Enhancing Efficiency of LP-based Heuristic Search in Optimal Planning

Master Thesis
by Renato Farruggio

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Florian Pommerening

February 7, 2024

Acknowledgments

I would like to thank Prof. Malte Helmert, head of the Artificial Intelligence Research Group at the University of Basel, for giving me the opportunity to write my thesis in his group. I want to thank Florian Pommerening for his guidance, for sharing his expertise and knowledge of LP-based heuristic search, the different heuristics, and Fast Downward with me, and for his diligent criticisms of my writing. I would also like to thank Augusto Blaas Corrêa, who sometimes replaced Florian Pommerening when he was not available for the weekly meetings. Next, I would like to thank the people from the AI Research Grop, in particular Remo Christen and Clemens Büchner, who always took some minutes out of their day to answer my questions about Fast Downward when I walked into their office. Next, I would like to thank my family and friends for their support during this time, for reading my thesis, giving feedback, and also giving kind words when needed, and listened to me when I tried to wrap my head around a topic. Thank you Berno, Katharina, Ken, Luka, Natascha, Samuel and Timotheus.

All your support is greatly appreciated.

Abstract

Optimal planning is an ongoing topic of research, and requires efficient heuristic search algorithms. One way of calculating such heuristics is through the use of Linear Programs (LPs) and solvers thereof. This thesis investigates the efficiency of LP-based heuristic search strategies of different heuristics, focusing on how different LP solving strategies and solver settings impact the performance of calculating these heuristics. Using the Fast Downward planning system and a comprehensive benchmark set of planning tasks, we conducted a series of experiments to determine the effectiveness of the primal and dual simplex methods and the primal-dual logarithmic barrier method. Our results show that the choice of the LP solver and the application of specific solver settings influence the efficiency of calculating the required heuristics, and showed that the default setting of CPLEX is not optimal in some cases and can be enhanced by specifying an LP-solver or using other non-default solver settings. This thesis lays the groundwork for future research of using different LP solving algorithms and solver settings in the context of LP-based heuristic search in optimal planning.

Contents

1 Introduction **4**

2 Background **4**
2.1 Linear Programs 4
2.2 Solving Linear Programs 6
2.2.1 Primal Simplex Algorithm 6
2.2.2 Dual Simplex Algorithm 7
2.2.3 Barrier Methods 7
2.2.4 Other Algorithms 7
2.3 Solver Options 7
2.3.1 Warm Starts 8
2.3.2 Preprocessing 8
2.3.3 Crossover 8
2.3.4 Twophase Warm Starts 8
2.4 Planning Tasks 10
2.5 Heuristics for Planning Tasks 12
2.5.1 Operator-Counting Heuristics 12
2.5.2 State Equation Heuristic 12
2.5.3 Delete Relaxation Heuristic 14
2.5.4 Optimal Cost Partitioning of Disjunctive Action Landmarks 15
2.5.5 Post-Hoc Optimization Heuristic 15
2.5.6 Potential Heuristic 16

3 Experiments **17**
3.1 Preliminaries 17
3.2 Barrier Method 18
3.3 Preprocessing 19

4 Conclusion **20**

5 Appendix **23**

1 Introduction

Optimal planning is the task of finding an optimal sequence of actions, that leads from an initial state to a goal state in a pre-defined environment. The widely known Rubik's Cube can be expressed as a planning task [18], but there are also more practical applications such as planning elevator movements [17] and many logistics tasks. Optimal planning is an ongoing topic of research, and each year numerous papers are submitted to the International Conference on Automated Planning and Scheduling (ICAPS). One branch of optimal planning is using heuristic-based search algorithms like A* that estimate the distance to the goal in each step. We focus on heuristics that can be formulated as linear programs (LPs). An LP is a mathematical optimization problem with the goal of finding variable values subject to given constraints, that optimize a given linear objective function. LP solvers are computer programs that find an optimal solution to a given LP, and are mostly used as a black box [23] in this context. We analyze the LPs of the heuristics of different planning tasks, hypothesise about the impact of different solving strategies applied by the LP solver on these problems, and run experiments to test the hypotheses.

Section 1 gives as an outline and an and serves as an overview over this work. In Section 2 we introduce various concepts that are important for this thesis. We start with linear programs, continue with planning tasks and relevant heuristics, and finally we introduce the reader to planner settings we examined. In Section 3, we present our experiments and findings. Finally, we conclude with Section 4 and give an outlook on what can be done in the future.

2 Background

In this section we introduce linear programs, planning tasks, and heuristics. These concepts are fundamental for this thesis.

2.1 Linear Programs

We start by introducing linear programs, before introducing heuristics that can be formulated as linear programs.

A linear program is a common optimization problem that is defined by a linear function, and linear constraints in the form of linear inequalities. The linear constraints define a feasible region in the form of a convex polytope. A convex polytope is a geometric object that can be represented as a finite intersection of half-spaces, where each half-space is defined by a linear inequality. It is a higher-dimensional generalization of a polygon in two dimensions or a polyhedron in three dimensions. In the case of a linear program, each face of the polytope corresponds to one relevant constraint of the linear program. The area within the polytope, including its borders, is called the feasible region.

Definition 2.1 (Linear program)**.** Let $A \in \mathbb{R}^{m \times n}$ be a matrix called the *constraint matrix*, let $b \in \mathbb{R}^m$ be a vector called the *bounds vector*, and let $c \in \mathbb{R}^n$ be a vector called the *cost vector*. Furthermore, let $f : x \mapsto c^T x$ be a function called the *objective function*. Then, a *Linear Program (LP)* is defined as

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \geq b \\ & x \geq 0 \end{array}$$

It has n variables and m constraints (not counting the non-negativity constraints). We can observe that this formulation is equivalent to a maximization problem

$$\begin{array}{ll} \text{maximize} & -c^T x \\ \text{subject to} & -Ax \leq -b \\ & x \geq 0 \end{array}$$

In particular, any linear program can be written as a minimization or a maximization problem.

Definition 2.2 (Feasible region)**.** Let

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \geq b \\ & x \geq 0 \end{array}$$

be a linear program. The *feasible region* of the LP is the set $\{x \mid x \in \mathbb{R}^n, Ax \geq b, x \geq 0\}$. If the feasible region of the LP is empty, then the problem is *infeasible*, otherwise the problem is *feasible*. The solution of the LP is a point x^* within the feasible region that is optimal with respect to the objective function. The value of the objective function evaluated at x^*, i.e. $f(x^*)$, is called the *objective value*. If the objective value of a feasible LP can be arbitrary low (or high, in the case of an maximization problem), then the problem is called *unbounded (feasible)*. Otherwise, it is called *bounded (feasible)*.

A problem is called *degenerate* if there exist different sets of faces of the polygon that define the optimal solution. In other words, the solution is over-determined, and the linear constraints that are relevant for the solution are not linearly independent. We also call a linear problem *degenerate*, if the solution is under-determined, in which case the objective function yields an optimal value for different points in the feasible region. In theory both these cases are not a problem when we are only interested in the objective value, but in practise the algorithms used to find these solutions can cycle through different sets of faces, in the prior case, or different points, in the latter case. Luckily, CPLEX has different tactics, for example perturbation methods[1, 8], to detect these cases and prevent cycling, so we don't have to worry about them.

Definition 2.3 (Primal LP and Dual LP)**.** For all linear programs there exists a corresponding dual linear program. Let

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \geq b \\ & x \geq 0 \end{array}$$

be a linear program called the *primal LP*. Its corresponding *dual LP* (or just *dual*) is the linear program

$$\begin{array}{ll} \text{maximize} & b^T y \\ \text{subject to} & A^T y \leq c \\ & y \geq 0 \end{array}$$

Note that the number of constraints and number of variables has switched, so for a primal LP with n variables and m constraints, its dual has m variables and n constraints (not counting the non-negativity constraints).

For any primal LP, the dual of its dual is again the primal LP. The relationship between the solutions of the primal LP and its dual LP are described by the Weak Duality theorem and Strong Duality theorem[1].

Theorem 2.1 (Weak Duality)**.** Let there be a primal LP and a corresponding dual LP. For any point x in the feasible region of the primal LP and any point y of the dual LP, it holds that

$$c^T x \geq b^T y$$

[1]The primary source is not available, but according to George Dantzigs foreword in [26], these theorems were first conjectured by John von Neumann and later rigorously proven by Albert W. Tucker and his group at Princeton.

This means that the objective value of the dual LP is a lower bound of the objective value of the primal LP. And similarly, the objective value of the primal LP is an upper bound of the objective value of the dual LP.

A direct consequence from the Weak Duality is that if an LP is unbounded then its dual is infeasible. It also follows, that if an LP is feasible, then its dual is bounded.

Theorem 2.2 (Strong Duality)**.** Let there be a primal LP and a corresponding dual LP. If the primal LP is bounded feasible and has an objective value $c^T x^*$, then the dual LP is bounded feasible and has an objective value $b^T y^*$, and it holds that

$$c^T x^* = b^T y^*$$

2.2 Solving Linear Programs

We will now introduce different strategies for solving linear programs. Since we use CPLEX for our experiments, we will discuss the solving strategies for linear programs that are implemented in CPLEX. The different strategies we consider are the primal simplex algorithm, the dual simplex algorithm, and the barrier method. We will also, very briefly, cover the network simplex algorithm, sifting algorithm, and concurrent optimizer, and justify why we did not use them.

2.2.1 Primal Simplex Algorithm

The primal simplex algorithm, also called the Dantzig Simplex Method, named after George Dantzig, is a typical approach to find optimal solutions of Linear Programs[12]. It works by starting at a vertex, i.e. a "corner", of the feasible region, and then moving along the edges of the feasible region to find the vertex with the highest evaluation of the objective function, while maintaining feasibility at all steps. Finding such an initial point is as hard as solving the LP, but it can be achieved by adding additional variables. But usually, if such a point is not known, it is a good approach to use the dual simplex algorithm instead, and this is what CPLEX does by default. But let us suppose that a point within the feasible region is known, for a linear program consisting of a linear objective function that we want to minimize, and a set of constraints, in the form of linear equations $Ax \geq b$, and non-negativity constraints for the variables $x_1, \ldots, x_n$. Each of the m constraints in $Ax \geq b$ is of the form $a_i x \geq b_i$, where a_i is the i-th row vector of A. In the Primal Simplex Method, slack variables $s_1, \ldots, s_m$ are introduced, together with non-negativity constraints $s_i \geq 0$ for all $1 \leq i \leq m$. The slack variables then get integrated into the LP, so that the constraints inequalities become equalities: $a_i x - s_i = b_i$ for $1 \leq i \leq m$. Any set with m elements from $\{x_1, \ldots, x_n, s_1, \ldots, s_m\}$ can form a *basis*. Each basis corresponds to a vertex (intersection of constraint bounds) in the solution space. A basis is feasible if all constraints of the problem are satisfied. In particular, if all basic variables satisfy the non-negativity constraints. The optimal solution can be represented by one of the feasible bases. Initially, the basis contains all m slack variables. The variables in the basis are called "basic variables", and the rest are the "nonbasic variables". In each iteration of the optimization, the optimizer performs a pivoting-step. As a result, one basic variable leaves the basis, and another non-basic variable enters the basis. After the optimization is completed, all nonbasic variables have the value 0, and usually, all basic variables are non-zero. If the problem is degenerate, then some basic variables can be 0 as well. In that case, the solver could cycle through several solutions. However, as mentioned before, CPLEX has methods to avoid such cycling. Since in each iteration, the optimizer searches for the non-basic variable that offers the largest improvement for the objective function, when added to the basis, there is an advantage when the problem has a lot more variables than constraints. We think that this is because the search space is high-dimensional with a lot fewer constraints than dimensions. This might sound counter-intuitive at first, because the pool of non-basic variables that the algorithm can choose from at each step is larger when there are more variables, and so more computations are required in each iteration. But the key point to regard here is that the search space becomes larger, and thus includes more options for improvement at each iteration. So, the chance of there being, and

being found, in particular, a short path to a solution, increases. So, while the computations at each iterations increases if there are more variables, the number of iterations decreases, which is the deciding factor.

2.2.2 Dual Simplex Algorithm

The dual simplex algorithm works similarly to the primal simplex algorithm, with a few key differences. Instead of starting at a point in the feasible region, it starts a point within the feasible region of the dual LP. It then iteratively finds vertices that remain in the feasible region of the dual LP, while trying to minimize the difference between the evaluation of the objective function of the primal LP and the evaluation of the objective function of the dual LP. When this difference is 0, the algorithm terminates and returns the basis and objective value of the primal LP.

2.2.3 Barrier Methods

In contrast to the simplex methods, the barrier methods move inside a convex feasible region, searching for the optimal solution from within. [2, 11].

There are different types of interior point methods. Karmakar has presented the first algorithm that can solve LPs efficiently in polynomial time, called the Karmakar's algorithm[13]. It was the first of a family of interior point methods. Interior point methods differ in the way they define the barrier function. Karmakar's method, for example, uses a potential reduction method.

The method used in CPLEX is a primal-dual logarithmic barrier method. We omit the exact details here; they can be found in various books, such as [11] and [2]. Barrier Methods have a few benefits over the simplex methods. It can parallelized, and it can handle degenerate problems better than simplex algorithms.

For us, an interesting practise is called crossover, in which the solution moves from a point in the interior of the feasible region to a close vertex. This happens as a last step of the barrier method, and can have benefits, but also comes with some downsides. We will discuss crossover a bit later, in Section 2.3.

2.2.4 Other Algorithms

The network simplex algorithm is a simplex algorithm that works under very specific constraints. These problems need to have the following property: Each column has exactly 2 non-zero entries, where the two coefficients are $a+1$ and $a-1$, or can at least be transformed to such a form[27]. The sifting algorithm works by solving a sequence of LP subproblems, starting with only a few columns and rows, and dynamically adding more to the problem while solving. The sifting algorithm can be very effective on problems with many more variables than equations[27]. The concurrent optimizer from CPLEX runs different optimizers concurrently, one on each thread. Some optimizers can also be run multi-threaded, for example the primal-dual logarithmic barrier optimizer.

We did not use the Network Simplex algorithm nor the Sifting algorithm, since the LPs we faced were not of the form described. We run all searches single-threaded, so that the results are deterministic, and so the concurrent optimizer is not interesting for our use case either.

2.3 Solver Options

CPLEX has many options for regulating how a problem should be solved. Some of these options determine what happens before the LP solving is started, others influence what happens during solving.

2.3.1 Warm Starts

For both the simplex and the barrier method, it is necessary to have a basis that represents a point within the feasible region. When such a basis is not available, then an additional step to compute one is required. Solving a problem with such an additional step is called a cold start. Such an initial basis can be provided by the user, but it can also have remained in the solver from the previously solved problem. When using simplex for solving an optimization problem, the basis of the solution found remains loaded in the solver until the solver shuts down, until the basis gets manually deleted by the user, or until another problem is loaded for which the saved basis does not lie within the feasible region of neither the primal nor the dual problem. If the saved basis can be used as an initial basis for the new problem, then it is called a *warm start* (or *advanced start*). When using warm starts in combination with the barrier method, then it will just continue with the simplex algorithm, since the barrier method requires a point in the interior, i.e. not on the edge, of the feasible region.

This setting by default is turned on in CPLEX, so whenever a basis is present in the solver before solving a problem, CPLEX will check if it is in the feasible region of the primal or dual problem, and then starts the primal or dual simplex, respectively, with the basis present as a starting basis.

2.3.2 Preprocessing

CPLEX has a range of options to preprocess any given LP. Preprocessing consists of presolving and aggregation. These steps contains several reformulations of the problem, such as replacing trivially fixed variables with their values, removing unnecessary constraints, etc. We cannot know for sure what happens during the two phases, because the documentation is tenuous. We could observe that the number of variables and constraints shrinks during this phase. This step is turned on by default, but can be turned off.

2.3.3 Crossover

The result of the barrier algorithm is a point that does not necessarily lie on the boundary of the feasible region, but is only guaranteed to find an optimal value. In this case, CPLEX does not generate a base like it does for the simplex algorithms. *crossover* shifts the point found to the nearest vertex, but does not change the objective value. Crossover reduces the number of nonzero values of the variables, because in the initial solution point, many variables have values like 0.00001 or 0.99999. This is, because by the nature of the method, the search finds values within the feasible region, i.e. not on the boundary, meaning that variables will not have the value 0.

Most of the time, crossover is needed. Crossover creates a basis, so that the problem can be warm-started. In this case, CPLEX will continue with a simplex algorithm using that basis for the LP, if a warm start is possible. Another advantage is, that if the variable assignments are important, then we are often times interested in solutions that have fewer nonzero values. In this case, crossover is also needed.

2.3.4 Twophase Warm Starts

A new approach that is not implemented in CPLEX, but can be achieved by combining its functionalities, is what we call a *twophase warm start*, where changes are made in two separate phases, between which an additional LP is solved, with the goal of allowing warm starts when this is otherwise not possible. The idea behind it is as follows. When only changing the objective function, then the feasible region remains the same. Therefore, in this case, any old solution found, will be in the feasible region of the primal LP, which can then be warm started. On the other hand, changing the constraint bounds of the primal LP corresponds to a change in the objective function of the dual LP. Therefore, in this case, the dual LP can be warm started for the same reason. With these observations, it stands to reason that it is possible to retain updates

made to the constraint bounds, and only update the objective function, then solve this auxiliary LP using the primal simplex method with a warm start, and then update the retained updates to the constraint bounds, and solve the resulting LP using dual simplex with a warm start. The order of these two steps could also be switched. While this allows for warm starts of problems that could otherwise not be warm started, note that the solution attained could very well be different from a solution attained by a cold start.

2.4 Planning Tasks

We now define SAS$^+$ planning tasks and related concepts, which serve as a foundation for introducing various heuristics later on. A *state space variable* (or just *variable*) is a symbol v with an associated *domain* $\text{dom}(v)$, which is a finite non-empty set of values. A *partial assignment* over variables V is a function f that maps a subset of V to elements of their domain: $f : X \subseteq V \to \bigcup_{v \in V} \text{dom}(v)$ such that $f(v) \in \text{dom}(v)$ for all $v \in X$. We refer to the set on which f is defined as $vars(f)$. A *state* s is a partial assignment over V, where s is defined for all variables in V, i.e. where $vars(s) = V$.

A *fact* is a partial assignment of exactly one variable. For example, for a variable v, the set $\{v \mapsto d \mid d \in \text{dom}(v)\}$ is a set of facts. Note that for any given state and variable v, there is exactly one true fact that includes v, and all other facts that include v are false (except for delete-free planning tasks, that will be introduced in 2.5.3).

Definition 2.4 (SAS$^+$ Planning Task)**.** A SAS$^+$ planning task (or just *task*) is a tuple $\Pi = \langle V, O, I, G, cost \rangle$, where

- V is a finite set of variables
- O is a set of operators of the form $\langle pre(o), \mathit{eff}(o) \rangle$, where the *preconditions* pre and *effects* eff are partial assignments over V
- I is a state called the start state
- G is a partial assignment that describes all goal states
- $cost$ is a function $cost : O \to \mathbb{R}_{\geq 0}$ that assigns a non-negative cost to each operator

We say that an operator o is *applicable* in a state s if s and $pre(o)$ are consistent, i.e. there is no $v \in V$ with $s(v) \neq pre(o)(v)$. If operator o is applicable in state s, then resulting state s' of applying o in s is written as $s' = s[o]$, and is defined as

$$s'(o) = \begin{cases} \mathit{eff}(o)(v) & \text{if } v \in vars(\mathit{eff}(o)) \\ s(v) & \text{otherwise.} \end{cases}$$

A *plan* π *for a state* s is a sequence of applicable operators $\langle o_1, \ldots, o_k \rangle$, such that $s[o_1][o_2] \ldots [o_k]$ is a goal state. We introduce $s[o_1, o_2, \ldots, o_k]$ as an abbreviation of $s[o_1][o_2] \ldots [o_k]$.

A *plan for a planning task* Π is a plan for the initial state of Π.

The *length of a plan* $\langle o_1, \ldots, o_k \rangle$ is the number of operators in the plan, i.e. the number k.

The *cost of a plan* $\langle o_1, \ldots, o_k \rangle$ is simply the sum of the costs of all the operators applied: $\sum_{i=1}^{k} cost(o_i)$.

Before we introduce heuristics next, we present an example task, that we can come back to afterwards. We use an instance of the Miconic task [17]. There are two floors $f0$ and $f1$, an elevator, and a passenger that wants to get from $f1$ to $f0$ using the elevator. The scenario is depicted in Figure 1.

Example planning task We define Π_{mic} to be the Miconic task instance from Figure 1, formally defined as $\Pi_{mic} = \langle V, O, I, G, cost \rangle$, with

- $V = \{elevator, passenger\}$ with
  - $\text{dom}(elevator) = \{f0, f1\}$
  - $\text{dom}(passenger) = \{f0, f1, e\}$
- $O = \{\text{up}, \text{down}\} \cup \{\text{enter-}f, \text{leave-}f\}$ for $f \in \{f0, f1\}$
  - $\text{up} = \langle elevator = f0, elevator = f1 \rangle$

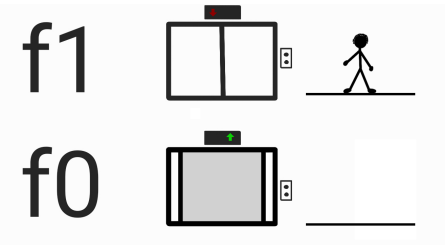

Figure 1: Example of a Miconic task, where the elevator is on floor $f0$ and the passenger is on floor $f1$. The goal is for the passenger to be on floor $f0$.

  - down $= \langle elevator = f1, elevator = f0 \rangle$
  - enter-$f = \langle \{elevator \mapsto f, passenger \mapsto f\}, \{passenger \mapsto e\} \rangle$ for $f \in \{f0, f1\}$
  - leave-$f = \langle \{elevator \mapsto f, passenger \mapsto e\}, \{passenger \mapsto f\} \rangle$ for $f \in \{f0, f1\}$
- $I = \{elevator \mapsto f0, passenger \mapsto f1\}$
- $G = \{passenger \mapsto f0\}$
- $cost(o) = 1$ for all $o \in O$

The optimal plan for Π_{mic} is $\langle \text{up}, \text{enter-}f1, \text{down}, \text{leave-}f0 \rangle$ with a cost of 4.

2.5 Heuristics for Planning Tasks

A common approach for optimal planning is the A* algorithm presented by Hart et al. [5], which uses a heuristic function (or just *heuristic*) that calculates a heuristic value for a given state. Informally, the heuristic can be understood as an estimation of the cheapest plan cost from a given state to a goal state. Formally, a *heuristic* is a function that maps states to real numbers or infinity. The *optimal heuristic* $h^*(s)$ of state s is the cost of the cheapest plan for s, or ∞ if no plan exists. In order for a heuristic to be useful for optimal planning with the A* algorithm, it has to be an admissible heuristic. A heuristic h is called *admissible*, if it never overestimates the true cost, i.e. if $h(s) \leq h^*(s)$ for all states s.

In this section, we cover a family of so called operator-counting heuristics, and a heuristic called potential heuristic.

2.5.1 Operator-Counting Heuristics

The heuristics in the family of operator-counting heuristics are based on linear programs, as defined in 2.1, with similar structures. Pommerening et al. [21] showed that these heuristics can be covered by a common framework based on the notion of *operator-counting constraints*. We use their definition:

Definition 2.5 (Operator-counting constraints [21])**.** Let Π be a planning task with operator set O, and let s be one of its states. Let $\mathcal{Y}$ be a set of non-negative real-valued and integer variables, including an integer variable Y_o for each operator $o \in O$ along with any number of additional variables. The variables Y_o are called *operator-counting variables*.
If π is a plan for s, we denote the number of occurrences of operator $o \in O$ in π with Y_o^π. A set of linear inequalities over $\mathcal{Y}$ is called an *operator-counting constraint* for s if for every plan π for s, there exists a feasible solution with $\mathrm{Y}_o = \mathrm{Y}_o^\pi$ for all $o \in O$.
A *constraint set* for s is a set of operator-counting constraints for s where the only common variables between constraints are the operator-counting variables.

The operator-counting heuristics are then defined as follows:

Definition 2.6 (Operator-counting heuristic)**.** An *operator-counting heuristic* of a state s is the objective value of an optimal solution to a linear program with the objective function

$$\text{minimize} \sum_{o \in O} \mathit{cost}(o)\mathrm{Y}_o$$

subject to a set of operator-counting constraints, where Y_o denotes the number of occurrences of operator o in a plan for state s.

From this family, we next introduce the state equation heuristic, the delete relaxation heuristic, the disjunctive action landmark heuristic, the optimal cost partitioning heuristic for abstractions, and the post-hoc optimization heuristic.

2.5.2 State Equation Heuristic

The term *state equation heuristic* was presented by Bonet [3]. The core idea is to use facts that can be true or false in each state. Applying an operator in a state can cause each fact to become true, false, or remain unchanged.

Let us consider the example task from Figure 1. It is clear that the passenger can only leave the elevator after having entered it. A possible fact we can use here is *passenger* $\mapsto e$. We can now express this constraint as $\mathrm{Y}_{\text{enter-}f0} + \mathrm{Y}_{\text{enter-}f1} - \mathrm{Y}_{\text{leave-}f0} - \mathrm{Y}_{\text{leave-}f1} \geq 0$. Such a constraint can be formulated for each possible fact of a task. We now say that the operators enter-$f0$ and enter-$f1$ *produce* the fact *passenger* $\mapsto e$, since before applying the operators the fact is false, and afterwards it is true. Likewise, the operators leave-$f0$ and leave-$f1$ *consume* it. The difference of the number of producers and consumers of an fact is called the *net-change* of the fact. Since a

fact that is not true in the initial state cannot be consumed more often than it is produced, the net-change of such facts is non-negative at all times. For facts that are true in the initial state, they can only be consumed once more than they are produced in each state.

Another fact we can consider is $\mathit{passenger} \mapsto f0$. The producer of said fact is leave-f0 and the consumer is enter-f0. We know that it must be true in the goal, and it is not true in the initial state, therefore the corresponding net-change must be at least 1, i.e. $\mathrm{Y}_{\text{leave-}f0} - \mathrm{Y}_{\text{enter-}f0} \geq 1$. Similarly, the fact $\mathit{passenger} \mapsto f1$, which is true initially, but in the goal state it may be false. So, a plan can consume it once more than it produces it, and we get the net-change constraint $\mathrm{Y}_{\text{leave-}f1} - \mathrm{Y}_{\text{enter-}f1} \geq -1$. The other two facts are $\mathit{elevator} \mapsto f0$ and $\mathit{elevator} \mapsto f1$. We know that $\mathit{elevator} \mapsto f0$ is true in the initial state, but it does not have to be true in the goal, therefore it can be consumed once more than it is produced, so the corresponding net-change must be at least -1, i.e. $\mathrm{Y}_{\text{down}} - \mathrm{Y}_{\text{up}} \geq -1$. For $\mathit{elevator} \mapsto f1$, we know that is not initially true and it also does not have to be true in the goal. It is produced by the operator up and consumed by the operator down. Therefore, the corresponding constraint must be $\mathrm{Y}_{\text{up}} - \mathrm{Y}_{\text{down}} >= 0$. Using all such facts, we can construct the following LP for the example task Π_{mic} in the initial state:

Minimize

$$\sum_{o \in O} \mathit{cost}(o) \mathrm{Y}_o$$

subject to

$$\begin{aligned}
\mathit{passenger} \mapsto e: \quad & \mathrm{Y}_{\text{enter-}f0} + \mathrm{Y}_{\text{enter-}f1} \\
& -\mathrm{Y}_{\text{leave-}f0} - \mathrm{Y}_{\text{leave-}f1} && \geq 0 \\
\mathit{passenger} \mapsto f0: \quad & \mathrm{Y}_{\text{leave-}f0} - \mathrm{Y}_{\text{enter-}f0} && \geq 1 \\
\mathit{passenger} \mapsto f1: \quad & \mathrm{Y}_{\text{leave-}f1} - \mathrm{Y}_{\text{enter-}f1} && \geq -1 \\
\mathit{elevator} \mapsto f0: \quad & \mathrm{Y}_{\text{down}} - \mathrm{Y}_{\text{up}} && \geq -1 \\
\mathit{elevator} \mapsto f1: \quad & \mathrm{Y}_{\text{up}} - \mathrm{Y}_{\text{down}} && \geq 0 \\
\text{for all } o \in O \quad & \mathrm{Y}_o && \geq 0
\end{aligned}$$

A solution to this is to set $\mathrm{Y}_{\text{enter-}f1}$ and $\mathrm{Y}_{\text{leave-}f0}$ to 1, and all other count-variables to 0. This gives us an objective value of 2.

To define the LP for the state equation heuristic formally, we first have to extend our definition of producers and consumers slightly. Since an operator can make a fact true, which was already true, the operator should not be classified as a producer. Similarly, if an operator makes a fact false which was already false before, it should not be counted as a consumer. For this reason, we distinguish between operators that *always produce* a fact, and operators that *sometimes produce* a fact. For an operator that always produces a fact, we know for a fact, that the fact has to be false before applying the operator, whereas for an operator that sometimes produces a fact, we do not know whether the fact is true or false before applying the operator. In the same way, we have operators that *always consume* a fact, and operators that *sometimes consume* a fact. Now that we have this classification, we can observe that the expression $\sum_{o \text{ always produces } a} \mathrm{Y}_o + \sum_{o \text{ sometimes produces } a} \mathrm{Y}_o - \sum_{o \text{ always consumes } a} \mathrm{Y}_o$ is an upper bound on the actual net-change of fact a for any given plan, because it overestimates the number of producers and underestimates the number of consumers. We can also get a lower bound. For that, define $S(a)$ to be 1 if the fact a is true in the current state, and 0 otherwise. Furthermore, define $G(a) = 1$ if the fact a is required to be true in the goal, and 0 if not. Then we can see that $G(a) - S(a)$ is a lower bound on the actual net-change of a for every plan for s. With that, we are equipped to define the constraints for the state equation heuristic:

$$\begin{aligned}
G(a) - S(a) \leq & \sum_{\substack{o \text{ always} \\ \text{produces } a}} \mathrm{Y}_o + \sum_{\substack{o \text{ sometimes} \\ \text{produces } a}} \mathrm{Y}_o - \sum_{\substack{o \text{ always} \\ \text{consumes } a}} \mathrm{Y}_o && \text{for all facts } a \\
\mathrm{Y}_o \geq & 0 && \text{for all } o \in O
\end{aligned}$$

2.5.3 Delete Relaxation Heuristic

The *delete relaxation* of a planning task Π is the planning task without the delete effects. It is called the *delete-free planning task* Π^+. Specifically, the delete relaxation of an SAS$^+$ task is an altered task where facts are not consumed. This means that once a fact is true, it can never become false.

Delete relaxations are extensively studied because they provide a foundation for calculating a heuristic value for a planning task by estimating the cost of an optimal plan to the original task [10]. The *delete relaxation heuristic* of a planning task Π is the optimal solution cost of the delete-free planning task Π^+ and is denoted as h^+. The optimal cost of the delete relaxation can be lower than the optimal cost of the original task because facts need to be made true only once. Therefore, the optimal cost of the delete relaxation serves as a lower bound on the optimal cost of the original planning task. Moreover, the maximal path length of the delete-free task equals the number of operators. This is because facts are never reverted from true to false, and hence, the optimal plan includes each operator at most once, even in the worst-case scenario.

For a planning task with a set of all possible facts A, we have the following optimization variables:

- $\mathrm{U}_o \in \{0,1\}$ for $o \in O$
  indicating whether o is part of the plan
- $\mathrm{R}_a \in \{0,1\}$ for $a \in A$
  indicating whether a is true at some point in the plan
- $\mathrm{F}_{o,a} \in \{0,1\}$ for $o \in O, a \in A$
  indicating whether o is the first operator in the plan that makes a true
- $\mathrm{T}_o \in \{0,\dots,|O|\}$ for $o \in O$
  indicating the time at which operator o is applied in the plan. The first operator can be applied at time 0 and value $|O|$ indicates that the operator was not used
- $\mathrm{T}_a \in \{0,\dots,|O|\}$ for $a \in A$
  indicating the time at which fact a was true for the first time. Value 0 means that a was already true in the state s

With that, and using the notion of producers from 2.5.2, we get the following constraints for the delete relaxation heuristic:

$$
\begin{array}{lll}
\mathrm{R}_a, \mathrm{U}_o, \mathrm{F}_{o,a} \in \{0,1\} & & \\
\mathrm{T}_o, \mathrm{T}_a \in \{0,\dots,|O|\} & & \text{for all } o \in O, a \in A \\
\mathrm{R}_a = 1 & & \text{for all } a \in G \\
S(a) + \sum_{o \text{ produces } a} \mathrm{F}_{o,a} \geq \mathrm{R}_a & & \text{for all } a \in A \\
\mathrm{U}_o \geq \mathrm{F}_{o,a} & \text{for all } o \in O, \text{ s.t. } & o \text{ sometimes produces } a \\
\mathrm{R}_a \geq \mathrm{U}_o & & \text{for all } o \in O, a \in \mathit{pre}(o) \\
\mathrm{T}_a \leq \mathrm{T}_o & & \text{for all } o \in O, a \in \mathit{pre}(o) \\
\mathrm{T}_o + 1 \leq \mathrm{T}_a + M(1 - \mathrm{F}_{o,a}) & \text{for all } o \in O, \text{ s.t. } & o \text{ sometimes produces } a \\
\mathrm{U}_o \leq \mathrm{Y}_o & & \text{for all } o \in O
\end{array}
$$

where S is a constant function that maps facts in s to 1 and all other facts to 0, and M is a large enough constant, such as $M = |O| + 1$.

These problems tend to get big and hard to solve. It has been shown that different relaxations can be adopted that give a lower heuristic value, which is less useful, but they are easier to obtain, and it is often advantageous to take this trade-off[10]. We consider two such relaxations. One is the LP-relaxations, which we used for all optimization problems in this thesis. This changes the constraints $\mathrm{R}_a \in \{0,1\}$ to $0 \leq \mathrm{R}_a \leq 1$ for all $a \in A$. Similarly, we change the constraints of U_o and of $\mathrm{F}_{o,a}$. The other relaxation we consider is the time-relaxation, which effectively ignores optimization all variables T_o for all $o \in O$ and T_a for all $a \in A$, and the two types of constraints that include these variables.

Furthermore, since this is a delete relaxation of a task, we also change the objective function, and replace Y_o by Y_o^+, which denotes the number of occurrences of operator o in the delete-free planning task. Note that this can only be 0 or 1.

We end up with the following LP:

$$
\begin{array}{lll}
\text{minimize} & \sum_{o \in O} \mathit{cost}(o) \mathrm{Y}_o^+ & \\
\text{subject to} & & \\
& 0 \leq \mathrm{R}_a \leq 1 & \text{for all } a \in A \\
& 0 \leq \mathrm{U}_o \leq 1 & \text{for all } o \in O \\
& 0 \leq \mathrm{F}_{o,a} \leq 1 & \text{for all } o \in O, a \in A \\
& \mathrm{R}_a = 1 & \text{for all } a \in G \\
& S(a) + \sum_{o \text{ produces } a} \mathrm{F}_{o,a} \geq \mathrm{R}_a & \text{for all } a \in A \\
& \mathrm{U}_o - \mathrm{F}_{o,a} \geq 0 & \text{for all } o \in O, \text{ s.t. } o \text{ sometimes produces } a \\
& \mathrm{R}_a - \mathrm{U}_o \geq 0 & \text{for all } o \in O, a \in \mathit{pre}(o) \\
& \mathrm{Y}_o^+ - \mathrm{U}_o \geq 0 & \text{for all } o \in O
\end{array}
$$

where R_a, U_o and $\mathrm{F}_{o,a}$ are defined as above for all $o \in O, a \in A$, without the restriction of them being in $\{0, 1\}$.

2.5.4 Optimal Cost Partitioning of Disjunctive Action Landmarks

A *disjunctive action landmark* [7] for a state s is a set of operators of which at least one must be part of any plan for s.

For example, let us consider a task with three operators o_1, o_2, o_3 with $\mathit{cost}(o_1) = \mathit{cost}(o_3) = 5$ and $\mathit{cost}(o_2) = 7$. Let $\{o_1, o_2\}$ and $\{o_2, o_3\}$ be two disjunctive action landmarks for state s. We can see that from each of the disjunctive action landmarks individually, the minimal cost of any plan for s is 5, by taking the cheapest operators from both sets. However, we can come to a better heuristic estimate of 7 by using the count-variables Y_i for the number of occurrences of each of the three operators o_i for $i \in \{1, 2, 3\}$ in any plan for s. We can require that $\mathrm{Y}_i \geq 0$. Furthermore, from the given landmarks we can derive the two constraints $\mathrm{Y}_1 + \mathrm{Y}_2 \geq 1$ and $\mathrm{Y}_2 + \mathrm{Y}_3 \geq 1$. We can now solve the LP of minimizing the objective function $5\mathrm{Y}_1 + 7\mathrm{Y}_2 + 5\mathrm{Y}_3$, subject to the previous five constraints to get a heuristic estimate of 7.

For the general case, consider an arbitrary task with operators O. Let $\mathcal{L}$ be a set of disjunctive action landmarks for state s. The constraints for the optimal cost partitioning for the disjunctive action landmark heuristic then are:

$$
\begin{array}{ll}
\sum_{o \in L} \mathrm{Y}_o \geq 1 & \text{for all } L \in \mathcal{L} \\
\mathrm{Y}_o \geq 0 & \text{for all } o \in O.
\end{array}
$$

The idea of using linear programs to derive heuristic estimates from landmarks originates from Karpas and Domshlak [14]. Keyder et al. [16] presented an improved formulation. The formulation here was first proposed by Bonet and Helmert [4], and they also showed that it is the dual of the representation by Keyder et al.

There are different ways of generating such landmarks automatically. We used the landmark generation method proposed by Richter et al. [22].

2.5.5 Post-Hoc Optimization Heuristic

An *abstraction* of a task is a simplified task, called *abstract task*, where different states are grouped together to abstract states. One way of generating an abstract task is by using an *abstraction function* α that maps the original states to the abstract states. This task is then called the *planning task induced by* α. We call these two tasks *original task* and *abstract task*,

and talk of original objects and abstract objects, e.g. states and operators. When not defined otherwise, we denote these objects with a superscript of the abstraction function, e.g. s^α and o^α. Usually, the abstract task has a lot fewer states than the original task. We say that an abstract state s_{abs} *contains* a state s iff $s_{abs} = \alpha(s)$. The initial state in the abstract task is the abstract state that contains the initial state from the original task. The abstract goal states are all abstract states that contain at least one original goal state. We have a transition from one abstract state a to another abstract state b iff there are two original state s and s', such that a contains s and b contains s', and there is an (original) transition from s to s'. The cost of the abstract transition from a to b then is the cost of the cheapest such original transition.

One way of defining such abstractions automatically is the systematic approach. In the *systematic approach*, we project states onto subsets of variables, called patterns, with a pattern size of no more than a pre-defined maximal pattern size. The number of abstractions grows exponentially in the number of maximal pattern size. In practice, a maximal pattern size of 2 is a good trade-off between functionality and efficiency. For example, if the maximal pattern size is 1, then for each variable there is an abstraction, and each abstract state corresponds to a value in the domain of the variable. For the miconic example with one variable and one passenger, the two abstractions are $\alpha^{\text{passenger}}$ and α^{elevator}. $\alpha^{\text{passenger}}$ maps all states onto the abstract state that corresponds to floor $f0$, floor $f1$, or the elevator e. So, there are 3 abstract states in the planning task induced by $\alpha^{\text{passenger}}$. Likewise, the planning task induced by α^{elevator} has two abstract states, one for floor $f0$ and one for floor $f1$. As an additional improvement, patterns that are a union of two disjoint patterns can be ignored if the union does not have more information than the individual patterns[20].

A trivial way of creating an admissible abstraction heuristic for a state s from several abstractions is to combine them by taking the maximum value of the optimal path costs in the abstract states: $\max_a h^\alpha(\alpha(s))$. But there exists a better way of calculating an admissible abstraction heuristic, by creating an *optimal cost partitioning* [15] over the abstract operators. The idea of cost partitioning in this context is to introduce a new (abstract) cost function $cost^\alpha$ for each abstraction that assigns new costs to abstract operators, such that the sum over all abstract instances of an operator is not greater than the original operator cost: $\sum_\alpha cost^\alpha(o^\alpha) \leq cost(o)$ for all operators $o \in O$.

For some tasks, or abstractions thereof, some operators do not contribute to the optimal solution cost, e.g. if they do not bring us closer to the goal state. We call an operator *relevant* when setting its cost to 0 changes the optimal solution cost. Otherwise, the operator is *irrelevant*. Using different abstractions, different operators might become irrelevant. Formally, we have the following definition.

For a task with operators O and a set of admissible heuristics H, the constraints for the *post-hoc optimization heuristic* [23] for a state s are as follows:

$$\begin{aligned} h(s) \leq & \sum_{\substack{o \in O \\ o \text{ is relevant} \\ \text{for } h \text{ in } s}} cost(o) \mathrm{Y}_o && \text{for all } h \in H \\ 0 \leq & \ \mathrm{Y}_o && \text{for all } o \in O \end{aligned}$$

Pommerening et al. [20] have shown that this is an admissible heuristic.

2.5.6 Potential Heuristic

We now introduce the potential heuristics. For the potential heuristic, we need potential functions, first introduced by Pommerening at al. [19]. The main idea of potential functions is to assign a numerical value, called *potential*, to all facts in a state s. The potential of fact $v \mapsto d$ is denoted as $\mathtt{P}_{v \mapsto d}$, where v is a variable and d is an element from the domain of v. The key idea here is to just calculate the LP once, and then use the result from it for the rest of the search.

For defining the LP, we use the same simplification as Pommerening et al., assuming that all variables in the effect of an operator o also occur in its precondition ($vars(\mathit{eff}(o)) \subseteq vars(\mathit{pre}(o))$) and that there is a unique goal state ($vars(G) = V$). In the implementation, we do not need this restriction. With that we get the following linear program:

$$\begin{aligned} \text{maximize} \quad & \mathtt{opt} \\ \text{subject to} \quad & \sum_{v \in V} \mathtt{P}_{v \mapsto G(v)} \leq 0 \\ & \sum_{v \in vars(\mathit{eff}(o))} (\mathtt{P}_{v \mapsto (\mathit{pre}(o))(v)} - \mathtt{P}_{v \mapsto (\mathit{eff}(o))(v)}) \leq \mathit{cost}(o) \text{ for all } o \in O \end{aligned}$$

The objective function $\mathtt{opt}$ can be chosen arbitrarily. Seipp et al. [24] have presented the following two evident choices for possible objective functions:

1. Maximizing the heuristic value of one state s, for example the initial state

$$\mathtt{opt}_s = \sum_{v \in V} \mathtt{P}_{v \mapsto s(v)}$$

2. Maximizing the average heuristic value of all states S

$$\mathtt{opt}_S = \frac{1}{|S|} \sum_{s \in S} \sum_{v \in V} \mathtt{P}_{v \mapsto s(v)}$$

aswell as more advanced potential functions, such as the sample based heuristic, and the diverse potentials heuristic.

3 Experiments

For the experiments we use the Fast Downward planning system v23.06 [6]. Calculations were performed at sciCORE[2] scientific computing core facility at University of Basel. We utilized an Intel Xeon E5-2660 Core running at 2.2 GHz and limited memory usage to 3584MB, time to 5 minutes, and 1 core per task. The results are very deterministic, with results between runs varying very little. Therefore we only used one run per experiment. The LPs are solved by CPLEX v22.1.1[9]. As a benchmark framework we use Downward Lab v7.4 [25]. We used the optimal strips plannig tasks from the AI Basel github repository[3] as our benchmarks.

For the rest of this section, we will talk of the arithmetic mean as the *average*. Furthermore, it is important to know that the tasks in the benchmark are defined in PDDL format. Fast Downward first translates such a task into an SAS$^+$ format, before the actual search starts. After this translation, the search starts. We define *total time* as the total time it takes the solver to find a plan. The only step not considered in total time is the time needed for the aforementioned translation. Finally, we define the *coverage* as the number of tasks for which a plan is found within the given time and memory constraints.

3.1 Preliminaries

There are different questions that are interesting to answer before trying to improve the current settings. By default, CPLEX can choose the solver to use itself. So, the obvious question here is, which one CPLEX actually chooses for the different heuristics. We found that CPLEX chooses either primal simplex or dual simplex, for all tasks, as can be seen in Table 1.

[2] http://scicore.unibas.ch/
[3] https://github.com/aibasel/downward-benchmarks

	SEH	DEL	OCP	PHO	IPOT	DPOT
Dual simplex for initial state	1602	1602	1602	1602	1602	1602
Primal simplex	14030725	895959	0	10721434	0	166612
Dual simplex	602773	2147267	4073024	1729194	0	0
Network simplex	0	0	0	0	0	0
Barrier	0	0	0	0	0	0
Sifting	0	0	0	0	0	0
Concurrent	0	0	0	0	0	0

Table 1: Comparison of which algorithm uses which LP solver method for all LPs after the initial LP for each task. The first row shows that CPLEX uses dual simplex for solving the LP in the initial state for all heuristics.

	SEH	DEL	OCP	PHO	IPOT	DPOT
Cold starts	**0**	**0**	565771	**0**	**0**	851

Table 2: Sum of cold starts required for all tasks in the benchmark per heuristic, ignoring the cold starts required for the first LP of each task.

3.2 Barrier Method

One metric that was interesting to look at was the number of cold starts. Since the first LP solved of each task is always a cold start, essentially by definition, we ignored these LPs. We found that only the optimal cost partitioning of landmarks heuristic (OCP) has a relevant number of cold starts, as can be seen in Table 2. The diverse potential heuristics have a sum of 851 cold starts, which is negligible. All other heuristics require no cold starts at all.

Based on that, we hypothesized that using the barrier algorithm might be beneficial for the OCP. We found that this was indeed the case, as can be seen in Table 3. For one did we see an increase in coverage by 2.4%. And even though the average presolve time increased from 0.99s to 1.03s, the total time the solver spent solving the LPs decreased by 17.4% and the average total time process of all tasks decreased by 14.4%. As expected did these improvements not occur in the other heuristics. The average total time per task increased by 450% for the state equation heuristic, by 296% for the delete relaxation heuristic, by 189% for the post-hoc optimization heuristic, and by 40% for the diverse potentials heuristic. For the initial state potential heuristic, there were no changes.

We expect the results to get worse when turning off crossover, since then, warm starts are not possible. And indeed, this is what we found. We omit the table here, but it can be found in the appendix. We found that all values were very similar, with the biggest differences being in lp solve time sum increasing by 5.8% to 12292.77, and the average total time increasing from 35.28s to 37.16s.

	OCP default	OCP barrier	Diff
Coverage	737	**755**	18
Search-out-of-time	1075	**1057**	-18
Lp solve time sum	14066.84	**11624.01**	-2442.82
Warm starts	582654	**0**	-582654
Presolve time	**0.99**	1.03	0.04
Total time	41.20	**35.28**	-5.92

Table 3: Comparison between the default setting and the barrier method for all but the first LP solved on the optimal cost partitioning of landmarks heuristic. All values are sums of all tasks that were solved by both algorithms, except for total time, which is the average over all tasks that were solved by both algorithms.

	IPOT default	IPOT no preprocessing	Diff
Coverage	834	**882**	48
Search-out-of-memory	645	**615**	-30
Search-out-of-time	331	**313**	-18
Total time	21.16	**16.33**	-4.83

Table 4: Comparison between the default setting and disabled preprocessing for the initial state potential heuristic. Total time is averaged over all tasks that, for both settings, either finished normally, ran out of memory of ran out of time. All other values are sums of all tasks that, for both settings, finished normally, ran out of memory or out of time. Bold values are the advantageous values.

	DPOT default	DPOT no preprocessing	Diff
Coverage	894	**897**	3
Search-out-of-memory	351	**360**	9
Search-out-of-time	565	**553**	-12
Total time	34.26	**32.69**	-1.57

Table 5: Comparison between the default setting and disabled preprocessing for the diverse potentials heuristic. Total time is averaged over all tasks that, for both settings, either finished normally, ran out of memory of ran out of time. All other values are sums of all tasks that, for both settings, finished normally, ran out of memory or out of time. Bold values are the advantageous values.

3.3 Preprocessing

In the case of the potential heuristics, we only solve a few heuristics once, before starting the search, and then use their values. Especially in the initial state heuristic, there is just one LP to be solved. This means that the time spent during preprocessing could be better spent solving the initial LP, since we only profit from the preprocessing once. We expect that more problems run out of memory, but fewer problems run out of time. We considered the initial state potential heuristic and the diverse potentials heuristic.

We found that disabling preprocessing enhances the efficiency of the potential heuristics, specially for the initial state potential heuristic. As apparent in Table 4, the coverage for the initial potential heuristic increased by 5.8%, and the total time decreased by 22.8%. Surprisingly, fewer problem ran out of memory.

But also for the diverse potentials heuristic, the results look promising. But in this case, the coverage only increased by 0.3%, and the total time decreased by 4.6%, as can be seen in Table 5

4 Conclusion

In this thesis we aimed for enhancing the efficiency of LP-based heuristic search in optimal plannig, focusing on the solving approaches of the LPs. We used the Fast Downward planning system, CPLEX, and a comprehensive set of benchmarks of planning tasks to understand how different LP solving strategies influence the performance of the search algorithms and discover new strategies through which planning systems can be improved.

Our experiments have demonstrated that the default settings of CPLEX are optimal in many cases. But it has also showed that the choice of the LP solving strategy, especially the barrier method, can substantially affect the efficiency of heuristic search under certain circumstances. In particular, solving the optimal cost partitioning of landmarks heuristic resulted in a significant improvement in both coverage and search time. This finding shows that selecting an appropriate LP solving strategy can improve the efficiency of the calculation of a heuristic value, and thus also the planning tasks that rely on it. Furthermore, we have found that disabling preprocessing for potential heuristics improves the overall efficiency of finding a plan for search tasks, especially for the initial state potential heuristic.

These results show that examining solver options for heuristic-based optimal planning offer a promising outlook. Further research can build on these findings by exploring a more selective settings application, based on the characteristics of the individual LPs, or by trying out other settings.

In conclusion, the work presented in this thesis contributes to our goal of enhancing LP-based heuristic calculations for optimal planning tasks. It also highlights the importance of the choice of LP solving strategies and solver settings and shows promising ways for future exploration.

References

[1] Carl M. Bender and Steven A. Orszag. *Advanced mathematical methods for scientists and engineers I: Asymptotic methods and perturbation theory.* Springer Science & Business Media, 2013.

[2] Dimitri P. Bertsekas. *Nonlinear Programming.* Athena Scientific, 1999.

[3] Blai Bonet. An Admissible Heuristic for SAS+ Planning Obtained from the State Equation. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, page 2268–2274. AAAI Press, 2013.

[4] Blai Bonet and Malte Helmert. Strengthening Landmark Heuristics via Hitting Sets. In *Proceedings of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, page 329–334. IOS Press, 2010.

[5] 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.

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

[7] Malte Helmert and Carmel Domshlak. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? *Proceedings of the International Conference on Automated Planning and Scheduling*, 19(1):162–169, 2009.

[8] Mark H. Holmes. *Introduction to perturbation methods.* Springer Science & Business Media, 2012.

[9] IBM (2022). IBM® ILOG® CPLEX® Optimization Studio v22.1.1. https://www.ibm.com/products/ilog-cplex-optimization-studio.

[10] Tatsuya Imai and Alex Fukunaga. On a Practical, Integer-Linear Programming Model for Delete-Free Tasks and its Use as a Heuristic for Cost-Optimal Planning. *Journal of Artificial Intelligence Research*, 54:631–677, 2015.

[11] Florian Jarre and Josef Stoer. *Optimierung.* Springer, 2004.

[12] Bernd Gärtner Jiří Matoušek. *Understanding and Using Linear Programming.* Springer Berlin, 2007.

[13] Narendra Karmarkar. A New Polynomial Time Algorithm for Linear Programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, page 302–311, New York, NY, USA, 1984. Association for Computing Machinery.

[14] Erez Karpas and Carmel Domshlak. Cost-Optimal Planning with Landmarks. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence*, page 1728–1733. Morgan Kaufmann Publishers Inc., 2009.

[15] Michael Katz and Carmel Domshlak. Optimal admissible composition of abstraction heuristics. *Artificial Intelligence*, 174(12):767–798, 2010.

[16] Emil Keyder, Silvia Richter, and Malte Helmert. Sound and Complete Landmarks for And/Or Graphs. In *Frontiers in Artificial Intelligence and Applications*, page 335–340. IOS Press, 2010.

[17] Jana Koehler and Kilian Schuster. Elevator Control as a Planning Problem. In *AIPS'00: Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, page 331–338. AAAI Press, 2000.

[18] Bharath Muppasani, Vishal Pallagani, Biplav Srivastava, and Forest Agostinelli. On Solving the Rubik's Cube with Domain-Independent Planners Using Standard Representations, 2023.

[19] Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp. From Non-Negative to General Operator Cost Partitioning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 29(1):3335–3341, 2015.

[20] Florian Pommerening, Gabriele Röger, and Malte Helmert. Getting the Most out of Pattern Databases for Classical Planning. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, page 2357–2364. AAAI Press, 2013.

[21] Florian Pommerening, Gabriele Röger, Malte Helmert, and Blai Bonet. LP-Based Heuristics for Cost-Optimal Planning. *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, 24:226–234, 2014.

[22] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks Revisited. In *AAAI*, volume 8, pages 975–982, 2008.

[23] Gabriele Röger and Florian Pommerening. Linear Programming for Heuristics in Optimal Planning. *Planning, Search, and Optimization : papers from the 2015 AAAI Workshop*, pages 69–76, 2015.

[24] Jendrik Seipp, Florian Pommerening, and Malte Helmert. New Optimization Functions for Potential Heuristics. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 193–201, 2015.

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

[26] Albert W. Tucker and Evar D. Nering. *Linear Programs and Related Problems*. Elsevier, 1992.

[27] Chen Yanover, Talya Meltzer, and Yair Weiss. Linear Programming Relaxations and Belief Propagation – An Empirical Study. *The Journal of Machine Learning Research*, 7:1887–1907, 2006.

5 Appendix

	OCP default	OCP barrier - no crossover	Diff
Coverage	737	**758**	19
Search-out-of-time	1073	**1054**	-19
Lp solve time sum	14660.82	**12292.77**	-2368.04
Warm starts	582654	**0**	-582654
Presolve time	**1.0**	1.02	0.02
Total time	42.89	**37.16**	-5.73

Table 6: Comparison between the default setting and the barrier method with crossover turned off for all but the first LP solved on the optimal cost partitioning of landmarks heuristic. All values are sums of all tasks that were solved by both algorithms, except for total time, which is the average over all tasks that were solved by both algorithms.