Graph-theoretic Properties of Control Flow Graphs and Applications

by

Neeraj Kumar

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

Waterloo, Ontario, Canada, 2015

Abstract

This thesis deals with determining appropriate width parameters of control flow graphs so that certain computationally hard problems of practical interest become efficiently solvable. A well-known result of Thorup states that the treewidth of control flow graphs arising from structured (goto-free) programs is at most six. However, since a control flow graph is inherently directed, it is very likely that using a digraph width measure would give better algorithms for problems where directional properties of edges are important. One such problem, *parity game*, is closely related to the μ-calculus model checking problem in software verification and is known to be tractable on graphs of bounded DAG-width, Kelly-width or entanglement.

Motivated by this, we show that the DAG-width of control flow graphs arising from structured programs is at most three and give a linear-time algorithm to compute the corresponding DAG decomposition. Using similar techniques, we show that Kelly-width of control flow graphs is also bounded by three. Additionally, we also show that control flow graphs can have unbounded entanglement. In light of these results, we revisit the complexity of the μ-calculus model checking problem on these special graph classes and show that we can obtain better running times for control flow graphs.

Acknowledgements

I am extremely grateful to my supervisors Therese Biedl and Sebastian Fischmeister for their guidance and support over these two years. It has been a great learning experience and has played a pivotal role in shaping my career. I would also like to thank Naomi Nishimura and Werner Dietl for being on my thesis committee.

I am also grateful to my friends here in Waterloo for all the good times and to my family for being so supportive of my academic endeavours.

Dedication

To my family.

Table of Contents

List of Tables **viii**

List of Figures **ix**

1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 3

1.3 Outline 4

2 Background **5**

2.1 Control Flow Graphs 5

2.2 Treewidth of Control Flow Graphs 12

2.2.1 Thorup's approach 13

2.3 Cops and Robber Game 16

2.4 History of Digraph Width Measures 17

2.5 Software Model Checking 19

2.5.1 Parity Game 19

2.5.2 Modal μ-calculus 20

2.5.3 μ-calculus Model Checking to Parity Games 21

3 DAG-width of Control Flow Graphs **23**

3.1 Cops and Robber Characterization 23

3.1.1 Cops and Robber on Control Flow Graphs 25

3.2 Constructing a DAG Decomposition 31

3.2.1 Winning Strategy implies DAG Decomposition 33

3.2.2 Computing the DAG Decomposition Directly 35

3.3 Properties of DAG-width 38

3.3.1 DAG-width Under Contraction and Subdivision 39

3.4 DAG-width of programs with labelled breaks 47

4 Other Digraph Width Parameters **52**

4.1 Kelly-width 52

4.1.1 Inert Robber Game and Kelly-width 53

4.1.2 Kelly-width of Control Flow Graphs 54

4.2 Entanglement 57

4.2.1 Entanglement of Control Flow Graphs 59

5 Conclusion and Open Problems **63**

References **66**

List of Tables

2.1 Various graph width parameters and their cops and robber characterization. Note that the digraph width measures (except the treewidth) must respect the edge orientations. 17

List of Figures

2.1 A sample program in STRUCTURED with the resulting control flow graph. Atomic statements are highlighted in gray. 7

2.2 Loop elements. (a) The abstract structure. Dashed edges must start in *belongs*(L), *cfg(S)* stands for control flow graph of statement S. (b) A control flow graph with loops L_i, L_j nested under L. The vertices enclosed in dash-dotted red, blue and green regions are *inside*(L), *inside*(L_i) and *inside*(L_j) respectively. The red, blue and green vertices are in *belongs*(L), *belongs*(L_i) and *belongs*(L_j) respectively. The vertices in black are in *belongs*(L_ϕ). . . . 11

2.3 A simple example. The goal is to check if the model above satisfies the formula $[\cdot](P \vee \langle\cdot\rangle Q)$ at s_0, given that the atomic propositions P and Q hold at the states s_1 and s_3 respectively. 21

3.1 An example control flow graph. See Figure 3.2 for an example run of our cops and robber strategy on this graph. 26

3.2 Strategy f applied on the graph in Figure 3.1. 27

3.3 The robber player has a winning strategy on G against two cops 30

3.4 A sample graph and its DAG decomposition on the right. 33

3.5 A digraph G and its DAG decomposition D computed by Algorithm 1. Note that the digraph G has $\Theta(|V|)$ edges but the DAG decomposition D has $\Theta(|V|^2)$ edges. 35

3.6 The DAG-width of G increases from k to $k+1$ after contraction. 40

3.7 An example illustrating our construction. In (a), we will add u to all the bags X_l shaded in gray, $l \in \{a, b, c, d\}$. The bag X_l in (b) will remain unchanged. 42

3.8 DAG decomposition on subdividing. The newly added bag X_k is shaded in gray. Note that in Case 3, u is introduced in both X_{i_1} and X_{i_2}, but different conditions namely 3b(i) and 3b(ii) apply. 46

3.9 The digraph $G(2, k)$ with $k = 2$. We say that the path $v_1 \rightsquigarrow v_5$ is cop-free if none of the vertices in $v_1 \rightsquigarrow v_5$ or $v_5' \rightsquigarrow v_1'$ is occupied by a cop. 48

3.10 (a) A control flow graph G_k with $k = 2$ and (b) its simplified version G'_k. The edges $L_i^{entry} \rightarrow L_i^{exit}$ are not shown for clarity. We simplify G_k by combining the `if`-chain (shaded in gray), contracting the dash-dotted edges and finally removing L_1^{exit}. Note that G'_k can be interpreted as a concatenation of two complete binary trees T and T' of height k, plus some extra edges and vertices. 50

4.1 A digraph G with Kelly-width 3 and DAG-width 4. The shaded region denotes the clique $C_1 = \{b, d, e\}$. Note that G has Kelly-width and DAG-width at least 3 as it contains a clique of size 3. 54

4.2 A lazy invisible robber has a winning strategy against two cops on this digraph. Note that at all times, the robber is at one of the shaded vertices. 56

4.3 (a) A control flow graph G. The $L^{entry} \rightarrow L^{exit}$ edges are not shown for clarity. We combine certain consecutive vertices of G together to obtain G' shown in (b). Note that G' can be interpreted as a concatenation of two complete binary trees T and T' of height k, plus some extra edges. The edges of T and T' are oriented downwards and upwards respectively. In this figure, $k = 2$. 60

Chapter 1

Introduction

1.1 Motivation

Software design, like all other human activities, is prone to errors and it is possible that the final product does not meet the initial specifications. This is especially important for safety-critical systems such as medical devices or avionics where consequences of an incorrect behaviour can be catastrophic. In order to address such concerns, the most widely accepted approach is to perform a formal verification of the system under consideration. In the formal approach, a system is modeled via a mathematical structure (such as a directed graph) and a specification is a list of mathematical properties which must hold in the structure if the system meets the required expectations. As an example of a property, consider a simple program that uses locks for synchronization. The following must hold:

On all program paths, a lock l_0 acquired at some point is eventually released.

A property is usually specified as a logical formula. However, the choice of the logic used depends on the amount of expressivity needed. As expected, simpler formulas are easier to verify whereas verifying certain complicated formulas can be computationally hard. The modal μ-calculus, a formalism introduced by Kozen [22], is known to encompass many traditional logics of programs and can express a wide range of properties [15].

This problem of deciding whether a computer system is correct with respect to its specification corresponds to an instance of the more general μ-calculus model checking problem. Formally, the μ-calculus model checking problem is the following: given a property π as a μ-calculus formula and a model M of the system under verification, the goal is to check whether or not the model M satisfies the property π. The model M of the system is usually a directed graph, e.g the so-called *Kripke structure* [23], where the vertices are labeled with appropriate propositions of the property π. For example, in the context of software systems, the *control-flow graph* of the program is the Kripke structure. In a control-flow graph (defined formally in Section 2.1), the vertices represent the basic blocks in the program and the edges represent the flow of control between them.

Since model checking is an important aspect of formal verification, it is desirable to have an efficient algorithm that could model check properties on transition systems of practical interest. Unfortunately, such an algorithm has not been found yet. The problem is known to be in NP $\cap$ co-NP [16], but a polynomial-time algorithm is not known so far. The best known result for a system of size n is $O(m \cdot n^{\lceil d/2 \rceil + 1})$ [12], where m is the size of the formula and d is its so-called *alternation depth*. It follows that if the alternation depth d is bounded by a small constant, the problem can be solved in polynomial time. Most model checking tools take advantage of this fact and therefore can model check properties of small alternation depth, which is usually the case in practice [7]. However, model checking more complicated properties is still a problem. An alternative approach is to efficiently solve the μ-calculus model checking problem on special graph classes and show that these classes cover cases of practical interest. There are algorithms in the literature that solve the model checking problem on graphs of bounded *treewidth* [17, 24], *DAG-width* [10, 17], *Kelly-width* [10] and *entanglement* [6]. (Formal definitions of these parameters will be given later.) In this thesis, we work towards the latter goal. More precisely, we show that control-flow graphs have bounded *DAG-width* and *Kelly-width*, thereby enabling us to use the DAG-width and Kelly-width based algorithms from [10, 17] for model checking software programs.

Among special graph classes, graphs of treewidth at most k have garnered considerable interest in algorithmic graph theory. Originally introduced as a graph theoretic tool in the seminal “graph minors” paper series of Robertson and Seymour [27], it quickly spawned a rich field of algorithmic research. Treewidth gives a notion of graph decomposition along which recursive algorithms can be built. By using dynamic programming [8] over the so-called tree decomposition, it is possible to solve many computationally hard problems efficiently when restricted to graphs of treewidth at most k. Using such techniques, Obdržálek [24] showed that the μ-calculus model checking problem on graphs of treewidth at most k is solvable in time $O(n \cdot (km)^2 \cdot d^{2((k+1)m)^2})$. Note that the running time is linear in n, the size of the system. What makes this result interesting is Thorup's proof [30] that the treewidth of control flow graphs arising from structured (goto-free) programs is at most 6. His proof comes with an associated tree decomposition (which is otherwise hard to find [3]). This allows us to use Obdržálek's algorithm for model checking software programs. However, despite the fact that the running time is linear in the size of the system, the algorithm is far from being practical due to the factor coming from $d^{2((k+1)m)^2}$. That is, with $k = 6$, verifying a formula of length $m = 1$ will have a running time of $O(n \cdot d^{98})$. Fearnley and Schewe's [17] improved result for bounded treewidth graphs brings the running time down to $O(n \cdot 7^{11} \cdot (d+1)^{23})$, but it still seems impractical.

Since treewidth is a metric for undirected graphs and control-flow graphs are directed, it was natural to look for graph width measures that consider the orientation of edges. Such measures (see [26] and Section 2.4 for a brief survey) can come in handy for solving problems where the directional properties of edges are important. Among such width measures, DAG-width [5] and Kelly-width [19] have attained reasonable success. Similar to treewidth, both these width measures give a notion of a graph decomposition on top of which efficient algorithms can be built. However, unlike tree decompositions where the decomposition structure is a tree, directed decompositions can have more than a linear number of edges. Since the running time of the algorithms also depends on the size of the decomposition, it is desirable that the corresponding directed decompositions have a linear number of edges. Fearnley and Schewe [17] showed that the μ-calculus model checking problem is efficiently solvable on graphs of bounded DAG-width. Bojanczyk et al. [10] gave fixed parameter tractability results by showing that the μ-calculus model checking problem can be solved in $O(f(k+m) \cdot n^c)$ time on classes of bounded DAG-width and Kelly-width, for some computable function f and some constant c. As usual, m is the size of the formula and k is the width parameter. Note that both these algorithms expect the corresponding directed decompositions as part of the input.

1.2 Contribution

Recall that explaining the structure of control flow graphs via treewidth is overly pessimistic since it ignores the directional properties of edges. In this thesis we study the width of control flow graphs for some directed width measures.

To this end, we show that the DAG-width of control flow graphs arising from structured (goto-free) programs is at most 3 and give a linear-time algorithm to find an associated DAG decomposition of small size. Using similar techniques, the Kelly-width of control flow graphs also turns out to be at most 3. The corresponding Kelly decomposition can be computed in linear time as well. Having found directed decompositions of a small width and a linear number of edges, we can obtain better running times for algorithms using these digraph width measures. For example, the DAG-width based algorithm from [17] for model checking with a single sub-formula runs in time $O(n^2 \cdot 3^5 \cdot (d+1)^{11})$. This is competitive with and probably more practical than the previous algorithms.

From a graph-theoretic perspective, it is desirable for a digraph width measure to be small on many interesting instances. The above result makes a case for both DAG-width and Kelly-width by demonstrating an application area where they are useful.

1.3 Outline

The remainder of the thesis is organized as follows. Chapter 2 reviews notation and some background material. We will also briefly review treewidth and the proof of Thorup's result [30]. Chapter 3 discusses DAG-width in detail and presents the proof of the DAG-width bound as well as the algorithm to find the associated DAG decomposition. We also present a few other general results pertaining to edge contraction and subdivision motivated by the fact that control flow graphs are usually simplified by contracting vertices that have one incoming and one outgoing edge. Chapter 4 discusses some other width parameters and their respective bounds for control flow graphs. Finally, we conclude in Chapter 5 by discussing some applications of our results and open problems.

Chapter 2

Background

In this chapter we establish the necessary background required for our main results. We start by defining control flow graphs and fixing some notation. We also list a few properties of control flow graphs that we will use in later chapters. Next we review Thorup's proof of the treewidth bound in Section 2.2, followed by a brief introduction to cops and robber games in Section 2.3 and some digraph width measures in Section 2.4. Finally in Section 2.5, we conclude this chapter by presenting details of the μ-calculus model checking problem on control flow graphs.

2.1 Control Flow Graphs

Informally speaking, a control flow graph of a program P is a directed graph where the vertices represent statements in the program and the edges represent the flow of control between them. In order to formally define control flow graphs, we will first need to define a program P formally. We assume that the programs we consider are goto-free, otherwise the corresponding control flow graph can be any digraph and so it will not have any special width properties.

Since we want our representation of a program to be independent of the choice of the programming language, we use a toy language STRUCTURED, an abstraction of the basic constructs supported by modern programming languages with the exception of unrestricted gotos. More complex constructs such as functions or switch statements can be emulated using these basic constructs and therefore we will not discuss them here for the sake of clarity. This is consistent with the approach adopted by Thorup [30].

Definition 1 (Structured Programs). *A program P in* STRUCTURED *starts with the keyword* `start`*, is followed by a* composite *program statement S and finally ends with the keyword* `stop`*. The statement S here is defined recursively as follows:*

$$
\begin{aligned}
S &\coloneqq S \;;\; S \;\;| \\
&\quad\; \texttt{loop}\ S\ \texttt{end-loop} \;\;| \\
&\quad\; \texttt{if}\ B\ \texttt{then}\ S\ \texttt{else}\ S\ \texttt{endif} \;\;| \\
&\quad\; \texttt{if}\ B\ \texttt{then}\ S\ \texttt{endif} \;\;| \\
&\quad\; s \\
B &\coloneqq B\ \texttt{and}\ B \;\;| \\
&\quad\; B\ \texttt{or}\ B \;\;| \\
&\quad\; b \;|\; \neg b
\end{aligned}
$$

Note that we have used s to represent atomic program statements and b to represent atomic Boolean expressions. By atomic we mean that these statements and expressions cannot be broken down further as per the rules mentioned above. A composite program statement S consists of more than one atomic statement. We have a few special atomic statements which mean the following:

- `break` results in a flow of control to the `end-loop` keyword of the nearest surrounding loop. Note that `break` always exits to the *nearest surrounding* loop; we do not allow *labelled breaks* in STRUCTURED (but see Section 3.4 for generalizations).
- `continue` results in a flow of control to the `loop` keyword of the nearest surrounding loop.
- `return` results in a flow of control to the end of the program, i.e to `stop`.

In Figure 2.1, we show an example of a complete program (binary search) written using the constructs in STRUCTURED.

Given a structured program P, we can think of the constructs (`start, end, loop, end-loop, if, then, else, endif, and, or`) and the atomic statements s and b as *program points*, since any branching occurs only at these points. Having said that, we can index these program points from $1 \ldots n$ in the order of their appearance in the program P. For example, `start` gets the index 1 and `stop` gets the index n, where n is the number of program points. All other program points get indices between 1 and n. See Figure 2.1.

```
$\texttt{start}_1$
$low \leftarrow 1, high \leftarrow n\ _2$;
$\texttt{loop}_3$
    $\texttt{if}_4$ $low > high\ _5$ $\texttt{then}_6$
        $break_7$
    $\texttt{endif}_8$ ;
    $mid \leftarrow low + \frac{(high - low)}{2}\ _9$ ;
    $\texttt{if}_{10}$ $key = a[mid]\ _{11}$$\texttt{then}_{12}$
        $return\ true\ _{13}$
    $\texttt{endif}_{14}$;
    $\texttt{if}_{15}$ $key < a[mid]\ _{16}$$\texttt{then}_{17}$
        $high \leftarrow mid - 1_{18}$
    $\texttt{else}_{19}$
        $low \leftarrow mid + 1_{20}$
    $\texttt{endif}_{21}$
    $continue_{22}$
$\texttt{end-loop}_{23}$ ;
$return\ false_{24}$
$\texttt{stop}_{25}$
```

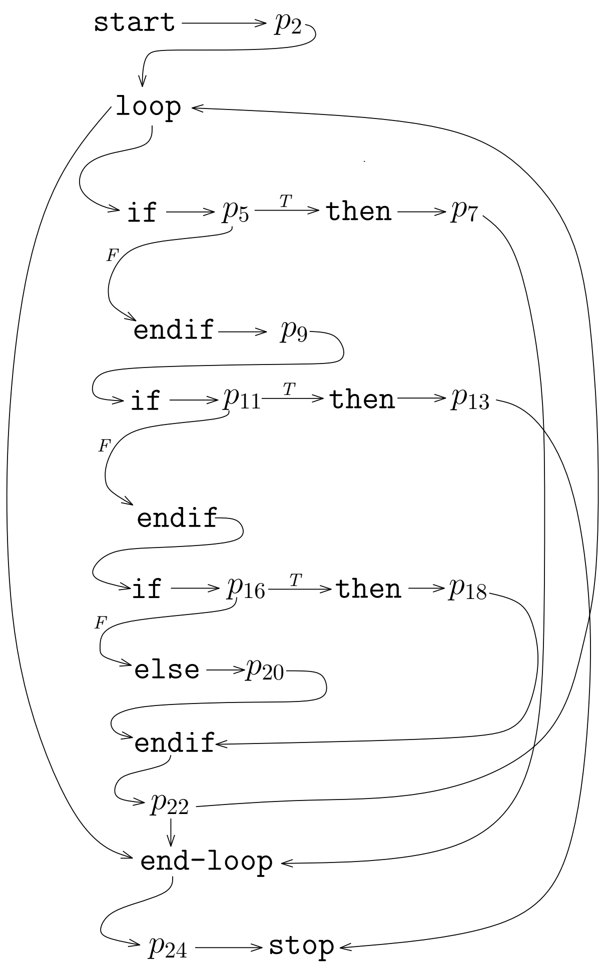


Figure 2.1: A sample program in Structured with the resulting control flow graph. Atomic statements are highlighted in gray.

Definition 2. *The* control flow graph *of a program P is a directed graph $G = (V, E)$, where a vertex $v_i \in V$ represents a program point $p_i \in P$ as defined above and an edge $(v_i, v_j) \in E$ represents the* flow of control *from p_i to p_j under some execution of the program P. The vertices v_1 =* `start` *and v_n =* `stop` *correspond to the first and last statements of the program, respectively.*

We use *cfg(S)* as a short-cut for the subgraph of G formed by the vertices corresponding to the program points in statement S.

In order to define control flow graphs more precisely, we will need to clarify what exactly we mean by "flow of control", or in other words, how exactly do we add the edges? We note that the potential immediate successors of a program point p_i are as follows. The corresponding vertex v_i in the control flow graph G can have at most the following immediate successors.

- *out*, the succeeding program point. This is typically p_{i+1}. The only exception occurs if p_{i+1} = `else`, in which case *out* is the `endif` corresponding to the `if-then-else` construct.
- If p_i is an atomic statement s, it can have three other immediate successors.
  - *exit*, the `end-loop` corresponding to the nearest surrounding loop. This happens when p_i is a `break` statement.
  - *entry*, the `loop` corresponding to the nearest surrounding loop. This happens when p_i is a `continue` statement.
  - *stop*, the end of program. This happens when p_i is the `return` statement.
- If p_i is a boolean expression b, it has two immediate successors.
  - *true*, the program point when b evaluates to true and
  - *false*, the program point when b evaluates to false.

Finally we introduce the following convention. For every loop construct `loop` S `end-loop` in the control flow graph G, we will add an edge from `loop` to `end-loop`. This is to clearly indicate the loop boundary and will be useful later. Note that adding extra edges does not affect our results concerning width parameters since if they hold for a bigger graph, they will automatically hold for its subgraphs.

For the sake of having fewer vertices in G, some representations of control flow graphs further remove a vertex v with *in-degree* and *out-degree* 1 by contracting the edge $(u, v) \in E$. For example, in Figure 2.1 we could suppress the vertices such as p_2 or p_7. (In Section 3.3.1, we show that this does not affect our results.) For more details on control flow graph construction and related concepts, we refer our readers to [1,2]. However, we borrow the following definition which will be used to gain some more insight into the structure of control flow graphs.

Definition 3 (Dominators)**.** *Let G be a control flow graph and $u, v \in V$. We say that u* dominates *v, if every directed path from* `start` *to v must go through u. Similarly, we say that u* post-dominates *v, if every directed path from v to* `stop` *must go through u.*

For reasons that will be clear in the later chapters, we are particularly interested in program points that result in directed cycles. As we will see later, these only occur at loops. Since we study loops a lot, it makes sense to define them by means of the vertex-pairs that correspond to them.

Definition 4 (Loop Element)**.** *We refer to a loop construct as a* loop element *L characterized by an entry point L^{entry} (which is the vertex corresponding to* `loop`*) and the exit point L^{exit} (which is the vertex corresponding to* `end-loop`*). See Figure 2.2.*

Recall that from Definition 1, for every loop element L in STRUCTURED, the only way to enter program points inside the loop is through L^{entry}. Moreover, the only way to exit from program points inside the loop is to L^{exit} or to the `stop` vertex. With this observation, we now note the following definitions and properties for loop elements.

- We define $\mathit{inside}(L)$ to be the set of vertices dominated by L^{entry} and not dominated by L^{exit}. Quite naturally, we define $\mathit{outside}(L)$ to be the set of vertices $(V \setminus \mathit{inside}(L))$. Note that $L^{entry} \in \mathit{inside}(L)$ but $L^{exit} \in \mathit{outside}(L)$. Moreover, if we ignore edges to `stop`, L^{exit} post-dominates vertices in $\mathit{inside}(L)$.
- For the purpose of simplification, we assume G to be enclosed in a hypothetical loop element L_ϕ. This is purely notational and we do not add extra vertices or edges to G. We have $\mathit{inside}(L_\phi) = V$ and $\mathit{outside}(L_\phi) = \emptyset$.
- We say that a loop element L_i is *nested* under L, iff $L_i^{entry} \in \mathit{inside}(L)$. Two distinct loop elements are either nested or have disjoint insides.

- We can now associate each vertex of G with a loop element as follows. We say that a vertex $v \in V$ *belongs to* L if and only if L is the nearest loop element such that L^{entry} dominates v. More precisely, $v \in belongs(L)$ if and only if $v \in inside(L)$, and there exists no L_i nested under L with $v \in inside(L_i)$.

- Every $v \in V$ belongs to exactly one loop element. `start` and `stop` (as well as any vertices outside all loops of the program) belong to L_ϕ.

- Finally, we say that a loop element L_i is *nested directly* under L, iff $L_i^{entry} \in belongs(L)$. In other words, L_i is nested under L and there exists no L_j nested under L such that L_i is nested under L_j.

For ease of understanding, we illustrate the above concepts with examples in Figure 2.2.

Recall that based on the order in which the statements appear in the program P, we defined a total order on the program points $p_i \in P$. This immediately implies a total order $\preceq_P$ on the vertices of the control flow graph G. Note that `start` is minimal in this ordering, whereas `stop` is maximal. Moreover, for a loop element L, $v \in inside(L)$ if and only if $L^{entry} \preceq_P v \prec_P L^{exit}$. We say that an edge $(u, v) \in E$ is a *forward edge* if $u \prec_P v$; otherwise we call it a *backward edge*.

From our definition of control flow graphs (Definition 2), it follows that for every edge (u, v) in a control flow graph G, the only case where $v \prec_P u$ is when v is the entry point of the nearest surrounding loop. Therefore, we have the following:

Lemma 1. *The backward edges are exactly those that lead from a vertex in $belongs(L)$ to L^{entry}, for some loop element L.*

Corollary 1. *Let C be a directed simple cycle for which all the vertices are in $inside(L)$ and at least one vertex is in $belongs(L)$, for some loop element L. Then $L^{entry} \in C$.*

Proof. Let v be a vertex in the cycle C such that $v \in belongs(L)$.

Since C is a directed cycle, it must contain back edges. One of these must lead from a program point after v (possibly itself) to a program point before v (possibly itself). Hence we have u, w with $u \preceq_P v \preceq_P w$ such that (w, u) is a backward edge.

Since (w, u) is a backward edge, it follows from Lemma 1 that $w \in belongs(L_x)$ and $u = L_x^{entry}$ for some loop element L_x. Now as all the vertices of the cycle C are in $inside(L)$, there are two cases, either $L_x = L$ or L_x is a loop element nested under L.

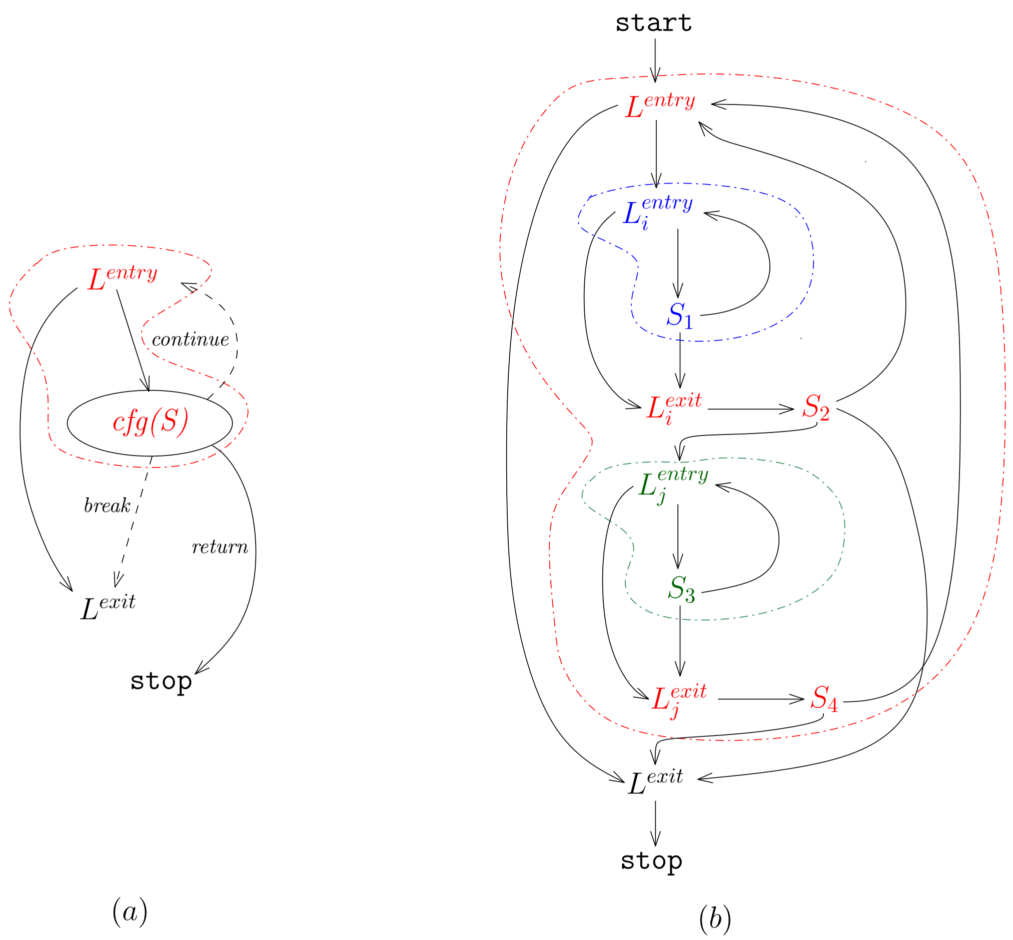


Figure 2.2: Loop elements. (a) The abstract structure. Dashed edges must start in *belongs*(L), *cfg(S)* stands for control flow graph of statement S. (b) A control flow graph with loops L_i, L_j nested under L. The vertices enclosed in dash-dotted red, blue and green regions are *inside*(L), *inside*(L_i) and *inside*(L_j) respectively. The red, blue and green vertices are in *belongs*(L), *belongs*(L_i) and *belongs*(L_j) respectively. The vertices in black are in *belongs*(L_ϕ).

Note that since $u \preceq_P v \preceq_P w$ and $u, w \in belongs(L_x)$, it follows that $v \in inside(L_x)$. However, $v \in belongs(L)$ which cannot hold if L_x is nested under L.

Therefore, $L_x = L$ must hold and hence $u = L^{entry} \in C$. $\square$

With this background, we will now discuss treewidth and then revisit Thorup's proof that control flow graphs have treewidth at most 6.

2.2 Treewidth of Control Flow Graphs

The *treewidth*, introduced in [27], is a graph theoretic concept which measures tree-likeness of an undirected graph. It uses a notion of graph decomposition defined as follows:

Definition 5 (Tree Decomposition). *Let $G = (V, E)$ be an undirected graph. A tree decomposition of G consists of a tree T and an assignment of bags $X_i \subseteq V$ to each node i of T such that:*

- $\bigcup X_i = V$.
- *For every k in the path from i to j, where $i, j, k \in V(T)$, $X_i \cap X_j \subseteq X_k$.*
- *If (v, w) is an edge of G, then there exists a bag X_i, $i \in V(T)$ that contains both v and w.*

The *width* of the decomposition is defined as $\max_{i \in V(T)} |X_i| - 1$. The *treewidth* of the graph is the minimum width over its all possible tree decompositions. Treewidth is often alternatively characterized in terms of an ordering of vertices of G defined as follows.

Definition 6 (Elimination Ordering). *An* elimination ordering *of an undirected graph $G = (V, E)$ is a linear ordering on the vertices V. Given an elimination ordering $\pi = (v_1, ..., v_n)$, we define G_{i-1} to be the graph obtained by 'eliminating' v_i from G_i. That is, define $G_n = G$. To obtain G_{i-1} from G_i remove v_i from G_i and add new edges (if needed) so that all the neighbors of v_i form a clique in G_{i-1}. These newly added edges are called* fill edges.

The width of an elimination ordering is defined as the maximum over all $i \in \{1, \ldots, n\}$ of the degree of v_i in G_i. The treewidth of G is the minimum width over all possible elimination orderings [14].

A third way to define treewidth via *k-complex listings* was used by Thorup [30].

Definition 7. *A* $(\leq k)$-complex listing *is an ordering of the vertices V of G such that for every vertex $v \in V$, there is a set $S(v)$ of at most k vertices preceding v in the listing, whose removal separates v from all the vertices preceding v in the listing.*

In other words, if $(v_1, ..., v_n)$ is a $(\leq k)$-complex listing of the vertices V, then there is no (undirected) path in $G[V \setminus S(v_i)]$ from v_i to v_j with $j < i$.

Note that if we add fill edges to G by eliminating vertices as per a $(\leq k)$-*complex listing* of the vertices of G, all the newly added edges for a vertex v_i are between the vertices in $S(v_i)$. Since the size of $S(v_i)$ is at most k, we have the following:

Observation 1 ([30]). *A $(\leq k)$-complex listing is the same as an elimination order of width k. The separator set for v_i consists of the neighbors of v_i in G_i.*

In the following section, we revisit Thorup's proof where he gives a $(\leq k)$-complex listing for control flow graphs with $k = 6$. Readers more familiar with elimination orders are encouraged to remember that the vertex order created is exactly the same, though the proof of correctness does not directly transfer.

2.2.1 Thorup's approach

In his proof [30], Thorup claims that the following order of recursively visiting the program points of a structured program (Definition 1) corresponds to a (≤ 6)-complex listing π.

Note that a program P in STRUCTURED is of the form `start` S `stop`. We first insert {`start` , `stop`} into π and then recursively visit S. There are the following cases:

1. S is an atomic statement s, or an atomic boolean expression b, $\neg b$.
   - Let v be the corresponding vertex in G. Append v to π.

2. S is of the form: S' ; S''
   - Append the vertex for $\{;\}$ to π and then recursively visit S' and S''.

3. S is of the form: `loop` S' `end-loop`
   - Let the corresponding loop element be L. Append $\{L^{entry}, L^{exit}\}$ to π, and then visit S' recursively.

4. S is a composite boolean expression of the form: B *op* B' where *op* $\in$ {`and`, `or`}.
   - Append the vertex for *op* to π and then visit B, B'.

5. S is of the form: `if` B `then` S' `else` S'' `endif`
   - Append the vertices for {`if`, `endif`, `then`, `else`} to π and recurse on B, S' and S''.

6. S is of the form: `if` B `then` S' `endif`
   - Append the vertices for {`if`, `endif`, `then`} to π and recurse on B and S'.

Claim 1. *The listing π computed above is (≤ 6)-complex.*

Proof. In order to show that the listing π is (≤ 6)-complex, it suffices to show that all the vertices get a separator of size at most 6. That is, for every vertex v_i we would like to find a set $S(v_i)$ of at most 6 vertices preceding v_i in π, such that after removing $S(v_i)$ from G, there is no path from v_i to v_j, with $j < i$.

Recall that the potential neighbors of a statement S are:

- the preceding statement or construct : *in*
- and the immediate successors: *out*, *entry*, *exit*, *stop*.

Likewise, the potential neighbors of an boolean expression B are:

- the preceding statement or construct : *in*
- and the immediate successors: *true* evaluation and *false* evaluation.

Note that the sequence of visiting statements described before ensures that all the neighbors of a statement S or a boolean expression B are visited before we visit words in S or B. Therefore, by removing all the neighbors of S, that is, {*in, out, exit, entry, stop*}, we can separate the first visited statement in S from all its predecessors in π. The idea is to apply this recursively and hence compute the separator sets for vertices in the listing π.

It is important to note that the {*in, out, exit, entry, stop*} may assume different values at each recursive call. For example, *entry* and *exit* always get the values L^{entry} and L^{exit} respectively where L is the nearest loop element surrounding the statement which is being visited currently. The *stop* always corresponds to the `stop` vertex. This will be more clear with the case-by-case analysis below.

Recall that we begin with `start` S `stop`. The words `start` and `stop` will get the separator sets $\{\phi\}$ and {`start`} respectively. We will now recurse on S with *in'* = `start`, *out'*= `stop`. Since there is no loop enclosing S, *entry'*= $\{\phi\}$ and *exit'*= $\{\phi\}$.

For a statement of the form $S = S'$; S'', the word ';' is visited first and therefore will get the separator set $\{$*in, out, exit, entry, stop*$\}$. We will now recurse on S' with $in' = in$ and out' = ';'. Similarly for S'', we recurse with in' = ';' and $out' = out$.

Proceeding similarly for `loop` S' `end-loop`, the word `loop` will get the separator set $\{$*in, out, exit, entry, stop*$\}$. Here *entry* and *exit* correspond to the nearest loop surrounding the current loop (if there is any). For `end-loop`, we have the separator set $\{$`loop`, *out, exit, entry, stop*$\}$. Finally, we recurse on S' with in' = `loop` and out' = `end-loop`.

For conditionals of the form `if` B `then` S' `else` S'' `endif`, the `if` will be visited first and hence gets the separator set $\{$*in, out, exit, entry, stop*$\}$. Proceeding similarly, `endif` will get the separator $\{$`if`, *out, exit, entry, stop*$\}$, `then` will get the separator set $\{$`if`, `endif`, *exit, entry, stop*$\}$ and `else` will get the separator set $\{$`if`, `endif`, `then`, *exit, entry, stop*$\}$. Finally, we will recurse on B with in' = `if` and $true$ = `then`, $false$ = `else`; recurse on S' with in' = `then` and out' = `endif`; and recurse on S'' with in' = `else` and out' = `endif`.

The atomic statements will get the separator set $\{$*in, out, exit, entry, stop*$\}$. An atomic boolean expression will get the separator set $\{$*in, true, false*$\}$.

For a boolean expression, B `or` B', '`or`' will get the separator set $\{$*in, true, false*$\}$. We then recurse on B with $in' = in$, $true' = true$ and $false'$ = `or`; and on B' with in' = `or`, $true' = true$ and $false' = false$. The case for '`and`' is similar.

To see that $Q = \{$`if`, `endif`, `then`, *exit, entry, stop*$\}$ is a separator set for `else`, consider any path in $G[V \setminus Q]$ from the vertex v_i of '`else`' to an earlier vertex v_j. We know that the path contains no `break`, `continue` or `return`, else it would contain *exit*, *entry* or *stop*. Also it must be entirely within B or S'', else it would contain `if`, `endif` or `then`. But all the statements inside B or S'' occur later in the order, contradicting that v_j comes earlier. The same reasoning can be applied to all other cases to show that the selected sets are indeed separators.

Note that the above algorithm will repeat until we hit an atomic statement or expression, and hence will compute the separator set of every vertex in the listing. We can see that in all these cases, the maximum size of the separator sets is 6. □

From the above result, it follows that treewidth of control flow graphs cannot exceed 6. Thorup also showed that this bound is tight by constructing a control flow graph that has treewidth exactly 6 [30]. Once we have the (≤ 6)-complex listing (which is also an elimination ordering), the associated tree decomposition can be computed in linear time [9]. Since graphs of treewidth at most k have $O(k \cdot |V|)$ edges, we have the following.

Corollary 2. *Every control flow graph $G = (V, E)$ has $O(|V|)$ edges.*

2.3 Cops and Robber Game

The *cops and robber* game on a graph G is a two-player pursuit evasion game in which k cops attempt to catch a robber. Most graph width measures have an equivalent characterization via a variant of the cops and robber game.

On a graph $G = (V, E)$, the cops and robber game is played as per the following general rules. Some variants of the game impose additional constraints on movement of the cops and the robber.

- The cop player controls k cops, which can occupy any k vertices in the graph. We denote this set as X where $X \in [V]^{\leq k}$. The robber player controls the robber which can occupy any vertex r.
- A play in the game is a (finite or infinite) sequence $(X_0, r_0), (X_1, r_1), \dots, (X_i, r_i)$ of positions taken by the cops and robbers. $X_0 = \emptyset$, i.e., the robber starts the game by choosing an initial position.
- In a transition in the play from (X_i, r_i) to (X_{i+1}, r_{i+1}), the cop player moves the cops not in $(X_i \cap X_{i+1})$ to $(X_{i+1} \setminus X_i)$. In the variants that we are interested in, this move happens by a helicopter, that is, there are no restrictions on X_{i+1} relative to X_i. We say that the robber is *visible* if the cops can see the robber at all times and therefore can plan out their next move based on the current position of the robber. In other words, $X_{i+1} = f(X_i, r_i)$ for some computable function f.

  The robber can see the helicopter landing and moves at a great speed along a cop-free path to another vertex r_{i+1}. More specifically, there must be a path from r_i to r_{i+1} in the graph $G \setminus (X_i \cap X_{i+1})$. This path must be a directed path if G is a digraph. Note that some variants of the game may impose additional constraints on this path.
- The play is winning for the cop player if it terminates with (X_m, r_m) such that $r_m \in X_m$. If the play is infinite, the robber player wins.

Definition 8. *(Reachable vertices) Let Reach(S, r) be the set of all vertices v such that there exists a path from r to v in $G[V \setminus S]$. We define Reach(X_i, r_i) to be the set of vertices reachable by the robber after the i^{th} move, that is when the cops occupy X_i and the robber occupies r_i.*

Additionally, we will use Reach$(X_i \cap X_{i+1}, r_i)$ to denote the set of vertices reachable from r_i, the current position of the robber, while the cops make their move from X_i to X_{i+1}.

Width measure	Constraints
Treewidth	Robber visible (undirected case)
DAG-width	Robber visible
Kelly-width	Robber invisible but lazy (only moves when a cop relocates to his current position)
Directed treewidth	Robber visible but must stay in the same strongly connected component
Entanglement	Robber visible but can only move to a successor. Cops can only move to current position of the robber.

Table 2.1: Various graph width parameters and their cops and robber characterization. Note that the digraph width measures (except the treewidth) must respect the edge orientations.

Definition 9. *(Monotone strategies)*

- *A strategy for the cop player is called* cop-monotone *if in a play consistent with that strategy, the cops never visit a vertex twice. More precisely, if $v \in X_i$ and $v \in X_k$ then $v \in X_j$, for all $i \leq j \leq k$.*
- *A strategy for the cop player is called* robber-monotone *if in a play consistent with the strategy, the set of vertices reachable by the robber is non-increasing. More precisely, $Reach(X_j, r_j) \subseteq Reach(X_i, r_i)$, for all $i \leq j$.*

Table 2.1 reviews some variants of the cops and robber game and lists the width parameter (most of which will be defined later) that correspond to it. For example, an undirected graph G has *treewidth* k if and only if $k+1$ cops can search G and successfully catch the robber [28]. Note that the cops can see the robber at all times. Moreover, Dendris et al. [14] showed that treewidth can be alternatively characterized by a variant of the cops and robber game where the robber is invisible but lazy, that is, only moves when a cop relocates to his current position. They showed that this variant also needs $k+1$ cops and a winning strategy for this variant of the game corresponds to an elimination ordering of width k.

2.4 History of Digraph Width Measures

Since treewidth is a measure for undirected graphs, it was natural to think of graph width measures that also take into account the orientation of edges. The objective was to find a measure which is general enough and, like treewidth, has nice algorithmic and graph theoretic properties. Motivated by this, several digraph width measures were proposed. In this section, we briefly discuss some of them that we will consider for our results. For the sake of locality, we defer the exact definitions to later chapters.

The first such measure, the *directed treewidth*, was proposed by Johnson, Robertson, Seymour, and Thomas [20] in 2001. The idea was to use a directed tree as the decomposition structure and partition the vertices of the underlying graph into bags (subject to certain conditions). The objective was to attain better running times for computationally hard problems on graphs of small directed treewidth. However, except for a few initial results, directed treewidth was not very successful. This was probably because the directed tree-decompositions are not intuitively related to the graphs they decompose and therefore, designing algorithms using these decompositions is complicated and error-prone. Moreover, unlike the game for undirected graphs, the cops and robber characterization for directed treewidth is not very clean. Specifically, a winning strategy against k cops only implies that there is a directed tree decomposition of width $3k + 1$.

In order to solve the problems associated with directed treewidth, Obdržálek [25] and Berwanger et al. [4] independently came up with a new connectivity measure for directed graphs, even giving it the same name *DAG-width*. The idea was to define a measure that tells how close a given digraph is to being a directed acyclic graph (DAG) (Section 3.1 gives a precise definition). The ulterior objective was to use the associated *DAG-decompositions* to efficiently solve computationally hard problems that are easy on DAGs. Moreover, DAG-width has a very clean characterization in terms of the cops and robber game on digraphs, which is a generalization of the game for treewidth to directed graphs. From this, it follows that the DAG-width is always upper bounded by treewidth plus one, since we can just ignore the edge directions and use the winning strategy from the game for treewidth. As expected, DAG-width has been quite successful and there are quite a few papers that present efficient algorithms using DAG-width [5, 10, 17, 26].

However, one common problem in all such algorithms comes from the fact that a DAG-decomposition can have up to $|V|^{k+2}$ edges. To address this problem, Hunter and Kreutzer proposed the notion of *Kelly-width* [19] that generalizes the concept of elimination orderings to directed graphs (see also Section 4.1). They characterize Kelly-width in terms of a variant of the cops and robber game called the *inert-robber* game in which the cops cannot see the robber and the robber can only move if there is a cop approaching its current position. In the case of undirected graphs, both the inert and the usual cops and robber game need the same number of cops [14]. However for the case of directed graphs, such a relationship has not been found yet, and consequently the corresponding width measures, Kelly-width and DAG-width, are uncomparable as of yet. In [19], the authors conjecture that Kelly-width and DAG-width lie within constant factors of one another.

One final width measure that is worth mentioning is *entanglement* [6] (see also Section 4.2), which captures the extent to which the directed cycles of a graph are entangled. Since there is no notion of graph decomposition associated with entanglement, the usability for dynamic-programming style algorithms is fairly limited. However, it is known to be closely connected to the μ-calculus model checking problem and therefore it may be worthwhile to study the entanglement of control flow graphs.

2.5 Software Model Checking

Recall that the main goal of this thesis is to characterize control flow graphs with appropriate width parameters such that certain computationally hard problems of practical interest become easier to solve on control flow graphs. One such application that we briefly discussed in Section 1.1 is the μ-calculus model checking problem. The problem is known to be in NP $\cap$ co-NP but a polynomial-time algorithm is not known so far. However, it is well known [29] that the μ-calculus model checking problem is polynomial-time reducible to the problem of finding a winner in *parity games*. Motivated by this, parity games were extensively studied and efficient algorithms were found for graphs of bounded widths. It follows that, using these algorithms combined with the bounded width property of control flow graphs, we may be able to efficiently solve the software model checking problem since it is essentially the μ-calculus model checking problem on control flow graphs.

In this section we will review the needed background to understand these algorithms for graphs of bounded widths. We will briefly discuss the modal μ-calculus, parity games and review how to convert the μ-calculus model checking to the problem of finding a winner in parity games. Finally, we conclude the chapter with a review of the treewidth-based algorithm for software model checking.

Note that the concepts presented in this section are not really needed for the main results in this thesis and are mostly for the sake of completeness. However, these concepts may be helpful in understanding the algorithmic applications of the main results.

2.5.1 Parity Game

A *parity game* $\mathcal{G}$ consists of a directed graph $G = (V, E)$, called the *game graph*, a partition of V into sets V_0 and $V_1 = V \setminus V_0$; and a parity function $\lambda : V \to \mathbb{N}$ (called *priority*) that assigns a natural number to each vertex of G.

The game $\mathcal{G}$ is played between two players P_0 and P_1 who move a shared token along the edges of the graph G. The vertices $V_0 \subset V$ and $V_1 = V \setminus V_0$ are assumed to be owned by P_0 and P_1 respectively. If the token is currently on a vertex in V_i (for $i = 0, 1$), then player P_i gets to move the token, and moves it to a successor of his choice. This results in a possibly infinite sequence of vertices called *play*. If the play is finite, the player who is unable to move loses the game. If the play is infinite, P_0 wins the game if the largest occurring priority is even, otherwise P_1 wins.

A solution for the parity game $\mathcal{G}$ is a partitioning of V into V_0^w and V_1^w, which are respectively the vertices from which P_0 and P_1 have a winning strategy. Clearly, V_0^w and V_1^w should be disjoint.

2.5.2 Modal μ-calculus

The modal μ-calculus (see [29] for a good introduction) is a fixed-point logic comprising a set of formulas defined by the following syntax:

$$\phi ::= X \mid P \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [\cdot]\phi \mid \langle\cdot\rangle\phi \mid \nu X\phi \mid \mu X\phi$$

Typically, a formula ϕ is evaluated on a directed graph (such as a state machine), where the vertices represent the *states* and the edges represent a *transition* between these states. We say that the formula ϕ holds at a state s if it evaluates to *true* at s. The symbols above mean the following.

- X is a propositional variable.
- P is an atomic proposition.
- $\neg$, $\wedge$, $\vee$ are usual boolean operators.
- $[\cdot]$, $\langle\cdot\rangle$ are the operators from modal logic. $[\cdot]\phi$ means the formula ϕ holds on all paths starting at state s whereas $\langle\cdot\rangle$ means that there is at least one path starting at s where ϕ holds.
- ν and μ are the maximal and minimal fixed point operators. Precise mathematical understanding of these operators requires some background on fixed point theory which is beyond the scope of this thesis. We refer interested readers to [11] for more details. Intuitively, these operators can be thought of as formalisms that allow the formula to be applied recursively. For example, if we want to say that the proposition

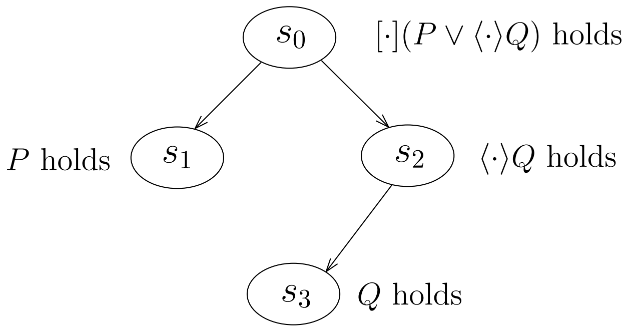

s_0
$[\cdot](P \vee \langle\cdot\rangle Q)$ holds
P holds
s_1
s_2
$\langle\cdot\rangle Q$ holds
s_3
Q holds

Figure 2.3: A simple example. The goal is to check if the model above satisfies the formula $[\cdot](P \vee \langle\cdot\rangle Q)$ at s_0, given that the atomic propositions P and Q hold at the states s_1 and s_3 respectively.

> P holds on all paths starting at state s, we will use $[\cdot]P$. However, if we wanted to say that P holds on all paths starting at state s in *all* computations, we will need to use $\phi(X) = \nu X\ (P \wedge [\cdot]X)$. That is, the maximal fixed point operator ν causes $\phi(X)$ to *unfold* an infinite number of times. On the other hand, with the minimal fixed point operator μ, $\phi(X)$ can only unfold a finite number of times. The combination of these two operators allows us to express a wide range of properties but at the cost of increased complexity. The *alternation depth* of a formula is the number of syntactic alternations between ν and μ. Recall that all known algorithms for model checking a μ-calculus formula are exponential in the alternation depth.

Given a formula ϕ, we say that a μ-calculus formula ψ is a *subformula* of ϕ if we can obtain ψ from ϕ by recursively decomposing as per the above syntax. For example, the formula $\nu X\ (P \wedge [\cdot]X)$ has five subformulas: $\nu X\ (P \wedge [\cdot]X)$, $(P \wedge [\cdot]X)$, P, $[\cdot]X$ and X. The *size* of a formula is the number of its subformulas.

2.5.3 μ-calculus Model Checking to Parity Games

A model $M = (S, T)$ of a system is a digraph with the set of states S as vertices and the transitions T as edges. The *μ-calculus model checking problem* consists of testing whether a given modal-μ-calculus formula ϕ applies to M (Figure 2.3 shows a simple example). As mentioned earlier, given M and ϕ, there exists a way to construct a parity game instance $\mathcal{G} = (G, \lambda, V_0, V_1)$ such that ϕ applies if and only if the parity game can be solved. See e.g. [29] for more details and some examples[1]. We note the following relevant points of this transformation:

R.1 Let $Sub(\phi)$ be the set of all subformulas of ϕ and $m = |Sub(\phi)|$ be the size of the formula ϕ. For every $\psi \in Sub(\phi)$ and $s \in S$ we create a vertex (s, ψ) in G. Therefore, $|V(G)| = m \cdot |S|$.

R.2 For every $s \in S$, let $V_s \subseteq V(G)$ be the set of vertices defined as $\{(s, \psi) \mid \psi \in Sub(\phi)\}$. Clearly, $|V_s| = m$. It holds that for any $s, t \in S$, there is an edge between any two vertices $u \in V_s$ and $v \in V_t$ of G only if $(s, t) \in T$.

R.3 The highest priority d in $\mathcal{G}$ is equal to the alternation depth of the formula ϕ plus two. That is, for a μ-calculus formula with no fixed-point operators, the number of priorities is 2.

We will now use the treewidth-based algorithm for parity games by Fearnley and Schewe [17] for solving the model checking problem on control flow graphs. The running time of their algorithm is $O(|V| \cdot (k+1)^{k+5} \cdot (d+1)^{3k+5})$, where k is the treewidth of the parity game graph G and d is the alternation depth.

For our case, let M be the model of the system (that is, a control flow graph), on which we want to check a formula ϕ. Using Thorup's result [30] we first obtain a tree decomposition $(\mathcal{T}, X)$ of M with width at most 6. This means that each bag of the tree decomposition contains at most 7 vertices. Using the translation above, we can obtain the parity game graph G for the model M and the formula ϕ. Note that using rule R.2 of this translation above, we can now obtain a tree decomposition $(\mathcal{T}', X')$ for the parity game graph G from $(\mathcal{T}, X)$ by replacing every $s \in X_i$ by V_s, for all the bags $X_i \in X$. Note that the width of $\mathcal{T}'$ will be $7 \cdot m - 1$.

At this point, we have the tree decomposition of the parity game graph G corresponding to the model M and the formula ϕ. We can now use Fearnley and Schewe's algorithm, with $k = 7 \cdot m - 1$. Note that even for smallest possible values $d = 2$ and $m = 1$, we have a running time of $O(|V| \cdot 7^{11} \cdot 3^{23}) = O(|V| \cdot 10^{20})$; which seems quite impractical. To this end, we observe that using a digraph width parameter such as DAG-width could give us better results if we can find a small bound on them for control flow graphs. We work towards this goal in the subsequent chapters.

[1]Alternatively, see pages 20-23: Obdržálek, Jan. Algorithmic analysis of parity games. PhD thesis, University of Edinburgh, 2006

Chapter 3

DAG-width of Control Flow Graphs

In the previous chapter, we discussed control flow graphs and briefly introduced the concept of DAG-width. We also know that the treewidth of control flow graphs is at most 6 and that the DAG-width of a digraph is always bounded by its treewidth plus one. It trivially follows that the DAG-width of control flow graphs is also bounded and is at most 7. However, since we ignore the directions of the edges, it is quite likely that this bound is not tight and we can do better. Since algorithms tailored to small DAG-width are typically exponential in the DAG-width, a smaller bound is likely to improve the running times of such algorithms for control flow graphs.

In this chapter, we address the problem of finding a tight bound on the DAG-width of control flow graphs. We start by reviewing the cops and robber characterization of DAG-width. Then we present our cops and robber strategy for control flow graphs with at most three cops, thereby showing that the DAG-width of control flow graphs is at most 3. We show that this bound is tight by giving an example of a control flow graph that has DAG-width exactly 3. In Section 3.2.2, we give an algorithm to compute the DAG decomposition of control flow graphs and argue its correctness. Finally in Section 3.3, we discuss and prove some general properties of DAG-width which can be useful in general and were needed for the proof of our main result.

3.1 Cops and Robber Characterization

The cops and robber game corresponding to DAG-width is a natural generalization of the equivalent game for treewidth to directed graphs. The cops can see the robber, who can only move along a cop-free path in order to evade capture. The following result is central to our proof:

Theorem 1. [5, Lemma 15 and Theorem 16] *A digraph G has DAG-width k if and only if the cop player has a cop-monotone winning strategy in the k-cops and robber game on G.*

Recall that a play is a sequence $(X_0, r_0), (X_1, r_1), \ldots, (X_i, r_i)$ of positions taken by the cops X_i and the robber r_i. Note that a (k-cop) strategy for the cop player is a function $f : [V]^{\leq k} \times V \to [V]^{\leq k}$. Put differently, the cops can see the robber when deciding where to move. A play is consistent with the strategy f if $X_{i+1} = f(X_i, r)$ for all i.

It is also important to note that for the game corresponding to DAG-width, a *cop-monotone* strategy is also *robber monotone* and vice-versa [5, Lemma 15].

Therefore, in order to prove that DAG-width of a graph G is at most k, it suffices to find a cop-monotone winning strategy for the cop player in the k-cops and robber game on G. In order to get a sense of DAG-width via the cops and robber game, we present a few examples below.

Example 1 (DAG-width examples)**.**

1. *The DAG-width of a directed acyclic graph is* 1*.*

   It is not hard to see that a single cop can catch the robber by placing himself at the current position of the robber. Since there are no cycles, the robber must move forward and will get stuck at a sink where the cop will catch him in the next iteration.

2. *The DAG-width of a directed cycle is* 2*.*

   In this case two cops can catch the robber by fixing one cop at some vertex of the cycle and by repeatedly placing the other cop at the current position of the robber.

3. *The DAG-width of a clique of size k is k.*

   For a directed graph G, a set of k vertices C forms a clique if for every pair of vertices u, v in C, both the edges (u, v) and (v, u) exist in G. Since there is a direct path from every vertex to every other vertex in C, the cops need to guard all k vertices of the clique C.

In the next section, we present a 3-cop strategy for control flow graphs and argue its correctness. We will later (in Section 3.2.2) give a second proof, not using the k-cops and robber game, of the DAG-width of control-flow graphs. As such, the proof presented below is not required for our main result, but is a useful tool for gaining insight into the structure of control-flow graphs, and also provides a way of proving a lower bound on the DAG-width.

3.1.1 Cops and Robber on Control Flow Graphs

Let $G = (V, E)$ be the control flow graph of a structured program P. Recall that we characterize a loop element L by its *entry* and *exit* points and refer to it by the pair (L^{entry}, L^{exit}).

We now present the following strategy f for the cop player in the cops and robber game on G with *three* cops.

1. We will throughout the game maintain that at this point $X(1)$ occupies L^{entry}, $X(2)$ occupies L^{exit}, and $r \in \mathit{inside}(L)$, for some loop element L.

   (In the first round $L := L_\phi$, where L_ϕ is the hypothetical loop element that encloses G. Regardless of the initial position of the robber, $r \in \mathit{inside}(L_\phi)$. The cops $X(1)$ and $X(2)$ are not used in the initial step.)

2. Now we move the cops:

   (a) If $r \in \mathit{belongs}(L)$, move $X(3)$ to r.

   (b) Else, since $r \in \mathit{inside}(L)$, we must have $r \in \mathit{inside}(L_i)$ for some loop L_i directly nested under L. Move $X(3)$ to L_i^{exit}.

3. Now the robber moves, say to r'. Note that $r' \in (\mathit{inside}(L) \cup \{\texttt{stop}\})$ since $r \in \mathit{inside}(L)$ and $X(1)$ and $X(2)$ block all paths from there to $(\mathit{outside}(L) \setminus \{\texttt{stop}\})$.

4. One of four cases is possible:

   (a) $r' = X(3)$. Then we have now caught the robber and we are done.

   (b) $r' = \texttt{stop}$. Move $X(3)$ to `stop` and we will catch the robber in next move since the robber cannot leave `stop`.

   (c) $r' \in \mathit{inside}(L_i)$, i.e., the robber stayed inside the same loop L_i, that it was before in Step 2(b). Go to step 5.

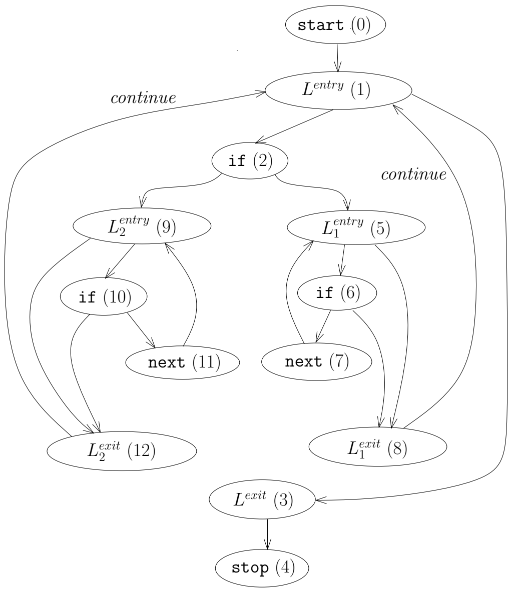


Figure 3.1: An example control flow graph. See Figure 3.2 for an example run of our cops and robber strategy on this graph.

(d) $r' \in (\mathit{inside}(L) \setminus \mathit{inside}(L_i))$, i.e. either 2(a) applied or the robber left the inside of the loop L_i that it was in. Go back to step 2. Note that this also covers the case when the robber moved to the inside of some other loop L_j directly nested under L.

5. We reach this case only if the robber r' is inside L_i, and $X(3)$ had moved to L_i^{exit} in the step before. Thus cop $X(3)$ now blocks movements of r' to $(\mathit{outside}(L_i) \setminus \{\texttt{stop}\})$. We must do one more round before being able to recurse:

   (a) Move $X(1)$ to L_i^{entry}.

   (b) The robber moves, say to r''. By the above, $r'' \in (\mathit{inside}(L_i) \cup \texttt{stop})$.

   (c) If $r'' = L_i^{entry}$, we have caught the robber. If $r'' = \texttt{stop}$, we can catch the robber in the next move.

   (d) In all remaining cases, $r'' \in \mathit{inside}(L_i)$. Go back to step 1 with $L \coloneqq L_i$, $X(2)$ as $X(3)$ and $X(3)$ as $X(2)$.

As an example in Figure 3.2, we show a possible sequence of transitions when the strategy f above is applied on a control flow graph (which is shown in all detail in Figure 3.3a.)

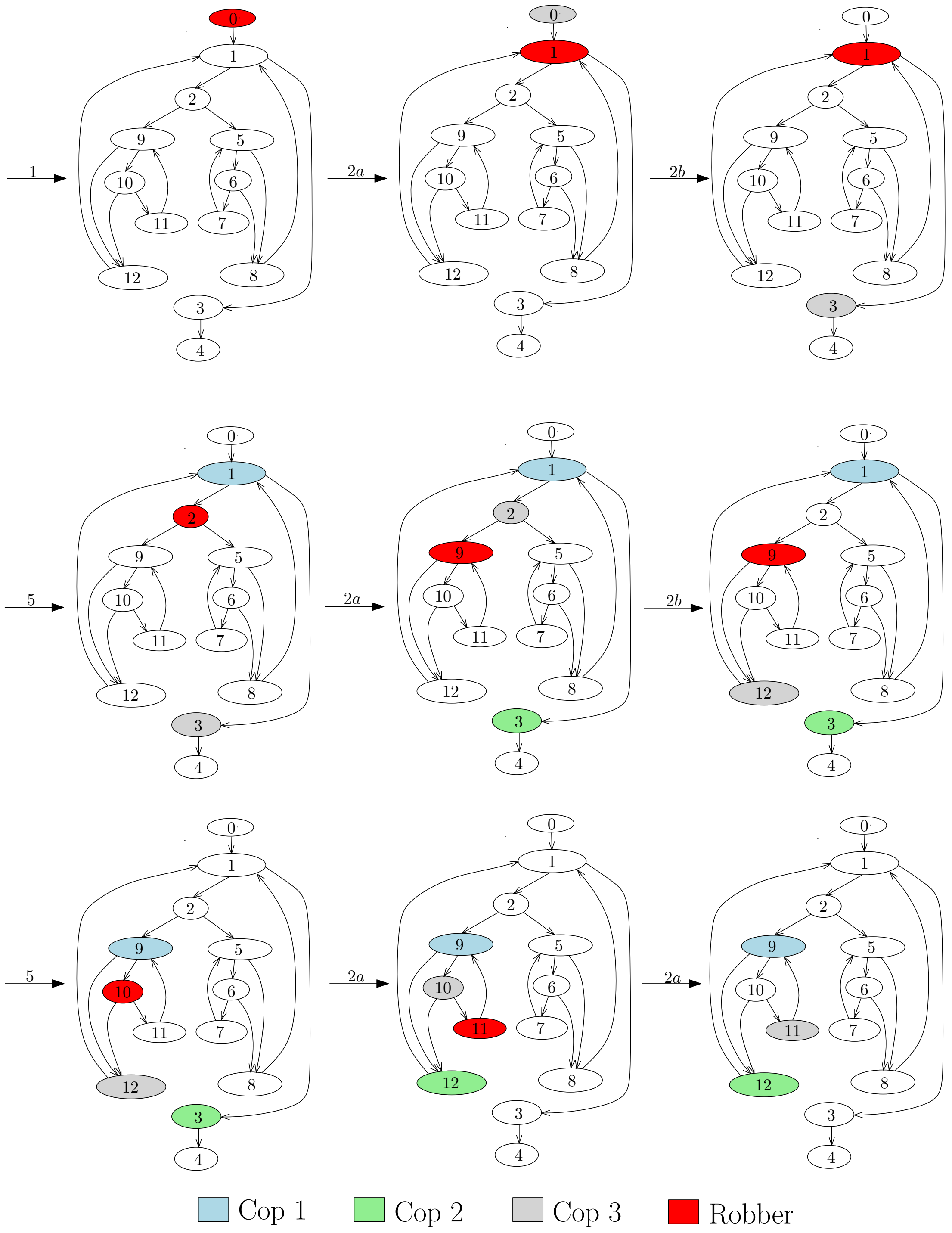


Figure 3.2: Strategy f applied on the graph in Figure 3.1.

We refer to the vertices by their indices and the labels on the transitions represent the corresponding steps of the strategy f above. We assume that the initial position of the robber is vertex 0, and he plays a lazy strategy, that is, he stays where he is unless a cop comes there, otherwise, he moves to the closest cop-free vertex. Note that in Figure 3.2, when the robber was at vertex 2, he could have moved to vertex 5 instead of going to vertex 9. However, since that case is symmetric to the robber moving to vertex 9, we do not show the corresponding transitions.

It should be intuitive that we make progress if we reach Step (5), since we have moved to a loop that is nested more deeply. It is much less obvious why we make progress if we reach 4(d). To prove this, we introduce the notion of a distance function $dist(v, L^{exit})$, which measures roughly the length of the longest path from v to L^{exit}, except that we do not count vertices that are inside loops nested under L. Formally:

Definition 10. *Let L be a loop element of G and $v \in inside(L)$. Define $dist(v, L^{exit}) = \max_P(|P \cap belongs(L)|)$, where P is a directed simple path from v to L^{exit} that uses only vertices in $inside(L)$ and does not use L^{entry}.*

Lemma 2. *When the robber moves from r to r' in step (3), then $dist(r', L^{exit}) \leq dist(r, L^{exit})$. The inequality is strict if $r \in belongs(L)$ and $r' \neq r$.*

Proof. Let P be the directed path from r to r' along which the robber moves. Notice that $L^{entry} \notin P$ since $X(1)$ is on L^{entry}. Let P' be the path that achieves the maximum in $dist(r', L^{exit})$; by definition P' does not contain L^{entry}.

$P \cup P'$ may contain directed cycles, but if C is such a cycle then no vertices of C are in $belongs(L)$ by Corollary 1. Let P_s be what remains of $P \cup P'$ after removing all directed cycles. Then $P_s \cap belongs(L) = (P \cup P') \cap belongs(L)$. Since P_s is a simple directed path from r to L^{exit} that does not use L^{entry}, therefore $dist(r, L^{exit}) \geq |P_s \cap belongs(L)| \geq |P' \cap belongs(L)| = dist(r', L^{exit})$ as desired. If $r' \neq r$, then P' cannot possibly include r while P_s does. So if we additionally have $r \in belongs(L)$, then the inequality is strict. $\square$

Lemma 3. *The strategy f is winning.*

Proof. Clearly the claim holds if the robber ever moves to `stop`, so assume this is not the case. Recall that at all times the strategy maintains a loop L such that two of the cops are at L^{entry} and L^{exit}. We provide a proof by induction on the number of loops that are nested within L.

So assume first that no loops are nested inside L. Then $inside(L) = belongs(L)$, and by Lemma 2 the distance of the robber to L^{exit} steadily decreases since $X(3)$ always moves onto the robber, forcing it to relocate. Eventually the robber must get caught.

For the induction step, assume that there are loops nested inside L. If we ever reach step (5) in the strategy, then the enclosing loop L is changed to L_i, which is inside L and hence has fewer loops inside and we are done by induction. But we must reach step (5) eventually (or catch the robber directly), because with every execution of (3) the robber gets closer to L^{exit}:

- If $r \in \mathit{belongs}(L)$, then this follows directly from Lemma 2 since $X(3)$ moves onto r and forces it to move.
- If $r \in \mathit{inside}(L_i)$, and we did not reach step (5), then r must have left L_i using L_i^{exit}. This must have happened while the cop $X(3)$ landed at L_i^{exit}, otherwise the robber could not have left. Since $\mathit{dist}(r, L^{exit}) = \mathit{dist}(L_i^{exit}, L^{exit})$ due to our choice of the distance-function and $L_i^{exit} \in \mathit{belongs}(L)$ (L_i was directly nested under L), we can view the robber as if he were initially at L_i^{exit} (which keeps the distance the same) and then moves to the new position r'. Note that since the cop $X(3)$ occupies L_i^{exit}, $r' \neq L_i^{exit}$ and by Lemma 2, the distance to L^{exit} strictly decreases.

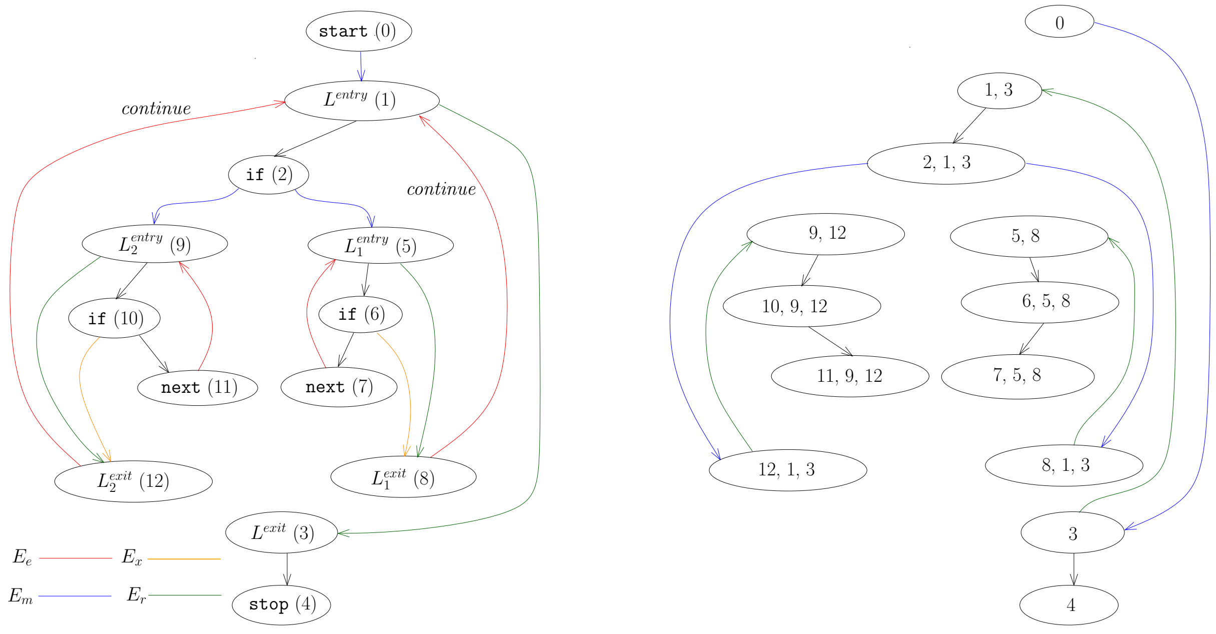
□

Lemma 4. *The strategy f is cop-monotone.*

Proof. We must show that the cops do not re-visit a previously visited vertex at any step of the strategy f. We note that since `stop` is a sink in G and the cops move to `stop` only if the robber was already there, it will never be visited again. Now the only steps which we need to verify are (2) and (5a).

Observe that while we continue in step (2), the cops $X(1)$ and $X(2)$ always stay at L^{entry} and L^{exit} respectively, and $X(3)$ *always* stays at a vertex in $\mathit{belongs}(L)$. (This holds because L_i was chosen to be nested directly under L in Case (2b), so $L_i^{exit} \in \mathit{belongs}(L)$.) Also notice that $\mathit{dist}(X(3), L^{exit}) = \mathit{dist}(r, L^{exit})$ for as long as we stay in step (2), because vertices in $\mathit{inside}(L_i)$ do not count towards the distance. In the previous proof we saw that the distance of the robber to L^{exit} strictly decreases while we continue in step (2). So $\mathit{dist}(X(3), L^{exit})$ also strictly decreases while we stay in step (2), and so $X(3)$ never re-visits a vertex.

During step (5), the cops move to L_i^{entry} and L_i^{exit} and from then on will only be at vertices in $\mathit{inside}(L_i) \cup \{L_i^{exit}\}$. Previously cops were only in $\mathit{belongs}(L)$ or in $\mathit{outside}(L)$. These two sets intersect only in L_i^{exit}, which is occupied throughout the transition by $X(3)$ (later renamed to $X(2)$). Hence no cop can re-visit a vertex and the strategy f is cop-monotone. □



(a) A control flow graph G (b) DAG Decomposition of G

Figure 3.3: The robber player has a winning strategy on G against two cops

With this, we have shown that the DAG-width is at most 3. This is tight.

Lemma 5. *There exists a control-flow graph that has DAG-width at least 3.*

Proof. Consider the graph in Fig. 3.3a. By Theorem 1, it suffices to show that the robber player has a winning strategy against two cops. We use the following strategy:

1. Start on vertex 5. We maintain the invariant that at this point the robber is at 5 or 6, and there is at most one cop on vertices $\{5, 6, 7\}$. This holds initially when the cops have not been placed yet.

2. If (after the next helicopter-landing) there will still be at most one cop in $\{5, 6, 7\}$, then move such that afterwards the robber is again at 5 or 6. (Then return to step (1).) The robber can always get to one of $\{5, 6\}$ as follows: If no cop comes to where the robber is now, then stay stationary. If one does, then get to the other position using cycle $5 \to 6 \to 7 \to 5$; this cannot be blocked since the one cop in $\{5, 6, 7\}$ is moving to the robber's positions and so no cop in $\{5, 6, 7\}$ is stationary.

3. If (after the next helicopter-landing) both cops will be in $\{5,6,7\}$, then "flee" to vertex 9 along the directed path $\{5 \text{ or } 6\} \rightarrow 8 \rightarrow 1 \rightarrow 2 \rightarrow 9$. This is feasible because there are only two cases. Either both the cops are about to land on the vertices $\{5,6,7\}$ or one of the cops is already in $\{5,6,7\}$ and the other one is about to land on some other vertex in $\{5,6,7\}$. In both these cases, the vertices on the path $8 \rightarrow 1 \rightarrow 2 \rightarrow 9$ are free and the robber can make his move.

4. Repeat the above steps with positions $\{9,10\}$, cycle $\{9,10,11\}$ and escape path $\{9 \text{ or } 10\} \rightarrow 12 \rightarrow 1 \rightarrow 2 \rightarrow 5$ symmetrically.

Thus the robber can evade capture forever by toggling between the two loop elements L_1 and L_2 and hence has a winning strategy. It is worth mentioning that the edge $(6,8)$ (and symmetrically $(10,12)$) is important to our construction because without that just one cop can block both the exit (8) and the cycle $(5,6,7)$ by placing himself at the entry (5), while the other cop will catch the robber in the blocked cycle. □

In summary:

Theorem 2. *The DAG-width of control-flow graphs is at most* 3 *and this is tight for some control-flow graphs.*

3.2 Constructing a DAG Decomposition

In the previous section, we showed that the DAG-width of a control-flow graph G is at most 3 using a winning strategy for the cops and robber game with three cops. In this section we show how to construct an associated DAG decomposition. One way to do that is to translate the winning strategy f into a DAG decomposition by the algorithm from [5, Theorem 16]. However, as we will discuss later, the algorithm is more complicated than it needs to be for control flow graphs and might create significantly more edges than needed. Therefore, we present an alternative method that directly computes the DAG decomposition from control flow graphs in linear time. First we need to define a DAG decomposition precisely.

Note that for a directed acyclic graph (DAG) D, we use $u \preceq_D v$ to denote that there is a directed path from u to v in D.

Definition 11 (DAG Decomposition)**.** *Let $G = (V, E)$ be a directed graph. A* DAG decomposition *of G consists of a DAG D and an assignment of bags $X_i \subseteq V$ to every node i of D such that:*

1. (Vertices covered) $\bigcup X_i = V$.

2. (Connectivity) *For any* $i \preceq_D k \preceq_D j$ *we have* $X_i \cap X_j \subseteq X_k$.

3. (Edges covered)

   (a) For any source j *in* D, *any* $u \in X_j$, *and any edge* (u, v) *in* G, *there exists a successor-bag* X_k *of* X_j *with* $v \in X_k$.

   (b) For every arc (i, j) *in* D, *any* $u \in (X_j \setminus X_i)$, *and any edge* (u, v) *in* G, *there exists a successor-bag* X_k *of* X_j *with* $v \in X_k$.

   Here a successor-bag *of* X_j *is a bag* X_k *with* $j \preceq_D k$.

Note that for the ease of understanding we have rephrased the edge-covering condition of the original in [5]. In Section 3.3, we show that both these conditions are equivalent.

A more intuitive way to think of the edge-covering is via the notion of *introduced* vertices defined below.

Definition 12 (Introduced vertices)**.** *Let* (D, X) *be a DAG decomposition of a digraph* G. *We say that a vertex* $v \in V(G)$ *is* introduced *in the bag* X_j *if* $v \in X_j$ *and either* j *is a source or there exists an arc* (i, j) *in* D *such that* v *does not belong to* X_i.

With this, the edge-covering condition simply states that if v is introduced in some bag X_j of D, then all its immediate successors must be present in some successor-bag X_s of X_j. We say that the edge (v, w) is *covered* in the successor-bag X_s if X_s contains the other end-point w. For example, in Figure 3.4 the vertex b is introduced in X_1 and the edges (b, a), (b, c), and (b, d) are covered in X_1, X_2 and X_3 respectively.

For ease of understanding, we now mention a few examples of DAG decompositions and note some important well-known results below.

- *A directed acyclic graph* $G = (V, E)$ *is a valid DAG decomposition of itself, that is,* $D = G$ *and* $X_v = \{v\}$ *for all* $v \in V$.

  It is not hard to see that conditions $1 - 3$ mentioned above apply.

- *For a directed circular graph* $G = (V, E)$ *(a graph containing a single directed cycle), the DAG* $D = G(V, E \setminus (p, q))$ *is a valid DAG decomposition if we set* $X_v = \{v, p\}$. *Here* (p, q) *is an arbitrarily chosen edge from the directed cycle and* $v \in V$.

  The idea is to break the cycle by taking off an edge and adding the tail of that edge to all the bags. Conditions $1 - 3$ are easy to verify.

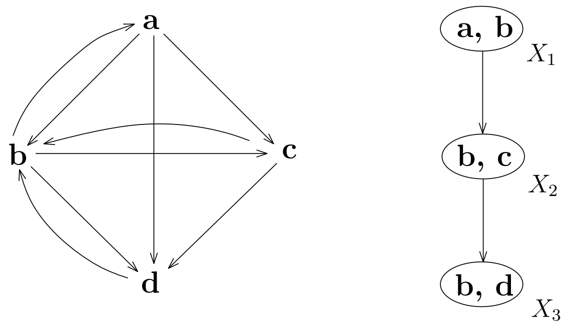


Figure 3.4: A sample graph and its DAG decomposition on the right.

- *If G has a tree decomposition of width k, then it has a DAG decomposition of width $k+1$.*

  Recall that an undirected graph has treewidth k if and only if $k+1$ cops can catch the robber on G. Since the cops and robber game for DAG-width is a generalization of the game for treewidth with directed edges, the winning strategy with $k+1$ cops also works in the game for DAG-width.

- *Computing a DAG decomposition of optimal width for an arbitrary graph is NP-hard [5, Theorem 25]*

  This follows from the NP-hardness of finding optimal tree decompositions. Consider an undirected graph G and make it directed by replacing every edge (u, v) by the directed edges $u \to v$ and $u \leftarrow v$. Recall that a DAG decomposition of $\overrightarrow{G}$ implies a winning strategy for the cops and robber game on $\overrightarrow{G}$, which also works for the undirected G. However, using this winning strategy for the cops and robber game on G, we can find a tree decomposition of G, which is known to be NP-hard.

3.2.1 Winning Strategy implies DAG Decomposition

Given a digraph $G = (V, E)$ and a winning strategy $f : [V]^{\leq k} \times V \to [V]^{\leq k}$ of the directed k-cops and robber game, the objective is to construct a DAG decomposition $(D, \mathcal{X})$ where D is a DAG and $\mathcal{X} := (X_d)_{d \in V(D)}$ is the set of bags of D. We review the algorithm of Berwanger et al. [5, Theorem 16] below. Recall that $\mathit{Reach}(X \cap X', r)$ is the set of vertices reachable from r, the current position of the robber, while the cops make their move from X to X'.

Algorithm 1 (By Berwanger et al. [5])**.**

1. Construct a digraph D' with the node set $[V]^{\leq k} \times V$ and an arc from (X, r) to (X', r') if $X' = f(X, r)$ and $r' \in \mathit{Reach}(X \cap X', r)$.

   In other words, the nodes (X, r) in D' correspond to the game position with cops on X and the robber on r and an arc indicates a round in play where the cops and robber have moved to X' and r' respectively.

2. Since the first move is to place no cops and the robber can start at any vertex $r \in V$, we select D to be the sub-digraph of D' induced by the set of vertices reachable from a node (ϕ, r) for all $r \in V$.

3. At this point we have the DAG D and we need to fill the bags $\mathcal{X} \coloneqq (X_d)_{d \in V(D)}$ of D. For a node $d = (X, r) \in V(D)$, we set $X_d = f(X, r)$. In other words, at any node $d = (X, r)$ of D, the bag X_d will contain the vertices which the cops will occupy in the next round from their current positions X.

In their proof of correctness, Berwanger et al. showed that the above method indeed computes a valid DAG decomposition of width k. However, it is not hard to see that D can have $|V|^{k+1}$ nodes and the set of vertices reachable by the robber $\mathit{Reach}(X \cap X', r)$, can be $\Theta(V)$, therefore step 1 can add $\Theta(|V|^{k+2})$ arcs.

For our winning strategy f for control flow graphs, we have $k = 3$ and therefore the resulting DAG decomposition will have $\Theta(|V|^5)$ arcs. It is possible that using some clever analysis of strategy f, we may be able to bring down the number of arcs. However, in the following we show that using Berwanger's algorithm above, we cannot do any better than $\Theta(|V|^2)$ arcs. Later in Section 3.2.2, we give an algorithm that creates a DAG decomposition of control flow graphs with at most $\Theta(|V|)$ arcs.

In order to show that Berwanger's algorithm creates more arcs in the DAG decomposition than needed, consider the graph $G = (V, E)$ from Figure 3.5. Since G is a DAG, we know that it is a valid DAG decomposition of itself. Clearly the number of edges in G is $\Theta(|V|)$. Now recall that the winning strategy on G with just one cop is to always place him at the current position of the robber. Since the cop's position always depends on the robber's current position, using Algorithm 1 above, we will construct a digraph D with $V(D) = V$. However, when the robber is at a vertex i of G, he can go to any vertex in $\{i + 1, \ldots, n\}$. Therefore, for each node $i \in V(D)$ Berwanger's algorithm creates $(|V| - i)$ arcs in D and hence the total number of arcs will be $\sum_{i \in 1..|V|}(|V| - i) = \Theta(|V|^2)$. Therefore, the DAG decomposition computed by Algorithm 1 will have $\Theta(|V|^2)$ edges, although there exists a DAG decomposition with $\Theta(|V|)$ edges.

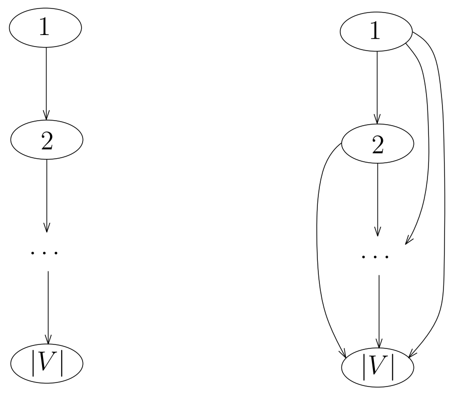


Figure 3.5: A digraph G and its DAG decomposition D computed by Algorithm 1. Note that the digraph G has $\Theta(|V|)$ edges but the DAG decomposition D has $\Theta(|V|^2)$ edges.

3.2.2 Computing the DAG Decomposition Directly

Let $G = (V, E)$ be a control-flow graph. We present the following algorithm to construct a DAG decomposition (D, X) of G.

Algorithm 2 (Construct the DAG)**.**

1. Start with $D = G$. That is $V(D) = V$ and $E(D) = E$.
2. Remove all backward arcs. Thus, let E_e be all backward edges of G; recall that each of them connects from a node $v \in \mathit{belongs}(L)$ to L^{entry}, for some loop element L. Remove all arcs corresponding to edges in E_e from D.
3. Remove all arcs leading to a loop-exit. Thus, let E_x be all edges (u, v) in G such that $u \in \mathit{belongs}(L)$ and $v = L^{\mathit{exit}}$ for some loop element L. Recall that these arcs are attributed to `break` statements. Remove all arcs corresponding to edges in E_x from D.
4. Re-route all arcs leading to a loop-entry. Thus, let E_m be all edges (u, v) in G such that $u \in \mathit{outside}(L) \setminus L^{\mathit{exit}}$ and $v = L^{\mathit{entry}}$ for some loop element L. For each such edge, remove the corresponding edge in D and replace it by an arc (u, L^{exit}). Let A_m be those re-routed arcs. Note that now $\mathit{indeg}_D(L^{\mathit{entry}}) = 0$ since we also removed all backward edges.

5. Reverse all arcs surrounding a loop. Thus, let E_r be the edges in G of the form (L^{entry}, L^{exit}) for some loop element L. For each edge, reverse the corresponding arc in D. Let A_r be the resulting arcs.

Note that there is a bijective mapping $\mathcal{D}$ from the vertices in G to the nodes in D. For ease of representation, assume that $v_1, v_2, .., v_n$ are the vertices of G and $1, 2, .., n$ are the corresponding nodes in D. We now fill the bags X_i:

For every vertex $v_i \in V$, set $X_i := \{v_i, L^{entry}, L^{exit}\}$, where L is the loop element such that $v_i \in belongs(L)$.

A sample decomposition computed by Algorithm 2 above is shown in Figure 3.3b. Clearly the construction can be done in linear time and digraph D has $O(|E|) = O(|V|)$ edges (Corollary 2). We note the following.

Observation 2. *For every arc $(i, j) \in E(D)$, $X_j \setminus X_i \subseteq \{v_j\}$.*

Proof. Note that for an arc $(i, j) \in E(D)$, there are two cases:

1. If $belongs(v_i) = belongs(v_j)$, then $X_i \cap X_j = \{L^{entry}, L^{exit}\}$ and the result follows.
2. If $belongs(v_i) \neq belongs(v_j)$, then v_i must be L^{exit} and v_j must be L^{entry} for some loop element L (by the above steps 4 and 5). In this case, clearly $X_i \cap X_j = L^{exit}$ and $X_j = \{L^{entry}, L^{exit}\}$. Therefore $X_j \setminus X_i = \{L^{entry}\} = \{v_j\}$ follows.

□

To prove correctness, we show the following:

Lemma 6. *The DAG decomposition (D, X) computed by Algorithm 2 satisfies the properties of a DAG decomposition.*

Proof. We need to argue that D is a DAG and satisfies the conditions (1-3) of Definition 11.

1. D **is a DAG**. We claim that $G - E_e$ is acyclic. For if it contained a directed cycle C, then let L be a loop element with $C \subseteq inside(L)$, but $C \nsubseteq inside(L_i)$ for any loop element L_i nested under L. Therefore C contains a vertex of $belongs(L)$. By Corollary 1 then L^{entry} belongs to C, so C contains a backward edge. This is impossible since we delete the backward edges.

Adding arcs A_m cannot create a cycle since each such arc is a shortcut for the 2-edge path from outside L to L^{entry} to L^{exit}. In $G - E_e - E_x$ there is no directed path from L^{entry} to L^{exit}, since such a path would reside inside L, and the last edge of it belongs to E_x. In consequence adding arcs A_r cannot create a cycle either. Hence D is acyclic.

Moreover, note that any directed path in D that begins in *inside*(L) can never reach *outside*$(L) \setminus$ `stop` since we have deleted/reversed all the edges to L^{exit} in D, that is, the edges in E_x and E_r.

2. **Vertices Covered**. By definition, each v_i is contained in its bag X_i.

3. **Connectivity**. Let $i \preceq_D k \preceq_D j$ be three nodes in D. Recall that their three bags are $\{v_i, L_i^{entry}, L_i^{exit}\}$, $\{v_k, L_k^{entry}, L_k^{exit}\}$ and $\{v_j, L_j^{entry}, L_j^{exit}\}$, where L_i, L_j, L_k are the loop elements to which v_i, v_j, v_k belong. There is nothing to show unless $X_i \cap X_j \neq \emptyset$, which severely restricts the possibilities:

   (a) Assume first that $L_i = L_j = L$. Thus v_i and v_j belong to the same loop element. Now since $k \rightsquigarrow j$ is a directed path in D, v_k must belong to the same loop element as v_j.
   So the claim holds since $X_i \cap X_j = \{L^{entry}, L^{exit}\} \subseteq X_k$.

   (b) If $L_i \neq L_j$, then the intersection can be non-empty only if $v_i = L_j^{exit}$ (recall that $i \preceq_D j$). But then $X_i \cap X_j = \{L_j^{exit}\}$, and the path from i to j must go from $\mathcal{D}(L_j^{exit})$ to $\mathcal{D}(L_j^{entry})$ to j. It follows that v_k also belongs to L_j and so $L_j^{exit} \in X_k$ and the condition holds.

4. **Edges Covered**. We only show the second condition; the first one is similar and easier since `start` is the only source. Let (i, j) be an arc in D. By Observation 2, v_j is the only possible vertex in $X_j \setminus X_i$. Let $e = (v_j, v_l)$ be an edge of G and let L be the loop element such that $v_j \in$ *belongs*(L). We have the following cases:

   (a) If $e \in (E_e \cup E_x \cup E_r)$, then $v_l \in \{L^{entry}, L^{exit}\}$ and $X_j = \{v_j, L^{entry}, L^{exit}\}$ itself can serve as the required successor-bag.

   (b) If $e \in E_m$, then $v_l = L^{entry}, v_j \in$ *outside*(L). We re-routed (v_j, L^{entry}) as arc $(j, \mathcal{D}(L^{exit}))$ and later added an arc $(\mathcal{D}(L^{exit}), \mathcal{D}(L^{entry}))$, so l is a successor of j and X_l can serve as the required successor-bag.

   (c) Finally, if $e \in E \setminus (E_e \cup E_x \cup E_r \cup E_m)$, then (j, l) is an arc in D and X_l is the required successor-bag.

□

From Lemma 6 above, our main result for this chapter follows.

Theorem 3. *Every control-flow graph $G = (V, E)$ has a DAG decomposition of width 3 with $O(|V|)$ vertices and edges. It can be found in linear time.*

3.3 Properties of DAG-width

Recall that in our definition for DAG-width via DAG decomposition (Definition 11), we gave a characterization of the edge-covering condition that differs from the one in the original definition given by Berwanger et al. [5]. This alternative characterization was particularly useful in proving the correctness of our results (Lemma 6). In this section we aim to show that our edge-covering condition is equivalent to the one given by Berwanger et al. We first review their concepts.

Definition 13 (Guarding)**.** *Let $G = (V, E)$ be a digraph and $W, V' \subseteq V$. We say that W* guards *V' if, for all $(u, v) \in E$, if $u \in V'$ then $v \in V' \cup W$.*

The original edge-covering condition was the following:

> (D3) For all edges $(d, d') \in E(D)$, $X_d \cap X_{d'}$ guards $X_{\succeq d'} \setminus X_d$, where $X_{\succeq d'}$ stands for $\bigcup_{d' \preceq_D d''} X_{d''}$. For any source d, $X_{\succeq d}$ is guarded by $\emptyset$.

For easier comparison we re-state here our edge-covering condition:

> (3a) For any edge (i, j) in $E(D)$, any vertex $u \in X_j \setminus X_i$, and any edge (u, v) in G, there exists a successor-bag X_k of X_j that contains v.
>
> (3b) For source j in D, any vertex $u \in X_j$, and any edge (u, v) in G, there exists a successor-bag X_k of X_j that contains v.

We will only show that the first half of (D3) is equivalent to (3a); one can similarly show that the second half of (D3) is equivalent to (3b). We first re-phrase (D3) partially by switching to our notation, and partially by inserting the definition of guarding; clearly (D3') is equivalent to the first half of (D3).

> (D3') For any edge (i, j) in $E(D)$, any vertex $u \in X_{\succeq j} \setminus X_i$ and any edge (u, v) in G, we have $v \in (X_{\succeq j} \setminus X_i) \cup (X_i \cap X_j)$.

Now, note that $(X_{\succeq j} \setminus X_i) \cup (X_i \cap X_j) = X_{\succeq j}$. The forward direction is easy to verify. For the other direction when $v \in X_{\succeq j}$, we have two cases. If $v \notin X_i$, then $v \in (X_{\succeq j} \setminus X_i)$ holds. Otherwise, if $v \in X_i$ then $v \in X_j$ by the connectivity condition. Hence, $v \in (X_i \cap X_j)$ holds.

So we can simplify (D3') again to the following equivalent:

> (D3") For any edge (i, j) in $E(D)$, any vertex $u \in X_{\succeq j} \setminus X_i$ and any edge (u, v) in G, we have $v \in X_{\succeq j}$.

At the other end, we can also simplify (3a), since we now have the shortcut $X_{\succeq j}$ for vertices in a successor-bag of X_j.

> (3a') For any edge (i, j) in $E(D)$, any vertex $u \in X_j \setminus X_i$, and any edge (u, v) in G, we have $v \in X_{\succeq j}$.

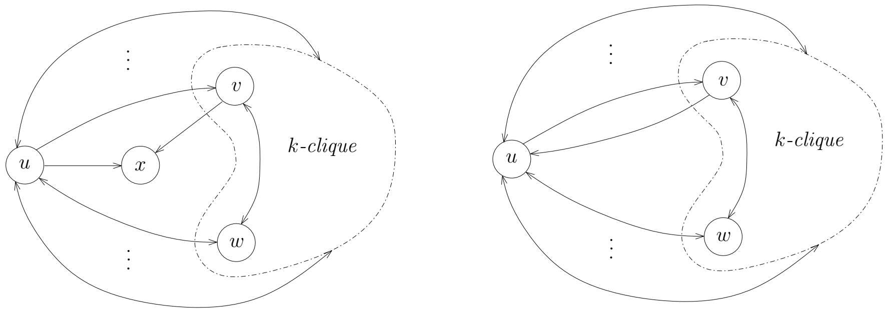
Thus (D3") and (3a') state nearly the same thing, except that for (D3") the claim must hold for significantly more vertices u. As such, (D3")$\Rightarrow$(3a') is trivial since $X_j \setminus X_i \subseteq X_{\succeq j} \setminus X_i$.

For the other direction, we need to work a little harder. Assume (3a') holds. To show (D3"), fix one such choice of edge (i, j) in $E(D)$ and (u, v) in $E(G)$ with $u \in X_{\succeq j} \setminus X_i$. We show that $v \in X_{\succeq j}$ using induction on the number of successors of j in D. If there are none, then $X_{\succeq j} = X_j$ and (D3") holds since (3a') does. Likewise (D3") holds if $u \in X_j \setminus X_i$ since (3a') holds. This leaves the case where $u \in X_{\succeq j} \setminus X_j$. Thus u belongs to some strict successor bag of X_j, and hence there exists an arc (j, k) with $u \in X_{\succeq k} \setminus X_i$. Node k has fewer successors than j, and so by induction (D3") holds for edge (j, k). We know $u \in (X_{\succeq k} \setminus X_i)$ and $u \notin X_j \setminus X_i$, so $u \in (X_{\succeq k} \setminus X_j)$. So applying (D3") we know $v \in X_{\succeq k} \subseteq X_{\succeq j}$ and hence (D3") also holds for edge (i, j).

We will now present a couple of results pertaining to edge contraction and subdivision, and their effect on DAG-width. Recall that contracting an edge $e = (u, v)$ is equivalent to deleting e, replacing the vertices u and v by a new vertex x and then making x adjacent to neighbors of both u and v preserving the directions. Subdividing an edge (u, v) is adding a vertex of degree 2 between u and v.

3.3.1 DAG-width Under Contraction and Subdivision

Recall that for a control flow graph, it is customary to contract vertices that have in-degree and out-degree 1 into a neighbour. Our main result, Theorem 3, hence is useless unless we



(a) A digraph G (b) After (u, x) was contracted

Figure 3.6: The DAG-width of G increases from k to $k + 1$ after contraction.

can argue that such contractions (and conversely, subdivisions of edges) do not increase the DAG-width. It is very easy to see that the treewidth does not increase when contracting or subdividing edges, but for DAG-width this does not obviously hold, and to our knowledge this question has not been considered in the literature.

In this section, we aim to address these questions. In contrast to treewidth, we show that the DAG-width may increase under contraction of an arbitrary edge (u, v). However, when v has in-degree 1, we show that the DAG-width does not increase, and how to compute quickly the DAG decomposition of G with (u, v) contracted. Likewise, we also show how to quickly compute a DAG decomposition of the same width when an edge is subdivided.

Theorem 4. *The DAG-width of a digraph $G = (V, E)$ may increase under contraction of an arbitrary edge.*

Proof. It suffices to find a digraph for which the DAG-width goes up on contraction. We use the digraph G in Fig 3.6a. Note that the vertices v and w are part of a k-clique. Recall that for a directed graph G, a set of k vertices C is a k-clique if for every pair of vertices u, v in C, both the edges (u, v) and (v, u) exist in G.

The vertex u is connected to every vertex in the k-clique by a bi-directional edge, except for v. The edge (u, v) connects u to v.

We will first show that the DAG-width of G is k. Since G contains a clique of size k as a subgraph, the DAG-width of G is at least k. To show that the DAG-width is exactly k, we present the following strategy for the cops and robber game on G with k cops:

- Fix $k - 1$ cops at all vertices of the k-clique, except v. We have one free cop, say X_k.

- Move X_k to u. Note that the robber is now restricted to $\{v, x\}$; from where it can never come back to u.
- Move X_k to v and then to x. Since x is a sink, cop X_k will catch the robber.

From Theorem 1, it follows that the DAG-width of G is k. After contracting (u, x), vertex u is now connected to every other vertex in the clique by a bi-directional edge and hence, there exists a clique of size $k + 1$. Therefore, the DAG-width of G increases to $k + 1$. □

We will now show that the DAG-width does not increase when contracting a vertex of in-degree 1.

Theorem 5. *The DAG-width of a digraph $G = (V, E)$ does not increase on contracting an edge $(u, v) \in E$ such that v has in-degree 1.*

Proof. Let G' be the graph obtained by contracting (u, v), where v has in-degree 1. It suffices to show that given a DAG decomposition D of G, we can construct a DAG decomposition of G' without increasing the size of the bags.

Note that as a result of contracting the edge (u, v), the vertex u will have new neighbours in G'. Therefore, the goal is to modify the original DAG decomposition D such that these newly added edges are also covered. We achieve this in two steps. First, we will construct a DAG decomposition D' by adding the vertex u to certain bags of D such that it is a valid DAG decomposition of G'. Next, we fix the width of D' by removing v from all the bags that contain it to get the DAG decomposition D''. We will start by constructing D' as follows.

```
For every bag $X_l$ of $D$ that contains $v$ but does not contain $u$ :
  If there is a successor bag $X_i$ of $X_l$ that contains both $v$ and $u$, add $u$ to
  $X_l$. Otherwise, leave $X_l$ unchanged.
```

Note that the only way bags in D' may have changed is by adding u to the bags that already contained v. Therefore, it is possible that the width of D' is one more than that of D. However, in the final step, we remove v from all the bags of D' containing it to obtain the DAG decomposition D'', so that the width of D'' is at most the one of D.

Now, we will show that D' as computed above is a valid DAG decomposition of G'.

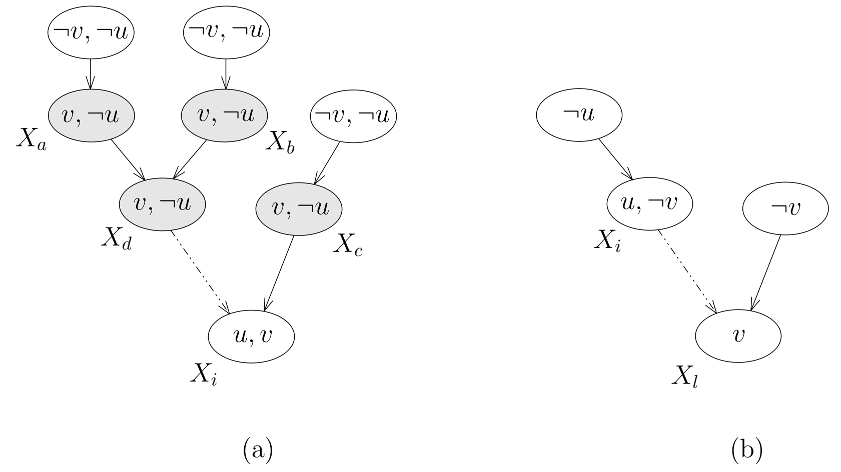


Figure 3.7: An example illustrating our construction. In (a), we will add u to all the bags X_l shaded in gray, $l \in \{a, b, c, d\}$. The bag X_l in (b) will remain unchanged.

Correctness Note that in our construction above, we only modify the contents of the bags in the DAG decomposition D and do not add or remove any bag. Therefore, the DAGs D and D' are exactly the same: just the contents of the bags have changed. We will refer to a bag X_i of D as X'_i in D'. Note that as usual we will use $i \preceq j$ to indicate that there is a path from i to j in D' (which was also there in D).

For all bags X'_l in D' which were modified in our construction, that is $X'_l \neq X_l$, we note that the following properties hold true. These will be used later in the proof.

P.1 $X'_l = X_l \cup \{u\}$.

This follows directly from our construction.

P.2 *There exists a successor j of l such that u was introduced in X_j and $v \in X_j$.*

Recall that we added u to X_l only if $u \notin X_l$, $v \in X_l$ and there existed a bag X_i, $l \prec i$ such that $u, v \in X_i$. It follows that there must exist a bag X_j where u first appears in the path from l to i. It is easy to see that X_j must contain v due to the connectivity condition on D.

P.3 *If u is introduced in X'_l, then v was introduced in X_l.*

Since u is introduced in X'_l, there exists an immediate predecessor k of l such that X'_k does not contain u. Moreover, since $X'_l \neq X_l$, we have $v \in X_l$ and a successor X_i with $u, v \in X_i$. If v were in X_k, then X_i would be a successor of X_k and hence we would have also added u to X_k. But this contradicts the fact that X'_k did not contain u. Therefore, we must have $v \notin X_k$, implying that v was introduced in X_l since $v \in X_l$ and $(k, l) \in E(D)$.

We will now verify that the vertex-covering, edge-covering and connectivity conditions are satisfied on D'. Recall that since we contract the edge (u, v), there are new edges (u, w) in G' for every edge (v, w) in G.

- *Vertex covering condition.* The vertex covering condition is trivially satisfied since we did not remove any vertex from the bags of D.
- *Edge covering condition.* We need to verify this condition only for the vertex u as all the other vertices of G' are introduced in the same bag in D' as they were in D, and so the edges are covered in D' because they were covered in D.

  Let X'_p be a bag in which u is introduced. There are two subcases:

  1. First consider an edge (u, x) with $x \neq w$.
     (a) If $X'_p = X_p$, then (u, x) is covered in D' since it was covered in D.
     (b) If $X'_p \neq X_p$, then by P.2 there exists a successor j of p such that u was introduced in X_j. Since (u, x) existed in G, x must be present in some successor-bag of X_j. Since $p \preceq j$, this same successor-bag now covers the edge (u, x) in D'.
  2. Now, consider the edge (u, w). We claim that there exists some successor bag X'_s of X'_p where v is introduced. This holds by P.3 if $X'_p \neq X_p$, so assume $X'_p = X_p$. We have two cases:
     (a) $v \in X_p$. Let X'_j be an immediate predecessor of X'_p with $u \notin X'_j$; this exists since u is introduced in X'_p. Then $v \notin X_j$, since otherwise the addition-rule would have applied to bags X_j and X_p and we would have added u to X_j. So $v \notin X_j$, and hence v is introduced in X'_p.
     (b) $v \notin X_p$. Since u is introduced in $X'_p = X_p$, the edge-covering condition for (u, v) implies that there exists X_t, $p \preceq t$ with $v \in X_t$. So $v \notin X'_p$, $v \in X'_t$, and somewhere along the path from p to t is a bag X'_s where v is introduced.

     So in both the cases above, we found a successor-bag X_s of X_p $(p \preceq s)$ where v was introduced. As (v, w) was an edge in G, w must be present in some successor-bag X_t of X_s. Bag X_t now covers the new edge (u, w) in D' since $p \preceq s \preceq t$.

- *Connectivity condition.* We want to verify that for all $p \preceq q \preceq r$, $X'_p \cap X'_r \subseteq X'_q$ holds.

  From the connectivity condition on D, we already have $X_p \cap X_r \subseteq X_q$. Recall that the only way we modify D is by adding u to some bags containing v. That is, $X'_i \subseteq X_i \cup \{u\}$ for all bags X_i. Hence $X'_p \cap X'_r \subseteq (X_p \cap X_r) \cup \{u\} \subseteq X_q \cup \{u\}$.

  There are two cases:

  1. If $u \notin X'_p \cap X'_r$, then $X'_p \cap X'_r = X_p \cap X_r \subseteq X'_q$ follows.
  2. If $u \in X'_p \cap X'_r$, then $u \in X'_p$, $u \in X'_r$ and it suffices to show that $u \in X'_q$. Assume for contradiction that $u \notin X'_q$ but $u \in X'_p$ and $u \in X'_r$. We have the following subcases:

     (a) $u \notin X_r$. By $u \in X'_r$ we have $v \in X_r$ and some successor-bag X_s of X_r that contains both u and v. Since X_s is also a successor-bag of X_q, this implies $v \notin X_q$ (otherwise we would have added u to X_q, but $u \notin X'_q$). So we have $u, v \notin X_q$ and $u, v \in X_s$. By the connectivity condition and $p \preceq q \preceq s$, we must have $u, v \notin X_p$. This makes $u \in X'_p$ impossible.

     (b) $u \in X_r$. Let X_n be the first bag on the path from X_q to X_r that contains u. Since $u \notin X_q$, vertex u is introduced in X_n. By the edge covering condition, some successor-bag X_s of X_n contains v. Since $u \in X_r$ and $u \notin X_q$, by the connectivity condition $u \notin X_p$. However $u \in X'_p$, therefore $v \in X_p$. Since $p \preceq q \preceq n \preceq s$ and $v \in X_p$, $v \in X_s$, we have $v \in X_q$ and $v \in X_n$. Now $v \in X_q$ and X_q has the successor-bag X_n containing both u and v, so we would have added u to X_q. This contradicts $u \notin X'_q$.

From above it follows that D' is a valid DAG decomposition for G'. Finally, since vertex v doesn not exist in G', the DAG decomposition D'' obtained by removing the vertex v from all bags containing it in D' is a valid DAG decomposition of G' and has the desired width. □

Theorem 6. *The DAG-width of a digraph $G = (V, E)$ does not increase when subdividing an edge.*

Proof. This follows trivially from the cops and robber characterization of DAG-width, but here we give a constructive proof to show that the size of the DAG decomposition stays the same. There are three possible ways of directing the two new edges when we subdivide the edge $(u \to w)$ by adding a vertex v of degree 2. (We are only considering those directions where the inverse operation, that is, removing v by contracting one of the two newly added edges, results in the original edge ($u \to w$).) We will show that, given a DAG-decomposition (D, X) of G, we can transform it to get a DAG-decomposition of G' in each of the three cases. (See also Figure 3.8.)

1. $u \leftarrow v \to w$. In this case, we first find some bag X_i in which u is introduced. Find a predecessor h of i which is a source in D (since D is acyclic such a predecessor always exists). Now create a new bag $X_k = \{v\}$ and connect X_k to X_h by adding an arc (k, h). Note that k will now replace h as a source in D. Also note that it was important for h to be a source in D because otherwise adding the edge (k, h) may introduce new vertices in X_h. The vertex-covering and the connectivity conditions are easy to verify. For the edge-covering condition, note that v is the only vertex introduced in X_h, so we just need to verify that its outgoing edges are covered. Clearly the edge (v, u) is covered in the successor-bag X_i. Moreover, since (u, w) was an edge in G it must be covered in some successor-bag X_j, $i \preceq j$. This same successor bag will now cover the edge (v, w) at k since $k \prec i$.

2. $u \to v \leftarrow w$. Find all the bags X_i in which either u or w is introduced. Create a new bag $X_k = \{v\}$ and connect it with every such X_i by adding the arcs (i, k). Note that k will be a sink in the DAG-decomposition. The vertex-covering, edge-covering, and connectivity conditions are easy to verify.

3. $u \to v \to w$. We note that for the special case when G is a DAG (width 1), subdivision also results in a DAG which is a valid DAG-decomposition of itself. For the general case, we will need to do some more work.

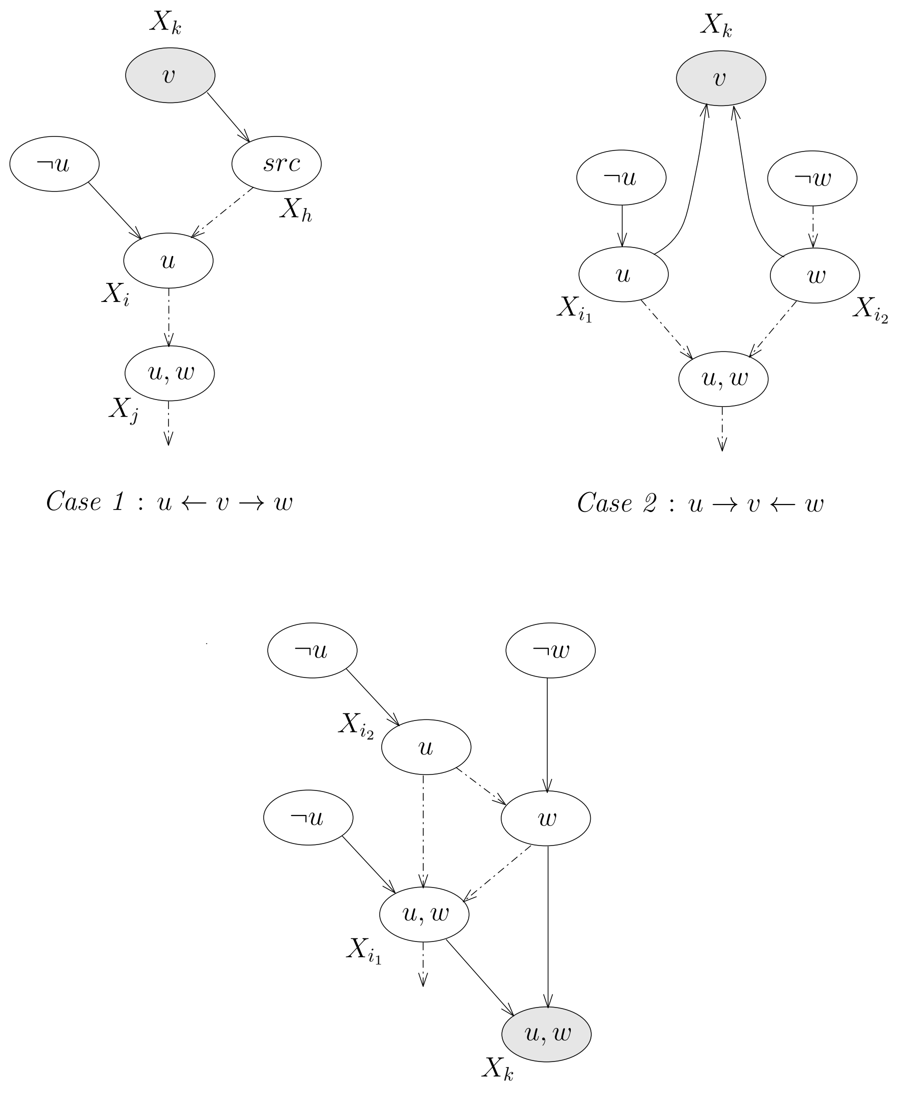
   (a) Create a new bag $X_k = \{v, w\}$.

   (b) For every bag X_i in which u was introduced, there are two cases:

      i. If X_i contains w, connect X_i to X_k by the arc (i, k).

      ii. If X_i does not contain w, then find all the successor-bags X_j in which w was introduced. Connect all these bags to X_k by the respective arcs (j, k).

   It is easy to see that the vertex-covering condition is satisfied. For the edge-covering condition, we note that the only vertex introduced in X_k is v. Since $w \in X_k$, the edge (v, w) is covered in the newly added bag X_k. Also, in the bags X_i where u is introduced, the edge (u, v) is covered in X_k: either via the arc $i \to k$ added in 3(b)i, or via some successor bag X_j with $i \preceq j$ and $j \to k$. Recall that such a X_j must exist by the edge-covering condition applied on the original edge (u, w) at X_i.



Case 3 : $u \rightarrow v \rightarrow w$

Figure 3.8: DAG decomposition on subdividing. The newly added bag X_k is shaded in gray. Note that in Case 3, u is introduced in both X_{i_1} and X_{i_2}, but different conditions namely 3b(i) and 3b(ii) apply.

For the connectivity condition, we note that for any $p \preceq_D q \preceq_D k$, $X_p \cap X_k \subseteq \{w\}$. The case $X_p \cap X_k = \emptyset$ is trivial. For the case $X_p \cap X_k = \{w\}$, we note that by our construction, $q \preceq_D k$ if and only if there exists a node $r \succeq_D q$ such that $w \in X_r$. Now since $p \preceq_D q \preceq_D r$ and both X_p and X_r contain $\{w\}$, X_q must contain $\{w\}$ as well. Hence we have, $X_p \cap X_k = \{w\} \subseteq X_q$.

□

3.4 DAG-width of programs with labelled breaks

Recall that in Section 2.1, we assumed the control flow graphs to be derived from structured programs (Definition 1). There, we did not consider `goto` statements and assumed that `break` statements can only lead to the nearest surrounding loop. However, some programming languages such as *Java* and *Ada* offer an additional construct called a *labelled* `break`. Such statements typically have a syntax '`break` *label*', where *label* is a unique name assigned to some loop that encloses this statement. Unlike traditional `break` statements which only lead to the exit of nearest surrounding loop, labelled `break` statements can lead to the exit points of any parent loop. For example, let L_4 be a loop element nested under L_2 which in turn is nested under L_1. The traditional `break` statement at a vertex $v_4 \in belongs(L_4)$ results in an edge to L_4^{exit} whereas a labelled `break` could also result in edges to L_2^{exit} and L_1^{exit}. (See the dotted edges in Figure 3.10).

Gustedt et al. [18] showed that Java programs have unbounded treewidth in the presence of labelled breaks. Burgstaller et al. [13] gave a similar result for Ada programs. In this section, we show that DAG-width of programs with labelled `break` statements is also unbounded. We construct a control flow graph G_k for which the robber always has a winning strategy against k cops, thereby showing that the DAG-width of G_k is at least $k+1$. Here k is an input to our construction. Note that our construction is inspired by [6, proof of Proposition 8]. We will start by proving a claim that will be useful later.

Let T and T' be two complete binary trees of height k. Now, consider the graph $G(2,k) = T \cup T'$. That is, $G(2,k)$ is a forest comprising the trees T and T'. Now we make it directed by orienting the edges in T to be away from the root and those in T' to be towards the root (See Figure 3.9). For the sake of clarity, we refer to the vertices in T by v and those in T' by v'. Observe that every vertex v in T has a corresponding *double* v' in T'.

We observe that the following holds for the k-cops and robber game on $G(2,k)$.

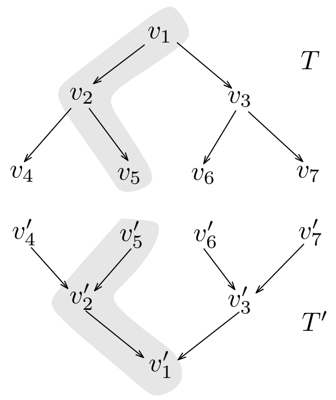


Figure 3.9: The digraph $G(2,k)$ with $k = 2$. We say that the path $v_1 \rightsquigarrow v_5$ is cop-free if none of the vertices in $v_1 \rightsquigarrow v_5$ or $v'_5 \rightsquigarrow v'_1$ is occupied by a cop.

Claim 2. *Let u be a leaf in T. Then, there always exists an ancestor v of u that has a path to a leaf w of T such that all the vertices in the path from v to w and their corresponding doubles in T' are not occupied by any of the k cops.*

Proof. Observe that the leaf u has $k+1$ ancestors in T including itself. Therefore, we can find $k+1$ vertex disjoint paths from the ancestors of u to some leaf w of T. We say that a path from an ancestor v of u to some leaf w is *blocked* if any of the vertices in $v \rightsquigarrow w$ or $w' \rightsquigarrow v'$ is occupied by the cops, otherwise we call it *cop-free*.

Since we only have k cops and all the paths are vertex disjoint, each cop can block at most one such path. However, we have $k+1$ such paths and therefore one of them must be cop-free. □

Theorem 7. *If labelled breaks are allowed, then for all $k > 0$, there exists a control flow graph G_k that has DAG-width at least $k+1$.*

Proof. In our construction for G_k, we have $k+1$ levels of nested loop elements. Assuming the top level to be 0, we have 2^i loop elements at level i. These are combined together by conditionals such that two loop elements at level i are directly nested under each loop element at level $i-1$ as shown in Figure 3.10(a). L_1 is the topmost loop element. For every loop element L_i we have a vertex v_i such that $v_i \in belongs(L_i)$. For all the loop elements at the leaf level, we have a chain of `if` statements that have outgoing edges to exit points of each of the ancestor loop elements. For example, in Figure 3.10(a), we have edges from $inside(L_4)$ to each of L_4^{exit}, L_2^{exit} and L_1^{exit}.

We now simplify G_k as follows.

1. Contract the `if`-chain at the leaf level to a single vertex v_i (shaded vertices in Figure 3.10(a)).

2. Contract the edges (L_i^{entry}, if) for all loop elements L_i (dash-dotted edges in Figure 3.10(a)).

3. Finally, remove L_1^{exit} from G_k as we do not need it for our proof.

Note that all the edges (u, v) that we contracted had *in-degree*$(u) = 1$, and therefore by Theorem 5, contracting them cannot increase the DAG-width. A simplified graph G'_k with $k = 2$ is shown in Figure 3.10(b).

Therefore, in order to show that G_k has DAG-width at least $k + 1$, it suffices to show that the robber has a winning strategy against k cops on G'_k. Now, the digraph G'_k can be thought of as being obtained from $G(2, k)$ by adding some extra vertices and edges. Recall that $G(2, k)$ is the digraph obtained by taking union of two complete binary trees T and T', and orienting the edges in T to be away from the root and those in T' to be towards the root. Now in G'_k, every L_i^{entry} in the upper tree T has the corresponding double v_i in T'. Additionally, every leaf u' in T' has a directed edge to every L_i^{exit} that is an ancestor of u'. Recall that these edges are due to labelled break statements. As a consequence, there is a directed path of length at most 2 from a leaf to any ancestor of T'.

It is not hard to see that Claim 2 also holds for G'_k since every leaf in T and T' still has $k + 1$ ancestors (including itself) that have vertex-disjoint paths to a leaf.

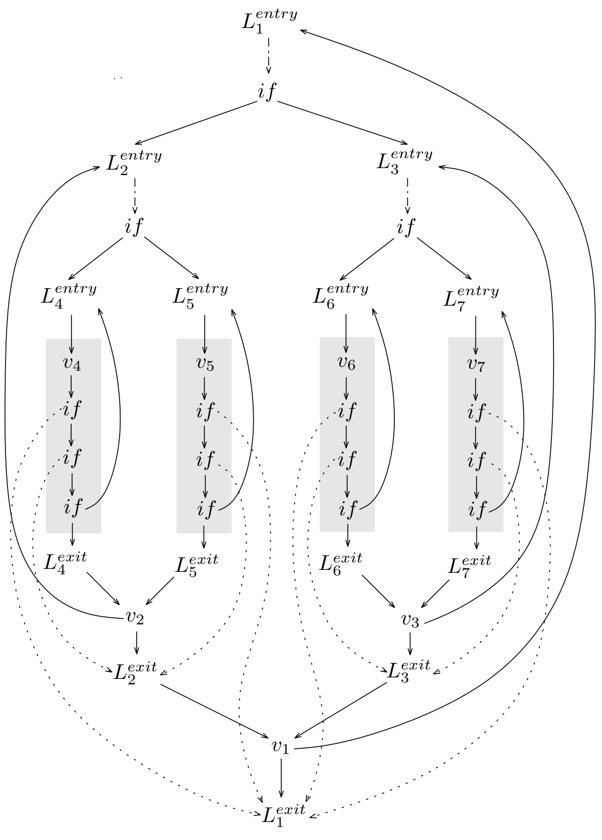
The robber uses the following strategy against k cops. Initially, he picks an arbitrary leaf in T'.

1. At this point, the robber maintains the invariant that he is at a leaf u' of the tree T'. That is, $r_i = u'$.

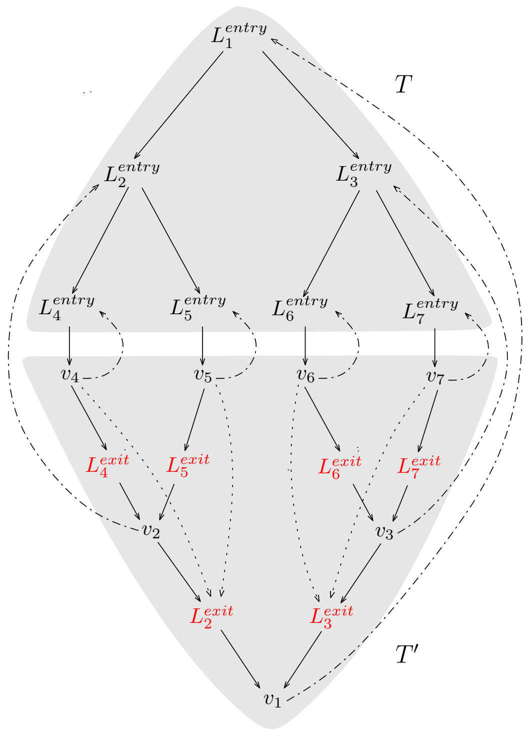
2. The cops decide to move to X_{i+1}.

3. The robber selects an ancestor v of u such that the paths $v \rightsquigarrow w$ and $w' \rightsquigarrow v'$ are cop-free with respect to X_{i+1}, the next move of the cops. This holds by Claim 2.

4. The robber now moves along the path $u' \rightarrow L_x^{exit} \rightarrow v' \rightarrow v \rightsquigarrow w \rightarrow w'$ such that $r_{i+1} = w'$.



(a)



(b)

Figure 3.10: (a) A control flow graph G_k with $k = 2$ and (b) its simplified version G'_k. The edges $L_i^{entry} \rightarrow L_i^{exit}$ are not shown for clarity. We simplify G_k by combining the `if`-chain (shaded in gray), contracting the dash-dotted edges and finally removing L_1^{exit}. Note that G'_k can be interpreted as a concatenation of two complete binary trees T and T' of height k, plus some extra edges and vertices.

Here, L_x^{exit} is an ancestor of u' that immediately precedes v'. Recall that by our construction, the edge $(u' \to L_x^{exit})$ always exists. Also note that all the vertices in this path do not contain a cop by our choice of v.

5. Go back to step 1 with $i = i + 1$ and $u' = w'$.

The robber can keep repeating the above steps and can evade capture forever.

□

Chapter 4

Other Digraph Width Parameters

In this chapter we will discuss some other digraph width measures and determine whether or not they are bounded for control flow graphs. The width measures we consider in this chapter are Kelly-width and entanglement. The choice of these measures is motivated by efficient algorithms for the μ-calculus model checking problem on graphs of bounded Kelly-width [10] and of bounded entanglement [6].

4.1 Kelly-width

The Kelly-width generalizes the concept of an elimination ordering (Definition 6) to directed graphs.

Definition 14 (Directed Elimination Ordering)**.** *An* elimination ordering *of a directed graph $G = (V, E)$ is a linear ordering on its vertices V. Given an elimination ordering $\pi = (v_1, ..., v_n)$, we define G_{i-1} to be the graph obtained by 'eliminating' v_i from G_i. That is, let G_n be G and let G_{i-1} be the digraph obtained from G_i by removing v_i from G_i and adding new edges (u, v) (if needed) for all $u, v \in V(G_i), u \neq v$ such that $(u, v_i) \in E(G_i)$ and $(v_i, v) \in E(G_i)$. The newly added edges are called* fill edges*.*

The graph $G' = (V, E \cup F)$ is called the *fill-in* graph of G with respect to the elimination ordering π. Here F is the set of all the fill edges that were added during the elimination.

The width of the elimination is defined as the maximum over all i of the out-degree of v_i in G_i. The *Kelly-width* of G is one more than minimum width across all possible elimination orderings. The extra plus one comes from the fact that the number of cops needed to search the graph will be one more than the width of the elimination.

As an example, consider G to be a directed acyclic graph with vertices $v_1, v_2, \ldots, v_n$ and edges (v_i, v_j) for all $j > i$. It is not hard to see that the ordering $\pi = (v_1, v_2, \ldots, v_n)$ is an elimination ordering of vertices in G with width zero. It follows that the Kelly-width of G is 1. In fact, Kelly-width of any directed acyclic graph is 1. This will be more clear with the cops and robber characterization of Kelly-width presented in the next section.

4.1.1 Inert Robber Game and Kelly-width

Kelly-width is related to a variant of cops and robber game called the *inert* robber game in which the robber is invisible but can only move when he sees a cop approach his current position. A play in the (k-cop) inert robber game on a directed graph $G = (V, E)$ is a sequence

$(X_0, R_0), (X_1, R_1), ..., (X_m, R_m)$

where $X_i \subseteq V, |X_i| \leq k$ is the set of positions for k cops whereas R_i is the set of potential robber locations. Initially, $X_0 = \emptyset$ and $R_0 = V$. If the cops catch the robber at the end of the play, then $R_m = \emptyset$. Since the robber can move only if the cop approaches its current position, we have:

$$R_{i+1} = \left(R_i \cup \bigcup_{v \in R_i \cap X_{i+1}} Reach(X_i \cap X_{i+1}, v) \right)$$

Recall that $Reach(X_i \cap X_{i+1}, v)$ is the set of vertices reachable form v in the digraph $G \setminus X_i \cap X_{i+1}$. Also recall that a strategy is called robber-monotone if in a play consistent with the strategy, the set of potential robber locations is non-increasing. In other words $R_j \subseteq R_i$ for all $i < j$.

We note the following result:

Theorem 8. *[19, Theorem 5] A digraph G has Kelly-width k if and only if k cops have a robber-monotone winning strategy in an inert robber game on G.*

Note that unlike the case of DAG-width, cop-monotonicity does not imply robber-monotonicity in the cops and robber game for Kelly-width. That is, it is possible that a

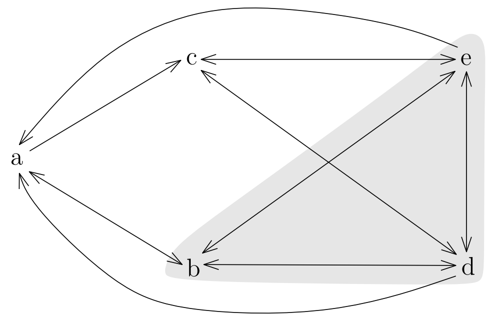


Figure 4.1: A digraph G with Kelly-width 3 and DAG-width 4. The shaded region denotes the clique $C_1 = \{b, d, e\}$. Note that G has Kelly-width and DAG-width at least 3 as it contains a clique of size 3.

robber-monotone winning strategy in the game for Kelly-width is not cop-monotone. In fact, we will later see that the winning strategy for the inert cops and robber game on a control flow graph is robber-monotone but not cop-monotone.

In Figure 4.1, we show a digraph for which DAG-width and Kelly-width are different. Note that G has two cliques of size 3: $C_1 = (b, d, e)$ and $C_2 = (c, d, e)$. One can see that three cops can catch a lazy invisible robber in G by successively placing themselves at $(b, d, e) \to (c, d, e) \to (c, d, b) \to (c, a, b)$. However, a robber that is not lazy always has a winning strategy against three cops, even if he is visible. He can start at any vertex in the clique C_1 and stay there as long as at least one of the three cops is outside C_1. If the third cop approaches a vertex in C_1, the robber will flee to vertex 'a' as every vertex in C_1 is connected to 'a' by a directed edge. Now, the robber will stay there until some cop leaves C_1. When this happens, the robber will come back to a vertex in C_1, and will keep doing this forever.

4.1.2 Kelly-width of Control Flow Graphs

The bound on Kelly-width of control flow graphs follows by adapting the cops and robber strategy from Section 3.1.1. There, we gave a winning strategy for catching a visible robber with three cops. Note that in the game for Kelly-width, the cops cannot see the robber. The idea now is to search the entire graph while making sure that the set of potential locations for the robber is non-increasing. We present the following modified strategy f_k :

1. Start with $L = L_\phi$, the outermost loop element.

2. Move cop $X(2)$ to L^{exit}.

3. Move cop $X(1)$ to L^{entry}.

4. Move $X(3)$ to every vertex in *belongs*(L) in a depth-first order.

5. For every loop element L_i directly nested inside L, go to step 2 with $L := L_i$.

6. Move to `stop` vertex.

Lemma 7. *The strategy f_k is winning.*

Proof. In step 4, the cop $X(3)$ searches every vertex in *belongs*(L). Therefore, if the robber was present on one of the vertices in *belongs*(L), he must flee to *inside*(L_i), for some L_i nested under L, or to `stop` in order to avoid capture. This holds because *belongs*$(L) \setminus L^{entry}$ does not contains a cycle (Corollary 1) and every time $X(3)$ visits the vertex r occupied by the robber, $dist(r, L^{exit})$ decreases (Lemma 2).

Note that the cops will try steps 2 through 5 for every loop element L_j directly nested inside L, even if the robber was not within *inside*(L_j). However, this does not change anything since the robber is lazy and must continue to stay within *inside*(L_i), the loop element it may have moved to in step 4 when the cops were searching L.

Nothing happens until the cops go to step 2 with $L = L_i$. Since $L_i^{exit} \in$ *belongs*(L), the cops have already checked for the robber at L_i^{exit} and placing a cop here simply forces the robber to stay within *inside*$(L_i) \cup \{$`stop`$\}$. Clearly, the robber is now restricted to a smaller set of vertices and will eventually be caught. If at any point the robber moved to the `stop` vertex, step 6 will ensure that he is caught. □

Lemma 8. *The strategy f_k is robber monotone.*

Proof. Note that since the robber is lazy, the set of potential robber locations R changes only when a cop moves to robber's current position. In step 2, it is guaranteed that the robber is not at L^{exit}. This holds initially since then $L = L_\phi$, and holds when we return to step 2, since by then we have checked all the vertices in *belongs*(L') where L' is the loop element to which L^{exit} belongs. Therefore, placing a cop at L^{exit} in step 2 cannot change anything. For step 4, due to the monotonicity of our distance function, the robber will be restricted to a smaller set of vertices if he was present in *belongs*(L) to begin with.

Finally for step 3, we note that if the robber was at L^{entry}, he must move to another vertex in *inside*(L), thereby reducing the set of potential robber locations by one. □

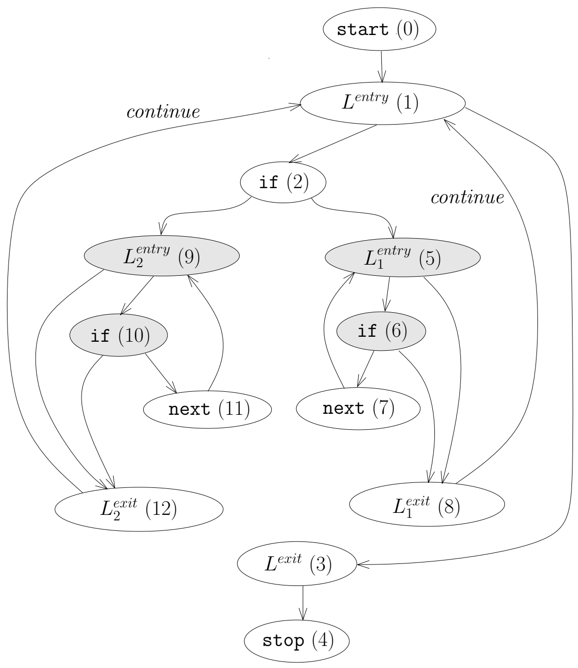


Figure 4.2: A lazy invisible robber has a winning strategy against two cops on this digraph. Note that at all times, the robber is at one of the shaded vertices.

However, it is important to note that the strategy is not *cop-monotone*, since the cops visit every L^{exit} twice.

The associated Kelly decomposition can be computed in linear time from the winning strategy f_k above using [19, Theorem 5 and Theorem 9]. From their proof, it also follows that the decomposition will have a linear size for control flow graphs.

It is not hard to see that the proof for the lower bound on the number of cops in the visible robber game (Lemma 5) carries over to the inert robber game. That is, the robber has a winning strategy against two cops on the digraph G in Figure 4.2. Recall that he does so by starting at one of the vertices in $\{5, 6\}$. That is, $r_i \in \{5, 6\}$. If $r_i \in X_{i+1} \neq \{5, 6\}$, then the robber keeps toggling between the vertices $\{5\}$ and $\{6\}$, thereby maintaining the invariant $r_i \in \{5, 6\}$. If $X_{i+1} = \{5, 6\}$, then he moves to the vertices $\{9, 10\}$ which are symmetric to $\{5, 6\}$.

Using this fact and the Lemmas 7 and 8, we have the following result.

Theorem 9. *The Kelly-width of control flow graphs is at most* 3 *and this is tight for some control flow graphs.*

4.2 Entanglement

Entanglement [6], as the name suggests, aims to measure the extent to which cycles of a directed graph are interwined. Like other digraph width measures, the entanglement of a digraph $G = (V, E)$ is defined via a variant of the cops and robber game on G with the following rules.

1. Initially, the robber selects any arbitrary vertex $r_0 \in V$ and all cops are outside G.
2. In any move, the cops can stay where they are or one cop moves to the current position $r \in V$ of the robber. Cops are not allowed to move to a position other than r.
3. The robber, in turn, *must* move to an immediate successor r' of r which is not occupied by the cops. That is, $(r, r') \in E$ and there is no cop at r'. It is important to note that the robber cannot stay at his current position, he must always move to an immediate successor.
4. The game terminates if the cops catch the robber or the robber is unable to move. Note that the cops can see the robber at all times and as in the previous versions, the robber knows exactly which vertices the cops will move to before he decides his own move.

The *entanglement* of the graph G is the minimal number k such that k cops can catch a robber on G in accordance with the rules mentioned above.

For example, if the digraph G is acyclic, the entanglement of G is zero. This holds because the robber must move at all times. As G is acyclic, he will eventually end up at a sink of G and hence cannot move any further.

If G contains directed cycles, then the entanglement of G is at least one. For a slightly more complicated example, consider a digraph with strongly connected components such that in each of these components, there is a *critical* vertex v_c whose removal makes the component acyclic. It is not hard to see that the entanglement of such a digraph is also one. Since all the cycles in that component must contain v_c, the robber will have to go there if he stays in that component. The cops can then simply place themselves at v_c, forcing the robber to move to another component in order to evade capture. However, he will eventually be caught at a terminal component.

Lemma 9. *The entanglement of a digraph $G = (V, E)$ does not increase on contracting an edge $(u, v) \in E$ for which u has out-degree* 1.

Proof. This follows from the cops and robber characterization of entanglement. Recall that the cops can either move to the current position of the robber or stay at their current positions. Therefore, it strategically makes sense for the cops to place themselves at a vertex only if they are going to stay there and block some path(s) for the robber in the future rounds.

Now, let f_k be the winning strategy with k cops. Consider the case when one of the cops moved to u at the end of the i^{th} round. Note that this can happen only if the robber was at u in the previous round, and he moved to its only successor v. Therefore, $r_i = v$ and $u \in X_i$. For the next round, the cops have three choices:

1. Do not move a cop.
2. Move the cop from some other vertex w to v.
3. Move the cop at u to v.

Let P_1, P_2 and P_3 be respectively the set of directed paths that are blocked by the cops, in each of the three cases above. Since every directed path that goes through u must also go through v, we have $P_1 \subseteq P_3$. In case 2, since there is no cop at w anymore, we have $P_2 \subseteq P_3$. It is easy to see that in all the cases, it is advantageous for the cops to switch from u from v, and hence they could have avoided moving to u in the first place.

Therefore, the winning strategy f_k would also have worked on G with the following modification:

The cops do not move in the round when they were supposed to move to u.
In the next round, the same cop which was supposed to move to u moves to v.

It is easy to verify that this modified strategy also works for G with the edge (u, v) contracted into a single vertex v. □

4.2.1 Entanglement of Control Flow Graphs

Berwanger et al. [6, Corollary 22] showed that parity games and hence the μ-calculus model checking problem can be solved in polynomial time on graphs of bounded entanglement. Therefore, if we could show that control flow graphs have bounded entanglement, we would obtain a polynomial-time algorithm for the μ-calculus model checking problem on control flow graphs. However, it turns out that control-flow graphs can have an unbounded entanglement. In this section, we will construct such a control flow graph.

We refer to the control flow graph in Figure 4.3(a). Recall that we used a similar construction in Section 3.4 to show unbounded DAG-width for programs with labelled breaks. Note that both these constructions are inspired from [6, Proposition 8].

Theorem 10. *For any $k \geq 0$, there exists a control flow graph G_k that has entanglement at least $k+1$.*

Proof. In our construction for the control flow graph G_k, we have a hierarchy of $k+1$ levels of nested loop elements combined using '`if`' statements as follows (see also Figure 4.3(a)).

Assuming the root is at level 0, the i^{th} level has 2^i loop elements. A loop element L at level i is directly nested under a loop element L_p at level $i-1$, via an '`if`' statement immediately following L_p^{entry}. We refer to these `if` statements by if. Note that the edges (L^{entry}, if) do not exist for the loop elements at the lowermost level k. Now for every loop element L, we have '`if`' statements immediately preceding L^{exit}. We refer to these by if' and give them an outgoing edge to L^{entry}.

We will now transform the control flow graph G_k to G'_k (see Figure 4.3(b)) by contracting certain edges of G_k as follows.

1. For every loop element L not at level k, remove the vertex if following L^{entry} by contracting the edge (L^{entry}, if).

2. For every loop element L, remove the vertex if' following L^{exit} by contracting the edge (L^{exit}, if'), if it exists.

In Figure 4.3(a), these edges are shown enclosed by the dotted region. Since all the edges we contracted had out-degree 1, it follows from Lemma 9 that the entanglement of G_k is at least as much as that of G'_k.

Recall from Section 3.4 that we used $G(2, k)$ (Figure 3.9) to denote the digraph obtained by taking the union of two complete binary trees T and T', and orienting the edges in T to

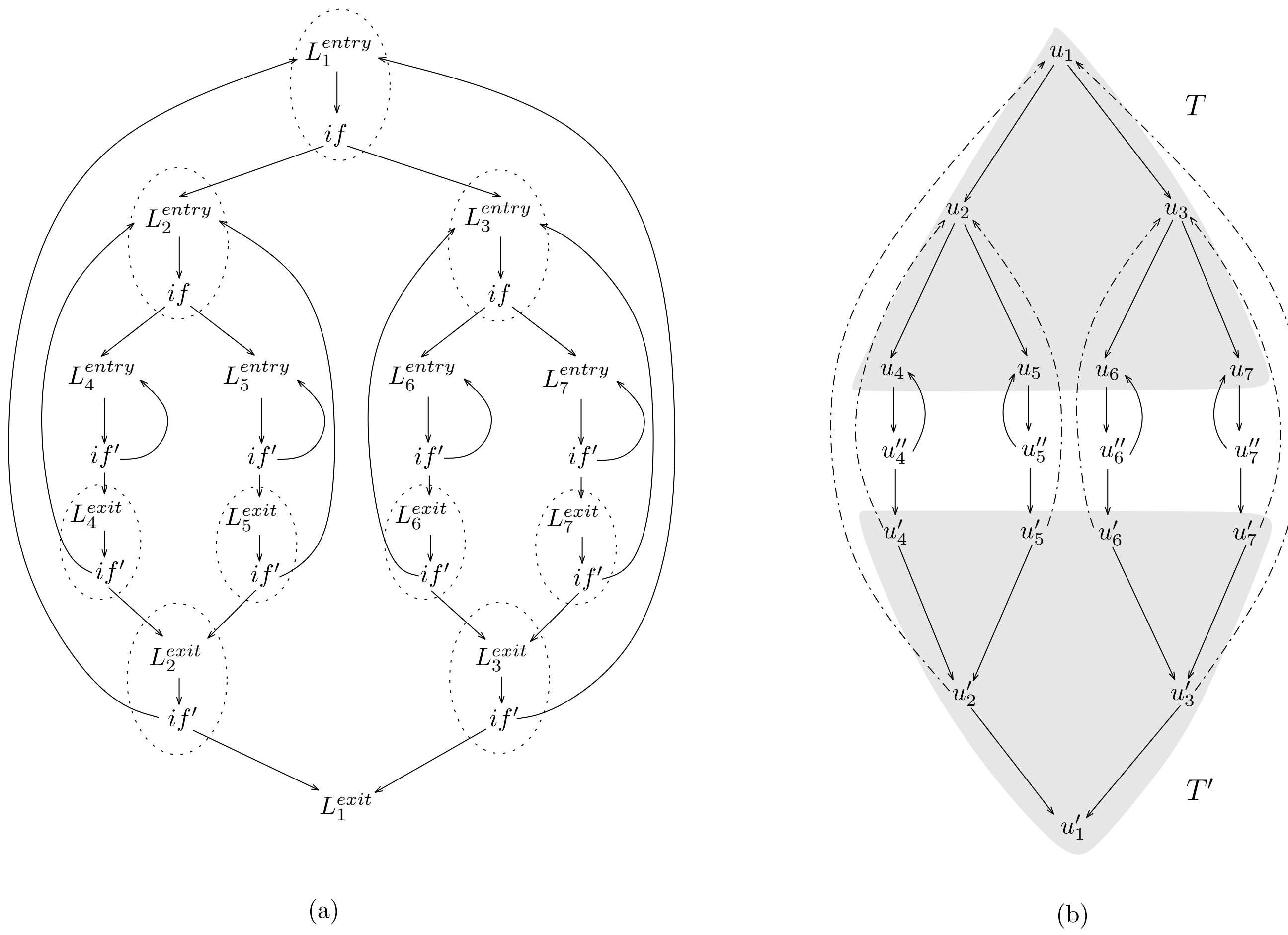


Figure 4.3: (a) A control flow graph G. The $L^{entry} \rightarrow L^{exit}$ edges are not shown for clarity. We combine certain consecutive vertices of G together to obtain G' shown in (b). Note that G' can be interpreted as a concatenation of two complete binary trees T and T' of height k, plus some extra edges. The edges of T and T' are oriented downwards and upwards respectively. In this figure, $k = 2$.

be away from the root and those in T' to be towards the root. The digraph G'_k constructed above can be thought of as being obtained from $G(2,k)$ by adding some extra vertices and edges. We will refer to the vertices L_j^{entry} in the upper tree T by u_j, and the vertices L_j^{exit} in the lower tree T' by u'_j. Therefore, every vertex v in T gets a *double* v' in T'. Since we still have the edges (if', L_j^{exit}) for a loop element L_j at the very last level, we will refer to these remaining vertices for if' by u''_j. Note that these vertices u'' connect the leaves u of T and u' of T'. We will call them *connectors*.

We will show that the robber always has a winning strategy against k cops on G'_k. In [6, Proposition 8], the authors very briefly showed that the entanglement of a digraph similar to our G'_k above is at least $k+1$. Here we build on their ideas and give a slightly different proof.

By Claim 2 we have that for every leaf u in T, there always exists an ancestor v of u that has a path to a leaf w of T such that all the vertices in the path from v to w and their corresponding doubles in T' are not occupied by the cops. Additionally in G'_k, we have extra vertices u'' connecting the leaves u and u' of T and T' respectively. However, adding extra vertices could only increase the number of cops needed and hence the claim also holds for G'_k. Therefore, for every leaf u of T we have an ancestor v such that the path $v \rightsquigarrow w \rightarrow w'' \rightarrow w' \rightsquigarrow v'$ is cop-free. Call this path P_v.

Starting at an arbitrary connector u'' in G'_k, the robber uses the following strategy:

1. At this point, the robber maintains the invariant that the path from the current chosen connector $r_i = u''$ to u'_1 (the root of T') does not contain a cop. Clearly, this holds initially, and we will argue it later for when we return here. Call this path Q_u.
2. Choose the lowest ancestor v of u such that the path $P_v = v \rightsquigarrow w \rightsquigarrow v'$ is cop-free with respect to X_{i+1}, the cops' next move. As argued before, such a path always exists. We have two cases:
   (a) If $v = u$, then the robber moves along the edge $u'' \rightarrow u$. Note that $u'' \notin X_{i+1}$, otherwise the robber would not have chosen $v = u$ in the first place. He then moves down the edge $u \rightarrow u''$ and goes back to step 1. The robber has not yet walked along the path Q_u, so no cop have moved there and the invariant holds.
   (b) If $v \neq u$, then the path $v \rightsquigarrow v'$ must go through the subtree of v that does *not* contain u.
   Assume for contradiction that the path goes through a node x in the subtree of v that contains u. Clearly, the sub-path $x \rightsquigarrow v'$ is also cop-free. However, this contradicts the claim that v was the lowest such ancestor. Continue at step 3.

3. Let x' be an ancestor of u'' along the path Q_u, that has the backward edge (x', v) (such edges are dash-dotted in Figure 4.3(b)). The robber now moves along $u'' \rightsquigarrow x' \to v \rightsquigarrow w \to w''$. That is, partly along Q_u ($u'' \rightsquigarrow x'$) and then the rest along P_v. Clearly, both these paths do not contain a cop.

4. Go back to step 1 with $u'' = w''$. Observe that Q_w (the path from w'' to v'_1) is cop-free since it consists of those parts of P_v and Q_u that the robber didn't walk along. This was cop-free before, and no cop can have moved here since cops can only move to where the robber was.

$\square$

Chapter 5

Conclusion and Open Problems

Motivated by applications in software verification, the original goal of this thesis was to obtain better algorithms for the μ-calculus model checking problem on control flow graphs. Since there are algorithms in the literature that solve the μ-calculus model checking problem on some special graph classes such as graphs of bounded DAG-width, Kelly-width and entanglement, the idea was to see if these graph classes include the class of control flow graphs. This serves two purposes: (a) it can potentially give a better algorithm for the μ-calculus model checking problem, and (b) it adds a subclass of significant practical interest to these graph classes.

Towards this goal, in Chapter 3, we showed that the DAG-width of control flow graphs is at most 3 and gave a linear-time algorithm to compute a DAG decomposition with a linear number of edges. In Section 4.1, we showed that Kelly-width of control flow graphs is also bounded by 3. Finally, in Section 4.2, we showed that control flow graphs can have unbounded entanglement.

The results for DAG-width and Kelly-width look promising for solving the μ-calculus problem on a control flow graph $G = (V, E)$. Recall from Section 2.5.3 that the traditional approach of solving the μ-calculus problem on any of these graph classes is by translating it to the problem of finding a winner in parity games. For the case of DAG-width, we can use the algorithm by Fearnley and Schewe [17]. This runs in $O(|V| \cdot M \cdot k^{k+2} \cdot (d+1)^{3k+2})$ time where k is the DAG-width of the resulting parity game graph G', d is the alternation depth and M is the number of edges in the DAG decomposition. Using our main result (Theorem 3), we can obtain a DAG decomposition (D, X) of G of width 3 and $M \in O(|V|)$. Recall from Section 2.5.3 (Rule R.1) that the game graph G' will have m vertices for each vertex in the control flow graph G. Here, m is the number of sub-formulas of the formula we wanted to verify on G. We can then obtain a DAG decomposition of G' from (D, X) by replacing every $v \in X_i$ with all the m vertices for v. Note that this will have width $3 \cdot m$ and $M \in O(|V|)$. Recall that even for the smallest possible values $m = 1$ and $d = 2$, the treewidth based algorithm runs in $O(|V| \cdot 7^{11} \cdot 3^{23}) = O(|V| \cdot 10^{20})$ time. For the same values, the DAG-width based algorithm runs in $O(|V|^2 \cdot 3^5 \cdot 3^{11}) = O(|V|^2 \cdot 10^7)$, which is better unless $|V| \geq 10^{13}$.

However, the algorithm above still has a major drawback. The exponent of $(d + 1)$ grows quite rapidly with increase in m, the length of the formula, and as such verifying bigger formulas still seems quite impractical. A natural question to consider is whether we really need all the $3 \cdot m$ vertices in a single bag of the DAG decomposition for G'. We believe that this could be done with fewer vertices. Another natural open problem is hence to develop even faster algorithms for parity games on digraphs that come from control flow graphs. Our simple DAG decomposition that is directly derived from the control flow graph might be helpful here. In [10], the authors approach the μ-calculus model checking problem on graphs of bounded DAG-width and Kelly-width using dynamic programming techniques along with some techniques from logic. They show that the μ-calculus model checking problem can be solved in $O(f(k+m) \cdot n^c))$ time on graphs of bounded DAG-width and Kelly-width. Here, f is some computable function and c is some constant. However, their proof is not constructive and the running time may have large constant factors. Nonetheless, a natural extension is to apply their techniques on our DAG decomposition for control flow graphs to see if it can attain better running times this way.

We conclude with some problems of more theoretical interest. In Theorem 5, we gave an algorithm to transform the DAG decomposition of a digraph G to a DAG decomposition for G', where G' is obtained from G by contracting an edge (u, v), such that v has in-degree 1. However, we could not find a similar algorithm for the case when u has out-degree 1. Future work could focus on this and using these results to solve the more general problem of computing near optimal DAG decompositions when contracting an edge (u, v) where v has in-degree k or u has out-degree k.

Another interesting problem is the *path avoiding forbidden pairs* (PAFP) by Kolman and Pangrác [21]. Given a digraph $G = (V, E)$, two fixed vertices $s, t \in V$, and a set F of pairs of vertices (called *forbidden pairs*), the goal is to find a directed path from s to t that contains at most one vertex from each pair in F. The problem is known to be NP-hard even when G is DAG. However, Kolman and Pangrác studied it under some special cases and gave a polynomial time algorithm for the case when G is a DAG and the forbidden pairs exhibit a special structure called the *hierarchical structure*. We believe that it can be worthwhile to study this problem on graphs of bounded DAG-width. More specifically, is there a polynomial time algorithm for the PAFP problem when G has bounded DAG-width and the forbidden pairs exhibit the hierarchical structure?

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Boston, MA, USA, 1986.

[2] F. E Allen. Control flow analysis. In *ACM SIGPLAN Notices*, volume 5, pages 1–19. ACM, 1970.

[3] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of Finding Embeddings in a k-Tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[4] D. Berwanger, A. Dawar, P. Hunter, and S. Kreutzer. DAG-Width and parity games. In *STACS 2006*, volume 3884 of *LNCS*, pages 524–536, 2006.

[5] D. Berwanger, A. Dawar, P. Hunter, S. Kreutzer, and J. Obdržálek. The DAG-width of directed graphs. *Journal of Combinatorial Theory, Series B*, 102(4):900–923, 2012.

[6] D. Berwanger and E. Grädel. Entanglement– A Measure for the Complexity of Directed Graphs with Applications to Logic and Games. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 209–223. Springer, 2005.

[7] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9(3):217–250, 2001.

[8] H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *ICALP'88*, volume 317 of *LNCS*, pages 105–118, 1988.

[9] H. L Bodlaender and A. M. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[10] M. Bojanczyk, C. Dittmann, and S. Kreutzer. Decomposition Theorems and Model-checking for the Modal μ-calculus. CSL-LICS '14, pages 17:1–17:10. ACM, 2014.

[11] J. Bradfield and C. Stirling. *Modal μ-calculi*, pages 721–756. Elsevier, 2007.

[12] A. Browne, E. M. Clarke, S. Jha, D.E Long, and W.R Marrero. An Improved Algorithm for the Evaluation of Fixpoint Expressions. *Theoretical Computer Science*, 178(1-2):237–255, 1997.

[13] B. Burgstaller, J. Blieberger, and B. Scholz. On the Tree Width of Ada Programs. *Reliable Software Technologies-Ada-Europe 2004*, pages 78–90, 2004.

[14] N. D Dendris, L. M. Kirousis, and D. M. Thilikos. Fugitive-search games on graphs and related parameters. *Theoretical Computer Science*, 172(1):233–254, 1997.

[15] A. E. Emerson. Model Checking and the μ-calculus. volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. American Mathematical Society, 1996.

[16] A. E. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for the μ-calculus and its fragments. *Theor. Comput. Sci.*, 258(1-2):491–522, 2001.

[17] J. Fearnley and S. Schewe. Time and Parallelizability Results for Parity Games with Bounded Tree and DAG Width. *Logical Methods in Computer Science,* www.lmcs-online.org, 9(2:06):1–31, 2013.

[18] J. Gustedt, O. A Mæhle, and J. A. Telle. The treewidth of Java programs. In *Algorithm Engineering and Experiments, ALENEX 2002*, volume 2409 of *LNCS*, pages 86–97. 2002.

[19] P. Hunter and S. Kreutzer. Digraph measures: Kelly decompositions, games, and orderings. *Theoretical Computer Science*, 399:206–219, 2008.

[20] T. Johnson, N. Robertson, P. D. Seymour, and R. Thomas. Directed tree-width. *Journal of Combinatorial Theory, Series B*, 82(1):138–154, 2001.

[21] P. Kolman and O. Pangrác. On the complexity of paths avoiding forbidden pairs. *Discrete Applied Mathematics*, 157(13):2871–2876, 2009.

[22] D. Kozen. Results on the Propositional μ-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[23] S. A. Kripke. Semantical Analysis of Modal Logic I. Normal Modal Propositional Calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963.

[24] J. Obdrzálek. Fast Mu-Calculus Model Checking when Tree-Width Is Bounded. In *CAV 2003*, volume 2725 of *LNCS*, pages 80–92, 2003.

[25] J. Obdržálek. DAG-width: Connectivity measure for directed graphs. In *SODA'06*, pages 814–821. ACM-SIAM, 2006.

[26] P. Hlinený R. Ganian, J. Kneis, A. Langer, J. Obdrzálek, and P. Rossmanith. Digraph Width Measures in Parameterized Algorithmics. *Discrete Applied Mathematics*, 168:88–107, 2014.

[27] N. Robertson and P. D. Seymour. Graph Minors. II. Algorithmic Aspects of Tree-Width. *J. Algorithms*, 7(3):309–322, 1986.

[28] P. D Seymour and R. Thomas. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B*, 58(1):22–33, 1993.

[29] C. Stirling. *Modal and Temporal Properties of Processes.* Springer-Verlag, New York, NY, USA, 2001.

[30] M. Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.