Lessons Learned from the CYK Algorithm for Parsing-based Verification of Hierarchical Plans

Simona Ondrčková, Roman Barták
Faculty of Mathematics and Physics
Charles University
Prague, Czech Republic
{ondrckova,bartak}@ktiml.mff.cuni.cz

Pascal Bercher
School of Computing
The Australian National University
Canberra, Australia
pascal.bercher@anu.edu.au

Gregor Behnke
Faculty of Engineering
ILLC, University of Amsterdam
The Netherlands
g.behnke@uva.nl

Abstract

Verification of hierarchical plans deals with the problem of whether an action sequence is causally consistent and can be obtained by a decomposition of a goal task. This second sub-problem (finding the decomposition) makes the verification problem NP-hard. The task decomposition structure is very close to a parsing tree of context-free grammar (CFG). Recently, the CFG-parsing algorithm by Cocke–Younger–Kasami (CYK) has been modified to verify hierarchical plans efficiently. Despite being fast, the algorithm can only handle totally-ordered planning domains. In this paper, we ask whether the ideas from the CYK algorithm can be extended to a more general parsing-based approach that covers *all* planning domains, i.e., including partially ordered ones. More specifically, we study the effect of modifying the domain model by limiting the number of sub-tasks in decomposition methods to two and the effect of changing the parsing strategy.

Introduction

Hierarchical planning is a form of planning that uses task decomposition. This is similar to how people solve tasks. We also tend to split difficult tasks into easier ones. Hierarchical planning can be used in a variety of different areas like robotics (Kaelbling and Lozano-Pérez 2011), machine learning (Mohr, Wever, and Hüllermeier 2018) or for automated assistance like for Do-It-Yourself home improvement projects (Bercher et al. 2021).

Plan verification is an opposite process to planning. It verifies whether a given goal task decomposes into the given plan. It can be used for instance in mixed initiative planning (Behnke et al. 2016).

There are currently three hierarchical plan verification approaches: via *SAT*, via *Planning*, and via *Parsing*. The SAT approach works by translating the plan verification problem into a boolean satisfiability problem (Behnke, Höller, and Biundo 2017). Verification via Planning translates the problem into a planning problem (Höller et al. 2022). Then, there are two approaches based on Parsing grammars. The first is a bottom-up approach inspired by parsing in grammars. We shall call it Bottom-up-Parsing Approach. (Barták, Maillard, and Cardoso 2018; Barták et al. 2020; 2021b).

Copyright © 2023 by the authors. All rights reserved.

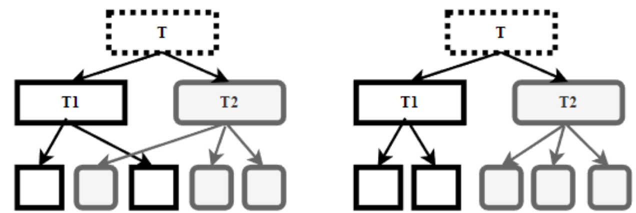

Figure 1: Example of task interleaving on the left and totally ordered tasks on the right

The second, which we call *CYK approach*, extends the Cocke–Younger–Kasami (CYK) algorithm, but only works for totally ordered HTN problems (Lin et al. 2023). There are also approaches that correct a plan if verification fails (Barták et al. 2021a; Lin, Grastien, and Bercher 2023), but that is not the focus of this article.

Both Parsing-based approaches focus on the similarity between hierarchical models and grammars (Höller et al. 2014; Höller et al. 2016; Barták and Maillard 2017). Actions correspond to terminal symbols in grammars and compound tasks correspond to non-terminal symbols. The plan verification question is akin to asking whether a word belongs to a language corresponding to given grammar. There are two main differences. First, actions can have preconditions and effects while terminal symbols do not. Second, the right-hand side of a grammar rule is totally ordered while the decomposition rules in hierarchical planning (called decomposition methods) can be partially ordered. Totally ordered domains are those in which ordering constraints of all methods induce total order on their tasks. They don't allow for interleaving (example in Figure 1) and if a domain is totally ordered then each task decomposes into a continuous series of actions.

Note that partially ordered methods can't be turned into totally ordered ones by simply enumerating all possible linearizations, which follows directly from complexity results showing that partially ordered HTN planning is undecidable (Geier and Bercher 2011; Erol, Hendler, and Nau 1996), whereas total-order HTN planning is not (Alford, Bercher, and Aha 2015; Erol, Hendler, and Nau 1996). Similarly, plan verification is also easier in the total-order setting: It can by done in poly-time in the total-order setting but is NP-complete in the partial-order setting (Behnke, Höller, and Biundo 2015; Bercher, Lin, and Alford 2022).

In this work we will focus on the Bottom-up-Parsing approach, because it is the most versatile of all approaches. As opposed to the SAT approach it can handle method preconditions (also the SAT approach is not strong in showing the invalidity of plans). As opposed to the planning approach it can handle *between conditions* and as opposed to the CYK algorithm it can handle partially ordered domains. It is also the only approach that can solve a variation of the plan verification problem without a given goal task (i.e., initial task network). However it is currently slower than the CYK approach and it solves less instances within the time limit than the Planning approach on valid instances. We aim to bring it on par with the other two algorithms.

The CYK approach uses a form of pre-processing, which we will call *2-regularization* (Behnke and Speck 2021) that limits the number of subtasks per task to a maximum of 2. When compared against the Bottom-up-Parsing approach this 2-regularization was attributed to its better performance. Due to the similarities between the CYK and the Bottom-up-Parsing approach we ask whether the 2-regularization could benefit the more general parsing inspired approach as well. In this article we aim to answer that question.

In previous publications of the Bottom-up-Parsing approach (Ondrčková et al. 2023) a different form of search (we shall call this search-system) was suggested as idea for possible future work. The focus of this article is mainly 2-regularization but we will also try to create a different search strategy.

The paper is structured as follows. First, we shall formally describe the hierarchical plan verification problem. Second, we will describe the Bottom-up-Parsing approach in detail. Third, we will introduce a grounder, which will be used to perform the 2-regularization and describe the 2-regularization. Fourth, we will explain the new search strategy. Finally, we will compare the new Bottom-up-Parsing approach with the other approaches.

Hierarchical Planning Verification

For hierarchical planning, which focuses on task decomposition (Erol, Hendler, and Nau 1996; Bercher, Alford, and Höller 2019), we use the STRIPS model (Fikes and Nilsson 1971) of actions. Each state is a set of propositions (P) that are true. Each action is given by a tuple $(\texttt{pre}(a), \texttt{eff}^+(a), \texttt{eff}^-(a))$, where $\texttt{pre}(a), \texttt{eff}^+(a), \texttt{eff}^-(a) \subseteq P$. The first set represents preconditions that must be true in order for the action to be valid and the other two represent the effects of the action on the state of the world. If all preconditions of an action are satisfied in a state S, we call the action applicable to state S. The state after the action was applied is defined by $\gamma(S, a) = (S \setminus \texttt{eff}^-(a)) \cup \texttt{eff}^+(a)$. An action sequence is executable if each action is applicable to the state before it. Let T be a compound task and $(\{T_1, ..., T_k\}, C)$ be a task network. The decomposition method M can be written as a rule that T decomposes to sub-tasks $T_1, ..., T_k$ under the constraints C:

$$T \rightarrow T_1, ..., T_k \ \ [C]$$

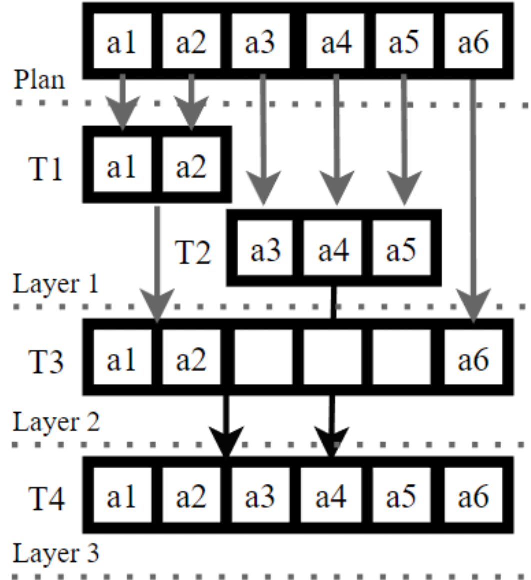

Figure 2: Example of parsing-based plan verification.

The order of sub-tasks is described explicitly in C, so the order in the sequence $T_1, ..., T_k$ does not matter. There are three types of constraints:

- $T_1 \prec T_2$: an ordering constraint meaning that in every plan, task T_1 is before task T_2.
- *before*(p, T): a precondition constraint meaning that in every plan, the proposition p holds in the state right before the first primitive sub-task of T is executed.
- *between*(T_1, p, T_2): a prevailing constraint meaning that in every plan, the proposition p holds in all the states between the last primitive sub-task of task T_1 and the first primitive sub-task of task T_2 is executed .

Hierarchical planning problems can be formalized as follows: *Given a description of tasks and their decompositions (in the domain model), initial state S and goal task G (in the problem instance), does an executable action sequence (plan) exist, such that G decomposes into it?* This sequence is the output.

Plan verification: *Given an action sequence (plan), initial state S and goal task G, can G be decomposed into the plan and is the plan executable?*

Plan Verification with Bottom-up-Parsing Algorithm

The Bottom-up-Parsing algorithm (Ondrčková et al. 2023; 2022; Barták et al. 2021b; 2020) begins by checking that preconditions of each action are satisfied. If not then the plan is invalid and the algorithm is finished. If they are satisfied, then it continues by building the decomposition structure layer by layer. An example is shown in Figure 2. This layer by layer building is akin to breadth-first search (BFS).

First each action of the plan will inform a method, where it is a sub-task that it's available. If all sub-tasks of a method are available, the method is *ready* - prepared for task creation. Then the algorithm picks the first method that

is *ready*. When creating a task from a method the algorithm creates all possible instances of the main task of the same method before it moves to the next *ready* method. We call this an *"at once"* system. For example we might have a method *deliver(From,To,What, Car)*, whose main task is *deliverT(To,What)*, with subtasks:*pick-up(From,Car,What)*, *drive(From,To,Car)*, *drop-off(To,What,Car)* Let these be available actions: *pick-up(L1,C1,P1)*, *drive(L1,L2,C1)*, *drive(L1,L3,C1)*, *drop-off(L2,P1,C1)* and *drop-off(L3,P1,C1)*. Then from this method two tasks are created: *deliverT(L2,P1)* and *deliverT(L3,P1)*.

After going through all methods, the algorithm creates all tasks that decompose only into actions. We shall call these layer 1 tasks. This is considered one *iteration* of the algorithm. Note that when a task is created the algorithm checks whether it's the goal task and whether it decomposes into all actions. If so, the plan is valid and the algorithm is finished. If not, the algorithm continues.

Created tasks will inform methods, where they are subtasks that they are available and if all sub-tasks of a method are ready, then the method is *ready*. These new methods will allow for creation of layer 2 tasks - tasks that either decompose only into layer 1 tasks or combination of layer 1 tasks and actions. This step would finish the next iteration of the algorithm. Essentially each iteration creates tasks of one layer. It continues this way layer by layer until it finds the goal task or until all possible tasks are created.

Task Creation In this section we will describe how the the Bottom-up-Parsing approach creates a new task. Let us assume we want to create task T using a method $T \rightarrow A, B, C, D$. It will begin by picking an instance from task A: $A1$ and checking it against the instance of task B: $B1$. If they are compatible it will take this combination of $A1B1$ and check against that instances of C and so on. It will do this all together, meaning that at the end it will get all possible instances of task T. It is important to note that if there is no compatible combination between A and B it will not try combining any C and D, because we already know that no task can be created. We always pick the newest sub-task first so if tasks A, B and C had instances created in iteration 2 but task D had instances created in iteration 4 we will pick D first and then try to combine it with A, B then C.

In previous work reordering of checks of tasks (starting with C for example, then A...) was suggested (Ondrčková et al. 2022) but the difference in performance was small. Note we will be using the Instance heuristic, which picks the subtasks that has the least instances first. Any reordering will still pick new sub-tasks first.

Grounder

In the latest article about the Bottom-up-parsing approach by Ondrčková et al. (2023) they utilize grounding to increase the performance of the approach. Grounding is a process that instantiates all variables with suitable constants. For this they used the grounder created by Behnke et al. (2020). This grounder can also perform the 2-regularization, which has not been used before on the Bottom-Up Parsing Approach.

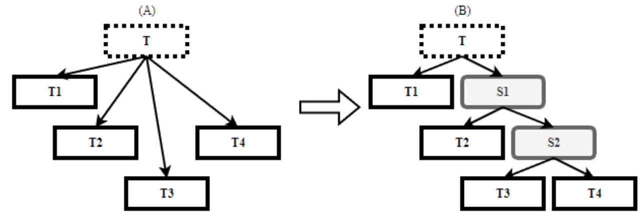

Figure 3: Example of 2 regularization.

2-regularization

As explained in Section the 2-regularization (Behnke and Speck 2021) is one of the reasons for better performance of the CYK algorithm compared to the Bottom-up-parsing approach on totally ordered domains (Lin et al. 2023). 2-regularization is a process of limiting the number of subtasks to a maximum of two by transforming the domain model. This is similar to one of the steps in transforming grammars into Chomsky normal form. For a task T with x subtasks (for $x > 2$), 2-regularization will create $x - 2$ new tasks. This can be seen in Figure 3, where part A represents the original task decomposition of task T and part B shows the decomposition after 2-regularization. Let us look at an example: $T \rightarrow T_1, T_2...T_x$. For this task T, 2-regularization will create $x - 2$ new tasks S such as this:

$$T \rightarrow T_1, S_1,$$

$$S_1 \rightarrow T_2, S_2;...;S_{x-3} \rightarrow T_{x-2}, S_{x-2};S_{x-2} \rightarrow T_{x-1}T_x$$

Greedy search

The latest version of the Bottom-up-parsing algorithm behaves like a BFS. However in previous publications it was suggested that moving to a different search strategy might benefit the algorithm (Ondrčková et al. 2023). So we decided to create a new greedy system that uses a static heuristic. In practice it will be similar to depth first search. We created a heuristic that calculates the minimum distance to the goal task g. This heuristic is calculated for each task at the beginning of the algorithm. The goal task's (and it's method's) goal distance is 0. Each method describing decomposition of some task S with minimum goal distance g, sets its subtasks' $\mathtt{T_1}...\mathtt{T_n}$ minimum goal distance to $\mathtt{g(S)} + \mathtt{1}$. A task T might be a sub-task in multiple methods $\mathtt{M_1}, ..\mathtt{M_n}$. If that is the case $\mathtt{g(T)} = \mathtt{min(g(M_1)}, ..\mathtt{g(M_n))} + \mathtt{1}$.

Then if multiple methods are ready, instead of picking the first one, the algorithm will pick the one with smallest g. In practice this means that we always try to pick the method that is the closest to the goal task.

We created two versions of this search. One that uses the *at once* system and one that creates tasks one by one. Essentially when a method is ready and selected, we only create one new task from this method and then we pick the next method with minimal heuristic value (this could be the same method again but a new method closer to the goal might become ready and therefore is selected). We call this the *one by one* system. There is no point in doing a BFS with the *one by one* system as after creating one task from the method the same method would just be selected again until all tasks of the method were created.

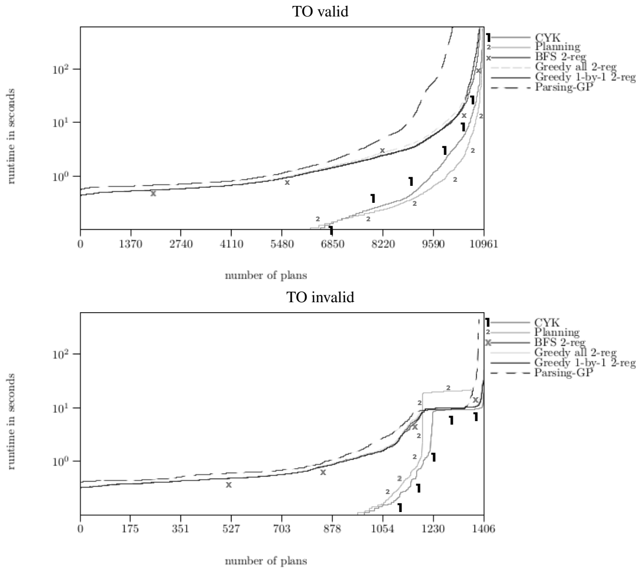

Figure 4: Runtime of verifiers on totally ordered domains

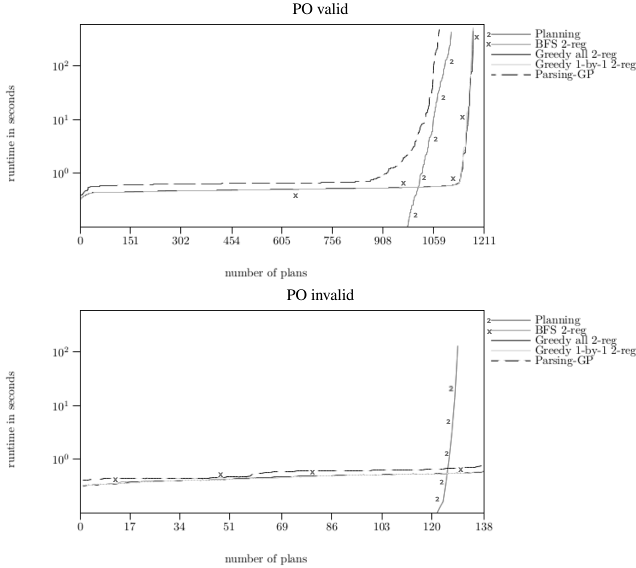

Figure 5: Runtime of verifiers on partially ordered domains

Empirical Evaluation

The experiments were run with a time limit of 10 minutes on Intel Xeon Gold 6130 processors with 8GB of RAM. For the best comparison we use the same 13714 instances that were used in previous experiments (Barták et al. 2021b; Höller et al. 2022; Ondrčková et al. 2022). We split the instances into four groups: to-valid and to-invalid for valid and invalid totally ordered domains and po-valid and po-invalid for partially ordered domains.

We run the test for 8 verifiers: *Parsing-GP*,*Planning*, *Planning 2*,*BFS 2*,*Greedy 1-by-1 2*,*Greedy all 2*, *Parsing GP 2* and *CYK 2*. *Parsing-GP* is the latest version of the Bottom-up-parsing approach as it was presented in (Ondrčková et al. 2023). *BFS 2*,*Greedy all 2* and *Greedy 1-by-1 2* are all new versions of the Bottom-up-parsing approach. *BFS 2* has implementation improvements over *Parsing GP*. Specifically some programming structures were changed (List to Dictionary) for faster input processing. *Greedy all 2* uses the greedy heuristic based search. *Greedy 1-by-1 2* also uses the greedy search strategy and it uses the one-by-one system instead of the at once system. Note that the algorithms are run either on grounded domains or grounded domains with 2 regularization (this is marked with 2 after the name of the algorithm). This is because it was shown in the previous work that the bottom-up-approach solves more instances on grounded domains (Ondrčková et al. 2023) and Planning uses grounded domains anyway. CYK algorithm by default uses 2-regularization. We did not include the SAT verifier as it was already shown in previous articles that the Bottom-up-parsing verifier outperforms it (Barták et al. 2021b).

Table 1 shows the number of solved instances for all verifiers in each of the four groups within the time limit.

First, let us look at the performance of our new versions (*BFS 2*,*Greedy all 2* and *Greedy 1-by-1*) against the previous version of the Bottom-up-parsing approach *Parsing-GP*. All three algorithms solve more instances in every group than (*Parsing-GP*). As can be seen in Figures 4 and 5 all three algorithms are also faster than the *Parsing-GP* version. Out of the three approaches *BFS 2* solves most instances. If compared to the *Parsing-GP* version it solves additional 6,8% of all instances (751 instances) on *to-valid* domains, 1% (14 instances) on *to-invalid*, 9,4% on (102 instances) *po-invalid* and on *po-valid* domains it solves one additional instance, which means it can now solve all *po-invalid* instances.

Next let us compare our best new version *BFS 2* against the *Planning* approach. Out of the four groups the *BFS 2* outperforms the *Planning* approach in three (on number of instances solved). In the remaining group of *to-valid* domains the *BFS 2* is now on par with *Planning* approach (less than 1% difference in total number of instances solved). In Figures 4 and 5 we can see that the BFS 2 approach has an overhead compared to the planning approach (on easier instances), but once the instances get difficult enough it outperforms the planning approach. The only exception are the *to-valid* domains, where both approaches are converging.

Finally, let us look at *BFS 2* and *CYK 2*. The CYK algorithm is faster on totally ordered domains. If we look at the number of instances solved, the *BFS 2* solves essentially the same amount when it comes to invalid totally ordered domains (1405 vs 1406). On valid totally ordered domains the *BFS 2* solves 10857 instances and CYK solves 10920 instances. The difference is 63 instances (0,57% of all instances). We can see that the Bottom-Up-Approach is now on par with the CYK algorithm in terms of number of instances solved and it still has the benefit of being able to solve partially ordered instances.

Question 1: What causes the increase in performance? Currently all three new versions of the Bottom-Up-Parsing approach outperform the *Parsing-GP* version. What is the main cause of the improvement? Is it 2-regularization, algorithmic improvement or search strategy? To answer this we ran an experiment, where we ran the *Parsing-GP* version on 2-regulated domains. The results can be seen in Table 1 under the *Parsing-GP 2* column. It solves 664 additional domains on *to-valid*. This is 88% of the 751 instances that the *BFS 2* solves additionally compared to *Parsing-GP*. On

	#inst	Parsing-GP	Planning	Planning 2	BFS 2	Greedy 1-by-1 2	Greedy all 2	Parsing-GP 2	CYK
plans	10961	10106 (92.20)	**10925 (99.67)**	10919 (99.62)	10857 (99.05)	10824 (98.75)	10839 (98.89)	10770 (98.26)	10920 (99.63)
inval-to	1406	1390 (98.86)	1364 (97.01)	1364 (97.01)	1405 (99.93)	1404 (99.86)	1405 (99.93)	1405 (99.93)	**1406 (100.00)**
po-plans	1211	1075 (88.77)	1112 (91.82)	996 (82.25)	1177 (97.19)	1174 (96.94)	**1178 (97.27)**	1170 (96.61)	NA
inval-po	138	136 (98.55)	129 (93.48)	130 (94.20)	**138 (100.00)**	**138 (100.00)**	**138 (100.00)**	**138 (100.00)**	NA

Table 1: Number of solved instances for different verifiers within the time limit.

Domain	#Plans	Verifier						
		Parsing-GP	Parsing-GP 2-reg	BFS	Greedy 1-by-1	Greedy all	Planning	CYK
AssemblyHierarchical	193	**193**	**193**	**193**	**193**	**193**	**193**	**193**
Depots	455	**455**	**455**	**455**	**455**	**455**	**455**	**455**
Entertainment	159	**159**	**159**	**159**	**159**	**159**	**159**	**159**
Freecell-Learned-ECAI-16	204	**204**	**204**	**204**	**204**	**204**	**204**	**204**
Hiking	565	**565**	**565**	**565**	**565**	**565**	**565**	**565**
Monroe-Fully-Observable	248	**248**	**248**	**248**	**248**	**248**	**248**	**248**
Monroe-Partially-Observable	217	**217**	**217**	**217**	**217**	**217**	**217**	**217**
Snake	183	**183**	**183**	**183**	**183**	**183**	**183**	**183**
Woodworking	494	**494**	**494**	**494**	**494**	**494**	**494**	**494**
Barman-BDI	423	390	420	**423**	**423**	**423**	421	**423**
Blocksworld-GTOHP	158	126	149	154	154	154	**158**	153
Childsnack	529	**529**	**529**	**529**	**529**	**529**	528	525
Elevator-Learned-ECAI-16	2812	2750	2784	2805	2802	2793	**2812**	**2812**
Logistics-Learned-ECAI-16	1108	723	1092	1103	1106	1106	**1108**	**1108**
Minecraft-Player	75	74	74	**75**	74	**75**	**75**	**75**
Multiarm-Blocksworld	443	381	416	428	396	421	**443**	**443**
Robot	117	90	90	92	89	90	**117**	**117**
Rover-GTOHP	509	**509**	508	**509**	**509**	**509**	**509**	**509**
Satellite-GTOHP	296	**296**	293	**296**	**296**	**296**	**296**	**296**
Transport	695	668	691	691	694	**695**	681	692
Blocksworld-HPDDL	172	165	165	166	166	163	**170**	168
Factories-simple	123	107	105	**122**	121	119	**122**	**122**
Minecraft-Regular	766	568	721	734	734	734	**755**	747
Towers	17	11	11	**12**	**12**	**12**	**12**	**12**

Table 2: Runtime of verifiers on totally ordered domains

partially ordered valid domains the *Parsing-GP 2* solves additional 95 instances, which is 93% out of the 102 additionally solved by *BFS 2*. This indicates that the main cause of improvement comes from 2-regularization. For invalid domains the *Parsing-GP 2* solves the same amount as the *BFS 2* version further proving our hypotheses.

Question 2: Why do the 2-regulated domains cause such an increase in performance? There are two reasons: retention of information and Last-subtasks-focus. Let us explain on *Parsing-GP* and *Parsing-GP 2*. The *Parsing-GP* algorithm and the previous versions do not retain information between iterations, but 2-regularization does. This allows us to avoid testing certain combinations of tasks multiple times.

Let us assume we have task $T \to A, B, C, D$ and let us assume we have 1 instance of A and 5 instances of B, C and D each. Also we shall assume that only one combination of C and D instances is valid: C1,D1. Lastly we shall assume that task A is the newest. Now let us look at what happens during two iterations of the Bottom-up-parsing algorithm.

In the first iteration the Bottom-up-parsing algorithm will reach this method. It will then set A to A1 and then check it with all instances Bs. If the combinations (of for example A1B1) are valid it will then check these combinations of AB with C and so on. It does this all at once. So it will compare all instances together to find all valid instances. This means it will do 1*5*5*5=125 comparisons. It will find 5 valid instances of T (A1C1D1 combined with every instance of B). Let us assume that during this iteration we get a new instance of A: A2. So once we reach the next iteration we go back to the same method, because we have a new sub-task option: A2. The algorithm will set A to A2 and check it with all the Bs and then these combinations A2B with all Cs and then all Ds. It will again do 1*5*5*5 comparisons. Clearly part of the computation is repetitive. The algorithm tried combining all Cs with all Ds, even though already in the first iteration we found out that only one combination is valid.

The 2-regularization helps retain this information. The decomposition rules for task T with 2-regularization look like this: $T \to A, T1$; $T1 \to B, T2$; $T2 \to C, D$. Figure 6 shows the process of task creation on 2-regulated domains. For the first iteration it only needs 35 checks as opposed to 125, this is due to the Last-Subtask-Focus which we will discuss later. At the next iteration after we get new A:A2, instead of recalculating the combinations again and therefore doing another 35 checks we actually retain previous information in T1 and we only have to do 5 more checks ($A2$ with all 5 instances of $T1$). In total the *Parsing-GP* version needs 2*125=250 checks to find the 10 valid instances, while the *Parsing-GP 2* only needs 40.

Essentially 2-regularization allows us to package valid combinations of a sequence of sub-tasks to the right of a given sub-task. Instances of $T1$ are all valid combinations

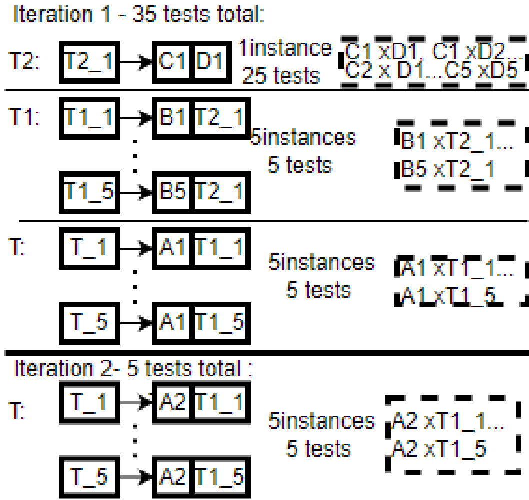

Figure 6: How 2 regularization creates tasks

of the original sub-tasks B,C,D. Instances of $T2$ are simply all valid combinations of C and D. So every-time we get new sub-task (like A2) we only have to check with the "package" of all the valid combinations to the right of it (T1 instances).

The 2-regularization should therefore be especially helpful on domains that are complicated and have multiple decompositions for the same tasks (which will cause us to go back to the same method in multiple iterations). In Table 2 we show number of instances solved per domain on *to-valid* domains. We can see significant improvements in performance on complicated domains such as Minecraft-Regular.

Due to the combination of the bottom-up nature of the approach with the 2-regularization the algorithm will always combine the last subtasks first (we call this Last-Subtask-Focus). This can be especially beneficial if some combinations of these tasks is invalid. If the tasks are in the right order and some of the tail end tasks like in the example above are invalid, 2-regularization can help us save number of tests by not checking combinations repeatedly (35 vs 125).

Question 3: Is it possible that non regulated domains might perform better than 2-regulated ones? Yes. Clearly 2-regularization only helps if the domain contains methods with more than 2 sub-tasks. Without the 2-regularization the tasks combination is based on Instances and task novelty (in our example that corresponds with going from left to right). It will try to combine A with B and then only combine the valid combinations of that with C. So if a combination of all sub-tasks A and B would be invalid, the original approach would not even try to combine it with sub-tasks C or D. This would save it multiple checks. Since 2-regularization has the Last-Subtask-Focus it would first create all valid combinations of C and D and package that into T2, then all valid combinations of B,C,D and package that into T1 and then try it with A only to find that no combination is valid.

Question 4: Doesn't the 2-regularization of the domains have the same effect as reordering task checks? We mean checking the sub-tasks in different order (not ordering in the domain). The answer is no. This is because of retention of information. Let's imagine we reordered the task checks so that the original approach would also have this Last-subtask-focus and start by combining these last subtasks. Let us imagine that we are at iteration 2 in the previous example. The 2-regulated approach would need 5 checks to find these new instances of T (by combining T1s with A2). The original approach would take A2 (it always takes new tasks first) and try to combine it with all Ds then all Cs (25 checks). Then that one valid combination A2D1C1 with all Bs (5 checks) for total of 30 checks. So it would recalculate the combination of Cs and Ds from iteration 1.

Question 5: How did changing to the Greedy search strategy and the one by one system affect performance? It didn't increase the performance of the algorithm in any significant way. We believe this might change if different heuristic was used. Currently we used a static heuristic. That means that it's calculated once at the start of the algorithm and then the algorithm greedily picks the task that is closest to the goal task. In future work we consider a dynamic heuristic that would combine the goal distance of a task with the layer of the task. This would be similar to A* algorithm.

Conclusion

In this article our goal was to study whether ideas presented in a recent publication of a new plan verification approach using the CYK algorithm (Lin et al. 2023) could be used in the current Bottom-up-parsing approach. If so, to integrate them and test the new version against other approaches. This is of interest because the Bottom-up-parsing approach is more versatile and it can solve partially ordered,while the CYK algorithm can only solve totally ordered domains.

As a secondary task we also tried to implement some of the ideas presented in previous works of the Bottom-up-parsing approach like moving away from the BFS like search. This was not effective but it has presented an idea for a heuristic that might benefit the approach in the future.

The main focus of this article was on 2-regularization, which limits the number of sub-tasks of a task to a maximum of two. This has proven to be very successful. While the Bottom-up-parsing approach is slower than the CYK algorithm on totally ordered domains, it solves nearly the same amount of instances within the time limit (less than 1% difference). It can also solve instances in partially ordered domains,which the CYK algorithm cannot. It also outperforms the planning approach (which is a an approach that can solve both totally and partially ordered instances). The Bottom-up-parsing approach now solves more instances than the planning approach in three out of four domain groups and it's on par with it in the last group (less than 1% difference).

Acknowledgments

Research is supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. S. Ondrčková is supported by the Charles University project GA UK number 280122.

References

Alford, R.; Bercher, P.; and Aha, D. 2015. Tight bounds for HTN planning. In *Proceedings of the 25th International*

Conference on Automated Planning and Scheduling (ICAPS 2015), 7–15. AAAI Press.

Barták, R., and Maillard, A. 2017. Attribute grammars with set attributes and global constraints as a unifying framework for planning domain models. In *PPDP 2017*, 39–48. ACM.

Barták, R.; Ondrčková, S.; Maillard, A.; Behnke, G.; and Bercher, P. 2020. A novel parsing-based approach for verification of hierarchical plans. In *Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2020)*, 118–125. IEEE.

Barták, R.; Ondrčková, S.; Behnke, G.; and Bercher, P. 2021a. Correcting hierarchical plans by action deletion. In *KR 2021*, 99–109. IJCAI.

Barták, R.; Ondrčková, S.; Behnke, G.; and Bercher, P. 2021b. On the verification of totally-ordered HTN plans. In *Proceedings of the 33rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2021)*, 263–267. IEEE.

Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of hierarchical plans via parsing of attribute grammars. In *ICAPS 2018*, 11–19. AAAI Press.

Behnke, G., and Speck, D. 2021. Symbolic search for optimal total-order HTN planning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021)*, 11744–11754. AAAI Press.

Behnke, G.; Höller, D.; Bercher, P.; and Biundo, S. 2016. Change the plan – How hard can that be? In *ICAPS 2016*, 38–46. AAAI Press.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On succinct groundings of HTN planning problems. In *AAAI 2020*, 9775–9784. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the complexity of HTN plan verification and its implications for plan recognition. In *ICAPS 2015*, 25–33. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2017. This is a solution! (... but is it though?) - verifying solutions of hierarchical planning problems. In *ICAPS 2017*, 20–28. AAAI Press.

Bercher, P.; Alford, R.; and Höller, D. 2019. A survey on hierarchical planning – one abstract idea, many concrete realizations. In *IJCAI 2019*, 6267–6275. IJCAI.

Bercher, P.; Behnke, G.; Kraus, M.; Schiller, M.; Manstetten, D.; Dambier, M.; Dorna, M.; Minker, W.; Glimm, B.; and Biundo, S. 2021. Do it yourself, but not alone: *Companion*-technology for home improvement – bringing a planning-based interactive DIY assistant to life. *Künstliche Intelligenz – Special Issue on NLP and Semantics* 35:367–375.

Bercher, P.; Lin, S.; and Alford, R. 2022. Tight bounds for hybrid planning. In *IJCAI-ECAI 2022*, 4597–4605. IJCAI.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity Results for HTN Planning. *Annals of Mathematics and AI* 18(1):69–93.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. In *IJCAI 1971*, 608–620.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 1955–1961. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *ECAI 2014*, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *ICAPS 2016*, 158–165. AAAI Press.

Höller, D.; Wichlacz, J.; Bercher, P.; and Behnke, G. 2022. Compiling HTN plan verification problems into HTN planning problems. In *ICAPS 2022*, 145–150. AAAI Press.

Kaelbling, L. P., and Lozano-Pérez, T. 2011. Hierarchical task and motion planning in the now. In *IROS 2011*, 1470–1477. IEEE.

Lin, S.; Behnke, G.; Ondrčková, S.; Barták, R.; and Bercher, P. 2023. On total-order HTN plan verification with method preconditions – an extension of the CYK parsing algorithm. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI 2023)*. AAAI Press.

Lin, S.; Grastien, A.; and Bercher, P. 2023. Towards automated modeling assistance: An efficient approach for repairing flawed planning domains. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI 2023)*. AAAI Press.

Mohr, F.; Wever, M.; and Hüllermeier, E. 2018. ML-plan: Automated machine learning via hierarchical planning. *Machine Learning* 107(8):1495–1515.

Ondrčková, S.; Barták, R.; Bercher, P.; and Behnke, G. 2022. On heuristics for parsing-based verification of hierarchical plans with a goal task. In *Proceedings of the 35th International Florida Artificial Intelligence Research Society Conference (FLAIRS 2022)*.

Ondrčková, S.; Barták, R.; Bercher, P.; and Behnke, G. 2023. On the impact of grounding on htn plan verification via parsing. In *Proceedings of the 15th International Conference on Agents and Artificial Intelligence (ICAART 2023)*. Scitepress.