CHALMERS
UNIVERSITY OF TECHNOLOGY

Evaluation of high level methods for efficient planning as satisfiability

Downloaded from: https://research.chalmers.se, 2026-09-16 09:26 UTC

Citation for the original published paper (version of record):
Erös, E., Dahl, M., Falkman, P. et al (2021). Evaluation of high level methods for efficient planning as satisfiability. IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2021-September. http://dx.doi.org/10.1109/ETFA45728.2021.9613254

N.B. When citing this work, cite the original published paper.

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, or reuse of any copyrighted component of this work in other works.

(article starts on next page)

Evaluation of high level methods for efficient planning as satisfiability

Endre Erős[1], Martin Dahl[1], Petter Falkman[1] and Kristofer Bengtsson[1]

***Abstract*—Fast planning algorithms play a key role in intelligent automation systems where control sequences are constantly calculated. In order to determine which algorithms increase planning performance, we evaluate and compare several high level planning methods on a set of standard benchmarks. We focus on planning as satisfiability as the leading approach for solving difficult planning problems.**

***Index Terms*—automated planning, planning as satisfiability, artificial intelligence, SAT solvers, intelligent automation.**

I. Introduction

In order to create truly flexible automation systems, they need to include planning algorithms that can compute schedules and sequences of operations automatically. Taking correctness of planning algorithms for granted, efficiency is the key trait such algorithms must have in order to be applicable in the industry. Hence, this paper investigates and evaluates performances for some commonly used techniques in automated planning such as invariants, skipping steps and subgoaling.

Planning is a deliberative decision making process which yields sequences of operations that drive state change towards a goal. Planning as satisfiability, where planning problems are encoded as logical formulas and handled by SAT solvers [1] was introduced by Kautz and Selman in 1992 [2]. In a SAT based approach, a solver determines whether there exists a satisfiable assignment for a given Boolean formula that encodes the plan. It turned out to be quite a valuable contribution since SAT based planning excels in solving hard combinatorial planning problems with a high number of variables [3].

Over the last two decades, many methods have emerged with the aim to improve the efficiency of SAT solving [4] and SAT based planning [5]. Additionally, coupling a SAT solver with theory solvers, for example theories such as linear arithmetic, bit vectors, or arrays, SMT based planning techniques [6] can encode and tackle real-world scenarios and complex application domains [7].

In this paper, we investigate and compare several relevant methods for planning as satisfiability. We do not however study the detailed tuning of these solvers, but instead focus on what we call *high level* methods to improve the planning performance, which for example include the structure of the model and how to call the solvers. To the extent of our knowledge, there are no other studies that perform such an evaluation and comparison of high level planning methods. Thus, we hope that our effort will be useful, particularly for those who aim to implement new satisfiability based planners.

The paper is organized in the following way. The next section presents, evaluates, combines and compares planning methods in an evolutionary way, using a number of standard planning benchmarks. Some drawbacks and special combinations are discussed in Section 3. Section 4 presents the combined results and the paper is concluded in Section 5.

II. Planning methods

In planning as satisfiability, a planning problem is encoded into a logical formula F_j, where j represents the number of plan steps. This formula is then tested for satisfiability by a solver, which either returns UNSAT for a failed planning attempt or it returns SAT and provides a satisfying assignment which is parsed to yield a plan. If the result from the solver is UNSAT, the planning problem is encoded into a logical formula of length $j+1$ and tested again by the solver. This goes on sequentially until the solver returns SAT with a satisfying assignment, or until a maximum limit on the plan length is breached. More formally:

Definition 2.1: A planning problem Ψ with a plan length j can be encoded by the logical formula F_j:

$$F_j = I(t_0) \wedge \left(\bigwedge_{k=0}^{j-1} \bigvee_{n=0}^{m} (T_n(t_k, t_{k+1}))\right) \wedge G(t_j) \qquad (1)$$

where I are clauses that encode the initial state at time-step t_0, T_n are clauses that encode the m transitions of the model for all successive time-steps t_k and t_{k+1}, and G are clauses that encode the goal state at the horizon t_j.

We are going to explore how modifications to this planning approach affect the efficiency of planning, and in some cases ease modeling. In order to do that, we implement a set of algorithms which make use of different planning methods. Each subsection explores some methods and compares their performances on a set of classical planning benchmarks.

The benchmarks that are chosen to test the methods are: *gripper* [8], *blocksworld* [9], *rovers* [10], *barman* [11] and *childsnack* [12]. These benchmarks are chosen to represent different strengths and weaknesses of the following planning methods. An overview of evaluated methods is shown in Figure 1.

[1]Endre Erős, Martin Dahl, Petter Falkman and Kristofer Bengtsson are with Chalmers University of Technology, Electrical Engineering, Systems and Control Department, Automation Research Group, Gothenburg, Sweden. *endree@chalmers.se*, *martin.dahl@chalmers.se*, *petter.falkman@chalmers.se*, *kristofer.bengtsson@chalmers.se*

978-1-7281-2989-1/21/$31.00 ©2021 IEEE

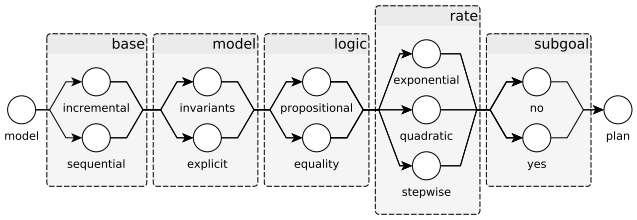

Fig. 1. Planning methods. There is a total of 48 method combinations, however we do not investigate all combinations in this paper. Starting from the left side, we evolve our algorithms with methods shown in this figure.

A. Incremental vs. sequential base

As mentioned earlier, a satisfiability based planning algorithm sequentially increases the plan length until a plan is found or until a limit is breached. The sequential algorithm (Algorithm 1) takes a planning problem and returns a result that either holds a plan or is empty, which represents that no solution was found.

An integer variable *step* keeps track of the step in the plan that the algorithm is currently at. At line 2 of Algorithm 1, the main loop checks whether the limit on the plan length is breached. It is important to limit the plan length so that the algorithm can terminate in case a solution can't be found, or where it takes a long time to calculate it.

A context *ctx* is created at line 3 to which the algorithm asserts a number of constraints, as shown between lines 4 and 9. The solver now checks the consistency of all assertions in the context *ctx* and if a satisfiable assignment exists, it is parsed into a plan and returned by the algorithm at lines 13 and 14. Otherwise, if no satisfiable assignment exists, the *step* variable is incremented by 1 and the whole procedure repeats while the statement $step \leq s_{max}$ holds.

Algorithm 1: ***Sequential***

```
   Input: (i, g, M, s_max)
   Output: planning_result
1  let step := 0;
2  while step ≤ s_max do
3      let ctx := new_context;
4      add constraint (ctx, i, 0);
5      add constraint (ctx, g, step);
6      let m_disj := disjunction for trans in M;
7      for s in (0 to step) do
8          add constraint (ctx, m_disj, s);
9      end
10     if check(ctx) == UNSAT then
11         step += 1;
12     else
13         let planning_result = parse(ctx.get_model);
14         return planning_result;
15     end
16 end
17 return new_empty_planning_result;
```

Algorithm 2: ***Incremental***

```
   Input: (i, g, M, s_max)
   Output: planning_result
1  let step := 0;
2  let ctx := new_context;
3  add constraint (ctx, i, step);
4  let bp := new_backtracking_point;
5  add constraint (ctx, g, step);
6  while step ≤ s_max do
7      step += 1;
8      if check(ctx) == UNSAT then
9          ctx.backtrack_to_level_bp;
10         let m_disj := disjunction for trans in M;
11         add constraint (ctx, m_disj, step);
12         let bp := new_backtracking_point;
13         add constraint (ctx, g, step);
14     else
15         let planning_result = parse(ctx.get_model);
16         return planning_result;
17     end
18 end
19 return new_empty_planning_result;
```

The downside of Algorithm 1 is that for every iteration where an assignment is not found, a new context has to be created. Gocht and Balyo showed in 2017 [13] that it is possible to achieve a significant speed-up by using an incremental SAT-solver. Instead of throwing away the context with all the accumulated data from previous results, the same context is used and constraints are just added on top.

The incremental algorithm (Algorithm 2) utilizes an incremental solver which makes it possible to add a *backtracking point* in each step so that the solver can choose which part of the context to save, and thus learn from previous attempts. In summary, some advantages of an incremental base solver are:

1) learnt clauses are kept
2) heuristic data is gathered
3) overhead from asserting the same clauses is reduced

At line 3 of Algorithm 2, the initial constraints for *step-0* are asserted into the context. Before asserting the goal, the algorithm creates a backtracking point so that if no satisfiable assignment was found, only the goal can be removed from the context, leaving the initial constraints.

At line 8, the algorithm checks the consistency of assignments in the context. If a satisfiable assignment is found, it is parsed and returned. Otherwise, the algorithm backtracks to the latest point, removing the assertions added after it from the context.

At line 11, the transitions are added, after which a new point is made so that the algorithm can backtrack to it if the assignments are not consistent with the goal assignment. This goes on until a plan is found or until a limit on the plan length is breached. Figure 2 compares planning efficiency between these two algorithms, where one utilizes an incremental solver and another one does not.

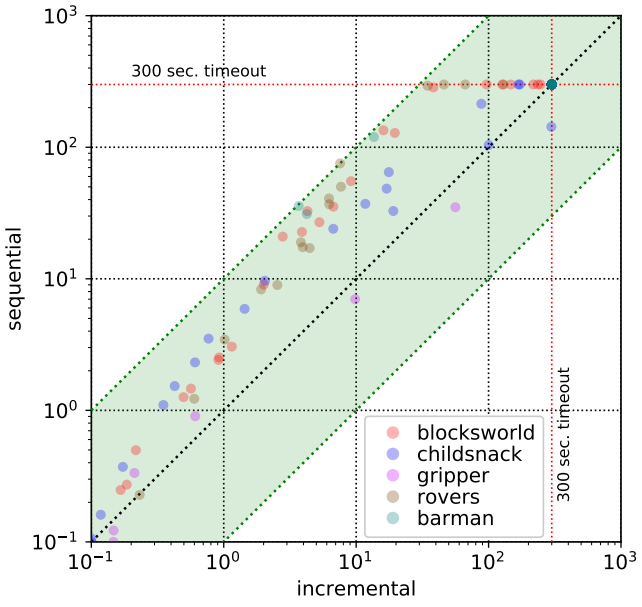

Fig. 2. This scatter plot shows that the algorithm that utilizes incremental solving is usually much faster in solving problems than the non-incremental algorithm because learned clauses are kept in the context. 73 (incremental) vs 61 (sequential) out of 200 instances were solved, where 97% are solved faster using the incremental base algorithm.

B. Invariants vs. explicit model

In the previous subsection, it was concluded that using the incremental base algorithm can substantially improve planning efficiency. From now on, we will only use the incremental algorithm as our base.

Invariants can be considered both as a modeling aid as well as a performance increasing method in automated planning. Invariants are specifications that must hold in every step of the calculated plan. More formally:

Definition 2.2: An *invariant* is a logical clause that must hold for all reachable states of Ψ, and can be encoded in the the logical formula F_j:

$$F_j = I(t_0) \wedge (\bigwedge_{k=0}^{j-1} \bigvee_{n=0}^{m} T_n(t_k, t_{k+1})) \wedge (\bigwedge_{k=0}^{j} N(t_k)) \wedge G(t_j)$$

where I are clauses that encode the initial state at time-step t_0, T_n are clauses that encode the m transitions of the model for all successive time-steps t_k and t_{k+1}, N are clauses that encode invariants for all time-steps t_k, and G are clauses that encode the goal state at the horizon t_j.

In essence, invariants prune states from the state space. They can be added to the model to forbid some undesired behavior, or to enforce tighter constraints and thus derive a more compact state space representation. In each case, the state space gets reduced and thus planning is more efficient.

For an existing problem, invariants can sometimes be synthesized to speed up planning [14]. However, invariants can often be a convenient tool to use for modelling different problems, but can in some cases be quite hard to use. Nevertheless, in order to illustrate how modelling using invariants can improve planning efficiency, let's look at a well known PDDL benchmark example, Blocksworld. The Blocksworld example is one of the most famous planning domains in artificial intelligence.

Imagine a set of blocks sitting on a table. The goal is to build one or more vertical stacks of blocks. The catch is that only one block may be moved at a time: it may either be placed on the table or placed atop another block. Because of this, any blocks that are, at a given time, under another block cannot be moved [9].

Invariants that should hold at any given time for the Blocksworld example are added to this problem:

1) b1 can't be on b2 if b2 is on b1
2) if holding any block, the gripper can't be empty
3) at most one block can be held
4) a block can't be on several different blocks
5) if a block is on the table, it is not on a block
6) if b1 is on b2, b2 is not clear

Now we have a much safer and smaller state space representation, and as an effect, planning is much faster. Figure 3 shows the difference between planning times with and without making use of planning invariants for a number of standard planning benchmarks.

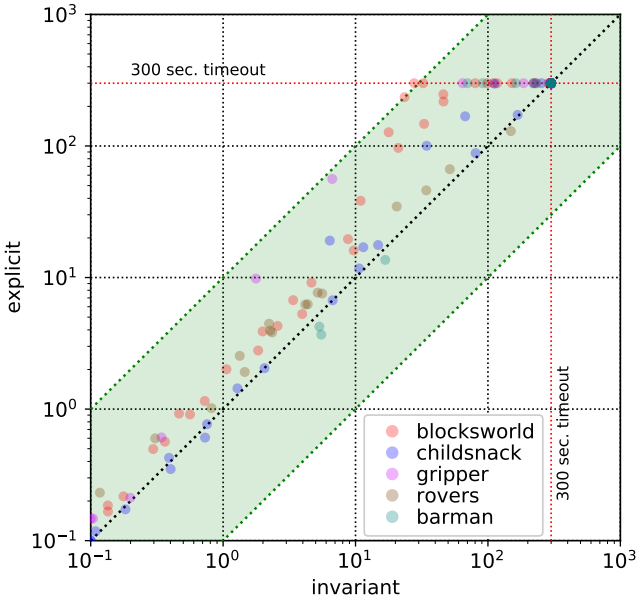

Fig. 3. On this scatter plot, it is shown that modelling the problem using invariants reduces the state space and thus speeds up planning , especially in the harder cases. 90 (invariants) vs 73 (explicit) out of 200 instances solved, where 97% are solved faster using invariants.

C. Equality vs. propositional logic

In the previous section, it was concluded that invariants have a beneficial effect for both the modeling and planning efficiency. From now on, we consider problems that are reinforced with additional invariants, if that is indeed possible.

Beside invariants, there are other things that can aid modeling. By using different *first-order* logic theories to increase expressiveness, some problems can be modeled in a much more convenient way compared to using pure propositional logic. In this paper, we only investigate equality logic since it is appropriate for modelling most classical planning problems.

Using a first-order theory with increased expressiveness to ease modeling can potentially lead to decreased decidability. Let's investigate the expressiveness and decidability aspects of both propositional and equality logic theories, in order to determine the beneficial effects on modelling and planning performances. A more comprehensive text on the problem of expressiveness vs. decidability, as well as decision procedures in general can be found in [15].

Definition 2.3: The following grammar defines the *syntax of formulas in propositional logic*:

$$formula : formula \wedge formula \mid \neg formula \mid atom$$
$$atom : Boolean - identifier \mid true \mid false$$

Example: Let's continue with the Blocksworld example and look at an action that is modeled in pure propositional logic:

$$\begin{aligned}
&(: action\ pick_up\\
&\quad : parameters\ (?x\ -\ block)\\
&\quad : precondition\\
&\quad\ (and\ (clear\ ?x)\\
&\qquad (ontable\ ?x)\\
&\qquad (handempty))\\
&\quad : effect\\
&\quad\ (and\ (not\ (ontable\ ?x))\\
&\qquad (not\ (clear\ ?x))\\
&\qquad (not\ (handempty))\\
&\qquad (holding\ ?x))
\end{aligned}$$

This action describes picking up a block from the table. In order to pick the block x up, the block has to be $ontable$ and $clear$ for picking up or placing another block on top of it. The hand has to be empty, as indicated by the $handempty$ variable. After the block has been picked up, it is no longer $ontable$, nor is it $clear$. The hand is no longer empty, it is now $holding$ the block x.

As it can be seen, to model this action for every block x, there has to be as many Boolean variables for $holding$, $ontable$ and $clear$ as there are defined blocks.

Let us now study the equality logic and the Blocksworld example using finite domain variables. If we restrict ourselves to using only finite domain variables, equality logic can be thought of as propositional logic where the atoms are equalities between variables or between variables and constants. A more formal definition of equality logic follows:

Definition 2.4: The following grammar defines the *syntax of formulas in equality logic*:

$$formula : formula \wedge formula \mid \neg formula \mid atom$$
$$atom : term\ =\ term$$
$$term : identifier \mid constant$$

where the identifiers are variables defined over a single finite domain of values. These values can be a subset of the set of Integers or Reals. Just the same, the variables can be of an enumeration type and define their own finite domain. Constants are elements from the same domain as identifiers.

Example: The same $pick_up$ action modeled using equality logic could look something along the lines of:

$$\begin{aligned}
&(: action\ pick_up\\
&\quad : parameters\ (?x\ -\ block)\\
&\quad : precondition\\
&\quad\ (and\ (clear\ ?x)\\
&\qquad (=\ (on\ ?x)\ \ ''table''))\\
&\quad : effect\\
&\quad (and\ (not\ (clear\ ?x))\\
&\qquad (=\ (on\ ?x)\ \ ''hand''))
\end{aligned}$$

Now, instead of using the Boolean variables $holding$ and $ontable$ for every block as well as the $handempty$ variable, we can use just one enumeration type variable on with a finite domain:

$$v_{on}^{D} = blocks \cup \{''table'',\ ''hand''\}$$

Both propositional and equality logic are NP-complete [16], [17], which means that they can model the same decision problems with not more than a polynomial difference in variables [15]. Certain problems are more conveniently modeled in equality logic compared to propositional logic, and for some other problems the opposite is true.

As for efficiency, the high level structure in the input equality logic formula can potentially be used to make the decision procedure work faster. This information may be lost if the problem is modelled directly in propositional logic [15].

Figure 4 compares planning efficiency between problems modeled in propositional logic and equality logic on 2 planning benchmarks.

D. Skipping steps

As it was mentioned before, the planning algorithms increment the plan length by one when an assignment is not found, after which the encoding for that length is tested for satisfiability. This is a good method when the yielded plan should be of minimal length. However, calculating the shortest length plan can sometimes be very slow, as Rintanen showed in [18] (shown in Figure 5).

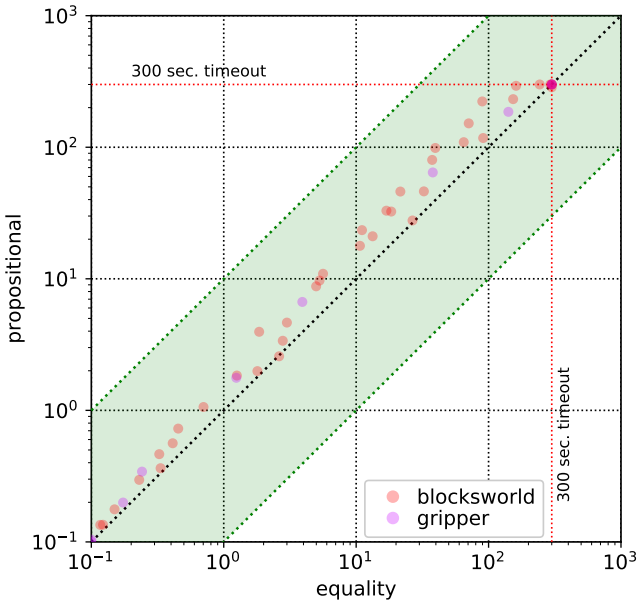

Fig. 4. This scatter plot shows that modelling problems using equality logic can sometimes increase planning efficiency. In some other cases, there is no high level structure that can be exploited by modelling the problem in equality logic, thus there is no big difference in performance. Thus, the appropriateness of using equality over propositional logic depends on the problem itself.

The evaluation cost of an unsatisfiable formula can be much higher than the evaluation cost of a satisfiable one, even if the latter is not of shortest length. Especially, when considering the sum of evaluation costs for all unsatisfiable formulas, it can be seen that the planner can spend a lot of time trying to find a satisfiable assignment. This is because the cost of evaluating the unsatisfiable formulas usually increases exponentially as the plan length increases [18].

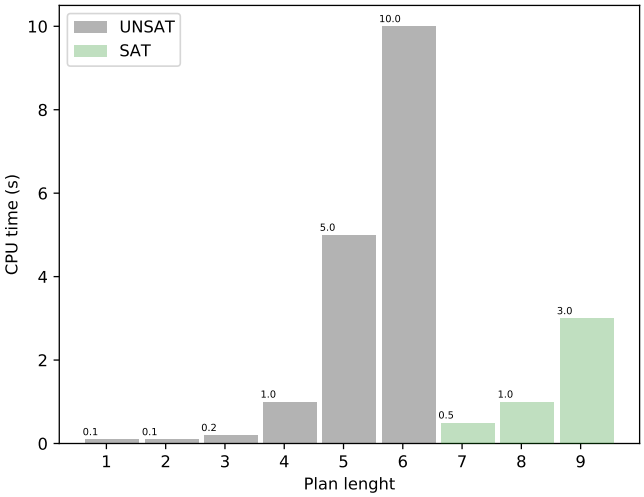

Fig. 5. Evaluation cost of the unsatisfiable formulas for plan lengths 1 to 6 and the satisfiable formulas for plan length 7 and higher, from [18]

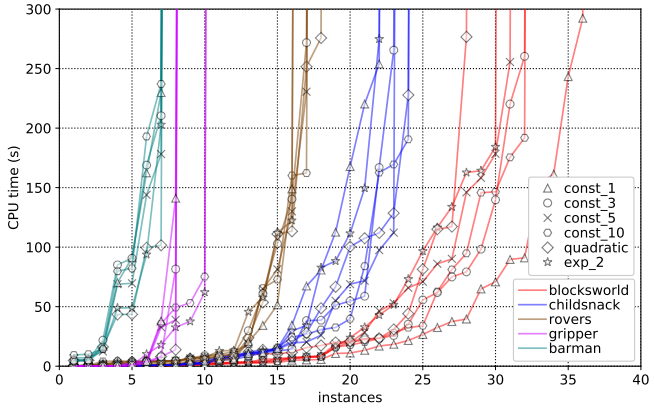

Fig. 6. This cactus plot shows the planning times of algorithms that skip planning steps in a certain way. The algorithms are essentially the same, it is only the step skipping rate that differs in them. Since skipping of steps is implemented on top of the incremental base algorithm, the solver has not learnt anything from the skipped iterations. In some cases, and as visible for the Blocksworld instance, skipping steps actually negates the good effects of using the incremental base solver. This will be further discussed later.

When finding the first satisfiable solution which yields the plan with the minimal length is not a strict requirement, an improvement in the planning time can sometimes be achieved. Rather than increasing the plan length by one after an assignment is not found, this improvement is realized by incrementing the plan length by a larger value. This allows us to skip some hard unsatisfiable instances which take a long time to evaluate.

As mentioned, skipping steps does not guarantee that the shortest plan will be yielded. For example, if we choose to evaluate only the encodings for the plan lengths 1, 2, 4 and 8 from Figure 5, we will skip the hard unsatisfiable instances 5 and 6, but also the first satisfiable instance 7. The plan length will be of length 8 instead of 7, but the benefit will be the decrease in planning time, 2.2s instead of 16.9s.

Incrementing the plan length after an assignment is not found can, for example, evolve in some constant or exponential rate. One could choose to solve for constantly increasing plan lengths $2i$ or $5i$ for integers $i \geq 1$, quadratic i^2, or for exponentialy increasing lengths 2^i for integers $i \geq 1$.

Figure 6 compares planning efficiency between planning algorithms that utilize different rates of skipping steps when using an incremental solver. Figure 7 compares the results when using a sequential solver.

E. Subgoaling

Skipping steps seems to help for some problems, and for others it does not. As we will discuss later, this is because we use the incremental base algorithm. Thus, we will continue to increment the rate stepwise.

When a goal is defined as a conjunction of several predicates, such predicates can be looked at as *subgoals*. Instead of asking the planner to reach the monolithic goal in one planning task, multiple planning tasks can be instantiated to plan for each subgoal. Having a simpler goal shortens the plan length

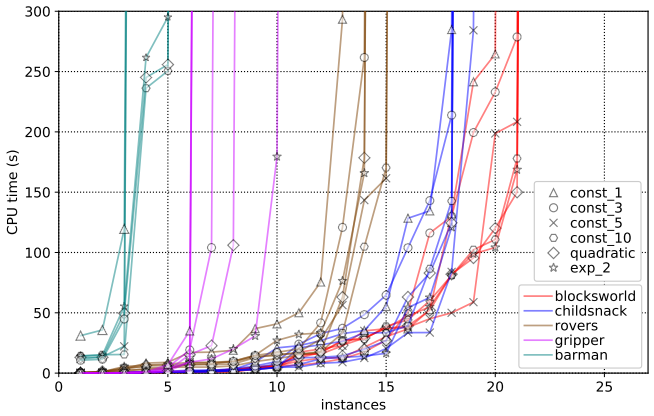

Fig. 7. The beneficial effect of skipping steps if much more visible if the base solver is non-incremental. This means that there is no drawback of skipping steps because the solver has not learnt anything from the skipped clauses.

and thus the planning time. As mentioned before, the cost of evaluating unsatisfiable formulas increases exponentially, thus it is usually faster to search for a large number of shorter plans than the other way around.

Algorithm 3 shows how subgoaling utilizes the incremental algorithm to solve for each conjunct of the monolithic goal. If that was indeed possible, the algorithm concatenates the results to yield the plan for the monolithic problem in line 18.

Subgoal ordering matters: If the subgoal order is not correct, finding a plan can sometimes be slower or even impossible. This depends quite much on the nature of the problem itself as planning problems often exhibit symmetry properties that could be exploited to speed up their solving.

Algorithm 3: ***Subgoal***

Input: *(i, G, M, s_{max})*
Output: *planning result*

```
1  let subresults := new_empty_list;
2  let first_sub := Incremental(i, G[0], M, s_max);
3  let n := 0;
4  subresults.push(first_sub);
5  Plan(first_sub, subresults, n, i, G, M, s_max);
6  function Plan(r, subresults, n, i, G, M, s_max) begin
7  if n < G.len() − 1 then
8  |   n = n + 1;
9  |   let new_i := if r.trace.len() == 0 then
10 |   |   i;
11 |   else
12 |   |   result.trace.tail();
13 |   end
14 |   let sub := Incremental(new_i, G[n], M, s_max);
15 |   subresults.push(sub);
16 |   Plan(sub, subresults, n, i, G, M, s_max);
17 else
18 |   return Concatenate(subresults);
19 end
```

For example, the two highly symmetrical problems Gripper and Childsnack benefit the most from subgoaling. There is no important subgoal order that has to be enforced in order to get a correct plan. On the other hand, the Blocksworld problem relies heavily on a correct order of subgoals.

F. Shortening the plan length

When planning with methods that skip steps or use subgoaling, yielded plans often have more steps than necessary. Sometimes, it is possible to remove chunks from a plan. An efficient way to do this is to detect loops in the plan and remove sections of it that lead back to an already visited state.

Usually, the requirement to visit the same state more than once is tracked by an additional variable, hence the states in the trace have the information about the complete state. If all variables are considered in a state, there should not be two of the same states in a plan. Having this in mind, we can remove parts of the plan to yield a valid plan of shorter length. Algorithm 4 shows how redundant sections of a plan are removed.

At line 5 of Algorithm 4, the *Find* function searches the plan for duplicate states and saves them, as well as their location in the plan, to a list. If duplicate states do exist, they cover a certain section of a plan that has to be removed. It can be that more than two duplicate states exist, as well as more than one pair of duplicates.

If that is the case, the section they cover can overlap in some way, as shown at B and C parts of Figure 8. Hence, the function *GetBiggest* finds the biggest section covered by two duplicates and the function *Remove* removes it from the plan. The plan is now shortened. However, it can be that it still contains some duplicate states. Because of this, the *Shorten* algorithm performs all the mentioned steps recursively until it exhausts all duplicates from a plan.

Algorithm 4: ***Shorten***

Input: *plan of length n*
Output: *plan of length m, $m \leq n$*

```
1  Shorten(plan);
2  function Shorten(plan) begin
3  let duplicates := new empty list;
4  for frame in plan do
5  |   duplicates.push(Find(frame.state, plan));
6  end
7  if not duplicates is empty then
8  |   let biggest := GetBiggest(duplicates, plan));
9  |   let new_trace := RemoveLoop(biggest, plan));
10 |   Shorten(new_plan);
11 else
12 |   return plan;
13 end
```

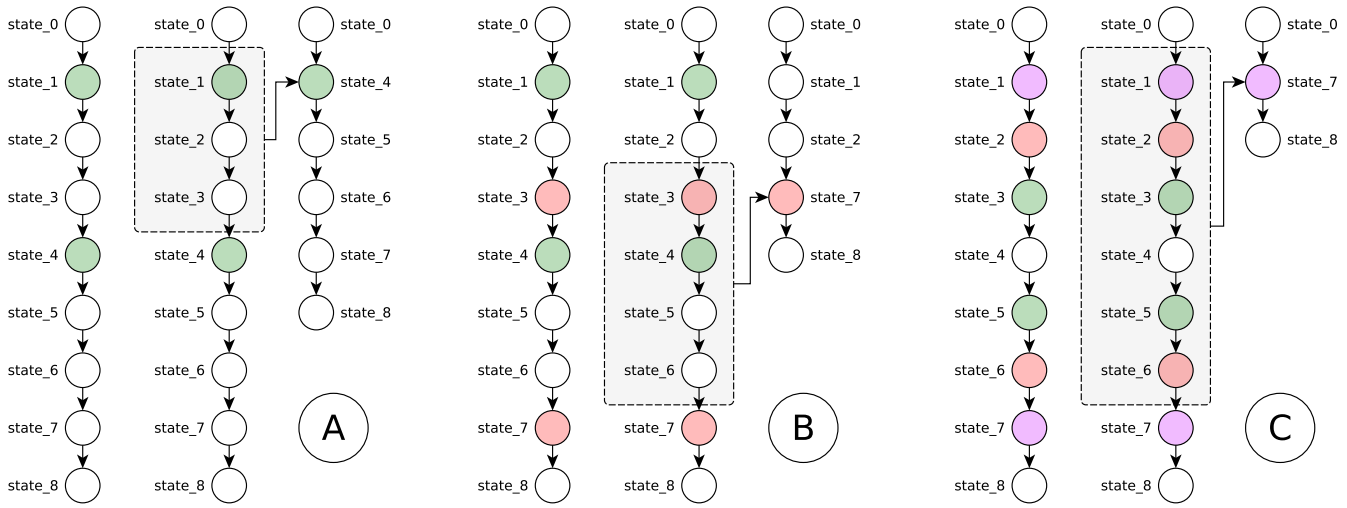

Fig. 8. Shorten plan scenarios. In scenario A, there is only one section of the plan that leads back to a same state (states 1 - 4), so the part is removed. In scenario 2, there are two overlapping loops, so in order to be more efficient, the algorithm removes the bigger loop. There might be loops still in the plan after removal so in the next iteration, the algorithm checks the plan for more loops. In scenario C, there is some nesting of loops, so the algorithm finds the biggest loop and removes both in one go.

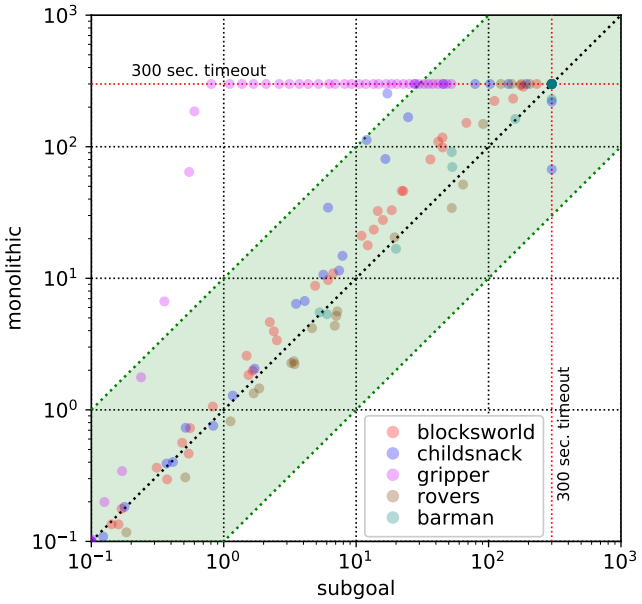

Fig. 9. On this scatter plot, it is shown that subgoaling will usually decrease planning time, especially in highly symmetrical problems.

G. Benchmarks

We ran the benchmarks on an Optiplex 9020 desktop PC with 8GB of RAM and an Intel Core i7-4790 CPU clocked at 3.60GHz. All algorithms used in this study were implemented using Z3's [19] quantifier free finite domain (QF_FD) theory, which supports propositional logic, bit-vector theories, pseudo-Boolean constraints, and enumeration data types. Figure 10 shows how different methods have improved the planning algorithm throughout the paper.

III. Discussion

Throughout this paper, we have investigated the effects of incremental solving, invariants, equality logic, skipping steps and subgoaling. We have done this in an evolutionary way, where a method from each subsection has improved the existing algorithm from the previous subsection. However, what about other method combinations and their performances?

What we learned from this study is that modelling with invariants always helps, both in easing the modeling itself and decreasing planning time. If we take that as a starting point of this discussion, we have to look at methods that fit well together with invariants. For example, incremental solving lets us keep some clauses in the context before we move on to the next step. Since invariants hold in each step of the plan, the clauses that encode them are never removed from the context.

Continuing on this, if we use an incremental base solver and skip some steps, the solver will not have the chance to learn new clauses from the skipped steps. However, what if we turn things around and use the sequential base together with skipping steps? Figure 7 shows the results of this combination. It can be seen that these results better match Rintatnen's [18] chart on Figure 5, thus it is more appropriate to combine non-incremental solving with skipping steps. Using non-incremental solving with skipping steps is probably beneficial if the computation is distributed among several cores. However, single-core planning benefits more using incremental solving, as seen on Figures 6 and 7.

Beside methods investigated in this paper, there is a multitude of other methods that weren't considered in this study. Probably one of the biggest contributors to speed in planning as satisfiability is parallel planning and it is presented in detail in [20]. Moreover, there is compositional planning as investigated in [21], and hierarchical planning [22].

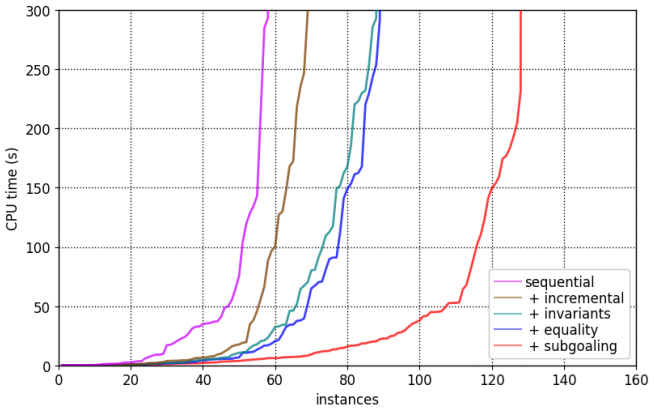

Fig. 10. Algorithm evolution throughout the paper.

Big performance improvements in planning as satisfiability can be unlocked when methods are altered on the low, SAT solving level. Some of these methods improve decision heuristics [23], restart heuristics [24] and data structures [25]. We refer to these methods as low level methods since they affect the performance of the solver itself. As such, they are of major interest in the field of planning as satisfiability, however they are out of the scope for this paper.

IV. Conclusion

We implemented and evaluated several high level planning methods using Z3 and showed how much each method contributes to an increase in planning performance. As discussed, this is by no means a complete study that covers all planning methods, however we feel that it is a useful reference and performance overview of some high level methods.

That being said, we see that there is great opportunity in studying automated planning, satisfiability, and method combinations that improve planing performance. Our next step is to use what we learned in this study in actual industrial implementations, as well as to investigate other high level methods that we mentioned in the discussions. Moreover, an investigative survey of low level planning methods, i.e. methods that operate on the SAT solving level is a planned step in our future work.

Acknowledgment

This research is supported by Chalmers University of Technology and VINNOVA under the projects Unification (contract nr. 2017-02245) and Unicorn (contract nr. 2017-03055).

References

[1] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518.

[2] H. Kautz and B. Selman, "Planning as satisfiability," in *Proceedings of the 10th European Conference on Artificial Intelligence*, ser. ECAI '92. USA: John Wiley & Sons, Inc., 1992, p. 359–363.

[3] J. Rintanen, "Search methods for classical and temporal planning," *Tutorials of the 21th European Conference on Artificial Intelligence (ECAI 2014)*, vol. 21, 2014.

[4] S. Alouneh, S. Abed, M. H. A. Shayeji, and R. Mesleh, "A comprehensive study and analysis on sat-solvers: advances, usages and achievements," *Artificial Intelligence Review*, pp. 1–27, 2018.

[5] J. Rintanen, "Planning as satisfiability: Heuristics," *Artificial Intelligence*, vol. 193, pp. 45 – 86, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370212001014

[6] J. E. Arxer, "Smt techniques for planning problems," 2018.

[7] A. Bit-Monnot, F. Leofante, L. Pulina, and A. Tacchella, "Smt-based planning for robots in smart factories," in *Advances and Trends in Artificial Intelligence. From Theory to Practice*, F. Wotawa, G. Friedrich, I. Pill, R. Koitz-Hristov, and M. Ali, Eds. Cham: Springer International Publishing, 2019, pp. 674–686.

[8] D. M. McDermott, "The 1998 ai planning systems competition," *AI Magazine*, vol. 21, no. 2, p. 35, Jun. 2000. [Online]. Available: https://ojs.aaai.org/index.php/aimagazine/article/view/1506

[9] F. Bacchus, "Aips 2000 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems," *AI Magazine*, vol. 22, no. 3, p. 47, Sep. 2001. [Online]. Available: https://ojs.aaai.org/index.php/aimagazine/article/view/1571

[10] D. Long and M. Fox, "The 3rd international planning competition: Results and analysis." *J. Artif. Intell. Res. (JAIR)*, vol. 20, pp. 1–59, 12 2003.

[11] A. Coles, A. Coles, A. Olaya, S. Jiménez, C. Linares López, S. Sanner, and S. Yoon, "A survey of the seventh international planning competition," *Ai Magazine*, vol. 33, pp. 83–88, 03 2012.

[12] M. Vallati, L. Chrpa, M. Grześ, T. L. McCluskey, M. Roberts, S. Sanner, and M. Editor, "The 2014 international planning competition: Progress and trends," *AI Magazine*, vol. 36, no. 3, pp. 90–98, Sep. 2015. [Online]. Available: https://ojs.aaai.org/index.php/aimagazine/article/view/2571

[13] S. Gocht and T. Balyo, "Accelerating sat based planning with incremental sat solving," in *ICAPS*, 2017.

[14] J. Rintanen, "An iterative algorithm for synthesizing invariants," in *AAAI/IAAI*, 2000.

[15] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, 2nd ed. Springer Publishing Company, Incorporated, 2016.

[16] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: Association for Computing Machinery, 1971, p. 151–158. [Online]. Available: https://doi.org/10.1145/800157.805047

[17] D. Kozen, "Positive first-order logic is np-complete," *IBM Journal of Research and Development*, vol. 25, no. 4, pp. 327–332, 1981.

[18] J. Rintanen, "Evaluation strategies for planning as satisfiability," in *Proceedings of the 16th European Conference on Artificial Intelligence*, ser. ECAI'04. NLD: IOS Press, 2004, p. 682–686.

[19] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[20] J. Rintanen, K. Heljanko, and I. Niemelä, "Planning as satisfiability: parallel plans and algorithms for plan search," *Artificial Intelligence*, vol. 170, no. 12, pp. 1031 – 1080, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370206000774

[21] E. Erős, M. Dahl, P. Falkman, and K. Bengtsson, "Towards compositional automated planning," 09 2020, pp. 416–423.

[22] D. Schreiber, D. Pellier, H. Fiorino, and T. Balyo, "Efficient sat encodings for hierarchical planning," 01 2019, pp. 531–538.

[23] J. Rintanen, "Heuristics for planning with sat," vol. 6308, 09 2010, pp. 414–428.

[24] J. Huang, "The effect of restarts on the efficiency of clause learning," in *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, ser. IJCAI'07. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, p. 2318–2323.

[25] M. W. Moskewicz, C. F. 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.