VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Doan Thanh Son

SOLVING JOB SHOP SCHEDULING PROBLEM – JSSP BY SAT ENCODING

Major: Computer Science

HA NOI - 2024

VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Doan Thanh Son

SOLVING JOB SHOP SCHEDULING PROBLEM – JSSP BY SAT ENCODING

Major: Computer Science

Supervisor: PhD. To Van Khanh

HA NOI - 2024

Acknowledgements

I would like to express my gratitude to University of Engineering and Technology for providing me with the opportunity to do this research. I am also thankful to PhD. To Van Khanh for his dedicated support, encouragement, and supervision.

Finally, a deep thank to family, relatives and friends - who are always with me during the most difficult times, always encouraging me in life and at work. This accomplishment would not have been possible without them.

In the report, there may be some unintentional errors. I sincerely hope to receive valuable feedback and suggestions from lecturers, colleagues, and supervisors to improve it further. Sincerely thank.

Declaration

I declare that the report has been composed by myself and the work has not been submitted for any other degree or professional qualification. I confirm that the work submitted is my own, except where work which has formed part of jointly-authored publications has been included. My contribution to this work has been explicitly indicated below. I confirm that appropriate credit has been given within this report where reference has been made to the work of others.

I certify that, to the best of my knowledge, my report does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my report, published or otherwise, are fully acknowledged in accordance with the standard referencing practices.

I take full responsibility and take all prescribed disciplinary actions for my commitments.

Signature

Doan Thanh Son

Abstract

Abstract: One common scheduling problem is the Job-Shop Scheduling Problem (JSSP) which includes a number of jobs and machines. Each job consists of a sequence of tasks, which must be performed in a given order, and each task must be processed on a specific machine. Each machine can only perform one task at a time. The goal is to find an optimal schedule that minimizes the makespan (total time to complete all jobs without causing congestion). This type of scheduling problem has numerous applications in manufacturing, such as in automobile and aircraft production, where different parts need to be processed separately. This thesis tries to solve JSSPs by translating them into Boolean Satisfiability Problem (SAT). The approach includes the encoding method based on the one proposed by Crawford and Baker [6], the makespan's lower bound and upper bound estimating method, an algorithm for quickly searching for the optimal makespan based on the binary search algorithm, and a method for finding additional optimal schedules alongside existing schedules. The experiments are conducted on three JSSPs: FT06, La03, and Orb07. For each problem, the optimal makespan and multiple optimal schedules are found. The experiments use a common SAT solver named MiniSAT.

Keywords*: scheduling, SAT encoding, job-shop scheduling, MiniSAT*

Table of Contents

Acronyms

CNF	Conjunctive normal form
JSSP	Job-shop scheduling problem
LB	Lower bound
SAT	Boolean statisfiability
UB	Upper bound

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, the thesis will present the definition of job-shop scheduling problems (JSSPs), emphasizing their practical applications in real-world scenarios.

1.1 Job-shop scheduling problem

The job shop scheduling problem is a common scheduling problem. The name originally came from the scheduling of jobs in a job shop, but the theme has wide applications beyond that type of instance. This problem is one of the best known combinatorial optimization problems, and was the first problem for which competitive analysis was presented, by Graham in 1966 [12].

The inputs to such problems are a list of *jobs* and a list of *machines*. The required output is a *schedule*. Each job comprises a series of *tasks* (operations) that must be executed in a specific order, and each task requires processing on a designated machine. Additionally, each task takes a fixed amount of *processing time* to complete. The goal is to efficiently schedule these tasks on the available machines to minimize the *makespan* – the total length of the schedule (that is, when all the jobs have finished processing), adhering to the following constraints:

- **Constraint 1:** No task can commence until the preceding task for the same job is finished.
- **Constraint 2:** Each machine can only handle one task at any given time.
- **Constraint 3:** Once a task has begun, it must continue uninterrupted until completion.

Below is a simple example of a JSSP, in which each task is labeled by a pair of numbers (m, t) where m is the number of the machine the task must be processed on and t is the processing time of the task. The numbering of jobs and machines starts at 0.

- $job_0 = [(0, 2), (2, 1), (1, 4)]$
- $job_1 = [(0, 3), (1, 2), (2, 2)]$
- $job_2 = [(1, 4), (2, 3), (0, 5)]$

In this example problem, there are 3 jobs ($job_0,\ job_1,\ job_2$) and 3 machines (machine 0, machine 1, machine 2). Each job contains 3 tasks. The first task of job_0, $(0, 2)$, must be processed on machine 0 in 2 units of time. The second task, $(2, 1)$, must be processed on machine 2 in 1 unit of time. The last task of job_0, $(1, 4)$, must be processed on machine 1 in 4 unit of time. The remaining tasks of job_1 and job_2 operate similarly to the previous ones.

Table 1.1: A schedule satisfies the example problem

Job index	**Task index**	**Machine**	**Processing time**	**Start time**	**End time**
0	0	0	2	3	5
0	1	2	1	7	8
0	2	1	4	8	12
1	0	0	3	0	3
1	1	1	2	4	6
1	2	2	2	8	10
2	0	1	4	0	4
2	1	2	3	4	7
2	2	0	5	7	12

A solution to the job shop problem is a *schedule* which means an assignment of a start time for each task meeting the constraints given above. Table 1.1 shows a possible solution for the example problem. The first four columns of the table are from the input, the start time column is the output, and the end time column is calculated based on the existing columns. Each schedule can be represented in the form of a diagram. The schedule in Table 1.1 is illustrated in Figure 1.1. In this diagram, each task is represented by a rectangle with length equals to the task's *processing time* and a pair of numbers $(job\_index, task\_index)$ inside. Both job index and task index start from 0. For

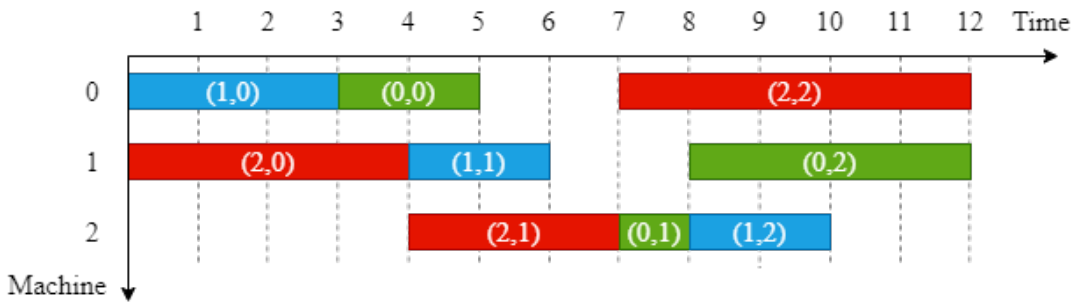

Figure 1.1: The diagram illustrates the schedule in Table 1.1

example, rectangle labeled $(2, 1)$ represents task 1 of job 2 which is executed on machine 2 and has processing time of 3 units. This task starts at time 4 and end at time 7.

Constraint 1: *No task can commence until the preceding task for the same job is finished.*

This constraint implies that within a job, tasks must be executed in the specified order provided in the input. A task can only begin once the preceding task of that job has been completed.

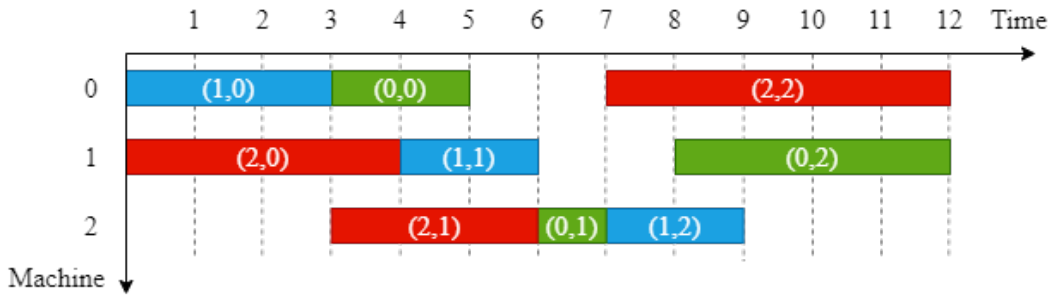

Figure 1.2: The schedule does not satisfy Constraint 1 (a)

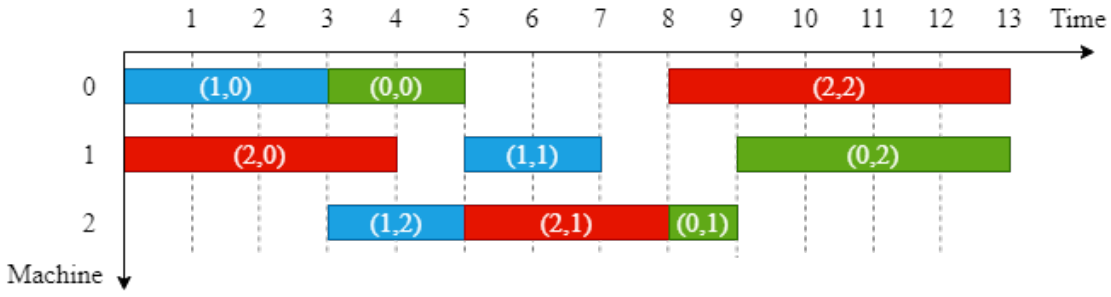

Figure 1.3: The schedule does not satisfy Constraint 1 (b)

Figure 1.2 shows a schedule which does not satisfy Constraint 1 because task 1 of

job 2 starts when task 0 of job 2 has not been completed.

Figure 1.3 shows a schedule which does not satisfy Constraint 1 because task 2 of job 1 starts and ends before task 1 of job 1 even starts. This means that these two tasks are not executed according to the order specified in the input.

Constraint 2: *Each machine can only handle one task at any given time.*

This constraint implies that two or more tasks cannot simultaneously run on a single machine.

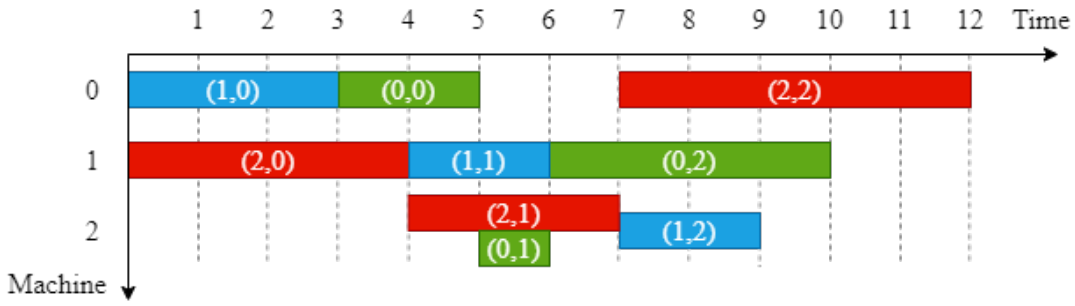

Figure 1.4: The schedule does not satisfy Constraint 2

Figure 1.4 shows a schedule which does not satisfy Constraint 2 because during the period from time 5 to time 6, both tasks 1 of job 2 and task 1 of job 0 are running, and both tasks require the use of machine 2.

Constraint 3: *Once a task has begun, it must continue uninterrupted until completion.*

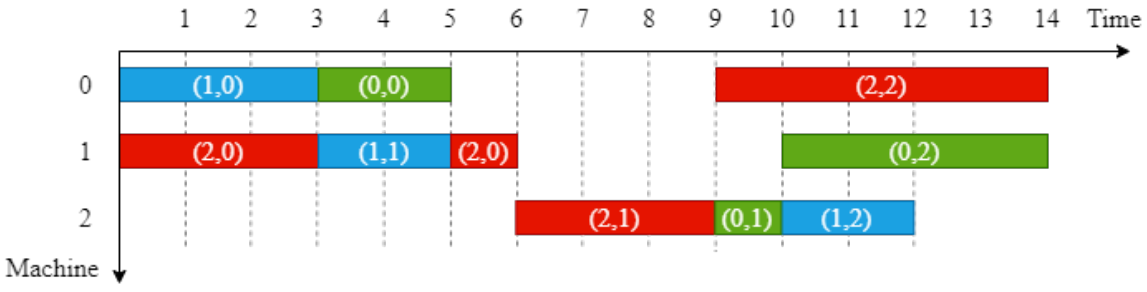

Figure 1.5: The schedule does not satisfy Constraint 3

Figure 1.5 shows a schedule which does not satisfy Constraint 3 because task 0 of job 2 is not executed continuously, but is interrupted to run task 1 of job 1 before resuming the remaining part.

The schedule in Figure 1.1 is a solution because it satifies all three constraints. The

tasks for each job are scheduled at non-overlapping time intervals, in the order given by the problem. Each machine only work on one task at a time and all tasks are not interrupted. The makespan of this solution is 12, which is the first time when all three jobs are complete. The objective of solving JSSPs is to minimize the makespan (find optimal makespan) while still satisfying the given constraints.

An instance of the JSSP can be modeled using a disjunctive graph [20]. Each task of a job is represented by a node in the graph, and all nodes within the disjunctive graph represent a potential candidate solution, in addition to two dummy nodes: S (source/starting vertex) and V (sink/terminating vertex). To determine the schedule, we must establish an ordering of all tasks processed on each machine. This can be achieved by transforming all undirected (disjunctive) edges into directed edges. In Figure 1.6, we illustrate the disjunctive graph of the example problem above.

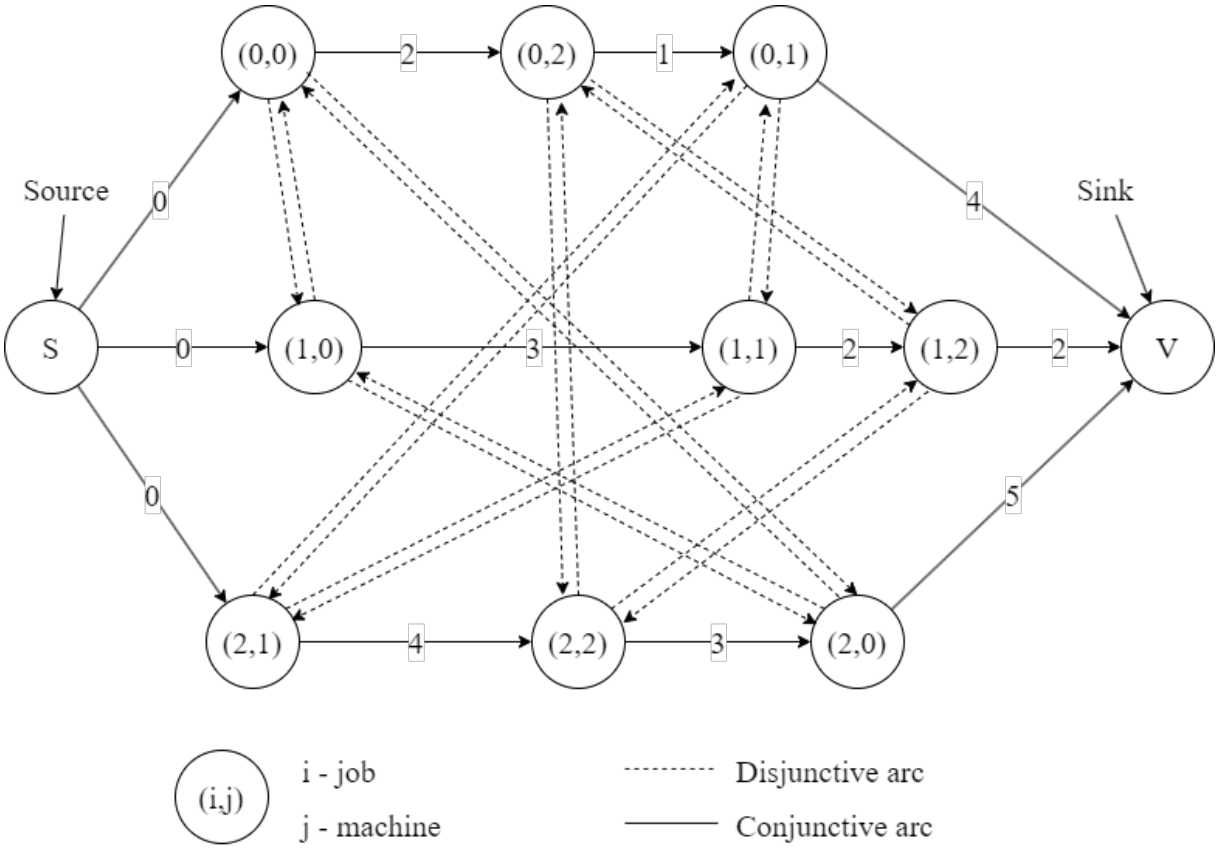

Figure 1.6: The disjunctive graph of the example problem

All jobs start at source S and finish at the sink V. There are eleven nodes all together, including two dummy nodes S and V, where node (0,0) is the processing of job 0 on machine 0, node (2,1) is the processing of job 2 on machine 1 and so forth. The label attached to an arc in Figure 1.6 represents processing time p, except the arcs going out

of the source have zero weights. For example, the arc from node (0,0) has weight $p = 2$, which represents that task 0 of job 0 has processing time equal to 2. The conjunctive arcs represent the given precedence between tasks in each job. For example, node (0,0), (0,2) and (0,1) is three tasks of job 0. Node (0,0) need to be executed before node (0,2), node (0,2) need to be executed before node (0,1). The disjunctive arcs represent the machine capacity constraints. For example, node (0,0), (1,0) and (2,0) are all executed on machine 0 and cannot be executed at the same time.

1.2 Application of job-shop scheduling

The JSSP is a common scheduling problem. This type of scheduling problem has been widely applied to address various real-world scheduling problems, such as in automobile and aircraft production, where different parts need to be manufactured separately using different machines. This section introduces an application JSSP in the green patient flow optimization problem tackled by M. Vali and his team [25]. Vali's study presents a flexible job shop scheduling problem to optimize patient flow and, thereby, minimize the total carbon footprint.

Optimizing patient flow is crucial for providing high-quality care in healthcare settings, especially hospitals. Improving patient flow not only benefits healthcare providers but also offers a way to refine healthcare services and enhance patient safety, outcomes, and satisfaction.

Table 1.2: The similarities between Green patient flow optimization and JSSP

Green patient flow optimization	**Job-shop scheduling**
patients	jobs
treatment operations	tasks
medical equipments	machines

The widespread use of complex equipment and technologies in various treatments, particularly in hospitals, consumes significant amounts of electricity and, consequently, can increase CO2 emissions. In addition, CO2 emissions from healthcare facilities in the world's largest economies contribute to approximately 4% of their national carbon footprints. Hospitals consume more energy per square meter of floor space than other non-residential buildings, largely due to their continuous operation. Given the focus on optimizing patient flow can reduce the operating time of equipment, thereby decreasing

carbon emissions.

The green patient flow management can be solved applying flexible job shop scheduling. A flexible job shop scheduling problem could be a generalized form of classical job shop scheduling where each operation can be executed on a qualified machine [19]. Table 1.2 show how to model the green patient flow problem as a job shop instance where patients resemble jobs, and treatment operations were seen as tasks of a specific job and medical equipments as machines.

Chapter 2

Related works and Background

2.1 Related works

The JSSP is an NP-hard problem that has been studied by numerous researchers. NP-hard means in the worst case, all known algorithms for solving such problems require run time exponential in the size of the problem. Various strategies have been proposed to tackle the problem, with research predominantly focusing on two avenues: the approximation method and the exact method.

Scholars have delved into a range of methods, utilizing heuristic and metaheuristic approaches to seek near-optimal solutions. Noteworthy strategies include the application of Genetic Algorithms, Estimation of Distribution Algorithms, Discrete Differential Evolution Algorithms, Linear Programming, Dynamic Programming, and Branch and Bound. These approaches aim to provide near-optimal schedules by efficiently navigating the intricate solution space of the JSSP. The selection of a particular method depends on factors such as problem size, constraints, and computational resources, reflecting ongoing efforts to refine and combine these techniques for improved approximations.

My focus centers on the exact solution, particularly addressing the Job Shop Scheduling Problem (JSSP) through the application of satisfiability (SAT) algorithms. Crawford and Baker [6] conducted series of experiments applying SAT algorithms to scheduling problems. Koshimura and his team [17] utilized the encoding method introduced by Crawford and Baker to prove optimal upper bounds of ABZ9 (678) and YN1 (884) are indeed optimal. They also improved the upper bound of YN2 nad lower bounds of ABZ8, YN2, YN3 and YN4. Hong Huang and Shaohua Zhou [15] optimized the SAT encoding method, thus reducing the number of clauses and their processing effi-

ciency in the solver, experimented on problems Ft20 and ABZ8. Popov [11] extended the application of SAT algorithms to solve task-resource scheduling problems in cloud computing systems, while Cruz-Chávez and Rivera-Lopez [7] applied a local search algorithm to the logical representation of the JSSP. Hamadi and Jabbour [13] proposed a novel clause learning method crucial for modern SAT solvers. Additionally, Micheli and Mishchenko [21] extended the SAT formulation to find a minimum-size network under delay constraints.

In this report, I propose the process to solve a problem using SAT encoding with the MiniSAT solver. My approach includes determining the lower bound and upper bound of the problem, an algorithm to quickly find the optimal makespan based on binary search, and a method to find different schedules that satisfy the same makespan.

2.2 SAT encoding and optimization techniques

2.2.1 Propositional logic

A proposition logic formula, also called boolean expression, is build from variables, operators, and parentheses. A boolean variable x can take one of two possible truth value 0 ($False$) or 1 ($True$).

A truth table is a tabular representation of all possible combinations of truth values for a logical expression. It systematically shows the outcomes of a logical operation or a set of propositions under different truth conditions. Each row in the table corresponds to a unique combination of truth values for the variables or propositions involved in the expression.

There are different operators:

- **Negation**: is a unary operator, is commonly referred to as NOT, often denoted by $\neg$. It negates or flips the value of the variables. All possible combinations of input variables are described in the Table 2.1.

Table 2.1: Truth table of negation operator

x	$\neg x$
1	0
0	1

- **Conjunction**: denoted by $\wedge$, commonly referred to as AND, is a binary operator between two variables that is only $True$ when both variables are $True$. All possible combinations of input variables are described in the Table 2.2.

Table 2.2: Truth table of conjunction operators

x	y	$x \wedge y$
1	1	1
1	0	0
0	1	0
0	0	0

- **Disjunction**: commonly referred to as OR, is a binary operator between two variables that is $True$ when at least one of the variables is $True$, denoted by $\vee$. All possible combinations of input variables are described in the Table 2.3.

Table 2.3: Truth table of disjunction operators

x	y	$x \vee y$
1	1	1
1	0	1
0	1	1
0	0	0

- **Exclusive-or**: commonly referred to as XOR, is a binary operator between two variables that is $True$ only when exactly one of the variables is $True$, denoted by $\oplus$. All possible combinations of input variables are described in the Table 2.4.

Table 2.4: Truth table of exclusive-or operators

x	y	$x \oplus y$
1	1	0
1	0	1
0	1	1
0	0	0

- **Implication**: The implication operator, commonly referred to as "if -then", written as $\rightarrow$, is a binary operator between two variables. It is $True$, unless the first variable is $True$, while the second variable is $False$. All possible combinations of input variables are described in the Table 2.5.

Table 2.5: Truth table of implication operators

x	y	$x \rightarrow y$
1	1	1
1	0	0
0	1	1
0	0	1

- **Biconditional**: The equivalence or "if and only if" operator, denoted by $\leftrightarrow$, is a binary operator between two variables that is $True$, only when both variables have the same value — either both $True$, or both $False$. All possible combinations of input variables are described in the Table 2.6.

Table 2.6: Truth table of biconditional operators

x	y	$x \leftrightarrow y$
1	1	1
1	0	0
0	1	0
0	0	1

2.2.2 Conjunctive normal form

A literal might be interpreted as the negation of a variable (negative literal) or as a variable (positive literal). A disjunction of literals or a single literal is called a clause. If a formula consists of a single clause or a conjunction of clauses, it is in conjunctive normal form (CNF). For example, x_1 is a positive literal, $\neg x_2$ is a negative literal, $(x_1 \vee \neg x_2)$ is a clause, the Formula 2.1 is in CNF.

$$(x_1 \vee \neg x2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge \neg x1 \tag{2.1}$$

Every propositional logic formula can be converted into an equivalent CNF using

the laws for Boolean Algebra; this form, however, may be exponentially longer. The basic laws of the Boolean Algebra are added in Table 2.7.

Table 2.7: Basic laws of Boolean Algebra

Law	**OR form**	**AND form**
Identity Law	$x \vee 0 = x$	$x \wedge 1 = x$
Idempotent Law	$x \vee x = x$	$x \wedge x = x$
Commutative Law	$x \vee y = y \vee x$	$x \wedge y = y \wedge x$
Associative Law	$x \vee (y \vee z) = (x \vee y) \vee z$	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
Distributive Law	$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
Inversion Law	$\neg(\neg x) = x$	$\neg(\neg x) = x$
De Morgan's Law	$\neg(x \vee y) = (\neg x) \wedge (\neg y)$	$\neg(x \wedge y) = (\neg x) \vee (\neg y)$

2.2.3 SAT problem

In context of computer science, the Boolean statisfiability problem (SAT) is the problem of deciding if there exists a truth assignment that satisfies a boolean formula.

INPUT: a boolean formula represented in CNF form.

OUTPUT:

- **SAT**: if there exists an assignment of values ($True$ or $False$) to its variables that makes the formula evaluate to $True$.
- **UNSAT**: if there exists no assignment of values ($True$ or $False$) to its variables that makes the formula evaluate to $True$.

Given a boolean formula in CNF form, SAT asks whether the variables can be replaced by the values $True$ or $False$ in order to make the formula evaluate to $True$. If no such assignment exists - the formula is $False$ for all possible variable assignments, we call it unsatisfiable. Otherwise, the formular is called satisfiable. For example, the formular $a \wedge \neg b$ is satisfiable because we can find the values $a = True$ and $b = False$ which make $a \wedge \neg b = True$. In constrast, $a \wedge \neg a$ is unsatisfiable because it is always $False$ whether $a = True$ or $a = False$. The Formula 2.1 is satisfiable, by choosing $x1 = False$, $x2 = False$ and $x3$ arbitrarily.

2.2.4 SAT encoding

Definition: *SAT Encoding is a method of transforming practical problems into propositional logic statements and applying SAT Solver to solve those.*

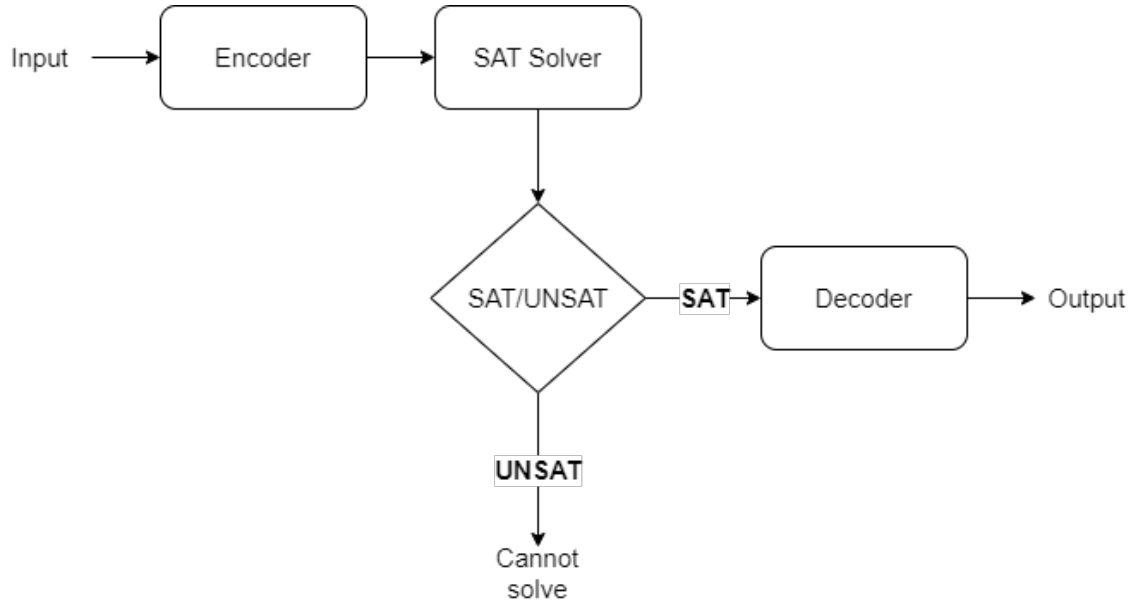

Figure 2.1: SAT execution flow chart

Most practical problems that can be solved using SAT encoding has the solving process following the flow depicted in Figure 2.1:

- ***Input***: The input to the flow is a text representation of the problem. The JSSPs are often represented in the form of standard specification or Taillard specification (Section 4.2.2).
- ***Encoder***: Encodes all the constraints of input to CNF. Encoder will generate a cnf file then pass it to the SAT Solver.
- ***SAT Solver***: From the cnf file, calculate the truth value assignment of all the propositional variables. If there does not exist a assignment of propositional variable values that satisfy the input formula, then return UNSAT; otherwise, return SAT along with those values passed to the Decoder.
- ***Decoder***: Reverses the encoding process, produce the problem output from the assignment of propositional variables.
- ***Output***: The output of the problem. For JSSPs, the output is an assignment of a start time for each task, which meets the problem constraints.

2.2.5 Applications of SAT encoding

Apart from its application to job shop scheduling problems, as presented in this thesis, SAT encoding is also employed to solve various other practical problems such as the Stable Marriage Problem [10] or a mathematical puzzle called *8-Queens Puzzle* [16]. In this section, the report describes the application of SAT encoding in solving the Course Timetabling Problem [4].

In 2012, Asín Achá and Nieuwenhuis introduce novel and robust techniques based on SAT and MaxSAT for handling the Curriculum-based Course Timetabling problem [4]. The MaxSAT problem is an optimization variant of the SAT problem, where the objective is to identify a model that maximizes the number of satisfied clauses. The organization and report on the results of the First and Second Max-SAT Evaluations was described in [3].

The Curriculum-based Course Timetabling Problem was introduced for track 3 of the 2nd International Timetabling Competition (ITC) in 2007, and its comprehensive description can be found in [8]. In brief, this problem involves the following elements:

- ***Courses***: A course consists of a specified number of lectures on a particular subject. Each course is associated with a teacher, a certain number of students, and a minimum number of days over which the lectures can be distributed. For instance, there could be a course c_1 with 50 students, taught by teacher t_1, comprising 5 lectures that should be spread across at least 3 different days of the week.

- ***Curriculums***: A curriculum is a set of courses that may share students. For example, curriculum k_1 might encompass four courses: c_1, c_2, c_3, and c_4.

- ***Rooms***: Courses take place in designated rooms, each having an assigned capacity (number of students, typically ranging from 20 to 1000).

- ***Days and hours***: Courses are scheduled weekly on specific days and hours, with each day having a predefined number of lecture hours during which courses can occur.

The primary objective is to devise an efficient assignment of courses to rooms and hours while adhering to various *hard* (necessary) and *soft* (desirable) constraints.

Hard Constraints in Curriculum-based Course Timetabling problem:

- Curriculum clashes: Courses belonging to the same curriculum must not be scheduled simultaneously.
- Teacher clashes: No two courses taught by the same teacher should overlap in time.
- Room clashes: A room cannot be occupied by multiple courses at the same time.
- Hour availability: Scheduling must account for hours when teachers are unavailable.
- Number of lectures: The exact number of lectures specified for each course should be scheduled.

Soft Constraints in Curriculum-based Course Timetabling problem:

- Room capacity: Courses should be scheduled in rooms with adequate capacity; a cost of 1 per student is incurred for each violation.
- Min working days: Courses should be scheduled over a specified minimum number of days, with a cost of 5 for each day under the minimum.
- Isolated lectures: Lectures within a curriculum should be scheduled adjacent to other lectures in the same curriculum; a violation incurs a cost of 2.
- Room stability: All lectures of a course should be scheduled in the same room, with a cost of 1 for each additional room required for a course.

Encoding variables in Curriculum-based Course Timetabling problem

Based on the aforementioned constraints, we define the following propositional variables:

- $ch_{c,h}$: course c occurs in hour h
- $cd_{c,d}$: course c occurs in day d
- $cr_{c,r}$: course c occurs in room r
- $kh_{k,h}$: curriculum k occurs in hour h

Encoding rules in Curriculum-based Course Timetabling problem

If a course is scheduled during hour h, it must also be scheduled on the corresponding day of that hour. We add the following clause:

$$\neg ch_{c,h} \vee cd_{c,day(h)}$$

If a course is scheduled on a day d, it must be assigned to at least one of the hours $(h_1, h_2, ..., h_n)$ within that day. We include the following clause:

$$\neg cd_{c,d} \vee ch_{c,h_1} \vee ... \vee ch_{c,h_n}$$

If a course is scheduled during hour h, all curricula $(k_1, k_2, ..., k_n)$ to which course c belongs must also be scheduled during hour h. We add the following clauses:

$$\neg ch_{c,h} \vee kh_{k_1,h}$$

$$\neg ch_{c,h} \vee kh_{k_2,h}$$

$$...$$

$$\neg ch_{c,h} \vee kh_{k_n,h}$$

If a curriculum k is scheduled during hour h, at least one course belonging to that curriculum $(c_1, c_2, ..., c_n)$ must also be scheduled within the same hour. We add the following clause:

$$\neg kh_{k,h} \vee ch_{c_1,h} \vee ... \vee ch_{c_n,h}$$

Curriculum clashes: No two courses c and c' belonging to the same curriculum may be scheduled at the same hour. We add the following clause :

$$\neg ch_{c,h} \vee \neg ch_{c',h}$$

Teacher clashes: No two courses c and c' with the same teacher may be scheduled to the same hour. We add the following clause:

$$\neg ch_{c,h} \vee \neg ch_{c',h}$$

Room clashes: No two courses c and c' may be scheduled to the same room at the same hour. We add the following clause:

$$\neg ch_{c,h} \vee \neg ch_{c',h} \vee \neg cr_{c,r} \vee \neg cr_{c',r}$$

Hour availability: For each course c with forbidden hours $h_1, h_2, ..., h_n$, we add the following clause:

$$\neg ch_{c,h_1}$$

$$\neg ch_{c,h_2}$$

$$...$$

$$\neg ch_{c,h_n}$$

Number of lectures: For each course c exactly $hours(c)$ literals of the following set $\{ch_{c,h_1}, ch_{c,h_2}, ..., ch_{c,h_n}\}$ must be true. This means the total number of elements in the set is equal to the number of hours allocated for the course. We add the following clause:

$$exactly(hours(c), \{ch_{c,h_1}, ch_{c,h_2}, ..., ch_{c,h_n}\})$$

Room capacity: For each room with capacity smaller than the course's number of students, we add the following clause:

$$\neg cr_{c,r}$$

Min working days: For each course c, at least $working\_days(c)$ literals of the set $\{cd_{c,d_1}, cd_{c,d_2}, ..., cd_{c,d_n}\}$ must be true. This means the total number of elements in the set is not lower than min working days. We add the following clause:

$$at\_least(working\_days(c), \{cd_{c,d_1}, cd_{c,d_2}, ..., cd_{c,d_n}\})$$

Isolated lectures: If curriculum k occurs in hour h, then k should also occur in an hour before or after in the same day. We add the following clause:

- If h is the first hour of a day:

$$\neg kh_{k,h} \vee kh_{k,h+1}$$

- If h is the last hour of a day:

$$\neg kh_{k,h} \vee kh_{k,h-1}$$

- If h is not the first nor the last of a day:

$$\neg kh_{k,h} \vee kh_{k,h-1} \vee kh_{k,h+1}$$

Room stability: Each course must be scheduled to exactly one room. So, the total number of elements in the set $\{cr_{c,r_1}, cr_{c,r_2}, ..., cr_{c,r_n}\}$ should be equals to 1. We add the following clause:

$$exactly(1, \{cr_{c,r_1}, cr_{c,r_2}, ..., cr_{c,r_n}\})$$

Chapter 3

Materials and Methods

3.1 Makespan, upper bound, lower bound

Under all constraints, the time required to complete all the jobs is called as the makespan L. Our method does not directly calculate the makespan but will instead assign a trial value to the makespan and then try to solve the problem with that assigned makespan.

If the problem can be solved with a certain value of makespan, we call that makespan value satisfiable. If the makespan $L = l$ is satisfiable, all the bigger makespan ($l + 1, l + 2, ...$) is also satisfiable. This can be proven by assuming that the problem output has been found with a makespan $L = l$. By adjusting the start time of the final task in the output and increasing it by 1, we obtain a new output that satisfies the problem with a makespan $L = l + 1$. Similarly, it can be proven that all makespans greater than l are satisfiable. Therefore, we call this value l the upper bound (UB) for the problem because the optimal result is definitely not larger than l. If any value lower than l is satisfiable, it can be a new UB instead of l.

If the makespan $L = l$ is not satisfiable, all jobs cannot be completed in time l. Therefore, in a shorter period of time, it is even more impossible to complete all jobs. All the lower makespan ($l - 1, l - 2, ...$) is also unsatisfiable. This value l can be called the problem lower bound (LB) because the optimal result is definitely bigger than l. If we find a value bigger than l which is also unsatisfiable, it can be a new LB instead.

Let S_L be a SAT problem generated by our encoding method under assumption that the makespan equals L. If we find a positive integer k such that S_{k-1} is unsatisfiable and S_k is satisfiable, the we conclude that the minimum makespan is k. Such a k divides

SAT problems S_i $(i \geq 1)$ into an unsatisfiable part S_i $(1 \leq i \leq k-1)$ and a satisfiable part S_i $(i \geq k)$.

For common JSSPs, most UB and LB have been established in various studies. When we know the UB, LB of the problem, we only need to find the optimal makespan within the range [LB, UB]. In cases of entirely new problems, we can only estimate the value of LB and UB.

3.2 Encoding method

The JSSP is represented in CNF and translated into a SAT problem by the encoding method based on the approach proposed by Crawford and Baker [6]. This section introduces the encoding method and provides an illustrative example of encoding for the simple problem introduced in Section 1.1. In this section, we call that problem the *sample* problem.

A JSSP consists of a set of n jobs $\{J_0, J_1, ..., J_{n-1}\}$ and a set of m machines $\{M_0, M_1, ..., M_{m-1}\}$. Each job J_l is a sequence of tasks $(O_0^l, O_1^l, ..., O_{q_l-1}^l)$ with q_l is the number of tasks in job J_l. Each task O_i^l is performed on machine $M_{O_i^l}$ for an uninterupted duration p_i^l - processing time. Machine $M_{O_i^l}$ belongs to set of m machines above. A *schedule* or an *output* is is a set of start times for each task O_i^l. The time required to finish all the jobs is called the *makespan* L. As previously introduced, we assign a value to the makespan for calculation.

There are three kinds of propositional variables:

- $pr_{i,j}^{l,k}$ stating that task O_i^l precedes O_j^k.
- $sa_{i,t}^l$ stating that task O_i^l starts at t or after.
- $eb_{i,t}^l$ stating that task O_i^l ends by t or before.

Scheduling constraints are translated by the following rules. Each rule represents a type of clause in our CNF encoding. The key objective here is the identification of the truth values of the propositional variables that satisfy all these rules, which means satisfying the conditions of the problem.

1. O_i^l precedes O_{i+1}^l:

$$pr_{i,i+1}^{l,l} \ (0 \leq l \leq n-1, \ 0 \leq i \leq q_l - 2)$$

This rule corresponds to the constraint that no task can begin until the previous task for the same job is completed. For the sample problem, we have the following clauses:

$pr_{0,1}^{0,0}$

$pr_{1,2}^{0,0}$

$pr_{0,1}^{1,1}$

$pr_{1,2}^{1,1}$

$pr_{0,1}^{2,2}$

$pr_{1,2}^{2,2}$

2. If O_i^l and O_j^k require the same machines ($M_{O_i^l} = M_{O_j^k}$), then O_i^l precedes O_j^k or O_j^k precedes O_i^l:

$$pr_{i,j}^{l,k} \vee pr_{j,i}^{k,l} \ (0 \leq l < k \leq n-1,\ 0 \leq i \leq q_l - 1,\ 0 \leq j \leq q_k - 1)$$

This rule corresponds to the constraint that each machine can only handle one task at any given time. For the sample problem, we have the following clauses:

$pr_{0,0}^{0,1} \vee pr_{0,0}^{1,0}$

$pr_{0,2}^{0,2} \vee pr_{2,0}^{2,0}$

$pr_{0,2}^{1,2} \vee pr_{2,0}^{2,1}$

...

$pr_{1,0}^{1,2} \vee pr_{0,1}^{2,1}$

3. Task O_i^l can only start after all previous task $\{O_0^l, O_1^l, ..., O_{i-1}^l\}$ of the same job J_l end. It requires at least $t = \sum_{u=0}^{i-1} p_u^l$. That is, O_i^l starts at time t or after:

$$sa_{i,t}^l \ (0 \leq l \leq n-1,\ 0 \leq i \leq q_l - 1,\ t = \sum_{u=0}^{i-1} p_u^l)$$

For the sample problem, we have the following clauses:

$sa_{0,0}^0$

$sa_{1,2}^0$

$sa_{2,3}^0$

...

$sa_{2,7}^2$

4. Conversely, to complete all the jobs by L, task O_i^l must end by time $t = L - \sum_{u=i+1}^{q_l-1} p_u^l$. The duration $\sum_{u=i+1}^{q_l-1} p_u^l$ is the minimum time required to complete all

later tasks $\{O^l_{i+1}, O^l_{i+2}, ..., O^l_{q_l-1}\}$ of the same job J_l.

$$eb^l_{i,t}\ (0 \leq l \leq n-1,\ 0 \leq i \leq q_l - 1,\ t = L - \sum_{u=i+1}^{q_l-1} p^l_u)$$

For the sample problem, we have the following clauses:

$eb^0_{0,L-5}$
$eb^0_{1,L-4}$
$eb^0_{2,L}$
...
$eb^2_{2,L}$

5. If O^l_i starts at t or after is True, it starts at $t-1$ or after is also True:

$$sa^l_{i,t} \rightarrow sa^l_{i,t-1}\ (0 \leq l \leq n-1,\ 0 \leq i \leq q_l - 1,\ 1 \leq t \leq L)$$

For the sample problem, we have the following clauses:

$sa^0_{0,1} \rightarrow sa^0_{0,0}$
$sa^0_{0,2} \rightarrow sa^0_{0,1}$
$sa^0_{0,3} \rightarrow sa^0_{0,2}$
...
$sa^0_{0,L} \rightarrow sa^0_{0,L-1}$
$sa^0_{1,1} \rightarrow sa^0_{1,0}$
$sa^0_{1,2} \rightarrow sa^0_{1,1}$
...
$sa^2_{2,L} \rightarrow sa^2_{2,L-1}$

6. If O^l_i ends by t or before is True, it ends by $t+1$ or before is also True:

$$eb^l_{i,t} \rightarrow eb^l_{i,t+1}\ (0 \leq l \leq n-1,\ 0 \leq i \leq q_l - 1,\ 0 \leq t \leq L-1)$$

For the sample problem, we have the following clauses:

$eb^0_{0,0} \rightarrow eb^0_{0,1}$
$eb^0_{0,1} \rightarrow eb^0_{0,2}$
...
$eb^0_{0,L-1} \rightarrow eb^0_{0,L}$
$eb^0_{1,0} \rightarrow eb^0_{1,1}$
...
$eb^2_{2,L-1} \rightarrow eb^2_{2,L}$

7. If O_i^l starts at t or after, it must end at $t + p_i^l$ or after. Therefore, it cannot end by $t + p_i^l - 1$ or before:

$$sa_{i,t}^l \rightarrow \neg eb_{i,t+p_i^l-1}^l \ (0 \leq l \leq n-1,\ 0 \leq i \leq q_l - 1,\ 0 \leq t \leq L - p_i^l + 1)$$

For the sample problem, we have the following clauses:
$sa_{0,0}^0 \rightarrow \neg eb_{0,1}^0$
$sa_{0,1}^0 \rightarrow \neg eb_{0,2}^0$
...
$sa_{2,L-4}^2 \rightarrow \neg eb_{2,L}^2$

8. If O_i^l start at t or after and O_i^l precedes O_j^k, O_j^k cannot start until O_i^l is finished. This means O_j^k can only start at $t + p_i^l$ or after. For each $pr_{i,j}^{l,k}$ from rule 1 and rule 2, we add the clause:

$$sa_{i,t}^l \wedge pr_{i,j}^{l,k} \rightarrow sa_{j,t+p_i^l}^k$$

$$(0 \leq l \leq n-1,\ 0 \leq k \leq n-1,\ 0 \leq i \leq q_l - 1,\ 0 \leq j \leq q_k - 1,\ 0 \leq t \leq L - p_i^l)$$

For the sample problem, we have the following clauses:
$sa_{0,0}^0 \wedge pr_{0,1}^{0,0} \rightarrow sa_{1,2}^0$
$sa_{0,0}^0 \wedge pr_{0,0}^{0,1} \rightarrow sa_{0,2}^1$
...
$sa_{2,L-5}^2 \wedge pr_{2,0}^{2,1} \rightarrow sa_{0,L}^1$

3.3 SAT solver

3.3.1 Definition of SAT solver

A SAT solver is an algorithm for establishing satisfiability. It takes the Boolean logic formula as input and returns SAT if it finds a combination of variables that can satisfy it or UNSAT if it can demonstrate that no such combination exists. In addition, it may sometimes return without an answer if it cannot determine whether the problem is SAT or UNSAT.

3.3.2 MiniSAT solver

This research uses MiniSAT solver developed by Niklas Sörensson and Niklas Een . It is a minimalistic, open-source SAT solver, developed for both researchers and developers; it is released under the "MIT license". Information about MiniSAT versions,

development history, and installation files can be found on the website hosted by MiniSAT's two authors [23]. The MiniSAT version used in this research is v1.13 [22].

MiniSAT introduces significant advancements in SAT-solving capabilities. The solver incorporates a resolution-based conflict clause minimization technique, rooted in self-subsuming resolution. Experiments demonstrate that this innovation can eliminate over 30% of redundant literals in conflict clauses, leading to reduced memory consumption and the generation of stronger clauses. Noteworthy improvements include an enhanced Variable State Independent Decaying Sum (VSIDS) decision heuristic, where variable activities decay by 5% after each conflict, outperforming the original VSIDS. The solver also introduces features such as binary clause implementation, aggressive clause deletion, and incremental SAT interface. The utilization of a two-literal watch scheme and conflict analysis further enhances the solver's efficiency. MiniSAT v1.13's commitment to minimizing learned clauses and optimizing decision heuristics positions it as a competitive and efficient SAT solver, particularly demonstrated by its participation in the SAT Competition 2005.

3.3.3 MiniSAT input format

The MiniSAT solver, similar to most SAT solvers, processes input provided in a simplified "DIMACS CNF" format, which is a simple text format and mostly-line-oriented format. Comment lines are identified by the letter "c" at the beginning, and they do not affect the input interpretation. The initial non-comment line serves to initiate the input and includes essential information such as the count of variables and clauses involved in the problem. This specific line is required to follow the structure:

```
p cnf [number of variables] [number of clauses]
```

The next non-comment lines define clauses. Every clause consists of a space-delimited sequence of 1-based variable indices. A positive value denotes a specific variable, while a negative value signifies the negation of a variable. The end of a clause is indicated by a final value of "0". The input file concludes once the last clause has been terminated.

Consider an example of CNF is:

$$(x_1 \vee \neg x_5 \vee x_4) \wedge (\neg x_1 \vee x_5 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_5)$$

The CNF expressions above would be written as MiniSAT input format:

```
c Here is a comment
```

```
p cnf 5 4
1 -5 4 0
-1 5 3 4 0
-3 4 0
-2 5 0
```

The "p cnf" line means that this is SAT problem in CNF format with 5 variables and 4 clauses. The first line after it is the first clause, meaning $(x_1 \vee \neg x_5 \vee x_4)$. The number 1 means x_1 and number -5 means $\neg x_5$. The solver's job is to find the set of boolean variable assignments that make all the clauses true.

3.3.4 MiniSAT output format

Upon execution, miniSAT provides several statistics related to its performance on standard error. Depending on whether a solution exists for the given problem, miniSAT will output either "SATISFIABLE" or "UNSATISFIABLE" (without quotation marks) to standard output. If a RESULT-OUTPUT-FILE is specified, miniSAT will write the results to that file. The first line will indicate whether the problem is satisfiable ("SAT") or unsatisfiable ("UNSAT"). In case of a satisfiable problem, the second line will contain an assignment of boolean variables that satisfies the given expression. It's important to note that while there might be multiple solutions, miniSAT is only required to provide one valid assignment.

For the provided example input from Section 3.3.3, the RESULT-OUTPUT-FILE will contain the following output:

```
SAT
1 2 -3 4 5 0
```

This output means that the example problem is satisfiable, with $x_1 = T,\ x_2 = T,\ x_3 = F,\ x_4 = T,\ x_5 = T$ (where T is True and F is False).

3.3.5 Get more solutions

In case of wanting to obtain an alternative solution, one straightforward approach is to create new input includes a new clause that negates the previous solution. Note that the count of clauses has increased, the count of variables is the same. In the context of our example above, we have a previous solution as:

```
1 2 -3 4 5 0
```

So the negation of previous solution is:

```
-1 -2 3 -4 -5 0
```

Appending the negation to the end of input file and increasing the count of clauses, we have the new input as follows:

```
c Here is a comment
p cnf 5 5
1 -5 4 0
-1 5 3 4 0
-3 4 0
-2 5 0
-1 -2 3 -4 -5 0
```

After putting this second input to MiniSat solver, the new output is:

```
SAT
1 -2 -3 4 5 0
```

The new solution $x_1 = T,\ x_2 = F,\ x_3 = F,\ x_4 = T,\ x_5 = T$ also satisfies the problem.

3.3.6 Generate cnf file after encoding

The clauses from Encoding step (Section 3.2) need to be translated to MiniSAT input format before put to the solver. This step can be done by using *bidirectional map* data structure. In computer science, a bidirectional map is an associative data structure in which the $(key,\ value)$ pairs from a one-to-one correspondence. Thus the binary relation is functional in each direction: each $value$ can also be mapped to a unique key. A pair $x,\ y$ thus provides a unique coupling between x and y so that y can be found when x is used as a key and x can be found when y is used as a key. Figure 3.1 describe a bidirectional map with some pairs of $(x,\ y)$.

The propositional variables is mapped to continuous positive numbers called *variable index* based on their order of appearance, starting from the number 1. This mapping allows the solver to distinguish between variables. Bidirectional map is used instead of

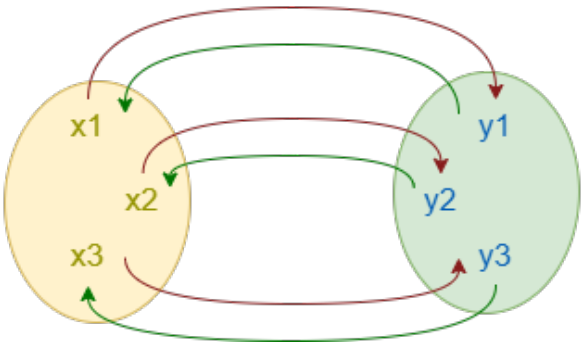

Figure 3.1: Bidirectional map for pairs of $(x,\ y)$

standard map because the decoding step needs to find variables when use *variable index* as a key.

Mapping for the sample problem in case of $L = 12$ is as follows:

$$pr_{0,1}^{0,0} \leftrightarrow 1$$

$$pr_{1,2}^{0,0} \leftrightarrow 2$$

...

$$sa_{0,1}^{0} \leftrightarrow 27$$

$$sa_{0,2}^{0} \leftrightarrow 28$$

...

$$eb_{0,0}^{0} \leftrightarrow 39$$

$$eb_{0,1}^{0} \leftrightarrow 40$$

...

$$eb_{2,11}^{2} \leftrightarrow 258$$

Each clause is transformed into a line in the solver input file using rules similar to those in Table 3.1. Each line is end with a number 0.

3.4 Decoding method

If the solver step results in satisfiable (SAT), we need a decoding step to convert the solver's output into the problem's output. The problem's output includes the start time for each task. From the Minisat output and bidirectional map, we found the truth

Table 3.1: Example of transformed clause for the sample problem in case of $L = 12$

Clause	**Line**
$pr_{0,1}^{0,0}$	1 0
$pr_{0,0}^{0,1} \vee pr_{0,0}^{1,0}$	7 8 0
$sa_{0,1}^{0} \rightarrow sa_{0,0}^{0}$	-27 25 0
$sa_{0,0}^{0} \rightarrow \neg eb_{0,1}^{0}$	-25 -40 0
$sa_{0,0}^{0} \wedge pr_{0,1}^{0,0} \rightarrow sa_{1,2}^{0}$	-25 -1 51 0

assignment for propositional variables. Call this truth assignment M. The start time s_i^l of operation O_i^l is given by extracting the variable $sa_{i,t}^l$ which has the biggest time and is still assigned as true in M . More precisely, s_i^l is given by which satisfies the following expression:

$$s_i^l \; = \; max \; of \; T_i^l \; where \; T_i^l \; = \; \{t \mid sa_{i,t}^l \; = \; True\}$$

For the sample problem with $L = 12$, the start after variables of task O_0^0 has the truth assignment as in Table 3.2. So the start time for this task is 3.

Table 3.2: Some truth assignment of the sample problem with $L = 12$

Variable	Truth assignment
$sa_{0,0}^{0}$	True
$sa_{0,1}^{0}$	True
$sa_{0,2}^{0}$	True
$sa_{0,3}^{0}$	True
$sa_{0,4}^{0}$	False
$sa_{0,5}^{0}$	False
...	...
$sa_{0,12}^{0}$	False

3.5 Estimate lower bound and upper bound

This section introduces my method for estimating LB and UB values for an entirely new problem that has not had these bounds determined in previous research. The aim is to estimate higher LB values and lower UB values, meaning that the closer the LB and UB are to the optimal result, the better. This approach ultimately helps reduce the

number of makespan calculations required.

To find the LB, we need to rely on the encoding rules outlined in Section 3.2. Some rules may potentially create negative time values if the makespan L is too low, as exemplified by Rule 4 with $t = L - \sum_{u=i+1}^{q_l-1} p_u^l$. The LB we seek is the smallest makespan L that satisfies the condition of ensuring no negative time values for all variables. First, encode the problem with $L = 0$, and find the smallest time value t_{min} among the variables. Value t_{min} will be a negative value. The LB is then derived as the absolute value of that time t.

The desired UB value needs to be a satisfiable makespan value. The simplest approach involves arranging task executions sequentially, similar to the task order in the input: all tasks of job 0 must finish before task 0 of job 1 can start and so on. The task execution order can be represented as follows:

$$O_0^0,\ O_1^0, ...,\ O_{q_0-1}^0,\ O_0^1,\ O_1^1, ..., O_{q_n-1}^{n-1}$$

Although this approach results in a large makespan value, it certainly guarantees the satisfaction of all three JSSP constraints. With this method, at any given point in time, only one task is executed, and only one machine is active. Therefore, it ensures that no conflicts or resource contention arise during the execution of tasks.

3.6 Find optimal makespan

After determining LB and UB, the optimal makespan is certainly within the range [LB, UB]. Because optimal makespan divides SAT problems into an unsatisfiable part and a satisfiable part (Section 3.1), I propose an algorithm based on binary search to quickly obtain the optimal makespan instead of searching all values in range [LB, UB]. The algorithm is described in detail as Algorithm 1. This algorithm has a complexity of $O(log_2(n))$ with $n = UB - LB$.

Algorithm 1 Algorithm to find the optimal makespan

Require: $0 < low < up$ (with low is lower bound, up is upper bound)

Ensure: op is optimal makespan

```
while low ≤ up do
    mid ← (low + up)/2
    if mid is satisfiable then
        if mid − 1 is not satisfiable then
            op ← mid
        else if mid − 1 is satisfiable then
            up ← mid − 1
        end if
    else if mid is not satisfiable then
        if mid + 1 is satisfiable then
            op ← mid + 1
        else if mid + 1 is not satisfiable then
            low ← mid + 1
        end if
    end if
end while
```

Chapter 4

Experiments and evaluations

4.1 Experimental environment

All experiments were conduct on the personal computer. I use Minisat SAT solver [22] to solve the problems. Details of the hardware configuration of the experimental computer are described in Table 4.1 below:

Table 4.1: Configuration of the experimental environment

Processor	AMD Ryzen 5 5500U with Radeon Graphics 2.10 GHz
RAM	24.0 GB
Operating system	Windows 11 Home Single Language version 23H2
Programming language	Python 3.11.5
SAT Solver	Minisat v1.13

4.2 Data analytics

4.2.1 Open job-shop scheduling problems

There is a variety of benchmark problems for job shop: the well-known FT10 with size of (10 jobs × 10 machines) and FT20 (20 jobs × 5 machines) from Fisher and Thompson [9], and La21, La24, La25 (15 jobs × 10 machines), La27, La29 (20 jobs × 10 machines), La38, La40 (15 jobs × 15 machines) from Lawrence [18], and ABZ7, ABZ8, ABZ9 (20 jobs × 15 machines) from Adams, Balas and Zawack [1] - a set of 10 problems considered to be hard to solve for classical job shop. The set of JSSP instances

mentioned comprises well-established benchmarks commonly used as benchmarks in the academic literature to evaluate the performance of optimization algorithms. The introduction of these instances can be traced back to various sources, and they have been widely adopted in research papers and competitions.

In this chapter, I try to solve these three open JSSPs: FT06, La03 and Orb07. FT06 consists of 6 jobs and 6 machines, originating from the same problem set as FT10 and FT20. La03 is a problem instance from the same set as La21, La24, and La25, consisting of 10 jobs and 5 machines. Orb07 has 10 jobs and 10 machines, was first introduced by Applegate and Cook in 1991 [2]. The reason for selecting these problems is their relatively small number of jobs and machines, which makes them suitable for the computing resources available. By choosing problems with fewer jobs and machines, the execution time can be kept reasonably short, preventing excessively long runtimes that could otherwise hinder the analysis and comparison.

All three problems have already been optimally solved using various methods. FT06 was first solved by Carlier J and Pinson É [5], La03 was first solved by Applegate and Cook [2] and Orb07 was first solved by Henning [14]. However, to validate the proposed method in this research, I will resolve these problems from scratch.

4.2.2 Input file specifications

Jobshop instances are typically expressed in one of two forms: standard specification or Taillard specification. First is the most widely used specification and second is the specification used by Taillard [24].

In the standard specification, the input data is provided as a text file. On the first line are two numbers, the first is the number of jobs and the second is the number of machines. Following there is a line for each job. The order for visiting the machines is presented together with the corresponding processing time. The numbering of the machines starts at 0. . For example an instance with 2 jobs and 3 machines can be represented by:

$$
\begin{array}{cccccc}
2 & 3 & & & & \\
1 & 6 & 2 & 7 & 0 & 5 \\
0 & 4 & 2 & 3 & 1 & 9
\end{array}
$$

The first job has three tasks. The first task is executed on machine 1 with processing time of 6. The second task is executed on machine 2 with processing time of 7. The third task is processed on machine 0 with processing time of 5. The order that the machines

are to be visited by the first job is 1, 2, 0. The second job has three tasks: (machine 0, time 4), (machine 2, time 3), (machine 1, time 9) and the order that the machines are to be visited by the second job is 0, 2, 1.

In the Taillard specification, the input data is also provided as a text file. On the first line are two numbers, the first is the number of jobs and the second the number of machines. Following there are two matrice: the first with a line for each job containing the processor times for each task, the second with the order for visiting the machines. The numbering of the machines starts at 1. For example the same instance as above would be presented as:

```
2 3
6 7 5
4 3 9
2 3 1
1 3 2
```

In this experiment, all three problems (FT06, La03, and Orb07) are all represented using a standard specification format. The FT06 problems with 6 jobs and 6 machines is represented as follows:

```
6 6
2 1 0 3 1 6 3 7 5 3 4 6
1 8 2 5 4 10 5 10 0 10 3 4
2 5 3 4 5 8 0 9 1 1 4 7
1 5 0 5 2 5 3 3 4 8 5 9
2 9 1 3 4 5 5 4 0 3 3 1
1 3 3 3 5 9 0 10 4 4 2 1
```

The La03 problem with 10 jobs and 5 machines is represented as follows:

10	5								
1	23	2	45	0	82	4	84	3	38
2	21	1	29	0	18	4	41	3	50
2	38	3	54	4	16	0	52	1	52
4	37	0	54	2	74	1	62	3	57
4	57	0	81	1	61	3	68	2	30
4	81	0	79	1	89	2	89	3	11
3	33	2	20	0	91	4	20	1	66
4	24	1	84	0	32	2	55	3	8
4	56	0	7	3	54	2	64	1	39
4	40	1	83	0	19	2	8	3	7

The Orb07 with 10 jobs and 10 machines is represented as follows:

10	10																		
0	32	1	14	2	15	3	37	4	18	5	43	6	19	7	27	8	28	9	31
0	8	3	12	4	49	8	24	9	52	6	19	7	23	5	19	2	17	1	32
0	25	7	19	3	27	2	45	6	21	4	15	1	13	5	16	8	43	9	19
0	24	1	18	3	41	8	29	5	14	2	17	4	23	7	15	6	18	9	23
0	27	6	29	1	39	3	21	5	15	2	15	9	25	7	26	8	44	4	20
4	17	0	15	6	51	8	17	2	46	3	16	9	33	7	25	5	30	1	25
4	15	3	31	0	25	9	12	2	13	1	51	8	19	7	21	6	12	5	26
4	8	6	29	9	25	3	15	8	17	2	22	7	32	5	20	0	11	1	28
2	41	6	10	8	32	7	5	4	21	9	59	3	26	1	10	5	16	0	29
2	20	9	7	5	44	8	22	6	33	3	25	7	29	4	12	1	14	0	0

These problems share similar characteristics: each job utilizes all machines, with each machine being used once by an task, implying that the total number of tasks within a job is equal to the number of machines. The only exception is the final task of the last job in the Orb07 problem, which has a processing time of 0. This can be considered as the corresponding job not running on machine 0 at any tasks.

Based on the input data, it can be observed that FT06 is the simplest problem among the three, having the smallest number of jobs and machines and the shortest processing times ranging from 0 to 10. A smaller processing time results in a smaller number of cases to evaluate and, consequently, a lower makespan value. The other two problems have processing times ranging from 0 to 100. Although Orb07 has more

machines, its average processing time is slightly smaller than that of La03. Thus, it is unclear which of these two problems is more complex without further analysis.

4.3 Experimental results

4.3.1 Lower bound and upper bound

Prior to solving the problems, it is necessary to establish the LB and UB. Using the previously mentioned method, the LB and UB for each problem are calculated and presented in Table 4.2.

Table 4.2: Results of LB and UB

Problem	**LB**	**UB**
FT06	39	197
La03	268	2383
Orb07	258	2407

It is evident that the current method for calculating the lower bound (LB) and upper bound (UB) is not yet optimal, as there is a significant gap between the LB and UB values. To find the optimal makespan value, it is necessary to test a wide range of makespan values.

4.3.2 Optimal makespan

Let S_L^P be a SAT instance from problem P generated by our encoding method under assumption that the makespan equals L. For each problem P, I generate about $2log_2(UB - LB)$ instances using the algorithm presented in Section 3.6 and solve each instances with our proposed encoding method. If the optimal solution is found early, the solving process can be stopped, so the number of instances required can be reduced. However, even in the worst-case scenario, the number of instances will not exceed the aforementioned number. Even though S_{LB}^P is definitely unsatisfiable, S_{UB}^P is definitely satisfiable as proven in Section 3.5, I still solve these two intances to verify the values.

I succeeded to solve three problems. The optimal makespans of FT06, La03 and Orb07 are 55, 597 and 397, respectively. These results are the same as the optimal makespan values obtained from other research studies. Table 4.3, 4.4 and 4.5 show the experimental results of S_L^{FT06}, S_L^{La03} and S_L^{Orb07}, respectively. The first column is

Table 4.3: Result of S_L^{FT06}

L	Number		CPU (secs)	Satisfiable
	Variables	Clauses		
39	3091	11525	1.43	no
50	3882	15023	1.79	no
51	3954	15341	1.79	no
53	4098	15977	1.85	no
54	4170	16295	2.09	no
55	**4242**	**16613**	**2.16**	**yes**
56	4314	16931	2.09	yes
61	4674	18521	2.25	yes
62	4746	18839	2.53	yes
86	6474	26471	3.34	yes
87	6546	26789	3.33	yes
136	10074	42371	5.09	yes
137	10146	42689	5.42	yes
197	14466	61769	7.80	yes

the makespan value L of the SAT instance. The second and third columns represent the size of the instance: the number of variables and clauses created during the encoding process. For each value of L, perform 10 solves and record the runtime for each run. The fourth column represents the average runtime of these 10 runs. This runtime includes all encoding, solving, and decoding steps. The last column shows the instance is satisfiable or not.

As predicted, the FT06 problem with smaller inputs, including fewer jobs, fewer machines, and shorter processing times, significantly reduces the size of the problem (the LB and UB, the number of variables and clauses) compared to the other two problems. Therefore, the solving time for instances of FT06 is also shorter, typically less than 10 seconds per instance.

The changes in the number of variables and clauses as L increases are visualized in the graphs presented in Figure 4.1. The horizontal axis of the graph represents the value of L, while the vertical axis shows the number of variables or clauses. Each dot in the graph corresponds to an instance, with the blue dots representing the number of variables and the orange dots indicating the number of clauses. Each pair of dots

Table 4.4: Result of S_L^{La03}

L	Number		CPU (secs)	Satisfiable
	Variables	Clauses		
268	27391	146620	18.00	no
531	53690	314940	38.57	no
532	53790	315580	38.05	no
596	60190	356540	46.25	no
597	**60290**	**357180**	**47.53**	**yes**
662	66790	398780	55.28	yes
663	66890	399420	55.55	yes
795	80090	483900	72.60	yes
796	80190	484540	83.44	yes
1324	132990	822460	139.88	yes
1325	133090	823100	137.50	yes
2383	238890	1500220	322.42	yes

which are collinear in a vertical line represent the same instance (same L). All blue dots collinear represents the linear increase of the number of variables as L increases. A similar situation occurs with the number of clauses. The size of SAT instances exhibits a linear increase as L grows because the number of clauses generated by each encoding rule depends on the value of L. For instance, rule 5 involves constraints $1 \leq t \leq L$ so it produces a number of clauses that increases linearly with L.

Observing the graph from Figure 4.1 reveals that the distance between dots decreases as they are near the optimal L. This demonstrates the operational steps of Algorithm 1, which continuously checks the value at the midpoint and halves the range between the LB and UB.

CPU time increases as L increases. This can be explained by the fact that the size of SAT instances increases, leading to a greater number of computational operations, which require more time to complete. The most time-consuming instance is S_{2407}^{Orb07}, taking 961.71 seconds per run. In total, 10 runs take 9617.1 seconds, which is nearly equivalent to 3 hours. In total, it takes over 9 hours to solve the entire Orb07 problem. While this time is still acceptable for a single problem, if I attempt to solve larger problems such as ABZ9 (20 jobs x 15 machines), the solving time could extend to several days or even months. This timeframe is not practical for real-world applications.

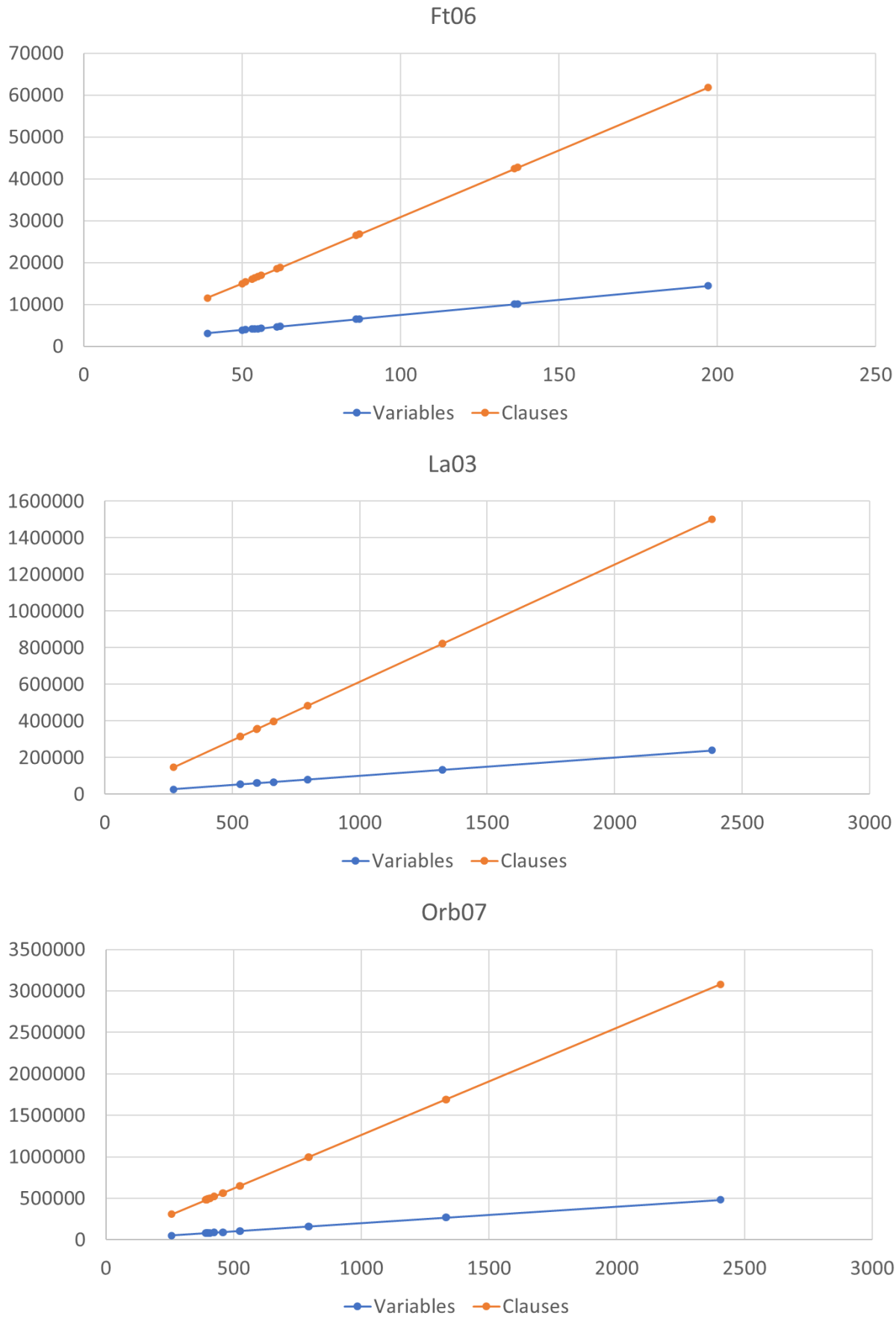

Figure 4.1: Relationship between variables, clauses, and L

Additionally, larger problems entail a massive number of variables and clauses, which current personal computer configurations cannot handle. In conclusion, while the current method has made progress in solving the given problems, its lengthy solving time and limitations in handling larger instances highlight the need for future improvements to enhance efficiency and scalability.

Table 4.5: Result of S_L^{Orb07}

L	**Number**		**CPU (secs)**	**Satisfiable**
	Variables	**Clauses**		
258	52792	308506	43.07	no
391	79392	480076	62.86	no
392	79592	481366	62.95	no
395	80192	485236	64.34	no
396	80392	486526	62.89	no
397	**80592**	**487816**	**73.20**	**yes**
398	80792	489106	73.29	yes
399	80992	490396	77.50	yes
406	82392	499426	75.57	yes
407	82592	500716	75.47	yes
423	85792	521356	79.09	yes
424	85992	522646	79.39	yes
457	92592	565216	85.72	yes
458	92792	566506	87.16	yes
524	105992	651646	102.01	yes
525	106192	652936	102.53	yes
793	159792	998656	178.87	yes
794	159992	999946	177.77	yes
1331	267392	1692676	372.83	yes
1332	267592	1693966	372.25	yes
2407	482592	3080716	961.71	yes

4.3.3 Find other schedules

With each optimal makespan, there are multiple suitable schedules. In the previous section, with each instance, the solver provided a set of variable values to demonstrate

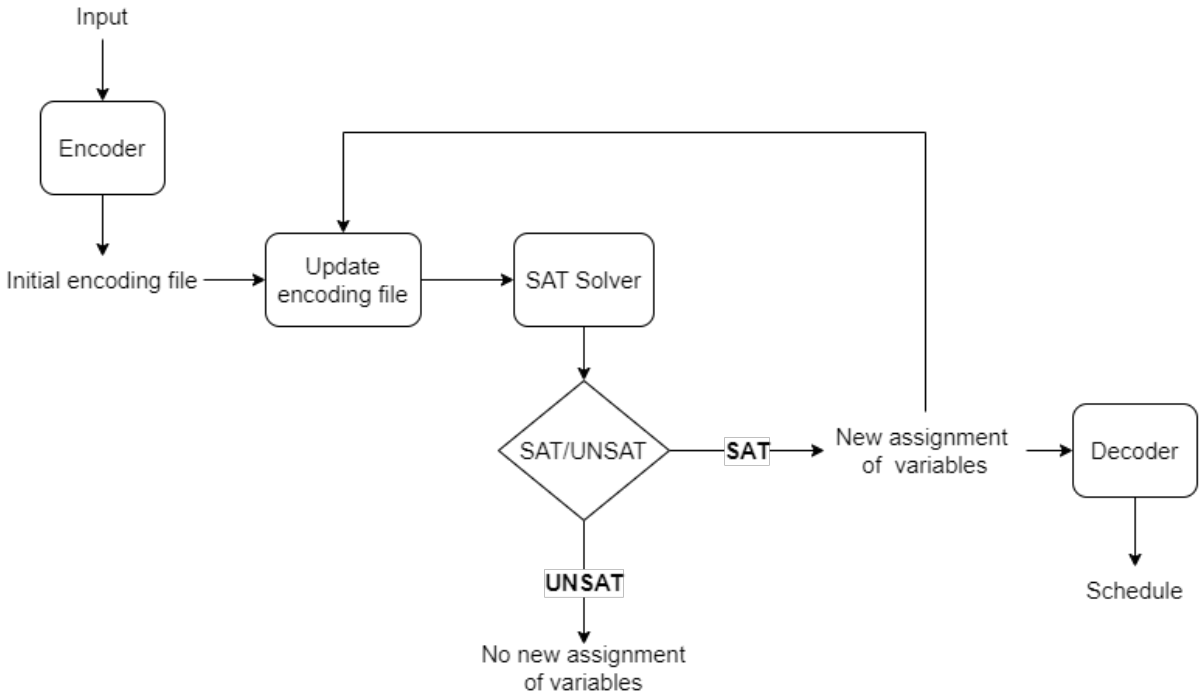

Figure 4.2: Find other schedules flow chart

whether the instance is satisfiable or not, resulting in a single schedule. However, in practical scenarios, it's often necessary to explore multiple schedules for comparison and selection of the most suitable one. Therefore, this section focuses on searching for additional schedules beyond the initially found one.

To find other schedules, I apply the method presented in section 3.3.5. The flow of this process is shown in Figure 4.2. From each assignment of variables found, I construct a new clause consisting of the negations of those variables, then append this clause to the end of the encoding file and update the number of clauses. I input the new encoding file into the solver, find the new assignment of variables, and repeat the process. In each iteration of finding new assignment, there is no need to redo the encoding step since the input conditions of the problem remain unchanged. Therefore, the initial encoding file can be reused, with new clauses added incrementally. The process only stops when the SAT solver returns UNSAT, indicating that no further assignments satisfying the conditions can be found.

Each new assignment of variables is input to the Encoder to find a corresponding schedule. However, the schedules found may be dupplicated because there are cases where changing the truth values of variables does not affect the start time, so the schedule remains unchanged. The experiment was conducted to find the schedules for the optimal makespan of 3 problems. The time limit for the each instance was set to 22000 seconds.

The number of assignments and schedules found is shown in Table 4.6. The number of schedules counts only distinct schedules. It can be observed that within a large time limit, despite finding a large number of assignments, only a few distinct schedules are found. The last column shows if the instance has found all possible assignments of variables. For all three problems, the solver has not returned UNSAT, meaning that if continued, it's still possible to find additional schedules.

Table 4.6: Results of finding other schedules for S_{55}^{FT06}, S_{597}^{La03} and S_{397}^{Orb07}

Instances	**Assignments**	**Schedules**	**Found all**
S_{55}^{FT06}	752	8	no
S_{597}^{La03}	237	9	no
S_{397}^{Orb07}	205	4	no

Figure 4.4 illustrates the change in time over each iteration of finding new assignments. The vertical axis represents the time measurement in seconds, while the horizontal axis depicts the order of assignments. It can be observed that as assignment order grows, the time increases exponentially. This is understandable because adding more clauses enlarges the size of the encoding file, requiring more computational time. For example, the first assignment of S_{55}^{FT06} only takes about 1 seconds to solve, but assignment number 752 takes about 80 seconds. If continuing to solve after 752, next assignments take even more time, and it's uncertain how much time it will take to find all possible assignments. It's impossible to find all schedules within polynomial time, reflecting the NP-hardness of JSSP.

4.3.4 Verify schedules

All schedules found are checked to see if they satisfy the problem's conditions. Each schedule are checked through 3 steps:

1. Calculate the end time of each task by sum of the start time and processing time. Check if the maximum end time is equal to makespan L.

2. Group tasks belonging to the same job, sort them in ascending order by start time. Check if the end time of the previous task is not greater than the start time of the following task.

3. Grouping tasks using the same machine, sort them in ascending by start time.

Check if the end time of the previous task is not greater than the start time of the following task.

The result of the experiment is that all schedules found from previous section satisfy the problem conditions. Each schedule can be represented as a Gantt chart, which visually displays the running order of tasks to see if they are satisfied or not. Figure 4.3 illustrates the Gantt charts of a schedule for instance S_{55}^{FT06}. The text inside each task is the job index. The tasks from same job have the same color.

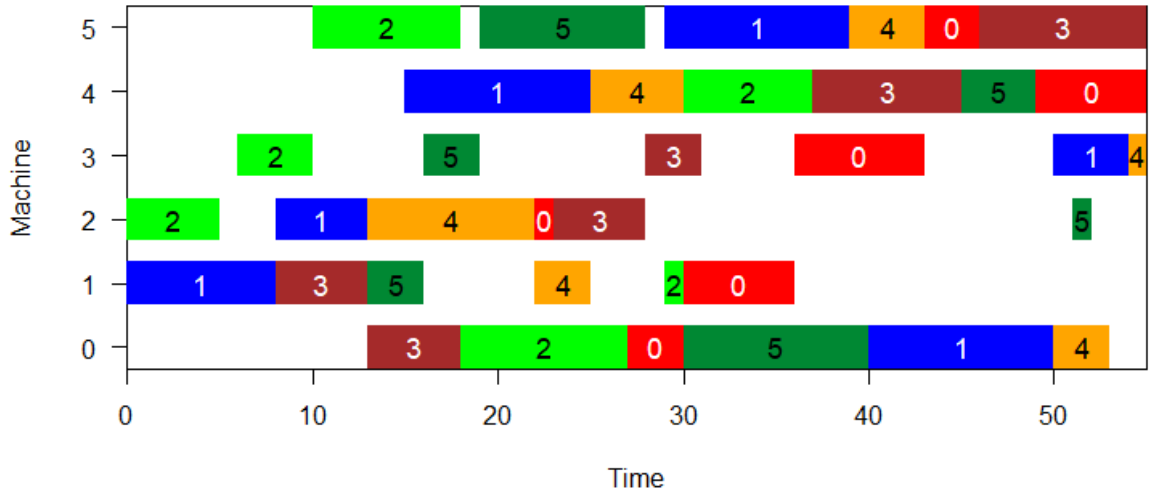

Figure 4.3: Gantt charts of a schedules for instance S_{55}^{FT06}

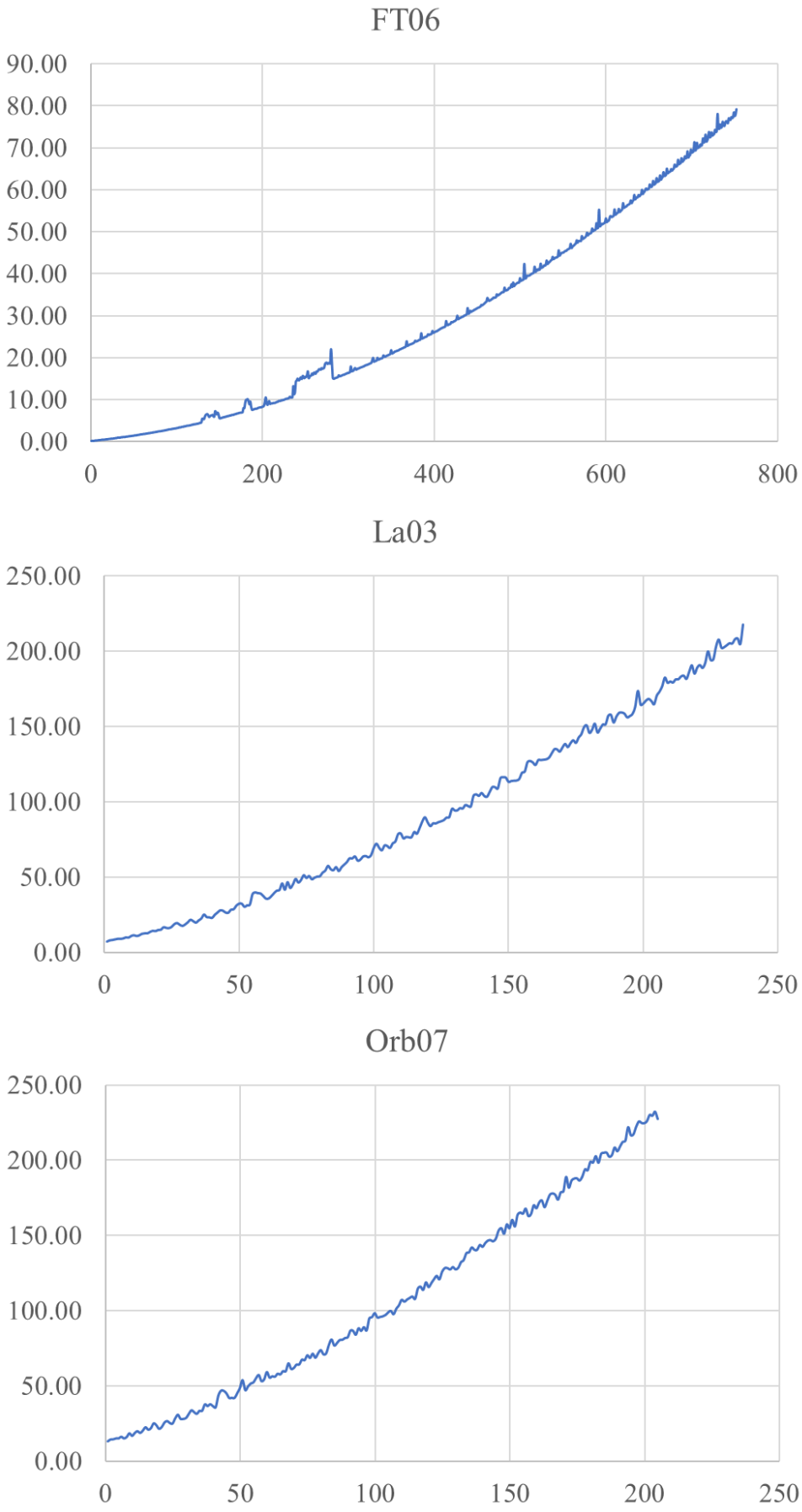

Figure 4.4: Changes in time over iterations of finding new assignments of variables

Chapter 5

Conclusions, limitations and future works

5.1 Conclusions

The thesis used SAT approach to solve JSSPs including: the encoding method based on the approach proposed by Crawford and Baker, the approach for estimating LB and UB, the algorithm for quickly obtain the optimal makespan.

The experiments were conducted on problems FT06, La03, and Orb07. By applying these methods, I successfully found the optimal makespans for all three problems which are 55, 597 and 397, respectively. The obtained LB and UB are reasonably acceptable. The exprerimental results also show that the size of SAT instances (number of clauses, variables) increases linearly as makespan grows.

Additionally, some other schedules were found besides the one found during the process of finding optimal makespan. The experiments were conducted with the optimal makespan of each problem. The results show that 8, 9, and 4 schedules is found for S_{55}^{FT06}, S_{597}^{La03}, and S_{397}^{Orb07}, respectively, with a time limit of 22000 seconds for each instances.

5.2 Limitations

The LB and UB estimating method are not yet optimal as there is a significant gap between the LB and UB values. The range of testing makespan values is still wide. The UB values are still large and not close to the optimal makespan, leading to the need to

compute with unnecessarily large makespan values.

In some cases with large makespan values, the number of variables and clauses can reach millions. This requires huge computation steps which is hard for typical personal computers to process and prolong runtime significantly. This aspect makes the method impractical for hard problems with a large number of machines and jobs.

During process of finding other schedules, the run time increases exponentially. Therefore, it's not possible to find all possible schedules within polynomial time.

5.3 Future works

With the results achieved during the research process, it also points to some future development directions:

- Improve LB and UB estimating, make it closer to the optimal makespan.
- Reducing the number of variables and clauses generated during the encoding process.
- Propose a more efficient method to find all possible schedules for each SAT instance.

References

[1] J. Adams, E. Balas, and D. Zawack, "The shifting bottleneck procedure for job shop scheduling," *Management science*, vol. 34, no. 3, pp. 391–401, 1988.

[2] D. L. Applegate and W. J. Cook, "A computational study of the job-shop scheduling problem," vol. 3, no. 2, pp. 149–156, May 1991, the JSSP instances used were generated in Bonn in 1986.

[3] J. Argelich, C.-M. Li, F. Manyà, and J. Planes, "The first and second max-sat evaluations," *JSAT*, vol. 4, pp. 251–278, 09 2008.

[4] R. Asín Achá and R. Nieuwenhuis, "Curriculum-based course timetabling with sat and maxsat," *Annals of Operations Research*, vol. 218, no. 1, p. 71–91, Feb 2012.

[5] J. Carlier and É. Pinson, "An algorithm for solving the job-shop problem," *Management Science*, vol. 35, no. 2, pp. 164–176, 1989, jstor: 2631909.

[6] J. M. Crawford and A. B. Baker, "Experimental results on the application of satisfiability algorithms to scheduling problems," in *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*, ser. AAAI'94. AAAI Press, 1994, p. 1092–1097.

[7] M. A. Cruz-Chávez and R. Rivera-López, "A local search algorithm for a sat representation of scheduling problems," *Lecture Notes in Computer Science*, p. 697–709, 2007.

[8] L. Di Gaspero, B. Mccollum, and A. Schaerf, "The second international timetabling competition (itc-2007): Curriculum-based course timetabling (track 3)," 01 2007.

[9] H. Fisher and G. L. Thompson, "Probabilistic learning combinations of local job-shop scheduling rules," *Industrial Scheduling*, pp. 225–251, 1963.

[10] I. P. Gent, R. W. Irving, D. F. Manlove, P. Prosser, and B. M. Smith, "A constraint programming approach to the stable marriage problem," in *Principles and Practice of Constraint Programming—CP 2001: 7th International Conference, CP 2001 Paphos, Cyprus, November 26–December 1, 2001 Proceedings 7*. Springer, 2001, pp. 225–239.

[11] A. Gorbenko and V. Popov, "Task-resource scheduling problem," *International Journal of Automation and Computing*, vol. 9, no. 4, p. 429–441, 2012.

[12] R. L. Graham, "Bounds for certain multiprocessing anomalies," *The Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, Nov 1966.

[13] Y. Hamadi, S. Jabbour, and J. Sais, "Control-based clause sharing in parallel sat solving," *Autonomous Search*, p. 245–267, 2011.

[14] A. Henning, "Praktische Job-Shop Scheduling-Probleme," Ph.D. dissertation, Friedrich-Schiller-Universität Jena, Jena, Germany, Aug. 2002, alternate url: https://nbn-resolving.org/urn:nbn:de:gbv:27-20060809-115700-4. [Online]. Available: http://www.db-thueringen.de/servlets/DocumentServlet?id=873

[15] H. Huang and S. Zhou, "An efficient sat algorithm for complex job-shop scheduling," *Proceedings of the 2018 8th International Conference on Manufacturing Science and Engineering (ICMSE 2018)*, 2018.

[16] M. Karpiński, "Cnf encodings of cardinality constraints based on comparator networks," *arXiv preprint arXiv:1911.00586*, 2019.

[17] M. Koshimura, H. Nabeshima, H. Fujita, and R. Hasegawa, "Solving open job-shop scheduling problems by sat encoding," *IEICE Transactions on Information and Systems*, vol. E93.D, no. 8, pp. 2316–2318, 2010.

[18] S. R. Lawrence, "Resource constrained project scheduling: An experimental investigation of heuristic scheduling techniques (supplement)," Ph.D. dissertation, Graduate School of Industrial Administration (GSIA), Carnegie-Mellon University, Pittsburgh, PA, USA, 1984.

[19] F. Pezzella, G. Morganti, and G. Ciaschetti, "A genetic algorithm for the flexible job-shop scheduling problem," *Computers & Operations Research*, vol. 35, no. 10, pp. 3202–3212, 2008, part Special Issue: Search-based Software Engineering. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054807000524

[20] O. Sitompul and E. Nababan, “Monte carlo acceptance criterion in optimizing job shop scheduling problems,” 06 2010.

[21] M. Soeken, G. De Micheli, and A. Mishchenko, “Busy man’s synthesis: Combinational delay optimization with sat,” in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*, 2017, pp. 830–835.

[22] N. Sörensson and N. Een, “Minisat v1.13-a sat solver with conflict-clause minimization,” *International Conference on Theory and Applications of Satisfiability Testing*, 01 2005.

[23] N. Sörensson and N. Eén, “The minisat page.” [Online]. Available: http://minisat.se

[24] E. Taillard, “Benchmarks for basic scheduling problems,” *European Journal of Operational Research*, vol. 64, no. 2, p. 278–285, Jan 1993.

[25] M. Vali, K. Salimifard, A. H. Gandomi, and T. J. Chaussalet, “Application of job shop scheduling approach in green patient flow optimization using a hybrid swarm intelligence,” *Computers & Industrial Engineering*, vol. 172, p. 108603, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835222005988