University of Basel

Mutex Based Potential Heuristics

Bachelor thesis

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

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Salomé Eriksson

Salome Müller
salo.mueller@unibas.ch
2017-063-058

06. 11. 2020

Acknowledgements

First, I want to thank Prof. Dr. Malte Helmert for the opportunity of writing my bachelor thesis in the AI group at the University of Basel. I am very grateful for the support of Dr. Salomé Eriksson, who patiently answered all my questions and guided me through this work. I would like to acknowledge Dr. Florian Pommerening for helping me understand the LP constructor of Fast Downward.
Further, I received a great deal of support and assistance from Lucas Galery Käser, as well as from my flatmates. Thank you all for keeping me nourished.
Last, I want to thank Ada Lovelace, the first programmer there was, for being an inspiration to women in computer science.
Calculations were performed at sciCORE (http://scicore.unibas.ch/) scientific computing center at University of Basel.

Abstract

One dimensional potential heuristics assign a numerical value, the potential, to each fact of a classical planning problem. The heuristic value of a state is the sum over the potentials belonging to the facts contained in the state. Fišer et al. (2020) recently proposed to strengthen potential heuristics utilizing mutexes and disambiguations. In this thesis, we embed the same enhancements in the planning system Fast Downward. The experimental evaluation shows that the strengthened potential heuristics are a refinement, but too computationally expensive to solve more problems than the non-strengthened potential heuristics.
The potentials are obtained with a Linear Program. Fišer et al. (2020) introduced an additional constraint on the initial state and we propose additional constraints on random states. The additional constraints improve the amount of solved problems by up to 5%.

Table of Contents

1 Introduction

One dimensional potential heuristics assign a potential, i.e., a numerical value, to each fact of a classical planning problem. They can be used in a heuristic search, the most common approach to solve such problems. The potentials are obtained with a Linear Program, which optimizes the potentials according to an optimization function. Different optimization functions yield different heuristics. The heuristic value of a state is the sum over the potentials of the fact in this state.
In this thesis, we reproduce the work of Fišer et al. (2020) who proposed to strengthen potential heuristics with mutexes and disambiguations. Mutexes are sets of facts, which can never appear together in any reachable state. They can be used to build disambiguation sets for partial states. They contain all remaining facts which can be assigned to this partial state, i.e., which are not mutex with any of the facts in the partial state.
As introduced by Fišer et al. (2020) we use mutexes and disambiguations to build a less restricted Linear Program. This does lead to better heuristics but the computation is too expensive and slightly less problems are solved.
Next, we use mutexes and disambiguations to strengthen the optimization functions. This leads to good single and ensemble heuristics, however not more problems can be solved, than with the non-strengthened optimization functions.
Further, we add the additional constraint on the initial state to the constraints of the LP, as Fišer et al. (2020) proposed. This does indeed enhance the performance, as the resulting heuristics solve more problems as the same heuristics computed without the additional constraint.
Taking this idea one step further, we add additional constraints on random states. Therefore, random states are generated with random walks, and constraints gained by optimizing the LP for this states are added. These additional constraints do not enhance the performance of the resulting heuristics in comparison to the constraint on the initial state. They are, however, better than using no additional constraint at all, and could be further enhanced.
Last, we compare our results to the evaluation of Fišer et al. (2020). It turns out, that the translation and pre processing of the planning tasks they use, as well as their way for generating of the potentials, are faster than ours.

2 Background

The goal of this chapter is to define and explain the terminology used in this thesis. For visualization, we use the 8-Tiles problem as an example. This is a classical planning problem, in which 8 tiles are arranged in a 3x3-grid. One spot remains empty, the goal is to bring the tiles in a specific order by sliding them around.

2.1 Planning Tasks

In order to solve a problem with an algorithm, we first need to establish mathematical expressions which represent the problem.

We define $\mathcal{V}$ as the finite set of **variables**, where each of the variables $V \in \mathcal{V}$ has a finite set of values called the **domain** $\mathrm{dom}(V)$. For 8-Tiles, the variables could be defined as the 9 fields in the grid $(v_1, \dots, v_9)$, and their domains hold the values of all tiles and the blank space (1 to 8 and 0 for the blank tile). A **fact** $f = \langle V, v \rangle$ consists of a variable $V \in \mathcal{V}$ and one of its values $v \in \mathrm{dom}(V)$. The fact for tile number 5 being in the first position would be $\langle v_1, 5 \rangle$. The set $\mathcal{F}_V$ holds all possible facts of variable $V \in \mathcal{V}$ while $\mathcal{F}$ is the set of all facts of this problem.

A **partial state** p of size t contains t facts of t different variables, i.e., it is the variable assignment over the variables $\mathrm{vars}(p) \subseteq \mathcal{V}$ with $|\mathrm{vars}(p)| = t$. We denote the value assigned to V in p as $p[V]$. In other words, $p = \{\langle V, p[v] \rangle | V \in \mathrm{vars}(p)\}$. If a partial state s is fully assigned i.e., $\mathrm{vars}(s) = \mathcal{V}$, we call s a **state**. A state s **extends** the partial state $p \subseteq s$, if $s[v] = p[v]$ for all $v \in \mathrm{vars}(p)$. The partial state $p = \{v_1 \mapsto 0, v_2 \mapsto 1\}$ represents all states where the first grid in the field of the 8-Tiles puzzle is the blank space while tile number one lies in the second field.

I is the **initial state**, in 8-Tiles this is some specific random order of the tiles. G is a partial state representing the **goal**. State s is a **goal state**, if it is an extension of G. In 8-Tiles G is a state representing one specific order of the tiles e.g. sorted by number: $s = \{v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3, v_4 \mapsto 4, v_5 \mapsto 5, v_6 \mapsto 6, v_7 \mapsto 7, v_8 \mapsto 8, v_9 \mapsto 0\}$.

$\mathcal{O}$ is a finite set of **operators**. Each $o \in \mathcal{O}$ has a precondition $\mathrm{pre}(o)$ and an effect $\mathrm{eff}(o)$ which are both partial states over $\mathcal{V}$, and a cost $\mathrm{c}(o) \in \mathbb{R}_0^+$.

The operator o is **applicable** in state s iff $\mathrm{pre}(o) \subseteq s$. We call the **resulting state** $o[\![s]\!]$. In

the resulting state $o[\![s]\!]$, $o[\![s]\!][v] = \text{eff}(o)[v]$ holds for all $v \in \text{eff}(o)$, while the other variables do not change, i.e., $o[\![s]\!][v] = s[v]$ for all $v \notin \text{eff}(o)$. In 8-Tiles, the operators encode the movement of one tile to the blank space. The precondition assures that the tile is next to the blank space, the effect swaps the values of the corresponding two variables, while all other tiles remain at the same position.
In order to reach the goal multiple operators need to be applied in a specific order. A sequence of operators $\pi = \langle o_1, \dots, o_n \rangle$ is called a path. It is applicable in s if o_1 is applicable in s and o_2 is applicable in $s_1 = o_1[\![s]\!]$ and so on. The resulting state is $\pi[\![s]\!] = s_n$. The cost of a plan is the sum over the cost of all contained operators, i.e., $\text{c}(\pi) = \sum_{i=1}^{n} \text{c}(o_i)$. A path π is an ***s*-plan**, if π is applicable in s and $\pi[\![s]\!]$ is a an extension of G and therefore a goal state. If it has minimal cost among all s-plans it is called **optimal**.
The set $\mathcal{R}$ is defined as the set of all **reachable** states. A state s is reachable, if a path π is applicable in I such that $\pi[\![I]\!] = s$. An operator o is reachable, if it is applicable in a reachable state. A state s is a **dead-end state** if it does not extend the goal state, and no s-plan exists.
We specify the tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, I, G \rangle$ where Π is a **planning task** in the SAS$^+$ formalism (Bäckström and Nebel, 1995).

2.2 Heuristics

One very common approach to solve planning tasks is heuristic search. A **heuristic** $h : \mathcal{R} \to \mathbb{R} \cup \{\infty\}$ estimates the cost of the optimal plan for a state $s \in \mathcal{R}$. The problem of 8-Tiles has uniform cost, as sliding a tile always costs the same, i.e. 1, and there are no other operators. Therefore, the cost of a s-plan of any state which is not a dead-end equals the amount of tiles, which need to be slid in the plan. The **optimal heuristic** $h^*(s)$ maps each state s to its actual optimal cost, or to ∞ if it is a dead-end state. We aim to approximate this heuristic.
Heuristics are used by search algorithms. They evaluate on where to continue the search based on the heuristic values of states. This thesis uses heuristics in the forward heuristic search where unreachable states are never expanded. Therefore they are defined over $\mathcal{R}$ instead of over all states and the following holds for reachable states only.
A heuristic is **admissible**, if it never overestimates the optimal heuristic, i.e., $h(s) \leq h^*(s)$. It is **goal-aware** iff $h(s) \leq 0$ for all reachable goal states, i.e., it recognizes a goal sate as such. Further, it is **consistent** iff $h(s) \leq h(o[\![s]\!]) + c(o)$ for all o applicable in s.
A heuristic which is goal-aware and consistent is also admissible. The search algorithm A* (Hart et al., 1968), which we will use for our evaluation (Ch. 4), always finds the optimal plan, if the used heuristic is admissible. We will therefore only consider admissible heuristics. One class of heuristics are potential heuristics which assign a numerical value to certain features of a planning task. One dimensional potential heuristics, which are subject of this thesis, assign a potential to each possible fact of the planning task (Pommerening et al., 2015).

Definition 1. *Let Π denote a planning task with facts $\mathcal{F}$. A **potential function** is a function $\mathtt{P} : \mathcal{F} \mapsto \mathbb{R}$. A **potential heuristic** for $\mathtt{P}$ maps each state $s \in \mathcal{R}$ to the sum of* *potentials of facts in s, i.e.,* $h^{\mathrm{P}}(s) = \sum_{f \in s} P(f)$.

The potentials themselves are obtained through optimization which will be further analyzed in Chapter 3.
One further approach is **ensemble heuristics**. Instead of only one heuristic, this approach uses multiple heuristics and chooses the highest value as heuristic value for for each state.

2.3 Mutexes and Disambiguations

A mutex is a set of facts which never appear together in any reachable state. If a partial state p in 8-Tiles holds $p[v_3] = 1$, then tile one may not be in any other spot of the grid, i.e., the fact $\langle v_3, 1\rangle$ is mutex with all other facts $\langle v, 1\rangle$ with $v \in \mathcal{V} \setminus \{v_3\}$.
The following definitions were introduced by Fišer et al. (2020).

Definition 2. *Let* Π *denote a planning task with facts* $\mathcal{F}$*. A set of facts* $\mathcal{M} \subseteq \mathcal{F}$ *is a* ***mutex*** *if* $M \nsubseteq s$ *for every reachable state* $s \in \mathcal{R}$

Further, we define the set $\mathcal{M}$. For a given set of mutexes M, $\mathcal{M}$ contains all mutexes which are in M and all further mutexes which can directly be inferred from M. For any (partial) state s it holds, that $s \in \mathcal{M}$ if s contains a subset of facts which are a mutex. For 8-Tiles, $\mathcal{M}$ would contain, among others, the partial state $p = \{\langle v_3, 1\rangle, \langle v_4, 1\rangle\}$, and all possible partial states extending p.
We can use $\mathcal{M}$ to derive disambiguations.

Definition 3. *Let* Π *denote a planning task with facts* $\mathcal{F}$ *and variables* $\mathcal{V}$*, let* $V \in \mathcal{V}$ *denote a variable, and let* p *denote a partial state. A set of facts* $F \subseteq \mathcal{F}_V$ *is called a* ***disambiguation*** *of* V *for* p *if for every reachable state* $s \in \mathcal{R}$ *such that* $p \subseteq s$ *it holds that* $F \cap s \neq \emptyset$ *(i.e.,* $\langle V, s[V]\rangle \in F$*).*

The disambiguation of a variable V for a partial state p is the set of facts $F \subseteq \mathcal{F}_V$ which occur in all reachable extended states of p. This means, that each fact of V which is not in F is a mutex with p. If F contains exactly one fact then p can be safely extended with that fact, as it is the only non-dead-end extension of the state. If F is the empty set every extended state of p is a dead-end. This knowledge can be used to prune operators o for which $p \subseteq \mathrm{pre}(o)$ and unreachable states $s \subseteq p$. If the goal state G is one of this states, the problem is unsolvable.
If a partial state s of the 8-Tiles problem holds $p[v_3] = 1$ and $p[v_2] = 1$, then it is a dead-end, as these facts are a mutex. If $p = \{v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3, v_4 \mapsto 4, v_5 \mapsto 5, v_6 \mapsto 6, v_7 \mapsto 7, v_8 \mapsto 8\}$ then p is not a dead-end and v_9 can safely be assigned with 0, as it is the only fact in $\mathcal{F}_{v_9}$ which does not form a mutex with any of the already assigned facts.
The set $\mathcal{M}_p = \{f | f \in \mathcal{F}, p \cup \{f\} \in \mathcal{M}\}$ is the set of facts which are mutex with p. All facts of a variable $f \subseteq \mathcal{F}_V$ not contained in $\mathcal{M}_p$ build the disambiguation F of V for p. In Chapter 3 we will use this to improve potential heuristics by narrowing down possible extensions of partial states.

3 Strengthening Potential Heuristics

Fišer et al. (2020) propose a method to improve potential heuristics with mutexes and disambiguations. This chapter contains the changes which are required to do so, regarding the transformation of a planning task into TNF and the adaption of the optimization functions. It shows how the equations which were later implemented (Section Implementation) are derived.

3.1 Potential Heuristics

When Pommerening et al. (2015) first introduced potential heuristics, they described to inequalities to assure the admissibility of the heuristic. The first one assures goal-awareness, the second one consistency.

Theorem 4. *Let* $\Pi = \langle \mathcal{V}, \mathcal{O}, I, G \rangle$ *denote a planning task,* $\mathtt{P}$ *a potential function, and for every operator* $o \in \mathcal{O}$*, let* $\mathrm{pre}^*(o) = \{\langle V, \mathrm{pre}(o)[V] \rangle | V \in \mathrm{vars}(\mathrm{pre}(o)) \cap \mathrm{vars}(\mathrm{eff}(o))\}$ *and* $\mathrm{vars}^*(o) = \mathrm{vars}(\mathrm{eff}(o)) \setminus \mathrm{vars}(\mathrm{pre}(o))$*. If*

$$\sum_{f \in G} \mathtt{P}(f) + \sum_{V \in \mathcal{V} \setminus \mathrm{vars}(G)} \max_{f \in \mathcal{F}_V} \mathtt{P}(f) \leq 0 \tag{3.1}$$

and for every operator $o \in \mathcal{O}$ *it holds that*

$$\sum_{f \in \mathrm{pre}^*(o)} \mathtt{P}(f) + \sum_{V \in \mathrm{vars}^*(o)} \max_{f \in \mathcal{F}_V} \mathtt{P}(f) - \sum_{f \in \mathrm{eff}(o)} \mathtt{P}(f) \leq c(o) \tag{3.2}$$

then the potential heuristic for $\mathtt{P}$ *is admissible.*

Eq. (3.1) of Theorem 4 assures goal-awareness of the potential heuristic. As all variables are assigned in a goal state, the potential of one fact per variable has to be summed up. For the variables $v \in \mathrm{vars}(G)$ we can simply use the potentials of their respective facts. Meanwhile we assume the worst case for the other variables, by using the maximal potential over their facts, as we do not know what fact they are assigned.
Eq. (3.2) assures consistency. Recall the general heuristics consistency equation $h(s) \leq h(o[\![s]\!]) + \mathrm{c}(o)$. It can be rewritten as $h(s) - h(o[\![s]\!]) \leq \mathrm{c}(o)$. As the facts which do not occur in the effect are the same in both s and $o[\![s]\!]$ we can leave them aside. For s we know

what facts of the variables of the preconditions are assigned and sum the potentials of the facts which are in the effect as well. For the variables which are in the effect but not in the precondition we proceed similarly to (3.1), as we do not know their values. The potentials of the facts in the effect can be used without modification for $o[\![s]\!]$.
The advantage of these equations is that they are not state-dependent, even though they do not tell us explicitly what the potentials should be. However, they can be used as the constraints for a linear program, the solution of which is a potential function that forms an admissible potential heuristic. More about this in Section 3.2.

3.1.1 Generalization with Mutexes

Mutexes can be used to reduce the domain of variables, which are not yet assigned in a partial state p. This property is very helpful, as it minimizes the amount of facts which are candidates for the not assigned variables in Equations (3.1) and (3.2) of Theorem 4. We perform this reduction with the following algorithm (Fišer et al., 2020).

Algorithm 1 Multi-fact fix-point disambiguation.

```
Input: A planning task $\Pi$ with variables $\mathcal{V}$ and facts $\mathcal{F}$, a partial state $p$, and a mutex-set $\mathcal{M}$.
Output: A set of disambiguations $\mathcal{D}_p$ of all variables $\mathcal{V}$ for $p$.
 1: $D_v \leftarrow \mathcal{F}_V$ for every $V \in \mathcal{V}$
 2: $A \leftarrow \mathcal{M}_p$
 3: change $\leftarrow$ True
 4: while change do
 5:     change $\leftarrow$ False
 6:     for all $V \in \mathcal{V}$ do
 7:         if $D_V \setminus A \neq D_V$ then
 8:             $D_V \leftarrow D_V \setminus A$
 9:             $A \leftarrow A \cup \bigcap_{f \in D_V} \mathcal{M}_{p \cup \{f\}}$
10:             change $\leftarrow$ True
11:         end if
12:     end for
13: end while
14: $\mathcal{D}_p \leftarrow \{D_V | V \in \mathcal{V}\}$
```

At the beginning of the algorithm, the set D_V contains all possible values for the variable $V \in \mathcal{V}$, while A contains all facts which are a mutex with p. In each iteration of the while-loop, all $f = \langle v, V \rangle$ which are in A and in D_V are removed from the corresponding D_V. On line 9, A is extended with all facts that form a mutex with all facts remaining in D_V, i.e., which are a mutex with $p \cup \{f\}$ for all $f \in D_V$.
In conclusion, after applying multi-fact fix-point disambiguation p can be extended with any fact in $\mathcal{D}_p$ without reaching a dead-end state. If any $D_V \in \mathcal{D}_p$ is empty, then p is already a dead-end itself.
This algorithm is used for several applications during the remainder of this chapter. In this section, it can be used to generalize Theorem 4 by removing the potentials of facts, which are a mutex with the corresponding goal or effect. Since we define heuristics over $\mathcal{R}$, the resulting potential heuristic is still admissible.

Theorem 5. *Let* $\Pi = \langle \mathcal{V}, \mathcal{O}, I, G\rangle$ *denote a planning task with facts* $\mathcal{F}$*, and let* $\mathtt{P}$ *denote a potential function, and*

(i) for every variable $V \in \mathcal{V}$*, let* $G_V \subseteq \mathcal{F}_V$ *denote a disambiguation of* V *for* G *s.t.* $|G_V| \geq 1$*, and*

(ii) for every operator $o \in \mathcal{O}$ *and every variable* $V \in \mathrm{vars}(\mathrm{eff}(o))$*, let* $E_V^o \subseteq \mathcal{F}_V$ *denote a disambiguation of* V *for* $\mathrm{pre}(o)$ *s.t.* $|E_V^o| \geq 1$.

If

$$\sum_{V \in \mathcal{V}} \max_{f \in G_V} \mathtt{P}(f) \leq 0 \tag{3.3}$$

and for every operator $o \in \mathcal{O}$ *it holds that*

$$\sum_{V \in \mathrm{vars}(\mathrm{eff}(o))} \max_{f \in E_V^o} \mathtt{P}(f) - \sum_{f \in \mathrm{eff}(o)} \mathtt{P}(f) \leq \mathrm{c}(o) \tag{3.4}$$

then the potential heuristic $\mathtt{P}$ *is admissible.*

Fišer et al. (2020) prove the theorem by showing that Equations (3.3) and (3.4) are generalizations of Equations (3.1) and (3.2), respectively.
The disambiguation G_V equals $D_V \in \mathcal{D}_G$ with $V \in \mathcal{V}$, which is generated by applying Algorithm 1 on the goal state. If it is empty for any of the variables, then the problem is unsolvable, as the goal contains a mutex and is therefore not a reachable state. The set E_V^o is equal to $D_V \in \mathcal{D}_{\mathrm{pre}(o)}$. If E_V^o is empty for any $V \in \mathrm{vars}(\mathrm{eff}(o))$, o is not applicable in any state.
Since we later use this equations as constraints for a Linear Program, we must bring them in a form which does not contain an optimization function. We do this by transforming the planning task into transition normal form.

3.2 Transition Normal Form

A planning task can always be compiled into Transition Normal Form or TNF (Pommerening and Helmert, 2015). In TNF, a planning task has a fully defined goal ($\mathrm{vars}(G) = \mathcal{V}$) and all variables of the effect are also in the precondition for each operator $o \in \mathcal{O}$ ($\mathrm{vars}(\mathrm{pre}(o)) = \mathrm{vars}(\mathrm{eff}(o))$). These properties are essential to form the Linear Program, which we will discuss in the next section. Any planing task $\Pi = \langle \mathcal{V}, \mathcal{O}, I, G\rangle$ can be transformed into TNF with the following rules cited from Fišer et al. (2020):

- Add a fresh value U to the domain of every variable.
- For every variable $V \in \mathcal{V}$ and every fact $f \in \mathcal{F}_V$, $f \neq \langle V, U\rangle$, add a new *forgetting* operator o_f with $\mathrm{pre}(o_f) = \{f\}$ and $\mathrm{eff}(o_f) = \{\langle V, U\rangle\}$ and the cost $\mathrm{c}(o_f) = 0$.
- For every operator $o \in \mathcal{O}$ and every variable $V \in \mathcal{V}$:
  - If $V \in \mathrm{vars}(\mathrm{pre}(o))$ and $V \notin \mathrm{vars}(\mathrm{eff}(o))$, then add $\langle V, \mathrm{pre}(o)[V]\rangle$ to $\mathrm{eff}(o)$.
  - If $V \in \mathrm{vars}(\mathrm{eff}(o))$ and $V \notin \mathrm{vars}(\mathrm{pre}(o))$, then add $\langle V, U\rangle$ to $\mathrm{pre}(o)$.

- For every $V \in \mathcal{V} \setminus \text{vars}(G)$ add $\langle V, U \rangle$ to G.

Each Variable $V \in \mathcal{V}$ gets a new value U in its domain, which can be seen as a sort of placeholder. The fact $\langle V, U \rangle$ can be assigned with cost 0, as the forgetting operator o_f which assigns it has no cost, regardless of the current state. The next point is to assure that for each operator the variables which are in the precondition are also in the effect.
If V is present in the preconditions of an operator $o \in \mathcal{O}$ but not in the effect, then we can simply add the fact $\langle V, pre(o)[V] \rangle$ to the effect. This is a formal change, but does not change the effect of the operator at all, as it would not have changed this fact anyway.
The case of an operator $o \in \mathcal{O}$ where V is in the effect but not in the precondition, is a little more complicated. Here, the precondition is changed such that it contains also the fact $\langle V, U \rangle$. If o was applicable in s before, then, after transforming the plan into TNF, the corresponding o_f needs to be applied beforehand in order to forget the value of V. This change of the variable is insignificant, as the value then gets changed by applying the operator anyways.
Last, all variables which were not included in the partial state G need to be added into it. If they are assigned the fresh value U, then the goal state can be reached from every state which expanded it before. Without creating more cost, the values of all unimportant variables are changed to the fresh value. The compilation into TNF can produce a planing task twice the size of the original task in worst case (Seipp et al., 2015).

3.2.1 Generalization with Mutexes

Similar to Section 3.1.1, these rules can be generalized with disambiguations. Therefore, to replace U we introduce the fresh values U_{G_V} and $U_{E_V^o}$. Instead of adding forgetting operators from every fact in every $V \in \mathcal{V}$, we use the disambiguation sets G_V and E_V^o.

- Add fresh value U_{G_V} to the domain of every $V \in \mathcal{V}$.
- For every variable $V \in \mathcal{V}$ and every fact $f \in G_V$, $f \neq \langle V, U_{G_V} \rangle$, add new *forgetting* operators o_{f_G} with $\text{pre}(o_{f_G}) = \{f\}$ and $\text{eff}(o_{f_G}) = \{\langle V, U_{G_V} \rangle\}$ and the cost $\text{c}(o_{f_G}) = 0$.
- For every $V \in \mathcal{V} \setminus \text{vars}(G)$ add $\langle V, U_{G_V} \rangle$ to G.
- For every operator $o \in \mathcal{O}$ add fresh value $U_{E_V^o}$ to the domain of every $V \in \mathcal{V}$:
  - If $V \in \text{vars}(\text{pre}(o))$ and $V \notin \text{vars}(\text{eff}(o))$, then add $\langle V, \text{pre}(o)[V] \rangle$ to $\text{eff}(o)$.
  - If $V \in \text{vars}(\text{eff}(o))$ and $V \notin \text{vars}(\text{pre}(o))$, then add $\langle V, U_{E_V^o} \rangle$ to $\text{pre}(o)$.
- For every variable $V \in \mathcal{V}$, every operator$o \in \mathcal{O}$ and every fact $f \in E_V^o$, $f \neq \langle V, U_{E_V^o} \rangle$, add new forgetting operators $o'_{f,o}$ with $\text{pre}(o'_{f,o}) = \{f\}$ and $\text{eff}(o'_{f,o}) = \{\langle V, U_{E_V^o} \rangle\}$ and the cost $\text{c}(o'_{f,o}) = 0$.

For the goal state, forgetting operators are only created for the facts in F_V which are not a mutex with G for every $V \notin \text{vars}(G)$. Similarly, facts in F_V which are a mutex with any $f \in \text{pre}(o)$ are not taken into account for all $o \in \mathcal{O}$ and every $V \in \text{vars}(\text{eff}(o))$. This creates at most $|\mathcal{O}| * |\mathcal{V}|$ $U_{E_V^o}$ and $|\mathcal{V}|$ U_{G_V} and the amount of forgetting operators is in the worst case the same, multiplied with the sum of the cardinalities of the domains of all $V \in \mathcal{V}$. In practice, Fišer et al. (2020) show that the transformation with disambiguations is never bigger than without disambiguations.

3.3 Linear Program

The formulas and rules which were defined in the previous two sections can now be used to form a Linear Program (LP).
An LP consists of LP-variables which are constrained by multiple inequalities (constraints) and which are part of an optimization function. An LP-solver then assigns each LP-variable a value, such that all constraints are satisfied and the optimization function is optimized. We will look at different optimization functions in the next section (3.4).
In order to find a potential heuristic, the LP-variables are the potentials of the facts, $P(f)$. Further, we introduce the the LP-variables $X_f = \mathtt{P}(f)$ for every $f \in \mathcal{F}$ and M_{G_V} and $M_{E_V^o}$ corresponding to U_{G_V} and $U_{E_V^o}$, respectively, with the constraint that $X_f \leq M_{G_V}$ for every $f \in G_V$ and $X_f \leq M_{E_V^o}$ for every $f \in E_V^o$. Using these variables, we can transform the Equations (3.3) and (3.4) into the constraints

$$\sum_{f \in G} X_f + \sum_{V \in \mathcal{V} \setminus \mathrm{vars}(G)} M_{G_V} \leq 0 \tag{3.5}$$

and

$$\sum_{f \in \mathrm{pre}^*(o)} X_f + \sum_{V \in \mathrm{vars}^*(o)} M_{E_V^o} - \sum_{f \in \mathrm{eff}(o)} X_f \leq c(o) \tag{3.6}$$

which assure admissibility. M_{G_V} and $M_{E_V^o}$ correspond to U_{G_V} and $U_{E_V^o}$, respectively, since these are the facts which are assigned if a variable is not defined in the goal state or in a precondition of an operator. Th LP can then be used to build the potential heuristic (Pommerening et al., 2015):

Definition 6. *Let f be a solution to the following LP:*
Maximize opt *subject to* (3.5) *and* (3.6), *where the objective function* opt *can be chosen arbitrarily.*
Then the function $\mathrm{pot}_{\mathrm{opt}}(\langle V, v\rangle) = f(\mathtt{P}_{\langle V,v\rangle})$ *is the* potential function optimized for opt *and* $h^{\mathtt{P}}$ *is the* potential heuristic optimized for opt.

Since 3.5 and 3.6 assure admissibility, any solution of the LP builds an admissible $h^{\mathtt{P}}$. The solution of the LP might differ vastly depending on the used optimization function.

3.4 Optimization Functions

An optimization function opt is a linear combination of the LP-variables. In our case, the goal is to have best possible heuristic value for as many states as possible. Using different optimization functions optimizes different aspects of a heuristic. The perfect heuristic would be achieved if we optimized the potentials for each state separately, but this is computationally too expensive.

In the first proposal for potential heuristics from Pommerening et al. (2015), the optimization for the initial state was used,

$$\mathrm{opt}_I = \sum_{f \in I} \mathtt{P}(f). \tag{3.7}$$

It optimizes the heuristic value for the initial state. The drawback is that facts which do not appear in the initial state are not taken into account.

Alternatively, we could optimize the potentials for all reachable states, as we are only interested in these:

$$\mathrm{opt}_{\mathcal{R}} = \frac{1}{|\mathcal{R}|} \sum_{s \in \mathcal{R}} \sum_{f \in s} \mathtt{P}(f). \tag{3.8}$$

It calculates the weighted sum of all facts, i.e., it multiplies the potential of a fact with the amount of reachable states containing this fact and normalizes it with the total amount of reachable states. The potentials generated with this optimization function would result in the heuristic with the maximal average heuristic value over all reachable states. Unfortunately, calculating this is, again, computationally expensive or even infeasible, if the planning task and therefore the size of $\mathcal{R}$ are big, as the set of reachable states is not known. To avoid this, we could sample some states $S \subseteq \mathcal{R}$, and calculate Equation (3.8) over these states, instead of $\mathcal{R}$:

$$\mathrm{opt}_{\hat{\mathcal{S}}} = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \sum_{f \in s} \mathtt{P}(f). \tag{3.9}$$

This equation can easily be used for ensemble heuristics. We could generate multiple sets of sample states, optimize for each and create a multiple potential heuristic for each of these sets. The optimization function could also assume uniform distribution and give all facts the same weight. Instead of going over all facts of all states, we would sum over the domains of all variables, optimizing the average value of all syntactic states:

$$\mathrm{opt}_{\mathcal{S}} = \sum_{\langle V,v \rangle \in \mathcal{F}} \frac{1}{|\mathrm{dom}(V)|} \mathtt{P}(\langle V, v \rangle). \tag{3.10}$$

We call this the all-states-potential optimization function. Both, (3.10) and (3.9), can be strengthened with mutexes and disambiguations.

3.4.1 Strengthening All State Potentials

To estimate the amount of reachable states containing $f = \langle V, v \rangle$, we will calculate the upper bound of these states, and try to lower it for each $f \in \mathcal{F}$. The product of the domains of all variables except V, since V is already assigned, is the total amount of states, reachable and non-reachable, which contain f. With Algorithm 1 from Section 3.1.1 we can remove all facts from all domains which are mutex with f. These facts could never be assigned in any reachable state containing f. Therefore, the product over all domains in the resulting disambiguation set is again an upper bound of appearances of f in $\mathcal{R}$.

This holds not only for a single fact, but can be applied to estimate the appearances of any partial state p. As the product of all domains in the disambiguation set is taken the value is zero, if p is a mutex itself.

We define $\mathcal{M}_p$ as the set of all facts which form a mutex with partial state p and $\mathcal{P}_k^{\{f\}}$ as all partial states of size k extending $\{f\}$. Then an upper bound for the amount of reachable states of size k containing f is

$$\mathcal{C}_f^k(\mathcal{M}) = \sum_{p \in \mathcal{P}_k^{\{f\}}} \prod_{V \in \mathcal{V}} |F_V \setminus \mathcal{M}_p|. \tag{3.11}$$

The Equation has the constraint to only extend states to size k, as it would get computationally too expensive to compute all reachable states for which f holds.
We will now use Equation (3.11) to calculate the weight of each $f \in \mathcal{F}$ as its relative appearance in all states extending any $p \in \mathcal{P}_k^{\{f\}}$ and form the following optimization function:

$$\text{opt}_{\mathcal{M}}^k = \sum_{f=\langle V,v\rangle \in \mathcal{F}} \frac{\mathcal{C}_f^k(\mathcal{M})}{\sum_{f' \in \mathcal{F}_V} \mathcal{C}_{f'}^k(\mathcal{M})} \mathtt{P}(f). \tag{3.12}$$

The result is a modification of opt_S (Eq. (3.10)) that does not assume uniformly distributed facts. Each fact is weighted according to its estimated appearance in all reachable states of size k in relation to the other facts of the corresponding variable. The sum of the weights for all facts of one variable is 1. We can use this not only as a single heuristic, but also in ensemble heuristics.

3.4.2 Strengthening Conditioned Ensemble Potentials

If we divide $\mathcal{R}$ into multiple subsets $S_i \subset \mathcal{R}$, with $i = 1, \dots, n$ and $S_1 \cup \dots \cup S_n = \mathcal{R}$, the solution of the LP with optimization function $\text{opt}_{\hat{S}_i}$ gives better heuristic values for the states in S_i than $\text{opt}_{\mathcal{R}}$. If we calculate the potentials for the optimization of all S_i, the resulting heuristics can be used as ensemble heuristics. This gives a result which lies somewhere between the all-states-potential heuristic and calculating the potentials for each individual state.
We will choose S_i as the states extending one particular partial state t, and use the potentials generated with an adjusted $\text{opt}_{\mathcal{M}}^k$ as one of multiple ensemble heuristics. For this we adapt Equation (3.11), such that it counts how many reachable states of size $|t| + k$ extend t:

$$\mathcal{K}_f^k(\mathcal{M}, t) = \sum_{p \in \mathcal{P}_{|t|+k}^{t \cup \{f\}}} \prod_{V \in \mathcal{V}} |\mathcal{F} \setminus \mathcal{M}_p|. \tag{3.13}$$

The corresponding optimization function uses the weights calculated with $\mathcal{K}_f^k(\mathcal{M}, t)$,

$$\text{opt}_{\mathcal{M}}^{t,k} = \sum_{f=\langle V,v\rangle \in \mathcal{F}} \frac{\mathcal{K}_f^k(\mathcal{M}, t)}{\sum_{f' \in \mathcal{F}_V} \mathcal{K}_f^k(\mathcal{M}, t)} \mathtt{P}(f). \tag{3.14}$$

For this thesis, the partial states t were sampled uniformly at random, as did Fišer et al. (2020). Future research could be to investigate other ways to choose them.

3.4.3 Adding Constraint on Initial State

Both $\text{opt}_{\mathcal{M}}^k$ and $\text{opt}_{\mathcal{M}}^{t,k}$ optimize the potentials in regard of the entire reachable state space. However, the importance of states may vary strongly, depending on where we start and what path is taken from there towards the goal. For example, a planning task with multiple goal states may have smaller heuristic values, even though the constraints enforce goal-awareness. This is why we now look at a third approach which gives the initial state more weight and add the constraint

$$\sum_{f \in I} \mathtt{P}(f) = h^{\mathtt{P}}_{I}(I) \tag{3.15}$$

to the Linear Program, where $h^{\mathtt{P}}_{I}$ denotes the potential heuristic optimized for the initial state (Eq. (3.7)). By calculating the potentials for this heuristic first and then taking it into account for the actual potentials, we force the solver to find potentials which guarantee high heuristic values for the initial state. This approach can be combined with all previously discussed optimization functions.

3.4.4 Adding Constraints on Random States

Taking the idea of the additional constraint a step further, we decided to add constraints on random states. For multiple states s we add the constraint

$$\sum_{f \in s} \mathtt{P}(f) = h^{\mathtt{P}}(s). \tag{3.16}$$

,

to the Linear Program, where $h^{\mathtt{P}}_{s}$ denotes the potential heuristic optimized for state s, i.e., $\mathrm{opt}_{\hat{\mathcal{S}}}$ ((3.9)) with $S = s$.

The states are generated with random walks. The length of the walks is binomially distributed around the ratio of the heuristic value of the initial state, $h^{\mathtt{P}}(I)$, divided by the average cost over all operators. Similarly to the additional constraint on the initial state, this gives more weight to states reachable with this amount of steps from the initial state. This rises the average heuristic value for the states in this area, which are the ones we are interested in.

Since the states are generated randomly, the positive effect of these additional constraints can not be guaranteed. To avoid this, a considerable amount of generated states would be necessary, but this is computationally expensive.

3.5 Implementation

Our implementation of mutex based potential heuristics is embedded in the planning system Fast Downward (Helmert, 2006) and written in C++.

We first implemented Algorithm 1, using the Fast Downward hm-heuristic with $m = 2$ to build a mutex table. The computation of the hm-heuristic in Fast Downward is very slow. Therefore, we implemented a new generator for the mutex table, which is very similar to the hm-heuristic, but optimized for finding mutexes and therefore significantly faster.

Fast Downward has a potential optimizer, which initializes and constructs an LP-solver. We implemented a new LP-constructor which builds the constraints according to Equations (3.5) and (3.6). For both the non-mutex and the mutex constructor, we added the option to use the additional constraints on the initial state and random states. They can be used individually

or combined. The additional constraints on random states do not take mutexes into account for generating the states. The Linear Program is optimized for each state individually, and the new constraint is added to the LP before optimizing for the next state.
Next, we implemented the optimization function $\mathrm{opt}^k_{\mathcal{M}}$. All partial states of cardinality one ($p_f = \{f\}$ for all $f \in \mathcal{F}$) are generated, and then recursively all partial states of size k extending p_f are created, using the disambiguation set $\mathcal{D}_{p_f}$. Even for small tasks the amount of extended states can grow very fast. For memory efficiency partial states are implemented as maps, containing the assigned facts only. Using these states, we calculate the weight of each fact with $\mathcal{C}^k_f(\mathcal{M})$. The weights are stored as vectors and passed to the potential optimizer, which uses them to generate the optimization function.
The optimization function $\mathrm{opt}^{t,k}_{\mathcal{M}}$ was implemented implicitly. As all former mentioned methods can handle partial states of cardinalities ≥ 1, only one new method was needed. It generates n mutex based potential heuristics, each of which uses a random state of size t. With each of these we perform the same procedure as above with p_f.
Running Fast Downward, the heuristics obtained with the strengthened optimization functions, called mutex based potential heuristics, can be used with the command `--search "astar(mutex_based_potential())"`, where `astar` can be replaced by any other search algorithm of Fast Downward. The size to which p_f should be extended can be set with `k`, the default value is `k=2`.
The command `--search "astar(mutex_based_ensemble_potential())"` uses the mutex based ensemble heuristics. The variables can be set manually and their respective default values are `t=1`, `k=2` and `n=50`. `k` must be greater than `t`, but smaller than the size of a fully extended state. `n` can be chosen arbitrarily, however the evaluation in Chapter 4 shows that the results are best with `n=10`.
Further, all potential heuristics can be solved using LP-constraints built with mutexes through the option `mutex=1`. The default value is `mutex=0` for all potential heuristics, except for the mutex based ones. The constraint for the initial state is added through `init-const=1`, the default value for this is 0 for all potential heuristics. With `rand-con-num` the amount of additional constraints on random states can be set, with the default value 0 none are added.
We will evaluate our implementation in the next chapter.

4 Experimental Evaluation

We tested our implementation of the strengthened potential heuristics on 1827 STRIPS problems with the official domains from the International Planning Competition (IPCs). Further, we used Downward Lab (Seipp et al., 2017) to set up the tests and ran them on the sciCORE high-performance computing infrastructure. We set the time limit for each problem to 30 minutes and the memory limit to 8 GB. The different heuristics were all used in an A* search (Hart et al., 1968).
We test whether the LP constraints built with disambiguations improve performance and compare the different optimization functions against each other. Further, we compare different ensemble heuristics and their respective amount of heuristics used, as well as the additional constraints on the initial state and on random states. Last, we compare our results to the results from the evaluation of Fišer et al. (2020).
We refer to the compared variants of heuristics as follows:

`lmc`:	The LM-Cut heuristic, introduced by Helmert and Domshlak (2015).
`init`:	The initial state potential heuristic (Eq. (3.7)).
`all`:	The all states potential heuristic (Eq. (3.9)).
`max`:	The maximization over `init` and `all`.
`div`:	The diversification heuristic (Seipp et al., 2015) with 1000 samples.
$\mathtt{S}_{\mathtt{i}}^{\mathtt{n}}$:	The sample based potential heuristic (Eq. (3.10)), n being the number of heuristics and i the number of samples per heuristic.
$\mathtt{M}_{\mathtt{k}}$:	The strengthened all states potential heuristic with $\mathrm{opt}^k_{\mathcal{M}}$ (Eq. (3.12)).
$\mathtt{K}_{\mathtt{k}}^{\mathtt{n}}$:	The strengthened ensemble potential heuristic with $\mathrm{opt}^{t,k}_{\mathcal{M}}$ (Eq. (3.14)) with $\lvert t\rvert = 1$ and n heuristics.
$\mathtt{L}_{\mathtt{k}}^{\mathtt{n}}$:	The same as $\mathtt{K}_{\mathtt{k}}^{\mathtt{n}}$, but with $\lvert t\rvert = 2$.
$\mathtt{J}_{\mathtt{k}}^{\mathtt{n}}$:	The same as $\mathtt{K}_{\mathtt{k}}^{\mathtt{n}}$, but with $\lvert t\rvert = 3$.

In addition, **`N`** refers to the non-strengthened LP, while **`D`** uses the LP strengthened with disambiguations. When the additional constraint on the initial state is used, **`I`** is appended to the name of the heuristic and **`R`** if the constraints on random states are used.
The attributes we use to compare different configurations are:

Coverage:	The amount of problems solved with the respective configuration.
Expansions:	The number of state expansions needed to solve the problem.
Total Time:	The total time needed to solve the problems without pre processing, in minutes.
Search Time:	The time needed for the search only, in minutes.
Out of Memory:	The amount of problems which failed due to a lack of memory.
Out of Time:	The amount of problems which failed due to a lack of time.

The attributes expansions, total time and search time are the geometric mean over all problems which were solved by all configurations. The search time is exclusively the time needed to perform the search. In all tables, the best value per attribute is written in bold. For coverage, this is the highest value and for the other attributes it is the lowest.

4.1 Results

The following table shows results for the heuristics which were already provided in the Fast Downward planning system.

	lmc	**all-N**	**init-N**	**max-N**	**div-N**	**S_1^{100}-N**	**S_{1000}^1-N**
Coverage	958	929	891	948	**963**	945	961
Expansions	**1287**	10244	22415	8270	6904	7181	9238
Total Time	0.57	**0.29**	0.54	0.33	0.74	0.94	0.33
Search Time	0.52	0.23	0.43	0.24	0.74	0.94	**0.22**
Out of Memory	**0**	870	911	854	623	170	844
Out of Time	852	11	8	8	224	695	**5**

Table 4.1: Test results for the already provided heuristics.

We see that `lmc`, `div-N` and S_{1000}^1-N have the highest coverage over all problems. Further, `lmc` has the lowest number of expansions and therefore the lowest out of memory errors. S_{1000}^1-N, on the other hand, is the fastest of the three configurations and has only few out of time errors.
As the same search algorithm is used for all configurations, the number of expansions can be used as an indicator for the quality of the heuristic value. Few expansions combined with a high coverage, e.g. with `lmc`, can be interpreted as an efficient search, with only few state expansions, due to a good heuristic value.
In the following subsections, we compare these results to the results we obtained with the features we implemented.

4.1.1 Mutex Based Linear Program

First, we compare the results from above with the ones obtained with the LP built with mutexes and disambiguations. The following table shows multiple configurations and their respective results.
The coverage is lower for these configurations. However, the number of expansions is better for all configurations with respect to their non-mutex version, except for `all`. The scatter

	all-D	init-D	max-D	div-D	S_1^{100}-D	S_{1000}^1-D
Coverage	879	881	932	837	853	**952**
Expansions	12101	18964	7863	**5269**	5503	7697
Total Time	0.68	0.90	0.84	4.18	3.29	**0.64**
Search Time	0.27	0.39	0.24	**0.19**	0.31	**0.19**
Out of Memory	824	824	770	560	**273**	726
Out of Time	75	**73**	76	381	652	100

Table 4.2: Test results for mutex based Linear Programs.

plot 4.1 shows how the the amount of expansions differ between the two different LPs for all solved problems for all and S_{1000}^1. Dots above the diagonal represent problems for which D has more expansions, dots below the diagonal represent problems for which N has more expansions. For all, more dots are above the diagonal, as it has a higher number of expansions. For S_{1000}^1, more dots are below the diagonal.

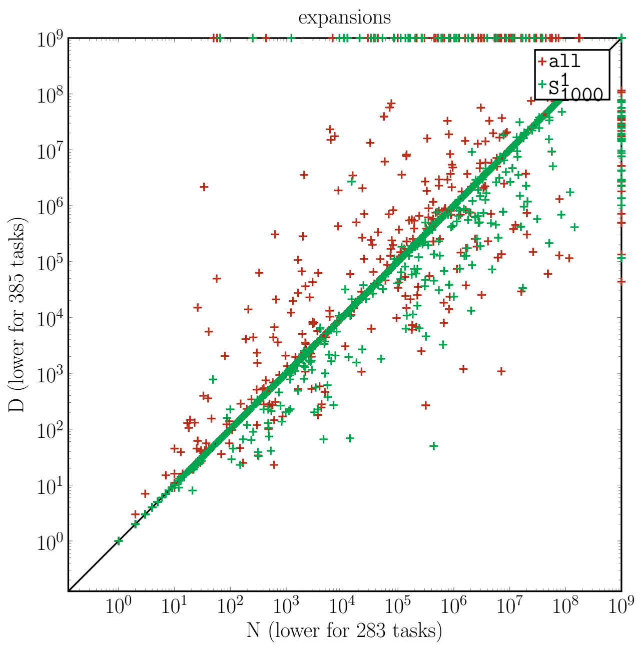

Figure 4.1: Number of expansions of all and S_{1000}^1 for both LPs per problem.

Due to the smaller amount of expanded states, less out of memory errors occur. The number of out of time errors, on the other hand, are higher. This is no surprise, as the total time is higher for all configurations as well. The search time is roughly the same, which indicates that the building of the mutex table and the mutex based LP is the most time consuming difference. Hence, it is the reason for the high amount of out of time errors and the relatively low coverage. This holds especially for div-D and S_1^{100}-D, where the total time is much higher than before and the difference between total and search time is immense. The search itself is a lot faster, which indicates good heuristic values. Both heuristics build the LP multiple times and, in our implementation, each time the mutex table needs to be computed from a fresh start.

We already greatly enhanced the performance of building the mutex table. However, finding a more efficient way for building the mutex table would lead to a higher coverage. To build the mutex table, we optimized the (very slow) Fast Downward hm-heuristic for $m = 2$ and ignored heuristic values, as we are only interested in the binary reachability. Before the optimization, the mutex table was built for 1450 problems, in 117 seconds on average. Now, 1776 mutex tables are built in 34 seconds on average. Thus, over 300 additional mutex tables can be built in less than 30 minutes, some of them in less than 20 seconds.
Comparing the configurations amongst each other, we see that the coverages of $\texttt{S}^{1}_{\texttt{1000}}\texttt{-D}$ and `max-D` are still good, as they only decrease by around 10 problems. So does the coverage of `init`. These are also the configurations, for which the expansions and search time sunk, as well as the out of memory errors. The other configurations are significantly worse with disambiguations. This does not hold for mutex based optimization functions.

4.1.2 Mutex Based Optimization Functions

The next table shows the results for our mutex based optimization functions. For the ensemble heuristics (`K`, `L` and `J`), we chose $n = 10$, since using 10 heuristics gave the best results on average (Table 4.4).

	$\texttt{M}_1\texttt{-D}$	$\texttt{M}_2\texttt{-D}$	$\texttt{K}^{10}_1\texttt{-D}$	$\texttt{K}^{10}_2\texttt{-D}$	$\texttt{L}^{10}_1\texttt{-D}$	$\texttt{L}^{10}_2\texttt{-D}$	$\texttt{J}^{10}_1\texttt{-D}$	$\texttt{J}^{10}_2\texttt{-D}$
Coverage	900	859	911	831	921	840	**922**	845
Expansions	8297	8240	6790	6847	6126	6273	6197	**6039**
Total Time	**0.59**	1.23	0.89	4.23	0.99	3.93	1.02	3.20
Search Time	**0.20**	**0.20**	0.86	4.08	0.86	3.78	0.89	3.04
Out of Memory	802	726	714	608	691	589	677	**586**
Out of Time	**77**	203	155	351	169	364	193	364

Table 4.3: Test results for mutex based potential heuristics.

For the four different categories, `M`, `K`, `L` and `J`, we see that the configurations with smaller k are always better. This is due to the higher time and memory consumption needed for bigger extensions, which can be concluded from the increasing sum of out of time and out of memory failures. We can also see that extending partial states by two facts is more time consumptive, as more out of time errors occur for these configurations. Our pretests showed, that for $k = 3$ the coverage decreases about 30% compared to $k = 1$. For even higher k, it would drop vastly, as the memory and time limits are not high enough to extend (multiple) partial states to this size.
Currently, the implementation for mutex based potential heuristics is able to work with any $k \in \mathbb{N}$ smaller than the size of a state. An optimization for $k = 1$ and $k = 2$ could, similarly to the optimization of building the mutex table, enhance the coverage as well.
However, the fact that the amount of expansions decreases for greater k and t shows that the approach for mutex based potential heuristics is good. For $|t| = 3$, the coverage and the number of expansions are better than for smaller t.
The comparison of the results for mutex based potential functions with Table 4.1 shows that the coverage is not higher than for any of the former configurations. However, less out

of memory errors occurred, while the out of time errors increased. This is, similar to the results for the mutex based LP (Sec. 4.1.1), due to the building of the mutex table.
We also tested the configurations $\texttt{M}_1\texttt{-N}$ and $\texttt{M}_2\texttt{-N}$, which have a slightly higher coverage. With the non-mutex based LP, the number of expansions and therefore the amount of out of memory errors are higher. The higher coverage is due to the saved effort by not considering mutexes for building LP.

4.1.3 Ensemble Heuristics

Table 4.4 shows the coverage for different sample based ensemble heuristics, for different amounts of used heuristics.

n	**5**	**10**	**50**	**100**	**250**	**500**
$\mathtt{S}_1$	**889**	888	867	852	815	773
$\mathtt{K}_1$	904	**911**	896	861	790	747
$\mathtt{K}_2$	835	831	**888**	865	669	615
$\mathtt{L}_1$	911	**921**	884	856	771	734
$\mathtt{L}_2$	**840**	**840**	796	756	670	615

Table 4.4: Comparison over different $\texttt{n}$ for different ensemble heuristics.

The best results are achieved with smaller n. In fact, the best coverage, 921, is achieved with $\texttt{L}_1^{10}$.
However, the number of expansions decrease with increasing n, indicating that the ensemble heuristics are more accurate when using more heuristics. But the more heuristics are used, the more out of time errors occur. This is both because more heuristics need to be computed and because they all need to be considered for each state which is evaluated.
The absolute numbers in Table 4.4 should be taken with a grain of salt, as they are produced with randomness and differ for each run. Running the tests multiple times shows that $n = 5$ is too small, as the variety of generated samples is not big enough to produce good results. For $n = 50$ on the other hand, the time consumption of computing all heuristics is to high. Further tests would be necessary, to see whether $n = 10$ really is the sweet spot, or if it lies somewhere else in between 5 and 50

4.1.4 Additional Constraint on the Initial State

The best coverage for all configurations is obtained with the additional constraints on the initial state.

	all-N-I	div-N-I	S^1_{1000}-N-I	M_1-D-I	M_2-D-I
Coverage	**965**	956	963	950	906
Expansions	8532	7741	9040	6585	**6561**
Total Time	**0.27**	0.70	0.33	0.60	1.21
Search Time	0.21	0.70	0.21	**0.17**	**0.17**
Out of Memory	837	716	843	729	**705**
Out of Time	8	127	**4**	115	183

Table 4.5: Test results for the additional constraint on the initial state.

None of the configurations shows a higher total time nor search time than before, despite the fact that the LP is solved twice. First, to optimize the heuristic value for the initial state, then for the actual optimization function. In addition, the number of expansions is smaller than before, and less time and memory errors occur in total. The plots 4.2 and 4.3 show very nicely how the number of expansions and the search time is smaller for `all-N-I`, compared to `all-N`.

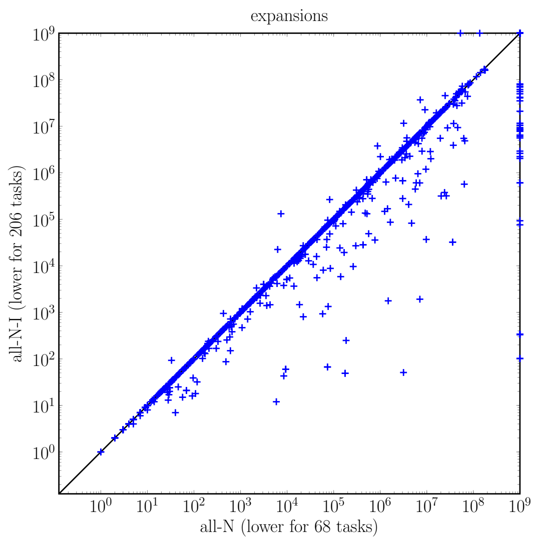

Figure 4.2: Number of expansions for `all` for both LPs per problem

It is remarkable, that `all-N-I` has a better coverage than `max-N` (948). Presumably, `max` chooses `init` for states which are close to the initial state and `all` otherwise. `all-N-I` on the other hand has high heuristic values on average, with a main focus on on the initial state. This results in two very similar heuristics. The higher coverage of `all-N-I` is due to its lower time consumption. The configuration `max-N` takes longer to compute, as it must generate two independent LPs. It takes less effort, to build an LP, solve it and then resolve it, as `all-N-I` does, since the presolving of the LP is only done once. As plot 4.4 shows, the search time is much lower for `all-N-I` than for `max`. This is because `max` must

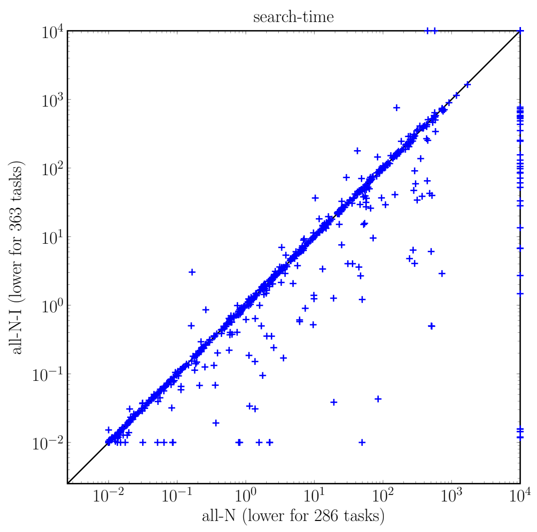

Figure 4.3: Search Time for `all` for both LPs per problem

consider two heuristics for each evaluated state.

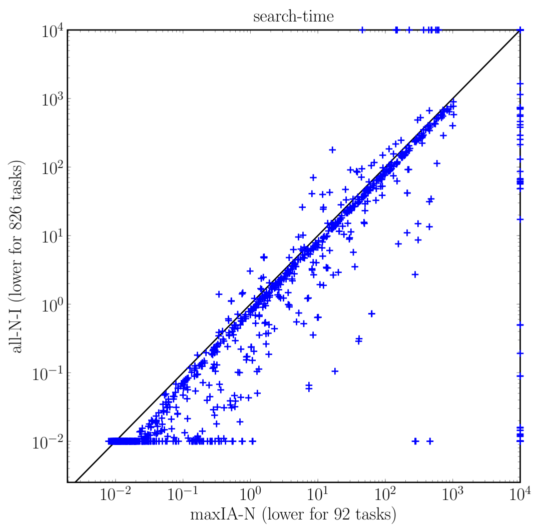

Figure 4.4: Search Time for `all` for both LPs per problem

4.1.5 Additional Constraints on Random States

In order to test the additional constraints on random states, we first tested different amounts of samples. The pretests showed that 5 samples are best, which is why all following results are for $n = 5$.

In comparison to Table 4.1, the coverage is higher with the additional constraints, as can be seen in Table 4.6. For `init-N-R` the number of expansions, search time and total time are better as well. For all other listed configurations, these attributes are better without the additional constraints on random states.

As we only used 5 samples, this could be a coincidence. A higher amount of samples would

	`all-N-R`	`init-N-R`	`div-N-R`	`M`$_2$`-D-R`
Coverage	**930**	898	905	863
Expansions	11672	16182	11306	**9190**
Total Time	**0.35**	0.44	0.76	1.51
Search Time	**0.34**	0.44	0.76	1.37
Out of Memory	858	899	798	**728**
Out of Time	16	**11**	105	205

Table 4.6: Test results for the additional constraints on random states.

be better, since an outlier in the samples could then be compensated with the other sample states. As is, with only 5 samples, if the random walk occurs in the wrong direction, the resulting potentials are distorted. However, for more samples, building the LP is not feasible inside our time limit. This is, because we implemented the additional constraint such that the Linear Program is optimized for one state after the other and the new constraint is added to the LP before optimizing for the next state. Another solution, which could take more samples into account, would be to optimize the LP for multiple samples at a time, and then add their respective heuristic values to the solver as a constraint.
Taking mutexes and disambiguations into account when generating the samples could possibly enhance the result as well.
In Table 4.7, the results for the combined additional constraints on the initial and on random states are listed. The results are better compared to only using the constraints on random states, but still not as good as with the sole use of the constraint on the initial state. In fact, all attributes lie between the configurations with only one type of additional constraints.

	`all-N-I-R`	`div-N-I-R`	`M`$_2$`-D-I-R`
Coverage	**963**	954	895
Expansions	8671	8897	**6245**
Total Time	**0.29**	0.74	1.36
Search Time	**0.29**	0.74	1.24
Out of Memory	827	798	**694**
Out of Time	**26**	58	206

Table 4.7: Test results for the combination of the additional constraints on the initial state and on random states.

We also tested different variants of `max`. Using `max` of `init` and `all` with the additional constraint on the initial state (`max(init-N, all-N-I)`), the coverage was 957 and therefore better than without the constraint. It was a little lower with the additional constraints on random states for both `init` and `all` (`max(init-N-R, all-N-R)`, 927). However, it was the only configuration, where the combined additional constraints yielded a better coverage, as the single use of the additional constraint of the initial state. For `max(init-N-R, all-N-I-R)`, the coverage was 960.

4.2 Comparison to Fišer et al.

In comparison to our results, Fišer et al. (2020) obtained a higher coverage using mutexes and disambiguations. In order to find out why, we used their code (`cpddl`[1]) for translating and pre processing the planning tasks and generating the potentials. The potentials where then used in the Fast Downward A* search. The results can be seen in Table 4.8.

	all-N	**all-D**	**max-D**	**div-D**	**S^1_{1000}-D**	**M_1-D**	**K^{50}_1-D**
Coverage	926	952	961	946	985	**962**	923
Expansions	37795	29386	47314	**24544**	26908	31841	49766
Total Time	0.61	0.52	0.80	**0.45**	0.48	0.56	0.71
Search Time	0.54	0.46	0.71	**0.40**	0.42	0.49	0.63
Out of Memory	833	807	792	**643**	770	791	812
Out of Time	**2**	**2**	5	92	3	5	5

Table 4.8: Test results obtained with `cpddl`

For `all-N`, the coverage is worse than in our implementation. All other configurations yield a higher coverage with `cpddl`.

Since `cpddl` uses a different preprocessor, we should not compare the relative attributes expansion, total time and search time with our former results.

What we can compare though are the out of time and out of memory errors. That they are significantly lower than before, especially the out of time errors, indicates that `cpddl` is faster than our implementation. As this holds for `all-N` as well, we can assume this is not only due to our implementation of the mutex calculation and the mutex based LP, but also due to the fact that `cpddl` uses a different preprocessor as Fast Downward.

[1] https://gitlab.com/danfis/cpddl/

5 Conclusions

In this work we consider potential heuristics to solve classical planning tasks. The potentials for each fact of the problem are computed with a Linear Program. The LP optimizes the potentials with regard to an optimization function, such that all given constraints hold.
Fišer et al. (2020) proposed to use mutexes and disambiguations for building the LP-constraints. It is an additional option, which can be used for all potential heuristics. However, the results we obtained with our implementation are not better than for the original LP. Our experimental evaluation shows that this is mainly due to the high amount of time needed to obtain the mutexes.
Further, we strengthen an optimization function with mutexes and disambiguations as introduced by Fišer et al. (2020) We give a weight to each fact of the problem, according to its relative appearance in all reachable states. The optimization function maximizes the sum over all potentials multiplied with the weight of their respective fact. The evaluation shows that the resulting heuristic is good, but not better than the heuristics generated with other optimization functions.
Like Fišer et al. (2020), we also strengthened ensemble heuristics. For this approach, multiple heuristics are computed and for each state the maximal value of all heuristics is used as the heuristic value. In our case, for each heuristic we uniformly randomly sample a partial state of a given size. The weights of the facts represent their relative appearance in all reachable states, which extend this partial state. They are then used to compute the potentials for this heuristic. The results are better for this approach as for the single heuristic. A partial state of greater starting size yields better results. The question which remains open is how to select the partial states, other than uniformly at random, in order to solve more problems.
Both of the approaches to strengthen optimization functions are good. Once the search has started, they are very efficient. However, computing the potentials needs a lot of time. The current implementation may extend partial states by an arbitrary amount of facts. But extending for more than two facts does decrease the amount of problems solved by using the resulting heuristic drastically. Optimizing the implementation for extending states by one or two additional facts could decrease the time needed and enhance the obtained results.
Last, we add additional constraints to the Linear Program. The constraints are obtained by solving the LP for a state, i.e., the initial state or a random state. The resulting heuristic

value for the respective state is then used in an additional constraint. The additional constraint on the initial state, proposed by Fišer et al. (2020), yields very good results. It exceeds the additional constraints on random states. The results of the combination of both the additional constraints lies in between the results obtained by using them separately in terms of quality and efficiency.

For the additional constraints on random states, we generate the states with a random walk. Using mutexes and disambiguations could improve the quality of the received states. Another approach to improve the results would be to solve the LP for multiple states at a time and add the respective constraints. This way, more states could be taken into account which would decrease the effect of a single outlier.

The comparison of our results to the results from the evaluation of Fišer et al. (2020) shows that using mutexes and disambiguations has more relevance in their implementations. The translation and pre processing of the planning tasks and generating the potentials takes longer in Fast Downward, leaving less time for the search.

Bibliography

Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

Daniel Fišer, Rostislav Horčík, and Antonín Komenda. Strengthening potential heuristics with mutexes and disambiguations. In *Proceedings of the Thirtienth International Conference on Automated Planning and Scheduling*, pages 124–133, 2020.

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

Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling*, pages 162–169, 2015.

Florian Pommerening and Malte Helmert. A normal form for classical planning tasks. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling*, pages 188–192. AAAI Press, 2015.

Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp. From non-negative to general operator cost partitioning. In *In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 3335 – 3341. AAAI Press, 2015.

Jendrik Seipp, Florian Pommerening, and Malte Helmert. New optimization functions for potential heuristics. 2015.

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

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

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

Author — Autor
Salome Müller

Matriculation number — Matrikelnummer
2017-063-058

Title of work — Titel der Arbeit
Mutex Based Potential Heuristics

Type of work — Typ der Arbeit
Bachelor thesis

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

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, 06. 11. 2020

Salome Müller

Signature — Unterschrift