University of Basel

Solving Delete-Relaxed Planning Tasks by Using Cut Sets

Bachelor 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. Thomas Keller and Cedric Geissmann

Marvin Buff
marvin.buff@stud.unibas.ch
2014-054-191

13/06/2017

Acknowledgments

I would like to express my thanks to Dr. Thomas Keller and Cedric Geissmann for their support and supervision of my work. I would also like to thank Professor Malte Helmert, who allowed me to write this thesis in his research group. Special thanks goes to Beat Bürgler who proofread this thesis.

Abstract

Classical domain-independent planning is about finding a sequence of actions which lead from an initial state to a goal state. A popular approach for solving planning problems efficiently is to utilize heuristic functions. A possible heuristic function is the perfect heuristic of a delete relaxed planning problem denoted as h^+. Delete relaxation simplifies the planning problem thus making it easier to find a perfect heuristic. However computing h^+ is still NP-hard problem [4].

In this thesis we discuss a promising looking approach to compute h^+ in practice. Inspired by the paper from Gnad, Hoffmann and Domshlak [9] about star-shaped planning problems, we implemented the Flow-Cut algorithm. The basic idea behind flow-cut to divide a problem that is unsolvable in practice, into smaller sub problems that can be solved. We further tested the flow-cut algorithm on the domains provided by the International Planning Competition benchmarks, resulting in the following conclusion: Using a divide and conquer approach can successfully be used to solve classical planning problems, however it is not trivial to design such an algorithm to be more efficient than state-of-the-art search algorithm.

Table of Contents

1 Introduction

Classical planning is about creating algorithms that can automatically solve planning problems. A solution to a planning problem is a sequence of actions (also called plan) leading from the initial state to a goal state. A well-known planning problem is the travelling salesperson problem (TSP), where a salesperson must visit every city on his list once with the condition that he takes the shortest route in regard to the overall travelling distance. A possible algorithm that could be used to solve the TSP is a blind search algorithm. A blind search algorithm like breadth first search (BFS) uses a search tree to consider states with increasing distance to the initial state. Solving a TSP instance with ten cities on the list would require a BFS algorithm to expand 3'628'800 states in the worst case. If just one city is added to the list, the number of states that had to be expanded increases to 39'916'800. If you go even further and add another city to the list, the needed number of states reaches 479'001'600. The number of states that have to be expanded (search cost) with BFS grows exponentially in the size of the problem variables, therefore limiting the problems that can be solved by today's computers.

To overcome this limitation, informed search algorithm use heuristic functions to reduce the search cost. A heuristic function gives an estimate on how far the current state is away from a goal state. By comparing the heuristic values of the other states, the algorithm can determine which direction looks most promising to evaluate, hence making the algorithm goal-oriented and reducing thereby the search cost. As an example, using greedy search algorithm to solve the TSP with twelve cities on the list, results in roughly 500 states that have to be expanded. However, finding and computing a good heuristic for a planning problem is not always possible or just too expensive. That is the reason why many algorithms in classical planning use delete relaxation. As the perfect heuristic for a delete-relaxed planning problem can be used to solve the original problem.

Delete-relaxation assumes that every fact persists once achieved by ignoring all negative effects of actions. The objective is to determine the exact goal distance for the relaxed plan, denoted by h^+. But as of yet delete relaxation heuristics can only calculate an estimate of h^+ [1] since h^+ itself is NP-hard to compute [4]. Three algorithms that provide an estimate for h^+ are the max, additive and ff heuristic. The max heuristic h^{max} [1] computes an estimate that is always lower bound to the h^+ value. Thus, providing an admissible estimate to h^+. The additive heuristic h^{add} [1] and ff-heuristic h^{ff} [4], on the other hand, compute an estimate that is always upper bound to the h^+ value and therefore not admissible. Thus cannot be used by most algorithms, as they need an admissible heuristic to compute an optimal solution.

The goal of this thesis is to create a domain-independent algorithm that can compute h^+ in practice. However, since this is a NP-hard problem we need to use a smart approach to get good results. Inspired by Divide and Conquer (further referenced as D&Q), we figure that dividing a planning task into smaller planning tasks could be beneficial to subsequently reduce the complexity to solve the problem from an original $O(2^n)$ to $O(2^{\frac{n}{2}})$. The algorithm we implemented works as follows: First, we derive the causal graph of the planning task, thereby enforcing delete relaxation. Resulting in a graph that represents our problem. The idea is to split this graph into three smaller sub-graphs: a left, middle and right graph. Like their name suggests is the order of them important, as it enables us to solve the resulting problems individually without considering going back and forth between left and right problems. The order we enforce is that all connections between the sub graphs must be directed from left over middle to right. Next, we merge the middle graph with the others, creating two new graphs with an overlapping part. At last we create new planning tasks from the two graphs and subsequently solve them. By combining the solution from the sub tasks, we can determine the solution of our original task. Important to notice here is that the overlapping part of the graphs enables us to solve the sub task separately and later assemble the solution.

The thesis is structured as follows: First, we will define the theoretical background needed for this thesis, like classical planning, heuristics and delete relaxation. Afterwards, we will depict the flow-cut algorithm we introduced and implemented to solve planning problems. We will continue by discussing the experiments conducted on the state-of-the-art planning problems and in the last chapter, we will provide a short summary and conclusion as well as an outlook into future work.

2 Background

In this chapter we will introduce notations and definitions, emphasizing on formalisms that are needed throughout this thesis. Furthermore we will provide a brief introduction in classical planning and some related topics like delete-relaxation and causal graphs.

2.1 Classical Planning

"Planning is the reasoning side of acting. It is an abstract, explicit deliberation process that chooses and organizes actions by anticipating their expected outcomes." [5, Ghallab et al., 2004, p.1]

Planning problems are formalized as planning tasks in order to provide a general description and thus be solved by generalized algorithms. In this thesis we are using the propositional STRIPS formalism [3, Bylander 1994] to describe a planning task formally.

Definition 1 (planning task)**.** *A* planning task *is a 4-tuple* $\Pi = \langle F, O, I, G \rangle$*, where*

- V *is a finite set of* propositional state variables *(also called* propositions*),*
- O *is a finite set of* operators*, each with associated* preconditions $pre(o) \subseteq F$*, add* effects $add(o) \subseteq F$*,* delete effects $del(o) \subseteq F$ *and the* cost *of applying the operator* o $cost(o) \in \mathbb{R}_0^+$*,*
- $I \subseteq F$ *is the* initial state*, and*
- $G \subseteq F$ *is a set of* goal variables*.*

A state $s \subseteq V$ is a set of variables, where every variable included in the set is true and the others are implicitly false. All possible states are defined as $S = P(V)$, where $P(V)$ is the power set over all variables. An operator o is *applicable* under the condition that the current state contains all variables $v \in pre(o)$. By applying the operator o, the state s is changed by deleting all variables $v \in del(o)$ and adding the variables $v \in add(o)$ resulting in a new state, $s[o] := s \cup add(o) \setminus del(o)$.

Definition 2 (plan)**.** *A plan π for a planning task is sequence of operators $o^1, \ldots ,o^n$ such that applying the operators to the initial state I is resulting in a goal state, $I[o_1][o_2] \ldots [o_n] \models G$. The cost of a plan π is the summed cost of all operators included in the plan π. A plan π is optimal if its cost is minimal compared to all other possible plans.*

Planning problems can be distinguished between satisficing and optimal planning. The objective of satisficing planning is to find a solution to the problem or to prove that none exists. Where in optimal planning, the goal is to find the optimal solution. In this thesis we will only consider optimal planning.

2.2 Heuristic Search

Search algorithm using *heuristic functions* (simply referenced as heuristics) are called *informed* search algorithms, because they are goal-oriented. This means they are taking information about how far away a state is from a goal state into consideration to find a *plan.* A heuristic maps each state of a planning task $\Pi = \langle F, O, I, G\rangle$ to a number $h : S \to \mathbb{N}_0 \cup \{\infty\}$. The perfect heuristic $h^\star$(s) returns the exact optimal distance to the goal for every state $s \in S$. An important property of heuristics is admissibility. A heuristic h is admissible if for every state $s \in S$, $h(s) \leq h^\star$(s), therefore it never overestimates the cost of a plan π. This is an important feature as many planning algorithms need an admissible heuristic to be able to find the optimal solution.
In this thesis, we design an algorithm that can exactly compute h^+, the perfect heuristic of a relaxed planning task Π^+. Computing h^+ is NP-hard [4, Hoffmann and Nebel 2001] but if successfully acquired it could be used as an admissible heuristic for the planning task Π.

2.3 Delete Relaxation

Delete relaxing a planning task $\Pi = \langle F, O, I, G\rangle$ removes all delete effects from its operators. Accordingly, variables can be added yet not be removed. Therefore, every variable remains true once it is achieved.

Definition 3 (delete relaxation)**.** *The relaxation of a* STRIPS planning task $\Pi = \langle F, O, I, G\rangle$*, is the four tuple $\Pi^+ = \langle F, \{o^+ | o \in O\}, I, G\rangle$, where the relaxation o^+ of o is the operator with*

- *pre(o^+) = pre(o),*
- *add(o^+) = add(o),*
- *cost(o^+) = cost(o) and*
- *del(o^+) = $\emptyset$.*

Many heuristics use delete relaxation as a foundation, because it simplifies the problem and still provides an accurate estimate. Delete relaxation works good, due to the variable that there are good and bad effects. The good effects lead us closer to the goal state (add effects). The bad effects (del effects) imply constraints and elongate the way to the goal. Consequently the good effects give a good heuristic estimate even when ignoring the negative.

2.4 Causal Graph

In this thesis, we use causal graphs derived from planning tasks $\Pi = \langle F, O, I, G \rangle$. A causal graph consists of vertices and edges, where the vertices are variables and the edges are depicted from the causal relation between the variables.

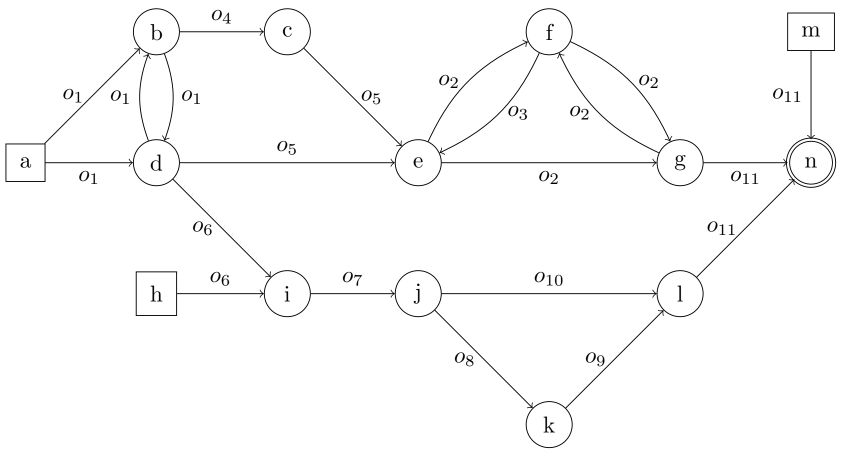

Figure 2.1: The causal graph derived from the example planning task.

Definition 4 (causal graph)**.** *A causal graph is a graph $C = \langle F, E \rangle$, that can be derived from a given planning task $\Pi = \langle F, O, I, G \rangle$, with*

- *set of vertices $VE = V$ and*
- *a set of edges E such that $\langle f_1, f_2 \rangle \in E$ iff there is an operator $o \in O$ with $f_1 \in pre(o)$ and $f_2 \in add(o)$.*

At this point we would like to introduce an example planning task used in this thesis. The causal graph shown in Figure 2.1 is derived from the planning task $\Pi = \langle F, O, I, G \rangle$ where:

- $V = \{a, b, c, \dots, l, m, n\}$
- $O = \{o_1, o_2, o_3, \dots\}$, $cost(o) = 1 \mid o \in O$ and precondition and effect are as followed:
  - o_1: $a \rightarrow b$ and d
  - o_2: $e \rightarrow f$ and g
  - $\dots$
- $I = \{v, a\}$
- $G = \{n\}$

By applying this procedure on the example planning task 2.4, we get the directed graph shown in Figure 2.1. Every node represents a variable and its shape indicates if it is an initial variable (shown as rectangle), goal variable (shown with two circles) or neither (shown as a single circle). The edges are labeled by the operator they are derived from and are directed from precondition to effect. Exception to this rule are the edges connecting two effects from the same action, indicated by two opposing edges side by side.

2.5 Strongly Connected Components

A directed graph $G = \langle V, E \rangle$ is *strongly connected* or *diconnected* when every vertex $v \in V$ is reachable from every other vertex $v' \in V$. A *strongly connected* sub-graph is called *strongly connected component* (SCC). Computing all SCC's of a directed graph G can be useful as a preprocessing, when an algorithm needs the SCC's or only works on graphs with a certain structure.

Applying a SCC algorithm on the causal graph computed in the previous chapter, results in the following nine SCC's: {{a}, {b, d}, {c}, {h}, {i}, {e, f, g}, {j, k, l}, {m}, {n}}, shown in Figure 2.2.

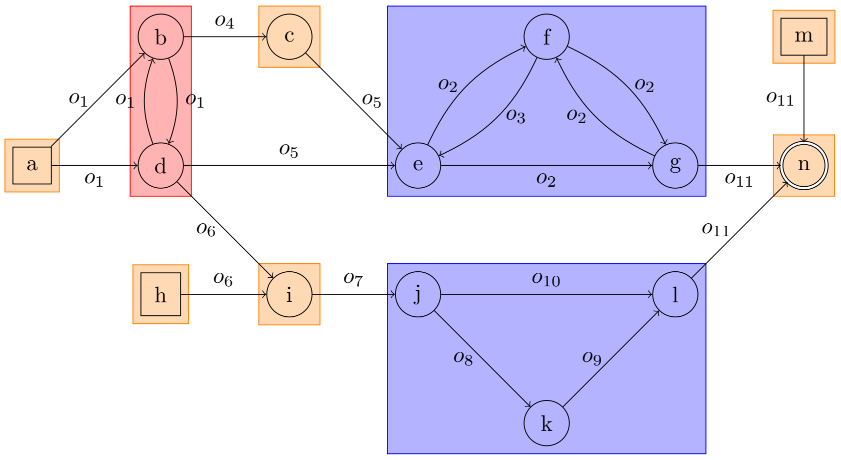

Figure 2.2: The example causal graph with all strongly connected components grouped and highlighted.

3 Flow-Cut Algorithm

The following sections discuss the implementation of the flow-cut algorithm, which is shown in Figure 1.

As mentioned before, our objective is to calculate the exact value of the delete relaxed heuristic h^+. The algorithm we introduced achieves this goal by performing two enclosed steps. First, the planning problem $\Pi = \langle F, O, I, G \rangle$ is split into three parts: A left, middle (also referred as *cut*) and right part, where all connections between them are directed from left to middle and from middle to right. Assuming that the problem can be split into such a structure, the algorithm combines the left and right part with the middle segment, thus creating different planning tasks with an overlapping section. Second, the newly created smaller sub-problems must be solved individually and afterwards be reassembled to get the solution of the delete-relaxed planning problem Π^+.

The idea behind this algorithm is to break down the original planning task into smaller ones with the intention of reducing the complexity from $O(2^n)$ to $O(2^{\frac{n}{2}})$. This works due to the complexity depending on the size of a planning task. However, creating and organizing sub-tasks creates an overhead that scales with the number of variables in the cut. As a result we are eager to make the cut as small as possible.

Algorithm 1 Flow-Cut Algorithm

```
procedure FLOW-CUT(P)                                  ▷ Solves the planning problem P
    $cg \leftarrow$ DERIVECAUSALGRAPH(P)
    $\langle$l,c,r$\rangle \leftarrow$ MAKEINITIALCUT(cg)   ▷ 3-tuple of sub-graphs,$\langle left, cut, right\rangle$
    $minimal \leftarrow false$
    while not $minimal$ do
        $minimal \leftarrow$ SHRINKCUT($\langle$l,c,r$\rangle$)      ▷ Hill climbing algorithm
    end while
    return CREATESOLVEPROBLEMS(subset,l,r,P)
end procedure
```

3.1 Preparation of the Planning Problem

Before the problem can be split into different parts, it must be modified to suit the task ahead. Therefore, we decided upon deriving the causal graph from the given planning problem, thus enforcing delete-relaxation and simultaneously providing the correct structure to apply a *SCC* algorithm. With the delete-relaxation in place, we can simplify the main planning task even more to consequently reduce the size of the final sub-tasks. First, we compute the *SCC*s of Π^+, as this is a low-cost operation with a linear complexity, and it provides us the information we need to split the graph in a further step. Second, we remove *SCC*s that are left over from the initial task but no longer hold any useful information. Like a *SCC* with no goal state and no other connections to other *SCC*s as through incoming edges, or a *SCC* with only an initial state but without outgoing edges. Both have no effect on the relaxed planning task Π^+ and can therefore be left out. In Figure 3.1 is a example shown of this procedure, vertex o and vertex p can be removed since they have no outgoing connection to the rest of the graph.

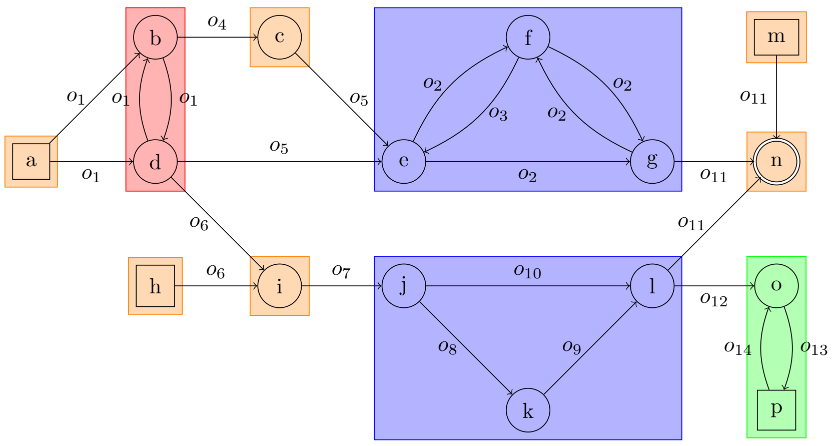

Figure 3.1: Causal graph with two vertices that could be removed as they have no influence on the solution (green highlighted).

3.2 Decomposition of the Planning Problem

The core aspect of the flow-cut algorithm is the determination of the cut, as its size has a major impact on the run-time of the algorithm, due to the number of sub-tasks created is depending on it. However, finding a good cut is not a trivial task as different problems have a fundamental diverse structure. As a basis we use the already established *SCC* algorithm. This provides us with the strict structure we need and gives us an estimate on which problems the algorithm can achieve good result. As example, we cannot make a good cut if the *SCC* algorithm only finds one *SCC*, since all facts are connected together. Yet having multiple *SCC* does not assure that a good cut can be made, as different compositions still could impose problems. With this in mind, *SCC*s still give an indication and can be useful to interpret the result in the end.
Having this covered we can go into the procedures shown in Figure 2 and 3.

Algorithm 2 Making Initial Cut Algorithm

```
procedure makeInitialCut(cg)                                  ▷ Returns initial cut
    sccs ← computeSCC(cg)
    for scc in sccs do
        if scc has only incoming or outgoing edges then
            remove scc from sccs
        end if
    end for
    ⟨l,c,r⟩ ← ⟨∅, sccs, ∅⟩
    ⟨l,c,r⟩ ← computeLR(c,cg)                    ▷ Fill with variables not in cut.
    return ⟨l,c,r⟩
end procedure
```

Algorithm 3 Shrinking the Cut Algorithm

```
procedure shrinkCut(⟨l,c,r⟩)               ▷ Returns whether the cut has been shrunk.
    sccs ← computeSCC(c)
    for scc in sccs do
        if scc has only incoming or outgoing edges then
            ⟨$l_2$,$c_2$,$r_2$⟩ ← computeCandidate(⟨l,c,r⟩,scc)
            ⟨l,c,r⟩ ← compareCandidates(⟨l,c,r⟩,⟨$l_2$,$c_2$,$r_2$⟩)
        end if
    end for
    if ⟨l,c,r⟩ has changed then
        return true
    else
        return false
    end if
end procedure
```

We distinguish between three different types of *SCC*s. Side *SCC*s that only have outgoing edges (left side) or only have incoming edges (right side). *SCC*s with incoming and outgoing edges (middle *SCC*s) are the most important ones for our algorithm, as they are connecting left and right side and therefore giving us the possibility to divide the graph. First, the algorithm builds an initial cut (shown in Procedure 2) by separating the middle *SCC*s from the others. Under certain circumstances this can already yield good results, e.g. when only one middle *SCC* is present with only a few variables. But in most cases, the size of the cut is too big and needs to be reduced for our algorithm to work efficiently. In order to remove *SCC*s from the cut, we implemented a downhill algorithm, where in every step one *SCC* is shifted from the cut to left or right. But foremost, the *SCC*s of the cut must be identified that are candidates to remove from the cut, now keep in mind that in the cut are only the facts that previously had incoming and outgoing edges. Under those circumstance, we can conclude that *SCC*s that no longer have an outgoing edge, only have outgoing connections to right and vice versa. Hence, those *SCC*s could be shifted to their respective side and are therefore candidates. Next, every candidate must now be evaluated and compared to each other. The condition we implemented to decide which candidate is better, is influenced by the consequential running complexity. The complexity O of a given cut is

$$\begin{aligned} 0(\langle left, cut, right\rangle) &= 2^C * 2^{|left|+|cut|} + 2^C * 2^{|right|+|cut|} \\ &= 2^{C+|left|+|cut|} + 2^{C+|right|+|cut|}, \end{aligned} \tag{3.1}$$

where C are the number of facts in the cut that are neither goal nor initial. For comparison, the worst case complexity O of an informed search algorithm is

$$O(n) = 2^n, \tag{3.2}$$

where n is the number of facts. In order to achieve a balanced right left proportion, we take the maximum from both sites. A good balance is favorable due to the complexity of solving a task growing exponentially. However having a small cut is even more important for this reason we added a weighting factor $w(= 0.5)$. This factor determines whether the algorithm favors a balanced right and left or rather over a smaller cut. Accordingly, we modified the complexity resulting in the following formula:

$$E(\langle left, cut, right\rangle) = max(C + w * (|left| + |cut|), C + w * (|right| + |cut|)) \tag{3.3}$$

For every candidate, this effort value E is calculated and the algorithm settles on the candidate with the lowest number. The algorithm repeats this process until the cut cannot be improved anymore. Due to the final cut, the algorithm makes an estimate on how time consuming solving the sub-problems are going to be and thus is able to decide whether it should interrupt at this point or continue.

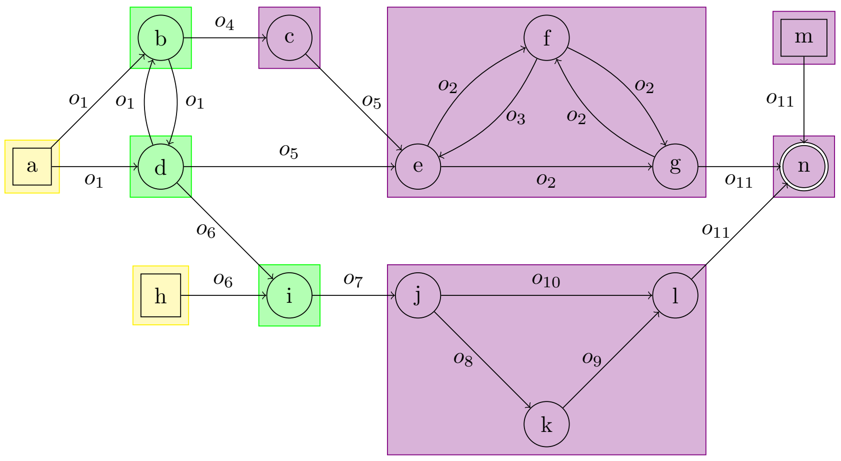

Figure 3.2: Causal graph with the SCCs highlighted to display the three sub-graphs: left(orange), middle(green), right(purple).

3.2.1 Decomposition Example

To illustrate the procedure explained above, we would like to make an example: Given the planning task $\Pi = \langle F, O, I, G\rangle$ described in Section 2.4 and the SCCs as shown in Figure 2.2 we define our initial cut as being all SCCs with incoming and outgoing edges:

$cut = \{\{b,d\},\{c\},\{i\},\{e,f,g\},\{j,k,l\}\}$. The next step is to shrink the cut. First, we determine the *SCC*s in the cut, which lost one connection and now only have either incoming or outgoing connections: $candidates = \{\{b,d\},\{e,f,g\},\{j,k,l\}\}$. Second, we compute the resulting cut for every candidate and rate it.

cut	$E(\langle left, cut, right\rangle)$
cut	16
$cut \setminus \{b,d\}$	14
$cut \setminus \{e,f,g\}$	13
$cut \setminus \{j,k,l\}$	13

As can be seen, comparing the candidates end in a tie. Assuming the algorithm chooses $\{e,f,g\}$ as the best candidate we end up with the following: $cut = \{\{b,d\},\{c\},\{i\},\{j,k,l\}\}$. Determining the candidates and removing the best concludes in one iteration, which is performed until no other *SCC*s can be removed without creating a direct connection from left to right.

3.3 Solving the Planning Problem

At last, the flow-cut algorithm acquired all the pieces needed to generate the planning tasks and consequently solving the planning problem as shown in the procedure 4. For all possible sub-graphs of the cut, excluding facts that are goal or initial, it creates two sub planning tasks Π_1 and Π_2. This distinction is needed, as it is possible that not all facts in the cut must comply or can be achieved, therefore imposing the effort of solving all the different sub-graphs. For the first task Π_1, it merges one sub-graph with left and for the second Π_2 it merges with right, resulting in two complementing tasks, which together yield the solution for our main problem when solved. The sub-tasks are solved with the $A^\star$ search algorithm deploying landmark cut heuristic [7][Helmert and Domshlak], producing different outcome for each task pair. The idea behind *landmark* cut heuristic is to find specific *landmark* facts that always have to be fulfilled to reach the goal state. These landmarks can then be used to estimate how advanced the current state is, by counting how many *landmark* facts already are achieved. *Landmark-cut* heuristic is admissible, hence can be used with the A^* algorithm to find the optimal solution. Last the lowest solution cost that was calculated must be found, as the solution with the lowest cost is the optimal solution for the search problem.

3.3.1 Solving Example

Given the planning task described in Section 2.4 and the cut as shown in Figure 3.2, we show how the algorithm in 4 works. The cut consists of following facts: $cut = \{b,d,i\}$, these are not initial nor goal facts. Since we do not know if all facts in the cut have or even can be achieved, we have to create planning tasks with every possible combination of the cut, shown in the following table:

Algorithm 4 createSolveProblems

```
$minCost \leftarrow \infty$
for every fact subset of cut do                    ▷ Only facts that are not initial or goal
    $leftSubP \leftarrow$ CREATESUBPROBLEM(subset,l,P)
    $rightSubP \leftarrow$ CREATESUBPROBLEM(subset,r,P)
    $cost \leftarrow$ SOLVEPROBLEMS(leftSubP,rightSubP)          ▷ Cost of plan(P)
    if $minCost > cost$ then
        $minCost \leftarrow cost$
    end if
end for
return $minCost$
```

cut	$l = \{a, b\}$	$r = \{c, e, f, g, j, k, l, n, m\}$
$\emptyset$	l	r
$\{b\}$	$l \cap \{b\}$	$r \cap \{b\}$
$\{d\}$	$l \cap \{d\}$	$r \cap \{d\}$
$\{i\}$	$l \cap \{i\}$	$r \cap \{i\}$
$\{b, d\}$	$l \cap \{b, d\}$	$r \cap \{b, d\}$
$\{b, i\}$	$l \cap \{b, i\}$	$r \cap \{b, i\}$
$\{d, i\}$	$l \cap \{d, i\}$	$r \cap \{d, i\}$
$\{b, d, i\}$	$l \cap \{b, d, i\}$	$r \cap \{b, d, i\}$

Every set of variables shown above represents one planning task that must be built, with the corresponding ones on the same row. Two planning tasks are corresponding when the left task provides all initial variables for the right task, by having them as goal variables. These tasks can now be solved and the lowest solution sum of all corresponding task is the h^+ value of the original problem.

4 Experiment Evaluation

The experiments were setup to evaluate the flow-cut algorithm on different planning domains in a state-of-the-art planning environment. The objective of the experiments was to test how the algorithm performs on different existing domains, thereby figuring out its strengths and weaknesses and how it could be improved in a further step.

4.1 Environment

This section specifies the setup for the experiments as well as the environment on which they were executed.
The flow-cut algorithm is implemented in C++ and integrated in the state-of-the-art planning system Fast Downward, which was introduced by Helmert[6] in 2006. Several heuristics are already implemented in Fast Downward and can be used to solve the sub-tasks Π_1 and Π_2. We use the $A^\star$ [8] algorithm with landmark-cut heuristic for these sub-tasks. To test all domains provided by the International Planning Competition (IPC) benchmarks, we make use of the computer cluster of the University of Basel, which has Intel Xeon E5-2660 CPUs running at 2.2 GHz. The domains of the benchmarks are from the IPC, however we excluded all domains with axioms. Furthermore, it is necessary for the flow-cut algorithm to solve many different tasks successively, therefore we wrote a python scripts that handles the different Fast Downward calls. To run the script on the cluster we utilized the lab framework. Finally, every problem had a computation time limit of 60 minutes, a RAM limit of 2 GB and the results were captured and summarized in the following sections.

4.2 Evaluation

In this section we will present and discuss the results of the experiments. We grouped the results in three different types. Every type will be discussed in its own section and summarized in the section 4.3. Type A are domains with at least one problem that has a cut smaller than 20. Type B domains have a cut bigger than 20 and were therefore not be solved as we implemented a hard limit to reduce computation time. At last, type C domains have a specific structure that does prevent us from making a cut.

The result tables in this section contain following categories:

- The **cut size** is the number of facts in the cut, inclusive initial facts and goal facts.
- The **left** and **right size** are the number of facts in their respective sub-problem, e.g. left size consists of all facts from the left part and middle part.
- **Solvable problems** are the number of problems that could be solved in the domain by the flow-cut algorithm.
- **Single *SCC*** are *SCC*s without any connection to the rest of the problem.
- **Left, right** and **middle *SCC*s** have incoming connections, outgoing connection or both.

4.2.1 Type A: Solvable

The eight domains shown in Table 4.1 have at least one problem that could successfully be solved by the flow-cut algorithm. In this section we will go through every one of these eight domains and analyze their structure.

Domain	Left-Size	Cut-Size	Right-Size	Solvable Problems
miconic	3-107	2-44	4-87	~50%
mystery	23-305	19-285	39-1581	1
no-mystery	23-305	19-285	39-1581	1
pathways	42-208	6-66	15-452	4
pathways-noneg	42-208	6-66	15-452	4
rovers	16-2684	6-14	6-1580	6
satellite	8-1753	4-603	10-762	5
trucks-strips	10-56	9-55	37-607	10

Table 4.1: All IPC domains that have at least one problem that can be solved with the flow-cut algorithm combined with the size of the different parts and how many problems can be solved.

Domain	Single *SCC*	Left Side *SCC*	Middle *SCC*	Right Side *SCC*
miconic	1-30	1	1-30	1-30
mystery	1-2	4-20	1-3	1
no-mystery	1-2	4-20	1-3	1
pathways	5-138	13-60	9-77	1-41
pathways-noneg	5-138	13-60	9-77	1-41
rovers	4-230	3-125	3-44	3-69
satellite	3-175	2-30	1-31	3-175
trucks-strips	3-21	1	2	5-20

Table 4.2: All IPC domains that have at least one problem that can be solved with the flow-cut algorithm combined with their *SCC*strucute.

4.2.1.1 Miconic

As shown in the table 4.1, all problems in the miconic domain have a constant left side *SCC* size of one. Even though this suggest that the left part is of constant size as well, it is not the case here. All parts are evenly balanced and give a good ground for the flow-cut algorithm. By comparing the parts over all problem, we came to the conclusion, that the miconic domain is a good example how a domain should behave to employ the flow-cut algorithm.

miconic	Left-Size	Cut-Size	Right-Size	h+
s1-0.pddl	3	2	4	3
s2-0.pddl	5	3	8	7
s3-0.pddl	13	4	6	10
s4-0.pddl	14	7	13	14
s5-0.pddl	16	7	16	17

Table 4.3: A few solvable planning problems from the miconic domain.

4.2.1.2 Mystery & No-Mystery

Analyzing the data to the mystery and no-mystery domain has shown, that even though some problems can be solved, the domains itselves are not suitable. Due to the fact, that the left part without the cut variables is always very small. Therefore the right part holds nearly all variables of the problem and as a consequence, the flow-cut algorithm loses all advantage, as the sub problems that are generated are as big as the original problem.

mystery	Left-Size	Cut-Size	Right-Size	h+
prob25.pddl	23	19	39	4

Table 4.4: The only planning problem from the mystery domain that has a cut size smaller than 20.

4.2.1.3 Pathways & Pathways-Noneg

Both pathways and pathways-noneg domain have the exact same results, hence can be discussed together. These domains produced the best results we encountered in our experiments. All *SCC* sizes are growing at a similar rate, resulting in balanced left and right parts. Even though the right part is roughly twice as big as the left part, it still is a good result due to the cut size being smaller than both.

pathways	Left-Size	Cut-Size	Right-Size	h+
p01.pddl	42	6	15	6
p02.pddl	39	11	37	12
p03.pddl	54	12	56	16
p04.pddl	67	19	76	timeout

Table 4.5: All planning problems from the pathways domain that have a cut size smaller than 20.

4.2.1.4 Rovers

The rover domain looks on the first glance similar to the pathways domain but after taking a closer look, it is visible that the right part consists only of variables from the cut. Leaving us in the same situation as the mystery domain.

4.2.1.5 Satellite

Examining the satellite domain shows a similar result to the pathways domain. The only difference being that the cut size scales worse in the satellite domain. It begins good with the cut being half as big as the other parts but soon the cut size converges on the others. An adjusted algorithm that considers the specific structure of this domain could possibly reduce the cut size and produce better results.

4.2.1.6 Trucks-Strips

Even though a few problem of this domain can be solved, the structure itself is not made for such an approach. Left side *SCC*s as well as middle *SCC*s are constant over all problems. Ultimately causing the right part to grow much bigger than the others. Making the flow-cut algorithm inefficient to use.

trucks-strips	Left-Size	Cut-Size	Right-Size	h+
p01.pddl	10	9	37	11
p03.pddl	13	12	54	17
p05.pddl	16	15	72	23
p07.pddl	13	12	78	20
p09.pddl	17	16	98	26

Table 4.6: A few solvable planning problems from the trucks-stips domain.

4.2.2 Type B: Big Cuts

The six domains shown in the Table 4.7 have all too big cuts to be solved efficiently. Even though the cuts do look promising, if you examine the *SCC*s it will be clear why the flow-cut algorithm is not efficient in solving these problems. The *SCC*s in the table mentioned before are split into four groups. Single *SCC*s are not connected with the rest of the problem, thus can be solved individually. Left side *SCC*s have only a outgoing connection and right side *SCC*s vice versa. The most important *SCC* type for our algorithm are middle *SCC*s, as we try to split the graph by removing these. Now, the problem with the domains below is not that they have no middle *SCC*, but that they have just one that nearly holds all facts of the problem. Leaving us no option to reduce the cut size as we have a strict structure to maintain.

A way to make the flow-cut more compatible with these domains, could be to allow connections from the right part to the middle part and middle part to left part. This would enable an algorithm to even break down *SCC*s. Yet this is not a trivial change, as one would need to handle that a solution plan could possibly apply alternating actions from both sides.

Domain	Left-Size	Cut-Size	Right-Size
childsnack-opt14-strips	72-192	50-139	64-174
nomystery-opt11-strips	30-264	29-163	44-356
parcprinter-08-strips	28-337	24-286	39-316
parcprinter-opt11-strips	55-255	44-204	50-284
tidybot-opt11-strips	277-763	269-757	269-759
tidybot-opt14-strips	277-763	269-757	269-759

Table 4.7: All domains where the cut is strict bigger than 20, thereby being too big to be useful.

Domain	Single *SCC*	Left Side *SCC*	Middle *SCC*	Right Side *SCC*
childsnack-opt14-strips	14-35	22-53	17-61	1
nomystery-opt11-strips	0	1	1	3-12
parcprinter-08-strips	9-35	4-51	3-70	2-11, 1
parcprinter-opt11-strips	11-33	11-46	6-70	3-11, 1
tidybot-opt11-strips	0	4	1	4
tidybot-opt14-strips	0	4	1	4

Table 4.8: The *SCC*structure of the domains with a cut that is bigger than 20.

4.2.3 Type C: No Cut

All domains shown in the Table 4.9 cannot be solved by the flow-cut algorithm due to them having no *SCC*s with incoming and outgoing edges. This highlights the main problem with the flow-cut algorithm, although the algorithm works, most domains inherent structure do not comply with the conditions we defined for our algorithm. Yet with the information from the *SCC* structure, we could introduce a new algorithm that is more adjusted to these domains.

4.3 Evaluation Summary

To summarize, we tested the flow-cut algorithm on 79 domains, resulting in six domains that successfully could be solved. With the best domains being miconic, pathways, pathways-noneg and satellite, showing that the flow-cut algorithm can successfully be used to compute h^+. The next step from here on, would be to make the other domains accessible. We identified the main problem for our algorithm, as being the incompatibility of our algorithm with the inherent structure of the domains. The issue being that most domains have no middle *SCC*s. This leads to the conclusion that cutting a planning problem needs to be more elaborate and adjusted to the problem it is supposed to work on, if it should be more efficient than state-of-the-art algorithms.

Domain	Single *SCC*	Middle *SCC*	Right Side *SCC*	Left Side *SCC*
airport	0	0	1	x
barman-mco14-strips	0	0	1	1
barman-opt11-strips	0	0	1	1
barman-opt14-strips	0	0	1	1
blocks	1	0	0	0
cavediving-14-adl	2	0	1	x
citycar-opt14-adl	x	0	1	x
depot	0	0	1	x
driverlog	0	0	x	1
elevators-opt08-strips	0	0	1	x
elevators-opt11-strips	0	0	1	x
floortile-opt11-strips	0	0	x	y
floortile-opt14-strips	0	0	x	3
freecell	0	0	1	x
ged-opt14-strips	1	0	0	0
grid	0	0	1	x
gripper	0	0	1	1
hiking-agl14-strips	0	0	1	x
hiking-opt14-strips	0	0	1	2
logistics00	0	0	x	y
logistics98	0	0	x	y
maintenance-opt14-adl	x	0	1	y
miconic-simpleadl	x	0	y	1
movie	13	0	0	0
mprime	0	0	1	1
no-mprime	0	0	1	1
openstacks-agl14-strips	0	0	1	x
openstacks-opt08-strips	0	0	1	x
openstacks-opt11-strips	0	0	1	x
openstacks-opt14-strips	0	0	1	x
openstacks-strips	5	0	1	10
parking-opt11-strips	1	0	0	0
parking-opt14-strips	1	0	0	0
pegsol-08-strips	1	0	0	0
pegsol-opt11-strips	1	0	0	0
pipesworld-notankage	1	0	0	0
pipesworld-tankage	1	0	0	0
psr-small	1	0	0	0
scanalyzer-08-strips	x	0	0	0
scanalyzer-opt11-strips	x	0	0	0
schedule	x	0	1	y
sokoban-opt08-strips	1	0	0	0
sokoban-opt11-strips	1	0	0	0
storage	1	0	0	0
tetris-opt14-strips	1	0	0	0
thoughtful-mco14-strips	x	0	1	y
tpp	0	0	x	y
transport-opt08-strips	0	0	1	x
transport-opt11-strips	0	0	1	x
transport-opt14-strips	0	0	1	x
visitall-opt11-strips	x	0	0	0
visitall-opt14-strips	x	0	0	0
woodworking-opt08-strips	x	0	y	z
woodworking-opt11-strips	x	0	y	z
zenotravel	0	0	x	y

Table 4.9: The *SCC* structure of all Domains where no cut could be generated since they have no middle *SCC*s.

5 Conclusion

The goal of this thesis was to implement an algorithm that can compute the h^+ heuristic value in practice. We achieved this by introducing the flow-cut algorithm. The idea is to divide a problem that is unsolvable in practice, into smaller sub problems that can be solved on their own. Additionally, we generated the strongly connected components for all the mentioned domains and thereby provide an indication on which domains splitting the problem can be beneficial to solve it.

To summarize our results shown in Chapter 4, we tested the flow-cut algorithm on 79 domains, resulting in six domains that can be solved successfully, thereby proving that our implementation works and can be used to solve classical planning problems. However, many domains cannot be solved with our approach. We decided to build the cut based on SCCs as it provides a useful foundation. SCCs allow us to enforce a specific structure, however the experiments have shown that most problems have no middle SCC and are therefore not solvable by the flow-cut algorithm. Consequently, we conclude that to achieve better result we would need to revise the decomposition of the planning task and possibly adjust the algorithm for certain domains.

One possible way to adjust the flow-cut algorithm is to change the number of parts the problem is split into. Gnad, Hoffmann and Domshlak have shown in the paper [9] that many classical search problems can lead back into a star like shape. If you combine that knowledge with our approach, you end up with a new algorithm that has many left and right parts connected through one middle part.
A second possibility to adjust the algorithm, could be by removing the condition described in Chapter 3, that all connections have to be directed from the left part to the middle part and from the middle part to the right part. This condition limits us in the creation of the cut, as we need a SCC that has incoming and outgoing connections. Yet, this approach generates a much bigger overhead, because the split parts would not be secluded from each other anymore and therefore require to be handled more complicated.

In conclusion, we have shown that a divide and conquer approach can successfully be applied to solve classical planning problems. Although the flow-cut algorithm did not outperform state-of-the-art search algorithms, it proves that this approach is a viable alternative and is a promising direction to conduct further research.

Bibliography

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

[2] Betz, C. Komplexität und Berechnung der h^{+}-Heuristik. Diplomarbeit, Albert-Ludwigs-Universität Freiburg. (2009).

[3] Bylander, T. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204 (1994).

[4] Hoffmann, J. and Nebel, B. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302 (2001).

[5] Ghallab, M., Nau, D., and Traverso, P. Automated Planning: Theory & Practice. (2004).

[6] Helmert, M. The fast downward planning system. *J.Artif. Intell. Res.(JAIR)*, 26:191–246 (2006).

[7] Helmert, M. and Domshlak, C. Landmarks, critical paths and abstractions: What's the difference anyway? *Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 162–169 (2009).

[8] Hart, P., Nilson, N. J., and Bertram, R. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*, pages 100–107 (1968).

[9] Gnad, D., Hoffmann, J., and Domshlak, C. From Fork Decoupling to Star-Topology Decoupling. *Proceedings of the 8th Annual Symposiium on combinatorial Search (SOCS'15)* (2015).

[10] Betz, C. and Helmert, M. Planning with h+ in theory and practice. In *Annual Conference on Artificial Intelligence*, pages 9–16. Springer (2009).

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

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

Author — Autor
Marvin Buff

Matriculation number — Matrikelnummer
2014-054-191

Title of work — Titel der Arbeit
Solving Delete-Relaxed Planning Tasks by Using Cut Sets

Type of work — Typ der Arbeit
Bachelor 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, 13/06/2017

Marvin Buff

Signature — Unterschrift