Combinatorial optimization for job sequencing with one common and multiple secondary resources by using a SAT solver augmented with a domain-specific heuristic

Artjom Pugatšov

Supervisors: Emir Demirović, Konstantin Sidorov, Maarten Flippo

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Artjom Pugatšov
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Konstantin Sidorov, Maarten Flippo, Jérémie Decouchant

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

This paper solves job sequencing with one common and multiple secondary resources (JSOCMSR) problem by encoding it as a Boolean satisfiability (SAT) problem and applying domain-specific heuristics to improve the SAT solver's performance. JSOCMSR problem is an NP-hard scheduling problem where each job utilizes two resources: a shared resource and a secondary job-dependent resource. First, the problem was modeled as an instance of SAT and then the SAT solver was augmented with a static greedy variable-ordering heuristic. This heuristic has led to significant improvement in the solver's speed compared to a generic SAT heuristic for problem instances of larger size.

1 Introduction

This paper discusses the job sequencing with one common and multiple secondary resources (JSOCMSR) problem. It is a scheduling problem, where each job uses two types of resources: 1) a common resource that is shared between all the jobs and is required only for a certain segment of the work, and 2) a secondary resource that is shared by only a subset of jobs, but is needed throughout their entire timespan. Additionally, the resources are non-sharable, meaning that no two jobs can use the same resource at the same time. The goal of the problem is to minimize the makespan of the solution, i.e. the latest finish time of a job.

The JSOCMSR problem has multiple real-world applications. One such application is in metal-forming. A metal-forming machine acts as the common resource and the molds act as the secondary resources. The molds are needed throughout the entire process, whereas the machine is only required for the metal-pouring part of the job. Another example of the problem's application is in scheduling cancer treatment appointments. It was first described by Horn et al [6] and was the original reason for researching JSOCMSR. This is a relatively new treatment, where a particle acceleration beam is used. The beam is expensive and typically only one beam is available. Whereas, there are multiple rooms where the therapy is conducted. The beam corresponds to the common resource and the rooms are the secondary resources.

JSOCMSR is an NP-Hard scheduling problem [6]. One common way of tackling such problems is by modeling them in terms of Boolean satisfiability problems (SAT) [1; 2].

By encoding the problems into SAT, a general SAT solver can be used to solve them. This way a wide range of problems can be solved using the same solver, as long as it is possible to model it with propositional logic. This allows for flexibility in solving many different problems and makes it possible to focus on improving a single solver algorithm, as opposed to developing specialized algorithms for each problem [2]. However, this approach does not take into account the specific properties of the problem when applying the solving algorithm. Employing problem-specific heuristics can further improve the performance of SAT-solving algorithms [14]. **The goal of this paper is to encode JSOCMSR as a SAT instance, develop a problem-specific heuristic for it and measure the performance of the SAT solver augmented by the heuristic.**

To accomplish this goal, JSOCMSR has been encoded into SAT. Then a heuristic that sorts variables based on the start time and duration of their corresponding jobs has been added to an existing SAT solver and experimentally tested. The heuristically augmented version has displayed significant improvement compared to the baseline heuristic, especially for the bigger problem instances.

The structure of this paper is as follows. Section 2 presents the related work done for JSOCMSR and also discusses other problems similar to it. Then section 3 gives a formal definition of the problem. Section 4 discusses the encoding of JSOCMSR into SAT. In section 5 the examined heuristics are described. Section 6 covers the experimental setup and the results of these experiments. Then section 7 discusses how the responsible research principles have been incorporated into the research. Section 8 provides a conclusion by summarizing the most important results and putting them into a broader context. And lastly, section 9 discusses the possible future work that can be done as a continuation of this research.

2 Related Work

JSOCMSR is a relatively new problem. It was first introduced and proven to be np-hard by Horn et al. in 2019 [6]. Before that, a similar problem of scheduling jobs with a shared resource was researched by Van der Veen et al [16]. The crucial difference is that compared to JSOCMSR, where each job has a pre-processing times during which the common resource is not used, this problem features no pre-processing times. This has made it possible to apply an exact polynomial-time algorithm to the problem, whereas pre-processing times make JSOCMSR NP-hard.

There have been two works that directly cover the JSOCMSR problem. The first work is by Horn et al., [6] where the problem was initially introduced and formalized. The authors have also proven JSOCMSR to be NP-hard and proposed an A* beam search algorithm for solving the problem. In the second work by Kaufmann et al., [8] a different algorithm was presented to solve the problem. A variable neighborhood search (VNS) technique has been applied in order to iteratively find better solutions. Both algorithms have been compared to a CSP encoding of the problem in their respective papers. Both approaches have been found to be more efficient than the CSP encoding approach. Despite that, it is worth noting that in both cases the CSP encoding was not augmented beyond the default heuristics employed by the CSP solvers that were used.

There exists a modified version of JSOCMSR that centers around prize-collecting (PC), instead of minimizing the makespan, called PC-JSOCMSR [7]. Even though PC-JSOCMSR is based on JSOCMSR, the methods required to solve the PC variant are significantly different from the ones required for JSOCMSR. Due to this, the prize-collecting version is not covered in this paper.

JSOCMSR properties are similar to the ones of the no-wait

flow-shop scheduling (NWFSS) problem. In this problem, any non-dominated solution is determined by the order in which the jobs start on the first machine [3]. Similarly, in JSOCMSR, any non-dominated solution corresponds to the order in which the jobs use the common resource [6]. This makes it possible to express solutions to both problems as a permutation of jobs and allows the use of similar heuristic approaches when solving both problems.

3 Job Scheduling With One Common and Multiple Secondary Resources

The following section first gives a formal definition of the problem. After that, an example problem instance is given.

3.1 Formal Description

An instance of JSOCMSR problem consists of the following components:

- $J = \{j_1, j_2, ..j_n\}$ - a set of n jobs that need to be scheduled.
- $R_s = \{r_1, r_2, ..r_m\}$ - a set of m secondary resources.
- r_0 - the common resource shared between all the jobs.

Each job j_i has a total processing time of $p_i > 0$ during which the secondary resource $r \in R_s$ is fully used. Also, each job j_i requires $p_i^0 > 0$ common resource processing time throughout which r_0 is utilized. Before the common resource is processed, there is pre-processing time p_i^{pre}. Processing of the common resource is done in parallel with the secondary resource, meaning that $p_i^0 \leq p_i - p_i^{pre}$. The common resource must be scheduled immediately after pre-processing has finished. This means that the need for the common resource starts at time p_i^{pre} and ends at time $p_i^{pre} + p_i^0$. This makes it possible to define the post-processing time as $p_i^{post} = p_i - p_i^{pre} - p_i^0$.

A solution for the problem has the form S = $[s_1, s_2, .., s_n]$ with $s_i \geq 0$, where s_i is the starting time of job j_i. The jobs must be scheduled in such a way that the solution is feasible, meaning that no two jobs use the same resource at the same time. The goal is to create a schedule such that the latest job finishes as early as possible. This implies the minimization of makespan L

$$L = \max_{0 \leq i \leq n} (s_i + p_i)$$

3.2 Example Problem

For the example, a simple problem instance with 5 jobs and 3 common resources was chosen. Figure 1 depicts the jobs. Each job consists of 3 parts: pre-processing, common-resource usage, and post-processing. The jobs that use the same common resource are on the same line and are colored in the same way. The middle part of each job is the time at which it uses the common resource.

The jobs cannot overlap with other jobs that use the same secondary resource. The time they use the common resource cannot overlap with that of any other job. In figure 1, the schedule is invalid, since there is an overlap in the use of the common resource for jobs I and III; IV and V. A valid schedule can be created by interchanging jobs I and III and scheduling IV at a later point. This schedule is shown in figure 2. The makespan of this schedule is 12 since the last job ends at time 12.

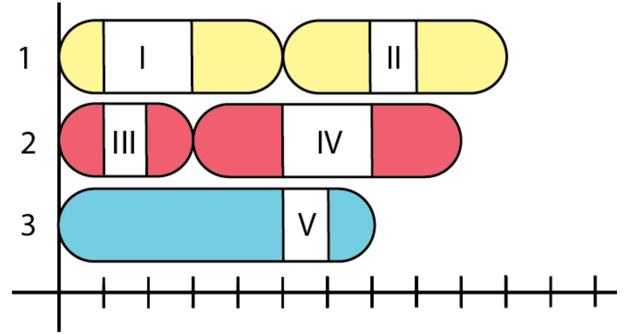

Figure 1: Example instance of JSOCMSR problem with 3 secondary resources and 5 jobs

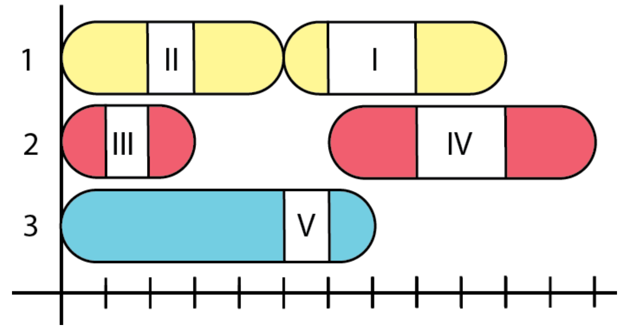

Figure 2: Example of a valid schedule for a small instance of JSOCMSR problem with makespan of 12

4 SAT Encoding

In order to solve the problem, it has to be first encoded in terms of a SAT instance. A SAT problem consists of a set of Boolean variables and the logical constraints that express the relationship between these variables, such as AND, OR, and NOT. In practice, the most widely used format by the solvers is conjunctive normal form (CNF), meaning that after creating a high-level encoding of the problem, it needs to be expressed in terms of CNF.

Since JSOCMSR is a minimization problem, a straightforward SAT encoding is not sufficient, as there are multiple valid solutions with varying makespans. As such, the problem was encoded as a maximum satisfiability problem (MaxSat).

MaxSat allows for a combination of soft and hard constraints. The hard constraints must be satisfied for any variable assignment. The soft constraints act as a measure of optimality and they might be left unsatisfied. Each soft constraint has a weight associated with it and the final measure of optimality is the sum of the weights of unsatisfied soft constraints. The fewer soft constraints are unsatisfied, the better the solution [9].

One of the requirements for a MaxSat model is that each part of the model needs to be discretized, as a SAT problem consisting of a set of boolean variables. Unfortunately, the original dataset used for the JSOCMSR problem contains very long duration times for the jobs ranging from 300 to 3000 unspecified time units. This means it is unfeasible to directly apply a SAT encoding to the dataset, as the number of clauses grows too fast. As such, it is necessary to apply a scaling procedure to the test set. All the durations of jobs have been divided by 150 to make the average duration of a job equal to 10. The scaling preserves the overall difficulty of the original problem but makes it feasible to encode the problem in terms of SAT.

The following section provides a formal MaxSat encoding of the problem. First, the variables and the hard constraints are introduced, then the soft constraints are described, and lastly, the encoding of high-level constraints is addressed.

4.1 Variables

There are 3 main types of variables that were used in modeling the problem: start time variables, resource usage variables, and makespan variables. The first two are essential for modeling the core constraints of the problem, and makespan variables are used to determine the makespan and help in the maximization process.

Start Time Variables

The start time variables determine when the jobs start. The set S of start time variables comprises elements, where each variable corresponds to a specific combination of a start time with its respective job. The possible start times range from 0 to the maximum start time T_{max}. The maximum start is not given directly by the problem but can be calculated by scheduling all the jobs sequentially, meaning that the maximum start time is $T_{max} = \sum_{i=1}^{n} p_i$. This makes it possible to define the set S as

$$S = \{s_{j,t} | \ j \ \in J, \ t \ \in [0, 1, 2, .., T_{max}]\}$$

Variable $s_{j,t}$ is set to true if and only if the job j starts at time t.

Resource Usage Variables

The job's resource usage variables signify the use of a particular resource. Since each job can only use two types of resources, the usage variables can be split into two sets: usage of the common resource U_{com} and usage of the secondary resource U_{sec}.

$$U_{com} = \{u_{com\ j,t} | \ j \ \in J, \ t \ \in [0, 1, 2, .., T_{max}]\}$$

Variable $u_{com\ j,t}$ is set to true if and only if the job j uses the common resource at the time t.

$$U_{sec} = \{u_{sec\ j,t} | \ j \ \in J, \ t \ \in [0, 1, 2, .., T_{max}]\}$$

Variable $u_{sec\ j,t}$ is set to true if and only if the job j uses its secondary resource at the time t.

Makespan Variables

The makespan variables determine if a specific time is less or equal to the latest time a job ends. This ensures that the number of these variables that are set to true is precisely equal to the makespan. The set M of makespan variables has a form:

$$M = \{m_t | \ t \ \in [0, 1, 2, ..., T_{max}]\}$$

Variable m_t is set to true if and only if the time t is less or equal to the makespan.

Additional Notation

For brevity, the additional notation describing subsets of variables is introduced. S_j is the set of all the possible start times variables of a job j. $U_{com\ t}$ is the set of all resource usage variables at time t. And $U_{sec\ t,r}$ is the set of all usage variables at time t which use the resource r.

4.2 Hard Constraints

The four main hard constraints describe the core constraints of the JSOCMSR problem. These constraints are:

1. Each job must be scheduled exactly once
2. No resource can be used by more than one job at a time
3. Job uses the secondary resource throughout its entire duration.
4. Job uses the common resource starting right after the preprocessing finishes, and ending when the postprocessing starts.

These constraints can be expressed by formulating them as additional constraints on top of the three basic logic ones. These constraints are logical implications, "Exactly-One" constraints which ensure that only one variable from the set is set to true, and "MaxOne" constraints which ensure that either one variable from the set is true, or no variable is true. These constraints can be then translated back into CNF by encoding each of them separately into CNF and then creating a conjunction of these translations. There exist different encodings for these higher-level constraints. The ones used for this research paper are discussed in section 4.4.

This makes it possible to define the 4 constraints formally:

1. $\forall j \in J : \text{ExactlyOne}(S_j)$
2. Since there are 2 types of usage variables, this constraint is composed of 2 sub-constraints:
$$\forall t \in [0, 1, 2, ..., T_{max}] : \text{MaxOne}(U_{com\ t})$$
$$\forall t \in [0, 1, 2, ..., T_{max}], \forall r \in R : \text{MaxOne}(U_{sec\ t,r})$$
3. $\forall s_{j,t} \in S : s_{j,t} \implies u_{sec\ j,t} \wedge u_{sec\ j,t+1} \wedge ... \wedge u_{sec\ j,t+p_j}$
4. $\forall s_{j,t} \in S : s_{j,t} \implies u_{com\ j,t+t_{pre}} \wedge u_{com\ j,t+p_i^{pre}+1} \wedge ... \wedge u_{com\ j,t+p_i^{pre}+p_i^0}$

The other additional hard constraints are meant to ensure the consistency of the makespan variables. These constraints take on forms of implications since makespan is fully dependent on the start times of jobs.

If a job J starts at time t, then the time at which it finishes is within the makespan.

$$\forall s_{j,t} \in S : s_{j,t} \implies m_{t+p_j}$$

The previous constraint does not take into account the full duration of the job but only considers the final end time. It is also required to ensure that when a time is within the makespan, all the previous times are also within the makespan. A straightforward way to ensure this would be to make an implication from each makespan variable to all the preceding makespan variables. This approach has the significant drawback of requiring a number of constraints proportional to $O(T_{max}^2)$. To reduce the number of constraints an inductive approach can be utilized. Each makespan variable ensures only that the variable directly following it is set to true. This way the number of constraints needed is proportional to $O(T_{max})$ instead.

$$\forall m_t \in (M \setminus \{m_0\}) : m_t \implies m_{t-1}$$

4.3 Soft Constraints

The soft constraints rely on the makespan variables. The better the solution, the fewer makespan variables are set to true.

$$\forall m_t \in M : \neg m_t$$

Since the makespan variables are strictly sequential, each soft constraint has a weight of 1.

4.4 High-level Constraint Encoding

Throughout the encoding, several constraints incompatible with CNF have been used. These constraints are MaxOne, ExactlyOne, and implication. The implication is very straightforward to translate into CNF, as

$$a \implies b \equiv \neg a \vee b$$

ExactlyOne can be formulated by additionally constraining MaxOne, such that at least one variable is set to true. This can be done by simply building a disjunction of all variables

$$\text{ExactlyOne}(a, b, c, ..) \equiv \text{MaxOne}(a, b, c, ..) \wedge (a \vee b \vee c \vee ...)$$

This means that the only constraint that needs a special encoding is MaxOne.

There are multiple ways of encoding MaxOne into CNF. The most straightforward approach that introduces no additional variables is creating a conjunction of disjunctions, where each pair of variables is represented within the disjunction and each variable is negated. This ensures that if any 2 variables are simultaneously true, at least one clause of the conjunction fails, and thus the entire statement is false. Unfortunately, this encoding requires a number of clauses proportional to $O(n^2)$, where n is the number of variables. This leads to a rapid increase in the size of the encoding.

There are other ways to encode MaxOne constraints that require fewer clauses. These encodings utilize additional variables. For this work, a Matrix encoding has been chosen. This encoding puts the variables into a square matrix and introduces auxiliary variables corresponding to each row and column in the matrix. Then MaxOne constraint is applied to the row and to the column variables separately. This ensures a one-to-one correspondence between tuples of row and column variables and original variables constrained by MaxOne. Then they are linked through implications. This encoding scales linearly in terms of clauses and additional variables [3].

5 Heuristic

SAT solvers use a systematic approach to problem-solving, exploring variable assignments and backtracking when conflicts arise. They find a solution to the problem by going over different assignments to the variables present in the encoding. At each step, it ensures that the next picked assignment is consistent with all the previous ones. If the current partial assignment leads to a conflict, it backtracks and tries a different variable assignment.

SAT solver's efficiency depends significantly on the heuristics used for variable and value selection. Selecting variables that lead to faster pruning reduces the search space and improves performance. For MaxSAT, finding an initial solution quicker allows for an upper bound that can further be used to restrict the search to only the solutions below the upper bound [13].

There are many different heuristics that are used within the SAT solvers. They can be classified into two categories: general-purpose heuristics and problem-specific heuristics. The former are most widely used and are shown to perform well for a variety of different SAT instances. Whereas the latter are not as widely used, even though they can lead to better performance [14].

This chapter introduces the heuristics whose performance has been measured. First, the baseline general-purpose Variable State Independent Decaying Sum (VSIDS) SAT heuristic is introduced, and then the JSOCMSR-specific variable selection heuristic is described.

5.1 VSIDS Heuristic

Variable State Independent Decaying Sum (VSIDS) is a general-purpose SAT heuristic for variable selection. It has been first introduced by Moskewicz et al. in 2001 [11]. Since then it has been shown to be effective for a variety of different SAT problems and is implemented by many state-of-the-art SAT solvers [15]. For these reasons, this heuristic has been chosen as the baseline for the performance evaluation.

VSIDS operates by keeping track of a dynamically-changing weight for each variable. The variable weight is increased when it is involved in a conflict. Then the solver needs to make a decision on what variable to split on next, the one with the highest weight is chosen. This way the variables that lead to conflicts are selected first, allowing for more frequent pruning of the search space. Besides increasing the weights of variables, VSIDS also periodically decreases the weights of all variables. This allows for more exploration and helps to prevent getting stuck on a small subset of variables [10].

5.2 Variable Selection Heuristic for JSOCMSR

In order to improve the performance of the solver, a greedy variable selection heuristic has been applied. The heuristic provides a static variable ordering based on the potential start times and overall job durations. This means that the solver

tries to schedule longer jobs first by putting them at the earliest available time slots.

The idea of scheduling longer jobs first comes from a well-known Nawaz-Enscore-Ham (NEH) heuristic for flow-shop scheduling. NEH greedily creates a schedule by iterating over the jobs based on their length, building an initial solution by adding the next job at every possible position, and picking the one with the lowest makespan [12]. This heuristic, despite its simplicity, has been shown to provide good initial solutions and is often used as a subroutine in cutting-edge algorithms for solving flow-shop scheduling [5].

In addition to employing the greedy heuristic, it has also been combined with VSIDS. This means that the initial weights of the variables are determined by the greedy variable ordering heuristic and then later dynamically modified by VSIDS during the solving process. This approach combines the dynamic search space exploration of VSIDS with problem-specific information, by providing an initial estimation of the variables' importance.

6 Experimental Setup and Results

Experiments have shown the effectiveness of the problem-specific value ordering heuristic. It is able to provide initial solutions much quicker than a general VSIDS heuristic which is especially beneficial for larger instances.

This section first describes the dataset that has been used in the experiments. Then it gives an overview of how the experiments were conducted. And lastly, it discusses the results of these experiments.

6.1 Dataset

In the experiment, a dataset based on the one compiled by Horn et al. [6] has been used. In this dataset, the number of secondary jobs considered is 2,3, and 5 to mimic the real-world application of the problem in cancer treatment. The dataset consists of 2 types of problem instances: balanced and skewed. The balanced dataset was generated by sampling the pre-processing, common resource usage, and post-processing times uniformly from $U(1, 1000)$. The usage of the secondary resource is also balanced, with every job being equally likely to require one of the secondary resources. For the skewed, the usage of common resource was sampled from $U(1, 2500)$ instead, making it a bigger bottleneck. Additionally, in the skewed instances around 50% of jobs require the same secondary resource and the other jobs are equally likely to require any of the other remaining secondary resources.

The dataset has been modified by scaling it down to make it feasible to encode it as a SAT instance. Since the encoding requires accounting for every possible start time of a job, the number of variables and clauses scales linearly with the maximum bound on the makespan. This maximum bound depends on the total duration of all the jobs. This means that encoding problem instances with longer average job durations requires significantly more variables and clauses. The original dataset has a high average job length of 1500 time units for the balanced and 2250 time units for the skewed dataset. The dataset has been scaled down in such a way that the average job length is 10 for the balanced and 11.25 for the skewed dataset by dividing the durations of the jobs by 150 and 200 respectively. Due to the scaling, and additionally due to the source code for the state-of-the-art algorithm by Horn et al. [6] not being publicly available, no direct comparison to its performance has been made. All the further discussion focuses on comparing the performance of the SAT solver under different heuristics.

Because of the computational constraints, it was decided to limit the original data set to 20 instances with similar configurations, meaning that they had the same number of jobs, number of secondary resources, and type. It was decided to focus on the more computationally complex instances that had the number of secondary resources set to 3 and 5.

Overall, each evaluation consisted of 480 instances. From which there were 24 unique configurations that are made of the following combination of factors:

1. Number of jobs: 10, 20, 50, 100, 150, 200.
2. Number of secondary resources: 3 and 5.
3. Type of the dataset: balanced and skewed.

6.2 Experiments

The encoding has been done separately from the main solving process in Java. Since the same encoding was shared between the tests of different heuristics, the speed of the encoding process has not been measured. For solving MaxSAT the Rust version of Pumpkin solver developed by the project supervisors was used. The solver was compiled using the Rustc compiler and the tests were run on the DelftBlue supercomputer's compute cluster with 2x Intel XEON E5-6248R 24C 3.0GHz CPUs and 6GB of memory dedicated for each problem instance [4].

Each run was given a time limit of 60 seconds. The encodings were done separately beforehand and are not accounted for in this limit, as both the baseline and the augmented versions share the same encoding. Throughout the runs, the makespans of all the best-so-far solutions have been recorded along with how much time has passed since the run started. Along with this information, the final state of the solution was noted. The states are:

1. Optimal - The solution is optimal
2. Satisfiable - A solution was found, but it is not guaranteed to be optimal
3. Unknown - No solution has been found

For each batch of 20 instances with similar configuration the following statistics have been gathered:

- $\bar{m}$ - average final makespan. If no makespan was found it is set to T_{max}.
- t_{first} - average time in seconds to an initial solution. If no solution was found it is set to 60 seconds.
- t_{best} - average time at which the best solution was found. It is set to 1 if no solution was found.
- $\%_{best}$ - the percentage of instances that are equal to the best value found by any heuristic.

- s - statuses of jobs in the batch. It is in the form x/y/z, where x is the number of optimal instances found by the heuristic, y is the number of satisfiable solutions and z is the number of unknown solutions.

In total 2 different configurations have been tested. These are:

- Baseline - VSIDS with no heuristic augmentation
- Variable ordering - only the greedy variable ordering heuristic
- Variable ordering + VSIDS - greedy variable ordering heuristic augmented with VSIDS

6.3 Results

The full results featuring the previously mentioned statistics are given in the table 1. The rest of the subsection discusses the results in more detail and highlights the most noteworthy observations.

Baseline

The baseline has performed well on smaller instances with a number of jobs n less or equal to 50. The table 2 provides an overview of the state of solutions to the problems. The instances are grouped by the number of jobs to make the table more readable. Both types and the number of secondary resources play a less significant role in the overall quality of the solution, though if the instance is skewed and/or has 5 secondary resources, it is slightly more likely to be satisfiable or unknown compared to similar problem instances. Amongst all the instances, only some instances with $n = 10$ are solved to proven optimality. For all instances with n in the range from 20 to 100, a solution is found, and for larger instances sometimes no solution is found within the time limit.

By examining the figure 3 showing the progression of the best-so-far solution for instances with $n = 150$ it can be seen that a significant portion of time is spent on discovering the initial solution. This leads to the solver often not finding any solutions for larger instances and thus not allowing it to improve upon the initial solutions, as there are none.

Variable Ordering Heuristic

Two heuristically augmented experiments have been conducted. One relies solely on picking the variables in the static order determined by the variable-ordering heuristic and the other additionally modifies the initial order by employing VSIDS. From the table 1 it can be seen that the version augmented with VSIDS generally displays better results, but it is still valuable to examine the plain version, as it provides insight into what is its contribution.

For the smaller instances with the number of jobs less or equal to 50, the addition of the heuristic does not have a significant impact on the performance of the solver. The final average makespans between VSIDS and VSIDS + heuristic are quite similar as can be seen from figure 4, though the plain heuristic version does not perform that well and is outperformed by both other versions.

Compared to the baseline, the main advantage of the heuristic is that it is able to quickly find an initial solution,

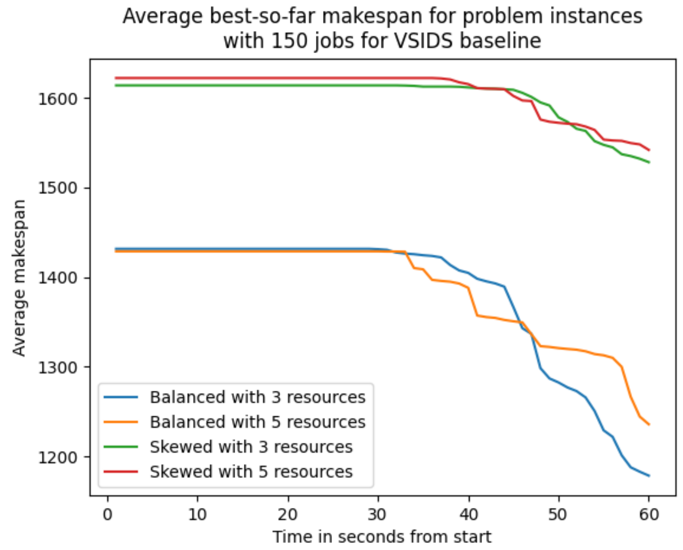

Figure 3: Average best-so-far makespan for different problem instances with the number of jobs equal to 150 when solved by the baseline solver.

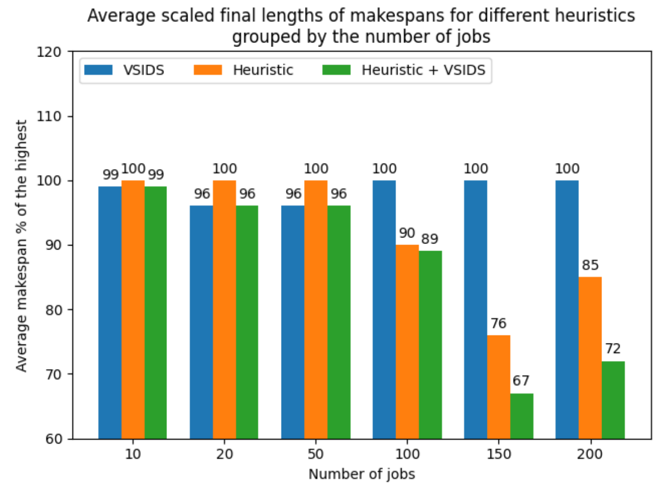

Figure 4: Average scaled final makespan achieved by different heuristics grouped by the number of jobs. 100% for each number of jobs corresponds to the highest final makespan and the average makespans achieved by other heuristics are scaled proportionally.

resulting in getting a satisfiable solution to all the problem instances in the dataset. The variable ordering heuristic points the solver toward the core variables of the problem that represent the start times of jobs. This provides a good initial solution that is then improved upon during the rest of the search.

For larger instances with 100 and more jobs, the heuristically augmented versions outperform the baseline model significantly, resulting in smaller average makespans as can be seen from the figure 4. From figure 5 it can be seen that for bigger instances with 150 jobs, an initial solution is found significantly quicker in only a couple of seconds compared to the baseline that takes more than 30 seconds to find an initial

			Baseline VSIDS					Heuristic					Heuristic + VSIDS				
Type	n	m	$\bar{m}$	$\%_b$	s	t_f	t_b	$\bar{m}$	$\%_b$	s	t_f	t_b	$\bar{m}$	$\%_b$	s	t_f	t_b
b	10	3	**52.7**	**100**	19/1/0	1.0	1.1	**52.7**	**100**	18/2/0	1.0	1.1	**52.7**	**100**	19/1/0	1.0	1.0
b	20	3	**91.5**	**95**	0/20/0	1.2	11.9	97.1	20	0/20/0	1.0	10.7	91.8	80	0/20/0	1.0	7.8
b	50	3	222.0	**60**	0/20/0	7.5	44.0	238.3	0	0/20/0	1.1	15.4	**221.6**	55	0/20/0	1.0	43.0
b	100	3	473.9	25	0/20/0	23.2	56.8	464.1	5	0/20/0	1.9	28.9	**450.7**	**80**	0/20/0	1.6	49.0
b	150	3	1178.4	0	0/15/5	46.4	44.2	751.1	15	0/20/0	3.0	51.4	**687.7**	**90**	0/20/0	2.6	39.4
b	200	3	1841.9	0	0/7/13	55.5	21.6	1539.8	20	0/20/0	4.8	59.9	**1215.5**	**80**	0/20/0	4.2	53.6
b	10	5	**44.1**	**100**	15/5/0	1.0	1.0	44.4	85	13/7/0	1.0	7.7	**44.1**	**100**	15/5/0	1.0	1.5
b	20	5	**84.7**	**80**	0/20/0	1.0	7.6	87.7	10	0/20/0	1.0	9.7	**84.7**	**80**	0/20/0	1.0	4.8
b	50	5	**213.6**	**90**	0/20/0	5.3	32.8	219.1	0	0/20/0	1.0	4.8	213.9	65	0/20/0	1.0	35.5
b	100	5	512.9	0	0/20/0	26.6	55.8	443.9	35	0/20/0	1.8	32.0	**439.6**	**80**	0/20/0	1.4	37.2
b	150	5	1235.8	5	0/15/5	51.2	44.6	887.8	15	0/20/0	3.1	59.9	**726.9**	**80**	0/20/0	2.8	44.0
b	200	5	1895.8	0	0/1/19	59.9	4.0	1486.0	30	0/20/0	4.8	57.5	**1178.7**	**70**	0/20/0	4.8	50.0
s	10	3	**78.8**	**95**	3/17/0	1.0	7.7	79.7	55	4/16/0	1.0	6.4	78.9	90	2/18/0	1.1	6.8
s	20	3	**152.9**	**85**	0/20/0	1.6	23.0	158.4	10	0/20/0	1.0	9.4	153.0	80	0/20/0	1.0	15.6
s	50	3	**368.6**	**70**	0/20/0	8.2	54.0	384.2	0	0/20/0	1.0	16.5	369.1	45	0/20/0	1.0	47.2
s	100	3	846.0	0	0/20/0	33.5	57.2	762.5	20	0/20/0	2.0	35.4	**758.5**	**85**	0/20/0	1.9	43.9
s	150	3	1528.4	0	0/15/5	50.1	44.1	1252.8	25	0/20/0	3.5	57.3	**1151.5**	**90**	0/20/0	4.0	42.2
s	200	3	2148.6	0	0/1/19	59.6	4.0	1966.5	15	0/20/0	6.0	60.0	**1698.0**	**85**	0/20/0	5.7	49.8
s	10	5	**73.2**	**100**	3/17/0	1.0	6.5	73.7	65	3/17/0	1.0	7.1	73.3	95	2/18/0	1.0	4.6
s	20	5	148.8	55	0/20/0	1.1	18.7	153.3	0	0/20/0	1.0	3.6	**148.3**	**75**	0/20/0	1.0	20.9
s	50	5	375.1	**55**	0/20/0	8.8	49.0	380.7	5	0/20/0	1.0	10.9	**374.6**	**55**	0/20/0	1.0	48.0
s	100	5	867.5	0	0/20/0	36.5	57.8	748.3	**70**	0/20/0	2.1	35.5	**748.1**	50	0/20/0	2.0	40.6
s	150	5	1542.1	0	0/8/12	54.5	23.2	1250.9	10	0/20/0	3.5	59.6	**1113.0**	**95**	0/20/0	3.2	38.4
s	200	5	2168.8	0	0/1/19	59.9	4.0	1818.0	**55**	0/20/0	5.8	57.5	**1736.2**	45	0/20/0	5.8	52.8

Table 1: Full statistics on the performance of the baseline compared to the heuristically augmented versions by problem type, number of jobs n, and number of resources m. The objectively best values are bolded.

Number of jobs	#Optimal	#Satisfiable	#Unknown
10	40	40	0
20	0	80	0
50	0	80	0
100	0	80	0
150	0	53	27
200	0	10	70

Table 2: Final solution states for baseline experiment grouped by the number of jobs.

solution as shown in the figure 3. This is not only specific to bigger instances but applies to all instances as seen from the table 1 by examining the average times at which the initial solution is found (t_f).

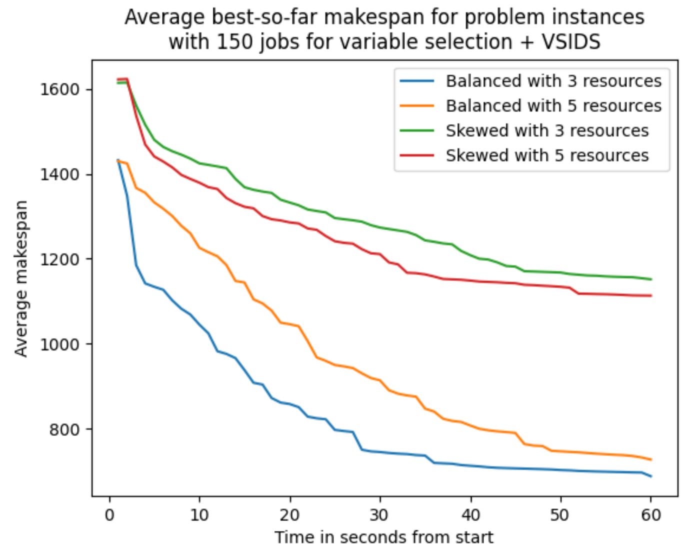

Figure 5: Average best-so-far makespan for different problem instances with the number of jobs equal to 150 when solved by the solver with static variable ordering and VSIDS.

7 Responsible Research

Throughout the project, a lot of effort was put into making the experiments repeatable. The encoding, heuristics, and the used datasets are described in full detail. Additionally, the related code used for encoding the problem is publicly available for inspection at "https://github.com/Artjom-Pugatsov/JSOCMSR_SAT".

Since in the result analysis problem instances with different parameters are often grouped together for brevity, a full breakdown of case-by-case results is given in a separate table 1. This is done to make the full results fully transparent

and let the reader themselves make sure that the conclusions about the performance have not been supported by cherry-picked data.

Moreover, it is worth mentioning that no sensitive data has been processed during this project. Even though the original dataset was inspired by cancer treatment scheduling, the actual dataset was randomly generated to only mimic the characteristics of real-world data. This means that it has no connections to any actual medical data.

8 Conclusions and Discussion

JSOCMSR is a relatively new problem. This research has taken a different approach to solving this problem by modeling it in terms of SAT and then applying a value selection heuristic augmented by VSIDS. The value selection heuristic first orders variables based on job length and start time and then employs VSIDS to additionally provide a dynamic variable ordering.

As shown in the evaluation of the experimental results, the developed value selection heuristic has demonstrated its effectiveness compared to a more general heuristic. The main advantage of the heuristic is quickly providing an initial solution which is thereafter improved upon by the VSIDS branching heuristic. It has significantly enhanced the performance of the SAT solver on the bigger instances while displaying similar performance for the smaller instances.

Despite the relative simplicity of the heuristic, it has demonstrated significant performance improvement for the SAT solver. This result provides further evidence of the viability of future research into problem-specific SAT heuristics.

9 Future Work

This research has centered around testing the impact of a variable ordering heuristic on the performance of the SAT solver. Thus in the experiments, the encoding of the problem has remained the same throughout all the tests. In the future, it could be of interest to check different encodings. One important parameter for the encoding is the maximum start time T_{max}. This parameter determines the latest time at which a job can be scheduled. Varying this parameter can drastically change the number of variables and clauses present in the encoding. The T_{max} that was used in the encoding is a straightforward sum of all the lengths of jobs. This provides a robust upper bound that does not make any assumptions about the underlying problem instance. For future research, it could be of interest to create additional heuristics for getting a lower bound on this parameter. One such approach could be to use a heuristic for calculating a smaller upper bound on this value by finding an initial solution before the problem is encoded.

When JSOCMSR was originally proposed, it was also modeled as a constraint satisfaction problem (CSP) [6]. This model has not been augmented with any additional JSOCMSR-specific heuristics when it was evaluated. It could be of interest to apply additional problem-specific heuristics to this model to further research their application for CSP solvers.

References

[1] Kenneth R. Baker and Dan Trietsch. *Principles of Sequencing and Scheduling*. Wiley Publishing, 2009.

[2] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[3] Jiatong Cai, Yating Su, Xixi Yang, Jionghao Min, and Yong Lai. A new sat encoding scheme for exactly-one constraints. *Journal of Physics: Conference Series*, 1288(1):012035, aug 2019.

[4] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 1). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.

[5] Luis Fanjul-Peyro and Rubén Ruiz. Ruiz, r.: Iterated greedy local search methods for unrelated parallel machine scheduling. european journal of operational research 207, 55-69. *European Journal of Operational Research*, 207:55–69, 11 2010.

[6] Matthias Horn, Günther Raidl, and Christian Blum. Job sequencing with one common and multiple secondary resources: An a∗/beam search based anytime algorithm. *Artificial Intelligence*, 277:103173, 09 2019.

[7] Matthias Horn, Günther R. Raidl, and Elina Rönnberg. A* Search for Prize-Collecting Job Sequencing with One Common and Multiple Secondary Resources. *Annals of Operations Research*, 302(2):477–505, July 2021.

[8] Thomas Kaufmann, Matthias Horn, and Günther R. Raidl. A variable neighborhood search for the job sequencing with one common and multiple secondary resources problem. In Thomas Bäck, Mike Preuss, André Deutz, Hao Wang, Carola Doerr, Michael Emmerich, and Heike Trautmann, editors, *Parallel Problem Solving from Nature – PPSN XVI*, pages 385–398, Cham, 2020. Springer International Publishing.

[9] Chu Min Li and Felip Manya. Maxsat, hard and soft constraints. In *Handbook of satisfiability*, pages 903–927. IOS Press, 2021.

[10] Jia Hui (Jimmy) Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. *CoRR*, abs/1506.08905, 2015.

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. *Proceedings - Design Automation Conference*, pages 530–535, 2001. 38th Design Automation Conference ; Conference date: 18-06-2001 Through 22-06-2001.

[12] Muhammad Nawaz, E Emory Enscore Jr, and Inyong Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95, 1983.

[13] Knot Pipatsrisawat, Akop Palyan, Mark Chavira, Arthur Choi, and Adnan Darwiche. Solving weighted max-sat problems in a reduced search space: A performance analysis. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):191–217, 2008.

[14] Jussi Rintanen. Planning as satisfiability: Heuristics. *Artificial Intelligence*, 193:45–86, 2012.

[15] Karem A Sakallah et al. Anatomy and empirical evaluation of modern sat solvers. *Bulletin of the EATCS*, (103):96–121, 2011.

[16] Jack A. A. van der Veen, Gerhard J. Woeginger, and Shuzhong Zhang. Sequencing jobs that require common resources on a single machine: A solvable case of the tsp. *Mathematical Programming*, 82(1):235–254, Jun 1998.