Augmenting VSIDS heuristic for the RCPSP/t by initializing activity values using domain-specific information

Tijs Lenssen[1]

Supervisors: Emir Demirović[1]**, Maarten Flippo**[1]**, Konstantin Sidorov**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

Abstract

The Variable State Independent Decaying Sum (VSIDS) heuristic is one of the most effective variable selection heuristics for Conflict-Driven Clause-Learning (CDCL) SAT solvers. It works by keeping track of activity values for each variable, which get *bumped* and *decayed* based on conflict analysis. These activity values usually start out arbitrarily at zero, which prompts the question if initializing these values can result in better performance.

This paper presents several adaptations of the VSIDS heuristic specialized for solving the Resource-Constraint Project Scheduling Problem with time varying resource availabilities and demands (RCPSP/t). The approach uses domain-specific knowledge to initialize the activity values of VSIDS with more relevant values. The experiments presented in this paper show that this domain-specific knowledge can indeed benefit the heuristic and can lead to better solve times, allowing the solver to find solutions for 17% more of the instances and find proven optimal solutions for 5% more instances of the PSPLIB data set.

1 Introduction

In the resource-constraint project scheduling problem (RCPSP), a set of activities have to be scheduled while respecting a set of precedence and resource constraints with the goal of minimising the makespan. The RCPSP and its variants find practical usage in the scheduling of a wide range of projects. Some examples are construction projects, medical research projects and aggregated production scheduling [1; 2; 3]. This, together with the fact that the problem has been shown to be strongly $\mathcal{NP}$-hard [4], has induced a lot of research into efficient algorithms to solve the problem and its extensions.

Several variants and extensions of the problem have been studied [5], one of which is the resource-constrained project scheduling problem with time varying resource availabilities and demands (RCPSP/t) [2], which is the subject of this paper. A foundation of research into the RCPSP/t has been laid in a case study by Hartmann [2]. In the paper, Hartmann discusses a formal mathematical model for the RCPSP/t and analyses a new heuristic algorithm to solve the problem. Later, Hartmann also analysed the applicability of different RCPSP heuristics to the RCPSP/t and adapted a genetic algorithm to the RCPSP/t [3]. More recently the problem was also included in [6], which discusses an SMT encoding of the problem. This encoding was the basis for a MAX-SAT formulation of the problem by Pleunes [7]. With this formulation, Pleunes has shown that a MAX-SAT solver [8] can be an effective tool for solving the RCPSP/t, outperforming the SMT-based approach for some large test instances.

These promising results are based on a MAX-SAT solver with the Variable State Independent Decaying Sum (VSIDS) heuristic [8], which is a variable selection heuristic for Conflict-Driven Clause-Learning (CDCL) SAT solvers [9; 10; 11]. It works by keeping track of the activity values for each variable, always choosing the variable with the highest activity value when a branching decision gets made. When a conflict arises it gets analysed and the activity values of related variables get *bumped*. Periodically all activity values also get *decayed*. These activity values usually start out arbitrarily at zero, which means the initial decisions are made essentially at random. This prompts the question if it is possible to initialise the activities with more relevant values to improve the initial decisions and achieve better performance.

In the past, there have been several papers that have suggested that domain-specific variations of VSIDS could achieve better results for some problems. For example [12] uses a similar technique for the RCPSP. It was able to improve the performance of the heuristic by employing a hybrid setup, combining VSIDS with a deterministic branching heuristic. First, it would run the deterministic branching heuristic then it would switch to VSIDS after some iterations.

This paper explores several domain-specific adaptations of the VSIDS heuristic for the RCPSP/t and compares their performance to the standard VSIDS heuristic. The augmented VSIDS heuristics work by setting the initial activity values of VSIDS based on properties from the RCPSP/t. Using data sets from the PSPLIB [13], we will show that we can construct an adapted heuristic that is able to achieve better results compared to the standard VSIDS heuristic.

With the augmented VSIDS heuristic we were able to find solutions for 17% more instances and certify the optimal solution for 5% more instances. We also show that the other similar methods of initialising the values achieve a similar improvement, with no significant difference between them.

We will start by giving a formal problem description of the RCPSP/t as well as the MAX-SAT formulation in section 2. In section 3 and section 4 we discuss how VSIDS works and the different heuristics that were implemented with the corresponding design choices that were made. Then, we discuss the experimental setup and the results in section 5 and section 6. Afterwards, the ethical aspects of the research are reflected on in section 7 and the results are analysed and compared to previous work in section 8. Finally, in section 9, we give a summary of the results as well as the prospects of possible future work.

2 Problem Description

This section covers the formal problem description of the RCPSP/t as well as the MAX-SAT formulation that is used by the solver. The RCPSP/t is an extension of the standard RCPSP. The problem definition in this paper is based on the version of the problem that was established in [2]. The MAX-SAT formulation will be based on the formulation from [7].

2.1 RCPSP/t

In the RCPSP, J tasks need to be scheduled, the tasks cannot be split up into multiple tasks (this is referred to as non-preemptive scheduling). The duration of a task j will be denoted as p_j. There are two types of constraints on the schedule: precedence constraints and resource constraints.

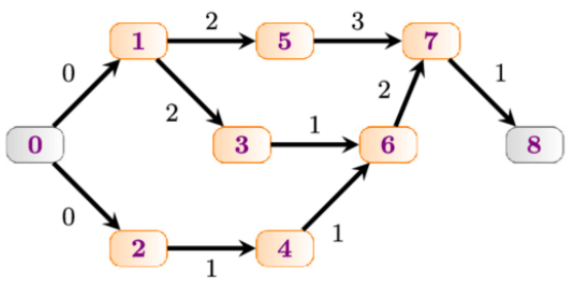

Figure 1: A precedence graph for a project with 7 tasks. Tasks 0 and 8 are added to mark the beginning and end of the project. An arrow between two tasks means one needs to be scheduled before another, with the value denoting the duration of the task. Adapted from [6].

The precedence constraints are formulated using predecessor sets P_j. Each task can have multiple predecessors, which means that for each predecessor, the predecessor needs to be finished before task j starts. There are two additional tasks that are included in the schedule, $j = 0$ and $j = J + 1$, which mark the start and end of the project. They both have a duration of 0 and precedence constraints in such a way that task 0 always has to be the first task, and $J + 1$ always has to be the last task. This means that the makespan of the project will be equal to the start time of task $J + 1$. In Figure 1, you can see an example of a precedence graph with 7 different tasks.

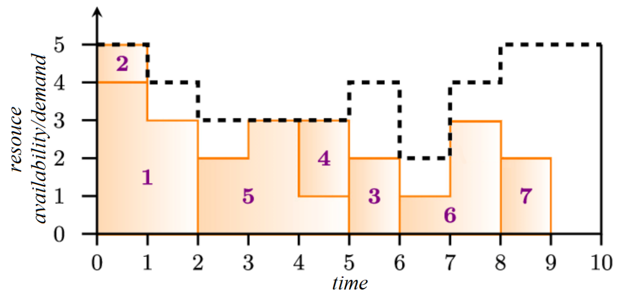

Figure 2: An optimal schedule for an instance of the RCPSP/t with one resource. The horizontal axis shows the different time slots and the vertical axis shows the amount of the resource that is available and demanded. The dotted line shows how much of the resource is available at any time, while the other shapes show the amount of the resource that is demanded by each task. Adapted from [6].

The resource constraints are based on K resources that can be required by tasks. Each resource has a certain capacity. In the RCPSP/t extension, the capacity of each resource can vary over time. We will denote the capacity of resource $k \in [1, K]$ at time $t \in [1, T]$ as R_{kt}. Because R_{kt} needs to be defined for the entire duration of the schedule, there is a predefined horizon T, which is the maximal time a schedule can take. If it is impossible to schedule all tasks in this time span, the instance will thus be unsatisfiable. The amount of resources each task requires also varies over time. The amount of resource k, task j requires will be denoted as r_{jkt} with $t \in [1, p_j]$. An example of an optimal schedule for the precedences of Figure 1 can be seen in Figure 2.

We will use $y_{jt} \in \{0, 1\}$ to denote that task j starts at time t iff $y_{jt} = 1$. This leads to the following mathematical model:

$$\min \quad \sum_{t=1}^{T} t \cdot y_{J+1,t} \tag{1}$$

s.t.

$$y_{jt} \in \{0,1\} \quad j \in [0, J+1], t \in [1, T] \tag{2}$$

$$\sum_{t=1}^{T} y_{jt} = 1 \quad j \in [0, J+1] \tag{3}$$

$$p_i + \sum_{t=1}^{T} t \cdot y_{it} \le \sum_{t=1}^{T} t \cdot y_{jt} \quad j \in [1, J+1], i \in P_j \tag{4}$$

$$\sum_{j=1}^{J} \sum_{q=0}^{p_j-1} r_{jkq} \cdot y_{j,t-q} \le R_{kt} \quad k \in [1, K], t \in [1, T] \tag{5}$$

The objective (1) minimizes the start time of the final task, thereby minimizing the makespan. Constraints (2) define the binary variables and constraints (3) make sure each task is scheduled exactly once. Finally, the precedence and resource constraints are enforced by (4) and (5) respectively.

2.2 Time window and lower bound

To reduce the search space, a preprocessing step is performed before the MAX-SAT generation, to determine the starting windows $STW(j) = [ES_j^*, LS_j^*]$ for each task j, consisting of the earliest and latest feasible start times (ES_j^* and LS_j^*). These are the earliest and latest times for which a task can be scheduled to create a feasible schedule. They also correspond with the earliest and latest feasible finish times ($EF_j^* = ES_j^* + p_j - 1$ and $LF_j^* = EF_j^* + p_j - 1$), which similarly are the earliest and latest time a task can still be running. The earliest start time of the start task will always be $ES_0 = 0$ and the latest finish time of the final task will be $LF_{J+1} = T$.

To determine the windows we make use of the method described in [2]. This method creates start and finish windows where precedence and resource feasibility is taken into account. For the earliest start times, precedence feasibility means that we have $ES_j^* \ge \max\{ES_i^* + p_i | i \in P_j\}$ and resource feasibility means that we have $r_{j,k,t-ES_j^*+1} \le R_{kt}$ for all times $t \in [ES_j^*, ES_j^* + p_j - 1]$. Similarly, we have that for the latest finish times we have that $LF_j^* \le \min\{LF_i^* - p_i | i \in P_j\}$ and $r_{j,k,t-LF_j^*-p_j+2} \le R_{kt}$ for all times $t \in [LF_j^* - p_j + 1, LF_j^*]$.

This step has several benefits. Firstly, it will significantly reduce the number of constraints necessary to formulate the problem as well as reduce the search space. Since it is impossible for a feasible schedule to have tasks scheduled outside their time window, we can ignore all times outside a task's window. We do this by removing them as variables and with them a bunch of constraints. Furthermore, the time windows can be used to determine a lower bound on the schedule.

Since the final task always needs to be scheduled we can use its earliest finish time EF^*_{J+1} as a lower bound.

2.3 MAX-SAT formulation

There are two types of constraints in a MAX-SAT problem, *hard* constraints, which always need to be satisfied for a feasible solution, and *soft* constraints, which need to be satisfied as much as possible. The soft constraints each have a weight, the goal of the solver will be to satisfy as many soft constraints as possible, so as to maximize the sum of the weights of the satisfied constraints.

The MAX-SAT formulation used by the solver is based on two types of boolean variables: *start* variables y_{jt} and *process* variables x_{jt}. The variable y_{jt} is $true$ if task i, starts at time t. The *process* variables are used to denote that a task is running at a certain time, variable x_{jt} is $true$ if task j is running at time t. Using these variables we can construct the MAX-SAT formulation.

Hard Constraints

The hard constraints are meant to force all solutions to be feasible schedules, it does not regard the goal of minimizing the makespan.

The first constraints enforce the tasks to be scheduled inside their start time window, make sure all tasks are scheduled and make sure the *process* variables are set correctly.

$$y_{0,0} \tag{6}$$

$$\bigvee_{s \in STW(j)} y_{js} \qquad \forall j \in [1, J+1] \tag{7}$$

$$\neg y_{J+1,t} \vee \neg y_{J+1,t+1} \qquad \forall t \in [ES^*_{J+1}, LS^*_{J+1} - 1] \tag{8}$$

Constraint (6) forces the start task to be scheduled at time 0. Constraints (7) force all tasks to be scheduled by enforcing they all have a starting time inside their start time window and constraints (8) force there to only be one end.

$$\neg y_{js} \vee x_{jt} \qquad \begin{array}{l} \forall j \in [1, J], \\ \forall s \in STW(j), \\ \forall t \in [s, s + p_j - 1] \end{array} \tag{9}$$

Constraints (9) force the process variables to be $true$ if they are within the task duration after the start of the task.

The next constraints enforce the precedence and resource constraints.

$$\neg y_{js} \quad \bigvee_{\substack{t \in [ES^*_j, \\ \min(s - p_j, LS^*_j)]}} y_{it} \qquad \begin{array}{l} \forall j \in [0, J+1], \\ \forall i \in P_j, \\ \forall s \in STW(j) \end{array} \tag{10}$$

The precedence constraints are enforced by constraints (10). If a variable y_{js} is scheduled all preceding tasks $i \in P_j$ need to be scheduled sufficiently before time s, namely somewhere in the window $[ES^*_j, \min(s - p_j, LS^*_j)]$.

$$\sum_{\substack{j \in [1, J+1] \text{ s.t.} \\ t \in [ES^*_j, LF^*_j] \\ e \in [0, p_j - 1] \text{ s.t.} \\ t - e \in STW(j)}} r_{jke} \cdot y_{j,t-e} \leq R_{kt} \qquad \begin{array}{l} \forall k \in R \\ \forall t \in [1, T] \end{array} \tag{11}$$

Constraints (11) enforce resource constraints. For each combination of resource and time, all resource demands are summed for tasks that are scheduled at that time, these demands should always be less than or equal to the availability of the resource at the same time. The constraints are expressed as *less than or equal* constraints for simplification, but they are converted to proper SAT constraints in CNF using the *RDD-1* method from [14]. This method also generates extra variables, which will be referred to as *resource* variables.

Finally, some redundant constraints are added.

$$\neg x_{jt} \vee x_{j,t+1} \vee y_{j,t-p_j+1} \qquad \begin{array}{l} \forall j \in [0, J+1], \\ \forall t \in [EC^*_j, \\ \quad LC^*_j - 1] \end{array} \tag{12}$$

Constraints (12) are not strictly necessary but have been shown to improve the solve times of the MAX-SAT solver [15].

Soft Constraints

Now we have constraints to get feasible schedules, we add soft clauses so the MAX-SAT solver can find a schedule with a minimised makespan. We do this by adding a weighted constraint for every possible start time of the end task, with decreasing values, such that it is beneficial for the end task to be scheduled as early as possible. I.e. we add constraints $y_{J+1,ES^*_{J+1}+t}$ with weight $|STW(J+1)| - t$ for $t \in [1, |STW(J+1)|]$.

3 MAX-SAT solvers

This section gives an overview of the MAX-SAT solver that is used in this research. First, the basics of how the MAX-SAT solver works will be explained as well as where heuristics come into play. Then one of the most used heuristics and the heuristic on which this research is based, VSIDS, will be explained.

3.1 CDCL

Before we can look at the effect of heuristics we have to know the basics of how a MAX-SAT solver works. For this research, the in-house MAX-SAT solver of the faculty was used. It is based on Conflict Driven Clause Learning or *CDCL*, a method first proposed by Marques-Silva and Sakallah [10; 9] and Bayardo and Schrag [11], which in turn is based on the DPLL algorithm already introduced in the 1960s [16; 17].

The DPLL algorithm can be described in several steps. First, it selects a literal and assigns a truth value to it, which literal gets selected is determined by the variable selection heuristic. Then it uses *unit propagation* to simplify the remaining clauses. In this step, it recursively checks if there are any clauses with only one unassigned literal, which means there is only one value which makes the clause satisfied, at which point it assigns that value to the variable. Whenever a conflict is encountered, i.e. some clauses become impossible to satisfy, it backtracks to the last decision. When all variables are satisfied it has found a feasible solution.

The main way the CDCL algorithm improves on the DPLL algorithm is by allowing backtracks to backtrack to earlier decision levels. After it has performed unit propagation and it has encountered a conflict, the conflict is analysed and used to ”learn” a clause that prevents that conflict from occurring. This analysis also finds the appropriate level to backtrack to, this can possibly jump over several irrelevant decisions.

3.2 VSIDS

An important factor for the efficiency of the solver is which variables are chosen as branching decisions [18]. Different branching heuristics can result in considerable differences in solve times. Many solvers (including the in-house solver, but also for example MiniSat [8]) make use of the Variable State Independent Decaying Sum (or VSIDS) heuristic, introduced by Chaff [19], which uses a dynamic order of variables based on activity.

The VSIDS heuristic works by first attaching an activity to each variable initialized at 0. Every time a clause is learned in the conflict analysis step of CDCL, the variables corresponding to that clause get *bumped*, meaning their activity value is increased. After a conflict, all the activity values are *decayed*, which means they get divided by a constant. Each time a branching decision is made the variable with the highest activity gets chosen. Ties are usually decided randomly.

The result of this heuristic is that the solver typically chooses variables that have recently appeared in conflicts. This is believed to benefit the solver because these conflicts are what drive the search process for difficult problems. Furthermore, the heuristic can be implemented with a relatively low overhead, which means only a little time is spent during the branching decision.

4 VSIDS for the RCPSP/t

This section gives an overview of the different adaptations of VSIDS that were studied in this paper.

A disadvantage of VSIDS is that the activity values are all equal at the start of the search, which means the initial decisions are random, and result in a lot of conflicts without learning a lot of useful clauses. An approach to solve this problem is by combining VSIDS with another heuristic, using the other heuristic for the first branching decisions then switching to VSIDS after some activity values have been set, a method that has already seen success for the standard RCPSP in [12]. Another method is to set the initial activity values to values other than 0, based on the RCPSP/t instance information. This is the approach of the heuristic that we will consider.

The variables used in the encoding can be divided into three kinds of variables: *start* variables, *process* variables and *resource* variables (as explained in subsection 2.3). The far majority of the variables are *resource* variables, but we expect they are usually not very good to select since each of them only appears in a few clauses. Our VSIDS adaptation will focus on the *start* variables because these variables directly influence the satisfiability and makespan.

Several ways of setting the activity values of the *start* variables were considered. The simplest possible adaptation is where all initial values of the *start* variable activities are set to a constant c while keeping all other activities at 0. This will effectively result in the heuristic always selecting *start* variables at the beginning of the search. Once enough conflicts have arisen and the *start* variables have decayed and the other variable have been bumped, the other variables will also be considered.

The other methods that are considered are methods where the activities are set using slightly more complex methods which are often considered for other scheduling problems. Firstly, setting the activity values based on the size of the start time window. Either initialising the largest windows with the highest values, referred to as LWH, or with the smallest windows initialised with the highest values, referred to as SWH. We also considered methods based on the position of the variable inside the start window. Either initialising the earliest variables of the window with the highest values, referred to as ESH, or the latest, referred to as LSH.

5 Experimental Setup

This section covers the experimental setup used to evaluate the new heuristic. To properly evaluate the performance of the different heuristics we ran experiments based on the data from PSPLIB [13]. A detailed explanation of how the test instances were generated can be found in [2].

In total, the PSPLIB contains twelve test sets for the RCPSP/t, six sets with projects consisting of 30 tasks (labelled J30t1 through J30t6), and six sets with projects consisting of 120 tasks (labelled J30t1 through J30t6). For our analysis, we make use of the J120t set because the projects in the J30t set were found to be too small to reliably compare the heuristics. The J120t set consists of $6 \cdot 600 = 3600$ instances. Due to time constraints, we were not able to run all experiments on the full data set, some experiments were run on a randomly selected subset of the experiments.

There are four main parameters used to generate the test sets, P^R, P^r, F^R and F^r [2]. P^R and P^r are probabilities that control whether a reduction is applied to a resource availability or demand respectively. Parameters F^R and F^r control by how much they are reduced. The values for the parameters can be found in Table 1. For example, for J120t1, each availability R_{kt} is set to $F^R \cdot R_{kt} = 0$ with a probability of $P^R = 0.05$ and kept at R_{kt} with a probability $1-P^R = 0.95$. And similarly, each demand r_{jkt} is set to $F^r \cdot r_{jkt} = 0$ with a probability of $P^r = 0.05$ and kept at r_{jkt} with probability $1 - P^r = 0.95$.

The experiments were run on the *DelftBlue* supercomputer [20] with a memory limit of 32GB. For each instance, the

Table 1: Parameters used in the generation of the different test sets J120t1 through J120t6 [2].

Set no.	1	2	3	4	5	6
P^R & P^r	0.05	0.1	0.2	0.05	0.1	0.2
F^R & F^r	0	0	0	0.5	0.5	0.5

time limit was set at 60s. During the experiments, all solutions that were found that improved over the previous best makespan were logged with the time at which they are found.

6 Results

In this section, we use figures and tables to show the results and make observations based on the data. First, the experiments on the full data set are shown using the standard VSIDS heuristic and the simple augmented heuristic with $c = 10^4$. Then some experiments are shown that compare the augmented heuristic to other variants with different values of c and different initialisation methods.

6.1 Simple VSIDS augmentation

In Table 2 you can see the number of proven satisfiable and optimal instances. We see a significant difference in the number of solutions that were proven satisfiable as well as the number of instances that were proven optimal. Using VSIDS the solver was able to prove satisfiability for 2402 of the 3600 instances, while the augmented VSIDS heuristic was able to prove satisfiability for 2806 instances, an increase of roughly 17%. It was also able to prove optimality for more instances, 1743 compared to 1660, an increase of roughly 5%.

Although the solver with the augmented VSIDS heuristic solves more instances overall, it does not perform better universally. On 26 instances, the solver with the standard VSIDS heuristic was able to prove optimality, while the solver with the augmented VSIDS heuristic was unable to. And on another 62, the solver with the standard VSIDS was able to find a solution while the solver with the augmented VSIDS heuristic did not.

Table 2: Number of proven optimal and satisfiable instances found using standard VSIDS and augmented VSIDS and the percentage of increase between them.

	VSIDS		aug. VSIDS		Increase	
	#sat	#opt	#sat	#opt	sat	opt
j120t1	405	284	461	303	14%	7%
j120t2	460	354	533	370	16%	5%
j120t3	534	464	585	479	10%	3%
j120t4	318	190	387	197	22%	4%
j120t5	319	184	409	198	28%	8%
j120t6	366	184	431	196	18%	7%
total	2402	1660	2806	1743	17%	5%

We also observe a difference between the different data sets. Both algorithms perform better in sets 1, 2 and 3, with $F^R = F^r = 0$ than in sets 4, 5 and 6, with $F^R = F^r = 0.5$. But we can see that augmented VSIDS also improves more compared to regular VSIDS on sets 4, 5 and 6, finding solutions for roughly 23% more instances, but only 13% for sets 1, 2 and 3.

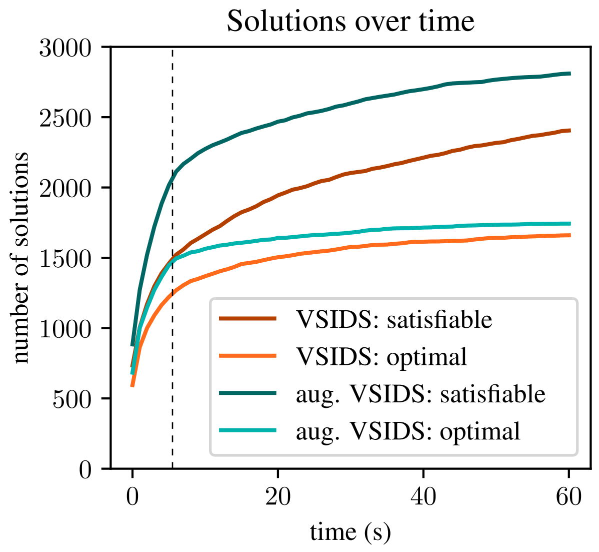

Figure 3: The number of first solutions and optimal solutions found over time for the original VSIDS and the augmented VSIDS.

In Figure 3, we see that as expected, the augmentation mostly has an effect at the beginning of the search. In the first 5s it is able to find a lot more solutions than the standard heuristic, but after these 5 seconds, it slows down.

6.2 Other variants

In Table 3 you can see experiments that were run for the augmented VSIDS heuristic with different values for the start variable activities c. They were all run on the same subset of 180 instances of the J120 set. You can see that while there is a big difference with $c = 0$ compared to the other values of c, all values $c > 0$ achieved very similar results.

Table 3: Number of proven satisfiable and optimal instances found for different values of c out of 180 instances.

c	0	1	10	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}
#sat	114	142	140	144	144	140	141	141	139	144	141	145
#opt	82	84	83	86	85	83	85	85	85	83	83	86

In Table 4 you can see the results of an experiment with negative values for c. Negative values of c essentially have the same result as initialising all variables except the $start$ variables. You can see that this does not seem to change the results significantly compared to $c = 0$.

Table 5 shows the results of experiments using the different methods to initialise the $start$ variables as described in section 4. The different methods seem to have no significant difference in performance compared to the simple method of setting all $start$ variable activities to a constant.

Table 4: Number satisfiable and proven optimal instances found for different values of c, including negative values, out of 500 instances.

c	-10^4	-1	0	1
#sat	323	328	319	385
#opt	230	237	232	242

Table 5: Number of satisfiable and proven optimal instances found for different variants of the augmented VSIDS. Details about the different variants can be found in section 4

mode	$c = 0$	$c = 10^4$	ESH	LSH	LWH	SWH
#sat	317	384	372	382	383	376
#opt	233	241	240	242	239	243

7 Responsible Research

This section reflects on the ethical aspects of this research and discusses the reproducibility of the methods used.

To make sure the results are reproducible, all methods have been thoroughly explained throughout the report and all necessary information has been made public. The data sets that were used are publicly available in PSPLIB [13]. How the data sets were converted to WCNF encodings can be found in subsection 2.3. And, the heuristics that were compared in this research are explained in detail in section 4, so they can easily be implemented by someone who wants to validate the results.

The raw data that was collected during our experiments can be found in a public repository[1], so they can be analysed and critiqued by anyone. They have not been altered in any way, and by using the full data set we have made sure the data was not cherry-picked and accurate conclusions could be drawn from them.

8 Discussion

The experiments show that the augmented VSIDS heuristic does improve the results of the solver compared to the standard VSIDS heuristic. Looking at Figure 3 it is likely due to the solver choosing better variables at the beginning of the search, seeing as that is when it is finding more solutions. The bigger increase of solutions in sets 4, 5 and 6 might be due to them having comparatively more *resource* variables. This is because the resource availabilities and demands are only halved instead of being set to zero. These extra *resource* variables could be the reason these sets are harder to solve, but also increase the effect of the augmentation since the augmentation works by increasing the activity values of *start* variables compared to *process* and *resource* variables.

Curiously we see very little difference in the way we set the *start* variable activities, with both different values for c and completely different initialisation methods. For different values of c, it does not seem to matter as long as c is bigger than 0. This probably means that the very first decisions have the biggest influence since this is when these heuristics still behave equally. This could be because after the first conflicts arise, the *start* variable values might increase more than the other values anyway, so having them start even higher has little impact. This would also explain why setting the *start* variables activities also has little impact since they would then also just influence the very first decisions.

Setting the value of c negative, which essentially has the same effect as increasing the other variable activities, achieves very similar results to $c = 0$ and does not seem to hurt the performance any further. This could be because there are many more *resource* variables than *start* variables, thus when $c = 0$ VSIDS is already far more likely to first select *resource* variables and it would not change much if $c < 0$. If the first variables that get selected have the biggest influence on the performance, this small difference in performance would thus make sense.

If we compare our results to the previous exact results in [6], we see that our method using MAX-SAT does not perform as well as their method using SMT for finding solutions. They were able to find a solution for nearly all instances of J120t even with very little time, while the MAX-SAT solver with the augmented heuristic was only able to find solutions for 2806 instances out of 3600. For finding optimal solutions, however, the results are closer to the SMT method. We found 1743 optimal solutions, while [6] was able to find 1932 optimal solutions for 60s per instance. It is important to note however that this might not be an accurate comparison since both experiments were run in different environments.

9 Conclusions and Future Work

In this paper, we have analysed an adaptation of the VSIDS heuristic for the resource-constraint project scheduling problem with time-dependent resource availabilities and demands (RCPSP/t). By setting initial values for the VSIDS activity values, we designed a problem-specific heuristic for the RCPSP/t. The experiments showed that the adapted heuristic allowed the solver to find more solutions in the same time span, finding solutions for 78% of the instances and finding the optimal solution for 48% of the instances compared to 66% and 46%, respectively, for the standard VSIDS heuristic.

We also analysed in which way the new heuristic improves on the standard VSIDS heuristic. We found that it mostly improves the beginning of the search and that after a while it does not have any significant effect. We also found it increases the performance the most for the test sets with $F^R = F^r = 0.5$.

We also compared different methods of initialising the *start* variable activities to see if we would be able to achieve further improvement, but we saw very little difference. Both changing the constant that we set the activity values to, and using totally different initialisation methods, seem to have very little effect on the performance.

Future research could explore if the initial values could be improved further. While we did test some different methods in this research and saw very little difference, there are many other ways to initialise the variables that could have a significant effect. Other potential improvements to VSIDS could be changing the bump amount per variable based on the problem

[1]https://github.com/t-lenssen

information. It would also be interesting to see the results of using a different heuristic for the first decisions and then swapping to VSIDS later.

Another area for future research could be using a similar method to improve VSIDS for other RCPSP extensions, and possibly other totally different problems. A lot of problems can be solved with MAX-SAT solvers. If you were able to generalise the way we choose initial activities it might be possible to achieve similar improvements for other problems as well.

References

[1] M. Vanhoucke, *The Westerschelde Tunnel Project*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 175–185. [Online]. Available: https://doi.org/10.1007/978-3-642-40438-2_9

[2] S. Hartmann, "Project scheduling with resource capacities and requests varying with time: a case study," *Flexible Services and Manufacturing Journal*, vol. 25, no. 1-2, pp. 74–93, 2013. [Online]. Available: https://dx.doi.org/10.1007/s10696-012-9141-8https://link.springer.com/article/10.1007/s10696-012-9141-8

[3] ——, *Time-Varying Resource Requirements and Capacities*. Cham: Springer International Publishing, 2015, pp. 163–176. [Online]. Available: https://doi.org/10.1007/978-3-319-05443-8_8

[4] J. Blazewicz, J. K. Lenstra, and A. H. G. R. Kan, "Scheduling subject to resource constraints: classification and complexity," *Discrete Applied Mathematics*, vol. 5, no. 1, pp. 11–24, 1983. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0166218X83900124

[5] S. Hartmann and D. Briskorn, "An updated survey of variants and extensions of the resource-constrained project scheduling problem," *European Journal of Operational Research*, vol. 297, no. 1, pp. 1–14, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221721003982

[6] M. Bofill, J. Coll, J. Suy, and M. Villaret, "Smt encodings for resource-constrained project scheduling problems," *Computers & Industrial Engineering*, vol. 149, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835220304873

[7] J. Pleunes, "Sat-based optimisation for the resource-constrained project scheduling problem with time-dependent resource capacities and requests," Bachelor thesis, Delft University of Technology, 2022. [Online]. Available: http://resolver.tudelft.nl/uuid:698692ab-3a12-4a66-ae6a-214d77d4a013

[8] N. Eén and N. Sörensson, *An Extensible SAT-solver*. Springer Berlin Heidelberg, 2004, pp. 502–518. [Online]. Available: https://dx.doi.org/10.1007/978-3-540-24605-3_37

[9] J. P. Marques-Silva and K. A. Sakallah, "Grasp: a search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999. [Online]. Available: https://dx.doi.org/10.1109/12.769433

[10] J. Marques Silva and K. Sakallah, "Grasp-a new search algorithm for satisfiability," in *Proceedings of International Conference on Computer Aided Design*, 1996, pp. 220–227.

[11] R. J. Bayardo and R. C. Schrag, "Using csp look-back techniques to solve real-world sat instances," in *AAAI'97/IAAI'97: Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence*, 1997, pp. 203–208.

[12] A. Schutt, T. Feydy, Peter, and Mark, "Solving the resource constrained project scheduling problem with generalized precedences by lazy clause generation," *arXiv pre-print server*, 2010. [Online]. Available: https://arxiv.org/abs/1009.0347

[13] R. Kolisch and A. Sprecher, "Psplib - a project scheduling problem library: Or software - orsep operations research software exchange program," *European Journal of Operational Research*, vol. 96, no. 1, pp. 205–216, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221796001701

[14] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger, "A new look at bdds for pseudo-boolean constraints," *Journal of Artificial Intelligence Research*, vol. 45, 2014.

[15] A. Horbach, "A boolean satisfiability approach to the resource-constrained project scheduling problem," *Annals of Operations Research*, vol. 181, no. 1, pp. 89–107, 2010. [Online]. Available: https://dx.doi.org/10.1007/s10479-010-0693-2

[16] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, 1960. [Online]. Available: https://dx.doi.org/10.1145/321033.321034

[17] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962. [Online]. Available: https://dx.doi.org/10.1145/368273.368557

[18] J. Marques-Silva, *The Impact of Branching Heuristics in Propositional Satisfiability Algorithms*. Springer Berlin Heidelberg, 1999, pp. 62–74. [Online]. Available: https://dx.doi.org/10.1007/3-540-48159-1_5

[19] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 530–535.

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