A Serverless Discrete Optimization Service in the Cloud Based on Boolean Circuit Satisfiability

by

Jialing Song

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2023

Author's Declaration

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

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

Abstract

This thesis discusses the implementation of a serverless cloud service designed for solving discrete optimization problems encoded as boolean circuit satisfiability. Boolean circuit satisfiability problem involves determining whether an input assignment exists that satisfies a given boolean circuit. A cloud service is a platform that offers on-demand computing resources and services over the Internet. In a serverless setup, the service automatically manages and scales resources, eliminating the need for manual server management.

The main objective of this thesis is to provide a comprehensive and efficient approach to address problems in NP (nondeterministic polynomial time) by utilizing boolean circuit satisfiability. Our novel cloud service offers clients a more universal, efficient, and user-friendly solution for solving problems in NP.

We proposed an enhanced boolean logical circuit that incorporates sub-circuits capable of performing mathematical operations, simplifying the reduction process and expanding the potential to tackle problems in NP across various domains. Our motivation for this work arises from the fact that the augmented boolean circuit satisfiability problem is NP-complete, and a wide range of discrete optimization problems can be reduced to it. By leveraging a reduction to the proposed enhanced version of boolean circuit satisfiability problem and using oracles such as conjunctive normal form (CNF) or Integer Linear Programming (ILP) solvers, our service can efficiently address problems in NP in which the decision version can be reduced to the improved circuit satisfiability problems within polynomial time.

This thesis covers in-depth knowledge about the fundamentals of computational complexity, presenting reductions from each of Karp's 21 NP-complete problems to the Circuit-SAT problem, demonstrating the versatility and applicability of our approach. Additionally, we discuss a software library we have built that enables the construction of circuits as well, allowing users to efficiently represent and solve problems using our cloud service. Furthermore, the thesis includes details about the service's working principles and deployment aspects.

Acknowledgements

I would like to sincerely express my gratitude to my graduate supervisor, Professor Mahesh Tripunitara, for his patient guidance and continuous support throughout my master's learning journey. His mentorship has been invaluable in shaping my academic growth and development.

I would like to extend my appreciation to Rahul Punchhi and Liran Tao, for their warmhearted help and support throughout my graduate career.

Dedication

This is dedicated to my parents Jie and Xiaoqian.

Table of Contents

List of Figures

List of Abbreviations

API Application Programming Interface 43, 55, 65

AWS Amazon Web Services 55

BFS Breadth-First Search 64, 65

CNF Conjunctive Normal Form 8, 10, 13, 16, 17

DAG Directed Acyclic Graph 5, 64, 65

DIMACS Discrete Mathematics and Theoretical Computer Science 17

ERWA Exponential Recency Weighted Average 18

FaaS Function as a Service 55

FVS Feedback Vertex Set 9, 26

IAM Identity and Access Management 65

IC Integrated Circuit 16

IoT Internet of Things 55

JSON JavaScript Object Notation 39

LRB Learning Rate Branching 18

MAB Multi-Armed Bandit 18

NP Nondeterministic Polynomial-time 1

P Polynomial-time 1

SAPS Scaling And Probabilistic Smoothing 17

SAT Satisfiability 13

TSP Traveling Salesman Problem 18

vCPUs virtual Central Processing Units 14

vGPUs virtual Graphics Processing Units 14

VMs Virtual Machines 15

VPC Virtual Private Cloud 15

VSIDS Variable State Independent Decaying Sum 18

List of Symbols

$\cap$ The "cap" symbol, which is used to indicate the intersection operation between two sets. 10

$\cup$ The "cup" symbol, which is used to indicate the union operation between two sets. 6, 10

$\emptyset$ The "empty set" symbol, which is used to indicate a set contains nothing. 12

$\forall$ The "for all" symbol, which is used to make universal general statements to every element in a set of objects. 2

$\in$ The "in" symbol, which is used to indicate the affiliation relationship between an element and a set. 2

$\wedge$ The "and" symbol, which is used to indicate the AND operation. 6

$\neg$ The "NOT" symbol, which is used to indicate the NOT operation. 6

$\notin$ The "notin" symbol, which is used to indicate the affiliation relationship between an element and a set. 6

$\subseteq$ The "subset equal" symbol, which is used to indicate the subset relationship between two sets. 10

$\vee$ The "or" symbol, which is used to indicate the OR operation. 6

Chapter 1

Introduction

With the rapid advancement of theoretical computer science, Nondeterministic Polynomial-time (NP) class has emerged as a prominent subject of investigation within the field of computational complexity classes. Numerous consequential computational problems arising in diverse domains such as science, engineering, and economics, can be effectively reformulated as problems in NP class. Effective resolutions of these challenges would yield substantial practical implications, including but not limited to enhancements in resource management methodologies, advancements in bio-molecular structural exploration, and the provision of more precise data analysis techniques. Additionally, research pertaining to the reduction of problems in NP and the development of efficient solving techniques holds the potential to offer insights into the long-standing and consequential "Polynomial-time (P) vs. NP" problem, that is, whether there exists a polynomial-time algorithm to solve the problem in NP. This question remains one of the most profound unresolved inquiries in computer science, and substantially impacts cryptography and many other domains related to security, given that numerous modern information security protocols are fundamentally constructed on the premise that $P \neq NP$.

This chapter first introduces the basic definition of abstract problem, algorithm, and computational complexity; then elicits the throughout research topic: problems in NP class. Then, a variety of examples of problems in NP and the process of reduction are illustrated. In the second part, the through-out-topic circuit satisfiability problem is introduced, which serves as the foundation for the solution proposed in the subsequent part of the thesis. In the end, 21 famous NP-complete problems are presented, which involve various categories of problems within the field of algorithms.

1.1 Computation Complexity

1.1.1 Problems and Algorithm

Problems in different fields of study are expressed in different formats. A statement of a "problem" in the computation area generally outlines the desired input and output relationship [7]. Besides, an "instance" of a problem is composed of the input needed to compute a solution to the problem. [7], and an "evidence" (a "certificate" or "witness") that is usually combined with a problem instance, is a piece of information that a deterministic polynomial-time algorithm can be used to verify the correctness. For example, the maximum value problem can be defined as:

Input: An array of n numerical elements: $A = [a_1, a_2, ..., a_n]$.
Output: An element a_i, where $1 \leq i \leq n$, such that $\forall\ a_j \in A$, where $1 \leq j \leq n$, $a_j \leq a_i$.

A valid instance of this problem is given the input sequence A as $[3, 7, 4, 2, 6, 8, 9, 0]$, and the corresponding certificate should be 9 since 9 is larger than or equal to any of the elements in array A.

Besides, it is of high necessity to introduce the concepts of "abstract problem" and "algorithm" because their further fine-grained classification and new definitions built upon them will be introduced later when computation complexity is involved.

Abstract Problem: Abstract problem is a binary relation on a set I of problem instance and a set S of problem solutions.[7].

Algorithm: Algorithm is a computational procedure which takes a set of values as inputs and generates a set of values as outputs [7].

1.1.2 Decision Problem and Optimization Problem

Decision problem: A problem with an output that can only be Yes or No [16].

Optimization problem: A problem which requires obtaining the best solution (associated with a maximum or minimum value) among all feasible solutions [7].

For example, to find the shortest path between two vertices in an undirected unweighted graph, the optimization version of it presents in the form that given a graph G and a pair of vertices $\langle u, v \rangle$, what is the shortest path connecting u and v with the least number of edges? The decision version of a problem usually can be converted from the optimization version by adding boundaries on the associated value [7]. Therefore, the decision version

of finding the shortest path is: given a graph G, a pair of vertices $\langle u, v \rangle$, and an integer k, if there exists a path from u to v which contains at most k edges?

The integer k mentioned above generally is the boundary of the related value for a possible solution. If there is an algorithm that could solve the decision version of a problem, then by modifying the boundaries on the associated value and other conditions, the optimization version can then be solved as well. Similarly, if an algorithm that could solve an optimization problem exists, then the decision version can be resolved by calculating the associated value based on the solution and comparing it to the boundary. Since the way of changing the boundary k and other attributes until reaching the extremum is usually more difficult than obtaining the related value of an optimized solution, the optimization version of a problem is considered "harder" than the decision version. The words "harder" and "easier" in this context refer to the fact that harder questions need more extra procedures to solve than easier ones.

In the finding shortest path example, if an algorithm exists for the optimization version (it returns the shortest path between u and v), the answer to the decision version can be obtained by counting the number of edges along this path and then comparing it to k. If the number of edges of the shortest path is even larger than the upper boundary k, then it is impossible to find another shortest path which satisfies this property, thus, the decision version should return "False"; otherwise, if the shortest path holds the condition, then the decision version is therefore "True". On the other hand, if there is an algorithm for the decision version, to get the shortest path in the optimization problem, first decrease k to find the number of edges in the shortest path. Then delete the start node (u) in G, and for every neighbour of u (denote as $u^{'}$), ask if there exists a path from $u^{'}$ to v which contains at most $k-1$ edges. If returns "True", then repeat the above procedure, if returns "False", try the next $u^{'}$ instead, until finding the satisfied vertex. Finally, all the deleted vertices in order compose the shortest path.

1.1.3 Deterministic and Non-deterministic Algorithm

When focusings on decision problems, deterministic and non-deterministic algorithms can be defined as below:

Deterministic Algorithm: A class of algorithms which exist only one state at a time before terminating.
Non-deterministic Algorithm: A class of algorithms which not only allow multiple states at a time when running, and hold the attributes that:

1. If the correct answer for the decision problem is "True", then at least one of the end

states when the non-deterministic algorithm terminates should be able to interpret it as "True".

2. If the correct answer for the decision problem is "False", then **none** of the end states when the non-deterministic algorithm halts could be interpreted as "True".

The non-deterministic property comes from the fact that instead of picking up parameters according to a specific scheme in a deterministic algorithm, a non-deterministic algorithm picks parameters to compute without any considerations.

1.1.4 Reduction

Reduction is the step to prove the "hardness" relationship between two problems. An instance of a problem refers to a particular input to that problem; [7]. For two decision problems A and B, the reduction from A to B is valid if and only if:

1. For every instance of A, there exists a polynomial time algorithm to convert it to an instance for problem B.

2. Assume there is an algorithm for problem B, the answer for an instance of A is "True" if and only if the algorithm returns "True" for the transformed instance for B.

The reduction from A to B above indicates the difficulty of solving problem A is no harder than the difficulty of solving problem B. Thus, if there doesn't exist an algorithm for solving A within polynomial time, then neither does B.

1.1.5 Computation Complexity Classes

Computation complexity is defined for the decision version of a problem. Below are several most common computation complexity classes:

P class: The collection of all decisions problems that can be resolved with an algorithm in polynomial time (there exists a constant k and an algorithm whose time-efficiency is $O(n^k)$) is in complexity class P.
NP class: The collection of all decisions problems that can be resolved with a non-deterministic algorithm in polynomial time (there exists a constant k and a non-deterministic algorithm whose time-efficiency is $O(n^k)$) is in complexity class NP. Besides, a problem

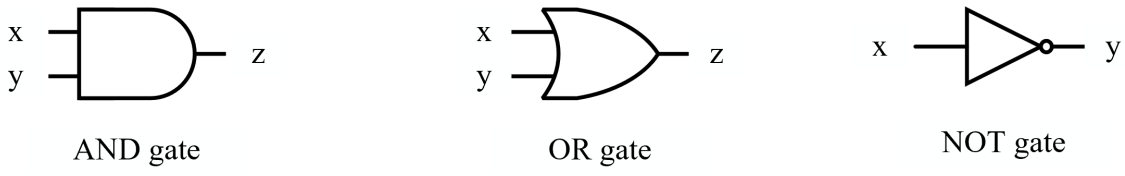
can also be categorized in NP if given a claimed solution of the problem, it can be verified within polynomial time.
NP-complete class: The collection of all decision problems in NP that any other problems in NP can be reduced to them within polynomial time.

Problems in NP are ubiquitous in everyday life, for example, checking if a given solution for Sudoku ($n \times n$) is valid can be accomplished in polynomial time in n, but it is not possible to find a solution within polynomial time, however, if there exists a non-deterministic algorithm which can non-deterministically pick numbers for each slot then verify, then return a correct solution is possible within polynomial time.

1.2 Logical Circuits and Circuit Satisfiability Problem

A logical circuit can be defined as a Directed Acyclic Graph (DAG) which consists of vertices and directed edges with the following constraints:

- Every directed edge is assigned a boolean value (0 or 1) and signifies the information flow from an input terminal to an output terminal in the overall graph.
- Every vertex in the graph represents an input or output terminal or a logical gate (AND gate, OR gate, and NOT gate). Logical gates work like functions that are able to conduct logical operations based on their own input boolean values, while input and output terminals are the boolean information sending and receiving elements in a graph.



x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

x	y
1	0
0	1

Figure 1.1: Three basic logical gates with their gate's operation.

AND gate performs "and" operation where $z = x \wedge y$, and $z = 1$ if and only if both input x and y are assigned with boolean value 1. OR gate performs "or" operation where $z = x \vee y$, and $z = 1$ if either input x or y, or both are assigned the boolean value 1. NOT gate performs the negation of the original input value, where $y = \neg\, x$.

- The indegree and outdegree of a NOT gate are both 1, while for AND and OR gates, their indegree is 2 and outdegree is 1 for each. The input terminal has an indegree of 0 and an outdegree of 1, while the output terminal has an indegree of 1 and an outdegree of 0.

A logical circuit G can also be composed of two sub-circuits G_1 and G_2, where $G_1 = \langle V_1, E_1 \rangle$, $G_2 = \langle V_2, E_2 \rangle$. The two sub-circuits hold properties that $\forall u \in V_1, u \notin V_2$, $\forall e \in E_1, e \notin E_2$ and vice versa. There are two ways to combine sub-circuits:

1. $G = G_1 \cup G_2$: graph G is the union of G_1 and G_2. The number of the input terminals in G is the sum of the number of input terminals in G_1 and G_2, and the same goes for output terminals as well. Besides, G_1 and G_2 do not share any common vertices or edges. Since both G_1 and G_2 are acyclic, their union does not contain any cycle either.

2. In the second way of combining G_1 and G_2, adjustments to G_1 and G_2 is made: new edges are established to connect vertices between sub-graph G_1 and G_2. Assume $u_1 \in V_1, u_2 \in V2$, in new graph G, there $\exists e \in E$ such that $e = \langle u_1, u_2 \rangle$ or $e = \langle u_2, u_1 \rangle$. In addition to the original G_1 and G_2 present in G, there are new edges directed from vertices in G1 to vertices in G2, or from vertices in G2 to vertices in G1. The constraint to this modification is to guarantee G is acyclic.

A circuit satisfiability (Circuit-SAT) problem can be stated as: given a logical circuit with a single output terminal, finding a possible assignment of boolean values to the edges originating from the input terminal, such that the value of the edge leading to the output terminal is 1. Below are two instances of the circuit satisfiability problem, and the only difference between them is that the OR gate in the upper right becomes an AND gate.

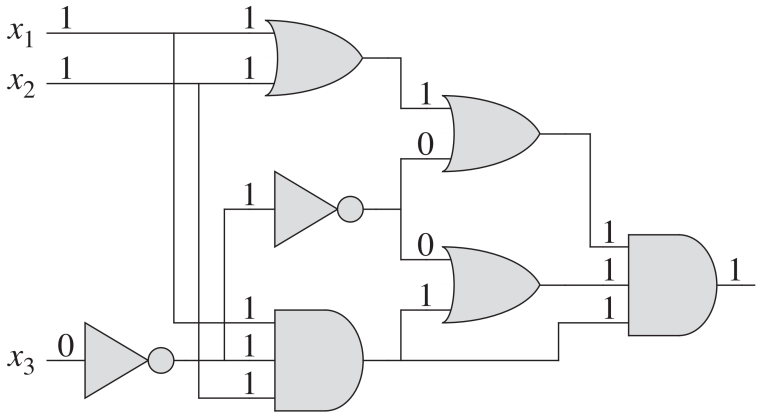


Figure 1.2: Instance (a), which is satisfiable since the certificate $x_1 = 1, x_2 = 1, x_3 = 0$ of this circuit results in the final output being 1. Source: Adapted from Introduction to algorithms [7]

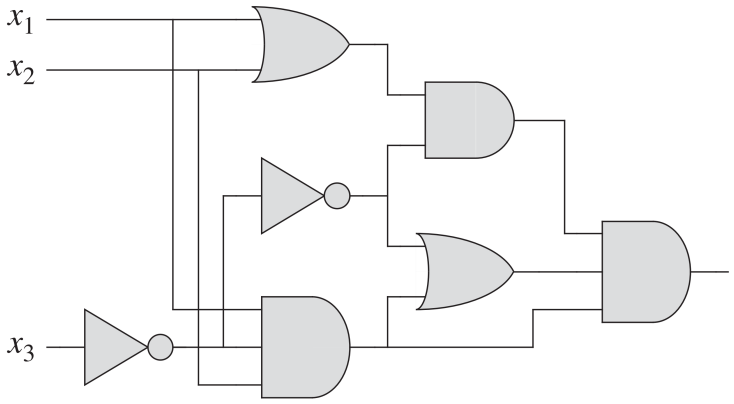


Figure 1.3: Instance (b), which is unsatisfiable since there is no assignment to the inputs of this circuit that will cause the final output to be 1. Source: Adapted from Introduction to algorithms [7]

Circuit-SAT problem can be proved to be in NP-complete class by reducing boolean

satisfiability problem to it, which is the first problem to be proved in the NP-complete class by Stephen Cook in 1971[6]. The service proposed in this thesis is based on solving circuit satisfiability problems reduced from other problems in NP, more details will be introduced in chapters 2 and chapter 3.

1.3 The Famous Karp's 21 NP-complete Problems

In 1972, Karp listed 21 famous NP-complete computational decision problems by reducing boolean satisfiability problem to each one of them. [15]. These classical problems not only come in various forms, with traces found in everyday life, but also serve as the foundation for many more complex problems in NP-complete. Below is the list of the decision version and optimization version of Karp's 21 problems which are NP-complete.

1. Boolean satisfiability problem: Given a boolean formula in Conjunctive Normal Form (CNF), if the variables can be assigned in a way that makes the CNF formula true.
2. 0-1 Integer programming: Given integer matrix $\mathbf{C}$ and an integer vector $\mathbf{d}$, if there exist a 0-1 vector $\mathbf{x}$ such that $\mathbf{Cx} = \mathbf{d}$.
3. Clique problem
   - Decision version: Given an undirected graph $G = \langle V, E \rangle$ and an integer k, if there exists a clique of size k for graph G?
   - Optimization version: Maximize integer k such that find out the maximum clique in a given undirected graph G.
4. Set packing
   - Decision version: Given a set $S = \{S_1, \ldots, S_n\}$, a universal set U, and an integer k, where collection S contains subsets of U. If subsets in S can form a set C of size k such that all subsets in C are pairwise disjoint?
   - Optimization version: Find the set C which holds the maximum number of mutually disjoint subsets (k).
5. Vertex cover problem
   - Decision version: Given a pair $\langle G, k \rangle$, where $G = \langle V, E \rangle$ is an undirected graph and $k \in \{1, \ldots, |V|\}$, does G have a vertex cover of size $\leq k$?

- Optimization version: Find the smallest vertex cover of graph G such that the size of vertex cover k reaches its minimum.

6. Set covering
   - Decision version: Given a set $S = \{S_1, \ldots, S_n\}$, a universal set U and an integer k, where S is a collection of subsets of U. If there exists a k-size (or less) set cover C, which consists of subsets in S and its union is U?
   - Optimization version: Find the smallest set cover for set S such that the size of set cover k reaches its minimum.
7. Feedback node set
   - Decision version: Given a graph $G = \{V, E\}$ and an integer k, if there exists a k-size Feedback Vertex Set (FVS) such that if all the vertices in FVS are removed, the whole graph will have no circles.
   - Optimization version: Find the smallest FVS for graph G such that the size of FVS k reaches its minimum.
8. Feedback arc set
   - Given a graph $G = \{V, E\}$, $E = \{e_1, \ldots, e_n\}$, and an integer k, if there is a k-size feedback arc set which is a set of edges such that G will not contain any circles if all edges in feedback arc set are removed.
   - Optimization version: Find the smallest feedback arc set for graph G such that the size of feedback arc set k reaches its minimum.
9. Directed Hamilton circuit
   - Decision version: If there exists a Hamiltonian path in the directed graph $G = \{V, E\}$ which makes it possible for visiting each vertex exactly once.
   - Optimization version: Find the longest Hamiltonian path in directed graph G.
10. Undirected Hamilton circuit
    - If there exists a Hamiltonian path in the undirected graph $G = \{V, E\}$ which makes it possible for visiting each vertex exactly once.
    - Optimization version: Find the longest Hamiltonian path in undirected graph G.

11. Satisfiability with at most 3 literals per clause: Given a boolean formula in 3-CNF format, if the variables can be assigned in a way that makes the 3-CNF formula true.

12. Chromatic number (Graph Coloring Problem): Given a graph $G = \{V, E\}$ and an integer k, if there is a way/function $c(\cdot)$ of labelling k colours to all vertices such that no two vertices connecting by an edge share the same colour, $c(u) \neq c(v)$ for $\forall(u, v) \in E$.

13. Clique cover

    - Decision version: Given a graph $G = \{V, E\}$ and an integer k, if there exists a way of partitioning all the vertices V in the graph G into k cliques.
    - Optimization version: Minimize the number of cliques k, such that find the smallest clique cover.

14. Exact cover

    - Decision version: Given a collection set $S = \{S_1, \ldots, S_n\}$, a universe set U and an integer k, where collection set S contains subsets of set U. If there exists a subcollection S^* of S whose size is k, such that every item in U is contained and only contained once in S^*.
    - Optimization version: Minimize k, the size of the subcollection S^*, such that find the smallest exact cover.

15. Hitting set

    - Decision version: Given a set $S = \{S_1, \ldots, S_n\}$, a universal set U, and an integer k, where S is a collection of subsets of U, and $S_1 \cup \ldots \cup S_n = U$. If there exists a k-size hitting set H, which is a subset of U, such that for every collection $S_i \in S$, the number of items shared by H and S_i is only 1, $|H \cap S| = 1$.
    - Optimization version: Minimize the number of items in hitting set H, so that the size of H reaches its minimum.

16. Steiner tree

    - Decision version: Given an instance of a Steiner tree problem in an undirected graph, $\{G = \langle E, V\rangle, T, k\}$, $V = \{u_1, \ldots, u_n\}$ and $E = \{e_1, \ldots, e_m\}$, with non-negative edges $w(e_i) \geq 0$. Terminal vertex set T is a subset of vertices of set V, and $k \in \mathbb{Z}$. Is there a tree $H = \langle E', V'\rangle$, which is a subgraph of G, such that $T \subseteq V'$ and the weight of the tree $w(H) \leq k$.

- Optimization version: Minimize k to find the minimum cost tree that connects all the terminals.

17. 3-Dimensional matching

    - Decision version: Given a 3-dimensional matching (3DM) problem denotes as $\{X, Y, Z, T, k\}$, where set X, Y and Z are three finite sets with no sharing elements, and set $T = X \times Y \times Z$, that is, $T = \{t_1, \ldots, t_s\}$, $t_i = \langle x_i, y_i, z_i \rangle$ where $x_i \in X$, $y_i \in Y$ and $z_i \in Z$. A 3-dimensional matching M, $M \subseteq T$, holds the property that for $\forall t_i \neq t_j$, $t_i \in M$, $t_j \in M$, and $t_i = \langle x_i, y_i, z_i \rangle$, $t_j = \langle x_j, y_j, z_j \rangle$, such that $x_i \neq x_j$, $y_i \neq y_j$, and $z_i \neq z_j$. If the size of the given 3DM is larger or equal to k?
    - Optimization version: Maximize the number of elements in M to find the biggest 3-dimensional matching set M.

18. Knapsack

    - Decision version: Given an integer k, a group of n items each has value: $\{v_1, \ldots, v_n\}$, and weight $\{w_1, \ldots, w_n\}$. Now by picking items into a knapsack, which has a limit to the weight of the picked items W, if the total value of picked items is greater than or equal to k.
    - Optimization version: What is the maximum value the knapsack can carry?

19. Job sequencing

    - Decision version: A description of job-shop scheduling problems: given two finite sets $M = \{M_1, \ldots, M_m\}$ (for machines) and $J = \{J_1, \ldots, J_n\}$ (for jobs), where there are m machines in the processing system, and n jobs are required to be processed. For each J_i, there is a list $L_i = [(M_{ord_1}, t_{ord_1}), (M_{ord_2}, t_{ord_2}), \ldots, (M_{ord_m}, t_{ord_m})]$ assigned to it to describe in what order J_i should be completed on different machines including its corresponding time, and $L = [L_1, \ldots, L_n]$. Every job here needs to go through all machines to be processed. Besides, each job must be processed in the order of the process. The task of scheduling is to arrange the processing, scheduling and sorting of all jobs, now given an integer k, if the time cost by the scheduled sequence of jobs is less than or equal to k.
    - Optimization version: How to arrange the order of jobs on each machine so that the total time spent is the least.

20. Partition: Given an instance of partition problem S, where S is a multi-set of positive integers. If the set S can be partitioned into two sets S_1 and S_2, and the sum of all

the elements in S_1 equals the sum of all the elements in set S_2.

21. Max cut

- Decision version: Given a graph $G = E, V$ and an integer k, where $V = \langle u_1, \ldots, u_n \rangle$ and $E = \langle e_1, \ldots, e_m \rangle$. If there exists a subset of vertex S_1 such that the number of edges of connecting set S_1 and its complementary subset S_2 is larger than or equal to k. $S_1 \cap S_2$ $\emptyset$ and $S_1 \cup S_2 = S$.
- Optimization version: Find the subset S_1 which makes the number of edges connecting it and S_2 achieve its maximum.

Chapter 2

Problem Solving

The diverse forms of problems within the NP class often permit the utilization of well-established strategies to address specific problems. For example, a boolean satisfiability problem can be solved by using an SAT solver. However, a notable gap persists in finding a unified solution template for problems in NP. At this stage, the reduction process becomes especially crucial in identifying inherent connections among problems within the same computational complexity class, since it helps identify the hardness of solving a problem and makes it possible to obtain the solution by solving other problems.

The core focus of this thesis revolves around addressing problems in NP in a universal, efficient, and convenient manner. This thesis proposes an optimization service grounded in boolean circuit satisfiability, designed to effectively address problems in the NP complexity class while providing sound and logical solutions. In this chapter, the problems to be tackled are provided and well examined first, along with the most common ways of solving problems in NP class nowadays: by using a CNF Satisfiability (SAT) solver. The solution concepts underlying the service proposed in this thesis are introduced next. Furthermore, drawing upon the service's approach to solving problems in NP, this chapter illustrates the application of the corresponding solving methods to Karp's 21 NP-Complete problems, ultimately offering practical resolutions for each.

2.1 Research Objectives

2.1.1 Problem examples

The research and development purpose of this service is to help people solve common problems in NP more conveniently. In the first chapter, the well-known 21 Karp's NP-C problems are well elucidated, encompassing a broad spectrum of applications and forming the basis for numerous intricate problems. For instance, the circuit satisfiability problem serves as a crucial component in computer science, facilitating the verification of logical reasoning system consistency. Similarly, the job sequencing problem frequently arises in production planning to optimize job and task scheduling sequences. Furthermore, the clique cover (partition into cliques) problem in an undirected graph, holds significant value in social network analysis as it aids in identifying community structures and social circles.

Apart from the aforementioned 21 problems in NP-Complete class, there are many other problems composed of naive problems in NP that are also the objects that this service intends to solve. Below are two illustrative questions:

1. A problem that may arise in distributed cloud resource management services is the allocation of supply resources to meet the demands of a distributed cloud environment. This issue involves determining how resources should be allocated to fulfill the requirements of a particular resource. To well present this problem, two types of resources are represented separately in undirected graphs.

    - **Demand Graph**: demonstrates the arrangement/architecture of the intended task and the resources it occupies, which is a set of sub-graphs consisting of nodes and edges.
    - **Supply Graph**: demonstrates the arrangement/architecture of the resources that is provided and can be used, which is a set of sub-graphs consisting of nodes and edges.

    Both the demand graph and supply graph have node attributes and edge attributes, which is the way to illustrate resources.

    - **Node Attributes**: node is used for representing hardware resources, and its attributes may include but are not restricted to the number of virtual Central Processing Units (vCPUs), virtual Graphics Processing Units (vGPUs), storage, and other components within compute or storage units (such as bare metal servers or database instances).

- **Edge Attributes**: edge is used for representing resources associated with network connection services, and its attributes may include but are not restricted to the time needed to establish Virtual Private Cloud (VPC) connections, or the cost involved in connecting different clouds.

For both vertices and edges, attributes are always an integer in this case. Finally, the problem requires a way of placement from the demand graph to the supply graph. To be more specific, "placement" means to allocate nodes/edges from the demand graph to nodes/edges of the supply graph, which implies that the resources a supply node/edges provide are capable of implementing the task from demand nodes/edges that are placed into it. Based on common sense, there are several rules for the placement:

- Demand nodes are to be placed on supply nodes only.
- A demand edge is placed on zero or one supply edge. Assume a vertex has an edge to itself with infinite value for all attributes, on which a demand edge may be placed.
- For a specific supply node/edge, it should be satisfied that for every single attribute it has, the sum of that specific attribute values from all the corresponding demand nodes/edges placed into it should be less than or equal to the single attribute value from that supply node/edge.

Besides, customized anti-affinity constraints can also be added as rules for the placements, such as certain two nodes in the demand graph cannot be placed in a same supply node, or certain two demand nodes have to be placed together in a node in the supply graph.

2. Another strong example relates to scheduling Virtual Machines (VMs) in server to prevent information leakage in side channels [14]. To enhance service efficiency and optimize resource utilization in cloud computing, it is typical for multiple clients to share a single server running multiple VMs. However, this approach exposes a vulnerability as attackers can exploit the lack of complete isolation between different VMs at the physical layer to execute sub-channel attacks, leading to the unauthorized disclosure of sensitive information from victims through these sub-channels. This problem involves three main subjects: epoch, virtual machine and server.

   - **epoch**: One discrete period of time.
   - **VM**: A VM is a unit of computation or execution used for processing client tasks. A client may possess several VMs and the workload on VMs may last several epochs.

- **Server**: A physical machine equipped with computational capacity that is responsible for running a certain number of VMs.

The placement of a VM on a server is considered valid when the server has sufficient capacity to run all the VMs it hosts at every epoch, which is similar to the problem introduced in the first example. Besides, this problem becomes more intricate as it requires addressing the issue of minimizing mutual sub-channel information leakage between different clients by changing the VMs running in each server at different epochs. Additionally, this thesis illustrates several scenarios for information leakage as well [14]:

- **Replication**: This situation arises when a malicious VM is co-located with multiple VMs belonging to the victim. Consequently, sensitive information runs the risk of being duplicated, as the malicious VM can intercept and extract various informative data fragments.
- **Collusion**: This scenario occurs when the victim's VMs are confronted by multiple attackers' VMs, all of which can collude to extract a greater amount of valuable information within a fixed epoch. For instance, each attacker's VM can be assigned the task of intercepting different pieces of information, thereby increasing the overall effectiveness of the data extraction process.

By obeying the rules for calculating the potential leaked data, it presents a significant challenge for cloud service providers to devise a strategy for distributing clients' (also possibly to be a potential attacker) VMs across different servers at different time periods.

2.1.2 Traditional Solutions

SAT Solver

Although NP-Complete problems cannot be solved within polynomial time of their inputs, there are ongoing efforts to develop programs that can efficiently tackle such problems. Among these, SAT solvers have emerged as particularly notable, due to their success in solving the boolean satisfiability problem. A SAT Solver takes a boolean formula in CNF form as input, and returns the answer if the formula is satisfiable or not, if so, it will provide one feasible solution as well. SAT Solver has been widely used in many fields since its release, including artificial intelligence, Integrated Circuit (IC) design, software verification, electronic hardware design etc.

SAT solvers typically necessitate a CNF input, commonly stored in a file formatted according to the Discrete Mathematics and Theoretical Computer Science (DIMACS) standard. The DIMACS format serves as a widely adopted standard for representing CNF problems, making it the prevalent input format for SAT solvers, and enables SAT solvers to efficiently process and solve boolean satisfiability problems. DIMACS files typically consist of two sections: the header and clause lines. The problem line, starting with "p", specifies the parameters of the CNF problem. It includes the problem type, the number of literals, and the number of clauses, all separated by spaces. The clause lines contain the contents of individual CNF clauses, with each number separated by spaces. Each number represents a literal, where positive numbers denote normal literals and negative numbers represent the negation of the corresponding literal. The clause lines are terminated by a 0 to indicate the end of each clause. Below is a simple example of CNF formula $(1 \vee \neg 2 \vee 3) \wedge (2 \vee 3) \wedge (1 \vee \neg 3) \wedge 2$ in DIMACS format.

Listing 2.1: CNF formula in DIMACS format

```
p cnf 3 4
1 -2 3 0
2 3 0
1 -3 0
2 0
```

Using a search algorithm, SAT Solvers usually determine the range of potential truth assignments for the variables. The process of exploring the assignment for input variables is progressive. For every search, the algorithm assigns a variable as a value and performs satisfiability reasoning, then update and analyzes the formula in the next search. Whenever the formula cannot be satisfied in a certain phase, the search function goes back to the last assignment of a variable and modify its value, iteratively propagating the assignment to verify if the input formula is satisfiable or not.

It is of high necessity for a SAT Solver to adopt an efficient search algorithm; otherwise, it will be indistinguishable from a normal program using brute force. Thus, to increase the effectiveness of the search, multiple procedures have been proposed; such as clause elimination procedure to simplify CNF [13], Scaling And Probabilistic Smoothing (SAPS) algorithm to make dynamic local search more efficient [12], variable elimination with subsumption and self-subsuming resolution in preprocessing to decrease the runtime of SAT solvers[10] etc.

There is a wide selection of SAT solvers available, among which MiniSat stands out as one of the most renowned. Initially presented in 2003 [21], MiniSat employs conflict-driven learning, watched literals technique, and various other algorithms to significantly

accelerate computation time while maintaining exceptional performance. It has become the cornerstone upon which many SAT solvers are built, owing to its ability to effectively incorporate clause learning techniques. Furthermore, the Maple series of SAT solvers, enhanced with machine learning-based heuristics developed by Vijay Ganesh et al., have made notable advancements in SAT solving [17]. In particular, MapleSAT has demonstrated improved performance by employing the Learning Rate Branching (LRB) heuristic technique which is built on a Multi-Armed Bandit (MAB) algorithm called Exponential Recency Weighted Average (ERWA), and diverging from the conventional Variable State Independent Decaying Sum (VSIDS) branching heuristic.

The advent of SAT solvers has opened up new possibilities for addressing prevalent problems in the NP domain. This entails reducing the target problem into a boolean satisfiability problem and subsequently employing an SAT solver to solve it. By inverting the solution provided by the SAT solver, one can obtain the desired answer for the original problem. This approach has proven to be an effective strategy for tackling problems in NP using the capabilities offered by SAT solvers. However, the process of reducing the original problem to a boolean satisfiability problem often proves to be intricate and burdensome. This is due to the inherent limitations of boolean satisfiability problems, which only permit three logic operators, whereas many other problems may involve more complex mathematical expressions. For instance, consider the exact cover problem, which relies on graph theory. The task of transforming a graph-based problem into a boolean logic-based problem for its solution is considerably complicated and convoluted. This process entails not only encapsulating the fundamental mathematical numbers and graph characteristics into boolean logic formulas (CNF form) but also meticulously representing each aspect of the basic logic in the exact cover problem through these formulas. It requires careful consideration of how to store and encode the essential components of the problem using boolean logic expressions, and the scale of the final CNF form is expected to be vast, which poses challenges in both the reduction process solving and testing.

Other Solvers

Apart from the SAT solvers that can solve the boolean satisfiability problem, solvers that can directly solve the graph colouring problem have also emerged. FastColor, a graph coloring problem solver released in 2017, introduces a novel reduction rule rooted in the degree bounded independent set concept [18]. As a result, its algorithm outperforms the approach proposed by Rossi in 2016 [20], delivering significantly enhanced performance. Concorde, a highly efficient solver for the Traveling Salesman Problem (TSP) developed by David L. et al. [5], excels at providing rapid solutions for large-scale TSP instances.

But still, it should be noted that the aforementioned solvers are specifically designed for a limited set of problems (graph colouring problem and TSP), which may restrict the solver's applicability. Their practicality significantly diminishes when confronted with problems that cannot be easily reduced to these domains, particularly those unrelated to graphs. Hence, this thesis focuses on the broader objective of utilizing solvers to address general problems in NP, transcending the limitations of several specific problems. Besides, it explores methodologies and approaches that enable the solver's applicability to a wide range of problems, aiming to enhance its versatility and effectiveness in solving various NP-complete problems.

2.2 Solution Ideas

This thesis presents professional solutions for commonly encountered problems in NP class. The approach first involves reducing a problem in NP to a circuit satisfiability problem in a non-traditional sense. The featured circuit not only encompasses the three very basic logical operators: AND, OR, and NOT gates, but also includes sub-circuits capable of performing arithmetic operations based on numerical numbers. Specifically, below are seven enhanced circuits that can be constructed in the service, and work as nested sub-circuits:

1. **AND circuit**: A big AND circuit which contains x input wires and perform AND logic.

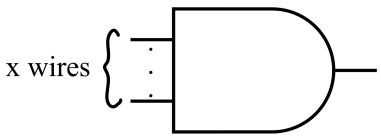


Figure 2.1: The AND circuit.

2. **OR circuit**: A big OR circuit which contains x input wires and perform OR logic.

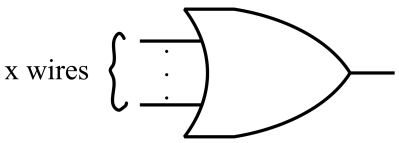


Figure 2.2: The OR circuit.

3. **Bit-Not circuit**: A circuit that takes 1-bit input, and has 1-bit output, which is the negation of the input.

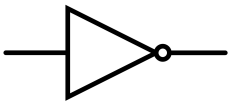


Figure 2.3: The Bit-Not circuit.

4. **IntegerAsCircuit circuit**: This circuit encodes a non-negative integer i and returns it in binary as a boolean circuit.

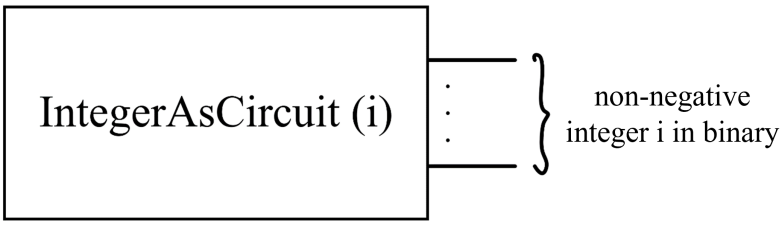


Figure 2.4: The IntegerAsCircuit circuit.

5. **Sum circuit**: The circuit has a list of wires representing n binary numbers as input and it outputs the number of wires equal to the sum of those binary values.

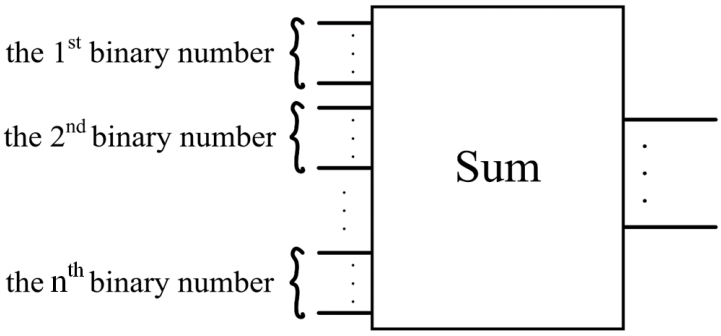


Figure 2.5: The Sum circuit.

6. **LessEquals circuit**: A circuit with $x + y$ input wires. It checks whether the binary number represented by the first x bits is less than or equals the binary number

represented by the next y bits. Its output wire will be assigned a boolean value of 1 if the first binary number in x wires is less than or equal to the second binary number in y wires, otherwise 0.

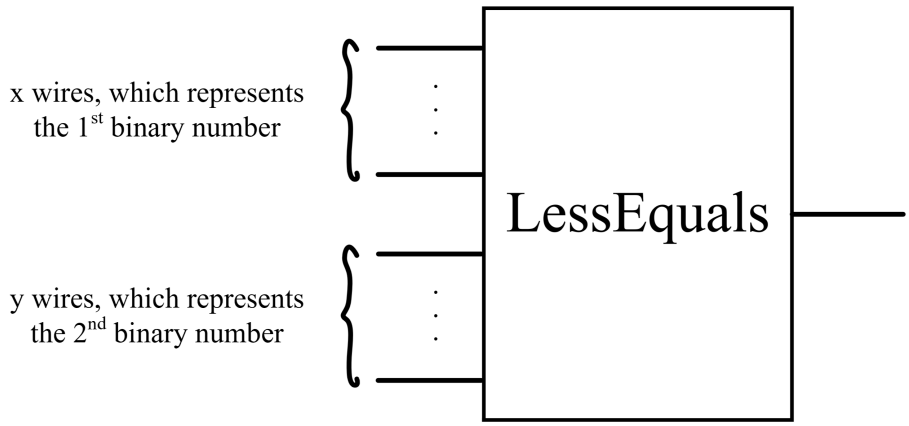


Figure 2.6: The LessEquals circuit.

7. **Choose circuit**: A circuit takes $1 + y$ input wires and uses the first input wire as the chooser-bit to choose the rest of y bits. If the chooser-bit is assigned a boolean value of 1, the output y wires will be the same as the y input wires; otherwise, they will all be set to a boolean value of 0 which represents the y input wires are not been selected.

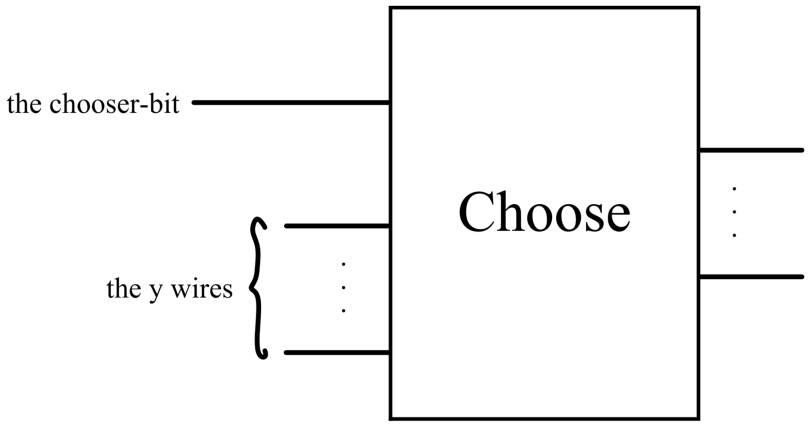


Figure 2.7: The Choose circuit.

All of the aforementioned circuits are fundamentally composed of AND, OR, and NOT gates. However, their ability to perform fundamental mathematical operations significantly simplifies the challenges posed by other problems in NP when they are reduced to the Circuit-SAT problem.

Then, to solve the circuit that probably contains the above extra four sub-circuits, it should be encoded into a JSON file that comprehensively includes all its information. The server proposed in this thesis takes that JSON file as input, effectively automates the circuit-building process, and determines its feasible solution. In chapter 3, a comprehensive exploration is provided regarding the library employed for constructing and solving circuits with these new features.

Before delving into the implementation details of the service, at the end of this chapter, the reduction process from all Karp's 21 NP-C Problems to Circuit-SAT Problems is presented as an example manual to provide references for other problems in NP.

2.3 Ways of Reducing Karp's 21 NP-C Problems to Circuit-SAT Problems

Below is the reduction process from all of Karp's 21 NP-complete problems to the Circuit-SAT problem. Karp's 21 NP-complete problems come in various forms and cover a wide range of domains. Clients can use this as a reference to reduce their own problems in NP to a Circuit-SAT problem. The ideas for all the reductions presented in Chapter 2.3 of this thesis were conceived independently by the author.

Boolean satisfiability problem

The boolean satisfiability problem (also known as SAT) is in the NP-complete class, currently, there is no deterministic algorithm that can solve this problem in polynomial time in its input. The reduction from SAT problem to Circuit-SAT problem is achieved by creating input wires for each unique literal, and constructing AND, OR and Bit-Not sub-circuits according to logical operators in the boolean formula.

0–1 integer programming

A 0-1 integer programming problem refers to a list of unknown variables and a few inequalities constraints. Its goal is to find the extreme value for a polynomial of unknowns. A canonical form of it is:

- maximize $\mathbf{c}^T\mathbf{x}$
- subject to $\mathbf{Ax} \leq \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$ and $\mathbf{x} \in \mathbb{Z}^n$

where $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$ are vectors and $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a matrix.

The circuit of the reduction from 0–1 integer programming problems to Circuit-SAT problems has $k * dim_{\mathbf{R}}(\mathbf{x})$ input wires, where k is the upper limit of the number of bits the unknown can occupy when represented in binary. Each unknown is represented by k input wires in binary for future calculation.

1. $\mathbf{Ax} \leq \mathbf{b}$ can be expressed as k independent equations and for every inequality, the inequality sign can be reified into a LessEquals sub-circuit. The right-hand side number b of the equation is been expressed as an IntegerAsCircuit and its output is input to the LessEquals sub-circuit. As for the left-hand side, each unknown x_i is multiplied by an integer a_i (since IntegerAsCircuit could only generate an integer). This is accomplished by using a Sum sub-circuit, which takes a_i elements, and then the k wires of x_i are input a_i times into the Sum sub-circuit. Finally, a Sum sub-circuit is again used to sum up all the output results of each unknown in the vector, its output is another part of the inputs of the LessEquals sub-circuit.
2. To determine the maximum of $\mathbf{c}^T\mathbf{x}$, the reduction circuit requires clients to use external code to control the input of an IntegerAsCircuit sub-circuit and let it generate a target value s. According to the feedback on the relationship between the previous target s and $\mathbf{c}^T\mathbf{x}$, the maximum can be gradually approached by continuously adjusting the value of s. First, we can convert $\mathbf{c}^T\mathbf{x}$ to a sub-circuit, the step is similar to the step above when we construct for the left-hand side of a single inequality. The output of the sub-circuit is part of the input for a LessEquals sub-circuit, and the other part of the input is the output of an IntegerAsCircuit for the target s, before finding the maximum value, the value of s is determined by the code used by clients. According to each previous return, clients code should be able to adjust s. For example, if $\mathbf{c}^T\mathbf{x}$ is less than or equal to the previous target s, then the code should increase s this turn, otherwise make it smaller.
3. An AND sub-circuit is used in the end to ensure all the constraints above are guaranteed.

Clique

A reduction from clique problem to circuit-sat is as follows. Given an instance of clique problem, $\langle G = \langle V, E\rangle, k\rangle$, if there exists a clique of size k for undirected graph G? A clique V' is a complete subgraph of G.

Let $V = \{u_1, \ldots, u_n\}$, for every vertex, an input wire is introduced and it is set to 1 if and only if this vertex is chosen to be included in the k-sized clique.

The constraints for the clique problem are encoded as below:

1. For two disconnected vertices $\{u_i, u_j\}$, their corresponding input wires are both input to an AND sub-circuit and then the output of the AND sub-circuit is then input to a Bit-Not sub-circuit. This is to prevent any disconnected vertices from being selected into the clique at the same time.

2. Input all the input wires into a Sum sub-circuit. Then, two LessEquals sub-circuits take the output wires of the Sum sub-circuit as the first and second groups of input, respectively. The other group of input wires for these two LessEquals sub-circuits are the output of IntegerAsCircuit for k. This is to check if the number of vertices in the final clique is exactly k.

3. An AND sub-circuit is constructed, with the output wires of the above sub-circuits serving as its inputs. This construction ensures that all the above constraints are satisfied, and its output wire becomes the output wire of the whole circuit.

Set packing

Suppose the input of an instance of set packing is $\langle\{S_1, \ldots, S_n\}, X, k\rangle$, where collection $S = \{S_1, \ldots, S_n\}$ contains subsets of set X. If there exists a collection C whose size is k or more, such that all sets in C are pairwise disjoint?

The circuit we construct has n input wires, wire i is set to 1 if and only if S_i has been chosen into collection C.

The reduction is as follows:

1. For every pair of subsets $\langle S_k, S_j\rangle$, where $S_k \cap S_j \neq \emptyset$, input the wires representing both subsets into an AND sub-circuit. Then, input the output of the AND sub-circuit to an OR sub-circuit.

2. An AND sub-circuit is built in the end collecting all the output of OR sub-circuits above, and its output is the output of the final whole circuit.

Vertex cover

A reduction from vertex cover to circuit-sat is as follows. Assume we are given an instance of vertex cover, $\langle G = \langle V, E\rangle, k\rangle$, if there exists a vertex cover of size at most k for undirected graph G? A vertex cover V' is a subset of V such that for all the edges $uv \in G, u \in V' \vee v \in V'$.

Assume $V = \{u_1, \ldots, u_n\}$. Our output circuit C has n input wires $\{i_1, \ldots, i_n\}$. An input wire is set to 1 if and only if u_i is chosen to be in the vertex cover.

We encode the constraints as follows:

1. For every edge $\langle u_i, u_j\rangle \in E$, at least one of them has to be in the vertex cover. So — for every edge $\langle u_i, u_j\rangle$, an OR with i_i, i_j as its input is built.
2. The integer sum $i_1 + \ldots + i_n \leq k$. So a Sum sub-circuit which inputs are all the input wires to the full circuit $i_1, \ldots, i_n$, an IntegerAsCircuit sub-circuit for k are constructed; and finally, a LessEquals sub-circuit is used to compare the output of the Sum and the IntegerAsCircuit.
3. Finally, an AND sub-circuit takes all the output wires of the above two constraints as inputs is built, and the output wire of this AND sub-circuit is the output wire of the full circuit.

Set covering

Given an instance of a set covering problem, $\langle \{S_1, \ldots, S_n\}, U, k\rangle$, where $S = \{S_1, \ldots, S_n\}$ is a collection of subsets of union set U. If there exists a k-size (or less) set cover C, which consists of subsets in S and the union is U? That is, if $C = \{S_{c_j}, \ldots, S_{c_k}\}$, then for all $e \in U$, $\exists S_{c_m} \in C$ such that $e \in S_{c_m}$.

The circuit constructed according to set covering has n input wires, $\{i_1, \ldots, i_n\}$, and wire i_j is set to 1 if and only if the j^{th} subset in collection S is selected into the set cover.

The encoding of constraints for this question is as follows:

1. For each subset $S_i \in S$, for every missing element not included in S_i, use an OR sub-circuit to receive the group of input wires whose corresponding subset contains the missing element. An AND sub-circuit is then built to receive the wire for the subset S_i and all the output wires of the previous OR sub-circuits. (If there is only one subset containing the missing element, then its wire can directly input to the AND sub-circuit of S_i).

2. Input all the wires to a Sum sub-circuit, and a LessEquals sub-circuit is built to compare the output of the Sum sub-circuit and the output of IntegerAsCircuit for k, to check if the size of the set cover exceeds k.

3. Finally an AND sub-circuit is built, which inputs are all the output wires above and its output is the output of the final whole circuit.

Feedback node set

Given an instance of a feedback node set problem: $\langle G = \{V, E\}, k\rangle$, if there exist a k-size FVS such that if all the vertices in FVS are removed, the whole graph will have no cycles.

Assume $V = \{u_1, \ldots, u_n\}$, the circuit for feedback node set problem has n input wires, $\{i_1, \ldots, i_n\}$, and a wire i_j is set to 1 if and only if the j^{th} vertex is chosen into the FVS. The reduction is shown below:

1. Clients should first determine all the cycles in graph G, and record every relevant index of vertices included. Then for all the vertices in every cycle, input their wires to an OR sub-circuit to check if at least one of them is selected to be in FVS.

2. Input all the wires to a Sum sub-circuit. The Sum sub-circuit's output, along with the output of an IntegerAsCircuit for k, are both input to a LessEquals sub-circuit to check if the number of the selected vertices is less than or equal to k.

3. Finally uses an AND sub-circuit to receive all the output wires above and its output is the output of the whole circuit.

Feedback arc set

An instance of a feedback arc set problem is $\langle G = \{V, E\}, k\rangle$, $E = \{e_1, \ldots, e_n\}$, if there is a k-size feedback arc with a set of edges such that it would not contain any cycles if all of its edges are removed.

Similar to above, instead of creating wires for each vertex, now create n input wires for each edge this time. A wire is set to 1 if and only if it is selected to feedback arc set. The reduction to Circuit-SAT problem is shown below:

1. Number all edges from 1 to n, then clients need to initially identify all the cycles within graph G and record all relevant indices of the included edges. For each cycle, in order to determine if at least one of the edges in it has been chosen to be in the feedback arc set, input the wires of all edges in the circuit to an OR sub-circuit.

2. Similarly, a Sum (receiving all the input wires) sub-circuit, a LessEquals sub-circuit and an IntegerAsCircuit for k are used to check if the number of the selected edges is smaller than k.

3. An AND sub-circuit whose inputs are all the output wires above is then used in the end to check if all the above constraints are satisfied.

Directed Hamiltonian path

Directed Hamiltonian path problem is to find out if there exists a path in the directed graph $G = \{V, E\}$ which makes it possible for visiting each vertex exactly once.

Assume $G = \{V, E\}$, $E = \{e_1, \ldots, e_n\}$ and $V = \{u_1, \ldots, u_m\}$, the reduced circuit possess n input wires each represent if an edge, an input wire i_j is set to 1 if and only if its corresponding edge e_j is selected into the undirected Hamiltonian path.

The reduction from directed Hamiltonian path problem to Circuit-SAT problem is shown below:

1. To make sure no vertices are visited twice or more, for every edge e_i, where $e_i = \langle u_a, u_b \rangle$, u_a is the source, u_b is the destination, all the wires representing edges whose destination is u_a is connected to a bit-NOT sub-circuit and pass to an AND sub-circuit.

2. To ensure all the vertices are visited, for every vertex, input all the wires whose corresponding edges connect to it to an OR sub-circuit.

3. Pass all the inputs to a SUM sub-circuit to calculate the number of selected edges. To verify the fact that the number of edges in Hamiltonian path in G equals the number of vertices minus 1, the output of the SUM is input to a LessEquals sub-circuit and the other part of its input is an IntegerAsCircuit for $m - 1$.

4. In the end, all the output wires above should be input to an AND sub-circuit to guarantee all the constraints are satisfied.

Undirected Hamiltonian path

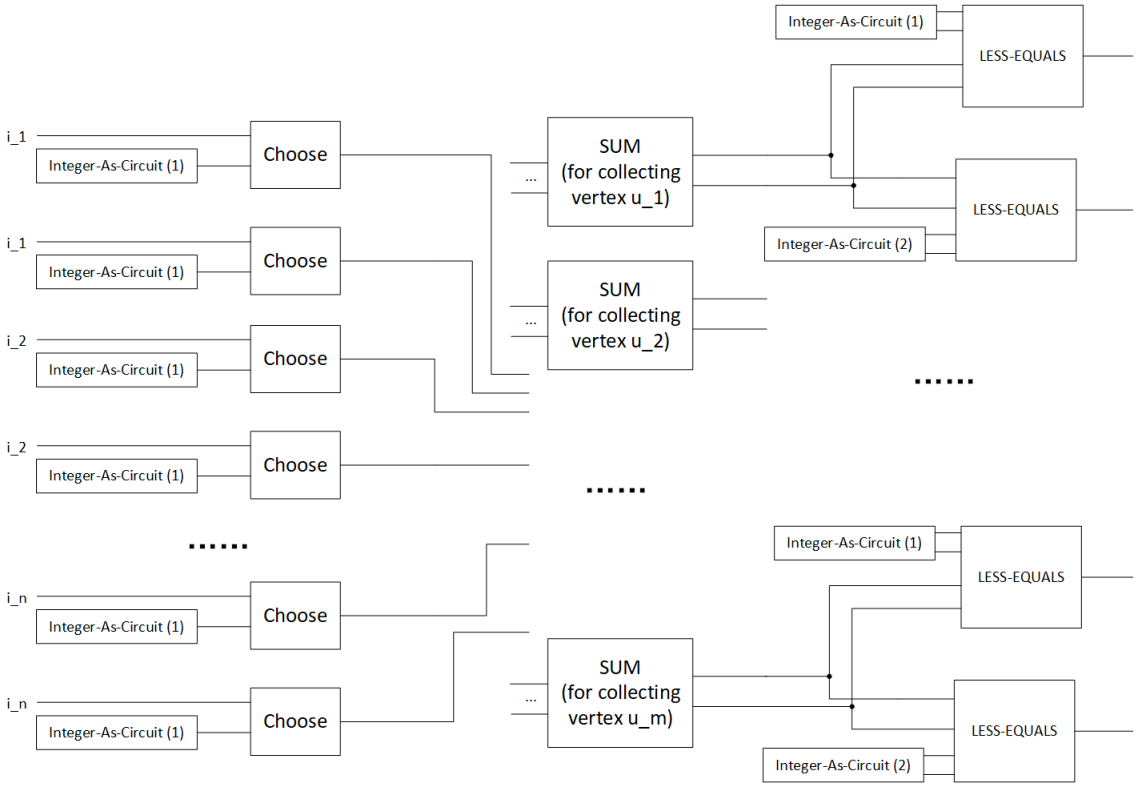
Similar to the directed Hamiltonian path problem, the undirected Hamiltonian path problem is asking if an undirected path in graph $G = \{V, E\}$ exists, such that each vertex will be visited precisely only once.

According to the given graph $G = \{V, E\}$, $E = \{e_1, \ldots, e_n\}$ and $V = \{u_1, \ldots, u_m\}$, the circuit has n input wires, $\{i_1, \ldots, i_n\}$, each represent an edge, and an wire i_j is set to 1 if and only if edge e_j is selected into the undirected Hamiltonian path.

The reduction from undirected Hamiltonian path problem to Circuit-SAT problem is shown below:

1. Input all the input wires to a LessEquals sub-circuit to check if the number of the selected edges in the path is exactly $m - 1$, m is the number of the vertices. Thus, the other part of the input is the output of an IntegerAsCircuit sub-circuit for m.

2. For each input wire, input it to two Choose sub-circuits both as the chooser-bit (first bit), and the rest of the input wires of the two Choose sub-circuits are the output of two individual IntegerAsCircuit (for 1) sub-circuits. Each of the IntegerAsCircuit sub-circuit represents a vertex to the edge connected, and in the entire whole circuit, all Choose sub-circuits that choose to represent the same vertex need to feed their outputs into a Sum sub-circuit. Later, the output of the Sum sub-circuit is the input of two LessEquals sub-circuits to check if the output is less than or equal to 2 and larger than or equal to 1.



That is, if an edge is selected in an undirected Hamiltonian path, the two vertices it connected will also be selected, finally by checking how many times each vertex has been counted (x), making sure $1 \leq x \leq 2$, then it is guaranteed that every vertex will only be visited once (only the start and end vertex will be counted once and others will be counted twice).

3. In the end, an AND sub-circuit is constructed whose input is all the output wires above, this is to make sure all the above constraints are satisfied.

Satisfiability with at most 3 literals per clause

3-SAT problem is also NP-complete, which formula contains 3 literals in every clause.

Similar to the original SAT problem, when solving 3-SAT problems by using the circuit solver, the input should be in accordance with the input of the formula, and each unknown variable is assigned to an input wire. The construction of the circuit sub-circuits should be aligned with the connection order between the operators in the formula.

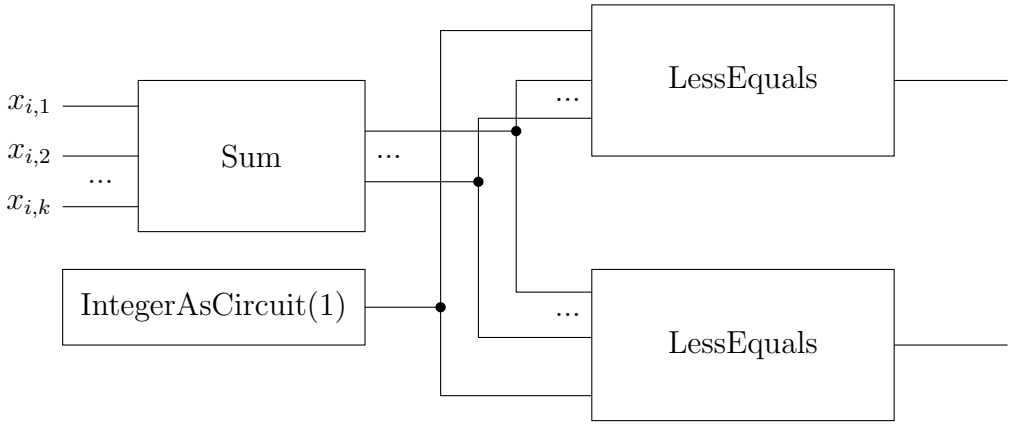
Graph Coloring Problem (Chromatic number)

Assume our input is an instance of graph colouring problem $\langle G = \langle V, E\rangle, k\rangle$, and ask if there is a way/function $c(\cdot)$ of labelling k colours to all vertices such that no two vertices connecting by an edge share the same colour, $c(u) \neq c(v)$ for $\forall(u, v) \in E$.

Let $V = \{u_1, \ldots, u_n\}$, for every vertex u_m, construct k input wires, $\{i_{m,1}, \ldots, i_{m,k}\}$. A wire $i_{m,j}$ is set to 1 if and only if u_m is set to the j^{th} color.

We encode the constraints as follows:

1. For every k input wire constructed according to a vertex, use a Sum sub-circuit to calculate their sum value, checking if it is equal to 1 since each vertex should be and only be assigned to one type of colour. Input both the output of the Sum sub-circuit and the output of an IntegerAsCircuit for 1 to two LessEquals sub-circuits to check if they are the same.



2. For every edge $\langle u, v\rangle$, check input wires $\{i_{u,j}, i_{v,j}\}$ for all $j \in k$ by inputting them to an AND sub-circuit and concatenate the AND with an OR sub-circuit, to make sure not assigning vertex u and v to the same colour.

3. An AND sub-circuit of the output wires of the above sub-circuits is constructed in the end to make sure all the above constraints are satisfied, whose output wire is the output wire of the full circuit.

Clique cover

Assume the input of clique cover is $\langle G = \{V, E\}, k\rangle$, if there exists a way of partitioning all the vertices V in the graph G into k cliques. Suppose $V = \{u_1, \ldots, u_n\}$, the circuit we built has $n * k$ input wires. Wire $i_{j,a}$, $j \in \{1, \ldots, n\}$ and $a \in \{1, \ldots, k\}$, is set to 1 if and only if decide to categorize the j^{th} vertex into the a^{th} clique.

The reduction from the clique problem to Circuit-SAT problem is as follows:

1. For each vertex, input its k wires into a Sum sub-circuit. Input its output and the output of an IntegerAsCircuit for 1 to two LessEquals sub-circuits to check if they are the same. This is to guarantee the constraint that one vertex should only be grouped into one clique.

2. For every two vertices which are not connected, they can not be put into the same clique. Thus, for $\forall(u, v)$ and $(u, v) \notin E$, input k pairs of wires $i_{u,a}$ and $i_{v,a}$, $a \in \{1, \ldots, k\}$, into an AND sub-circuit and then concatenate its output to a Bit-Not sub-circuit.

3. Inputs all the output wires above to an AND sub-circuit in the end, this is to make sure all the constraints above are satisfied at the same time.

Exact cover

Suppose our input is $\langle\{S_1, \ldots, S_n\}, X\rangle$, where collection set $S = \{S_1, \ldots, S_n\}$ contains subsets of set X. If there exists a subcollection S^* of S whose size is k, such that every item in U is contained and only contained once in S^*. Our full circuit has n input wires $i_1, \ldots, i_n$ where the wire i_j is set to 1 if and only if S_j is in the exact cover.

We encode the constraints as follows:

1. For each pair S_j, S_k that is chosen to be in our solution, we ensure that their intersection is empty.

   That is, for each pair of input wires i_j, i_k, which have the property $S_j \cap S_k \neq \emptyset$, we create an AND sub-circuit with i_j and i_k as the inputs and then place a Bit-Not sub-circuit directly after its output.

2. For each member $x \in X$, an OR sub-circuit is built, whose inputs $i_{k_1}, \ldots, i_{k_q}$ are all the input wires to the full circuit which corresponding subset contains element x, where $x \in \{S_{k_1} \cap \ldots \cap S_{k_q}\}$.

3. An AND sub-circuit of the output wires above is constructed in the end to make sure all the above constraints are satisfied, whose output wire is the output wire of the full circuit.

Hitting set / Set cover problem

Given a set $S = \{S_1, \ldots, S_n\}$, a universal set U, and an integer k, where S is a collection of subsets of U, and $S_1 \cup \ldots \cup S_n = U$. If there exists a k-size hitting set H, which is a subset of U, such that for every collection $S_i \in S$, the number of items shared by H and S_i is only 1, $|H \cap S| = 1$.

Steiner tree

Given an instance of a Steiner tree problem in an undirected graph, $\{G = \langle E, V \rangle, T, k\}$, $V = \{u_1, \ldots, u_n\}$ and $E = \{e_1, \ldots, e_m\}$, with non-negative edges $w(e_i) \geq 0$. The terminal vertex set T is a subset of vertices of set V, and $k \in \mathbb{Z}$. Is there a tree $H = \langle E^{'}, V^{'} \rangle$, which is a subgraph of G, such that $T \subseteq V^{'}$ and the weight of the tree $w(H) \leq k$.

The circuit has m inputs for each edge, and an input wire is set to 1 if and only if the edge is selected into the tree. The reduction from this problem to Circuit-SAT problem is shown below:

1. To check that all the vertices in the terminal set are included, for every vertex $u_k \in T$, all the edges connected to it are passed to an OR sub-circuit. This is used to ensure that at least one of the edges is chosen, and the vertex is in the final output tree H.

2. To ensure all the vertices in H are connected, for each pair of vertices $\{u_i, u_j\}$ ($u_i \in T$ and $u_j \in T$), clients should record all the possible pass connecting $u_i \in T$ and $u_j \in T$. For every possible path, use an AND sub-circuit to receive all the input of edges included. Finally, the output of the AND for each path should then be input to a SUM sub-circuit, this is to calculate how many paths are there between $u_i \in T$ and $u_j \in T$. The output of the SUM will be input to a LessEquals sub-circuit, the other part of the LessEquals input is the output of an IntegerAsCircuit for value 1. This is to make sure there exists only one path between two vertices in the tree H and prevent cycles.

3. Every input wire is passed to a Choose sub-circuit as the chooser bit, and the other input is the output of an IntegerAsCircuit for the weight of that edge. The output

of all the Choose sub-circuits is input to a Sum sub-circuit to calculate the sum of the weight of selected edges. Then the output of the Sum sub-circuit and an IntegerAsCircuit for k is compared, and both are input to a LessEquals sub-circuit.

4. An AND sub-circuit is used to take all the sub-circuit output above as input to ensure those constraints are satisfied. Its output is the final output of the entire circuit.

3-dimensional matching

Given a 3-dimensional matching (3DM) problem denotes as $\{X, Y, Z, T, k\}$, where set X, Y and Z are three finite sets with no sharing elements, and set $T = X \times Y \times Z$, that is, $T = \{t_1, \ldots, t_s\}$, $t_i = \langle x_i, y_i, z_i \rangle$ where $x_i \in X$, $y_i \in Y$ and $z_i \in Z$. A 3-dimensional matching M, $M \subseteq T$, holds the property that for $\forall t_i \neq t_j$, $t_i \in M$, $t_j \in M$, and $t_i = \langle x_i, y_i, z_i \rangle$, $t_j = \langle x_j, y_j, z_j \rangle$, such that $x_i \neq x_j$, $y_i \neq y_j$, and $z_i \neq z_j$. If the size of the given 3DM is larger or equal to k?

The circuit for 3DM problem contains s inputs $\{i_1, \ldots, i_s\}$ where s is the size of the set T. A wire i_j is set to 1 if and only if the j^{th} element in T is selected into set M. The reduction from 3DM problem to Circuit-SAT problem is shown below:

1. For each pair of subset $\langle t_i, t_j \rangle$, both t_i and $t_j \in T$, if they contain any common elements, for example, $x_i = x_j$ or $y_i = y_j$ or $z_i = z_j$, then pass the i^{th} and j^{th} input wires into an AND sub-circuit and then the output of the AND sub-circuit should then pass to an OR sub-circuit. This is to make sure any two triples with no empty intersection should not be selected into a 3DM at the same time.

2. A LessEquals sub-circuit is used to compare the number of elements in M with k. One part of its input is the output of an IntegerAsCircuit sub-circuit for k, and another part of the input is the output wires of a Sum sub-circuit. The Sum takes all s input wires into it and figures out how many of them are chosen.

3. Finally to guarantee all the constrains above, an AND sub-circuit is used to accept all the sub-circuits output wires, and its output is the final output of the whole circuit.

Knapsack

Following is the reduction from Knapsack to Circuit-SAT. Assume there are n values each has value: $\{v_1, \ldots, v_n\}$, and the weight of each item is given: $\{w_1, \ldots, x_n\}$. Now by picking

items into a knapsack, which has a limit to the weight of the picked items W, ask what is the maximum value it can carry.

The problem can be solved by using binary search on the final possible values. Every time after deciding the value V, we encode the constraints:

Build a circuit with n input wires, $\{i_1, \ldots, i_n\}$, the wire is set to 1 if and only if this item is selected to be placed in the knapsack.

1. Connect each input wire i_j to a Choose sub-circuit, meanwhile, the other input of the Choose sub-circuit is the output of IntegerAsCircuit for w_j. Then concatenate all Choose outputs into a Sum, later the output wires of the Sum and another IntegerAsCircuit for W are both input to a LessEquals sub-circuits to check if the weight of the selected items exceeds the limit W.

2. Similarly, as for checking the value, connect each input wire i_j to a Choose sub-circuit, meanwhile, the other input of the Choose is the output of IntegerAsCircuit for v_j. Then concatenate all Choose outputs into a Sum, later the output wires of the Sum and another IntegerAsCircuit for V are both input to a LessEquals sub-circuit to check if the expected value V is less than or equal to the sum of the picked items.

3. Finally an AND sub-circuit is built, which inputs are all the output wires of the above constraints and its output is the output of the final whole circuit.

Job sequencing

A description of job-shop scheduling problems: given two finite sets $M = \{M_1, \ldots, M_m\}$ (for machines) and $J = \{J_1, \ldots, J_n\}$ (for jobs), where there are m machines in the processing system, and n jobs are required to be processed. For each J_i, there is a list $L_i = [(M_{ord_1}, t_{ord_1}), (M_{ord_2}, t_{ord_2}), \ldots, (M_{ord_m}, t_{ord_m})]$ assigned to it to describe in what order J_i should be completed on different machines including its corresponding time, and $L = [L_1, \ldots, L_n]$. Every job here needs to go through all machines to be processed. Besides, each job must be processed in the order of the process. The task of scheduling is to arrange the processing, scheduling and sorting of all jobs, so that the performance indicators can be optimized while the constraints are met.

There are four constraints that should be considered:

1. One machine can only process at most one job at a certain time.

2. A job cannot appear on multiple machines at the same time (it can only work on one machine).

3. Each job can only be processed once on one machine.

4. For each job, every process is carried out on a specific machine, and the subsequent process may only begin until the preceding process is finished.

An instance of job-shop scheduling problem is given $\{M, J, L, k\}$, if there exists a way of arranging all the jobs so that the final finish time is within k. The reduction of job sequencing problem to Circuit-SAT problem is shown below.

The circuit holds $m \times n \times t$ inputs, where t is the sum of time for all jobs to complete on every different machine. An input wire $M_k\_J_i\_T_x$ ($k \in [1, \ldots, m]$, $i \in [1, \ldots, n]$ and $x \in [0, \ldots, t]$) is set to 1 if and only if decide to make machine M_k process job J_i in time slot T_x. The worst arrangement is to let jobs from J_1 to J_n be completed in order, and it takes t times. Assume the overall time axis is divided into t segments, here the time slot T_x denotes the x^{th} slot from the jobs start.

1. First, to make sure every machine can only process at most one job at a time slot, so for every machine $k \in [1, \ldots, m]$ and every time slot $x \in [0, \ldots, t]$, a Sum sub-circuit is used to take all the wires of $\{M_k\_J_1\_T_x, M_k\_J_2\_T_x, \ldots, M_k\_J_n\_T_x\}$ as inputs and its output are passed to a LessEquals sub-circuit to check if for machine M_k at time x there is only one or even no job is processing.

2. Second, in order to ensure that the same job cannot appear in more than two job-shops at the same time, for every job $i \in [1, \ldots, n]$ and every time slot $x \in [0, \ldots, t]$, each pair of $\langle M_a\_J_i\_T_x, M_b\_J_i\_T_x \rangle$ ($a \neq b$ and $a, b \in [1, \ldots, m]$) needs an AND sub-circuit which concatenates with an OR to guarantee these two inputs are equal at the same time.

3. Third, to guarantee each job should be processed and only be processed once in each machine, the number of time slots occupied by a job should be continued and equal to the provided duration. Thus, for every machine $k \in [1, \ldots, m]$ and every job $i \in [1, \ldots, n]$, the given required working time r is $L_i[k][1]$; then all input wires $\{M_k\_J_i\_T_1, M_k\_J_i\_T_2, \ldots, M_k\_J_i\_T_t\}$ should pass to a Sum sub-circuit and determine if the total working time duration is r. Besides, for every consecutive r group of input wires in $\{M_k\_J_i\_T_1, M_k\_J_i\_T_2, \ldots, M_k\_J_i\_T_t\}$, input each group of wires into AND sub-circuits, the outputs of all the AND sub-circuits are passed to an OR sub-circuit. This is to make sure a job does not stop during processing.

4. Fourth, to make sure the final decision sequence is in alignment with the given order of each job, for every job $i \in [1, \ldots, n]$, get the sequence of the required processing machines $seq_i = [M_{ord_1}, M_{ord_2}, \ldots, M_{ord_m}]$, from the second machine M_{ord_2}, check if the previous machine has this job occupied for the corresponding required time long. By following the above sub-circuit, once a group of consecutive wires $\{M_k\_J_i\_T_x, M_k\_J_i\_T_{x+1}, \ldots, M_k\_J_i\_T_{x+r}\}$ are set to 1, check if the number of "1's" in the input wires $\{M_{k-1}\_J_i\_T_0, M_{k-1}\_J_i\_T_1, \ldots, M_{k-1}\_J_i\_T_{x-1}\}$ is the same as the required processing time for the previous machine and the same job J_i.

5. Finally, an AND sub-circuit is built to accept all the output wires above, and its output is the output of the whole circuit.

Partition

Given an instance of partition problem S, where S is a multi-set of positive integers. If the set S can be partitioned into two sets S_1 and S_2, and the sum of all the elements in S_1 equals the sum of all the elements in set S_2.

Assume $S = \{s_1, \ldots, s_n\}$, there are n elements in it. The circuit for the partition question has n input wires, and an input wire i_j is set to 1 if and only if the j^{th} elements are selected into set S_1.

The reduction of the constraints of partition problems is as follows:

1. Every input wire i_j ($j \in [1, \ldots, n]$) is set as a chooser wire of a Choose sub-circuit, and the other part of the Choose sub-circuit is the output of IntegerAsCircuit sub-circuit for the value of the i^{th} element in S. Then all the output wires of Choose sub-circuits then pass to a Sum sub-circuit to get the summation. Assume the sum of all the elements in S is T. Later the output of IntegerAsCircuit sub-circuit for $T/2$ and the output of the Sum sub-circuit are input to two LessEquals sub-circuits with different orders and check if they are the same.

Max cut

Given a max cut problem $G = \langle E, V \rangle$, where $V = \langle u_1, \ldots, u_n \rangle$ and $E = \langle e_1, \ldots, e_m \rangle$. If there exists a subset of vertex S_1 such that the number of edges of connecting set S_1 and its complementary subset S_2 reaches its maximum. $S_1 \bigcap S_2 = \emptyset$ and $S_1 \bigcup S_2 = S$.

The corresponding circuit should be capable to check if the number of edges connecting S_1 and S_2 equals some integer k. Then every time by changing the number of k, we can finally reach its maximum.

The circuit for the above sub-question has n input wires where n is the number of vertices. A wire i_j is set to 1 if and only if the j^{th} vertex is selected into set S_1.

Below is the reduction of the above sub-question to a Circuit-SAT problem.

1. For every edge $e_i = \langle u_j, u_k \rangle$ in G, input the two corresponding wires for u_j and u_k into a XOR sub-circuit. An off-the-shelf XOR sub-circuit can be used, or built by using combinations of AND, OR, and Bit-Not sub-circuits. The outputs of all the XOR sub-circuits are then passed to a Sum sub-circuit, which represents the number/sum of the edges connecting two sets. Then similarly, the output of IntegerAsCircuit sub-circuit for k and the output of the Sum sub-circuit are passed to two LessEquals sub-circuits with different orders to check if the number is equal to k.

Since $k \in [0, \ldots, m]$ ($E = \langle e_1, \ldots, e_m \rangle$), to find the maximization, if the current selection k is unsatisfiable, then the k for next try can be set to $k/2$; and if the current selection k is satisfiable, the program can try $k = (k + m)/2$ for the next term until we reach the max border.

Chapter 3

The Circuit Construction Service

The service proposed has versatile applications across various domains, due to the enhanced convenience in constructing sub-circuits involving digital-related operations compared to traditional circuits that only consist of AND, OR, and NOT gates, the service supported in this thesis can be applied to solve a wider range of problems, ranging from resource allocation, program scheduling to path selection and beyond.

To solve a problem in NP, once it is reduced to the Circuit-SAT problem which may include the enhanced sub-circuits introduced in Chapter 2.2, the proposed circuit construction service can be utilized to address it. The process involves several steps: firstly, clients should format the circuit to be built in JSON and pass it to the service, allowing the Circuit Construction Service to decode and retrieve the circuit's pertinent details. Secondly, the Circuit Construction Service utilizes the *circuit* library to construct and solve the circuit, and if a circuit is satisfiable, a certificate will be provided alongside the solution. Lastly, once the user receives the answer, they can deduce the final result according to the reduction process. By following these steps, the circuit construction service enables efficient and professional circuit-solving capabilities.

All the steps mentioned above are presented thoroughly in chapter 3 along with several examples. Besides, the service is deployed on AWS Lambda, a serverless computing platform provided by Amazon Web Services, which eliminates the need to consider server provisioning and maintenance during the deployment process and make responses to incoming requests. The deployment details are elaborated at the end of this chapter.

3.1 Main Components

3.1.1 JSON Encoding of a Circuit

Before constructing a circuit, it is essential to establish a standardized expression format for circuits that can be parsed by the service. In this thesis, JavaScript Object Notation (JSON) is used as the input format for the service, which encompasses all the information about the circuit to be constructed. After receiving a JSON encoding of a circuit, the input is parsed during the initial phase of the service's construction process.

JSON is a lightweight and widely adopted data interchange format used for storing and representing structured data. It employs name/value pairs as the fundamental building blocks to store information. JSON's popularity stems from its simplicity and efficiency in transmitting data between servers. Its flexibility and language independence enable seamless parsing and generation across multiple programming languages, making it exceptionally versatile and widely utilized in various industries.

JSON exhibits a straightforward and intuitive syntax characterized by name/value pairs enclosed in curly braces and separated by commas. The syntax of JSON originates from the syntax used for JavaScript objects, which holds the following properties [1]:

- JSON object is presented within curly braces, which consists of a collection of name/-value pairs separated by commas.
- Array holds organized sequences of values and is enclosed within square brackets.
- The value of JSON data could be a string (enclosed in double quotes), a number, a boolean (true or false), null, or even nested objects.

The versatility of the values enables JSON to represent intricate data structures and facilitate effortless information interpretation and communication across a wide range of platforms. Below is a short code snippet depicting a JSON object for a game:

Listing 3.1: A JSON object for a game.

```
{
  "game": "The Legend of Zelda: Breath of the Wild",
  "releaseYear": 2017,
  "platforms": ["Nintendo Switch", "Wii U"],
  "publisher": "Nintendo",
  "is3D": true
}
```

The JSON format of a circuit that the server can parse has the following characteristics:

1. Each JSON encoding is specifically designed to store a single circuit.

2. The initial name within the JSON object is labeled as "InputWire", serving to indicate the number of input wires connected to the circuit and the corresponding value for this name should be an integer.

3. The subsequent name, "Gates", is responsible for preserving the precise interconnections present within the circuit. Its value holds a list of sub-circuits, with each sub-circuit being an autonomous JSON object consisting of five name/value pairs including "ID", "Type", "Input", "Output" and "Info", which collectively encompass essential information regarding the sub-circuits identification, type, input connections, output connections, and additional details.

As for a sub-circuit, the name/value pairs in the JSON object are described as follows:

1. **ID**: In a circuit, each input wire and sub-circuit is assigned a unique ID. Starting from ID=1, all input wires are assigned IDs in sequential order. For example, if there are 4 input wires, each input wire should receive a distinct ID ranging from 1 to 4. The IDs for each sub-circuit are assigned sequentially after the number of input wires. Using the previous example, all sub-circuits should be assigned non-repeating IDs greater than 4. Besides, the value here should only be a number.

2. **Type**: The "Type" field simply denotes the type of a sub-circuit, and its corresponding value is a string chosen from the following options: "And", "Or", "BitNot", "LessEquals", "IntegerAsCircuit", "Sum", and "Choose".

3. **Input**: The value of the "Input" name for a sub-circuit is a list of sub-circuit IDs, and if there are no input wires towards it, it keeps empty. The IDs in it represent the sub-circuits whose outputs serve as inputs to the current sub-circuit. Additionally, the order of the IDs in the input gate list corresponds to the construction order of the current sub-circuit. For instance, for sub-circuits such as LessEquals and Sum where the input order matters, the gate associated with the first ID in the input ID list establishes its outputs as the initial group of inputs for the current sub-circuit. This pattern continues with the subsequent IDs in the list, dictating the input order for subsequent groups within the sub-circuit.

4. **Output**: The value of the "Output" name for a sub-circuit is a list of sub-circuit IDs, and if its output is the one output wire for the whole circuit, it remains empty.

The IDs in the list represent the sub-circuits whose inputs serve as outputs to the current sub-circuit.

5. **Info**: The "Info" name is initialized as -1 by default for all types of sub-circuits, except for the IntegerAsCircuit. The IntegerAsCircuit sub-circuit requires a positive integer as input and generates the corresponding binary representation of that. Therefore, the value of the "Info" name for the IntegerAsCircuit sub-circuit is the intended number to be generated.

Below is an example of a circuit encoding in JSON format, which takes 3 wires as input and contains two AND sub-circuits and an OR sub-circuit. The output of the AND sub-circuit with ID=6 is the output of the entire circuit.

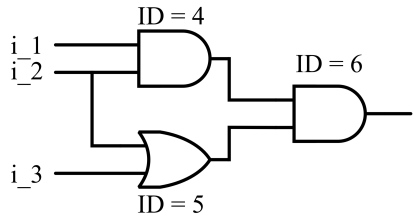


Figure 3.1: An example circuit.

Listing 3.2: A JSON object for the circuit in figure 3.1.

```
{
  "InputWire": 3,
  "Gates": [
    {"ID": 4,
    "Type": "And",
    "Input": [1,2],
    "Output": [6],
    "Info": -1},
    {"ID": 5,
    "Type": "Or",
    "Input": [2,3],
    "Output": [6],
    "Info": -1},
    {"ID": 6,
    "Type": "And",
    "Input": [4,5],
    "Output": [],
    "Info": -1}
  ]
}
```

3.1.2 Circuit Library

Circuit-SAT Library Circuit-SAT Library was designed and developed by my supervisor Mahesh Tripunitara, which provides multiple functions for building and solving an instance for a Circuit-SAT problem. In chapter 2.2, four new sub-circuits are introduced along with their specific functions and features. In this thesis, these mentioned sub-circuits are treated as black boxes, emphasizing solely their inputs, outputs, and operations. While these sub-circuits are primarily constructed using AND, OR, and NOT gates, their intricate implementation intricacies are beyond the preview of this thesis.

The *circuit* library offers functionalities for manipulating the *Circuit* object, which contains nested classes *Circuit.Gate*, *Circuit.GateType*, and *Circuit.Wire*. In the context of the Circuit-SAT problem, a wire refers to an instantiation of the *Circuit.Wire* class, and a gate refers to an instantiation of the *Circuit.Gate* class. The *GateType* class is an enumeration type designed to represent various types of logic gates. It provides a

set of enum constants, including AND, OR, and NOT, which can be utilized within the construction function *Gate(Circuit.GateType t, Circuit.Wire[] in, Circuit.Wire out)* of the *Circuit.Gate* class. The *Circuit.Wire* and *Circuit.Gate* classes both have a common method *getId()* which allows retrieving their respective IDs within a circuit. Additionally, the *getInputTo()* and *getOutputOf()* methods of the *Circuit.Wire* class provide access to the two *Circuit.Gate* objects that an instantiated *Circuit.Wire* object is connected to. On the other hand, the *getInputs()* and *getOutput()* methods of the *Circuit.Gate* class allow obtaining the input wire list and output wire list, respectively, for an instantiated gate object.

The fields of the *Circuit* class encompass the collections of gates, inputs, outputs, and wires. The gate field represents the set of all instantiated gate objects contained within a *Circuit* object; similarly, the wire field represents the set of all instantiated *Circuit.Wire* objects. Besides, The inputs and outputs fields respectively represent the input wire list and output wire list of a *Circuit* object. Besides, the *Circuit* class also offers a variety of methods that can be applied to instantiated objects to modify the internal circuit structure.

- **resetIDs()**: This Application Programming Interface (API) needs to be called to update the cache in the library before every new circuit is built.
- **Circuit()**: The constructor function, by utilizing it to instantiate objects, the resulting instances represent the Circuit-SAT problem to be solved.
- **union(Circuit c)**: Accept another circuit, denoted as c, as an input and perform a union operation with the *Circuit* object that invokes the *union()* function. As a result, the invoking circuit then becomes the final union circuit. The inputs of the c circuit are added as inputs of the invoking circuit, and the same applies to the outputs. Please be aware that this routine does not verify the uniqueness of IDs or any other constraints, and it simply performs a basic union operation.
- **getInputs()**: Return a list of the input wires of the *Circuit* object that invokes the *getInputs()* function.
- **getOutputs()**: Return a list of the output wires of the *Circuit* object that invokes the *getOutputs()* function.
- **fuse(Circuit.Wire p, Circuit.Wire t)**: Fuse wires p and t together to merge them into a single wire. This operation is commonly used to connect the output wire of one (sub-)circuit to the input wire of another (sub-)circuit, and the fused wire is returned as the result. It is important to note that the argument p represents the "permanent"

wire that will be retained, while the argument t represents the "throw-away" wire that will be discarded.

- **removeAsInput(Circuit.Wire w)**: Take the wire w (which should be in the input wire list) and remove it from the input wire list. This function is usually called before the *fuse()* method, since in most cases, it is of high necessity to first remove the *Circuit.Wire* object from the input wire list of a Circuit before fusing it with another wire.

- **removeAsOutput(Circuit.Wire w)**: Take the wire w (which should be in the output wire list) and remove it from the output wire list. This function is usually called before the *fuse()* method, since in most cases, it is of high necessity to first remove the *Circuit.Wire* object from the output wire list of a Circuit before fusing it with another wire.

Moreover, the *Circuit* class has sub-classes: BigAndOr, BitNot, Choose, IntegerAsCircuit, LessEquals, Sum, etc., and the objects generated by instantiating these sub-classes represent sub-circuits with the corresponding functionalities, and these sub-classes also inherit the aforementioned methods from their parent class.

- **BigAndOr(boolean isOr, int nInputWires)**: Create a big AND or OR circuit based on the provided parameters. The first argument, a boolean value, determines whether an OR gate should be created (isOr = true) or an AND gate should be created (isOr = false). The second argument, nInputWires, specifies the number of input wires for the sub-circuit. For instance, if nInputWires is equal to 2, only one gate will be generated as the sub-circuit since each AND and OR gate requires two inputs. If nInputWires is set to 4, the resulting circuit will consist of three gates. The number of output wires in the sub-circuit is always 1 as usual.

- **BitNot()**: Create a circuit that takes 1-bit input, and has 1-bit output, which is the negation of the input.

- **Choose(int y)**: Construct a circuit to select (or not select) the specified y bits. In a Choose circuit, the first input wire represents the chooser bit, followed by the rest y input wires. Therefore, the total number of input wires is equal to 1 + y, and the circuit's output consists of y wires. If the first chooser bit is set to 1, the Choose circuit will select the y input wires, and the output of the circuit will be the same as the selected y wires. On the other hand, if the first chooser bit is set to 0, the y wires will not be selected, and all output bits will be assigned a boolean value of 0.

- **IntegerAsCircuit(int i)**: Encode and return the non-negative integer i in binary as a boolean circuit.
- **LessEquals(int x, int y)**: Return a circuit with x+y input wires. The circuit compares whether the binary number represented by the first x bits is less than or equal to the binary number represented by the next y bits. If this comparison holds true, the output wire will be assigned a boolean value of 1; otherwise, it will be assigned a value of 0.
- **Sum(java.util.List<java.lang.Integer> x)**: Return a circuit that has the number of input wires equal to the sum of integer values in x. The input is to be perceived as $|x|$ binary numbers, and the circuit represents the sum of those binary numbers. Thus, the number of output wires of the circuit is at most $log_2|x| + max_i x$, where $max_i x$ is the largest integer value in x.

For example, the subsequent code snippet depicts the functions employed in the construction of a circuit c and an OR gate (which has 2 inputs) in it, with an assumption that the output of the OR gate is not the final output of circuit c.

Listing 3.3: Define a new circuit c and an OR gate.

```
public static String ConstructCitcuitC() {
    // Define a new circuit c.
    Circuit.resetIDs()
    Circuit c = new Circuit()

    // Build an OR gate with 2 inputs.
    Circuit firstOR = new BigAndOr(true, 2)
    // Union the OR gate with circuit c.
    c.union(firstOR)
    // Remove the OR gate output wires form the circuit's output.
    c.removeAsOutput(firstOR.getOutput.get(0)
    ...
}
```

The following code describes the statements used for solving the Circuit-SAT problem based on circuit c. After the *cnf.dimacs* file is generated by calling function *cnfSatToFile(Circuit c, java.lang.String loc)*, the service uses SAT4J, which is a Java library for solving CNF-SAT problems, to solve the CNF-SAT problem in file *cnf.dimacs*.

The *CircuitUtils* class in the circuit library provides functions for reducing the Circuit-SAT problem represented by circuit c to a CNF-Satisfiability problem. Once a circuit is constructed, the *cnfSatToFile(Circuit c, java.lang.String loc)* method is used to convert the circuit c to a one-output CNF-Satisfiability problem and stores the formula in a DIMACS format file[1].

During the process of reducing a Circuit-SAT problem according to the circuit c to a boolean satisfiability problem, all the input wires in c are mapped to the literals in the SAT problem in the order. For example, if there are 10 input wires to c, the reduced CNF formula will contain 10 literals $x_1, x_2, \ldots, x_{10}$ as well, which correspond to those 10 input wires in order. On the other hand, if the CNF formula is satisfiable, the boolean assignments to the first 10 literals given by SAT solver are the assignments to the 10 input wires of the original circuit c.

[1]The DIMACS format file employs the standardized DIMACS format to store the CNF formula, which can be parsed and solved by SAT solvers. In this paper, it stores the CNF formula resulting from the reduction from the Circuit-SAT encoded in the JSON input received by the service.

Listing 3.4: Solve the Circuit-SAT problem.

```
// Import the external libraries SAT4J for future access.
import org.sat4j.specs.ISolver;
import org.sat4j.minisat.SolverFactory;
import org.sat4j.reader.Reader;
import org.sat4j.reader.DimacsReader;

public static String ConstructCitcuitC() {
    ...
    // Convert circuit c, a Circuit-SAT problem, into a CNF-Satisfiability
       problem in DIMACS.
    CircuitUtils.cnfSatToFile(c, "cnf.dimacs");

    // Use SAT4J solver to solve the generated boolean SAT problem.
    ISolver solver = SolverFactory.newDefault();
    solver.setTimeout(3600);
    Reader reader = new DimacsReader(solver);
    reader.parseInstance("cnf.dimacs");

    // ISolver object holds method isSatisfiable() to check if the problem
       is satisfiable.
    if (solver.isSatisfiable()) {
        int[] solution = solver.model();
        return = "Solution found: " + Arrays.toString(solution);
    } else {
        return "Unsatisfiable!";
    }
}
```

3.1.3 Working Principle

This service, upon receiving the JSON-encoded representation of a circuit, leverages the *circuit* library introduced in the previous section (chapter 3.1.2) to construct the circuit. Firstly, during the process of parsing the original input's string-formatted circuit information, the service utilizes the *JSONObject.fromObject()* function to instantiate the input content as a *JSONObject* object. Subsequently, the service can invoke the built-in functions of the *JSONObject* object in Java to extract the contents of relevant values based on

the name's type. Finally, through the parsing of the input circuit, each sub-circuit in the list associated with the "*Gates*" name in the original input is instantiated as a *Gate* object. The *Gate*[2] class is a newly defined internal class within the service, facilitating convenient access to the circuit information being constructed. Each *Gate* object encompasses five attributes:

- *id*, corresponds to the name "*ID*" in the JSON encoding[3] of each sub-circuit.
- *type*, corresponds to the name "*Type*" in the JSON encoding of each sub-circuit.
- *input_id_list*, corresponds to the name "*Input*" in the JSON encoding of each sub-circuit.
- *output_id_list*, corresponds to the name "*Output*" in the JSON encoding of each sub-circuit.
- *info_num*, corresponds to the name "*Info*" in the JSON encoding of each sub-circuit.

Specifically, a *Gate* object is instantiated based on the information within a sub-circuit's JSON encoding. The instantiated attributes such as "*id*", "*type*", and "*info_num*" correspond to the values of the "*ID*", "*Type*", and "*Info*" of the respective sub-circuit. The "*input_id_list*" and "*output_id_list*" attributes of the *Gate* object keep track of the *ID*s of the other sub-circuits connected to it in the input and output directions respectively.

After obtaining all the sub-circuit information within a circuit, to ensure the accuracy of subsequent computations, the service first performs error checking on all the generated *Gate* objects, including but not limited to verifying that their *id* attributes are unique positive integers and that the content of the *type* attribute belongs to the range of sub-circuit types provided by the *circuit* library. Additionally, specific error checks are performed for different types of *Gate* objects. For example, for all *Gate* objects with a *type* equal to *LessEquals*, their *input_id_list* can only contain two integers. For all objects with a type equal to *IntegerAsCircuit*, their *info_num* attribute must be a non-negative integer, and the *input_id_list* must be an empty list.

The service also performs overall validity checks on the circuit. For instance, the circuit composed of all generated *Gate* objects must have one and only one output wire, meaning

[2]Please note that the term "Gate" is used as a class name in the code, and each instantiated *Gate* object is responsible for storing information about a sub-circuit. Here, "Gate" is not referring to the logic gates introduced in chapter 1, but rather to sub-circuits presented in chapter 2, which perform specific logical or mathematical operations.

[3]The JSON encoding of a sub-circuit is described in chapter 3.1.1

that there should be only one *Gate* object whose *input_id_list* is an empty list. The matching of input and output relationships among *Gate* objects is also included in the scope of the checks. If *Gate* A's *input_id_list* contains the ID of *Gate* B, *Gate* B's *output_id_list* must also include the ID of *Gate* A.

Once the service completes the error checking process, which includes but is not limited to the aforementioned checks, it proceeds to classify all the sub-circuits to be constructed into a regular group and a special group based on their types. The regular group includes sub-circuits such as *AND*, *OR*, *Bit-Not*, and *IntegerAsCircuit*, which the number of input wires could be determined based on its list of input IDs. For example, when constructing an *AND* sub-circuit, the number of input wires can be determined based on the *input_id_list* attribute of the corresponding instantiated *Gate* object, allowing the construction of that *AND* sub-circuit. A special group consists of sub-circuits with unknown specific numbers of input wires, such as *Choose*, *LessEquals*, and *Sum*. For instance, when constructing a *LessEquals* sub-circuit, two exact numbers x and y are needed for the construction function, and x and y are both the number of wires that a binary representation of a number contains, so it is not possible to determine the exact number of wires included in each input solely based on two IDs in its *input_id_list*.

To construct the final circuit, the function first instantiates a *Circuit* object, c, by calling the constructor of the *Circuit* class from the *circuit* library. Then, add the corresponding number of input wires to the circuit according to the required quantity. A tracking list is used to record the constructed sub-circuits and their corresponding IDs. To ensure subsequent retrieval and other operations related to the stored sub-circuits in this thesis, a new class named *CircuitID* has been introduced. This class comprises three attributes:

- the ID of a sub-circuit
- the actual sub-circuit itself
- the count of input wires already fused for that sub-circuit's input terminal (set to 0 by default).

After creating each sub-circuit, an instance of the *CircuitID* object is instantiated based on its ID and stored in the tracking list. Next, traverse all the *Gate* objects that need to be built, the corresponding sub-circuits belonging to the normal group are constructed first. The number of input wires for each sub-circuit is determined by parsing the *input_id_list* of its corresponding *Gate* object, and the sub-circuit is instantiated accordingly. Then, the newly instantiated sub-circuit is merged with the circuit c using the *union()* function. Based on its *input_id_list* and *output_id_list*, the decision is made whether to use the *removeAsInput()* and *removeAsOutput()* functions remove the input and output wires of the

new sub-circuit from the input and output wires of the circuit c. In addition, wire fusion within circuit c will be conducted after all the sub-circuits are constructed. In the end, the CircuitID object instantiated according to the newly constructed sub-circuit is added to the tracking list.

As for constructing the sub-circuit in the special group, the service utilizes a while loop to perform the following operations as long as the length of the tracking list is less than the number of sub-circuits to be constructed. The service iterates through all the sub-circuits awaiting construction, and for every sub-circuits belonging to the special group, it first checks if all its source sub-circuits, identified by whether the sub-circuit ID is in the *input_id_list* or not, have already been constructed (i.e., present in the tracking list). If all the source sub-circuits have been built, the service accesses them from the tracking list to obtain the output wire count of their corresponding circuits. However, if there are IDs in the *input_id_list* that correspond to sub-circuits not yet constructed (i.e., not present in the tracking list), the service will skip those particular special sub-circuits and try to instantiate them in the next iteration of the while loop. Through repeated iterations of the while loop, the circuit is gradually constructed step by step, starting from the input direction and progressing toward the output direction.

After all the sub-circuits have been constructed and unioned with the final circuit c, the service proceeds to fuse the input and output wires between all the sub-circuits. By iterating through the tracking list generated in the previous step, which records all the completed sub-circuits, for each sub-circuit, the service again traverses the *input_id_list* attribute of its corresponding *Gate* object. It fuses each output wire of the sub-circuit associated with every ID in the *input_id_list* with the input wire of that sub-circuit in sequence. In addition, if an ID belongs to an input wire of circuit c, the service connects that input wire to the corresponding sub-circuit's input.

The following pseudocode illustrates the logic used to construct an entire circuit, along with algorithms for creating an AND sub-circuit, a Sum sub-circuit, and the *Fuse()* function.

Algorithm 1 Circuit Construction - AND Circuit

1: **Inputs:**

- the circuit that the AND sub-circuit belongs to – c
- the tracking gate list recording all the constructed sub-circuits – tracking_gate_list
- the *Gate* object which contains all the information of building the AND circuit – normalGate.

2: **Output:** None
3: **function** GATEANDCONSTRUCTION(c, tracking_gate_list, normalGate)
4: **Circuit** and_Gate = **new BigAndOr(false, normalGate.input_id_list.size())**
5: c.union(and_Gate)
6: **for** $i = 0$ to $normalGate.input\_id\_list.size() - 1$ **do**
7: c.**removeAsInput(and_Gate.getInputs().get(i))**
8: **if not** normalGate.output_id_list.isEmpty() **then** ▷ Determin if the output of the AND sub-circuit is the final output of the circuit c.
9: **for** $i = 0$ to $normalGate.output\_id\_list.size() - 1$ **do**
10: c.**removeAsOutput(and_Gate.getOutputs().get(i))**
11: tracking_gate_list.add(CircuitID(and_Gate, normalGate.id))
12: return

Algorithm 2 Circuit Construction - Sum Sub-circuit

1: **Inputs:**

- the circuit that the Sum sub-circuit belongs to – c
- the tracking gate list recording all the constructed sub-circuits – tracking_gate_list
- the *Gate* object which contains all the information of building the AND circuit – specialGate.

2: **Output:** None
3: **function** GATESUMCONSTRUCTION(c, numWires, tracking_gate_list, specialGate)
4: **for all** srcId ∈ specialGate.input_id_list **do** ▷ Check if the two input source gates have all been constructed.
5: **if** srcId > numWires **then**
6: flag ← false
7: **for all** constructedGate ∈ tracking_gate_list **do**
8: **if** constructedGate.id == srcId **then**
9: flag ← true
10: break for
11: **if not** flag **then** return
12: sumInput ← [·] ▷ Input source gates have all been constructed.
13: **for all** constructedGate ∈ tracking_gate_list **do**
14: **if** constructedGate.id ∈ specialGate.input_id_list **then** sumInput.add(constructedGate**.getOutputs().size()**)
15: **Circuit** sum_Gate = **new Sum(sumInput)**
16: c.union(sum_Gate)
17: **for** $i = 0$ to $specialGate.input\_id\_list.size() - 1$ **do**
18: c.**removeAsInput(sum_Gate.getInputs().get(i))**
19: **if not** specialGate.output_id_list.isEmpty() **then** ▷ Determin if the output of the Sum sub-circuit is the final output of the circuit c.
20: **for** $i = 0$ to $specialGate.output\_id\_list.size() - 1$ **do**
21: c.**removeAsOutput(sum_Gate.getOutputs().get(i))**
22: tracking_gate_list.add(CircuitID(sum_Gate, specialGate.id))
23: return

Algorithm 3 Fuse Wire

Inputs:

- the final circuit - c
- the gate list contains all the sub-circuits information – gateList
- the input wire list of circuit c – input_wire_list
- the tracking gate list recording all the constructed sub-circuits –tracking_gate_list

Output: None

function FUSEWIRE(c, gateList, input_wire_list)

    numWires ← input_wire_list.wires()

    for all Gate ∈ gateList **do**

        if not Gate.input_id_list.isEmpty() **then**

            for all source ∈ Gate.input_id_list **do**

                if source <= numWires **then** ▷ The source ID corresponds an input wire

                    des ← circuitTemp, circuitTemp ∈ tracking_gate_list & circuitTemp.id == gate.id

                    c.**fuse(input_wire_list.get(source-1, des.sub_circuit.getInputs().get(des.numFused))**

                    des.numFused + = 1

                else ▷ The source ID corresponds to a sub-circuit.

                    src ← circuitTemp, circuitTemp ∈ tracking_gate_list & circuitTemp.id == source

                    des ← circuitTemp, circuitTemp ∈ tracking_gate_list & circuitTemp.id == gate.id

                    for $i = 0$ to $src.sub\_circuit.getOutputs().size()$ **do**

                        c.**fuse(src.sub_circuit.getOutputs().get(i), des.sub_circuit.getInputs().get(des.numFused))**

                        des.numFused + = 1

    return

Algorithm 4 Circuit Construction - The Entire Circuit

```
1: Inputs: the number of input wires – numWires, a list of circuits to be constructed – gateList. Output: the constructed circuit c
2: function CircuitConstruction(numWires, ArrayList<Gate> gateList)
3:     input_wire_list, tracking_gate_list, special_gate_list, normal_gate_list ← [·], [·], [·], [·]
4:     for all gate ∈ gateList do
5:         if gate.type ∈ {"And", "BitNot", "Or", "IntegerAsCircuit"} then
6:             normal_gate_list.add(gate)
7:         else if gate.type ∈ {"Choose", "Sum", "LessEquals"} then
8:             special_gate_list.add(gate)
9:     Circuit.ResetIDs()
10:    circuit.Circuit c = new circuit.Circuit()
11:    for i = 0 to numWires − 1 do
12:        c.addNewInput()                    ▷ Add numWires input wires to circuit c.
13:        input_wire_list.add(c.getInputs.get(i))
14:    for all normalGate ∈ normal_gate_list do                ▷ Construct normal gates.
15:        if normalGate.type == "And" then
16:            AndConstruct(c, tracking_gate_list, normalGate)
17:        else if normalGate.type == "Or" then
18:            OrConstruct(c, tracking_gate_list, normalGate)
19:        else if normalGate.type == "Not" then
20:            NotConstruct(c, tracking_gate_list, normalGate)
21:        else if normalGate.type == "IntegerAsCircuit" then
22:            IACConstruct(c, tracking_gate_list, normalGate)
23:    while tracking_gate_list.size() < gateList.size() do        ▷ Construct special gates.
24:        for all specialGate ∈ special_gate_list do
25:            if specialGate ! = constructedGate.sub_circuit, constructedGate ∈ tracking_gate_list then
26:                if specialGate.type == "Choose" then
27:                    ChooseConstruct(c, numWires, tracking_gate_list, specialGate)
28:                else if specialGate.type == "LessEquals" then
29:                    LEConstruct(c, numWires, tracking_gate_list, specialGate)
30:                else if specialGate.type == "Sum" then
31:                    SumConstruct(c, numWires, tracking_gate_list, specialGate)
32:    FuseWire(c, gateList, input_wire_list, tracking_gate_list)
33:    return c
```

Finally by converting the Circuit-SAT problem based on the constructed circuit c to a CNF-SAT problem by calling the function *CircuitUtils.cnfSatToFile()*, the problem is then solved by using solver SAT4J later. After obtaining the result, if it is satisfiable, the boolean values corresponding to each input wire will also be provided, and the solution to the original problem is also answered.

3.1.4 AWS Lambda Deployment

Cloud computing services have emerged as a rapidly growing industry within the realm of computer engineering over the past few decades. These services enable clients to effortlessly access the resources and services they require through the network dynamically, without the need for physical servers. By efficiently managing and orchestrating a vast array of interconnected computing resources, cloud computing services form a cohesive computing resource pool, which seamlessly delivers on-demand services to users, allowing for scalability and expansion as needed. Amazon Web Services (AWS) is one of the most prominent and largest cloud computing providers globally and it is the platform used for server deployment in this thesis. AWS offers an extensive suite of cloud computing services encompassing elastic computing, storage, databases, Internet of Things (IoT) capabilities, and a diverse range of managed products. Leveraging AWS, individuals can effectively address a wide spectrum of software development requirements, all while alleviating the dependency on physical servers.

In 2014, AWS introduced AWS Lambda, pioneering the revolutionary concept of serverless computing. Serverless computing, also referred to as Function as a Service (FaaS), represents a cloud computing model empowering developers to write and deploy code as self-contained functions, eliminating the burden of managing the underlying infrastructure [11]. Within this architecture, the cloud service provider dynamically allocates resources and scales applications in response to the workload. AWS Lambda seamlessly processes events by executing function instances, whereby functions serve as the fundamental building blocks. Users and other AWS services can effortlessly invoke functions by dispatching JSON-formatted events, which are meticulously processed and executed within the Lambda environment, thereby enabling a diverse range of use cases. Once deployed, users gain the flexibility to directly invoke functions using the Lambda API or seamlessly configure AWS services and resources to trigger the execution of these functions [4].

Service Deployment on AWS Lambda

1. **Code preparation and packaging**
   Maven is employed and serves as a robust tool for organizing and managing code and its associated dependencies. The directory structure of the optimization service is as follows:

```
/JAVA-build-lambda
├── pom.xml
├── src
│   ├── main
│   │   └── java
│   │       ├── circuit
│   │       └── dev.jackie
│   │           ├── handlers
│   │           │   └── CirConSolveHandler
│   │           └── ...(source code)...
│   └── test
│       ├── java
│       └── circuit
└── target
```

The root directory of the project, named *JAVA-build-lambda*, contains a project description file called *pom.xml*. This file is responsible for documenting the required dependencies for the source code and serves as the unique identifier for this Maven project. The directory *src/main/java/dev/jackie* is designated for storing the Java source code, while *src/main/-circuit* is the directory for the *circuit* library introduced in chapter 3.1.2. The *src/test/java* directory is specifically assigned for housing the test source code, while the *src/test/circuit* directory is dedicated to storing the generated test resources of the *circuit* library. Finally, upon compilation and packaging, all resulting files are consolidated and located within the *target* directory.

Within the *pom.xml* file, the following dependencies are declared, which encapsulate crucial functionalities and features that is used by the source code:

- *org.apache.commons:commons-lang3:3.11*: Apache Commons Lang library, version 3.11. It provides a broad variety of functionality for standard string manipulation and object utility functions.
- *commons-beanutils:commons-beanutils:1.7.0*: Apache Commons BeanUtils library, version 1.7.0. The manipulation of JavaBeans is supported by this library.

- *commons-collections:commons-collections:3.2.1*: Apache Commons Collections library, version 3.2.1. It offers broad effective data-handling implementations.
- *commons-logging:commons-logging:1.1.1*: Apache Commons Logging library, version 1.1.1. This library provides a simple and unified logging interface.
- *net.sf.ezmorph:ezmorph:1.0.6*: Ezmorph library, version 1.0.6. The library is used for Java object conversions.
- *org.ow2.sat4j:org.ow2.sat4j.core:2.3.5*: OW2 SAT4J library, version 2.3.5. This library offers a powerful solver for boolean satisfiability problems, which helps to solve the final cnf.dimacs file generated by the *circuit* library.
- *net.sf.json-lib:json-lib:2.4*: Json-lib library, version 2.4. It facilitates the processing and manipulation of JSON data.
- *org.junit.jupiter:junit-jupiter:RELEASE*: The latest version of the JUnit Jupiter library, which is employed for writing and executing unit tests.
- *com.amazonaws:aws-lambda-java-core:LATEST*: The most recent version of the AWS Lambda Java Core library, providing seamless functionality support when deploying Java functions on AWS Lambda.

To make AWS Lambda able to invoke the whole service as a function, AWS Lambda Java SDK provides *RequestHandler* interface to define handlers for Lambda functions. The function signature of the handler function is shown below, class *CirConSolveHandler()* is designed to implement the *RequestHandler* interface along with explicit generic type parameters. Within *CirConSolveHandler()*, method *public String handleRequest(Map<String, String> input, Context context)* is defined to take a *Map<String, String>* object as one of the input parameter, which contains the information needed to build a circuit. Besides, it takes a *Context* object as input as well, which contains the run-time execution information of the Lambda function. The *handleRequest()* function constructs a circuit according to the input, solves the corresponding Circuit-SAT problems and returns the answer in a string.

```java
package dev.jackie.handlers;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.io.*;
import java.util.Map;

import dev.jackie.Construct_Circuit;

no usages
public class CirConSolveHandler implements RequestHandler<Map<String, String>, String> {
    no usages
    public String handleRequest(Map<String, String> input, Context context) {
```

Figure 3.2: The handler function *handleRequest()*.

Finally, to upload the code as a function to AWS Lambda, the whole Maven is packaged into a .zip archive, including all the necessary dependencies and resources required. At this point, all the necessary preparations before uploading the code have been completed.

2. **Create an AWS Lambda function**
   To deploy the above function in AWS Lambda, firstly, search and navigate to the AWS Lambda page in the browser. Select "Create function" and enter a desired function name. Choose Java 11 as the runtime and select x86_64 as the architecture. Keep the remaining options at their default settings. Secondly, after the function is created, click on "Edit" in the "Runtime settings" section; and in the Handler field, enter the path *dev.jackie.handlers.CirConSolveHandler::handleRequest* to the handler function. Finally, Upload the zip package containing the Java function.

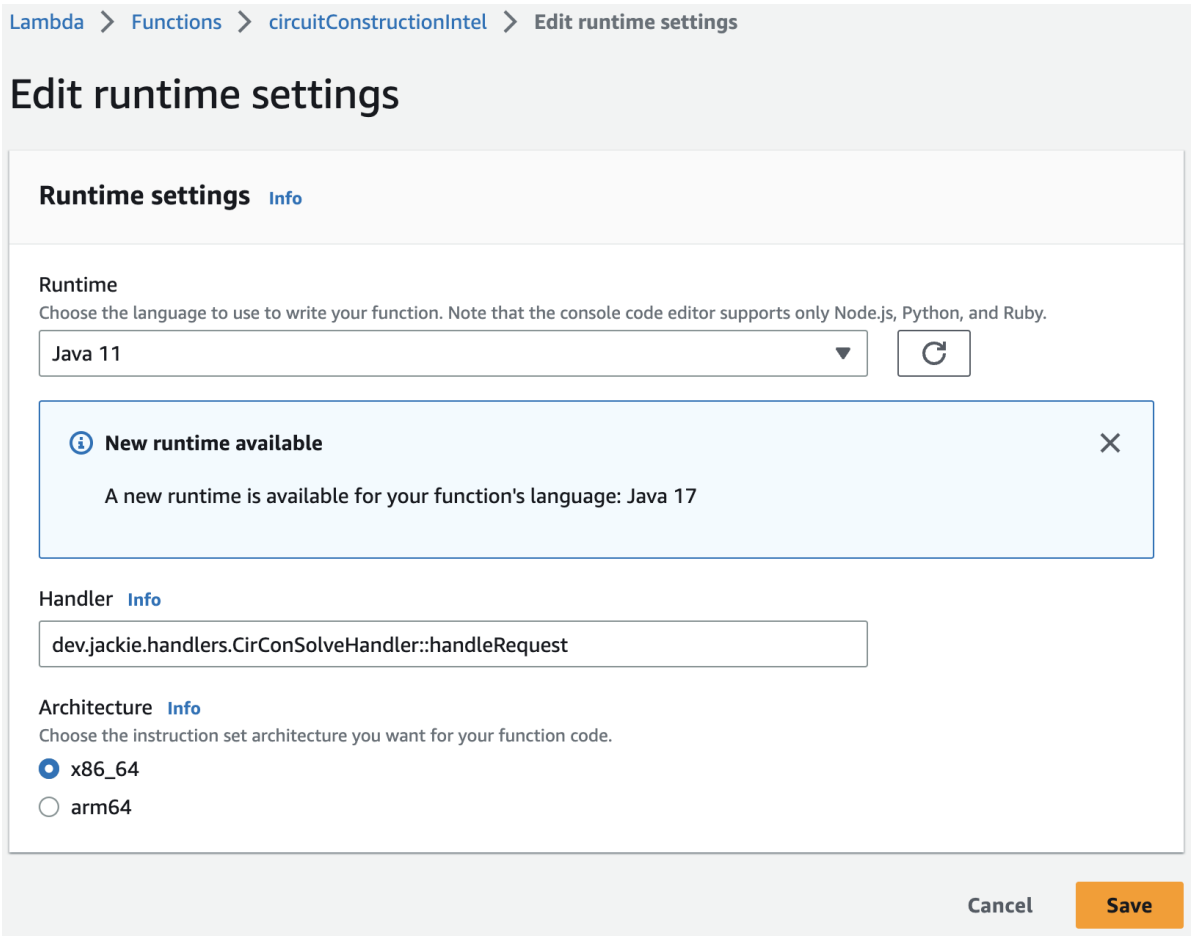


Figure 3.3: Adding the path to function *handleRequest()*.

During the first step of creating a function, the desired execution role can also be modified as needed, which determines the permissions granted to AWS Lambda functions for accessing AWS resources and services during runtime. In this thesis, the function deployment utilizes the default option "create a new role with basic Lambda permissions", since the role provided by this option already holds sufficient permissions for the function.

3. **Event**
   Once the function is deployed to AWS Lambda, it can be triggered by an event. In the case of the handler function *handleRequest()*, the event is expected to be in the form of a name/value pair of strings, where the name in the event is the string "fileContent", and the corresponding value is the JSON-encoded representation (as described in Section

3.1.1) of a circuit converted to a string format. Once an event triggers the *handleRequest()* function, the JSON encoding of the circuit is parsed based on the value corresponding to the "fileContent" name. This allows for further operations such as circuit construction to be performed and Circuit-SAT problems to be solved.

According to the previous introduction, the deployment of this optimization service on AWS Lambda eliminates the need for manual consideration of hardware and infrastructure provisioning, such as memory allocation and vCPUs. Besides, the service demonstrates event-driven characteristics as well, eliminating the requirement for continuous running in the background. Regarding billing, services are charged solely based on actual resource consumption during execution, without any fees for idle state service and resources. This pay-per-use pricing model ensures efficient resource allocation and optimal cost management for the deployed services. Hence, this streamlined deployment process not only enhances convenience but also demonstrates cost-effectiveness.

3.2 Result Demonstration

To test the service deployed on AWS Lambda and demonstrate its usage, the following knapsack is used for testing.

Given an integer $k = 17$, a group of 3 items each has weight: $\{2, 5, 8\}$, and value $\{4, 7, 10\}$. Now by picking items into a knapsack, which has a limit to the weight of the picked items $W = 14$, if the total value of picked items is greater than or equal to k?

In order to utilize the service for problem-solving purposes, the first step involves referencing the methods discussed in chapter 2.3 regarding the reduction of the knapsack problem to the Circuit-SAT problem. The circuit takes 3 input wires as input, $\{i_1, \ldots, i_n\}$, The wire is assigned a value of 1 if and only if the corresponding item is selected to be included in the knapsack. In order to verify whether the total weight of the selected items exceeds the specified limit $W = 14$, connect each input wire i_j to a Choose gate. Additionally, the other input of the Choose subcircuit is connected to the output of the IntegerAsCircuit subcircuit for w_j. Subsequently, the Choose outputs are combined into a SUM subcircuit. Afterwards, the output wires of the SUM subcircuit, along with another IntegerAsCircuit subcircuit for W, are connected as inputs to a LessEquals sub-circuit in order. Likewise, for verifying the value, the aforementioned subcircuit structure can be utilized to check if the value of the selected items exceeds k. The only distinction lies in converting the numerical output of the IntegerAsCircuit from weight to value. Finally, an AND gate is constructed, taking the outputs of the LessEquals subcircuits from the

two aforementioned subcircuits as inputs. The output of the AND gate serves as the final output of the entire circuit. figure 3.4 illustrates the resultant reduced Circuit-SAT problem.

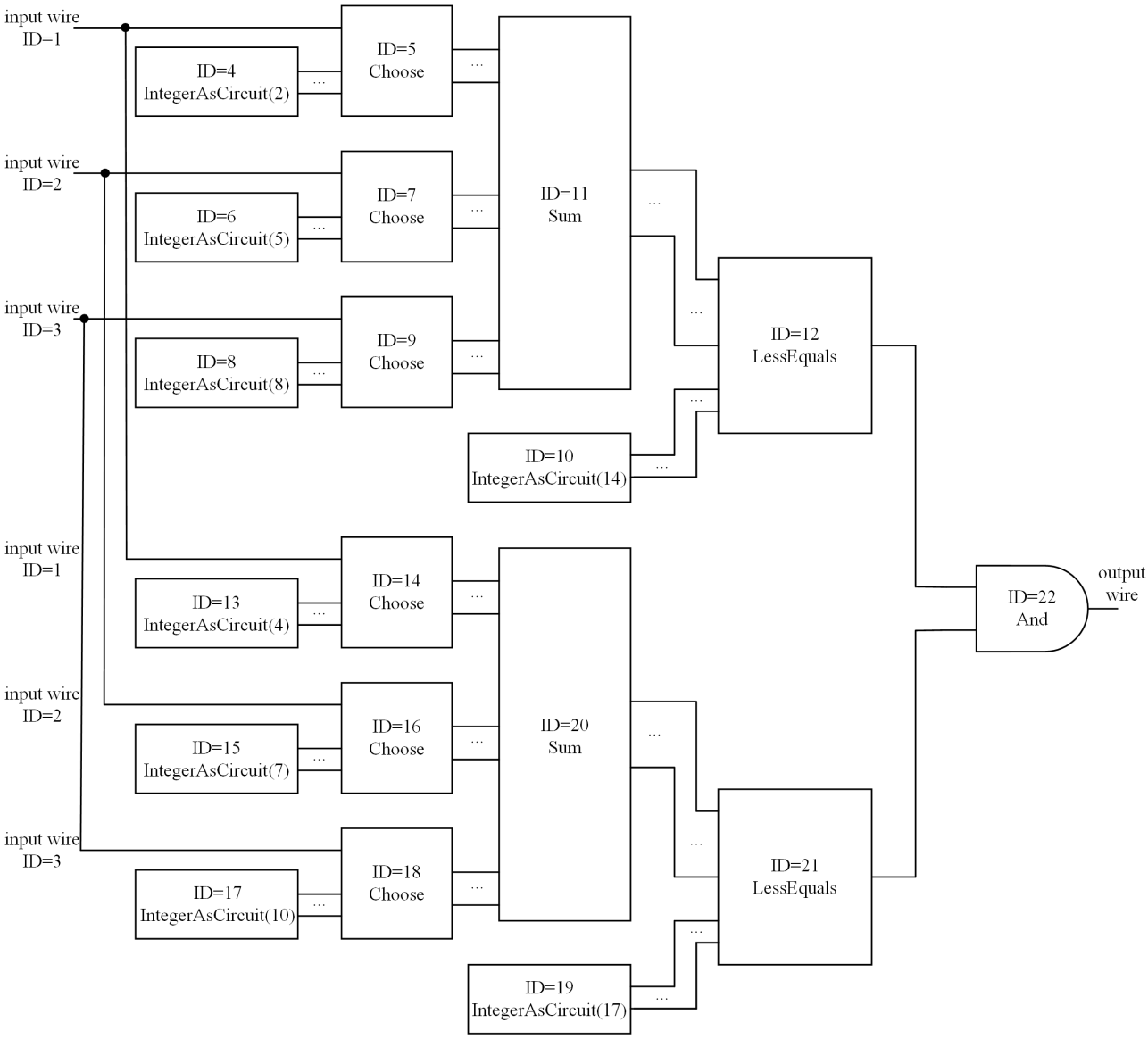


Figure 3.4: The Circuit-SAT problem reduced from the given knapsack problem.

Then according to the circuit in figure 3.4, the JSON encoding of the circuit should be

provided as the value of a name/value pair in the event to trigger the function deployed on AWS Lambda. The answer to this problem is as shown below. According to the answer, if the second item (weight of 5, value of 7) and the third item (weight of 8, value of 10) are chosen while excluding the first item (weight of 2, value of 4), it would satisfy the condition of selecting goods with a total weight not exceeding $W = 14$ and a value greater than or equal to $k = 17$. In fact, for this problem, the answer returned by the service is entirely accurate.

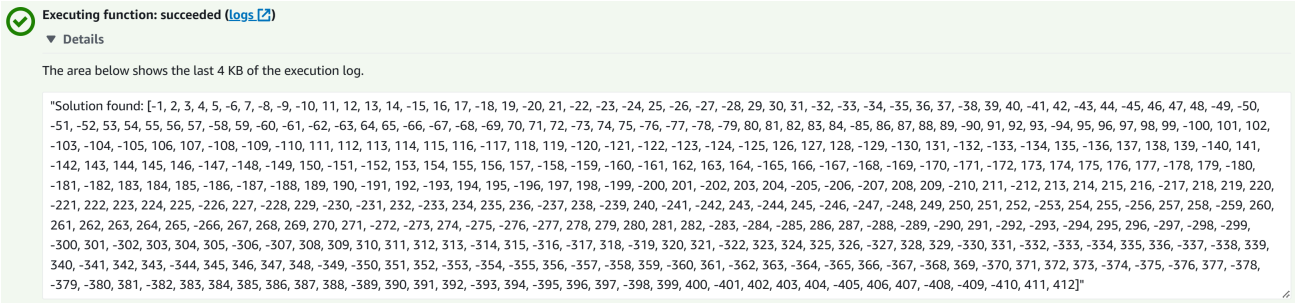


Figure 3.5: The returned results for the satisfiable knapsack problem.

If we modify the aforementioned knapsack problem by changing the value of k to 18, the problem will become unsatisfiable because there is no way to select items such that the total value of the chosen items exceeds or equals $k = 18$ while ensuring the total weight of the selected items remains no greater than $W = 14$. The circuit obtained by reducing this modified problem needs to replace the IntegerAsCircuit subcircuit that originally generated 17 with an IntegerAsCircuit subcircuit that generates 16. The answer returned by the service after inputting the JSON encoding of the corresponding circuit for this variant problem is as below.

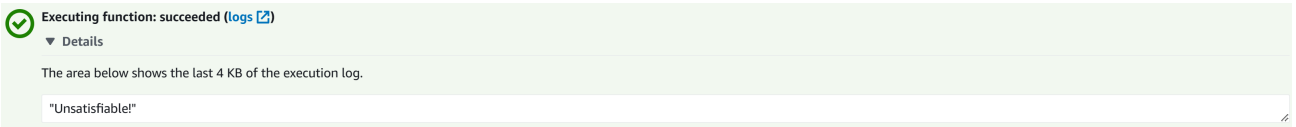


Figure 3.6: The returned results for the unsatisfiable knapsack problem.

The above test results further demonstrate that the service deployed on the AWS Lambda cloud is capable of constructing circuits based on the content of the provided JSON encoding and solving the related Circuit-SAT problems.

Chapter 4

Summary and Future Work

4.1 Summary of the Results

This thesis provides a comprehensive exploration of problems in the NP complexity class and proposes a novel serverless optimization service in the cloud based on boolean circuit satisfiability to address these problems.

The first chapter introduces the concept as well as the significance of the NP class in computational complexity, highlighting its relevance to various domains such as science, engineering, and economics. Chapter 1.1 provides a demonstration of the fundamental concepts related to the thesis topic, including the concepts of problems, algorithms, computational complexity classes, reduction etc.; which aims to lay the foundation for understanding the most relevant and intricate concepts pertaining to this thesis. In chapter 1.2, boolean circuit and Circuit-SAT problem is introduced, which is the core processing target of the proposed service. Additionally, chapter 1.3 presents examples of problems in NP to further bring forth the subsequent discussion on the idea and resolution of problems in NP.

In chapter 2, an optimization service grounded in boolean circuit satisfiability as a universal and efficient approach to solving problems in NP is primarily proposed. Chapter 2.1 elaborates general examples of problems in NP that this thesis aims to solve, discusses solvers specifically designed for several specific problems in NP, and addresses the need for a unified solution template for problems in NP class. Therefore, in chapter 2.2, the underlying solution concept of the service is proposed, along with the novel circuit incorporated in the proposed service of this thesis, emphasizing its fundamental role in enabling the service to address problems in NP with greater ease and efficiency. The application of the service

to Karp's 21 NP-Complete problems is presented in chapter 2.3 as a reference for the reduction of other problems in NP to the Circuit-SAT problem.

The third chapter provides a detailed explanation of the operational principles of the service, including the JSON format required by the circuits that can be parsed by the service (chapter 3.1.1), the circuit library that the service relies on (chapter 3.1.2), and the algorithmic logic involved in circuit construction (chapter 3.1.3). Additionally, chapter 3.1.4 provides a comprehensive, step-by-step guide on the steps of deploying the service on AWS Lambda, accompanied by the relevant concepts. In chapter 3.2, a specific knapsack problem is used as a case study for analysis and solution. Starting from reducing it to a Circuit-SAT problem, to the JSON representation of the circuit, and finally constructing the actual circuit and obtaining the answer, we have outlined the detailed process of using this service from start to finish.

4.2 Future Work

However, although this thesis proposes a service that offers a relatively universal and simpler solution to problems in NP, there is still significant room for improvement in this service. This mainly includes optimization of the algorithm during circuit construction to reduce its time and space complexity, and in the deployment service stage, auxiliary steps such as security certification could be added to ensure the security of the server.

In fact, as one of the key components of this paper, the circuit library, the diversity of sub-circuit functionalities it provides significantly influences the overall operations the final circuit can perform. If the diversity of sub-circuits within the circuit library is expanded, it might potentially make the reduction of other problems to the Circuit-SAT problem more convenient.

When considering improvements to the internal logic of circuit construction within the service, a viable approach is to store the entire circuit as a DAG within the service instead of using a simple list. By treating logical sub-circuits as individual nodes, the process can begin from the source nodes (corresponding to the input wires), and traverse the remaining nodes using the Breadth-First Search (BFS) algorithm. Since the order in which sub-circuits within a circuit are processed can be influenced by the input JSON file, the BFS algorithm prevents the issue encountered in the original algorithm, where sub-circuits that still lack completed input connections might be repeatedly traversed. For instance, when facing a sub-circuit like `LessEquals`, which requires a specific number of input wires before constructing, if its input sub-circuits have not yet been constructed, this sub-circuit, as well as its fellow-up sub-circuits, would be continually traversed in subsequent loops. In

contrast, the BFS algorithm in the DAG ensures that when traversing each node, at least one sub-circuit in its input direction has been constructed. If all the nodes outputting to it have already been built, the corresponding sub-circuit can be smoothly constructed. Otherwise, it remains in the queue awaiting subsequent iterations until its direct input connections are established.

During the deployment phase of the service, AWS provides the Identity and Access Management (IAM) service, allowing fine-grained control over resource access and identity authentication for servers [3]. In the case of deploying the service proposed in this thesis on AWS Lambda, IAM can be utilized to add different users and grant them corresponding access permissions. This ensures that users can interact with the server smoothly while safeguarding the confidentiality of the server's information as well as preventing mutual information leakage between users. Configuring AWS API Gateway is equally important in the deployment. It allows seamless integration with AWS Lambda, facilitating access to backend services by creating RESTful APIs to interact with resources in Lambda [2]. Additionally, API Gateway collaborates with IAM to provide identity authentication, ensuring the security of information for both the server and clients [2]. The configuration of API Gateway will offer future clients of this service a more convenient and flexible way of accessing it.

Besides, the service can be optimized with multi-threading capabilities to efficiently process requests, allowing clients to obtain processing results more effectively.

Overall, this thesis contributes to the advancement of concepts regarding computation complexity of the NP class. It introduces a serverless optimization service that relies on boolean circuit satisfiability along with an innovation of the functionality that a circuit can carry. The service is capable of parsing the JSON encoding of the circuit provided as input, enabling the construction of the circuit and subsequently determining whether the corresponding Circuit-SAT problem is satisfiable. Through innovative functional enhancements to the traditional circuit in a Circuit-SAT problem, this service significantly simplifies the complexity associated with reducing problems to this particular challenge, making it possible to universally and efficiently solve problems in NP.

References

[1] Introducing JSON. https://www.json.org/json-en.html. Accessed: August 16, 2023.

[2] AWS Gateway – Create, maintain, and secure APIs at any scale. https://aws.amazon.com/api-gateway/, September 30 2022. Accessed: July 07, 2023.

[3] AWS IAM – Securely management. https://aws.amazon.com/iam/, September 30 2022. Accessed: July 07, 2023.

[4] AWS Lambda – Serverless compute. https://docs.aws.amazon.com/lambda/index.html, September 30 2022. Accessed: July 07, 2023.

[5] David Applegate, Robert Bixby, Vašek Chvátal, William Cook, Daniel Espinoza, Marcos Goycoolea, and Keld Helsgaun. Certification of an optimal tsp tour through 85,900 cities. *Operations research letters*, 37(1):11–15, 2009.

[6] Stephen Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[7] Thomas Cormen. *Introduction to algorithms*. MIT Press, Cambridge, Mass, 3rd ed. edition, 2009.

[8] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[9] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[10] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. *Lecture notes in computer science*, 3569:61–75, 2005.

[11] Joseph Hellerstein, Jose Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. 2018.

[12] Pascal Van Hentenryck. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In *Lecture Notes in Computer Science*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248. Springer Berlin / Heidelberg, Germany, 2002.

[13] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for cnf formulas. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, pages 357–371, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[14] Nahid Juma, Jonathan Shahen, Khalid Bijon, and Mahesh Tripunitara. The overhead from combating side-channels in cloud systems using vm-scheduling. *IEEE transactions on dependable and secure computing*, 17(2):422–435, 2020.

[15] Richard Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008*, pages 219–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[16] Dexter Kozen. *Automata and computability*. Undergraduate Texts in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 1st ed. 1977. edition, 1977.

[17] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9710 of *Lecture Notes in Computer Science*, pages 123–140, Cham, 2016. Springer International Publishing.

[18] Jinkun Lin, Shaowei Cai, Chuan Luo, and Kaile Su. A reduction based method for coloring very large graphs. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, page 517–523. AAAI Press, 2017.

[19] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings - ACM IEEE Design Automation Conference*, DAC '01, pages 530–535. ACM, 2001.

[20] Ryan Rossi and Nesreen Ahmed. Coloring large complex networks. *Social network analysis and mining*, 4(1):1–37, 2014.

[21] Niklas Sörensson and Niklas Een. An extensible sat-solver. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, Berlin, Heidelberg, 2004. IT-fakulteten, Springer Berlin Heidelberg.

[22] Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *Digest of Technical Papers - IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '02, pages 442–449, Piscataway NJ, 2002. ACM.