Shortening Plans by Local Re-Planning

Tomáš Balyo[1] and Roman Barták[1]
[1]Faculty of Mathematics and Physics
Charles University
Prague, Czech Republic
{tomas.balyo, roman.bartak}@mff.cuni.cz

Pavel Surynek[1,2]
[2] Kobe University
5-1-1 Fukae-minamimachi, Higashinada-ku
Kobe 658-0022, Japan
pavel.surynek@mff.cuni.cz

***Abstract*—There exist planning algorithms that can quickly find sub-optimal plans even for large problems and planning algorithms finding optimal plans but only for smaller problems. In this paper we attempt to integrate both approaches. We present an anytime technique for improving plan quality, in particular decreasing the plan makespan, via substituting parts of the plan by makespan-optimal sub-plans. The technique guarantees optimality though it is primarily intended to quickly improve plan quality. We experimentally compare various approaches to local improvements and we show that our method has significantly better makespan score than the SASE planner, which is one of the best optimal planners.**

Keywords-planning; makespan optimization

I. INTRODUCTION

Classical AI planning deals with the problem of finding a partially ordered set of actions that transfers the world from some initial state to a state satisfying a certain goal condition. As it is typical for many hard combinatorial optimization problems, there is a trade-off between the plan quality and the time necessary to find the plan. There exist planners such as LPG [6] and SGPLAN [9] that generate quickly plans for many problem domains but the plan quality may be bad while other planners such as SASE [10] generate optimal plans with the trade-off of longer runtime. In this paper we attempt to improve the quality of plans generated by sub-optimal planners by doing local optimizations of the plans. More precisely, we will be improving makespan of parallel plans by optimizing sub-plans using SAT-based techniques such as SASE that are successful for finding makespan-optimal plans.

The idea of making local repairs in a sub-optimal plan to improve the plan towards optimal makespan or towards better visual quality has already appeared in domain-dependent planning. In [13] and [15] authors try to improve solutions of *cooperative-path finding* (CPF) where the task is to plan movements of agents so that they do not collide with each other. Improvements are done by ad-hoc local changes such as avoiding busy locations, eliminating agent's waiting, and replacing movements along long paths with shortcuts. In these cases the pattern of improvement is fixed which may cause that redundancies of a priori unknown pattern remain in the solution. Applying SAT-based mechanisms to improve solutions of CPF problems where sub-solutions are replaced by makespan-optimal ones has been proposed in [14]. This approach is trying to overcome the limitation imposed by the fixed set of improvement patterns. The SAT-based approach allows discovering redundancy or inefficiency of any pattern. A special domain-dependent propositional encoding of the CPF problem has been used.

Since domain-independent propositional encodings for planning such as SATPLAN [11] or SASE [10] are already available, we tried to apply the SAT-based solution improvement technique from CPF also in the domain-independent planning in this work. This is however not the first time that certain quality improvements in SAT-based domain-independent planning have been considered. In [7] the authors proposed to optimize the number of actions in the iterative process where a plan containing fewer actions is generated at each iteration. This is achieved through adding preference constraints into the propositional formulation of the problem.

From the broader perspective, our work is also related to techniques for compression of schedules [5] that apply the *critical path* method to reduce the makespan and to increase the parallelism of the schedule. Contrary to this approach, the parallelism is not restricted by the number of execution units but rather by the semantics of a planning instance in our case.

In this paper we propose a method that works as follows. We start with a sub-optimal plan and select one of its sub-plans. From this sub-plan we generate a new planning problem and using a SAT-based technique we find a makespan-optimal plan for this problem. This optimal plan substitutes the original sub-plan in the large plan and this way we decrease the makespan of the plan. The process is repeated with other sub-plans until a given stopping criterion is reached. We will propose several methods how to select the sub-plan and we compare these methods experimentally using the problems from the International Planning Competition [12].

The paper is organized as follows. We will first formally describe the planning problem to be solved. Then we will give details of our methods, namely how we define the planning problem from the sub-plan and how we find the makespan optimal plan for this problem. A large portion of the text will be devoted to the experimental study of the proposed methods and suggesting possible ways how to improve the method.

II. FORMAL BACKGROUND

In this paper we focus on optimizing makespan of parallel plans. Parallel planning deals with the problem of finding a sequence of sets of independent actions called a *parallel plan* that transfers the world from a given initial state to a state

satisfying a given goal condition. The *state* is defined as a set of propositions that are true in a given state. The goal condition is defined as a set of propositions that must hold in the goal state – state S satisfies the goal condition G if $G \subseteq S$. Each action is a triple (Cond, PosEff, NegEff) of sets of propositions such that $PosEff \cap NegEff = \emptyset$. Action A is *applicable* to state S if $Cond_A \subseteq S$. If the action A is applicable to state S then the resulting state after applying the action is $\gamma(S, A) = (S \cup PosEff_A \setminus NegEff_A)$. We can apply a sequence of actions $A_1,\ldots, A_n$ to state S, if A_1 is applicable to state S and each A_i is applicable to $\gamma(\ldots\gamma(\gamma(S, A_1), A_2)\ldots, A_{i-1})$. We denote the final state $\gamma(S, (A_1,\ldots, A_n))$. Action A is relevant to a goal G if $PosEff_A \cap G \neq \emptyset$ and $NegEff_A \cap G = \emptyset$. The *regression set* for goal G and action A that is relevant to G is $\gamma^{-1}(G, A) = ((G \setminus PosEff_A) \cup Cond_A)$. The regression set characterizes the set of states (via a goal condition) such that goal G is reachable from any of these states by applying action A. We can define the regression set for G and for a sequence of actions $A_1,\ldots, A_n$ if A_1 is relevant to G and each A_i is relevant to $\gamma^{-1}(\ldots\gamma^{-1}(\gamma^{-1}(G, A_1), A_2)\ldots, A_{i-1})$. We denote the final regression set $\gamma^{-1}(G, (A_1,\ldots, A_n))$. Two actions A and B are called *independent* if $NegEff_B \cap (Cond_A \cup PosEff_A) = \emptyset$ and $NegEff_A \cap (Cond_B \cup PosEff_B) = \emptyset$. Independent actions do not influence each other. Hence if independent actions are applicable to some state then they can be applied in any order to obtain an identical state. Formally, the set P of pairwise independent actions is applicable to state S if $\forall A \in P\ Cond_A \subseteq S$. The resulting state is then $\gamma(S, P) = (S \cup (\cup_{A \in P} PosEff_A) \setminus (\cup_{A \in P} NegEff_A))$. The parallel plan is *makespan optimal*, if there does not exist any shorter parallel plan.

Another way to formalize the planning problem is using the *SAS+ formalism* [8] based on *multivalued state variables* instead of propositions. A world state is specified by the values of all state variables. The goal conditions and the preconditions of actions are expressed as required values of certain state variables and the effects of actions are assignments of values to some state variables. An action is applicable to a state if its preconditions are satisfied in the given state, i.e., if the state variables included in the action's preconditions are properly assigned. The resulting state after applying an action has some of its variables set to the new values as specified by the action's effects. Two actions are independent if they use disjunctive sets of state variables. A set of independent actions is applicable to a state if all the actions in the set are applicable to the state. A set of such actions transforms a state to a new state where the effects of all actions are applied. A parallel plan is a sequence of applicable sets of independent actions, which transfer the initial state into a state satisfying the goal condition. The makespan of the parallel plan is the length of the sequence. Classical proposition formulation can be automatically converted to SAS+ formalism [8] for which efficient SAT-encoding exists [10]. We use this encoding in our approach.

III. Methodology

Assume that we have a sub-optimal parallel plan and we want to shorten its makespan. We propose a method of local improvements of the plan that is based on selecting a sub-plan of the plan, finding a better sub-plan, and substituting the original sub-plan by this better sub-plan in the original plan. To formulate this method precisely we must answer several questions. First, how is the sub-plan selected? Second, how is a better sub-plan found? Our idea is based on using existing planning techniques, namely the SAT-based approach, to find a better plan. Hence, the second question consists of two additional questions. How does a sub-plan define a planning problem? How do we solve optimally that planning problem? The final question is how many times we should repeat this local improvement process.

In the following sections we shall answer the above questions. We will first show how to formulate a planning problem from the sub-plan selected from a larger plan and how to improve the sub-plan by finding a plan with a smaller makespan. Then we will show how to select the sub-plans for optimizations, in particular, we will present several ideas of shifting a window over the plan to define where the local improvements will be realized. The stopping criterion will also be presented there.

IV. Local Enhancements

Assume that we have a parallel plan consisting of a sequence of sets of independent actions $P_1,\ldots, P_n$ and this plan transfers the world from the state S to a state satisfying the goal condition G. Let $P_i,\ldots,P_j$, $i<j$, be a sub-sequence of the above plan to be optimized. The makespan of this sub-plan is (j-i+1) so optimizing the sub-plan means finding a parallel plan $P'_1,\ldots, P'_k$ such that $k < (j-i+1)$ and plan $P_1,\ldots, P_{i-1}, P'_1,\ldots, P'_k, P_{j+1},\ldots, P_n$ reaches the goal G from state S. In other words, we attempt to substitute the original sub-plan by a shorter sub-plan that is consistent with the rest of the plan. This new shorter plan is found by applying classical SAT-based techniques to a specific planning problem. We shall now show how to formulate this planning problem and how to solve it.

A. Identifying Local Sub-Problems

Let $P_1,\ldots, P_n$ be a plan reaching goal G from state S and $P_i,\ldots, P_j$ be a sub-sequence of that plan We formulate the planning problem P_{sub} to be solved as follows. The initial state of P_{sub} is defined as the state $\gamma(S, (P_1,\ldots, P_{i-1}))$; if i = 1 then we define the initial state of P_{sub} as S. Analogously, the goal G_{sub} of P_{sub} is defined as $\gamma^{-1}(G, (P_n,\ldots, P_{j+1}))$; if j = n then the goal G_{sub} equals G. Obviously, if we find a plan $P'_1,\ldots, P'_k$ that solves P_{sub} then P'_1 is applicable to $\gamma(S, (P_1,\ldots, P_{i-1}))$ etc. so $(P_1,\ldots, P_{i-1}, P'_1,\ldots, P'_k)$ is a valid plan that reaches goal G_{sub}. Because $G_{sub} = \gamma^{-1}(G, (P_n,\ldots, P_{j+1}))$ contains the precondition of P_{j+1} (see the definition of the regression set), the action set P_{j+1} is applicable to state $\gamma(S, (P_1,\ldots, P_{i-1}, P'_1,\ldots, P'_k))$. Similarly propositions from the precondition of P_{j+2} are either among the effects of P_{j+1} or in G_{sub}. Hence P_{j+2} is applicable to state $\gamma(S, (P_1,\ldots, P_{i-1}, P'_1,\ldots, P'_k, P_{j+1}))$. Together $(P_1,\ldots, P_{i-1}, P'_1,\ldots, P'_k, P_{j+1},\ldots, P_n)$ is a valid plan that reaches goal G from state S. Figure 1 illustrates the process of substituting a sub-plan with a shorter sub-plan.

B. Optimizing Local Sub-Plans

As mentioned above, we use a SAT technique to optimize the sub-plan. The basic idea is the following. For the planning

problem P_{sub} and a parameter k we create a SAT formula F_k which is satisfiable if and only if there is a parallel plan for P_{sub} of size k or shorter. To obtain this SAT formula we use the SASE encoding, which is described in [10]. If the formula F_k is satisfiable, then we can efficiently extract a parallel plan of size k (or shorter) from its satisfying assignment.

A standard approach for solving planning problems via SAT is to iteratively solve F_k for $k=1,2,\ldots$ until F_k is satisfiable (Kautz and Selman, 1999). Since we have an upper bound on the length of the solution of P_{sub} (it is the size s of the sub-plan we want to improve), we can start searching for a better sub-plan by solving from F_{s-1} downwards, i.e., solving F_{s-1}, F_{s-2} ,… until we find an unsatisfiable formula F_m. At that moment we can conclude that the optimal plan of P_{sub} has the size $m+1$. Then, from the satisfying assignment of F_{m+1} we extract the optimal plan for the local sub-problem. If the first formula F_{s-1} is unsatisfiable, then the makespan of the sub-plan cannot be improved, since it is already makespan optimal.

Generating and solving SAT formulas in a downward fashion has several advantages over the standard approach in our application. First, the expected number of generated and solved formulas is usually lower for long sub-plans. Second, the SASE encoding allows us to create the sequence of SAT formulas incrementally if proceeding downwards. More precisely, we can generate F_{i-1} from F_i by only adding new clauses to F_i. Some SAT solvers support incremental SAT solving and can therefore solve a sequence of such formulas more efficiently if they are supplied in an incremental way to the solver.

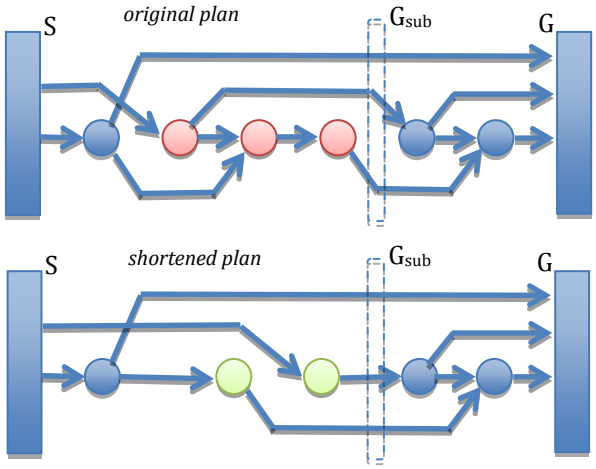

Figure 1. Substituting a sub-plan by a shorter sub-plan (arrows indicate causal relations between the action).

V. Plan Window Shifting

In this section we give some methods how to select the local sub-problems (or plan windows) we wish to improve.

The first and the simplest idea is selecting the windows randomly. We select two random numbers between 1 and the length of the plan and try optimizing the window between them. We repeat selecting random windows until we run out of time or we select the entire plan as our window and successfully optimize it.

Another approach is to systematically shift a window of a certain size through the plan. A *Systematical window shifting (SWS) procedure* has three parameters: *(window size*, *window shift*, and *fixed point)*. It works by moving a window of the specified size through a plan from its beginning increasing its starting position by the window shift parameter until the end of the plan is reached. Note that the last window in the iteration may be shorter than the specified *window size*. The fixed-point parameter of the SWS procedure specifies whether the iteration is repeated if any window has been improved. In particular, if any of the windows during the shifting is improved and the fixed-point parameter is set to the value *True*, then the SWS procedure with the same parameters is repeated. Figure 2 gives a pseudo-code of procedure SWS.

There are many ways to set the parameter *window shift*. In our experiments we tried two methods – shifting the window by the *window size* (then the subsequent windows are not overlapping), we call this method *fullstep*; and shifting the window by the half size of the window (the next window shares its first half with the previous window), we call this method *halfstep*.

The remaining question is what window size should be used. For domain independent planning we prefer a parameter-free solving method so we propose two methods both based on incrementally increasing the window size. The method called *turbo* increases the window size by 1 and the method called *exponential* (or just *expo*) increases the window size by the factor of 3/2 (NextSize ← 3/2*Size). While *turbo* focuses more on smaller windows, *expo* tries to reach larger windows faster. For both methods we suggest starting with the initial size 3 where some makespan improvement may happen. Theoretically, window size 2 may also lead to improving plan makespan, but we believe that such improvement is so easy that the planner generating the initial plan would find it.

When optimizing a window, we also specify a time limit for its optimization to avoid spending too much time on optimizing hard windows. The time limit depends only on the size of window to be optimized and the remaining time (we assume that the entire plan optimization process is launched with a time limit). The time limit is computed using the following formula:

$$Limit = RemainingTime / (PlanSize / windowSize)$$

The idea behind this formula is that we give a window so much time, that in the case of timeouts the remaining time would still be enough to optimize the rest of the plan with windows of the same size.

```
SWS(P, size, shift, fp)
  repeat
    Start <- 1
    Change <- false
    repeat
      P' <- optimize(P, Start, Size)
      if P' ≠ P then
        Change <- true
        P <- P'
      endif
      Start <- Start + Shift
    until Start >= makespan(P)
  until not(fp and Change)
```

Figure 2. Procedure for window shifting over plan P.

VI. EXPERIMENTAL STUDY

To evaluate properties of the proposed methods we did an experimental study comparing various combinations of the methods. In particular, we compared eight systematic approaches as specified in Table 1 and we also used the random method where the maximal window size was 20. This size was deduced from the experiment with the expo method that explored large windows but almost never found any improvement for window of size larger than 20 due to time limit. We used the LPG planner [6] to generate the initial plans. Because we are improving the makespan, we used the SASE [10] planner to compare the quality of plans generated by our method. The SASE planner is currently one of the best makespan-optimal planners.

Our experimental study focused on three aspects. Our initial goal was improving plan makespan, so we compared cumulative makespan of all methods. Of course, we want the method to be competitive also in terms of runtime, so we also compared the time efficiency of the methods. We also examined the above-mentioned properties of our methods using the performance score functions proposed in the learning track of IPC-6 [4]. The *speed score* of a system S over a set of benchmark problems is the sum of the scores over all the test problems in the set. The score of a system S on a problem P is 0 if P is unsolved in the given time limit and T_P^*/T_P otherwise, where T_P^* is the lowest measured CPU time required to solve P by any of the tested systems and T_P is the CPU time of the evaluated system S on the problem P. Analogously we define the *makespan score*, the only difference is, that instead of CPU time we consider the makespan of the resulting plans. For both scores higher values indicate better performance. Finally, we looked inside the methods and we tried to find the reason for the behavior of the methods.

All experiments were run on a cluster of identical PC computers with Intel Core i7 920@2.67GHz processors, 6GB of main memory running Gentoo Linux 2.0.3 (kernel version 3.0.26). We used SAT4J [1] to implement our SAT-based window improvements, while SASE used the precosat [2] SAT solver. All auxiliary procedures were implemented in Java.

TABLE I. DESCRIPTION OF METHOD PARAMETERS

method	win. size increase	window shift	fixed point
expo-fullstep	size*3/2	size	no
expo-fullstep-fp	size*3/2	size	yes
expo-halfstep	size*3/2	size/2	no
expo-halfstep-fp	size*3/2	size/2	yes
turbo-fullstep	size+1	size	no
turbo-fullstep-fp	size+1	size	yes
turbo-halfstep	size+1	size/2	no
turbo-halfstep-fp	size+1	size/2	yes

A. Plan Quality

Our initial motivation was improving the plan quality, namely makespan, for a planner that can find plans quickly, but with lower quality of plans. Table 3 shows the comparison of cumulative makespan for all 189 solved problems (total makespan) and for 135 problems that both SASE and our method can solve (joint makespan). The table shows that plans generated by LPG have more than five times larger makespan than the plans generated by SASE and that all our methods significantly reduce the makespan. In fact, the plans improved by our methods are very close to the optimal plans produced by SASE. Table 4 gives a comparison of makespan scores. All our methods outperform both SASE and LPG, with turbo-halfstep-fp being the method with the best performance. Both tables show that it is worth to scan the windows systematically over the plan rather than trying them completely randomly. The experiment also confirms the expectation that using the iterations with fixed points leads to better plans and that overlapping windows also contribute to smaller makespan. The experiment also shows that it is better to increase the size of the window conservatively between the iterations. In summary, it seems that to get the largest quality improvement, it is better to use smaller windows that are overlapping and to increase the window only when no improvement with the current window size can be found.

TABLE II. THE NUMBER OF SOLVED PROBLEMS

domain		SASE	Our method (via LPG)	Common
depots	22	16	22	16
driverlog	20	17	20	17
freecell	20	6	4	4
pipesworld	50	34	31	28
rovers	40	17	40	17
storage	30	15	30	15
tpp	30	30	22	22
zenotravel	20	16	20	16
solved	232	151	189	135

TABLE III. COMPARISON OF PLAN MAKESPAN

method	total makespan		joint makespan	
LPG	14531	100%	6752	548%
expo-fullstep	4411	30%	1911	155%
expo-fullstep-fp	3434	24%	1665	135%
expo-halfstep	3577	25%	1660	135%
expo-halfstep-fp	3138	22%	1550	126%
turbo-fullstep	3426	24%	1589	129%
turbo-fullstep-fp	3156	22%	1563	127%
turbo-halfstep	3076	21%	1515	123%
turbo-halfstep-fp	3013	21%	1540	125%
random-20	6351	44%	1935	157%
SASE			1232	100%

TABLE IV. COMPARISON OF MAKESPAN SCORES

method	makespan score	ΔLPG	ΔSASE
LPG	71,27	0,00	-75,38
expo-fullstep	158,33	87,05	11,67
expo-fullstep-fp	170,41	99,14	23,76
expo-halfstep	169,48	98,21	22,83
expo-halfstep-fp	177,16	105,88	30,50
turbo-fullstep	172,67	101,40	26,02
turbo-fullstep-fp	175,57	104,30	28,92
turbo-halfstep	177,65	106,38	31,00
turbo-halfstep-fp	179,53	108,25	32,87
random-20	159,76	88,49	13,11
SASE	146,65	75,38	0,00

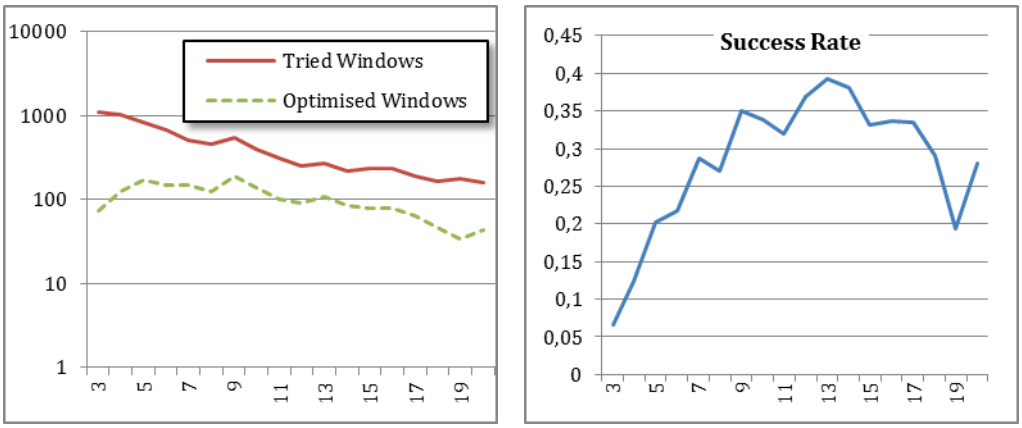

Figure 3. Description of scanning windows based on their size (X axis) for the **random** method.

B. *Efficiency*

In this section we will focus on the runtimes of the methods. Table 5 shows the cumulative runtimes for 135 problems solved by all methods. The clear winner is the LPG planner. Note that the runtimes of our methods do not include the time to find the initial plan (the runtime of LPG needs to be added). Table 6 contains the speed scores of the methods and shows an even more significant lead of LPG than the total runtime comparison.

There are not significant differences between the runtimes and speed scores of our methods. This is because we are doing local improvements until we find the optimal plan (the last window covers completely the plan) or the time limit is reached. For larger plans we frequently reach the time limit, which is reflected in the total runtime. In general, our methods are about one order of magnitude slower than LPG but only a few times slower than SASE.

We also measured when we found the last improvement of the plan (improving runtime). The results are quite interesting and promising for future improvements. For the best combination of methods turbo-halfstep-fp, the time to find the best plan is very close to the runtime of SASE while we still have the advantage that thanks to LPG we can find more plans. Nevertheless, it seems that our methods in the core form waste about 40% of runtime by continuing attempts to improve the plan. Finding a better stopping criterion would improve the runtime while preserving the quality of found plans.

TABLE V. COMPARISON OF RUNTIMES (IN SECONDS)

method	total runtime	improving runtime	efficacy
LPG	6 604		
expo-fullstep	63 195	41 409	66%
expo-fullstep-fp	54 568	35 540	65%
expo-halfstep	55 652	40 340	72%
expo-halfstep-fp	52 018	31 798	61%
turbo-fullstep	52 446	34 637	66%
turbo-fullstep-fp	53 834	30 478	57%
turbo-halfstep	50 251	33 749	67%
turbo-halfstep-fp	52 568	29 472	56%
random-20	63 082	49 647	79%
SASE	23 220		

TABLE VI. COMPARISON OF SPEED SCORES

method	speed score	ΔLPG	ΔSASE
LPG	167,02	0,00	147,06
expo-fullstep	5,89	-161,13	-14,07
expo-fullstep-fp	6,51	-160,51	-13,45
expo-halfstep	6,46	-160,56	-13,50
expo-halfstep-fp	6,44	-160,58	-13,52
turbo-fullstep	6,73	-160,28	-13,23
turbo-fullstep-fp	6,34	-160,67	-13,61
turbo-halfstep	6,85	-160,17	-13,11
turbo-halfstep-fp	6,61	-160,40	-13,34
random-20	5,41	-161,61	-14,55
SASE	19,96	-147,06	0,00

C. *Looking Inside*

So far we looked at global properties of the compared methods. From the comparison of runtimes, it seems that we are wasting some time by continuing to run the local optimizations without obtaining further improvement. In this section, we shall look inside the proposed methods to uncover other possible inefficiencies that may help to further improve the methods. In particular, we will explore the number of windows that each method tries to optimize for each window size and we look at how many of these windows actually lead to improving the makespan.

Figure 3 shows the number of windows as the relation of the window size for the random method. Recall that in this method, we placed the windows with uniform distribution of their size randomly in the plan (we used windows of maximal size 20). We measured the number of tried windows and the number of windows that lead to improvement of the plan (left) and we computed the ratio (success rate) between these numbers (right). The results are not very surprising. The number of tried windows decreases exponentially with the size of the windows and the very small windows are less successful during optimizations. In general, only a small number of tried enhancements lead to real improvement of makespan that shows that the random method is very inefficient. An interesting observation may be that windows of size 13 are particularly successful during optimizations. We also did an experiment with the unlimited size of the windows, and the windows of size 13 were also among the peaks of the success

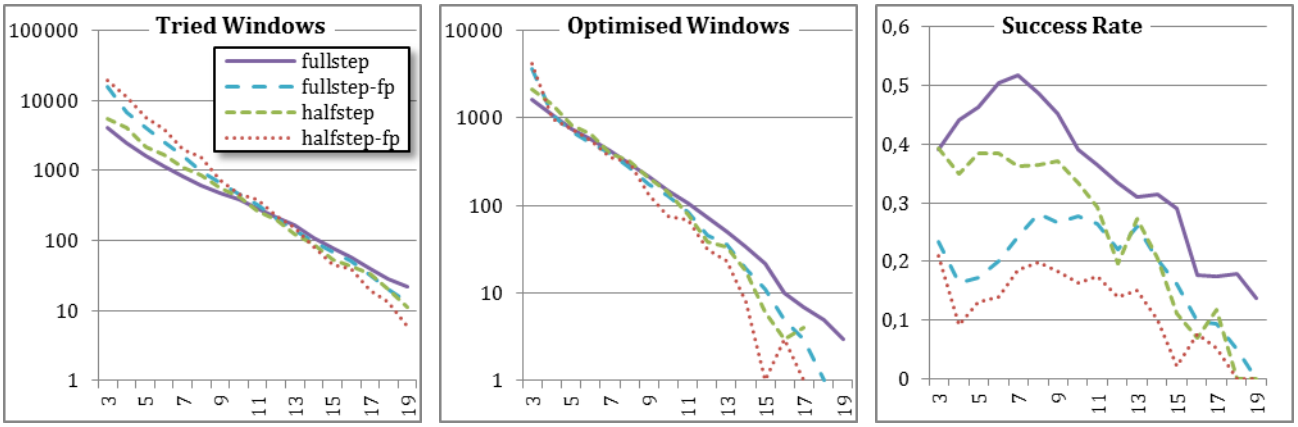

Figure 4. Description of scanning windows based on their size (X axis) for the **turbo** methods.

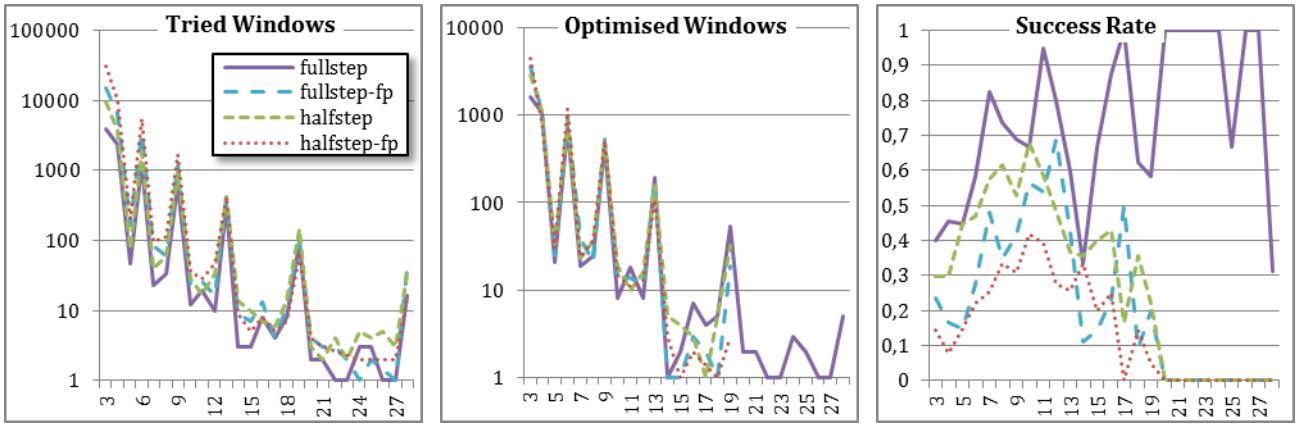

Figure 5. Description of scanning windows based on their size (X axis) for the **expo** methods.

rate. Note, that this is a cumulative result for all the tested domains and a more detailed study of particular domains may reveal whether this window size is really common for different domains.

Figure 4 shows the same measurements for all four versions of the turbo method. The graph comparing the success rate per window size is of particular interest. The graph clearly shows that using the fixed point in iterations significantly degrades the success rate of optimizing the windows. In other words, we are wasting time by trying to optimize the windows of the same size again and again while any window of a given size is still being improved. Nevertheless, recall that the fixed-point strategy generally reached better plans and the runtime was comparable to the methods without the fixed point. The reason could be that the fixed-point strategies spent more time with smaller windows where the optimization is much faster. The graphs showing the number of tried windows confirm this hypothesis. The conclusion is that optimizing smaller windows contribute a lot to the overall plan makespan.

The graphs also show that using overlapping windows decreases the number of successful optimizations independently of whether the fixed point is or is not used. The explanation could be that if we optimize a given window then the overlapping window has already its first half optimal (for the halfstep method) so the chance for further improvement is lower. Still, overlapping windows lead to better plans (Table 3). The reason could be that this method allows the planner to reallocate faster the actions to proper layers of the parallel plan.

In summary, the experiment shows that there is still a big possibility for improving the runtime by focusing on windows with a higher chance for improvement. The open question is how to effectively identify such windows.

Figure 5 shows basically the same study as above for the expo method. The behavior is identical to the turbo method though the graphs look different. It is interesting to see that the fullstep method achieves very good efficiency for larger windows while this method produces the worst plans (Table 3). The reason could be that this method does the optimization of larger windows, which is time consuming, while the same improvement can be done using the smaller windows much faster. This confirms our hypothesis that it is better to focus on smaller windows.

The peaks in the graphs showing the number of windows in Figure 5 correspond to the sizes of windows that are actively tried by the expo method. Recall that we start with the window of size 3 and for each next iteration we increase the size of the window by half so we get windows of size 4, 6, 9, 13, 19, 28.

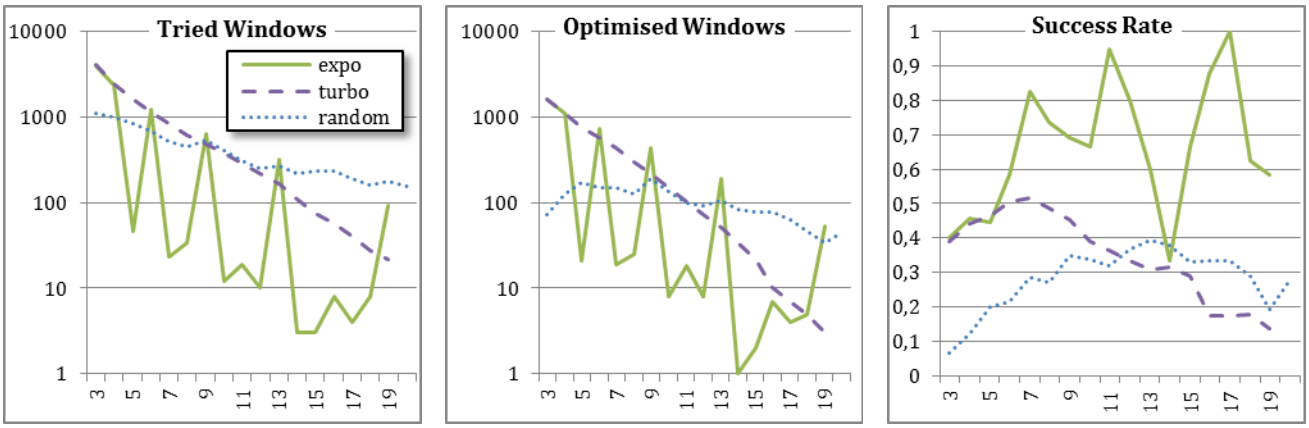

Figure 6. Comparison of all methods.

Nevertheless, the method may also explore windows of different sizes than those describe above. These are the windows covering the rest of the plan at the end of iterations.

To conclude the study, we compared the methods with the best success rates in a single graph. In particular, Figure 6 shows the fullstep versions without the fixed point of turbo and expo methods and the random method. The graphs show that the random method exploits uniformly windows of all sizes with very low success rate of actual improvements. The turbo method focuses on smaller windows and though it reaches the worst success rate for larger windows, the turbo method still finds the plans with the smallest makespan. The expo method has a very good success rate for the larger windows (most attempts lead to plan improvement), but because optimizing large plans is time consuming, the overall makespan is still worse than for the turbo method.

VII. Conclusions

In this paper we proposed a method for improving quality of plans by doing local enhancements of sub-optimal plans. In particular, we focused on improving plan makespan by optimizing sub-plans using a SAT-based method. We experimentally compared several methods how to select a sub-plan and its size for optimization. The best results were achieved by systematically exploring partially overlapping sub-plans with the constant increase of sub-plan size between the iterations. This method gives both the smallest overall makespan and the smallest runtime among the explored methods. All the methods significantly improved quality of plans produced by the LPG planner and made the quality comparable to plans generated by the optimal SASE planner. In fact, we achieved significantly better makespan score than the SASE planner thanks to solving more problems. Though the methods are still slower than SASE, they can find solutions for more problems thanks to exploiting the LPG planner (but any sub-optimal planner can be used to find the initial plan). The general conclusion from the experimental study is that it is worth optimizing a larger number of smaller sub-plans than trying a smaller number of larger sub-plans. The reason is that the runtime to optimize a sub-plan increases significantly with the size of the sub-plan.

By deeper experimental comparison we identified several points of the method that can contribute to better runtime. Based on our observations the runtime can be further improved by omitting sub-plans that cannot be improved and by better stopping criterion of the method. For example, sub-plans that cannot be improved can be discovered through some kind of fast planning-graph reasoning [3] or simply because they were already tried. This second criterion is slightly more complicated because even if the sub-plan has already been explored, the goal condition may change if a sub-plan in the later part of the plan has been changed. Regarding the stopping criterion, it seems that the algorithm can stop with some maximal window size as the large windows rarely lead to improvement if the smaller windows were carefully optimized and the remaining runtime is limited. Exploring these opportunities is part of future research.

Acknowledgements

Research is supported by the Czech Science Foundation under the contract P103/10/1287 and by the Grant Agency of Charles University under contracts no. 266111 and 600112.

References

[1] Berre, D.L., Parrain, A. 2010. *The Sat4j library, release 2.2.* JSAT 7 (2-3) (2010), pp. 59–64.

[2] Biere, A. 2012. *Precosat home page.* http://fmv.jku.at/precosat/ [accessed on February 2012]

[3] Blum, A. L. and Furst M. L. 1997. *Fast planning through planning graph analysis.* Artificial Intelligence, Volume 90 (1-2), 281-300, AAAI Press.

[4] Fern, A., Khardon, R., and Tadepalli, P. 2008. Learning track of the 6th international planning competition. http://eecs.oregonstate.edu/ipc-learn/ [accessed on July, 2012]

[5] Ge, Y., Yun, D. Y. Y. 1996. *Simultaneous Compression of Makespan and Number of Processors Using CRP*, Proceedings of IPPS 1996, pp. 332-338, IEEE Computer Society.

[6] Gerevini, A., Serina, I. 2002. *LPG: a Planner based on Local Search for Planning Graphs.* Proceedings of AIPS-2002, pp. 13-22, AAAI Press.

[7] Giunchiglia, E., Maratea, M. 2009. *Improving Plan Quality in SAT-Based Planning"*, Proceedings of AI*IA 2009, pp. 253-263, LNCS 5883, Springer.

[8] Helmert, M. 2006. *The Fast Downward Planning System*. Journal of Artificial Intelligence Research (JAIR) 26 (2006), pp. 191–246, AAAI Press.

[9] Hsu Ch-W., Wah, B. W., Huang, R., and Chen, Y. 2006. *Handling Soft Constraints and Preferences in SGPlan.* Proceedings of the ICAPS 2006 Workshop on Preferences and Soft Constraints in Planning.

[10] Huang, R., Chen, Y., Zhang, W. 2010. *A Novel Transition Based Encoding Scheme for Planning as Satisfiability.* Proceedings of AAAI 2010, pp. 89-94, AAAI Press.

[11] Kautz, H., Selman, B. 1999. *Unifying SAT-based and Graph-based Planning,* Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999), pp. 318-325, Morgan Kaufmann.

[12] Koenig, S. 2012 (editor). International Planning Competition (IPC), http://ipc.icaps-conference.org/, University of Southern California, [accessed on July, 2012].

[13] Surynek, P. 2011. Redundancy Elimination in Highly Parallel Solutions of Motion Coordination Problems, Proceedings of ICTAI 2011, pp. 701-708, IEEE Press.

[14] Surynek, P. 2012. A SAT-Based Approach to Cooperative Path-Finding Using All-Different Constraints, SoCS 2012, in press.

[15] Wang, K. C., Botea, A., Kilby, P. 2011. *Solution Quality Improvements for Massively Multi-Agent Pathfinding*. Proceedings of AAAI 2011, AAAI Press.