UNIVERSITÄT BASEL

Tunnel-Based Pruning for Classical Planning

Bachelor's Thesis

Natural Science Faculty of the University of Basel
Department of Computer Science
Artificial Intelligence
http://ai.cs.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Martin Wehrle, Silvan Sievers

Daniel Federau
daniel.federau@stud.unibas.ch
11-057-197

08.01.2015

Acknowledgments

At this point I would like to thank Martin Wehrle and Silvan Sievers for their help and insight. The weekly meetings helped a lot in understanding the topic. I would also like to thank Prof. Dr. Malte Helmert for allowing me to write this bachelor's thesis in the area of Artificial Intelligence.

Abstract

One huge topic in Artificial Intelligence is the classical planning. It is the process of finding a plan, therefore a sequence of actions that leads from an initial state to a goal state for a specified problem. In problems with a huge amount of states it is very difficult and time consuming to find a plan. There are different *pruning methods* that attempt to lower the amount of time needed to find a plan by trying to reduce the number of states to explore. In this work we take a closer look at two of these pruning methods. Both of these methods rely on the last action that led to the current state. The first one is the so called *tunnel pruning* that is a generalisation of the tunnel macros that are used to solve *Sokoban* problems. The idea is to find actions that *allow* a tunnel and then prune all actions that are not in the tunnel of this action. The second method is the partition-based path pruning. In this method all actions are distributed into different partitions. These partitions then can be used to prune actions that do not belong to the current partition.

The evaluation of these two pruning methods show, that they can reduce the number of explored states for some problem domains, however the difference between pruned search and normal search gets smaller when we use heuristic functions. It also shows that the two pruning rules effect different problem domains.

Table of Contents

1 Introduction

A lot of research is done to improve the performance of planner for classical planning. The goal of a planner is to find a plan for a problem, therefore a sequence of actions that when executed leads from a specific initial state to a goal state. The search for a plan can potentially consume a lot of time, especially when the planner has to explore a huge amount of states. There are different approaches to boost the performance of a planner. One approach is *action pruning*. The goal is to find states that can be pruned without destroying the plan. In this bachelor's thesis we use two different pruning methods and implement them. These two methods were introduced by Nissim et al. [1]. The two pruning methods have in common, that they try to finish a task before starting another. This can lead to a reduction of states because we have a better focus during the search. They also have in common that they use the last action that led to the current state to decide whether we can prune actions or not. An action allows pruning if it fulfils certain conditions. These condition differ for the different pruning methods.

The first method we use is the so called *Tunnel Pruning*. In this method we first have to check every action whether they allow a tunnel or not. An action allows a tunnel, if it satisfies certain conditions. We take a closer look on these conditions in chapter 3.1. If an action allows a tunnel we have to then find all actions that belong to that tunnel. Once we find the tunnel for the action we can use it to prune actions. During the search, if we are in a state s and the action a that led to s allows a tunnel, then we only have to consider actions that are in the tunnel of a for the next step. This can cause that a number of actions are pruned.

The second pruning method is the *partition based path pruning*. This method uses the *partitioned SAS+ problem* described in section 3.2.1 to split all actions into different partitions. This partitions can then be used to prune actions. The difficult part for this method is to find a useful decomposition into different partitions. Section 3.2.3 gives an overview over the process of decomposition.

In order to evaluate these methods, we implemented them in the *Fast Downward planner* [2]. Details for the implementation are described in section 3.1.1 and 3.2.4. The results of our evaluation are shown in chapter 4. We compared our solution with the standard A*-algorithm. In order to do so we used three different configurations. First we executed

a uninformed search with no heuristic information and then we used two different heuristic functions. The results show that the pruning methods can reduce the number of explored states for different problem domains. But the quality of the pruning differs for different domains. Also the two different pruning methods prune in different domains although the idea behind them is very similar.

2
Preliminaries

This chapter gives an overview of the theoretical background that is used in this work. The first section describes the concept of planning and introduces associated terms. Section 2.2 gives an overview over heuristic search, in particular over the informed search algorithm A* [3].

2.1 SAS+ Planning

The solving of *planning problems* is an important branch of artificial intelligence and is used in a huge amount of applications, for example, solving logistic problems, like optimally distributing packages to their destination.
Planning is the process of finding a sequence of actions that leads from a specific initial state to a goal condition. The planner needs a description of the problem in an appropriate *planning language*, e.g. *PDDL* [4]. This planning language is used to map the given problem to a number of variables that can be used by the planner. These variables can be assigned to different values. An assignment that maps a value to each variable is called a *state*. The first state which is used as the starting point for the planner is the *initial state*. Now the planner has to find a solution for the given initial state. It executes *actions* which change the values of the variables until it finds an assignment for the variables that is equal to a specified *goal state*. A sequence of legal actions which lead from the initial state to a goal state is called a *plan*.
Every action has a non-negative amount of costs. The cost of the plan is the sum of all costs of the actions which are applied in the plan.
There are two different kinds of planners: Cost-optimal planners guarantee that the generated plan is optimal, which means the total costs are minimal. On the other side, satisficing planners can return a plan whose costs are suboptimal. In this work we only consider cost-optimal planners.
In this bachelor's thesis we use the SAS+ planning formalism [5]. A planning problem that uses the SAS+ formalism, is defined by the 5-tuple $\Pi = \langle \mathcal{V}, s_0, s_*, \mathcal{A}, cost \rangle$.

- $\mathcal{V} = \{v_1, ..., v_n\}$ is a set of finite-domain variables. Each variable $v \in \mathcal{V}$ can be assigned a value from its finite domain D_v. If a variable v is assigned with a value p from its

finite domain we use (v, p) as notation.

- s_0 is the initial state and s_* is a partial assignment of variables which describes the different goal states. This means not all variables need a value to fulfil the goal.

- $\mathcal{A}$ is a set of actions. Every action $a = \langle pre(a), prevail(a), \mathit{eff}(a)\rangle \in \mathcal{A}$ is defined by its *preconditions*, *prevail conditions* and *effects*. These three are partial assignments of all variables in $\mathcal{V}$. An action is applicable in a state s if the pre- and prevail conditions are satisfied in s. The values of all the other variables do not matter here.

  A prevail condition is a condition that is required, but its action does not change the value of the variable specified in this prevail condition. In contrast, the value of a variable of a precondition will be changed after executing its action. The notation $(v, p) \in pre(a)$ means that the variable v has to have the value p, so that the action a can be executed. This notation can also be used for the effects: $(v, p) \in \mathit{eff}(a)$ means that the variable v is assigned with the value p after executing action a. After the action is applied, the values of the variables are changed according to the effects of the action. The values of all the other variables stay the same.

  For simplification reasons we use the following notation like Nissim et al. [1] do in their work: If an action a has the same variable in its *pre* and in its *eff*, for example if it has $(v, p) \in pre(a)$ and $(v, p') \in \mathit{eff}(a)$, then we say that a has a *pre-post* condition (v, p, p').

- *cost* is a function that maps a non-negative value to every action. These values describe how much it costs to execute an action.

2.2 Heuristic Search

To improve the performance of the planner we can use *heuristic functions*. The idea is to approximate the distance, so the cheapest path cost, between the current state and the nearest goal state. Formally, a heuristic function h is a function that maps a non-negative value to every state.
The closer the heuristic value is to the real distance, the more efficient the search algorithm is that uses this heuristic. But on the other hand the computation time for the heuristic rises. Therefore a good heuristic has to be as close as possible to the real distance while also being fast to compute.
Search algorithms that use heuristic values are called *informed search algorithms*. The search algorithm we use in this work is $A*$ because it delivers optimal plans. A* uses a cost function f that determines which state will be visited next. For a state s, f is defined as $f(s) = g(s) + h(s)$. The first part $g(s)$ describes the cost from the initial state to s. $h(s)$ is the heuristic value for this state s.
$f(s)$ estimates the cost of the whole path from the initial state over the current state s to a nearest goal state. A* uses f to determine which state will be explored next. The state with the smallest value of f is picked.
To guarantee that the plan received from A* is optimal, the heuristic that is used must be *admissible*. That means that the heuristic value has to be lower or equal than the true distance to the goal for every state.

3
Action Pruning

During the process of finding a plan, the planner potentially has to traverse a lot of states. In problems with a large state space this can lead to long computation times. To reduce the number of states to explore and therefore to increase the performance of the planner we can use a method called *action pruning*. The objective is to reduce the state space by finding applicable actions for every state that can be pruned so that the plan is still optimal. The pruning methods we use in this work rely on the last action that led to the current state s. The idea is to prune applicable actions in s according to the last action under certain criteria. These criteria differ for the different pruning methods. The paper "Tunneling and Decomposition-Based State Reduction for Optimal Planning" by R. Nissim, U. Apsel and R. Brafman [1] describes two different pruning methods. The first method is called *Tunnel Pruning*. The goal is to find actions that induce a *tunnel* and prune all actions that are not part of the tunnel. The second one is *partition-based path pruning*. The general idea is to divide all actions into different partitions and prune actions that do not belong to the current partition. In this bachelor's thesis these two pruning methods were implemented in the *Fast Downward* planner [2] and evaluated.
The two pruning methods we use are *optimality preserving*. A pruning method ρ is optimality preserving if the optimal plan π for every SAS+ problem, which is not pruned, is *effect-equivalent* to the optimal plan π' for the same problem pruned by ρ. Effect-equivalent means that the cost of the two plans have to be equal and they have to consist of the same states. But the order of the visited states in the plans can be different. So π has to be a permutation of π'.

3.1 Tunnel Pruning

The first time the idea of tunnel macros for pruning actions was used for a solver for the puzzle game *Sokoban* by Hiroyuki Imabayashi from 1981. The playing field of Sokoban consists of a 2D-Grid of square fields and is surrounded by walls. The player controls a character that can move around the fields and can push boxes from one field to another. The goal is to push boxes from their starting position to a specified goal field. Of course there are obstacles like walls that complicate the solving of the problem. In this context, a

tunnel, as its name implies, is a path with "width" of one which is surrounded by walls and has an entry and an exit. Figure 3.1 shows an example of a Sokoban field with a tunnel in the middle. Once the player pushes a box into the tunnel, the most reasonable action is to push the box to the end of the tunnel before executing any other actions. So the idea is to prune all other actions until the box reaches the exit of the tunnel.

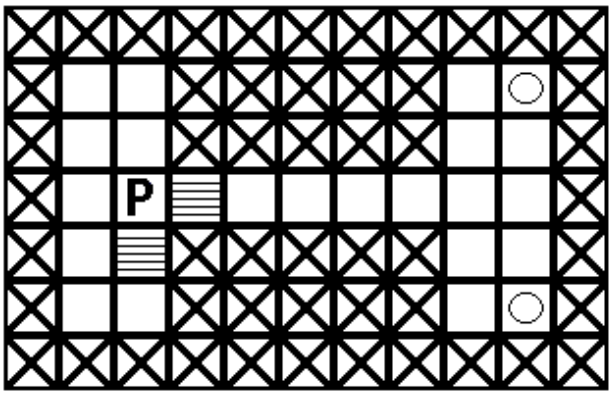

Figure 3.1: Example of a sokoban field with tunnel in the middle.
P = Player, Circle = Goal Field, Striped = Box

In this work we use a generalisation of the idea from Sokoban that can be used for every problem domain. To use tunnel pruning we first have to find an action a that *allows* a tunnel. Once we find this action we have to search for other actions that belong to the tunnel of a. Then we can use this computed $tunnel(a)$ to prune actions.
Nissim et al. identify the following conditions that an action $a = \langle pre(a), prevail(a), \mathit{eff}(a)\rangle$ has to fulfil to allow a tunnel:

- The action a has to write a variable v_i that is needed to fulfil a goal state but the value p of v_i is not the same as the value of v_i in the goal. This means it exists $(v_i, p) \in \mathit{eff}(a)$ such that $(v_i, p') \in s_*$ and $p \neq p'$. Action a needs to write a goal variable because if it does not, it will not contribute to the solution and it will be unnecessary to follow the tunnel. If the goal variable will be fulfilled after executing a, then we already reached our goal and further following of the tunnel would only destroy this condition.

- Now we have to compare a with every other action. We have to look at all actions that have a precondition where $(v_i, p_i) \in \mathit{eff}(a)$. Every action a' that that has a precondition $(v_i, p_i) \in \mathit{eff}(a)$ also has to satisfy these conditions:

  1. Any action a' has a pre-post condition (v_i, p_i, p_i') where $(v_i, p_i) \in \mathit{eff}(a)$.
  2. The preconditions of a' are satisfied by $s_{min}(a)$, where $s_{min}(a)$ is the union of *prevail* and *eff* of a, formally: $s_{min}(a) = \{prevail \cup \mathit{eff}\}$. $s_{min}(a)$ is the minimal partial assignment that is always true after executing action a. This condition ensures, that it is always possible to execute a' after executing a.
  3. a' only changes the same variables as a does. This means a has the same variables in $\mathit{eff}(a)$ as a' in its effects.

If one action a' does not fulfil these three conditions, then a does not allow a tunnel. If all actions that have a precondition in $\mathit{eff}(a)$ satisfy these three conditions, a allows a tunnel.

If a satisfies the list of conditions and allows a tunnel, we need to find all actions that belong to *tunnel(a)*: Every action that has a pre-post condition (v_i, p_i, p_i') or prevail condition (v_i, p_i), where $(v_i, p_i) \in \mathit{eff}(a)$, belongs to *tunnel(a)*. If an action does not allow a tunnel, then we can not prune and *tunnel(a)* consists of all actions $\mathcal{A}$. Note that *tunnel(a)* only consists of actions that can be applied immediately after executing a.
Since we now have computed *tunnel(a)* we can use it to prune actions:

Pruning Rule 1 After executing action a, prune all actions that are not in *tunnel(a)*.

If we enter a tunnel we want to reach its end before executing any other actions unrelated to the tunnel, like in the sokoban example.

3.1.1 Implementation

Since the computation of the tunnels is not state-dependent, we can pre-compute all tunnels during the initialisation before the search. Once we have all the tunnels we only have to look them up in every search step. We use the previous action a that led to the current state and check if it allows a tunnel. If it does, then we only have to consider all actions that are in $\mathit{tunnel}(a)$. These are the applicable actions for this search step. If a does not allow a tunnel we have to compute all applicable actions normally without any pruning.
The information of which actions allow a tunnel and the inherent tunnels is saved in a *tunnel_map*. Each entry of the map is a pair. The first part of the pair is the action a that allows a tunnel. The second part is a vector with all the actions that belong in the tunnel of a. Only actions that induce tunnel are saved in the map. So when a problem domain does not have any tunnels, the tunnel-map will be empty. So only the pre computation and detecting of tunnels during the initialisation costs some more time compared to the normal A*-search. Since we have to compare every action with all the other actions, the method that finds actions that allow a tunnel and the corresponding tunnel has quadratic time complexity $O(n^2)$. This can be neglected in huge problems, because the search time is much higher, than the pre-computation time. We have a closer look on the performance of tunnel pruning in chapter 4.1.
In order to fill the tunnel-map we have to check all actions whether they fulfil the conditions from 3.1 or not. This is done in the *create_tunnel_map()* method: We go through every action in the current problem and check all the different conditions.
Once we confirm that an action a allows a tunnel, we have to find *tunnel(a)*. To do so, we have go again through all other actions and find all actions that have a pre-post or a prevail condition in the effects of a. All actions that fulfil these, are saved in the *tunnel_map*.

3.2 Partition Based Pruning

The second pruning method we implemented is the *partition based path pruning* [1]. In order to use this method we introduce the partitioned SAS+ problem in 3.2.1. This is then used for the pruning rule in 3.2.2. In 3.2.3 we talk about the difficulties of finding a good partition. Then the section 3.2.4 describes how the pruning method was implemented.

3.2.1 Partitioned SAS+ problem

For the partition-based pruning we need to introduce the *partitioned SAS+ problem*. A partitioned SAS+ problem is defined by the 5-tuple $\Pi = \langle \mathcal{V}, s_0, s_*, \{\mathcal{A}_i\}_{i=1}^{k}, cost \rangle$ for $1 \leq i \leq k$ where $\mathcal{A}_i$ is a set of actions which belong to *partition* i. It is relatively similar to the normal SAS+ problem. The only difference is that the actions can be mapped to a number of different partitions. These partitions are disjoint, which means, every action can only belong to one partition. The partitioning is used to classify actions as *public* or *private*. An action is public if it is not *commutative* [6] with some action from another partition. In this context commutative means that two different actions a, a' do not change the value of the same variable and that a does not change the value of a variable in $pre(a')$ or $prevail(a')$ and vice versa. So the two actions are not commutative, if some variable v occurs in either $pre(a')$ or $prevail(a')$ and $\mathit{eff}(a)$, or v occurs in either $pre(a)$ or $prevail(a)$ and $\mathit{eff}(a')$, or v occurs in $\mathit{eff}(a)$ and $\mathit{eff}(a')$. So if you change the order of two consecutive commutative actions in a plan, the plan is still legal and reaches the same goal with the same cost. All other actions are private.

To get a better understanding of this problem we can use an example from the Logistics domain. The goal is to move a number of packages from its starting location to a specified destination. In order to do so, the packages can be loaded to various types of vehicles, for example trucks or planes, then the vehicles move to a certain location where the packages can be unloaded. Every location belongs to a town. Inside the town each location is connected to other locations with paths. Trucks can only move inside a town but they can reach every location by following the present paths. Planes connect different towns, but for simplification reasons we only use trucks as vehicles. We can describe this domain by using the *SAS+* formalism: Every package and truck is described by a variable. The values for these variables are the present locations. So every location has a different value. There are also values for each truck that the package variables can assume, which means, that the package is on a truck. There are three possible actions, namely *move*, *load* and *unload*. *Move* changes the location for one truck. *Load* removes a package from its location and puts it on the truck if the truck is on the same location as the package. *Unload* removes the package from the truck and places it in the same location as the truck.

In this example we have two towns, two trucks and two packages. Every town has three locations, which are connected. Picture 3.2 shows the starting configuration. To reach the goal, *truckA* has to load *packageA*, then move to *loc3* over *loc2* and finally has to unload *packageA*. *truckB* has to do the same in his town with *packageB*.

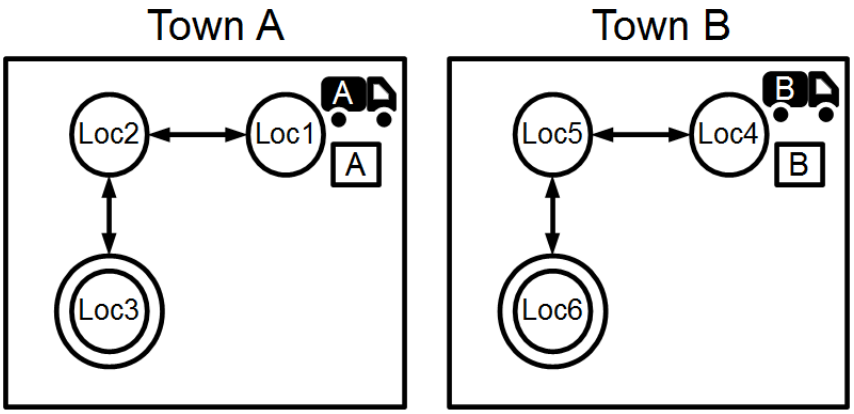

Figure 3.2: Example of a starting configuration for the logistic domain.
Destination = double circle (Loc3, Loc6)

The next step is to find a useful partition for the described problem. Since the vehicles act independently and every action only affects one vehicle, it is a good idea to map all actions that involve a single vehicle to one partition. Therefore, all move, load and unload actions that change the value of the variable of *truckA* belong to partition 1 and the actions that affect *truckB* belong to partition 2. Since the two trucks are located in different towns this also means that the variables of the packages can only be changed by actions of one partition. Now we can describe which actions are public or private. In this example all actions are private, because all actions in partition 1 do not change values of variables(*packageB*, *truckB*) from partition 2 and vice versa. Therefore all actions are commutative to all actions of the other partition.
But this changes, when there are two different trucks in one town, like in picture 3.3. When we use partitions for every truck, like in the previous example, then the value of the variable of the package can be changed by the *unload* and *load* action from every partition. Therefore only the *move* action is private, the other two are public. This makes sense, because, for example, you can not change the order of *unload packageA from truckA* and *load packageA into truckB* when the two trucks are on the same location.

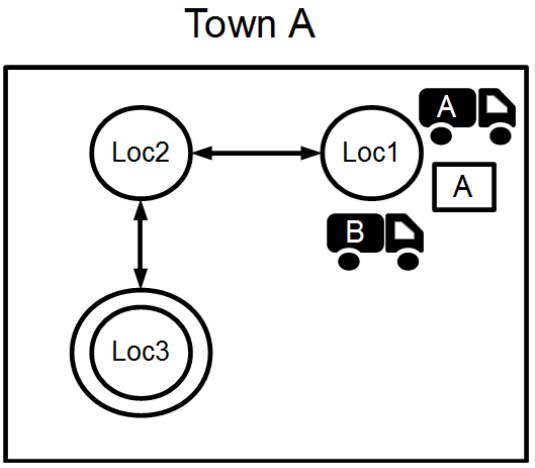

Figure 3.3: Example with two trucks in one town.
Destination = double circle (Loc3)

3.2.2 Pruning Rule

In this section we use the partitioned SAS+ problem described in the previous paragraph to establish a pruning rule. We assume that every action of the problem belongs to a partition and every action is either private or public. The pruning rule by Nissim et al. reads as follows:

Pruning Rule 2 If the previous action $a \in \mathcal{A}_i$ that led to the current state s is private, then prune all actions that do not belong to partition $\mathcal{A}_i$.

The idea is to finish one task before starting another. To illustrate this we can use the example from picture 2 of the previous section. Here the *move* actions are private when there are multiple trucks in one town. When the previous action was a move action for truckA, all other actions for truckB are pruned. This makes sense because we want to move truckA until it reaches its destination and can unload/load the package, before moving the other trucks.

3.2.3 Decomposition

The biggest problem for the partition based path pruning is to find an accurate decomposition of the problem into different partitions. When we have a specific problem domain, e.g. like in the examples from section 3.2.1, it is relatively easy to find a fitting partition. But for a lot of other domains the decomposition is not so obvious. In the logistic domain we have a number of different vehicles, that can execute certain actions independently. So it is only natural to allocate one partition for every vehicle. The challenge is to find a good decomposition rule that can be applied for many different problem domains. A good decomposition has a lot of private actions, because we can only prune in a state s if the last action that led to s was private. But on the other hand a good decomposition also needs a high amount of partitions, because if we, for example, just have one partition, all the actions are private, but then you can not prune actions. Therefore, another problem is to find out how many partitions you have to use to solve a problem. In the logistic example 3.2.1 the perfect number of partitions is two, because there are two different vehicles. But if the number of

partitions is too high, the probability that there are a lot of private actions is very low. If you use too little partitions, the effect of pruning when we encounter a private action is very low, because a partition contains a huge number of actions.

3.2.4 Implementation

Due of lack of time, we were not able to implement everything we wanted for the partition based path pruning. Especially it was not possible to implement the *path pruning A* (PP-A*)* algorithm by Nissim et al. PP-A* is a modification of the classical A* that uses another data structure for the nodes of the *open list*. The open list saves all states that still need to be explored. In PP-A* every node not only saves just the state s, it also saves a set of actions that led to s. So the information about the path from a previous state to s is saved. This information is then used to handle duplicate states. PP-A* guarantees that the used partition based path pruning is optimality preserving. It is not clear, if the normal A* is optimality preserving in combination with the partition based pruning. In our implementation we used the normal A*-algorithm and implemented the pruning rule that is executed in each search step.
Similar to the implementation for the tunnel pruning, we can pre-compute the decomposition of the actions during the initialisation. In order to do so, we have a method called *create_partition_map()* where all actions are distributed to different partitions and then it checks whether an action is private or public. An action $a = \langle pre(a), prevail(a), eff(a)\rangle$ is private, if it is *commutative* to all actions of the other partitions. In order to check this, we first have to compare *eff(a)* with the pre- and prevail conditions of every other action from a different partition. If a variable in *eff(a)* is equal to a variable in any other pre- or prevail condition, then a is public and we can stop the comparison. If not, then we have to check if a variable in *pre(a)* or *prevail(a)* occurs in an effect of any action in a different partition. If a variable does, then a is public and we can terminate. If a passes this test, we have to compare *eff(a)* with all other effects from actions in a different partition and look for same variables. If one variable occurs in *eff(a)* and in any other effect of a action in another partition, then a is public.
In our implementation we added another rule for the distinction between private and public: All actions, that change the value of a variable to a goal value in their effect, are considered public. This prevents that the search reaches a dead end. To illustrate this, we can use the logistic example from section 3.2.1 with two trucks in two different towns: *TruckA* already loaded *packageA* and moved to its destination location. *TruckA* only has to unload the package. But since this unload action is private, we prune all actions that do not belong in its partition, namely all actions that consider *truckB*. Now we can not execute actions for *truckB* to move *packageB* to its destination. So the planner returns that there is no solution. To prevent this, we consider all actions public, that change a value from a variable to its goal value.
Now, we have all the information needed for the pruning. This information is stored in a vector called *partition_map*. Each entry of the *partition_map* consists of the action, the partition to where the action belongs and if the action is private or public. After the initialisation, this information is stored for every action in the problem. The saved information of the *partition_map* is then used to prune. During the search we can then just lookup in each search step whether the previous action a, that induced the current state, is private

or public and prune if necessary. If a is private then we can prune all actions that do not belong to the partition of a.
As stated in section 3.2.3 it is not easy to find a useful decomposition. We implemented two different ways to find a partitioning. First we tried to distribute all actions randomly to a partition. Some experiments showed that this approach is not very useful. Even in small problems with few actions, the probability that an action is private is very small. So in almost every problem there were only public actions. So there were no chances to prune actions. The second idea was to use a approach that iteratively tries to improve the quality of the decomposition. First we allocate every action randomly to one partition. Then we swap the partition for two randomly chosen actions. Then we have to check again what actions are public or private. If the swap lowers the amount of private actions, we have to undo these changes and randomly select two other actions and repeat this procedure until a maximal amount of iterations is reached. The experimental results in chapter 4.2 show how this method performs.
Due to lack of time we unfortunately could not test other possible and more powerful decomposition methods. Another idea was to use a local search approach, that specifically finds public actions and swaps them until the number of private actions does not increase any more.

4

Experimental Results

In this chapter the experimental results of our implemented solution in *Fast Downward* are described. We used a number of different benchmark problem domains, that were already implemented in Fast Downward. These were solved with the A* search algorithm with and without tunnel pruning respectively partition based pruning. Every problem has a maximal computation time of half an hour and a maximal search memory of 2GB. If a problem can not be solved before reaching the maximal values, the problem is considered as unsolvable. We also used different heuristics for the evaluation. First of all we executed a blind search for A* without any information about the distance to a goal state. Then we used two different highly used heuristics: h^{max} [7] which uses delete-relaxation to estimate the distance to the goal and h^{LM-cut} [8] which is based on landmarks.

4.1 Tunnel Pruning

This sections describes the results for the evaluation of the tunnel pruning method. First the results for the blind search, then the ones for the heuristic informed search.

4.1.1 Evaluation with uninformed search

Table 4.1 gives a short summary for the outcome of the computation with uninformed search over all problems. It compares the coverage, therefore how many problems could be solved without reaching the time and memory limitations, the mean number of states that were evaluated/expanded/generated and the average of the total time and of the search time.

Summary	**A*-blind**	**A*-blind with tunneling**
coverage - sum	519	**524**
evaluations - geometrical mean	94520.69	**89966.54**
expansions - geometrical mean	63373.28	**60559.96**
generated - geometrical mean	394897.81	**369708.18**
search time - geometrical mean	0.58	0.58
total time - geometrical mean	0.64	**0.63**

Table 4.1: Summary of blind search and blind search with tunneling

The number of evaluated, expanded and generated states is smaller for the tunnel-pruned search than for the normal A*-search. So there are problem domains where the tunnel pruning rule is used. Since the difference is not very high, we can assume that only a few domains benefit from this pruning rule. Table 4.2 confirms this. In only seven out of 43 domains there is a difference between the number of states that are evaluated or expanded. It is interesting to note that although the number of states that are considered is always smaller in A* with tunneling, the average search and total time is almost identical to the search without pruning. Maybe with an optimized implementation we could shorten the time for the computation of the tunnels.

	evaluations		expansions	
	A*-blind	**A*-blind+T**	**A*-blind**	**A*-blind+T**
driverlog (7)	240401.85	**220189.62**	157819.23	**144356.62**
logistics00 (10)	44966.84	**41146.45**	36450.41	**33842.89**
logistics98 (2)	266071.07	**263554.35**	201674.32	**200381.33**
nomystery-opt11-strips (8)	75588.70	**12394.16**	61402.55	**11518.43**
psr-small (49)	1132.29	**999.96**	996.78	**893.63**
satellite (5)	36461.39	**36461.34**	27364.43	**27364.34**
zenotravel (8)	23195.52	**23082.08**	11321.26	11321.26

Table 4.2: Evaluations and expansions of all domains with different results

Table 4.2 and 4.3 show that there are big differences in the efficiency of the pruning method for the different domains. In some domains (*logistics98*, *zenotravel*) the difference is very small. The domain profits almost nothing from the pruning method. On the other hand the domain *nomystery-opt11-strips* benefits the most. In this domain over six times less states were evaluated and generated and about five times less states were expanded.

Generated	**A*-blind**	**A*-blind + tunneling**
driverlog (7)	1502474.06	**904353.17**
logistics00 (10)	267588.87	**229870.51**
logistics98 (2)	2682296.23	**2630398.61**
nomystery-opt11-strips (8)	190460.33	**28199.37**
psr-small (49)	2687.40	**2374.94**
satellite (5)	408445.14	**388765.69**
zenotravel (8)	120523.08	**119790.28**

Table 4.3: Generated states of all domains with different results

4.1.2 Evaluation with informed search

Now we compare the results for the informed search with the two heuristics h^{max} and h^{LM-cut}. The summary in table 4.4 shows that the difference between the informed search with and without pruning becomes even smaller than in the uninformed search. Especially with the h^{LM-cut} heuristic the gap is almost non-existent. This shows that h^{LM-cut} itself is a really strong heuristic that does not gain much performance with tunnel pruning. But tunnelling has a bigger effect with h^{max} than with h^{LM-cut}. Because of the information from the heuristics the search becomes more focused and the number of visited states becomes smaller. So the probability that an action, that allows a tunnel, is visited becomes smaller.

Like in the blind search, the search and total time is almost the same for pruning and without pruning. Therefore on average it does not hurt to run tunnel pruning but there is also no time gain while using pruning.

Summary	h^{max}	h^{max} + T	h^{LM-cut}	h^{LM-cut} + T
coverage - sum	575	**580**	770	**773**
evaluations - geometrical mean	23854.97	**22939.53**	2432.26	**2415.74**
expansions - geometrical mean	12255.34	**11923.80**	733.55	**732.22**
generated - geometrical mean	79911.11	**76217.63**	4764.10	**4722.50**
search time - geometrical mean	0.66	0.66	**0.29**	0.30
total time - geometrical mean	0.71	**0.70**	**0.31**	0.32

Table 4.4: Summary of A* search with different heuristics

The trend from Table 4.4 also shows in the evaluated, expanded and generated states for the different problem domains (Table 4.5, Table 4.6 and Table 4.7). The number of domains where tunnelling reduces states is even smaller than in the blind search. In the h^{max} search the domains that profit the most from tunnelling are *driverlog* and *nomystery-opt11-strips*. In the other domains the difference of considered states is very insignificant or non-existent. In h^{LM-cut} the gap is even smaller. Only two domains profit from tunnelling. This means that only in two domains the search reached an action that allowed a tunnel, in all other problems the planner found a solution without entering a tunnel. It is interesting to note that tunnelling has a huge impact in the *nomystery-opt11-strips* domain for the search with h^{max}, but it has no impact for the search with h^{LM-cut}.

evaluations	h^{max}	h^{max} + T	h^{LM-cut}	h^{LM-cut} + T
driverlog (7)	76185.65	**66863.99**	1147.65	**902.79**
logistics00 (10)	17835.40	**17290.17**	591.02	591.02
logistics98 (2)	86081.18	86081.18	722.25	722.25
nomystery-opt11-strips (8)	28028.98	**6845.99**	326.04	326.04
psr-small (49)	868.33	**783.51**	581.35	**551.34**
satellite (5)	23904.87	**23697.90**	524.05	524.05
zenotravel (8)	7151.79	7151.79	175.09	175.09

Table 4.5: Evaluated states of all domains with different values for h^{max} and h^{LM-cut}

expansions	h^{max}	h^{max} + T	h^{LM-cut}	h^{LM-cut} + T
driverlog (7)	31344.45	**29613.85**	197.36	**187.33**
logistics00 (10)	12020.59	**11804.12**	160.05	160.05
logistics98 (2)	51184.43	51184.43	89.85	89.85
nomystery-opt11-strips (8)	11562.18	**4182.30**	86.07	86.07
psr-small (49)	602.30	**567.23**	386.24	**377.05**
satellite (5)	15090.07	**15090.00**	74.84	74.84
zenotravel (8)	2523.20	2523.20	25.32	25.32

Table 4.6: Expanded states of all domains with different values for h^{max} and h^{LM-cut}

All in all, the experiments show that the tunnelling pruning method works best in the uninformed search without any heuristic informations. However the amount of domains that profit from tunnel pruning is not very high and the time difference is almost zero for

generated	h^{max}	h^{max} + T	h^{LM-cut}	h^{LM-cut} + T
driverlog (7)	305146.78	**200715.82**	1917.94	**1407.24**
logistics00 (10)	88175.99	**84184.13**	1195.44	1195.44
logistics98 (2)	684862.87	684862.87	1200.47	1200.47
nomystery-opt11-strips (8)	51219.64	**13068.59**	407.44	407.44
psr-small (49)	1750.15	**1570.91**	1119.64	**1055.81**
satellite (5)	226115.80	**215444.90**	1095.83	1095.83
zenotravel (8)	27527.33	27527.33	260.64	260.64

Table 4.7: Generated states of all domains with different values for h^{max} and h^{LM-cut}

normal A*-search and A*-search with tunnelling. So in average tunnelling does not increase the amount of time for the search, but it also does not lower the amount of time. If you use the powerful h^{LM-cut} heuristic, than tunnel pruning achieves almost nothing.

4.2 Partition Based Pruning

In this section the evaluation of our implementation of the partition based pruning (PB) is described. For the evaluation we used our decomposition method from section 3.2.4 that randomly swaps partitions for two different actions. We used 100000 iterations in order two find a good decomposition with a high amount of private action. We also used a fixed number of partitions. In this case we used five different partitions to distribute all actions. Like in the previous section, we first show the results of the comparison with the normal A* for the blind search and then the results for the heuristic search.

4.2.1 Evaluation with uninformed search

Table 4.8 shows that even with our simple decomposition the number of evaluated, expanded and generated states is smaller in A* with partition based pruning than in the normal A*-search. Therefore our iterative random approach finds private actions. But as the data shows the difference of considered states is very low. One reason for that may be the fixed number of partitions. Maybe if the number of partitions scales with the number of actions, the performance of our approach would be better. For problems with a high amount of actions we could use more partitions than for problems with a low amount.
The table also shows that the time used with our solution is drastically higher than for the normal search. This also shows in the sum of the coverage of problems. The normal A*-search covers more problems than our solution. The main reason for this is the iterative swapping of partitions. In each iteration after we swap the partition of two actions, we have to check all actions whether there are private or public. This check has quadratic time complexity. Of course, if we lower the maximal amount of iterations then the total time would become lower, but then the possibility is high that the number of private actions becomes smaller too.
The biggest difference between the normal search and our partition based search is the number of generated states. Table 4.9 shows generated states for different problem domains where the amount of states differ. The table shows that there are a few problem domains where our solution prunes a lot of states. For example, in the pathway-noneg domain the number of generated states is about 60% higher for the normal A* than for our approach

Summary	**A*-blind**	**A*-blind with PB**
coverage - sum	**519**	471
evaluations - geometrical mean	73046.78	**72626.14**
expansions - geometrical mean	49965.87	**49696.35**
generated - geometrical mean	317640.78	**308234.08**
search time - geometrical mean	**0.42**	25.72
total time - geometrical mean	**0.46**	25.73

Table 4.8: Summary of blind search and blind search with partition based pruning

and in the satellite domain the difference is about 25%. This shows that even with our simple decomposition method the number of pruned actions can become relatively high in some problem domains. With a better and more powerful decomposition method it is most likely possible to increase the number of pruned states and to prune in even more problem domains.

Generated	**A*-blind**	**A*-blind + PB**
pathways-noneg (4)	108196.54	**67089.28**
rovers (5)	50138.80	**33874.08**
satellite (5)	408445.14	**304807.57**
tpp (6)	3688.45	**3465.07**
woodworking-opt08-strips (7)	403869.63	**400930.01**

Table 4.9: Generated states of all domains with different results

4.2.2 Evaluation with informed search

Now we discuss the results for the comparison of the informed search with and without partition based pruning from Table 4.10. Similar to the results of the tunnel pruning, the difference between normal and pruned search becomes smaller when we use heuristic functions. The more powerful a heuristic function is the less useful the pruning becomes. So again in the case of the search with h^{LM-cut} the difference of explored states is the smallest. The other figures are similar to the results of the blind search. The number covered problems is smaller for our solution with pruning and the time difference is really high.

Summary	h^{max}	h^{max} + PB	h^{LM-cut}	h^{LM-cut} + PB
coverage - sum	**576**	519	**770**	694
evaluations - geometrical mean	19968.19	**19712.85**	2158.48	**2145.82**
expansions - geometrical mean	10661.38	**10626.76**	669.62	**672.03**
generated - geometrical mean	68047.41	**66199.80**	4245.45	**4205.47**
search time - geometrical mean	**0.47**	25.72	**0.22**	25.05
total time - geometrical mean	**0.51**	25.63	**0.24**	25.05

Table 4.10: Summary of A* search with different heuristics

The results for the specific domains in Table 4.11 also confirm that the pruning method is more efficient with h^{max} than with h^{LM-cut}. It is interesting to note that the number of generated states is even higher for the search with our pruning implementation in h^{LM-cut} than for the normal A* search with h^{LM-cut} for the problem domain *tpp*.

generated	h^{max}	h^{max} + PB	h^{LM-cut}	h^{LM-cut} + PB
pathways-noneg (4)	17409.61	**10757.51**	533.81	**432.02**
rovers (5)	15679.64	**11442.28**	858.57	**750.54**
satellite (5)	226115.80	**176417.40**	1095.83	**1046.81**
tpp (6)	2604.81	**2428.67**	**194.84**	195.60
woodworking-opt08-strips (7)	38339.35	**37808.66**	132.86	132.86

Table 4.11: Generated states of all domains with different values for h^{max} and h^{LM-cut}

Similar to the tunnelling method, the partition based path pruning lowers the amount of explored states in a few problem domains. However the most domains are not effected by the pruning method. With a better decomposition method we could possibly increase the efficiency and especially decrease the time for the computation of the partitions which is the biggest problem of this approach.

5 Conclusion

The evaluation for our implementation of the two different pruning methods shows, that they can be used to reduce the number of states for some problem domains. But the most domains were not effected by the pruning rules. This means that it is very difficult to find good pruning methods that can be applied for a huge amount of problem domains. Our results also show that pruning is more efficient when used in uninformed search. If we use pruning with information from heuristics, the effect becomes smaller. The more powerful a heuristic is, the lower the difference between normal and pruned search becomes. The evaluation also reveals that, for the most part, different domains are effected by the two different pruning rules, although they share the same intuition, namely to finish a task before starting another.
But on the other hand it also shows that there is a huge possibility to increase the performance of our implementations. Especially in the case of the partition based pruning. The biggest challenge for this pruning method is to find a good decomposition. Without a good decomposition method the efficiency and usefulness of the partition based pruning becomes very low. But even with our approach the pruning shows decent pruning rates for a few problem domains. So with a better decomposition method this pruning method probably can be really powerful and effect even more problem domains.

5.1 Future Work

Due to the lack of time we only used simple and few decomposition methods for the partition based pruning. An interesting question would be if we could boost the performance of our implementation with different decomposition methods. In the paper "Tunneling and Decomposition-Based State Reduction for Optimal Planning" [1] the authors use the external tool *Metis* [9] to find a good partition. *Metis* is a graph partitioning package that divides graphs into different partitions. In order to use this package, they mapped all actions to an *action graph*. An action graph is a graph that has a node for every action in the problem. Two nodes are connected if the corresponding actions are not commutative. The authors of the paper use this action graph as input for *Metis* which cuts this graph into partitions. These are then used for the pruning rule. They also introduce a *symmetry score* which

shows how good a decomposition is. In order to find a good partitioning, they create a huge number of decompositions and then compare the symmetry score and pick the best.
It would be interesting to implement this decomposition method and to see how much performance we gain. Or maybe there are other useful decomposition methods that can be used in our implementation.

Bibliography

[1] Nissim, R., Apsel, U., and Brafman, R. I. Tunneling and Decomposition-Based State Reduction for Optimal Planning. pages 624–629 (2012).

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

[3] Hart, P. E., Nilsson, N. J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107 (1968).

[4] Fox, M. and Long, D. PDDL2. 1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research(JAIR)*, 20:61–124 (2003).

[5] Bäckström, C. and Nebel, B. Complexity results for SAS+ planning. *Computational Intelligence*, 11(4):625–655 (1995).

[6] Geffner, P. H. H. and Haslum, P. Admissible heuristics for optimal planning. In *Proceedings of the 5th Internat. Conf. of AI Planning Systems (AIPS 2000)*, pages 140–149 (2000).

[7] Bonet, B. and Geffner, H. Planning as heuristic search. *Artificial Intelligence*, 129(1):5–33 (2001).

[8] Helmert, M. and Domshlak, C. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *ICAPS* (2009).

[9] Karypis, G. and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392 (1998).

Declaration of Authorship

UNIVERSITÄT BASEL

PHILOSOPHISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Erklärung zur wissenschaftlichen Redlichkeit
(beinhaltet Erklärung zu Plagiat und Betrug)

Bachelorarbeit / ~~Masterarbeit~~ *(nicht Zutreffendes bitte streichen)*

Titel der Arbeit *(Druckschrift)*:
Tunnel-Based Pruning for Classical Planning

Name, Vorname *(Druckschrift)*: Federau, Daniel

Matrikelnummer: 11-057-197

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe.

Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Diese Erklärung wird ergänzt durch eine separat abgeschlossene Vereinbarung bezüglich der Veröffentlichung oder öffentlichen Zugänglichkeit dieser Arbeit.

☒ ja ☐ nein

Ort, Datum: Basel, 08.01.2015

Unterschrift: Daniel Federau

Dieses Blatt ist in die Bachelor-, resp. Masterarbeit einzufügen.

UNI BASEL