

# **Partition-Based Pruning for Classical Planning Revisited**

Bachelor's thesis

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

Examiner: Prof. Dr. Malte Helmert  
Supervisor: Dr. Martin Wehrle

Alexander Steinacher  
[alex.steinacher@stud.unibas.ch](mailto:alex.steinacher@stud.unibas.ch)  
12-917-480

12.01.2017

## Abstract

In classical planning the objective is to find a sequence of applicable actions that lead from the initial state to a goal state. In many cases the given problem can be of enormous size. To deal with these cases, a prominent method is to use heuristic search, which uses a heuristic function to evaluate states and can focus on the most promising ones. In addition to applying heuristics, the search algorithm can apply additional pruning techniques that exclude applicable actions in a state because applying them at a later point in the path would result in a path consisting of the same actions but in a different order. The question remains as to how these actions can be selected without generating too much additional work to still be useful for the overall search. In this thesis we implement and evaluate the partition-based path pruning method, proposed by Nissim et al. [1], which tries to decompose the set of all actions into partitions. Based on this decomposition, actions can be pruned with very little additional information. The partition-based pruning method guarantees with some alterations to the A\* search algorithm to preserve its optimality. The evaluation confirms that in several standard planning domains, the pruning method can reduce the size of the explored state space.

# Table of Contents

<b>Abstract</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Preliminaries</b>	<b>2</b>
2.1 Planning . . . . .	2
2.2 Informed Search . . . . .	3
2.2.1 A*-algorithm . . . . .	3
2.3 Action Pruning . . . . .	4
<b>3 Partition Based Path Pruning</b>	<b>5</b>
3.1 Pruning Rule . . . . .	5
3.2 Partitioning . . . . .	6
3.3 Path Pruning A* . . . . .	7
3.4 Implementation . . . . .	8
3.4.1 PP-A* . . . . .	9
3.4.2 Retracing . . . . .	9
<b>4 Experiments</b>	<b>10</b>
4.1 Uninformed Search . . . . .	10
4.2 Informed search . . . . .	13
4.3 Symmetry Score . . . . .	15
4.4 Partitioning . . . . .	16
<b>5 Conclusion</b>	<b>17</b>
5.1 Future Work . . . . .	17
<b>Bibliography</b>	<b>18</b>
<b>Declaration on Scientific Integrity</b>	<b>19</b>

# 1

## Introduction

In classical planning the task at hand is to find an algorithm that can find a plan for a given problem. A plan is a sequence of actions leading, when applied, from the initial state to a goal state of the problem. These actions are each connected to a cost and if a plan with the lowest possible sum of action costs is required, we talk about optimal planning, and its result would be an optimal plan. In many cases only optimal paths are acceptable as a solution. In these cases the used algorithm has to guarantee to only return a solution if it's optimal. One of the biggest problems in planning is the size of the state space that a complex problem can induce. To counter this problem multiple approaches are being researched. One of them is pruning. Pruning introduces a method to analyze a given state and excludes based on this information a part of the applicable actions in that state. Pruning has the effect that fewer states are explored and this directly results in a smaller size of the problem.

In this thesis we discuss a pruning method introduced by Nissim et al. [1], called partition based pruning. This method decomposes the set of all actions in partitions and prunes the actions applicable in a state based on what action has led into that state. Partition based pruning is optimality preserving, meaning when used with an alteration of the commonly used A\* algorithm, it produces an optimal plan. The altered A\* algorithm is called Path-Pruning-A\* and it is described in section 3.3. The most important step for this pruning method is the decomposition of the set of actions, because the quality of the partitioning has a great impact on the usefulness of the pruning in the search process. For this step we implemented a local search algorithm using the symmetry score introduced by Nissim et al. to produce high value partitions at little computational expense.

Finally the thesis contains an evaluation of partition based pruning and its influence in various search scenarios. The results show that the pruning method has a highly varying influence on the efficiency of the search. For some problems the experiments show a strong improvement in the number of states generated when using partition based pruning, while in others the pruning method has no influence at all on the size of the explored space. Additionally the usefulness of the symmetry score for evaluation of partitions and the local search algorithm for partitioning are discussed.

# 2

## Preliminaries

In this chapter we introduce the preliminaries needed for this thesis.

### 2.1 Planning

Planning in artificial intelligence is referred to as the research of algorithms to find a sequence of applicable actions leading from the initial state to a goal state. In this thesis we consider the SAS+ formalism [5] which describes a planning problem as a 5-tuple:

$$\Pi = \langle V, s_0, s_*, A, cost \rangle \quad (2.1)$$

- $V = \{v_1, v_2, \dots, v_n\}$  is a set of variables which can be assigned to a value from the finite domain  $D_v$
- An complete assignment of the variables in  $V$  to a value of its domain is called a state  $s$ . The set of all possible assignments is the set of all states  $S$ .
- $s_0$  defines the assignment of each variable in  $V$  which describes the initial state of the problem.
- $s_*$  is a partial assignment of the variables  $V$  which defines the goal conditions.
- $A$  is a set of actions, each defined as  $a = \langle pre(a), eff(a) \rangle$  where  $pre(a)$  and  $eff(a)$  are partial assignments of  $V$  declaring the preconditions and effects of the action  $a$ . An action  $a$  is applicable in a state  $s$  if the partial assignment of  $pre(a)$  complies with the assignment that defines the state  $s$ . When an action  $a$  is applied the assignment of all variables of the state  $s$  set in  $eff(a)$  are remapped to their value in  $eff(a)$ . The resulting state is called the successor state  $s'$  of  $s$  when applying  $a$  and  $a$  is called a creating action of  $s'$ .
- $cost$  is a function that assigns every action to a non-negative value describing the cost of applying the action.

The search-space induced by a planning problem is a directed graph generated by creating a vertex for every state in  $S$  and adding a directed edge between two vertices if there exists

an action  $a \in A$  that is applicable in the first state resulting in the second state.

A sequence of actions is referred to as a path. If this sequence can be applied starting at the initial state and ends in a goal state, it is called a plan. The length of a plan is the number of actions it contains, its cost is the sum of costs of all its actions. If a plan has the lowest possible cost in a problem it's classified as an optimal plan. In planning we distinguish between satisficing and optimal planning. Optimal planning only accepts a cheapest possible plan as the solution to a planning problem, where in satisficing planning any plan is valid as a solution.

For the this thesis we use the notation for the set of all variables that are assigned in a partial assignment  $p$   $vars(p) := \{v \in V \mid p \text{ assigns a value } \in dom(v) \text{ to } v\}$ . For a partial assignment  $p$  and variable  $v \in vars(p)$ , we denote the value of  $v$  in  $p$  with  $p[v]$ . We say that an action  $a_i$  *destroys* a precondition of an action  $a_j$  if there exists  $v \in vars(eff(a_i)) \cap vars(pre(a_j))$  such that  $eff(a_i)[v] \neq pre(a_j)[v]$ . Additionally an action  $a_i$  *achieves* a precondition of an action  $a_j$  if there exists  $v \in vars(eff(a_i)) \cap vars(pre(a_j))$  such that  $eff(a_i)[v] = pre(a_j)[v]$ . We also say that an action  $a$  *affects* a variable  $v$  if  $v \in vars(eff(a))$ .

## 2.2 Informed Search

Informed search tries to distinguish between good and bad states and prefers good ones in the search process. For this purpose a heuristic function is defined.

$$h : S \rightarrow \mathbb{R}_{\geq 0} \cup \{\infty\} \quad (2.2)$$

This function calculates based on information about the state  $s \in S$  an estimation of the remaining path cost from this state to a nearest goal state. Intuitively the closer this estimation is to the actual cost, the better the heuristic. But good heuristics are often expensive to calculate what causes them to be less useful because of the computational overhead. Because of this a good middle ground has to be found between cheap calculation and accurate estimation.

### 2.2.1 A\*-algorithm

A\* is a form of best-first-search which is used very often in modern applications because of its efficiency and simplicity. In search algorithms the exploration of the search space is often represented as search nodes. A node  $n$  represents a state plus additional information, e.g. the path cost  $g(n)$  from the initial state to the state of the node and a pointer to its predecessor node. When a node is expanded all applicable actions in its state  $s$  are used and for each successor state a new node is created with the expanded node as predecessor node. Best-first-search expands in each iteration, starting at the initial state, the search node with the lowest value of an evaluation function  $f(n)$ . In the case of A\* this evaluation function is calculated from the combination of the path-cost from the initial node to the current node  $g(n)$  and the heuristic value of the current state  $h(n.state)$ . In some cases, to find solutions with low cost, or even to guarantee an optimal solution to be found, states that already have been reached and expanded prior in the search are reopened, when they are reached again with a lower path-cost. If the heuristic function is admissible and consistent [6], A\*

can guarantee to find optimal solutions even without reopening any states.

Figure 2.1: pseudo-code for A\* with reopening

---

```

1 open_list:={}
2 closed_list:={}
3 open_list.insert(new node(none, none, initial_state))
4 while not open_list.is_empty():
5     n := open_list.pop_lowest();
6     if not closed_list.contains(n.state) or get_closed_g(n.state) > get_g(n):
7         closed_list.insert(n.state)
8         if is_goal(n.state):
9             return get_path(n)
10        for each <action, succ_state> in expand(n.state)
11            if heuristic(succ_state) < infinity
12                if not closed_list.contains(succ_state)
13                    open_list.insert(new node(n, action, succ_state))
14                else if get_closed_g(succ_state) > get_g(n)+action.cost
15                    reopen(get_closed_node(succ_state), action)
16 return no_solution

```

---

The A\*-algorithm (figure 2.1) works with two data-structures, the closed list and the open list. The open list contains the search nodes which have yet to be expanded and it initially contains the root node with the initial state of the search problem. The closed list contains states that have been expanded and it is initially empty. For each iteration a node with the lowest f-value is extracted from the open list and checked if its state already is contained within the closed list and if so if the node connected to this state has a lower or equal g value than the current node. If this is not the case, the node's state is tested against the goal conditions, if they are met the path to the node is returned and the search finishes. For non-goal states the node is expanded, meaning all applicable actions in the state and their successor states are generated. If the successor state is not already present in the closed list or the new path to the state is less expensive a new node is generated and inserted in the open list. Inserting a previously closed node again in the open list is called reopening. When the open list is empty and no goal has been reached, the search concludes without a solution.

### 2.3 Action Pruning

Action pruning is a technique to reduce the number of actions applied in a given state to reduce the number of states that have to be explored. The idea behind this is that we exclude actions that are not promising to lead to good states or create redundant work. In the A\* algorithm the pruning would be an extra step when expanding a state (pseudo-code line 10). Instead of using all applicable actions, only the actions that are not pruned by the pruning rule are used in the expansion process. The pruning rule can operate based on different information to prune actions, for example on the current state or on the creating actions of the current node.

# 3

## Partition Based Path Pruning

Two actions are *commutative* if they neither achieve nor destroy a precondition of the other and they don't affect the same variable in different ways when applied. Such two actions  $a_i$  and  $a_j$  when applied consecutively in a plan  $\langle \dots a_i, a_j, \dots \rangle$  can be switched in the sequence to create a plan  $\langle \dots a_j, a_i, \dots \rangle$  with the same cost that is also valid. When searching without further guidance or pruning both of these paths are explored, generating redundant work. Intuitively with the partition based path pruning method actions applied in a given node are pruned based on the action that has led to the node. This way many of these paths aren't explored, resulting in a smaller search space. The idea behind this method is that when the work on a subgoal of the problem is begun, it should be finished before using any actions that do not contribute to this goal and can still be applied later in the process. For this pruning method the actions of an SAS+ problem have to be divided into partitions resulting in a slightly altered definition of the problem.

$$\Pi = \langle V, s_0, s_*, \{A_i\}_{i=1}^k, cost \rangle \text{ with } A_i \cap A_j = \emptyset \text{ for } i \neq j \text{ and } \bigcup_{i=1}^k A_i = A \quad (3.1)$$

Now the actions have been divided into  $k$  partitions and they can be classified as *private* or *public*. An action  $a_i \in A_j$  is public if there exists an action  $a_n \in A_m$  with  $j \neq m$  and  $a_i$  and  $a_n$  are not commutative. Additionally an action  $a$  is also public if it achieves a goal condition of the problem, meaning there exists a variable  $v \in vars(eff(a)) \cap vars(s_*)$  such that  $eff(a)[v] = s_*[v]$ .

### 3.1 Pruning Rule

NIssim et al. [1] proposed the following pruning rule for the partition-based pruning method:

In a state with a private creating action  $a \in A_i$ , all actions outside of it's partition  $A_i$  are pruned.

The idea of this rule is that any path containing multiple private actions of the same partition that do not have a public action of their partition in the sequence between them are moved directly in front of the next public action maintaining their order. This is allowed because



of the commutativity property of private actions with actions of other partitions. As shown by Nissim et al. [1] this retains the completeness and optimality of  $A^*$  because for each path that is not explored with pruning, there exists a path that is explored and contains the same actions as the pruned path but with the before mentioned sequencing rule in place.

### 3.2 Partitioning

Before using the pruning rule in the search, the actions have to be divided into their partitions. This can for example be done by generating an action graph containing a node for each action in the action set  $A$  of the problem. Two action nodes are connected by an undirected edge in the graph if their actions are not commutative, meaning they do not have conflicting effects or destroy or achieve a precondition of the other. Now this graph can be partitioned and result directly in the needed partitioned problem.

Since there are exponential many ways to choose the partitioning, a measurement for how useful a given partitioning would potentially be in the search process is needed. For this purpose Nissim et al. [1] introduce a symmetry score, offering an easily calculated estimation of the value of given partitions. The calculation for the symmetry score  $\Gamma$  for a given partitioning is as follows:

$$\Gamma(\{A_i\}_{i=1}^k) := \sum_{i=1}^k \left( \frac{|\{a|a \in A_i \text{ and } a \text{ is private}\}|}{|A_i|} * \frac{|A \setminus A_i|}{|A|} \right) \quad (3.2)$$

The idea of this symmetry score is as follows: We calculate the probability of the appearance of a private action followed by an action belonging to another partition, because only these actions can be pruned. This calculation is not always accurate, since it regards any action to have the same probability to be applied at any point in the search. However it estimates the usefulness of the decomposition in an intuitive way and has accurate values in the extreme cases where all actions are mapped to one partition or every action is the only action of it's partition. In both of these cases the symmetry score will result in 0, since one of the two factors in the formula remains 0.

For the graph partitioning process the graph partitioning tool METIS [4] was used, which is capable to partition bidirectional graphs very efficiently. The same tool was used by Nissim et al. METIS takes an adjacency list of the graph as input which can easily be generated without much computational effort. The adjacency list can also be used to distinguish between public and private actions after the decomposition process is complete. METIS can be used to generate a user specified number of partitions with the objective that the partitions are connected with as few edges as possible. This objective is called edge-cut.

In addition to Nissim et al. we introduce a local search algorithm (figure 3.1) to choose the number of partitions to be used, which operates based on the symmetry score. The local search starts with 2 partitions, generates the partitioning with METIS, calculates the symmetry score, adjusts the number of partitions by adding a step size and generates the new partitions. The step size starts at 2 and doubles as long as the symmetry score improves. After the symmetry score stops improving while increasing the number of partitions, the area around the number of partitions with the most promising symmetry score is searched

and the most promising partitioning is used for the search. We decided to stop the search after 5 iterations without any improvement to remove the risk of using too much time on partitioning but in the experiments this limit was never reached because the number of partitions used by the algorithm stayed low.

Figure 3.1: pseudo-code for local search

---

```

1  number_of_partitions = 2
2  step_size = 2
3  iterations_wo_improvement = 0
4  max_iterations_wo_improvement = 5
5  best_partition = none
6  best_symmetry_score = 0
7  max_step_reached = false
8  switched = false
9
10 while step_size > 0 and iterations_wo_improvement < max_iterations_wo_improvement
11     partitioning = METIS.part_graph_kway(action_graph, number_of_partitions)
12     symmetry_score = get_symmetry_score(partitioning)
13     if( best_symmetry_score <= symmetry_score)
14         best_partitioning = partitioning
15         best_symmetry_score = symmetry_score
16         if(max_step_reached)
17             step_size = step_size / 2
18         else
19             step_size = step_size * 2
20             iterations_wo_improvement = 0
21     else
22         max_step_reached = true
23         if( switched )
24             step_size = -step_size
25             switched = false
26         else
27             step_size = step_size / 2
28             switched = true
29             iterations_wo_improvement++
30     if(max_symmetry_score == 0)
31         return no useful partitioning found
32     number_of_partitions = number_of_partitions + step_size
33 return best_partition

```

---

### 3.3 Path Pruning A\*

Nissim et al.[1] introduced path pruning A\* called PP-A\*, which is based on the A\* algorithm, described in section 2.2.1, with slight alterations which allow the algorithm to prune based on the creating actions of a state while preserving optimality. The used pruning method needs to have the following properties:

- The pruning method is optimality preserving
- The pruning method prunes based on the previously applied action.

Figure 3.2: pseudo-code for PP-A\*

---

```

1 open_list:={}
2 closed_list:={}
3 open_list.insert(new node(none, none, initial_state))
4 while not open_list.is_empty():
5     n := open_list.pop_lowest();
6     if not closed_list.contains(n.state) or get_closed_g(n.state) > get_g(n):
7         closed_list.insert(n.state)
8         if is_goal(n.state):
9             return get_path(n)
10        actions = prune_actions(n, applicable_actions)
11        for each <action, succ_state> in expand(n.state, actions)
12            if heuristic(succ_state) < infinity
13                if not closed_list.contains(succ_state)
14                    if open_list.contains(succ_state) && get_open_g(succ_state) == get_g(n)+action.cost
15                        get_open_node(succ_state).add_creating_action(action)
16                    else
17                        open_list.insert(new node(n, action, succ_state))
18                else
19                    closed_node = get_closed_node(succ_state)
20                    if get_closed_g(succ_state) > get_g(n)+action.cost
21                        closed_node.remove_all_creating_actions()
22                        closed_node.add_creating_action(action)
23                        reopen(closed_node, action)
24                    else if get_closed_g(succ_state) == get_g(n)+action.cost
25                        get_closed_node(succ_state).add_creating_action(action)
26                        reopen(closed_node, action)
27 return no_solution

```

---

PP-A\* (figure 3.2) is different from A\* in a few aspects. First of all, a search node has to contain all actions which led to its state with the lowest cost. This is needed because all of these actions are used in the pruning process.

The second alteration is that while expanding, an action is applied if there exists an incoming action that allows its use with respect to the pruning rule in the node's set of generating actions.

The last change alters how reaching the same state multiple times is handled. If an already open state is reached again with the same path cost via another action it is added to the list of generating actions in the search node, if the new path-cost is lower, the action list is replaced with only the new action. In the case that a state that is closed is reached again with the same path-cost it's reopened, when expanding such a node only actions that are now possible with the new generating actions are applied.

### 3.4 Implementation

The implementation for our test has been designed as an extension for the Fast Downward [2] planning system. Fast Downward can be used to perform search tasks on problems presented in the propositional PDDL representation [3]. It contains multiple search algorithms and various heuristics and pruning methods that can be used.

### 3.4.1 PP-A\*

The PP-A\* algorithm was realized as an extension of the built-in A\* implementation, adding the changes to the search described in section 3.3. Since our implementation is only intended to run with the partition based pruning method, some aspects of the PP-A\* algorithm were slightly altered to improve performance or adapt it to Fast Downward.

A node contains instead of all actions creating the state with the same path-cost, only one private action per partition or a public action, to reduce memory usage. This is allowed because in the pruning process, the actual action itself is not relevant only its public or private property and its partition if its private, are needed.

### 3.4.2 Retracing

Another change that has to be made is when retracing the plan after the search has concluded. Since multiple generating actions can be saved in a node, a choice has to be made for which action is used for the resulting plan in each state. Normally any of the saved actions can be chosen since all of them have a path with the same cost attached to them, but in the case of 0-cost actions being present the retracing can result in cycles. These cycles are created by paths that are explored with a sum of 0 cost returning to an already contained state. These can not be ignored in the search, because they can potentially result in additional actions becoming applicable in this state. This was handled by specifically not using 0-cost actions in the retracing that were added when reopening a state with the same cost.

# 4

## Experiments

Experiments were performed to evaluate the overall value of the partition based pruning method for different problems. Additionally the application of the pruning method in informed and uninformed search were evaluated. Besides the overall performance, the usefulness of the symmetry score and the local search to find good partitions were explored. For the experiments a set of benchmark problems provided with the Fast Downward planning system were used, containing multiple problems from several standard planning domains. The tests have a memory limit of 2 GB and a runtime limit of 30 min per run.

First of all the experiments verify the optimality of the PP-A\* search with partition based pruning, for all problem instances a plan with optimal cost was found or the search ran out of memory or time. The value of partition based pruning depends as expected heavily on the problem domain, especially on the commutativity of the actions.

### 4.1 Uninformed Search

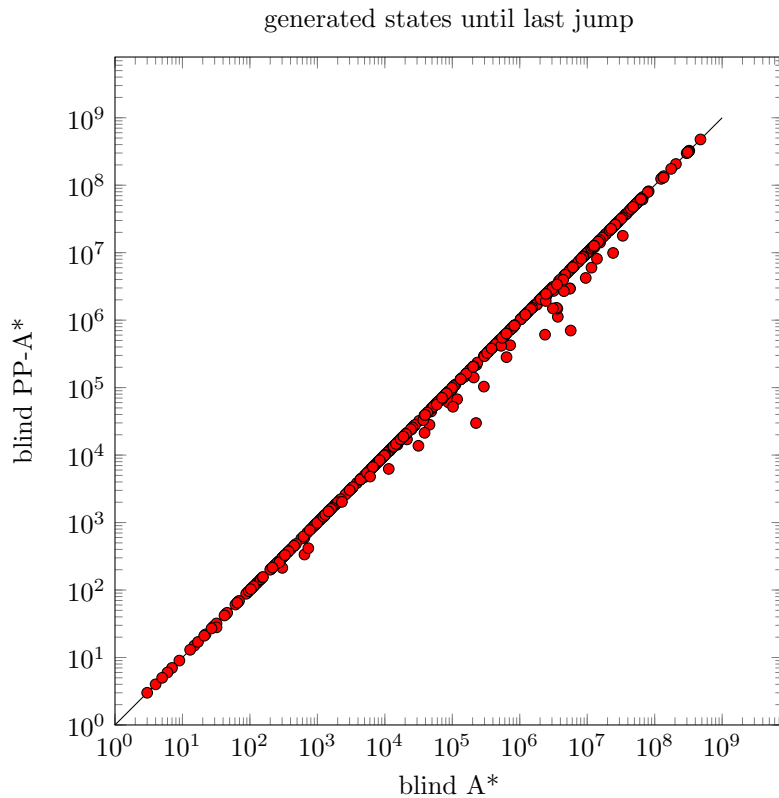
Here we discuss the results of the experiments run with uninformed search, meaning using Fast Downward's blind heuristic. Mainly the comparison between blind A\* and blind PP-A\* with partition based pruning is shown in this section. To inspect the change in size of the search space caused by partition based pruning we compare the number of states generated excluding the last f-layer with blind A\* to the number of PP-A\* with partition based pruning. A state counts as generated if it is ever reached during the search. An expansion is when the successor states of a state are generated, if a state is reopened during the search it can be expanded multiple times and counts every time towards the number of expansions.

Table 4.1: Uninformed search results summary

Summary	blind A*	blind PP-A*
cost - Sum	6'395'625	6'395'625
generated states- Sum	5'925'421'104	<b>5'742'188'449</b>
expansions - Sum	593'886'589	<b>588'347'885</b>
memory - Sum	<b>94'808'800</b>	143'537'164
coverage - Sum	<b>598</b>	578

Overall the amount of generated States with PP-A\* is lower than with A\*. The improvement varies heavily between the problem instances. For example, the problems with a clear multi agent structure show very high improvements with PP-A\*, because they are easy to partition (one partition per agent), the best example for this is the satellite domain where the number of generated states is about 60% reduced compared to A\*.

Figure 4.1: Comparison of generated states with and without pruning for uninformed search



Another interesting behavior that can be seen in figure 4.1 is that in smaller problems, the pruning method does not have much of an influence. This is because of the fact that small problems tend to have fewer actions and are thus more difficult to partition in way that a lot of actions become private. In some cases the pruning method shows no improvements in the size of the explored search space. This is caused by the problem not having actions that can be partitioned in a way that actions can be pruned in the search or by the fact that our local search doesn't always find these partitions.

Table 4.2: Uninformed search results by domain

domain	A* gen.	PP-A* gen.	A* cov.	PP-A* cov.
airport	113'000'242	112'848'865	21	21
driverlog	116'045'495	115'411'418	7	7
elevators-opt08-strips	95'683'946	90'100'870	10	9
elevators-opt11-strips	95'115'562	89'544'286	8	7
floortile-opt11-strips	65'976'949	61'229'356	2	1
logistics00	11'434'652	11'434'652	10	10
logistics98	4'554'852	4'549'483	2	2
mprime	52'948'085	51'672'008	18	17
mystery	33'346'122	33'232'555	15	14
openstacks-opt08-strips	6'676'982	6'669'495	18	18
openstacks-opt11-strips	6'652'254	6'644'835	13	13
parcprinter-08-strips	7'945'013	7'824'901	10	9
parcprinter-opt11-strips	7'929'566	7'810'574	6	5
pathways-noneg	5'176'108	4'728'130	4	4
psr-small	13'354'909	13'350'791	48	47
overs	135'111'806	129'623'061	5	5
satellite	5'926'926	745'421	4	4
scanalyzer-08-strips	709'100'492	707'927'453	12	9
scanalyzer-opt11-strips	532'842'564	532'282'403	9	6
tetris-opt14-strips	46'595'649	46'325'490	7	7
tpp	119'411	68'803	5	5
transport-opt08-strips	52'687'623	27'323'838	11	11
transport-opt11-strips	52'628'620	27'288'429	6	6
transport-opt14-strips	18'867'064	8'794'897	4	5
woodworking-opt08-strips	40'894'144	18'840'918	7	7
woodworking-opt11-strips	38'105'951	18'032'722	2	2
zenotravel	10'750'124	5'946'238	7	7
<b>Sum</b>	<b>2'279'471'111</b>	<b>2'140'251'892</b>	<b>271</b>	<b>258</b>
Others	3'470'460'433	3'470'460'433	327	317
<b>Sum</b>	<b>5'749'931'544</b>	<b>5'610'712'325</b>	<b>598</b>	<b>575</b>

No improvements can be seen in about 40% of the domains. This also explains the fact that the overall memory usage is lower without partition based pruning, since the memory needed per node is higher in PP-A\* because additional creating actions have to be saved where A\* only ever saves one creating action. This leads to higher memory usage of PP-A\* while solving problems where no good partitions can be found. The number of expanded states shows a very similar pattern as the number of generated states. The coverage of PP-A\* is slightly lower than the one of A\*, this is again directly caused by the increased memory usage of PP-A\*, and in the cases where not a lot of pruning takes place, the memory usage can not be equalized by generating a lower amount of states.

## 4.2 Informed search

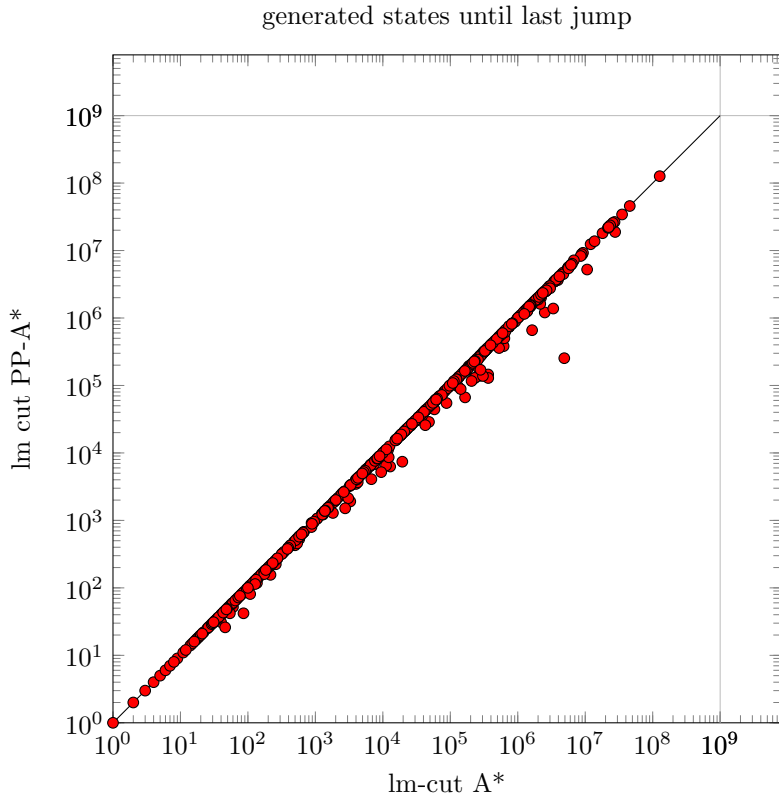
The experiments for the informed search were conducted with the LM-cut heuristic [7]. LM-cut is a very powerful heuristic that is efficiently calculable using landmarks. We compare the statistics of A\* against PP-A\* when both are using the LM-cut heuristic. Overall the improvements with respect to state generation when using PP-A\* are much less noticeable when using a powerful heuristic. This can be explained with the fact that a heuristic stops the search from exploring some of the same states which would not be explored because of the pruning method.

Table 4.3: Informed search results summary

Summary	lm-cut A*	lm-cut PP-A*
cost - Sum	6'395'625	6'395'625
generated until last jump - Sum	478'746'717	<b>475'332'039</b>
expansions until last jump - Sum	55'301'339	<b>55'254'598</b>
memory - Sum	<b>20'604'712</b>	58'668'740
coverage - Sum	873	873

Like in the uninformed search difference in the number of generated states is lower with PP-A\*, but the difference is not as noticeable. The domains having the highest difference in generated states are the same as in the uninformed search because the partitioning process is the same in both cases resulting in the same pruning in a given state.

Figure 4.2: Comparison of generated states with and without pruning for informed search





An interesting difference can be seen in the number of expansions, because in comparison to the uninformed search, when using informed search the fact that PP-A\* reopens states when reached with the same path cost using a new action comes into play. These states are not reopened when using the normal A\* algorithm. The reopening of these states can result in the possibility of PP-A\* expanding them multiple times even though the path-cost to the state did not change in between.

Table 4.4: Informed search results by domain

domain	A* gen.	PP-A* gen	A* cov.	PP-A* cov.
driverlog	3'674'560	3'640'485	13	13
elevators-opt08-strips	62'928'198	55'933'649	22	22
elevators-opt11-strips	51'633'260	50'314'479	18	18
floortile-opt11-strips	35'034'404	34'103'855	7	7
floortile-opt14-strips	62'025'753	60'215'404	6	6
logistics00	12'849'032	12'144'973	20	20
logistics98	2'230'437	2'150'078	6	6
mprime	2'559'414	2'559'407	22	22
mystery	29'697'441	29'697'361	17	17
openstacks-opt08-strips	5'707'994	5'700'527	19	18
openstacks-opt11-strips	5'684'374	5'676'955	14	13
openstacks-opt14-strips	2'543'816	2'543'816	3	2
parcprinter-08-strips	30'208'366	20'133'498	19	19
parcprinter-opt11-strips	30'208'357	20'133'489	14	14
pathways-noneg	2'158'087	1'624'680	5	5
pipesworld-notankage	12'063'640	12'063'118	17	17
psr-small	23'710'452	23'710'102	49	48
rovers	1'894'389	1'466'287	7	7
satellite	5'057'697	322'179	7	12
scanalyzer-08-strips	13'942'542	13'942'542	16	12
scanalyzer-opt11-strips	140'983'160	140'387'216	13	10
tetris-opt14-strips	4'768'909	4'730'899	6	6
tidybot-opt14-strips	500'349	500'349	9	9
tpp	233'191	130'970	6	6
transport-opt08-strips	426'278	249'540	11	11
transport-opt11-strips	423'275	247'833	6	6
transport-opt14-strips	3'396'223	1'645'447	6	6
woodworking-opt08-strips	6'109'688	5'382'655	17	19
woodworking-opt11-strips	6'109'399	5'382'438	12	13
zenotravel	7'921'205	7'293'315	13	13
<b>Sum</b>	<b>566'683'890</b>	<b>524'027'546</b>	<b>400</b>	<b>397</b>
Others	300'422'896	300'422'896	474	474
<b>Sum</b>	<b>867'106'786</b>	<b>824'450'442</b>	<b>874</b>	<b>871</b>

The experiments show that, when using a powerful heuristic the impact partition based pruning has on the search becomes less noticeable, but when the partitioning process results in valuable partitions it results in a considerable amount of states that are not explored because of the pruning method. In the best case we had in the experiments the (satellites domain) the reduction of generated states is about 40% compared to the A\* algorithm when using the LM-cut heuristic.

### 4.3 Symmetry Score

In this section we discuss the symmetry score as proposed by Nissim et al. [1]. The symmetry score has the purpose of evaluating given partitions in how valuable they are for the pruning method. Based on this score we chose the used partitions in our experiments. Here we want to see how much of a correlation there is between the symmetry score and the reduction of the search state size due to pruning. In all of the cases where the symmetry score remained at 0 no improvement for the search can be seen, which shows that the symmetry score can identify bad partitions.

Table 4.5: Symmetry score - generated states evaluation

<b>problem</b>	<b>s-score</b>	<b>blind A*</b>	<b>blind PP-A*</b>	<b>LM-cut A*</b>	<b>LM-cut PP-A*</b>
satellite 01	0	576	576	14	14
satellite 02	0	13'245	13'245	17	17
satellite 03	0.88	224'751	29'870	86	42
satellite 04	0.89	5'688'354	701'730	0	0
satellite 05	0.53			3'290	1911
satellite 06	0.65			165'598	66'610
satellite 07	2.71			4'888'692	253'585
transport 01	0.08	302	213	5	5
transport 02	0.08	11'432	6'236	128	114
transport 03	0.08	3'094'556	1'489'999	48'269	28'805
transport 04	0.07	33'642'535	17'801'547	206'707	116'501
transport 05	0.09	734	416	46	26
transport 06	0.09	102'280	52'027	1'820	1'290
transport 07	0.09	3'508'797	1'523'191	12'847	6'289
transport 08	0.11	644	336	60	52
transport 09	0.08	45'891	28'208	2'764	1'510
transport 10	0.06	721'372	424'985	11'154	6'490
transport 11	0.06	11'559'080	5'996'680	142'478	88'458
woodworking 01	0.66	87'070	60'610	54	42
woodworking 02	1.55	293'533	103'512	0	0
woodworking 03	0.72			40	31
woodworking 04	0.25	37'186	32'900	19	19
woodworking 05	1.68	24'171'231	9'915'004	31	31
woodworking 06	1.32	2'312	2'027	0	0
woodworking 07	3.54	2'368'092	609'147	216	156
woodworking 08	2.35	13'934'720	8'117'718	49	49
woodworking 09	1.82			273'904	139'085

in table 4.5 we can see the symmetry score of the partitions used by our algorithm and the number of generated states with and without pruning. It is clearly visible that the higher the symmetry score, the higher the impact of the pruning tends to be. However based on the symmetry score alone, it can not reliably be predicted how much the pruning will improve the search especially when using a strong heuristic like LM-cut. Overall we can say that the symmetry score provides a useful estimate of the value of given partitions for partition-based pruning.

## 4.4 Partitioning

In the partitioning process our approach was to use a local search algorithm to get the number of used partitions in addition to the graph partitioning method proposed by Nissim et al. [1], since in the paper the method used to find the number of partitions is not described. From the experimental data we can see that our approach can find good partitions in some of the problems. In about 40% of the cases no valuable partition could be found resulting in no pruning during the search. The local search concludes in all of the cases in less than 10 iterations which results in less preprocessing time used, but overall the speed increase due to pruning in the search can not counteract the needed time to find a good partition unless the problem is very large in size and the found partitions are exceptionally good for the pruning method. Another downside of the local search is that it assumes that if for 2 partitions the symmetry score results in 0 no good partitions can be found, when potentially a large number of partitions could have a symmetry score  $> 0$  but finding this number if it exists would require a lot of time to calculate.

Overall our approach can result in good partitions in very few iterations for problems that can be partitioned in a low number of partitions.

# 5

## Conclusion

In this thesis we have shown that the partition based pruning method can have a significant impact on the size of the explored space. The overall improvement varies highly between the problems and especially between the problem domains. The pruning method is most effective when used on problem domains which have a multi agent structure to them because they can be partitioned easily. For many problems our implementation for partitioning is not able to find partitions which are useful for the pruning method and thus do not improve the size of the explored search space and have even a negative impact on memory usage and time needed, because the usage of the pruning method takes additional computational resources. Partition based pruning also has more impact on the performance of uninformed search than of informed search because the heuristic can hinder the search from exploring states which would also not be explored when using partition based pruning.

### 5.1 Future Work

Nissim et al. proposes a second pruning method in the paper called action tunneling pruning, which can be combined with partition based pruning and should further improve the search efficiency, while both use some of the same information and prune based on creating actions of a state. It would be interesting to see further tests where both of the pruning methods work together and how much this would improve the overall performance of the search.

Another possible continuation of this work is to further explore the partitioning process and test other approaches on finding partitions. This includes investigating alternatives to the symmetry score to decide if a partition is useful in the search.

Our implementation of PP-A\* could also be improved to lower memory usage and runtime. For example another data structure to save all creating actions of a node could be changed to improve memory usage, this was difficult to implement because the Fast Downward Planner we worked with is not designed to save more than one creating action per node.

## Bibliography

1. Nissim, R., Apsel, U., and Brafman, R. I. “Tunneling and Decomposition-Based State Reduction for Optimal Planning”, pp. 624–629 (2012).
2. Helmert, M. “The Fast Downward Planning System”. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246 (2006).
3. 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).
4. “A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs”. George Karypis and Vipin Kumar. *SIAM Journal on Scientific Computing*, Vol. 20, No. 1, pp. 359–392, 1999.
5. Christer Bäckström and Bernhard Nebel, “Complexity results for SAS+ planning”, *Computational Intelligence*, 11, 625–656, 1995
6. Judea Pearl, “Heuristics: Intelligent Search Strategies for Computer Problem Solving”, Addison-Wesley, 1984
7. Malte Helmert and Carmel Domshlak, “Landmarks, critical paths and abstractions: what’s the difference anyway?”, in ICAPS, 2009

# Declaration on Scientific Integrity

## Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud  
beinhaltet Erklärung zu Plagiat und Betrug

**Author — Autor**

Alexander Steinacher

**Matriculation number — Matrikelnummer**

12-917-480

**Title of work — Titel der Arbeit**

Partition-Based Pruning for Classical Planning Revisited

**Type of work — Typ der Arbeit**

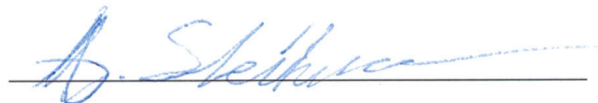
Bachelor's thesis

**Declaration — Erklärung**

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

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.

Basel, 12.01.2017



Signature — Unterschrift