Why Midas would be a terrible secretary:

Using a greedy approach to enhance SAT for the Preemptive Resource-Constrained project scheduling problem with set up time

George Hellouin de Menibus[1]

Supervisor(s): Dr Emir Demirović[1]**, Maarten Flippo**[1]**, Konstantin Sidorov**[1]

[1]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: George Hellouin de Menibus
Final project course: CSE3000 Research Project
Thesis committee: Dr Emir Demirović, Maarten Flippo, Konstantin Sidorov , Jérémie Decouchant

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

Abstract

This paper presents a new greedy heuristic to extend SAT Solvers when solving the Preemptive resource-constrained project scheduling problem (PRCPSP-ST). The heuristic uses domain-specific knowledge to generate a fixed order of variable selection. We also extend previous work into encoding PRCPSP-ST by providing an alternative upper bound. The heuristic was tested against VSDIS on the J12 dataset. These experiments show that it performed, on average, six times slower than VSDIS.

1 Introduction

Satisfiability (SAT) solvers provide great versatility in solving any problem expressed as a set of Boolean clauses. Once encoded, no knowledge of the problem is needed for efficient searching. Starting 20 years ago, efficient and scalable algorithms for SAT have proven useful when dealing with NP-Hard problems[1]. Any optimization to the solver is an optimization to all. Interestingly, Solvers have been found to perform "unreasonably well"[2] for real-world problems and are deployed in many industrial processes, such as the automotive industry[3].

However, it comes at a cost. The solver only takes advantage of the knowledge of the problem space in the encoding. Encoding is the method used to feed information to the solver. If the encoding cannot capture relations or other knowledge, they don't contribute to finding a solution. Heuristics such as "schedule early" or "minimize lateness" are difficult to transmit.

This information loss raises the question at the heart of our investigation: How do we add domain-specific information to a SAT solver? Would a SAT solver with problem-specific knowledge outperform one without?

To investigate this potential improvement, we selected the Resource Constrained project scheduling problem with activity splitting and setup time (PRCPSP-ST). It is a derivative of the generic Resource Constrained project scheduling problem (RCPSP). This general scheduling problem concerns projects with precedence and resource requirements, and its subproblem adds preemption and setup time.

For a concrete example, imagine a factory producing cars. Each machine has a specific power draw, and the factory can only output a fixed amount of electricity every hour. The factory can not power all devices at the same time. Furthermore, parts may require others to be assembled before being built. A window can only be completed if its frame has been created. We can preempt work, meaning it can be stopped before completion to be finished later. Each preemption pays a cost called setup time.

Numerous works on scheduling, including with setup time exist. We can split previous work into two approaches: Custom algorithms for scheduling and SAT modelling with some heuristics. Mario Vanhouke and Jose Coelho have published much[4]–[7] in the domain of Resource-constrained project scheduling, including with setup time, but their work concerns domain-specific algorithmic. Combining domain-specific heuristics and SAT exists, such as [8] and [9], but they don't look at this sub-problem. Lastly, the work of [10] will be drawn upon for modelling the problem in SAT, but it does not implement a domain-specific heuristic.

We developed a naive greedy approach. Instead of the dynamic variable ordering found in the current state-of-the-art solvers, we implemented a static arrangement computed alongside encoding. This heuristic relies on reducing the problem (removal of resource constraint) and sorting the variables according to start time and size. We compared this to Variable State Independent Decaying Sum (VSIDS) over the J12 dataset. Our approach was between 3 to 10 times slower than VSIDS.

This paper is organized as follows. Section 2 develops on some of the previous works that influenced this paper. Section 3 describes both PRCPSP-ST and SAT solvers. Following it, Section 4 details our efforts to extend previous work and the development of our heuristic. Next, Section 5 explains the experimental setup as well as the results. Section 6 dives into the steps we took to make our research reproducible. A discussion of results can be found in Section 7. Lastly, the conclusion and future works are in Section 8

2 Related Work

This section discusses related work in scheduling, heuristics, and encoding, diving into some specific papers and how they inspired this one.

M. Vanhoucke and J. Coelho[5] proposed a fascinating approach to solving PRCPSP-ST. Their domain-specific algorithm has two repeating steps: A SAT solver that generates correct schedule permutations and a genetic algorithm that finds the optimal schedule for that permutation. Their approach combines the strengths of global search, the solver and local search, the genetic algorithm. However, they do not compare their algorithm with current state-of-the-art scheduling, and therefore it is hard to tell what the effectiveness of their algorithm is. To avoid this drawback, we elected to compare our heuristic to VSIDS, the current state-of-the-art heuristic for SAT.

J. Vermeulen [10] proposes an encoding to translate any PRCPSP-ST problem into wncf, allowing the solver to propose a potential schedule. The main drawback was its low satisfiability for some data instances, notably the J30 instance. Scheduling should always be satisfactory if the assumptions of encoding are not broken. It is always possible to execute projects sequentially. This encoding was replicated for this paper, and the main steps can be found later in the paper. Some corrections to its methods can also be found later in the paper, though their effect on satisfiability is hard to tell, given that we used a different data set.

Greedy approaches for scheduling are well known and actively studied[11]. The work of [12] uses a greedy heuristic to solve RCPSP, the parent problem of PRCPSP-ST. The greedy heuristic is used to avoid enumerating all possible solutions and retaining only the promising ones. While being very different in approach to SAT, having the heuristic guide the search for promising solutions was the primary inspiration behind our development.

3 Formal Problem Description

This section aims to explain all the terminology related to PRCPSP/ST and briefly explain SAT solvers. It builds primarily on the works of [5].

3.1 RCPSP

To define PRCPSP-ST, we first need to define its parent problem, RCPSP.

Formally, RCPSP is represented as a graph $G(N, A)$ where N and A are:

- N, the Set of nodes which each represent a project
- A, the edges, representing the pairs with a finish-start antecedent relationship with time lag 0.

We assume $G(N, A)$ to be acyclic.

We use R to denote the Set of renewable resources which projects will use. Each resource in R has a capacity, a_k, $k = 1..|R|$.

Each project has a deterministic duration, d_i and requires some renewable resource in set R. We express this as $r_{i,k}$. We assume that $r_{i,k} \leq a_k, i \in N, k \in R$. No individual project has a consumption above availability.

Project	Antecedents	Duration	Resource Usage
A		2	0 1
B	A	2	1 1
C	A	2	1 0
D	C	2	0 1
E	B, D	0	0 0

Table 1: Set of example projects

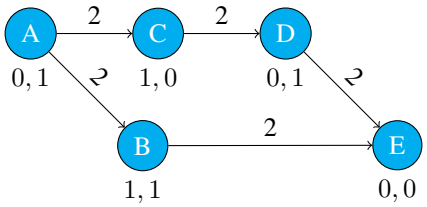

Figure 1: Graph Representation of Table 1

Table 1 shows an example set of projects. A graph representation can be seen in figure 1.

Once scheduled, activities can be represented with a Gnatt chart, as seen in figure 2. We can see the parallel work being done on B and C as they are executed at the same time. Given the properties of antecedent relations, projects can start when antecedent projects end. Thus, this schedule is done by time 6.

However, this example does not account for resource usage. Resource constraint prevents perfect parallelization of all tasks, and is why this problem is NP-hard. Section 4 has a deeper exploration of this.

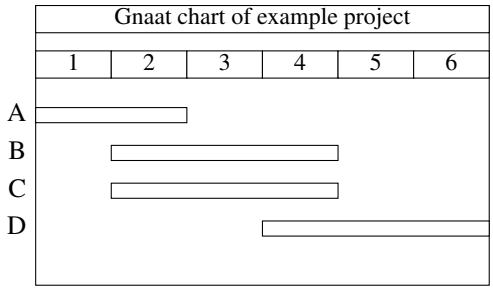

Figure 2: Gnatt chart of possible schedule for example projects

3.2 PRCPSP-ST

PRCPSP/ST is a strongly NP-hard algorithmic problem[13] with the same characteristics as RCPSP. A set of projects, N, and a set A represent antecedent relationships among projects. The enactment of projects requires the use of some amount of renewable resources.

The difference stems from preemption and setup time. When projects are pre-empted, they are stopped and started at some later time. Preemption incurs a setup time penalty, so the second segment of the project would take longer than its duration had it not been preempted. The objective is to minimize the makespan, the time between the start of the first project and the end of the last project.

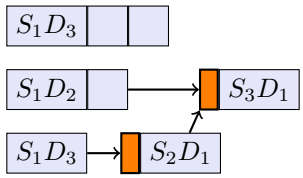

Figure 3: Possible Ways to preempt a Project of duration 3. Orange represents the set up time cost

To model preemption and activity split, [5] introduced the concept of "segments". They correspond to potential splits of a project. In the words of M. Vanhoucke and J. Coelho[5], "[...] an activity segment $S_x^i D_y$ is any integer part of an activity with duration d_i for which $x \in \{1...d_i\}$ and $y \in \{1...d_i - x + 1\}$." Figure 3 shows a visualization of a project of duration three. Segments are denoted $S_x^i D_y$, for a segment belonging to project i, starting at point x and having duration y. For a project of duration d_i, there exist $\frac{d_i * (d_i - 1)}{2}$ segments and $2^{d_i - 1}$ possible combinations that can recreate the original project.

When multiple projects are introduced, and we need to consider multiple combinations for each project, the problem grows dramatically. If all projects have the same duration, there will be $(2^{d_n})^n$ possible combinations of segments to schedule.

Many types of setup times have been proposed[5], but for this paper, we will only look at fixed setup times. Having a dynamic setup time, though closer to reality, is beyond the scope of this research. Set up time for segment $S_x^i D_y$ is denoted $t_{S_x^i D_y}$. The fixed setup cost is denoted f

$$t_{S_x^i D_y} = \begin{cases} 0 \text{ if } x = 1 \\ f \text{ if } x > 1 \end{cases}$$

Segments play a vital role in the encoding developed in Section 3.4.

3.3 SAT solvers

SAT Solvers are a powerful modern tool used to deal with NP-Hard problems. Their versatility stems from the Cook-Levin theorem, which states that every NP problem can be reduced to SAT in polynomial time[14], otherwise known as NP-completeness.

To solve these NP-hard problems, the solvers refine brute force search. First laid out in 1961, the Davis–Putnam–Logemann–Loveland (DPLL) algorithm searches the space in three repeating steps: Variable selection, value assignment and propagation. One of the Boolean variables is selected, given a value. In each constraint, it is substituted by the value DPLL has assigned. Then we propagate the value. Setting a variable X to true and propagating it is similar to saying, "If X is true, and X and Y are true, then Y must be true." Propogationing may arrive at conflicts, i.e. contradictions, which require backtracking. For example: If X is false, X Or Y is true, but Y is false because of other constraints, X cannot be false. In such cases, the algorithm backtracks to a previous decision and tries again.

SAT solvers have gone through many iterations, and modern solvers primarily rely on CDCL[15], conflict-driven learning. One of the current favoured implementations of CDCL is VSDIS. A full exploration of VSDIS is beyond the scope of this paper, as the reason it is so successful is still an area of active research[15].

Briefly summarised, VSIDS looks at the conflicts among variables and uses this to pick the next. Variables with high conflict tend to be selected first[15]. A priority queue (or similar data structure) keeps track of variable ordering, prioritizing based on weights. Weights are updated for variables present in clauses that get learned and decay as iterations continue. Despite being proposed twenty years ago, this remains the core approach to variable selection[16].

3.4 Encoding

We based our encoding on [10] and [5]. We begin by turning our schedule N into the enhanced schedule $N*$ by replacing each project with all possible combinations of its segments. Projects which had precedence relations pass those off to their segments. If Project X has Y as an antecedent relation, all segments $S_1 D_*$ have the last Y segments as antecedents. Lastly, we add a dummy start and end. All segments with no antecedents have a dummy start as an antecedent, and all tasks with no decedents have a dummy end as one. A more detailed explanation of this process can be found in [5]

To encode the enhanced schedule, we develop two variables to represent the problem. S variables represent the starting time of a project segment, and U variables, represent if a project is taking place at that time. Formally we can say:

Let $s_{j,t} \in \{0, 1\}$, where 1 means segment j is scheduled at time t.

Let $u_{i,t} \in \{0, 1\}$, where 1 means segment i is taking place at time t.

Let d_j represent the duration of project or segment j.

Let N represent the set of projects and $N*$ represent the set of segments. Let R denote the set of resource capacities.

Let $A(x, y)$ denote that segment x is the antecedent of y. Let es_i denote the earliest starting time of segment i. The shortest distance between a project and the dummy start project is the earliest start time. Let ls_i denote the latest starting time of segment i. The method used to obtain this value is discussed later in section 4. Unless otherwise specified, all time values t for segments rest between es_i and ls_i.

Hard Clauses are clauses that must evaluate to be true. These variables are used in the following hard clauses:

- Completion Clauses ensure that the combination of segments scheduled represents the entire project. We use the subset $C*_{i,l}$ to group all segments that contain time segment l of i. They are expressed as:

$$l \in \{0...d_j\}; i \in C_{j,l} \bigvee s_{i,t}$$

- Consistency Clauses ensure that u variables are set if the corresponding start time is set.

$$l \in \{t...d_i\}; i \in N*; \neg s_{i,t} \vee u_{i,l}$$

- Set precedence clauses ensure that no project is started before the completion of all its precedents

$$(j, i) \in A; \neg s_{i,t} \bigvee_{l \in es_j ... es_i - d} s_{j,l}$$

- Resource Clauses. Using a Pseudo-Boolean weighted sum, we can capture that at no point in time should the resource usage surpass its limit. Explaining the transformation into CNF is beyond this paper's scope, but more information can be found in [17].

$$k \in R; \sum_{i=1}^{n} u_{i,t} * r_{i,k} \leq a_k$$

To capture optimally, we use soft clauses. Depending on the encoding, these clauses have a weight, which the solver will try to minimize or maximize. Since we are trying to minimize the makespan, we use the start time of the dummy end to measure this.

$$i = \text{ dummy end}; t \text{ as weight}; s_{i,t}$$

4 Original Contribution

We based our encoding of PRCPST on [10] and [5]. We had to adapt the upper bound used because it proved ill-suited to the resource constraints. We then developed a heuristic based on this encoding.

4.1 Extension

A vital component of the encoding is the latest start time. It determines the upper bound used in variable generation take. The upper bound is necessary as we cannot encode infinite

time. Increasing the upper bound dramatically increases the number of variables, clauses and search space.

By changing the upper bound, we adopted the encoding more closely to suit PRCPST. The original paper[10] favours a critical path method, often used in project scheduling. The critical path corresponds to the longest path in the Directed Acyclic Graph (DAG), representing project precedence relations. An illustration can be seen in Figure 4. However, because of the resource constraint, some tasks can monopolize their scheduled time. This forces linear project execution and means a schedule may exceed the limit predicted by the critical path. An example of such a schedule can be seen in FIG 5.

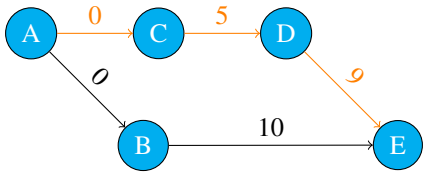

Figure 4: Example of Dependency graph with critical path analyse. The orange line shows the critical path(Length of 14)

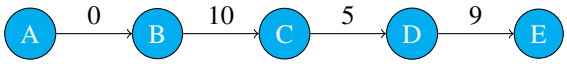

Figure 5: Order of projects if executed sequentially because resource cap is 10

We instead used the horizon, which is the sum of all durations. On a J30 instance, the critical path would be around 30-40. The horizon would be 120. The challenges this caused us can be found in Section 5.

4.2 Heuristic development

Part of the challenge of developing a heuristic for a SAT solver rests on what part of the solver the heuristic should intervene at. There are several possible interception points. A heuristic could inform the solver by providing initial starting values for specific variables. A heuristic could also act in the encoding, such as the previously mentioned critical path method providing an upper bound to values. We choose to focus on variable selection, as it most easily maps back to the domain.

As explained in section 3, current approaches favour dynamic variable selection. This provides flexibility and adaptability. Variables can be picked because they often cause conflict or because they appear a lot, and giving them a value at the start ensures high propagation.

This flexibility comes at the cost of greater complexity and needing to reshuffle the order after each conflict. We used a more straightforward approach since few previous works combine domain-specific heuristics with domain-specific knowledge. Fixed order variable traversal can be computationally cheap and potentially produce a viable solution, which can then converge to the optimal solution.

Current efforts in Scheduling heuristics focus on approaches which are opposite to Sat Solvers. Genetic algorithms [18] are a form of local search, fundamentally different from the solver's exhaustive search. Marrying these two approaches is beyond the scope of this paper.

For these reasons, we favoured a greedy heuristic to explore this new approach. The intuition is as follows: u variables are always defined in relation to s variables in consistency clauses. Previous research has shown situations where splitting, especially for smaller tasks, is worse than no split [5]. Therefore segments with no set-up time should be attempted first, providing a baseline. Additionally, when comparing s variables, we want to prioritize early start times. Lastly, we want to attempt to schedule longer segments before shorter ones, as they are less likely to incur set-up time penalties in their descendants.

In Pseudocode, this is expressed as follows:

Algorithm 1 Heuristic order

```
procedure HEURISTIC ORDERING(S, U, Aux)   ▷ Lists for each variable type
    S ← Sort by key S                     ▷ Sort by set up time incr, start time inc, duration dec
    vars ← Append S U Aux
    return vars
```

This ordering is passed as a comment in the encoded file to the SAT solver. This ensures that the same file can be used by a solver with the heuristic and one without, and doesn't require significant changes to the functionality of the solver, only the order it visits variables.

Both of these approaches are left open as potential future work.

5 Experimental Setup and Results

This section details some of the challenges faced in generating the wcnf files, as well as the experimental setup and the results we obtained.

5.1 Hardware Limitations

This subsection elaborates on some of the technical challenges encountered during encoding and the steps we took to overcome them.

We used the Pumpkin Solver, currently in development by Dr. Emir Demirovi, Maarten Flippo, and Konstantin Sidorov It is written in Rust. Before heuristic development, we decided that the transformation from schedule files into WCNF should also be done in Rust to increase code reuse if the heuristic needed access to any structures. Rust's performance gains compared to other languages[19] were also seen as a comparative advantage. However, the specifics of Rust's lifetime and ownership model clash with Direct Acyclic graphs. To handle this, we deployed an arena, a memory structure that does not deallocate anything until it is destroyed.

The next challenge came from the translation of the resource clauses. Transforming Psuedo-Boolean into CNF required using an external library, as developing a maximum totalizer was beyond the scope of this paper. However, no

such library exists for Rust, and the maximum totalizer found inside the pumpkin solver proved challenging to extract. Instead, we use Rust's foreign function interface and call a Python library to handle this case. However, the Python call was being done inside the arena, and since Python has no manual memory deallocation, python instances would remain in memory until the end of the program.

Had we been dealing with RCPSP, that is, no pre-emption nor set-up time, this problem may have yet to surface. However, PRCPSP-ST has a significantly greater memory load, as each project generates d^2 segments and 2^d combination. u variables in the resources clause belong to one segment, not any individual project. Lastly, correcting the critical path method meant our upper bound was significantly higher than other encoding attempts. For the J30 instance, horizon produces a latest starting time above 100, and the critical path around 30. The amount of u variables is drastically more significant in our proposed encoding than in previous work.

Set-up time added another complicating factor. One scheduling file generates one wcnf file for each set-up time, which meant six files per original RCPSP instance for our chosen range.

Combining all this led to a significant setback: Memory leaking. The python call lived inside the arena and therefore was not deallocated, as references to it long outlived the code it was responsible for generating. As more files were generated, more fragments of previous generations lived on and rapidly clogged memory, halting the code once the memory was full.

5.2 Set up

We collected 90 instances from the J12 dataset. We selected this dataset due to the hardware constraints imposed by the upper-bound encoding. J30 proved too large and complex to encode in a reasonable time.

To gather data for PRCPSP-ST, we can use any representation of RCPSP and generate its segments and encoding. Our data sets came from the Project Scheduling Problem Library. However, because of the need to use a smaller dataset, and the lack of smaller instances for RCPSP, we instead transformed a Multi-Mode J12 instance into RCPSP. We did this by removing the additional modes and non-renewable resources.

We rebuilt the encoding in Rust to maximize performance and used it to generate six different encodings per file corresponding to different setup times. Five hundred forty file wcnf files were generated in total. After generating the files, We batch-processed them into the solver and collected results for each setup time. We focused on time taken and the number of optimal vs unsatisfiable solutions. Because of the parallelized nature of the workload manager, all eight concurrent threads had access to 32 GB of shared memory. They ran on an AMD Ryzen 5 3600 6-Core Processor at a base frequency of 3.6 GHz.

5.3 Results

Method	Avg solve time(s)	# of Timeouts	# of Errors
Greedy	1.07	5	54
VSIDS	0.162	0	54

Table 2: Processed Results of the Runs

Setup	Average Time (s)	Avg T for Opt	Avg T Unsat
1	0.77	0.193	0.04
2	0.70	1.01	0.04
3	1.07	0.557	0.05
4	1.11	0.590	0.04
5	1.29	0.793	0.05
6	1.46	1.00	0.05

Table 3: Metrics for heuristic

Setup	# Optimal	#UnSat	# Errors
1	40	26	23
2	61	23	6
3	64	18	7
4	67	16	6
5	70	13	6
6	70	13	6

Table 4: Overall Run info Heuristic

Setup	Average Time (s)	Avg T for Opt	Avg T Unsat
1	0.09	0.169	0.04
2	0.12	0.19	0.04
3	0.16	0.22	0.04
4	0.17	0.20	0.04
5	0.20	0.24	0.04
6	0.24	0.29	0.05

Table 5: Metrics for VSIDS

Setup	# Optimal	#UnSat	# Errors
1	41	26	23
2	61	23	6
3	65	18	7
4	68	16	6
5	71	13	6
6	71	13	6

Table 6: Overall Run info VSIDS

The error column corresponds to errors produced by the Pumpkin solver. Given that this solver is still in development, most of these were ”to do” style errors, with specific cases not being handled correctly.

6 Responsible Research

We have made the source code available at https://github.com/gdemenibus/SatHeuristic to increase transparency. The original J12 schedules and their wcnf encoding alongside the collection and work scripts are also hosted in this repo. A readme file also gives a high-level overview of the code and the splits.

To adequately represent the work that came before us, we tested the encoder to reproduce examples worked out by hand and worked out examples on more minor instances.

The potential for misuse of this research is practically none. This paper demonstrates that this greedy approach is ill-suited for this problem and should not be considered for enhancement.

7 Discussion

The results in table 2 show that the heuristic performed significantly worse. There is a factor of 6 between the time it took VSIDS and Greedy. Both produced errors on the same files. Therefore, we can isolate solver issues from the potential explanation. Both arrived at unsatisfiable solutions within similar timeframes, pointing to an error with the files producing impossible schedules (Such as not being a DAG).

There are several potential explanations for this significant divergence that merit further consideration.

The first is that the heuristic is a suboptimal choice for this problem. Other heuristics may better indicate what activity should be scheduled next. Our proposed heuristic is ignorant of the resource constraints and, therefore, may spend too long attempting to allocate jobs to slots which are "full." A heuristic that takes resource constraints into account may prove more useful.

The second is with the heuristic's static nature. The proposed greedy heuristic does not consider the problem's current state; it only dictates a fixed order of traversal. When designing the heuristic, we saw this as a positive, avoiding the need to compute weights at each decision point. However, VSIDS' dynamic approach, bumping high conflict clauses, gives it great flexibility. A more dynamic ordering may be more successful, not based on the boolean clauses but instead on what they represent.

It is also possible that domain-specific heuristics are poorly suited for scheduling. Current heuristics in scheduling focus on metaheuristics [20]. These usually involve genetic algorithms or tabu search, which are more general approaches adapted to scheduling. They also fulfill a different role than SAT, as they are local search algorithms, whereas SAT is an exhaustive search algorithm.

Lastly, VSIDS and other domain-blind methods may consistently outperform any domain-specific heuristic. These approaches are designed to search the problem space efficiently, and much research has gone into explaining why[16]. Variables represent the problem, but the efficient searching of variable space and domain space could be different.

8 Conclusions and Future Work

This paper has presented the key components to encoding PRCPSP-ST into conjunctive normal form, as well as an extension of previous work to capture the upper bound. We describe the greedy heuristic we developed for static variable ordering. Augmenting a SAT solver with this greedy heuristic is detrimental to performance, on average performing 6 times worse.

Future work could focus on providing more insight as to why, by collecting the number of decisions or a measure of how quickly good solutions are found. Additionally, there remain a number of schedules that produce unsatisfiable files. This would point to an error in the encoding, but the cause would have to be investigated.

Several approaches could be taken to add a dynamic element to this heuristic. Adding an element to the sort that is derived from the state of the solver, such as the number of conflicts or amount of previous propagations, and recomputing the order at every assignment, or every few assignments, could make it respond dynamically to the changing circumstances.

To merge it with VSIDS, this ordering could provide the original weights that VSIDS will modify. This approach risks providing very little impact, as the growth rate of weights in VSIDS is steep. VSIDS could be modified to reset weights to their original values every X iteration, ensuring the heuristic is tried repeatedly as more variable values are fixed.

References

[1] F. van Harmelen, F. van Harmelen, V. Lifschitz, and B. Porter, *Handbook of Knowledge Representation*. San Diego, CA, USA: Elsevier Science, 2007, ISBN: 0444522115.

[2] V. Ganesh and M. Y. Vardi, *On the unreasonable effectiveness of sat solvers.* 2020.

[3] C. SINZ, A. KAISER, and W. KÜCHLIN, "Formal methods for the validation of automotive product configuration data," *AI EDAM*, vol. 17, no. 1, pp. 75–97, 2003. DOI: 10.1017/S0890060403171065.

[4] J. Coelho and M. Vanhoucke, "An exact composite lower bound strategy for the resource-constrained project scheduling problem," *Computers & Operations Research*, vol. 93, pp. 135–150, 2018, ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2018.01.017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S030505481830025X.

[5] M. Vanhoucke and J. Coelho, "Resource-constrained project scheduling with activity splitting and setup times," *Computers & Operations Research*, vol. 109, pp. 230–249, 2019, ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2019.05.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054819301170.

[6] mario vanhoucke and josé coelho, "An approach using sat solvers for the rcpsp with logical constraints," *european journal of operational research*, vol. 249, no. 2, pp. 577–591, 2016, ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2015.08.044. [Online]. Available: https://www.sciencedirect.com/science/article/pii/s0377221715008000.

[7] M. Vanhoucke and J. Coelho, "A tool to test and validate algorithms for the resource-constrained project scheduling problem," *Computers & Industrial Engineering*, vol. 118, pp. 251–265, 2018, ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2018.02.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835218300378.

[8] J. Rintanen, "Planning as satisfiability: Heuristics," *Artificial Intelligence*, vol. 193, pp. 45–86, 2012, ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2012.08.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370212001014.

[9] J. Rintanen, "Heuristics for planning with sat," in *Principles and Practice of Constraint Programming – CP 2010*, D. Cohen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 414–428, ISBN: 978-3-642-15396-9.

[10] J. Vermeulen, *Resource-constraint project scheduling with task preemption and set up tomes by boolean satisfiability encoding and satisfiability (sat) solver*, , 2022.

[11] R. Pellerin, N. Perrier, and F. Berthaut, "A survey of hybrid metaheuristics for the resource-constrained project scheduling problem," *European Journal of Operational Research*, vol. 280, no. 2, pp. 395–416, 2020, ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2019.01.063. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221719300980.

[12] Y. Liu, J. Zhou, A. Lim, and Q. Hu, "A tree search heuristic for the resource constrained project scheduling problem with transfer times," *European Journal of Operational Research*, vol. 304, no. 3, pp. 939–951, 2023, ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2022.05.014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221722003897.

[13] J. Blazewicz, J. Lenstra, and A. Kan, "Scheduling subject to resource constraints: Classification and complexity," *Discrete Applied Mathematics*, vol. 5, no. 1, pp. 11–24, 1983, ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(83)90012-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0166218X83900124.

[14] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71, Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158, ISBN: 9781450374644. DOI: 10.1145/800157.805047. [Online]. Available: https://doi.org/10.1145/800157.805047.

[15] Carsten Sinz and Tomáš Balyo, *Practical sat solving*, https://baldur.iti.kit.edu/sat/files/2019/l08.pdf, Accessed: 2023-06-05, 2015.

[16] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki, *Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers*, 2015. arXiv: 1506.08905 [cs.LO].

[17] S. Joshi, R. Martins, and V. Manquinho, "Generalized totalizer encoding for pseudo-boolean constraints," in *Lecture Notes in Computer Science*, Springer International Publishing, 2015, pp. 200–209. DOI: 10.1007/978-3-319-23219-5_15. [Online]. Available: https://doi.org/10.1007%5C%2F978-3-319-23219-5_15.

[18] R. Ruiz, *State-of-the-art flowshop scheduling heuristics*, https://www.youtube.com/watch?v=F3Ykma1eqnY, Accessed: 2023–04-25, 2021.

[19] W. Bugden and A. Alahmar, "Rust: The programming language for safety and performance," *arXiv preprint arXiv:2206.05503*, 2022.

[20] R. Kolisch and S. Hartmann, "Experimental investigation of heuristics for resource-constrained project scheduling: An update," *European Journal of Operational Research*, vol. 174, no. 1, pp. 23–37, 2006, ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2005.01.065. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221705002596.