# Control of Non-deterministic Transition Systems for Linear Temporal Logic Specifications

by

Zhibing Sun

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2021

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

Consider the formal synthesis problem in the continuous dynamical systems. A systematic approach is the abstraction-based method: constructing an abstraction of the original continuous system in the discrete space, and then consider the control problem in the abstraction. We consider our abstraction as a non-deterministic transition system (NTS), and describe our control specifications as linear temporal logic (LTL) formulas. The standard procedure is to convert the LTL formula as a deterministic automaton (DA) with a corresponding accepting condition, taking the product of the NTS and DA, and then solve it as an infinite game in the product space. The winning condition of the infinite game corresponds to the accepting condition of the DA, which often has polynomial or even exponential theoretical worst case time complexity.

With the development of more powerful computation platforms, the enormous scale of the NTS with millions of states and billions of transitions becomes relatively affordable. Such space complexity suggests that practical implementations should solve the infinite games in linear time at most. This motivates us to examine the gap between theoretical complexity upper bounds for solving infinite games and the run-time complexity of practical implementations of abstraction-based methods for solving temporal logic control problems. We give detailed analysis of different games such as reachability, safety, Büchi, co-Büchi, generalized Büchi, generalized co-Büchi, and Rabin from an algorithmic and implementation perspective to try to show that the theoretical upper bound is indeed too loose in practice. Better still, we implement an efficient Büchi solver to systematically solve for the control specifications as general as deterministic Büchi automata (DBA) translatable LTL formulas. The experimental results show that the program always terminates within one or two iterations. We conclude that as a result of the huge gap between theory and implementation, we can expect a linear time complexity to solve for the infinite game in practice, thus it is feasible to solve complicated specifications efficiently using abstraction-based methods.

**Acknowledgements**

First and foremost, I would like to express my sincere gratitude to my supervisor Professor Jun Liu for his constructive advice, strong support, wide tolerance, and positive feedback throughout my master's studies at the University of Waterloo.

Additionally, I am grateful to Professor Xinzhi Liu and Professor Stephen Smith for serving as my committee members and providing precious comments.

Last but not least, I am thankful to my hybrid system lab group for their companion and discussions, especially Dr. Yinan Li. I appreciate the friendship from the people that I have met here, especially Dongchang Li. Supports from our department, faculty, and university are sincerely acknowledged.

## Dedication

To my family including my girlfriend.

# Table of Contents

# List of Tables

# List of Figures

# List of Acronyms

**BFS** breadth first search 2, 46, 50, 53, 54, 58, 61, 189, 190, 225

**DA** deterministic automaton 2, 3, 11, 18, 26, 31, 167

**DAG** directed acyclic graph 71, 191, 207, 208

**DBA** deterministic Büchi automaton 3, 4, 30, 136, 146–149, 153, 154, 172, 179, 187

**DCA** deterministic co-Büchi automaton 136, 146, 149, 150, 159

**DE** difference equation 8, 9, 166

**DFA** deterministic finite automaton 143, 144

**DFS** depth first search 2, 189, 198

**DRA** deterministic Rabin automaton 30, 162, 164

**DSA** deterministic Streett automaton 30

**FIFO** first in first out 190

**FILO** first in last out 198, 199, 205, 206, 210

**GDBA** generalized deterministic Büchi automaton 152, 153

**GDCA** generalized deterministic co-Büchi automaton 157, 159

**GNBA** generalized non-deterministic Büchi automaton 30

**LTL** linear temporal logic 1–3, 10, 11, 15, 18, 25, 167

**NA** non-deterministic automaton 18

**NBA** non-deterministic Büchi automaton 2, 30

**NTS** non-deterministic transition system 2, 3, 7, 10, 11, 16, 17, 26, 34, 187

**ODE** ordinary differential equation 7, 8, 10, 166

**SCC** strongly connected components 70, 199, 206, 207

# Chapter 1

# Introduction

Formal methods are systematic approaches of studying a certain specifications such as verification and synthesis in a dynamical system. It balances the theoretical rigorousness and the computational efficiency.

In the formal verification problem [3], our goal is to check whether the execution of a system always satisfies a certain property, often expressed as a temporal logic formula such as a linear temporal logic (LTL) formula. While formal verification has been well-studied in the past thirty years, its dual problem of formal synthesis has also been studied for decades in computer science [29]. However, the synthesis of controller for continuous dynamical system hasn't aroused much attention until recently [34], [4]. In the formal synthesis problem, our goal is to synthesize or to control a system according to a temporal logic control specification.

We look into the formal synthesis problem from the perspective of an abstraction-based approach, which converts a control problem in a continuous dynamical system to a control problem in its abstraction in the discrete space. Our particular interest into this method is that, it uses the strategy of divide and conquer that separates a complicated control problem in a continuous dynamical system into two subproblems: constructing the abstraction and solving the control problem in the abstraction. With this separation, the continuous portion of the problem is restricted in the construction of the abstraction, which allows us to consider the various control specifications described as LTL formulas in the discrete space.

Central to the abstraction-based approach for formal verification and synthesis of an infinite-state system is, namely, the construction of the abstraction. Roughly, the abstraction can be considered as a finite graph, whose nodes map the partition of sets of states

of the original continuous system, and whose edges project the trajectories of the original continuous system of the equivalence classes. The abstraction of a continuous dynamical system can be constructed by over-approximating the dynamics using symbolic models [39], [32], [24] or interval analysis [26]. Such abstractions are conservative in order to preserve soundness. A sound abstraction guarantees that any discrete controller synthesized using the abstraction can be implemented in the original system to guarantee soundness in terms of satisfying LTL specifications. non-deterministic transition system (NTS) is commonly used as the model for the abstraction to more precisely approximate the original system.

In this thesis, we focus on the control problems in the NTS, which is the discrete portion of the continuous control problem. We describe our control specification as an LTL formula, which can be automatically converted into a graph based automaton with a corresponding accepting condition by some tools such as Spot [13]. The token in the automaton tracks the status of the agent in the NTS. In both verification and control problems, we first take the product of the NTS and the automaton, and then consider the problem in an infinite two player game formulation [15]. The infinite game models the interactive behavior of the agent in the system and the uncertainty of the environment. We use one player to represent the agent and the other to represent the uncertainty of the environment. The game structure captures the worst uncertainty of the environment to include any uncertainty of the NTS.

In a verification problem, we would like to check whether all the trajectories satisfy the accepting condition of the specification. Since one counter-example is sufficient enough to falsify the verification, the idea of solving a verification problem is to find a contradiction. It only requires traversing the graph once by the variation of some standard searching approaches such as breadth first search (BFS) or depth first search (DFS), thus the time complexity is linear w.r.t. the size of the graph, and a non-deterministic Büchi automaton (NBA) is enough to do the tracking.

In a control problem, we would like to find all the initial set of states such that all the controlled trajectories initiated in this set satisfy the accepting condition of the control specification regardless of the non-determinism of the NTS. We enumerate all the possible trajectories to construct the product graph, and we require a deterministic automaton (DA) to track the trajectory uniquely. The winning condition of the game is converted from the accepting condition of the DA. A special case is that, if the control specification $\varphi$ is defined by the system states in the NTS, then the infinite game is played in the NTS directly.

The time complexity of the commonly converted games such as Büchi and co-Büchi is polynomial, whereas the time complexity of the more general Rabin and Street game even

has exponential or factorial terms [16], [28]. This shows that the control problem is much more complicated than the verification problem. However, due to the enormous size of the NTS, and the exponential blow up w.r.t. the number of dimensions when scaling, only linear time complexity is feasible in practice. This motivates us to show that the theoretical polynomial worst case time complexity does not often happen in practice; otherwise the abstraction-based method wouldn't have much implementation value in practice.

With the help of the computational tools such as SCOTS [32] and ROCS [23] that construct the NTS of a continuous dynamical system, we implement a program to solve the control problems in NTS with the control specifications as general as the deterministic Büchi automaton (DBA) translatable LTL formulas. The experimental results show that the program terminates within one or two iterations w.r.t. the size of the product space of NTS and DBA, which implies a linear time complexity. This converts the bottleneck of the abstraction-based method from time complexity to space complexity, which further motivates us to look back into the continuous system to try to reduce the size of the abstraction for future work.

Since the result of the implementation may vary significantly case by case, it is not convincing without any theoretical analysis. Much efforts in this thesis is contributed to organically connect the NTS, the DA, and the LTL together to convert our control problem in NTS into an infinite game. The infinite game structure is more abstract than the control problem itself, which allows us to discuss from a high level algorithmic perspective. In each particular game, we give detailed analysis in an effort to show that a polynomial algorithm in theory is likely to be linear in practice without even adding heuristic optimizations.

## 1.1 Contributions of the Thesis

The main contributions of the thesis are as follows.

- In chapter 5, we give detailed analysis of each game from an algorithmic and implementation perspective. We present the algorithms in a form that is close to implementation. We give algorithms from the perspective of player 1 for both of the dual games. We give relatively tighter upper bounds and heuristic optimizations for games such as Büchi and generalized Büchi. We give examples of worst case scenarios for different games. We give a sound but not complete algorithm for the $k$ pairs Rabin game with $\mathcal{O}(kn^2m)$ time complexity.

- In chapter 6, we give a procedure of how to solve different control problems in NTS more efficiently. We give detailed quantification and complexity analysis of the different control problems, including reachability, safety, Büchi, co-Büchi, generalized Büchi, generalized co-Büchi, and Rabin.
- We implemented an efficient Büchi solver in ROCS [23] that systematically solves for the control specifications as general as DBA translatable LTL formulas.

## 1.2 Organization of the Thesis

The rest of the thesis is organized as follows.

- In chapter 2 Abstraction, we will discuss the brief idea of how to project the dynamical system from the continuous space to the discrete space as an abstraction.
- In chapter 3 Linear Temporal Logic (LTL), we will discuss the connection among $\omega$-language, automata, and linear temporal logic in the automata theory.
- In chapter 4 Formulation of Infinite Games, we will discuss the problem formulation of infinite games which is closely related to the automata theory in chapter 3.
- In chapter 5 Solution of Infinite Games, we will discuss the algorithms and the complexity analysis of different infinite games. The background for graph theory, data structure, and algorithm can be found in Appendix A.
- In chapter 6 Non-deterministic Transition System (NTS), we will discuss the conversion of a control problem in NTS to an infinite game problem, and how to perform the simulation in the continuous space using the information in the discrete space.
- In chapter 7 Implementations, Conclusions and Future Work, we will discuss some analysis and results related to the implementations. We will conclude our analysis and implementation results and discuss several directions of future work.
- In Appendix A, we will introduce some background in graph theory, data structure and algorithm as preparation for chapter 5.
- In Appendix B we will discuss JiushaoQin's algorithm to explain what we mean by encoding and decoding in various scenarios.

# Chapter 2

# Abstraction

In this chapter, we will discuss how to project the dynamical system from the continuous space to the discrete space as an abstraction. We will leave the mapping from the discrete space back to the continuous space in the simulation section in chapter 6 when we have our winning set and winning strategy in the discrete space ready for simulation in the continuous space.

Since the focus of this thesis falls in the discrete space, we will only give the idea of the conversion between continuous space and the discrete space. There are actually several approaches of constructing the abstraction of a continuous dynamical system using some numerical methods.

## 2.1 Motivation

In this section, we will discuss the motivation of converting a continuous problem to a discrete problem.

**Problems in a Continuous System**

First we would like to explain our motivation to work in the abstraction of the continuous dynamical system in the discrete space instead of in the continuous space directly.

There are mainly two problems in a continuous system:

1. infinite states in the continuous space;

2. expensive to compute repeatedly in the continuous space.

1. **Infinite States in the Continuous Space**

   In theory, there are infinitely many states in the continuous space. However, when it comes to computation, infinitely many states may not be representable or feasible to compute directly. We would like a computer to solve a problem for us in finite time using finite space, otherwise it is not practical. Therefore, it is inevitable for us to discretize the continuous system at some point.

2. **Expensive to Compute Repeatedly in the Continuous Space**

   In computation, real numbers are more expensive to compute than integers. Also, the general control specifications require at least polynomial time to solve. Therefore, it is more efficient to convert the dynamical system from the continuous space to the discrete space in linear time, then solve the control problem in the discrete space in polynomial time, and finally map back to the continuous space in linear time. In this way, we only need to compute the dynamics in the continuous space once. Solving the control problem in the continuous space directly requires repetitive computing of the dynamics in the continuous space, thus is more expensive in computation.

These two problems in a continuous system motivates us to separate a continuous control problem into two parts:

1. mapping from the continuous space to the discrete space and mapping from the discrete space back to the continuous space;

2. solving the control problem in the discrete space.

**Setting Abstraction-based Method as Reference**

Another motivation for us to adopt the abstraction-based method is that, since the idea of the method is quite intuitive and has a wide range of applications, we would like to set it up as a reference to the other more advanced methods. We would like to play with the data structure and optimize the graph searching algorithms in the discrete space to the extreme to see how it performs in space and time. The implementations somehow show that the theoretical high complexity is almost acceptably linear in practice. We would like to compare it with the other methods that focus more in the continuous space to figure out a better way to balance the techniques in the continuous space and the discrete space. We would also like to consider the result of the abstraction-based method as a reference when we implement other more complicated methods to see whether their results are compatible.

## 2.2 Abstraction – from Continuous to Discrete

In this section, we will discuss how to project a continuous dynamical system to a discrete dynamical system as an abstraction.

We will first give the definition of a continuous dynamical system and the definition of its abstraction in the form of an NTS. Then we give the procedure to construct the NTS from the continuous dynamical system.

**Definition 2.2.1** (continuous dynamical system)**.** *A continuous dynamical system is a 4-tuple* $(X_c, \Sigma_c.\xi_c, L)$*, where*

1. $X_c$ *is the set of continuous system states;*
2. $\Sigma_c$ *is the set of continuous control actions;*
3. $\xi_c$ *is a transition function described by an ordinary differential equation (ODE)* $\dot{x}(t) = f(x(t), \sigma)$*;*
4. $L \subseteq 2^{AP}$ *is the set of labels.*

The abstraction of a continuous system is a discretization of the continuous system. Here we choose the model of an NTS as our model for abstraction.

**Definition 2.2.2** (non-deterministic transition system)**.** *A Non-deterministic Transition System (NTS) is a 4 tuple* $(X, \Sigma, \xi, L)$*, where*

1. $X_d$ *is the set of discrete system states;*
2. $\Sigma_d$ *is the set of discrete control actions;*
3. $\xi_d : X_d \times \Sigma_d \to 2^{X_d}$ *is a non-deterministic transition function;*
4. $L \subseteq 2^{AP}$ *is the set of labels.*

### Idea of the Abstraction

Now we give the idea of the abstraction. The main idea of the abstraction is to map a continuous system into a discrete graph according to the equivalence relation or the partition. There are mainly three factors from the continuous dynamical system that influence the abstraction:

- the choice of the partition of the set of continuous system states $X_c$;

  We partition the set of continuous system states $X_c$ into different regions.

- the choice of finitely many continuous control actions from $\Sigma_c$;

  We select a finite number of continuous control actions from $\Sigma_c$.

- the choice of the unit time step $\Delta t$.

  We select a fixed unit time step $\Delta t$. This also implies that we detect the status of the agent in the system per $\Delta t$ time. We may also choose to detect the status of the agent in the system after applying each control action, where the time step may not be unified.

After we fix the unit time step $\Delta t$, we can convert the ODE $\dot{x}(t) = f(x(t), \sigma)$ to a difference equation (DE) $x_{t+1} = f(x_t, \sigma)\Delta t + x_t$. Then we can compute the transitions of one time step of the dynamics according to some numerical scheme of our choice.

**Discussion.** *Here we only give a conceptual idea of the conversion of dynamics from continuous to discrete. The more rigorous way to present is to over-approximate by adding validated upper bounds.*

*A commonly used numerical scheme is the* $4^{th}$ *order Runge-Kutta method.*

We will discuss the mapping from the continuous dynamical system to the NTS from 5 perspectives: system states, control actions, transition function, non-determinism and compatibility for errors.

**System States**

We partition the set of continuous system states $X_c$ into $n_0$ regions, i.e.,

$X_c = R_1 \oplus \cdots \oplus R_{n_0}$, and map it to the set of discrete system states $X_d = \{x_{di} | 1 \leq i \leq n_0\}$. Such mapping projects a set of infinite points in the same region $x_{ci} \in R_i \subseteq X_c$ to a node $x_{di} \in X_d$, $1 \leq i \leq n_0$. In other words, we map a set of infinite points into one integer. We consider the system states as nodes in the discrete graph.

The most intuitive idea to discretize the continuous space is to use the uniform grid, which uses a linear data structure. We set up precision for each dimension, and then partition the continuous space according to the precision. We also have other methods that use non-linear data structures to partition the continuous space such as using bisection method to partition each dimension of the state space until desired precision, or partition the state space as triangles or polygons.

**Control Actions**

We map the finite number of continuous control actions $\{\sigma_{ci}|1 \leq i \leq |\Sigma_d|\} \subseteq \Sigma_c$ to the finite number of discrete control actions $\Sigma_d = \{\sigma_{di}|1 \leq i \leq |\Sigma_d|\}$. Such mapping is 1-1 correspondence. We consider the control actions as either colouring on the edges, or nodes depending on different graph representations in the discrete graph.

We can use the method of uniform grid or bisection to select the finite set of continuous control actions as our representatives.

**Transition Function**

For a set of infinite points in each region, we calculate the DE under each control action $\sigma_c$ for one time step $\Delta t$, i.e., $x'_c = f(x_c, \sigma_c)\Delta t + x_c$. The continuous transition is given by $(x_c, \sigma_c, x'_c)$. The projection of this continuous transition to the discrete graph is given by $(x_d, \sigma_d, x'_d)$. The continuous transitions that starts in the same region $x_c \in R$ under the same control action $\sigma_c$, and ends in the same region $x'_c \in R'$ forms an equivalence class which can be represented by $(x_d, \sigma_d, x'_d)$ according to the projection. The equivalence class allows us to use one discrete transition to represent infinitely many continuous transitions. This allows us to compute each continuous transition only once, and the number of discrete transitions is also greatly reduced. In practice, we only need to compute finitely many continuous transitions in each region to conservatively approximate the behavior of infinitely many continuous transitions starting in the same region. It is likely that only some boundary points or even the corner points plus some approximate bounds are sufficient enough according to different numerical schemes, which is computationally efficient. We consider the transitions as edges in the discrete graph.

**Non-determinism**

The continuous transitions that start in the same region may end in different regions under the same control action. For example, we may have $(x_{c1}, \sigma_c, x_{c2})$ and $(x'_{c1}, \sigma_c, x_{c3})$, where $x_{c1}, x'_{c1} \in R_1$, $x_{c2} \in R_2$, and $x_{c3} \in R_3$. The projection to the discrete space is thus $(x_{d1}.\sigma_d, x_{d2})$ and $(x_{d1}.\sigma_d, x_{d3})$. Equivalently, we have $\xi_d(x_{d1}, \sigma_d) = \{x_{d2}, x_{d3}\}$ in the form of a non-deterministic transition. The two graph representations of the non-determinism of the transition function is shown in Figure 2.1.

Since we would like to use a single node to represent a set of infinite points in a region, the projection of the transitions may go to several different nodes under the same control action. This is where the non-determinism comes from.

**Compatibility for Errors**

The numerical errors, measuring errors, perturbations, and uncertainties can all be considered in the non-determinism.

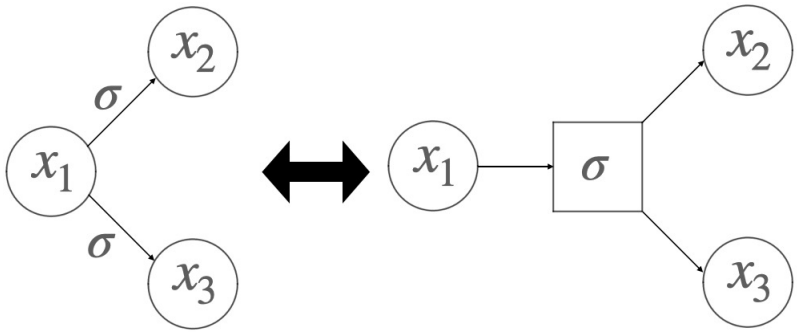


Figure 2.1: Two Graph Representations of the Non-determinism of the Transition Function

For the labelling, we use under approximation for the positive atomic propositions and over approximation for the negative atomic propositions.

The gap between the nominal abstraction and the real continuous system is increasingly closing by the non-deterministic transition function as we increase the precision of our refinement.

## 2.3 Problem Formulation

**Problem Formulation in Continuous Setting**

Given a continuous dynamical system $(X_c, \Sigma_c, \xi_c, L)$ and a control specification $\varphi$, find the initial set and the winning control strategy such that all the trajectories induced by the initial state and the sequence of control inputs inductively satisfy the accepting condition of $\varphi$.

We consider control synthesis in a dynamical system. Therefore, the problem consists of two parts: the dynamical system and the control specifications. We consider our system as a continuous dynamical system described by the ODE. The control specifications can be directly defined by the states in the dynamical system. A more general and systematic approach is to describe our control specification as an LTL formula.

**Approach:**

We adopt a systematic abstraction-based approach:

1. Project the dynamical system from the continuous space to the discrete space as an NTS.

2. Describe the control specification as an LTL formula $\varphi$.

3. Convert $\varphi$ to a DA with a corresponding accepting condition Acc in an automation process.

4. Convert the control problem in NTS to an infinite two player game problem.

5. Solve for the winning set and the winning control strategy of the control problem.

6. Use the information in the discrete space to guide the controller back in the original continuous space.

Considering the control problem in the abstraction of the original continuous system allows us to consider the control problem in the discrete setting.

**Problem Formulation in Discrete Setting**

Given an NTS = $(X_d, \Sigma_d, \xi_d, L)$ and a control specification $\varphi$, find the initial set and the winning control strategy such that all the trajectories induced by the initial state and the sequence of control inputs inductively satisfy the accepting condition of $\varphi$ regardless of the non-determinism of the NTS.

A brief explanation of the problem formulation in the discrete setting is given at the end of chapter 3 when we have discussed the form of the control specification as an LTL formula. The detailed definition of the problem formulation is given in chapter 6 after we have discussed the formulation and the solution of infinite games in chapter 4 and 5 respectively.

## 2.4 More about Non-determinism

We use the equivalence relation and the partition to map between a continuous dynamical system and its abstraction. Therefore, there must be some trade-offs, which is the non-determinism. We will show how the non-determinism connects the continuous dynamical system and its abstraction.

In Figure 2.1, we can see how it works in an NTS: at a system state $x_{d1}$, we choose a control action $\sigma_d$, and it will randomly go to the next system state $x_{d2}$ or $x_{d_3}$. Therefore, we can consider the NTS as a bipartite graph of state nodes and action nodes.

**How does the NTS match up with the continuous system?**

Consider the non-deterministic transition function $\xi_d(x_{d1}, \sigma_d) = \{x_{d2}, x_{d3}\}$ in the NTS. It implies that back in the continuous space, there exist continuous transitions $(x_{c1}, \sigma_c, x_{c2})$ and $(x'_{c1}, \sigma_c, x_{c3})$, where $x_{c1}, x'_{c1} \in R_1$, $x_{c2} \in R_2$ and $x_{c3} \in R_3$.

Therefore, if we want all the continuous transitions $(x_{c1}, \sigma_c, x_c)$ for one time step $\Delta t$, where $x_{c1} \in R_1$ to contribute to the satisfaction of the control specification $\varphi$ in the continuous space, we require all the non-deterministic transitions $\xi(x_{d1}, \sigma_d)$ for one time step to contribute to the satisfaction the control specification $\varphi$ in the discrete space.

Conversely, if a non-deterministic transition $(x_{d1}, \sigma_d, x_{d2})$ violates the satisfaction of the control specification $\varphi$ in the NTS, then it implies that back in the continuous system, the exist some point $x_{c1} \in R_1$ such that $(x_{c1}, \sigma_c, x_{c2})$ violates the satisfaction of the control specification $\varphi$.

**Defects of a Deterministic Transition System**

The non-determinism leads to uncertainty. So why don't we choose the deterministic transition system as our model for the abstraction instead?

The reason is that there are mainly two defects of a deterministic transition system:

1. if we only consider the control actions that lead to deterministic transitions, then the number of available transitions may be too restrictive;
2. it is even possible that no deterministic transitions exist in the first place.

## 2.5 Pros and Cons

The pros and cons both come from the comparison between the properties of a discrete problem and a continuous problem.

**Pros**

1. **Turn an infinite system into a finite system**

   Mapping a continuous system into a discrete graph turns an infinite system into a finite system. In other words, this may turn the problem from unsolvable to solvable, even though the size of the system might be huge as we increase the precision.

2. **Save Computation for Later Work**

We only compute the necessary transitions in the continuous space once, then we convert the continuous problem into a discrete one in the abstraction.

Also, most infinite games converted from our control problem in the NTS have polynomial time complexity. Therefore, it would be even better for us to compute the continuous transitions just once.

3. **Less Unpredictable Errors in Discrete Space**

   We may have more confidence in solving a discrete problem than in solving a continuous problem for the reason that there may be more unpredictable errors in the continuous space. The abstraction-based method decomposes a continuous problem to the construction of the abstraction in the continuous space and a discrete problem. We only need to focus on the continuous space when projecting the continuous system to its abstraction in the discrete space. Then we can apply the efficient graph searching algorithms in the discrete space which are easier to debug as a result of less uncertainty.

   Mapping from the discrete space back to the continuous space is already contained in the mapping from the continuous space to the discrete space because their relation is like that of the necessary condition and the sufficient condition.

4. **Less Requirements on Assumptions**

   By discretizing the continuous problem in the first place, we are taking the advantage of the computational power of a computer to violently enumerate the system. Therefore, this may avoid the problem that the continuous dynamical system doesn't possess nice assumptions such as convexity.

5. **Nice Extension Properties**

   The abstraction-based method converts a continuous problem to a discrete graph searching problem. Therefore, it is convenient for us to add other factors into consideration such as stochastics and weights.

**Cons**

1. **Model Mismatch**

   There may be a mismatch of the abstraction and the original system by inaccurate over-approximations.

2. **Expensive in Space for Refinement**

   Theoretically, it is possible that the abstraction-based method can be sound and complete as we refine the abstraction up to any precision. However, in practice it is quite expensive if the set of continuous system states $X_c$ and the set of continuous control actions $\Sigma_c$ have several dimensions and the requirement for the precision is high. The blow up is exponential w.r.t. the number of dimensions. For example, if we have $k$ dimensions and we equally partition each dimension into two parts, then the number of states will blow up by a factor of $2^k$.

3. **Expensive in Time for Refinement**

   The carry on aftermath of the exponential blow up in space is an exponential blow up in time as we refine our abstraction. This comparison is for the abstraction-based method with different scaling.

   We can expect a linear time complexity in practice for solving a theoretically polynomial time complexity problem in the discrete space. Therefore, the abstraction-based method may still outperform other methods that work directly in the continuous space in time with a trade-off of extra space complexity.

# Chapter 3

# Linear Temporal Logic (LTL)

In this chapter, we will discuss the mechanism to define our control specifications. We will first introduce the $\omega$-regular languages which is the formulation we use to depict infinite sequences [15, Chapter 1]. All the propositions involving infinite sequences is related to the $\omega$-regular languages. We will then discuss automaton which is a graph based mechanism that we use to keep track of the status of the agent in a system [3, Chapter 4], [15, Chapter 1], and [7]. The accepting condition of an automaton is defined by the formulation of the $\omega$-regular languages, which is define in the same fashion as the winning condition of an infinite two player game [15, Chapter 2]. Finally, we will introduce LTL, which is the general language we use to define our control specification [3, Chapter 5]. Our interest in the LTL is that, an LTL formula can be converted to an automaton in an automation process.

Since there is rich literature in the automaton theory and LTL, we only extract out the necessary ingredients to define our control problem.

## 3.1 Omega-Regular Languages

In this section, we will introduce the notations and the basic ingredients to define the $\omega$-regualar language.

**Omega**

We use $\omega$ to denote a non-negative integer or the "infinity": $\omega \in \{0, 1, 2, 3, \cdots, \infty\}$. We use $\omega$ to denote "infinity" most of the time.

**Atomic Propositions**

$AP = \{a_i | 1 \leq i \leq |AP|\}$ is the finite set of atomic propositions, with length $|AP|$ being its cardinality. The different $a \in AP$ are used to label the different properties of a state $x \in X$ in the NTS.

**Label**

$L \subseteq 2^{AP}$ is the set of labels consisting of the $|AP|$ atomic propositions, where $2^{AP}$ is the powerset of $AP$.

A label $l \in L$ can therefore be considered as a $|AP|$-tuple of 0 and 1 representing the status of the $|AP|$ atomic propositions, where the 0 and 1 on the $i^{th}$ projection of the tuple stands for $\neg a_i$ and $a_i$ respectively.

It follows that each label $l \in L$ can also be considered as a number in binary representation, or in decimal representation. We consider a label $l$ in the more compact decimal representation most of the time unless evaluating the Boolean functions defined on $AP$s. Since $L$ is the set of atomic propositions in logic, we prefer to call $L$ as the set of labels than alphabet.

**Word**

A word is a sequence of labels.

The set of finite words over $L$, denoted by $L^*$, is a set of finite sequences of labels. A finite word $w \in L^*$ has length $|w| = \omega$ being a non-negative integer. We denote a finite word $w$ as $w = w(0)w(1)\cdots w(n) = l_0 l_1 \cdots l_n$ with $w(i) = l_i \in L$, $|w| = n$.

The set of infinite words (or $\omega$-words) over $L$, denoted by $L^\omega$, is a set of infinite sequences of labels. An infinite word $\alpha \in L^\omega$ has length $|w| = \omega$ being infinity. We denote an infinite word $\alpha$ as $\alpha = \alpha(0)\alpha(1)\cdots = l_0 l_1 \cdots$ with $\alpha(i) = l_i \in L$, $|\alpha| = \infty$.

**Language**

A language is a set of words.

A set of finite words over a given set of labels $L$, denoted by $\mathcal{L}^*$, is called a finite language.

A set of $\omega$-words over a given set of labels $L$, denoted by $\mathcal{L}^\omega$, is called an $\omega$-language.

**Occurrence**

The occurrence of a word $w$ or $\alpha$ is the set of labels $l \in L$ occurring in the word $w$ or $\alpha$. Formally, $\mathrm{Occ}(w) = \{l \in L | \exists i \geq 0 \text{ s.t. } w(i) = l\}$ or $\mathrm{Occ}(\alpha) = \{l \in L | \exists i \geq 0 \text{ s.t. } \alpha(i) = l\}$. The number of occurrences of a label $l \in L$ in a word $w$ or $\alpha$ is denoted by $|w|_l$ or $|\alpha|_l$.

**Infinity Set**

The infinity set of an $\omega$-word $\alpha \in L^\omega$ is the set of labels $l \in L$ occurring infinitely many often in $\alpha$. Formally, $\text{Inf}(\alpha) = \{l \in L | \forall i \geq 0, \exists j \geq i \text{ s.t. } \alpha(j) = l\}$.

**Discussion.** *The occurrence and the infinity set can be applied to different sequences in different scenarios:*

- *A trajectory $\rho_x$ is a sequence of system states in the NTS;*
- *A run $\rho_q$ is a sequence of automaton states in the automaton;*
- *A play $\pi_v$ is a sequence of nodes in the infinite game.*

*We use the the occurrence or the infinity set of a sequence and some states of the system to define the specification:*

- *For a control specification defined by the states in the NTS, the accepting condition of a trajectory is defined as a proposition using the occurrence set $Occ(\rho_x)$ or the infinity set $Inf(\rho_x)$, and the subsets of $X$;*
- *For an automaton, the accepting condition of a run is defined as a proposition using the occurrence set $Occ(\rho_q)$ or the infinity set $Inf(\rho_q)$, and the accepting set $Acc$;*
- *For an infinite game, the winning condition of a play is defined as a proposition using the occurrence set $Occ(\pi_v)$ or the infinity set $Inf(\pi_v)$, and the objective set $Obj$.*

*Therefore, we use the $\omega$-regular language to connect the NTS, the automaton, and the infinite game.*

## 3.2 Deterministic Automaton

Since the idea of an automaton is to use a finite machine to keep track of an infinite process, our discussion will be focused on automata of finite size, i.e., the number of states $n_1$ and transitions $m_1$ of the automaton is finite, which will be abbreviated as automaton. By finite automaton, we mean the input word $w$ and the accepting condition of the automaton is finite. This differentiates from the automaton with an infinite input word $\alpha$ and an infinite accepting condition called $\omega$-automaton, where $\omega$ denotes infinity as in $\omega$-word and $\omega$-language.

The general automata are non-deterministic. However, since our focus is on the control synthesis, which requires keeping track of every automaton state $q$ in a run $r$. Therefore, in order to guarantee the run of an automaton induced by the initial state $q_0$ and the input word $\alpha$ is unique, the automaton must be deterministic.

A non-deterministic automaton is used in the falsification of a model-checking problem, and in the construction of an automaton from an LTL formula.

### 3.2.1 Definition of Deterministic Automaton

**Definition 3.2.1** (deterministic automaton)**.** *A Deterministic Automaton (DA) is a 5-tuple* $DA = (Q, L, \tau, q_0, Acc)$*, where*

- $Q$ *is the set of automaton states;*
- $L \subseteq 2^{AP}$ *is the set of labels;*
- $\tau : L \times Q \to Q$ *is a deterministic transition function;*
- $q_0 \in Q$ *is the unique initial state;*
- $Acc \subseteq 2^Q$ *is the set of final states in a finite automaton or the set of accepting states in an* $\omega$*-automaton.*

**Remark.** *The counterpart of a DA is the more general case non-deterministic automaton (NA), where the generalization only lies in the initial states and the transition function: the initial states is generalized from a unique initial state* $q_0 \in Q$ *to a set of initial states* $Q_0 \subseteq Q$*, and the transition function is generalized from a deterministic transition function* $\tau : L \times Q \to Q$ *to a non-deterministic transition function* $\tau : L \times Q \to 2^Q$*.*

**Discussion.** *Now we will discuss the use of a DA. We place a token on the initial state* $q_0$ *of the DA, and then move the token according to the transition function* $\tau$ *to track the status of the agent in a system:*

- *a DA can be considered as a graph, where the automaton states* $Q$ *are the nodes and the transition function* $\tau$ *is hidden in the edges;*
- *a token is initially placed on the initial state* $q_0$*;*
- *for the transition function* $\tau$*, we input the label* $l$ *of the current NTS state* $x$ *and the current DA state* $q$*, and output the next DA state* $q'$*, i.e.,* $\tau(l, q) = q'$*; we move the token from* $q$ *to* $q'$*.*

### 3.2.2 Classification of Automata

An automaton can be classified and correspondingly abbreviated as

$\{\emptyset, G\} \times \{D, N\} \times \{F, B, C, R, S\} \times \{A\}$ according to:

- the number of objectives in the accepting condition: one (omit) or more ($G$ for generalized);
- its determinacy: deterministic ($D$) or non-deterministic ($N$);
- its accepting condition: finite ($F$), Büchi ($B$), co-Büchi ($C$), Rabin ($R$), and Streett ($S$);
- $A$ is for automaton.

**Remark.** *We use $\mathcal{A}$ to denote an automaton when its determinacy is not specified.*

*For the accepting conditions, only the first one $F$ requires finite words $w$ as input; all the latter ones require infinite words $\alpha$ as input. This is why the first one is called a finite automaton (FA), and the latter ones are called $\omega$-automata.*

**Example.** *Two examples of the abbreviations of automata are $NFA$ for non-deterministic finite automaton and $GDBA$ for generalized deterministic Büchi automaton.*

### 3.2.3 Run

The concept of run is the mechanism we use to keep track of an infinite process using an automaton.

A run is a sequence of automaton states.

**Definition 3.2.2** (finite run)**.** *Let $DFA = (Q, L, \tau, q_0, Acc)$ be a deterministic finite automaton. A finite run of $DA$ on a finite word $w = w(0)w(1)\cdots w(n) = l_0 l_1 \cdots l_n \in L^*$ is a finite sequence of atomaton states $r = r(0)r(1)\cdots r(n+1) = q_0 q_1 \cdots q_{n+1} \in Q^{n+2}$ induced by $r(0) = q_0$ and $r(i+1) = \tau(l_i, r(i)), 0 \leq i \leq n$, where $w(i) = l_i \in L$ and $r(i) = q_i \in Q$, $|w| = n, |r| = n+1$.*

**Discussion.** *In the non-deterministic case, given a finite word $w$, the runs*

*$r = r(0)r(1)\cdots r(n+1) = q_0 q_1 \cdots q_{n+1} \in Q^{n+2}$ are induced by $r(0) = q_0 \in Q_0$ and $r(i+1) \in \tau(l_i, r(i)), 0 \leq i \leq n$.*

*Thus one finite word $w$ would induce* $\sum_{r(0)=q_0\in Q_0}\left(\prod_{i=0}^{n}|\tau(l_i,r(i))|\right)$ *many different runs, where the non-determinacy $|\tau(l_i,r(i))|$ roughly grows exponentially w.r.t. the length of the input word $|w|=n$. Roughly, the number of finite runs induced by a single word $w$ is $|Q_0|\cdot|\tau|^n$.*

**Definition 3.2.3** (infinite run)**.** *Let $DA=(Q,L,\tau,q_0,Acc)$ be a deterministic $\omega$-automaton. An infinite run of $DA$ on an $\omega$-word $\alpha=\alpha(0)\alpha(1)\cdots=l_0l_1\cdots\in L^\omega$ is an infinite sequence of automaton states $\rho=\rho(0)\rho(1)\cdots=q_0q_1\cdots\in Q^\omega$ induced by $\rho(0)=q_0$ and $\rho(i+1)=\tau(l_i,\rho(i)), i\geq 0$, where $\alpha(i)=l_i\in L$ and $\rho(i)=q_i\in Q$, $|\alpha|=|\rho|=\infty$.*

**Discussion.** *In the non-deterministic case, given an $\omega$-word $\alpha$, the runs $\rho=\rho(0)r(1)\cdots=q_0q_1\cdots\in Q^\omega$ are induced by $r(0)=q_0\in Q_0$ and $r(i+1)\in\tau(l_i,r(i))$, $i\geq 0$.*

*In the control synthesis, we would like to use runs to keep track of all the possibilities in an infinite process, the number of runs induced by a single input word must thus be finite and small, otherwise not feasible. However, when the automaton is non-deterministic, the number of runs induced by a single word $w$ in the finite case is already $|Q_0|\cdot|\tau|^n$, not to say in the infinite case where $n\to\infty$.*

*When the automaton is deterministic, $|Q_0|=|\tau|=1$, thus the infinite run $\rho$ induced by an $\omega$-word $\alpha$ is unique, which satisfies our requirement. That's why we must use $DA$ instead of $NA$ in control synthesis.*

### Acceptance of Run, Word, and Language

The acceptance of run, word and language are inductively defined. Without specifying the determinacy, let $\mathcal{A}$ be an automaton.

In a finite automaton $\mathcal{A}$, the acceptance of a finite run $r$, a finite word $w$, and a finite language $\mathcal{L}^*$ is inductively defined as:

1. a finite run $r=r(0)r(1)\cdots r(n+1)=q_0q_1\cdots q_{n+1}\in Q^{n+2}$ of $\mathcal{A}$ is accepted by $\mathcal{A}$ iff it satisfies the accepting condition of $\mathcal{A}$;
2. a finite word $w=w(0)w(1)\cdots w(n)=l_0l_1\cdots l_n\in L^*$ is accepted by $\mathcal{A}$ iff there exists a run $r$ of $\mathcal{A}$ on $w$ that satisfies the accepting condition of $\mathcal{A}$;
3. the finite language accepted by $\mathcal{A}$ is the set of finite words in $L^*$ accepted by $\mathcal{A}$. In other words, the finite language recognized by $\mathcal{A}$ is defined as
   $\mathcal{L}^*(\mathcal{A})=\{w\in L^*|\ \mathcal{A}\text{ accepts }w\}$.

In an $\omega$-automaton $\mathcal{A}$, the acceptance of an infinite run $\rho$, an $\omega$-word $\alpha$, and an $\omega$-language $\mathcal{L}^\omega$ is inductively defined as:

1. an infinite run $\rho = \rho(0)\rho(1)\cdots = q_0q_1\cdots \in Q^\omega$ of $\mathcal{A}$ is accepted by $\mathcal{A}$ iff it satisfies the accepting condition of $\mathcal{A}$;
2. an $\omega$-word $\alpha = \alpha(0)\alpha(1)\cdots = l_0l_1\cdots \in L^\omega$ is accepted by $\mathcal{A}$ iff there exists a run $\rho$ of $\mathcal{A}$ on $\alpha$ that satisfies the accepting condition of $\mathcal{A}$;
3. the $\omega$-language accepted by $\mathcal{A}$ is the set of $\omega$-words in $L^\omega$ accepted by $\mathcal{A}$. In other words, the $\omega$-language recognized by $\mathcal{A}$ is defined as $\mathcal{L}^\omega(\mathcal{A}) = \{\alpha \in L^\omega |\ \mathcal{A} \text{ accepts } \alpha\}$.

### 3.2.4 Accepting Condition of a Run

To close the logic for the definitions of the acceptance of run, word, and language, we need to define the accepting conditions for the runs, which is also how the accepting conditions of the automata are defined. The accepting condition of $\mathcal{A}$ is defined as a proposition using the occurrence set $\mathrm{Occ}(\rho)$ or the infinity set $\mathrm{Inf}(\rho)$, and the accepting set Acc.

**Finite Automaton**

The first accepting condition is for the finite automaton, where the input word $w$ to induce the run $r$ is finite.

For a finite automaton $FA$, $\mathrm{Acc} = F \subseteq Q$ is the set of final states. A finite run $r$ is accepted by $FA$ iff one of the states in $F$ occurs in $r$. Formally, $r$ is accepting iff $\mathrm{Occ}(r) \cap F \neq \emptyset$.

**$\omega$-Automaton**

The rest of the accepting conditions are for the $\omega$-automaton, where the input word $\alpha$ to induce the run $\rho$ is infinite.

These accepting conditions enjoy a dual property, i.e., Büchi and co-Büchi, generalized Büchi and generalized co-Büchi, Rabin and Streett are all mutually dual to each other, which can be seen from how they are defined. The duality allows us to look into a problem from both sides. Closer relations between the two duals will be shown later in the infinite two player games.

- **Büchi Condition** For a Büchi automaton $BA$, $\mathrm{Acc} = F \subseteq Q$ is the set of accepting states. An infinite run $\rho$ is accepted by $BA$ iff at least one of the states in $F$ occurs infinitely many often in $\rho$. Formally, $\rho$ is accepting iff $\mathrm{Inf}(\rho) \cap F \neq \emptyset$.

- **co-Büchi Condition** For a co-Büchi automaton $CA$, an infinite run $\rho$ is accepted by $CA$ iff only the accepting states occur in $\rho$ eventually, or equivalently, the rejecting states occur only finitely many often in $\rho$. Therefore, there are two ways to define the co-Büchi accepting condition:

  1. Acc $= F \subseteq Q$ is the set of accepting states. $\rho$ is accepting iff $\text{Inf}(\rho) \subseteq F$.
  2. Acc $= F \subseteq Q$ is the set of rejecting states. $\rho$ is accepting iff $\text{Inf}(\rho) \cap F = \emptyset$.

- **Generalized Büchi Condition** For a generalized Büchi automaton $GBA$, Acc $= \{F_1, \cdots, F_k\}$, where $F_i \subseteq Q$, $1 \leq i \leq k$ are the $k$ sets of accepting states. An infinite run $\rho$ is accepted by $GBA$ iff at least one of the states in $F_i$ occurs infinitely many often in $\rho$ for all $i$, $1 \leq i \leq k$. Formally, $\rho$ is accepting iff $\bigwedge_{i=1}^{k} (\text{Inf}(\rho) \cap F_i \neq \emptyset)$.

- **Generalized co-Büchi Condition** Similar to the co-Büchi condition, there are also two ways to define the accepting condition for a Generalized co-Büchi automaton $GCA$:

  1. Acc $= \{F_1, \cdots, F_k\}$, where $F_i \subseteq Q$, $1 \leq i \leq k$ are the $k$ sets of accepting states. An infinite run $\rho$ is accepted by $GCA$ iff only the states in $F_i$ occur in $\rho$ eventually for some $i$, $1 \leq i \leq k$. Formally, $\rho$ is accepting iff $\bigvee_{i=1}^{k} (\text{Inf}(\rho) \subseteq F_i)$.
  2. Acc $= \{F_1, \cdots, F_k\}$, where $F_i \subseteq Q$, $1 \leq i \leq k$ are the $k$ sets of rejecting states. An infinite run $\rho$ is accepted by $GCA$ iff the states in $F_i$ occur only finitely many often in $\rho$ for some $i$, $1 \leq i \leq k$. Formally, $\rho$ is accepting iff $\bigvee_{i=1}^{k} (\text{Inf}(\rho) \cap F_i = \emptyset)$.

- **Rabin Condition** For a Rabin Automaton $RA$, Acc $= \{(G_1, B_1), \cdots, (G_k, B_k)\}$, where $G_i, B_i \subseteq Q$, $1 \leq i \leq k$. "G" refers to the set of "Good" states that we would like to occur infinitely many often in a run $\rho$, whereas "B" refers to the set of "Bad" states that we would like to occur only finitely many often in $\rho$. An infinite run $\rho$ is accepted by $RA$ iff at least one of the states in $G_i$ occurs infinitely many often in $\rho$ and the states in $B_i$ occur only finitely many often in $\rho$ for some $i$, $1 \leq i \leq k$. Formally, $\rho$ is accepting iff $\bigvee_{i=1}^{k} ((\text{Inf}(\rho) \cap G_i \neq \emptyset) \wedge (\text{Inf}(\rho) \cap B_i = \emptyset))$.

- **Streett Condition** For a Streett Automaton $SA$, $\text{Acc} = \{(G_1, B_1), \cdots, (G_k, B_k)\}$, where $G_i, B_i \subseteq Q$, $1 \leq i \leq k$. "G" refers to the set of "Good" states that we would like to occur infinitely many often in a run $\rho$, whereas "B" refers to the set of "Bad" states that we would like to occur only finitely many often in $\rho$. An infinite run $\rho$ is accepted by $SA$ iff at least one of the states in $G_i$ occurs infinitely many often in $\rho$ or the states in $B_i$ occur only finitely many often in $\rho$ for all $i$, $1 \leq i \leq k$. Formally, $\rho$ is accepting iff $\bigwedge_{i=1}^{k} ((\text{Inf}(\rho) \cap G_i \neq \emptyset) \vee (\text{Inf}(\rho) \cap B_i = \emptyset))$.

### 3.2.5 Expressive Power

The language of an automaton $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of words accepted by $\mathcal{A}$. We also say that $\mathcal{A}$ recognizes $\mathcal{L}(\mathcal{A})$.

The expressive power of an automaton $\mathcal{A}$ is therefore measured by the number of words accepted by $\mathcal{A}$.

We say two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same expressive power, or are equivalent iff $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$. We say an automaton $\mathcal{A}_1$ is more expressive than $\mathcal{A}_2$ if $\mathcal{L}(\mathcal{A}_1) \supseteq \mathcal{L}(\mathcal{A}_2)$, and vise versa. In other words, $\mathcal{A}_1$ is more expressive than $\mathcal{A}_2$ if all the words accepted by $\mathcal{A}_2$ are also accepted by $\mathcal{A}_1$, and there exists a word accepted by $\mathcal{A}_1$ that is not accepted by $\mathcal{A}_2$.

We use $G$ to denote "generalized" for an accepting condition with more than one objective.

For the determinacy, we use $D$ and $N$ to denote "deterministic" and "non-deterministic" respectively.

For the accepting conditions, we use $B$, $R$, and $S$ to denote "Büchi", "Rabin", and "Streett" respectively.

We have the following results:

- $GBA$ can be converted to a $BA$, and $BA$ is a special case of $GBA$, therefore they have the same expressive power in both deterministic and non-deterministic case.
- $DBA$ is less expressive than $NBA$.
- $RA$ and $SA$ have the same expressive power in both deterministic and non-deterministic case, which recognize the complete $\omega$-regular language.
- $NBA$ also recognizes the complete $\omega$-regular language.

### 3.2.6 More about the Transition Function

- Implicitly, there is a Boolean function $bf : L \to \{0, 1\}$ on each edge of the automaton.
- For each of the automaton state $q_i$, let $bf_{i\alpha}$ be the Boolean function on the out-going edge $(q_i, q_\alpha)$, where $q_\alpha \in POST(q_i)$. Consider all of the Boolean functions on its out-going edges, we have two properties:
  1. the disjunction of all the Boolean functions is logic 1:
  $$\bigvee_{q_\alpha \in POST(q_i)} bf_{i\alpha} = 1;$$
  2. the conjunction of any two distinct Boolean functions is logic 0:
  $$bf_{i\alpha} \wedge bf_{i\beta} = 0, \alpha \neq \beta, \forall q_\alpha, q_\beta \in POST(q_i).$$
- The DA state $q$ chooses the unique transition on which the Boolean function $bf$ given the input $l$ outputs logic 1 to go to the next DA state $q'$.
- At an NTS state $x$ and a DA state $q$, we input the label $l$ at $x$ to the deterministic transition function $\tau$, and go through the transition on which the Boolean function $bf$ given the input $l$ returns logic 1 to the next DA state $q'$. We express this process as $\tau\_next(l, q) = q'$.

We can see that there can be two forms of the transition function:

1. calculate the transition function $\tau\_next$ on the fly;
2. pre-calculate the transition function into a $\tau\_prime$ table.

In practice, it is more efficient to use a $\tau\_prime$ table. More analysis will be given when we discuss taking the product of an NTS and a DA.

### 3.2.7 Non-deterministic Automaton

In this section, we will explain why it can't be used to solve a control problem and what it can be used for.

There are two differences between a deterministic automaton and a non-deterministic automaton: the transition function and the initial state.

1. **Transition Function**

   **In DA:** $T : L \times Q \to Q$ is a deterministic transition function.

   **In NA:** $T : L \times Q \to 2^Q$ is a non-deterministic transition function.

   IN an NA, some of the states $q_i$ have non-deterministic transitions, i.e., the Boolean functions $bf$ on the out-going edges of $q_i$ given the input $l$ that output logic 1 is not unique.

   In other words, the conjunction of two distinct Boolean functions on the out-going edges of $q_i$ may not be logic 0, which violates the second property of the transition function in a DA.

2. **Initial State**

   **In DA:** $q_0 \in Q$ is the unique initial state.

   **In NA:** $Q_0 \subseteq Q$ is the set of initial states, which may not be unique.

**Discussion.** ***Why can't we use an NA to solve a control problem?***

*As discussed above, given a label $l$ to a NA state $q_i$, there may be more than one transitions on which the Boolean function returns logic 1, which causes us to lose track of the status of the token.*

*This is why an NA can't be used to verify a run or to solve a game.*

*An NA can only be used in a posteriori explanation, i.e., the reason why the run of the token satisfies the accepting condition of the NA or not.*

**Discussion.** ***What can we use an NA for?***

*There are mainly two places that we uses an NA:*

1. *solving a verification problem in model checking;*
2. *converting an LTL formula to an automaton.*

## 3.3 Linear Temporal Logic

Since we would like to take advantage of the automation tools such as Spot [13] to help us convert an LTL formula to an automaton, we will only give the syntax of the LTL. We refer the reader to [3, Chapter 5] for a complete discussion of LTL. The arithmetics and

the reasoning of the LTL is much more complicated than that of the propositional logic. Therefore, our focus is on using the LTL formula to define our control specification, and leave the conversion from an LTL formula to a DA in an automation process.

This is similar to how we address a maximum flow problem in computer science: we only focus on setting up the problem, and then leave the solving process to the linear programming solver, which reduces our working load and divide a big problem into sub-problems.

### 3.3.1 Syntatic Rules

The basic ingredients of LTL formulas are the logic 1, the atomic propositions $a \in AP$, the Boolean connectors conjunction ("and") $\wedge$ and negation $\neg$, and two basic temporal modalities $X$ (reads "next") and $U$ (reads "until"). The propositional logic can be recursively defined by the former four ingredients, thus the construction of an LTL formula can be considered as propositional logic plus the two temporal modalities $X$ and $U$.

The negation of the logic 1 is the logic 0. The atomic proposition $a \in AP$ stands for one of the $|AP|$ dimensions in the state label $l \in 2^{AP}$ in an NTS.

The Boolean connector conjunction ("and") $\wedge$ is a binary infix operator that takes two LTL formulas $\varphi_1$ and $\varphi_2$ as arguments. $\varphi_1 \wedge \varphi_2$ holds at the current moment iff both $\varphi_1$ and $\varphi_2$ hold at the current moment.

The Boolean connector negation $\neg$ is a unary prefix operator that takes one LTL formula $\varphi$ as an argument. $\neg\varphi$ holds at the current moment iff $\varphi$ doesn't hold at the current moment.

The "next" modality $X$ is a unary prefix operator that takes one LTL formula $\varphi$ as an argument. $X\varphi$ holds at the current moment if $\varphi$ holds in the next moment.

The "until" modality $U$ is a binary infix operator that takes two LTL formulas $\varphi_1$ and $\varphi_2$ as arguments. $\varphi_1 U \varphi_2$ holds at the current moment either $\varphi_2$ holds at the current moment, or $\varphi_1$ holds from the current moment on until $\varphi_2$ holds in some future moment. In other words, $\varphi_2$ must hold at some moment starting from the current moment, and $\varphi_1$ must hold at each moment before $\varphi_2$ holds.

**Definition 3.3.1** (Syntax of LTL)**.** *A propositional Linear Temporal Logic (LTL) formula $\varphi$ over a set of atomic propositions $AP$ is recursively defined as*

$$\varphi ::= 1 | a | \varphi_1 \wedge \varphi_2 | \neg\varphi | X\varphi | \varphi_1 U \varphi_2,$$

*where $a \in AP$ is an atomic proposition and $\varphi$, $\varphi_1$ and $\varphi_2$ are LTL formulas.*

In words, a propositional Linear Temporal Logic (LTL) formula $\varphi$ over a set of atomic propositions $AP$, formula for short, is inductively defined by the following four rules:

1. The logic 1 is a formula.
2. Any atomic proposition $a \in AP$ is a formula.
3. If $\varphi$, $\varphi_1$, and $\varphi_2$ are formulas, then $\varphi_1 \wedge \varphi_2$, $\neg\varphi$, $X\varphi$, and $\varphi_1 U \varphi_2$ are also formulas.
4. Nothing else is a formula.

### 3.3.2 Precedence Order

In a hope to simplify a formula by shortening its length and omitting the brackets whenever possible, it is standard to use derived operators and to declare a precedence order on the basic and derived operators.

A standard precedence order assigns a higher priority to the unary operators than the binary operators. The unary operators "negation" $\neg$ and "next" $X$ bind equally strong. The temporal operator "until" takes precedence over the conjunction operator $\wedge$, and the conjunction operator $\wedge$ takes precedence over the other derived Boolean operators. This is because $\varphi_1 \wedge \varphi_2$ holds if and only if both $\varphi_1$ and $\varphi_2$ hold, which has more restriction than the other derived Boolean operators.

The operator "until" $U$ is right-associative, i.e., $\varphi_1 U \varphi_2 U \varphi_3 \Leftrightarrow \varphi_1 U (\varphi_2 U \varphi_3)$; the unary operators are also right-associative; all the other operators are left-associative.

**Discussion.** *The right-associative property of the "until" operator $U$ somehow indicates that for a control specification described by the LTL formula, we should start from right to left to solve the problem. This indication also holds for the derived temporal operators $F$ ("eventually" operator) and $G$ ("always" operator), and the joint temporal operators $GF$ and $FG$.*

*From the perspective of graph searching problem, we should search the graph backwards through the in-going edges. We should start our searching from the right-most objective to the left. It may require repetitive searching until fixed point iteration for the specifications involving $GF$ and $FG$.*

### 3.3.3 Derived Operators

In this subsection, we will discuss the derived propositional operators, the derived temporal operators, and the joint temporal operators.

**Derived Propositional Operators**

The full power of propositional logic can be obtained by using only the Boolean connectors $\wedge$ and $\neg$: other common Boolean connectors such as disjunction $\vee$, implication $\rightarrow$, equivalence (not exclusive or) $\leftrightarrow$, and parity (exclusive or) $\oplus$ can be derived as follows:

$$\begin{aligned}
\varphi_1 \vee \varphi_2 &:= \neg(\neg\varphi_1 \wedge \neg\varphi_2) && \text{disjunction (“or”)} \\
\varphi_1 \rightarrow \varphi_2 &:= \neg\varphi_1 \vee \varphi_2 && \text{implication} \\
\varphi_1 \leftrightarrow \varphi_2 &:= (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) && \text{equivalence (“xnor”)} \\
\varphi_1 \oplus \varphi_2 &:= (\varphi_1 \wedge \neg\varphi_2) \wedge (\varphi_2 \wedge \neg\varphi_1) && \text{parity (“xor”).}
\end{aligned}$$

**Discussion.** *A tricky thing about using LTL formulas to describe the specification of a graph searching problem is that, the propositions composed of atomic propositions that are equivalent may not be equivalent for the sets composed of the set of nodes labeled by these atomic propositions.*

*For example, let $A_1$, $A_2$ be the set of nodes labeled by the atomic propositions $a_1$ and $a_2$ respectively. In propositional logic, we have $a_1 \rightarrow a_2 \equiv \neg a_1 \vee a_2 \equiv (a_1 \wedge a_2) \vee \neg a_1$. For the sets, we have $\bar{A}_1 \cup A_2 \equiv (A_1 \cap A_2) \cup \bar{A}_1$. $A_1 \cap A_2$ and $\bar{A}_1$ are distinct, whereas $\bar{A}_1$ and $A_2$ may have intersection. Therefore, for an LTL formula such as $\varphi = GFa_1 \rightarrow GFa_2$ that requires repetitive searching, the two different interpretations of implications “$\rightarrow$” from proposition to set may result in different winning sets. The reason is that the one with intersection has “larger” sets than the two sets that are distinct, thus may have larger winning sets when we require repetitive searching.*

**Derived Temporal Operators**

There are two important dual temporal operators that can be derived from the “until” operator $U$:

1. the “eventually” operator $\Diamond$ or $F$ (finally): $F\varphi := 1U\varphi$;
2. and the “always” operator $\Box$ or $G$ (globally): $G\varphi := \neg F\neg\varphi$.

“Eventually”, aka “finally”, stands for the current moment or some future moment; and “always”, aka “globally”, stands for from the current moment on and forever.

We choose to use $F$ and $G$ to express the "eventually" operator and the "always" operator instead of using $\Diamond$ and $\Box$ for the reason that the former is more readable and contains more effective information than the later.

$F\varphi$, reads "eventually $\varphi$", means that $\varphi$ will eventually be satisfied. More specifically, $F\varphi$ holds at the current moment if $\varphi$ holds at the current moment or some future moment. Since $Fa$ can be interpreted as eventually reach a state labeled with $a$, $Fa$ is also known as the reachability objective.

$G\varphi$, reads "always $\varphi$", means that $\varphi$ is always satisfied. More specifically, $G\varphi$ holds at the current moment if $\varphi$ holds from the current moment on and forever. In other words, $G\varphi$ holds if eventually $\neg\varphi$ never happens. If we consider the states labeled with $a$ as the safe region, then $Ga$ can be interpreted as always stay in the safe region. That's why $Ga$ is also known as the safety objective.

**Joint Temporal Operators**

$F$ and $G$ are often jointly used together as combinations to obtain two new temporal modalities: $GF$ ("always eventually", aka Büchi) and $FG$ ("eventually always", aka co-Büchi).

$GF\varphi$, reads "always eventually $\varphi$", means that $\varphi$ will eventually be satisfied for a first time, and then it will be satisfied for a second time, a third time, $\cdots$, up to infinitely many times. Therefore, "always eventually $\varphi$" can also be phrased as "infinitely many often satisfy $\varphi$". $GFa$ can thus be interpreted as always eventually reach, or infinitely many often reach the states labeled with $a$, which is also known as the Büchi objective. Since $GFa$ requires reaching $a$ infinitely many times, the Büchi objective is stronger than the reachability objective $Fa$ which requires reaching $a$ only once.

$FG\varphi$, reads "eventually always $\varphi$", means that $\varphi$ will eventually always be satisfied, which allows a finitely many violations of $\varphi$. Therefore, "eventually always $\varphi$" can also be phrased as "finitely many often violate $\varphi$". $FGa$ can thus be interpreted as eventually always stay in the states labeled with $a$, or finitely many often reach the states labeled with $\neg a$, which is also known as the co-Büchi objective. Since $FGa$ allows reaching $\neg a$ finitely many times, the co-Büchi objective is weaker than the safety objective $Ga$ which requires always staying in $a$.

### 3.3.4 Conversion from LTL Formula to Automaton

Since we leave the conversion from LTL formula to automaton to the automation process, we will only give the brief idea

**Length of a Formula**

The length of an LTL formula $\varphi$, denoted $|\varphi|$, is the number of operators in $\varphi$. Since only the asymptotic size $\Theta(|\varphi|)$ is needed in the complexity analysis, we directly count the number of operators appearing in $\varphi$ without restricting the use of the original operators defined in the syntax.

**Conversion from LTL Formula to Automaton**

An LTL formula $\varphi$ can be algorithmically converted into a NBA. The intermediate step is to construct a generalized non-deterministic Büchi automaton (GNBA), and then convert the GNBA into an equivalent NBA. The semantics of $\varphi$ is encoded in the states and the transitions of the automaton.

**Benefits of Automatic Conversion**

We prefer to use a computer to convert an LTL formula to a DA instead of by human for the following reasons:

- it is complicated and easy to make mistakes to do the conversion by human;
- the automation process provides a systematic approach to do the conversion;
- we can encode all the optimizations in the automation process according to the arithmetics of the LTL to reduce the size of the automaton to increase the efficiency.

**Discussion.** *After $\varphi$ is converted into an NBA, we try to determinize it into a DBA, which is not always possible. However, we can always determinize an NBA into a deterministic Rabin automaton (DRA) or a deterministic Streett automaton (DSA). The trade-off is that, a Rabin game or a Streett game is much more complicated and has much higher time complexity than a Büchi game.*

**Remark.** *The conversion from an LTL formula $\varphi$ to an NBA can lead to exponential blowup w.r.t. the length of $\varphi$.*

### 3.3.5 LTL and DA in Control Setting

**Human and Computer**

We would like to combine the strong points of human and computer to help us solve the control problem.

Humans are good at reasoning, thus it is relatively easy for us to convert our control specification from human language to an LTL formula.

Computers are good at computing in graph structure, thus a graph based automaton is friendly to computers. The merit of an automaton is that it can track the status of an agent in the NTS in an infinite process using only a small finite graph.

**Connection between LTL and DA**

Our particular interest in the logic languages such as LTL is that, an LTL formula $\varphi$ can be converted into an automaton in an automation process. This allows us to formulate the control specification as an LTL formula $\varphi$, and then leave the rest of converting $\varphi$ to a DA with a corresponding accepting condition to the computer.

Once the LTL formula is converted to a DA, we try to solve the control problem as a graph searching problem.

**What is our control problem?**

The trajectory of the agent $\rho_x$ satisfies the accepting condition of the LTL formula $\varphi$ if its input word $\alpha_l$ satisfies the accepting condition of the DA converted from $\varphi$. Our control problem is to find the winning set and the winning strategy such that all the trajectory initiated in the winning set induced by the winning strategy satisfy the accepting condition of $\varphi$ regardless of the non-determinism of the NTS.

**How do we solve it?**

We can see that there is a hidden product graph behind the NTS and the DA. Therefore, we enumerate all the possible paths of the agent, token pair to construct the product graph. We take the product of the NTS and the DA, and then convert the control problem to an infinite two player game in the product arena converted from the product NTS. The winning condition of the game corresponds to the accepting condition of the DA. We then use the corresponding algorithm of the game to solve the winning set and the winning strategy by computer.

**Discussion.** *The product graph connects the NTS and the DA, and once we convert the logic from an LTL formula to a graph as DA, the control problem can be automatically solved by the computer as a graph searching problem.*

*For the conversion from an LTL formula $\varphi$ to a DA, even there may be an exponential blow up w.r.t. $|\varphi|$ in theory, this is actually not the case in practice for the following reasons:*

- *the length of $\varphi$ is short;*

- *the automation tool can simplify the conversion in both space and time.*

*The size of the NTS is much larger than the size of the DA, thus the size of the NTS is the dominating factor.*

*An assumption for our control problem to be feasible to solve is that, the size of the DA is small, and the game in the product space can be solved in almost linear time; otherwise the control problem is not feasible to solve as a result of the large size of the NTS.*

**Remark.** *Since we don't really have an infinite run, the ability of a DA to keep track of the status of the agent in the NTS is limited. So how can we tell whether a trajectory satisfies the accepting condition of the LTL formula $\varphi$? This is done by mapping the accepting condition of the DA converted from $\varphi$ to the winning condition of an infinite game in the product space. The satisfaction of the trajectory is constructed and showed inductively in solving the game.*

# Chapter 4

# Formulation of Infinite Games

In this chapter, we will introduce the infinite two player games on directed graphs. We will define what a game is, how it is played, when a player wins, what his winning strategy is, etc. We will introduce the properties such as determinacy, time independent strategies and duality. For more about the formulation of infinite games, one may refer to [15, Chapter 2].

## 4.1 Game

A game is comprised of an arena and an objective set. The arena is the environment where the game is played, and the objective set defines the winning condition the game. We will first study the arena and then add the different objectives on top of the arena.

### 4.1.1 Arena

**Definition 4.1.1** (arena)**.** *An arena $\mathcal{A} = (V, E)$ is a pair of nodes $V = V_1 \oplus V_2$ and edges $E \subseteq V \times V$.*

We use "arena" in short for a two player game graph.

We abuse the notation of $\mathcal{A}$ for both automaton and arena for the reason that the winning condition of a two player game is defined analogously to the accepting condition of an automaton: a run is accepted by an automaton $\mathcal{A}$ iff it satisfies the accepting condition

of $\mathcal{A}$, whereas a play wins a game in an arena $\mathcal{A}$ iff it satisfies the winning condition of the game.

An infinite two player game is played by two players, named player 1 and player 2. Since the two players are dual to each other, we will simply fix $i \in \{1, 2\}$ when consider player $i$, and the other player is consequently denoted as $\bar{i} = 3 - i$.

An arena is a finite directed graph where the nodes are partitioned into player 1 nodes ($\bigcirc$) and player 2 nodes ($\Box$), i.e., $V = V_1 \oplus V_2 \quad \equiv \quad (V_1 \cup V_2 = V) \wedge (V_1 \cap V_2 = \emptyset)$. In the general case, we don't restrict our arena to be a bipartite graph with the corresponding partition $V = V_1 \oplus V_2$. We will show later, that an NTS can naturally be represented as a bipartite arena. More discussions between control problems in NTS and two player games in arena will be saved for later when the entire game is introduced.

**Remark.** *We allow self-loops (A.1.4) but not multi-edges (A.1.5) in an arena.*

*We also require each node to have at least one out-going edge to guarantee that the infinite flow of the game would never terminate.*

### 4.1.2 Play

Recall that in the graph theory, a path is a sequence of nodes, which can be both finite and infinite. Here, the play is the same as the path in the game setting. A play is a sequence of arena nodes, which can also be both finite and infinite. Since our focus is on the infinite games, the play we indicate is an infinite play if not specified.

A finite play $p = p(0)p(1)\cdots p(n) = v_0 v_1 \cdots v_n \in V^*$ is a finite sequence of arena nodes, where $p(0) = v_0$ is the initial node and $p(j+1) \in Post(p(j))$, $0 \leq j < n$, $|p| = n$.

An infinite play $\pi = \pi(0)\pi(1)\cdots = v_0 v_1 \cdots \in V^\omega$ is an infinite sequence of arena nodes, where $\pi(0) = v_0$ is the initial node and $\pi(j+1) \in Post(\pi(j))$, $j \geq 0$, $|\pi| = \infty$.

After we have introduced what a play is, we can further introduce how an infinite play is played by the two players:

- we start the game by placing a token on an initial node $v_0$, thus $\pi(0) = v_0$;
- if the token is on a player $i$ node, then it is player $i$'s turn to play the game;
- player $i$ plays the game by moving the token to one of the successors of $v_0$, i.e., $\pi(1) \in Post(v_0)$;

- this process continues as when the node $\pi(j) = v_j$ that the token is on is a player $i$ node, player $i$ plays the game by moving the token to one of the successors of $v_j$, i.e., $\pi(j+1) \in Post(v_j)$.

More precisely, an infinite play is induced by:

1. Initially placing the token on $\pi(0) = v_0$;
2. if $\pi(j) = v_j \in V_i$, then player $i$ moves the token to $\pi(j+1) = v_{j+1} \in Post(v_j)$, $j \geq 0$, $i \in \{1, 2\}$.

A play is consequently the path traveled by the token in a game.

When the token is on a player $i$ node $v \in V_i$, the nodes $v'$ that player $i$ can move the token to is defined by the edge relation of the arena $v' \in Post(v)$. Moving the token each time is also called making a move. A special thing about a bipartite arena is that the two players take turns to make a move.

### 4.1.3 Objective Set

We use Obj $\subseteq 2^V$ to denote the objective set in an arena $\mathcal{A}$ that contains one or more labeled sets. These labeled sets are used to define the winning condition of the game.

For example, in a Büchi game, Obj $= T$, where $T$ is the target set. For player 1, the winning condition is to place the token infinitely many often on a node in $T$. Player 2 wins the game if the objective for player 1 is not achieved.

Based on the arena and the objective set of a game, now we can define what a game is.

**Definition 4.1.2** (game)**.** *A game $\mathcal{G} = (\mathcal{A}, Obj)$ is a pair of arena and objective set, where $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$ is the arena of the game, and $Obj \subseteq 2^V$ is the objective set used to define the winning condition of the game.*

### 4.1.4 Winning Condition of a Play

Player $i$ declares to be the winner of a play $\pi$ in the game $\mathcal{G}$ iff $\pi$ satisfies the winning condition of $\mathcal{G}$ for player $i$. In other words, the play $\pi$ is a winning play for player $i$ in a game $\mathcal{G}$ iff $\pi$ satisfies the winning condition of $\mathcal{G}$ for player $i$.

The winning condition of a play $\pi$ in a game $\mathcal{G}$ is always positively defined for player 1; player 2 is the winner of $\pi$ in $\mathcal{G}$ if $\pi$ doesn't satisfy the winning condition for player 1. Therefore, the infinite two player games we consider are all 0-sum games with no ties.

The winning condition of a play in a game corresponds to the accepting condition of a run in an automaton. The winning condition of a play in a game is defined as a proposition using the occurrence set $\mathrm{Occ}(\pi)$ or the infinity set $\mathrm{Inf}(\pi)$, and the objective set Obj.

**Finite Game**

The first winning condition is for the finite reachability game, where the play $p$ is finite. The finite reachability game can be extended to an infinite reachability game by extending the finite play $p$ to an infinite play $\pi$ according to $\pi(i+1) \in \mathit{Post}(\pi(i))$ once the target set is reached.

For a finite reachability game, $\mathrm{Obj} = T \subseteq V$ is the set of target nodes. A finite play $p$ wins the reachability game iff one of the nodes in $T$ occurs in $p$. Formally, $p$ wins iff $\mathrm{Occ}(p) \cap T \neq \emptyset$.

The finite reachability game corresponds to the finite automaton $FA$, whereas the target set $T$ in the reachability game corresponds to the final set $F$ in $FA$.

**Infinite Game**

The rest of the winning conditions are for the infinite games, where the play $\pi$ is infinite.

We use $G$ to denote "generalized" for a game with more than one objective. We use $B$, $C$, $R$, and $S$ to denote "Büchi", "co-Büchi", "Rabin", and "Streett" respectively. These winning conditions enjoy a dual property, i.e., Büchi and co-Büchi, generalized Büchi and generalized co-Büchi, Rabin and Streett are all mutually dual to each other, which can be seen from how they are defined.

The duality allows us to look into a problem from both sides, which gives us a clearer picture of the game. For example, when the algorithm that we use to solve for the winning set of a game $\mathcal{G} = (\mathcal{A}, \mathrm{Obj})$ terminates, the winning set of the two players $\mathcal{W}_1$ and $\mathcal{W}_2$ will partition the nodes of the arena $\mathcal{A} = (V, E)$, i.e., $V = \mathcal{W}_1 \oplus \mathcal{W}_2$. This also helps us to understand why an algorithm has a certain worst case time complexity, and why this worst case time complexity is not likely to reduce.

The duality also allows us to consider only the behaviour of player 1 instead of also looking into the behaviour of player 2, which simplifies the analysis by half. This is because the behaviour of the player 2 in a game $\mathcal{G}$ can be seen as the behaviour of the player 1 in its dual game $\bar{\mathcal{G}}$.

A special case, also the simplest case of duality is between a finite reachability game and an infinite safety game. The analysis for the later more complicated games are all based on the dual property between these two games.

- **Safety Game** For a safety game, Obj $= S \subseteq V$ is the set of safe nodes and correspondingly, $V/S$ is the set of unsafe nodes. An infinite play $\pi$ wins the safety game iff only the nodes in $S$ occur in $\pi$, or equivalently, the nodes in $V/S$ never occur in $\pi$. Formally, $\pi$ wins iff $\mathrm{Occ}(\pi) \subseteq S$, or equivalently, $\mathrm{Occ}(\pi) \cap (V/S) = \emptyset$.

- **Büchi Game** For a Büchi game $B\mathcal{G}$, Obj $= T \subseteq V$ is the set of target nodes. An infinite play $\pi$ wins the $B\mathcal{G}$ iff at least one of the nodes in $T$ occurs infinitely many often in $\pi$. Formally, $\pi$ wins iff $\mathrm{Inf}(\pi) \cap T \neq \emptyset$.

  The Büchi game $B\mathcal{G}$ corresponds to the Büchi automaton $BA$, whereas the target set $T$ in $B\mathcal{G}$ corresponds to the accepting set $F$ in $BA$.

- **co-Büchi Game** For a co-Büchi game $C\mathcal{G}$, Obj $= S \subseteq V$ is the set of safe nodes and correspondingly, $V/S$ is the set of unsafe nodes. An infinite play $\pi$ wins the $C\mathcal{G}$ iff only the nodes in $S$ occur in $\pi$ eventually, or equivalently, the nodes in $V/S$ occur only finitely many often in $\pi$. Formally, $\pi$ wins iff $\mathrm{Inf}(\pi) \subseteq S$, or equivalently, $\mathrm{Inf}(\pi) \cap (V/S) = \emptyset$.

  The co-Büchi game $C\mathcal{G}$ corresponds to the co-Büchi automaton $CA$, whereas the safe set $S$ or the unsafe set $V/S$ in $C\mathcal{G}$ corresponds to the accepting set or the rejecting set $F$ in $CA$;

- **Generalized Büchi Game** For a generalized Büchi game $GB\mathcal{G}$, Obj $= \{T_1, \cdots, T_k\}$, where $T_i \subseteq V$, $1 \leq i \leq k$ are the $k$ sets of target nodes. An infinite play $\pi$ wins the $GB\mathcal{G}$ iff at least one of the nodes in $T_i$ occurs infinitely many often in $\pi$ for all $i$, $1 \leq i \leq k$. Formally, $\pi$ wins iff $\bigwedge_{i=1}^{k} (\mathrm{Inf}(\pi) \cap T_i \neq \emptyset)$.

  The generalized Büchi game $GB\mathcal{G}$ corresponds to the generalized Büchi automaton $GBA$, whereas the $k$ target sets $T_1, \cdots, T_k$ in $GB\mathcal{G}$ corresponds to the $k$ accepting sets $F_1, \cdots, F_k$ in $GBA$.

- **Generalized co-Büchi Game** For a generalized co-Büchi game $GC\mathcal{G}$,

  Obj $= \{S_1, \cdots, S_k\}$, where $S_i \subseteq V$, $1 \leq i \leq k$ are the $k$ sets of safe nodes and correspondingly, $\{V/S_1, \cdots, V/S_k\}$ are the $k$ sets of unsafe nodes. An infinite play

$\pi$ wins the $GC\mathcal{G}$ iff only the nodes in $S_i$ occur in $\pi$ eventually for some $i$, $1 \leq i \leq k$, or equivalently, the nodes in $V/S_i$ occur only finitely many often in $\pi$ for some $i$, $1 \leq i \leq k$. Formally, $\pi$ wins iff $\bigvee_{i=1}^{k} (\text{Inf}(\pi) \subseteq S_i)$, or equivalently, $\bigvee_{i=1}^{k} (\text{Inf}(\pi) \cap (V/S_i) = \emptyset)$.

The generalized co-Büchi game $GC\mathcal{G}$ corresponds to the generalized co-Büchi automaton $GCA$, whereas the $k$ safe sets $S_1, \cdots, S_k$, or the $k$ unsafe sets $V/S_1, \cdots, V/S_k$ in $GC\mathcal{G}$ corresponds to the $k$ accepting sets or the $k$ rejecting sets $F_1, \cdots, F_k$ in $GCA$.

- **Rabin Game** For a Rabin game $R\mathcal{G}$, Obj $= \{(G_1, B_1), \cdots, (G_k, B_k)\}$, where $G_i, B_i \subseteq V$, $1 \leq i \leq k$. "G" refers to the set of "Good" nodes that we would like to occur infinitely many often in a play $\pi$, whereas "B" refers to the set of "Bad" nodes that we would like to occur only finitely many often in $\pi$. An infinite play $\pi$ wins the $R\mathcal{G}$ iff at least one of the nodes in $G_i$ occurs infinitely many often in $\pi$ and the nodes in $B_i$ occur only finitely many often in $\pi$ for some $i$, $1 \leq i \leq k$. Formally, $\bigvee_{i=1}^{k} ((\text{Inf}(\pi) \cap G_i \neq \emptyset) \wedge (\text{Inf}(\pi) \cap B_i = \emptyset))$.

  The Rabin game $R\mathcal{G}$ corresponds to the Rabin automaton $RA$, whereas the $k$ Rabin pairs Obj $= \{(G_1, B_1), \cdots, (G_k, B_k)\}$ in $R\mathcal{G}$ corresponds to the $k$ Rabin pairs Acc $= \{(G_1, B_1), \cdots, (G_k, B_k)\}$ in $RA$.

- **Streett Game** For a Streett game $S\mathcal{G}$, Obj $= \{(G_1, B_1), \cdots, (G_k, B_k)\}$, where $G_i, B_i \subseteq V$, $1 \leq i \leq k$. "G" refers to the set of "Good" nodes that we would like to occur infinitely many often in a play $\pi$, whereas "B" refers to the set of "Bad" nodes that we would like to occur only finitely many often in $\pi$. An infinite play $\pi$ wins the $S\mathcal{G}$ iff at least one of the nodes in $G_i$ occurs infinitely many often in $\pi$ or the nodes in $B_i$ occur only finitely many often in $\pi$ for all $i$, $1 \leq i \leq k$. Formally, $\bigwedge_{i=1}^{k} ((\text{Inf}(\pi) \cap G_i \neq \emptyset) \vee (\text{Inf}(\pi) \cap B_i = \emptyset))$.

  The Streett game $S\mathcal{G}$ corresponds to the Streett automaton $SA$, whereas the $k$ Streett pairs Obj $= \{(G_1, B_1), \cdots, (G_k, B_k)\}$ in $S\mathcal{G}$ corresponds to the $k$ Streett pairs Acc $= \{(G_1, B_1), \cdots, (G_k, B_k)\}$ in $SA$.

## 4.2 Winning Condition of a Game

In this section, we will introduce the winning condition of a game, and why it is difficult to determine the winner of a play.

The winning condition of a game indicates the objective of the game that we are playing. For example, we are playing the Büchi game if the winning condition is a Büchi condition.

The winning condition of a game is based on the winning condition of a play.

Since we always stand on the behalf of player 1, the winning condition is positively defined for player 1, and the winning condition for player 2 is just the complement of player 1. For this reason, the winning condition of a game refers to the winning condition for player 1 of the game for simplicity.

### 4.2.1 Problem Formulation of Infinite Game

In the previous section, we introduced the concept of game and the winning condition of a play in different games. But what exactly are we solving for in each game? This leads to the problem formulation of an infinite game.

**Problem Formulation of Infinite Game**

Given an infinite game $\mathcal{G} = (\mathcal{A}, \text{Obj})$, where $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$ is the arena and $\text{Obj} \subseteq 2^V$ is the objective set used to define the winning condition of $\mathcal{G}$, find the winning set $\mathcal{W}_1 \subseteq V$ and the winning strategy $\Pi_1$ for $v \in V_1 \cap \mathcal{W}_1$ for player 1 such that all the plays $\pi = v_0 v_1 v_2 \cdots$ induced by the initial node $v_0 \in \mathcal{W}_1$, $\Pi_1$, and $\Pi_2$ inductively satisfy the winning condition of $\mathcal{G}$ regardless of $\Pi_2$, where $\Pi_2$ is any strategy for player 2.

From the problem formulation, we can see that we want to find the winning set $\mathcal{W}_1$ and the winning strategy $\Pi_1$ for player 1 in each infinite game. To fully understand what the problem formulation is talking about, we need to introduce three concepts: strategy, determinacy, and duality.

In the next subsection, we will discuss why it is difficult to determine the winner of a play directly, which lead us to the concept of strategy.

To understand why player 1 is the determined winner of all the plays initialized at $v_0 \in \mathcal{W}_1$ induced by the winning strategy $\Pi_1$ for player 1 and any strategy for player 2 $\Pi_2$, we need to introduce the concept of determinacy.

With the concept of strategy and determinacy, we can define the winning set.

To understand why we only need to consider the winning set and the winning strategy of player 1 in each game, we need to introduce the concept of duality.

### 4.2.2 Difficulty in Determining the Winner of a Play

In the previous section, we have introduced the winning play and the different winning conditions of a play in different games. Since the most of the games we consider are infinite games, the infinity set of a play is hard to define in practice since we can only have finite information of the play.

Consider an infinite play $\pi$ in an infinite game $\mathcal{G} = (\mathcal{A}, \text{Obj})$. Recall that the infinity set of an infinite play $\pi \in V^\omega$ is the set of nodes $v \in V$ occurring infinitely many often in $\pi$, where $V$ is the set of nodes in the arena $\mathcal{A}$.

Formally, $\text{Inf}(\pi) = \{v \in V | \forall i \geq 0, \exists j \geq i \text{ s.t. } \pi(j) = v\}$.

Assume the Game $\mathcal{G}$ starts at moment 0, and the two players make a move at each moment. Then at the $j^{th}$ moment, we only have the information of the infinite play $\pi$ from moment 0 to moment $j$, i.e., a finite sub-play $p = \pi(0)\pi(1)\cdots\pi(j)$.

Since we only have the information of the history of the play $\pi$, but not the information of the future of $\pi$, it is not possible to determine $\text{Inf}(\pi)$, which is closely related to the winning conditions of a play. Therefore, it is likely to be either hard to verify, or hard to falsify the winning conditions of a game, which makes the winner of the play undetermined.

For a play $\pi$ in a game $\mathcal{G}$, there are two exceptions:

1. in a reachability game, if the token reaches a target node, then player 1 wins the game; otherwise the winner of $\pi$ is undetermined;

2. in a safety game, if the token reaches an unsafe node, then player 2 wins the game; otherwise the winner of $\pi$ is undetermined.

We can see that the two exceptions are dual to each other, and the winner is still undetermined if the token hasn't reached a target node in a reachability game or the token hasn't reached an unsafe node in a safety game.

Therefore, in order to determine the winner of an infinite game, we need to introduce two new concepts, strategy and determinancy to resolve the problem of the infinity set of the play.

We are particularly interested in whether the winner of a play $\pi$ in a game $\mathcal{G}$ is already determined in the first place as early as the token is placed on an initial node $v_0 \in V$. We claim the answer is positive, which means if player $i$ is the determined winner of all the plays initiated at $\pi(0) = v_0 \in V$, then there exists a strategy for player $i$ such that regardless of any strategy player $\bar{i}$ plays, all the induced plays are winning plays for player $i$. Player $i$ can satisfy the winning condition of the play $\pi$ because by following the strategy for player $i$, we can inductively show that the infinity set in the winning condition can be satisfied again and again, up to infinitely many times, which resolves the problem of $\mathrm{Inf}(\pi)$.

Before looking deep into strategy and determinancy, we will show an example of a finite game.

**Number Counting Game**

In a number counting game, the two players take turns to count numbers from 1 to $n$. Each time a player must count 1 number up to $a$ numbers, where $1 \leq a \leq n$. The player who counts the number $n$ wins the game.

The idea of the strategy for the two players are constructed in the same way: forcing the other player into an $a + 1$ loop.

Suppose player 1 counts first.

If $n = a$, then player 1 wins the game by counting from 1 to $a$.

If $n = a + 1$, then player 2 can always count the number $a + 1$ and wins the game. The strategy for player 2 is as follows: suppose player 1 counts $b$ numbers from 1 to $b$, where $1 \leq b \leq a$, then player 2 counts $a + 1 - b$ numbers from $b + 1$ to $a + 1$ and wins the game. We can see that the latter player who counts in the $a + 1$ loop wins the game.

If $n = m(a + 1)$, $m > 0$, then player 2 wins the game by following the strategy in the $n = a + 1$ case, because player 2 is the latter player who counts in the $a + 1$ loops.

If $n = m(a + 1) + k$, $m > 0, 1 \leq k \leq a$, then player 1 wins the game by first counting from 1 to $k$, then the game converts to counting from $k + 1$ to $n = m(a + 1) + k$, with player 2 counting first. Now player 1 is the latter player who counts in the $a + 1$ loops and wins the game.

The determinancy of a game means that, given a certain condition of the game, the winner of the game is determined if this player plays correctly and the other player will always lose even before the game is played.

In the number counting game, the determinancy for $n = m(a+1)+k, m \geq 0, 0 \leq k \leq a$ is as follows: when $k = 0$, the player who counts first wins; when $1 \leq k \leq a$, the player who counts first loses.

We can generalize the game a little bit by setting different counting numbers for the two players: one player must count 1 number up to $a$ numbers each time, the other player must count 1 number up to $b$ numbers each time.

If we let $1 \leq b < a \leq n$, then the player who can count more numbers each time always has a winning strategy. The player who can count $a$ numbers each time can always force the other player into a $b+1$ loop no matter who counts first, because this player can count from 1 number up to $a \geq b+1$ numbers.

### 4.2.3 Strategy

In a two player game, it is player $i$'s turn to move the token if the token is placed on a player $i$ node. Therefore, the strategy of player $i$ is the guide or the rule for player $i$ to move the token when it is his turn.

Consider a play $\pi$ in a game $\mathcal{G}$. Suppose at the current moment $j \geq 0$, $\pi(j) \in V_i$, then the token is on a player $i$ node and it is player $i$'s turn to make a move. The strategy for player $i$ at the moment $j$ can at most take reference of the finite sub-play $p = \pi(0)\pi(1)\cdots\pi(j)$. Therefore, the input for the strategy is a sub-play of $p$.

In general, the strategies are time dependent, which requires the information of the history of the play to make a decision for the next move.

**Definition 4.2.1** (time dependent strategy)**.** *For a play $\pi$ in a game $\mathcal{G}$, a time dependent strategy for player $i \in \{1,2\}$ at moment $j$ is a function $\Pi_i : V^* \to V$, where $p' \in V^*$ is a sub-play of $p = \pi(0)\pi(1)\cdots\pi(j)$, $\pi(j) \in V_i$. The memory of the strategy is defined as the size of the input of the function, i.e., $1 \leq |p'| \leq j$.*

If $\Pi_i$ takes exactly $p = \pi(0)\pi(1)\cdots\pi(j)$ at moment $j$, then $|p| = j$, and as $j$ goes to infinity, the memory of the strategy for player $i$ at moment $j$ goes to infinity, which is infeasible in practice.

Therefore, we require the memory of our strategy to be finite, i.e., doesn't grow linearly w.r.t. the length of the play.

All the games we consider have finite memory for strategy.

#### Time Independent Strategy

In this subsubsection, we will discuss the time independent strategy.

A common and maybe the most ideal case of strategy is when the strategy is independent of the history of the play, i.e., the player only needs the information of the node at the current moment where the token is on to decide which node to go to for the next moment.

From the time perspective, such strategy is memoryless because it is independent of the past.

From the space perspective, such strategy is also local, or positional since we only need the information of the current node where the token is on to decide which node to go to next.

When we talk about "memory", it can indicate a process is time dependent, or it can indicate the amount of space required to do a job in computer science. Therefore, to avoid the ambiguity, we will only use the later meaning of "memory", and call the time independent strategy as "local" or "positional" instead of "memoryless".

**Definition 4.2.2** (time independent strategy)**.** *A time independent strategy for player* $i \in \{1, 2\}$ *is a function* $\Pi_i : V_i \to V$*. Such strategy is also called local, positional, or memoryless.*

For player 1, games with one objective such as reachability, safety, Büchi, and co-Büchi, have local strategy; games with more than one objective such as generalized co-Büchi and Rabin also have local strategy since these two objectives are of an "or" condition, meaning that we only need to satisfy one of the $k$ objectives to satisfy the winning condition. Therefore, each node only focusing on satisfying one objective is sufficient to satisfy the winning condition.

### Time Dependent Strategy

In this subsubsection, we will discuss the time dependent strategy.

For player 1, games with more than one objective such as generalized Büchi and Streett require a more complicated structure of strategy that is time dependent.

Since these two objectives are of an "and" condition, meaning that we need to satisfy all the $k$ objectives to satisfy the winning condition.

If we would like to satisfy $k$ objectives infinitely many often, then a naive strategy is to repeatedly satisfy the objectives from objective 1 to objective $k$ in order.

Therefore, we need to keep track of which of the $k$ objectives is to satisfy at the current moment. Since there are $k$ objectives to satisfy, we need to store $k$ local strategies at each node.

From the general definition of the strategy of player 1 at moment $j$, we need to at least store a sub-play of $\pi$ containing the satisfaction of the latest $k-1$ objectives to guide the player to choose the strategy for the next objective. Once which of the $k$ objectives is to satisfy is determined, the strategy is again local. We can now see that the time dependent part for the strategy comes from determining which of the $k$ objectives is to satisfy.

In practice, we can simply reduce the time and space complexity of storing a sub-play of $\pi$ containing the satisfaction of the latest $k-1$ objectives by a counter indicating which of the $k$ objectives is to satisfy: when the counter $count = j$, it means we would like to satisfy the $j^{th}$ objective; once the $j^{th}$ objective is satisfied, it increments to $count = j+1$. Whether the counter repeats from 1 to $k$ or simply increments and then $mod\ k$ when in use is a choose of design.

We can see that even in games like generalized Büchi, the strategy for player 1 still only consists of $k$ local strategies for the $k$ objectives at each node, with an additional counter indicating which of the $k$ objectives is to satisfy. Therefore, the time dependent strategy we consider is still almost time independent: the time dependent strategy consists of a time independent strategy where each node takes $\mathcal{O}(k)$ space, and the additional dynamic counter takes $\mathcal{O}(1)$ space, and $\mathcal{O}(1)$ time each time to determine which local strategy to choose.

The winning condition of the Streett game with $k$ Streett pairs requires satisfying the $k$ Streett conditions at the same time. Each Streett condition is in the form of a disjunction rather than a single case in the generalized Büchi. Therefore, the strategy for the Streett game with $k$ Streett pairs is even much more complicated than that of the generalized Büchi, since it also needs to coordinate the conflicts among the $k$ Streett conditions. In other words, we can easily construct a local strategy for each Streett condition, but the satisfaction of one Streett condition may violate the other. In the worst case, the conflict exists only if we consider the $k$ Streett conditions together. Therefore, there doesn't exist a simple local strategy for each Streett condition, and we will omit the discussion for the $k$ pairs Streett game here. In fact, the theoretical upper bound of the local strategy for the $k$ pairs Streett game can be up to $\mathcal{O}(k!)$ due to the permutation [16], [28].

### Winning Strategy

In this subsubsection, we will discuss the winning strategy.

The winning strategy of a player is defined for an initialized play $\pi$ with $\pi(0) = v_0 \in V$.

If a play $\pi$ starts with $\pi(0) = v_0 \in V$, we say $\pi$ is an initialized play at $v_0$.

**Definition 4.2.3** (winning strategy)**.** *For an initialized play $\pi$ with $\pi(0) = v_0 \in V$ in a game $\mathcal{G}$, a strategy $\Pi_i$ is a winning strategy for player $i$ if any play $\pi$ induced by $\pi(0) = v_0$, $\Pi_i$, and $\Pi_{\bar{i}}$ is a winning play for player $i$, where $\Pi_{\bar{i}}$ is any strategy for player $\bar{i}$.*

It is clear that if player $i$ has a winning strategy $\Pi_i$ for an initialized play $\pi$ at $v_0$, then player $i$ is the determined winner for any play $\pi$ initialized at $v_0$ by following $\Pi_i$.

What remains to discuss is whether there is a determined winner for a play $\pi$ initialized at each $v_0 \in V$ in a game $\mathcal{G}$, which leads to the discussion of determinacy.

### 4.2.4 Determinacy

The determinacy of a game $\mathcal{G}$ tells us that, given certain conditions of the game, the winner of the game is determined even before the game is actually played.

In other words, if a game $\mathcal{G}$ given certain conditions has a determined winner player $i$, then player $i$ has a winning strategy $\Pi_i$ such that any play $\pi$ induced by $\Pi_i$ and $\Pi_{\bar{i}}$ is a winning play for player $i$, where $\Pi_{\bar{i}}$ is any strategy for player $\bar{i}$.

A game is undetermined if the winner of the game can't be predicted, i.e., there exists a winning strategy for neither player 1 nor player 2.

The condition we consider in the discussion of determinacy of a game is to initialize our play $\pi$ at each $v_0 \in V$ in a game $\mathcal{G}$.

A game $\mathcal{G}$ initialized at $v_0 \in V$ is a determined game if it has a determined winner.

A game $\mathcal{G}$ is a determined game if each initialization of the game $v_0 \in V$ has a determined winner.

### 4.2.5 Winning Set

The winning set of a game is defined based on the concepts of strategy and determinancy. The strategy of a player $i$ guides the player to move the token when the token is on a player $i$ node. The determinancy shows that the winner of all the plays initialized at each node $v_0 \in V$ is determined if that player plays correctly.

The winning set of a game $\mathcal{G}$ is defined by the winning set of the initialized play in the game.

**Definition 4.2.4** (winning set)**.** *The winning set of player $i \in \{1, 2\}$ in a game $\mathcal{G}$, denoted by $\mathcal{W}_i$, is the set of nodes $v_0 \in V$ such that player $i$ has a winning strategy for any play $\pi$ initialized at $v_0$.*

For a game $\mathcal{G}$, our focus is to solve for the winning sets for the two players, with their corresponding winning strategies respectively. The winning strategy of each player can be solved in the process of solving the winning sets by the particular algorithm of the corresponding game. When the algorithm terminates, the winning sets of the two players partition the set of nodes of the arena $\mathcal{A}$, i.e., $V = \mathcal{W}_1 \oplus \mathcal{W}_2$. For this reason, the games that we consider are all determined games.

**Remark.** *The construction of the winning strategies, the proof of soundness and completeness of the winning sets, and the complexity analysis of the algorithm will be all included in the construction of the algorithm of the corresponding game. The fact that a game is determined follows directly from the proof of completeness and soundness of the winning sets. A short answer for the reason that the games we consider are all determined games would be the winning conditions of the two players in each game we consider are mutually exclusive, i.e., these games are 0-sum games with no ties.*

### Time Independent Property of Winning Play

The winning condition of a reachability game or a safety game is based on the occurrence set of a play, i.e., $\mathrm{Occ}(\pi)$, which is a finite condition.

The winning conditions of the other games are based on the infinity set of a play, i.e., $\mathrm{Inf}(\pi)$, which is an infinite condition.

For the simplicity of the discussion of the winning set, we will focus on the more complicated winning conditions based on $\mathrm{Inf}(\pi)$ for now, and leave the discussion for the simpler ones based on $\mathrm{Occ}(\pi)$ until we look into these two games.

We haven't discussed the local winning strategies are assigned on which nodes for each player, but let us assume the two players always follow their winning strategies if assigned for now.

Since the fundamental ingredient of the algorithm for each game originates from the linear BFS algorithm in a classic graph searching problem which is time independent and has local strategy, the algorithms for most of the games we consider are also time independent and have local strategy. For the ones that are not time independent such as generalized Büchi, the time dependent part only depends on a unit counter to determine which of the $k$ objectives is to satisfy, whereas the strategy remains local.

For $v_0 \in W_i$, there exists a winning strategy $\Pi_i$ for player $i$ such that any play $\pi$ induced by $\pi(0) = v_0$, $\Pi_i$, and $\Pi_{\bar{i}}$ is winning for player $i$, where $\Pi_{\bar{i}}$ is any strategy for player $\bar{i}$. We can see that the winning play $\pi$ of player $i$ is induced by the winning strategy of player $i$, $\Pi_i$. Since $\Pi_i$ is time independent, $\pi$ is also time independent. Also, since $\pi$ is an infinite play, truncating any finite part from the front of the play doesn't change the infinity set of the play. Therefore, the evaluation of the winning condition of the play doesn't change, and the winner of the play doesn't change. In other words, if we truncate a play $\pi$ initialized at $\pi(0) \in \mathcal{W}_i$ after any moment $j > 0$, then player $i$ remains to be the determined winner for the truncated play $\pi' = \pi(j+1)\pi(j+2)\cdots$. Therefore, the truncated play $\pi'$ can also be seen as a play initialized at $\pi(j+1) \in \mathcal{W}_i$. Since player $i$ is the determined winner of $\pi'$ for any $j > 0$, we have that any play $\pi$ initialized from $\mathcal{W}_i$ stays in $\mathcal{W}_i$. Formally, $(\pi(0) \in \mathcal{W}_i) \rightarrow (\mathrm{Occ}(\pi) \subseteq \mathcal{W}_i)$. It follows that the plays initialized from $\mathcal{W}_1$ and $\mathcal{W}_2$ are disjoint.

Now we can see that from the time independent property of the play, the definition of the winning set can be generalized from the initial node of a play to any node in a play.

**Definition 4.2.5** (time independent winning set)**.** *In a game $\mathcal{G}$ that is time independent, the winning set of player $i$, denoted by $\mathcal{W}_i$, is the set of nodes $v \in V$ such that player $i$ has a winning strategy for any play $\pi$ with $v$ occurring in $\pi$. Also, $\pi$ always stays in $\mathcal{W}_i$. Formally, $(v \in (\mathcal{W}_i \cap Occ(\pi))) \rightarrow (Occ(\pi) \subseteq \mathcal{W}_i)$.*

The time independent property allows us to look into a play $\pi$ from any moment $j \geq 0$ instead of always starting from moment 0. This allows us to look into the arena $\mathcal{A}$ locally when constructing the winning sets and the winning strategies of the game $\mathcal{G}$.

In a game $\mathcal{G} = (\mathcal{A}, \mathrm{Obj})$, since the set of nodes $V$ in the arena $\mathcal{A}$ can be partitioned by either the type of nodes $V_1$ and $V_2$, or the winning sets $\mathcal{W}_1$ and $\mathcal{W}_2$, $V$ can also be partitioned into four sets by combining these two partitions. Formally, with $V = V_1 \oplus V_2 = \mathcal{W}_1 \oplus \mathcal{W}_2$, we have $V = (V_1 \cap \mathcal{W}_1) \oplus (V_1 \cap \mathcal{W}_2) \oplus (V_2 \cap \mathcal{W}_1) \oplus (V_2 \cap \mathcal{W}_2)$. Now we can discuss where we can assign winning strategies for the two players.

Since player $i$ plays when $v \in V_i$, and player $i$ wins when $v \in \mathcal{W}_i$, the winning strategy $\Pi_i$ is assigned for $v \in (V_i \cap \mathcal{W}_i)$. Since $V_i \cap \mathcal{W}_{\bar{i}} \subseteq \mathcal{W}_{\bar{i}}$ is in the winning set of player $\bar{i}$, regardless of what strategy player $i$ use, player $\bar{i}$ will win the game. Therefore, no strategies are assigned in $V_i \cap \mathcal{W}_{\bar{i}}$. The strategy distribution on arena can be seen in table 4.1.

| $V$ | $V_1$ | $V_2$ |
|---|---|---|
| $\mathcal{W}_1$ | $V_1 \cap \mathcal{W}_1$<br>$\Pi_1$ | $V_2 \cap \mathcal{W}_1$<br>$\forall\ \Pi_2$ |

| | | |
|---|---|---|
| $\mathcal{W}_2$ | $V_1 \cap \mathcal{W}_2$<br>$\forall\ \Pi_1$ | $V_2 \cap \mathcal{W}_2$<br>$\Pi_2$ |

Table 4.1: Strategy Distribution on Arena

**Relation between the Winning Set and the Winning Strategy**

The winning set and the winning strategy of player $i$ always come in pair. The winning set is inductively guaranteed by the winning strategy. Conversely, the winning strategy of player $i$ is defined in nodes $v \in V_i \cap \mathcal{W}_i$. Therefore, they can only be considered jointly but not separately.

**Judgment of Winner**

After we have discussed time independent strategy, determined game and winning set, we can now look back and ask how to determine a winner of a game? Why can we predict the winner of a determined game before the game is actually played?

Consider a game $\mathcal{G}$ whose winning condition is defined based on the infinity set of the play $\text{Inf}(\pi)$. Since we can't practically keep track of an infinite play $\pi$ and we don't know how $\pi$ will be played in the future, the winner can neither be verified nor be falsified by analyzing the $\pi$ itself. In fact, the way we show a player $i$ is the winner of $\mathcal{G}$ is by induction of the winning strategy $\Pi_i$: if player $i$ declares to be the winner of a game initializing at $\pi(0) \in \mathcal{W}_i$, then we can inductively show that any play $\pi$ induced by $\pi(0) \in \mathcal{W}_i$, $\Pi_i$, and $\Pi_{\bar{i}}$ satisfies the winning condition of the play regardless of any $\Pi_{\bar{i}}$. Player $i$ is the determined winner as long as he always follows the winning strategy $\Pi_i$ to play the game. The winning strategy is solved and proved in the algorithm of the corresponding game, which is a backward process starting from the result.

Even for games whose winning condition is defined based on the occurrence set of the play $\text{Occ}(\pi)$, we still need the winning strategy to inductively verify or falsify the winner because such play can still be infinite. For example, in a reachability game, we need to inductively show whether the token will reach a target node in $T$; in a safety game, we need to inductively show whether the token will always stay in the safe nodes $S$.

### 4.2.6 Duality

Consider a game $\mathcal{G} = (\mathcal{A}, \text{Obj})$, we say it is a certain game for player $i$ when its winning condition $\varphi$ is positively defined for player $i$. Player $\bar{i}$ wins the game if player $i$ doesn't.

Therefore, the winning condition for player $\bar{i}$ is $\neg\varphi$. If we take the negation into $\varphi$, then $\neg\varphi$ is in the form of a positively defined winning condition of the dual game $\bar{\mathcal{G}}$ of this game $\mathcal{G}$. Therefore, a game $\mathcal{G}$ and its dual game $\bar{\mathcal{G}}$ is essentially the same game, just from the perspective of different players. If from the perspective of player $i$ its a game $\mathcal{G}$, then from the perspective of player $\bar{i}$ its a dual game $\bar{\mathcal{G}}$.

For example, the winning condition for a reachability game is $\mathrm{Occ}(p) \cap T \neq \emptyset$ where $T$ is the target set, and the winning condition for a safety game is $\mathrm{Occ}(\pi) \cap (V/S) = \emptyset$ where $S$ is the safe set. Let $\varphi = (\mathrm{Occ}(\pi) \cap T \neq \emptyset)$ be the winning condition for player 1, then the winning condition for player 2 would be $\neg\varphi = (\mathrm{Occ}(\pi) \cap T = \emptyset)$. We can see that in the same game, the winning condition for player 1 is a reachability condition, whereas the winning condition for player 2 is a safety condition if we consider the target set $T$ as the unsafe set $V/S$.

We can see that the duality of a game comes from the mutually exclusive property of the winning conditions of two players.

**Benefits of Duality**

The duality enables us to look into a game $\mathcal{G}$ from the sides of both players. This helps especially in the construction of the algorithm to solve for the winning sets and the winning strategies of the game. The algorithm for a game $\mathcal{G}$ and its dual game $\bar{\mathcal{G}}$ is essentially the same: the algorithm for $\mathcal{G}$ is for solving the winning set and the winning strategy of player 1 in $\mathcal{G}$, and the algorithm for $\bar{\mathcal{G}}$ can be used for solving the winning set and the winning strategy of player 2 in $\mathcal{G}$. If the winning set for one player $\mathcal{W}_i$ is determined in a game, then the winning set for the other player $\mathcal{W}_{\bar{i}}$ is also determined as a result of the partition relation of the winning sets $V = \mathcal{W}_1 \oplus \mathcal{W}_2$.

Therefore, in the algorithm for $\mathcal{G}$, we only need to show the winning set and the winning strategy for player 1. Leaving the winning strategy for player 2 in $\bar{\mathcal{G}}$ saves a lot of analysis. For example, the winning strategy for player 1 in generalized Büchi game and Street game requires a counter, dually the winning strategy for player 2 in generalized co-Büchi game and Rabin game also requires a counter. The winning strategies in all the other cases are local. This tells us that the winning strategies for the two players in a game $\mathcal{G}$ may be constructed differently.

The other benefit of the duality is that, sometimes a property of a game $\mathcal{G}$ is intuitive from the perspective of player 1 but not from the perspective of player 2, therefore it is hard to analyze the dual game $\bar{\mathcal{G}}$ from the perspective of player 1. However, if we analyze from the perspective of player 2 in the dual game $\bar{\mathcal{G}}$ which is equivalent as analyzing from the perspective of player 1 in game $\mathcal{G}$, then the result would be intuitive. The reason for this to hold is that the algorithms for $\mathcal{G}$ and $\bar{\mathcal{G}}$ operate almost in the same way. For

example, it is hard to do a relatively tight complexity analysis for generalized co-Büchi, but we know it must be the same as that of the dual game generalized Büchi because the algorithms for the dual games are essentially the same.

The slight differences of the algorithms for $\mathcal{G}$ and $\bar{\mathcal{G}}$ is that, the algorithm is particularly designed from the perspective of player 1 to achieve some optimization.

There is often a one layer difference in algorithms for $\mathcal{G}$ and $\bar{G}$ in each iteration if we don't tailor the algorithm for a specific game. The improvement can accumulate if we use the tailored algorithm as the iteration number goes up.

Another benefit about the algorithms for $\mathcal{G}$ and $\bar{\mathcal{G}}$ is that we can do some heuristic optimizations. We don't really know which of the two algorithms is slightly more efficient, but we can borrow the idea from the bi-direction BFS to implement the two algorithms together for better performance. This also induces a relatively tighter bound for the worst case time complexity.

Focusing on the behaviour of only one player also benefits if we don't care about the winning strategy of the other. For example, in control synthesis, we often use player 1 to depict the agent of the system and player 2 to depict the uncertainty of the environment. Therefore, we only care about the winning set and the winning strategy for player 1. If we stand on behalf of player 1 to look into a game $\mathcal{G}$, we only care about the winning set and the winning strategy of player 1, which can achieve some optimality by not considering the winning strategy for player2. We do not need to look into $\mathcal{W}_2$ to solve for the winning strategy in $V_2 \cap \mathcal{W}_2$, which allows us to work in a monotonically reducing sub-arena in the algorithm. Solving the winning strategy for player 2 in $\mathcal{G}$ is the same as solving the winning strategy for player 1 in $\bar{\mathcal{G}}$. Therefore, it is redundant and distractive if we present the algorithm for solving the winning strategy of player 2 in $\mathcal{G}$.

# Chapter 5

# Solution of Infinite Games

In this chapter, we will discuss the algorithms for solving the winning set and the winning strategy of player 1 in different games. One may refer to [15], [9], [10], [8], [28], and [16] for more related contents.

## 5.1 Introduction

Since a game $\mathcal{G}$ and its dual game $\bar{\mathcal{G}}$ are essentially the same game, we will present the dual games in pairs, i.e., reachability and safety, Büchi and co-Büchi, generalized Büchi and generalized co-Büchi, Rabin and Streett. We will extract the one pair Rabin and Streett game out before we introduce the more general $k$ pair Rabin and Streett game to decompose the complexity of analysis.

Since the role of player 2 in a game $\mathcal{G}$ is identical to the role of player 1 in the dual game $\bar{\mathcal{G}}$, we will present on behalf of player 1 in each game. This benefits us to only compute for the winning set and winning strategy of player 1 in the algorithm for $\mathcal{G}$, and leave the computation of the winning strategy of player 2 in $\mathcal{G}$ in the algorithm for $\bar{\mathcal{G}}$. Fixing our role to be player 1 also simplifies our notation since all the behavior is from the perspective of player 1. We will use $i$ for our current number of iteration, and $\bar{i}$ for the number of our fixed point iteration.

**Formulation and Solution of Infinite Games**

The formulation of an infinite game is defined based on an infinite play $\pi$, where the token moves along the out-going edges between the nodes. In the algorithm to solve for

the winning set and the winning strategy of player 1 in a game, we do the computation backwards along the in-going edges between the nodes.

When we talk about a node $v$, what we mean is that we assume the token is on $v$. For example, when we say a node $v' \in Pre(v)$ can move to $v$, what we mean is that we assume the token is on $v'$, and it can move through the edge $(v', v)$ to $v$.

**Sub-arena**

Since the arena $\mathcal{A}$ is monotonically decreasing as the algorithm implements, we will introduce the concept of sub-arena before we look into the actual problems. The sub-arena $\mathcal{A}' = (V', E')$ of the arena $\mathcal{A} = (V, E)$ is an arena induced by the set of nodes $V' \subseteq V$, where the edges in $E'$ are the edges in $E$ such that the two end nodes are in $V'$.

**Definition 5.1.1** (sub-arena)**.** *Given an arena $\mathcal{A} = (V, E)$, where $V = V_1 \oplus V_2$ is the set of nodes and $E \subseteq V \times V$ is the set of edges. The sub-arena $\mathcal{A}'$ induced by the set of nodes $V' \subseteq V$ is defined as $\mathcal{A}' = (V', E')$, where $V' \subseteq V$ and $E' = (V' \times V') \cap E$.*

**Remark.** *Note that a sub-arena may not satisfy the condition that all the nodes have an positive out-degree, but this can be resolved by doing a safety process which will be discussed in the safety game.*

Recall that the set of nodes $V$ in the arena $\mathcal{A} = (V, E)$ can also be partitioned by the winning sets of the two players, i.e., $V = \mathcal{W}_1 \oplus \mathcal{W}_2$. The two sub-arenas $\mathcal{A}_1$, $\mathcal{A}_2$ induced by $\mathcal{W}_1$ and $\mathcal{W}_2$ are disjoint. This is because the set of edges $E$ can be partitioned as $E = (\mathcal{W}_1 \times \mathcal{W}_1) \cap E \oplus (\mathcal{W}_1 \times \mathcal{W}_2) \cap E \oplus (\mathcal{W}_2 \times \mathcal{W}_1) \cap E \oplus (\mathcal{W}_2 \times \mathcal{W}_2) \cap E$. The two sub-arenas are defined as $\mathcal{A}_1 = (\mathcal{W}_1, (\mathcal{W}_1 \times \mathcal{W}_1) \cap E)$ and $\mathcal{A}_2 = (\mathcal{W}_2, (\mathcal{W}_2 \times \mathcal{W}_2) \cap E)$. Therefore, the set of edges that connect $\mathcal{A}_1$ and $\mathcal{A}_2$, i.e., $(\mathcal{W}_1 \times \mathcal{W}_2) \cap E$ and $(\mathcal{W}_2 \times \mathcal{W}_1) \cap E$ is never visited by the token. This shows that a play $\pi$ initializing in an sub-arena $\mathcal{A}_1$ or $\mathcal{A}_2$ will always stay in that sub-arena.

## 5.2 Reachability and Safety

In this section, we will introduce the reachability game and the safety game. The algorithms used in these two games, i.e., reach$(\cdot)$ and safe$(\cdot)$, will serve as atomic algorithms in the latter more complicated algorithms.

### 5.2.1 Reachability Game

In this subsection, we will introduce the reachability game that requires reaching the target set $T$ in at least 0 steps.

Reachability is the only finite game that we consider, where the play $p$ is finite.

**Winning Condition of the Game**

In a reachability game $\mathcal{G} = (\mathcal{A}, \text{Obj})$, the objective set $\text{Obj} = T \subseteq V$ is the set of target nodes. The winning condition for player 1 is to force the token to eventually reach a node in $T$ in at least 0 step. Formally, player 1 wins iff $\text{Occ}(p) \cap T \neq \emptyset$.

**Idea of the Algorithm**

The fundamental idea of the algorithm is the BFS.

Our goal is to force the token to eventually reach a node in $T$. Therefore, if the token is initially on a node $v$ in $T$, then this node naturally satisfies our winning condition, i.e., $T \subseteq \mathcal{W}_1$.

Now we look for the set of nodes that is guaranteed to reach a node in $T$, therefore we look for the predecessors of $T$, i.e., $Pre(T)$. The player 1 nodes in $Pre(T)$ are willing to reach $T$, whereas the player 2 nodes in $Pre(T)$ are reluctant to reach $T$. Therefore, a player 1 node in $Pre(T)$ is guaranteed to reach $T$ if there is an out-going edge to $T$, and a player 2 node in $Pre(T)$ is guaranteed to reach $T$ if all the out-going edges go to $T$. In other words, a player 2 node in $Pre(T)$ can only be forced to reach $T$.

Formally, the set of nodes in $Pre(T)$ that is guaranteed to reach $T$ is defined as

$pre(T) = \{v \in V_1 | Post(v) \cap T \neq \emptyset\} \oplus \{v \in V_2 | Post(v) \subseteq T\}$.

We consider the nodes in $T$ as on layer 0 of $\mathcal{W}_1$, and the nodes in $pre(T)/T$ as on layer 1 of $\mathcal{W}_1$. Each node in $\mathcal{W}_1$ can be assigned a layer number only once.

Next we consider the set of nodes that is guaranteed to reach a node on layer 1 of $\mathcal{W}_1$ that hasn't been assigned a layer number. This can be computed by applying $pre(\cdot)$ to the layer 1 nodes to find the layer 2 nodes of $\mathcal{W}_1$ that haven't been assigned a layer number.

We continue this process until no more layer can be assigned, and the maximum number of layer $\bar{i}$ is the number of our fixed point iteration. Since our arena is finite, this process must terminate. We can see that each layer of $\mathcal{W}_1$ partitions $\mathcal{W}_1$. Also, because of the unique layer number of each node in $\mathcal{W}_1$, each node and each edge is visited at most once in the algorithm, which gives a linear time complexity w.r.t. $|V|$ and $|E|$.

**Algorithm**

Algorithm 1 is the algorithm we use to solve for the reachability game. It is a variation of the classic BFS algorithm in a graph searching problem. The procedure to implement BFS is as follows:

- First we add the initial nodes into the queue as base.
- Each time we pick out the node at the head of the queue, and search for its adjacent nodes according to the enqueue rule.
- If an adjacent node hasn't been searched yet, then we add it to the tail of the queue.
- We continue this process until the queue is empty.

**Algorithm 1:** $\mathcal{A}'(\mathcal{W}_1) = \text{reach}_{\geq 0}(\mathcal{A}, T)$

**Input:**

1. $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$;
2. $T \subseteq V$;

**Output:**

1. $\mathcal{A}'(\mathcal{W}_1)$;

```
1  Initialize V as unvisited;
   /* Enqueue T as base.                                          */
2  for (queue = head = tail; all v ∈ T;) do
3      mark v as visited;
4      v.flag = 0;
5      *tail++ = v;
   /* BFS to do reach.                                            */
6  for (; head < tail; head++) do
7      for (v = *head; all v' ∈ Pre(v);) do
8          if v' ∈ V_1 then
9              if v' is not visited then
10                 mark v' as visited;
11                 v'.flag = v.flag + 1;
12                 v'.strategy = v;
13                 *tail++ = v';
14         else if v' ∈ V_2 then
15             v'.outdeg − −;
16             if v'.outdeg == 0 then
17                 mark v' as visited;
18                 v'.flag = v.flag + 1;
19                 *tail++ = v';
   /* W_1 is the entire queue.                                    */
20 W_2 = V/W_1;
```

**Result:**

1. Winning set $\mathcal{W}_1 \oplus \mathcal{W}_2 = V$;
2. Layer number of each $v \in \mathcal{W}_1$ is labeled by $v.flag \geq 0$;
3. Winning strategy of each node in $V_1 \cap \mathcal{W}_1$ is computed;

**Enqueue Rules of Different BFS**

The difference in different variations of Breadth First Search is the rule to enqueue an adjacent node. The enqueue rules in different variations of BFS are given by:

- Classic searching problem: $post(T) = \{v | Pre(v) \cap T \neq \emptyset\}$;
- Topological sort: $post(T) = \{v | Pre(v) \subseteq T\}$;
- Reachability for $\mathcal{A}(V_1, E)$: $pre(T) = \{v \in V_1 | Post(v) \cap T \neq \emptyset\}$;
- Reachability for $\mathcal{A}(V_2, E)$: $pre(T) = \{v \in V_2 | Post(v) \subseteq T\}$.

Here, in our reachability game, the enqueue rule is given by

$pre(T) = \{v \in V_1 | Post(v) \cap T \neq \emptyset\} \oplus \{v \in V_2 | Post(v) \subseteq T\}$.

**Proof of Soundness and Completeness**

**Soundness**

Conceptually, the process of solving $\mathcal{W}_1$ in a reachability game can be seen as a fixed point iteration induced by $R_0$ and $R_{i+1} = R_i \cup pre(R_i)$, where $R_0 = T$ and $pre(\cdot)$ is given above. The fixed point iteration terminates in the $\bar{i}^{th}$ iteration when $pre(R_{\bar{i}}) = \emptyset$, i.e., $R_{\bar{i}+1} = R_{\bar{i}} \cup pre(R_{\bar{i}})$. The number of fixed point iteration $\bar{i}$ is thus defined as the minimum number of $i$ such that $pre(R_i) = \emptyset$, or equivalently $R_{i+1} = R_i \cup pre(R_i)$. Since the arena is finite, the iteration must terminate in $\bar{i} \leq |V|$ iterations. We have $\mathcal{W}_1 = R_{\bar{i}}$ and $\mathcal{W}_2 = V/\mathcal{W}_1$. This gives the proof of soundness.

**Completeness**

From the enqueue rule $pre(\cdot)$, the reason why $v \in \mathcal{W}_2$ can't reach $T$ is straightforward:

- if $v \in V_1$, then $Post(v) \cap \mathcal{W}_1 = \emptyset$;
- if $v \in V_2$, then $Post(v) \cap \mathcal{W}_2 \neq \emptyset$.

Therefore, $v \in \mathcal{W}_2$ can never reach $\mathcal{W}_1$, thus never reach $T$. This gives the proof of completeness.

**Algorithmic Perspective of BFS**

In practice, we partition $\mathcal{W}_1$ into $\bar{i}+1$ layers, where $layer_0 = R_0$, and $layer_i = R_i/R_{i-1}$ for $1 \leq i \leq \bar{i}$. Instead of computing $pre(R_i)$ to get $R_{i+1} = R_i \cup pre(R_i)$ in the $i^{th}$ iteration, we are actually computing $pre(layer_i)$ to get $layer_{i+1} = pre(layer_i)/R_i$, where $R_i = layer_0 \oplus layer_1 \oplus \cdots \oplus layer_i$. Since $\mathcal{W}_1 = layer_0 \oplus layer_1 \oplus \cdots \oplus layer_{\bar{i}}$, each node

and each edge in the arena is visited at most once. This shows that the time complexity for solving the reachability game is linear w.r.t. $|V|$ and $|E|$. The relation between $R_i$ and $layer_i$ is given below.

$$\begin{aligned} R_0 &= T & layer_0 &= R_0 \\ R_1 &= R_0 \cup pre(R_0) & layer_1 &= R_1 / R_0 \\ R_2 &= R_1 \cup pre(R_1) & layer_2 &= R_2 / R_1 \\ &\vdots & &\vdots \\ R_{\bar{i}} &= R_{\bar{i}-1} \cup pre(R_{\bar{i}-1}) & layer_{\bar{i}} &= R_{\bar{i}} / R_{\bar{i}-1} \end{aligned}$$

**Strategy**

For $v \in \mathcal{W}_1$, $v.flag$ marks the layer number of $v$. Suppose a node $v \in \mathcal{W}_1$ is on layer $i$. If $v \in V_1 \cap \mathcal{W}_1$, then $v$ can reach $T$ in at least $i$ steps, and $i$ is one more than the layer number of $v.strategy$; if $v \in V_2 \cap \mathcal{W}_1$, then $v$ can reach $T$ in at most $i$ steps, and $i$ is at least one more than the layer number of any successors of $v$. Also, the nodes in $T$ is on layer 0. Therefore, as long as player 1 follows the given strategy, the layer number of all the plays $p$ initialized on a node $v_0 \in \mathcal{W}_1$ is monotonically decreasing to 0, thus eventually reaching a node in $T$. This shows that such strategy is indeed a winning strategy for player 1. The local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.

**Complexity Analysis**

In a reachability game $\mathcal{G} = (\mathcal{A}, \text{Obj})$, $\mathcal{A} = (V, E)$ is the arena and $\text{Obj} = T \subseteq V$ is the objective set. $|V| = n$ is the number of nodes and $|E| = m$ is the number of edges.

**Proposition 5.2.1.** *A reachability game can be solved in $\mathcal{O}(m)$ space, $\mathcal{O}(m)$ time, and the local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Space Complexity:** $\mathcal{O}(n + m)$

Since the size of the arena is depicted by $n$ nodes and $m$ edges, the space complexity of the algorithm is $\mathcal{O}(n + m)$.

**Time Complexity:** $\mathcal{O}(n + m)$

Guaranteed by each node being enqueued at most once, each node and each edge is thus visited at most once, and the time complexity of the algorithm is thus linear w.r.t. $n$ and $m$, i.e., $\mathcal{O}(n + m)$.

### 5.2.2 Variation of the Reachability Game

In this subsection, we will introduce a variation of the reachability game that requires reaching the target set $T$ in at least one step. This is one of the atomic algorithms that we will use in the latter more complicated polynomial algorithms. Since this reachability game is just a variation of the previous one, we will only present the differences for simplicity.

In the previous reachability game, we apply $\text{reach}_{\geq 0}(\mathcal{A}, T)$ to find the set of nodes that can reach $T$ in at least 0 steps. We consider nodes in $T$ as on layer 0, and is naturally in the reachable set of $T$.

Here, we will introduce a variation of the reachability game which is used to solve for the set of nodes that can reach $T$ in at least 1 step, which has a higher restriction that a node $v$ in $T$ is in $\text{reach}_{\geq 1}(\mathcal{A}, T)$ iff $v$ can reach $T$ in at least one step. By iteratively applying this algorithm, we can find the set of nodes that can reach $T$ two times, three times, up to infinitely many often times in at least one step, which satisfies a Büchi objective of infinitely many often times reaching $T$.

When we start the BFS for $\mathcal{A}'(\mathcal{W}_1) = \text{reach}_{\geq 1}(\mathcal{A}, T)$, the layer number of the nodes in $T$ is 0, which doesn't mean $T \subseteq \mathcal{W}_1$ yet. A node $v$ in $T$ is in $\mathcal{W}_1$ iff its layer number turns positive, which means $v$ can reach $T$ in at least 1 step. Therefore, the layer number of the nodes in $\mathcal{W}_1$ starts from 1 instead of 0.

**Winning Condition of the Game**

In this variation of the reachability game $\mathcal{G} = (\mathcal{A}, \text{Obj})$, the objective set $\text{Obj} = T \subseteq V$ is the set of target nodes. The winning condition for player 1 is to force the token to eventually reach a node in $T$ in at least one step.

Formally, player 1 wins iff $\text{Occ}(p(i)) \cap T \neq \emptyset, i \geq 1$.

**Idea of the Algorithm**

This algorithm is a variation of the previous one, thus the fundamental idea of the algorithm is still the BFS.

In this reacability game, the layer number of $\mathcal{W}_1$ is from 1 to $\bar{i}$ instead of from 0 to $\bar{i}$ in the previous one. The set of nodes in $T$ is initially on layer 0, but it can be assigned a positive layer number once if it can reach $T$ in at least one step. All the other nodes in $V/T$ can only be assigned a layer number once. Everything else is identical to the previous reachability game.

**Algorithm**

Algorithm 2 is the algorithm we use to solve for the variation of the reachability game.

**Algorithm 2:** $\mathcal{A}'(\mathcal{W}_1) = \text{reach}_{\geq 1}(\mathcal{A}, T)$

**Input:**

1. $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$;
2. $T \subseteq V$;

**Output:**

1. $\mathcal{A}'(\mathcal{W}_1)$;

```
1  Initialize V as unvisited;
   /* Enqueue T as base.                                          */
2  for (queue = head = tail; all v ∈ T;) do
       /* Do not mark v as visited.                               */
3      v.flag = 0;
4      *tail++ = v;
   /* BFS to do reach.                                            */
5  for (i = 1, marker=tail, W1 = ∅; head < tail; i++, head++) do
6      if head == maker then marker = tail, i++;
7      for (v = *head; all v' ∈ Pre(v);) do
8          if v' ∈ V1 then
9              if v' is not visited then
10                 W1+ = v';
11                 mark v' as visited;
12                 v'.flag = i;
13                 v'.strategy = v;
14                 if v' ∉ T then *tail++ = v';
15         else if v' ∈ V2 then
16             v'.outdeg − −;
17             if v'.outdeg == 0 then
18                 W1+ = v';
19                 mark v' as visited;
20                 v'.flag = i;
21                 if v' ∉ T then *tail++ = v';
22 W2 = V/W1;
```

**Result:**

1. Winning set $\mathcal{W}_1 \oplus \mathcal{W}_2 = V$;
2. Layer number of each $v \in \mathcal{W}_1$ is labeled by $v.flag \geq 1$;
3. Winning strategy of each node in $V_1 \cap \mathcal{W}_1$ is computed;

**Fixed Point Iteration**

Conceptually, the process of solving $\mathcal{W}_1$ in this variation of the reachability game can be seen as a fixed point iteration induced by $R_1$ and $R_{i+1} = R_i \cup pre(R_i)$, where $R_1 = pre(T)$ and $pre(\cdot)$ is given by $pre(T) = \{v \in V_1 | Post(v) \cap T \neq \emptyset\} \oplus \{v \in V_2 | Post(v) \subseteq T\}$ as in the previous one. The fixed point iteration terminates in the $\bar{i}^{th}$ iteration when $pre(R_{\bar{i}}) = \emptyset$, i.e., $R_{\bar{i}+1} = R_{\bar{i}} \cup pre(R_{\bar{i}})$. The number of fixed point iteration $\bar{i}$ is thus defined as the minimum number of $i$ such that $pre(R_i) = \emptyset$, or equivalently $R_{i+1} = R_i \cup pre(R_i)$. Since the arena is finite, the iteration must terminate in $\bar{i} \leq |V|$ iterations. We have $\mathcal{W}_1 = R_{\bar{i}}$ and $\mathcal{W}_2 = V/\mathcal{W}_1$.

**Algorithmic Perspective of BFS**

In practice, we partition $\mathcal{W}_1$ into $\bar{i}$ layers, where $layer_1 = R_1$, $layer_i = R_i/R_{i-1}$ for $2 \leq i \leq \bar{i}$. $layer_0 = T/R_{\bar{i}}$ is not in $\mathcal{W}_1$. Instead of computing $pre(R_i)$ to get $R_{i+1} = R_i \cup pre(R_i)$ in the $i^{th}$ iteration, we are actually computing $pre(layer_i)$ to get $layer_{i+1} = pre(layer_i)/R_i$, where $R_i = layer_1 \oplus layer_2 \oplus \cdots \oplus layer_i$.

Since $\mathcal{W}_1 = layer_1 \oplus layer_2 \oplus \cdots \oplus layer_{\bar{i}}$, each node and each edge in the arena is visited at most once. This shows that the time complexity for solving this variation of the reachability game is linear w.r.t. $|V|$ and $|E|$. The relation between $R_i$ and $layer_i$ is given below.

$$
\begin{array}{ll}
R_1 = pre(T) & layer_1 = R_1 \\
R_2 = R_1 \cup pre(R_1) & layer_2 = R_2/R_1 \\
R_3 = R_2 \cup pre(R_2) & layer_3 = R_3/R_2 \\
\vdots & \vdots \\
R_{\bar{i}} = R_{\bar{i}-1} \cup pre(R_{\bar{i}-1}) & layer_{\bar{i}} = R_{\bar{i}}/R_{\bar{i}-1} \\
 & \hline
 & layer_0 = T/R_{\bar{i}}
\end{array}
$$

In the implementation, it can be even more concise as shown in algorithm 2. The trick is that, in the conventional BFS, we mark a node $v$ as visited when we enqueue $v$, but here we **do not** mark $v$ as visited when we enqueue $v$. Also, we enqueue a node only if it hasn't been enqueued yet to guarantee that we enqueue each node only once in the BFS. The $marker$ is used to separate the different layers, which can be removed together with $v'.flag$ and $i$ if we don't care about the layer numbers. We can even remove the $v'.strategy$ if we only want the winning set but not the winning strategy, or in the intermediate process of a polynomial algorithm such as Büchi for the reason that only the strategy assigned in the last iteration is real, whereas all the strategies assigned in the previous ones are nominal.

**Strategy**

The reasoning to show the given strategy is a winning strategy for player 1 is similar to that in the previous game. As long as player 1 follows the given strategy, the layer number of all the plays $p$ initialized on $v_0 \in \mathcal{W}_1$ is monotonically decreasing to 1, and the next node $v \in T$ can have any layer number from 0 to $\bar{i}$. The local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.

**Complexity Analysis**

Since this algorithm is also a variation of the BFS, the space complexity and the time complexity are both linear w.r.t. the number of nodes $|V| = n$ and the number of edges $|E| = m$ in the arena $\mathcal{A} = (V, E)$, i.e., $\mathcal{O}(n + m)$.

**Discussion. *Relation Between the Two Reachability Games***

*Let $\mathcal{A}'(R_{\geq 0}) = reach_{\geq 0}(\mathcal{A}, T)$ and $\mathcal{A}'(R_{\geq 1}) = reach_{\geq 1}(\mathcal{A}, T)$. Then the set of nodes in $T$ that can't reach $T$ again, denoted by $T'$, is given by $T' = R_{\geq 0}/R_{\geq 1}$ or $T' = T/R_{\geq 1}$. In other words, $R_{\geq 0} = T' \oplus R_{\geq 1}$.*

*We will give more discussions when we talk about the Büchi game.*

### 5.2.3 Safety Game

In this subsection, we will introduce the safety game that requires always staying in the safe set $S$. We will present the safety game in a dual way of the reachability game.

Safety is the first infinite game that we consider, where the play $\pi$ is infinite.

**Winning Condition of the Game**

In a safety game $\mathcal{G} = (\mathcal{A}, \text{Obj})$, the objective set $\text{Obj} = S \subseteq V$ is the set of safe nodes and correspondingly, $V/S$ is the set of unsafe nodes. The winning condition for player 1 is to force the token to always stay in $S$, or equivalently, never reach $V/S$. Formally, player 1 wins iff $\text{Occ}(\pi) \subseteq S$, or equivalently, $\text{Occ}(\pi) \cap (V/S) = \emptyset$.

**Idea of the Algorithm**

The idea of the safety algorithm is to delete the unsafe nodes in $S$ that can reach $V/S$, and the remaining nodes in $S$ are the safe ones that can always stay in $S$. The fundamental idea of the algorithm is still the BFS, thus it enjoys all the nice properties of BFS.

The essence of a safety game is to solve a reachability game starting from the unsafe nodes $V/S$ to delete the unsafe nodes that can escape from the safe nodes $S$. Therefore,

when we say "doing a safety", what we mean is to do a reachability to delete the unsafe nodes. The player 2 nodes in $Pre(V/S)$ are willing to reach $V/S$, whereas the player 1 nodes in $Pre(V/S)$ are reluctant to reach $V/S$. Therefore, a player 2 node in $Pre(V/S)$ is guaranteed to reach $V/S$ if there is an out-going edge to $V/S$, and a player 1 node in $Pre(V/S)$ is guaranteed to reach $Pre(V/S)$ when all the out-going edges go to $V/S$. In other words, a player 1 node in $Pre(V/S)$ can only be forced to reach $V/S$.

**Algorithm**

Algorithm 3 is the algorithm we use to solve for the safety game.

**Algorithm 3:** $\mathcal{A}'(\mathcal{W}_1) = \text{safe}(\mathcal{A}, S)$

**Input:**
1. $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$;
2. $S \subseteq V$;

**Output:**
1. $\mathcal{A}'(\mathcal{W}_1)$;

```
1  Initialize V as safe;
   /* Enqueue V/S as base.                                          */
2  for (queue = head = tail; all v ∈ V/S;) do
3      mark v as unsafe;
4      *tail++ = v;
   /* BFS to delete unsafe nodes.                                   */
5  for (; head < tail; head++) do
6      for (v = *head; all v' ∈ Pre(v);) do
7          if v' ∈ V_2 then
8              if v' is safe then
9                  mark v' as unsafe;
10                 *tail++ = v';
11         else if v' ∈ V_1 then
12             v'.outdeg--;
13             if v'.outdeg == 0 then
14                 mark v' as unsafe;
15                 *tail++ = v';
   /* W_2 is the entire queue.                                      */
16 W_1 = V/W_2;
```

**Result:**
1. Winning set $\mathcal{W}_1 \oplus \mathcal{W}_2 = V$;
2. Winning strategy of each node $v \in V_1 \cap \mathcal{W}_1$ is computed by $v.strategy = \{v' | \forall v' \in Post(v)\ s.t.\ v' \in \mathcal{W}_1\}$;

### Enqueue Rules of Different Safety Game

We use the idea of BFS to solve for the different safety games, while the differences lies in the different enqueue rules:

- Safety for $\mathcal{A}(V_1, E)$: $pre(V/S) = \{v \in V_1 | Post(v) \subseteq (V/S)\}$;
- Safety for $\mathcal{A}(V_2, E)$: $pre(V/S) = \{v \in V_2 | Post(v) \cap (V/S) \neq \emptyset\}$.

Here in our safety game, the enqueue rule is given by

$pre(V/S) = \{v \in V_2 | Post(v) \cap (V/S) \neq \emptyset\} \oplus \{v \in V_1 | Post(v) \subseteq (V/S)\}$, which is the combination of the enqueue rules for the safety games of $\mathcal{A}(V_1, E)$ and $\mathcal{A}(V_2, E)$. The ideas of the safety games of $\mathcal{A}(V_1, E)$ and $\mathcal{A}(V_2, E)$ originate from the topological sort and the classic graph searching problem respectively.

**Proof of Soundness and Completeness**

**Completeness**

$\mathcal{W}_2$ is computed as the set of nodes that are guaranteed to reach $V/S$ regardless of how player 1 plays. Therefore, $v \in \mathcal{W}_2$ will eventually reach $V/S$, thus violating the winning condition of always staying in $S$. This gives the proof of completeness.

**Soundness**

$\mathcal{W}_1$ is the set of safe nodes after deleting the unsafe nodes, i.e., $\mathcal{W}_1 = V/\mathcal{W}_2$.

From the enqueue rule $pre(\cdot)$ above, the reason why $v \in \mathcal{W}_1$ can always stay in $S$ is straightforward:

- if $v \in V_2$, then $Post(v) \cap \mathcal{W}_2 = \emptyset$;
- if $v \in V_1$, then $Post(v) \cap \mathcal{W}_1 \neq \emptyset$.

Therefore, $v \in \mathcal{W}_1$ can always stay in $\mathcal{W}_1 \subseteq S$. This gives the proof of soundness.

**Strategy**

From the proof of soundness, we can see that $v \in V_2 \cap \mathcal{W}_1$ can not move to $\mathcal{W}_2$ in the next step since $Post(v) \cap \mathcal{W}_2 = \emptyset$. In other words, $Post(v) \subseteq \mathcal{W}_1 \subseteq S$.

Also, $v \in V_1 \cap \mathcal{W}_1$ can move to $\mathcal{W}_1$ in the next step since $Post(v) \cap \mathcal{W}_1 \neq \emptyset$. It follows that the strategy for $v \in V_1 \cap \mathcal{W}_1$ can be any successor of $v$ in $\mathcal{W}_1$.

Formally, for $v \in V_1 \cap \mathcal{W}_1$, $v.strategy = \{v' | \forall v' \in Post(v) \ s.t. \ v' \in \mathcal{W}_1\}$.

With this strategy, $Post(\mathcal{W}_1) \subseteq \mathcal{W}_1 \subseteq S$, thus inductively any play $\pi$ initialized in $\mathcal{W}_1$ is trapped in $\mathcal{W}_1 \subseteq S$. Therefore, the token always stays in $\mathcal{W}_1 \subseteq S$ and such strategy is indeed a winning strategy for player 1. The local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.

**Complexity Analysis**

Since this algorithm is also a variation of the BFS, the space complexity and the time complexity is both linear w.r.t. the number of nodes $|V| = n$ and the number of edges $|E| = m$ in the arena $\mathcal{A} = (V, E)$, i.e., $\mathcal{O}(n + m)$.

**Proposition 5.2.2.** *A safety game can be solved in $\mathcal{O}(m)$ space, $\mathcal{O}(m)$ time, and the local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Discussion.** ***More about Safety Game***

*Since we don't care that much about nodes in $\mathcal{W}_2$ can reach $V/S$ in how many steps, we don't label the layers in the safety algorithm. This can be done in a similar process as in a reachability game if we want to.*

*An optimization of the safety game is that, we can enqueue $pre(V/S)$ instead of $V/S$ as base nodes, which allows us to restrict to the sub-arena $\mathcal{A}'(S)$. This small change will safe us one step of enqueue calculation in a co-Büchi game, which can accumulate in a polynomial algorithm.*

*$reach_{\geq 0}(\cdot, \cdot)$, $reach_{\geq 1}(\cdot, \cdot)$, together with $safe(\cdot, \cdot)$ are all variations of BFS. These linear algorithms are the atomic algorithms that we use to construct the latter more complicated polynomial algorithms.*

## 5.3 Büchi and co-Büchi

In this section, we will introduce the Büchi game and the co-Büchi game. The algorithms used in these two games are constructed by the atomic algorithms introduced in the previous section.

### 5.3.1 Büchi Game

In this subsection, we will introduce the Büchi game that requires reaching the target set $T$ infinitely many often.

**Winning Condition of the Game**

In a Büchi game $B\mathcal{G} = (\mathcal{A}, \text{Obj})$, the objective set $\text{Obj} = T \subseteq V$ is the set of target nodes. The winning condition for player 1 is to force the token to reach $T$ infinitely many often. Formally, player 1 wins iff $\text{Inf}(\pi) \cap T \neq \emptyset$.

### Idea of the Algorithm

The basic idea of the algorithm is to iteratively apply $\text{reach}_{\geq 1}(\cdot, \cdot)$ until fixed point iteration.

We find the set of nodes that can reach $T$ in at least 1 step once after applying $\text{reach}_{\geq 1}(\mathcal{A}'(V), T)$ once. We find the set of nodes that can reach $T$ in at least 1 step twice after iteratively applying $\text{reach}_{\geq 1}(\mathcal{A}'(V), T)$ twice. After applying $\text{reach}_{\geq 1}(\cdot, \cdot)$ each time, the nodes in $T$ that can't reach $T$ again are deleted, thus the size of $T$ reduces by at least 1 each time. The set of nodes after iterating $\text{reach}_{\geq 1}(\cdot, \cdot)$ $i$ times can reach $T$ in at least 1 step for at least $i$ times. We continue this process until the iteration terminates in the $\bar{i}^{th}$ iteration when the size of $T$ doesn't reduce any more, thus the size of the reachable set of $T$ doesn't reduce, even if we iterate for more times. Now this reachable set of $T$ after $\bar{i}$ iterations forms a closed loop that guarantees any node in this set can reach a node in $T$, thus inductively satisfies the winning condition of infinitely many often reach $T$.

### Algorithm

Algorithm 4 is a naive algorithm we use to solve for the Büchi game. It's a fixed point iteration of $\text{reach}_{\geq 1}(\cdot, \cdot)$.

**Algorithm 4:** $\mathcal{A}'(\mathcal{W}_1) = \text{Büchi}(\mathcal{A}, T)$ Fixed Point Iteration of Reach

**Input:**
1. $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$;
2. $T \subseteq V$;

**Output:**
1. $\mathcal{A}'(\mathcal{W}_1)$;

```
for (V.iterate = 0, V' = V, T' = T, i = 1;; i++) do
    A'(R_1) = reach_{>=1}(A'(V'), T');
    R_1.iterate = i;
    if V' == R_1 then break;
    V' = R_1;
    T' = T' ∩ V';
W_1 = V';
W_2 = V/W_1;
```

**Result:**
1. Winning set $\mathcal{W}_1 \oplus \mathcal{W}_2 = V$;
2. Iterate number of each $v \in V$ is labeled by $v.iterate \geq 0$;
3. Winning strategy of each node in $V_1 \cap \mathcal{W}_1$ is computed by $\text{reach}_{\geq 1}(\cdot, \cdot)$;

### Proof of Soundness and Completeness

**Soundness**

Conceptually, the process of solving $\mathcal{W}_1$ in a Büchi game can be seen as a fixed point iteration induced by $\mathcal{A}'(R_{i+1}) = \text{reach}_{\geq 1}(\mathcal{A}'(R_i), T_i)$, where $R_1 = V$, $T_1 = T$, and $T_{i+1} = R_{i+1} \cap T_i$. The fixed point iteration terminates in the $\bar{i}^{th}$ iteration when $T_{\bar{i}+1} = T_{\bar{i}}$, i.e., $R_{\bar{i}+1} = R_{\bar{i}}$. The number of fixed point iteration $\bar{i}$ is thus defined as the minimum number of $i$ such that $T_{i+1} = T_i$, or equivalently $R_{i+1} = R_i$. The nodes in $R_i$ can reach $T$ in at least 1 step for at least $i$ times. Since the fixed point iteration is achieved after the $\bar{i}^{th}$ iteration, a closed loop is formed in $R_{\bar{i}}$. The nodes in $R_{\bar{i}}$ can thus reach $T$ infinitely many often inductively. Since the size of $T$ reduces by $|R_i/R_{i+1}| \geq 1$ in the $i^{th}$ iteration, the iteration must terminate in $\bar{i} \leq |T|$ iterations. We have $\mathcal{W}_1 = R_{\bar{i}}$ and $\mathcal{W}_2 = V/\mathcal{W}_1$. This gives the proof of soundness.

**Completeness**

In algorithm 4, each node $v \in V$ is labeled with a iterate number indicating the number of times $v$ participated in the iterations.

We have $\mathcal{W}_1 = \{v | v.iterate = \bar{i}\}$, and $\mathcal{W}_2 = \{v | 0 \leq v.iterate < \bar{i}\}$.

Nodes in $\mathcal{W}_1$ has $v.iterate = \bar{i}$, indicating $v$ has participated in all of the $\bar{i}^{th}$ iterations for $1 \leq i \leq \bar{i}$.

Nodes in $\mathcal{W}_2$ has $v.iterate = i < \bar{i}$, indicating $v$ has only participated in the first $i^{th}$ iterations from 1 to $i$. Such nodes can reach $T$ in at least 1 step for $i$ times, but not the $(i+1)^{th}$ time. Therefore, $v \in \mathcal{W}_2$ can at most reach $T$ in at least 1 step for $i < \bar{i}$ times which is finite, thus can't reach $T$ infinitely many often times. This gives the proof of completeness.

**Alternative Algorithm**

Algorithm 5 is an improved algorithm based on algorithm 4 that we use to solve for the Büchi game. It's a fixed point iteration of $\text{reach}_{\geq 1}(\cdot, \cdot)$ and $\text{safe}(\cdot, \cdot)$. $\text{reach}_{\geq 0}(\cdot, \cdot)$ is for the use of presenting the sub-arena in the $i^{th}$ iteration only.

Algorithm 5 does an extra safety process that deletes the nodes that can't reach $T$ for the next time after doing each reach iteration. After the $i^{th}$ iteration of $\text{reach}_{\geq 1}(\cdot, \cdot)$, we find the set of nodes that can reach $T$ in at least 1 step for $i$ times. Before we iterate $\text{reach}_{\geq 1}(\cdot, \cdot)$ for an $(i+1)^{th}$ time to find the set of nodes that can reach $T$ for an $(i+1)^{th}$ time, we apply the safety algorithm $\text{safe}(\cdot, \cdot)$ to delete the nodes that can't reach $T$ for an $(i+1)^{th}$ time since these nodes can reach $\mathcal{W}_2$ and then get trapped in $\mathcal{W}_2$. We do an extra safety after each reach iteration in a hope that the nodes in $T$ can be deleted in the safety

process in order to reduce the number of iterations of reachability.

**Algorithm 5:** $\mathcal{A}'(\mathcal{W}_1) = \text{Büchi}(\mathcal{A}, T)$ Fixed Point Iteration of Reach and Safe

**Input:**
1. $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$;
2. $T \subseteq V$;

**Output:**
1. $\mathcal{A}'(\mathcal{W}_1)$;

```
for (V.iterate = 0, V' = V, T' = T, i = 1; ; i++) do
    A'(R_0) = reach_{≥0}(A'(V'), T');
    A'(R_1) = reach_{≥1}(A'(V'), T');
    V' = R_0;
    if R_0 == R_1 then R_1.iterate = i; break;
    A'(V') = safe(A'(R_0), R_1);
    T' = T' ∩ V';
    if T' == ∅ then V' = ∅; break;
    V'.iterate = i;
W_1 = V';
W_2 = V/W_1;
```

**Result:**
1. Winning set $\mathcal{W}_1 \oplus \mathcal{W}_2 = V$;
2. Iterate number of each $v \in V$ is labeled by $v.iterate \geq 0$;
3. Winning strategy of each node in $V_1 \cap \mathcal{W}_1$ is computed by $\text{reach}_{\geq 1}(\cdot, \cdot)$;

### Proof of Soundness and Completeness

#### Soundness

Conceptually, the process of solving $\mathcal{W}_1$ in a Büchi game using the improved algorithm can be seen as an alternating fixed point iteration induced by $\mathcal{A}'(R_{i+1}) = \text{reach}_{\geq 1}(\mathcal{A}'(R_i), T_i)$ and $\mathcal{A}'(R_{i+1}) = \text{safe}(\mathcal{A}'(R_i), R_{i+1})$, where $R_0 = V$, $T_0 = T$, and $T_{i+1} = T_i \cap R_{i+1}$. The fixed point iteration terminates in the $\bar{i}^{th}$ iteration when $T_{\bar{i}+1} = T_{\bar{i}}$, i.e., $R_{\bar{i}+1} = R_{\bar{i}}$. The number of fixed point iteration $\bar{i}$ is thus defined as the minimum number of $i$ such that $T_{i+1} = T_i$, or equivalently $R_{i+1} = R_i$. The nodes in $R_i$ can reach $T$ in at least 1 step for at least $i$ times. Since the fixed point iteration is achieved after the $\bar{i}^{th}$ iteration, a closed loop is formed in $R_{\bar{i}}$. The nodes in $R_{\bar{i}}$ can thus reach $T$ infinitely many often inductively. Since the size of $T$ reduces by $|R_i/R_{i+1}| \geq 1$ in the $i^{th}$ iteration, the iteration must terminate in $\bar{i} \leq |T|$ iterations. We have $\mathcal{W}_1 = R_{\bar{i}}$ and $\mathcal{W}_2 = V/\mathcal{W}_1$. This gives the proof of soundness.

#### Completeness

In algorithm 5, each node $v \in V$ is labeled with a iterate number indicating the number of times $v$ participated in the iterations.

We have $\mathcal{W}_1 = \{v | v.iterate = \bar{i}\}$, and $\mathcal{W}_2 = \{v | v.iterate < \bar{i}\}$.

Nodes in $\mathcal{W}_1$ has $v.iterate = \bar{i}$, indicating $v$ has participated in all of the $\bar{i}^{th}$ iterations for $1 \leq i \leq \bar{i}$.

Nodes in $\mathcal{W}_2$ has $v.iterate = i < \bar{i}$, indicating $v$ has only participated in the first $i^{th}$ iterations from 1 to $i$. Such nodes can reach $T$ in at least 1 step for $i$ times, or reach $\mathcal{W}_2$ after the $(i+1)^{th}$ time. Therefore, $v \in \mathcal{W}_2$ can at most reach $T$ in at least 1 step for $i < \bar{i}$ times which is finite, thus can't reach $T$ infinitely many often times. This gives the proof of completeness.

**Strategy**

The strategies in the two algorithms of the Büchi game are the strategies computed in the $\text{reach}_{\geq 1}(\cdot, \cdot)$ process. The strategies of the two algorithms should be the same since the winning sub-arena $\mathcal{A}'(\mathcal{W}_1)$ is the same. If we apply $\text{reach}_{\geq 1}(\cdot, \cdot)$ again in $\mathcal{A}'(\mathcal{W}_1)$, the strategy for $v \in V_1 \cap \mathcal{W}_1$ should be the same.

Since the reachable set of $T$ is reducing after each iteration, the strategy of the remaining $V_1$ nodes may be updating each time, thus only the strategy in the last iteration $\bar{i}$ is the real strategy. All the strategies computed in the previous iterations are nominal. Therefore, we can choose to compute the strategy in each iteration, or after the fixed point iteration is achieved depending on the implementation.

For $v \in V_1 \cap \mathcal{W}_1$, from the analysis of the strategy in a reachability game, we know that such strategy can guarantee $v$ to reach a node $v' \in T \cap \mathcal{W}_1$ in at least one step, and $v'$ can reach a node $v'' \in T \cap \mathcal{W}_1$ in at least one step again, thus inductively $v$ can reach $T$ infinitely many often times. This shows that such strategy is indeed a winning strategy for player 1. The local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.

**Complexity Analysis**

In a Büchi game $B\mathcal{G} = (\mathcal{A}, \text{Obj})$, $\mathcal{A} = (V, E)$ is the arena and $\text{Obj} = T \subseteq V$ is the objective set. $|V| = n$ is the number of nodes and $|E| = m$ is the number of edges.

**Proposition 5.3.1.** *A Büchi game can be solved in $\mathcal{O}(m)$ space, $\mathcal{O}(nm)$ time, and the local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Space Complexity:** $\mathcal{O}(n+m)$

Since the size of the arena is depicted by $n$ nodes and $m$ edges, the space complexity of the algorithm is $\mathcal{O}(n+m)$.

**Time Complexity:** $\mathcal{O}(nm)$

In the first naive Büchi algorithm 4, each $\text{reach}_{\geq 1}(\cdot,\cdot)$ costs $\mathcal{O}(m)$. $|T|$ reduces by at least one after each iteration, thus there can be at most $\bar{i} \leq |T| \leq n$ iterations. Therefore, the time complexity of the algorithm is polynomial w.r.t. $n$ and $m$, i.e., $\mathcal{O}(|T| \cdot m) = \mathcal{O}(nm)$.

In the second improved Büchi algorithm 5, the analysis is the same as in algorithm 4, with an additional cost for the safety process. The total cost of $\text{safe}(\cdot,\cdot)$ is linear, i.e., $\mathcal{O}\left(\sum_{i=1}^{\bar{i}} \text{safe}(\cdot,\cdot)\right) = \mathcal{O}(m)$. Therefore, with an additional cost of at most linear $\mathcal{O}(m)$, we are likely to reduce the number of iterations if any $v \in T$ is deleted in the safety process, which may result in a smaller multiple of $\mathcal{O}(m)$. This is usually the case in practice. The time complexity of the algorithm is thus polynomial w.r.t. $n$ and $m$, i.e, $\mathcal{O}(|T| \cdot m) = \mathcal{O}(nm)$.

**More Detailed Analysis**

Now we have that the time complexity of the Büchi algorithm is $\mathcal{O}(|T| \cdot m) = \mathcal{O}(nm)$, where the number of iterations is upper bounded by $|T|$. Our goal is to see whether we can reduce this upper bound or even make it a constant to make the algorithm linear.

We can easily get a relatively smaller lower bound inspired by the improved Büchi algorithm 5, or from the dual game co-Büchi in the next subsection.

After each iteration of $\text{reach}_{\geq 1}(\cdot,\cdot)$, $|T|$ reduces by at least one. We should also notice that the new unsafe trap after each iteration also has size at least 1. This shows that the target nodes and the unsafe nodes are deleted "in pairs". Therefore, we should have deleted at least two nodes in each iteration. It follows that we have a relatively smaller lower bound $\mathcal{O}(min(|T|, n - |T|) \cdot m) = \mathcal{O}(nm)$.

**Remark.** *We explicitly give this relatively smaller lower bound.*

**Is it possible to have a linear algorithm for Büchi?**

It seems like doing a safety can save us many iterations of reachability. So is it possible to reduce the number of iterations to a constant to have a linear algorithm for Büchi? The answer is no, and we will give the reason after we provide two linear algorithms for Büchi when the arena has only one type of player nodes.

1. If we only have player 1 nodes in the arena, i.e., $\mathcal{A} = (V_1, E)$.

   This is a classic graph in the graph theory. We can first use a linear algorithm to solve for the strongly connected components (SCC) of $V_1$. The remaining arena is

a directed acyclic graph (DAG). We treat all the nodes in the same SCC as one node, and choose the non-trivial SCCs that contain a target node as the target set, and then apply $\text{reach}_{\geq 0}(\cdot, \cdot)$ to find $\mathcal{W}_1$. Since all the nodes are $V_1$, player 1 is fully controllable of the arena. Therefore, $v \in \mathcal{W}_1$ iff $v$ is reachable to a non-trivial SCC that contains a node in $T$. The reason is that, once $v$ reaches a non-trivial SCC that contains a node in $T$, it can infinitely many often reach a target node in the SCC, thus satisfying the winning condition. Conversely, if $v$ can't reach a non-trivial SCC that contains a node in $T$, then $v$ can reach a node in $T$ at most finitely many times since there is no loop that contains a target node which doesn't satisfy the winning condition.

Since solving SCC and reachability are both linear $\mathcal{O}(n+m)$, the time complexity for solving the Büchi game in $\mathcal{A} = (V_1, E)$ is also linear $\mathcal{O}(n+m)$.

2. If we only have player 2 nodes in the arena, i.e., $\mathcal{A} = (V_2, E)$.

   Player 2 is fully controllable of the arena. This Büchi game can be solved by doing one iteration of reachability and safety in the algorithm 5. The time complexity is clearly linear $\mathcal{O}(n+m)$. The reason is that, in a safety game for $\mathcal{A} = (V_2, E)$, all the $V_2$ nodes that are reachable to the unsafe set is deleted. The remaining set doesn't lose the reachability property.

We have given two linear algorithms to solve for the Büchi game when the arena has only $V_1$ or $V_2$. So why is it hard when the arena has both $V_1$ and $V_2$? Even the fully uncontrollable $\mathcal{A} = (V_2, E)$ case has linear algorithm. In other words, why does the improved algorithm 5 still requires multiple iterations to become polynomial?

This can be answered by how the algorithm is performed. After applying $\text{reach}_{\geq 1}(\cdot, \cdot)$, we find the set of nodes that can reach a node in $T$ in at least one step. Then we apply $\text{safe}(\cdot, \cdot)$ to delete the unsafe nodes that can reach $\mathcal{W}_2$.

Now the current set is safe in a sense that it won't be trapped in $\mathcal{W}_2$. However, it may lose the property of reachability that it can reach a node in $T$. This is because a target node that the current set can reach before is deleted in the safety process. This happens when we have an "SCC" without a target node that contains $V_1$ nodes. We can't delete this "SCC" since it has edges that can self-circular inside, but it doesn't have a target node so it doesn't satisfy reaching a target node.

The difference between the current safety and the next reachability are such "SCC"s that can't be deleted in the safety process. Therefore, the most efficient way is to solve for the reachability again, but then we lose the property of safety again. The purpose for us

to alternatively iterate reachability and safety is to gain the property of reachability and safety at the same time when the fixed point is achieved. This is the reason that requires us to apply reachability and safety multiple times.

Back to the Büchi game in $\mathcal{A} = (V_2, E)$, we won't have an "SCC" without a target node after doing a reachability followed by a safety since such "SCC" won't be included in the reachable set in the first place, or it must be deleted in the safety since all the nodes are $V_2$. Therefore, the set after applying the reachability and the safety preserves the property of both reachability and safety, thus satisfies the winning condition.

The difficulty of solving a Büchi game in $\mathcal{A} = (V = V_1 \oplus V_2, E)$ lies in the switching roles of player 1 and player 2 in reachability and safety. $V_1$ are easy to include in reachability, while $V_2$ are easy to delete in safety. The most tricky part comes from $V_1$ being hard to delete in safety because of "SCC", which requires doing reachability again.

**Construction of $\mathcal{O}(nm)$ Example**

It is easy to construct an example that requires $\mathcal{O}(nm)$ time to solve. We will show the performance of the naive algorithm 4 and the improved algorithm 5 by constructing multiple variations of examples. Our final example requires $\mathcal{O}(nm)$ time which shows the worst case for solving Büchi is polynomial, and it can be as worse as $\mathcal{O}(nm)$.

1. **Basic set up** We alternatively lie $V_1$ nodes and $V_2$ nodes in a line, with a $V_1$ node with self-loop being the head. Each node points to its former node. Each $V_2$ node is a target node. The picture is shown in Figure 5.1.

   For $\mathcal{A} = (V, E)$, $|V| = n$ and $|E| = m$.

   Now, $\mathcal{O}(m) = \mathcal{O}(n)$.

   For algorithm 4, it takes about $\mathcal{O}(n)$ iterations of reachability and each reachability takes $\mathcal{O}(n)$ time, thus taking $\mathcal{O}(n^2) = \mathcal{O}(nm)$ time.

   For algorithm 5, it takes only one iteration of reachability and safety, thus taking linear $\mathcal{O}(n + m) = \mathcal{O}(n)$ time.

   From this example, we can see that algorithm 5 outperforms algorithm 4.

2. **First variation** Add self-loop to each $V_1$ node. The picture is shown in Figure 5.2.

   Now, $\mathcal{O}(m) = \mathcal{O}(n)$.

   For algorithm 4, it takes about $\mathcal{O}(n)$ iterations of reachability and each reachability takes $\mathcal{O}(n)$ time, thus taking $\mathcal{O}(n^2) = \mathcal{O}(nm)$ time.

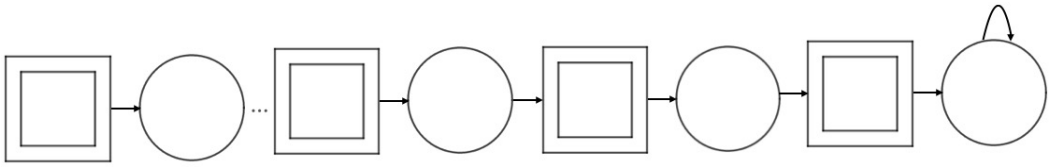

Figure 5.1: Büchi Example 1

For algorithm 5, it takes about $\mathcal{O}(n)$ iterations of reachability and safety, thus taking $\mathcal{O}(n^2) = \mathcal{O}(nm)$ time. We can see that the reason for the time complexity to be polynomial is the self-loops of the $V_1$ nodes, which are of the simplest form of non-trivial SCC. However, we can do an heuristic optimization to the arena by deleting the self-loops of $V_1$ nodes, and then the algorithm turns linear again.

Therefore, from this example, we can see that algorithm 5 still outperforms algorithm 4.

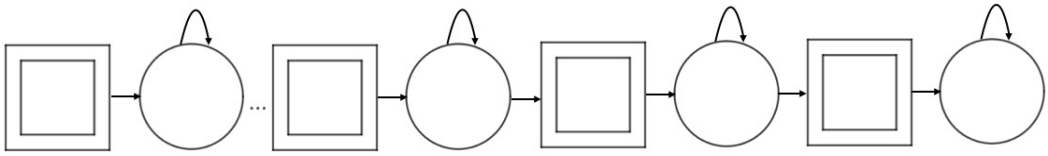

Figure 5.2: Büchi Example 2

3. **Second variation** For each $V_1$ node, add edges to all of its former nodes.

   For each $V_2$ node, add edges to all of its latter nodes. We can also add self-loops for the $V_2$ nodes. The picture is shown in Figure 5.3.

   Now, $\mathcal{O}(m) = \mathcal{O}(n^2)$.

   For algorithm 4, it takes about $\mathcal{O}(n)$ iterations of reachability and each reachability takes $\mathcal{O}(n^2) = \mathcal{O}(m)$ time, thus taking $\mathcal{O}(n^3) = \mathcal{O}(nm)$ time.

   For algorithm 5, it takes about $\mathcal{O}(n)$ iterations of reachability and safety, thus taking $\mathcal{O}(n^3) = \mathcal{O}(nm)$ time.

   If we don't know how the arena is constructed, then it is impossible for us to solve the Büchi game without iteratively applying reachability (and safety). A subtle change can change $\mathcal{W}_1$ from $\emptyset$ to $V$.

   Therefore, from this example, we can see that algorithm 5 even takes $\mathcal{O}(n+m)$ more operations than algorithm 4. The extra operations come from the extra linear safety process.

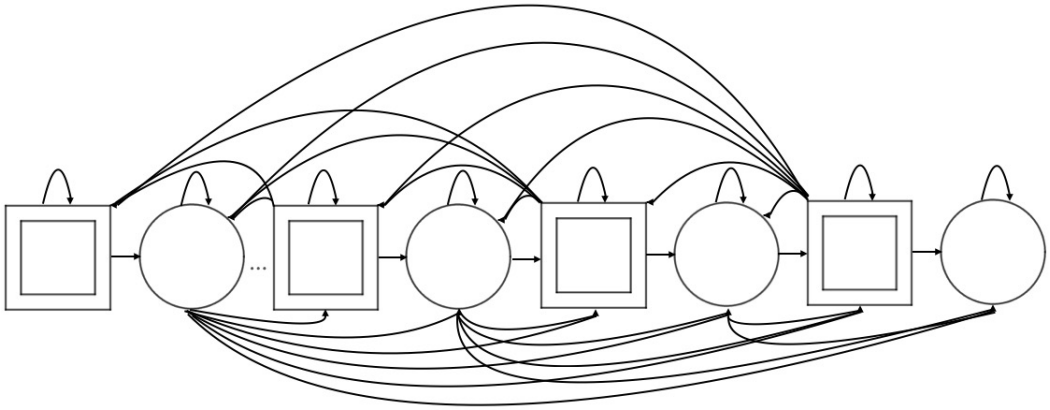

Figure 5.3: Büchi Example 3

### Heuristic Optimization

The problem that causes algorithm 5 to be polynomial is the "SCC"s that can't be deleted in a safety, but can be deleted in the next reachability. Therefore, we give some heuristic optimizations to try to delete these "SCC"s by simplifying the arena and breaking down a big problem into small subproblems. This is using the idea of reducing branches in the graph searching problems, and the idea of divide and conquer.

The nodes in the arena can be partitioned by the types of player or whether it is a target node. Therefore, we partition $V$ into four sets of nodes labeled from 0 to 3, the smaller number has relatively higher priority:

$0 : v \in V_1 \cap T; \quad 1 : v \in V_2 \cap T; \quad 2 : v \in V_1 \cap (V/T); \quad 3 : v \in V_2 \cap (V/T).$

The picture is shown in Figure 5.4.

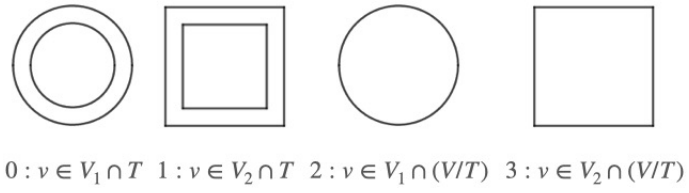


Figure 5.4: Labels of Different Nodes

1. **Delete self-loops** We can delete the self-loops of a node if we know the its type. Deleting each self-loop takes $\mathcal{O}(1)$ time. We will analyze what we can do to a node $v$ of each type 0, 1, 2, 3 when we delete its self-loop.

- 0: $v \in \mathcal{W}_1$, $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V), v)$, $\mathcal{W}_1 += R$.
- 1: We can delete its self-loop first.
  If $v.outdeg == 0$, then $v \in \mathcal{W}_1$, $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V), v)$, $\mathcal{W}_1 += R$.
- 2: We can delete its self-loop first.
  If $v.outdeg == 0$, then $v \in \mathcal{W}_2$, $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V), V/\{v\})$, $\mathcal{W}_2 += V/S$.
- 3: $v \in \mathcal{W}_2$, $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V), V/\{v\})$, $\mathcal{W}_2 += V/S$.

2. **Delete loop between a pair of nodes** $(u, v)$ We can also delete most of the loops between a pair of nodes $(u, v)$. Deleting each loop between $(u, v)$ takes $\mathcal{O}(1)$ time. Since $u$ and $v$ each has 4 types, there are a total of 16 cases. Each $(u, v)$ pair can be represented by a number encoded by $u \times 4 + v$ from 0 to 15. We can cancel out the repetitive cases such as $(u, v)$ and $(v, u)$ to reduce to 10 cases. We will analyze what we can do to a loop between a pair of nodes $(u, v)$.

   - 0: $u, v \in \mathcal{W}_1$, $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V), \{u, v\})$, $\mathcal{W}_1 += R$.
   - 1: $u \to v$
     If $v.outdeg == 0$, then $u, v \in \mathcal{W}_1$, $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V), \{u, v\})$, $\mathcal{W}_1 += R$.
   - 2: $u, v \in \mathcal{W}_1$, $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V), \{u, v\})$, $\mathcal{W}_1 += R$.
   - 3: $u \to v$
     If $v.outdeg == 0$, then $u, v \in \mathcal{W}_1$, $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V), \{u, v\})$, $\mathcal{W}_1 += R$.
   - 5: We can combine $u$, $v$ into 1 node $v'$.
     If $v'.outdeg == 0$, then $u, v \in \mathcal{W}_1$, $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V), \{u, v\})$, $\mathcal{W}_1 += R$.
   - 6: $u \leftarrow v$
     If $u.outdeg == 0$, then $u, v \in \mathcal{W}_1$, $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V), \{u, v\})$, $\mathcal{W}_1 += R$.
   - 7: This is the only case that can't be simplified.
   - 10: We can combine $u$, $v$ into 1 node $v'$.
     If $v'.outdeg == 0$, then $u, v \in \mathcal{W}_2$, $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V), V/\{u, v\})$, $\mathcal{W}_2 += V/S$.
   - 11: $u \leftarrow v$
     If $u.outdeg == 0$, then $u, v \in \mathcal{W}_2$, $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V), V/\{u, v\})$, $\mathcal{W}_2 += V/S$.
   - 15: $u, v \in \mathcal{W}_2$, $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V), V/\{u, v\})$, $\mathcal{W}_2 += V/S$.

3. **Combine SCC** The more general case is to calculate the SCCs directly, and then combing each SCC into a single node of that type. Calculating all the SCCs takes

$\mathcal{O}(n+m)$ in total. However, classifying the SCCs allows us to separate a big problem into several small subproblems, which is using the idea of divide and conquer. We can use an inequality to represent this idea:

$$n = \sum_{i=1}^{m} k_i, \quad \mathcal{O}(f(n)) \geq \mathcal{O}(n), \quad f(n) \geq \sum_{i=1}^{m} f(k_i).$$

We will analyze what we can do after we contract an SCC of a certain type to a single node $v$ of that type.

- 0: $v \in \mathcal{W}_1$, $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V), v)$, $\mathcal{W}_1 += R$.
- 1: If $v.outdeg == 0$, then $v \in \mathcal{W}_1$, $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V), v)$, $\mathcal{W}_1 += R$.
- 2: If $v.outdeg == 0$, then $v \in \mathcal{W}_2$, $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V), V/\{v\})$, $\mathcal{W}_2 += V/S$.
- 3: $v \in \mathcal{W}_2$, $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V), V/\{v\})$, $\mathcal{W}_2 += V/S$.
- SCC of 0 and 2: $v \in \mathcal{W}_1$, $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V), v)$, $\mathcal{W}_1 += R$.
- After each iteration of reachability and safety in the improved algorithm 5, we can add a calculation of SCCs to the entire remaining sub-arena to see whether it has been partitioned into several disjoint sub-arenas. If that is the case, then we can apply algorithm 5 to each of these disjoint sub-arenas instead of solving them together, which will give us some optimality.

If these heuristic optimizations still can't delete the "SCC"s, then we have to iterate multiple times. These "SCC"s are composed of player 1 nodes and player 2 nodes, which can only be deleted in the next reachability.

However, since our arena is not deliberately designed in practice, the safety process can help us reduce most of the iterations by deleting the unsafe target nodes. Therefore, we can expect an almost linear time complexity $\mathcal{O}(n + m)$ to solve a Büchi game in practice even without adding any heuristic optimizations.

### 5.3.2 co-Büchi Game

In this subsection, we will introduce the co-Büchi game that requires eventually always staying in the safe set $S$. We will present the co-Büchi game in a dual way of the Büchi game.

**Winning Condition of the Game**

In a co-Büchi game $C\mathcal{G} = (\mathcal{A}, \text{Obj})$, the objective set $\text{Obj} = S \subseteq V$ is the set of safe nodes and correspondingly, $V/S$ is the set of unsafe nodes. The winning condition for player 1 is to force the token to eventually always stay in $S$, or equivalently, finitely many times reach $V/S$. Formally, player 1 wins iff $\text{Inf}(\pi) \subseteq S$, or equivalently, $\text{Inf}(\pi) \cap (V/S) = \emptyset$.

**Idea of the Algorithm**

The co-Büchi algorithm we will present is the dual algorithm of the improved Büchi algorithm 5. The basic idea of the algorithm is to iteratively apply $\text{safe}(\cdot, \cdot)$ and $\text{reach}_{\geq 0}(\cdot, \cdot)$ until fixed point iteration.

Our goal is to force the token to eventually always stay in $S$. We consider the problem backwards, which is to find the set of nodes that satisfies our final condition first. The final condition is to find the nodes $X \subseteq S$ that can always stay in $S$, which is a safety game. It is clear that $X \subseteq \mathcal{W}_1$. Then we find the set of nodes that can reach $X$ by applying $\text{reach}_{\geq 0}(\cdot, \cdot)$, which also satisfies eventually always stay in $S$. In this reachability process, we first find nodes in $V/S$, then we may find nodes back in $S$. This reachable set of $X$ is in $\mathcal{W}_1$, so in the next iteration, we are working in the sub-arena $\mathcal{A}'(V/\mathcal{W}_1)$, which is monotonically decreasing. We have to repeat the process of safety and reachability because in the remaining sub-arena, there may contain a safe set $X \subseteq S$ again. In each iteration, at least 1 node in $S$ is determined to be in $\mathcal{W}_1$. We continue this process until the iteration terminates in the $\bar{i}^{th}$ iteration when the safe set $X \subseteq S$ is empty. Then no more nodes can satisfy eventually always staying in $S$ since there is no safe set $X \subseteq S$.

**Algorithm**

Algorithm 6 is the algorithm we use to solve for the co-Büchi game. It's a fixed point

iteration of safe$(\cdot,\cdot)$ and reach$_{\geq 0}(\cdot,\cdot)$.

**Algorithm 6:** $\mathcal{A}'(\mathcal{W}_1)$ = co-Büchi$(\mathcal{A}, S)$ Fixed Point Iteration of Safe and Reach

**Input:**
1. $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$;
2. $S \subseteq V$;

**Output:**
1. $\mathcal{A}'(\mathcal{W}_1)$;

```
for (V.iterate = 0, V' = V, S' = S, i = 1, W_1 = ∅; ; i++) do
    A'(X) = safe(A'(V'), S');
    if X == ∅ then break;
    A'(R) = reach_{≥0}(A'(V'), X);
    W_1 += R;
    R.iterate = i;
    if (R == X) ∨ (R == V') then break;
    V' -= R;
    S' = S' ∩ V';
    if S' == ∅ then break;
W_2 = V/W_1;
```

**Result:**
1. Winning set $\mathcal{W}_1 \oplus \mathcal{W}_2 = V$;
2. Iterate number of each $v \in V$ is labeled by $v.iterate \geq 0$;
3. Winning strategy of each node in $V_1 \cap \mathcal{W}_1$ is computed by safe$(\cdot,\cdot)$ and reach$_{\geq 0}(\cdot,\cdot)$ in the corresponding process;

### Proof of Soundness and Completeness

### Soundness

Conceptually, the process of solving $\mathcal{W}_1$ in a co-Büchi game can be seen as a fixed point iteration induced by $\mathcal{A}'(X_i) = \text{safe}(\mathcal{A}'(V_i), S_i)$ and $\mathcal{A}'(R_i) = \text{reach}_{\geq 0}(\mathcal{A}'(V_i), X_i)$, where $V_1 = V$, $S_1 = S$, $V_{i+1} = V_i - R_i$ and $S_{i+1} = S_i \cap V_{i+1}$. The fixed point iteration terminates in the $\bar{i}^{th}$ iteration when $X_{\bar{i}} = \emptyset$. The number of fixed point iteration $\bar{i}$ is thus defined as the minimum number of $i$ such that $X_i = \emptyset$. $\mathcal{W}_1 = R_1 \oplus R_2 \oplus \cdots \oplus R_{\bar{i}}$ and $\mathcal{W}_2 = V_{\bar{i}} = V/\mathcal{W}_1$. The nodes in $R_i$ can leave $S$ for at most $i$ times, $1 \leq i \leq \bar{i}$. If a node $v \in \mathcal{W}_1$ is not in $S$, then it will return to $S$ within $|V|$ steps which is finite. Since the fixed point iteration is achieved after the $\bar{i}^{th}$ iteration, the nodes in $\mathcal{W}_1$ can leave $S$ for at most $\bar{i}$ times which is finite. Therefore, the nodes in $\mathcal{W}_1$ will eventually stay in $S$, even though we can't determine when that eventually is. What we do know is that the nodes in $\mathcal{W}_1$

can violate staying in $S$ for at most $\bar{i}$ times. Since the size of $S$ reduces by $|S_i/S_{i+1}| \geq 1$ in the $i^{th}$ iteration, the iteration must terminate in $\bar{i} \leq |S|$ iterations. This gives the proof of soundness.

**Completeness**

Since the fixed point iteration is achieved after the $\bar{i}^{th}$ iteration, the safe set $X_{\bar{i}} \subseteq S_{\bar{i}}$ is empty. Therefore, no nodes in $\mathcal{W}_2 = V_{\bar{i}}$ can eventually stay in $S$.

In algorithm 6, each node $v \in V$ is labeled with an iterate number indicating the number of times $v$ participated in the iteration of reachability.

We have $\mathcal{W}_1 = \{v | 1 \leq v.iterate \leq \bar{i}\}$, and $\mathcal{W}_2 = \{v | v.iterate = 0\}$.

Nodes in $\mathcal{W}_1$ has $1 \leq v.iterate = i \leq \bar{i}$, indicating $v$ has participated in the $i^{th}$ iteration of reachability, and $v$ can leave $S$ for at most $i$ times.

Nodes in $\mathcal{W}_2$ has $v.iterate = 0$, indicating $v$ can't be attracted into a safe set. In other words, $v \in V_2 \cap \mathcal{W}_2$ has a strategy from the Büchi game to force the token infinitely many often reach $V/S$, thus violating the winning condition. This gives the proof of completeness.

**Strategy**

Consider the nodes $v \in V_1 \cap \mathcal{W}_1$. If $v$ is determined in a safety process, i.e., $v \in X_i$, then $v.strategy$ is given by the safety algorithm to "stay" in $X_i \subseteq S$. If $v$ is determined in a reachability process, $v \in R_i/X_i$, then $v.strategy$ is given by the reachability algorithm to reach $X_i \subseteq S$. We quoted "stay" for the nodes in $X_i \subseteq S$, since a node $v' \in V_2 \cap X_i$ may choose to stay in $X_i$, or, to leave $X_i$ to enter $R_{\hat{i}}$, $\hat{i} < i$, but then we know the nodes in $R_{\hat{i}}$ are in $X_{\hat{i}} \subseteq S$ or can reach $X_{\hat{i}} \subseteq S$ in finite steps. Since $\hat{i}$ can only be reduced to 1, and $X_1 \subseteq S$ is a real safe set of $S$, thus inductively $v$ can leave $S$ for at most finitely many times.

We can see that the iterate number of the nodes in a play $\pi$ initiated at $v_0 \in \mathcal{W}_1$ is non-increasing:

- The iterate number stays the same in the process from $R_i$ to reach $X_i \subseteq S$, or staying in $X_i \subseteq S$.
- The iterate number is decreasing in the process from $X_i$ to $R_{\hat{i}}$, $\hat{i} < i$.

We can't determine when that "eventually" is to satisfy always staying in $S$ because it may depend on player 2 to choose whether to stay in $X_i$ or leave $X_i$. But even if player 2 chooses to leave $X_i$, it can still leave $S$ for at most $\bar{i}$ times, or more precisely, for at most

$|\mathcal{W}_1/S|$ steps, which is finite. This allows us to claim that our strategy satisfies eventually always staying in $S$. If player 2 violates staying in $S$, then we can argue that it's not the time for "eventually" yet, and player 2 can violate staying in $S$ for at most finitely $\bar{i}$ times or $|\mathcal{W}_1/S|$ steps, so that our argument still holds.

This shows that such strategy is indeed a winning strategy for player 1. The local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.

**Complexity Analysis**

In a co-Büchi game $C\mathcal{G} = (\mathcal{A}, \text{Obj})$, $\mathcal{A} = (V, E)$ is the arena and $\text{Obj} = S \subseteq V$ is the objective set. $|V| = n$ is the number of nodes and $|E| = m$ is the number of edges.

**Proposition 5.3.2.** *A co-Büchi game can be solved in $\mathcal{O}(m)$ space, $\mathcal{O}(nm)$ time, and the local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Space Complexity:** $\mathcal{O}(n + m)$

Since the size of the arena is depicted by $n$ nodes and $m$ edges, the space complexity of the algorithm is $\mathcal{O}(n + m)$.

**Time Complexity:** $\mathcal{O}(nm)$

In the co-Büchi algorithm 6, each $\text{safe}(\cdot, \cdot)$ costs $\mathcal{O}(m)$. $|S|$ reduces by at least one after each iteration, thus there can be at most $\bar{i} \leq |S| \leq n$ iterations. The total cost of $\text{safe}(\cdot, \cdot)$ is thus $\mathcal{O}(|S| \cdot m) = \mathcal{O}(nm)$.

The total cost of $\text{reach}_{\geq 0}(\cdot, \cdot)$ is linear, i.e., $\mathcal{O}\left(\sum_{i=1}^{\bar{i}} \text{reach}_{\geq 0}(\cdot, \cdot)\right) = \mathcal{O}(m)$. It is likely to reduce the number of iterations if any $v \in S$ is determined in the reachability process, which may result in a smaller multiple of $\mathcal{O}(m)$. This is usually the case in practice since co-Büchi is the dual game for Büchi. The time complexity of the algorithm is thus polynomial w.r.t. $n$ and $m$, i.e, $\mathcal{O}(|S| \cdot m) = \mathcal{O}(nm)$.

**More about co-Büchi Game**

In the co-Büchi game, we can restrict the arena of the safety algorithm to $\mathcal{A}'(S)$. The safe set is defined as $S/pre(V/S)$. This makes the algorithm polynomial in $S$ and linear in $V/S$, which can save all the unnecessary operations in $V/S$. This shows that one step of enqueue calculation $pre(V/S)$ in the first iteration allows us to work in $\mathcal{A}'(S)$ instead of in $\mathcal{A}'(V)$, which saves us all the calculations in $V/S$. Such savings accumulate in a polynomial algorithm like co-Büchi.

Similar to the analysis for the Büchi algorithm, we can easily get the same relatively smaller lower bound for co-Büchi. After each iteration of safe$(\cdot,\cdot)$, $|S|$ reduces by at least one. We should also notice that the new reachable set $R_i/X_i$ after each iteration also has size at least 1, otherwise the safe set in the next iteration would be empty and the fixed point will be achieved. We can see that the safe nodes and the reachable nodes are determined "in pairs". Therefore, we should have determined at least two nodes in each iteration for $\mathcal{W}_1$. It follows that we have a relatively smaller lower bound $\mathcal{O}(min(|S|, n-|S|)\cdot m) = \mathcal{O}(nm)$.

We can also get this relatively smaller lower bound for Büchi and co-Büchi when considering these two games together: the target set $T$ in a Büchi game is the unsafe set $V/S$ in its dual co-Büchi game. Since Büchi and co-Büchi are essentially the same game from the perspective of different players, the time complexity for the two games should be the same. From the perspective of the Büchi game, we have $\mathcal{O}(|T|\cdot m) = \mathcal{O}(|V/S|\cdot m)$; from the perspective of the co-Büchi game, we have $\mathcal{O}(|S|\cdot m)$. This directly gives us a tighter bound by taking the min of these two: $\mathcal{O}(min(|S|, n-|S|)\cdot m) = \mathcal{O}(nm)$.

Note that $min(|S|, n-|S|) \leq \frac{n}{2}$. Actually, if we really want to do a detailed analysis to see how small the coefficient for $\mathcal{O}(nm)$ can be, then we can introduce the average degree of the arena $a$ given by $m = a\cdot n$.

In the $i^{th}$ iteration, it takes $\mathcal{O}(a\cdot i)$ operations. The number of nodes reduces by at least two in each iterations, thus $\sum_{i=0}^{min(|S|,n-|S|)} a\cdot(n-2i) \leq a\cdot(1+n)\cdot\frac{n}{2}\cdot\frac{1}{2} \approx \frac{an^2}{4} = \frac{1}{4}nm$.

Therefore, the polynomial part of the time complexity is given by

$\mathcal{O}\left(\sum_{i=1}^{min(|S|,n-|S|)} a\cdot i\right) = \mathcal{O}(\frac{1}{4}nm)$, with a coefficient $\frac{1}{4}$.

## 5.4 Generalized Büchi and Generalized co-Büchi

In this section, we will introduce the generalized Büchi game [8] and the generalized co-Büchi game. The algorithms used in these two games are constructed by modifying the algorithms for Büchi and co-Büchi respectively.

### 5.4.1 Generalized Büchi Game

In this subsection, we will introduce the generalized Büchi game that requires reaching the target sets $T_i$ infinitely many often for all $i$, $1 \leq i \leq k$.

**Winning Condition of the Game**

In a generalized Büchi game $GB\mathcal{G} = (\mathcal{A}, \text{Obj})$, the objective set $\text{Obj} = \{T_1, \cdots, T_k\}$, where $T_i \subseteq V$, $1 \leq i \leq k$ are the $k$ sets of target nodes. The winning condition for player 1 is to force the token to reach $T_i$ infinitely many often for all $i$, $1 \leq i \leq k$. Formally, player 1 wins iff $\bigwedge_{i=1}^{k} (\text{Inf}(\pi) \cap T_i \neq \emptyset)$.

**Three Methods to Solve the Game**

There are at least three methods to solve a generalized Büchi game with $k$ target sets in $\mathcal{A}$:

1. **Method 1** Solve it as a generalized Büchi game in $\mathcal{A}$.
2. **Method 2** Convert the game to a Büchi game in $k$-copies of $\mathcal{A}$.
3. **Method 3** Convert the game to a Büchi game in $(k+1)$-copies of $\mathcal{A}$.

In a generalized Büchi game with $k$ Büchi objectives, we require satisfying all of the $k$ Büchi objectives. Therefore, a generalized $k$ Büchi objective is also considered as the conjunction of $k$ Büchi objectives as shown in the form of the winning condition.

As in method 2 and 3, we unfold the arena $\mathcal{A}$ to $k$ or $(k+1)$ copies to convert the generalized Büchi game to a Büchi game. The benefit of introducing the generalized Büchi algorithm in method 1 is for efficiency in both time and space.

Method 1 is the composition of method 2 and 3, and method 2 and 3 are the decompositions of method 1. We will first introduce method 2 and 3 to give the conversion from a generalized Büchi game to a Büchi game, and then introducing a new generalized Büchi algorithm in method 1.

**Method 2**

In this subsubsection, we will discuss method 2 that converts the generalized $k$ Büchi game to a Büchi game in $k$-copies of $\mathcal{A}$.

Since a game $\mathcal{G} = (\mathcal{A}, \text{Obj})$ is a pair of arena $\mathcal{A}$ and objective set Obj, we will show the conversion by constructing the Büchi game $\mathcal{G}' = (\mathcal{A}', \text{Obj}')$ from the generalized Büchi game $\mathcal{G} = (\mathcal{A}, \text{Obj})$.

**Arena**

The procedure to convert the arena $\mathcal{A}$ of the generalized Büchi game to the arena $\mathcal{A}'$ of the Büchi game is as follows:

1. list $k$-copies of $\mathcal{A}$ from 1 to $k$, i.e., $\mathcal{A}_1, \cdots, \mathcal{A}_k$;
2. on $\mathcal{A}_i$ for $1 \leq i \leq k$, for each $v \in T_i$, connect each of its out-going edges from $\mathcal{A}_i$ to the corresponding $\mathcal{A}_{i+_k 1}$, where $a +_k b$ is defined as $(a \bmod k) + b$.

The idea behind this construction of arena $\mathcal{A}'$ is that, when $T_i$ is reached in $\mathcal{A}_i$, the out-going edges bridge us to $\mathcal{A}_{i+_k 1}$. In other words, the token can reach $\mathcal{A}_{i+_k 1}$ from $\mathcal{A}_i$ iff it can reach $T_i$ in $\mathcal{A}_i$. Therefore, $\mathcal{A}'$ forms a closed loop, and the bridges are the outgoing edges of $T_i$ in $\mathcal{A}_i$.

**Objective Set**

The procedure to convert the objective set Obj of the generalized Büchi game to the objective set Obj$'$ of the Büchi game is as follows: we can choose any of the target set $T_i$ in $\mathcal{A}_i$ as our target set for the Büchi game.

W.L.O.G., we consider $T_1$ in $\mathcal{A}_1$ as our target set, since the other choices are only a matter of permutations.

The idea behind this construction of objective set Obj$'$ is that, from the construction of $\mathcal{A}'$, reaching $T_1$ in $\mathcal{A}_1$ infinitely many often is equivalent to reaching $T_i$ in $\mathcal{A}_i$ infinitely many often for all $i$, $1 \leq i \leq k$. This is because $\mathcal{A}'$ forms a closed loop and is connected by each $T_i$ in $\mathcal{A}_i$.

**Winning Condition**

The winning condition of the Büchi game $\mathcal{G}' = (\mathcal{A}', \text{Obj}')$ is defined as reaching $T_1$ in $\mathcal{A}_1$ infinitely many often, i.e., $\text{Inf}(\pi) \cap T_1 \neq \emptyset$, $T_1$ is in $\mathcal{A}_1$.

**Winning Set from Büchi back to generalized Büchi**

The winning set in any copy of $\mathcal{A}$ in $\mathcal{G}'$ is the winning set for $\mathcal{G}$ since the winning set in each copy of $\mathcal{A}$ in $\mathcal{G}'$ is identical. This result is not surprising since $\mathcal{A}'$ can be seen as piling up $k$-copies of $\mathcal{A}$.

**Winning Strategy from Büchi back to generalized Büchi**

The mapping of the strategy from the Büchi game $\mathcal{G}'$ back to the generalized Büchi game $\mathcal{G}$ is straightforward: we start from using the strategy in $\mathcal{A}_1$ to reach $T_1$, and then using the strategy in $\mathcal{A}_2$ to reach $T_2$, etc. Generally, we use the strategy in $\mathcal{A}_i$ to reach $T_i$. We satisfy the generalized Büchi objective by reaching $T_1, \cdots, T_k$ in a row over and over again. The local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.

**Complexity Analysis**

In a generalized Büchi game $GB\mathcal{G} = (\mathcal{A}, \text{Obj})$, the objective set $\text{Obj} = \{T_1, \cdots, T_k\}$, where $T_i \subseteq V$, $1 \leq i \leq k$ are the $k$ sets of target nodes. $|V| = n$ is the number of nodes and $|E| = m$ is the number of edges.

Now we consider the converted Büchi game $\mathcal{G}' = (\mathcal{A}', \text{Obj}')$.

**Proposition 5.4.1.** *Converting a generalized $k$ Büchi game to a Büchi game in $k-$copies of $\mathcal{A}$ can be solved in $\mathcal{O}(km)$ space, $\mathcal{O}(knm)$ time, and the local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Space Complexity:** $\mathcal{O}(km)$

Since $\mathcal{A}'$ is $k$-copies of $\mathcal{A}$, and $\mathcal{O}(|\mathcal{A}|) = \mathcal{O}(n+m)$, we have $\mathcal{A}' = (V', E')$, $\mathcal{O}(|V'| = kn)$, $\mathcal{O}(|E'| = km)$, and $\mathcal{O}(|\mathcal{A}'|) = \mathcal{O}(k(n+m)) = \mathcal{O}(km)$.

**Time Complexity:** $\mathcal{O}(knm)$

Since the target set for the Büchi game is $T_1$ in $\mathcal{A}_1$, $|T_1| \leq n$.

From the complexity analysis of the Büchi game, the time complexity for $\mathcal{G}'$ is given by $\mathcal{O}(|T_1| \cdot |E'|) = \mathcal{O}(|T_1| \cdot km) = \mathcal{O}(k \cdot |T_1| \cdot m) = \mathcal{O}(knm)$.

**Method 3**

In this subsubsection, we will discuss method 3 that converts the generalized $k$ Büchi game to a Büchi game in $(k+1)$-copies of $\mathcal{A}$.

The conversion in method 3 is similar to that in method 2, which can be seen as a rougher version based on method 2. The method 3 adds an additional copy of $\mathcal{A}$ from method 2 as a counter, and choose this copy of $\mathcal{A}$ as the target set for the Büchi game.

Since a game $\mathcal{G} = (\mathcal{A}, \text{Obj})$ is a pair of arena $\mathcal{A}$ and objective set Obj, we will show the conversion by constructing the Büchi game $\mathcal{G}'' = (\mathcal{A}'', \text{Obj}'')$ from the generalized Büchi game $\mathcal{G} = (\mathcal{A}, \text{Obj})$.

**Arena**

The procedure to convert the arena $\mathcal{A}$ of the generalized Büchi game to the arena $\mathcal{A}''$ of the Büchi game is as follows:

1. list $(k+1)$-copies of $\mathcal{A}$ from 1 to $k+1$, i.e., $\mathcal{A}_1, \cdots, \mathcal{A}_{k+1}$;
2. on $\mathcal{A}_i$ for $1 \leq i \leq k$, for each $v \in T_i$, connect each of its out-going edges from $\mathcal{A}_i$ to the corresponding $\mathcal{A}_{i+1}$;
3. on $\mathcal{A}_{k+1}$, for each $v \in \mathcal{A}_{k+1}$, connect each of its out-going edges from $\mathcal{A}_{k+1}$ to the corresponding $\mathcal{A}_1$.

The idea behind this construction of arena $\mathcal{A}''$ is that, for $1 \leq i \leq k$, when $T_i$ is reached in $\mathcal{A}_i$, the out-going edges bridge us to $\mathcal{A}_{i+1}$; the out-going edges in $\mathcal{A}_{k+1}$ directly bridge us to $\mathcal{A}_1$. In other words, the token can reach $\mathcal{A}_{i+1}$ from $\mathcal{A}_i$ iff it can reach $T_i$ in $\mathcal{A}_i$ for $1 \leq i \leq k$, and $\mathcal{A}_{k+1}$ only serves as a counter since if the token is in $\mathcal{A}_{k+1}$, then it will go to $\mathcal{A}_1$ in the next move anyway. Therefore, $\mathcal{A}''$ forms a closed loop, and the bridges are the outgoing edges of $T_i$ in $\mathcal{A}_i$ connecting $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ for $1 \leq i \leq k$, and $\mathcal{A}_{k+1}$ closes the loop by connecting $\mathcal{A}_1$.

**Objective Set**

The procedure to convert the objective set Obj of the generalized Büchi game to the objective set Obj$''$ of the Büchi game is as follows: we can choose any of the target set $T_i$ in $\mathcal{A}_i$, for $1 \leq i \leq k$, or the nodes in $\mathcal{A}_{k+1}$ as our target set for the Büchi game.

Choosing any of the target set $T_i$ in $\mathcal{A}_i$, for $1 \leq i \leq k$, is the same as in method 2, and is less efficient since it takes one more copy of $\mathcal{A}$. Therefore, we only consider choosing the nodes in $\mathcal{A}_{k+1}$ as our target set for the Büchi game $\mathcal{G}''$ here.

The idea behind this construction of objective set Obj$''$ is that, from the construction of $\mathcal{A}''$, reaching $\mathcal{A}_{k+1}$ infinitely many often is equivalent to reaching $T_i$ in $\mathcal{A}_i$ infinitely many often for all $i$, $1 \leq i \leq k$. This is because nodes in $\mathcal{A}_{k+1}$ directly go to $\mathcal{A}_1$ in the next move, and reaching $\mathcal{A}_{k+1}$ again requires traveling through each $\mathcal{A}_i$ in $\mathcal{A}''$.

**Winning Condition**

The winning condition of the Büchi game $\mathcal{G}'' = (\mathcal{A}'', \text{Obj}'')$ is defined as reaching $\mathcal{A}_{k+1}$ infinitely many often, i.e., $\text{Inf}(\pi) \cap V_{k+1} \neq \emptyset$, $V_{k+1}$ is in $\mathcal{A}_{k+1}$.

**Winning Set from Büchi back to generalized Büchi**

The winning set in any copy of $\mathcal{A}$ in $\mathcal{G}''$ is the winning set for $\mathcal{G}$ since the winning set in each copy of $\mathcal{A}$ in $\mathcal{G}''$ is identical. This result is not surprising since $\mathcal{A}''$ can be seen as piling up $(k+1)$-copies of $\mathcal{A}$.

**Winning Strategy from Büchi back to generalized Büchi**

The mapping of the strategy from the Büchi game $\mathcal{G}''$ back to the generalized Büchi game $\mathcal{G}$ is straightforward: we start from using the strategy in $\mathcal{A}_1$ to reach $T_1$, and then using the strategy in $\mathcal{A}_2$ to reach $T_2$, etc. Generally, we use the strategy in $\mathcal{A}_i$ to reach $T_i$, for $1 \leq i \leq k$. We satisfy the generalized Büchi objective by reaching $T_1, \cdots, T_k$ in a row over and over again. Reaching $\mathcal{A}_{k+1}$ each time means we have reached each of $T_1, \cdots, T_k$ once again. The local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.

**Complexity Analysis**

In a generalized Büchi game $GB\mathcal{G} = (\mathcal{A}, \text{Obj})$, the objective set $\text{Obj} = \{T_1, \cdots, T_k\}$, where $T_i \subseteq V$, $1 \leq i \leq k$ are the $k$ sets of target nodes. $|V| = n$ is the number of nodes and $|E| = m$ is the number of edges.

Now we consider the converted Büchi game $\mathcal{G}'' = (\mathcal{A}'', \text{Obj}'')$.

**Proposition 5.4.2.** *Converting a generalized $k$ Büchi game to a Büchi game in $(k+1)-$copies of $\mathcal{A}$ can be solved in $\mathcal{O}((k+1)m)$ space, $\mathcal{O}((k+1)nm)$ time, and the local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Space Complexity:** $\mathcal{O}((k+1)m)$

Since $\mathcal{A}''$ is $(k+1)$-copies of $\mathcal{A}$, and $\mathcal{O}(|\mathcal{A}|) = \mathcal{O}(n+m)$, we have $\mathcal{A}'' = (V'', E'')$, $\mathcal{O}(|V''| = (k+1)n)$, $\mathcal{O}(|E''| = (k+1)m)$, and $\mathcal{O}(|\mathcal{A}''|) = \mathcal{O}((k+1)(n+m)) = \mathcal{O}((k+1)m)$.

**Time Complexity:** $\mathcal{O}((k+1)nm)$

Since the target set for the Büchi game is $V_{k+1}$, which is the set of nodes in $\mathcal{A}_{k+1}$, $|V_{k+1}| = n$.

However, if we look closer, we can see that choosing $V_{k+1}$ as the target set is equivalent as choosing $T_k$ in $\mathcal{A}_k$ as the target set. This is because the only nodes in $\mathcal{A}_{k+1}$ that have in-going edges are the ones whose predecessors are the $T_k$ in $\mathcal{A}_k$. The other nodes in $\mathcal{A}_{k+1}$ have in-degree 0 and will not form a closed loop. In other words, $Pre(V_{k+1}) = T_k$, $T_k$ is in $\mathcal{A}_k$.

Therefore, when computing the time complexity of the Büchi game, we can treat the target set as $T_k$ in $\mathcal{A}_k$, $|T_k| \leq n$.

From the complexity analysis of the Büchi game, the time complexity for $\mathcal{G}''$ is given by $\mathcal{O}(|V_{k+1}| \cdot |E''|) = \mathcal{O}(|T_k| \cdot (k+1)m) = \mathcal{O}((k+1) \cdot |T_k| \cdot m) = \mathcal{O}((k+1)nm)$.

We can change $T_k$ to any $T_1, \cdots, T_k$ since the other choice is only a matter of permutation.

### Method 1

In this subsubsection, we will discuss method 1 that solves the generalized $k$ Büchi game using the generalized Büchi algorithm in $\mathcal{A}$.

Method 1 works in $\mathcal{A}$, thus can be seen as a more compact expression of method 2 and 3. Since method 1 is more compact, it is more efficient than method 2 and 3 in implementations.

#### Idea of the Algorithm

The basic idea of the generalized Büchi algorithm is to iteratively apply the Büchi algorithm until fixed point iteration.

The question is what is the order for us to apply the Büchi algorithm? Our focus is always on achieving the Büchi objective for $T_1$:

- step 1: when $T_1$ hasn't achieved Büchi objective, we apply the improved Büchi algorithm 5 for $T_1$ to achieve Büchi objective for $T_1$;
- step 2: when $T_1$ has achieved Büchi objective, we apply the Büchi algorithm for $T_2, \cdots, T_k$ to see whether $T_2, \cdots, T_k$ achieve the Büchi objective;

  if one of $T_2, \cdots, T_k$ doesn't achieve the Büchi objective, then we go back to step 1; else the Büchi objective has been achieved for each $T_1, \cdots, T_k$, thus the generalized Büchi objective is achieved.

#### Algorithm

Algorithm 7 is the algorithm we use to solve for the generalized Büchi game. It's a

fixed point iteration of the $k$ Büchi objectives.

**Algorithm 7:** $\mathcal{A}'(\mathcal{W}_1) = \text{Generalized Büchi}(\mathcal{A}, T_1, \cdots, T_k)$

**Input:**
1. $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$;
2. $T_i \subseteq V$, $1 \leq i \leq k$;

**Output:**
1. $\mathcal{A}'(\mathcal{W}_1)$;

```
for (V' = V, T'_1 = T_1, ··· , T'_k = T_k, i = 1, j = 0; i ≤ k; j++) do
    T'_i = T'_i ∩ V';
    A'(R_0) = reach_{≥0}(A'(V'), T'_i);
    A'(R_1) = reach_{≥1}(A'(V'), T'_i);
    if R_0 == V' then i++; continue;
    V' = R_0;
    if |R_0| > |R_1| then
        A'(V') = safe(A'(R_0), R_1);
        if V' == ∅ then break;
    i = 1;
W_1 = V';
W_2 = V/W_1;
```

**Result:**
1. Winning set $\mathcal{W}_1 \oplus \mathcal{W}_2 = V$;
2. Winning strategy of each node in $V_1 \cap \mathcal{W}_1$ for each Büchi objective is computed by $\text{reach}_{\geq 1}(\cdot, \cdot)$;

### Proof of Soundness and Completeness

#### Soundness

The generalized Büchi algorithm terminates when all of the $k$ target sets satisfy the Büchi objectives. Therefore, the nodes in $\mathcal{W}_1$ can reach $T_i$ infinitely many often for all $1 \leq i \leq k$ by reaching $T_1, \cdots, T_k$ in a row again and again. This gives the proof of soundness.

#### Completeness

For any $v \in \mathcal{W}_2$, it must have been deleted in the iterations for one of the Büchi objectives $T_i$, $1 \leq i \leq k$. Therefore, $v$ can't reach $T_i$ infinitely many often for all $i$, $1 \leq i \leq k$. This gives the proof of completeness.

#### Strategy

The strategy for reaching $T_i$ is computed in the $\text{reach}_{\geq 1}(\mathcal{A}'(V'), T_i')$ process for each $1 \leq i \leq k$.

Similar to the Büchi game, only the strategy in the last iteration when all the $k$ Büchi objectives are achieved at the same time is the real strategy. All the strategies computed in the previous iterations are nominal.

For $v \in V_1 \cap \mathcal{W}_1$, from the analysis of the strategy in a reachability game, we know that the strategy to reach $T_i$ can force the token to reach $T_i$, for $1 \leq i \leq k$.

Our strategy to satisfy the generalized Büchi objective is to reach $T_1, \cdots, T_k$ in a row again and again. We use a unit counter to indicate the current target $T_i$ to reach. After reaching $T_i$ using the strategy for reaching $T_i$, the counter increments to $i+1$ and we use the strategy for reaching $T_{i+1}$ to reach $T_{i+1}$. For the counter, we clear the number to 1 each time when $T_k$ is reached.

By repeatedly reaching $T_1, \cdots, T_k$, we inductively show that $v \in \mathcal{W}_1$ can reach $T_i$ infinitely many often for all $i$, $1 \leq i \leq k$. This shows that such strategy is indeed a winning strategy for player 1.

The local strategy takes $k$ units for each node $v \in V_1 \cap \mathcal{W}_1$ since we need one unit for the local strategy for each of the $k$ Büchi objectives. We can see that the strategy for the generalized Büchi game consists of $k$ local strategies for the $k$ Büchi objectives, and an additional counter indicating the current Büchi objective. Therefore, the strategy for the generalized Büchi game is still almost local.

**Complexity Analysis**

In a generalized Büchi game $GB\mathcal{G} = (\mathcal{A}, \text{Obj})$, $\mathcal{A} = (V, E)$ is the arena and

$\text{Obj} = \{T_1, \cdots, T_k\}$ is the objective set, where $T_i \subseteq V$, $1 \leq i \leq k$ are the $k$ sets of target nodes. $|V| = n$ is the number of nodes and $|E| = m$ is the number of edges.

**Proposition 5.4.3.** *A generalized $k$ Büchi game can be solved using generalized Büchi algorithm in $\mathcal{O}(m)$ space, $\mathcal{O}(knm)$ time, and the local strategy takes $k$ units for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Space Complexity:** $\mathcal{O}(n+m)$

Since the size of the arena is depicted by $n$ nodes and $m$ edges, the space complexity of the algorithm is $\mathcal{O}(n+m)$.

**Time Complexity:** $\mathcal{O}(knm)$

We always focus on achieving the Büchi objective for $T_1$.

After each iteration of reachability and safety before the Büchi objective for $T_1$ is achieved, the size of $|T_1|$ reduces by at least one.

When $T_1$ has achieved Büchi objective, we apply the Büchi algorithm for $T_2, \cdots, T_k$ to see whether $T_2, \cdots, T_k$ achieve the Büchi objective.

If $T_i$, for $2 \leq i \leq k$, doesn't achieve the Büchi objective, then the size of $|T_i|$ reduces by at least one. We claim that the size of $|T_1|$ also reduces by at least one.

The reason is that, when the Büchi objective for $T_1$ is achieved, we have

$\mathcal{A}'(R_1) = \text{reach}_{\geq 1}(\mathcal{A}'(R_1), T_1')$, and $\mathcal{A}'(R_1)$ is the current sub-arena. If $T_i$ doesn't achieve that Büchi objective, for $2 \leq i \leq k$, we have $\mathcal{A}'(R_1') = \text{reach}_{\geq 1}(\mathcal{A}'(R_1), T_i')$, and $|R_1'| < |R_1|$.

We claim that $T_1' \subseteq R_1$ but $T_1' \nsubseteq R_1'$ by contradiction.

Suppose $T_1' \subseteq R_1'$, then $\mathcal{A}'(R_1) = \text{reach}_{\geq 1}(\mathcal{A}'(R_1), T_1')$ gives $\mathcal{A}'(R_1) = \text{reach}_{\geq 1}(\mathcal{A}'(R_1), R_1')$.

The reachability property tells us that, if $\mathcal{A}'(R_1') = \text{reach}_{\geq 1}(\mathcal{A}'(R_1), T_i')$, then

$\mathcal{A}'(R_1') = \text{reach}_{\geq 1}(\mathcal{A}'(R_1), R_1')$, since $T_i' \subseteq R_1'$.

This gives $R_1' = R_1$, which contradicts to $|R_1'| < |R_1|$.

Therefore, after the iteration of reachability for $T_i$, $2 \leq i \leq k$, whose Büchi objective is not achieved yet, the size of $|T_1|$ reduces by at least one.

In algorithm 7, doing reachability for each target set $T_i$, i.e., $\text{reach}_{\geq 1}(\cdot, \cdot)$, costs $\mathcal{O}(m)$. $|T_1|$ reduces by at least one after doing reachability for at most $k$ target sets, which costs $\mathcal{O}(km)$. Since $|T_1| \leq n$, the total cost of $\text{reach}_{\geq 1}(\cdot, \cdot)$ is given by

$\mathcal{O}(|T_1| \cdot km) = \mathcal{O}(k \cdot |T_1| \cdot m) = \mathcal{O}(knm)$.

After algorithm 7 terminates, $j$ gives the total number of iteration for $\text{reach}_{\geq 1}(\cdot, \cdot)$ and $\text{safe}(\cdot, \cdot)$, which is upper bounded by $j \leq |T_1| \cdot k \leq nk$.

The total cost of $\text{safe}(\cdot, \cdot)$ is linear, i.e., $\mathcal{O}\left(\sum_{i=1}^{j} \text{safe}(\cdot, \cdot)\right) = \mathcal{O}(m)$.

Therefore, the time complexity of the algorithm is given by $\mathcal{O}(k \cdot |T_1| \cdot m) = \mathcal{O}(knm)$.

**When does this worst case time complexity occur?**

In each iteration, the Büchi objective is not achieved for the last target set $T_k$ only, $|T_1|$ reduces by 1 each time, and $\mathcal{O}(|T_1|) = \mathcal{O}(n)$, which gives the worst case time complexity $\mathcal{O}(|T_1| \cdot km) = \mathcal{O}(k \cdot |T_1| \cdot m) = \mathcal{O}(knm)$. We can see that this worst case time complexity rarely happens in practice unless deliberately designed. Therefore, we can expect a linear time complexity $\mathcal{O}(km)$ in practice.

### Comparison among the Three Methods

In this subsubsection, we will give the comparison among the three methods.

From the complexity analysis of the three methods, we can see that method 1 performs better than method 2, and method 2 performs slightly better than method 3.

For the space complexity, method 1 uses one copy of arena $\mathcal{O}(m)$, while method 2 and 3 use $k$- and $(k+1)$- copies of arena, i.e., $\mathcal{O}(km)$ and $\mathcal{O}((k+1)m)$ respectively.

For the time complexity, method 1 and 2 use $\mathcal{O}(knm)$ time, while method 3 uses $\mathcal{O}((k+1)nm)$ time.

These information is shown in table 5.1.

| | Space Complexity | Time Complexity | Local Strategy |
|---|---|---|---|
| Method 1 | $\mathcal{O}(m)$ | $\mathcal{O}(knm)$ | $k$ |
| Method 2 | $\mathcal{O}(km)$ | $\mathcal{O}(knm)$ | 1 |
| Method 3 | $\mathcal{O}((k+1)m)$ | $\mathcal{O}((k+1)nm)$ | 1 |

Table 5.1: Comparison among Three Methods for $GB\mathcal{G}$

Now we give some more detailed analysis among the three methods.

Even though method 1 and method 2 have the same worst case time complexity $\mathcal{O}(knm)$, method 1 still outperforms method 2 in practice. This is because the key point to accelerate the fixed point iteration for Büchi is the linear safety process: in method 1, we can apply the safety algorithm once the reachability algorithm detects that a Büchi objective is not achieved; in method 2, however, we can apply the safety algorithm only after applying reachability algorithm in $k$ arenas for the $k$ target sets each time. In method 3, we need to apply reachability algorithm in $(k+1)$ arenas before applying safety algorithm each time, which is even worse.

In method 2 and 3, when we convert a generalized Büchi game to a Büchi game, first we convert the arena $\mathcal{A}$ to $\mathcal{A}'$ or $\mathcal{A}''$. The idea behind the two ways of converting the arena is the same as converting a generalized Büchi automaton to a Büchi automaton.

### Heuristic Optimization of the Time Complexity

Since the time complexity in method 1 and 2 is given by $\mathcal{O}(k \cdot |T_1| \cdot m) = \mathcal{O}(knm)$, and the order of the target sets is just a matter of permutation, we can always place the target set with the smallest size first. Therefore, the relatively optimal time complexity is given

by $\mathcal{O}(k \cdot \{min_{1 \leq i \leq k} |T_i|\} \cdot m) = \mathcal{O}(knm)$. A similar result also applies for method 3, i.e., $\mathcal{O}((k+1) \cdot \{min_{1 \leq i \leq k} |T_i|\} \cdot m) = \mathcal{O}((k+1)nm)$.

**Remark.** *In [8], the heuristic improvements end here. We continue to give two more steps of heuristic improvements below.*

From the analysis of Büchi and co-Büchi, we know that the multiple iterations of $\mathcal{O}(m)$ comes from the size of the target set in Büchi or the safe set in co-Büchi. Therefore, the time complexity for method 1 and 2 can be further reduced to

$\mathcal{O}(k \cdot min_{1 \leq i \leq k}(|T_i|, n - |T_i|) \cdot m) = \mathcal{O}(knm)$. A similar result also applies for method 3, i.e., $\mathcal{O}((k+1) \cdot min_{1 \leq i \leq k}(|T_i|, n - |T_i|) \cdot m) = \mathcal{O}((k+1)nm)$.

We can reduce the time complexity even further by replacing the target set with the currently smallest $min_{1 \leq i \leq k}(|T'_i|, n - |T'_i|)$ to the first place each time after detecting a Büchi objective that is not achieved. This is because the target set with the smallest $|T'_i|$ or $n - |T'_i|$ may change during the iterations. The upper bound is the same as the previous one, i.e., $\mathcal{O}(k \cdot min_{1 \leq i \leq k}(|T_i|, n - |T_i|) \cdot m) = \mathcal{O}(knm)$. However, this optimization can only apply to method 1 because it involves changing the target sets.

The effort above is trying to reduce the theoretical upper bound for the generalized Büchi game. In practice, however, we can expect an almost linear time complexity $\mathcal{O}(km)$, thus applying method 1 directly should be efficient enough. From the analysis, we can see that the reason for us to use the generalized Büchi algorithm whenever applicable is to improve the efficiency in both space and time.

**Disjunction of Büchi Objectives**

A generalized Büchi game with $k$ Büchi objectives can be considered as a conjunction of $k$ Büchi objectives. What about a disjunction of $k$ Büchi objectives?

For a conjunction of $k$ Büchi objectives, we need to satisfy all of the $k$ Büchi objectives $T_i$, for $1 \leq i \leq k$. In other words, we need to infinitely many often reach $T_i$ for all $i$, $1 \leq i \leq k$. The time complexity is $\mathcal{O}(knm)$.

For a disjunction of $k$ Büchi objectives, we only need to satisfy some of the $k$ Büchi objectives $T_i$, for $1 \leq i \leq k$. In other words, we only need to infinitely many often reach $T_i$ for some $i$, $1 \leq i \leq k$. Therefore, we can take the union of the $k$ target sets as one target set $\bigcup_{i=1}^{k} T_i$. The disjunction of $k$ Büchi objectives is thus equivalent to a Büchi objective with target set $\bigcup_{i=1}^{k} T_i$. The space and time complexity are $\mathcal{O}(m)$ and $\mathcal{O}(nm)$ respectively as in the analysis of a Büchi game.

### 5.4.2 Generalized co-Büchi Game

In this subsection, we will introduce the generalized co-Büchi game that requires eventually always staying in the safe set $S_i$ for some $i$, $1 \leq i \leq k$.

Similar to the generalized Büchi game, we can also convert the generalized co-Büchi game to a co-Büchi game using $k$- or $(k+1)$- copies of arena. The analysis and the results are similar to the generalized Büchi game as a merit of the duality.

Therefore, we will only present the method of using the generalized co-Büchi algorithm to solve the generalized co-Büchi game. We will present the generalized co-Büchi game in a dual way of the generalized Büchi game.

**Winning Condition of the Game**

In a generalized co-Büchi game $GC\mathcal{G} = (\mathcal{A}, \text{Obj})$, the objective set $\text{Obj} = \{S_1, \cdots, S_k\}$, where $S_i \subseteq V$, $1 \leq i \leq k$ are the $k$ sets of safe nodes and correspondingly, $\{V/S_1, \cdots, V/S_k\}$ are the $k$ sets of unsafe nodes. The winning condition for player 1 is to force the token to eventually always stay in $S_i$ for some $i$, $1 \leq i \leq k$, or equivalently, finitely many times reach $V/S_i$ for some $i$, $1 \leq i \leq k$. Formally, player 1 wins iff $\bigvee_{i=1}^{k} (\text{Inf}(\pi) \subseteq S_i)$, or equivalently, $\bigvee_{i=1}^{k} (\text{Inf}(\pi) \cap (V/S_i) = \emptyset)$.

In a generalized co-Büchi game with $k$ co-Büchi objectives, we only require satisfying some of the $k$ co-Büchi objectives. Therefore, a generalized $k$ co-Büchi objective is also considered as the disjunction of $k$ co-Büchi objectives as shown in the form of the winning condition.

**Idea of the Algorithm**

The basic idea of the generalized co-Büchi algorithm is to iteratively apply the co-Büchi algorithm until fixed point iteration.

Similar to the generalized Büchi algorithm, the order for us to apply the co-Büchi algorithm is to always focus on achieving the co-Büchi objective for $S_1$:

- step 1: when $S_1$ hasn't achieved co-Büchi objective, we apply the co-Büchi algorithm for $S_1$ to achieve co-Büchi objective for $S_1$;
- step 2: when $S_1$ has achieved co-Büchi objective, we apply the co-Büchi algorithm for $S_2, \cdots, S_k$ to see whether $S_2, \cdots, S_k$ achieve the co-Büchi objective;

if one of $S_2, \cdots, S_k$ doesn't achieve the co-Büchi objective, then we go back to step 1; else no more co-Büchi objective can be achieved for any $S_1, \cdots, S_k$, thus the generalized co-Büchi objective is achieved.

### Algorithm

Algorithm 8 is the algorithm we use to solve for the generalized co-Büchi game. It's a fixed point iteration of the $k$ co-Büchi objectives.

**Algorithm 8:** $\mathcal{A}'(\mathcal{W}_1) = \text{Generalized co-Büchi}(\mathcal{A}, S_1, \cdots, S_k)$

**Input:**
1. $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$;
2. $S_i \subseteq V$, $1 \leq i \leq k$;

**Output:**
1. $\mathcal{A}'(\mathcal{W}_1)$;

```
for (V' = V, S'_1 = S_1, ..., S'_k = S_k, i = 1, j = 0, W_1 = ∅; i ≤ k; j++) do
    S'_i = S'_i ∩ V';
    A'(S) = safe(A'(V'), S_i);
    if S == ∅ then i++; continue;
    A'(R) = reach_{≥0}(A'(V'), S);
    W_1 += R;
    if R == V' then break;
    V' -= R;
    i = 1;
W_2 = V/W_1;
```

**Result:**
1. Winning set $\mathcal{W}_1 \oplus \mathcal{W}_2 = V$;
2. Winning strategy of each node in $V_1 \cap \mathcal{W}_1$ is computed by $\text{safe}(\cdot, \cdot)$ and $\text{reach}_{\geq 1}(\cdot, \cdot)$ in the corresponding process;

### Proof of Soundness and Completeness

#### Soundness

The generalized co-Büchi algorithm terminates when no more safe sets in the $k$ safe sets satisfy the co-Büchi objectives. Therefore, the nodes in $\mathcal{W}_1$ can eventually always stay in $S_i$ for some $1 \leq i \leq k$ by following the local strategy constructed through the iteration. This gives the proof of soundness.

#### Completeness

After the generalized co-Büchi algorithm 8 terminates, the remaining nodes in the arena $V'$ are the nodes in $\mathcal{W}_2$. The nodes in $\mathcal{W}_2$ can't be attracted to $\mathcal{W}_1$. We apply the generalized co-Büchi algorithm again in $\mathcal{W}_2$ to analyze the behavior of the nodes in $\mathcal{W}_2$.

After the safety process $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V'), S_i)$, $S$ must be empty for all $i$, $1 \leq i \leq k$. Therefore, the nodes in $\mathcal{W}_2$ can't eventually always stay in $S_i$ for any $i$, $1 \leq i \leq k$. In other words, for any $v \in \mathcal{W}_2$, dually, it must satisfy the generalized Büchi objective of infinitely many often reach the unsafe set $V/S_i$ for all $i$, $1 \leq i \leq k$. Therefore, $v$ can't eventually always stay in $S_i$ for some $i$, $1 \leq i \leq k$. This gives the proof of completeness.

**Strategy**

The strategy for the generalized co-Büchi algorithm is computed by $\text{safe}(\cdot,\cdot)$ and $\text{reach}_{\geq 1}(\cdot,\cdot)$ in the corresponding process. Each strategy is computed once since the current sub-arena $\mathcal{A}'$ is monotonically decreasing and $\mathcal{W}_1$ is monotonically increasing through the process of iterating the co-Büchi objectives.

The reason of the strategy for the generalized co-Büchi game is similar to that for the co-Büchi game, which guarantees that play $\pi$ to eventually always stay in $S_i$ for some $1 \leq i \leq k$.

The difference is that, we don't know which $S_i$ the token will eventually always stay in, unless it is the initial real safe set that activates the iteration.

We also can't determine when that "eventually" is to satisfy eventually always stay in $S_i$ for some $1 \leq i \leq k$ because it may depend on player 2 to choose whether to stay in $S_i$ or leave $S_i$. However, once left $S_i$, the token is being forced to reach an $S_j$, $1 \leq j \leq k$. The total number of times and steps that the token can leave $S_i$ for all $i$, $1 \leq i \leq k$ is bounded by $\left|\mathcal{W}_1/(\bigcup_{i=i}^{k} S_i)\right|$, which is finite. This allows us to claim that our strategy satisfies eventually always staying in $S_i$ for some $i$, $1 \leq i \leq k$. If player 2 violates staying in $S_i$ for some $i$, $1 \leq i \leq k$, then we can argue that it's not the time for "eventually" yet, and player 2 can violate staying in $S_i$ for some $i$, $1 \leq i \leq k$ for at most $\left|\mathcal{W}_1/(\bigcup_{i=i}^{k} S_i)\right|$ steps, so that the argument still holds. This shows that such strategy is indeed a winning strategy for player 1.

The local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$ since even though we have $k$ co-Büchi objectives, we only need to satisfy one of them to satisfy the generalized co-Büchi condition. Therefore, we only need 1 unit for the local strategy to focus on 1 co-Büchi objective for each node $v \in V_1 \cap \mathcal{W}_1$.

**Complexity Analysis**

In a generalized co-Büchi game $GC\mathcal{G} = (\mathcal{A}, \text{Obj})$, $\mathcal{A} = (V, E)$ is the arena and

Obj $= \{S_1, \cdots, S_k\}$ is the objective set, where $S_i \subseteq V$, $1 \leq i \leq k$ are the $k$ sets of safe nodes and correspondingly, $\{V/S_1, \cdots, V/S_k\}$ are the $k$ sets of unsafe nodes. $|V| = n$ is the number of nodes and $|E| = m$ is the number of edges.

**Proposition 5.4.4.** *A generalized $k$ co-Büchi game can be solved using generalized co-Büchi algorithm in $\mathcal{O}(m)$ space, $\mathcal{O}(knm)$ time, and the local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Space Complexity:** $\mathcal{O}(n + m)$

Since the size of the arena is depicted by $n$ nodes and $m$ edges, the space complexity of the algorithm is $\mathcal{O}(n + m)$.

**Time Complexity:** $\mathcal{O}(knm)$

Similar to the generalized Büchi game, we always focus on achieving the co-Büchi objective for $S_1$.

Whenever doing a safety for $S_i$ is not empty for some $i$, $1 \leq i \leq k$, the size of $|S_i|$ reduces by at least one. The consequent reachability process may reduce $|S_i|$ further more.

In algorithm 8, doing safety for each safe set $S_i$, i.e., safe$(\cdot, \cdot)$, costs $\mathcal{O}(m)$, where $1 \leq i \leq k$. $|S_i|$ reduces by at least one after doing safety for at most $k$ safe sets, which costs $\mathcal{O}(km)$. Since $\left|\bigcup_{i=1}^{k} S_i\right| \leq |V| = n$, the total cost of safe$(\cdot, \cdot)$ is given by

$$\mathcal{O}\left(\left|\bigcup_{i=1}^{k} S_i\right| \cdot km\right) = \mathcal{O}\left(k \cdot \left|\bigcup_{i=1}^{k} S_i\right| \cdot m\right) = \mathcal{O}(knm).$$

After algorithm 8 terminates, $j$ gives the total number of iteration for safe$(\cdot, \cdot)$ and reach$_{\geq 0}(\cdot, \cdot)$, which is upper bounded by $j \leq \left|\bigcup_{i=1}^{k} S_i\right| \cdot k \leq nk$.

The total cost of reach$_{\geq 0}(\cdot, \cdot)$ is linear, i.e., $\mathcal{O}\left(\sum_{i=1}^{j} \text{reach}_{\geq 0}(\cdot, \cdot)\right) = \mathcal{O}(m)$.

Therefore, the time complexity of the algorithm is given by

$$\mathcal{O}\left(k \cdot \left|\bigcup_{i=1}^{k} S_i\right| \cdot m\right) = \mathcal{O}(knm).$$

**When does this worst case time complexity occur?**

In each iteration, doing a safety is not empty for the last safe set $S_k$ only, $|S_k|$ reduces by 1 each time. When $S_k$ is empty, doing a safety is not empty for the second last safe set $S_{k-1}$ only, etc. With $\mathcal{O}\left(\left|\bigcup_{i=1}^{k} S_i\right|\right) = \mathcal{O}(n)$, the worst case time complexity is given by $\mathcal{O}\left(\left|\bigcup_{i=1}^{k} S_i\right| \cdot km\right) = \mathcal{O}\left(k \cdot \left|\bigcup_{i=1}^{k} S_i\right| \cdot m\right) = \mathcal{O}(knm)$. We can see that this worst case time complexity rarely happens in practice unless deliberately designed. Therefore, we can expect a linear time complexity $\mathcal{O}(km)$ in practice.

**More Detailed Analysis**

Note that we can't deduct an upper bound as tight as $\mathcal{O}(k \cdot |S_1| \cdot m) = \mathcal{O}(knm)$ directly since generalized Büchi algorithm enjoys a conjunction property that the reduction of any $|T_i|$ implies the reduction of $|T_1|$ for $1 \leq i \leq k$, whereas generalized co-Büchi algorithm only has a disjunction property that doesn't satisfy a corresponding implication.

However, with the merit of the duality, if we consider the generalized co-Büchi game as a generalized Büchi game from the perspective of player 2, then we obtain the conjunction property that the reduction of any $|V/S_i|$ implies the reduction of $|V/S_1|$ for $1 \leq i \leq k$. This gives $\mathcal{O}(k \cdot |V/S_1| \cdot m) = \mathcal{O}(knm)$, where $V/S_i$ is considered as $T_i$ in a generalized Büchi game for player 2.

Other heuristic optimizations in the generalized Büchi game also applies for the generalized co-Büchi game here. The time complexity can be reduced to $\mathcal{O}(k \cdot min_{1 \leq i \leq k}(|S_i|, n - |S_i|) \cdot m) = \mathcal{O}(knm)$. We can reduce the time complexity even further by replacing the safe set with the currently smallest $min_{1 \leq i \leq k}(|S'_i|, n - |S'_i|)$ to the first place each time after detecting a new co-Büchi objective.

From the duality, we know that the analysis and results for the generalized co-Büchi should be exactly the same as that for the generalized Büchi since they are like the two sides of one coin.

In practice, we can also expect an almost linear time complexity $\mathcal{O}(km)$, thus applying algorithm 8 directly should be efficient enough.

**Conjunction of co-Büchi Objectives**

This is the dual problem of the disjunction of Büchi objectives.

A generalized co-Büchi game with $k$ co-Büchi objectives can be considered as a disjunction of $k$ co-Büchi objectives. What about a conjunction of $k$ co-Büchi objectives?

For a disjunction of $k$ co-Büchi objectives, we only need to satisfy some of the $k$ co-Büchi objectives $S_i$, for $1 \leq i \leq k$. In other words, we only need to eventually always stay in $S_i$ for some $i$, $1 \leq i \leq k$. The time complexity is $\mathcal{O}(knm)$.

For a conjunction of $k$ co-Büchi objectives, we need to satisfy all of the $k$ co-Büchi objectives $S_i$, for $1 \leq i \leq k$. In other words, we need to eventually always stay in $S_i$ for all $i$, $1 \leq i \leq k$. Therefore, we can take the intersection of the $k$ safe sets as one safe set $\bigcap_{i=1}^{k} S_i$. The conjunction of $k$ co-Büchi objectives is thus equivalent to a co-Büchi objective with safe set $\bigcap_{i=1}^{k} S_i$. The space and time complexity are $\mathcal{O}(m)$ and $\mathcal{O}(nm)$ respectively as in the analysis of a co-Büchi game.

## 5.5 One Pair Rabin and Streett

In this section, we will introduce the one pair Rabin game and the one pair Streett game.

This is a warm up for the more general Rabin and Street games with $k$ pairs. We choose to extract the one pair Rabin and Streett games out as an intermediate step because the 1 pair Rabin and Streett games are already quite complicated; the $k$ pairs Rabin and Streett games are even more comprehensive, which involve analysis of inner loops and outer loops. Therefore, we will discuss the core inner loops of the algorithms in this section for one pair Rabin and Streett games, and leave the discussion of the additional outer loops of the algorithms in the next section for $k$ pairs Rabin and Streett games based on the discussion in this section. Also, the worst case scenario is also different in Rabin and Streett games for one pair and $k$ pairs.

The algorithms used in the one pair Rabin and Streett games are constructed by combining the algorithms for Büchi and co-Büchi together.

### 5.5.1 One Pair Rabin Game

In this subsection, we will introduce the one pair Rabin game that requires reaching the "Good" set $G$ infinitely many often times and the "Bad" set $B$ finitely many often times.

**Winning Condition of the Game**

In a one pair Rabin game $R\mathcal{G}$, Obj $= \{(G, B)\}$, where $G, B \subseteq V$. "G" refers to the set of "Good" nodes that we would like to reach infinitely many often times, whereas "B" refers to the set of "Bad" nodes that we would like to reach only finitely many often times.

The winning condition for player 1 is to force the token to reach $G$ infinitely many often and reach $B$ only finitely many often. Formally, player 1 wins iff

$(\text{Inf}(\pi) \cap G \neq \emptyset) \wedge (\text{Inf}(\pi) \cap B = \emptyset)$.

**Partition of the Arena**

We assume that the "Good" set $G$ and the "Bad" set $B$ are disjoint, i.e., $G \cap B = \emptyset$. Otherwise the Rabin pair $(G, B)$ is equivalent to $(G/B, B)$. The reason is that, if the intersection of $G$ and $B$ is not empty, i.e., $G \cap B \neq \emptyset$, then in order to satisfy the one pair Rabin condition that reaching some nodes in $G$ infinitely many often and reaching all nodes in $B$ finitely many often, $(G \cap B) \subseteq B$ can't be reached for infinitely many often times, and thus can be reached for at most finitely many often times. Therefore, $G \cup B$ can be partitioned as $G/B$ and $B$, i.e., $G \cup B = (G/B) \oplus B$.

With $M = V/(G \cup B)$, we can partition $V$ as $V = G \oplus M \oplus B$, where "M" refers to "Middle" or "Mutual".

If $M = \emptyset$, then $V = G \oplus B$. The winning condition can be simplified as

$(\text{Inf}(\pi) \cap G \neq \emptyset) \wedge (\text{Inf}(\pi) \cap B = \emptyset) \Leftrightarrow (\text{Inf}(\pi) \cap G \neq \emptyset) \wedge (\text{Inf}(\pi) \subseteq G) \Leftrightarrow \text{Inf}(\pi) \subseteq G$, which is a co-Büchi condition.

**Idea of the Algorithm**

The basic idea of the one pair Rabin algorithm is to iteratively apply a Büchi algorithm within a co-Büchi algorithm until fixed point iteration. It can be seen as a generalization of the co-Büchi algorithm.

As always, the algorithm is constructed backwards according to the winning condition of the one pair Rabin game. $\mathcal{W}_1$ is initially empty.

We would like the token to reach some nodes in $G$ infinitely many often and reach all nodes in $B$ finitely many often. Therefore, the token must eventually always stay in $V/B$, i.e., in $G \oplus M$. This is done by considering $G \oplus M$ as the safe set and do a safety process,

i.e., $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V'), G' + M')$. Since we also need to infinitely many often reach $G$, we consider $G' \cap S$ as the target set and solve a Büchi game within $S$ to get $X$, i.e., $X = \text{Büchi}(\mathcal{A}'(S), G' \cap S)$. The nodes in $X$ satisfies always staying in $V/B$ and infinitely many often reaching $G$, thus satisfies the winning condition of the game. We consider $X$ as our base set in the winning set to activate our iteration and apply the reachability algorithm to find the reachable set of $X$, i.e., $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V'), X)$. The nodes in $R$ satisfies eventually always staying in $V/B$ and infinitely many often reaching $G$, thus satisfies the winning condition of the game. We add $R$ to the winning set $\mathcal{W}_1$ and reduce the current set of nodes in the arena $V'$ by $R$. This finishes our one iteration. We repeat this process until $\mathcal{W}_1$ doesn't increase any more. We may need to apply this iteration multiple times because after doing the reachability from $X$ to $R$, we may induce a new safe set in the remaining $V'/R$, which activates our next iteration.

We know that the co-Büchi algorithm is a fixed point iteration of safety and reachability. Here in the one pair Rabin algorithm, we insert a Büchi between safety and reachability in each iteration. Therefore, the one pair Rabin algorithm is a fixed point iteration of Büchi within co-Büchi.

**Algorithm**

Algorithm 9 is the algorithm we use to solve for the one pair Rabin game. It's a fixed

point iteration of Büchi within co-Büchi.

**Algorithm 9:** $\mathcal{A}'(\mathcal{W}_1) = \text{One Pair Rabin}(\mathcal{A}, G, B)$

**Input:**
1. $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$;
2. $G, B \subseteq V$;

**Output:**
1. $\mathcal{A}'(\mathcal{W}_1)$;

$G' = G = G/B, M' = M = V - G - B, B' = B$;
**for** $(V' = V, j = 0, \mathcal{W}_1 = \emptyset; ; j++)$ **do**
  $G' = G' \cap V', M' = M' \cap V', B' = B' \cap V'$;
  $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V'), G' + M')$;
  **if** $S == \emptyset$ **then** break;
  $X = \text{Büchi}(\mathcal{A}'(S), G' \cap S)$;
  **if** $X == \emptyset$ **then** break;
  $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V'), X)$;
  $\mathcal{W}_1 += R$;
  $V' -= R$;
$\mathcal{W}_2 = V/\mathcal{W}_1$;

**Result:**
1. Winning set $\mathcal{W}_1 \oplus \mathcal{W}_2 = V$;
2. Winning strategy of each node in $V_1 \cap \mathcal{W}_1$ is computed by Büchi$(\cdot, \cdot)$ and reach$_{\geq 0}(\cdot, \cdot)$ in the corresponding process;

### Proof of Soundness and Completeness

#### Soundness

The one pair Rabin algorithm 9 is a generalization of the co-Büchi algorithm 6, thus it shares a similar reasoning as the co-Büchi algorithm.

The one pair Rabin algorithm terminates when $\mathcal{W}_1$ doesn't increase any more. In each iteration, the nodes in $X$ satisfies always staying in $V'/B'$ and infinitely many often reaching $G'$ in the current $V'$, thus satisfies the winning condition of the game. The nodes in $R$ satisfies eventually always staying in $V'/B'$ and infinitely many often reaching $G'$ in the current $V'$, thus satisfies the wining condition of the game.

For a token in $\mathcal{W}_1$, if it doesn't stay in $X$ of a certain iteration, then it must go through an $R$ to an $X$ of a previous iteration. Since the $X$ in the first iteration satisfies always staying in $V/B$ and infinitely many often reaching $G$, the nodes in $\mathcal{W}_1$ satisfies eventually

always staying in $V/B$ and infinitely many often reaching $G$, which is the winning condition of the game. This gives the proof of soundness.

**Completeness**

After the one pair Rabin algorithm 9 terminates, the remaining nodes in the arena $V'$ are the nodes in $\mathcal{W}_2$. The nodes in $\mathcal{W}_2$ can't be attracted to $\mathcal{W}_1$. We apply one more iteration of the one pair rabin algorithm to analyze the behavior of the nodes in $\mathcal{W}_2$.

After the safety process $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V'), G' + M')$:

- if $S$ is empty, then the nodes in $\mathcal{W}_2$ can be forced to reach $B$ infinitely many often, which violates the winning condition of the game;
- otherwise $S$ is not empty:
  - the nodes in $\mathcal{W}_2/S$ can be forced to reach $B$;
  - then we apply the Büchi algorithm in $S$, i.e., $X = \text{Büchi}(\mathcal{A}'(S), G' \cap S)$, and $X$ must be empty since the fixed point iteration has been achieved in the previous iteration, thus the nodes in $S$ can be forced to reach $G$ only finitely many often times, which violates the winning condition of the game;
  - if we would like to reach $G$ infinitely many often times, we may need to reach $\mathcal{W}_2/S$ first, which also leads to reaching $B$ infinitely many often times, which violates the winning condition of the game.

Therefore, the nodes in $\mathcal{W}_2$ can't satisfy the winning condition of the game. This gives the proof of completeness.

**Strategy**

The strategy for the one pair Rabin algorithm is computed by $\text{Büchi}(\cdot,\cdot)$ and $\text{reach}_{\geq 0}(\cdot,\cdot)$ in the corresponding process. Each strategy is computed once since the current sub-arena $\mathcal{A}'$ is monotonically decreasing and $\mathcal{W}_1$ is monotonically increasing through the process of the iterations.

Suppose a token is initially placed on a node in $\mathcal{W}_1$. If the token is on a node in $B$, then the token will be forced to $V/B$ by the strategy computed in $\text{reach}_{\geq 0}(\cdot,\cdot)$. If the token is on a node in $V/B$, then the token will be forced to reach $G$ infinitely many often times by the strategy computed in $\text{Büchi}(\cdot,\cdot)$. The token can reach $B$ for at most finitely $|\mathcal{W}_1 \cap B|$ steps, and can inductively reach $G$ infinitely many times. This shows that such strategy is indeed a winning strategy for player 1. The local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.

**Complexity Analysis**

In a one pair Rabin game $R\mathcal{G}$, $\mathcal{A} = (V, E)$ is the arena and Obj $= \{(G, B)\}$ is the objective set, where $G, B \subseteq V$. $|V| = n$ is the number of nodes and $|E| = m$ is the number of edges.

**Proposition 5.5.1.** *A one pair Rabin game can be solved in $\mathcal{O}(m)$ space, $\mathcal{O}(n^2m)$ time, and the local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Space Complexity:** $\mathcal{O}(n + m)$

Since the size of the arena is depicted by $n$ nodes and $m$ edges, the space complexity of the algorithm is $\mathcal{O}(n + m)$.

**Time Complexity:** $\mathcal{O}(n^2m)$

Similar to the analysis for co-Büchi, in each iteration, the safety process costs $\mathcal{O}(m)$, the Büchi process costs $\mathcal{O}(nm)$, and $|G|$ and $|B|$ reduces by at least one.

After algorithm 9 terminates, $j$ gives the total number of iterations for safe$(\cdot,\cdot)$, Büchi$(\cdot,\cdot)$ and reach$_{\geq 0}(\cdot,\cdot)$, which is upper bounded by $j \leq |G \oplus M| \leq |V| = n$. The total cost of safe$(\cdot,\cdot)$ is given by $\mathcal{O}(n \cdot m) = \mathcal{O}(nm)$.

The total cost of Büchi$(\cdot,\cdot)$ is given by $\mathcal{O}(n \cdot nm) = \mathcal{O}(n^2m)$.

The total cost of reach$_{\geq 0}(\cdot,\cdot)$ is linear, i.e., $\mathcal{O}\left(\sum_{i=1}^{j} \text{reach}_{\geq 0}(\cdot,\cdot)\right) = \mathcal{O}(m)$.

Therefore, the time complexity of the algorithm is given by

$\mathcal{O}(nm + n^2m + m) = \mathcal{O}(n^2m)$.

**When does this worst case time complexity occur?**

We set $|G| \approx |B| \approx \frac{1}{4}n$, and $|M| \approx \frac{1}{2}n$. Since the Büchi process is the dominating term, we would like to maximize the Büchi process.

We design our one pair Rabin algorithm to take $j \approx \frac{1}{4}n$ iterations.

The picture is shown in Figure 5.5.

In the first iteration, we let half of $M$ be deleted in the safety process $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V'), G' + M')$. Since $\frac{1}{2}|M| \approx \frac{1}{2} \cdot \frac{1}{2}n = \frac{1}{4}n$ nodes participate in the safety process, the cost for the first safety process is $\mathcal{O}(\frac{1}{4}m)$.

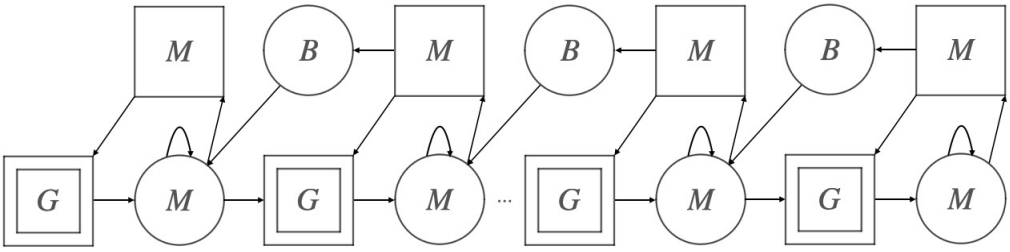


Figure 5.5: Rabin Example

One node in this half of $M$ will be added to the safe set $S$ in each of the $\frac{1}{4}n$ iterations.

In the first iteration of the Büchi process $X = \text{Büchi}(\mathcal{A}'(S), G' \cap S)$,

$|G| + \frac{1}{2}|M| \approx \frac{1}{4}n + \frac{1}{2} \cdot \frac{1}{2}n = \frac{1}{2}n$ nodes participate in the iteration for Büchi, with $G$ being the target nodes. Therefore, the cost for the first Büchi process is

$\mathcal{O}(|G| \cdot \frac{1}{2}m) = \mathcal{O}(\frac{1}{4}n \cdot \frac{1}{2}m) = \mathcal{O}(\frac{1}{8}nm)$. $X$ consists of one node in $G$ and two nodes in $M$.

In the first iteration of the reachability process $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V'), X)$, we attract one node in $B$, thus the cost for the first reachability process is $\mathcal{O}(1)$.

The number of nodes participate in the safety process reduces by one in each iteration, thus the total cost for safety is $\mathcal{O}(\frac{1}{4}m \cdot \frac{1}{4}n \cdot \frac{1}{2}) = \mathcal{O}(\frac{1}{32}nm)$.

The number of nodes participate in the Büchi process reduces by two in each iteration, thus the total cost for Büchi is $\mathcal{O}(\frac{1}{8}nm \cdot \frac{1}{4}n \cdot \frac{1}{2}) = \mathcal{O}(\frac{1}{64}n^2m)$. $X$ consists of one node in $G$ and two nodes in $M$ in each iteration.

One node in $B$ is attracted to $\mathcal{W}_1$ in each reachability process, thus the total cost for reachability is $\mathcal{O}(\frac{1}{4}m)$.

Therefore, the worst case time complexity is given by

$\mathcal{O}(\frac{1}{32}nm + \frac{1}{64}n^2m + \frac{1}{4}m) = \mathcal{O}(\frac{1}{64}n^2m) = \mathcal{O}(n^2m)$.

We can see that this worst case time complexity rarely happens in practice unless deliberately designed, and the coefficient is quite small. Therefore, we can expect a linear time complexity $\mathcal{O}(m)$ in practice.

**Common Mistake**

It seems like the winning condition of a one pair Rabin game is in the form of a conjunction of Büchi and co-Büchi objectives. However, if we analogize it to a generalized Büchi algorithm by taking conjunction of Büchi and co-Büchi until fixed point iteration, there will be a problem. Such algorithm is complete but not sound. The problem is that, the winning set may satisfy the Büchi and the co-Büchi objectives independently. However, there may be a conflict in the play if we consider the Büchi objective and the co-Büchi objective together.

A counter-example is shown in Figure 5.6. We may satisfy the Büchi condition of infinitely many often reaching $G$ by playing $\pi_1 = (GBM)^\omega$, and satisfy the co-Büchi condition of finitely many often reaching $B$ by playing $\pi_2 = M^\omega$. However, no play can satisfy the Büchi and the co-Büchi conditions together. $\pi_1$ violates finitely many often reaching $B$, whereas $\pi_2$ violates infinitely many often reaching $G$. The reason is that there is a conflict of coordinating these two conditions together in the play. The proposed one pair Rabin algorithm 9 can resolve this.

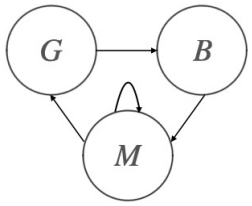


Figure 5.6: Rabin Counter-Example

### 5.5.2 One Pair Streett Game

In this subsection, we will introduce the one pair Streett game that requires reaching the "Good" set $G$ infinitely many often times or the "Bad" set $B$ finitely many often times.

We will present the one pair Streett game in a dual way of the one pair Rabin game.

**Winning Condition of the Game**

In a one pair Streett game $S\mathcal{G}$, Obj $= \{(G, B)\}$, where $G, B \subseteq V$. "G" refers to the set of "Good" nodes that we would like to reach infinitely many often times, whereas "B" refers to the set of "Bad" nodes that we would like to reach only finitely many often times.

The winning condition for player 1 is to force the token to reach $G$ infinitely many often or reach $B$ only finitely many often. Formally, player 1 wins iff

$(\text{Inf}(\pi) \cap G \neq \emptyset) \vee (\text{Inf}(\pi) \cap B = \emptyset)$.

**Partition of the Arena**

We can't assume that the "Good" set $G$ and the "Bad" set $B$ are disjoint, i.e., $G \cap B = \emptyset$. The reason is that, if the intersection of $G$ and $B$ is not empty, i.e., $G \cap B \neq \emptyset$, then in order to satisfy the one pair Streett condition that reaching some nodes in $G$ infinitely many often or reaching all nodes in $B$ finitely many often, $G \cap B$ can be reached for either infinitely many often times or only finitely many often times: reaching $(G \cap B) \subseteq G$ infinitely many often times can satisfy the winning condition according to reaching some nodes in $G$ infinitely many often times, whereas reaching $(G \cap B) \subseteq B$ only finitely many often times can satisfy the winning condition if all nodes in $B$ are reached for only finitely many often times according to reaching all nodes in $B$ finitely many often times. Therefore, we can not partition $G \cup B$.

However, with $M = V/(G \cup B)$, we can partition $V$ as $V = (G \cup B) \oplus M$, where "M" refers to "Middle" or "Mutual".

If $M = \emptyset$ and $G \cap B = \emptyset$, then $V = G \oplus B$. The winning condition can be simplified as $(\text{Inf}(\pi) \cap G \neq \emptyset) \vee (\text{Inf}(\pi) \cap B = \emptyset) \Leftrightarrow (\text{Inf}(\pi) \cap G \neq \emptyset) \vee (\text{Inf}(\pi) \subseteq G) \Leftrightarrow \text{Inf}(\pi) \cap G \neq \emptyset$, which is a Büchi condition.

**Idea of the Algorithm**

There is no neat way to present the one pair Streett algorithm from the perspective of player 1, and the idea for the one pair Streett algorithm is basically the same as the one pair Rabin algorithm. Therefore, we won't explicitly present a one pair Streett algorithm. Instead, we will give a conversion to solve it as a one pair Rabin game using algorithm 9 from the perspective of player 2. The conversion is straightforward: switching the roles of $V_1$ and $V_2$, considering the good nodes "G" in $S\mathcal{G}$ as bad nodes "$\bar{B}$" in $R\mathcal{G}$, and considering the bad nodes "B" in $S\mathcal{G}$ as good nodes "$\bar{G}$" in $R\mathcal{G}$.

**Proof of Soundness and Completeness**

The reasoning for the proof of soundness and completeness for the one pair Streett game is the same as the proof of completeness and soundness for the one pair Rabin game.

**Strategy**

The strategy for the one pair Streett algorithm is computed by safe$(\cdot,\cdot)$ and Büchi$(\cdot,\cdot)$ in the corresponding process of the one pair Rabin algorithm 9. From the perspective of player 1, these two processes are reach$_{\geq 0}(\cdot,\cdot)$ and co-Büchi$(\cdot,\cdot)$ respectively. The reasoning is given in the proof of completeness for the one pair Rabin game. The local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.

### Complexity Analysis

In a one pair Streett game $S\mathcal{G}$, $\mathcal{A} = (V, E)$ is the arena and Obj $= \{(G, B)\}$ is the objective set, where $G, B \subseteq V$. $|V| = n$ is the number of nodes and $|E| = m$ is the number of edges.

**Proposition 5.5.2.** *A one pair Streett game can be solved in $\mathcal{O}(m)$ space, $\mathcal{O}(n^2m)$ time, and the local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Space Complexity:** $\mathcal{O}(n + m)$

Since the size of the arena is depicted by $n$ nodes and $m$ edges, the space complexity of the algorithm is $\mathcal{O}(n + m)$.

**Time Complexity:** $\mathcal{O}(n^2m)$

We convert the one pair Street game as a one pair Rabin game for player 2. Therefore, it has the same time complexity as the one pair Rabin game which is $\mathcal{O}(n^2m)$.

### When does this worst case time complexity occur?

We can refer the worst case scenario to that in the one pair Rabin game since from duality they are the same problem. This worst case time complexity rarely happens in practice unless deliberately designed, and the coefficient is quite small. Therefore, we can expect a linear time complexity $\mathcal{O}(m)$ in practice.

### Common Mistake

It seems like the winning condition of a one pair Streett game is in the form of a disjunction of Büchi and co-Büchi objectives. However, if we analogize it to a generalized co-Büchi algorithm by taking disjunction of Büchi and co-Büchi until fixed point iteration, there will be a problem. Such algorithm is sound but not complete. The problem is that, we may miss the winning set that does not satisfy the Büchi or the co-Büchi objective independently. However, there may be a conflict in the play if we consider the Büchi objective and the co-Büchi objective together, which turns out satisfying the winning condition.

A counter-example is shown in Figure 5.7. We may violate the Büchi condition of infinitely many often reaching $G$ by playing $\pi_1 = M^\omega$, and violate the co-Büchi condition of finitely many often reaching $B$ by playing $\pi_2 = (BGM)^\omega$. However, no play can violate the Büchi or the co-Büchi conditions together. $\pi_1$ satisfies finitely many often reaching $B$, whereas $\pi_2$ satisfies infinitely many often reaching $G$. The reason is that there is a conflict of violating these two conditions together in the play. The proposed conversion from the one pair Streett game to the one pair Rabin game using algorithm 9 can resolve this.

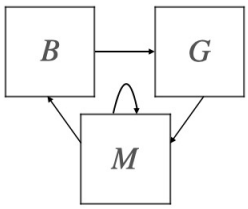


Figure 5.7: Streett Counter-Example

## 5.6 Rabin and Streett

In this section, we will only introduce the $k$ pairs Rabin game but not the $k$ pairs Streett game. The reason is that we will only present a sound but not complete algorithm for the $k$ pairs Rabin game, which turns out to be a complete but not sound algorithm for the $k$ pairs Streett game from duality.

A sound and complete algorithm for the $k$ pairs Rabin game has much higher time complexity than a sound but not complete one, which involves factorial w.r.t. the number of Rabin pairs $k$, i.e., $k!$ [16], [28]. Also, it involves recursion into different sub-problems which is hard in implementation and has high time complexity.

The $k$ pairs Rabin game has a local strategy, whereas the $k$ pairs Streett game may not have a local strategy for each of the $k$ Streett objectives, which may involve a complicated design that requires much memory.

The reason why Rabin and Streett games are much more tricky than the generalized Büchi game or the generalized co-Büchi game is that, they involve two different objectives, i.e., Büchi and co-Büchi. It is possible to have conflicts when we want to satisfy these two objectives together in one play if we are not careful enough. In a $k$ pairs Rabin game, it is also likely that we can not satisfy any of the $k$ Rabin conditions independently, but the play satisfies the winning condition when we consider all the $k$ Rabin conditions together. We will give a counter-example to show why our algorithm is not complete.

### 5.6.1 Rabin Game

In this subsection, we will introduce the $k$ pairs Rabin game that requires reaching the "Good" sets $G_i$ infinitely many often times and the "Bad" sets $B_i$ finitely many often times for some $i$, $1 \leq i \leq k$.

**Winning Condition of the Game**

In a $k$ pairs Rabin game $R\mathcal{G}$, Obj $= \{(G_1, B_1), \cdots, (G_k, B_k)\}$, where $G_i, B_i \subseteq V$, $1 \leq i \leq k$. "G" refers to the set of "Good" nodes that we would like to reach infinitely many often times, whereas "B" refers to the set of "Bad" nodes that we would like to reach only finitely many often times.

The winning condition for player 1 is to force the token to reach $G_i$ infinitely many often and reach $B_i$ only finitely many often for some $i$, $1 \leq i \leq k$. Formally, player 1 wins iff $\bigvee_{i=1}^{k} ((\text{Inf}(\pi) \cap G_i \neq \emptyset) \wedge (\text{Inf}(\pi) \cap B_i = \emptyset))$.

**Partition of the Arena**

The partition from the 1 pair Rabin can be generalized to the $k$ pairs Rabin. For $1 \leq i \leq k$, $G_i \cup B_i$ can be partitioned as $G_i/B_i$ and $B_i$, i.e., $G_i \cup B_i = (G_i/B_i) \oplus B_i$. With $M_i = V/(G_i \cup B_i)$, we can partition $V$ as $V = G_i \oplus M_i \oplus B_i$, where "M" refers to "Middle" or "Mutual". If $M_i = \emptyset$ for $1 \leq i \leq k$, then $V = G_i \oplus B_i$. The winning condition can be simplified as

$$\begin{aligned}\bigvee_{i=1}^{k} ((\text{Inf}(\pi) \cap G_i \neq \emptyset) \wedge (\text{Inf}(\pi) \cap B_i = \emptyset)) &\Leftrightarrow \bigvee_{i=1}^{k} ((\text{Inf}(\pi) \cap G_i \neq \emptyset) \wedge (\text{Inf}(\pi) \subseteq G_i)) \\ &\Leftrightarrow \bigvee_{i=1}^{k} ((\text{Inf}(\pi) \subseteq G_i)),\end{aligned}$$

which is a generalized co-Büchi condition.

**Idea of the Algorithm**

We will present a sound but not complete algorithm for the $k$ pairs Rabin algorithm. The basic idea of the $k$ pairs Rabin algorithm is to iteratively apply the one pair Rabin algorithm until fixed point iteration. It can be seen as a generalization of the generalized co-Büchi algorithm.

Similar to the generalized co-Büchi algorithm, the order for us to apply the one pair Rabin algorithm is to always focus on achieving the Rabin objective for $(G_1, B_1)$:

- step 1: when $(G_1, B_1)$ hasn't achieved Rabin objective, we apply the one pair Rabin algorithm for $(G_1, B_1)$ to achieve Rabin objective for $(G_1, B_1)$;
- step 2: when $(G_1, B_1)$ has achieved Rabin objective, we apply the one pair Rabin algorithm for $(G_2, B_2), \cdots, (G_k, B_k)$ to see whether $(G_2, B_2), \cdots, (G_k, B_k)$ achieve the Rabin objective;

if one of $(G_2, B_2), \cdots, (G_k, B_k)$ doesn't achieve the Rabin objective, then we go back to step 1; else no more Rabin objective can be achieved for any $(G_1, B_1), \cdots, (G_k, B_k)$, thus the $k$ pairs Rabin objective is achieved.

### Algorithm

Algorithm 10 is the algorithm we use to solve for the $k$ pairs Rabin game. It's a fixed point iteration of the $k$ Rabin objectives.

**Algorithm 10:** $\mathcal{A}'(\mathcal{W}_1) = \text{Rabin}(\mathcal{A}, G_1, B_1, \cdots, G_k, B_k)$

**Input:**
1. $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$;
2. $G_i, B_i \subseteq V$, $1 \leq i \leq k$;

**Output:**
1. $\mathcal{A}'(\mathcal{W}_1)$;

$G'_i = G_i = G_i/B_i, M'_i = M_i = V - G_i - B_i, B'_i = B_i$, $1 \leq i \leq k$;
**for** $(V' = V, i = 1, j = 0, \mathcal{W}_1 = \emptyset; i \leq k; j++)$ **do**
  $G'_i = G'_i \cap V', M'_i = M'_i \cap V', B'_i = B'_i \cap V'$;
  $\mathcal{A}'(S) = \text{safe}(\mathcal{A}'(V'), G'_i + M'_i)$;
  **if** $S == \emptyset$ **then** $i++$; continue;
  $X = \text{Büchi}(\mathcal{A}'(S), G'_i \cap S)$;
  **if** $X == \emptyset$ **then** $i++$; continue;
  $\mathcal{A}'(R) = \text{reach}_{\geq 0}(\mathcal{A}'(V'), X)$;
  $\mathcal{W}_1 += R$;
  $V' -= R$;
  $i = 1$;
$\mathcal{W}_2 = V/\mathcal{W}_1$;

**Result:**
1. Winning set $\mathcal{W}_1 \oplus \mathcal{W}_2 = V$;
2. Winning strategy of each node in $V_1 \cap \mathcal{W}_1$ is computed by Büchi$(\cdot, \cdot)$ and reach$_{\geq 0}(\cdot, \cdot)$ in the corresponding process;

### Proof of Soundness

The $k$ pairs Rabin algorithm 10 is a generalization of the generalized co-Büchi algorithm 8, thus it shares a similar reasoning as the generalized co-Büchi algorithm.

The $k$ pairs Rabin algorithm terminates when no more Rabin pairs in the $k$ Rabin pairs satisfy the Rabin objectives. The nodes in $\mathcal{W}_1$ can satisfy one of the $k$ Rabin objectives

by following the local strategy constructed through the iteration. This gives the proof of soundness.

**Counter-Example for Showing not Complete**

Even if we are doing some smart counting by tracking the out-degree of the nodes to build up connections among different Rabin objectives, we are still considering the $k$ Rabin objectives independently. The reason is that in each Rabin objective, it is a conjunction of Büchi objective and co-Büchi objective. The play to violate one Rabin objective may satisfy the other. We may not be able to satisfy any of the $k$ Rabin objectives independently. However, it is possible when we consider the $k$ Rabin objectives together.

A counter-example is shown in Figure 5.8. We may violate the first Rabin condition by playing $\pi_1 = M_1^\omega$, and violate the second Rabin condition by playing $\pi_2 = (B_2G_2)^\omega$. However, no play can violate the two Rabin conditions together. $\pi_1$ satisfies the second Rabin condition, whereas $\pi_2$ satisfies the first Rabin condition. The reason is that there is a conflict of violating these two conditions together in the play. That's why our $k$ pairs Rabin algorithm 10 is not complete.

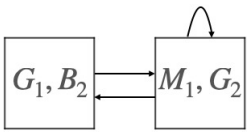


Figure 5.8: $k$ Pairs Rabin Counter-Example

**Strategy**

The strategy for the $k$ pairs Rabin algorithm is computed by Büchi$(\cdot,\cdot)$ and reach$_{\geq 0}(\cdot,\cdot)$ in the corresponding process. Each strategy is computed once since the current sub-arena $\mathcal{A}'$ is monotonically decreasing and $\mathcal{W}_1$ is monotonically increasing through the process of the iterations.

The reason of the strategy for the $k$ pairs Rabin game is similar to that for the generalized co-Büchi game, which guarantees that play $\pi$ satisfies some of the $k$ Rabin objectives. We don't know which Rabin objective we will eventually satisfy, but we know we can eventually satisfy some of them by the construction of the algorithm. This shows that such strategy is indeed a winning strategy for player 1.

The local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$ since even though we have $k$ Rabin objectives, we only need to satisfy one of them to satisfy the $k$ pairs Rabin condition. Therefore, we only need 1 unit for the local strategy to focus on 1 Rabin objective for each node $v \in V_1 \cap \mathcal{W}_1$.

**Complexity Analysis**

In a $k$ pairs Rabin game $R\mathcal{G}$, $\mathcal{A} = (V, E)$ is the arena and

Obj $= \{(G_1, B_1), \cdots, (G_k, B_k)\}$ is the objective set, where $G_i, B_i \subseteq V$, $1 \leq i \leq k$. $|V| = n$ is the number of nodes and $|E| = m$ is the number of edges.

**Proposition 5.6.1.** *A $k$ pairs Rabin game can be solved by a sound but not complete algorithm in $\mathcal{O}(m)$ space, $\mathcal{O}(kn^2m)$ time, and the local strategy takes 1 unit for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Proposition 5.6.2.** *A $k$ pairs Streett game can be solved by a complete but not sound algorithm in $\mathcal{O}(m)$ space, $\mathcal{O}(kn^2m)$ time, and the local strategy takes $k$ units for each node $v \in V_1 \cap \mathcal{W}_1$.*

**Space Complexity:** $\mathcal{O}(n + m)$

Since the size of the arena is depicted by $n$ nodes and $m$ edges, the space complexity of the algorithm is $\mathcal{O}(n + m)$.

**Time Complexity:** $\mathcal{O}(kn^2m)$

Similar to the analysis for generalized co-Büchi, in each iteration, the safety process costs $\mathcal{O}(m)$, the Büchi process costs $\mathcal{O}(nm)$, and $|G_i|$ and $|B_i|$ reduces by at least one after $k$ iterations.

After algorithm 10 terminates, $j$ gives the total number of iterations for safe$(\cdot, \cdot)$, Büchi$(\cdot, \cdot)$ and reach$_{\geq 0}(\cdot, \cdot)$, which is upper bounded by $j \leq |\bigcup_{i=1}^{k} G_i \oplus M_i| \cdot k \leq |V| \cdot k = nk$. The total cost of safe$(\cdot, \cdot)$ is given by $\mathcal{O}(nk \cdot m) = \mathcal{O}(knm)$.

The total cost of Büchi$(\cdot, \cdot)$ is given by $\mathcal{O}(nk \cdot nm) = \mathcal{O}(kn^2m)$.

The total cost of reach$_{\geq 0}(\cdot, \cdot)$ is linear, i.e., $\mathcal{O}\left(\sum_{i=1}^{j} \text{reach}_{\geq 0}(\cdot, \cdot)\right) = \mathcal{O}(m)$.

Therefore, the time complexity of the algorithm is given by

$\mathcal{O}(knm + kn^2m + m) = \mathcal{O}(kn^2m)$.

**When does this worst case time complexity occur?**

In each iteration, $X$ is not empty for the last Rabin pair $(G_k, B_k)$ only,

$\mathcal{O}(|G_k|) = \mathcal{O}(|B_k|) = \mathcal{O}(n)$, and $|G_k|$, $|B_k|$ reduces by 1 each time. In other words, after iterating each Rabin objective, we always have empty winning sets for the first $(k-1)$ Rabin

objectives, and only one node in the winning set for the last Rabin objective. Solving each Rabin objective costs $\mathcal{O}(nm)$, thus obtaining one node in the winning set costs $\mathcal{O}(knm)$. Since there are a total of $n$ nodes in $V$, the total time is given by $\mathcal{O}(kn^2m)$.

We can see that this worst case time complexity rarely happens in practice unless deliberately designed, and the coefficient is quite small. Therefore, we can expect a linear time complexity $\mathcal{O}(km)$ in practice.

**How to make the Rabin algorithm complete?**

Here we briefly discuss how to construct a complete algorithm based on the sound but not complete $k$ pairs Rabin algorithm 10.

Suppose after we apply algorithm 10, $\mathcal{W}_2$ is not empty. We apply algorithm 10 again. If the set $S$ is empty for each Rabin objective after doing safety, then our algorithm is complete. The corresponding $k$ pairs Street game is reduced to a generalized $k$ Büchi game, and the generalized Büchi algorithm 7 gives a sound and complete solution.

If the set $S$ is not empty for some Rabin objective $i$, then we know that after we iterate a Büchi process, $X$ will be empty. We instead solve a $(k-1)$ Rabin game in $S$ for the remaining $(k-1)$ Rabin objectives. This is where the recursion comes from which requires us to solve sub-games within sub-games. The time complexity can be as worse as $\mathcal{O}(mn^{k+1}kk!)$ [27], which is not feasible in practice.

# Chapter 6

# Non-deterministic Transition System (NTS)

In this chapter, we will introduce the problem formulation and the solution of some control problems defined in NTS. We will discuss the relation of an NTS in our control setting and an arena in the infinite two player game setting. We will define the control problem in an NTS and solve it by converting it to a graph searching problem as a infinite two player game in a bipartite arena. For related contents about control problems in NTS, one may refer to [4, Chapter 5], [20], and [40].

## 6.1 Analysis of the NTS and DA

In this section, we will give a quantification and some analysis of the Non-deterministic Transition System (NTS) and the Deterministic Automaton (DA). Such quantification will be used in the complexity analysis of solving each control problem. This section serves as a set up for the problem formulation in the next section.

### 6.1.1 Analysis of the NTS

We will give a definition of the NTS followed by a quantification. We will also give two graph representations of NTS, and show that the NTS is itself in the form of a bipartite arena introduced in the infinite two player game.

**Definition 6.1.1** (non-deterministic transition system)**.** *A Non-deterministic Transition System (NTS) is a 4-tuple* $(X, \Sigma, \xi, L)$*, where*

1. $X$ *is the set of system states;*
2. $\Sigma$ *is the set of control actions;*
3. $\xi : X \times \Sigma \to 2^X$ *is a non-deterministic transition function;*
4. $L \subseteq 2^{AP}$ *is the set of labels.*

**Quantification of the NTS**

After giving the definition of NTS, we will define some quantifiers to help us quantify the NTS.

- $n_0 = |X|$ is the number of system states;
- $|\Sigma|$ is the number of control actions;
- $m_0 = \sum_{x \in X} \sum_{\sigma \in \Sigma} |\xi(x, \sigma)|$ is the number of system transitions;
- $|AP|$ is the number of atomic propositions;
- $|L| \leq 2^{|AP|}$ is the number of different labels;
- $|\xi| = \sum_{x \in X} \sum_{\sigma \in \Sigma} \kappa\left(\xi(x, \sigma)\right)$ is the number of non-deterministic transition functions, or the total number of control actions of the system;
- $\bar{\sigma} \approx \dfrac{\text{number of non-deterministic transition functions}}{\text{number of system states}} = \dfrac{|\xi|}{n_0} \leq |\Sigma|$ is the average number of control actions of each system state;
- $\bar{\delta} \approx \dfrac{\text{number of system transitions}}{\text{number of non-deterministic transition functions}} = \dfrac{m_0}{|\xi|}$ is the average number of non-determinism;
- $\bar{\sigma}\bar{\delta} \approx \dfrac{\text{number of system transitions}}{\text{number of system states}} = \dfrac{m_0}{n_0}$ is the average degree of the system.

**Non-determinism**

Consider a non-deterministic transition function of the simplest form $\xi(x_1, \sigma) = \{x_2, x_3\}$. It's non-deterministic transitions can be written as two 3-tuples: $(x_1, \sigma, x_2)$ and $(x_1, \sigma, x_3)$. Here we define the number of transitions as the number of non-determinism, which is $|\xi(x_1, \sigma)| = 2$ in this case.

**Counting Function**

$\kappa$ is a counting function that counts the number of transition functions: 1 if a transition function $\xi(x, \sigma)$ exists, 0 otherwise.

$$\kappa(\xi) = \begin{cases} 1 & \text{if } \xi \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

**Two Graph Representations of the NTS**

Consider a non-deterministic transition function $\xi(x_1, \sigma) = \{x_2, x_3\}$. It can be written as two 3-tuples: $(x_1, \sigma, x_2)$ and $(x_1, \sigma, x_3)$. It has two graph representations: placing $\sigma$ on the edge and placing $\sigma$ as a new node. Here we give the two graph representations and their corresponding quantification. The picture is shown in Figure 6.1.

1. **Place $\sigma$ on the Edge**

   If we consider $x_1$, $x_2$, and $x_3$ as nodes and place $\sigma$ on the edges, then we have two multi-edges: $x_1 \xrightarrow{\sigma} x_2$ and $x_1 \xrightarrow{\sigma} x_3$.

   **Quantification of the Graph**

   Number of nodes: $n = n_0$.

   Number of multi-edges: $m = m_0$.

2. **Place $\sigma$ as a New Node**

   If we consider $x_1$, $x_2$, and $x_3$ as player 1 nodes and $\sigma$ as a player 2 node, then we have three edges: $(x_1, \sigma)$, $(\sigma, x_2)$, and $(\sigma, x_3)$.

   **Quantification of the Graph**

   Number of nodes: $n = n_0 + |\xi| = n_0 + \bar{\sigma} n_0 \leq n_0 + |\Sigma| \cdot n_0 = \mathcal{O}(n_0)$.

   Number of edges: $m = m_0 + |\xi| = \mathcal{O}(m_0)$.

**Remark.** *1. The second representation has $|\xi|$ more nodes and edges than the first one.*

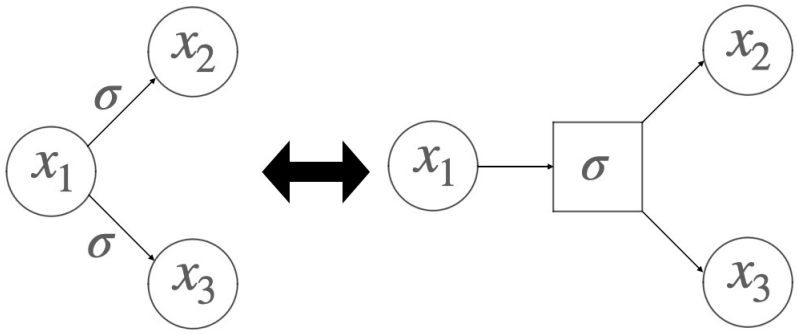


Figure 6.1: Two Graph Representations of NTS

2. *These two graph representations are equivalent. It's just different aspects of understanding NTS. The design of data structure in the implementation is another story, which is still different from these two.*

3. *In the first representation, each system state has about $\bar{\sigma}$ control actions. Therefore, it is more compact and involves multi-edges.*

4. *The second representation forms a bipartite graph. More precisely, a bipartite arena in the two player game setting, where the system states are the player 1 nodes, and the control actions are the player 2 nodes. Therefore, the NTS is just an application of the arena in the two player game.*

5. *The second representation is a decomposition of the first one, which is easier for us to understand the algorithms in the two player game setting. It is easier to understand because it is less compact, no multi-edges, and can use the duality property of the two players.*

6. *We consider the control specification in the NTS as a two player game, but the status of the two players are different: since the system states are the player 1 nodes and the control actions are the player 2 nodes, we only care about the winning set and the winning strategies of the player 1.*

7. *From now on, we consider the NTS as a bipartite arena.*

8. *In the complexity analysis, we use the first representation for simplicity; in the understanding of the algorithms, we use the second representation.*

### 6.1.2 Analysis of the DA

We will give a definition of the DA followed by a quantification. We will also give two representations of the deterministic transition function $\tau$.

**Definition 6.1.2** (deterministic automaton)**.** *A Deterministic Automaton (DA) is a 5-tuple* $DA = (Q, L, \tau, q_0, Acc)$*, where*

- $Q$ *is the set of automaton states;*
- $L \subseteq 2^{AP}$ *is the set of labels;*
- $\tau : L \times Q \to Q$ *is a deterministic transition function;*
- $q_0 \in Q$ *is the unique initial automaton state;*
- $Acc \subseteq 2^Q$ *is the set of final states in a finite automaton or the set of accepting states in an* $\omega$*-automaton.*

#### Quantification of the DA

After giving the definition of DA, we will define some quantifiers to help us quantify the DA.

- $n_1 = |Q|$ is the number of automaton states;
- $m_1 = |\tau|$ is the number of automaton transitions;
- $|AP|$ is the number of atomic propositions;
- $|L| \leq 2^{|AP|}$ is the number of different labels;
- The accepting condition Acc will be discussed later when we specify the different automata.

#### More on the Transition Function and the Labels

The deterministic transition function $\tau$ has two representations: as a boolean function $\tau\_next$ and as a look-up table $\tau\_prime$.

The labels $L$ have two representations: the binary representation and the decimal representation.

We will show the two representations of the Labels $L$ in the two representations of the deterministic transition function $\tau$.

The two representations of the deterministic transition function $\tau$ are as follows.

1. **As a Boolean Function** $\tau\_next$
   - When we compute the boolean function $\tau\_next$, we use the binary representation of the labels.
   - The boolean function $\tau\_next : L \times Q \to Q$ has $n_1$ cases and $m_1$ boolean functions, corresponding to the $n_1$ nodes and $m_1$ edges.
   - Computing each boolean function costs $\mathcal{O}(|AP|)$.
   - Encoding from binary representation to decimal representation costs $\mathcal{O}(|AP|)$.
2. **As a Look-Up Table** $\tau\_prime$
   - When we check the look-up table $\tau\_prime$, we use the decimal representation of the labels.
   - The look-up table $\tau\_prime : L \times Q \to Q$ is of size $2^{|AP|}n_1$.
   - Decoding from decimal representation to binary representation costs $\mathcal{O}(|AP|)$.

## 6.2 Problem Formulation and Procedure of Solution

After we have introduced the concept of winning set and winning strategy in chapter 4, now we can formally define our control problem in NTS.

**Problem Formulation**

Given an NTS $= (X, \Sigma, \xi, L)$ and a control specification $\varphi$, find the winning set $\mathcal{W}_0 \subseteq X$ and the winning control strategy $\Pi : X^* \to \Sigma$ such that all the trajectories $\rho = x_0 x_1 x_2 \cdots$ induced by the initial state $x_0 \in \mathcal{W}_0$ and the sequence of control inputs $\alpha = \sigma_0 \sigma_1 \sigma_2 \cdots$ inductively satisfy the accepting condition of $\varphi$ regardless of the non-determinism of the NTS.

**How is a control specification defined?**

In the set of system states $X$ in NTS, we use different atomic propositions in $AP$ to describe different subsets of $X$. For example, we can define the set of states labeled by $a_1$ as the target set $T_1$, the set of states labeled by $a_2$ as the safe set $S_2$. Also, we can define

the set of states labeled by $a_3$ as the unsafe set, which can be used to depict the obstacles and the boundary.

We can give each state $x \in X$ several meanings by assigning different atomic propositions to $x$, which forms $l$, the label of $x$. $l$ is a set of atomic propositions and thus a subset of $AP$, i.e., $l \subseteq AP$.

Therefore, from the correspondence of the subset of $X$ and the atomic propositions $AP$, a control specification $\varphi$ can be defined by the subsets of $X$, or by the atomic propositions $AP$.

Describing control specification using atomic propositions allows us to describe the control specification using logic languages such as Linear Temporal Logic (LTL), which can be converted to a deterministic automaton (DA) in an automation process.

For example, suppose our control specification $\varphi$ is a reachability objective, then we can define $\varphi$

1. by the subsets of $X$: eventually reach $T$, where $T \subseteq X$ is the set of states labeled by $a$;
2. by the atomic propositions $AP$: eventually reach the set of states labeled by $a$;
3. by the LTL formula $\varphi = Fa$.

**How is a trajectory constructed?**

A trajectory $\rho = x_0x_1x_2\cdots$ is a sequence of system states $x_i \in X$, for $i \geq 0$, which can be finite ($r$) or infinite ($\rho$) depending on the control specification $\varphi$.

A sequence of control inputs $\alpha = \sigma_0\sigma_1\sigma_2\cdots$ is a sequence of control actions $\sigma_i \in \Sigma$, for $i \geq 0$, which can be finite ($w$) or infinite ($\alpha$) depending on the trajectory.

We borrow the notation of run and word from the $\omega$-automata theory to represent trajectory and control inputs in our control problem. The corresponding relation is as follows. A run $\rho = q_0q_1q_2\cdots$ is a sequence of automaton states $q \in Q$, whereas a trajectory $\rho = x_0x_1x_2\cdots$ is a sequence of system states $x \in X$. An input word $\alpha = l_0l_1l_2\cdots$ is a sequence of labels $l \in L$, whereas a sequence of control inputs $\alpha = \sigma_0\sigma_1\sigma_2\cdots$ is a sequence of control actions $\sigma \in \Sigma$.

At the moment $i$, for $i \geq 0$, the agent is on state $x_i$. Based on the information such as $x_i$, our strategy $\Pi$ gives a control action $\sigma_i$, and the non-determinism of the NTS leads the agent to $x_{i+1} \in \xi(x_i, \sigma_i)$. The trajectory is inductively extended by this procedure.

### 6.2.1 Procedure to Solve Control Problems in NTS

In this subsection, we will introduce the procedure to solve control problems in NTS in an effort to improve the space complexity and the time complexity of solving a certain control problem in NTS.

**Classification by Control Specifications**

We try to convert our control problem to a graph searching problem in the form of an infinite two player game. The procedure is classified by the control specifications: if the control specification can be described as a certain game $\mathcal{G}$ by the states in NTS, then we use the particular algorithm of $\mathcal{G}$ to solve for $\mathcal{G}$ in NTS; else we describe our control specification as an LTL formula and convert it to a DA with a corresponding accepting condition Acc.

**Procedure of General Solution**

Describing the control specification as an LTL formula $\varphi$ allows us to give a general solution of the control problems in NTS:

1. describe the control specification as an Linear Temporal Logic (LTL) formula $\varphi$;
2. convert $\varphi$ to a Deterministic Automaton DA with a corresponding accepting condition Acc in an automation process;
3. take product of NTS and DA to construct the product space $P\mathcal{A}$;
4. use the Acc of DA to define the objective set Obj in $P\mathcal{A}$, and the accepting condition of DA to define the winning condition of the game $\mathcal{G}$ in $P\mathcal{A}$;
5. solve the game $\mathcal{G}$ with the winning condition described by Obj in $P\mathcal{A}$;
6. map the winning set and the winning strategy in $P\mathcal{A}$ back to NTS.

**Procedure to Determine the Accepting Condition of DA**

The procedure to determine the accepting condition of DA is as follows:

- if $\varphi$ can be converted to a finite automaton, then we convert $\varphi$ to a deterministic finite automaton $DFA$ (all the finite automaton can be determinized);
- else we convert $\varphi$ to an $\omega$-automaton;

- if $\varphi$ can be converted to a deterministic Büchi automaton, then we convert $\varphi$ to a deterministic Büchi automaton $DBA$;
- else if $\neg\varphi$ can be converted to a deterministic Büchi automaton, then we convert $\varphi$ to a deterministic co-Büchi automaton $DCA$;
- else we convert $\varphi$ to a deterministic Rabin automaton $DRA$ or a deterministic Streett automaton $DSA$ ($DRA$ and $DSA$ are both $\omega$-regular complete, just the conversion for one may be more succinct than the other for a specific $\varphi$).

**Comparison between the Two Methods Classified by Control Specification**

We can define the control specification in two ways. Here we give a comparison between the two methods of solving control problems in NTS classified by the two methods of defining control specifications.

**Control Specification Described by States in NTS**

If we describe the control specification by the states in NTS, then we can only address some simple specifications. We need to design each specific algorithm for each specification, which is not compatible. However, the benefit of designing each specific algorithm for each specification is that, we can take advantage of the property of the specific specification to improve the efficiency of the algorithm.

**Control Specification Described by LTL formula**

The reason we would prefer to describe our control specification $\varphi$ as a logic formula such as an LTL formula is that, $\varphi$ can be converted into a deterministic automaton DA in an automation process. The size of the DA is optimized by the automation tool. The automaton process saves the working load and reduces the chance of making mistakes.

Then we take product of the NTS and the DA and solve a game $\mathcal{G}$ in the product space $NTS \otimes DA$. The winning condition of the game $\mathcal{G}$ is defined by the accepting condition of the DA. The accepting condition of DA can be classified into certain types, and the LTL formulas that can be converted to DAs with the same type of accepting condition can be solved using the same algorithm in the game, which is more compatible.

After solving the game $\mathcal{G}$ in the product space, we map the information from the product space back to NTS.

This gives a systematic approach of solving control problems in NTS.

**Example**

Of course, for some common specifications described by the states in NTS, we can easily construct the algorithms for these specifications by gluing the existing algorithms. It is more efficient to solve the control problem in NTS than in the product space.

Here we give one example:

1. Reach $T_1$ and then reach $T_2$.

   This is a generalization of the reachability problem reach $T$. It can have variations by whether we allow reaching $T_2$ at the same moment as reaching $T_1$, or reaching $T_2$ at a latter moment than reaching $T_1$. Here we consider the former case.

   We start from $T_2$ to solve for the reachable set of $T_2$, and then solve for the reachable set of the intersection of the reachable set of $T_2$ and $T_1$.

   We can describe this problem as a game $\mathcal{G} = (\mathcal{A}, \text{Obj})$, where $\mathcal{A} = (V, E)$ and $\text{Obj} = \{T_1, T_2\}$. The procedure is as follows: $\mathcal{A}'(R_2) = \text{reach}_{\geq 0}(V, T_2)$, $T = R_2 \cap T_1$, $\mathcal{A}'(\mathcal{W}_1) = \text{reach}_{\geq 0}(V, T)$, where the winning set is $\mathcal{W}_1$, and the winning strategy is computed in $\mathcal{A}'(\mathcal{W}_1) = \text{reach}_{\geq 0}(V, T)$.

   Given $|V| = n$ and $|E| = m$, the space complexity is $\mathcal{O}(n + m)$ and the time complexity is $\mathcal{O}(2(n + m))$. The time complexity is given by computing the reachability twice, each taking $\mathcal{O}(n + m)$ time.

   This control specification can be turned into a LTL formula $\varphi = F(a \wedge Fb)$, where $a$ and $b$ are the labels for $T_1$ and $T_2$ respectively.

   $\varphi$ can be converted to a $DFA$ with 3 states and 5 transitions.

   Given $|V| = n$ and $|E| = m$, the space complexity is $\mathcal{O}(3(n + m))$ and the time complexity is $\mathcal{O}(3(n+m))$. The time complexity is the same as the space complexity for a game converted from a $DFA$.

   From the comparison, we can see that it is more efficient to solve a control problem in NTS than in the product space.

### 6.2.2 Organization of Rest of the Chapter

The idea of solving a control problem in NTS is to convert it to an infinite two player game. The purpose of this chapter is to give the conversion and then solve the infinite game using the algorithms given in the previous chapter.

A control problem in NTS consists of the environment NTS and the control specification $\varphi$.

An infinite two player game $\mathcal{G} = (\mathcal{A}, \text{Obj})$ consists of the arena $\mathcal{A}$ and the winning condition of the game defined by the objective set Obj.

The matching between the control problem and the infinite game is now straightforward:

- control problem corresponds to infinite game;
- environment corresponds to arena;
- accepting condition of control specification $\varphi$ corresponds to winning condition of $\mathcal{G}$.

The control specification is classified by a proposition using states in NTS and as an LTL formula. The control specification also classifies the two methods of solving the control problem.

We will discuss the general part of the conversion in the following two sections, which is converting from environment to arena. We will leave the conversion of accepting condition of control specification to winning condition of the game when we discuss each specific control specification in the latter section.

## 6.3 Control Specification Defined by States in NTS

In this section, we will introduce the procedure to solve the control problem in NTS with control specification $\varphi$ defined by states in NTS.

### 6.3.1 Conversion from NTS to Arena

We will show that NTS is a bipartite arena and discuss the specialties of NTS from a general arena. We will first give the definition of a bipartite graph, and then the definition of a bipartite arena. We will conclude by showing NTS is a bipartite arena.

**Definition 6.3.1** (bipartite graph)**.** *A graph $G = (V, E)$ is bipartite if $V = V_1 \oplus V_2$ and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$.*

In other words, in a bipartite graph, the set of nodes $V$ is partitioned as two sets $V_1$ and $V_2$, and there only exist edges from $V_1$ to $V_2$, or from $V_2$ to $V_1$.

**Definition 6.3.2** (bipartite arena)**.** *An arena $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$ is bipartite if $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$.*

In other words, since there is a natural partition of $V$ into $V_1$ and $V_2$, we bipartite the edges in a bipartite arena according to this partition of nodes.

Consider an NTS $= (X, \Sigma, \xi, L)$ and an arena $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$. The conversion of NTS to arena $\mathcal{A}$ is as follows:

- $V_1 = X$ is the set of player 1 nodes;
- $V_2 = \{\sigma_x | x \in X, \sigma \in \Sigma, |\xi(x, \sigma)| > 0\}$ is the set of player 2 nodes;
- $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges constructed by
  $E \cup (V_1 \times V_2) = \{(x, \sigma_x) | x \in X, \sigma \in \Sigma, |\xi(x, \sigma)| > 0\}$ and
  $E \cup (V_2 \times V_1) = \{(\sigma_x, x') | x \in X, \sigma \in \Sigma, |\xi(x, \sigma)| > 0, x' \in \xi(x, \sigma)\}$.

We convert an NTS to an arena $\mathcal{A}$ according to the second graph representation of NTS. This conversion clearly shows that an NTS can be considered as a bipartite arena.

**Quantification of Arena by Quantifiers of NTS**

Now we give a quantification of the bipartite arena $\mathcal{A}$ converted from the NTS by the quantifiers of NTS.

Recall that in the quantification of an NTS,

- $n_0 = |X|$ is the number of system states;
- $|\Sigma|$ is the number of control actions;
- $m_0 = \sum_{x \in X} \sum_{\sigma \in \Sigma} |\xi(x, \sigma)|$ is the number of system transitions;
- $|\xi| = \sum_{x \in X} \sum_{\sigma \in \Sigma} \kappa\left(\xi(x, \sigma)\right)$ is the number of non-deterministic transition functions, or the total number of control actions of each system state;
- $\bar{\sigma} \approx \dfrac{\text{number of non-deterministic transition functions}}{\text{number of system states}} = \dfrac{|\xi|}{n_0} \leq |\Sigma|$ is the average number of control actions of each system state.

The quantification of the bipartite arena $\mathcal{A}$ converted from the NTS is thus given by

- $|V_1| = |X| = n_0$;

- $|V_2| = |E \cup (V_1 \times V_2)| = |\xi| \approx \bar{\sigma} n_0 \leq |\Sigma| n_0$;
- $|E \cup (V_2 \times V_1)| = \sum_{x \in X} \sum_{\sigma \in \Sigma} |\xi(x, \sigma)| = m_0$;
- $|V| = |V_1| + |V_2| = n_0 + |\xi| \approx n_0 + \bar{\sigma} n_0 = (1 + \bar{\sigma}) n_0 \approx \bar{\sigma} n_0$;
- $|E| = |E \cup (V_1 \times V_2)| + |E \cup (V_2 \times V_1)| = |\xi| + m_0 \approx \bar{\sigma} n_0 + m_0 \approx m_0$.

For simplicity, if we would like to do the complexity analysis using two quantifiers, then we can quantify the number of nodes $|V| = n_0$ and the number of edges $|E| = m_0$ as in a classic graph or an arena. The average number of control actions $\bar{\sigma}$ is absorbed in the operations of each system state $x$.

If we would like to do a more detailed complexity analysis, then we can add the average number of control actions $\bar{\sigma}$ as a third quantifier.

Our complexity analysis for the NTS is quantified by two quantifiers $n_0$ and $m_0$ for simplicity. If we want to take $\bar{\sigma}$ into consideration, then we can simply replace $n_0$ by $\bar{\sigma} n_0$ in the result for space complexity. $\bar{\sigma}$ doesn't affect the time complexity.

**Mapping behind Conversion**

Considering the NTS as a bipartite arena is an application of the graph searching method in the control setting. In this conversion, we consider the system states as player 1 nodes and the control actions as the player 2 nodes.

In the infinite two player game, we always "play" the game from the perspective of player 1. Therefore, the player 1 nodes are fully controllable, and the player 2 nodes are fully uncontrollable.

In the NTS, the system states are fully controllable since we can choose a control action at each system state; however, the control actions are fully uncontrollable since we can not choose the next system state to go to after applying a control action as a result of the non-determinism of the NTS.

Therefore, the bipartite arena in the infinite two player game is a suitable model to depict the controllable and the uncontrollable part of the NTS. The player 1 nodes are often used to depict the system states, while the player 2 nodes are often used to depict the uncertainty of the environment. In fact, from the property that the infinite two player game is a 0-sum game, it can capture the worst case uncertainty of the environment, thus giving a conservative result that is guaranteed to be sound. The missing of completeness

is almost unavoidable when constructing the abstraction from continuous space to discrete space, yet it can be alleviated in a cost of refinement to increase precision.

Looking at a non-deterministic transition function $x' \in \xi(x, \sigma)$ from the perspective of the bipartite arena, at the player 2 node $\sigma_x$, it is possible to go to any of the node $x'$ such that $x' \in \xi(x, \sigma)$. This is because when constructing the abstraction, there exist points $x_c$ in one region such that after taking control action $\sigma_c$ in one time step, the agent will reach $x'_c$. The mapping from the continuous space to the discrete space gives the non-deterministic transition function $x' \in \xi(x, \sigma)$.

However, we may not really have the worst case uncertainty in the dynamics all the time, i.e., acting as player 2 in each step. This is where the conservatism comes from. Therefore, the agent may still satisfy the accepting condition of the control specification even if we consider some non-deterministic transitions as deterministic ones. This is the motivation for further research by relaxing the non-determinism by adding probabilities to the transitions. The probabilities can be approximated from the frequency of the samplings, or through theoretical analysis.

For example, consider a non-deterministic transition function $\xi(x_1, \sigma) = \{x_1, x_2\}$. In the infinite two player game setting, if the objective of player 1 is to escape from $x_1$, then this is impossible because the player 2 can trap player 1 in $x_1$ by choosing $(x_1, \sigma, x_1)$. In the control setting, when we apply the control action $\sigma$ at $x_1$, we may get trapped at $x_1$ by the self-loop of $x_1$. However, it is also possible that the agent get trapped in $x_1$ in the first few steps and then escape to $x_2$, which has a potentially larger chance of satisfying the control specification.

**Specialties of Bipartite Arena**

Since our focus is on solving the control problem in NTS, we will discuss the specialties of bipartite arena from arena in a control setting. We tailor our settings and functions in the infinite two player game to fit our control problem.

Since we consider NTS as a bipartite arena, the system states $x$ are considered as the player 1 nodes $V_1$ and the control actions $\sigma$ are considered as the player 2 nodes $V_2$. Also, the edges are from $x$ to $\sigma$, or from $\sigma$ to $x$. There are no edges from $x$ to $x$, or from $\sigma$ to $\sigma$. A play $\pi = x_0\sigma_0x_1\sigma_1\cdots$ in a bipartite arena is thus an alternating sequence between $x$ and $\sigma$, initializing from $x$.

Therefore, our emphasis is on $V_1$, and $V_2$ is secondary. We only consider the system states $x$ in the winning set, and the winning strategy of $x$, which are control actions $\sigma$. We omit the winning set and the winning strategy for player 2.

### 6.3.2 Control Specification Defined by States in NTS

In this subsection, we will introduce the procedure to solve the control problem in NTS with the control specification $\varphi$ defined by the states in NTS.

If the accepting condition of the control specification $\varphi$ is in the form of the winning condition of a game $\mathcal{G}$, then we convert $NTS$ to a bipartite arena $\mathcal{A}$, followed by converting the control problem with control specification $\varphi$ in NTS to a infinite game $\mathcal{G}$ in the bipartite arena $\mathcal{A}$.

**Same Logic behind Control Problem and Infinite Two Player Game**

After the matching between NTS and bipartite arena, we will show that the general logic behind the control problem and the infinite two player game is the same. The detailed conversion will be discussed in each specific game because that depends on the specific objective set Obj and the winning condition, which may differ from game to game.

We will tailor our discussion of the infinite game to fit our control problem. We will show the correspondence of control problem and infinite game by presenting the related materials in pairs.

**Problem Formulation of Control Problem**

Given an NTS $= (X, \Sigma, \xi, L)$ and a control specification $\varphi$, find the winning set $\mathcal{W}_0 \subseteq X$ and the winning control strategy $\Pi : X^* \to \Sigma$ such that all the trajectories $\rho = x_0 x_1 x_2 \cdots$ induced by the initial state $x_0 \in \mathcal{W}_0$ and the sequence of control inputs $\alpha = \sigma_0 \sigma_1 \sigma_2 \cdots$ inductively satisfy the accepting condition of $\varphi$ regardless of the non-determinism of the NTS.

The key point of solving a control problem is to solve for the winning set $\mathcal{W}_0 \subseteq X$ and the winning control strategy $\Pi : X^* \to \Sigma$ in NTS. The winning set $\mathcal{W}_0 \subseteq X$ is the set of initial system states to place our agent. By placing our agent initially in a system state $x_0 \in \mathcal{W}_0$, and applying the winning control strategy $\Pi : X^* \to \Sigma$ at each system state, a sequence of control inputs $\alpha = \sigma_0 \sigma_1 \sigma_2 \cdots$ is induced. The output of the winning control strategy is a control action $\sigma \in \Sigma$. We can also expect that all the trajectories $\rho = x_0 x_1 x_2 \cdots$ induced by $x_0$ and $\alpha$ according to the non-deterministic transition function $x_{i+1} \in \xi(x_i, \sigma_i)$, $i \geq 0$ to inductively satisfy the accepting condition of the control specification $\varphi$, regardless of the non-determinism of NTS. If $\varphi$ is described by the subsets in $X$, then the accepting condition of $\varphi$ is defined as a proposition using the occurrence set $\mathrm{Occ}(\rho)$ or the infinity set $\mathrm{Inf}(\rho)$, and the subsets of $X$; otherwise $\varphi$ is in the form of an LTL formula and the accepting condition of $\varphi$ will be discussed later.

**Problem Formulation of Infinite Game**

Given an infinite game $\mathcal{G} = (\mathcal{A}, \text{Obj})$, where $\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1))$ is the bipartite arena and $\text{Obj} \subseteq 2^V$ is the objective set used to define the winning condition of $\mathcal{G}$, find the winning set $\mathcal{W}_1 \subseteq V_1$ and the winning strategy $\Pi_1 : V^* \to V_2$ for $v \in \mathcal{W}_1$ such that all the plays $\pi = v_0 v_1 v_2 \cdots$ induced by the initial node $v_0 \in \mathcal{W}_1$, $\Pi_1$, and $\Pi_2$ inductively satisfy the winning condition of $\mathcal{G}$ regardless of $\Pi_2$, where $\Pi_2 : V^* \to V_1$ is any strategy for player 2.

The key point of solving an infinite game $\mathcal{G}$ is to solve for the winning set $\mathcal{W}_1 \subseteq V_1$ and the winning strategy $\Pi_1 : V^* \to V_2$ in $\mathcal{A}$. The winning set $\mathcal{W}_1 \subseteq V_1$ is the set of initial nodes to place our token. By placing our token initially in a node $v_0 \in \mathcal{W}_1$, and applying the winning strategy $\Pi_1 : V^* \to V_2$ at each $V_1$ node, and any strategy $\Pi_2 : V^* \to V_1$ at each $V_2$ node, a play $\pi = v_0 v_1 v_2 \cdots$ is induced. We can expect that all the plays $\pi = v_0 v_1 v_2 \cdots$ induced by $v_0$, $\Pi_1$ and $\Pi_2$ according to the edges $(v_i, v_{i+1}) \in E$, $i \geq 0$ to inductively satisfy the winning condition of $\mathcal{G}$, regardless of $\Pi_2$. The winning condition of $\mathcal{G}$ is defined as a proposition using the occurrence set $\text{Occ}(\rho)$ or the infinity set $\text{Inf}(\rho)$, and the objective set Obj.

Consider the system states $X$ as player 1 nodes $V_1$, and the control actions $\Sigma$ as player 2 nodes $V_2$. The play is thus $\pi = x_0 \sigma_0 x_1 \sigma_1 x_2 \sigma_2 \cdots$, which can be considered as a trajectory $\rho = x_0 x_1 x_2 \cdots$ induced by the control inputs $\alpha = \sigma_0 \sigma_1 \sigma_2 \cdots$.

From our analysis, the control problem and the infinite game matches up, thus they are equivalent.

## 6.4 Control Specification Defined as an LTL Formula

In this section, we will introduce the accepting condition of LTL in NTS and the procedure to solve the control problem in NTS with the control specification $\varphi$ defined as an LTL formula.

### 6.4.1 Accepting condition of LTL in NTS

In this subsection, we will introduce the accepting condition of an LTL formula $\varphi$ in NTS.

In a control problem in NTS, consider the control specification $\varphi$ defined as an LTL formula. What exactly does it mean for a trajectory $\rho = x_0 x_1 x_2 \cdots$ to satisfy the accepting condition of the LTL formula $\varphi$?

We first convert $\varphi$ to a Deterministic Automaton DA in an automation process. The accepting condition of DA is defined by the accepting condition of the run $\rho$. The accepting condition of DA is defined as a proposition using the occurrence set $\text{Occ}(\rho)$ or the infinity set $\text{Inf}(\rho)$, and the accepting set Acc.

Recall that a run $\rho_q = q_0 q_1 q_2 \cdots$ is a sequence of automaton states $q \in Q$, and a word $\alpha_l = l_0 l_1 l_2 \cdots$ is a sequence of labels $l \in L$. Each label $l_i$ is also the label of the system state $x_i$ in the trajectory $\rho_x = x_0 x_1 x_2 \cdots$, for $i \geq 0$. A run $\rho_q = q_0 q_1 q_2 \cdots$ is uniquely induced by the unique initial automaton state $q_0 \in Q$ and the input word $\alpha_l = l_0 l_1 l_2 \cdots$ according to the deterministic transition function $\tau(l_i, q_i) = q_{i+1}$, $i \geq 0$. An input word $\alpha_l$ satisfies the accepting condition of DA if its induced run $\rho_q$ satisfies the accepting condition of DA.

Therefore, a trajectory $\rho_x = x_0 x_1 x_2 \cdots$ satisfies the accepting condition of the LTL formula $\varphi$ if its input word $\alpha_l = l_0 l_1 l_2 \cdots$ satisfies the accepting condition of DA converted from $\varphi$.

Now we have four sequences: the trajectory $\rho_x = x_0 x_1 x_2 \cdots$, the control inputs

$\alpha_\sigma = \sigma_0 \sigma_1 \sigma_2 \cdots$, the run $\rho_q = q_0 q_1 q_2 \cdots$, and the input word $\alpha_l = l_0 l_1 l_2 \cdots$. They are induced jointly by the initial states $x_0$ and $q_0$, the control strategy $\Pi : \cdot \to \Sigma$, the non-deterministic transition function of the NTS $\xi : X \times \Sigma \to 2^X$, and the deterministic transition function of the DA $\tau : L \times Q \to Q$, where $l_i$ is the label of $x_i$, $i \geq 0$. We will discuss more about how these four sequences work jointly later when we talk about how to use the winning set and the winning control strategy constructed in the discrete space to do simulation in the continuous space.

### 6.4.2 Procedure to Solve the Control Problem

We have defined our control problem where the control specification $\varphi$ is an LTL formula. Now it comes to how do we solve the problem?

Consider the NTS and the DA converted from the LTL formula $\varphi$ as two graphs. The agent is initiated at $x_0 \in X$ in the NTS and moves according to the non-deterministic transition function of the NTS $\xi : X \times \Sigma \to 2^X$. The token is initiated at $q_0 \in Q$ in the DA and moves according to the deterministic transition function of the DA $\tau : L \times Q \to Q$. In other words, the agent moves in the NTS according to the trajectory $\rho_x$ and the control inputs $\alpha_\sigma$, whereas the token moves in the DA according to the run $\rho_q$ and the input word $\alpha_l$.

If we consider the NTS and the DA together, the system state $x$ and the automaton state $q$ come in pairs at each time step, i.e., $(x, q)$. We can see that there is a product graph

of NTS and DA hidden implicitly behind. Since we consider the NTS as a bipartite arena, it turns out that we can convert the product NTS to a product bipartite arena $P\mathcal{A}$ as well. By constructing the product NTS explicitly, we enumerate all the possible extensions of the agent, token pair $(x, q)$ initiated from $x_0 \in X$ and $q_0 \in Q$ in the forward direction. Then we convert our control problem as a game $\mathcal{G}$ defined in the bipartite product arena $P\mathcal{A}$ and solve it in the backward direction. After we solve the game $\mathcal{G}$, we map the winning set $\mathcal{W}_1$ and the winning strategy $\Pi_1$ from $P\mathcal{A}$ back to the winning set $\mathcal{W}_0$ and the winning control strategy $\Pi$ in the NTS.

### 6.4.3 Product of NTS and One DA

**How does an NTS work?**

- We consider an NTS as a bipartite arena of system states $X$ and control actions $\Sigma$: an edge can only point from a state $x$ to an action $\sigma$ or from an action $\sigma$ to a state $x'$, where $|\xi(x, \sigma)| > 0$ and $x' \in \xi(x, \sigma)$.
- At a state $x$, we may choose to apply more than one actions. If we choose an action $\sigma$, it will go through one of its non-deterministic transitions $\xi(x, \sigma)$ and go to the next state $x'$, i.e., $x' \in \xi(x, \sigma)$.

**How does a DA work?**

- At an NTS state $x$ and a DA state $q$, we input the label $l$ of $x$ to the deterministic transition function $\tau : L \times Q \to Q$, and go through the transition on which the boolean function $bf : L \to \{0, 1\}$ given the input $l$ returns logic 1 to the next DA state $q'$. We express this process as $\tau\_next(l, q) = q'$.

From how the NTS and the DA work jointly together, we can see there is a hidden product graph behind, and the process of taking product is to explicitly construct this product graph out.

**How to take product?**

- The product states are $(x, q)$ where $x \in X$ and $q \in Q$.
- The initial states of the product are $(x, q_0)$ where $x \in X$ and $q_0 \in Q$ is the unique initial DA state.

- For an NTS transition $(x, \sigma, x')$ given by $x' \in \xi(x, \sigma)$,
- and a DA transition $(q, q')$ derived from $\tau\_next(l, q) = q'$,
- the product transition is given by $((x, q), \sigma, (x', q'))$.

**Definition of Product NTS**

After we have introduced how an NTS and a DA work together as in a product space, now we give a formal definition of the product NTS.

**Definition 6.4.1** (product NTS)**.** *For an $NTS = (X, \Sigma, \xi, L)$ and a $DA = (Q, L, \tau, q_0, Acc)$, the product NTS is a 5-tuple $NTS \otimes DA = (X_P, \Sigma, \xi_P, L, X_0)$, where*

- $X_P = X \times Q$ *is the set of product states;*
- $\Sigma$ *is the set of control actions as in NTS;*
- $\xi_P : X_P \times \Sigma \to 2^{X_P}$ *is a non-deterministic transition function such that* $(x', q') \in \xi_P((x, q), \sigma)$ *iff* $|\xi(x, \sigma)| > 0$*,* $x' \in \xi(x, \sigma)$*, and* $\tau(l, q) = q'$*;*
- $L \subseteq 2^{AP}$ *is the set of labels as in NTS and DA;*
- $X_0 = X \times \{q_0\}$ *is the set of initial product states.*

**Quantification of the Product NTS**

After giving the definition of the product NTS, we will give a quantification of the product NTS based on the quantifiers of the NTS and the DA.

**NTS**

- $n_0 = |X|$ is the number of system states;
- $|\Sigma|$ is the number of control actions;
- $m_0 = \sum_{x \in X} \sum_{\sigma \in \Sigma} |\xi(x, \sigma)|$ is the number of system transitions;
- $|AP|$ is the number of atomic propositions;
- $|L| \leq 2^{|AP|}$ is the number of different labels;

- $|\xi| = \sum_{x \in X} \sum_{\sigma \in \Sigma} \kappa\left(\xi(x, \sigma)\right)$ is the number of non-deterministic transition functions, or the total number of control actions of the system;
- $\bar{\sigma} \approx \dfrac{\text{number of non-deterministic transition functions}}{\text{number of system states}} = \dfrac{|\xi|}{n_0} \leq |\Sigma|$ is the average number of control actions of each system state.

**DA**

- $n_1 = |Q|$ is the number of DA states;
- $m_1 = |\tau|$ is the number of DA transitions.

**Product NTS** $NTS \otimes DA$

- $|X_P| = |X \times Q| = n_0 n_1$ is the number of product states;
- $|\Sigma|$ is the number of control actions;
- $\sum_{(x,q) \in X_P} \sum_{\sigma \in \Sigma} |\xi_P((x,q),\sigma)| = \sum_{x \in X} \sum_{\sigma \in \Sigma} |\xi(x,\sigma)| \cdot |Q| = m_0 n_1$ is the number of product transitions.
- $|AP|$ is the number of atomic propositions;
- $|L| \leq 2^{|AP|}$ is the number of different labels;
- $|\xi_P| = \sum_{(x,q) \in X_P} \sum_{\sigma \in \Sigma} \kappa(\xi_P((x,q),\sigma)) = \sum_{x \in X} \sum_{\sigma \in \Sigma} \kappa\left(\xi(x,\sigma)\right) \cdot |Q| = |\xi| \cdot n_1$ is the number of non-deterministic transition functions, or the total number of control actions of the product NTS;
- $\bar{\sigma} \approx \dfrac{\text{number of non-deterministic transition functions}}{\text{number of product states}} = \dfrac{|\xi_P|}{n_0 n_1} = \dfrac{|\xi| \cdot n_1}{n_0 n_1} = \dfrac{|\xi|}{n_0} \leq |\Sigma|$ is the average number of control actions of each product state;
- $|X_0| = |X \times \{q_0\}| = n_0 \cdot 1 = n_0$ is the number of initial product states.

The two quantifiers that we care about the most are the number of product states $n = n_0 n_1$ and the number of product transitions $m = m_0 n_1$ when we convert the product NTS $NTS \otimes DA$ to a product arena $P\mathcal{A}$. A third quantifier, the average number of control actions of each product state, $\bar{\sigma}$ may be also of interest if we would like a more detailed analysis. Therefore, we will give a brief derivation and discussion of these three quantifiers here.

**Number of Product States:** $n = n_0 n_1$

There are $n_0$ NTS states $x$ and $n_1$ DA states $q$.

Therefore, the number of product states $(x, q)$ is $n = n_0 n_1$.

**Number of Product Transitions:** $m = m_0 n_1$

There are $m_0$ NTS transitions $(x, \sigma, x')$ and $n_1$ DA states $q$.

Each pair of NTS transition $(x, \sigma, x')$ and DA state $q$ corresponds to a product transition $((x, q), \sigma, (x', q'))$, where $x' \in \xi(x, \sigma)$, $\tau(l, q) = q'$ and $l$ is the label of $x$.

Therefore, the number of product transitions $((x, q), \sigma, (x', q'))$ is $m = m_0 n_1$.

Note that the number of product transitions is only related to the number of NTS transitions $m_0$ and the number of DA states $n_1$, but not the number of DA transitions $m_1$. The number of DA transitions $m_1$ is involved in the time complexity to take product, which will discussed later in the implementation chapter.

**Average Number of Control Actions of Each Product State:** $\bar{\sigma} = \dfrac{|\xi|}{n_0}$

There are $|\xi_P| = |\xi| \cdot n_1$ control actions, and $n_0 n_1$ product states in the product NTS. Therefore, the average number of control actions of each product state is

$\bar{\sigma} \approx \dfrac{|\xi_P|}{n_0 n_1} = \dfrac{|\xi| \cdot n_1}{n_0 n_1} = \dfrac{|\xi|}{n_0} \leq |\Sigma|$, which is the same as the average number of control actions of each system state in the NTS. In implementation, $\bar{\sigma}$ of the product NTS is probably smaller than that of the NTS due to the optimizations when taking product. This is another reason that we would like to depict the complexity using only the first two quantifiers.

**Conversion from Product NTS to Product Arena**

After introducing the product NTS, we now give a conversion from the product NTS $NTS \otimes DA$ to the product arena $P\mathcal{A}$ to convert the control problem in NTS to an infinite game in $P\mathcal{A}$. This conversion is very much similar to the conversion from NTS to arena $\mathcal{A}$.

Consider a product NTS $NTS \otimes DA = (X_P, \Sigma, \xi_P, L, X_0)$ and a product arena $P\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$. The conversion of product NTS to product arena $P\mathcal{A}$ is as follows:

- $V_1 = X_P$ is the set of player 1 nodes;
- $V_2 = \{\sigma_{(x,q)} | (x,q) \in X_P, \sigma \in \Sigma, |\xi_P((x,q),\sigma)| > 0\}$ is the set of player 2 nodes;
- $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges constructed by
  $E \cup (V_1 \times V_2) = \{((x,q), \sigma_{(x,q)}) | (x,q) \in X_P, \sigma \in \Sigma, |\xi_P((x,q),\sigma)| > 0\}$ and
  $E \cup (V_2 \times V_1) = \{(\sigma_{(x,q)}, (x',q')) | (x,q) \in X_P, \sigma \in \Sigma, |\xi_P((x,q),\sigma)| > 0,$
  $(x',q') \in \xi_P((x,q),\sigma)\}$.

We can see that the product arena $P\mathcal{A}$ is a bipartite arena, i.e.,
$P\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1))$. The conversion from product NTS to product arena shows that the product NTS can be considered as a product bipartite arena.

**Quantification of Product Arena by Quantifiers of Product NTS**

Now we give a brief quantification of the bipartite product arena $P\mathcal{A}$ converted from the product NTS by the quantifiers of the product NTS.

The quantification of the bipartite product arena $P\mathcal{A}$ converted from the product NTS $NTS \otimes DA$ is given by

- $|V_1| = |X_P| = n_0 n_1$;
- $|V_2| = |E \cup (V_1 \times V_2)| = |\xi_P| \approx \bar{\sigma} n_0 n_1 \leq |\Sigma| n_0 n_1$;
- $|E \cup (V_2 \times V_1)| = \sum_{(x,q) \in X_P} \sum_{\sigma \in \Sigma} |\xi_P((x,q),\sigma)| = m_0 n_1$;
- $|V| = |V_1| + |V_2| = n_0 n_1 + |\xi_P| \approx n_0 n_1 + \bar{\sigma} n_0 n_1 = (1 + \bar{\sigma}) n_0 n_1 \approx \bar{\sigma} n_0 n_1$;
- $|E| = |E \cup (V_1 \times V_2)| + |E \cup (V_2 \times V_1)| = |\xi_P| + m_0 n_1 \approx \bar{\sigma} n_0 n_1 + m_0 n_1 \approx m_0 n_1$.

In complexity analysis, if we use two quantifiers for simplicity, then we quantify $|V| = n = n_0 n_1$, and $|E| = m = m_0 n_1$. We can also add a third quantifier $\bar{\sigma}$ for more details. If we want to take $\bar{\sigma}$ into consideration, then we can simply replace $n_0$ by $\bar{\sigma} n_0$ in the result for space complexity. $\bar{\sigma}$ doesn't affect the time complexity.

The conversion of the accepting condition of DA Acc to the objective set Obj in the game $\mathcal{G} = (P\mathcal{A}, \text{Obj})$, and the the conversion of the accepting condition of the control specification $\varphi$ to the winning condition of the game $\mathcal{G}$ will be discussed separately in each control problem.

### 6.4.4 Multiple Product

In this subsection, we will give a generalization of the product space to multi-product space. The product of NTS and multiple DAs and the product of multiple DAs may be used in solving the control problem when the control specification $\varphi$ can be converted as a conjunction of DBAs and a disjunction of deterministic co-Büchi automaton (DCA)s.

#### Product of NTS and Multiple DAs

**Definition 6.4.2** (multi-product NTS)**.** *For an* $NTS = (X, \Sigma, \xi, L)$ *and* $k$ *DAs* $DA_i = (Q_i, L, \tau_i, q_{i0}, Acc_i)$, $1 \leq i \leq k$, *the multi-product NTS is a 5-tuple* $NTS \otimes DA_1 \otimes \cdots \otimes DA_k = (X_{MP}, \Sigma, \xi_{MP}, L, X_0)$, *where*

- $X_{MP} = X \times Q_1 \times \cdots \times Q_k$ *is the set of multi-product states;*
- $\Sigma$ *is the set of control actions as in NTS;*
- $\xi_{MP} : X_{MP} \times \Sigma \to 2^{X_{MP}}$ *is a non-deterministic transition function such that* $(x', q_1', \cdots, q_k') \in \xi_{MP}((x, q_1, \cdots, q_k), \sigma)$ *iff* $|\xi(x, \sigma)| > 0$, $x' \in \xi(x, \sigma)$, *and* $\tau_i(l, q_i) = q_i'$, $1 \leq i \leq k$;
- $L \subseteq 2^{AP}$ *is the set of labels as in NTS and each DA;*
- $X_0 = X \times \{q_{10}\} \times \cdots \times \{q_{k0}\}$ *is the set of initial multi-product states.*

#### Quantification of the Multi-Product NTS

After giving the definition of the multi-product NTS, we will give a quantification of the multi-product NTS based on the quantifiers of the NTS and the $k$ DAs.

**NTS**

- $n_0 = |X|$ is the number of system states;

- $|\Sigma|$ is the number of control actions;
- $m_0 = \sum_{x \in X} \sum_{\sigma \in \Sigma} |\xi(x, \sigma)|$ is the number of system transitions;
- $|AP|$ is the number of atomic propositions;
- $|L| \leq 2^{|AP|}$ is the number of different labels;
- $|\xi| = \sum_{x \in X} \sum_{\sigma \in \Sigma} \kappa\left(\xi(x, \sigma)\right)$ is the number of non-deterministic transition functions, or the total number of control actions of the system;
- $\bar{\sigma} \approx \dfrac{\text{number of non-deterministic transition functions}}{\text{number of system states}} = \dfrac{|\xi|}{n_0} \leq |\Sigma|$ is the average number of control actions of each system state.

**$k$ DAs**

- $n_i = |Q_i|$ is the number of $DA_i$ states;
- $m_i = |\tau_i|$ is the number of $DA_i$ transitions.

**Multi-Product NTS** $NTS \otimes DA_1 \otimes \cdots \otimes DA_k$

- $|X_{MP}| = |X \times Q_1 \cdots \times Q_k| = \prod_{i=0}^{k} n_i$ is the number of multi-product states;
- $|\Sigma|$ is the number of control actions;
- $\sum_{(x, q_1, \cdots, q_k) \in X_{MP}} \sum_{\sigma \in \Sigma} |\xi_P((x, q_1, \cdots, q_k), \sigma)| = \sum_{x \in X} \sum_{\sigma \in \Sigma} |\xi(x, \sigma)| \cdot \prod_{i=1}^{k} |Q_i| = m_0 \cdot \prod_{i=1}^{k} n_i$ is the number of multi-product transitions.
- $|AP|$ is the number of atomic propositions;
- $|L| \leq 2^{|AP|}$ is the number of different labels;

- $$\begin{aligned} |\xi_{MP}| &= \sum_{(x,q_1,\cdots,q_k)\in X_{MP}} \sum_{\sigma\in\Sigma} \kappa(\xi_{MP}((x,q_1,\cdots,q_k),\sigma)) \\ &= \sum_{x\in X}\sum_{\sigma\in\Sigma} \kappa\left(\xi(x,\sigma)\right)\cdot\prod_{i=1}^{k}|Q_i| = |\xi|\cdot\prod_{i=1}^{k} n_i \end{aligned}$$

  is the number of non-deterministic transition functions, or the total number of control actions of the multi-product NTS;

- $$\begin{aligned} \bar{\sigma} &\approx \frac{\text{number of non-deterministic transition functions}}{\text{number of multi-product states}} \\ &= \frac{|\xi_{MP}|}{\prod_{i=0}^{k} n_i} = \frac{|\xi|\cdot\prod_{i=1}^{k} n_i}{\prod_{i=0}^{k} n_i} = \frac{|\xi|}{n_0} \leq |\Sigma| \end{aligned}$$

  is the average number of control actions of each multi-product state;

- $|X_0| = |X\times\{q_10\}\times\cdots\times\{q_k0\}| = n_0\cdot\prod_{i=1}^{k} 1 = n_0$ is the number of initial multi-product states.

The two quantifiers that we care about the most are the number of multi-product states $n = \prod_{i=0}^{k} n_i$ and the number of multi-product transitions $m = m_0\prod_{i=1}^{k} n_i$ when we convert the multi-product NTS $NTS\otimes DA_1\otimes\cdots\otimes DA_k$ to a multi-product arena $MP\mathcal{A}$. A third quantifier, the average number of control actions of each multi-product state, $\bar{\sigma}$ may be also of interest if we would like a more detailed analysis. Therefore, we will give a brief derivation and discussion of these three quantifiers here.

**Number of Multi-Product States:** $n = \prod_{i=0}^{k} n_i$

There are $n_0$ NTS states $x$ and $n_i$ $DA_i$ states $q_i$, $1\leq i\leq k$.

Therefore, the number of product states $(x, q_1, \cdots, q_k)$ is $n = \prod_{i=0}^{k} n_i$.

**Number of Multi-Product Transitions:** $m = m_0 \prod_{i=1}^{k} n_i$

There are $m_0$ NTS transitions $(x, \sigma, x')$ and $n_i$ $DA_i$ states $q_i$, $1 \leq i \leq k$.

Each copy of NTS transition $(x, \sigma, x')$ and $DA_i$ state $q_i$, $1 \leq i \leq k$ corresponds to a multi-product transition $((x, q_1, \cdots, q_k), \sigma, (x', q'_1, \cdots, q'_k))$ where $x' \in \xi(x, \sigma)$, $\tau_i(l, q_i) = q'_i$ and $l$ is the label of $x$.

Therefore, the number of multi-product transitions $((x, q_1, \cdots, q_k), \sigma, (x', q'_1, \cdots, q'_k))$ is $m = m_0 \prod_{i=1}^{k} n_i$.

Note that the number of multi-product transitions is only related to the number of NTS transitions $m_0$ and the number of $DA_i$ states $n_i$, but not the number of $DA_i$ transitions $m_i$. The number of $DA_i$ transitions $m_i$ is involved in the time complexity to take multi-product, which will discussed later in the implementation chapter.

**Average Number of Control Actions of Each Multi-Product State:** $\bar{\sigma} = \dfrac{|\xi|}{n_0}$

There are $|\xi_{MP}| = |\xi| \cdot \prod_{i=1}^{k} n_i$ control actions, and $\prod_{i=0}^{k} n_i$ multi-product states in the multi-product NTS. Therefore, the average number of control actions of each multi-product state is

$\bar{\sigma} \approx \dfrac{|\xi_{MP}|}{\prod_{i=0}^{k} n_i} = \dfrac{|\xi| \cdot \prod_{i=1}^{k} n_i}{\prod_{i=0}^{k} n_i} = \dfrac{|\xi|}{n_0} \leq |\Sigma|$, which is the same as the average number of control actions of each system state in the NTS. In implementation, $\bar{\sigma}$ of the multi-product NTS is probably smaller than that of the NTS due to the optimizations when taking multi-product. This is another reason that we would like to depict the complexity using only the first two quantifiers.

**Conversion from Multi-Product NTS to Multi-Product Arena**

After introducing the multi-product NTS, we now give a conversion from the multi-product NTS $NTS \otimes DA_1 \otimes \cdots \otimes DA_k$ to the multi-product arena $MP\mathcal{A}$ to convert the

control problem in NTS to an infinite game in $MP\mathcal{A}$. This conversion is also very much similar to the conversion from NTS to arena $\mathcal{A}$.

Consider a multi-product NTS $NTS \otimes DA_1 \otimes \cdots \otimes DA_k = (X_{MP}, \Sigma, \xi_{MP}, L, X_0)$ and a multi-product arena $MP\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq V \times V)$. The conversion of multi-product NTS to multi-product arena $MP\mathcal{A}$ is as follows:

- $V_1 = X_{MP}$ is the set of player 1 nodes;
- $V_2 = \{\sigma_{(x,q_1,\cdots,q_k)} | (x, q_1, \cdots, q_k) \in X_{MP}, \sigma \in \Sigma, |\xi_{MP}((x, q_1, \cdots, q_k), \sigma)| > 0\}$ is the set of player 2 nodes;
- $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges constructed by
  $E \cup (V_1 \times V_2) = \{((x, q_1, \cdots, q_k), \sigma_{(x,q_1,\cdots,q_k)}) | (x, q_1 \cdots, q_k) \in X_{MP},$
  $\sigma \in \Sigma, |\xi_{MP}((x, q_1, \cdots, q_k), \sigma)| > 0\}$ and
  $E \cup (V_2 \times V_1) = \{(\sigma_{(x,q_1\cdots,q_k)}, (x', q'_1, \cdots, q'_k)) | (x, q_1, \cdots, q_k) \in X_{MP},$
  $\sigma \in \Sigma, |\xi_{MP}((x, q_1, \cdots, q_k), \sigma)| > 0, (x', q'_1, \cdots, q'_k) \in \xi_{MP}((x, q_1, \cdots, q_k), \sigma)\}$.

We can see that the multi-product arena $MP\mathcal{A}$ is a bipartite arena, i.e.,
$MP\mathcal{A} = (V = V_1 \oplus V_2, E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1))$. The conversion from multi-product NTS to multi-product arena shows that the multi-product NTS can be considered as a multi-product bipartite arena.

**Quantification of Multi-Product Arena by Quantifiers of Multi-Product NTS**

Now we give a brief quantification of the bipartite multi-product arena $MP\mathcal{A}$ converted from the multi-product NTS by the quantifiers of the multi-product NTS.

The quantification of the bipartite multi-product arena $MP\mathcal{A}$ converted from the multi-product NTS $NTS \otimes DA_1 \otimes \cdots \otimes DA_k$ is given by

- $|V_1| = |X_{MP}| = \prod_{i=0}^{k} n_i$;
- $|V_2| = |E \cup (V_1 \times V_2)| = |\xi_{MP}| \approx \bar{\sigma} \prod_{i=0}^{k} n_i \leq |\Sigma| \prod_{i=0}^{k} n_i$;
- $|E \cup (V_2 \times V_1)| = \sum_{(x,q_1,\cdots,q_k) \in X_{MP}} \sum_{\sigma \in \Sigma} |\xi_{MP}((x, q_1, \cdots, q_k), \sigma)| = m_0 \prod_{i=1}^{k} n_i$;

- $|V| = |V_1| + |V_2| = \prod_{i=0}^{k} n_i + |\xi_{MP}| \approx \prod_{i=0}^{k} n_i + \bar{\sigma} \prod_{i=0}^{k} n_i = (1 + \bar{\sigma}) \prod_{i=0}^{k} n_i \approx \bar{\sigma} \prod_{i=0}^{k} n_i$;
-

$$\begin{aligned} |E| &= |E \cup (V_1 \times V_2)| + |E \cup (V_2 \times V_1)| \\ &= |\xi_{MP}| + m_0 \prod_{i=1}^{k} n_i \approx \bar{\sigma} \prod_{i=0}^{k} n_i + m_0 \prod_{i=1}^{k} n_i \approx m_0 \prod_{i=1}^{k} n_i. \end{aligned}$$

In complexity analysis, if we use two quantifiers for simplicity, then we quantify $|V| = n = \prod_{i=0}^{k} n_i$, and $|E| = m = m_0 \prod_{i=1}^{k} n_i$. We can also add a third quantifier $\bar{\sigma}$ for more details. If we want to take $\bar{\sigma}$ into consideration, then we can simply replace $n_0$ by $\bar{\sigma} n_0$ in the result for space complexity. $\bar{\sigma}$ doesn't affect the time complexity.

The conversion of the accepting condition of $DA_i$ $\text{Acc}_i$ to the objective set Obj in the game $\mathcal{G} = (MP\mathcal{A}, \text{Obj})$, and the the conversion of the accepting condition of the control specification $\varphi$ to the winning condition of the game $\mathcal{G}$ will be discussed separately in each control problem.

### Product of Multiple DAs

We sometimes take the product of multiple DAs with the same accepting condition, i.e., Büchi or co-Büchi, together to construct a multi-product DA when we need to consider these multiple DAs together.

**Definition 6.4.3** (multi-product DA)**.** *For $k$ DAs*
*$DA_i = (Q_i, L, \tau_i, q_{i0}, Acc_i)$, $1 \leq i \leq k$, the multi-product DA is a 5-tuple*
*$DA_1 \otimes \cdots \otimes DA_k = (Q_{MP}, L, \tau_{MP}, q_0, Acc)$, where*

- *$Q_{MP} = Q_1 \times \cdots \times Q_k$ is the set of multi-product DA states;*
- *$L \subseteq 2^{AP}$ is the set of labels as in each DA;*
- *$\tau_{MP} : L \times Q_{MP} \to Q_{MP}$ is a deterministic transition function such that $\tau_{MP}(l, (q_1, \cdots, q_k)) = (q'_1, \cdots, q'_k)$ iff $\tau_i(l, q_i) = q'_i$, $1 \leq i \leq k$;*

- $q_0 = \{q_{10}\} \times \cdots \times \{q_{k0}\}$ *is the unique initial multi-product DA state;*
- $Acc = \{F_1, \cdots, F_k\} \subseteq 2^{Q_{MP}}$, $F_i = Q_1 \times \cdots \times Q_{i-1} \times Acc_i \times Q_{i+1} \times \cdots \times Q_k$ *is the set of final states in a finite automaton or the set of accepting states in an* $\omega$*-automaton.*

**Quantification of the Multi-Product DA**

After giving the definition of the multi-product DA, we will give a quantification of the multi-product DA based on the quantifiers of the $k$ DAs.

**$k$ DAs**

- $n_i = |Q_i|$ is the number of $DA_i$ states;
- $m_i = |\tau_i|$ is the number of $DA_i$ transitions;
- $|q_{i0}| = 1$ is the number of initial $DA_i$ states;
- $1 \leq |\mathrm{Acc}_i| \leq n_i$ is the number of accepting states in $DA_i$.

**Multi-Product DA $DA_1 \otimes \cdots \otimes DA_k$**

- $|Q_{MP}| = |Q_1 \times \cdots \times Q_k| = \prod_{i=1}^{k} n_i$ is the number of multi-product DA states;
- $|\tau_{MP}| = \prod_{i=1}^{k} |\tau_i| = \prod_{i=1}^{k} m_i$ is the number of multi-product DA transitions;
- $|AP|$ is the number of atomic propositions;
- $|L| \leq 2^{|AP|}$ is the number of different labels;
- $|q_0| = |\{q_1 0\} \times \cdots \times \{q_k 0\}| = \prod_{i=1}^{k} 1 = 1$ is the number of initial multi-product states;
- $|F_i| = |Q_1 \times \cdots \times Q_{i-1} \times \mathrm{Acc}_i \times Q_{i+1} \times \cdots \times Q_k| = \prod_{j=1}^{k} n_j \cdot \frac{|\mathrm{Acc}_i|}{n_i}$ is the number of the $i^{th}$ accepting set of multi-product DA.

## 6.5 Detailed Analysis of Each Control Specification

We will classify the control specifications as reachability, safety, Büchi, co-Büchi, generalized Büchi, generalized co-Büchi, and Rabin.

In each case, we will also discuss the two methods classified by the control specification: defining by the NTS states and as an LTL formula. The latter is the generalization of the former. For each of the two methods, we will give the conversion of the accepting condition of a trajectory $\rho_x$ of the control specification $\varphi$ to the winning condition of a play $\pi_v$ of the infinite game $\mathcal{G} = (\mathcal{A}, \mathrm{Obj})$. We will then use the conclusion from the previous chapter to give a complexity analysis.

Briefly, the target set for the NTS and the arena $\mathcal{A}$ is the same.

The safe set of $\mathcal{A}$ consists of the safe set of NTS and its control actions. However, when computing time complexity, we can only consider the safe set in NTS which are the $V_1$ nodes in $\mathcal{G}$. We don't need to consider the control actions of the safe set of NTS which are the $V_2$ nodes in $\mathcal{G}$. The reason is that, the $V_2$ nodes can't form an "$SCC$" by itself, thus the safe set computed after each safe$(\cdot, \cdot)$ must contain at least 1 $V_1$ node if not empty.

### 6.5.1 Reachability

In this subsection, we will introduce the control specification classified as reachability in NTS.

The generalization of a reachability specification is the LTL formulas that can be converted to a deterministic finite automaton (DFA).

#### Control Specification Defined by States in NTS

We convert the control problem with a reachability control specification in NTS to a reachability game in $\mathcal{A}$.

#### Control Specification

For a reachability control specification $\varphi$, $T \subseteq X$ is the set of target states labeled with $a$. A finite trajectory $r_x$ satisfies $\varphi$ iff one of the states in $T$ occurs in $r_x$. Formally, $r_x$ satisfies $\varphi$ iff $\mathrm{Occ}(r_x) \cap T \neq \emptyset$.

#### Conversion to a Reachability Game

Consider a finite reachability game $\mathcal{G} = (\mathcal{A}, \text{Obj})$. $\mathcal{A} = (V, E)$ is the bipartite arena converted from NTS, where $V = V_1 \oplus V_2$ is the set of nodes and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = T \subseteq V_1$ is the set of target nodes. A finite play $p_v$ wins the reachability game iff one of the nodes in $T$ occurs in $p_v$. Formally, $p_v$ wins iff $\text{Occ}(p_v) \cap T \neq \emptyset$.

**Complexity Analysis**

In a reachability game $\mathcal{G} = (\mathcal{A}, \text{Obj})$, $\mathcal{A} = (V, E)$ is the arena and $\text{Obj} = T \subseteq V_1$ is the objective set. $|V| = n_0$ is the number of nodes and $|E| = m_0$ is the number of edges. We have the following results according to proposition 5.2.1 from the reachability game.

**Space Complexity:** $\mathcal{O}(n_0 + m_0) = \mathcal{O}(m_0)$.

**Time Complexity:** $\mathcal{O}(n_0 + m_0) = \mathcal{O}(m_0)$.

**Number of Control Actions on Each Winning State:** 1.

**Control Specification Defined as an LTL Formula**

We convert the control problem with a DFA translatable LTL formula as control specification in NTS to a reachability game in $P\mathcal{A}$.

For a reachability control specification $\varphi$, we convert $\varphi$ to a DFA. $\varphi = Fa$ is a characteristic LTL formula that can be converted to a DFA, where $a$ is the label of the set of target states $T \subseteq X$. The picture of DFA is shown in Figure 6.2.

We can also convert the finite reachability game to an infinite reachability game by converting the finite run $r_q$ to an infinite run $\rho_q$, and converting the DFA to a DBA or DCA by adding a self-loop with logic 1 for $q_1$. The picture of converting from DFA to DBA or DCA is also shown in Figure 6.2.

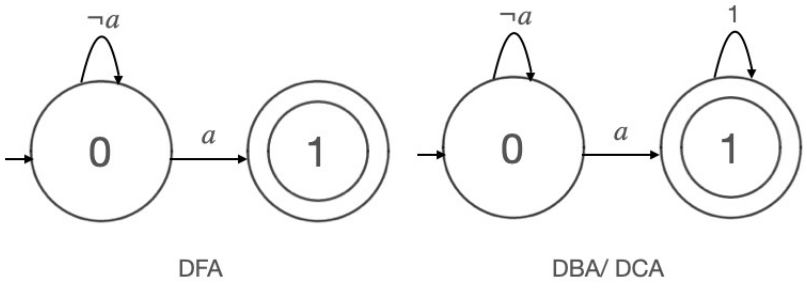


Figure 6.2: DFA and DBA/ DCA of $\varphi = Fa$, with initial state $q_0$ and final or accepting state $F = \{q_1\}$

For a $DFA = (Q, L, \tau, q_0, \text{Acc})$, the accepting condition $\text{Acc} = F \subseteq Q$ is the set of final states. A finite run $r_q$ is accepted by DFA iff one of the states in $F$ occurs in $r_q$. Formally, $r_q$ is accepting iff $\text{Occ}(r_q) \cap F \neq \emptyset$.

**Satisfaction of LTL**

A finite trajectory $r_x$ satisfies $\varphi$ iff its finite input word $w_l$ satisfies the accepting condition of DFA converted from $\varphi$.

**Conversion to a Reachability Game**

In the product NTS $NTS \otimes DA$, $T = \{(x, q) | x \in X, q \in F\} \subseteq X_P$ is the set of target states.

Consider a finite reachability game $\mathcal{G} = (P\mathcal{A}, \text{Obj})$. $P\mathcal{A} = (V, E)$ is the bipartite product arena converted from product NTS, where $V = V_1 \oplus V_2$ is the set of nodes and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = T \subseteq V_1$ is the set of target nodes. A finite play $p_v$ wins the reachability game iff one of the nodes in $T$ occurs in $p_v$. Formally, $p_v$ wins iff $\text{Occ}(p_v) \cap T \neq \emptyset$.

**Complexity Analysis**

In a reachability game $\mathcal{G} = (P\mathcal{A}, \text{Obj})$, $P\mathcal{A} = (V, E)$ is the product arena and

$\text{Obj} = T \subseteq V_1$ is the objective set. $|V| = n = n_0 n_1$ is the number of nodes and

$|E| = m = m_0 n_1$ is the number of edges. We have the following results according to proposition 5.2.1 from the reachability game.

**Space Complexity:** $\mathcal{O}(n + m) = \mathcal{O}(n_0 n_1 + m_0 n_1) = \mathcal{O}(m_0 n_1)$.

**Time Complexity:** $\mathcal{O}(n + m) = \mathcal{O}(n_0 n_1 + m_0 n_1) = \mathcal{O}(m_0 n_1)$.

**Number of Control Actions on Each Winning State:** 1.

Specifically, when $\varphi = Fa$, the DFA has $n_1 = 2$ and $m_1 = 2$ as shown in Figure 6.2. The control problem can be solved in $\mathcal{O}(2m_0)$ space and time. This shows that solving a reachability problem in NTS directly is more efficient than taking product.

### 6.5.2 Safety

In this subsection, we will introduce the control specification classified as safety in NTS.

This is a special problem in NTS. It's the dual problem of a reachability problem, and the first infinite specification that we consider. When we express the safety specification

as an LTL formula $\varphi = Ga$, it can be converted to either a DBA or a DCA. The converted DA has only 1 liveness state, thus can be solved in the NTS directly. The picture of DBA/ DCA converted from $\varphi = Ga$ is shown in Figure 6.3.

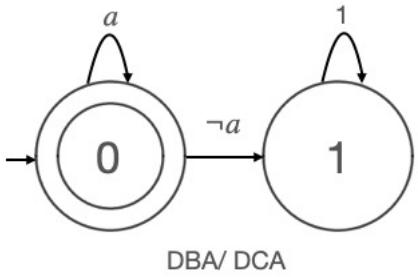


Figure 6.3: DBA/ DCA of $\varphi = Ga$, with initial state $q_0$ and accepting state $F = \{q_0\}$

An important use of the safety specification is that, when we consider an infinite specification, we assume that each node in the arena has positive out-degree to guarantee an infinite play. However, this assumption may not hold for our NTS computed in the abstraction for efficiency. Therefore, we need to consider the obstacles and the boundary as the unsafe nodes and solve a safety game to resolve this. We prefer to leave this step when we solve the control problem.

When we need to take the product to solve a control problem, we also solve a safety game before taking the product. After taking the product, we solve a safety game again before solving the actual problem in $P\mathcal{A}$ for efficiency as a safety game can be solved in linear time.

### Control Specification Defined by States in NTS

We convert the control problem with a safety control specification in NTS to a safety game in $\mathcal{A}$.

#### Control Specification

For a safety control specification $\varphi$, $S \subseteq X$ is the set of safe states labeled with $a$ and correspondingly, $X/S$ is the set of unsafe states labeled with $\neg a$. An infinite trajectory $\rho_x$ satisfies $\varphi$ iff only the states in $S$ occur in $\rho_x$, or equivalently, the states in $X/S$ never occur in $\rho_x$. Formally, $\rho_x$ satisfies $\varphi$ iff $\mathrm{Occ}(\rho_x) \subseteq S$, or equivalently, $\mathrm{Occ}(\rho_x) \cap (X/S) \neq \emptyset$.

#### Conversion to a Safety Game

Consider an infinite safety game $\mathcal{G} = (\mathcal{A}, \text{Obj})$. $\mathcal{A} = (V, E)$ is the bipartite arena converted from NTS, where $V = V_1 \oplus V_2$ is the set of nodes and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = S \subseteq V$ is the set of safe nodes consisting of the safe set $S$ in the NTS and its control actions. An infinite play $\pi_v$ wins the safety game iff only the nodes in $S$ occur in $\pi_v$, or equivalently, the nodes in $V/S$ never occur in $\pi_v$. Formally, $\pi_v$ wins iff $\text{Occ}(\pi_v) \subseteq S$, or equivalently, $\text{Occ}(\pi_v) \cap (V/S) \neq \emptyset$.

**Complexity Analysis**

In a safety game $\mathcal{G} = (\mathcal{A}, \text{Obj})$, $\mathcal{A} = (V, E)$ is the arena and $\text{Obj} = S \subseteq V$ is the objective set. $|V| = n_0$ is the number of nodes and $|E| = m_0$ is the number of edges. We have the following results according to proposition 5.2.2 from the safety game.

**Space Complexity:** $\mathcal{O}(n_0 + m_0) = \mathcal{O}(m_0)$.

**Time Complexity:** $\mathcal{O}(n_0 + m_0) = \mathcal{O}(m_0)$.

**Number of Control Actions on Each Winning State:** 1.

### 6.5.3 Büchi

In this subsection, we will introduce the control specification classified as Büchi in NTS.

The generalization of a Büchi specification is the LTL formulas that can be converted to a DBA.

**Control Specification Defined by States in NTS**

We convert the control problem with a Büchi control specification in NTS to a Büchi game in $\mathcal{A}$.

**Control Specification**

For a Büchi control specification $\varphi$, $T \subseteq X$ is the set of target states labeled with $a$. An infinite trajectory $\rho_x$ satisfies $\varphi$ iff one of the states in $T$ occurs infinitely many often in $\rho_x$. Formally, $\rho_x$ satisfies $\varphi$ iff $\text{Inf}(\rho_x) \cap T \neq \emptyset$.

**Conversion to a Büchi Game**

Consider an infinite Büchi game $B\mathcal{G} = (\mathcal{A}, \text{Obj})$. $\mathcal{A} = (V, E)$ is the bipartite arena converted from NTS, where $V = V_1 \oplus V_2$ is the set of nodes and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = T \subseteq V_1$ is the set of target nodes. An infinite play $\pi_v$ wins the

Büchi game iff one of the nodes in $T$ occurs infinitely many often in $\pi_v$. Formally, $\pi_v$ wins iff $\text{Inf}(\pi_v) \cap T \neq \emptyset$.

**Complexity Analysis**

In a Büchi game $B\mathcal{G} = (\mathcal{A}, \text{Obj})$, $\mathcal{A} = (V, E)$ is the arena and $\text{Obj} = T \subseteq V_1$ is the objective set. $|V| = n_0$ is the number of nodes and $|E| = m_0$ is the number of edges. We have the following results according to proposition 5.3.1 from the Büchi game.

**Space Complexity:** $\mathcal{O}(n_0 + m_0) = \mathcal{O}(m_0)$.

**Time Complexity:** $\mathcal{O}(n_0 m_0)$.

**Number of Control Actions on Each Winning State:** 1.

**Control Specification Defined as an LTL Formula**

We convert the control problem with a DBA translatable LTL formula as control specification in NTS to a Büchi game in $P\mathcal{A}$.

For a Büchi control specification $\varphi$, we convert $\varphi$ to a DBA. $\varphi = GFa$ is a characteristic LTL formula that can be converted to a DBA, where $a$ is the label of the set of target states $T \subseteq X$. The picture of DBA is shown in Figure 6.4.

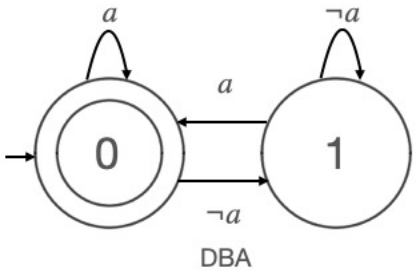


Figure 6.4: DBA of $\varphi = GFa$, with initial state $q_0$ and accepting state $F = \{q_0\}$

For a $DBA = (Q, L, \tau, q_0, \text{Acc})$, the accepting condition $\text{Acc} = F \subseteq Q$ is the set of accepting states. An infinite run $\rho_q$ is accepted by DBA iff one of the states in $F$ occurs infinitely many often in $\rho_q$. Formally, $\rho_q$ is accepting iff $\text{Inf}(\rho_q) \cap F \neq \emptyset$.

**Satisfaction of LTL**

An infinite trajectory $\rho_x$ satisfies $\varphi$ iff its infinite input word $\alpha_l$ satisfies the accepting condition of DBA converted from $\varphi$.

**Conversion to a Büchi Game**

In the product NTS $NTS \otimes DA$, $T = \{(x, q) | x \in X, q \in F\} \subseteq X_P$ is the set of target states.

Consider an infinite Büchi game $B\mathcal{G} = (P\mathcal{A}, \text{Obj})$. $P\mathcal{A} = (V, E)$ is the bipartite product arena converted from product NTS, where $V = V_1 \oplus V_2$ is the set of nodes and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = T \subseteq V_1$ is the set of target nodes. An infinite play $\pi_v$ wins the Büchi game iff one of the nodes in $T$ occurs infinitely many often in $\pi_v$. Formally, $\pi_v$ wins iff $\text{Inf}(\pi_v) \cap T \neq \emptyset$.

**Complexity Analysis**

In a Büchi game $B\mathcal{G} = (P\mathcal{A}, \text{Obj})$, $P\mathcal{A} = (V, E)$ is the product arena and $\text{Obj} = T \subseteq V_1$ is the objective set. $|V| = n = n_0 n_1$ is the number of nodes and $|E| = m = m_0 n_1$ is the number of edges. $|T| = n_0 \cdot |F|$ is the number of target nodes.

We have the following results according to proposition 5.3.1 from the Büchi game.

**Space Complexity:** $\mathcal{O}(n + m) = \mathcal{O}(n_0 n_1 + m_0 n_1) = \mathcal{O}(m_0 n_1)$.

**Time Complexity:** $\mathcal{O}(nm) = \mathcal{O}(n_0 n_1 \cdot m_0 n_1) = \mathcal{O}(n_0 m_0 n_1^2)$.

**Number of Control Actions on Each Winning State:** 1.

Specifically, when $\varphi = GFa$, the DBA has $n_1 = 2$, $m_1 = 4$ and $|F| = 1$ as shown in Figure 6.4. The control problem can be solved in $\mathcal{O}(2m_0)$ space and $\mathcal{O}(|T| \cdot m) = \mathcal{O}(n_0 \cdot 2m_0) = \mathcal{O}(2n_0 m_0)$ time. This shows that solving a Büchi problem in NTS directly is more efficient than taking product.

### 6.5.4 co-Büchi

In this subsection, we will introduce the control specification classified as co-Büchi in NTS.

The generalization of a co-Büchi specification0 is the LTL formulas that can be converted to a DCA. An LTL formula $\varphi$ can be converted to a DCA if $\neg\varphi$ can be converted to a DBA. To convert the DCA for $\varphi$ from the DBA for $\neg\varphi$, we simply consider the accepting set of DBA as the rejecting set of DCA.

**Control Specification Defined by States in NTS**

We convert the control problem with a co-Büchi control specification in NTS to a co-Büchi game in $\mathcal{A}$.

### Control Specification

For a co-Büchi control specification $\varphi$, $S \subseteq X$ is the set of safe states labeled with $a$ and correspondingly, $X/S$ is the set of unsafe states labeled with $\neg a$. An infinite trajectory $\rho_x$ satisfies $\varphi$ iff only the states in $S$ occur in $\rho_x$ eventually, or equivalently, the states in $X/S$ occur only finitely many often in $\rho_x$. Formally, $\rho_x$ satisfies $\varphi$ iff $\mathrm{Inf}(\rho_x) \subseteq S$, or equivalently, $\mathrm{Inf}(\rho_x) \cap (X/S) = \emptyset$.

### Conversion to a co-Büchi Game

Consider an infinite co-Büchi game $C\mathcal{G} = (\mathcal{A}, \mathrm{Obj})$. $\mathcal{A} = (V, E)$ is the bipartite arena converted from NTS, where $V = V_1 \oplus V_2$ is the set of nodes and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\mathrm{Obj} = S \subseteq V$ is the set of safe nodes consisting of the safe set $S$ in the NTS and its control actions. An infinite play $\pi_v$ wins the co-Büchi game iff only the nodes in $S$ occur in $\pi_v$ eventually, or equivalently, the nodes in $X/S$ occur only finitely many often in $\pi_v$. Formally, $\pi_v$ wins iff $\mathrm{Inf}(\pi_v) \subseteq S$, or equivalently, $\mathrm{Inf}(\pi_v) \cap (V/S) = \emptyset$.

### Complexity Analysis

In a co-Büchi game $C\mathcal{G} = (\mathcal{A}, \mathrm{Obj})$, $\mathcal{A} = (V, E)$ is the arena and $\mathrm{Obj} = S \subseteq V$ is the objective set. $|V| = n_0$ is the number of nodes and $|E| = m_0$ is the number of edges. We can only consider the $V_1$ nodes in $S$ when computing the time complexity. We have the following results according to proposition 5.3.2 from the co-Büchi game.

**Space Complexity:** $\mathcal{O}(n_0 + m_0) = \mathcal{O}(m_0)$.

**Time Complexity:** $\mathcal{O}(n_0 m_0)$.

**Number of Control Actions on Each Winning State:** 1.

### Control Specification Defined as an LTL Formula

We convert the control problem with a DCA translatable LTL formula as control specification in NTS to a co-Büchi game in $P\mathcal{A}$.

For a co-Büchi control specification $\varphi$, we convert $\varphi$ to a DCA. $\varphi = FGa$ is a characteristic LTL formula that can be converted to a DCA, where $a$ is the label of the set of safe states $S \subseteq X$. The picture of DCA is shown in Figure 6.5.

For a $DCA = (Q, L, \tau, q_0, \mathrm{Acc})$, the accepting condition $\mathrm{Acc} = F \subseteq Q$ is the set of accepting states. An infinite run $\rho_q$ is accepted by DCA iff only the states in $F$ occur in $\rho_q$ eventually. Formally, $\rho_q$ is accepting iff $\mathrm{Inf}(\rho_q) \subseteq F$.

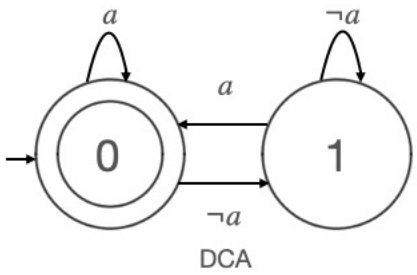


Figure 6.5: DCA of $\varphi = FGa$, with initial state $q_0$ and accepting state $F = \{q_0\}$

Equivalently, we can also define the accepting condition $\text{Acc} = (Q/F) \subseteq Q$ as the set of rejecting states. An infinite run $\rho_q$ is accepted by DCA iff the states in $Q/F$ occur only finitely many often in $\rho_q$. Formally, $\rho_q$ is accepting iff $\text{Inf}(\rho_q) \cap (Q/F) = \emptyset$.

**Satisfaction of LTL**

An infinite trajectory $\rho_x$ satisfies $\varphi$ iff its infinite input word $\alpha_l$ satisfies the accepting condition of DCA converted from $\varphi$.

**Conversion to a co-Büchi Game**

In the product NTS $NTS \otimes DA$, $S = \{(x, q) | x \in X, q \in F\} \subseteq X_P$ is the set of safe states.

Consider an infinite co-Büchi game $C\mathcal{G} = (P\mathcal{A}, \text{Obj})$. $P\mathcal{A} = (V, E)$ is the bipartite product arena converted from product NTS, where $V = V_1 \oplus V_2$ is the set of nodes and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = S \subseteq V$ is the set of safe nodes consisting of the safe set $S$ in the product NTS and its control actions. An infinite play $\pi_v$ wins the co-Büchi game iff only the nodes in $S$ occur in $\pi_v$ eventually, or equivalently, the nodes in $V/S$ occur only finitely many often in $\pi_v$. Formally, $\pi_v$ wins iff $\text{Inf}(\pi_v) \subseteq S$, or equivalently, $\text{Inf}(\pi_v) \cap (V/S) = \emptyset$.

**Complexity Analysis**

In a co-Büchi game $C\mathcal{G} = (P\mathcal{A}, \text{Obj})$, $P\mathcal{A} = (V, E)$ is the product arena and

$\text{Obj} = S \subseteq V$ is the objective set. $|V| = n = n_0 n_1$ is the number of nodes and

$|E| = m = m_0 n_1$ is the number of edges. We can only consider the $V_1$ nodes in $S$ when computing the time complexity, thus $|S| = n_0 \cdot |F|$ is the number of safe nodes.

We have the following results according to proposition 5.3.2 from the co-Büchi game.

**Space Complexity:** $\mathcal{O}(n + m) = \mathcal{O}(n_0 n_1 + m_0 n_1) = \mathcal{O}(m_0 n_1)$.

**Time Complexity:** $\mathcal{O}(nm) = \mathcal{O}(n_0 n_1 \cdot m_0 n_1) = \mathcal{O}(n_0 m_0 n_1^2)$.

**Number of Control Actions on Each Winning State:** 1.

Specifically, when $\varphi = FGa$, the DCA has $n_1 = 2$, $m_1 = 4$ and $|F| = 1$ as shown in Figure 6.5. The control problem can be solved in $\mathcal{O}(2m_0)$ space and

$\mathcal{O}(|S| \cdot m) = \mathcal{O}(n_0 \cdot 2m_0) = \mathcal{O}(2n_0 m_0)$ time. This shows that solving a co-Büchi problem in NTS directly is more efficient than taking product.

### 6.5.5 Generalized Büchi

In this subsection, we will introduce the control specification classified as generalized Büchi in NTS.

The generalization of a generalized Büchi specification is the LTL formulas that can be converted to a generalized deterministic Büchi automaton (GDBA).

#### Control Specification Defined by States in NTS

We convert the control problem with a generalized Büchi control specification in NTS to a generalized Büchi game in $\mathcal{A}$.

**Control Specification**

For a generalized Büchi control specification $\varphi$, $T_i \subseteq X$, $1 \leq i \leq k$ are the $k$ sets of target states labeled with $a_i$. An infinite trajectory $\rho_x$ satisfies $\varphi$ iff one of the states in $T_i$ occurs infinitely many often in $\rho_x$ for all $i$, $1 \leq i \leq k$. Formally, $\rho_x$ satisfies $\varphi$ iff $\bigwedge_{i=1}^{k} (\text{Inf}(\rho_x) \cap T_i \neq \emptyset)$.

**Conversion to a generalized Büchi Game**

Consider an infinite generalized Büchi game $GB\mathcal{G} = (\mathcal{A}, \text{Obj})$. $\mathcal{A} = (V, E)$ is the bipartite arena converted from NTS, where $V = V_1 \oplus V_2$ is the set of nodes and

$E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = \{T_1, \cdots, T_k\}$, where $T_i \subseteq V_1$, $1 \leq i \leq k$ are the $k$ sets of target nodes. An infinite play $\pi_v$ wins the generalized Büchi game iff one of the nodes in $T_i$ occurs infinitely many often in $\pi_v$ for all $i$, $1 \leq i \leq k$. Formally, $\pi_v$ wins iff $\bigwedge_{i=1}^{k} (\text{Inf}(\pi_v) \cap T_i \neq \emptyset)$.

### Complexity Analysis

In a generalized Büchi game $GB\mathcal{G} = (\mathcal{A}, \text{Obj})$, $\mathcal{A} = (V, E)$ is the arena and $\text{Obj} = \{T_1 \cdots, T_k\}$, where $T_i \subseteq V_1$, $1 \leq i \leq k$ is the objective set. $|V| = n_0$ is the number of nodes and $|E| = m_0$ is the number of edges. Here we give the complexity analysis according to the three methods given in the generalized Büchi game.

- **Method 1**

  We have the following results according to proposition 5.4.3.

  **Space Complexity:** $\mathcal{O}(n_0 + m_0) = \mathcal{O}(m_0)$.

  **Time Complexity:** $\mathcal{O}(kn_0m_0)$.

  **Number of Control Actions on Each Winning State:** $k$.

- **Method 2**

  We have the following results according to proposition 5.4.1.

  **Space Complexity:** $\mathcal{O}(k(n_0 + m_0)) = \mathcal{O}(km_0)$.

  **Time Complexity:** $\mathcal{O}(kn_0m_0)$.

  **Number of Control Actions on Each Winning State:** 1.

- **Method 3**

  We have the following results according to proposition 5.4.2.

  **Space Complexity:** $\mathcal{O}((k + 1)(n_0 + m_0)) = \mathcal{O}((k + 1)m_0)$.

  **Time Complexity:** $\mathcal{O}((k + 1)n_0m_0)$.

  **Number of Control Actions on Each Winning State:** 1.

### Control Specification Defined as an LTL Formula

We convert the control problem with a GDBA translatable LTL formula as control specification in NTS to a generalized Büchi game in $MP\mathcal{A}$.

For a generalized Büchi control specification $\varphi = \bigwedge_{i=1}^{k} \varphi_i$, where each $\varphi_i$, $1 \leq i \leq k$ can be converted to a DBA, we convert $\varphi$ to a GDBA.

$\varphi = \bigwedge_{i=1}^{k} GFa_i$, where $a_i$ is the label of the set of target states $T_i \subseteq X$, $1 \leq i \leq k$ is the naive generalized Büchi objective. We prefer to solve it as a generalized Büchi game in $\mathcal{A}$ directly. It is even more efficient to convert it to a DBA than to a GDBA.

For a $GDBA = (Q, L, \tau, q_0, \text{Acc})$, the accepting condition $\text{Acc} = \{F_1, \cdots, F_k\}$, where $F_i \subseteq Q$, $1 \leq i \leq k$ are the $k$ sets of accepting states. An infinite run $\rho_q$ is accepted by GDBA iff one of the states in $F_i$ occurs infinitely many often in $\rho_q$ for all $i$, $1 \leq i \leq k$. Formally, $\rho_q$ is accepting iff $\bigwedge_{i=1}^{k} (\text{Inf}(\rho_q) \cap F_i \neq \emptyset)$.

**Satisfaction of LTL**

An infinite trajectory $\rho_x$ satisfies $\varphi$ iff its infinite input word $\alpha_l$ satisfies the accepting condition of GDBA converted from $\varphi$.

**Conversion to a Generalized Büchi Game**

Consider the generalized Büchi control specification $\varphi = \bigwedge_{i=1}^{k} \varphi_i$, $1 \leq i \leq k$, where each $\varphi_i$ can be converted to a DBA $DA_i$ with $\text{Acc}_i$ being the accepting set. Conceptually, there are two ways to construct the multi-product NTS without considering the implementations.

1. Construct a multi-product NTS with the NTS and the $k$ DAs.

   $T_i = \{(x, q_1, \cdots, q_k) | x \in X, q_i \in \text{Acc}_i, q_j \in Q_j, j \neq i\} \subseteq X_{MP}$, $1 \leq i \leq k$ are the $k$ sets of target states in the multi-product NTS.

2. Construct a multi-product DA with the $k$ DAs and then take the product of NTS and multi-product DA to construct a multi-product NTS.

   $\text{Acc} = \{F_1, \cdots, F_k\}$, where $F_i = \{(q_1, \cdots, q_k) | q_i \in \text{Acc}_i, q_j \in Q_j, j \neq i\} \subseteq Q_{MP}$, $1 \leq i \leq k$ are the $k$ sets of accepting states in the multi-product DA.

   $T_i = \{(x, q_1, \cdots, q_k) | x \in X, q_i \in \text{Acc}_i, q_j \in Q_j, j \neq i\} \subseteq X_{MP}$, $1 \leq i \leq k$ are the $k$ sets of target states in the multi-product NTS.

In the implementation, we may encode the $k$ DA dimensions into 1 to reduce the multi-product NTS into a product NTS. There are several ways of constructing the multi-product NTS in implementation, which will be discussed in the implementation chapter later.

Consider an infinite generalized Büchi game $GB\mathcal{G} = (MP\mathcal{A}, \text{Obj})$. $MP\mathcal{A} = (V, E)$ is the bipartite multi-product arena converted from multi-product NTS, where $V = V_1 \oplus V_2$

is the set of nodes and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = \{T_1, \cdots, T_k\}$, where $T_i \subseteq V_1$, $1 \leq i \leq k$ are the $k$ sets of target nodes. An infinite play $\pi_v$ wins the generalized Büchi game iff one of the nodes in $T_i$ occurs infinitely many often in $\pi_v$ for all $i$, $1 \leq i \leq k$. Formally, $\pi_v$ wins iff $\bigwedge_{i=1}^{k}(\text{Inf}(\pi_v) \cap T_i \neq \emptyset)$.

**Complexity Analysis**

In a generalized Büchi game $GB\mathcal{G} = (MP\mathcal{A}, \text{Obj})$, $MP\mathcal{A} = (V, E)$ is the multi-product arena and $\text{Obj} = \{T_1 \cdots, T_k\}$, where $T_i \subseteq V_1$, $1 \leq i \leq k$ is the objective set.

$|V| = n = \prod_{i=0}^{k} n_i$ is the number of nodes and $|E| = m = m_0 \prod_{i=1}^{k} n_i$ is the number of edges.

$|T_i| = n \cdot \frac{|\text{Acc}_i|}{n_i} = \frac{\prod_{j=0}^{k} n_j \cdot |\text{Acc}_i|}{n_i}$, $1 \leq i \leq k$ is the number of target nodes for $T_i$.

Here we give the complexity analysis according to the three methods given in the generalized Büchi game.

- **Method 1**

  We have the following results according to proposition 5.4.3.

  **Space Complexity:** $\mathcal{O}(n+m) = \mathcal{O}(m) = \mathcal{O}\left(m_0 \prod_{i=1}^{k} n_i\right)$.

  **Time Complexity:** $\mathcal{O}(knm) = \mathcal{O}\left(k \cdot \prod_{i=0}^{k} n_i \cdot m_0 \prod_{i=1}^{k} n_i\right) = \mathcal{O}\left(kn_0m_0(\prod_{i=1}^{k} n_i)^2\right)$.

  **Number of Control Actions on Each Winning State:** $k$.

- **Method 2**

  We have the following results according to proposition 5.4.1.

  **Space Complexity:** $\mathcal{O}(k(n+m)) = \mathcal{O}(km) = \mathcal{O}\left(km_0 \prod_{i=1}^{k} n_i\right)$.

  **Time Complexity:** $\mathcal{O}(knm) = \mathcal{O}\left(k \cdot \prod_{i=0}^{k} n_i \cdot m_0 \prod_{i=1}^{k} n_i\right) = \mathcal{O}\left(kn_0m_0(\prod_{i=1}^{k} n_i)^2\right)$.

  **Number of Control Actions on Each Winning State:** 1.

- **Method 3**

  We have the following results according to proposition 5.4.2.

  **Space Complexity:** $\mathcal{O}((k+1)(n+m)) = \mathcal{O}((k+1)m) = \mathcal{O}\left((k+1)m_0\prod_{i=1}^{k} n_i\right)$.

  **Time Complexity:**

$$\begin{aligned}\mathcal{O}((k+1)nm) &= \mathcal{O}\left((k+1)\cdot\prod_{i=0}^{k} n_i \cdot m_0\prod_{i=1}^{k} n_i\right)\\ &= \mathcal{O}\left((k+1)n_0m_0(\prod_{i=1}^{k} n_i)^2\right).\end{aligned}$$

  **Number of Control Actions on Each Winning State:** 1.

**Most General Form of the Generalized Büchi Control Specification**

The most general form of the generalized Büchi control specification is given by $\varphi = \bigwedge_{i=1}^{k_1}\varphi_i \wedge \bigwedge_{j=1}^{k_2} GFa_j$, where $\varphi_i$, $1 \leq i \leq k_1$ is a DBA translatable LTL formula and $a_j$, $1 \leq j \leq k_2$ is the label of the set of target states $T_j \subseteq X$.

The most efficient way of solving this control specification $\varphi$ in NTS is to construct a multi-product NTS with the NTS and the $k_1$ DAs, convert the multi-product NTS to a multi-product arena, and then solve a generalized Büchi game with $k_1+k_2$ Büchi objectives in the multi-product arena. The idea is to solve it as a generalized Büchi game in the smallest possible arena.

$T_i = \{(x, q_1, \cdots, q_{k_1}) | x \in X, q_i \in \text{Acc}_i, q_j \in Q_j, j \neq i\} \subseteq X_{MP}$, $1 \leq i \leq k_1$ are the $k_1$ sets of target states in the multi-product NTS given by $\varphi_i$.

$\hat{T}_j = \{(x, q_1, \cdots, q_{k_1}) | x \in T_j, q_i \in Q_i\} \subseteq X_{MP}$, $1 \leq j \leq k_2$ are the extra $k_2$ sets of target states in the multi-product NTS given by $a_j$.

In a generalized Büchi game $GB\mathcal{G} = (MP\mathcal{A}, \text{Obj})$, $MP\mathcal{A} = (V, E)$ is the multi-product arena and $\text{Obj} = \{T_1 \cdots, T_{k_1}, \hat{T}_1, \cdots, \hat{T}_{k_2}\}$, where $T_i, \hat{T}_j \subseteq V_1$, $1 \leq i \leq k_1$, and $1 \leq j \leq k_2$ is the objective set. $|V| = n = \prod_{i=0}^{k_1} n_i$ is the number of nodes and $|E| = m = m_0\prod_{i=1}^{k_1} n_i$ is

the number of edges. $|T_i| = n \cdot \frac{|\text{Acc}_i|}{n_i} = \frac{\prod_{j=0}^{k_1} n_j \cdot |\text{Acc}_i|}{n_i}$, $1 \leq i \leq k_1$ is the number of target nodes for $T_i$.

$|\hat{T}_j| = n \cdot \frac{|T_j|}{n_0} = \prod_{i=1}^{k} n_i \cdot |T_j|$, $1 \leq j \leq k_2$ is the number of target nodes for $\hat{T}_j$.

We have the following results according to proposition 5.4.3 from method 1 of generalized Büchi game.

**Space Complexity:** $\mathcal{O}(n+m) = \mathcal{O}(m) = \mathcal{O}\left(m_0 \prod_{i=1}^{k_1} n_i\right)$.

**Time Complexity:**

$$\begin{aligned} \mathcal{O}((k_1+k_2)nm) &= \mathcal{O}\left((k_1+k_2) \cdot \prod_{i=0}^{k_1} n_i \cdot m_0 \prod_{i=1}^{k_1} n_i\right) \\ &= \mathcal{O}\left((k_1+k_2) n_0 m_0 (\prod_{i=1}^{k_1} n_i)^2\right). \end{aligned}$$

**Number of Control Actions on Each Winning State:** $k_1 + k_2$.

### 6.5.6 Generalized co-Büchi

In this subsection, we will introduce the control specification classified as generalized co-Büchi in NTS.

The generalization of a generalized co-Büchi specification is the LTL formulas that can be converted to a generalized deterministic co-Büchi automaton (GDCA).

#### Control Specification Defined by States in NTS

We convert the control problem with a generalized co-Büchi control specification in NTS to a generalized co-Büchi game in $\mathcal{A}$.

**Control Specification**

For a generalized co-Büchi control specification $\varphi$, $S_i \subseteq X$, $1 \leq i \leq k$ are the $k$ sets of safe states labeled with $a_i$ and correspondingly, $X/S_i$ are the $k$ sets of unsafe states labeled with $\neg a_i$. An infinite trajectory $\rho_x$ satisfies $\varphi$ iff only the states in $S_i$ occur in $\rho_x$ eventually for some $i$, $1 \leq i \leq k$, or equivalently, the states in $X/S_i$ occur only finitely many often in $\rho_x$ for some $i$, $1 \leq i \leq k$. Formally, $\rho_x$ satisfies $\varphi$ iff $\bigvee_{i=1}^{k}(\text{Inf}(\rho_x) \subseteq S_i)$, or equivalently, $\bigvee_{i=1}^{k}(\text{Inf}(\rho_x) \cap (X/S_i) = \emptyset)$.

**Conversion to a generalized co-Büchi Game**

Consider an infinite generalized co-Büchi game $GC\mathcal{G} = (\mathcal{A}, \text{Obj})$. $\mathcal{A} = (V, E)$ is the bipartite arena converted from NTS, where $V = V_1 \oplus V_2$ is the set of nodes and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = \{S_1, \cdots, S_k\}$, where $S_i \subseteq V$, $1 \leq i \leq k$ are the $k$ sets of safe nodes consisting of the safe set $S_i$ in the NTS and its control actions. An infinite play $\pi_v$ wins the generalized co-Büchi game iff only the nodes in $S_i$ occur in $\pi_v$ eventually for some $i$, $1 \leq i \leq k$, or equivalently, the nodes in $V/S_i$ occur only finitely many often in $\pi_v$ for some $i$, $1 \leq i \leq k$. Formally, $\pi_v$ wins iff $\bigvee_{i=1}^{k}(\text{Inf}(\pi_v) \subseteq S_i)$, or equivalently, $\bigvee_{i=1}^{k}(\text{Inf}(\pi_v) \cap (V/S_i) = \emptyset)$.

**Complexity Analysis**

In a generalized co-Büchi game $GC\mathcal{G} = (\mathcal{A}, \text{Obj})$, $\mathcal{A} = (V, E)$ is the arena and $\text{Obj} = \{S_1 \cdots, S_k\}$, where $S_i \subseteq V$, $1 \leq i \leq k$ is the objective set. $|V| = n_0$ is the number of nodes and $|E| = m_0$ is the number of edges. We can only consider the $V_1$ nodes in $S_i$ when computing the time complexity. We have the following results according to proposition 5.4.4 from the generalized co-Büchi game.

**Space Complexity:** $\mathcal{O}(n_0 + m_0) = \mathcal{O}(m_0)$.

**Time Complexity:** $\mathcal{O}(kn_0m_0)$.

**Number of Control Actions on Each Winning State:** 1.

### Control Specification Defined as an LTL Formula

We convert the control problem with a GDCA translatable LTL formula as control specification in NTS to a generalized co-Büchi game in $MP\mathcal{A}$.

For a generalized co-Büchi control specification $\varphi = \bigvee_{i=1}^{k} \varphi_i$, where each $\varphi_i$, $1 \leq i \leq k$ can be converted to a DCA, we convert $\varphi$ to a GDCA.

$\varphi = \bigvee_{i=1}^{k} FGa_i$, where $a_i$ is the label of the set of safe states $S_i \subseteq X$, $1 \leq i \leq k$ is the naive generalized co-Büchi objective. We prefer to solve it as a generalized co-Büchi game in $\mathcal{A}$ directly. It is even more efficient to convert it to a DCA than to a GDCA.

For a $GDCA = (Q, L, \tau, q_0, \text{Acc})$, the accepting condition $\text{Acc} = \{F_1, \cdots, F_k\}$, where $F_i \subseteq Q$, $1 \leq i \leq k$ are the $k$ sets of accepting states. An infinite run $\rho_q$ is accepted by GDCA iff only the states in $F_i$ occur in $\rho_q$ eventually for some $i$, $1 \leq i \leq k$. Formally, $\rho_q$ is accepting iff $\bigvee_{i=1}^{k} (\text{Inf}(\rho_q) \subseteq F_i)$.

Equivalently, we can also define the accepting condition $\text{Acc} = \{Q/F_1, \cdots, Q/F_k\}$, where $(Q/F_i) \subseteq Q$, $1 \leq i \leq k$ as the $k$ sets of rejecting states. An infinite run $\rho_q$ is accepted by GDCA iff the states in $Q/F_i$ occur only finitely many often in $\rho_q$ for some $i$, $1 \leq i \leq k$. Formally, $\rho_q$ is accepting iff $\bigvee_{i=1}^{k} (\text{Inf}(\rho_q) \cap (Q/F_i) = \emptyset)$.

### Satisfaction of LTL

An infinite trajectory $\rho_x$ satisfies $\varphi$ iff its infinite input word $\alpha_l$ satisfies the accepting condition of GDCA converted from $\varphi$.

### Conversion to a Generalized co-Büchi Game

Consider the generalized co-Büchi control specification $\varphi = \bigvee_{i=1}^{k} \varphi_i$, $1 \leq i \leq k$, where each $\varphi_i$ can be converted to a DCA $DA_i$ with $\text{Acc}_i$ being the accepting set. Conceptually, there are two ways to construct the multi-product NTS without considering the implementations.

1. Construct a multi-product NTS with the NTS and the $k$ DAs.

$S_i = \{(x, q_1, \cdots, q_k) | x \in X, q_i \in \text{Acc}_i, q_j \in Q_j, j \neq i\} \subseteq X_{MP}$, $1 \leq i \leq k$ are the $k$ sets of safe states in the multi-product NTS.

2. Construct a multi-product DA with the $k$ DAs and then take the product of NTS and multi-product DA to construct a multi-product NTS.

   $\text{Acc} = \{F_1, \cdots, F_k\}$, where $F_i = \{(q_1, \cdots, q_k) | q_i \in \text{Acc}_i, q_j \in Q_j, j \neq i\} \subseteq Q_{MP}$, $1 \leq i \leq k$ are the $k$ sets of accepting states in the multi-product DA.

   $S_i = \{(x, q_1, \cdots, q_k) | x \in X, q_i \in \text{Acc}_i, q_j \in Q_j, j \neq i\} \subseteq X_{MP}$, $1 \leq i \leq k$ are the $k$ sets of safe states in the multi-product NTS.

In the implementation, we may encode the $k$ DA dimensions into 1 to reduce the multi-product NTS into a product NTS. There are several ways of constructing the multi-product NTS in implementation, which will be discussed in the implementation chapter later.

Consider an infinite generalized co-Büchi game $GC\mathcal{G} = (MP\mathcal{A}, \text{Obj})$. $MP\mathcal{A} = (V, E)$ is the bipartite multi-product arena converted from multi-product NTS, where $V = V_1 \oplus V_2$ is the set of nodes and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = \{S_1, \cdots, S_k\}$, where $S_i \subseteq V_1$, $1 \leq i \leq k$ are the $k$ sets of safe nodes consisting of the safe set $S_i$ in the multi-product NTS and its control actions. An infinite play $\pi_v$ wins the generalized co-Büchi game iff only the nodes in $S_i$ occur in $\pi_v$ eventually for some $i$, $1 \leq i \leq k$, or equivalently, the nodes in $V/S_i$ occur only finitely many often in $\pi_i$ for some $i$, $1 \leq i \leq k$. Formally, $\pi_v$ wins iff $\bigvee_{i=1}^{k} (\text{Inf}(\pi_v) \subseteq S_i)$, or equivalently, $\bigvee_{i=1}^{k} (\text{Inf}(\pi_v) \cap (V/S_i) = \emptyset)$.

**Complexity Analysis**

In a generalized co-Büchi game $GC\mathcal{G} = (MP\mathcal{A}, \text{Obj})$, $MP\mathcal{A} = (V, E)$ is the multi-product arena and $\text{Obj} = \{S_1 \cdots, S_k\}$, where $S_i \subseteq V$, $1 \leq i \leq k$ is the objective set.

$|V| = n = \prod_{i=0}^{k} n_i$ is the number of nodes and $|E| = m = m_0 \prod_{i=1}^{k} n_i$ is the number of edges. We can only consider the $V_1$ nodes in $S_i$ when computing the time complexity, thus $|S_i| = n \cdot \frac{|\text{Acc}_i|}{n_i} = \frac{\prod_{j=0}^{k} n_j \cdot |\text{Acc}_i|}{n_i}$, $1 \leq i \leq k$ is the number of safe nodes for $S_i$.

We have the following results according to proposition 5.4.4 from the generalized co-Büchi game.

**Space Complexity:** $\mathcal{O}(n+m) = \mathcal{O}(m) = \mathcal{O}\left(m_0 \prod_{i=1}^{k} n_i\right)$.

**Time Complexity:** $\mathcal{O}(knm) = \mathcal{O}\left(k \cdot \prod_{i=0}^{k} n_i \cdot m_0 \prod_{i=1}^{k} n_i\right) = \mathcal{O}\left(kn_0m_0(\prod_{i=1}^{k} n_i)^2\right)$.

**Number of Control Actions on Each Winning State:** 1.

### Most General Form of the Generalized co-Büchi Control Specification

The most general form of the generalized co-Büchi control specification is given by $\varphi = \bigvee_{i=1}^{k_1} \varphi_i \vee \bigvee_{j=1}^{k_2} FGa_j$, where $\varphi_i$, $1 \leq i \leq k_1$ is a DCA translatable LTL formula and $a_j$, $1 \leq j \leq k_2$ is the label of the set of safe states $S_j \subseteq X$.

The most efficient way of solving this control specification $\varphi$ in NTS is to construct a multi-product NTS with the NTS and the $k_1$ DAs, convert the multi-product NTS to a multi-product arena, and then solve a generalized co-Büchi game with $k_1 + k_2$ co-Büchi objectives in the multi-product arena. The idea is to solve it as a generalized co-Büchi game in the smallest possible arena.

$S_i = \{(x, q_1, \cdots, q_{k_1}) | x \in X, q_i \in \text{Acc}_i, q_j \in Q_j, j \neq i\} \subseteq X_{MP}$, $1 \leq i \leq k_1$ are the $k_1$ sets of safe states in the multi-product NTS given by $\varphi_i$.

$\hat{S}_j = \{(x, q_1, \cdots, q_{k_1}) | x \in S_j, q_i \in Q_i\} \subseteq X_{MP}$, $1 \leq j \leq k_2$ are the extra $k_2$ sets of safe states in the multi-product NTS given by $a_j$.

In a generalized co-Büchi game $GC\mathcal{G} = (MP\mathcal{A}, \text{Obj})$, $MP\mathcal{A} = (V, E)$ is the multi-product arena and $\text{Obj} = \{S_1 \cdots, S_{k_1}, \hat{S}_1, \cdots, \hat{S}_{k_2}\}$, where $S_i, \hat{S}_j \subseteq V_1$, $1 \leq i \leq k_1$, and $1 \leq j \leq k_2$ is the objective set. $|V| = n = \prod_{i=0}^{k_1} n_i$ is the number of nodes and $|E| = m = m_0 \prod_{i=1}^{k_1} n_i$ is the number of edges. $|S_i| = n \cdot \frac{|\text{Acc}_i|}{n_i} = \frac{\prod_{j=0}^{k_1} n_j \cdot |\text{Acc}_i|}{n_i}$, $1 \leq i \leq k_1$ is the number of safe nodes for $S_i$.

$|\hat{S}_j| = n \cdot \frac{|S_j|}{n_0} = \prod_{i=1}^{k} n_i \cdot |S_j|$, $1 \leq j \leq k_2$ is the number of safe nodes for $\hat{S}_j$.

We have the following results according to proposition 5.4.4 from the generalized co-Büchi game.

**Space Complexity:** $\mathcal{O}(n+m) = \mathcal{O}(m) = \mathcal{O}\left(m_0 \prod_{i=1}^{k_1} n_i\right)$.

**Time Complexity:**

$$\begin{aligned}\mathcal{O}((k_1+k_2)nm) &= \mathcal{O}\left((k_1+k_2)\cdot\prod_{i=0}^{k_1} n_i \cdot m_0 \prod_{i=1}^{k_1} n_i\right) \\ &= \mathcal{O}\left((k_1+k_2)n_0 m_0 (\prod_{i=1}^{k_1} n_i)^2\right).\end{aligned}$$

**Number of Control Actions on Each Winning State:** 1.

### 6.5.7 Rabin

In this subsection, we will introduce the control specification classified as Rabin in NTS.

The generalization of a Rabin specification is the LTL formulas that can be converted to a DRA.

A special case is that, we convert the control problem into a one pair Streett game if possible.

#### Control Specification Defined by States in NTS

We convert the control problem with a Rabin control specification in NTS to a Rabin game in $\mathcal{A}$.

#### Control Specification

For a Rabin control specification $\varphi$, $G_i, B_i \subseteq X$, $1 \leq i \leq k$. $G_i$ is the set of "Good" states labeled with $g_i$ and correspondingly, $B_i$ is the set of "Bad" states labeled with $b_i$. An infinite trajectory $\rho_x$ satisfies $\varphi$ iff one of the states in $G_i$ occurs infinitely many often in $\rho_x$ and the states in $B_i$ occur only finitely many often in $\rho_x$ for some $i$, $1 \leq i \leq k$. Formally, $\rho_x$ satisfies $\varphi$ iff $\bigvee_{i=1}^{k} ((\text{Inf}(\rho_x) \cap G_i \neq \emptyset) \wedge (\text{Inf}(\rho_x) \cap B_i = \emptyset))$.

### Conversion to a Rabin Game

Consider an infinite Rabin game $R\mathcal{G} = (\mathcal{A}, \text{Obj})$. $\mathcal{A} = (V, E)$ is the bipartite arena converted from NTS, where $V = V_1 \oplus V_2$ is the set of nodes and

$E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = \{(G_1, B_1), \cdots, (G_k, B_k)\}$, $1 \leq i \leq k$, where $G_i \subseteq V_1$ is the set of "Good" nodes and $B_i \subseteq V$ is the set of "Bad" nodes consisting of the "Bad" set $B_i$ in the NTS and its control actions. An infinite play $\pi_v$ wins the Rabin game iff one of the nodes in $G_i$ occurs infinitely many often in $\pi_v$ and the nodes in $B_i$ occur only finitely many often in $\pi_v$ for some $i$, $1 \leq i \leq k$. Formally, $\pi_v$ wins iff $\bigvee_{i=1}^{k} ((\text{Inf}(\pi_v) \cap G_i \neq \emptyset) \wedge (\text{Inf}(\pi_v) \cap B_i = \emptyset))$.

### Complexity Analysis

In a Rabin game $R\mathcal{G} = (\mathcal{A}, \text{Obj})$, $\mathcal{A} = (V, E)$ is the arena and

$\text{Obj} = \{(G_1, B_1), \cdots, (G_k, B_k)\}$, where $G_i \subseteq V_1$, $B_i \subseteq V$, $1 \leq i \leq k$ is the objective set. $|V| = n_0$ is the number of nodes and $|E| = m_0$ is the number of edges. We can only consider the $V_1$ nodes in $B_i$ when computing the time complexity. We have the following results according to proposition 5.6.1 from the Rabin game. The time complexity of the algorithm 10 is sound and complete when $k = 1$, and is sound but may not be complete when $k > 1$.

**Space Complexity:** $\mathcal{O}(n_0 + m_0) = \mathcal{O}(m_0)$.

**Time Complexity:** $\mathcal{O}(kn_0^2 m_0)$.

**Number of Control Actions on Each Winning State:** 1.

The following results also hold according to proposition 5.6.2 if it is a Streett game. The time complexity of the algorithm 10 is sound and complete when $k = 1$, and is complete but may not be sound when $k > 1$.

**Space Complexity:** $\mathcal{O}(n_0 + m_0) = \mathcal{O}(m_0)$.

**Time Complexity:** $\mathcal{O}(kn_0^2 m_0)$.

**Number of Control Actions on Each Winning State:** $k$.

### Control Specification Defined as an LTL Formula

We convert the control problem with a Rabin translatable LTL formula as control specification in NTS to a Rabin game in $P\mathcal{A}$.

Any LTL formula $\varphi$ can be converted to a DRA.

$\varphi = \bigvee_{i=1}^{k}(GFg_i \wedge FGb_i)$, where $g_i$ is the label of the set of "Good" states $G_i \subseteq X$, and $b_i$ is the label of the set of "Bad" states $B_i \subseteq X$, $1 \leq i \leq k$ is the naive Rabin objective. We prefer to solve it as a Rabin game in $\mathcal{A}$ directly.

For a $DRA = (Q, L, \tau, q_0, \text{Acc})$, the accepting condition $\text{Acc} = \{(G_1, B_1), \cdots, (G_k, B_k)\}$, where $G_1, B_i \subseteq Q$, $1 \leq i \leq k$. "G" refers to the set of "Good" states that we would like to occur infinitely many often in a run $\rho_q$, whereas "B" refers to the set of "Bad" states that we would like to occur only finitely many often in $\rho_q$. An infinite run $\rho_q$ is accepted by DRA iff one of the states in $G_i$ occurs infinitely many often in $\rho_q$ and the states in $B_i$ occur only finitely many often in $\rho_q$ for some $i$, $1 \leq i \leq k$. Formally, $\rho_q$ is accepting iff $\bigvee_{i=1}^{k}((\text{Inf}(\rho_q) \cap G_i \neq \emptyset) \wedge (\text{Inf}(\rho_q) \cap B_i = \emptyset))$.

**Satisfaction of LTL**

An infinite trajectory $\rho_x$ satisfies $\varphi$ iff its infinite input word $\alpha_l$ satisfies the accepting condition of DRA converted from $\varphi$.

**Conversion to a Rabin Game**

In the product NTS $NTS \otimes DA$, $\hat{G}_i = \{(x, q) | x \in X, q \in G_i\} \subseteq X_P$ is the set of "Good" states and correspondingly, $\hat{B}_i = \{(x, q) | x \in X, q \in B_i\} \subseteq X_P$ is the set of "Good" states, $1 \leq i \leq k$.

Consider an infinite Rabin game $R\mathcal{G} = (P\mathcal{A}, \text{Obj})$. $P\mathcal{A} = (V, E)$ is the bipartite product arena converted from product NTS, where $V = V_1 \oplus V_2$ is the set of nodes and $E \subseteq (V_1 \times V_2 \oplus V_2 \times V_1)$ is the set of edges. $\text{Obj} = \{(\hat{G}_1, \hat{B}_1), \cdots, (\hat{G}_k, \hat{B}_k)\}$, $1 \leq i \leq k$, where $\hat{G}_i \subseteq V_1$ is the set of "Good" nodes and $\hat{B}_i \subseteq V$ is the set of "Bad" nodes consisting of the "Bad" set $\hat{B}_i$ in the product NTS and its control actions. An infinite play $\pi_v$ wins the Rabin game iff one of the nodes in $\hat{G}_i$ occurs infinitely many often in $\pi_v$ and the nodes in $\hat{B}_i$ occur only finitely many often in $\pi_v$ for some $i$, $1 \leq i \leq k$. Formally, $\pi_v$ wins iff $\bigvee_{i=1}^{k}\Big((\text{Inf}(\pi_v) \cap \hat{G}_i \neq \emptyset) \wedge (\text{Inf}(\pi_v) \cap \hat{B}_i = \emptyset)\Big)$.

**Complexity Analysis**

In a Rabin game $R\mathcal{G} = (P\mathcal{A}, \text{Obj})$, $P\mathcal{A} = (V, E)$ is the product arena and $\text{Obj} = \{(\hat{G}_1, \hat{B}_1) \cdots, (\hat{G}_k, \hat{B}_k)\}$, where $\hat{G}_i \subseteq V_1$, $\hat{B}_i \subseteq V$, $1 \leq i \leq k$ is the objective set.

$|V| = n = n_0 n_1$ is the number of nodes and $|E| = m = m_0 n_1$ is the number of edges. We can only consider the $V_1$ nodes in $\hat{B}_i$ when computing the time complexity, thus $|\hat{G}_i| = n_0 \cdot |G_i|$ is the number of "Good" nodes for $\hat{G}_i$ and correspondingly, $|\hat{B}_i| = n_0 \cdot |B_i|$ is the number of "Bad" nodes for $\hat{B}_i$.

We have the following results according to proposition 5.6.1 from the Rabin game. The time complexity of the algorithm 10 is sound and complete when $k = 1$, and is sound but may not be complete when $k > 1$.

**Space Complexity:** $\mathcal{O}(n + m) = \mathcal{O}(n_0 n_1 + m_0 n_1) = \mathcal{O}(m_0 n_1)$.

**Time Complexity:** $\mathcal{O}(kn^2 m) = \mathcal{O}(k(n_0 n_1)^2 \cdot m_0 n_1) = \mathcal{O}(k n_0^2 m_0 n_1^3)$.

**Number of Control Actions on Each Winning State:** 1.

The following results also hold according to proposition 5.6.2 if it is a Streett game. The time complexity of the algorithm 10 is sound and complete when $k = 1$, and is complete but may not be sound when $k > 1$.

**Space Complexity:** $\mathcal{O}(n + m) = \mathcal{O}(n_0 n_1 + m_0 n_1) = \mathcal{O}(m_0 n_1)$.

**Time Complexity:** $\mathcal{O}(kn^2 m) = \mathcal{O}(k(n_0 n_1)^2 \cdot m_0 n_1) = \mathcal{O}(k n_0^2 m_0 n_1^3)$.

**Number of Control Actions on Each Winning State:** $k$.

## 6.6 Simulation — from Discrete back to Continuous

In this section, we will discuss how to do simulation in the continuous space using the information in the discrete space.

A discrete controller can be refined and implemented on the original continuous system. Such mapping is guaranteed in the construction of the abstraction from continuous space to discrete space: a sound abstraction guarantees that any discrete controller synthesized using the abstraction can be implemented in the original system to guarantee soundness in terms of satisfying LTL specifications.

Simulation is an approach to do conservative substitute experiments before, or when we don't have access, to do experiments in the real system.

We will first discuss the relation between the continuous dynamical system and its abstraction, and then classify the discussion of simulation according to how the control specification is defined.

### 6.6.1 Relation between Continuous Space and Discrete Space

In this subsection, we will discuss the relation between the continuous space and the discrete space. We will discuss the counterparts between the continuous space and the discrete space in pairs.

**System States**

In the continuous space, the set of continuous system states $X_c$ is partitioned into $n_0$ regions, i.e., $X_c = R_1 \oplus \cdots \oplus R_{n_0}$. The partition is performed by setting up precision for each dimension using the method of uniform gird.

In the discrete space, the set of discrete system states $X_d$ is also of size $|X_d| = n_0$. The $n_0$ regions in the continuous space is 1-1 correspondence to the $n_0$ system states in the discrete space.

The conversion of a continuous system state $x_{c1} \in R_1$ to a discrete system state $x_{d1}$ is an encoding process since the continuous system states may have several dimensions. We will represent such conversion as $x_{c1} \xrightarrow{encode} x_{d1}$.

**Control Actions**

In the continuous space, we choose the finite set of continuous control actions by setting up precision for each dimension using the method of uniform grid out of $\Sigma_c$, which is 1-1 correspondence to the finite set of discrete control actions $\Sigma_d$ of size $|\Sigma_d|$.

The conversion of a discrete control action $\sigma_d$ to a continuous control action $\sigma_c$ is a decoding process since the continuous control actions may have several dimensions. We will represent such conversion as $\sigma_d \xrightarrow{decode} \sigma_c$.

**Time Step**

In the continuous space, we fix the unit time step as $\Delta t$.

In the discrete space, we abstract the unit time step $\Delta t$ as 1.

The LTL formula and the automaton share the same time step as the dynamical system.

**Transition Function**

In the continuous space, the transition function is initially described by the ODE $\dot{x}(t) = f(x(t), \sigma)$. After we have fix the unit time step $\Delta t$, we convert the ODE to a DE $x_{t+1} = f(x_t, \sigma)\Delta t + x_t$.

A continuous transition $(x_{c1}, \sigma_c, x_{c2})$ projects to a discrete transition $(x_{d1}, \sigma_d, x_{d2})$, where $x_{c1} \in R_1$, $x_{c2} \in R_2$, $R_1$ maps to $x_{d1}$, $\sigma_c$ maps to $\sigma_d$, and $R_2$ maps to $x_{d2}$.

**Labelling**

In the discrete space, we under approximate the positive atomic propositions of that in the continuous space, whereas we over approximate the negative atomic propositions of that in the continuous space.

**Agent and Token**

Since there is dynamics in the dynamical system, we place an agent in the dynamical system in both continuous and discrete case. Since there is no dynamics in an automaton or an arena, we place a token in the automaton or an arena.

Consider the control synthesis in NTS. When the control specification $\varphi$ is defined by an LTL formula, we convert it to a DA with a corresponding accepting condition. Then the agent in the NTS and the token in the DA move in pairs at each time step. If we construct a product NTS by the NTS and the DA, then the agent in the NTS and the token in the DA combine into one new agent. If we construct a multi-product NTS by the NTS and multiple DAs, then the agent in the NTS and the tokens in the multiple DAs combine into one new agent.

The agent in the continuous dynamical system and the token in the DA also move in pairs at each time step.

In the simulation, the agent is initially placed on $x_c \in \mathcal{W}_{0c}$ in the continuous space, and correspondingly, the agent is initially placed on $x_d \in \mathcal{W}_{0d}$ in the discrete space. The token is initially placed on the unique initial automaton state if there is any DA.

**Trajectory and a Sequence of Control Inputs**

A trajectory $\rho = x_0 x_1 x_2 \cdots$ is a sequence of system states $x_i \in X$, for $i \geq 0$, which can be finite $(r)$ or infinite $(\rho)$ depending on the control specification $\varphi$.

A sequence of control inputs $\alpha = \sigma_0 \sigma_1 \sigma_2 \cdots$ is a sequence of control actions $\sigma_i \in \Sigma$, for $i \geq 0$, which can be finite $(w)$ or infinite $(\alpha)$ depending on the trajectory.

In the continuous space, a sequence of control inputs $\alpha_c = \sigma_0 \sigma_1 \sigma_2 \cdots$ is a sequence of control actions $\sigma_i \in \Sigma_c$, for $i \geq 0$. A trajectory $\rho_c = x_0 x_1 x_2 \cdots$ is a sequence of system states $x_i \in X_c$, for $i \geq 0$, where $x_{i+1} = f(x_i, \sigma_i)\Delta t + x_i$.

In the discrete space, a sequence of control inputs $\alpha_d = \sigma_0 \sigma_1 \sigma_2 \cdots$ is a sequence of control actions $\sigma_i \in \Sigma_d$, for $i \geq 0$. A trajectory $\rho_d = x_0 x_1 x_2 \cdots$ is a sequence of system states $x_i \in X_d$, for $i \geq 0$, where $x_{i+1} \in \xi(x_i, \sigma_i)$.

### 6.6.2 Control Specification Defined by States in NTS

After we have done the control synthesis in the NTS, we have computed the winning set $\mathcal{W}_{0d} \subseteq X_d$ and the winning control strategy $\Pi : \cdot \to \Sigma_d$. We initially choose a continuous system state $x_{c0} \in X_c$ and project it to the NTS by $x_{c0} \xrightarrow{encode} x_d$. If $x_d \in \mathcal{W}_{0d}$, then the induced trajectory initiated at $x_{c0}$ by our abstraction based method should always satisfy the accepting condition of the control specification $\varphi$ as long as the abstraction is constructed conservative enough; otherwise the induced trajectory may not always satisfy $\varphi$. We assume the initial continuous system state $x_{c0} \in \mathcal{W}_{0c}$, and classify the simulation according to the different winning control strategies.

**Local Strategy**

Here we give the procedure to simulate the trajectory for control specifications that only require local strategy. It takes 1 unit for each $x_d \in \mathcal{W}_{0d}$ to record the local winning control strategy. Such control specifications include reachability, safety, Büchi, co-Büchi, generalized co-Büchi, Rabin and one pair Streett. The trajectory induced by the local strategy is constructed by iterating the following process:

- $x_c \xrightarrow{encode} x_d$;
- $x_d.strategy = \sigma_d$;
- $\sigma_d \xrightarrow{decode} \sigma_c$;
- $x_c' = f(x_c, \sigma_c)\Delta t + x_c$;
- $x_c \leftarrow x_c'$;

**Local Strategy + Counter**

Here we give the procedure to simulate the trajectory for control specifications that require local strategy and a counter. It takes $k$ units for each $x_d \in \mathcal{W}_{0d}$ to record the local winning control strategy since there are $k$ objectives in the control specifications.

The counter increments to indicate the next objective when the current objective is satisfied. We initialize the counter $i$ as 0, and define a module plus $a +_k b$ as $(a \bmod k) + b$. Such control specifications include generalized Büchiand $k$ pairs Streett. The simulation for the $k$ pairs Streett may not be sound as a result of a conflict among different objectives.

The trajectory induced by the local strategy and the counter is constructed by iterating the following process:

- $x_c \xrightarrow{encode} x_d$;
- if $x_d$ satisfies the $i^{th}$ objective, then $i \leftarrow i +_k 1$;
- $(x_d, i).strategy = \sigma_d$;
- $\sigma_d \xrightarrow{decode} \sigma_c$;
- $x_c' = f(x_c, \sigma_c)\Delta t + x_c$;
- $x_c \leftarrow x_c'$;

We can set up necessary or sufficient conditions to verify or falsify our simulation by counting the number of occurrence of the objective set such as the target set or the safe set:

- in a reachability objective, a sufficient condition to falsify the simulation is that, the trajectory doesn't reach a target set within $|\mathcal{W}_{0d}|$ steps;
- in a safety objective, a sufficient condition to falsify the simulation is that, the trajectory reaches an unsafe state;
- in a Büchi objective, a necessary condition to verify the simulation is that, the trajectory should reach the target set within $|\mathcal{W}_{0d}|$ steps each time;
- in a co-Büchi objective, a sufficient condition to falsify the simulation is that, the trajectory reaches the unsafe states more than $|X_d/S|$ times;
- in a generalized Büchi objective, a necessary condition to verify the simulation is that, the trajectory should reach each target set within $|\mathcal{W}_{0d}|$ steps each time;
- in a generalized co-Büchi objective, a sufficient condition to falsify the simulation is that, the trajectory reaches some states in $\mathcal{W}_{0d}/(\bigcup_{i=i}^{k} S_i)$ more than $\left|\mathcal{W}_{0d}/(\bigcup_{i=i}^{k} S_i)\right|$ times;
- in a Rabin objective, a necessary condition to verify the simulation is that, the trajectory should reach some states in $\bigcap_{i=i}^{k} B_i$ for at most $\left|\bigcap_{i=i}^{k} B_i\right|$ times, and reach $G_i$ within $|\mathcal{W}_{0d}|$ steps each time for some $i$, $1 \leq i \leq k$;

- in a one pair Streett objective, a necessary condition to verify the simulation is that, if the trajectory reaches a state in $B$, then the trajectory should reach a state in $G$ within $|\mathcal{W}_{0d}|$ steps each time.

More detailed verification for the simulation of the Rabin objective requires considering the $k$ Rabin pairs together to check whether there is some Rabin pair that satisfies the accepting condition.

The Streett objective requires satisfying the $k$ Streett conditions at the same time, where each Streett condition requires satisfying either an infinite number of occurrence of the "Good" set or a finite number of occurrence of the "Bad" set. The trajectory can choose to satisfy either of the two to satisfy a Streett condition. However, it is quite complicated to track the switching between the finite and the infinite accepting condition, and to coordinate among the $k$ Streett conditions to avoid conflicts. Therefore, it requires much memory to construct or to verify a trajectory that satisfies a Streett objective. This is the reason why we omit the construction and the verification here.

### 6.6.3 Control Specification Defined as an LTL Formula

We will classify our discussion w.r.t. the product NTS and the multi-product NTS.

**Control Synthesis in the Product NTS**

After we have done the control synthesis in the product NTS, we have computed the winning set $\mathcal{W}_P \subseteq X_P$ and the winning control strategy $\Pi : \cdot \rightarrow \Sigma_d$. The winning set of the NTS $\mathcal{W}_0$ is the projection of the product states $(x, q_0) \in (\mathcal{W}_P \cap X_0)$ to the NTS state dimension $X$: $\mathcal{W}_0 = \{x|(x, q_0) \in (\mathcal{W}_P \cap X_0)\}$. $\mathcal{W}_P$ has a meaning iff $\mathcal{W}_0$ is not empty. We need the winning control strategy in each $\mathcal{W}_P$ state to do simulation. We locate a $\mathcal{W}_P$ state by the agent, token pair in the NTS and the DA.

**Control Synthesis in the Multi-Product NTS**

After we have done the control synthesis in the multi-product NTS, we have computed the winning set $\mathcal{W}_{MP} \subseteq X_{MP}$ and the winning control strategy $\Pi : \cdot \rightarrow \Sigma_d$. The winning set of the NTS $\mathcal{W}_0$ is the projection of the multi-product states $(x, q_{10}, \cdots, q_{k0}) \in (\mathcal{W}_{MP} \cap X_0)$ to the NTS state dimension $X$: $\mathcal{W}_0 = \{x|(x, q_{10}, \cdots, q_{k0}) \in (\mathcal{W}_{MP} \cap X_0)\}$. $\mathcal{W}_{MP}$ has a meaning iff $\mathcal{W}_0$ is not empty. We need the winning control strategy in each $\mathcal{W}_{MP}$ state to do simulation. We locate a $\mathcal{W}_{MP}$ state by the combination of the agent, and the $k$ tokens in the NTS and the $k$ DAs.

Since we already gave a detailed analysis of the simulation for the control specification defined by the States in NTS in the previous subsection, here we only give the procedure for the simulation of the control synthesis in the multi-product NTS that requires only local strategy for simplicity. The control synthesis in the product NTS is a special case of this where $k = 1$, and we can refer the simulation of the control specification that requires local strategy and a counter to the previous subsection.

We initially choose a continuous system state $x_{c0} \in X_c$ and project it to the NTS by $x_{c0} \xrightarrow{encode} x_d$. If $x_d \in \mathcal{W}_{0d}$, then the induced trajectory initiated at $x_{c0}$ by our abstraction based method should always satisfy the accepting condition of the control specification $\varphi$ as long as the abstraction is constructed conservative enough; otherwise the induced trajectory may not always satisfy $\varphi$. We assume the initial continuous system state $x_{c0} \in \mathcal{W}_{0c}$, and give the procedure to simulate the trajectory for control specifications that only require local strategy. It takes 1 unit for each $x_{MP} \in \mathcal{W}_{MP}$ to record the local winning control strategy. The token of $DA_i$ is initially placed on $q_i = q_{i0}$, $1 \leq i \leq k$.

The trajectory induced by the local strategy is constructed by iterating the following process:

- $x_c \xrightarrow{encode} x_d$;
- $(x_d, q_1, \cdots, q_k).strategy = \sigma_d$;
- $\tau(l, q_i) = q_i'$, where $l$ is the label of $x_d$, $1 \leq i \leq k$;
- $\sigma_d \xrightarrow{decode} \sigma_c$;
- $x_c' = f(x_c, \sigma_c)\Delta t + x_c$;
- $x_c \leftarrow x_c'$, $q_i \leftarrow q_i'$, $1 \leq i \leq k$;

We verify whether the trajectory of the agent satisfies the accepting condition of the LTL formula by verifying whether the runs of the tokens satisfy the accepting conditions of the DAs.

# Chapter 7

# Implementations, Conclusions and Future Work

## 7.1 Implementations

In this section, we will give some analysis and examples about implementations.

### 7.1.1 Different Ways of Taking Multi-Product

We construct the multi-product NTS to solve a generalized Büchi control specification $\varphi = \bigwedge_{i=1}^{k} \varphi_i$, where each $\varphi_i$, $1 \leq i \leq k$ can be converted to a DBA $DA_i$ with $\text{Acc}_i$ being the accepting set.

We give five approaches of solving such problem and compare their performances.

1. Incrementally taking product of NTS and each of the $k$ DAs and solve a generalized $k$ Büchi game [40].
2. Simultaneously taking product of NTS and the $k$ DAs together and solve a generalized $k$ Büchi game.
3. Simultaneously taking product of the $k$ DAs to construct a multi-product DA, which can be encoded to a GDBA. Then we take product of the NTS and the GDBA together and solve a generalized $k$ Büchi game [5].

4. Take $k$ copies of multi-product NTS and solve a Büchi game.

5. Take $(k+1)$ copies of multi-product NTS and solve a Büchi game.

In method 1, the multi-product NTS is propagated as $NTS$, $NTS \otimes DA_1$, $NTS \otimes DA_1 \otimes DA_2$, $\cdots$, upto $NTS \otimes DA_1 \otimes \cdots \otimes DA_k$. This approach doesn't take advantage of the conjunction property of the generalized Büchi objective, and requires reallocating the memory and increasing the tuple number as we incrementally take the product. Also, there may be redundant parts as we take the product with an additional DA.

In method 2, the multi-product NTS is constructed from $NTS$ to $NTS \otimes DA_1 \otimes \cdots \otimes DA_k$ directly and the $k+1$ tuple is also encoded directly. We take advantage of the conjunction property of the generalized Büchi objective, and the entire process is monotonically increasing.

In method 3, it constructs a multi-product DA first and then take the product with the NTS instead of constructing the multi-product NTS directly as in method 2. This method doesn't take advantage of the structure of the NTS in the first place, thus is slightly less efficient than method 2, but still better than method 1.

Method 4 and 5 converts the generalized Büchi game to a Büchi game in a cost of $k$ or $(k+1)$ times of space cost, which is not advised as in the analysis for the generalized Büchi.

Therefore, we propose method 2 which performs the best, and then method 3 and method 1 are slightly less efficient, whereas method 4 and method 5 are even less efficient.

### 7.1.2 Example

In this subsection, we will give two case studies of our implementations. The program is included in the tool ROCS [23].

**Problem Formulation**

Consider the model of a robot car [1] depicted in Figure 7.1.

The kinematics of the model is given by

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos(\gamma+\theta)\cos(\gamma)^{-1} \\ v\sin(\gamma+\theta)\cos(\gamma)^{-1} \\ v\tan(\phi) \end{bmatrix}, \tag{7.1}$$

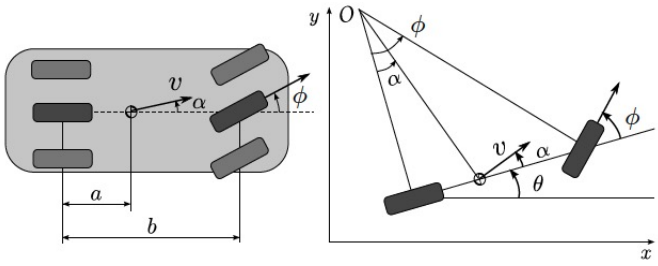


Figure 7.1: Model of Robot Car

where $(x, y)$ is the planar position of center of the vehicle, $\theta$ is its orientation, the control variable $v$ and $\phi$ is the velocity and steering angle, respectively, and $\gamma = \arctan(\tan(\phi)/2)$.

We consider the 3 dimensional state space $\mathcal{X} = [0, 10] \times [0, 10] \times [-\pi, \pi]$ and the 2 dimensional control space $\mathcal{U} = [-1, 1] \times [-1, 1]$.

$\mathcal{O} = \{o_1, o_2, o_3, o_4\}$ is the set of obstacles, where

- $o_1 = [1.6, 5.7] \times [4.0, 5.0] \times [-\pi, \pi]$;
- $o_2 = [3.0, 5.0] \times [5.0, 8.0] \times [-\pi, \pi]$;
- $o_3 = [4.3, 5.7] \times [1.8, 4.0] \times [-\pi, \pi]$;
- $o_4 = [5.7, 8.5] \times [1.8, 2.5] \times [-\pi, \pi]$.

$A_1$, $A_2$, $A_3$ are the 3 sets of target sets, where

- $A_1 = [1.0, 2.0] \times [0.5, 2.0] \times [-\pi, \pi]$;
- $A_2 = [0.5, 2.5] \times [7.5, 8.5] \times [-\pi, \pi]$;
- $A_3 = [7.1, 9.1] \times [4.6, 6.4] \times [-\pi, \pi]$.

The workspace in the $(x, y)$-plane is depicted in Figure 7.2.

We consider two control specifications:

- $\varphi_1$: Patrol the three areas $A_1$, $A_2$, and $A_3$ indefinitely.

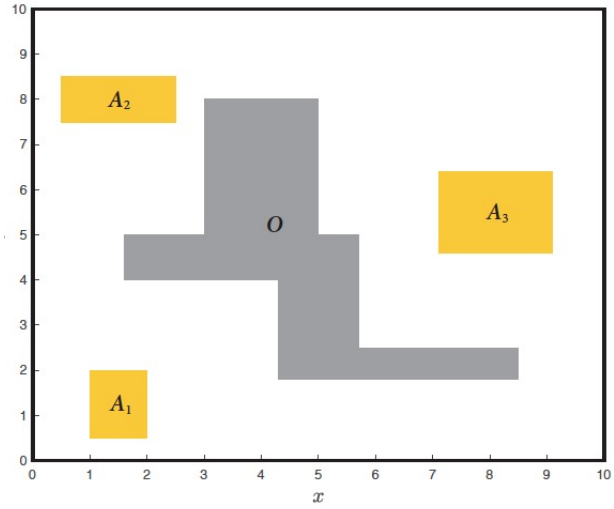


Figure 7.2: Workspace

- $\varphi_2$: The robot should eventually visit area $A_1$, and from there, it should eventually visit area $A_2$, and from there, it should eventually visit $A_3$, and then go back to $A_1$ without visiting $A_2$.

The robot should remain in the set $\mathcal{X}$, while avoiding the obstacle sets in $\mathcal{O}$, and the control input should take values in $\mathcal{U}$.

**Solution**

We use an exact discrete-time model of (7.1) with sampling time $\tau = 0.3$s for the control synthesis. We partition the control space $\mathcal{U}$ with the sample grid $\mu = 0.3$. The set of discrete control action is of size $|\Sigma_d| = 7 \times 7 = 49$. We first use tools such as SCOTS [32] and ROCS [23] to construct the abstraction of the original system in the form of an NTS.

Now we classify our discussion according to the two control specifications.

**Task 1**

We write the control specification as an LTL formula $\varphi_1 = GFa_1 \wedge GFa_2 \wedge GFa_3$, where $a_1$, $a_2$, and $a_3$ are the labels of $A_1$, $A_2$, and $A_3$ respectively.

Then we use the tool Spot [13] to automatically convert $\varphi_1$ to a DBA, shown in Figure 7.3.

It has 4 states and 13 transitions.

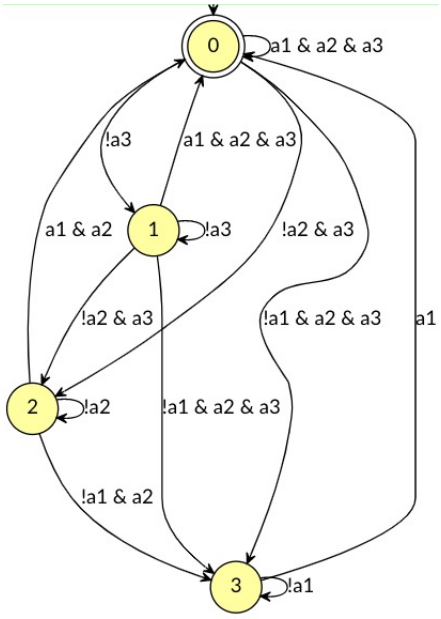


Figure 7.3: DBA for $\varphi_1$

The data for Task 1 is shown in Table 7.1. The Büchi game is solved in one iteration, thus is linear in practice.

**Remark.** *Solving $\varphi_1$ as a generalized Büchi game in the NTS is more efficient than solving a Büchi game in the product space.*

### Task 2

We write the control specification as an LTL formula

$\varphi_2 = F(a_1 \wedge F(a_2 \wedge F(a_3 \wedge (\neg a_2 U a_1))))$, where $a_1$, $a_2$, and $a_3$ are the labels of $A_1$, $A_2$, and $A_3$ respectively.

Then we use the tool Spot [13] to automatically convert $\varphi_2$ to a DBA, shown in Figure 7.4.

It has 5 states and 14 transitions.

The data for Task 2 is shown in Table 7.2. The Büchi game is solved in one iteration, thus is linear in practice.

**Remark.** *$\varphi_2$ is a generalized reachability objective, thus we are actually solving a reachability game instead of a Büchi game. Therefore, $\varphi_2$ has linear time complexity.*

| Precision in Each Dimension | 0.2 | 0.1 | 0.05 |
|---|---|---|---|
| Number of System States | 91035 | 724271 | 5696541 |
| Number of System Transitions | 40873755 | 344373735 | 2793066498 |
| Time for Abstraction (s) | 12.1694 | 121.186 | 988.595 |
| Number of Product States | 288629 | 2403698 | 19086724 |
| Number of Product Transitions | 148329535 | 1320400864 | 10845804208 |
| Time for Synthesis (s) | 6.0155 | 56.026 | 547.3567 |
| Total Time (s) | 18.1849 | 177.212 | 1535.9517 |
| Memory (GB) | 1.67 | 24.28 | 199.18 |

Table 7.1: Data for Task 1

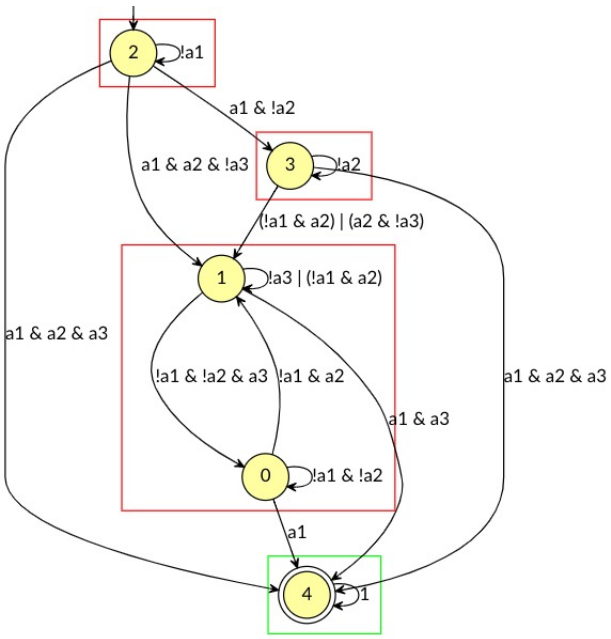


Figure 7.4: DBA for $\varphi_2$

| Precision in Each Dimension | 0.2 | 0.1 | 0.05 |
|---|---|---|---|
| Number of System States | 91035 | 724271 | 5696541 |
| Number of System Transitions | 40873755 | 344373735 | 2793066498 |
| Time for Abstraction (s) | 12.1694 | 121.186 | 988.595 |
| Number of Product States | 360860 | 3005693 | 23864389 |
| Number of Product Transitions | 185457682 | 1651151151 | 13560950147 |
| Time for Synthesis (s) | 7.7206 | 84.4351 | 635.0992 |
| Total Time (s) | 19.89 | 205.6211 | 1623.6942 |
| Memory (GB) | 3.17 | 29.68 | 243.41 |

Table 7.2: Data for Task 2

**Discussion.** *The large consumption of memory is a big bottleneck of the abstraction-based method. A personal computer can only run a precision of 0.2. However, with the support of the computation platform Canada Compute that provides memory up to 1 TB, the memory issue as been largely resolved.*

## 7.2 Conclusions

A systematic approach of solving a control problem in NTS is to convert it into an infinite game such as Büchi or Rabin, which has polynomial or higher theoretical worst case time complexity. However, due to the enormous size of the NTS, the control problem is feasible to solve only if it can be solved in linear time in practice. The main effort of this thesis is to show that we can expect a linear time complexity in practice even though the theoretical time complexity is polynomial or higher. We input much effort in both analysis and implementation to show the feasibility of the abstraction-based method in implementation.

For the analysis, we showed that an implicit tighter upper bound exists in a polynomial game, and some heuristic optimizations to reduce the time complexity. We give examples of worst case scenarios that is deliberately designed for high complexity which is not likely to happen in a natural system like NTS. Actually, Büchi game always terminates in one or two iterations in implementation, which shows that the basic algorithm is efficient enough without adding any heuristic optimizations. Our attention has thus been dragged from the time complexity back to the space complexity. In other words, our motivation for future work is to consider the reduction of the abstraction in the continuous space. The current available tools only solve control specifications such as reachability, safety, Büchi and co-Büchi in the NTS, which can not handle more general control specifications. Our

efficient Büchi solver [23] provides a more systematic approach of solving all the control specifications that can be translated into a DBA.

## 7.3 Future Work

In this section, we will discuss the directions of our potential future work.

We would like to explore our future work from the following six perspectives:

1. Reduction and refinement of the abstraction from a numerical analysis perspective;

   We would consider constructing the abstraction using different numerical schemes.

2. Parallel computing in construction of the abstraction;

   Since the computation of each uniform grid is independent of each other, the computation can be paralleled.

3. Combination of the abstraction and specification from a data structure and algorithm design perspective;

   Since we can expect a linear time complexity for solving Büchi game in practice, we would consider a design of non-linear data structure to store the abstraction. We would refine the abstraction and solve a Büchi game alternatively in an effort to reduce space complexity and time complexity.

4. Describing the problem as a Rabin game or Parity game from a completeness perspective;

5. Addition of rewards and costs from an optimization perspective;

6. Addition of the probability into the model from a stochastic perspective.

   We would consider the uncertainty of the system described as probability such as Markov Decision Process (MDP).

# References


[1] Karl Johan Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton, 2008.

[2] Ebru Aydin Gol, Mircea Lazar, and Calin Belta. Language-guided controller synthesis for linear systems. *IEEE Trans. Automat. Contr.*, 59(5):1163–1176, 2014.

[3] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of Model Checking*. MIT press, 2008.

[4] Calin Belta, Boyan Yordanov, and Ebru Aydin Gol. *Formal Methods for Discrete-Time Dynamical Systems*. Springer International Publishing, 2017.

[5] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, and Barbara Jobstmann. Robustness in the presence of liveness. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 410–424, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'Ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[7] Udi Boker. Why these automata types? In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 57 of *EPiC Series in Computing*, pages 143–163. EasyChair, 2018.

[8] Krishnendu Chatterjee, Wolfgang Dvorák, Monika Henzinger, and Veronika Loitzenbauer. Conditionally optimal algorithms for generalized büchi games. *CoRR*, abs/1607.05850, 2016.

[9] Krishnendu Chatterjee and Monika Henzinger. An $o(n^2)$ time algorithm for alternating Büchi games. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 1386–1399. SIAM, 2012.

[10] Krishnendu Chatterjee, Thomas A Henzinger, and Nir Piterman. Algorithms for Büchi games. In *Proceedings of 3rd Workshop on Games in Design and Verification*, 2006.

[11] L. de Alfaro, T.A. Henzinger, and R. Majumdar. From verification to control: dynamic programs for omega-regular objectives. In *Proc. of LICS*, pages 279–290, 2001.

[12] Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. Symbolic algorithms for infinite-state games. In *CONCUR*, pages 536–550. Springer-Verlag Berlin Heidelberg, 2001.

[13] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and $\omega$-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, October 2016.

[14] Nathanaël Fijalkow and Florian Horn. The surprizing complexity of generalized reachability games. *arXiv*, 2010.

[15] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata Logics, and Infinite Games*. Springer Berlin Heidelberg, 2002.

[16] Florian Horn. Streett games on finite graphs. In *In GDV'05*, 2005.

[17] Luc Jaulin. *Applied Interval Analysis: with Examples in Parameter and State Estimation, Robust Control and Robotics*, volume 1. Springer Science & Business Media, 2001.

[18] Charanjit S Jutla. Determinization and memoryless winning strategies. *Inf. Comput.*, 133(2):117–134, mar 1997.

[19] Bakhadyr Khoussainov and Anil Nerode. *Automata Theory and its Applications*. Birkhäuser Boston, 2001.

[20] Marius Kloetzer and Calin Belta. Dealing with nondeterminism in symbolic control. In *Proc. of HSCC*, pages 287–300, 2008.

[21] Marius Kloetzer and Calin Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Trans. Automat. Contr.*, 53(1):287–297, feb 2008.

[22] Yinan Li and Jun Liu. Invariance control synthesis for switched nonlinear systems: An interval analysis approach. *IEEE Trans. Automat. Contr.*, 63(7):2206–2211, 2018.

[23] Yinan Li and Jun Liu. ROCS: A robustly complete control synthesis tool for nonlinear dynamical systems. In *Proc. of HSCC*, pages 130–135, 2018.

[24] Jun Liu. Robust abstractions for control synthesis: Robustness equals realizability for linear-time properties. In *Proc. of HSCC*, pages 101–110, 2017.

[25] Robert McNaughton. Infinite games played on finite graphs. *Ann. Pure Appl. Log.*, 65(2):149–184, 1993.

[26] Ramon E Moore. *Interval Analysis*. Prentice-Hall, 1966.

[27] N. Piterman and A. Pnueli. Faster solutions of rabin and streett games. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 275–284, 2006.

[28] Nir Piterman and Amir Pnueli. Faster solutions of rabin and streett games. In *Proc. of LICS*, pages 275–284, 2006.

[29] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simonetta Ronchi Della Rocca, editors, *Automata, Languages and Programming*, pages 652–671, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[30] Giordano Pola, Antoine Girard, and Paulo Tabuada. Approximately bisimilar symbolic models for nonlinear control systems. *Automatica*, 44(10):2508–2516, 2008.

[31] Stefan Ratschan and Zhikun She. Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *ACM Trans. Embed. Comput. Syst.*, 6(1):8, 2007.

[32] Gunther Reissig, Alexander Weber, and Matthias Rungger. Feedback refinement relations for the synthesis of symbolic controllers. *IEEE Trans. Automat. Contr.*, 62(4):1781 – 1796, 2017.

[33] Matthias Rungger, Jr Mazo Manuel, and Paulo Tabuada. Specification-guided controller synthesis for linear systems and safe linear-time temporal logic. In *Proc. of HSCC*, pages 333–342, 2013.

[34] Paulo Tabuada. *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer Science & Business Media, 2009.

[35] Moshe Y. Vardi. *An automata-theoretic approach to linear temporal logic*, pages 238–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[36] Eric M. Wolff, Ufuk Topcu, and Richard M. Murray. Automaton-guided controller synthesis for nonlinear systems with temporal logic. In *Proc. of ICIRS*, pages 4332–4339, 2013.

[37] Eric M. Wolff, Ufuk Topcu, and Richard M. Murray. Efficient reactive controller synthesis for a fragment of linear temporal logic. In *Proc. of ICRA*, pages 5033–5040, 2013.

[38] Boyan Yordanov, Jana Tumova, Ivana Cerna, Jiří Barnat, and Calin Belta. Temporal logic control of discrete-time piecewise affine systems. *IEEE Trans. Automat. Contr.*, 57(6):1491–1504, 2012.

[39] Majid Zamani, Giordano Pola, Manuel Mazo Jr., and Paulo Tabuada. Symbolic models for nonlinear control systems without stability assumptions. *IEEE Trans. Automat. Contr.*, 57(7):1804–1809, 2012.

[40] M. H. Zibaeenejad and J. Liu. Auditor product and controller synthesis for nondeterministic transition systems with practical ltl specifications. *IEEE Transactions on Automatic Control*, 65(10):4281–4287, 2020.

[41] Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.

# APPENDICES

# Appendix A

# Preliminaries

In this chapter, we will give some background introduction in graph theory and algorithms.

This is the background for the more complicated graph based problems of the infinite two player games in chapter 5. We would like to introduce some of the basic terminologies and algorithms in the Graph Theory first. Experienced readers with a computer science background may skip this chapter. However, it is still advised to review some of the properties and the mechanisms of the fundamental algorithms as the more complicated algorithms will be defined and derived from the results in this chapter.

The discussions in this chapter will be detailed and elementary, whereas the discussion in the mean thesis will focus on the higher level ideas based on the results in this chapter.

## A.1 Basic Terminologies

In this section, we will introduce some basic terminologies in graph theory.

**Graph**

Here we define some of the terminologies related to graph.

**Definition A.1.1** (graph)**.** *The environment/ arena/ graph is a pair $G = (V, E)$, where $V$ is the set of vertices/ nodes/ points/ states, $|V| = n$; $E \subseteq V \times V$ is the set of edges/ arcs/ links/ transitions, $|E| = m$.*

**Definition A.1.2** (edge)**.** *An edge $e = (u, v)$ is a pair of nodes, ordered or unordered.*

- *In an undirected graph, such edge looks like $u - v$; in a directed graph, such edge looks like $u \to v$, where the arrow points from $u$ to $v$. From now on, we only consider edges in a directed graph.*

- *We say $(u, v)$ is an out-going edge of $u$, and an in-going edge of $v$. The out-degree(outdeg) of a node $v$ is the number of out-going edges of $v$; the in-degree(indeg) of $v$ is the number of in-going edges of $v$.*

- *The predecessors of $v$ are the nodes that point to $v$, and the successors of $v$ are the nodes that point from $v$. Sometimes we use Pre and Post to abbreviate predecessors and successors. Formally, the Pre and Post of $v$ are defined as $Pre(v) = \{u|(u, v) \in E\}$ and $Post(v) = \{u|(v, u) \in E\}$, respectively.*

**Definition A.1.3** (subgraph)**.** *A graph $G' = (V', E')$ is a subgraph of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.*

A subgraph $G' = (V', E')$ of a graph $G = (V, E)$ is often induced by the subset of nodes $V' \subseteq V$ with the relevant edges $E' \subseteq E$. Therefore, $G'$ is also called the $V'$ induced subgraph of $G$.

**Definition A.1.4** (self-loop)**.** *A self-loop is an edge that connects a node to itself.*

**Definition A.1.5** (multi-edges)**.** *The edges between two nodes are called multi-edges if there are more than one edge between these two nodes.*

**Definition A.1.6** (walk)**.** *A walk is a finite or infinite sequence of edges that joins a sequence of nodes.*

**Definition A.1.7** (trail)**.** *A trail is a walk with no repetitive edges.*

**Definition A.1.8** (path)**.** *A path is a walk with no repetitive nodes.*

**Remark.**

- *If without specified, we consider $|V| = n$ and $|E| = m$ as conversion.*

- *We only consider finite graphs here, i.e., the number of nodes and edges are both finite.*

- *Graphs are classified as directed or undirected:*

    1. *The undirected graphs are the simpler ones, since there is no direction/arrow on an edge. Therefore, as long as there is an edge between two nodes, these two nodes are connected to each other from both ends.*

2. *The directed graphs are also called digraphs, where there is a direction/arrow on an edge. This makes the direction one-way and may lose some connectivity, therefore is more complicated.*
3. *We can now see that an undirected graph can be seen as a special case of a directed graph if we add two-way arrows to all the edges of the graph.*

- *If without specified, we consider directed graphs only.*
- *Conventionally, the graphs are represented in out-going edges.*
- *However, due to the structure of the problem we consider, we have to work on graphs represented in in-going edges most of the time, which is counterintuitive.*
  1. *Most problems in Graph Theory are defined on graphs represented in out-going edges.*
  2. *However, for problems related to two-player games such as Reachability, Safety, Büchi and co-Büchi, we have to work on graphs represented in in-going edges.*
  3. *Perhaps the only place that we will work on graphs represented in out-going edges with problems in two-player games is when taking the product of an NTS and DBA(s), yet we still represent the graph of the product in in-going edges, and then solve a particular game based on that product graph.*
- *We allow self-loops in the graph.*
- *Since we would like to guarantee an infinite walk in the graph, all the nodes are guaranteed to have* $outdeg(v) > 0$.

### Complexity Analysis

Here we define some of the terminologies for complexity analysis. Complexity analysis refers to the analysis of the time or the space complexity of an algorithm or program, where we use the order of growth of some parameters in the program to approximate the execution time or the amount of memory used of the program.

There are three bounds to depict the approximations: $\mathcal{O}(\cdot)$, $\Omega(\cdot)$ and $\Theta(\cdot)$.

**Definition A.1.9** (Big-O notation $\mathcal{O}(\cdot)$)**.** *We use big-O notation* $\mathcal{O}(\cdot)$ *to denote asymptotic upper bound. Formally, for a given function* $g(n)$*, we denote by* $\mathcal{O}(g(n))$*, pronounced "big-oh of g of n", for the set of functions:*

$$\mathcal{O}((g(n)) = \{f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ s.t. } 0 \leq f(n) \leq c \cdot g(n), \ \forall\ n \geq n_0\}.$$

**Definition A.1.10** (Big-$\Omega$ notation $\Omega(\cdot)$)**.** *We use big-$\Omega$ notation $\Omega(\cdot)$ to denote asymptotic lower bound. Formally, for a given function $g(n)$, we denote by $\Omega(g(n))$, pronounced "big-omega of g of n", for the set of functions:*

$$\mathcal{O}(g(n)) = \{f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ s.t. } 0 \leq c \cdot g(n) \leq f(n),\ \forall\ n \geq n_0\}.$$

**Definition A.1.11** (Big-$\Theta$ notation $\Theta(\cdot)$)**.** *We use big-$\Theta$ notation $\Theta(\cdot)$ to denote asymptotic tight bound. Formally, for a given function $g(n)$, we denote by $\Theta(g(n))$, pronounced "big-theta of g of n", for the set of functions:*

$$\Theta(g(n)) = \{f(n) : \exists \text{ positive constants } c_1,\ c_2 \text{ and } n_0 \text{ s.t. } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),\ \forall\ n \geq n_0\}.$$

**Remark.**

- *There are mainly two criteria of complexity analysis: the worst case analysis and the average analysis.*
- *The worst case analysis is the more common one, therefore the notation that we use the most in analysis is the over approximation $\mathcal{O}(\cdot)$.*
- *The reason is that:*
  1. *for the worst case analysis, it is easier for us to directly calculate an upper bound, even though it may not be tight;*
  2. *for the average analysis, we have to enumerate and sum up all the cases to take the average, which is much harder or even impossible.*
- *When the complexity is the summation of several terms, we only care about the largest term that dominates, while omitting the other smaller terms.*
- *Take the linear complexity $\mathcal{O}(n+m)$as an example, where $n$ is the number of nodes and $m$ is the number of edges. This complexity is often simplified as $\mathcal{O}(m)$:*
  1. *If $m \gg n$, i.e., $m$ is much larger than $n$, then $m$ is the dominating term. Therefore, $\mathcal{O}(m+n) = \mathcal{O}(m)$.*
  2. *Since $m \geq n$ always holds, and $m + n \leq 2m$, it follows that $\mathcal{O}(m+n) \leq \mathcal{O}(2m) = \mathcal{O}(m)$.*
- *The constant $c$ is often neglected in theory, but can be important in practice.*
- *Even programs of the same order of complexity can have a big difference in the performance in practice. The reason is that the constant $c$ may differ a lot, even up to some order of complexity.*

- *The motivation to improve an algorithm when the order of complexity is hard to reduce is to minimize the constant c.*

## A.2 Sorting and Searching

They are two of the things that a computer program does the most, and are most basic and important.

**Sorting**

Since sorting is not quite related in this thesis, we just list a few well-known results here for reference.

1. The theoretical lower bound for sorting an arbitrary list of $n$ numbers is $\Omega(n \log n)$.
2. Sorting can increase the efficiency of searching: for an arbitrary list of $n$ numbers, searching an element is of both worst case and average case $\mathcal{O}(n)$; for an sorted list of $n$ numbers in $\Omega(n \log n)$ time, searching an element is of both worst case and average case $\mathcal{O}(\log n)$.
3. The 3 efficient sorting algorithms that meet this bound are quicksort (qsort), merge sort and heap sort.
4. Each algorithm has pros and cons with respect to time, space, and stability.
5. These algorithms often perform poorly on already sorted data or almost sorted data, which is quite practical in real-world data.
6. Therefore, we may need to combine several sorting algorithms together to create more sophisticated algorithms to expect better performance in practice.

**Searching**

The searching problem originates from graph traversal, which refers to the visitation of each node in the graph. The two most common searching algorithms are BFS and DFS.

In a single node graph traversal starting from $s$, both searching algorithms can find all the nodes that are reachable from $s$, where each node and each edge can be visited at most once. Therefore, the time complexity for performing such algorithms are both linear w.r.t. the number of nodes $n$ and the number of edges $m$, i.e., $\mathcal{O}(n + m)$. The difference is just the traversal order. Most algorithms in this thesis are based on the variation of BFS, so less will be discussed w.r.t. DFS.

## A.3 Breadth First Search (BFS)

The Breadth First Search is a searching algorithm that is breadth oriented. It traverse the graph layer by layer. Therefore, it has a nice property that it not only gives a path from the starting node(s) to a traversed node, but that path is also a shortest one. Here we use "a" shortest one instead of "the" shortest one for the reason that such shortest paths may not be unique.

BFS uses a data structure called queue in implementation. This data structure is an analogy to the queue in real life. For example, we stand in a queue when waiting for a bus. One who stands in the queue earlier gets on the bus earlier. Such property of a queue can be summarized as first in first out (FIFO). We usually use an horizontal array to imitate this process, where we delete from the head and insert from the tail.

Before we get into BFS, we introduce the data structure and some terminologies related to perform this algorithm first.

**Definition A.3.1** (queue)**.** *A queue $\mathcal{Q}$ is a linear data structure with the property FIFO. Some terminologies related to queue are:*

- *enqueue: add/ insert an item to the end of the queue;*
- *dequeue: pick out/ delete the first item of the queue;*
- *head: the position of the first item of the queue, i.e., where we dequeue;*
- *tail: the position after the last item of the queue, i.e., where we enqueue.*

### A.3.1 Traversing a Graph from a Single Node Using BFS

One of the simplest implementation of BFS is the graph traversal from a single starting node $s$. This is to find all the nodes that are reachable from $s$. Note that this process may not traverse the whole graph if there exists a node that is not reachable from $s$.

**Remark.** *There are several functions that we can add to this single node graph traversal:*

- *Label each of the traversed node with a number that indicates the least number of steps to reach this node from the starting node $s$, i.e., its layer. $s$ is labelled with number 0.*

1. *When traversing the graph using BFS, this labelling also indicates the smallest labelling. For any node $v$ traversed in this single node graph traversal, the path from $s$ to $v$ is a shortest path. This can be proved by Mathematical Induction.*
2. *Since the layer of each traversed node indicates the least number of steps to reach this node from the starting node $s$, the layer of each traversed node is unique. Therefore, the nodes in different layers form a partition of the traversed graph.*
3. *If we only care about whether a node is traversed or not, we can just label the node as visited or unvisited.*

- *Label each of the traversed node with a strategy/ action/ previous step/ path/ parent that indicates how this node is reached from a node in the previous layer.*

1. *The process of implementing BFS search forms a tree structure, where the starting node $s$ is the root node.*
2. *The traversed nodes form the leaf nodes of the tree.*
3. *On each traversed node $v$, we record the node in the previous layer that first reach this node as $u$.*
   - *In fact, $u$ is the parent node of $v$, and $u$ is unique to $v$, whereas $u$ can have several children nodes $v$'s.*
   - *The layer of a child node is one more than that of its parent node.*
   - *Using a child node to record its unique parent node is actually the reverse storage of a tree.*
   - *All the traversed $(u, v)$ pairs form the branches of the tree.*
   - *Instead of just recording the node $u$ in the previous layer that first reach this node $v$ as parent, we can also record all the nodes $u's$ in the previous layer that reach this node $v$ as parent, which will give us all the shortest paths from $s$ to $v$. The trade-off is a more complicated data structure to record all the parents of $v$ and the tree structure is broken, but is still a DAG.*
4. *Therefore, a BFS searching tree is formed by these leaf nodes and branches.*
5. *The path from the starting node $s$ to any traversed node $v$ can be constructed by a backward recursive algorithm 12 path$(v)$. We recursively call this path$(v)$ to construct a path from $s$ to $v$. Each node records its previous step. In other words, each child node records its parent node.*

### Algorithm

In this subsubsection, we will give an algorithm to implement BFS single node graph traversal and a follow up recursive algorithm for path construction.

### Sketch of the Algorithm

When we do a single node graph traversal using BFS from $s$, we set two marks to each node $v$: flag and parent. $v.flag$ marks whether that node is visited or not, it can also indicate the number of layer that $v$ is on if required; $v.parent$ marks the unique parent node of $v$, which is used in the recursive algorithm 12 $path(v)$ to construct a shortest path after implementing BFS.

We start our algorithm by initializing our two marks $v.flag$ and $v.parent$ as -1. $v.flag$ equals to -1 implies $v$ hasn't been visited yet, and it is on layer "infinity". $v.parent$ equals to -1 implies $v$ has no parent node. To set up the queue $\mathcal{Q}$, we set $s.flag$ as 0, which implies $s$ is visited and it is on layer 0. Then we add $s$ into the queue.

Next we formally proceed our search by repeating the following process until the queue is empty:

1. pick the currently first node $v$ out of the queue;
2. print $v$ to present a BFS traversal order;
3. for each unvisited successor $v'$ of v:
   (a) mark $v'$ as visited, set its layer as one more than that of $v$;
   (b) set $v$ as the parent node of $v'$;
   (c) add $v'$ to the end of the queue.

We only add node that hasn't been visited into the queue, which would save us some

operations and guarantee that each node can be enqueued at most once.

**Algorithm 11:** BFS Single Node Graph Traversal

**Input:**
1. $G = (V, E)$.
2. Starting node $s$.

**Output:**
1. The traversal order of the graph.

```
1 Initialization: Let Q be a queue, V.flag = −1, V.parent = −1;
2 for (s.flag = 0, Q.enqueue(s); Q is not empty; ) do
3     for (v = Q.dequeue( ), print v; all v' ∈ Post(v); ) do
4         if v'.flag == −1 then
5             Q.enqueue(v'), v'.flag = v.flag + 1, v'.parent = v;
```

**Result:**
1. Nodes traversed: $\mathcal{W} = \{v | v.flag > -1\}$;
2. If $v.parent > -1$, then $v.parent$ is the unique parent node of $v$.

### Path Construction

We can construct a shortest path from $s$ to any traversed node $t$ after applying BFS by calling the recursive algorithm 12 $path(t)$, which back track from $t$ to $s$ through the chain of the unique parent node recorded at each node.

In a recursive step at node $v$:

1. if $v$ has a parent node, then we recursively call its parent node first;
2. print the step number $v.flag$ and node $v$;

This is a post order, i.e., we print $v$ after its parent is printed. To construct a path

from $s$ to $t$, we simply call the recursive function $path(t)$, which is a shortest path.

**Algorithm 12:** Path Construction of Single Node Graph Traversal

**Input:**
1. BFS traversed graph $G = (V, E)$.
2. Starting node $s$.
3. Terminating node $t$ that we would like to search for a path from $s$.

**Output:**
1. The path from $s$ to $t$.

```
1 The recursive algorithm path(v) is defined as:
2 path(v){
3 if v.parent ≥ 0 then path(v.parent);
4 print step # v.flag : v;
5 }
```

**Result:**
1. This function $path(v)$ is based on the BFS graph traversal.
2. To construct a path from $s$ to $t$, we simply call the recursive function $path(t)$, which is a shortest path.
3. $v.flag$ indicates which intermediate step $v$ is at in the path.

### Proof

In this subsubsection, we show that BFS produces a shortest path by mathematical induction.

When implementing the BFS, we record the number of layers $n$ of the traversed nodes, which indicates the least number of steps for $s$ to reach a certain node. The starting node $s$ is on layer 0, and the nodes that are not searched are on layer "infinity". For some node $u$ on the $k^{th}$ layer, if a node $v$ is searched from $u$, then $v$ is on one more layer than $u$, i.e., $v$ is on the $(k+1)^{th}$ layer. We also record the unique parent node $u$ of $v$ at $v$.

**Theorem A.3.1.** *Traversing a graph from a single node s using BFS searches all the nodes that are reachable from s. For any node t that is searched in the BFS, a path from s to t is constructed, and that path is a shortest path.*

This is how our proof goes:

1. show BFS provides a shortest path by mathematical induction;

2. show BFS searches all the nodes that are reachable from $s$ by the finite property of the graph;

3. show how a shortest path is constructed by BFS using recursion.

**Proof by Mathematical Induction**

**Statement:** It takes at least $n$ steps for the starting node $s$ to reach a node on the $n^{th}$ layer.

**Base Case:** $n = 0$. The starting node $s$ is on the $0^{th}$ layer, so it takes at least 0 steps to reach itself.

**Induction Hypothesis:** $n = k$. Suppose it takes at least $k$ steps for the starting node $s$ to reach some node $u$ on the $k^{th}$ layer.

**Induction step:** Then for any node $v$ that is searched from $u$, $v$ is on one layer more than $u$. Therefore, $v$ is on the $(k+1)^{th}$ layer. Before $v$ is searched from $u$, $s$ can not reach $v$ within finite steps, therefore $s$ can not reach $v$ within $k$ steps. Also, there exists a path $s \dots uv$ that connects $s$ and $v$, and that it takes at least $k$ steps for $s$ to reach $u$ and 1 step for $u$ to reach $v$. Therefore, it takes at least $k+1$ steps for $s$ to reach some node $v$ on the $(k+1)^{th}$ layer.

We conclude that it takes at least $n$ steps for the starting node $s$ to reach a node on the $n^{th}$ layer. This ends the mathematical induction.

Since the graph is finite, this BFS process would terminate eventually, and the nodes that are still not searched at that point are the ones that are not reachable from $s$. The reason is that these nodes are on layer "infinity", which takes at least infinitely many steps for $s$ to reach them. In other words, these nodes are not reachable from $s$. This proves that BFS searches all the nodes that are reachable from $s$.

Since we record the unique parent node of each traversed node at that node, we can therefore trace back from any traversed node $t$ to $s$, which constructs a path from $s$ to $t$. Suppose $t$ is on the $n^{th}$ layer, we have shown by induction that it takes at least $n$ steps for $s$ to reach $t$, therefore the path from $s$ to $t$ constructed by BFS is a shortest path.

**Complexity Analysis**

In this subsubsection, we will show some complexity analysis for the BFS single node graph traversal and the path construction.

For a graph $G = (V, E)$, $|V| = n$ is the number of nodes, $|E| = m$ is the number of edges. $m_i$ is the number of out-going edges of node $i$, where $1 \leq i \leq n$. The relation between $n$ and $m$ is that:

$$\sum_{i=1}^{n} m_i = m \tag{A.1}$$

A hidden property of graph traversal is that, each node and each edge is visited at most once with the merit of marking each traversed node, so that no node or edge would be searched repeatedly. Therefore, the size of the graph traversal is upper bounded by the size of the entire graph $G = (V, E)$. The time and space complexity of graph traversal are both linear w.r.t. $n$ and $m$.

Time Complexity: $\mathcal{O}(n + m)$.

Space Complexity: $\mathcal{O}(n + m)$.

The size of the queue $\mathcal{Q}$ is $\mathcal{O}(n)$, because each node can be enqueued at most once.

Time Complexity for path construction: $\mathcal{O}(n)$.

### A.3.2 Traversing a Graph from a Set of Nodes using BFS

Instead of traversing a graph using BFS from a single initial node $s$, we can also start the BFS from a set of initial nodes $S$. The only difference from algorithm 11 is that, we enqueue the set of nodes $S$ instead of a single node $s$ initially, and that all the nodes in $S$ are labelled as layer 0.

If we would like to traverse the entire graph using BFS, we can simply apply the single node graph traversal multiple times until all the nodes in the graph are visited. The time and space complexity for traversing the entire graph is still $\mathcal{O}(n + m)$.

### A.3.3 Heuristic and Optimization of BFS

In this subsection, we will discuss some higher level ideas of the heuristic and optimization of BFS.

#### Bidirectional BFS

In this subsubsection, we will discuss bidirectional BFS, which is a heuristic optimization of BFS.

BFS originates from a searching problem: suppose we consider our graph as a maze, where $s$ is the starting point and $t$ is the terminal point. There are three natural questions to ask:

1. Determine whether there exists a path from $s$ to $t$;
2. if exists, provide such a path;
3. moreover, find a shortest one.

The first question asks about the existence of a path, the second one asks about finding such a path if exists, and the third one is a follow up optimization problem of finding a shortest one.

From the previous analysis, we already know that BFS is competent of solving all these three problems. Here we would like to further introduce some heuristic ideas to optimize BFS. When trying to find a path through a maze, we may get stuck in the middle of the maze starting from $s$. An intuitive idea to resolve this is to back track from $t$. In this way, the problem is greatly simplified.

Similarly, when applying BFS to do a single node graph traversal from $s$, we can expect the size of the layer to increase fast, which is likely to be an exponential growth. One way to resolve this is to apply BFS layer by layer between $s$ and $t$ alternatingly, until two searches intersect. This is the bidirectional alternating search. By searching through two ends instead of one, we can expect the number of traversed nodes to reduce a considerable amount.

A more detailed approach from simply alternatingly applying BFS from two ends is that, we always apply BFS on the side where the number of nodes on the current layer is smaller. This is the bidirectional balanced search. With this control, the number of traversed nodes can be further reduced. Note that for all these three BFS methods, the number of total layers are the same, i.e., the length of the paths are the same, all being the shortest. The difference is that the number of traversed/ searched nodes is decreasing.

In an undirected graph, we can apply the bidirectional searches directly; in a directed graph, we search from $s$ through out-going edges and search from $t$ through in-going edges. We can expect the algorithms for searching and path construction to be much more complicated when applying a bidirectional BFS.

#### Optimizations with BFS

In this subsubsection, we will discuss some of the optimization problems with BFS.

Finding an optimal path through a maze can be more complicated:

- adding some obstacles;
- adding some special rules;
- adding some rewards (positive weights);

  We can apply Dijkstra's algorithm to solve a single-source shortest path problem with non-negative weights in $\mathcal{O}(n^2)$ time and $\mathcal{O}(n+m)$ space.

- adding some costs (negative weights).

  We can apply Floyd's algorithm to solve a shortest path problem with positive or negative weights with no negative cycles for each pair of nodes in the graph in $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space.

  We can apply Bellman-Ford algorithm to solve a single-source shortest path problem with positive or negative weights and with negative cycles in $\mathcal{O}(nm)$ time and $\mathcal{O}(n+m)$ space.

These weighted variables form an optimization problem, which involves the idea of dynamical programming to solve. However, all these variants of a searching problem are based on the simple linear searching method BFS.

## A.4 Depth First Search (DFS)

The Depth First Search is another searching algorithm that is depth oriented. It traverse the graph one way down, as deep as possible, until no further path exists. Then it back track and go along the first unsearched path. This process continues until no more node is reachable. Since DFS also enumerates all the possible paths, it can give a path from the starting node(s) to a traversed node, but that path is not likely to be a shortest one.

DFS uses a data structure called stack in implementation. This can be understood as a pile of books in a box: we have to remove the books on top before we can get the books at the bottom. Such property of a stack can be summarized as first in last out (FILO). We

usually use an vertical array to imitate this process, where all operations such as inserting or deleting can be implemented from the top only.

A more common approach, however, is to write a recursive algorithm, which is much shorter, easier to understand and easier to write. The principle behind the recursive algorithm is still using the data structure stack. A potential problem that a recursive algorithm may cause is that, if the recursive steps are too deep, it may cause a stack overflow, which means the system provided stack is not large enough. In that situation, we have to explicitly implement the data structure stack.

Recall from the searching problem in section A.3.3, we can also determine whether there exists a path from $s$ to $t$ using DFS, and provide such a path if exists, yet there is no guarantee that such a path is a shortest one. Therefore we have to use BFS instead of DFS even though implementing DFS with a recursive algorithm is much easier. However, the worst case of BFS and DFS are the same. That is, if a path from $s$ to $t$ doesn't exist, then we have to implement a single node graph traversal to traverse all the nodes from $s$ using BFS or DFS. The cost of the two searches are both linear, which is $\mathcal{O}(m+n)$. The difference is just the traversal order.

Despite DFS can't be used to find a shortest path as BFS, it has other useful applications that BFS can't competent such as finding all the SCCs, which we will talk about in the next section A.5.

Before we get into DFS, we introduce the data structure and some terminologies related to perform this algorithm first.

**Definition A.4.1** (stack)**.** *A stack $\mathcal{S}$ is a linear data structure with the property FILO. Some terminologies related to stack are:*

- *push: add/ insert an item to the top of the stack;*
- *pop: pick out/ delete an item of the stack from the top;*
- *peek: the item on top of the stack.*

### A.4.1 Traversing a Graph from a Single Node Using DFS

DFS can also perform a graph traversal from a single starting node $s$. The difference between BFS and DFS to perform a single node graph traversal is the data structure and the traversal order.

### Algorithm

In this subsubsection, we will give three algorithms to implement DFS single node graph traversal. The path construction can be shared with the one in BFS.

### Sketch of the Algorithm

There are several approaches to perform a DFS:

1. recursive algorithm;
2. iterative algorithm that pushes a new node into the stack before checking visitation;
3. iterative algorithm that pushes a new node and its next edge to visit into the stack after checking visitation.

### First Version of DFS

The first algorithm is the easiest to implement, since the data structure of a stack has been implicitly taken care of by the recursive algorithm.

Similar to the single node graph traversal from $s$ using BFS, we set the same two marks to each node $v$ in our DFS: flag and parent. $v.flag$ marks whether that node is visited or not, it can also indicate the number of layer that $v$ is on if required; $v.parent$ marks the unique parent node of node $v$, which is used in the recursive algorithm 12 $path(v)$ to construct a path after implementing DFS.

We start our algorithm by initializing our two marks $v.flag$ and $v.parent$ as -1. $v.flag$ equals to -1 implies $v$ hasn't been visited yet, and it is on layer "infinity". $v.parent$ equals to -1 implies $v$ has no parent node.

Now we formally define our recursive algorithm 13 $DFS(v)$. For each recursive call on node $v$, we do the following process:

1. print $v$ to present a DFS traversal order;
2. for each unvisited successor $v'$ of v:
   (a) mark $v'$ as visited, set its layer as one more than that of $v$;
   (b) set $v$ as the parent node of $v'$;
   (c) call $DFS(v')$.

To implement this recursive DFS single node graph traversal algorithm 13 from $s$, we initialize $s.flag$ as 0 to imply that $s$ is visited and it is on layer 0. Then we simply call $DFS(s)$.

**Algorithm 13:** Recursive DFS Single Node Graph Traversal

**Input:**
1. $G = (V, E)$.
2. Starting node $s$.

**Output:**
1. The traversal order of the graph.

**Initialization:** Let $V.flag = -1, s.flag = 0, V.parent = -1$;
The recursive algorithm $DFS(v)$ is defined as:
$DFS(v)\{$
**for** (***print*** $v$; ***all*** $v' \in Post(v)$;) **do**
    **if** $v'.flag == -1$ **then**
        $v'.flag = v.flag + 1, v'.parent = v, DFS(v')$;
$\}$

**Result:**
1. We simply call $DFS(s)$ to run this algorithm;
2. Nodes traversed: $\mathcal{W} = \{v | v.flag > -1\}$;
3. If $v.parent > -1$, then $v.parent$ is the unique parent node of $v$.

**Second Version of DFS**

The second algorithm explicitly implements the operations on a stack. Unlike BFS which checks the visitation of a new node before adding it into the queue to avoid repetitive enqueueing, here we push a new node into the stack without checking its visitation in this version of iterative DFS algorithm, which results in repetitively pushing new nodes into the stack. This process presents the forward searching and the backward tracking, which is unavoidable in DFS. A trade-off from repetitively pushing new nodes into the stack is to record a node and its next edge to visit at each unit on the stack, which requires a more complicated data structure. That version of DFS will be presented in the next algorithm.

Since we push a new node into the stack without checking its visitation, our mark $v.flag$ can only record the visitation of a node $v$, but not its layer in this version of DFS algorithm. We mark $v$ as unvisited or visited by setting $v.flag$ as 0 or 1 respectively. $v.parent$ functions the same as before.

We start our algorithm by initializing our two marks $v.flag$ as 0 (unvisited) and $v.parent$ as -1 (no parent node). In addition, we explicitly set up the stack $\mathcal{S}$. We set

$s$ as the parent node of all its successors and then push $s$ into the stack.

Next we formally proceed our search by repeating the following process until the stack is empty:

1. take the peek node $v$ out of the stack;
2. if $v$ is unvisited:
   (a) print $v$ to present a DFS traversal order;
   (b) mark $v$ as visited;
   (c) for each unvisited successor $v'$ of $v$:
      i. set $v$ as the parent node of $v'$;
      ii. push $v'$ into the stack.

Although each node can be pushed into the stack more than once, we can still push the unvisited nodes only into the stack to save some operations.

**Algorithm 14:** Iterative DFS Single Node Graph Traversal without Iterator

**Input:**
1. $G = (V, E)$.
2. Starting node $s$.

**Output:**
1. The traversal order of the graph.

```
1 Initialization: Let S be a stack, V.flag = 0, V.parent = −1;
2 for (; all v' ∈ Post(s); ) do v'.parent = s;
3 for (S.push(s); S is not empty; ) do
4     v = S.pop( );
5     if v.flag == 0 then
6         print v, v.flag = 1;
7         for (; all v' ∈ Post(v); ) do
8             if v'.flag == 0 then v'.parent = v, S.push(v');
```

**Result:**
1. Nodes traversed: $\mathcal{W} = \{v | v.flag == 1\}$;
2. If $v.parent > -1$, then $v.parent$ is the unique parent node of $v$.

**Third Version of DFS**

The third algorithm is an explicit presentation of the first one. It checks the visitation of a new node before adding it into the stack. For each unit of the stack, instead of just recording a node in the previous algorithm, its next edge to visit is also recorded here. The information of the next edge to visit is used to avoid the repetitive searching of the edges while back tracking.

We start our algorithm by initializing $v.flag$ as -1 (unvisited) and $v.parent$ as -1 (no parent node), which is the same as the first algorithm. In addition, we explicitly set up the stack $\mathcal{S}$. We print $s$ and set $s.flag$ as 0, which implies $s$ is visited and it is on layer 0. Then we push $(s, s')$ as a unit into the stack, which indicates node $s$ and its next edge to visit $(s, s')$.

Next we formally proceed our search by repeating the following process until the stack is empty:

1. take the peek item $(v, v')$ out of the stack;
2. set $(v, v')$ as the current visiting edge of node $v$;
3. if $v$ has unvisited edge $(v, v'')$, then push the next edge to visit of $v$, i.e., $(v, v'')$ into the stack;
4. if $v'$ is unvisited:
   (a) rename $v'$ as $w$;
   (b) print $w$ to present a DFS traversal order;
   (c) mark $w$ as visited, set its layer as one more than that of $v$;
   (d) set $v$ as the parent node of $w$;
   (e) push the next edge to visit of $w$, i.e., $(w, w')$ into the stack;

The third algorithm reveals the mechanism of the implicit stack data structure of the first

recursive algorithm.

**Algorithm 15:** Iterative DFS Single Node Graph Traversal with Iterator

**Input:**
1. $G = (V, E)$.
2. Starting node $s$.

**Output:**
1. The traversal order of the graph.

**Initialization:** Let $\mathcal{S}$ be a stack, $V.flag = -1, V.parent = -1$;
A unit $(v, v')$ in the stack records a node $v$ and its next edge to visit $(v, v')$.
**for** ($\boldsymbol{print}$ $s, s.flag = 0, \mathcal{S}.push((s, s'))$; $\mathcal{S}$ *is not empty*;) **do**
    $(v, v') = \mathcal{S}.pop(\ )$;
    **if** $v$ *has unvisited edge* $(v, v'')$ **then** $\mathcal{S}.push((v, v''))$;
    **if** $v'.flag == -1$ **then**
        $w = v'$, **print** $w, w.flag = v.flag + 1, w.parent = v, \mathcal{S}.push(w, w')$;

**Result:**
1. Nodes traversed: $\mathcal{W} = \{v | v.flag > -1\}$;
2. If $v.parent > -1$, then $v.parent$ is the unique parent node of $v$.

**Remark.**
- *The second algorithm can not record the information of the layer of a traversed node since it pushes a new node into the stack before checking its visitation and its layer number is still undetermined.*
- *In the second algorithm, even though we may have nodes that are pushed into the stack more than once, it doesn't violate the traversal property that each node and each edge is visited at most once. The reason is that the repetitive pushing of a node into the stack represents the process of visiting different edges, yet each edge is still visited at most once.*
- *All of these three algorithms can construct a path by sharing the same algorithm 12 path(v) from BFS.*
- *The third algorithm is an explicit implementation of the first one, with three merits:*
    1. *an iterative algorithm runs faster than a recursive one;*
    2. *an iterative algorithm costs less memory than a recursive one;*
    3. *we can self-define a larger capacity to a stack if we define it explicitly.*

- *The DFS traversal order given by the first and the third algorithm are the same, which is different from that of the second algorithm.*

  *The reason is that, in the second algorithm, the edges are traversed in backward order, which presents the process of FILO.*

  *In the first and the third algorithm, the edges are traversed in forward order. This doesn't violate the property of FILO since the order between the nodes is still FILO.*

- *The DFS traversal order given by the first and the third algorithm has a name lexicographic order, also known as the dictionary order. That is, such order follows the rules of how words are ordered in a dictionary.*

#### Complexity Analysis

In this subsubsection, we will show some complexity analysis for the DFS single node graph traversal.

For a graph $G = (V, E)$, $|V| = n$ is the number of nodes, $|E| = m$ is the number of edges. Similar to the complexity analysis of BFS, DFS also possesses the property of graph traversal that each node and each edge is visited at most once. Also, the size of the graph traversal is upper bounded by the size of the entire graph $G = (V, E)$. It follows that the time and space complexity of graph traversal is linear with respect to $n$ and $m$.

Time Complexity: $\mathcal{O}(n + m)$.

Space Complexity: $\mathcal{O}(n + m)$.

The size of the stack $\mathcal{S}$ is $\mathcal{O}(n)$ in the first and the third algorithm, and is $\mathcal{O}(m)$ in the second algorithm, although the size of each unit of the stack is a little larger in the former than the later.

### A.4.2 Traversing a Graph from a Set of Nodes using DFS

Instead of traversing a graph using DFS from a single initial node $s$, we can also start the DFS from a set of initial nodes $S$ by pushing the set $S$ into the stack. However, there are some differences between BFS and DFS in this process:

- in BFS, when we enqueue the set of nodes $S$ instead of a single node $s$ into the queue initially, these nodes in $S$ are considered as partition layer 0;

- in DFS, pushing the set of nodes $S$ into the stack followed by performing DFS presents the same traversal order as performing the single node graph traversal one by one for each node in $S$, a result of the FILO property.

If we would like to traverse the entire graph using DFS, we can simply apply the single node graph traversal multiple times until all the nodes in the graph are visited. Finding all the SCCs of a graph is of a similar process with a DFS based algorithm.

The time and space complexity for traversing the entire graph is still $\mathcal{O}(n+m)$.

## A.5 Strongly Connected Component (SCC)

Connectivity is an important feature of a graph. It talks about whether two or more nodes are connected to each other. If so, then we can "shrink" these nodes into one which simplifies the graph.

In the directed graphs, a graph is said to be strongly connected if every node is mutually reachable. The entire graph can then be seen as one connected "big" node.

If a graph is not strongly connected, then it can be partitioned into subgraphs (A.1.3) of Strongly Connected Components (SCCs) that are themselves strongly connected.

The time complexity for performing algorithms to solve for SCCs are all linear w.r.t. the number of nodes $n$ and the number of edges $m$, i.e., $\mathcal{O}(n+m)$.

Since the algorithms for solving SCCs are not quite related to this thesis and are quite complicated, we only list the properties related to SCCs here without providing the algorithms.

**Definition A.5.1.** *(strongly connected) A directed graph is said to be strongly connected if for each pair of nodes in the graph, there are paths between them in both directions.*

*A pair of nodes in a directed graph is said to be strongly connected to each other if there are paths between them in both directions.*

**Definition A.5.2.** *(Strongly Connected Component) A Strongly Connected Component (SCC) of a directed graph is a maximal subgraph (A.1.3) that is strongly connected.*

**Definition A.5.3.** *(A Trivial Strongly Connected Component) A Strongly Connected Component (SCC) of a directed graph is trivial if it is a single node without self-loop (A.1.4).*

In other words, the binary relation of being strongly connected is an equivalence relation, and each strongly connected component is an equivalence class.

**Remark.**

- *SCCs are often partitioned in the pre-processing of a graph:*
  1. *After partitioning a directed graph $G$ into SCCs, if we contract each SCC into a single node, then the resulting graph is a Directed Acyclic Graph (DAG), the condensation of $G$, which results in a reduction in the size of the graph.*
  2. *A topological sort can then be performed in the DAG and produce a topological order, which will be discussed in the next section.*
  3. *Such order is an order of dependency that can avoid repetitive computing which may result in a reduction in the complexity of the algorithms applied to the graph.*
- *SCCs can be found by applying some DFS based algorithms such as Kosaraju's algorithm, Tarjan's strongly connected components algorithm and path-based strong component algorithm, with different features.*
- *All these algorithms are linear w.r.t. the number of nodes $n$ and the number of edges $m$ in both time and space complexity, i.e., $\mathcal{O}(n+m)$.*
- *The strong connectivity of a directed graph can also be tested using these algorithms.*

## A.6 Topological Sort

In this section, we will introduce topological sort and give two algorithms to perform it.

A topological sort or a topological ordering is a linear ordering of the nodes in a Directed Acyclic Graph (DAG). In such ordering, for each directed edge $(u, v) \in E$, $u$ comes before $v$.

Topological sort presents the dependency of the nodes in a DAG, and such sort may not be unique. A DAG can be constructed from a directed graph by contracting each SCC of the graph into a single node.

If we use a DAG to represent the flow diagram of a project, then a topological sort presents a possible working flow of the project, i.e., an order of which task should be done before another.

The complexity of algorithms may be reduced if performed in a topological sorted DAG. The reason is that the one-way order of a topological sort eliminates repetitive computing.

Here we present two algorithms to implement a topological sort. The time complexity are both linear w.r.t. the number of nodes $n$ and the number of edges $m$, i.e., $\mathcal{O}(n+m)$.

**Definition A.6.1.** *(DAG) A Directed Acyclic Graph (DAG) is a directed graph without cycles.*

**Definition A.6.2.** *(cycle) If there exists a cycle in a directed graph, then there exists a pair of nodes $u$ and $v$ such that there exist a path from $u$ to $v$ and a path $from$ $v$ to $u$.*

A topological sort can only be performed in a DAG. Otherwise if a cycle exists, then the one-way structure of a topological sort is violated.

### A.6.1 BFS Based Topological Sort

In the original definition of a topological sort, for each directed edge $(u, v)$ from $u$ to $v$, $u$ comes before $v$. That is, a node $v$ can be placed in the ordering only when it has no in-going edges $v \leftarrow u$, i.e., the in-degree of $v$ is 0. It follows that there are two hidden properties in a topological sort:

1. The first node is of in-degree 0;
2. the last node is of out-degree 0.

The implementation of a BFS based topological sort is based on the idea that the first node is of in-degree 0: we first enqueue all the nodes with in-degree 0, then we start a BFS and reduce the graph by deleting the visited nodes and edges of the graph, which will create more nodes with in-degree 0. We enqueue these new nodes with in-degree 0 and continue this process until the queue is empty.

Instead of enqueueing a new node when we first visit it in a BFS graph traversal, here we enqueue a new node when its in-degree reduces to 0. The difference just the rule to enqueue a new node: either there exists an edge, or the node is of in-degree 0.

#### Algorithm

In this subsubsection, we will give an algorithm to perform BFS based topological sort.

**Sketch of the Algorithm**

When we do a BFS based topological sort in a DAG, we start by setting the in-degree of each node in the graph. We set up the queue $\mathcal{Q}$ by enqueueing all the nodes with in-degree 0.

Next we formally proceed our search by repeating the following process until the queue is empty:

1. pick the currently first node $v$ out of the queue;
2. print $v$ to present a topological order;
3. for each successor $v'$ of v:
   (a) reduce the in-degree of $v'$ by 1;
   (b) if the in-degree of $v'$ reduces to 0, then enqueue $v'$.

We enqueue a new node when its in-degree reduces to 0.

**Algorithm 16:** BFS Based Topological Sort in a DAG

**Input:**
1. $G = (V, E)$.

**Output:**
1. A topological order of the graph.

```
1 Initialization: Let Q be a queue, set V.indeg;
2 for (Q.enqueue(V.indeg == 0); Q is not empty; ) do
3 |   for (v = Q.dequeue( ), print v; all v' ∈ Post(v); ) do
4 |   |   v'.indeg − −;
5 |   |   if v'.indeg == 0 then Q.enqueue(v') ;
```

**Result:**
1. This algorithm can also be used to check whether a directed graph is a DAG;
2. The topological order of a DAG may not be unique.

**Remark.**
- *This algorithm can also be used to check whether a directed graph is a DAG: if the number of nodes presented in the topological sort is fewer than the number of nodes of the graph, then the remaining graph has at least one non-trivial SCCs.*
- *Now we see that BFS can have different variations based on the enqueueing rules:*
  1. *if there exists an edge to reach a node in a graph traversal;*

2. *if for all edges to reach a node/ if the in-degree of a node reduces to 0 in a topological sort;*
3. *a combination of the two cases exist and for all in a two-player Reachability game which will be presented in chapter 5.*

#### Complexity Analysis

In this subsubsection, we will show some complexity analysis for the BFS based topological sort.

For a graph $G = (V, E)$, $|V| = n$ is the number of nodes, $|E| = m$ is the number of edges.

Since a BFS based topological sort is just a variation of BFS, the time and space complexity are both linear w.r.t. $n$ and $m$.

Time Complexity: $\mathcal{O}(n + m)$.

Space Complexity: $\mathcal{O}(n + m)$.

The size of the queue $\mathcal{Q}$ is $\mathcal{O}(n)$, because each node can be enqueued at most once.

### A.6.2 DFS Based Topological Sort

We can also use DFS to do a topological sort in a DAG.

The implementation of a DFS based topological sort is based on the idea that the last node is of out-degree 0: we recursively call DFS in a post order, i.e., print itself after searching its successors. The first printed node is therefore of out-degree 0. We call DFS multiple times until all the nodes in the graph are visited.

The graph must be a DAG, otherwise the recursive process will not terminate, even though this can also be used to check whether a directed graph is a DAG.

The printed order is a reversed topological order of the graph. If we would like a forward order of a topological sort, we can first apply multiple DFS to traverse the entire graph and store these sorted nodes in a stack, and then print them all together at the end. The FILO property of a stack reverses the reversed topological order and therefore produces a forward topological order. In other words, topological sorting between in-degree 0 and out-degree 0 is of reversed order.

### Algorithm

In this subsubsection, we will give an algorithm to perform DFS based topological sort.

**Sketch of the Algorithm**

When we do a DFS based recursive topological sort in a DAG, we mark $v$ as unvisited or visited by setting $v.flag$ as 0 or 1 respectively. We initialize each node as unvisited.

Now we formally define our recursive algorithm 17 $DFS(v)$. For each recursive call on node $v$, we do the following process:

1. for each unvisited successor $v'$ of $v$, call $DFS(v')$;
2. mark $v$ as visited;
3. print $v$ to present a reversed topological order;

We call $DFS(v)$ multiple times on unsorted nodes until the entire graph is traversed. The printed order is a reversed topological order of the graph.

A new node is printed when its out-degree reduces to 0.

---

**Algorithm 17:** DFS Based Recursive Topological Sort in a DAG

---

**Input:**
1. $G = (V, E)$.

**Output:**
1. A reversed topological order of the graph.

**Initialization:** Let $V.flag = 0$;
The recursive algorithm $DFS(v)$ is defined as:
$DFS(v)\{$
**for** (; ***all*** $v' \in Post(v)$;) **do**
  **if** $v'.flag == 0$ **then** $DFS(v')$;
$v.flag = 1$, **print** $v$;
$\}$

**Result:**
1. We call $DFS(v)$ on unsorted nodes until the entire graph is traversed;
2. If the recursion doesn't terminate, then the directed graph isn't a DAG;
3. The topological order of a DAG may not be unique.

---

**Remark.**
- *This algorithm can also be used to check whether a directed graph is a DAG: if the recursion doesn't terminate, then the remaining graph has at least one non-trivial SCCs. However, this is not recommended since the algorithm is supposed to be designed in a DAG and we would like our program to terminate.*
- *Topological order between in-degree 0 and out-degree 0 is of reversed order.*
- *Now we see that DFS can also have different variations:*
  1. *pre order: print before searching successors in a graph traversal;*
  2. *post order: print after searching successors in a topological sort.*

#### Complexity Analysis

In this subsubsection, we will show some complexity analysis for the DFS based topological sort.

For a graph $G = (V, E)$, $|V| = n$ is the number of nodes, $|E| = m$ is the number of edges.

Since a DFS based recursive topological sort is just a variation of DFS, the time and space complexity are both linear w.r.t. $n$ and $m$.

Time Complexity: $\mathcal{O}(n + m)$.

Space Complexity: $\mathcal{O}(n + m)$.

The size of the implicit stack $\mathcal{S}$ is $\mathcal{O}(n)$, because each node can be visited at most once.

## A.7 Games

In this section, we will discuss graph based problems in the form of games.

Now we consider some of the games on a classic graph. Games are in a more vivid and interactive way of describing a problem.

Different games have different goals for winning. For different games, the target set $T \subseteq V$ have different meanings. For each of the game, we want to solve for the winning set $W \subseteq V$ and the winning strategy/ controller for the winning set of the game.

All the problems are defined on graphs with in-going edges because of the properties of the games we consider. The controller of the games we consider are all local, i.e., in order

to create a strategy to win a game, we just need to record a strategy at each node we are currently at, without considering the structure of the entire graph.

Since we would like to preserve some optimality w.r.t. the shortest path, the algorithms we present here are all variations of BFS instead of DFS.

## A.8 Reachability

In this section, we will discuss the reachability game.

Reachability is among one of the simplest specifications. The time complexity is linear w.r.t. the number of nodes $n$ and the number of edges $m$, i.e., $\mathcal{O}(n+m)$. However, it serves as an atomic algorithm for many other more complicated games.

### A.8.1 Problem Description

**Problem Description:** Given a target set $T \subseteq V$, we want to find all the nodes that can reach $T$, either including or excluding $T$.

Recall from the graph traversal problem starting from a set of nodes $S$ using BFS in subsection A.3.2, if a node $v$ is on layer $i$, it implies $S$ can reach $v$ in at least $i$ steps. Similarly, in a reachability game, if a node $v$ is on layer $i$, it implies $v$ can reach $T$ in at least $i$ steps.

Here the reachability game is just a reverse problem of the graph traversal: graph traversal is to find all the nodes that $S$ can reach, and reachability game is to find all the nodes that can reach $T$. In implementation, such difference is simply by changing a graph with out-going edges to a graph with in-going edges.

The two variations of the reachability game correspond to the set of nodes that can reach $T$ in at least 0 steps and the set of nodes that can reach $T$ in at least 1 steps. They are quite similar in implementation, with different applications.

Here we define the set difference operator / or $-$.

**Definition A.8.1** (set difference operator)**.** *For two sets $A$ and $B$, the set difference of $A$ and $B$ is defined as $A/B = A - B = \{x | x \in A \text{ and } x \notin B\}$.*

In other words, the elements in the set difference of $A$ and $B$ are the ones in $A$ but not in $B$.

### A.8.2 Reach T in at least 0 steps

In this subsection, we will discuss the reachability game in at least 0 steps.

The set of nodes that can reach $T$, including $T$ is the same as the set of nodes that can reach $T$ in at least 0 steps. Here the nodes in $T$ is considered as in the reachable set of $T$.

#### Algorithm

In this subsubsection, we will give an algorithm to perform reachability in at least 0 steps.

**Sketch of the Algorithm**

When we search for the nodes that can reach $T$ in at least 0 steps, we set three marks to each node $v$: $isT$, $flag$ and $strategy$. $v.isT$ marks whether $v$ is in $V$ or not; $v.flag$ marks whether $v$ is visited or not, it can also indicate the number of layer that $v$ is on if required; $v.strategy$ marks the local strategy of $v$, which is used in the iterative algorithm 19 $path(v)$ to construct a shortest path after implementing the reachability algorithm.

We start our algorithm by initializing our three marks. We mark $v.isT$ as 1 for the nodes in $T$ and 0 for the nodes not in $T$, i.e., in $V/T$. We mark $v.flag$ as -1 to indicate $v$ hasn't been visited yet, and it is on layer "infinity". We mark $v.strategy$ as -1 to indicate $v$ hasn't assigned a strategy yet. To set up the queue $\mathcal{Q}$, we set $T.flag$ as 0, which implies nodes in $T$ is visited and can reach $T$ in at least 0 steps. Then we add all the nodes in $T$ into the queue.

Next we formally proceed our search by repeating the following process until the queue is empty:

1. pick the currently first node $v$ out of the queue;
2. print $v$ to indicate $v$ is in the set that can reach $T$ in at least 0 steps;
3. for each unvisited predecessor $v'$ of $v$:
   (a) mark $v'$ as visisted, set its layer as one more than that of $v$;
   (b) set $v$ as the local strategy of $v'$;
   (c) add $v'$ to the end of the queue.

Here we initially mark the nodes in $T$ as visited and add them into the queue. The set that can reach $T$ in at least 0 steps are the nodes labeled with $v.flag \geq 0$, where the nodes labeled with $v.flag$ equal to 0 are the nodes in $T$, and the nodes labeled with $v.flag > 0$ are the nodes not in $T$, but can reach $T$ through the strategy $v.strategy$.

**Algorithm 18:** Reachability in at least 0 steps

**Input:**
1. $G = (V, E)$;
2. $T \subseteq V$;

**Output:**
1. Winning set $W$ of $V$;
2. Winning strategy of each state in $W$;

**Initialization:** Let $\mathcal{Q}$ be a queue,
$T.isT = 1, (V/T).isT = 0,\ V.flag = -1, V.strategy = -1$;
**for** $(T.flag = 0, \mathcal{Q}.enqueue(T); \mathcal{Q}$ *is not empty;*$)$ **do**
  **for** $(v = \mathcal{Q}.dequeue(\ )$,***print*** $v$; ***all*** $v' \in Pre(v);)$ **do**
    **if** $v'.flag == -1$ **then**
      $\mathcal{Q}.enqueue(v'),\ v'.flag = v.flag + 1, v'.strategy = v$;

**Result:**
1. $\mathcal{W} = \{v | v.flag \geq 0\}$;
2. If $v.flag == 0$, $v \in T$; if $v.isT == 1$, $v \in T$;
3. If $v.flag \geq 0$, then a path from $v$ can be constructed to reach a node in $T$ through $v.strategy$ in at least 0 steps;
4. If $v.flag == -1$, then $v$ is not in $T$ and can't reach $T$.

### Path Construction

We can construct a shortest path from a node $t$ in the reachable set to a node in $T$ after applying the reachability algorithm by calling the iterative algorithm 19 $path(t)$, which links $t$ to a node in $T$ through a chain of the strategy recorded at each node. Since it is in forward direction, this path construction algorithm can either be recursive or iterative.

In the iterative algorithm starting from node $v$, we repeat the following process:

1. print the step number $v.flag$ and node $v$;
2. if $v$ is in $T$, then break the loop;
3. assign $v.strategy$ to $v$.

This path construction is in pre order, i.e., we print $v$ before its strategy is printed. To construct a path from $t$ to a node in $T$, we simply call the iterative function $path(t)$, which is a shortest path.

**Algorithm 19:** Path Construction of Reachability in at Least 0 Steps

**Input:**

1. Reachability searched graph $G = (V, E)$.
2. Target set $T \subseteq V$.
3. Node $t$ in reachable set that we would like to search for a path to reach $T$ in at least 0 steps.

**Output:**

1. The path from $t$ to a node in $T$.

```
1 The iterative algorithm path(v) is defined as:
2 path(v){
3 for (;;) do
4 |   print step # v.flag : v;
5 |   if v.isT == 1 then break;
6 |   v = v.strategy;
7 }
```

**Result:**

1. This function $path(v)$ is based on the reachability search in at least 0 steps.
2. To construct a path from $t$ to $T$, we simply call the iterative function $path(t)$, which is a shortest path.
3. $v.flag$ indicates which intermediate step $v$ is at in the path.

### A.8.3 Reach T in at least 1 steps

In this subsection, we will discuss the reachability game in at least 1 steps.

The set of nodes that can reach $T$, excluding $T$ is the same as the set of nodes that can reach $T$ in at least 1 steps. Here the nodes in $T$ is not considered as in the reachable set of $T$. This design is used when we would like to visit $T$ multiple times, or infinitely many times (Büchi game).

#### Algorithm

In this subsubsection, we will give an algorithm to perform reachability in at least 1 steps.

**Sketch of the Algorithm**

When we search for the nodes that can reach $T$ in at least 1 steps, the set up is similar to that of reaching $T$ in at least 0 steps. We set three marks to each node $v$: $isT$, $flag$ and $strategy$. $v.isT$ marks whether $v$ is in $V$ or not; $v.flag$ marks whether that node is visited or not, it can also indicate the number of layer that $v$ is on if required; $v.strategy$ marks the local strategy of $v$, which is used in the iterative algorithm 21 $path(v)$ to construct a shortest path after implementing the reachability algorithm.

Here we add another two indicators $marker$ and $cnt$ to determine the layer number of each node. $marker$ is a position indicator of the queue. Recall that BFS is searching layer by layer, and $marker$ partitions the nodes in each layer. $cnt$ is a layer indicator that indicates which layer the to be enqueued nodes are on. This two indicators are essential because otherwise the layer number for the nodes in $T$ may be wrong as a result of repetitive counting.

We start our algorithm by initializing our three marks. We mark $v.isT$ as 1 for the nodes in $T$ and 0 for the nodes not in $T$, i.e., in $V/T$. We mark $v.flag$ as -1 to indicate $v$ hasn't been visited yet, and it is on layer "infinity". We mark $v.strategy$ as -1 to indicate $v$ hasn't assigned a strategy yet.

To set up the queue $\mathcal{Q}$, we set $T.flag$ as 0 and then we add all the nodes in $T$ into the queue. We set $marker$ to the position of the tail of the queue, which is a partition between the nodes in the layer 0 and the layer 1. We set $cnt$ to be 1 which indicates the already enqueued nodes in $T$ are on layer 0, and the nodes to be enqueued are on layer 1.

Now, $v.flag$ equals to -1 implies $v$ is not in $T$ and hasn't been visited yet. $v.flag$ equals to 0 implies $v$ is in $T$ and hasn't been visited yet. Note that nodes that can reach $T$ in at least 1 steps should have $v.flag \geq 1$.

Next we formally proceed our search by repeating the following process until the queue is empty:

1. if the position indicator $marker$ is at the position of the head of the queue:
   (a) set $marker$ to the position of the tail of the queue;
   (b) accumulate the layer indicator $cnt$ by 1;
2. pick the currently first node $v$ out of the queue;
3. for each unvisited predecessor $v'$ of $v$:
   (a) print $v'$ to indicate $v'$ is in the set that can reach $T$ in at least 1 steps;

(b) mark $v'$ as visisted, set its layer to be $cnt$;

(c) set $v$ as the local strategy of $v'$;

(d) if $v'$ is not in $T$, then add $v'$ to the end of the queue.

Here we initially mark the nodes in $T$ as unvisited and add them into the queue. After that, we only enqueue unvisited nodes that are not in $T$, which guarantees that each node in $V$ is enqueued at most once.

- The nodes labeled with $v.flag$ equal to -1 are the ones not in $T$, and can't reach $T$ in at least 1 steps;
- the nodes labeled with $v.flag$ equal to 0 are the ones in $T$, but can't reach $T$ in at least 1 steps;
- the nodes labeled with $v.flag \geq 1$ are the ones that can reach $T$ in at least 1 steps through the strategy $v.strategy$.

**Algorithm 20:** Reachability in at least 1 steps

**Input:**

1. $G = (V, E)$;
2. $T \subseteq V$;

**Output:**

1. Winning set $W$ of $V$;
2. Winning strategy of each state in $W$;

**Initialization:** Let $\mathcal{Q}$ be a queue, $marker$ be a position indicator, $cnt = 1$ be a layer indicator, $T.isT = 1, (V/T).isT = 0,\ V.flag = -1, V.strategy = -1$;

**for** $(T.flag = 0, \mathcal{Q}.enqueue(T), marker = \mathcal{Q}.tail(\ );\ \mathcal{Q}$ *is not empty;*$)$ **do**

  **if** $\mathcal{Q}.head(\ ) == marker$ **then** $marker = \mathcal{Q}.tail(\ ), cnt++$;

  **for** $(v = \mathcal{Q}.dequeue(\ );$ ***all*** $v' \in Pre(v);)$ **do**

    **if** $v'.flag < 1$ **then**

      **print** v', $v'.flag = cnt, v'.strategy = v$;

      **if** $v'.isT == 0$ **then** $\mathcal{Q}.enqueue(v')$;

**Result:**

1. $\mathcal{W} = \{v | v.flag \geq 1\}$;
2. If $v.isT == 1$, $v \in T$;
3. If $v.flag == 0$, $v \in T$, but $v$ can't reach $T$ in at least 1 steps;
4. If $v.flag \geq 1$, then a path from $v$ can be constructed to reach a node in $T$ through $v.strategy$ in at least 1 steps;
5. If $v.flag == -1$, then $v$ is not in $T$ and can't reach $T$.

### Path Construction

We can construct a shortest path from a node $t$ in the reachable set to a node in $T$ after applying the reachability algorithm by calling the iterative algorithm 21 $path(t)$, which links $t$ to a node in $T$ through a chain of the strategy recorded at each node. Since it is in forward direction, this path construction algorithm can either be recursive or iterative.

In the iterative algorithm starting from node $v$, first we print the step number $v.flag$ and node $v$, then we repeat the following process:

1. assign $v.strategy$ to $v$
2. print the step number $v.flag$ and node $v$;
3. if $v$ is in $T$, then break the loop.

We end the loop when we reach a node in $T$. Since we have one step at the beginning before the loop, the path is at least of length 1.

The node $v$ in $T$ may not have $v.flag$ equals to 0. It can be $v.flag > 0$ if $v$ can reach $T$ in at least 1 steps. In that case, $T$ can be visited multiple times until $v.flag$ equals to 0. If $v.flag$ is always greater than 0, then $T$ can be visited infinitely many times, which is a Büchi objective.

This path construction is in pre order, i.e., we print $v$ before its strategy is printed. To construct a path from $t$ to a node in $T$, we simply call the iterative function $path(t)$, which is a shortest path.

**Algorithm 21:** Path Construction of Reachability in at Least 1 Steps

**Input:**

1. Reachability searched graph $G = (V, E)$.
2. Target set $T \subseteq V$.
3. Node $t$ in the reachable set that we would like to search for a path to reach $T$ in at least 1 steps.

**Output:**

1. The path from $t$ to a node in $T$.

```
1 The iterative algorithm path(v) is defined as:
2 path(v){
3 for (print step # v.flag : v;;) do
4     v = v.strategy;
5     print step # v.flag : v;
6     if v.isT == 1 then Break;
7 }
```

**Result:**

1. This function $path(v)$ is based on the reachability search in at least 1 steps.
2. To construct a path from $t$ to $T$, we simply call the iterative function $path(t)$, which is a shortest path.

### A.8.4 Complexity Analysis

In this subsection, we will show some complexity analysis for the reachability game.

For a graph $G = (V, E)$, $|V| = n$ is the number of nodes, $|E| = m$ is the number of edges.

Since the two versions of the reachability problem are both variations of BFS, the time and space complexity are both linear w.r.t. $n$ and $m$.

Time Complexity: $\mathcal{O}(n+m)$.

Space Complexity: $\mathcal{O}(n+m)$.

The size of the queue $\mathcal{Q}$ is $\mathcal{O}(n)$, because in both cases, each node can be enqueued at most once.

**Remark** (Differences and similarities between the two reachability games).

- *Differences:*
  - *When initially enqueueing $T$:*
    - *in the first case, we mark $T$ as visited;*
    - *in the second case, we mark $T$ as unvisited.*
  - *Reachable set:*
    - *in the first case, $\mathcal{W} = \{v | v.flag \geq 0\}$;*
    - *in the second case, $\mathcal{W} = \{v | v.flag \geq 1\}$;*
  - *application:*
    - *in the first case, it is the solution of a standard reachability game;*
    - *in the second case, it is used in reaching a target set $T$ multiple times, or even infinitely many often times (Büchi objective).*
- *Similarities:*
  - *the layer number in both cases indicates the smallest number of steps to reach a node in $T$;*
  - *the time and space complexity in both cases are both $\mathcal{O}(n+m)$.*

## A.9 Safety

In this section, we will discuss the safety game.

Safety is another simple specification. The time complexity is linear w.r.t. the number of nodes $n$ and the number of edges $m$, i.e., $\mathcal{O}(n+m)$. Safety is an application of the reachability algorithm.

### A.9.1 Problem Description

**Problem Description:** Given the target set $T$, we want to find all the nodes in $T$ that can always stay in $T$.

Here $T$ is considered as the safe region, and $V/T$ is considered as the unsafe region. $T$ and $V/T$ forms a partition of the node set $V$, i.e., $T \oplus (V/T) = V$. We want to find all the nodes in $T$ that can always stay in $T$, so our working space is $T$. Nodes in $T$ that are forced to leave $T$ and enter $V/T$ in one step are the ones whose successors are all in $V/T$, therefore they have no choice but to enter $V/T$.

This reminds us of the topological sort using BFS, where we enqueue a node when its in-degree reduces to 0. Similar to the reachability game, we also compute backwards in a graph with in-going edges in the safety game. The enqueued unsafe nodes are the ones with out-degree to $T$ equal to 0. The nodes in the safety set that can always stay in T are the ones in $T$ excluding the unsafe nodes.

We borrow the partition operator $\oplus$ from linear algebra to give a more detailed description of the graph.

**Definition A.9.1** (partition operator)**.** *A partition operator $\oplus$ is a binary operator that describes the partition of a set, i.e., $\{V = V_1 \oplus V_2\} \equiv \{V_1 \cup V_2 = V \text{ and } V_1 \cap V_2 = \emptyset\}$. We can also align more than one partition operator, i.e., $V = V_1 \oplus V_2 \oplus V_3$.*

In other words, the union of the partitioned sets is the universal set, and the partitioned sets are mutually disjoint.

### A.9.2 Algorithm

In this subsection, we will give an algorithm to perform safety.

**Sketch of the Algorithm**

When we search for the nodes in $T$ that can always stay in $T$, we set three marks to each node $v$: $outdegT$, $flag$ and $strategy$. $v.outdegT$ is the number of out-going edges to $T$ of $v$, i.e., out-degree to $T$ of $v$; $v.flag$ marks whether $v$ is safe or not; $v.strategy$ marks the local strategies of $v$, which guarantees $v$ to stay in $T$ if $v$ is in the safety set.

We start our algorithm by initializing our three marks. We set the number of out-going edges to $T$ for nodes in $T$. We mark $v.flag$ as 1 for the nodes in $T$ to indicate safe nodes and 0 for the nodes in $V/T$ to indicate unsafe nodes respectively. We mark $v.strategy$ as

-1 to indicate $v$ hasn't assigned a strategy yet. To set up the queue $\mathcal{Q}$, we set $v.flag$ as 0 for all the nodes in $T$ with out-degree to $T$ 0, which implies these nodes are unsafe because they will be forced to enter the unsafe $V/T$ in the next step. Then we add all these unsafe nodes in $T$ into the queue.

Next, we formally proceed our search of unsafe nodes in $T$ by repeating the following process until the queue is empty:

1. pick the currently first node $v$ out of the queue;
2. for each predecessor $v'$ of $v$ in $T$:
   (a) reduce the out-degree to $T$ of $v'$ by 1;
   (b) if the out-degree to $T$ of $v'$ reduces to 0, then enqueue $v'$ and mark $v'$ as unsafe.

Nodes in $T$ that can always stay in $T$ are the ones with $v.flag$ equal to 1. For a node $v$ in the safety set, the strategies of $v$ are its safe successors. Such safe successors always exist because safe nodes have out-degree to $T$ greater than 0.

Now we can conclude that the process of finding the safety set of $T$ is to delete the unsafe nodes in $T$. The process of finding the unsafe nodes in $T$ forms a reachability game

to the unsafe region $V/T$, therefore it is an application of the reachabililty algorithm.

**Algorithm 22:** Safety

**Input:**
1. $G = (V, E)$;
2. $T \subseteq V$;

**Output:**
1. Winning set $W$ of $V$;
2. Winning strategy of each state in $W$;

**Initialization:** Let $\mathcal{Q}$ be a queue, set $T.outdegT$, $T.flag = 1, (V/T).flag = 0, V.strategy = -1$;
**for** *(; **all** $v \in T$;)* **do**
  **if** $v.outdegT == 0$ **then**
    $\mathcal{Q}.enqueue(v), v.flag = 0$;
**for** $(;\mathcal{Q}$ *is not empty*$;)$ **do**
  **for** $(v = \mathcal{Q}.dequeue(\ )$; ***all*** $v' \in Pre(v);)$ **do**
    **if** $v'.flag == 1$ **then**
      $v'.outdegT - -$;
      **if** $v'.outdegT == 0$ **then**
        $\mathcal{Q}.enqueue(v'), v'.flag = 0$;

**Result:**
1. $\mathcal{W} = \{v | v.flag == 1\}$;
2. For $v \in \mathcal{W}$, $v.strategy = \{u | (v, u) \in E, u.flag == 1\}$.

### A.9.3 Complexity Analysis

In this subsection, we will show some complexity analysis for the safety game.

For a graph $G = (V, E)$, $|V| = n$ is the number of nodes, $|E| = m$ is the number of edges.

Since the safety problem is an variation of the topological sort using BFS, the time and space complexity are both linear w.r.t. $n$ and $m$.

Time Complexity: $\mathcal{O}(n + m)$.

Space Complexity: $\mathcal{O}(n + m)$.

The size of the queue $\mathcal{Q}$ is $\mathcal{O}(|T|) = \mathcal{O}(n)$, because each node in $T$ can be enqueued at most once.

## A.10 Summary

In this section, we will sum up this chapter.

We give a background introduction in graph theory and algorithms, where the linear searching algorithm Breadth First Search (BFS) is the most fundamental one. Topological sort and some problems defined as games such as reachability and safety can be solved based on the variations of BFS.

# Appendix B

# Model

## B.1 JiushaoQin's Algorithm or Horner's Algorithm

In this section, we will introduce JiushaoQin Algorithm and its applications.

### B.1.1 Algorithm and Analysis

JiushaoQin's Algorithm is the known fastest approach to evaluate a polynomial of degree $n$.

Consider an $n^{th}$ degree polynomial

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0. \tag{B.1}$$

Generally, if we evaluate an $n^{th}$ degree polynomial directly, we need at most $n$ additions and $\frac{n(n+1)}{2}$ multiplications. The complexity is therefore polynomial $\mathcal{O}(n^2)$.

However, if we rewrite it as

$$\begin{aligned} f(x) &= a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \\ &= (a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1)x + a_0 \\ &= ((a_n x^n + a_{n-1} x^{n-1} + \cdots + a_3 x + a_2)x + a_1)x + a_0 \\ &\vdots \\ &= (\ldots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0, \end{aligned}$$

then we only need at most $n$ additions and $n$ multiplications.

Therefore, this algorithm improves a polynomial complexity $\mathcal{O}(n^2)$ to a linear complexity $\mathcal{O}(n)$. Better still, multiplication is actually more expensive than addition. Therefore, this algorithm saves even more.

### B.1.2 Number System

In this subsection, we will introduce the first application of JiushaoQin's algorithm which is the number system.

A number system is the system that we use to represent numbers. Decimal number system with base 10 is the number system that we use in our daily life. Binary number system with base 2 is the most popular number system that we use in computer. Other commonly used number systems in computer are octal number system with base 8 and hexadecimal number system with base 16.

Generally, a number system with base $d \geq 2$, $d \in \mathbb{N}$, has $d$ digits from 0 to $d-1$. We consider the left most digit as the most important digit, whereas the right most digit as the least important digit. Therefore, a number in the number system with base $d$ can be represented as $\overline{a_n a_{n-1} \cdots a_1 a_0}$, where $0 \leq a_i < d$ for $0 \leq i < n$, and $0 < a_n < d$.

For easy understanding, we may consider our commonly used decimal number system as a bridge between different number systems.

Converting a number $\overline{a_n a_{n-1} \cdots a_1 a_0}$ in the number system with base $d$ to the decimal number system can be done by plugging $x = d$ into equation B.1, where $0 \leq a_i < d$ for $0 \leq i < n$, and $0 < a_n < d$.

The inverse process of converting the same number $y = f(d)$ from the decimal number system to a number $\overline{a_n a_{n-1} \cdots a_1 a_0}$ in the number system with base $d$ is as follows:

- we mod $y$ with $d$ to get the right most digit $a_0$, i.e., $y \equiv a_0 \pmod{d}$;
- we substitute $y$ by its floor function after dividing $d$, i.e., $y = \lfloor y/d \rfloor$;
- we repeat this process for $i$ from 0 to $n$, i.e., $y \equiv a_i \pmod{d}$ and $y = \lfloor y/d \rfloor$;
- this process terminates when $y$ becomes 0 after the iteration for $i = n$.

### B.1.3 Multi-Dimension Array

In this subsection, we will introduce the second application of JiushaoQin's algorithm which is the multi-dimension array.

The multi-dimension array can be considered as an application of the generalized number system.

In a number system with base $d$, we fix the base to be $d$ on each position. We can also assign different $d_i$ as base for different positions $0 \leq i \leq n$. Therefore, a number in the number system with base $d_i$ on position $i$ can be represented as $\overline{a_n a_{n-1} \cdots a_1 a_0}$, where $0 \leq a_i < d_i$ for $0 \leq i < n$, and $0 < a_n < d_n$.

Converting a number $\overline{a_n a_{n-1} \cdots a_1 a_0}$ in the number system with base $d_i$ on position $i$ to the decimal number system can be done by applying the JiushaoQin's algorithm $y = (\ldots((a_n \cdot d_{n-1} + a_{n-1}) \cdot d_{n-2} + a_{n-2}) \cdot d_{n-3} + \cdots + a_1) \cdot d_0 + a_0$.

The inverse process of converting the same number $y$ from the decimal number system to a number $\overline{a_n a_{n-1} \cdots a_1 a_0}$ in the number system with base $d_i$ on position $i$ is as follows:

- we mod $y$ with $d_0$ to get the right most digit $a_0$, i.e., $y \equiv a_0 \pmod{d_0}$;
- we substitute $y$ by its floor function after dividing $d_0$, i.e., $y = \lfloor y/d_0 \rfloor$;
- we repeat this process for $i$ from 0 to $n$, i.e., $y \equiv a_i \pmod{d_i}$ and $y = \lfloor y/d_i \rfloor$;
- this process terminates when $y$ becomes 0 after the iteration for $i = n$.

Such generalization of the number system can be applied directly in the setting of the multi-dimension array.

For a multi-dimension array $\text{array}[d_n][d_{n-1}]\cdots[d_1][d_0]$, there are $d_i$ units for dimension $i$, $0 \leq i \leq n$. On dimension $i$, the index $a_i$ is within $0 \leq a_i < d_i$. This $(n+1)$-dimension array $\text{array}[d_n][d_{n-1}]\cdots[d_1][d_0]$ is equivalent as the one-dimension array $\text{array}[\prod_{i=0}^{n} d_i]$. This equivalence relation can be shown by showing the 1-1 correspondence of the address of an element $\text{array}[a_n][a_{n-1}]\cdots[a_1][a_0]$ in the multi-dimension array and the address of an element $\text{array}[y]$ in the one-dimension array.

The mapping is given by $y = (\ldots((a_n \cdot d_{n-1} + a_{n-1}) \cdot d_{n-2} + a_{n-2}) \cdot d_{n-3} + \cdots + a_1) \cdot d_0 + a_0$ as in the generalized number system.

The mapping between the generalized number system and the multi-dimension array is thus straightforward:

- a generalized number system with base $d_i$ on position $i$ for $0 \leq i \leq n$ corresponds to a multi-dimension array array$[d_n][d_{n-1}]\cdots[d_1][d_0]$;
- a number $\overline{a_n a_{n-1}\cdots a_1 a_0}$ in the above number system corresponds to an element array$[a_n][a_{n-1}]\cdots[a_1][a_0]$ in the above multi-dimension array;
- the conversion of a number $\overline{a_n a_{n-1}\cdots a_1 a_0}$ in the above number system to a number $y$ in the decimal number system corresponds to the conversion of the address of an element array$[a_n][a_{n-1}]\cdots[a_1][a_0]$ in the above multi-dimension array to the address of an element array$[y]$ in the one-dimension array array$[\prod_{i=0}^{n} d_i]$;
- the inverse conversion of a number $y$ in the decimal number system to a number $\overline{a_n a_{n-1}\cdots a_1 a_0}$ in the above number system corresponds to the inverse conversion of the address of an element array$[y]$ in the one-dimension array array$[\prod_{i=0}^{n} d_i]$ to the address of an element array$[a_n][a_{n-1}]\cdots[a_1][a_0]$ in the above multi-dimension array;

### B.1.4 Encode and Decode

In this subsection, we will introduce the third application of JiushaoQin's algorithm which is the procedure to encode and decode.

We can generalize the equivalence relation of a multi-dimension array array$[d_n][d_{n-1}]\cdots[d_1][d_0]$ and a one-dimension array array$[\prod_{i=0}^{n} d_i]$ to the equivalence relation of a multi-dimension system and a one-dimension system.

We consider the procedure of converting $\overline{a_n a_{n-1}\cdots a_1 a_0}$ to $y$ as encode, whereas consider the procedure of converting $y$ to $\overline{a_n a_{n-1}\cdots a_1 a_0}$ as decode. Either way has linear time complexity $\mathcal{O}(n)$.

We encode when we don't need the detailed information of the element but only its index. An encoded index saves us space and time for later computations. An encoded index takes 1 unit.

We decode when we need the detailed information of the element to do computations. A decoded index enables us to have access of the information of the element directly, which also saves us time. A decoded index takes $n$ units.

Here we list all the places that we used the technique of encoding and decoding.

- System state
- Control action
- Label
- States in the product NTS
- States in the multi-product NTS
- States in the multi-product automaton