ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie

Institut für
Künstliche Intelligenz

Optimization of Partially Ordered Plans Through Justification Algorithms

Bachelorarbeit an der Universität Ulm

Vorgelegt von
Linus Diepold

Gutachter:
Prof. Dr. Susanne Biundo-Stephan

Betreuer:
Dr. Pascal Bercher
Cornelia Olz

2019

Fassung September 13, 2019

Satz: PDF-LaTeX 2_ε

Abstract

The purpose of planning is to get from an initial state to a goal state by ordering plan steps between them. The creation of optimal plans demands a price in runtime performance therefore some planning algorithms create non-optimal plans. In order to deal with those Eugene Fink and Qiang Yang presented Algorithms which are able to optimize a specific set of non-optimal plans. They categorize plan steps inside of a plan as either justified or non-justified. Every non-justified plan step can be removed without hurting the correctness of the plan. In this work these algorithms will be applied on Partial-order (PO) and Partial-order-causal-link (POCL) plans. Furthermore, the effectiveness of the algorithm will be evaluated to provide a proof of concept. While the justification algorithms optimize plans by removing plan steps, plans can also be optimized by a reordering of the plan steps to reduce its makespan. In this work both methods will be evaluated and compared by using metrics like reduction of plan steps, reduction of makespan, and necessary computation time.

Contents

1 Introduction

Creating and executing a plan will get you from a starting point to a goal. The faster a plan is being created the faster it can be executed. That is why planning systems want to produce plans with as little computation time as possible. Some planning systems will crate non-optimal plans in order to enhance their performance.

This work will present ways to deal with non-optimal plans through optimization. The plan justification approach follows the idea of removing plan steps in non-optimal plans that are not necessary for reaching a goal. For example, a plan is given which is supposed to provide coffee supply for an office. The first plan step is "boil hot water", the second is "filter coffee" and the third and final plan step is "boil hot water". Instead of trying to search better options to brew coffee, justification algorithms will search plan steps that can be removed while still achieving the goal. For the purpose of identifying unnecessary plan steps [FY92] have defined justification criteria. If there is a plan step that can not satisfy a justification criteria, the plan step is unjustified and can be removed. However, if a plan step is justified it could still be unnecessary for achieving the goal. In general finding optimal plans can be a very high complex problem depending on the planning domain [Hel03]. That is why there are 4 types of justification that vary in strength. The stronger the justification the harder it gets for a plan step to fulfill the criteria. Therefore, algorithms using stronger justification types can potentially remove more plan steps. Along with the strength of justification the runtime of the related algorithm increases. The best possible plan is called perfectly justified, which is the strongest justification type. Finding a perfectly justified plan is a NP-complete problem [FY92]. Since computation is an important property to optimization algorithms, the presented justification algorithms are limited to polynomial time. Therefore, perfect justification will not be evaluated in this work. For the same reason the tested justification algorithms are also not able to rearrange ordering constraints to repair a plan, which is a NP-complete problem as showed in [OB19]. While the concept of justification can be used in total-order plans as in the example above, the focus of this work is on the application on PO and POCL plans [MR91]. The concept of plan step justification can also be used to provide explanations for a plan to a user. Which can increase the trust and ability to understand a plan to a user which was shown in [BBG+14].

Since PO and POCL plans allow nonlinear ordering of plan steps, multiple plan steps can be executed at the same time. The more plan steps that can be executed parallel the better the plan is. The makespan of a plan describes the length of the longest path from the initial state to the first goal state. It serves as a metric on how fast a plan can

be executed thanks to parallel execution of plan steps. If the justification algorithms remove plan steps that cannot be executed parallel to another plan step, they reduce the makespan of a plan.

In addition to that a second method of makespan optimization is presented in this work. This method will not remove plan steps, instead it will reorder the plan steps in a way that allows for more parallel execution. This is accomplished by translating the problem of creating an ordering of plan steps into a hierarchical planning problem which can be solved by the hybrid planning system PANDA [BKB14]. For this the ordering constraints and causal links of a plan have to be removed and the plan steps have to be handled as primitive tasks. Then PANDA is able to solve these hierarchical planning problems with a makespan minimizing A-star heuristic. Therefore, PANDA is able to insert ordering constraints in a way that creates an optimal makespan for the given plan steps.

2 Formal Framework

Planning problems in general want to change the world state through actions to achieve some sort of goal. STRIPS [FN71] is a notation to formalize such problems. A planning domain $D = (V, A)$ in STRIPS is defined by a finite set of boolean state variables V and a finite set of actions A. The current state s of a planning domain is defined by the set of state variables. For every planning Domain there is a set of $S = 2^V$ possible states. A planning problem $P = (D, s_{init}, g)$ consists of a planning domain, an initial state $s_{init} \in S$ and a goal description $g \subseteq V$ which is a set of state variables. In order to solve a planning problem the initial state needs to be changed with actions to achieve a goal state. A state s is a goal state S_g if $g \subseteq s$. An action $a \in A$ is defined by a 3-tuple: $(p(a), e^-(a), e^+(a))$. $p(a)$ is the set of preconditions, which contains all state variables that have to be true, at the time a is performed. $e^-(a)$ is the set of delete effects. If the action a is being performed, all state variables of the set $e^-(a)$, that are true in the current state are negated in the next state. $e^+(a)$ is the set of add effects. If the action a is performed, all literals of $e^+(a)$ are true in the next state. A state variable cannot be set false and true at the same time, that is why $\forall a \in A\ e^-(a) \cap e^+(a) = \emptyset$. Therefore, every state variable is either true or false in every valid state. In state s an action a can be performed if and only if $p(a) \subseteq s$. If the action a is performed in the state s, a new state s' is being obtained with $s' = (s \setminus e^-(a)) \cup e^+(a)$.

2.1 TO-Planning

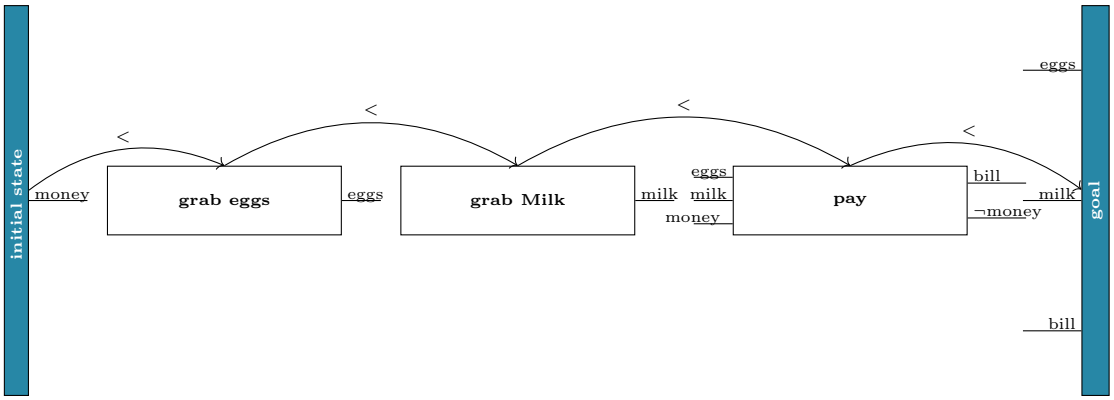
A total-order plan (TO plan) Π is a solution to a planning problem and consists of a sequence of actions. The sequence of a TO plan is transforming the state of the planning domain, starting with the initial state and finishing with a goal state.

Definition 1. Establishment in total-order plans *Let Π be a total-order plan. Let ps_1 and ps_2 be two plan steps of the plan $\Pi, ps_1, ps_2 \in \Pi, v \in e^+(action(ps_1))$, and $v \in p(action(ps_2))$. Then ps_1 establishes v for ps_2 if*

- *1. $ps_2 \prec ps_1$, and*
- *2. $\forall ps \in \Pi$, if $ps_2 \prec ps \prec ps_1$ then $v \notin e^+(action(ps))$ and $v \notin e^-(action(ps))$*

Definition 2. Correctness of a TO plan *A sequence of actions* Π *is a TO plan for a planning problem if it is legal and achieves the goal. A TO plan is legal if for every action* $a_1 \in \Pi$ *every precondition* $v \in p(a)$ *has been established by another plan step* $a_2 \in \Pi$*. A TO plan achieves the goal if the application of* Π *on the initial state results in a goal state.*

Figure 2.1 provides an example for a plan which describes the task of getting groceries and paying for them. In this example the state variable *money* is true in the initial state. This state needs to be transformed till a goal state is reached which has to contain the set of variables $\{milk, eggs, bill\}$.



.

Figure 2.1: A total-order plan for the problem in Figure 2.1. The arrows imply the order in which the actions are preformed. Preconditions are on the left side of an action while add and delete effects are on the right side

2.2 PO Planning

As an alternative to total-order plans, planning problems can be solved with partial-order plans (PO plans). A PO plan $\Pi = (PS, \prec)$ consists of a set of plan steps PS and a set of ordering constraints $\prec$. To allow actions to be executed more than once, we need to differentiate between the actions of the planning domain, and the set of plan steps. In a planning domain there is the set of all available actions. To reach the goal they might need to be executed multiple times, or they could even be skipped if they are not necessary at all. The actions that are used in a PO plan need to have a fixed amount of occurrences. So for example if an action needs to be executed twice in a PO plan, the set of plan steps contains two occurrences of the same action. To avoid ambiguities, the plan steps $ps \in PS$ are tuples $ps = (l, a)$ consisting of an action a and a unique label l (e.g. numbers). An action a that is part of a plan step $ps = (l, a)$ can get referred by using the function $action(ps) = a$ An ordering constraint between two plan steps ps_1, ps_2 is defined by: $ps_1 \prec ps_2$. This constraint means that ps_1 has to be performed before ps_2 is performed. Ordering constraints are transitive so if $ps_1 \prec ps_2$ and $ps_2 \prec ps_3$ would imply $ps_1 \prec ps_3$. A linearization $\overline{\Pi}$ of a PO plan Π is a total-order version of the list of plan steps, that do not violate any of the ordering constraints. In a PO plan the set of ordering constraints is transitive which means if the ordering constraints $ps_1 \prec ps_2$ and $ps_2 \prec ps_3$ are given the ordering constraint $ps_1 \prec ps_3$ also holds. Therefore, some ordering constraints, which are already implied by the transitivity, can be added to a PO plan without changing the ordering of the plan.

Definition 3. Correctness of a PO plan *A Tuple $\Pi = (PS, \prec)$ is a PO plan for a planning problem if and only if every linearization of Π is a TO plan for the planning problem.*

Definition 4. Establishment in PO plans [KTY91] *A plan step ps_1 with $v \in e^+(action(ps_1))$ establishes a variable v for ps_2 with $v \in p(action(ps_2))$ in a PO plan if and only if the following conditions hold: .*

- $ps_1 \prec ps_2$
- $\forall ps_3 \in \Pi$ *if* $ps_1 \prec ps_3 \prec ps_2$ *then* $v \notin e^+(action(ps_3), e^-(action(ps_3))$

Definition 5. White Knights [KN96] *If there are two plan steps ps_1, ps_2 in a PO plan with $v \in e^+(action(ps_1)), p(action(ps_2))$ and $ps_1 \prec ps_2$ then ps_1 does not establish v for ps_2 if there is a plan step ps_3 with $ps_1 \prec ps_3 \prec ps_2$ and $v \in e^-(action(ps_3))$. A plan step ps_4 is called a White Knight if $v \in e^+(action(ps_4))$ and $ps_3 \prec ps_4 \prec ps_2$ holds.*

In Figure 2.3 plan steps grab eggs and grab milk are ordered after the initial state and ordered before pay. The goal is ordered after pay. Notice there is no constraint between grab eggs and grab milk, that means they can be ordered arbitrarily.

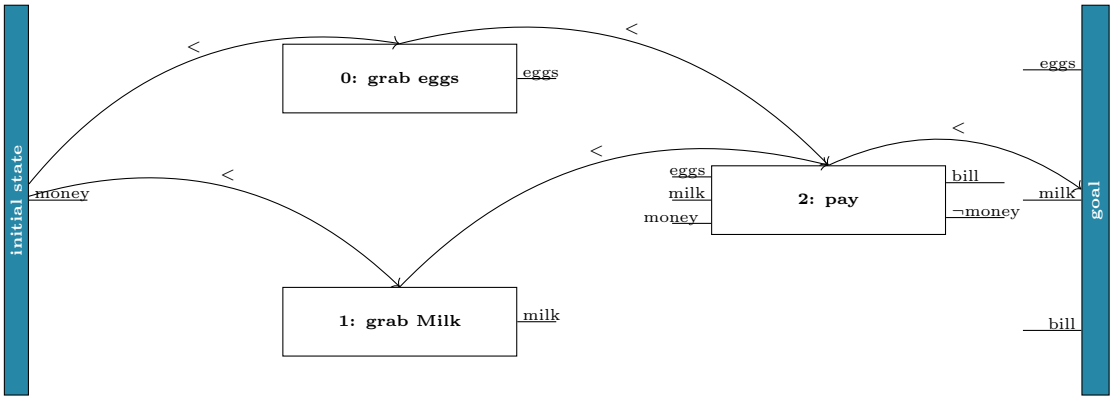


Figure 2.2: Partial-order plan for the problem of Figure 2.1.
In the graphic representation an arrow depicts an ordering constraint e.g. an arrow from plan step 2 to plan step 4 means $2 \prec 4$.

2.3 POCL Planning

Partial-order-causal-link plans (POCL plans) extend PO plans by a set of so-called causal links CL [MR91]. So a POCL plan Π is defined by the tuple $(D, \prec, CL)$ with D being a planning domain, $\prec$ a set of ordering constraints and CL a set of causal links. The purpose of a causal links is to specify that certain plan steps establishing state variables for other plan steps. A causal link is a 3-tuple (ps_1, v, ps_2) consisting of two plan steps ps_1, ps_2 and a state variable $v \in p(action(ps_2)), v \in e^+(action(ps_1))$. ps_1 is called the *producer*, and ps_2 is called the *consumer* of the literal v. If a POCL plan contains a causal link (ps_1, v, ps_2), it means ps_1 establishes v for ps_2. If ps_1 gets performed, the causal link protects the literal v from being deleted before the execution of ps_2. Every causal link implies an ordering constraint: $ps_1 \prec ps_2$. While In a PO plan every plan step possibly can establish a variable for another plan step, in POCL plans every established variable has one fixed producer and consumer.

In POCL plans every plan starts with the action *init* which generates the state s_{init}. $e^+(init)$ contains all state variables of s_{init}. *init* has neither delete effects nor preconditions. A plan reaches a goal g if an action *goal* can be executed, such an action has neither add nor delete effects. $p(goal)$ contains all state variables of the goal description g. A PO plan has achieved a goal state if *goal* is executable.

If there is a plan step ps_3 with $v \in e^-(ps_3)$ that may be executed after ps_1 and before ps_2 the related POCL plan has a causal threat. This is the case if the plan contains neither the constraint $ps_3 \prec ps_2$ nor $ps_1 \prec ps_3$. In order to fix such a threat, ps_3 has to be either promoted or demoted. Meaning ps_3 needs either be ordered before ps_1 ($ps_3 \prec ps_1$) or after ps_3 ($ps_2 \prec ps_1$).

Definition 6. Correctness of POCL plan [MR91] *A 3-tuple* $\Pi = (D, \prec, CL)$ *is a POCL plan for a planning problem if*

- *For every plan step* $ps \in \Pi$ *every of its preconditions* $v \in action(ps)$ *is protected by exactly one causal link* $\exists! cl \in CL, cl = (ps', v, ps)$ *with* $ps' \neq ps$
- *The plan contains a plan step with the action goal* $\exists ps_g \in Pi$ *with* $action(ps_g) = goal$
- *The plan has no causal flaw* $\forall ps \in \Pi, \forall cl = (ps_1, v, ps_2) \in CL$ *with* $v \in e^-(action(ps) \Rightarrow ps \prec ps_2 \vee ps_1 \prec ps$

An example for such a plan is visualized in Figure 2.3. solving the planning problem of Figure 2.1.

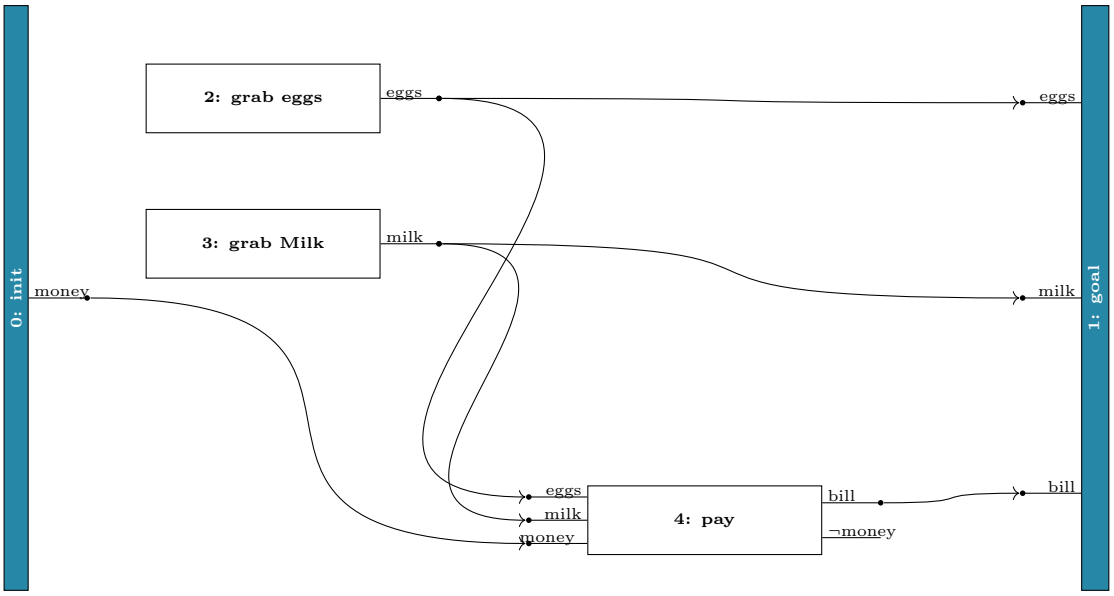


Figure 2.3: The PO plan of Figure 2.2. extended with causal links
every arrow depicts a causal link with the producer on the emitting and consumer at the receiving end

2.4 Makespan

The Makespan is a metric for PO and POCL plans. It describes the length of the longest path from the initial state to the first goal state. A path in a PO/POCL plan is defined inside of the directed graph $G = \langle PS, \prec \rangle$, as a sequence of plan steps $(ps_1, ps_2, ..., ps_n)$ with $ps_i \prec ps_{i+1}$. The length of a path p equals the makespan of a plan Π if for every path in p' in Π with $|p'| < |p|$. In a TO plan the makespan is equal to the length of its

sequence of actions. In PO and POCL plans parallel execution of plan steps is allowed that is why there can be multiple paths from the initial state to a goal state. If there are a lot of parallel executable plan steps in a PO or POCL plan the makespan will be much smaller than the amount of overall plan steps. In general a small makespan is preferred because parallel executable plan step have the potential to be faster performed than strictly ordered plan steps. The makespan is independent of causal links that is why there is an algorithm to calculate the makespan of both PO and POCL plans. Algorithm 1 from [Bäc98] is capable to calculate the makespan from a given PO or POCL plan.

input : Π a PO or POCL plan
output: The makespan m of Π
Construct directed graph $G = \langle PS, \prec \rangle$ with a goal node g;
foreach $ps \in PS$ **do**
    $m(ps) \leftarrow 0$;
    while $PS \neq \emptyset$ **do**
        Select some node $ps \in PS$ without predecessors in PS;
        foreach $ps' \in PS$ *s.t.* $ps \prec ps'$ **do**
            $m(ps') \leftarrow max(m(ps'), m(ps) + 1)$;
            $PS \leftarrow PS \setminus \{ps\}$
        end
    end
end
return $m(g)$;

Algorithm 1: Calculate the makespan of a given PO or POCL plan

The algorithm works up every node starting with the first and ending with the last node to be executed. For every inspected node all of its successor gets assigned with their makespan value, which is either already assigned, or is the makespan of the inspected node plus one. The makespan of the goal node is the makespan of the plan.

3 Justification Algorithms

The basic idea behind Justification is to define a criterion for a plan step that describes its necessity for a plan. Different kinds of justification have been formalized and applied in algorithms by Fink and Yang [FY92]. These algorithms deployed on PO and POCL plans can find unjustified plan steps that can be removed in order to create smaller, less complex plans. Smaller plans are in general more useful since they contain less "useless" plan steps and are therefore easier and faster to execute, so the goal is to remove as many plan steps as possible. For some plans the amount of unjustified plan steps can vary on the justification type. The algorithms can be sorted by their justification strength. A stronger algorithm can remove all plan steps that can be removed by a weaker algorithm and possibly more. However, the runtime of the related algorithm also increases with the justification strength. There is a tradeoff between the amount of removed plan steps and the computation time.

kind of justification	**calculation time**	stronger justification
greedy justification	$\mathcal{O}(P \cdot \lvert\Pi\rvert^5])$	$\uparrow$
well justification	$\mathcal{O}(P \cdot \lvert\Pi\rvert^4])$	$\downarrow$
backward justification	$\mathcal{O}(E \cdot \lvert\Pi\rvert^2])$	weaker justification

Figure 3.1: Three different types of justification and the runtime of its related algorithms on a PO plan Π, with $|\Pi|$ being the number of plan steps, E the number of effects $E = \sum_{ps \in \Pi} |e^+(action(ps))|$ and P the number of preconditions $P = \sum_{ps \in \Pi} |p(action(ps))|$. Table taken from [FY92].

For each of the three justification types in figure 3.1 there is an algorithm for PO and POCL plans. The characteristics of these algorithms will be discussed in the following sections.

3.1 Backward Justification

Definition 7. Backward Justification [FY92] *Let Π be a PO plan that achieves the goal g. A plan step $ps \in \Pi$ is called backward justified if $\exists v \in e^+(action(ps))$ such that ps possibly establishes v either for the goal g or for another backward justified plan step. We say that ps possibly establishes a variable v for a plan step ps' in a partially ordered plan Π if ps establishes v for ps' in at least one linearization of Π.*

```
   Data: let Π̄ be some linearization of Π
 1 for ps := (last plan step of Π̄) to (first plan step of Π̄ ) do
 2 |   Justified=False;
 3 |   foreach v ∈ e⁺(action(ps)) do
 4 |   |   if ∃ps′ ∈ Π s.t. ps establishes v for ps′ or ps establishes v for g then
   |   |   |   /* ps is backward justified                                    */
 5 |   |   |   Justified=True;
 6 |   |   end
 7 |   end
 8 |   if Justified=False then
   |   |   /* ps is not backward justified                                   */
 9 |   |   remove ps from the plan Π;
10 |   end
11 end
```

Algorithm 2: [FY92] a backward justified subplan of a given plan

Algorithm 2 iterates through every variable in the add effect of each plan step and checks if these variables are established for other plan steps. If there is a plan step that is not part of any establishment, the plan step is not justified in the given linearization of the plan. Therefore, the plan step is also not justified in the related PO plan and can be removed.

As mentioned in section 2.2 a causal link (ps_1, l, ps_2) means that ps_1 establishes l for ps_2. So every plan step in a POCL plan which is a producer of a causal link, does establish a variable for another plan step or goal in every linearization of the plan. So every plan step that is producer of a causal link is a backward justified plan step. This the reason why backward justification is easy to establish with POCL plans. On the flip side most planners already create backward justified POCL plans, therefore the related algorithm will most likely not optimize the plan. Above reasons show that the optimization of PO plans through backward justification is more effective than the optimization of POCL plans.

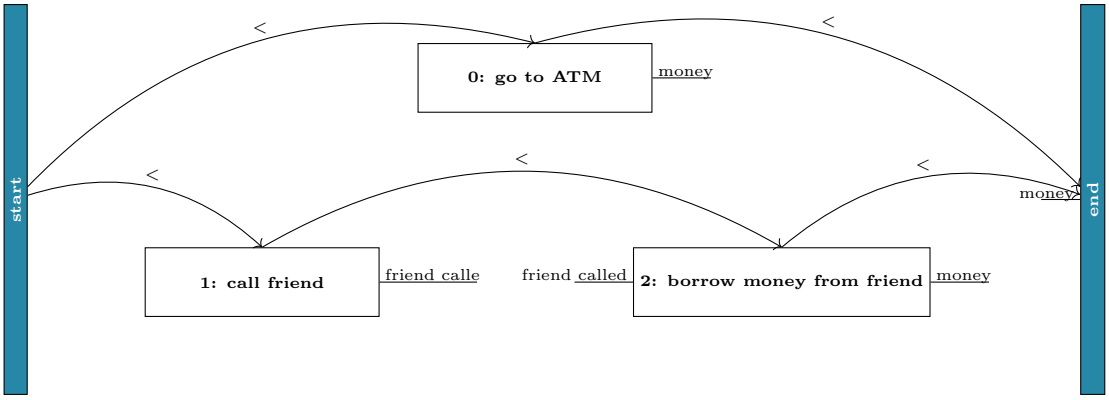


Figure 3.2: Example: PO plan with two alternate linearizations

In PO plan there are no causal links that define the establishment of a variable. In two different linearizations of a PO plan one variable can be established by two different plan steps as can be seen in the example of Figure 3.2. Plan step 1 establishes 'called friend' for plan step 2, and the plan steps 0 and 2 establish 'money' for the goal. The PO plan in Figure 3.2 allows for three different linearizations:

1. $0 \rightarrow 1 \rightarrow 2$
2. $1 \rightarrow 2 \rightarrow 0$
3. $1 \rightarrow 0 \rightarrow 2$

In the first linearization plan step 0 is not backward justified, because plan step 2 establishes also the variable "money" but later. In the second linearization plan step 1 and 2 are both not backward justified. The algorithm first removes plan step 2 because plan step 0 is executed later. Then plan step 1 is removed because plan step 2 is deleted, so there is no precondition that needs the establishment of 'friend called'. Notice the optimization of plan step 1 is only possible because the algorithm removes the plan steps from last to first. The third linearization would be optimized like the first one because plan step 0 is executed before plan step 2. The example also showcases that backward justification can depend on the observed linearization. The algorithm deals with this problem by choosing a random linearization of the PO plan. The random decision of the linearization results in some non-deterministic justifications. A strategy to get an optimal and deterministic result would be to observe every possible linearization and choose the optimization with the least remaining plan steps. If there are plan steps that have no add effect, then they are always going to be removed, because there is no linearization where those plan step could possibly establish a variable. However, plan steps that have variables in their add effects that are already true in the initial state cannot be removed by the backward justification algorithm.

Regarding the complexity of backward justification, for every variable v which is part of an add effect the algorithm needs to check whether it is part of an establishment or

not which equals the complexity $\mathcal{O}(E = \sum_{ps \in \Pi} |e^+(action(ps))|)$. Then the algorithm needs to check for every plan step that is ahead, whether v can be established for this plan step or not with the complexity $\mathcal{O}(|\Pi|)$. Then for every possible establishment the algorithm needs to check, if there can be a plan step with v as a delete effect which also has the complexity of $\mathcal{O}(|\Pi|)$. Therefore, the overall running time of the algorithm is $\mathcal{O}(E \cdot |\Pi|^2])$.

3.2 Well Justification

Definition 8. Well Justification [FY92] *A plan step ps_i in a total-order plan $\overline{\Pi}$ is called well-justified if $\exists v \in e^+(action(ps))$ such that ps_i establishes v for some plan steps or for the goal g, and v does not hold before ps_i, that is $v \notin s_{i-1}$. A plan step in a partially ordered plan is called well-justified if it is well-justified in at least one linearization of the plan.*

A difference between well justification and backward justification is that a variable v that is already true in a state s can no longer be established by other plan steps. If the variable v is true either in the initial state or through the add effect of ps_1 and there are two plan steps ps_2 ps_3 with $ps_1 \prec ps_2 \prec ps_3$ and ps_2 establishes v for ps_3. ps_2 would be backward justified but not well justified, given that there is no plan step ps_4 with $v \in e^-(action(ps_3))$ that can be executed before ps_2. The backward justifications criteria are included in the definition of well justification alongside the above explained criterion. Therefore, every well justified plan step is also backward justified, but not every backward justified plan step is well justified. Because every plan step is only well justified if it establishes some variable that was not established before (or is true in the initial state) the removal of a well justified plan step will lead to the incorrectness of the plan. Therefore, the following lemma holds.

Lemma 1. [FY92] *A plan step is well-justified if and only if we cannot remove it from the plan without violating correctness of the plan.*

The next theorem follows directly from the lemma.

Theorem 1. [FY92] *A plan is well-justified if and only if there is no plan step that can be removed without violating correctness of the plan.*

In order to find a well-justified subplan of a PO plan, algorithm 2 needs to be applied.

```
   Data: let Π be a PO plan
1  repeat
2  |  foreach ps ∈ Π do
3  |  |  if Π without ps is legal and achieves the goal then
4  |  |  |  remove ps from Π;
5  |  |  end
6  |  end
7  until no plan step is removed durning the last execution of the loop;
```

Algorithm 3: Finding a well justified subplan of a given PO plan

Algorithm 2 tries to remove all plan steps one by one and then checks whether the plan is still legal. If a plan step gets removed, some establishments might be removed with it. Therefore, if a plan step is being removed the algorithm tries again to remove plan steps. This leads to the removal of plan steps that only establish variables for not well justified plan steps. This algorithm uses a similar non-determinism like the backward justification algorithm. Depending on the choice of the plan step in line 2, different subplans will emerge. The choice is, in general, random and an optimal reduction of plan steps for a plan is achieved if every possible order of plan step choices is computed and the plan with the least amount of plan steps is selected. This random ordering also determines if a chain of plan steps can be removed. For example in Figure 3.2. plan step 2 needs a variable that is created by plan step 1, so if plan step 2 is being removed plan step 1 is also being removed. If plan step 1 is attempted to be removed first the plan without it is not legal anymore, because the precondition of plan step 2 is not satisfied anymore, therefore 2 needs to be eliminated first. This kind of ordering problem is already addressed in the backward justification algorithm because the plan steps are being selected in a reversed order. One way to optimize the well justification algorithm would be to select the plan steps with the least plan steps ordered behind them first.

In order to execute algorithm 2, we need to be able to check the condition in line 3. This can be accomplished with algorithm 3.

```
   input  : Π a PO plan
   output: True if Π is correct, False otherwise
 1 foreach ps_1 ∈ Π do
 2 |   foreach v ∈ p(action(ps_1)) do
 3 |   |   established ← False;
 4 |   |   foreach ps_2 ∈ predecessors(ps_1) do
 5 |   |   |   if ∃v ∈ e^+(action(ps_2)) and v ∈ p(action(ps_1)) then
 6 |   |   |   |   established ← true;
 7 |   |   |   |   foreach
   |   |   |   |    ps_3 ∈ possiblePredecessors(ps_1) ∩ possibleSuccessors(ps_2) do
 8 |   |   |   |   |   if v ∈ e^−(ps_3) then
 9 |   |   |   |   |   |   established ← False;
10 |   |   |   |   |   end
11 |   |   |   |   end
12 |   |   |   end
13 |   |   end
14 |   |   if established=False then
15 |   |   |   return False;
16 |   |   end
17 |   end
18 end
19 return True ;
```

Algorithm 4: Checking the correctness of a PO plan

Algorithm 3 checks for every precondition variable of every plan step if there is a predecessor plan step that establishes this variable. Therefore, the algorithm needs to check if there is a predecessors with the variable in its add effects and if there could be a delete effect which destroys the establishment. For this computation Algorithm 3 uses the functions *predecessors*, *successors*, *possiblePredecessors* and *possibleSuccessors*, that each return a set of plan step. They are defined as follows.

- predecessors(ps_1)=$\{ps \in \Pi \setminus ps_1 \mid ps \prec ps_1\}$
- successors(ps_1)=$\{ps \in \Pi \setminus ps_1 \mid ps_1 \prec ps\}$
- possibleSuccessors(ps_1)=$\{ps \in \Pi \setminus ps_1 \mid ps \notin predecessors\}$
- possiblePredecessors(ps_1)=$\{ps \in \Pi \setminus ps_1 \mid ps \notin successors\}$

To compute these functions efficiently an ordering matrix needs be created. An ordering matrix is a squared matrix with the height/length of the amount of plan steps, filled with boolean values. For an ordering matrix *om* the value *om*[1][2] would state whether the ordering constraint $ps_1 \prec ps_2$ would hold. As discussed in section 2.2 ordering constraints are transitive, to avoid redundant computation the ordering matrix can be replaced by its transitive closure. This can be accomplished with the Floyd-Warshall Algorithm [Flo62].

From such a transitive closure the sets *predecessors*, *successors*, *possiblePredeccessors* and *possibleSuccesors* can directly be computed.

The well justification algorithm can also be extended for POCL plans unlike the backward justification algorithm. This is because of the way the algorithm selects the plan steps in line 2. If a plan step that produces a causal link gets removed from the plan, it can still be a legal plan if there is another plan step that can also produce the same variable. So if replacements for all removed causal links are added before checking the correctness of the plan an optimization can be accomplished. Algorithm 6 searches such replacements for causal links that have been produced by a removed plan step.

In order to extend the well justification algorithm to POCL plans the condition in line 3 needs to be computed differently. If we remove a plan step from a POCL plan that produced a causal link the plan is illegal because there would be an open precondition. Therefore, replacements for such causal links need to be found without creating causal threats. This problem is known as Remove and Repair and is addressed in [OB19].

Data: let Π be a POCL plan
1 **repeat**
2 **foreach** $ps \in \Pi$ **do**
3 **if** *replacementFound*(Π,{ps})=(*True*,Π') ***and*** *checkCausalThreats*(Π')=*True* **then**
4 $\Pi \leftarrow \Pi'$
5 **end**
6 **end**
7 **until** *no plan step is removed durning the last execution of the loop*;

Algorithm 5: Finding a well justified subplan of a given POCL plan

Algorithm 5 describes the extended well justification algorithm for POCL plans. The condition in line 3 of algorithm 3 is replaced by two algorithms that are described in algorithm 5 and 6.

input : $\Pi = (PS, \prec, CL)$ a POCL plan, PS' a set of plan steps that should be removed
output: (True,Π with new Causal Links and without PS') if replacements could be found, (False,Π) otherwise

```
1  oldCL ← CL;
2  foreach ps1 ∈ PS' do
3      foreach cl ∈ CL do
4          if cl = (ps1, v1, ps2) then
5              foreach ps3 ∈ possiblePredeccessors(ps2) do
6                  if ∃v1 ∈ p(action(ps3)) then
7                      replacementFound ← True;
8                  end
9                  if replacementFound=True then
10                     remove cl from CL;
11                     add cl' := (ps3, v1, ps2) to CL;
12                 end
13                 else
14                     return (False,Π with oldCL);
15                 end
16             end
17         end
18     end
19 end
20 remove PS' from Π;
21 return (True,Π with CL);
```

Algorithm 6: Finding replacements for causal links: searchReplacements(Π,PS')

Algorithm 6 needs to replace all causal links (ps_1, v, ps_2) where a removed plan step ps_1 was the producer with a causal link that has a new producer ps_3. In order to accomplish this every $possiblePredeccessors(ps_2)$ is checked if they can establish v.

input : $\Pi = (PS, \prec, CL)$ a pocl plan
output: True, if Π has no causal threat, False otherwise

```
1  foreach cl = (ps1, v, ps2) ∈ CL do
2      foreach ps ∈ Π do
3          if v ∈ e⁻(action(ps)) and
             ps ∈ possibleSuccesors(ps1) ∪ possiblePredeccessors(ps2) then
4              return False;
5          end
6      end
7  end
8  return True;
```

Algorithm 7: Checking a POCL plan for causal threats: checkCausalThreats(Π)

Algorithm 7 checks for every causal link (ps_1, v, ps_2) if there is a plan step that can be ordered between ps_1 and ps_2 that has v in its delete effects.

In the well justification algorithms (algorithm 2 and 4) there are $\mathcal{O}(|\Pi|^2)$ attempts at removing plan steps from the plan. To check whether a PO plan is legal a running time of $\mathcal{O}(P \cdot |\Pi|^2)$ ($P = \sum_{ps \in \Pi} |p(action(ps))|$) is needed. For POCL plans the running time of searchReplacements and checkCausalThreats are both $\mathcal{O}(|CL| \cdot |\Pi|)$. So the overall running time of the PO algorithm is $\mathcal{O}(P \cdot |\Pi|^4)$ while the running time of the POCL algorithm is $\mathcal{O}(P \cdot |CL| \cdot |\Pi|^3)$.

3.3 Greedy Justification

To check if a plan step ps is greedily justified in a plan Π algorithm 7 [FY92] needs to be applied. If ps is greedily justified the original plan, Π is returned. If ps is not greedily justified, a reduced plan Π' is returned. In Π' ps and possibly other also not greedily justified plan steps have been removed.

input : Π a PO plan, $ps \in \Pi$ a plan step
output: A possibly reduced PO plan Π
1 remove ps from Π' ;
2 **repeat**
3 | Illegals := findIllegalsPO(Π);
4 | EarliestIllegals := $\{ps' \in \text{Illegals} \mid (\forall ps_1 \in \Pi) : ps_1 \prec ps' \Rightarrow ps_1 \notin \text{Illegals}\}$;
5 | remove all plan steps of the set EarlieastIllegals from Π;
6 **until** Π *does not contain illegal plan steps*;
7 **if** Π *still achieves the goal and is legal* **then**
8 | **return** Π *with removed plan steps*;
| `/* Π is a legal subplan of the initial plan */`
9 **else**
10 | **return** *the initial plan* Π;
| `/* ps in the initial plan is greedily justified */`
11 **end**

Algorithm 8: GreedyJustifyChecking(Π,ps)

A plan step ps in a PO plan Π is considered illegal if a precondition $v \in p(action(ps))$ is not established. This is the case if there is no $ps' \in \Pi$ that establishes v for ps according to Definition 1. Therefore, the set of illegals in PO plans is defined by

$illegals = \{ps \in \Pi \mid \exists v \in p(action(ps)) \forall ps' \in \Pi$: ps' does not establishes v for ps$\}$

The set of illegal plan steps in a PO plan can be found with algorithm 8 which is a modified version of algorithm 3.

```
   input  : Π a PO plan with possibly illegal plan steps
   output: illegals, a set containing all illegal plan steps of Π
 1 illegals ← ∅ ;
 2 foreach ps_1 ∈ Π do
 3 |   foreach v ∈ p(action(ps_1)) do
 4 |   |   established ← False;
 5 |   |   foreach ps_2 ∈ predecessors(ps_1) do
 6 |   |   |   if ∃v ∈ e^+(action(ps_2)) and v ∈ p(action(ps_1)) then
 7 |   |   |   |   established ← true;
 8 |   |   |   |   foreach
   |   |   |   |    ps_3 ∈ possiblePredecessors(ps_1) ∩ possibleSuccessors(ps_2) do
 9 |   |   |   |   |   if v ∈ e^−(ps_3) then
10 |   |   |   |   |   |   established ← False;
11 |   |   |   |   |   end
12 |   |   |   |   end
13 |   |   |   end
14 |   |   end
15 |   |   if established=False then
16 |   |   |   add ps_1 to illegals
17 |   |   end
18 |   end
19 end
20 return illegals ;
21 ö
```

Algorithm 9: Finding illegal plan steps in a PO plan: findIllegalsPO(Π)

Well justification can also be applied to POCL plans. An illegal plan step in a POCL plan is defined by either missing a causal link to protect one of its precondition or by being the consumer of a threatened causal link. Let $ps_1, ps_2 \in \Pi$ be two plan steps of a POCL plan. ps_1 is considered an illegal plan step if:

- $\exists v \in p(action(ps_1))$ and $\neg\exists cl \in CL$ with $cl = (ps_2, v, ps_1)$ or
- $\exists cl \in Cl$ with $cl = (ps_2, v, ps_1)$ and $\exists ps_3 \in possiblePredecessors(ps_1) \cap possibleSuccesors(ps_2)$ with $v \in e^-(action(ps_3))$

In order to find the illegal plan steps of a POCL plan algorithm 9 needs to be applied.

```
   input  : Π a POCL plan with possibly illegal plan steps
   output: illegals a set containing all illegal plan steps of Π
 1 illegals ← ∅ ;
 2 foreach ps_1 ∈ Π do
 3 |   foreach v ∈ p(action(ps_1)) do
 4 |   |   protected ← False ;
 5 |   |   foreach cl ∈ CL do
 6 |   |   |   if cl = (ps_2, v, ps_1) then
 7 |   |   |   |   protected ← True ;
 8 |   |   |   |   foreach
   |   |   |   |    ps_3 ∈ possiblePredecessors(ps_1) ∩ possibleSuccesors(ps_2) do
 9 |   |   |   |   |   if v ∈ e^-(ps_3) then
10 |   |   |   |   |   |   protected ← False ;
11 |   |   |   |   |   end
12 |   |   |   |   end
13 |   |   |   end
14 |   |   end
15 |   |   if protected=False then
16 |   |   |   add ps_1 to illegals;
17 |   |   end
18 |   end
19 end
```

Algorithm 10: Finding illegal plan steps in a POCL plan: findIllegalsPOCL(Π)

With algorithm 9 and the algorithms from section 3.2 (algorithm 4 and 5) GreedyJustifyChecking can be extended to POCL plans through algorithm 10.

input : Π a POCL plan, $ps \in \Pi$ a plan step
output: A possibly reduced PO plan Π
1 remove ps from Π ;
2 $PS' \leftarrow \emptyset$;
`/*` PS' `tracks all removed plan steps */`
3 **repeat**
4 | Illegals := findIllegalsPOCL(Π) ;
5 | EarliestIllegals := $\{\alpha' \in \text{Illegals} \mid (\forall \alpha_1 \in \Pi) : \alpha_1 \prec \alpha' \Rightarrow \alpha_1 \notin \text{Illegals}\}$;
6 | remove all plan steps of the set EarlieastIllegals from Π;
7 | add EarlieastIllegals to PS';
8 **until** Π *does not contain illegal plan steps*;
9 **if** *replacementFound(*Π,PS'*)=(True,*Π′*)* ***and*** *checkCausalThreats(*Π′*)=True* **then**
10 | **return** Π′ *with removed plan steps*;
| `/*` Π′ `is a legal subplan of the initial plan */`
11 **end**
12 **else**
13 | **return** *the initial plan* Π;
| `/*` ps `in the initial plan is greedily justified */`
14 **end**

Algorithm 11: GreedyJustifyChecking(Π,ps) extended to POCL plans

While GreedyJustifyChecking(Π,ps) (algorithm 7 and 9) only examines one plan step, GreedyJustification(Π) (algorithm 11) tries to remove all non greedy justified plan steps from Π.

input : Π a PO or POCL plan
output: Π a PO or POCL plan with possibly reduced plan steps
1 **foreach** $ps \in \Pi$ **do**
2 | Π′ ← GreedyJustifyChecking(Π,ps);
3 | **if** ps *is not greedily justified in* Π **then**
4 | | **return** *GreedyJustification(*Π′*)*
5 | **end**
6 **end**
7 **return** Π

Algorithm 12: GreedyJustification(Π)

Greedy justification creates a stronger justification than well justification, but not a perfect one. A perfect justified plan would be defined by creating a plan with no legal subplan. However, the problem of creating a perfect justified subplan is NP-complete. That is why greedy justification is used, it delivers an almost perfect justification in

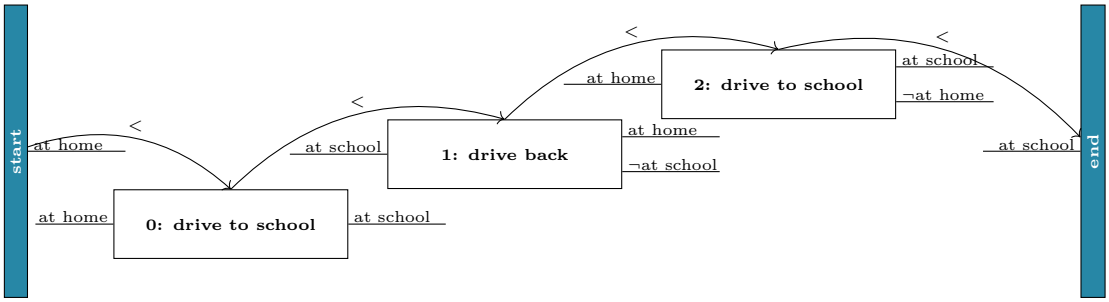


Figure 3.3: In this PO plan all plan steps are well and backward justified. However the greedy justification algorithm would remove either the first or the last two plan steps.

polynomial time. Figure 3.4 shows the big improvement to well and backward justification, the elimination of action cycles that destroy and reestablish variables. The actions *driveSchool* and *driveBack* destroy and reestablish the variable *atschool*. However, they are both well justified because they establish preconditions for other plan steps. They can only be removed using greedy justification. The greedy justification algorithm also includes some non-deterministic decisions. Whether the first or the last two plan steps are going to be removed depends on line 2 of GreedyJustification(Π). The iteration of "drive to school" which is chosen first is going to be removed. As already mentioned for well justification, to get the best optimization every possible order of choices needs to be computed and the solution with the least plan steps has to be selected.

The algorithm GreedyJustifyChecking for PO plans requires $\mathcal{O}(P \cdot |\Pi|^3)$ time. For POCL plans GreedyJustifyChecking needs $\mathcal{O}(P \cdot |\Pi| \cdot |CL|^2)$ time. The algorithm GreedyJustification calls GreedyJustifyChecking at most $|\Pi|^2$ times. So the overall complexity of the PO plan algorithm is $\mathcal{O}(P \cdot |\Pi|^5)$, while the complexity for the POCL plan algorithm is $\mathcal{O}(P \cdot |\Pi|^3 \cdot |CL|^2)$.

4 Evaluation Justification Algorithms

The theoretical foundation for justification algorithms has already been established in [FY92], therefore the purpose of this work is to test the justification algorithms in order to provide a proof of concept. To achieve this an implementation for the algorithms that were discussed in Chapter 3 has been tested with planning domains from the IPCs (International Planning Competitions). With the PANDA [BKB14] planning system which has been developed by the Institute for Artificial Intelligence at Ulm University, solutions to these domains have been computed. The computations of the justification algorithms took place on a Lenovo L480 Thinkpad with an Intel Core i5-8250U CPU at 1.6 GHz. The solutions are given in the form of POCL plans which can be transformed to PO plans by simply removing the causal links.

Well and greedy justification have been tested with both POCL plans and PO plans. Therefore, every plan has been tested with 4 different justification algorithms. The tests on backward justification have been unsuccessful because the related algorithm did not produce optimization on any of the tested plans, which is caused by the fact that the test plans have been created by a POCL planning system and are therefore backward justified by definition.

To evaluate an optimization five different metrics have been used.

- Plan step-optimized plans: The amount of plans where plan steps were removed
- Makespan-optimized plans: The amount of plans with reduced makespan
- Optimized plan steps: The total amount of reduced plan steps
- Optimized makespan: The total amount of reduced makespan
- Optimization ratio: $(\frac{|\text{optimized plans}|}{|\text{Tested plans}|})$
- Plan step ratio: $(\frac{|\text{output plan steps}|}{|\text{input plan steps}|})$
- Average plan step optimization: $(\frac{\sum_{\Pi \in \text{plan step optimized plans}} \frac{|\text{output plan steps of } \Pi|}{|\text{input plan steps of } \Pi|}}{|\text{plan step optimized plans}|})$
- Makespan ratio: $(\frac{|\text{output makepspan}|}{|\text{input makespan}|})$
- Average makespan optimization: $(\frac{\sum_{\Pi \in \text{makespan optimized plans}} \frac{|\text{output makespan of } \Pi|}{|\text{input makespan of } \Pi|}}{|\text{makespan optimized plans}|})$
- Computation time: The time it took to compute the optimizations

Algorithm	Well justification on PO	Well justification on POCL
Makespan optimized plans	248736	81327
Plan Step optimized plans	270300	100899
Optimized plan steps	783381	386845
Optimized makespan	536471	145243
Optimization ratio	0.586	0.219
Plan step ratio	0.906	0.948
Average plan step optimization	0.851	0.856
Makespan ratio	0.913	0.969
Average makespan optimization	0.842	0.861
Computation time in min	11.674	6.351

Table 4.1: Test Result for well justification with 1 random seed

Algorithm	Greedy justification on PO	Greedy justification on POCL
Makespan optimized plans	432708	431834
Plan Step optimized plans	453117	452423
Optimized plan steps	1791062	1893068
Optimized makespan	1444539	1545904
Optimization ratio	0.982	0.981
Plan step ratio	0.786	0.774
Average plan step optimization	0.773	0.757
Makespan ratio	0.765	0.749
Average makespan optimization	0.754	0.737
Computation time in min	20.083	18.791

Table 4.2: Test Result for greedy justification with 1 random seed

In the test set there are 461 363 different plans that contain an overall amount of 8 358 208 plan steps. The sum over the makespan of all plans is 6 152 343.

As discussed in Chapter 3, all of the justification algorithms have a non-deterministic element. The ordering in which the plan steps are being checked for their justification is non-deterministic and can have an impact on the outcome of the optimization. In order to take this problem into account, every algorithm will be executed with 10 different random seeds. Additionally, the same test set will be provided with only one random seed therefore the non-deterministic impact can be evaluated. For the tests with 10 random seeds 10 possibly different optimizations can arise. The results in table 4.3 and 4.4 were computed by choosing one of the 10 optimizations with the highest amount of optimized plan steps.

Tables 4.1 and 4.2 show that the greedy justification algorithms generate a better overall optimization which is excepted since greedy justification is a stronger justification type. Both types of the greedy justification algorithms are optimizing over 10% more plan

Algorithm	Well justification on PO	Well justification on POCL
Makespan optimized plans	248737	81327
Plan Step optimized plans	270300	100899
Optimized plan steps	790174	386845
Optimized makespan	543264	145243
Optimization ratio	0.586	0.219
Plan step ratio	0.905	0.948
Average plan step optimization	0.849	0.856
Makespan ratio	0.912	0.969
Average makespan optimization	0.839	0.862
Computation time in min	108.738	66.767

Table 4.3: Test result for well justification with 10 random seeds

steps than their well justification counterparts. Even more significant is the differnce in the optimization ratio. While the greedy justification algorithms find an optimization for almost every plan, the well justification algorithms show much smaller optimization ratios with 0.586 for PO plans and 0.219 for POCL plans. Also notable are the differences between the optimization of PO plans and POCL plans. For well justification algorithms every optimization metric except the computation time is much better for PO plans. Especially the amount of optimized plans and as a result of this also the optimization ratio. For PO plans the amount of optimized plans are almost doubled in comparison those of the POCL plans. The PO algorithm has a higher computation time since every removal of a plan step can cause the loop inside of the well justification algorithms to restart. Therefore, a lower plan step ratio does result in a higher computation time. For greedy justification the difference in optimization metrics for PO and POCL plans is much lower. In fact on regarding table 4.2 greedy POCL algorithm preforms better at optimizing makespan than its PO counterpart. This effect is due to the random seed that has been used. With another random seed the PO algorithm might preform better as can be seen in Table 4.4.

In general better results for PO plans are excepted since the lack of causal links make PO plans easier modifiable and more optimizations are possible. There are a lot of plans where an optimization can be done on the PO plan but not on the POCL plan. But there are also some plans that result in a better optimization for POCL plans than for PO plans. This can happen if there is a plan with multiple possible optimizations with some removing more and some removing less plan steps. If the optimization that is removing less plan steps can not be chosen by the POCL algorithm, because of its added restrictions, the weaker optimization will only be possible in the PO algorithm. Therefore, such plans work better with the POCL algorithm.

In the tested set of plans there are not many plans with different optimization options. Therefore, the non-deterministic element of the algorithms did not matter for most plans. Nevertheless, slight improvements can be seen by comparing table 4.1 and 4.2

Algorithm	Greedy justification on PO	Greedy justification on POCL
Makespan optimized plans	432777	431834
Plan Step optimized plans	453117	452423
Optimized plan steps	1939801	1933375
Optimized makespan	1593006	1586213
Optimization ratio	0.982	0.981
Plan step ratio	0.768	0.769
Average plan step optimization	0.751	0.757
Makespan ratio	0.741	0.742
Average makespan optimization	0.730	0.737
Computation time in min	213.512	183.604

Table 4.4: Test result for greedy justification with 10 random seeds

with table 4.3 and 4.4. The only algorithm where the usage of multiple different seeds had no impact is the well justification algorithm on POCL plans. The obvious downside of using 10 random seeds is the increase in computation time by an approximate factor of 10.

In order to provide a perspective on the size of the plans (amount of plan steps), the justification algorithms have been tested on a second set of plans. The second set is designed to provide a greater diversity of plan sizes. On the first set most of the plans had an approximate amount of 20-30 plan steps. In the second set the plans have a range of 5 to 70 plan steps. The algorithms will use 1 random seed per plan. Table 4.5 to 4.8 display the test results for the second set while being sorted by length of the plans. To provide a more compact view the plans with similar plan length (e.g. size >10,size >20, size >30) have been put together.

In table 4.5 to 4.8 the biggest differences in optimization performances can be observed in the smaller plans (size 0 to 20). The greedy algorithms perform the best on smaller plans and get worse the bigger the plans become. The plan and makespan ratios of well justification algorithms do not change a lot with the plan size. Therefore, the bigger the plans the smaller the difference in optimization metrics between the algorithms. Although, the lack of plans with a size >60 is causing some inconsistencies on the optimization ratios. The difference between the PO and POCL algorithms in the second test set is similar to those from the first test. The PO algorithm perform, slightly better than its POCL counterpart and scale similarly with their Plan size.

Plan size	>0	>10	>20	>30	>40	>50	>60
Plans	1908	1563	1812	830	1202	123	28
Plan step optimized plans	298	270	1786	822	1202	123	4
Makespan optimized plans	298	270	1249	656	1100	120	3
Optimized plan steps	495	508	4714	2348	2856	444	4
Optimized makespan	495	508	1848	1027	1561	233	3
Optimized plan ratio	0.844	0.827	0.014	0.010	0.000	0.000	0.857
Plan step ratio	0.967	0.976	0.895	0.917	0.947	0.928	0.998
Average plan step optimization	0.787	0.864	0.894	0.915	0.947	0.928	0.986
Makespan ratio	0.961	0.971	0.909	0.914	0.929	0.900	0.993
Average makespan optimization	0.756	0.853	0.874	0.893	0.924	0.898	0.936
Computation time in min	0.001	0.025	0.158	0.362	1.787	0.192	0.069

Table 4.5: Results sorted by plan size for well justification on PO plans

Plan size	>0	>10	>20	>30	>40	>50	>60
Plans	1908	1563	1812	830	1202	123	28
Plan step optimized plans	118	0	1786	822	1202	123	4
Makespan optimized plans	118	0	1249	656	1100	120	3
Optimized plan steps	135	0	4714	2348	2709	434	4
Optimized makespan	135	0	1848	1027	1477	233	3
Optimized plan ratio	0.938	1.000	0.014	0.010	0.000	0.000	0.857
Plan step ratio	0.991	1.000	0.895	0.917	0.950	0.929	0.998
Average plan step optimization	0.844	1.000	0.894	0.915	0.950	0.929	0.986
Makespan ratio	0.989	1.000	0.909	0.914	0.933	0.900	0.993
Average makespan optimization	0.819	1.000	0.874	0.893	0.927	0.898	0.936
Computation time in min	0.002	0.019	0.087	0.176	0.705	0.073	0.033

Table 4.6: Results sorted by plan size for well justification on POCL plans

Plan size	>0	>10	>20	>30	>40	>50	>60
Plans	1908	1563	1812	830	1202	123	28
Plan step optimized plans	1880	1258	1807	822	1202	123	4
Makespan optimized plans	1876	1182	1278	659	1100	120	3
Optimized plan steps	6993	4880	4872	2358	2884	444	4
Optimized makespan	6731	4575	1954	1037	1593	233	3
Optimized plan ratio	0.015	0.195	0.003	0.010	0.000	0.000	0.857
Plan step ratio	0.528	0.773	0.891	0.917	0.946	0.928	0.998
Average plan step optimization	0.524	0.718	0.891	0.914	0.947	0.928	0.986
Makespan ratio	0.468	0.739	0.903	0.913	0.928	0.900	0.993
Average makespan optimization	0.464	0.682	0.871	0.893	0.922	0.898	0.936
Computation time in min	0.002	0.039	0.374	0.708	3.787	0.503	0.322

Table 4.7: Results sorted by plan size for greedy justification on PO plans

Plan size	>0	>10	>20	>30	>40	>50	>60
Plans	1908	1563	1812	830	1202	123	28
Plan step optimized plans	1880	1258	1807	822	1202	123	4
Makespan optimized plans	1876	1182	1278	659	1100	120	3
Optimized plan steps	6993	4952	4872	2358	2741	434	4
Optimized makespan	6731	4647	1954	1037	1509	233	3
Optimized plan ratio	0.015	0.195	0.003	0.010	0.000	0.000	0.857
Plan step ratio	0.528	0.770	0.891	0.917	0.949	0.929	0.998
Average plan step optimization	0.524	0.714	0.891	0.914	0.949	0.929	0.986
Makespan ratio	0.468	0.735	0.903	0.913	0.931	0.900	0.993
Average makespan optimization	0.464	0.678	0.871	0.893	0.926	0.898	0.936
Computation time in min	0.006	0.037	0.355	0.570	3.114	0.518	0.338

Table 4.8: Results sorted by plan size for greedy justification on POCL plans

5 Makespan Optimization

There are more ways to optimize the makespan of a plan than just the removal of plan steps as it is done by the justification algorithms. In this chapter the goal is to minimize the makespan of a plan without changing any the set of plan steps. If no plan steps may be removed, the ordering of the plan steps has to be changed.

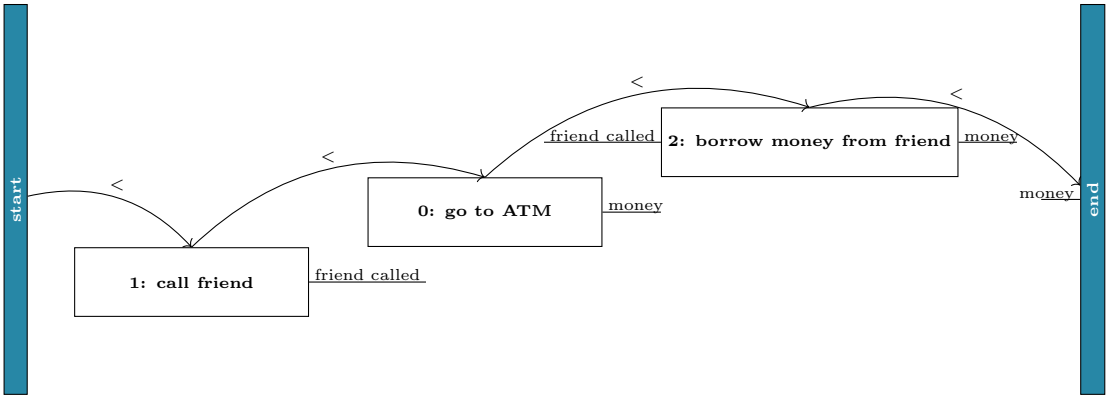


Figure 5.1: Example: PO-plan with non-optimal ordering and a makespan of length 3

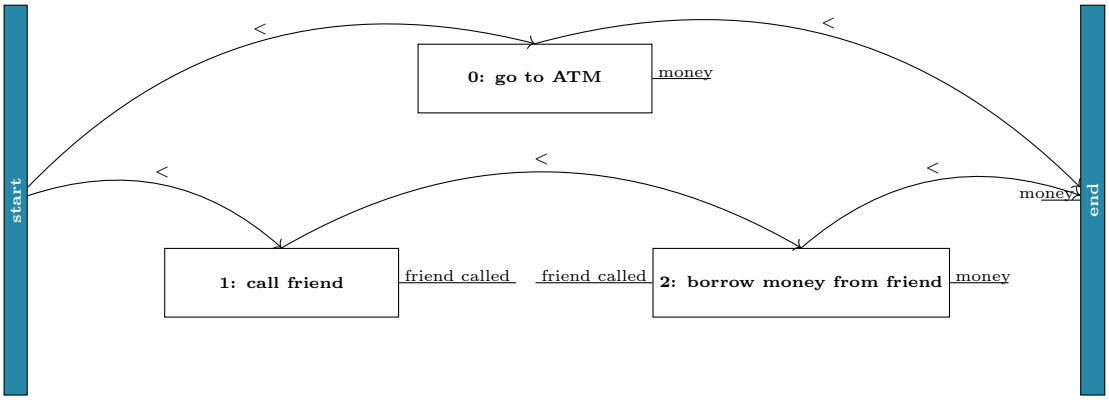


Figure 5.2: Example: PO-plan with optimal ordering and a makespan of length 2

Figure 5.1 and 5.2 show how the ordering of plan steps can impact the makespan of a plan by using an example from Chapter 3.

For this approach we are using the hybrid planning system PANDA [BKB14]. Hybrid Planning is a mixture of Hierarchical Task Network planning [EHN94] and POCL planning [MR91]. Since, the goal is to find new ordering constraints for the given plan steps. We define a hybrid planning problem that is solved by a PO plan with possibly new ordering constraints to the given plan steps.

A hybrid planning domain is defined by the tuple (T_a,T_p,M). T_a and T_p are sets of primitive and abstract tasks and M is a set of decomposition methods. A primitive task is an unlabeled plan step i.e. an action. An abstract task can be decomposed into a various amount of primitive and abstract tasks using decomposition methods. Using abstract tasks can cause a variation in the resulting plan steps. Therefore, to neither remove nor add additional plan steps, all given plan steps will be primitive tasks in our hybrid planning domain.

A hybrid planning problem is defined by a hybrid planning domain and an initial Plan P_{init}. It is solved by inserting ordering constraints and decomposition of abstract tasks, resulting in a correct PO plan P_{sol}.

To find an optimal ordering of plan step, we need to discard all possibly suboptimal ordering constraints in order to find better constraints. Therefore, P_{init} will not contain any ordering constraints.

In conclusion the hybrid planning problem for reordering the PO plan $\Pi = (PS, \prec)$ has to satisfy the following criteria.

- $T_a = \emptyset$
- $M = \emptyset$
- $T_p = PS$
- $P_{init} = (PS, \prec)$ with $\prec = \emptyset$ and $PS = \{init, goal\}$

Since, there are no abstract tasks PANDA solves this Hybrid planning problem with just the insertion of ordering constraints. PANDA is also able to run on an A-star heuristic that minimizes the makespan. In this configuration PANDA will create ordering constraints that will have the minimal possible makespan for the given plan steps. The new plan will have the same plan steps as the old plan while also having makespan optimal ordering constraints.

6 Evaluation Makespan Optimization

In this chapter the effectiveness of the optimization strategy from Chapter 5 will be evaluated. The metrics that have been used are similar to the metrics in Chapter 3. Plan step related metrics will not be featured because the amount of plan steps is not affected by the optimizations of Chapter 5. The set of tested plans were selected similarly to the second test set of Chapter 3 which vary in size from <10 up to 50 plan steps. In total 7429 plans have been tested. The added makespan of all plans is 79530. Table 6.1 provides the compact results for all plans, while in table 6.2 the tested plans are sorted by their size.

Plans	7429
Optimized Plans	2198
Optimization Ratio	0.295
Optimized Makespan	3454
Makespan Ratio	0.957
average optimization	0.888
Computation Time in min	922.110

Table 6.1: Optimization results for all 7429 plans

In comparison, the makespan ratio is worse than all makespan ratios from the justification algorithms. Also, the optimization ratio is relatively low but is able to beat the results of the well justification algorithm on POCL plans. The average makespan optimization is on the same level as those of the justification algorithm optimizations. Therefore, a low percentage of plans have been optimized, but the optimizations that have been made are significant. The computation time is really high, in comparison the slowest justification algorithm (greedy justification on POCL plans) required for a similar amount of plans only 5 minutes.

Regarding table 6.2 the most significant change throughout the size of the tested plans is the optimized plan ratio. The bigger the plans get the more likely an optimization can be found. Although the average makespan optimization seems to get worse for plans with a size >10, in total more makespan is getting optimized on the bigger plans. Those differences are most significant for plans with a size <10 plan steps. Only 2.9% of those plans were optimized with an average makespan optimization of 74.8%. 74.8% seems like a really high average optimization, but we have to consider on a plan size <10 the removal of only one plan step will yield a makespan optimization of at least

Plan size	>0	>10	>20	>30	>40
Plans	1921	1547	2054	911	996
Optimized problems	56	215	1010	431	507
Optimized plan ratio	0.029	0.139	0.492	0.473	0.509
Optimized makespan	72	231	1353	740	1126
Average makespan optimization	0.748	0.887	0.883	0.892	0.875
Makespan ratio	0.993	0.985	0.937	0.944	0.937
Computation time in min	85.855	78.549	156.861	68.005	473.858

Table 6.2: Optimization results sorted by the plan size

90%. For plans that have more than 20 plan steps the results have rather insignificant variations.

Overall justification algorithms provide better makespan optimization in less computation time. On top of that, justification algorithms have the ability to remove plan steps. Therefore, justification algorithms are more effective for optimizing plans. Although it has to be considered that both methods create different kinds of optimization which means they can be used consecutively on the same plans.

7 Conclusion

In this work the justification algorithms which were featured in [FY92] have been deployed on PO and POCL plans, this led to two versions of the well and greedy justification algorithms, a PO and a POCL version. These have been tested on a set of plans with more than 400 000 plans. The tests were able to prove that using a stronger justification type will yield better optimization results, while also using a significantly higher computation time. This confirms the justification hierarchy that has been described in [FY92]. Another important observation is the superiority of PO optimization over POCL optimization due to increased ability to change plans. The PO algorithms also had higher computation times, which results in another tradeoff between computation time and optimization effectiveness. The combination of these factors leads to the following hierarchy between the four algorithms.

Greedy justification on PO plans	better optimization
Greedy justification on POCL plans	↑
Well justification on PO plans	↓
Well justification on POCL plans	lower computation time

Additionally, a way to reduce the makespan of a plan by rearranging its ordering constraints has been tested. For this the initial ordering constraints were removed and the resulting incomplete plan would be transformed in a hybrid planning problem, which would be solved with the hybrid planning system PANDA [BKB14] by inserting ordering constraints that result in an optimal makespan. This method proved to have much higher computation time than justification algorithms. Also all the justification algorithms were better in optimizing the makespan of the tested plans. Therefore, the rearrangement of ordering constraints can be used as a way to further optimize plans with already optimal plan steps. For general optimization purposes the justification algorithms with their ability to remove plan steps proved to be more effective.

Bibliography

[Bäc98] Christer Bäckström. Computational aspects of reordering plans. *Journal of Artificial Intelligence Research*, 9:99–137, 1998.

[BBG+14] Pascal Bercher, Susanne Biundo, Thomas Geier, Thilo Hoernle, Florian Nothdurft, Felix Richter, and Bernd Schattenberg. Plan, repair, execute, explain – how planning helps to assemble your home theater. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014*. AAAI, 2014.

[BKB14] Pascal Bercher, Shawn Keen, and Susanne Biundo. Hybrid planning heuristics based on task decomposition graphs. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014*. AAAI Press, 2014.

[DM91] Thomas L. Dean and Kathleen R. McKeown, editors. *Proceedings of the 9th National Conference on Artificial Intelligence, Volume 2*. AAAI Press / The MIT Press, 1991.

[EHN94] Kutluhan Erol, James A. Hendler, and Dana S. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems, University of Chicago, Chicago, Illinois, USA, June 13-15, 1994*, pages 249–254. AAAI, 1994.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[FN71] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[FY92] Eugene Fink and Qiang Yang. Formalizing plan justifications. *Proceedings of the 9th Conference of the Canadian Society for Computational Studies of Intelligence*, pages 9–14, 1992.

[Hel03] Malte Helmert. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262, 2003.

[KN96] Subbarao Kambhampati and Dana S. Nau. On the nature and role of modal truth criteria in planning. *Artificial Intelligence*, 82(1-2):129–155, 1996.

[KTY91] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In Dean and McKeown [DM91], pages 692–697.

[MR91] David A. McAllester and David Rosenblitt. Systematic nonlinear planning. In Dean and McKeown [DM91], pages 634–639.

[OB19] Conny Olz and Pascal Bercher. Eliminating redundant actions in partially ordered plans – A complexity analysis. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, IiCAPS 2019*, pages 310–319. AAAI Press, 2019.

Eidesstattliche Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig angefertigt zu haben. Dabei wurden nur die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Ulm, den September 13, 2019 _____

(Linus Diepold)