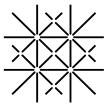

University of Basel

A* Tie-Breaking Strategies in Fast Downward

Bachelor's Thesis

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

Examiner: Prof. Dr. Malte Helmert
Supervisor: Remo Christen

Mirco Franco
m.franco@stud.unibas.ch
2021-052-022

30.06.2025

Acknowledgments

First of all I would like to give a huge thank you to my supervisor Remo Christen. Since the first day of this thesis he was a big help with his useful feedback, interesting discussions and important guidance. I am very glad to have had him as a helping hand and advisor as I always enjoyed the weekly meetings we had. Without him this bachelor's thesis would not have been possible. He always gave me hope and encouraged me on proceeding on my way. Additionally I would also like to thank Prof. Dr. Malte Helmert for suggesting this interesting topic and giving me the opportunity to work on this thesis. I learned a lot while working on this thesis and had a lot of fun implementing an interesting approach into a state-of-the-art problem solver.

Abstract

Planning is a fundamental problem in artificial intelligence. The goal of a planning problem is to find a sequence of actions that lead a given starting state to a predefined goal state. The A^* search algorithm is a widely used to solve cost-optimal planning problems, where the goal is to find a sequence with minimal cost. This search algorithm navigates through the different paths from node to node with the help of the f-value each node holds. But if multiple nodes have the same f-value, tie-breaking strategies have to find the optimal way to reach the goal. This thesis implements and evaluates the tie-breaking strategy *depth-diversification* within the Fast Downward planning system. Additionally a *distance-to-go* approach as a tie-breaking strategy is tested and evaluated as well. The default tie-breaker are ineffective, especially for zero-cost domains, where multiple nodes often have the same f-value. The findings show that structured tie-breaking strategies like depth-diversification and distance-to-go heuristics can make a difference and perform better than default tie-breaking strategies.

Table of Contents

Acknowledgments **ii**

Abstract **iii**

1 Introduction **1**

2 Background **4**
2.1 Planning Task 4
2.2 Heuristics 6
2.3 Best-First Search Algorithm 8
2.4 Plateau 10

3 Tie-Breaking Strategies **11**
3.1 Zero-Cost Domains 11
3.2 Default Tie-breaking 15
3.3 Depth-Diversification 16
3.4 Implementation 18
3.5 Distance-to-go 19

4 Evaluation **21**
4.1 Setup 21
4.2 Zero-Cost Domains 22
4.2.1 Configurations in Zero-Cost Domains 23
4.2.2 Comparison to Asai and Fukunaga (2017) 25
4.2.3 Detailed Evaluation 27
4.3 IPC-Domains 30
4.3.1 Comparison to Zero-Cost Domains 33

5 Conclusion **34**
5.1 Configuration Comparison 34
5.2 Unexpected Findings 35
5.3 Future Work 35

Bibliography **37**

1 Introduction

Planning is a subfield of Artificial Intelligence. It is about finding a plan for certain problems in an automated way. A plan in this context means a sequence of actions taken to get to a predefined goal from a given starting point. There are different kinds of problems that can be solved. To solve a problem one could use a problem-specific problem solver, which can only be used for problems of the same kind. But we will only look at general problem solving methods and not problem-specific approaches. This means that the problem solver is a general algorithm where the developer of the search algorithm does not know what kind of problem it has to solve. Because of that these general problem solvers can be used widely for many different problems. In a classical planning problem we have many paths that can be followed. Classical planning addresses the fundamental problem of finding a sequences of actions that lead to a certain goal from a given initial state. Often it is desired to find the optimal path to a goal. For cost-optimal planning this means that the total cost of the actions taken in the sequence should be minimized. To do that we want to follow the path that costs the least and we want to expand as few states as possible while following the path. The A^* search algorithm is widely used for optimal planning. It uses heuristic functions that estimate which of the successor states the most promising is. The A^* search algorithm guarantees optimality of the solution as long as the heuristic function has certain characteristics that will be looked at in detail in Section 2.2. But there are cases where two or more nodes n in a path look exactly the same with respect to their path cost $g(n)$ and heuristics $h(n)$. Adding these two values together we get the f-value. When the f-value is the same for multiple nodes then we need to decide which path we should take. This means we have to break the tie. The presence of zero-cost actions increases the number of nodes in a path that have the same values for their cost and their heuristic. A plateau is such a set of nodes, that have the same the path cost $g(n)$ and estimation to the nearest goal $h(n)$. In this paper we will look at different tie-breaking strategies we can use to determine which path will be best, not only with respect to the path cost, but also on how many states have to be checked until reaching the goal. There are the standard default tie-breaking strategies *first-in-first-out (fifo)*, *last-in-first-out (lifo)* and *random order (ro)*. But we can see in the example in Figure 1.1 that these default tie-breakers do not always find the path that needs the fewest expansion of nodes. In the example we will expand the nodes in a different order depending

on the tie-break. For this example we will assume that the successor nodes get generated in alphabetical order. For *fifo* the expansion order would be: $\langle A, B, C, D, E, F, G, H\rangle$ while *lifo* needs even more expansions with: $\langle A, C, G, F, K, J, B, E, D, I, H\rangle$. The *random order* is not structured and not deterministic but could look like this: $\langle A, B, E, C, D, F, J, H\rangle$.

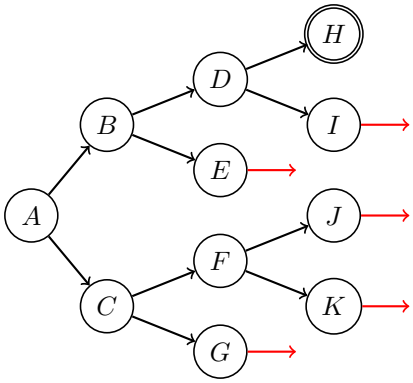

Figure 1.1: This is an example of nodes $A, \ldots, K$ that have the same path cost (g-value) and the same estimation value to the nearest goal (h-value). Therefore they form a plateau. Both these values can be the same when the actions (marked as arrows) have cost zero. The red arrows here are non-zero actions and lead to a node that is not in the plateau anymore. Those nodes outside the plateau will be ignored in this example as the node H already holds a goal state and will terminate the search.

To diversify this more and get an even distribution over the nodes, Asai and Fukunaga (2017) proposed a depth-based tie-breaking strategy. This approach sorts the nodes of a plateau into depth buckets sorted by their layer of depth. In the example of Figure 1.1 these buckets would be $\langle A\rangle$, $\langle B, C\rangle$, $\langle D, E, F, G\rangle$ and $\langle H, I, J, K\rangle$. Now instead of just starting to select the first or last node that got expanded, we select one node at the time from each depth bucket. How the node in the depth bucket gets selected is again a matter of the chosen default tie-breaker. The order of expansion for the depth-diversification with *fifo* as the default tie-breaker would be: $\langle A, B, D, C, H\rangle$. This uses the least expanded nodes and therefore has optimal expansion. But depth-diversification with *lifo* will expand all nodes before the goal in H is found. This example does not show the general case as it is easy to find a different case where *lifo* expands less nodes than *fifo*. For example if not H but K would be the goal state, then *lifo* would work better with depth-diversification than *fifo*.
To see which configurations of tie-breaking strategies and heuristics achieve the best performance, we conduct a series of experiments. For these experiments we use Fast Downward (Helmert, 2006) as the basis for our implementation of the depth-diversification. Fast Downward is a general planning problem solver. For this paper we added the implementation of different tie-breaking strategies to Fast Downward and tested them out.
The goal of this thesis is to implement depth-diversification and then evaluate its effectiveness on IPC-domains as well as domains that contain mostly zero-cost actions. Those so called zero-cost domains are interesting because by creating large plateaus we can test the tie-breaking strategies better. Additionally different configurations of tie-breaking strategies will be evaluated, including distance-to-go heuristics. A distance-to-go heuristic is a

heuristic function that focuses on the amount of steps needed to reach a goal rather than the cost of the path. The evaluation will be compared to previous results from the paper by Asai and Fukunaga (2017).
In Chapter 2 we will provide necessary context for the thesis for a better understanding on classical planning, heuristics and search algorithms. Next in Chapter 3 the fundamental idea of depth-diversification is discussed, as well as the zero-cost domains and the distance-to-go heuristics. Lastly we will discuss the results of the experiments in Chapter 4 before we draw a conclusion in Chapter 5.

2
Background

First we will introduce some background concepts needed for this thesis. We will explain what a planning task is and what it is used for. Then we will see what a heuristic is used for and which characteristics it can have. Lastly we will see how a best-first search algorithm works and what kind we will use for this thesis.

2.1 Planning Task

A planning task is a structured approach to solve a problem. Such problems can be of various kinds like organizing the logistic of a business or synthesizing molecules from given atoms. A solution of such a planning problem is a sequence of actions that have to be taken to get to a predefined goal.
When we explicitly describe such a planning problem, we call it a state space search problem. The state space search problem describes all possible states explicitly.

Definition 1. *(State space) A state space is formalized as a 6-tuple* $\mathcal{S} = \langle S, A, cost, T, s_I, S_G \rangle$ *where:*

- S *is a finite set of states*
- A *is a finite set of actions*
- $cost$ *is a function* $cost : A \to \mathbb{R}_0^+$
- T *is the transition relation between states, with* $T \subseteq S \times A \times S$
- s_I *is the initial state*
- S_G *is a finite set of goal states, with* $S_G \subseteq S$

We see that a state space consists of different states where one of them is the starting point called initial state s_I and a number of them are goal states. Actions each have a cost which is mapped to it through the cost function. The actions can bring one state s to another state s' via a transition. A transition $\langle s, a, s' \rangle \in T$ is a triple consisting of the current state

s (predecessor), the action a, and the successor state s'. Another way to write the transition is $s \xrightarrow{a} s'$.
A planning task is a compact description of a planning problem. It efficiently describes the general problem without getting too big, but still includes all states. In contrast to the state space search the planning task describes the states implicitly.
Let us define and explain some terms needed to understand the following chapter.

State: In a planning problem we have states. Each state represents how the situation of the problem is at a given time. For that each object in the problem setting is assigned a value. For example in a logistics problem we have the object or more formal the variable *package-A* that has the status *at-location-A*. We call this an assignment of the value *at-location-A* to the variable *package-A*. If all variables in a problem have a concrete value assigned to them, we call it a total assignment. A partial assignment is a state where not all variables are assigned.

Action: To get from one state to another we need something to change. An action is a procedure that changes the values of some variables of the problem setting. In the logistics example, we can have the action *move-from-A-to-B*. This action would take the variable *package-A* and move it to another location. For that we need the precondition *at-location-A* to be true for the variable *package-A*, before we can apply the action to the state. After applying the action, the value of the variable *package-A* has changed.

Path: A path is a sequence of actions that start with a state s and end in another state s'. Formal it is defined as a sequence of transitions such that the intermediate states s and s' match the transition.

$$\text{A path from } s \text{ to } s' \text{ is a sequence } \langle\langle s_0, a_0, s_1\rangle, \langle s_1, a_1, s_2\rangle, \ldots, \langle s_n, a_n, s_{n+1}\rangle\rangle \text{ such that } s_0 = s \text{ and } s_{n+1} = s'.$$

We can write $s \to s'$ for a path from s to s'. Here the number of actions between s and s' can be arbitrary.
For this thesis we will only consider classical planning. A planning problem is called classical if the problem environment has the following properties:

- Static: the environment will not change without an action
- Deterministic: each action will always lead to the same successor from e given predecessor
- Fully observable: there is no hidden information, the state of the environment is always given
- Discrete: there is only a finite number of states and actions
- Single-agent: there is only one agent that can change the environment or take actions

Because we only look at deterministic state spaces in classical planning, the transitions have to be deterministic in $\langle s, a \rangle$. This means that $s \xrightarrow{a} s_1$ and $s \xrightarrow{a} s_2$ can only be in T together if $s_1 = s_2$. Or more formal: $\forall s \in S, \forall a \in A : \langle s, a, s_1 \rangle \in T \wedge \langle s, a, s_2 \rangle \in T \Rightarrow s_1 = s_2$.
We call a path a solution or a plan if it starts with the initial state s_i and ends in one of the goal states $s_g \in S_G$.
For this thesis we only consider optimal planning, which will always give us an optimal solution (minimal path costs) for a given problem. To formalize planning tasks we use the finite domain representation formalism $SAS+$ proposed by Bäckström and Nebel (1995).

Definition 2. *(SAS+) This definition is based on the work of Bäckström and Nebel (1995). A planning task problem consists of a 4-tuple* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle$ *where:*

- $\mathcal{V}$ *is a finite set of state variables* v_i
- $\mathcal{O}$ *is a finite set of actions, such that an action* $a \in \mathcal{O}$ *has preconditions, postconditions and a cost:* $\langle pre(a), post(a), cost(a) \rangle$
- s_0 *is the initial state*
- s_* *is the goal which is a partial assignment*

Each variable can be assigned a value. Assigning a value d to the variable v is denoted with $v \mapsto d$. Additionally each variable has a finite domain $dom(v)$, that denotes which values d the variable v can be assigned to. A state in the $SAS+$ formalism is a total assignment of all variables $v \in \mathcal{V}$ such that $\{v \mapsto d \mid v \in \mathcal{V}, d \in dom(v)\}$ The state s_0 is a total assignment and the goal s_* is a partial assignment of the variables in $\mathcal{V}$. The pre- and postconditions $pre(a)$ and $post(a)$ are partial assignments too. To be able to apply the action a in the current state s all variables of the partial assignment in $pre(a)$ have to match with the values of the variables of the current state s. The action a is only applicable in a state s if each assignment in $pre(a)$ is also assigned in state s. With that we get a set of actions that can be applied to the state s. In cost-optimal planning we want to find the action in this set that will lead us to the goal while minimizing the path cost. To find the action that looks most promising we can use a heuristic function that computes a estimation for each successor node.

2.2 Heuristics

A heuristic is a function that maps each state of the planning problem to a heuristic value. This value is an estimation of the cost to the next goal state beginning from the current state s, for which this heuristic value is computed. A heuristic value is always computed to one state s that is considered at the moment. In this thesis we consider different heuristic functions that will focus more on the path cost or more on the number of steps needed to reach the goal state. The heuristic function is used in a search algorithm, which we will see later in Section 2.3. With the computed heuristic value the search algorithm is guided in the direction of the most promising path according to the heuristic. For cost-optimal planning this is the path, that has the lowest path cost. But of course there are heuristics which are

better and such that are not as good in guiding the algorithm. The perfect heuristic h^* is the heuristic that always gives back the true value of the cost from the state s to a goal state. A good heuristic function for cost-optimal planning will approximate this value as close as possible, while a bad heuristic can not get a good approximation. Each heuristic function has different characteristics, that influence the quality of the search.

Definition 3. *(Safety)*

A heuristic function h is safe, if for every state s it holds that if $h(s) = \infty$ then also $h^*(s) = \infty$.

This means that a heuristic function is called safe, if the h-value of a state s is only ∞ when there really is no solution from the state s to a goal state. The heuristic value is never ∞ when there is still a solution possible from the state s.

Definition 4. *(Goal-awareness)*

A heuristic function h is goal-aware, if for every goal states $s \in S_G$ it holds that $h(s) = 0$.

Goal-awareness implies that a heuristic function assigns the value 0 to every goal state in the planning problem. The heuristic recognizes a goal state when the state is evaluated.

Definition 5. *(Admissibility)*

A heuristic function h is admissible, if for every state s it holds that $h(s) \leq h^*(s)$.

With admissibility the heuristic ensures that for every state $s \in S$, it estimates the cost to reach the goal not more than the actual minimal cost. In other words, the heuristic never overestimates the true cost from a state s to the goal, which ensures that it is optimistic.

Definition 6. *(Consistency)*

A heuristic h is consistent, if $h(s) \leq cost(a) + h(s')$ for all transitions $s \xrightarrow{a} s'$

This means that a heuristic function is consistent, if the estimated distance to the goal (the h-value) is not greater than the cost to a successor combined with the successor's h-value. Or more formal: for every transition $s \xrightarrow{a} s'$ with cost $cost(a)$, the estimated cost from s to the goal is not more than the cost of reaching s' plus the estimated cost from s' to the goal. This condition ensures that the estimated total cost along a path never decreases.
In this thesis we only consider optimal planning. The cost of a solution path should be minimal. To ensure that the solution is cost-optimal, the choice of the search algorithm is important. Some algorithms can not guarantee optimality and some need specific heuristics to guarantee optimality. In the next Section 2.3 we will see that the search algorithm A^* is used for this thesis. A^* can guarantee optimality but only if it uses an admissible heuristic. The first heuristics we used in this thesis is the admissible *LM-cut* heuristic (Helmert and Domshlak, 2009). The *LM-cut* heuristic works by repeatedly finding so called landmark cuts to achieve a good approximation of the cost to the next goal state. Another heuristic we used is the *Merge and Shrink* heuristic (Helmert et al., 2007) or just *M&S. Merge and Shrink* is an admissible heuristic that takes simple transition systems for each state variable. These

transition systems are then iteratively merged together. To control the complexity of the system it gets abstracted by grouping together states. This abstraction is called shrinking. With these heuristic functions we can now use a search algorithm to solve planning problems.

2.3 Best-First Search Algorithm

Best-first search algorithms are search algorithms that prioritize paths that seem promising. In the search we have nodes n that have a corresponding state $s(n)$ to it. The best-first search uses the f function to decide which node is the most promising successor. This f function can be chosen differently and gives us different search algorithms. We will only consider the search algorithm A^* in this thesis. A^* was proposed in 1968 by Hart et al. (1968) and can guarantee an optimal solution with a suitable heuristic. The f function of A^* takes the g-value of the node n and the h-value of the corresponding state $s(n)$ and combines them to get a good estimation on how good this path is. Here the g-value is the actual cost of the path from the initial state s_I to the current state s. The heuristic value h estimates the rest of the path from the current state s to the nearest goal state $s_g \in S_G$. Which heuristic function A^* is using can be chosen and determines how well the search algorithm works. A^* will always find an optimal solution when using an admissible heuristic.
Algorithm 1 shows how the A^* search algorithm could look like. We need an OpenList and a ClosedList to keep track of the seen nodes and states. The OpenList holds the nodes that are not yet looked at but are successors of nodes that we already expanded. Expanding means that a node gets selected and removed from the OpenList. Then the successors of the node will be generated and put into the OpenList. Which node gets selected is object of the search algorithm and its configuration. A search algorithm has to order the nodes in the OpenList, for example by the heuristic value. The problem arises that more than one node can be of the same order and have the same values. For these situations the search algorithm has to define a tie-breaking strategy to select one of the nodes with the same value. Later in Section 3.2 we will see how this can be done in particular. After a node is selected and expanded, its state will be put into the ClosedList. The procedure starts with the initial node n_0 which is the root node and holds the initial state s_I.

A^* gives us an optimal solution independent on the tie-breaking strategy. Because of that we can omit the tie-break in the algorithm 1. Nevertheless it is necessary to implement a tie-breaking strategy in practice, to decide which node should be expanded next when two nodes have the same f-value. Many search algorithms will just use a default tie-breaking strategy like *first-in-first-out*, *last-in-first-out* or *random*. The tie-breaking strategy *first-in-first-out* or short *fifo* will expand the nodes in order of generation. So every node gets put into a list as they are generated and then *fifo* will choose the node that was put into the list first. In contrast *last-in-first-out* or *lifo* will choose the last node that was added to the list. Lastly *random* or *ro* chooses the next node randomly from the list. But we will see later, that there are better options for tie-breaking strategies to choose from. A tie-breaking strategy is denoted as a criterion that decides which node to choose. To show which tie-

Algorithm 1 A*

```
1: procedure ASTAR
2:     OpenList ← PriorityQueue ordered by f(n) = g(n) + h(s(n))
3:     ClosedList ← empty set
4:     g(n_0) ← 0
5:     f(n_0) ← h(n_0)
6:     Insert n_0 into OpenList with priority f(n_0)
7:     while OpenList is not empty do
8:         n ← OpenList.remove_next()
9:         if is_goal(s(n)) then
10:            return extract_path(n)
11:        end if
12:        Add s(n) to ClosedList
13:        for each m ∈ successors(n) do
14:            g_new ← g(n) + cost(n, m)
15:            if m ∈ ClosedList and g_new ≥ g(m) then
16:                continue
17:            end if
18:            if s(m) ∉ OpenList or g_new < g(m) then
19:                g(m) ← g_new
20:                f(m) ← g(m) + h(s(m))
21:                if m ∉ OpenList then
22:                    Insert s(m) into OpenList with priority f(m)
23:                else
24:                    Update priority of s(m) to be f(m)
25:                end if
26:            end if
27:        end for
28:    end while
29:    return Unsolvable
30: end procedure
```

breaking strategies are used we write it as follows: [$criterion_1$, $criterion_2$, ... , $criterion_k$]. If multiple nodes have the same $criterion_1$, the next $criterion_2$ has to be computed and compared. After that the $criterion_3$ is needed, but only when both criterions before are equal for multiple nodes. This can go on until the last criterion that has to be a tie-breaker that always only chooses one node.

For A^* these criterions would be just $[f]$ or $[g+h]$. In the original paper for A^* (Hart et al., 1968), it is stated that a node should be chosen by the smallest f-value and the tie-break should happen arbitrarily, but always in favor of any goal state. Hansen and Zhou (2007) additionally state that A^* achieves the best performance, when using the h-value as a tie-breaker. In practice this is also commonly used to take the h-value as a first tie-breaker to get $[f, h]$. The optimality of A^* is still given independent of the following criterions. It does not matter for cost-optimality, which node from the set of nodes with the same f-value is chosen. The h-value can just break the tie and still guarantee optimality.

2.4 Plateau

But now when both the f-value and the h-value are the same, we need another tie-breaking strategy to decide the next best node to expand. For this final tie-break we can use one of the default tie-breaking strategies *fifo*, *lifo* or *ro*. Because these three strategies will always choose only a single node. Any tie-breaking strategy that gives back only one node at a time, can be used for the final tie-break. We call a set of nodes with the same f- and h-values in A^* a plateau. Such a plateau will be denoted by *plateau(f, h)* or with concrete values (for example: *plateau(5,3)*). In the case where A^* is considered only with $[f]$ there would be $[f]$-plateaus where the h-value does not have to be the same for two nodes to be in the same *plateau(f)*. In the same way there can be plateaus with the other criterions too, but we will not consider them here. The entrance of any *plateau(f,h)* is the node whose parent is not in the same *plateau(f,h)*.

Definition 7. *If* $f(m) \neq f(n) \vee h(m) \neq h(n)$ *with* m *as predecessor of* n*, then* n *is an entrance to the* plateau$(f(n), h(n))$.

We call the distance from the entrance to the node n the *depth* $d(n)$ of the node n. The depth $d(n)$ of a node n is zero when the node n and its parent node m are not in the same plateau. If both nodes n and the parent node m are in the same plateau, the depth $d(n)$ of n is one more than the depth $d(m)$ of the parent node m.

Definition 8. *For node* n *and its predecessor node* m*: if* $f(m) = f(n) \wedge h(m) = h(n)$*, then* m *and* n *are in the same* plateau$(f(n), h(n))$ *and* $d(n) = d(m) + 1$*. Else* n *is an entrance to the* plateau$(f(n), h(n))$ *and has depth* $d(n) = 0$.

The *final plateau* is the last plateau and contains at least one goal state. After reaching the *final plateau* the search algorithm could theoretically be finished within the next expansion if the tie-break chooses the right node with the goal state. Because of that the tie-breaking strategy can be important in certain situations and can either increase the amount of resources needed or it can reach the goal immediately.

3
Tie-Breaking Strategies

While the default tie-breaking strategies are widely used, they do not follow any informed strategy. Because of that we would not expect these strategies to produce an effective order of expansion. The advantage of *fifo*, *lifo* and *ro* is that they are implemented easily. But Asai and Fukunaga (2017) show that these default tie-breaking strategies are not effective for certain domains.

3.1 Zero-Cost Domains

An action a is called zero-cost action if $cost(a) = 0$. If the action cost is non-zero ($cost(drive) > 0$), we call it a positive or non-zero action. Domains that contain mostly zero-cost actions are therefore called zero-cost domains. These domains still have to have at least one positive action or else it would not matter which actions are taken in which order because the total cost at the end would always be zero. Zero-cost domains are useful to model certain scenarios that domains with only positive action costs can not model. With the help of zero-cost actions the search algorithm does not only focus on the cost of the different actions but rather give the cost a meaning. For example the traditional DRIVERLOG domain, consists of a truck that has to drive around different locations to deliver packages. A cost-optimal plan for this domain minimizes the path cost and because all actions are positive the search has to consider all actions. Each step has a certain action cost and so all steps are relevant for the total cost. The zero-cost domain DRIVERLOG-FUEL from Asai and Fukunaga (2017) modifies the cost model to have a more fuel-centric view. Only the driving action *drive* has a positive action cost, while the picking up (*pick*) and dropping off (*drop*) of packages are zero-cost actions. Intuitively this corresponds to the idea that driving around uses up fuel, while the handling of the packages do not influence the amount of fuel spent. In the PDDL model, this intuition is realized by assigning a positive action cost to the driving action and zero to all other actions. This example intuitively shows how zero-cost actions can help in modeling more realistic cost structures. But most of the benchmark domains in planning competitions do not consider zero-actions. The most commonly used benchmark set is the set of domains and instances used in the past International Planning Competitions (IPC). Most of these IPC-domains use positive action costs for most actions in the domain. Asai

and Fukunaga (2017) created domains that have many zero-cost actions, but always at least one action with positive action cost. Even though most IPC-domains do not even consider zero-cost actions, it is still meaningful to look at zero-cost domains. Zero-cost domains can be used to study tie-breaking strategies and their effectiveness. With many zero-cost actions the need of tie-breaking strategies and their impact on the result increases. This is because zero-cost actions can lead to the formation of plateaus.
But zero-cost domains bring a problem with them. A zero-cost action does never change the path cost from one node to another.

Proposition 1. *The path cost from the current node n to another node n_k is zero if all actions used to go from n to n_k are zero-cost actions.*
If $n \xrightarrow{a_1} n_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} n_k$, where a_i are zero-cost actions with $i = 1, \ldots, k$ Then $g(n) = g(n_1) = \ldots = g(n_k)$ because no action adds cost to the path.

Just like the g-value which does not increase because the cost of the action is zero, also the h^*-value often does not change either. Because we only look at admissible heuristics h that try to approximate h^*, we can say that the following proposition will hold for h too. Each node n_i that is accessible by zero-cost actions from a current node n will need the same positive actions to reach a goal state as the current node n. Therefore the h^*-value of all nodes n and n_i (with $i = 1, \ldots k$) will be the same.

Proposition 2. *The heuristic value of the current node n will be the same as for another node n_k if n_k is reachable from n by only applying zero-cost actions. The set of positive actions needed to get from n_k to the goal node n_g is the same set for n because we can get to n_k without any positive actions.*
If $n \xrightarrow{a_1} n_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} n_k \rightarrow n_g$, where a_i are zero-cost actions with $i = 1, \ldots, k$ and positive actions from n_k to n_g with n_g being a node with a goal state. Then $h^*(n) = h^*(n_1) = \ldots = h^*(n_k)$ because the set of positive actions leading to a goal state are the same for each node.

If we consider this true for all admissible heuristics h, we can find that this leads to the same f-value between the current node n and its successors n_i. Because of that all nodes n_i and the node n are now in the same *plateau(f, h)*. Additionally n can have more than one applicable zero-cost action. With multiple applicable actions we can get to multiple successors from the same node n that end up in the same plateau. All of these successors can again have multiple successors from zero-cost actions and the plateau will grow even more. This behavior can continue further and the plateau can grow level-wise per generation of successors. We will call each new level of successors a new depth layer. In Figure 3.1 we can see an example of such a plateau. We see that the plateau can grow exponentially in depth to the branching factor. In the example of Figure 3.1 the branching factor is three and the plateau contains 3^n nodes with n being the depth value of the deepest depth layer. While expanding the first nodes we get other nodes that are put into the list of open nodes. From those nodes some will be in the same plateau and for them we have to decide which node is the most promising. For that we have to apply another tie-breaker to select a single node to expand next.

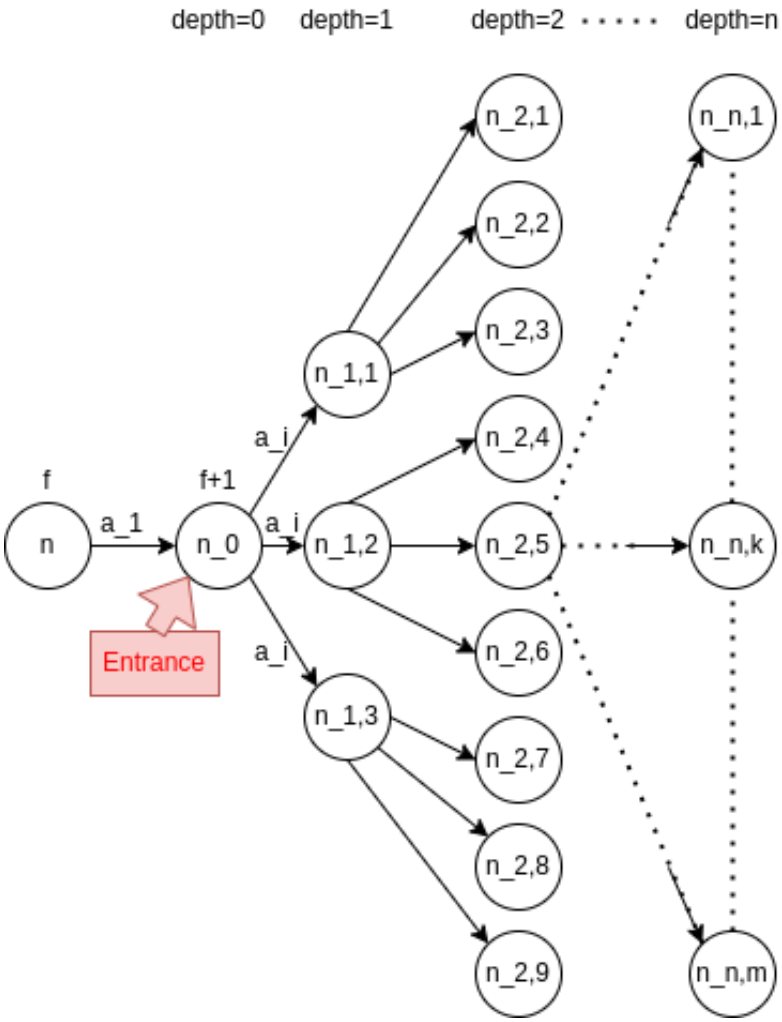

Figure 3.1: A plateau is created with zero-cost actions a_i which do not change the f-value of the successor node. Here the node n does have another f-value while all other nodes in the plateau(f+1, h) = $\{n_0, n_{1,1}, n_{1,2}, \ldots, n_{n,m}\}$ have the same f-value f+1. The action a_1 has a cost greater than zero while all other actions a_i after this have a cost of 0 (zero-cost actions). The node n_0 is the entrance of the plateau as it is the first node with the f-value f+1.

Let us look at a more concrete example. The zero-cost domain GRIPPER-MOVE is about a robot called Robby. The robot has two gripper-arms which can pick up one ball each. We will look at a specific instance from this domain and simply it by reducing the number of gripper-arms to only one. Here we have two different balls 1 and 2. They can be either in room A or room B. Lastly we have the robot Robby. Robby has in this example one gripper-arm which can hold one ball at the time. Both balls are in room A at the initial state and have to be in room B at the goal state. The domain defines three actions: *move, drop* and *pick*. To pick up a ball with the gripper-arm the action *pick* is used and to drop the ball that is held by the arm the *drop* action is used. Both *drop* and *pick* are zero-cost actions. The last action *move* changes the position of the robot from the current room to the other. This is the only positive action in this domain and has a action cost of one.

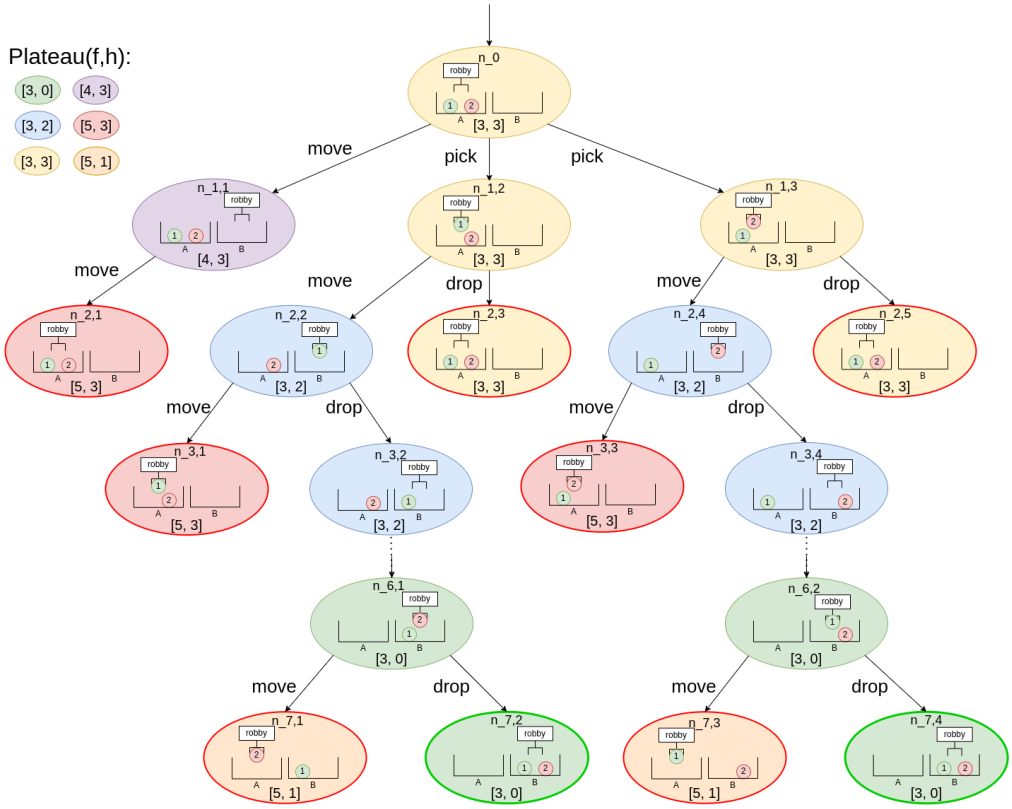

Figure 3.2: An instance of the GRIPPER-MOVE domain as an example of forming a plateau. The actions *pick* and *drop* are zero-cost actions, while only *move* has a cost of one. All nodes that belong to the same *plateau*(f, h^*) have the same color. The nodes with a red outline hold states that where already seen and will not be expanded by the search. The green outline denotes a node that holds a goal state. The graph is not complete because we skipped some nodes for the sake of readability.

In Figure 3.2 we can see a part of the visualization of the instance from the GRIPPER-MOVE example. Let the search algorithm for this example be A^* with the tie-breaking strategies $[f, h^*]$ and $f = g + h^*$. The search starts at the node n_0 that holds the initial state. This starting node n_0 has the h-value $h(n_0) = 3$, because in any optimal path we only need the *move* action three times. The g-value of the starting state is zero, which makes the f-value $f(n_0) = 3$. If we use the *move* action we reach the node $n_{1,1}$. The h-value $h(n_{1,1})$ increases by one because now we would have to apply *move* four times to get to any goal node. Additionally the path cost $g(n_{1,1})$ is increased by one too, which gives us $f(n_{1,1}) = 5$. The f-value of the goal nodes $f(n_{7,2}) = f(n_{7,4}) = 3$ is smaller than $f(n_{1,1}) = 5$. Because of that we know that A^* with an admissible heuristic would not expand $n_{1,1}$ before reaching the goal. The nodes $n_{2,3}, n_{2,5}, n_{3,1}, n_{3,4}, n_{7,1}$ and $n_{7,3}$ are such states too and marked with a red circle in Figure 3.2. All these nodes will never be expanded in this example. So the next actions to look at are either *pick* on the first ball or *pick* on the second. Both ways will lead to a similar path and later to a solution. The node $n_{1,2}$ and $n_{1,3}$ both have the same h-value as the starting node n_0. Additionally because *pick* is a zero-cost action the g-value will not increase either. We find that all three nodes have the same f-values

$f(n_0) = f(n_{1,2}) = f(n_1, 3) = 3$. In this case we already have a plateau from the start with n_0 as the entrance node. We denote it as *plateau(3, 3)* with $f = 3$ and $h = 3$. This plateau is marked yellow in Figure 3.2. Additionally all other plateaus are marked with different colors. Applying *drop* to either $n_{1,2}$ or $n_{1,3}$ will give as another node $n_{2,3}$ or $n_{2,5}$ in the same plateau. The positive action *move* changes the g-value for the nodes $n_{2,2}$ and $n_{2,4}$ respectively. Both nodes have the same path cost $g(n_{2,2}) = g(n_{2,4}) = 1$ and the heuristic value $h(n_{2,2}) = h(n_{2,4}) = 2$. The f-value $f(n_{2,2}) = f(n_{2,4}) = 3$ is still the same as for the nodes $n_0, n_{1,2}$ and $n_{1,3}$ but the h-value does not match anymore. In the case of only looking at tie-breaking strategies $[f]$ without the h-value we would additionally have a *plateau(3)*. Then the nodes $f(n_0) = f(n_{1,2}) = f(n_1, 3) = f(n_{2,2}) = f(n_{2,4}) = 3$ would all be in the same *plateau(3)* because they have the same f-value. But for this example we will only consider $[f, h]$-plateaus where both the f- and the h-value are the same. This means that $n_{2,2}$ and $n_{2,4}$ would be not in the same $[f, h]$-plateau as the others but still in the *plateau(3)* because of the f-value. We apply more actions to the expanded nodes until we reach a node that holds any goal state. In this example the goal nodes are $n_{7,2}$ and $n_{7,4}$. To find the most efficient path we still have to decide which node from the plateaus we should expand first. For that we need to take a closer look at different tie-breaking strategies.

3.2 Default Tie-breaking

Let us first use a default tie-breaking to find the next node to expand. If we use *first-in-first-out* we would order all nodes in order of generation inside a list. Because we do not know how the algorithm chooses which successor gets generated first we can not say correctly how the list would look like. But we can at least say which nodes where before other. For the example in Figure 3.2 we know that the starting node n_0 is generated before its successor nodes $n_{1,1}, n_{1,2}$ and $n_{1,3}$. What we do not know is which of these three successor nodes gets generated first. The nodes that are next in the list will be the successors of these three nodes $n_{1,1}, n_{1,2}$ and $n_{1,3}$. Again we do not know the order of these successor nodes. We only know that they will be expanded only after all predecessors are expanded first. This tie-breaking strategy lets the search algorithm behave the same as a breadth-first search inside a plateau. With this approach we would start at the most shallow levels of depth and gradually continue on to the deepest part of the plateau. If none of the nodes in the depth layers from 0 to n-1 contains any goal state we have to expand almost every node before finding the goal. Because of that we could use *last-in-first-out*. This approach behaves like a depth-first search inside the plateau. The deepest nodes will be expanded first. As the list grows with each expansion *lifo* will choose the current last node in the list. So this tie-breaking strategy focuses on the deeper layers rather than the first layers. The search algorithm would then need much time in the last depth layer with depth = n. This last layer is exponentially bigger than the others as every layer grows exponentially by the branching factor of the tree (in figure 3.1: branching factor = 3). The last default tie-breaking strategy *random* does a better job in looking through the different depth layers evenly but is not structured. All three used default tie-breaking strategies may search in the wrong order and then have to expand a large amount of nodes before finding any goal. This could lead them

to failing to reach the goal in the given time limit.

3.3 Depth-Diversification

To solve these problems of inefficiency Asai and Fukunaga (2017) recommend a new tie-breaking strategy called *depth-diversification*. This depth-diversification is effectively searching evenly over all depth layers. We denote the depth-diversification criterion as $\langle d \rangle$ and write $[f, h, \langle d \rangle]$ for A^* with h tie-break and depth-diversification. The idea is to sort all nodes of a plateau in so called depth buckets. A depth bucket is not more than a priority queue inside the plateau. Each depth bucket corresponds to a depth layer in the plateau. The nodes then get sorted by their depth and put in the corresponding depth bucket. With depth-diversification the depth bucket with the highest depth value is selected at first. The depth value is the number of how many layers the node is away from the entrance of the plateau. A default tie-breaking strategy then selects one of the nodes from the selected depth bucket, which will be expanded next. There is a counter with the highest d-value that is then decreased by one. It shows from which depth bucket the next node should be expanded. The counter starts at the deepest depth bucket and progressively decreases after choosing one node in the current depth bucket. When reaching depth zero it loops around and starts again at the deepest depth bucket. This behavior ensures that all different depth layers get searched evenly. The problem that occurs with *fifo* and *lifo* is solved with the depth-diversification. This is because with depth-diversification there is always a chance of finding the solution in a short amount of time even when it is concentrated near the entrance or at the deeper layers. When going over the `gripper-move` example in Figure 3.2 we find that *lifo* expands the least nodes with 10 expanded nodes. The example runs with the perfect heuristic h^* so it is informed and finds the solution path fast.. But with *fifo* we have to expand 16 nodes while the depth-diversification only needs 13. So for this small example one default tie-breaker is better than the other while the depth-diversification is a good middle ground. For exponentially bigger instances the difference between the expanded nodes can grow exponentially too. This shows that the depth-diversification is more reliable than the default tie-breakers as both *fifo* and *lifo* can be in a position where they expand a large amount of nodes depending on the problem.

To understand better how the depth-diversification works, we can look at the Algorithms 2, 4, 3 and 5. The Algorithm 2 shows the method that gets called when a new node gets generated. The new node n gets inserted into the OpenList with the $key = [f, h]$. The parent node m is needed to compare the f- and h-values of n and m. With these values we can find out if both nodes are in the same plateau or not and update the d-value of the new node n. To find the the right plateau, we use the key that then sorts the node into the corresponding selector.

The method `Add` is described in Algorithm 3 and adds the node n into the depth buckets of the selector. Figure 3.3 shows how the OpenList and the selectors are built. Each circle represents a node inside the depth bucket. The number of nodes inside can vary and does not have to be evenly distributed like the figure might show. The new node gets added to the selector with the key-value as the index. Inside the selector the node gets sorted into

the depth bucket with the d-value of the new node n as the index.

Algorithm 2 Node Insertion into Open List

```
1: procedure DivDoInsertion(current_node, entry, parent_node, depth_value)
2:     if current_node is root node then
3:         Compute h(current_node), f(current_node)
4:         key ← [f(current_node), h(current_node)]
5:         depth_value(current_node) ← 0
6:     else
7:         Compute g(current_node), h(current_node), f(current_node)
8:         key ← [f(current_node), h(current_node)]
9:         if g(current_node) = g(parent_node) and
10:        f(current_node) = f(parent_node) then
11:            Node current_node and parent_node are in same plateau
12:            depth_value(current_node) ← depth_value + 1
13:        else
14:            // New plateau
15:            depth_value(current_node) ← 0
16:        end if
17:    end if
18:    Identify correct Selector for key
19:    Call Selector.Add(entry, depth_value)
20: end procedure
```

Algorithm 3 Selector Add Entry

```
1: procedure Add(entry, depth_value)
2:     if depth bucket at index depth_value does not exist then
3:         Create new depth bucket
4:     end if
5:     Insert entry into bucket at depth depth_value
6:     return
7: end procedure
```

To expand the next node, the search will call the method `RemoveMin` of the OpenList. We can see this method in the Algorithm 4. In the OpenList the selector with the numerical lowest key gets selected. This means we will look at the plateau with the lowest $[f, h]$-value pair. For this selector the method `RemoveNext` gets called. The selecting of the next node to expand is done in the selector which we can see in Algorithm 5.

Algorithm 4 Remove Minimal Entry from Open List

```
1: procedure RemoveMin
2:     Select Selector bucket with lowest key
3:     result ← Selector.RemoveNext(tiebreaking_criteria)
4:     if Selector is empty then
5:         Remove Selector
6:     end if
7:     return result
8: end procedure
```

In the method `RemoveNext` inside the selector, lays the logic of the tie-break. Each selector

has its own counter that points to a depth bucket. In Figure 3.3 the counter points at the depth bucket with d-value of two. So the next node to expand will be one of this depth bucket. Which of these nodes it is, decides the tie-breaking criteria in the switch cases in Algorithm 5. The counter gets decreased at the beginning of the method to avoid problems with the initialization. If the counter gets below zero it gets rewind to the highest d-value that corresponds to an unempty depth bucket. Lastly when the node is selected in the depth bucket, it will be removed from there und returned to the search algorithm.

Algorithm 5 Selector Remove Next Entry

```
1: procedure RemoveNext(tiebreaking_criteria)
2: // The counter gets initialized with -1 for the first node in the Selector
3:     Decrease counter by 1
4:     if counter < 0 then
5:         Rewind counter to deepest non-empty depth bucket
6:     end if
7:     if tiebreaking_criteria == FIFO then
8:         Select and remove first entry from depth bucket
9:     else if tiebreaking_criteria == LIFO then
10:        Select and remove last entry from depth bucket
11:    else
12:        Select and remove random entry from depth bucket // RANDOM
13:    end if
14:    return selected entry
15: end procedure
```

3.4 Implementation

In Figure 3.3 we can see how we implemented the depth-diversification ourselves. The implementation is done in Fast Downward and can be combined with any other tie-breaking strategy. In the OpenList we have different selectors that represent a plateau. These selectors are containers that are indexed by the f- and h-value in the case of A^* with depth-diversification. In general the index could be any combination of tie-breaking criterion up until the second-last criterion. The second-last criterion then has to be the depth value $\langle d \rangle$ and the last has to be a tie-breaker that always chooses exactly one node for example one of the default tie-breakers. So the key that indexes the selector looks like this: $[criterion_1, criterion_2, \ldots, criterion_n]$. We denote a tie-breaking strategy with criterions 1 to n and depth-diversification as $[criterion_1, criterion_2, \ldots, criterion_n, \langle d \rangle, *]$ with * being one of the default tie-breaking strategies. Inside each selector there are the depth buckets that are indexed by the $\langle d \rangle$-value. The nodes that get added to the OpenList will first be sorted by the first n tie-breaking criterions and then put into the depth bucket corresponding to their depth layer.

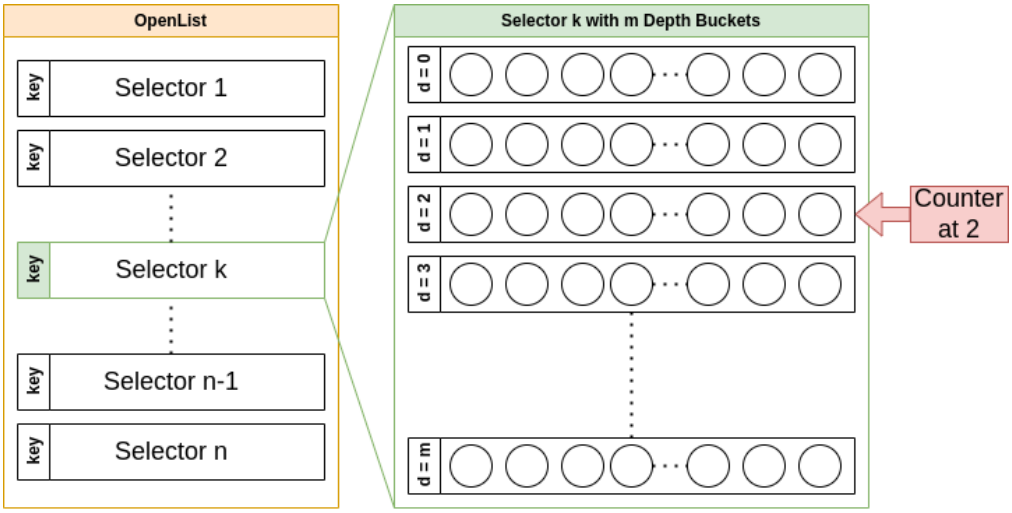

Figure 3.3: With the depth-diversification the OpenList sorts the nodes first by the key $[criterion_1, criterion_2, \ldots, criterion_n]$ into so called selectors. Inside the selectors the nodes get sorted again by the depth value of the plateau. The counter is in this example at the depth layer 2 and one node will be from the list with d = 2. A default tie-breaking strategy will decide which node will be chosen. After that the counter will continue at depth d = 1.

3.5 Distance-to-go

To further improve depth-diversification we can use an inadmissible distance-to-go estimate. A distance-to-go estimate is a heuristic function that uses a heuristic like *LM-cut* or *Merge and Shrink*. But instead of calculating the h-value with the real action costs, it will replace every action cost with one. With this adjustment even the zero-cost actions cost the same as other actions. Because of that the heuristic can predict the number of actions needed until some goal state is reached. We denote $\hat{h}$ as the distance-to-go heuristic of a heuristic h. In general distance-to-go heuristics are inadmissible because they do not underestimate for example zero-cost actions or actions with a cost less than one. The optimality of A^* is still given when the distance-to-go heuristic is only used as a tie-breaker and not to compute the f-value. It does not matter if any criterion after the f-value uses an inadmissible heuristic because the first criterion provides optimality on its own. With this knowledge we can use multi-heuristic strategies like $[f, h, \hat{h}]$ or $[f, \hat{h}]$ as long as $f = g+h$ with h being an admissible heuristic. Furthermore does the second criterion not even have to be related to the heuristic in the first criterion f. We can use *LM-cut* for the computation of f to ensure optimality, while we can use another heuristic for the second criterion. For the second heuristic we can choose the *FF* heuristic as a distance-to-go estimate and denote it as $\hat{h}^{FF}$. The *Fast Forward* heuristic or just *FF* heuristic (Hoffmann and Nebel, 2001) is a planning heuristic that estimates the path cost by ignoring delete effects of actions. With the performing of a greedy forward search, it counts the number of actions needed in the simplified search. In contrary to *LM-cut* this *FF* heuristic is not admissible.

Now if we use a distance-to-go heuristic in our tie-breaking strategy we can get $[f, h, \hat{h}]$, $[f, \hat{h}]$ or $[f, \hat{h}^{FF}]$. But with that we are not done. the distance-to-go criterions can still leave plateaus with the same $\hat{h}$- or $\hat{h}^{FF}$-values and therefore need another tie-breaker. For

that we can use one of the default tie-breaking strategies *fifo*, *lifo* or *ro*. With that we can combine the different tie-breaker together and get $[f, h, \hat{h}, *]$, $[f, \hat{h}, *]$ or $[f, \hat{h}^{FF}, *]$ with $*$ as one of the default tie-breakers. These tie-breaking strategies will improve the search performance in contrast to $[f, h, *]$ as we will later see in Chapter 4. Additionally we can combine the distance-to-go heuristic with the depth-diversification. The $\langle d \rangle$ criterion will be added after the distance-to-go criterion and will be again followed by a default tie-breaker: $[f, \hat{h}^{FF}, \langle d \rangle, *]$. With that the distance-to-go criterion has a bigger impact on the search than the depth-diversification. The distance-to-go criterion will sort the nodes into plateaus where the depth-diversification then can tie-break the nodes that still have the same f- and $\hat{h}^{FF}$-values. Lastly the default tie-breaker is needed to ultimately break the tie and select a single node from the plateau.

4
Evaluation

In order to determine the effectiveness of our implementation we conducted a series of experiments. First we will see how the setup of the experiments was done to then see in the later sections how these experiments turned out. We will look at the impact of our implementation on the zero-cost and IPC-domains and compare them to the experiment results of Asai and Fukunaga (2017).

4.1 Setup

To evaluate the efficiency of the depth-diversification we ran experiments to test the implementation we did in Fast Downward. Each experiment was conducted with a 5 min time limit and a memory limit of 3.5 GiB. The CPU we used for all the experiments is a Core Intel Xeon Silver 4114 2.2 GHz Processor. The instances that these experiments ran on come from Asai and Fukunaga (2017) and contain 28 different zero-cost domains. Overall we have 620 instances with zero-cost actions that we used to run our experiments on. Asai and Fukunaga (2017) created these zero-cost domains based of the IPC-benchmarks. They modified the problems such that the majority of actions are zero-cost actions while always at least one action has a positive action cost. The non-zero actions are chosen to, for example, optimize energy usage for transportation-type domains or minimize resource usage for assembly-type domains. The number of instances per domain is smaller than the IPC-domains with 1104 instances. This is achieved by choosing 5 to 40 instances from each domain. The instances are evenly chosen by difficulty and size of the problem instance. Additionally we also ran all of the experiments on the IPC-domains too. The IPC-domains contain 1104 instances distributed over 35 domains with 5 to 150 instances per domain.
In Chapter 3 we already looked at the different tie-breaking criterions there are. For the experiment we used the tie-breaking strategies listed in Table 4.1.
Every configuration of tie-breaking strategies was run for each default tie-breaking strategy (denoted as $*$ in the Table 4.1).
With the same configurations we can compare our results to the results of Asai and Fukunaga (2017). Every configuration that uses a random tie-break was run ten times and then averaged to get a comparable result. The f-value was computed by $f = g + h$ for all

Strategy	Description
Baseline:	
$[f, h, *]$	standard A^* search with h tie-breaking
$[f, h, \langle d \rangle, *]$	h tie-breaking with depth-diversification
Distance-to-Go:	
$[f, \hat{h}, *]$	tie-breaking with inadmissible distance-to-go heuristic $\hat{h}$
$[f, h, \hat{h}, *]$	combination of standard h and distance-to-go $\hat{h}$ tie-breaking
$[f, \hat{h}^{FF}, *]$	tie-breaking with distance-to-go version of *FF* heuristic
Distance + Depth:	
$[f, \hat{h}^{FF}, \langle d \rangle, *]$	combination of distance-to-go $\hat{h}^{FF}$ and depth-diversification

Table 4.1: Overview of different tie-breaking strategies we used for our experiments. All experiments were done with $f = g + h$, half with $h =$ *LM-cut* and half with $h =$ *Merge and Shrink*.

configurations. For the heuristic h the landmark heuristic *LM-cut* was used. While we focused on the results where the *LM-cut* heuristic was used, we also conducted the same experiments again with the same configurations but replaced *LM-cut* with *Merge and Shrink*. It is interesting to see if the choice of the heuristic functions changes the result, but this thesis focuses more on the different configurations of tie-breaking strategies. For the *Merge and Shrink* heuristic we did not find any settings in the paper by Asai and Fukunaga (2017). Because of that we choose the recommended settings from the documentation[1].
The important parts of the settings can be seen in Table 4.2.

Parameter	Setting
Merge strategy	SCC-DFP
Shrink strategy	non-greedy shrink_bisimulation
Label reduction	exact (before_shrinking=true, before_merging=false)
Max states	50k
Threshold before merge	1

Table 4.2: Settings of the *Merge and Shrink* heuristic using the recommended specifications from the documentation.

4.2 Zero-Cost Domains

First we will look at the results for the zero-cost domains. We will take a closer look at these experiments because we expect to see more interesting results than with the IPC-domains. At first we have to know which configurations we used and how well they performed in the series of experiments. After that we will compare our results with the paper of Asai and Fukunaga (2017). Lastly we will discuss interesting details and anomalies in different zero-cost domains.

[1] https://www.fast-downward.org/latest/documentation/search/Evaluator/#merge-and-shrink_heuristic, accessed on 2025-06-17

Sorting Criteria	LM-cut with zero-cost domains (620)	
	Results	Asai and Fukunaga
Baselines		
$[f, h, \mathit{fifo}]$	241	256
$[f, h, \mathit{lifo}]$	245	279
$[f, h, \mathit{ro}]$	241	261.9 ± 1.4
$[f, h, \langle d \rangle, \mathit{fifo}]$	257	284
$[f, h, \langle d \rangle, \mathit{lifo}]$	247	264
$[f, h, \langle d \rangle, \mathit{ro}]$	258	288.1 ± 1.6
Distance-to-Go		
$[f, \hat{h}, \mathit{fifo}]$	272	295
$[f, \hat{h}, \mathit{lifo}]$	278	303
$[f, \hat{h}, \mathit{ro}]$	275	301.0
$[f, h, \hat{h}, \mathit{fifo}]$	279	305
$[f, h, \hat{h}, \mathit{lifo}]$	282	309
$[f, h, \hat{h}, \mathit{ro}]$	277	305.9 ± 2.1
$[f, \hat{h}^{FF}, \mathit{fifo}]$	313	337
$[f, \hat{h}^{FF}, \mathit{lifo}]$	311	340
$[f, \hat{h}^{FF}, \mathit{ro}]$	314	341.0 ± 2.2
Distance + Depth		
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{fifo}]$	**317**	340
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{lifo}]$	**317**	342
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{ro}]$	316	**344.3 ± 1.8**

Table 4.3: Coverage of different configurations in comparison to Asai and Fukunaga (2017) with ***LM-cut*** heuristic on the **zero-cost domains**.

4.2.1 Configurations in Zero-Cost Domains

Let us first focus on which configuration solved the most instances within the time and memory limits. This coverage is a good indicator on how well the tie-breaking strategy works. The configurations can be assigned to three categories: baseline, distance-to-go and distance + depth. We can see all configurations divided in the categories in Table 4.1. We have the baseline configurations that test the common A^* configuration with $[f, h, *]$. Additionally we add the configurations with the depth-diversification added after the h tie-break. We see that this standard depth-diversification configuration solves more instances than the baseline strategies without $\langle d \rangle$. This shows that the depth-diversification increases the coverage in the zero-cost domains. The distance-to-go category uses $\hat{h}$ as a distance-to-go variant of the h function. Additionally we have the distance-to-go version of the *FF* heuristic as a tie-breaker. The runs with the distance-to-go heuristics $\hat{h}$ scores an even better result compared to the baseline runs. This is topped slightly with the combination of h criterion and $\hat{h}$ criterion. The distance-to-go version of the *FF* heuristic again achieves a higher coverage than the distance-to-go runs before. We learn that the distance-to-go approach for the second or third criterion increases the coverage. Therefore we can conclude that it is effective for zero-cost domains. This can be explained with that the distance-to-go heuristics will try to minimize the number of actions taken until a goal is reached. This is because all actions cost one. In cost-optimal planning this leads the tie-breaking to rather

Sorting Criteria	M&S with zero-cost domains (620)	
	Results	Asai and Fukunaga
Baselines		
$[f, h, \mathit{fifo}]$	284	280
$[f, h, \mathit{lifo}]$	278	301
$[f, h, \mathit{ro}]$	268	287.7 ± 3.2
$[f, h, \langle d \rangle, \mathit{fifo}]$	297	302
$[f, h, \langle d \rangle, \mathit{lifo}]$	290	288
$[f, h, \langle d \rangle, \mathit{ro}]$	295	308.1 ± 2.1
Distance-to-Go		
$[f, \hat{h}, \mathit{fifo}]$	289	308
$[f, \hat{h}, \mathit{lifo}]$	286	305
$[f, \hat{h}, \mathit{ro}]$	288	307.3 ± 1.5
$[f, h, \hat{h}, \mathit{fifo}]$	294	307
$[f, h, \hat{h}, \mathit{lifo}]$	291	306
$[f, h, \hat{h}, \mathit{ro}]$	292	307.8 ± 1.4
$[f, \hat{h}^{FF}, \mathit{fifo}]$	335	336
$[f, \hat{h}^{FF}, \mathit{lifo}]$	328	331
$[f, \hat{h}^{FF}, \mathit{ro}]$	331	**337.9 ± 2.1**
Distance + Depth		
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{fifo}]$	**336**	337
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{lifo}]$	330	333
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{ro}]$	332	**337.6 ± 1.3**

Table 4.4: Coverage of different configurations in comparison to Asai and Fukunaga (2017) with ***Merge and Shrink*** heuristic on the **zero-cost domains**.

select nodes that appear to be closer to a goal than others. Therefore the search algorithm will not expand as many states in a plateau as with the normal version of h. When more states have to be expanded the search takes longer and therefore some instances can not be solved in the limited time given.

Only the combination of $\hat{h}^{FF}$ and depth-diversification solves more instances than the runs with $\hat{h}^{FF}$ alone. The difference between these two configurations is not big but we can still see that depth-diversification seems to improve the coverage on the zero-cost domains. Zero-cost domains benefit from the depth-diversification and even more so from the distance-to-go estimation. Comparing the difference between the default tie-breakers of the different configurations we see that the final default tie-break does not have a big impact on the coverage overall. We find the best performing configurations $[f, \hat{h}^{FF}, \langle d \rangle, \mathit{fifo}]$ and $[f, \hat{h}^{FF}, \langle d \rangle, \mathit{lifo}]$. Both happen to have solved 317 instances out of the 620 instances in the zero-cost benchmark domains.

When looking at the results with the *Merge and Shrink* heuristic we find that the coverage did not change significantly in comparison to the runs with *LM-cut*. The only difference is that *Merge and Shrink* achieves a higher coverage with every configuration. The best performing configuration is $[f, \hat{h}^{FF}, \langle d \rangle, \mathit{fifo}]$ too and the lowest coverage has $[f, h, \mathit{ro}]$. This is the same for the experiments with *LM-cut* on the zero-domains.

Sorting Criteria	**Average Results**	Difference	**Asai and Fukunaga**
$[f, h, *]$	242.33	$\xrightarrow{+5.9\%}$	256.63
$[f, h, \langle d \rangle, *]$	254	$\xrightarrow{+9.7\%}$	278.70
$[f, \hat{h}, *]$	275	$\xrightarrow{+9.0\%}$	299.66
$[f, h, \hat{h}, *]$	279.33	$\xrightarrow{+9.8\%}$	306.63
$[f, \hat{h}^{FF}, *]$	312.66	$\xrightarrow{+8.5\%}$	339.33
$[f, \hat{h}^{FF}, \langle d \rangle, *]$	316.66	$\xrightarrow{+8.0\%}$	342.10

Table 4.5: Relative difference of the averages of the different configurations with ***LM-cut*** between our results and Asai and Fukunaga (2017).

4.2.2 Comparison to Asai and Fukunaga (2017)

When comparing our results to the results from Asai and Fukunaga (2017), we find similar conclusions. In our experiments the configuration with $\hat{h}^{FF}$ in combination with depth-diversification and either *fifo* or *lifo* deliver the highest coverage. The best performing tie-breaking strategy from Asai and Fukunaga (2017) is the same configuration but with *random* default tie-breaking. In general our results have a lower absolute number. We can explain this with the difference in the CPU and machines the experiments ran on. But we also see that for Asai and Fukunaga (2017) the increase from the coverage of the baseline configurations to the highest coverage with $\hat{h}^{FF}$ and depth-diversification is of about 35%. The increase in our experiments is only of 32%. So our implementation gives us a smaller increase.

The averages from the different configurations with *LM-cut* compared relatively to the average results of Asai and Fukunaga (2017) give us a rough overview of how close our results are in comparison to their paper. Factors like the CPU and machine the experiments ran on do not interfere with these relative differences. In Table 4.5 we see that the results from Asai and Fukunaga (2017) are on average about 8 to 9% higher than our results. The configurations without depth-diversification use the unmodified Fast Downward and can be a good pointer to how much the hardware interferes with the results. We find a variance from +5.9% up to +9.8% increase. The configurations with depth-diversification and our modified version of Fast Downward do not exceed this variance. This suggests that our implementation is not responsible for the decrease of coverage even if it is possible that it has some effect on the results. We can see this development in Figure 4.1 where the curve of our zero-cost experiments with *LM-cut* (green) is very similar to the curve from the results of Asai and Fukunaga (2017) (red). The curves of the IPC-domain experiments are discussed later in Section 4.3. The average of the $[f, h, *]$ configuration results from our experiments are higher in relativity to the results of Asai and Fukunaga (2017) in comparison to the other configurations. This explains the higher increase from the baseline experiments to the best performing configuration in their paper.

When we look at the *Merge and Shrink* experiments we find even smaller differences between the paper and our experiments on the zero-cost domains. We can see in Table 4.6 that the variance of the increase is from +1% up to +6.2%. This means that our results are closer to the results of Asai and Fukunaga (2017) with the *Merge and Shrink* heuristic. But we

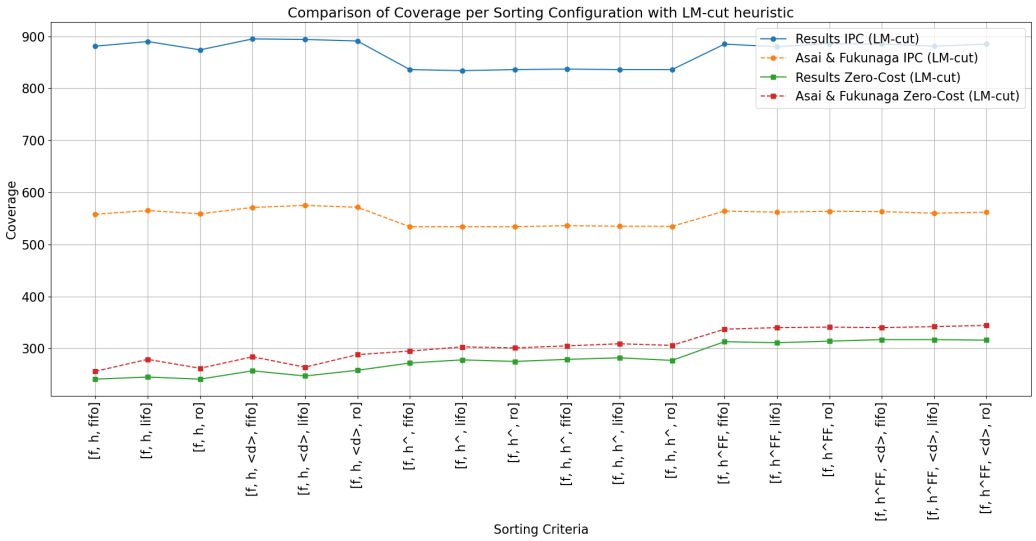

Figure 4.1: This figure shows the coverage in our experiments with ***LM-cut*** compared to the results of Asai and Fukunaga (2017). The results of the zero-cost domains (green and red) can be compared to the IPC-domains (blue and orange). The curves all show a similar development. This shows that our results are similar to the results of Asai and Fukunaga (2017). There is a noticeable difference between the IPC-domains and the zero-cost domains for the first six configurations (baseline runs).

Sorting Criteria	**Average Results**	Difference	**Asai and Fukunaga**
$[f, h, *]$	276.67	$\xrightarrow{+4.7\%}$	289.57
$[f, h, \langle d \rangle, *]$	294.00	$\xrightarrow{+1.8\%}$	299.37
$[f, \hat{h}, *]$	287.67	$\xrightarrow{+6.2\%}$	306.77
$[f, h, \hat{h}, *]$	292.33	$\xrightarrow{+5.0\%}$	306.93
$[f, \hat{h}^{FF}, *]$	331.33	$\xrightarrow{+1.1\%}$	334.97
$[f, \hat{h}^{FF}, \langle d \rangle, *]$	332.67	$\xrightarrow{+1.0\%}$	335.87

Table 4.6: Relative difference of the averages of the different configurations with ***Merge and Shrink*** between our results and Asai and Fukunaga (2017) on the zero-cost domains.

can also see that the difference is low for the configurations with depth-diversification and $\hat{h}^{FF}$ heuristic. The configurations that contain either depth-diversification or $\hat{h}^{FF}$ have a difference of only about 1% while the others have differences from 5 up to 6.2%. This could suggest that our implementation did not increase the coverage as much as expected. But the differences are small and the depth-diversification improvement still achieves the best performance. In Figure 4.2 we can see that the curve for our results (green) is almost identical to the results of Asai and Fukunaga (2017) (red). The largest differences in coverage are for the baseline runs where we achieved less coverage than their paper. For the configurations with the depth-diversification or the $\hat{h}^{FF}$ the results match well.

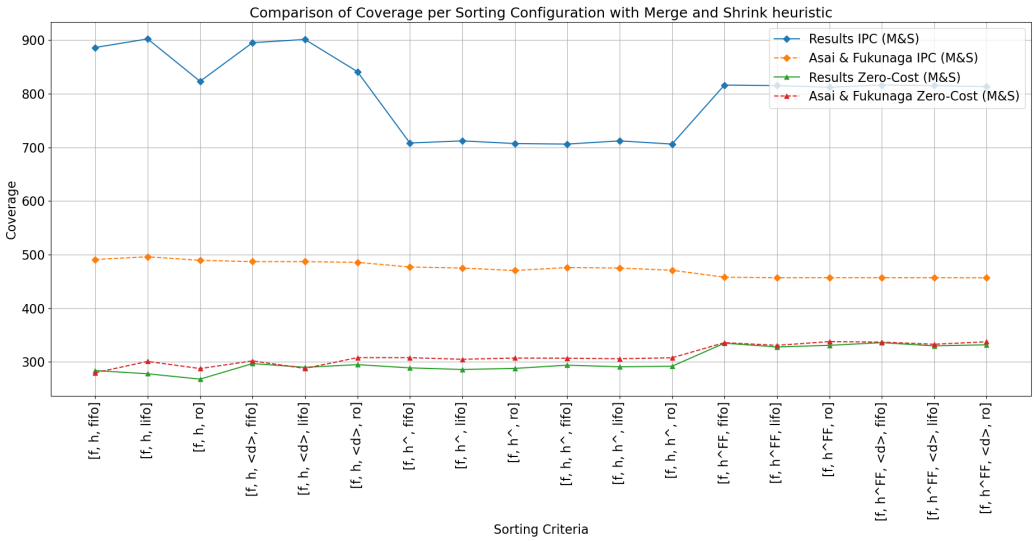

Figure 4.2: This figure shows the coverage in our experiments with ***Merge and Shrink*** compared to the results of Asai and Fukunaga (2017). The results of the zero-cost domains (green and red) can be compared to the IPC-domains (blue and orange). The curves for the zero-cost domains show a similar development while we have a noticeable dip in the IPC-domains for the configurations with $\hat{h}$. Apart from that, our results are similar to the results of Asai and Fukunaga (2017).

4.2.3 Detailed Evaluation

We can look deeper into the individual domains and find some anomalies. In Table 4.7 we can see the best and worst performing configurations for the different zero-cost domains. Most domains do not have great differences between the configurations. Because of that we only listed the domains that looked interesting.

In the ELEVATORS-UP domain the distance-to-go approaches achieve the highest coverage with a large gap to the standard $[f, h, *]$ configurations. This domain tries to solve the problems of bringing people from one floor to the other with elevators. This can be explained

Domain	**Best performing**		**Worst performing**	
	Strategy	Coverage	Strategy	Coverage
elevators-up	$[f, \hat{h}, *]$, $[f, h, \hat{h}, *]$, $[f, \hat{h}^{FF}, \langle d \rangle, *]$	20	$[f, h, *]$	7
miconic-up	$[f, \hat{h}^{FF}, \mathit{lifo}]$, $[f, \hat{h}^{FF}, \langle d \rangle, \mathit{lifo}]$	22	$[f, \hat{h}, \mathit{fifo}]$, $[f, h, \hat{h}, \mathit{fifo}]$	14
mprime-succumb	$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{fifo}]$	32	$[f, h, \mathit{lifo}]$	14
parking-movecc	$[f, \hat{h}^{FF}, *]$, $[f, \hat{h}^{FF}, \langle d \rangle, *]$	20	baseline	0
scanalyzer-analyze	$[f, h, \hat{h}, \mathit{lifo}]$	18	$[f, h, \mathit{ro}]$, $[f, \hat{h}, \mathit{fifo}]$	9
woodworking-cut	distance-to-go & depth	20	baseline	5-10

Table 4.7: Best and worst performing tie-breaking strategies per domain for **zero-cost domains** with ***LM-cut***. Only the domains with interesting differences are listed.

as the only non-zero action in the ELEVATORS-UP domain is the action *move-up-fast*. That means that you can get to the goal with the other three actions *move-up-slow, move-down-fast* and *move-down-slow*. With this many zero-cost actions there are many nodes that will lead to a goal without adding cost. For this the distance-to-go approach minimizes the number of actions and therefore the number of nodes that get expanded. The standard strategy with only the heuristic as a tie-breaker can not differentiate which zero-cost action is the best when many nodes have a h^*-value of zero.
The MICONIC-UP domain is a very similar domain that contains people that want to move to their goal floor and one lift that can move those people. Here the action *up* is the only non-zero action so moving the lift down, boarding and departing all cost zero. In contrast to the ELEVATORS-UP domain, we see that not all distance-to-go configurations achieved a high coverage. We have a noticeable difference between the configurations with $\hat{h}^{FF}$ and the ones with $\hat{h}$. Even the standard baseline configurations performed better. This could be because of the overhead of computing $\hat{h}$ which is still about as uninformed as the heuristic h. The $\hat{h}^{FF}$ configuration has overhead too, but it is robuster against a high amount of zero-cost actions in the domain. Additionally $\hat{h}^{FF}$ works with delete-relaxation what ignores the negative effects of actions. In a elevator or lift problem this means that the lift is seen as staying simultaneously in every floor it has seen before. This brings a huge advantage because it can ignore many actions for moving back to floors where the lift already was.
For the domains MPRIME-SUCCUMB, PARKING-MOVECC and WOODWORKING-CUT we can come to the same conclusion. We find that the configurations with the distance-to-go and the depth-diversification improvements achieve far better coverage as the standard baseline configurations. This is not surprising as the improvements specifically where made to solve instances of zero-cost domains. Noticeable is that none of the baseline configurations where able to solve a single PARKING-MOVECC instance while the configurations with $\hat{h}^{FF}$ solved all 20 instances in the given time. In this domain the goal is to park some cars in the right order. There are curbs where a car $c1$ can be parked and then another car $c2$ can park behind the car $c1$ in the curb. The actions *move-curb-to-car, move-car-to-car* and *move-car-to-curb* are all zero-cost actions while *move-curb-to-curb* is the only non-zero action in the domain. Each instance can be solved with only zero-cost actions. This is the same with the ELEVATORS-UP domain. These two domains are the only ones that have a total cost of zero when solved optimally. For these problems the standard baseline configurations are not suited as they can not differentiate which path expands less nodes. The distance-to-go approaches can identify better which path needs more actions and therefore expands more nodes.
Lastly the we will look at the SCANALYZER domain. This planning domain was introduced by Helmert and Lasinger (2010) for the sixth International Planning Competition (IPC-2008). The problem that has to be solved in the domain is the scanning and analyzing of plants in a greenhouse. For that conveyer belts have to transport a batch of plants from the greenhouse into a imaging chamber. Each batch of plants is placed on one conveyer belt and has to be moved at least once to the imaging chamber. The batch size can vary but we will consider the batch size as half the amount of plants that fit on one conveyer belt. The zero-cost variant SCANALYZER-ANALYZE of this domain contains the zero-cost

Configuration	fifo	lifo	random	Factor
$[f, h, *]$ with instance `p11.pddl`	**52**	19'386	11'591.20	$\times$ 372.80
$[f, \hat{h}, *]$ with instance `p15.pddl`	1'548	**29**	1'444.30	$\times$ 53.37
$[f, h, \hat{h}, *]$ with instance `p14.pddl`	27'919	**519**	7'999.86	$\times$ 53.79

Table 4.8: The number of expansions of specific instances of the SCANALYZER-ANALYZE domain that show a noticeable difference between the final default tie-breakers.

actions *rotate-2* and *rotate-4*. These two actions can swap two batches on the conveyer belts. Additionally there are the non-zero actions *analyze-2* and *analyze-4*. The actions with **-4* are used for larger rotations when we have two batches on one conveyer belt. In the experiments we find some differences in the results. The configurations in the Table 4.8 use the $\hat{h}$ tie-breaker in both the best and the worst performing strategy. The difference here lies in the final tie-breaker. Table 4.8 shows three instances where there is a noticeable difference in number of expanded nodes between *fifo* and *lifo*. For most instances of the same configuration either *fifo* or *lifo* is strongly better than the other. The three instances in Table 4.8 just show the extreme cases. For $[f, h, *]$ the worst coverage over all instances is in combination with *ro*. The instance `p11.pddl` expands the most nodes with *lifo* while *fifo* expands 372.8 times less nodes. The other worst performing strategy $[f, \hat{h}, *]$ is worst in combination with *fifo*. In the instance `p15.pddl` we see this with a magnification factor of 53.37 times the expanded nodes of the result with *lifo*. A similar factor can be seen with the $[f, h, \hat{h}, *]$ configuration in the instance `p14.pddl`. But this configuration performs best in the SCANALYZER-ANALYZE domain. For the coverage the difference in expansion number does not matter in this configuration as *fifo* solves 17 and *lifo* 18 instances. The order of expansion is important for the *fifo/lifo* tie-breaking. But we do not know which action the algorithm applies first. This is a matter of the internal implementation. Because of that we can not be sure which nodes are put into the OpenList first when the previous tie-breaks are the same. In all three cases this order of expansion had a significant impact on the amount of expansions in total. Therefore we can conclude that the order of applying the actions can matter in certain scenarios and can increase the number of nodes expanded.

Let us look at another example where the final tie-breaker made a significant difference. The PATHWAYS-FUEL domain solves the problem with sequences of chemical reactions in a biological organism. In the domain we have the zero-cost actions *choose, initialize* and *synthesize* that are needed to make a molecule available for a chemical reaction. The only two non-zero actions *associate* and *associate-with-catalyze* are there to use up two molecules to make a third new molecule. In the experiments none of the configurations was able to solve more than five instances out of the 30 existing. This instance `p05.pddl` shows some significant differences between the configurations with different default tie-breakers. For the baseline configurations $[f, h, *]$ and $[f, h, \langle d \rangle, *]$ we can explain the difference by stating that *lifo* works like a depth-first-search inside the plateau. There the deeper nodes hold states that have more molecules available to react with. This is because the zero-cost actions can accumulate molecules without increasing the cost. Then at a deeper layer the non-zero actions can be applied easier because the preconditions are more likely to match. But this does not explain why the configurations $[f, \hat{h}, *]$ and $[f, h, \hat{h}, *]$ expanded so much more

Configuration	fifo	lifo	random	Factor
$[f, h, *]$	4'173'908	**290**	566'145.80	× 14'392.78
$[f, h, \langle d \rangle, *]$	325'339	**423**	1'114'393.50	× 2'634.50
$[f, \hat{h}, *]$	**55**	156'712	39'234.50	× 2'849.31
$[f, h, \hat{h}, *]$	**43**	156'712	*none*	× 3'644.47

Table 4.9: The number of expansions of the instance `p05.pddl` of the PATHWAYS-FUEL domain. The results show a significant difference between the final default tie-breakers.

nodes with *lifo* but needed even less expansions with *fifo*. The distance-to-go heuristic $\hat{h}$ makes the paths with many zero-cost actions costly as $\hat{h}$ is computed with all unit action cost. This makes the nodes that are closer to the current node more attractive in the sense of cost-minimization. So the search finds paths that only apply the necessary zero-cost actions faster with *fifo*. The paths with many zero-cost actions will be only expanded if the closer nodes already got expanded. Even tough one of the default tie-breakers performed significantly better than the other with certain configuration, no configuration was able to solve the instance `p06.pddl`. While the instance `p05.pddl` needs six variables to be assigned in the goal state, the instance `p06.pddl` increases the amount to eight variables. This of course increases the difficulty but still does not explain why the well performing configurations from the instance `p05.pddl` can not solve the problem within the memory and time limitations. This example shows that we can never know which default tie-breaking works best for a general problem. It is always dependent on the domain. For that reason we implemented teh depth-diversification that allows us to not rely on luck wether *fifo* or *lifo* is better for any specific problem. The instance `p05.pddl` of the PATHWAYS-FUEL domain shows this well, because the number of expansions for the configuration $[f, \hat{h}^{FF}, \langle d \rangle, *]$ are balanced with *fifo:* 31, *lifo:* 31 and *ro:* 31.80.

4.3 IPC-Domains

For the IPC-domains we used the same configuration of tie-breaking strategies as before with the zero-cost domains. The coverage of the different configurations can be seen in Table 4.10. We find a small increase of coverage for the baseline runs with depth-diversification $[f, h, \langle d \rangle, *]$ in comparison to the standard runs with $[f, h, *]$. These runs with the depth-diversification are overall the best performing ones with $[f, h, \langle d \rangle, \mathit{fifo}]$ achieving the highest coverage. The configurations with the distance-to-go heuristic $\hat{h}$ did not perform as well as the other configurations. When we compare the average of the runs with $\hat{h}$ with the average of the baseline runs we find a decrease of coverage of almost 6%. For the both $[f, \hat{h}^{FF}, *]$ and $[f, \hat{h}^{FF}, \langle d \rangle, *]$ we find a decrease of more than 5%. These last two configurations both share the same average. When comparing the configurations $[f, \hat{h}^{FF}, *]$ and $[f, \hat{h}^{FF}, \langle d \rangle, *]$ we can not find any big differences even when looking at the individual domains. The distance-to-go approaches on their own do not perform as well as the combination of the distance-to-go heuristics with depth-diversification. We can explain this based on the fact that the search algorithm has to compute two different heuristic values for each state. This is because we use another heuristic for the h that is used to get f and for the $\hat{h}$. The generated overhead from

Sorting Criteria	LM-cut with IPC domains (1104)	
	Results	Asai and Fukunaga
Baselines		
$[f, h, \mathit{fifo}]$	881	558
$[f, h, \mathit{lifo}]$	890	565
$[f, h, \mathit{ro}]$	874	558.9 ± 2.1
$[f, h, \langle d \rangle, \mathit{fifo}]$	**895**	571
$[f, h, \langle d \rangle, \mathit{lifo}]$	894	**575**
$[f, h, \langle d \rangle, \mathit{ro}]$	891	571.4 ± 1.7
Distance-to-Go		
$[f, \hat{h}, \mathit{fifo}]$	836	534
$[f, \hat{h}, \mathit{lifo}]$	834	534
$[f, \hat{h}, \mathit{ro}]$	836	534 ± 2.1
$[f, h, \hat{h}, \mathit{fifo}]$	837	536
$[f, h, \hat{h}, \mathit{lifo}]$	836	535
$[f, h, \hat{h}, \mathit{ro}]$	836	534.7 ± 1.5
$[f, \hat{h}^{FF}, \mathit{fifo}]$	885	564
$[f, \hat{h}^{FF}, \mathit{lifo}]$	880	562
$[f, \hat{h}^{FF}, \mathit{ro}]$	886	563.7 ± 1.4
Distance + Depth		
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{fifo}]$	885	563
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{lifo}]$	881	560
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{ro}]$	885	561.9 ± 1.4

Table 4.10: Coverage of different configurations in comparison to Asai and Fukunaga (2017) with ***LM-cut*** heuristic on the **IPC domains**.

the computation of the second heuristic is is most likely the cause of the decreased coverage. This overhead explains why $[f, \hat{h}^{FF}, *]$ and $[f, \hat{h}^{FF}, \langle d \rangle, *]$ are so similar. The computation of the distance-to-go heuristic uses up so much resources that the search reaches the limits. The depth-diversification is not able to increase the performance because the plateaus get too big before the tie-break even happens.. Therefore the depth-diversification can not make any difference in these two settings. But because the *FF* heuristic is a powerful heuristic function itself, we still manage to get a comparable result to the baseline configurations without depth-diversification. Comparing it to the zero-cost domains we see that with zero-cost actions the distance-to-go heuristics increase the coverage. In the IPC-domains this increase is not notable. But the depth-diversification has a larger impact on the coverage increase as in the zero-cost domains.

When we compare the default tie-breakers in the different configurations, we see *lifo* does not perform as well in the distance-to-go configurations. In contrary in the standard baseline runs *lifo* outperforms the other two default tie-breaker. For the baseline run with depth-diversification, the results are rather balanced between *fifo* and *lifo*.

Now let us look at the overall results with the heuristic *Merge and Shrink* on the zero-cost domains. We find that the baseline configurations $[f, h, *]$ and $[f, h, \langle d \rangle, *]$ achieved the highest coverage. From those configurations $[f, h, \mathit{lifo}]$ performed best with 902 out of 1'104 instances. Slightly worse was the configuration $[f, h, \langle d \rangle, \mathit{lifo}]$ with 901 of 1'104 instances

Sorting Criteria	Merge and Shrink with IPC domains (1104)	
	Results	Asai and Fukunaga
Baselines		
$[f, h, \mathit{fifo}]$	886	491
$[f, h, \mathit{lifo}]$	**902**	**496**
$[f, h, \mathit{ro}]$	823	489.4 ± 1.0
$[f, h, \langle d \rangle, \mathit{fifo}]$	895	487
$[f, h, \langle d \rangle, \mathit{lifo}]$	901	487
$[f, h, \langle d \rangle, \mathit{ro}]$	841	485.6 ± 1.5
Distance-to-Go		
$[f, \hat{h}, \mathit{fifo}]$	708	477
$[f, \hat{h}, \mathit{lifo}]$	712	475
$[f, \hat{h}, \mathit{ro}]$	707	470.4 ± 0.9
$[f, h, \hat{h}, \mathit{fifo}]$	706	476
$[f, h, \hat{h}, \mathit{lifo}]$	712	475
$[f, h, \hat{h}, \mathit{ro}]$	706	470.9 ± 0.9
$[f, \hat{h}^{FF}, \mathit{fifo}]$	816	458
$[f, \hat{h}^{FF}, \mathit{lifo}]$	815	457
$[f, \hat{h}^{FF}, \mathit{ro}]$	812	457 ± 1.3
Distance + Depth		
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{fifo}]$	816	457
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{lifo}]$	815	457
$[f, \hat{h}^{FF}, \langle d \rangle, \mathit{ro}]$	813	456.8 ± 1.2

Table 4.11: Coverage of different configurations in comparison to Asai and Fukunaga (2017) with ***Merge and Shrink*** heuristic on the **IPC domains**.

solved. But on average the baseline configurations with depth-diversification $[f, h, \langle d \rangle, *]$ with 879 solved instances performed better than the baseline configurations without depth-diversification (870.33 instances). The average was taken over the three default tie-breakers. It stands out that the configurations with the default tie-breaker *random* achieved noticeable less coverage than the other two default tie-breakers. But this is only for the baseline configurations. In all other configurations the *random* tie-breaker performs similar to *fifo* in most cases.

The distance-to-go runs with $\hat{h}$ achieved lower coverage than all other runs. But the configurations with $\hat{h}^{FF}$ increase the coverage to the same level as the configurations $[f, \hat{h}^{FF}, \langle d \rangle, *]$. This is the opposite of the results from Asai and Fukunaga (2017). Our results show on average an increase of almost 15% from the $\hat{h}$-distance-to-go configurations to the $\hat{h}^{FF}$ configurations with and without depth-diversification. In the paper of Asai and Fukunaga (2017) they state a decrease of the coverage of 4%. These differences are most likely because we used *Merge and Shrink* with our own settings. It is possible that Asai and Fukunaga (2017) used different settings and because of that also achieved different results.

4.3.1 Comparison to Zero-Cost Domains

In Figure 4.1 we can see that the overall curve development is similar for all four curves with *LM-cut*. The most noticeable difference between the zero-cost and IPC-domains are the first six configurations. These baseline runs $[f, h, *]$ and $[f, h, \langle d \rangle, *]$ perform better in the IPC-domains than the distance-to-go runs $[f, \hat{h}, *]$ and $[f, h, \hat{h}, *]$. But for the zero-cost domains this is the opposite case. We find that the baseline configurations work better in the IPC-domains than in the zero-cost domains. Noticeable here is that we have a dip for the configuration $[f, h, ro]$ for both the IPC- and teh zero-cost domains. For the configurations with $\hat{h}^{FF}$ the curve goes up again for both domain sets. This shows that the baseline configurations have trouble in solving zero-cost domain problems more than solving IPC-domains. This makes sense as the h-value can not differentiate between two paths that contain many zero-cost actions and will therefore choose either path with an uninformed default tie-breaking. For the baseline runs the following paths look equally promising:

PathA: $n_0 \xrightarrow{a_1} n_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} n_n \xrightarrow{b} n_g$ with $a_1, \ldots, a_n$ as zero-cost actions, b as a non-zero action and n_g as a goal.

PathB: $n_0 \xrightarrow{b} n_g$ with b as a non-zero action and n_g as a goal

The runs for the distance-to-go combination with depth-diversification do not change the development of the curve much for either of the results.

If we look at the specific domains we can compare SCANALYZER-08-STRIPS and SCANALYZER-OPT11-STRIPS with the modified zero-cost domain SCANALYZER-ANALYZE. The main difference between those domains is of course that in SCANALYZER-ANALYZE only one action has a non-zero cost. The other two domains from the IPC-domain set only have non-zero actions. We find significant differences in SCANALYZER-ANALYZE between the different default tie-breaks as the final tie-breaking strategy. In the IPC-domains of the SCANALYZER problem we do not find any noticeable anomalies. So the zero-cost actions can be identified as the cause of the anomalies.

5
Conclusion

The aim of this thesis was to implement the depth-diversification tie-breaking within Fast Downward. With that we wanted to see if the depth-diversification performs better than the original implementation with the default tie-breaker *fifo*. Additionally we wanted to improve the performance with different configurations with distance-to-go heuristics, especially for zero-cost domains of Asai and Fukunaga (2017) where bigger plateaus appear more often than in the IPC-domains. Bigger plateaus increase the importance of tie-breaking strategies as the default tie-breakers are often only effective for certain problems.

5.1 Configuration Comparison

We first want to look at which configurations solved the most instances in the different domains. We found out that there can be a significant difference between *fifo* and *lifo* as a final tie-breaker but otherwise have the same configurations. The baseline configuration with $[f, h, *]$ show significantly worse results in the zero-cost domains compared to other configurations with distance-to-go heuristics and depth-diversification. The average coverage achieved by $[f, h, *]$ is only 242.33 out of 620 zero-cost instances which is only about 39% of all instances solved. In contrast, the baseline configurations perform better than the distance-to-go configurations with $\hat{h}$ in the IPC-domains. There we have a average coverage of 887.5 out of 1'104 IPC-instances (80.39%) over all baseline configurations. But in both zero-cost and IPC-domains the configurations with $\hat{h}^{FF}$ achieve a high coverage. In the zero-cost domain with an average of 314.66 of 620 (50.75%) and in the IPC-domains with 883.67 of 1'104 (80.04%) over the configurations $[f, \hat{h}^{FF}, *]$ and $[f, \hat{h}^{FF}, \langle d \rangle, *]$. Combining $\hat{h}^{FF}$ with depth-diversification improves the results even more. In the zero-cost domains the configuration of $[f, \hat{h}^{FF}, \langle d \rangle, *]$ achieve the highest coverage. This shows that both the improvement through $\hat{h}^{FF}$ and the depth-diversification is effective for zero-cost domains. These improvements also increase the coverage for IPC-domains. But for the IPC-domains the best performing algorithm is the baseline configuration with the depth-diversification improvement $[f, h, \langle d \rangle, *]$. This baseline configuration with depth-diversification solves on average 893.33 of 1'104 instances (80.92%) and is therefore slightly better performing than the configurations with $\hat{h}^{FF}$. We find similar results compared to the results of Asai and

Fukunaga (2017). But our absolute values for coverage in the zero-cost experiment were about 5 to 9% less than in the paper from Asai and Fukunaga (2017) with *LM-cut*. For *Merge and Shrink* this decrease is reduced even more to about 1 to 6%. These result comparisons suggest hardware-related factors as the reason for the difference, because the relative improvements remain consistent. The experiments show that the choice of the tie-breaking strategies is indeed important and can make significant differences in the results.
We showed that the depth-diversification and the distance-to-go improvements did indeed improve the amount of solved instances. This was expected because the depth-diversification prevents the search from wasting too much time and resources in either shallow (breadth-first like) or deep (depth-first like) layers of a plateau. Depth-diversification ensures more balanced explorations across all depth layers. This is particularly significant in zero-cost domains where the plateaus can grow exponentially. Additionally the distance-to-go heuristic $\hat{h}^{FF}$ is beneficial for domains like ELEVATORS-UP or MICONIC-UP. In these domains the delete-relaxation of the *FF* heuristic allows the elevator to appear simultaneously on all floors it has visited. In this relaxed state of the problem, the need to account for the expensive moving actions is omitted. This enables the $\hat{h}^{FF}$ heuristic to focus on the number of depart and board actions that make the difference in a plateau. Because of that $hath^{FF}$ provides a better performance for distinguishing between different paths with many zero-costs. A distance-to-go heuristic can differentiate between paths with a different number of required actions even when the action all have a cost of zero.

5.2 Unexpected Findings

Let us look at more specific into some domains. The domains ELEVATORS-UP and PARKING-MOVECC both require only zero-cost actions to reach any goal node. Because of that the optimal solution of each instance has a total cost of zero. The domains with this characteristics showed the biggest differences in coverage between the different configurations. In the PARKING-MOVECC domain both configurations with $\hat{h}^{FF}$ achieved a coverage of 20 out of 20 instances while all baseline configurations could not even solve a single instance. This example shows that the baseline configurations can be trapped in big plateaus because of zero-cost actions. While the distance-to-go configurations can navigate through the paths more efficiently without creating unnecessary big plateaus.
Another interesting finding is that in the SCANALYZER-ANALYZE and also the PATHWAYS-FUEL domain the differences between the final default tie-breakers were greater than expected for some instances. The biggest difference in number of expansions was achieved with the baseline configuration $[f, h, *]$ where we have a increase from $[f, h, \mathit{lifo}]$ (290 expansions) to $[f, h, \mathit{fifo}]$ (4'173'908 expansions) by a factor of over 14'000. This finding shows the unpredictability of action ordering inside the search.

5.3 Future Work

We will take a look at further topics that could be studied and improvements that could be made in connection with this thesis. One thing that could be extended is the configuration

of the distance-to-go heuristics. At the moment the distance-to-go heuristic $\hat{h}$ takes the original heuristic h and replaces every action cost with the value one. This weights only the amount of actions taken and does not consider the cost of the action itself. Corrêa et al. (2018) proposed the idea of adding a value c to the action-cost instead of replacing it with one. With that Corrêa et al. (2018) expand the experiments of Asai and Fukunaga (2017) with a cost adaption approach. This approach values the action-cost while still accounting for the amount of the actions needed. It could lead to an interesting result to see if this idea that adapts the the action cost by c performs better than the distance-to-go heuristic $\hat{h}$. Additionally it would be interesting to see if the cost adaption improves or reduces the coverage depending on the values of c.

Another idea would be to test out more heuristics and take a deeper look at the differences they make. For this thesis we focused on one heuristic (*LM-cut*) and only shortly compared the results to the findings with the *Merge and Shrink* heuristic. Other heuristic functions may be able to achieve even better results with the configurations we tested in the experiments.

Lastly Asai and Fukunaga (2017) mention that they also tried the depth-diversification with a randomizer instead of the counter. This could increase the performance as well, because the different depth-buckets would be selected randomly. The random selection omits the problem that *fifo* and *lifo* have with their breadth-first and depth-first like search inside a plateau. The depth-diversification with randomizer could still find the solution in reasonable time without getting stuck at the lower or higher depth layers. But as a randomizer is not deterministic, this would not be suitable to show in a thesis because the results would have to be tested more rigorously to show a consistent result.

Bibliography

Asai, M., and Fukunaga, A. Tie-breaking strategies for cost-optimal best first search. *Journal of Artificial Intelligence Research (JAIR)*, 58:67–121, 2017.

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

Corrêa, A. B.; Pereira, A. G.; and Ritt, M. Analyzing tie-breaking strategies for the A* algorithm. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4715–4721, 2018.

Hansen, E. A., and Zhou, R. Anytime heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 28:267–297, 2007.

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

Helmert, M. The Fast Downward planning system. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246, 2006.

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

Helmert, M., and Lasinger, H. The scanalyzer domain: Greenhouse logistics as a planning problem. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 234–237, 2010.

Helmert, M.; Haslum, P.; Hoffmann, J.; et al. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 176–183, 2007.

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