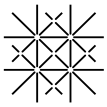

University of Basel

Optimized Computation of the Additive and FF heuristics

Bachelor thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
https://ai.dmi.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Patrick Ferber, MSc.

Cyrill Imahorn
cyrill.imahorn@stud.unibas.ch
2019-051-333

July 20, 2022

Acknowledgments

I would like to thank my supervisor Patrick Ferber, who always had an open ear for questions and provided helpful answers and support. It was a pleasure to work with him. I would also like to thank Prof. Dr. Malte Helemert for making it possible for me to write this bachelor thesis and giving me the chance to work on a topic in which I am genuinely interested. Finally, I would like to thank my family, who has been very supportive both during this thesis and throughout my education so far.

Abstract

Planning tasks are important and difficult problems in computer science. A widely used approach is the use of delete relaxation heuristics to which the additive and FF heuristic belong. Those two heuristics use a graph in their calculation, which only has to be constructed once for a planning task but then can be used repeatedly. To solve such a problem efficiently it is important that the calculation of the heuristics are fast. In this thesis the idea to achieve a faster calculation is to combine redundant parts of the graph when building it to reduce the number of edges and therefore speed up the calculation. Here the reduction of the redundancies is done for each action within a planning task individually, but further ideas to simplify over all actions are also discussed.

Table of Contents

1 Introduction

Planning problems are important and difficult problem in computer science and its application. For example a famous problem is the game Go, which, after there were computers better in Chess than the best humans, became the next big milestone. And it was only recent, where a computer beat the best human player with AlphaGo [8], which used a combination of planning techniques and neural networks to play the game.
A key component in many classical planning algorithms are heuristic searches, where a heuristic solves a simplified version of the problem to estimate how expedient an action is. There are many different approaches for such heuristics with different strengths. One important class of heuristics are the delete relaxation heuristics, in which the additive and FF heuristics are included.
Since a heuristic value can be calculated millions of times for even small planning tasks, a fast computation of the heuristic values is crucial for the run time of heuristic based searches like the Greedy Best-First-Search. This means that even milliseconds can have a significant impact on the total computation time.

This bachelor thesis focus one exactly this in the classical planning is Fast Downward [2], which is a domain-independent classical planning tool, which is to most used tool for classical planning tasks.
The additive and FF heuristic use a graph, which is repeatedly updated to calculate the heuristic values. This graph has to be constructed only once for a planning task and be used to calculate the heuristic value of all possible actions in the task, by just resetting the values of the nodes in the graph and assign different starting values depending on the state of the search. This means the construction of the graph is allowed to be a bit more computational intensive, if the updating of the graph becomes less intensive, which is the idea behind this thesis. The goal is to achieve a faster calculation of the heuristic values by reducing number of edges within the graph, because they are directly proportional to how many updates are needed for a change of a value of one node. The basic idea is to reduce redundant parts of the graph by combining them into fewer nodes.

2 Background

This chapter provides the necessary understanding of planning tasks and how to solve them.

2.1 Planning Task

We work with planning tasks in nite-domain representation(FDR)[3]. Therefore we de ne a planning task like the following:

De nition 1. A planning task is a 4-tuple = hV; s_I; O; i where

- V is a nite set of variables each with a nite domain dom(V)
- s_I is the initial state, which is a total assignment overV,
- O is a nite set of operators, and
- is the goal, which is a partial assignment of the variables v 2 V.

In our case we only consider cases where all variables inV have a nite domain.

A planning task formally describes di erent states, the transitions between them, the initial state as well as the goal states.

A state s is described by an unique assignment of all the variables inV. For example if a planning task has two variablesV = f v_1; v_2g over the domains dom(v_i) = f T; F g, then there are four possible states:f v_1 7! T; v_2 7! T g , f v_1 7! T; v_2 7! F g, f v_1 7! F; v_2 7! T g and f v_1 7! F; v_2 7! F g. The size of a setS containing all possible states of a planning task can be estimated byjSj k^|V|, where k is the domain size of the variable with the biggest domain.

The possible transitions between di erent states are de ned by operators. Operators have three aspects: They have a precondition, which are partial assignments of the state variables which must be identical to the current state for the operator to be applicable. They have an e ect, which says how some state variables change and they have a cost, which says how expansive this operator is. An e ect can also have additional conditions, where the e ect is

called "conditional e ect" and the additional conditions of such an e ect are called "e ect conditions". If an operator is used, all of its e ects without additional conditions are applied and conditional e ects are only applied if their e ect conditions are satis ed. Lets de ne an operator a bit more formal:

De nition 2. An operator o has three properties:

- Precondition pre(o): A partial assignment a state must have foro to be applicable.
- E ect e (o): How o e ects the assignment of the variablesV and
- Cost cost(o): How expensive it is to applyo, where cost(o) 0.

The applied e ect changes the assignment of the state variables therefore the assignment represents a new state. This means that the operators describe transitions between states. For all operators o_i, which are applicable at a state$s$, there is a new states_i^0 which are the new variable assignments wheno_i is applied to s. This states s_i^0 are called the successor states of$s$.

Planning tasks allow us to represent a lot of complex problems. The goal is to nd a plan = $\langle o_1; :::; o_n \rangle$ which is a sequence of operators, which lead from the initial states_I to a state s_G that ful lls the requirements of . It is also possible to have additional requirements to a plan $_i$ for example a requirement could be, that the plan should have a small cost, where the cost of a plan $_i$ is de ned as cost($_i$) = $\sum_{o_i 2 \ _i} \text{cost}(o_i)$.
One possible approach to solve a planning task is to try to estimate for each successor state s^0 of a current state how good they are, respectively how much it costs to get from those states to get to a goal state. This leads us to heuristics.

2.2 Heuristic

A heuristic h is a function used to estimate how much it costs to get from a state$s$ to a goal state by solving an approximation of the original problem. This meansh maps a state to a real positive number: $h : S \mapsto R_0^+ [1$. Therefore a heuristic can be used to estimate how well suited a state is to get to any goal state compared to other states [6]. If$h(s) = 1$ the heuristic estimates that no goal state is reachable from the state$s$.
There are di erent possible ways to implement a heuristic but generally they simplify the problem by reducing the complexity. One way to achieve this is with a delete relaxation, this is explained further in the following section.
There are four main characteristics look for in a heuristic:

- Safe: A safe heuristic h does never estimate a state$s$ to not have a path to a goal state, if a path exists. This guarantees, that no state is estimated to be a dead end, if it is not.
- Goal-Aware : A heuristic h is goal-aware, if it estimates the cost to reach the nearest goal to be zero, for all states that are goal states.

- Admissible : A heuristic is admissible, if its estimation is always smaller or equal to the true minimal cost to reach the nearest goal from a given state. Therefore h never overestimates the true minimal cost.
- Consistent : A heuristic h is consistent if its estimation for a state s is less or equal than the its estimation for all successor states of s plus the cost to reach the successor state.

2.3 Delete Relaxation Heuristics

A delete relaxation heuristic approximates a problem by assuming that a variable not only has it current assignment but also all assignments it has had. This means that an operator never leads to a state, that is further away from the goal state. It can only bring us nearer or has neither positive nor negative effect besides the cost.
The neat thing about this, is we can solve such a relaxed problem with a directed graph $G = \langle V; E \rangle$ with vertices V and edges E, which can be solved in polynomial time [5]. For each variable $v \in V$, there is a node n_v and for each operator $o \in O$ node n_o. A node n_o of an operator has an edge from all effects of the operator to n_o and has an edge from n_o to all variable nodes n_v with $v \in vars(pre(o))$, where vars maps a partial assignment! to the variables involved in the !. Additionally, there is a node n_I for the initial state s_I, which has incoming edges from all variable nodes n_v with $v \in s_I$ as well as an goal node n_*, which has an outgoing edge to all variable nodes n_v that need to be true for the goal to be achieved.
Now it can be tested whether such a relaxed problem is solvable or not with a few simple rules.

- A variable node becomes true, if any of its outgoing edges goes to a true node.
- A operator node becomes true, if all of its outgoing edges go to true nodes.
- The initial node is always true and all other nodes start with being false.
- The goal node becomes true, if all nodes at its outgoing edges are true.

By applying this rules repeatedly to such a graph, the graph eventually does not change anymore. If at this point the goal node is true, the problem is solvable, else it is not solvable. With a few changes, this can also be used to estimate the cost to reach the goal node:

- Instead of using true or false for the value of the nodes, we use integers. If the integer is equal to -1 the node is false and if the integer is greater or equal to 0 the node is true.
- The initial node has value 0 and all others start with value -1.
- Variable nodes change there value to the smallest value of nodes of its outgoing edges that are greater than -1, unless their value smaller than that smallest value, but bigger than -1.

- Operator nodes have value -1 if any of the nodes at their outgoing edges has value 1. If all those values are greater than 1 it uses some functionu over all those values and adds its cost to it.
- The goal node also has value 1 if any of the nodes at its outgoing has value 1. Else it has the value that is given by a function u over all those values.

With this rules the graph eventually also stops changing and the value of the goal node represents the estimated cost to achieve the goal state from the initial state. If the value is equal to 1 the goal state is not reachable.
Regarding the function u used in the operator node and the goal node, there are to commonly used alternatives. If u gives us the maximum value of its inputs it is the so called maximum heuristic h^{max}, which is admissible and consistent but it generally vastly under estimates the true cost of the relaxed problem. Ifu returns the sum of all its inputs it is the Additive heuristic h^{add} which is neither admissible nor consistent but generally leads to better estimates [1]. Also $h^{add}(s)$ results are always bigger or equal to the optimal solution cost of the relaxed problem.
Lets make an example:
A planning task $= \langle V; s_I; O; \rangle$, with the variables $V = \{P_1; P_2; P_3; P_4\}$ with the domains $dom(v) = \{T; F\}$; for all $v \in V$, the initial state $s_I = \{P_1 \mapsto T; P_2 \mapsto T; P_3 \mapsto F; P_4 \mapsto F\}$, the operators $O = \{o_1; o_2\}$, where $cost(o_1) = 2$, $e(o_1) = \{P_2 \mapsto T; P_3 \mapsto T\}$, $pre(o_1) = \{P_1 \mapsto T\}$, $cost(o_2) = 1$, $e(o_2) = \{P_4 \mapsto T\}$ and $pre(o_2) = \{P_3 \mapsto T\}$ and $= \{P_2 \mapsto T; P_3 \mapsto T; P_4 \mapsto T\}$. The graph for the heuristic of the initial state can be seen in gure 2.1(a). Once the graph looks like gure 2.1(b), it will not change anymore. As no can easily be seen the Additive heuristic h^{add} estimates $h^{add}(s_I) = 5$ for the relaxed task.

(a) Initial graph (b) Fixpoint of graph

Figure 2.1: The initial and nal state of the graph for the Additive heuristic for the described planning task. The variables are round nodes and the operators and the nodes for the goal and for the initial state are square nodes.

Since h^{add} generally overestimates the cost by quite a lot, there is a post processing which reduces the estimate by only taking the sum of all operator costs directly involved in making the goal true. This can be done by keeping track of which operator set the value of each variable node. At the end it is possible to backtrack from the goal state to the initial state over all operator used in the transitions. This heuristic is called FF heuristic h^{FF} [4] and is also neither admissible nor consistent and isn't even deterministic, since the order of the updates in the graph is not de ned. But it is generally much closer to the optimal solution cost of the relaxed problem than the result of h^{add}.
In this thesis only the Additive and FF heuristic are looked at.

2.4 Operator Domination

The Fast Downward implementation introduces operator domination [2]. An operator o_1 is dominated by an other operator o_2 if:

1. e (o_1) = e (o_2), and
2. pre(o_2) pre(o_1), and
3. cost(o_2) cost(o_1), and either
4. a) 2. or 3. is strict, or
   b) id(o_2) < id(o_1), with id(o_i) being a index of the operator o_i in a vector holding a references to all operators.

This means, that if o_1 can be applied also o_2 can be applied and it is generally a better option, since it has a smaller cost or a smaller set of preconditions or both in case of 4a, in case of 4b the operator o_1 is not actually dominated, but it is equivalent to the operator o_2. In both cases it is su cient to look at the dominating operator and the dominated operator can be removed. This allows us to reduce the number of updates that have to be done, without losing any precision of the calculation at least in a delete relaxation heuristic. But this means for all operator nodes it has to be checked if they are dominated.

2.5 Greedy Best-First-Search

A simple but quite powerful algorithm to nd a solution for a planning task is the Greedy Best-First-Search (GBFS) [6]. GBFS uses a heuristic h to estimate how much it costs to get from the given initial state s_I to the closest goal state. From a given state s it calculates the heuristic value $h(s')$ for all successor states s' from s and stores those states together with the operator used to reach it, as well as s in a priority queue open sorted by their heuristic value and stores s_i in a List closed to indicate, that all s' have been added to open. It then repeatedly pops the element s_{min} with the smallest heuristic value, inserts all reachable states, which are directly reachable from s_{min} and are not already in closed, to open and adds s_{min} to closed until either a newly reached state is a goal state (rendering the problem solved) or there are no more states in open (rendering the problem unsolvable). The algorithm is shown in gure 2.2.

```
Greedy Best-First-Search:
open = PriorityQueue ordered by h
if h(init()) < ∞:
    open.insert(<h(init()), init(), None>)
closed = HashSet of states
while open is not empty():
    <h, state, op> = open.pop()
    if state is not in closed:
        closed.insert(state)
        if state is goal:
            return path to state
        for each <state', op> in succ(state):
            if h(state') < ∞:
                open.insert(<h(state'), state', op>)
return unsolvable
```

Figure 2.2: Pseudocode of the Greedy Best-First-Search algorithm. init() returns the initial state s_I of the planning task and succ(s) returns all pairs of successor states of a given state s and the operator used to get to the the successor state from s.

GBFS is often very fast and if the heuristic used is safe it is even complete, meaning that it always terminates and if a solution exists it will always find one. But GBFS is suboptimal which means the found solution must not necessarily be the best possible solution and can even be arbitrarily bad.

3
Implementation

In this chapter the goal is to show, what changes were made in the implementation and where there were problems and how they were solved.

3.1 Status Quo

In the original implementation of Fast Downward there is a straight forward approach to generating the graph for solving the relaxed problem. It splits the operator nodes into multiple similar nodes. One for each e ect of the operator as a C++ class called UnaryOperator. This has some signi cant advantages: the overhead of building the graph is small and conditional e ects can just have additional edges from its operator node to the e ect conditions. But it also has some major disadvantages. If a variable node (implemented as a C++ class called Proposition) updates its value, not only has there to be a check for all operators which depend on that variable, but all e ects that depend on that variable. This is not a big deal, if every operator has on average one e ect, but the number of nodes, that have to check if they have to update their value on average is directly proportional to the average number of e ects per operator. This promises better run times if we can combine such nodes. For example let there be v state variables and o operators with an average number of e e ects and an average number of p preconditions. With this implementation this means that every time a Proposition updates its value on average $\frac{p \cdot o \cdot e}{v}$ UnaryOperator have to be update with the new Propositions value and for each e ected UnaryOperator it has to be checked, if they now have all there preconditions ful lled and if so update their e ects of them.

For example lets look at an operator O with $pre(O) = \{P_1\}$ and the e ects $e(O) = \{e_1; e_2; e_3\}$ with $e_1 = \{P_3\}$, $e_2 = \{P_4\}$ and $e_3 = \{P_5\}$; with e ect condition P_2. The graph generated for this operator (with other elements of the graph ignored) would look like the gure 3.1.

The removal of dominated e ects is done with method simplify. Regarding the simplifying of dominated e ects the original implementation removes dominated e ects individually, by just looking at each UnaryOperator individually. It Iterates over all UnaryOperators adding them to a hashmap and uses the preconditions as well as the e ect as the key and

Figure 3.1: Graph of an operator with two effect and one conditional effect, which would be generated by the original implementation. The state variables are represented by circles and the UnaryOperators by squares.

the cost and the index of the UnaryOperator as the value. If there already exists an element in the hashmap with that key, it checks if the cost is smaller than the cost of the already inserted element and if so it replaces the value with the cost and the UnaryOperator index of the cheaper. After this first iteration it iterates a second time over the UnaryOperator and first checks, if the entry in the hashmap with the key generated from this operator has the same operator index, if not, this means there is an other UnaryOperator which has the same precondition but cheaper cost or equal cost but smaller index, so the operator is dominated and can be removed. If it is not dominated at this point, it checks for all subsets of the UnaryOperator in the same way, this reveals if there is a dominating operator with a smaller subset of the preconditions. But since the number of subsets grows exponentially to the elements of the set, this check is only done, for precondition sets with at most 5 elements. The removal of dominated effects can have an effect on the result of the FF heuristic, since a dominated UnaryOperator o_1 can be applied before its dominating UnaryOperator o_2 and stay in the result when $cost(o_1) = cost(o_2)$, leading to a different backtracking path of used operators. Also since we remove effects that are not dominated but equivalent with a higher index and this index is not strictly defined this can lead to different results depending on the implementation. This simplifying has only to be done once for each planning task, since the operators and whether they are dominated or not does not depend on a state, but on the task.

In the original implementation a data structure ArrayPools is used, which is basically a long vector. The idea is, that an UnaryOperator does not have to store its preconditions itself, but it holds only two pieces of information about it, the rest is stored in the ArrayPool. The UnaryOperator only knows, at which index of the ArrayPool it has its first entry and how many elements in the ArrayPool it has. So instead of each UnaryOperator having its own vector with its preconditions, the ArrayPool holds this data as one segment in its vector. This allows all UnaryOperators to have the same size, because the data that can be

of different size is outsourced. It also seems to have a positive impact on the memory, but to that later more (Section 3.7). The ArrayPools are also used for the Propositions to safe the UnaryOperators dependent on the Proposition.

3.2 Optimization Idea

A relatively easy attempt to make such an optimization is to bundle all effects of an operator except of the conditional effects in the same node n_{Op}. Conditional effects can treat n_{Op} as a variable node, since the function u used in an operator node can be applied transitively for h^{add} (as well as for h^{max}) if the cost is added separately: Let A_o be the set of preconditions of an operator o ($A_o = pre(o)$) and B_o be the set of effect conditions of an effect of o. With $\hat{} = A_o \setminus B_o$ and $\hat{B_o} = B_o \cap \hat{}$, than:

$$cost(o) + \sum_{v \in (A_o \cup B_o)} n_v.value = cost(o) + \sum_{v \in A_o} n_v.value + \sum_{v \in \hat{B_o}} n_v.value$$

This reduces the number of updates needed, when a variable node updates.

For example let there be v state variables and o operators with an average number of e effects without effect conditions and an average number of p preconditions. Also let e_c be the average number of conditional effects with p_c preconditions on average. With the optimization idea the average number of OperatorNodes that have to be updated for every Proposition that updates is $\frac{p \cdot o \cdot (1 + e_c)}{v} + \frac{p_c \cdot e_c \cdot o}{v}$. Which seems bigger than the number of updates described in section 3.1 and it can be depending on the planning task. But often the number of conditional effects e_c is very small or even zero, but the number of effect e can be big. The number of updates under the assumption that e_c is zero would be $\frac{p \cdot o}{v}$ which then would be significantly smaller than $\frac{p \cdot o \cdot e}{v}$.

For example the same operator as described in section 3.1 for an operator O with $pre(O) = \{P_1\}$ and the effects $e(O) = \{e_1, e_2, e_3\}$ with $e_1 = \{P_3\}$, $e_2 = \{P_4\}$ and $e_3 = \{P_5\}$, with effect condition P_2. The graph generated for this operator (with other elements of the graph ignored) would look like the figure 3.2.

As can be seen the optimization idea generally reduces the number of nodes and edges in the graph.

An even more advanced idea would be to search for common subsets of preconditions and use them likewise as intermediate nodes with no direct effects them self. This could further reduce the number of edges in the graph and therefore the number of updates significantly. Here is a small example: an operator O_1 with $pre(O_1) = \{P_1, P_2, P_3, P_4\}$ and the effect $e(O_1) = \{P_5\}$ and an operator O_2 with the $pre(O_2) = \{P_2, P_3, P_4\}$ and the effect $e(O_2) = \{P_6\}$. These two operators have an overlapping in there precondition $pre(O_1) \cap pre(O_2) = \{P_2, P_3, P_4\}$, therefor the graph can be simplified. As shown in figure 3.3.

The reduction of the edges of the graph can increase rapidly with more operators and promises even further speed up in the computation. This however is not part of this thesis and remains as future work.

Figure 3.2: Graph of an operator with two e ect and one conditional e ect, which would be generated by an implementation with the optimization idea. The state variables are represented by circles and the UnaryOperators by squares.

(a) with current implementation (b) further optimization

Figure 3.3: The graphs of two operators with overlapping preconditions. In (a) it is shown as the graph would look like in the original implementation as well as with the adaptation of this thesis. In (b) the graph is build with the discussed further reduction of edges.

3.3 Problem 1: Graph nodes

With the original implementation of UnaryOperators and Propositions the graph was simple, every Proposition knew about which UnaryOperators were dependent on it. And every UnaryOperator knew which Proposition it changes as well as all propositions, it had as preconditions (This part is relevant for the FF backtracking).
For the optimization idea, this was not ideal, since now an operator node could have Propositions as well as an operator as precondition and it could have not only multiple e ects, but also multiple conditional e ects depending on it. Therefore it would be more useful

to have a class GraphNode with subclasses PropositionNode and OperatorNode. With this the preconditions of an OperatorNode can just be a GraphNode and it must not care about whether its an OperatorNode or a PropositionNode. But this lead to the first problem, which took a while to identify:
The preconditions of a OperatorNodes are saved in a vector which holds a pointer to each GraphNode. The Problem was, that the OperatorNodes were also saved in a vector, which grew while all the OperatorNodes were build. But when that vector grew over its allocated size, it had to allocate a new and bigger space in the memory and move all its entries there. This rendered all pointers in the preconditions vector as useless, since the pointers in it now pointed somewhere in the memory, where the GraphNodes used to be, but not anymore. As solution the OperatorNodes were not longer saved in a vector, but randomly allocated with the C++ command new. This meant that the pointers remind valid, regardless of how many OperatorNodes were added. To still be able to iterate over all OperatorNodes as well as releasing the allocated memory at the end, the vector which originally held the UnaryOperators was used to now hold pointers to the OperatorNodes.
The updating of the nodes is implemented as a recursive function of those nodes, but the problem with that is that the ArrayPool can not be accessed directly from the nodes, since the ArrayPools are object variables of the class RelaxationHeuristic, which is independent of the GraphNodes, and the effect was no longer just one Proposition index, but could be multiple pointers to propositions and other operators. So the ArrayPools were ignored for the moment and the preconditions and effects were given to the nodes as vectors. But this has a major effect on the calculation speed as discussed later (Section 3.7).

3.4 Problem 2: To high costs in conditional effects

In the original implementation the cost of an operator cost(o) was just used as the default cost value of the UnaryOperator. And after each updated precondition the cost value of that precondition was added to the cost value of the UnaryOperator until all preconditions were updated. Then it used this sum of all precondition cost values as well as cost(o) to update the effect of the precondition.
In my implementation I first naively used the same approach. But this lead to wrong results. The problem being conditional effects. Conditional effects nodes added the cost of the operator themselves, but the cost has already been added in the OperatorNode it uses as precondition, therefore the cost has been added twice. An idea to solve this, would be to just add the cost in the first operator node and ignore the operator cost in all succeeding operator nodes (conditional effect nodes). But with future compatibility in mind, there is the possibility, that an OperatorNode is a precondition for multiple different operators, therefore it is not possible to just add a operator cost to its cost counter, since it is not given, that all operators depending on this node have the same cost. But since all OperatorNodes already hold the operator cost in a separate variable, to allow the graph to be reset easily, it is possible to just ignore the operator cost in the cost counter, and only apply the operator cost directly to update a state variable. This allows the counter in the OperatorNode to be used freely for all operator that depend on the node.

3.5 Problem 3: simplify had to be changed

The simplify method also had to be changed. In the original version each UnaryOperator corresponded to an effect, so if an effect was dominated the whole node was dominated and could been removed. In the optimized version, an OperatorNode can hold more than one effect and be the precondition of other effects. So each effect must be check individually if it is dominated. And if all effects are dominated, and the OperatorNode is not a precondition of a conditional effect, then the operator node is fully dominated and could been removed. For this, it was useful, how the vector operator_nodes that holds all pointers to all operator nodes was constructed. The order of the entries in operator_nodes corresponds not only to the order of how the operators were defined in the planning task but also all OperatorNodes of conditional effects are later in the vector, than the OperatorNode they use as precondition. This allows to iterate backwards over the vector and checking for each entry, if the effects in the OperatorNode where dominated. If all effects where dominated and no other OperatorNode uses it as precondition, the node can be deleted (remove all edges from preconditions to the node, delete the node from operator_nodes and free the allocated memory). At any point in simplify, if an OperatorNode N_o is checked, all OperatorNodes, that uses N_o as precondition already have been checked, since we iterate over the vector backwards and those nodes are later in the vector than N_o. So at no point the removal of N_o has an impact to any already checked node.
The implementation of this also lead to a lot of bugs. For example, it is highly relevant to be mindful when checking the effects of an OperatorNode. Since if an effect is dominated and therefore removed from the vector of effects, the index of all effects in the vector after the removed effect are decreased by one. If the iteration over the effects happens by just simply increasing the index, the first element after the removed effect is overlooked. Instead of keeping track of this, it is much more simple to iterate backwards over the effects, since this way the removal of an effect does not change the indexes of unchecked effects.

3.6 Problem 4: Correctness of the FF heuristic

To verify that the implementation of FF heuristic h^{FF} was correct there were to main possibilities: formally show that the implementation is correct or have the same results as in the base implementation for all test cases. To formally show that the implementation labor-intensive, but it is much easier to show that the results are equivalent to the already thought to be correct original implementation. This is not a proof, that the implementation is correct, but if the results are equal in all 2742 test cases there is a high probability.
The problem with this approach is that h^{FF} is not deterministic in its definition. So to have the same results, all updates of all operators must be in the exact same order as in the original implementation. The update order directly corresponds to the order in the vector operator_nodes and the order in which the effects are saved within a node. So in the original version, where every node holds one effect, the order of updates was the same as in the planning task, but in the optimized version, there are multiple effects in each node, the order in the node is generally the same, except for conditional effects, since they are in a different node, which is updated after all effects inside the first node. To counter this, the

simplest fix was to changed the order in the original implementation and for each operator first append all normal effects as UnaryOperators to the list and than add the conditional effects to the list before going to the next operator. With this change the update order is the same. But the results for h^{FF} were still different.

The next assumption was, that the reason for the difference was the implementation of the queue which holds all nodes that had a change in their precondition and maybe need to be updated.
The original implementation uses a class AdaptiveQueue for this queue. AdaptiveQueues start of as a bucket queue which is efficient as long as the keys are small and a limited number of elements per key. But if there is are big keys or a large number of keys, it becomes inefficient and the AdaptiveQueues change to a heap queue which is more efficient for such keys. The problem with this is, that the switch from bucket to heap queue can effect the order within the queue and therefore lead to a different order in the updating of nodes and therefore lead to different non deterministic heuristic values for the FF heuristic. To solve this both the original implementation and the new one were changed to only use the bucket queue.
But this also did not solve the difference in the h^{FF} results. So the code had to be debugged very carefully which showed multiple bugs and details which had an effect on the h^{FF} results. One of those details is here further explained:

In the simplify method there is a hashmap used, to determine for each effect with the same preconditions the cheapest operator. To achieve this the hashmap uses as key a pair with the first entry being the preconditions of the effect and the second entry being the effect. The values of the hashmap also were a pair, with the first entry being the cost of the cheapest found such key pair and the second entry being the operator. In the original implementation, the operator was represented by the index of the UnaryOperator in the list of all UnaryOperators and in the new implementation the operator is represented by a pointer to the OperatorNode. When the hashmap is build, it is iterated over all operator nodes and over all their effects, a key is generated and if there is no entry with this key, the key along with the value is inserted in the hashmap. If there already exists a entry with this key, the value of the entry in the hashmap $value_{old}$ is compared with the value of the operator which is to be inserted $value_{new}$. In the original original implementation this looks like the following:

$$\text{if } (\ value_{new} < value_{old}\ )\ \{\ \text{hashmap.insert(key,}\ value_{new}\text{);}\}$$

Since the compared variables are of type pair C++ uses the overwritten implementation of the binary operator "<", which compares the second element of the pair if the first element of the pairs are equal. In the original implementation the index of the UnaryOperators are deterministic so the hashmap will look always the same for the same input. This achieved that operators with smaller indexes dominated operators with bigger indexes. But in the new implementation pointers are used to identify an OperatorNode and those pointers are not deterministically assigned. This means, that for a given key in the hashmap the value

can di er to the original implementation for e ects with the same precondition and cost. This can lead to di erent h^{FF} results, since the operator that set the value of a state variable di ers the backtracking of h^{FF} can return a di erent set of operators with a di erent sum of costs.
To achieve the hashmap entries deterministically and equal to the original implementation the fact that the e ects are in the same order in the list of all operators in both the original and the new implementation could be used. Since the lling of the hashmap was done by iterating forwards over the list of operators the index in each iteration grew monotonically. So in the original implementation the hashmap will always contain the operator with the lowest index of all operators with the smallest cost and with this the requirement that the operator with the lowest index dominates other equal good operators is achieved. So if we iterate over the list of operators and just compare the rst element of the value pair, and only replace the value if the new cost is smaller than the old cost, we get the same result as in the original implementation.
After a lot of debugging, the h^{FF} results were the same in all 2742 test cases. With this showing that the implementation is very likely to be correct, all measures to achieve the same results but which were not necessary for a correct algorithm could be removed, for e ciency sake.

3.7 Problem 5: Speed

The calculation times with this new implementation have shown to be much slower, than the original implementation. The rst idea to combat this was to remove the recursive part in the code introduced to update nodes in the graph.
The updating of the graph in the code follows a simple logic. At the start, all operators that do not require any precondition are processed, meaning all proposition nodes that are e ected by such a operator updates its cost and is added to a queue.
After the rst element of the queue is removed and all OperatorNodes that use this PropositionNode as a precondition are informed, that the proposition has a new cost. The e ected OperatorNodes then check, if all preconditions are ful lled, if so it updates all its e ects and add them to the queue. After which it is repeated for the new rst element of the queue. This is repeated until all goal state variables are ful lled or there are no more elements in the queue.
In the original implementation each UnaryOperator directly updates its e ect, but in the new implementation this is no strictly true, since there could be other OperatorNodes depending on this OperatorNode. In the recursive implementation this is solved, well recursive, when an OperatorNode is updated, it updates its cost counter and calls, in its update function, the update function of all nodes it e ects (whether they are an OperatorNode or PropositionNode). If it is also a OperatorNode it does exactly the same thing, but if it is a PropositionNode, it updates its self, adds itself to the queue and returns. This is a neat solution of the problem but recursion adds its own overhead, since there are many function calls.
A iterative implementation does not have this overhead and should be faster. So in a new

implementation instead of calling the update function recursively there is an additional queue of OperatorNodes and the e ects are separated into e ected PropositionNodes and OperatorNodes. First it updates all the PropositionNodes directly depending on the current OperatorNode after which all OperatorNodes directly depending on the current OperatorNode are updated and if they are ful lled are added to the secondary queue. Then it iterates over the secondary queue with the same logic, until it is empty. This has the same e ect as the recursive solution but removes the recursive function calls.
But this made the calculation time even longer. So the next idea was that the problem was the cache. Since there could be a lot of nodes, not all are held in the cache at all times and they needed to be loaded from the memory, which adds additional time. Therefore the ArrayPools where reintroduced to hold all the data previously held by vectors in the GraphNodes. This has the e ects, that a node is much smaller in memory space, all nodes of the same type have the same size and the ArrayPool can be hold in the memory reducing the loading time. The iterative solution allowed to implement the ArrayPools again without major redesign, since the update function was no longer part of the graph nodes, but was achieved in the relaxation heuristic class. Just the usage of ArrayPools sped up the calculation time, but it was still signi cantly slower than to original or even the rst recursive implementation. Only when the size of the nodes where reduced the times came close to the original implementation.
After this, I also reimplemented the recursive variant to make use of the ArrayPools. To allow the update function access to the ArrayPools of the heuristic class, I reimplemented the function as part of the heuristic class instead of as part of the GraphNodes, which takes the needed nodes as inputs. This however only had very minor positive e ect to the normal recursive implementation and still being way slower than the iterative solution.

4 Results

In this chapter the results gathered are presented. All compared implementations and their pseudonyms are listed here:

base: The original implementation as of March the 21th 2022.

recursive-base : A recursive implementation, where the neighbours of each node is stored in the node with vectors.

recursive-reduced : A recursive implementation, where the size of each node was reduced by using ArrayPools.

iterative-reduced : A iterative implementation, where the size of each node was reduced by using the ArrayPools.

Some implementations are not included, like a recursive implementation without the use of ArrayPools, to make the data less cluttered, and they were generally strictly worse than the shown ones. To compare the implementations three main characteristics are compared:

Coverage : The number of tasks an implementation solved within the time and memory limit.

Search Time : The time needed for the GBFS for a planning task without the construction of the graph.

Total Time : The time needed to build the delete relaxation heuristic graph plus the search time plus any time needed for clean up at the end like free reserved memory.

To test the di erent implementations the benchmark environment of Fast Downward [7] was used, which allowed to run searches in 76 di erent domains of planning tasks in a total of 2'742 di erent tasks for each of the implementation for both h^{add} and h^{FF}. To avoid extremely long test runs, a search was terminated if it took longer than 30 minutes. This can generate a bias, when comparing di erent implementations. Since some implementations that are faster and solve a task within the time limit with high search and total time, and a slower implementation did not nish within the limit. When comparing the two

the additional data points of the faster implementation can have a signi cant e ect on the average. Therefore when comparing the implementations, only the planning task solved by both are considered.
Further hardware resource limitation were 40 minutes with 3.6GB of RAM on a single core of an Intel Xeon E5-2600 processor, for each task.

4.1 Overall Comparison

To compare the di erent implementations the geometric mean over the search times and total times is calculated (see gure 4.1). But since not all domains have the same number of tasks in the test environment, the geometric mean is rst calculated over all the tasks within a domain and in a second step the geometric mean over all those domain means is calculated. This gives a better approximation for a random task, when all domains have the same probability.

	Implementation	Coverage	$\bar{R}_S$	$\bar{R}_T$
add	base	1817	1.000	1.000
	iterative-reduced	1810	1.008	1.057
	recursive-reduced	1793	1.202	1.237
	recursive-base	1787	1.212	1.234
FF	base	1760	1.000	1.000
	iterative-reduced	1771	0.958	1.005
	recursive-reduced	1754	1.119	1.154
	recursive-base	1741	1.155	1.181

Table 4.1: Di erent implementations compared with the base. $\bar{R}_S$ and $\bar{R}_T$ are the geometric mean of the relativeSearch time, respectiveT otal time over all tasks solved by every implementation. The Coverage is out of the 2742 di erent planning tasks. The subset of tasks which were solved from all implementations and therefore used to calculated the means has 1781 entries for the Additive heuristic and 1732 for the FF heuristic.

The coverage is close for all implementations which already indicates, that no implementation is signi cantly better than the other over all. But the iterative-reduced seems to be better than the both recursive variants and even better than the base implementation for the FF heuristic. But the three new implementations are already strictly ordered with the iterative solution being better than the recursive implementations and the recursive-reduced being better than the recursive-base implementation in both the Additive and the FF heuristic.
The average relative search time for the Additive heuristic is best for the base implementation, but the iterative-reduced implementation is not far behind with only 0 :8%, so it is almost as fast. The recursive-reduced and recursive-base implementations are signi cantly slower in average with 20.2% and 21.2% more search time needed in average.
For the average search time for the FF heuristic the iterative-reduced implementation is even faster than for the base implementation by 4.2%. But the iterative implementations are again signi cantly slower than the base implementation.
The average relative total times are a big bigger compared to the relative search times.

This is expected, since the overhead in building the graph is bigger than for the base implementation. But in the FF case its only 0:5% for the iterative-reduced, so it has about the same time needed as the base implementation. But again the recursive implementations are signi cantly slower.
This overview is however a strongly reduced impression of the data and it makes sense to also compare the implementations on a ner level, namely the domains, since di erent domains can vary strongly in there composition which can have a strong impact on how e cient the di erent implementations solve them.

4.2 Comparison by Domain

A data point in the gures 4.1, 4.2, 4.3, 4.4 corresponds to the following formula:

$$x_{\text{domain, implementation}} = \log_{10}\left(\frac{X_{\text{domain, implementation}}}{X_{\text{domain, base}}}\right)$$

, where X corresponds to the arithmetic mean of the search or total time of a given domain and implementation. The recursive-base implementation is no longer included, since it is generally worse than the other two new implementations.
The gure 4.1 like the others shows how strongly the search time can vary between the di erent domains and the implementations. As can be easily seen both the iterative-reduced and the recursive-reduced implementation are in many domains signi cantly faster than the base implementation. Also the iterative-reduced implementation seems to almost always has a smaller average search time than the recursive-reduced version.
The gure 4.2 shows the average relative total times. The di erence seem to be generally a bit smaller, so there seems to be less variation between the implementations. The values in the di erent domains seem to be similar to the ones in the gure 4.1 which makes sense, since the main time consumption of solving a task should lay in the search and not in the construction. Non the less already the rst line shows that agricola-sat18-strips has a slower total time than the base but it had a faster faster search times. This means the solving took for all tasks in this domain longer with the new implementations than with the old one even thou the search was faster. But since the search growth faster than the building of the graph for more complex tasks, there comes a point at which it is probably more sensible to use the new implementation than the base with growing complexity even thou for this set it took in average more time to solve. This assumption seems to be re ected in the test data. The most complex task in this domain solved by both the base and the iterative-reduced implementation took about 1555 seconds with the base implementation but only about 789 seconds with the iterative implementation.
For the FF heuristic search times (gure 4.3) are similar to the Additive heuristic times, but the extreme cases are less extreme and the times are generally closer to the base than with the Additive heuristic.
The average relative total times in gure 4.4 are generally slightly left shifted to the ones in the Additive heuristic. This means that total time relative to the the base are a bit faster for the FF heuristic than for the Additive. Therefore the optimization has a bigger positive e ect for the FF heuristic than for the Additive heuristic.

Figure 4.1: The search time of di erent implementations of the Additive heuristic relative to the original implementation ("base:add") compared on a logarithmic scale, split up into di erent domains of experiments it was tested in.

Over all implementations it shows that the iterative-reduced is generally better than the recursive-reduced. Also whether it makes sense to use the new implementation is highly deepened on the domain and the complexity of the task. For small tasks it often makes sense to use the base implementation but in many domains, it makes sense to use the iterative-reduced implementation, even if it has a longer average total time, if the search time is faster with growing complexity. Also overall the average positive e ect seems to be bigger for the FF heuristic than the Additive heuristic.

Figure 4.2: The total time of di erent implementations of the Additive heuristic relative to the original implementation ("base:add") compared on a logarithmic scale, split up into di erent domains of experiments it was tested in.

Figure 4.3: The search time of di erent implementations of the FF heuristic relative to the original implementation ("base: ") compared on a logarithmic scale, split up into di erent domains of experiments it was tested in.

Figure 4.4: The total time of di erent implementations of FF heuristic relative to the original implementation ("base: ") compared on a logarithmic scale, split up into di erent domains of experiments it was tested in.

5 Conclusion

The implementations with the optimization idea are not strictly better compared to the original implementation, but there are many domains for which it is de nitely the better choice and more so with increased complexity of the task.
During the implementation I realized how optimized the implementation already was. Even thou the base used an approach for building the graph which leads to a generally more computationally expansive search, than the one used for the new implementation, it was quit di cult to come near its speed with the more promising idea, since the base was quit well implemented in terms of memory use, etc.
To continue the optimization the idea discussed in section 3.2 seems to be promising, especially for more complex problems, since it has additional complexity in the construction of the graph but it has the possibility to signi cantly speed up the search. And since this optimization is over more than one operator at the time, it is also expected to have a positive e ect in even more domains.

I have learned a lot in the course of this work, not only about planning tasks and in particular about Additive and FF heuristics, but also about C++ and optimizing the run time of code written in C++.. And although the results are not quite as positive as I had hoped at the beginning, I am happy with the outcome and with the experience as a whole.

Bibliography

[1] Blai Bonet and Hector Geffner. Planning as heuristic search. Artificial Intelligence, 129 (1-2):5–33, 2001.

[2] Malte Helmert. The Fast Downward planning system. Journal of Artificial Intelligence Research, 26:191–246, 2006.

[3] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. Artificial Intelligence, 173:503–535, 2009.

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

[5] Keyder, Emil and Geffner, Hector. Heuristics for planning with action costs revisited. In ECAI 2008, pages 588–592. IOS Press, 2008.

[6] Stuart J Russell and Peter Norvig. Artificial intelligence: A modern approach, 2010.

[7] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://doi.org/10.5281/zenodo.790461, 2017.

[8] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587):484–489, 2016.

University of Basel
Faculty of Science

! "#$%&%'() *+) *+ , #(" * '(-(#+. * '" / &('0+

*+ , -." / + , #&0&1 2 - .030%+4 , &4 , &5.0#+03+$ 6 &0 , / &730" / 8&
930 , $.0%+4 , &:34 6 & ; 23 6 0 , &43+#+ , 0.&

9+%.2&4:&9<2$+$=& Optimized Computation of the Additive and FF heuristics

>0 6 2& ! $$2$43=& Prof. Dr. Malte Helmert
>0 6 2&@%" / 2 , %=& Cyrill Imahorn
A0%3+ - ". 0%+4 , &>4B=& 19-051-333

C +%<& 6 D& $+# , 0%" 32& E& / 2 - .032& %<0%& %<+$& $" F 6 +$$+4 , & +$& 6 D& 4G , & G43H& 0 , / & %<0%& E& <0 I 2& : "..D& 0 - H , 4G.2 / #2 / & %<2& 0$$+$%0 , - 2& 32 - 2+ I 2 / & + , & - 4 6 J.2%+ , #& %<+$& G43H& 0 , / & %<0%& +%& - 4 , %0+ , $& , 4& 6 0%23+0.& %<0%& <0$& , 4%& F22 , &:43 6 0..D& 0 - H , 4G.2 / #2 / B& E& <0 I 2& 6 2 , %+4 , 2 / & 0..& $4" 3 - 2& 6 0%23+0.$& "$2 / &0 , / &<0 I 2& - +%2 / &%<2$2&+ , &0 - - 43 / 0 , - 2&G+%<&32 - 4# , +$2 / &$ - +2 , %+:+ - &3". 2$B&

5.0 - 2K& 10%2=&& Wil, 19.07.2022 &@%" / 2 , %=&

C +..&%<+$&G43H&F2&J" F.+$<2 / L&

- (●) >4&
- ( ) M2$B& C +%<& 6 D& $+# , 0%" 32& E& - 4 , :+3 6 &%<0%& E& 0#322&%4&0& J" F.+ - 0%+4 , &4:&%<2&G43H&*J3+ , %N / +#+%0.8& + , & %<2& .+F303DK& 4 , & %<2& 32$203 - <& / 0%0F0$2& 4:& %<2& O , +I 23$+%D& 4:& P0$2.& 0 , / N43& 4 , & %<2& / 4 - " 6 2 , %&$23I 23&4:&%<2& / 2J03% 6 2 , %B&Q+H2G+$2K& E& 0#322&%4&%<2&F+F.+4#30J<+ - &32:232 , - 2&+ , & %<2& - 0%0.4#&@Q@5&*@G+$$&Q+F303D&@23I+ - 2&5.0%:43 6 8B&* - 34$$&4" %&0$&0JJ.+ - 0F.28&

5" F.+ - 0%+4 , &0$&4:=&&

5.0 - 2K& 10%2=&&

5.0 - 2K& 10%2=&& ! $$2$$43=&&&