Concept Languages as Expert Input for Generalized Planning

Rik de Graaff
University of Basel, Basel, Switzerland

July 6, 2020

Acknowledgements

I would like to thank my supervisors Augusto B. Corrêa and Dr. Florian Pommerening for the countless hours they spent discussing and giving feedback on my thesis, as well as Prof. Dr. Malte Helmert for having built such a supportive environment in his research group. Lastly, I would like to thank my partner, Tatiana Gruosso, for putting up with my wildly inconsistent working schedule during these past few months.

Abstract

Classical planning is an attractive approach to solving problems because of its generality and its relative ease of use. Domain-specific algorithms are appealing because of their performance, but require a lot of resources to be implemented. In this thesis we evaluate concepts languages as a possible input language for expert domain knowledge into a planning system. We also explore mixed integer programming as a way to use this knowledge to improve search efficiency and to help the user find and refine useful domain knowledge.

Contents

1 Introduction

The oft-repeated maxim "physics, not advice" (McDermott, 2000), states that a specification of a planning domain should simply describe the physical properties of the domain and refrain from encoding advice to the planner on how to solve tasks in the domain. This is sensible because the person specifying the domain might have incorrect ideas about the best solution strategies. It is a separation of concerns. The specification of domain and planning are different problems and should not be intertwined. However, the ultimate goal of planning should arguably be to help solve real-world problems. They require close to no domain knowledge to be applied to a domain. All that is required, is that the problem is formulated in a format the planning system can understand. A*(Hart et al., 1968) and greedy best-first search in combination with an informative domain-independent heuristic are considered the state of the art in optimal and suboptimal classical planning respectively. But these approaches to problem-solving mostly can not compete with domain-specific algorithms precisely because of their generality. In this thesis we want to take a step towards finding a golden middle way between classical planning and domain-specific solutions.

To do so we consider adding advice for the planner back in, but as a step separate from the specification of the domain. In doing so, we will be straying very close to generalized planning, where domains of planning tasks with a common underlying structure are automatically analyzed and the results are used to solve individual tasks within that domain much more efficiently. Francès et al. (2019a) presented one such attempt. They introduced an automated way to discover an informative heuristic[1] in the form of a weighted sum of features expressed in a concept language strictly less powerful than first-order logic. They found that these features often represent aspects that have a clear meaning to humans which makes them supremely suited a language for humans to give advice to planning systems. A weakness of this approach is that the expressiveness of the logic as well as how the features can be combined to form a heuristic must be kept at a minimum to prevent a blow-up of the search space of possible heuristics. This resulted in an inability to express a useful heuristic for certain domains.

In this thesis, we will explore how useful concept languages are as a means to express expert knowledge of a domain. We will consider a standard concept language and a few extensions to this concept language, a few different features based on these languages, as well as different ways of combining

[1]To be precise, the heuristics are guaranteed to be descending and dead-end avoiding (Seipp et al., 2016).

these features into the final heuristic and attempt to identify the advantages and downsides of each extension.

We start by formalizing our setting by defining planning tasks, concept languages, how these two can be combined, and what sort of features we want to be able to express. We then describe a workflow that aides a domain expert with specifying informative features and automatically learns a heuristic based on these features. On the practical side, we implement this workflow as a tool and provide a few notes on the nuts and bolts. Finally, we present some experimental results based on our attempts to use the tool ourselves interspersed with fictitious dialogues between two people learning to use the tool as a way to illustrate the capabilities and peculiarities of this workflow.

2 Background

This section will formalize classical planning and the concept language we will be using throughout the paper. We will then clarify how these two elements can be combined. Finally, we will define the heuristics that are ultimately the subject of this thesis.

2.1 Classical Planning

In this thesis, we will consider planning tasks which can be represented as a tuple $\langle \mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$. $\mathcal{P}$ is a set of predicate symbols with an associated arity. $\mathcal{A}$ is a set of action schemas. $\mathcal{C}$ is a set of constants. If $a \in \mathcal{A}$ then $\mathit{par}(a)$ is a set of parameters, $\mathit{pre}(a)$ any logical formula over the predicates in $\mathcal{P}$ and their negations applied to $\mathit{par}(a) \cup \mathcal{C}$, and $\mathit{eff}(a)$ a conjunction of a subset of the same predicates and their negations. $\mathit{cost}(a) \in \mathbb{R}_{\geq 0}$ is called the cost of an action schema. $\mathcal{O}$ is a set of objects. $\mathcal{I}$ is called the initial state and is a full assignment of truth values to all predicates in $\mathcal{P}$ applied to $\mathcal{C} \cup \mathcal{O}$. $\mathcal{G}$ is the goal condition and is a consistent conjunction of a subset of the same predicates and their negations applied to $\mathcal{C} \cup \mathcal{O}$. $\mathcal{A}$ and $\mathcal{O}$ together induce the set of ground actions of the task. This set is obtained by replacing the parameters $\mathit{par}(a)$ in $\mathit{pre}(a)$ and $\mathit{eff}(a)$ of action schemas a by all possible combinations of objects from $\mathcal{O}$. The cost of a ground action is the same as that of its action schema. We say two planning tasks are part of the same domain if they share $\mathcal{P}$, $\mathcal{A}$, and $\mathcal{C}$. In this case we call $\langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$ the domain of the planning tasks.

Many classical planning tasks expressed in PDDL (McDermott et al., 1998) fit our definition quite naturally. We consider predicates and types

declared in a PDDL domain file to be the predicates of the task. Likewise, the action schemas in the domain file are the action schemas in the planning task. Objects defined in the domain file are considered constants, whereas those defined in the task file make up the objects. The initial state and goal as defined in PDDL are also simply the initial state and goal condition of the task. PDDL contains some features which do not map cleanly onto our definition of planning tasks, but these can be compiled away.

A planning task induces a state space $\langle S, L, T, s_I, S_G \rangle$. Every state $s \in S$ represents a unique assignment of truth values to all predicates in $\mathcal{P}$ applied to $\mathcal{C} \cup \mathcal{O}$. We call this assignment (or interpretation) $I(s)$ and write $I(s) \vDash \varphi$ if a formula φ holds under the interpretation $I(s)$. Every label $l \in L$ corresponds to a ground action induced by $\mathcal{A}$ and $\mathcal{O}$. We will call this ground action $act(l)$. $T \subseteq S \times L \times S$ is the set of transitions such that $\langle s, l, s' \rangle \in T$ if and only if $I(s) \vDash \mathit{pre}(act(l))$ and $I(s') \vDash \mathit{eff}(act(l))$ and the set of predicates $I(s)$ and $I(s')$ assign different truth values to is minimal. $s_I \in S$ such that $\mathcal{I} = I(s_I)$ is the initial state. $S_G \subseteq S$ consists of all states s such that $I(s) \vDash \mathcal{G}$.

The successors $\mathit{succ}(s)$ of a state s are all states s' such that $\langle s, l, s' \rangle \in T$ for any $l \in L$. A path between two states s_0 and s_n is a sequence $\langle l_1, \ldots, l_n \rangle$ such that there exists a sequence of states $\langle s_1, \ldots, s_{n-1} \rangle$ with $\langle s_{i-1}, l_i, s_i \rangle \in T$ for all $i \in \{1, \ldots n\}$. The cost of a path $\mathit{cost}(\langle l_1, \ldots, l_n \rangle)$ is $\sum_{i=1}^{n} \mathit{cost}(act(l_i))$. An optimal path between two states is a path with minimal cost. A plan is a path from the initial state to any goal state. An optimal plan is a plan with minimal cost. *Reachable* states are states to which a path from the initial state exists and *solvable* states are states from which a path to any goal state exists. We call a state *alive* if it is both solvable and reachable, but not a goal state.

A heuristic for a planning task is a function $h : S \to \mathbb{R}$. The perfect heuristic h^* is the heuristic which is equal to the cost of an optimal path between the state and any goal state. A generalized heuristic is a function that is defined for all states of all tasks in a given domain.

2.2 Concept Languages

Concept languages or *description logics* are a family of logic-based representation languages, most of which are more expressive than propositional logic but still decidable (Baader et al., 2007). Concept languages deal with *individuals*; classes of individuals which are called *concepts* and can be thought of as properties; and binary relationships between individuals which are called *roles*. Members of the family of concept languages differentiate from one another by which *constructors* for concepts and roles they allow. We will

consider various concept languages which will all be supersets of the standard language $\mathcal{SOI}$ with the added constructor role-value-map.

Syntax. Concepts and roles are defined inductively based on finite sets of named concepts, roles, and individuals. Top $\top$ and bottom $\bot$ are both concepts. Any *named concept* is a concept and any *named role* is a role. If $a_1, \ldots, a_n$ are *named individuals*, C and C' are concepts, and R and R' are roles, then the following holds: the *negation* or *complement* of a concept $\neg C$, the *union* $C \sqcup C'$ and the *intersection* $C \sqcap C'$ of two concepts, the *existential restriction* $\exists R.C$ and the *universal restriction* $\forall R.C$ of a concept by a role, the *nominal concept* $\{a_1, \ldots, a_n\}$, and the *role-value-map* $R = R'$ are all concepts. The *inverse* of a role R^{-1}, the *transitive closure* of a role R^+ and the *composition* of two roles $R \circ R'$ are also roles.

Semantics. The semantics of this description logic are given by a *universe* Δ and a *model* $.^{\mathcal{M}}$. This model maps every individual a to an object $a^{\mathcal{M}} \in \Delta$, every named concept C to a set of objects $C^{\mathcal{M}} \subseteq \Delta$, and every named role R to a relation between objects $R^{\mathcal{M}} \subseteq \Delta^2$. The interpretation of all other concepts and roles is defined recursively as follows. As before, $a_1, \ldots, a_n$ are named individuals, C and C' are concepts, and R and R' are roles.

$$
\begin{aligned}
\top^{\mathcal{M}} = \Delta, \bot^{\mathcal{M}} = \emptyset, (\neg C)^{\mathcal{M}} &= \Delta \backslash C^{\mathcal{M}}, \\
(C \sqcup C')^{\mathcal{M}} = C^{\mathcal{M}} \cup C'^{\mathcal{M}}, (C \sqcap C')^{\mathcal{M}} &= C^{\mathcal{M}} \cap C'^{\mathcal{M}}, \\
(\exists R.C)^{\mathcal{M}} &= \{x \mid \exists y : (x,y) \in R^{\mathcal{M}} \wedge y \in C^{\mathcal{M}}\}, \\
(\forall R.C)^{\mathcal{M}} &= \{x \mid \forall y : (x,y) \in R^{\mathcal{M}} \rightarrow y \in C^{\mathcal{M}}\}, \\
\{a_1, \ldots, a_n\}^{\mathcal{M}} &= \{a_1^{\mathcal{M}}, .., a_n^{\mathcal{M}}\}, \\
(R = R')^{\mathcal{M}} &= \{x \mid (x,y) \in R^{\mathcal{M}} \leftrightarrow (x,y) \in R'^{\mathcal{M}}\}, \\
(R^{-1})^{\mathcal{M}} &= \{(x,y) \mid (y,x) \in R^{\mathcal{M}}\}, \\
R^{+\mathcal{M}} &= \{(x_0, x_n) \mid \exists x_1, \ldots x_{n-1} \\
&\qquad : (x_{i-1}, x_i) \in R^{\mathcal{M}} \text{ for all } i \in \{1, \ldots, n\}\}, \\
(R \circ R')^{\mathcal{M}} &= \{(x,y) \mid \exists z : (x,z) \in R^{\mathcal{M}} \wedge (z,y) \in R'^{\mathcal{M}}\}.
\end{aligned}
$$

Extensions. We also consider the following extensions to this base concept language.

Definition 1 *For all comparison operators* $\sim \in \{=, >, <, \geq, \leq\}$ *we define* qualified cardinality restrictions *introduced by Hollunder and Baader (1991). Let* R *be a role and* C *a concept, the interpretation of the qualified cardinality*

restriction $\sim n\ R.C$ *is*

$$(\sim n\ R.C)^{\mathcal{M}} = \{x \mid \#\{y \mid (x,y) \in R^{\mathcal{M}} \wedge y \in C^{\mathcal{M}}\} \sim n\}.$$

This extension of description logics is commonly referred to as $\mathcal{Q}$.

We further introduce a way to deal with n-ary relations with $n > 2$. To this end we introduce *n-ary roles*. When a role has an arity n which is higher than two, we will make this explicit by indicating this with a subscript, for instance: R_n. We extend $.^{\mathcal{M}}$ such that $R_n^{\mathcal{M}} \subseteq \Delta^n$. In order to make n-ary roles work with the concept language we will define a constructor which makes it possible to reduce the arity of a role. To make this useful, this constructor will be a combination of a selection and a projection.

Definition 2 *Let R_n be an n-ary role, $m \leq n$ a natural number, and C a concept. The interpretation of the* atomic selection *of R_n, ${R_n}^{m \in C}$ is*

$$({R_n}^{m \in C})^{\mathcal{M}} = \{(x_1, \ldots, x_n) \mid x_m \in C^{\mathcal{M}}\}.$$

Definition 3 *Let R_n be an n-ary role, and $m \leq n$ a natural number. The interpretation of the* projection *of R_n, ${R_n}^{\bar{\pi}(m)}$ is*

$$({R_n}^{\bar{\pi}(m)})^{\mathcal{M}} = \{(x_1, \ldots, x_{m-1}, x_{m+1}, \ldots, x_n) \mid \exists x_m.(x_1, ..., x_n) \in R_n^{\mathcal{M}}\}.$$

Using these two definitions, we can now define the *partial restriction* of an n-ary role $R_n[a_1, \ldots, a_n]$, where $a_1, \ldots, a_n$ are either individuals or concepts. For each a_i, if a_i is a concept, then we simply apply the selection $.^{i \in a_i}$ to the role, if it is an individual, we apply both the selection $.^{i \in \{a_i\}}$ as well as the projection $.^{\bar{\pi}(i)}$. For example, if x is an individual and C a concept, the complex role $R_3[\top, x, C]$ would expand to $(({R_3}^{2 \in \{x\}})^{3 \in C})^{\bar{\pi}(2)}$. It should be noted that the projection and selection together form a generalization of the existential restriction, but that the partial restriction does not.

We also introduce the *universal quantifier* and the *existential quantifier* as

$$(\forall a \in C.R_n)^{\mathcal{M}} = \bigcap_{x \in C^{\mathcal{M}}} R_n(a)^{\mathcal{M}(a=x)},$$
$$(\exists a \in C.R_n)^{\mathcal{M}} = \bigcup_{x \in C^{\mathcal{M}}} R_n(a)^{\mathcal{M}(a=x)}.$$

In these definitions, $.^{\mathcal{M}(a=x)}$ denotes the model which is identical to $.^{\mathcal{M}}$ with the addition of the new named individual a such that $a^{\mathcal{M}} = x$. These quantifiers are particularly useful when used in combination with the partial restriction. The introduction of the universal and existential quantifier and

the partial restriction make the concept language much more expressive, at the cost of a lot of nice computational guarantees which make concept languages attractive in the first place. However, we will only be concerned with evaluating concepts and roles under a given interpretation, so this is of no importance.

Finally, we introduce *role intersection* and *role union* which are defined as

$$(R \sqcap R')^{\mathcal{M}} = R^{\mathcal{M}} \cap R'^{\mathcal{M}},$$
$$(R \sqcup R')^{\mathcal{M}} = R^{\mathcal{M}} \cup R'^{\mathcal{M}}.$$

2.3 Concept Languages of Planning Tasks

We consider planning tasks represented in PDDL. We consider objects and constants – objects which are defined in the domain file – to be individuals, unary predicates to be named concepts, binary predicates to be named roles, and n-ary predicates to be n-ary roles is $n > 2$. We consider the universe to be constant across all states and to be identical to the set of individuals. Every state s of the state space induced by the planning task defines a model $.^{\mathcal{M}(s)}$. $a^{\mathcal{M}(s)} = a$ for all named individuals a and states s. For every named concept C, $C^{\mathcal{M}(s)} = \{x \mid I(s) \vDash C(x)\}$, and for every named role R, $R^{\mathcal{M}(s)} = \{(x, y) \mid I(s) \vDash R(x, y)\}$.

We also introduce a *goal constructor* $._G$ which can be applied only to named roles and concepts with the following semantics. If C is a named concept and R is a named role and s is a state in a state space with a set of goal states S_G then

$$(C_G)^{\mathcal{M}(s)} = \bigcap_{s' \in S_G} C^{\mathcal{M}(s')},$$
$$(R_G)^{\mathcal{M}(s)} = \bigcap_{s' \in S_G} R^{\mathcal{M}(s')}.$$

The concept language of a planning domain $\langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$ is the language which has the concepts and roles derived from the predicates in $\mathcal{P}$ as its named concepts and roles as well as the individuals derived from the constants in $\mathcal{C}$ as its named individuals. We also introduce a named concept for each nullary predicate that have the interpretation Δ in states where the predicates hold and $\emptyset$ in states where they do not.

2.4 Generalized Potential Heuristics

Francès et al. (2019a) introduced *generalized potential heuristics* as a weighted

sum over features which map states to integers.

Definition 4 *Let S be a set of states which are not necessarily part of the same state space and $\mathcal{F}$ a set of features $f : S \to \mathbb{Z}$. Let $w : \mathcal{F} \to \mathbb{R}$ be a weight function mapping features to weights. The value of the potential heuristic with features $\mathcal{F}$ and weights w on a state $s \in S$ is*

$$h(s) = \sum_{f \in \mathcal{F}} w(f) \cdot f(s).$$

In this thesis we will consider generalized potential heuristics which are well-defined on all states s that are part of any state space induced by any planning task belonging to the same domain. This requires that all the features must be well-defined on the same states. To this end, we will express features in terms of concepts and roles in the concept language of the domain. We call these features the features of this planning domain. As a starting point, we use the same features Bonet et al. (2016) introduced.

Definition 5 *Let C and C' be concepts and R a role in the concept language of a planning domain. The value of the* cardinality feature $|C|$ *in the state s is $|C^{\mathcal{M}(s)}|$. The value of the* minimal distance feature $dist(C, R, C')$ *in s is $\min n$ such that $\exists x_0, \dots x_n$ with $x_0 \in C^{\mathcal{M}(s)}, x_n \in C'^{\mathcal{M}(s)}$ and $(x_{i-1}, x_i) \in R^{\mathcal{M}(s)}$ for all $i \in \{1, \dots, n\}$. If such a sequence of individuals $x_0, \dots x_n$ does not exist, we define the value to be 0.*

Extensions. We will also consider the following extensions to the set of possible features.

Definition 6 *Let f_1 and f_2 be features of a planning domain. The value of the* product feature $f_1 \cdot f_2$ *in the state s is $f_1(s) \cdot f_2(s)$.*

Definition 7 *Let h be a generalized heuristic of the domain $\mathcal{D}$. The value of the* heuristic feature f_h *in the state s in the domain $\mathcal{D}$ is $h(s)$.*

By allowing any generalized heuristic of the domain as a feature, we allow a set of features to contain both human intuition about a domain and information state of the art heuristics are good at detecting.

Complexity. If X is a feature, role or concept consider its *complexity* $\mathcal{K}(X)$ to be the size of its syntax tree. For example, $\mathcal{K}(dist(C, R, C')) = 1 + \mathcal{K}(C) + \mathcal{K}(R) + \mathcal{K}(C')$. The complexity of $\top$ and $\bot$ is 0, the complexity of all heuristic features, named concepts, named roles, and nominal concepts is 1. This definition differs slightly from the one introduced by Francès et al. (2019a). They defined the complexity of a feature simply as the sum of the complexities of all the involved concepts and roles. We deviate from their definition because we introduced features which are defined inductively.

3 Learning Heuristics

To quantify the value of extensions to the concept language and feature space bring, we will learn several generalized potential heuristics with the features that we expressed with them, and analyze their performance. Each of the approaches takes as inputs a set of alive states from a common domain $\mathcal{S}$ and a set of hand-crafted *candidate features* $\mathcal{F}$ of the same domain. We will find heuristics $h : \mathcal{S} \to \mathbb{R}$ by solving some mixed integer program with variables $W = \{w_f \in \mathbb{R} \mid f \in \mathcal{F}\}$ and consider $h(s)$ to equal $\sum_{f \in \mathcal{F}} w_f \cdot f(s)$. The expression $f(s)$ here represents the value of the feature f in the state s. For our first approach, we follow Francès et al. (2019a) and compute the simplest heuristic which is descending and dead-end avoiding on $\mathcal{S}$. We say a heuristic h is descending on a state s if s has at least one successor with a heuristic value that is less than $h(s) - 1$. We say a heuristic h is dead-end avoiding on a state s if every unsolvable successor of s has a heuristic value greater than or equal to $h(s)$. A descending and dead-end avoiding heuristic on a set of states $\mathcal{S}$ is a heuristic which is descending and dead-end avoiding on every solvable, non-goal state in $\mathcal{S}$. A heuristic with this property guides most standard greedy algorithms directly to a goal. The mixed integer program for this heuristic is a program $\mathcal{M}_{strict}(\mathcal{S}, \mathcal{F})$ with $w_f \in \mathbb{R}$ they defined as

$$\begin{aligned}
&\min_{W} \sum_{f \in \mathcal{F}} [w_f \neq 0] \mathcal{K}(f) && \text{subject to} \\
&\bigvee_{s' \in succ(s)} h(s) \geq h(s') + 1 && \text{for all } s \in \mathcal{S} \\
&\bigwedge_{\substack{s' \in succ(s), \\ s' \text{ unsolvable}}} h(s') \geq h(s) && \text{for all } s \in \mathcal{S}.
\end{aligned}$$

It is worth noting that neither this mixed integer program nor any of the other two are linear. However, all three can be solved by modern MIP solvers after some rewriting as described by Francès et al. (2019b). In broad terms, it requires introducing indicator constraints for disjunctions and two binary variables for the indicator function in the objective.

Since we do not expect to find a descending and dead-end avoiding heuristic for more complex domains like Sokoban, we also introduce a version of the above linear program with slack variables. This mixed integer program $\mathcal{M}_{slack}(\mathcal{S}, \mathcal{F})$ with variables $w_f \in \mathbb{R}$ and additional slack variables

$u_s, v_{s,s'} \in \mathbb{R}$ is defined as

$$\begin{aligned}
&\min_{W,u,v} \sum u_{s,s'} + \sum v_{s,s'} && \text{subject to}\\
&\quad \bigvee_{s' \in succ(s)} h(s) + u_s \geq h(s') + 1 && \text{for all } s \in \mathcal{S}\\
&\quad \bigwedge_{\substack{s' \in succ(s),\\ s' \text{ unsolvable}}} h(s') + v_{s,s'} \geq h(s) && \text{for all } s \in \mathcal{S}\\
&\quad u_s \geq 0 \quad \text{for all } s \in \mathcal{S}\\
&\quad v_{s,s'} \geq 0 \quad \text{for all } s \in \mathcal{S}, s' \in succ(s), s' \text{ unsolvable.}
\end{aligned}$$

Note here that we no longer take the complexity of a feature into account and simply compute the generalized potential heuristic which is closest to being descending and dead-end avoiding using all the features provided. The reasoning behind this is that, since the features are hand-crafted by a domain expert, we expect most of them to be useful. Provided the set of states is sufficiently large and varied, a feature f which is not informative will essentially act as noise and will not be useful to minimize the slack variables. We can, therefore, expect w_f to be close to 0 and the domain expert can choose to discard it at their discretion to make the heuristic more computationally efficient and, hopefully, generalize better.

The final heuristic aims to locally approximate the perfect heuristic h^*. We obtain this heuristic by solving the mixed integer program $\mathcal{M}^*(\mathcal{S}, \mathcal{F})$ with $w_f \in \mathbb{R}$ and additional slack variables $u_{s,s'}, v_{s,s'} \in \mathbb{R}$

$$\begin{aligned}
&\min_{W,u,v} \sum |u_{s,s'}| + \sum v_{s,s'} && \text{subject to}\\
&\quad \bigwedge_{\substack{s' \in succ(s),\\ s' \text{ solvable}}} h(s) - h(s') + u_{s,s'} = h^*(s) - h^*(s') && \text{for all } s \in \mathcal{S}\\
&\quad \bigwedge_{\substack{s' \in succ(s),\\ s' \text{ unsolvable}}} h(s') + v_{s,s'} \geq \max_{\substack{t \in succ(s) \cup \{s\}\\ t \text{ solvable}}} h(t) && \text{for all } s \in \mathcal{S}\\
&\quad v_{s,s'} \geq 0 \quad \text{for all } s \in \mathcal{S}, s' \in succ(s), s' \text{ unsolvable.}
\end{aligned}$$

The first type of constraint encapsulates how well the heuristic approximates the perfect heuristic locally. A heuristic h which fulfills these constraints with all $u_{s,s'} = 0$ can only differ the perfect heuristic by a constant on all states in $\mathcal{S}$ and their solvable successors. The second type of constraint deals with unsolvable successors of solvable states. Since the value of the features, and by extension of the heuristic, can only be a finite number, the heuristic

can not indicate an unsolvable state by having an infinite value. Instead, we strengthen the requirement from $\mathcal{M}_{slack}(\mathcal{S}, \mathcal{F})$ and compare the value of the heuristic for the unsolvable state to that for its predecessor and all of its solvable successors. The same considerations about generalization and computational efficiency we made for $\mathcal{M}_{slack}(\mathcal{S}, \mathcal{F})$ apply here too.

The mixed integer program $\mathcal{M}^*(\mathcal{S}, \mathcal{F})$ with $w_f \in \mathbb{R}$ can be equivalently expressed as the purely linear program

$$
\begin{aligned}
\min_{W,u,v} \sum_{\substack{s' \in succ(s),\\ s' \text{ solvable}}} u_{s,s'} + \sum v_{s,s'} \qquad & \text{subject to} \\
\bigwedge_{\substack{s' \in succ(s),\\ s' \text{ solvable}}} u_{s,s'} \geq h(s) - h^*(s) - h(s') + h^*(s') \qquad & \text{for all } s \in \mathcal{S} \\
\bigwedge_{\substack{s' \in succ(s),\\ s' \text{ solvable}}} u_{s,s'} \geq h(s') - h^*(s') - h(s) + h^*(s) \qquad & \text{for all } s \in \mathcal{S} \\
\bigwedge_{\substack{s' \in succ(s),\\ s' \text{ unsolvable}}} \bigwedge_{\substack{t \in succ(s) \cup \{s\}\\ t \text{ solvable}}} v_{s,s'} \geq h(t) - h(s') \qquad & \text{for all } s \in \mathcal{S} \\
u_{s,s'} \geq 0 \quad \text{for all } s \in \mathcal{S}, s' \in succ(s), s' \text{ solvable} & \\
v_{s,s'} \geq 0 \quad \text{for all } s \in \mathcal{S}, s' \in succ(s), s' \text{ unsolvable.} &
\end{aligned}
$$

3.1 Reporting Difficult States

When solving $\mathcal{M}_{slack}(\mathcal{S}, \mathcal{F})$ and $\mathcal{M}^*(\mathcal{S}, \mathcal{F})$, we can identify states which the heuristic captures badly by considering the slack variables with the highest value. We can automatically present the domain expert with these states and their successors as well as the feature values and total heuristic value for these states. These can serve as a basis for the expert to identify which important aspects of the domain are not adequately expressed by the features they specified. They can then extend $\mathcal{F}$ with features they expect might alleviate the problem.

What results from these basic elements is an iterative process. The domain expert starts out by expressing some features $\mathcal{F}$ of the domain they expect are significant. They can then solve one or both of the mixed integer programs $\mathcal{M}_{slack}(\mathcal{S}, \mathcal{F})$ and $\mathcal{M}^*(\mathcal{S}, \mathcal{F})$ with $\mathcal{S}$ being the alive states of a few small tasks of the domain. After refining $\mathcal{F}$ as described above, the expert can solve the mixed integer programs again with the new set of features. When the resulting heuristic captures everything in the sample states $\mathcal{S}$ reasonably well, the expert can add additional tasks and thereby expand

$\mathcal{S}$. These steps can be repeated until the expert is satisfied with the resulting heuristic or improving upon it becomes infeasible due to the size of the mixed integer program.

There are two logical improvements to this process which we do not explore in this thesis. The first is the ability to generate constraints incrementally, sampling states from $\mathcal{S}$ randomly and adding more as needed. However, we do give the domain expert the ability to manually specify a sample rate that determines what portion of states from state spaces are added into $\mathcal{S}$. The second is the capacity to improve the heuristic using data from states seen during searches with the heuristic. Francès et al. (2019a) introduced approaches to incorporate these improvements into the calculation of a descending and dead-end avoiding heuristic in their setting where the feature pool is automatically generated and can grow unboundedly. These approaches do not translate easily to a setting where the objective is to minimize slack variables and features are handcrafted. We, therefore, consider these possible improvements to be the subject of future work.

4 Implementation

We implemented everything needed to formulate, evaluate, and refine these generalized potential heuristics as described in the previous section using Python and C++. The concepts were integrated into the Fast Downward Planning System (Helmert, 2006). Features based on concepts can be formulated in a manchester-like notation[2], and be parsed and evaluated on states by Fast Downward. The mixed integer programs and the general workflow were implemented in a Python script using the CPLEX Python API [3]. We used version 12.10 CPLEX. The source code for this tool is available online (de Graaff, 2020).

4.1 Notes on Implementation

There are a few things to keep in mind when implementing this process. Firstly, the resulting heuristic is not guaranteed to always be positive. Using such a heuristic, therefore, requires either a planner and search algorithm which can handle negative heuristic values, or some lower bound on the heuristic value to be added to the heuristic. The heuristic might not even be a whole number. If the planner can only handle whole numbers the heuristic might need to be rounded. To prevent this rounding from causing unforeseen

[2]https://www.w3.org/TR/owl2-manchester-syntax/
[3]https://pypi.org/project/cplex/

trouble it is advisable to multiply the weights of such a heuristic with some number to make all weights whole numbers or at least large enough that it is unlikely to make two meaningfully different heuristic values round to the same number.

Secondly, the inclusion of heuristic features opens the possibility of infinite feature values. This must either be handled explicitly or modeled by a sufficiently large positive number to work with the learning step. When a safe heuristic is infinite on a state, the learned heuristic can also be infinite on that heuristic, regardless of what weight is assigned to it. By default,the Fast Downward planner only supports positive integers as heuristic values and represents an infinite heuristic value has -1. We opted to allow make changes that allow negative integers as well and represent infinity as a large integer.

Finally, static information of a task, such as the types of objects in PDDL and the value of variables that have only one reachable value which might be filtered by a preprocessing step must be retained if it is to be used in the concept-based features. Ideally, this should be stored separately, so that the interpretation of concepts, roles, and features which are comprised entirely of static named roles and concepts can be cached. Using the same mechanism, we can also cache the interpretation of concepts, roles, and features that are used multiple times. These two caching strategies can make for a tremendous speed-up, ranging from an order of magnitude to linear in the number of evaluated states and polynomial in the number of objects depending on the task and features used. The Fast Downward planner also does not retain static information when translating from PDDL to SAS by default (Helmert, 2009) and we opted to extend the translator to output the static information as a separate file. We also implemented both caching strategies by tagging expressions which are made up of only static components for state space-level caching and tagging expressions which are used more than once in at least two different parent expression for state-level caching.

5 Use Cases

In this section, we will illustrate how this process can work in praxis by presenting fictitious back and forths between Amelia – a domain expert – and Fred – who has prior knowledge of the system. After each dialogue we will include a more formal discussion of the use of the tool for that domain. This discussion will include some results from experiments to support some of the sentiments raised in the dialogue. The main measure these experiments will look at is the total evaluation score of the eager greedy search using the

found heuristics. The evaluation score depends on the number of evaluations a search algorithm needs to solve a task. It is 1 when the task required no evaluations and exponentially approaches 0 as the number of evaluations tends to infinity. If an algorithm fails to solve a task it also gets a score of 0. This ensures that the results for the largest tasks don't dominate the other results and allows a fair comparison when not all algorithms were able to solve all tasks. All heuristics were found on an Intel i5-7400 CPU with a time limit of half an hour for CPLEX. If CPLEX exceeded the time limit, one of the current best solutions was used, even if it was not provably optimal. All experiments were run on Intel Xeon E5-2660 CPUs with a time limit of 5 minutes.

5.1 VisitAll

The setting of the domain VisitAll is an undirected graph. On this graph, there is a single agent, which must visit every node in the graph.

FRED: Hopefully you've got a sense of how this system could help us exploring general solutions for problem domains from that explanation.

AMELIA: I believe so.

FRED: Excellent. Just to get our feet wet, the first domain we'll be taking a look at is VisitAll.

AMELIA: I'm familiar.

FRED: I'm told you're an expert. What is the first thing that comes to mind as a feature that might be helpful to the system?

AMELIA: Well, since the goal is to visit all locations, the number of unvisited locations is an obvious starting point.

FRED: That makes sense. We can express that as $|\neg visited|$.

AMELIA: Did you just speak in mathematical symbols?

FRED: I did; it's a skill I picked up a few years ago.

AMELIA: I bet that comes in handy in your profession. Another feature I think we could... [FRED *interrupts* AMELIA.]

FRED: I think we should start simple and see what the system does with it. I've written up a file with your feature and we can try it out on the smallest graph with just four locations.
[FRED *shows* AMELIA *the file visitall.cpt on his screen*]

visitall.cpt

```
not_visited = not visited.
return {#not_visited}.
```

[*and then types the following into his console.*]

```
./general_optimizer.py
    --domain visitall-opt11-strips
    --tasks problem02-full
    --samples-path ./samples
    --evaluators "concepts(./concepts/visitall.cpt)"
    --methods slack
    --steps sample evaluate optimize
```

FRED: This shouldn't take too long. [*A few seconds later, the program has finished running and the bottom of the console displays the following.*]

```
state in task problem02-full
slack: 1
values:
at_robot   visited
loc_x1_y1  loc_x0_y1
           loc_x1_y0
           loc_x1_y1
features: [1]
h: 25
h*: 2
successors
operator: move loc-x1-y1 loc-x1-y0
values:
at_robot   visited
loc_x1_y0  loc_x0_y1
           loc_x1_y0
           loc_x1_y1
features: [1]
h: 25
h* 1
operator: move loc-x1-y1 loc-x0-y1
values:
at_robot   visited
loc_x0_y1  loc_x0_y1
           loc_x1_y0
           loc_x1_y1
features: [1]
h: 25
h* 1

feature weights are:
0: (# not_visited)
weight: 25
```

FRED: As expected, the heuristic isn't perfect yet. The program is telling us that when the robot is in the bottom right corner and has visited the top right and bottom corner, neither moving up-

wards nor moving to the left decreases the heuristic value. Can you tell me the simplest feature that you think might help with this problem?

AMELIA: I expected as much. You mentioned a distance feature; can you express the distance between the robot and the nearest unvisited location?

FRED: Of course, that would be $dist(at\text{-}robot, connected, \neg visited)$. Let me try adding that. [FRED *edits the visitall.cpt file to read*]

visitall.cpt

```
not_visited = not visited.
distance_to_next = dist(at_robot, connected, not_visited).
return {#not_visited, distance_to_next}.
```

[*and then runs the program again.*]

FRED: Now it found a descending heuristic for this task by assigned a weight of 25 to the amount of unvisited locations and a weight of one to the distance.

AMELIA: Why 25?

FRED: That's somewhat arbitrarily the maximum weight. These mixed integer program solvers often prefer extreme values when the exact value doesn't affect their objective value. [FRED *types away at his keyboard and looks at his screen for a few seconds.*] This seems to work for bigger instances as well.

AMELIA: This maximum weight of 25 will become a problem at some point.

FRED: How so?

AMELIA: When the graph becomes too large, the distance to the next unvisited location might become larger than 25. May I produce a revenge task for this heuristic?

FRED: Be my guest. [FRED *turns his laptop towards* AMELIA*, who starts defining a task with 27 locations in a row and the robot starting on the second location.*]

AMELIA: Try running the program on this. [FRED *runs the program again.*]

FRED: I see. When the robot visits the location at one end of the graph and only the location on the other end is unvisited, the distance becomes so large that visiting a new location doesn't make up for it.

AMELIA: Exactly, and all you need to do to fix this problem is multiply the number of unvisited locations with the diameter of the graph.

FRED: Will simply the number of locations do as well? I don't think we can express the diameter of a graph.

AMELIA: I don't see why not. [FRED *edits the visitall.cpt one more time to*]

visitall.cpt

```
not_visited = not visited.
distance_to_next = dist(at_robot, connected, not_visited).
return {#T * #not_visited, distance_to_next}.
```

[*and runs the program once more.*]

FRED: That does seem to have fixed the issue. Are there any more hidden problems like this?

AMELIA: This heuristic now perfectly captures the strategy of always moving towards the closest unvisited node. I don't think there are any more problems.

FRED: If there are, they would hopefully become evident while using the heuristic. This heuristic should always lead directly to a goal, so it should be obvious when it doesn't.

AMELIA: That's an awfully imprecise guarantee.

FRED: It is; I've heard they will work on making it more concrete in a future update.

Discussion. In this first foray into using this tool, Amelia and Fred discovered a simple descending heuristic for the domain VisitAll. This dialogue exemplifies what sorts of considerations one must make when using this tool and shows what sort of help the tool provides. The VisitAll domain provides a very strong argument for the inclusion of the multiplication feature since a descending heuristic can not be expressed without it.

We compared the performance of the heuristic described in the dialogue with the FF heuristic. The results can be found in Figure 1 and show that the heuristic found dominates the FF heuristic entirely.

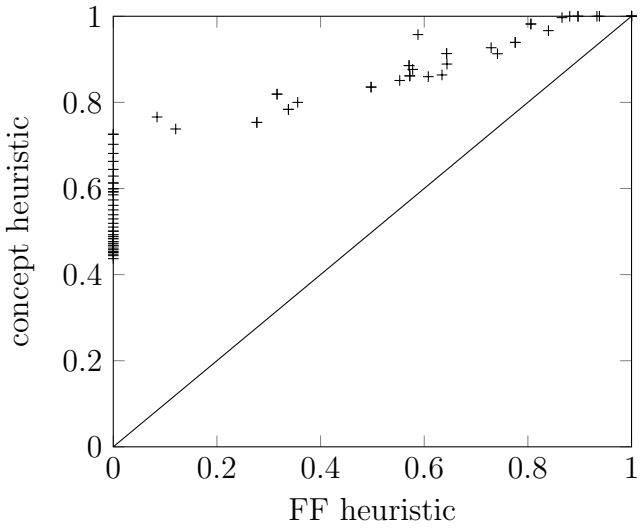


Figure 1: Evaluation scores of eager greedy search with concept heuristic and the FF heuristic for VisitAll tasks.

5.2 Logistics

The Logistics domain deals with packages which must be delivered from one location to another. There are trucks which can transport packages within cities and airplanes which can transport packages from one city to another but only between airports. Loading a package into or out of a vehicle has a cost of one, as do flying an airplane and driving a truck.

Amelia: Can I have a few minutes of your time?
Fred: Absolutely. What's on your mind?
Amelia: I have been working on the Logistics domain with this planning tool and I am running in place somewhat.
Fred: What exactly is the problem?
Amelia: Well, I tried simply using the FF heuristic (Hoffmann, 2001) as a starting point to see what the system would do with it. I optimized it for two small tasks and one of the states on which the heuristic does not descend had a successor which progressed towards the goal where the only difference was that a truck drove towards a package that needed to be transported. To capture the

difference between those two states I added

$$\begin{aligned} \mathit{real\_at} &:= \mathit{at} \sqcup (\mathit{in} \circ \mathit{at}) \\ \mathit{right\_loc} &:= \mathit{package} \sqcap \neg(\mathit{real\_at} = \mathit{at}_G) \\ \mathit{wrong\_loc} &:= \mathit{package} \sqcap \neg \mathit{right\_loc} \\ \mathit{at\_truck} &:= \exists \mathit{at}.\exists \mathit{at}^{-1}.\mathit{truck} \\ \mathit{at\_airplane} &:= \exists \mathit{at}.\exists \mathit{at}^{-1}.\mathit{airplane} \\ \mathit{at\_vehicle} &:= \mathit{at\_truck} \sqcup \mathit{at\_airplane} \\ \mathit{waiting\_packages} &:= (\exists \mathit{at}.\mathit{location}) \sqcap \mathit{wrong\_loc} \sqcap \neg \mathit{at\_vehicle} \end{aligned}$$

as concepts and $|\mathit{waiting\_packages}|$ as a feature.

FRED: Did that improve the heuristic?

AMELIA: Yes, the sum of the slack variables went down from 492 to 126 on the two small tasks I was using. I then added

$$\begin{aligned} \mathit{right\_city} &:= \mathit{package} \sqcap (\mathit{real\_at} \circ \mathit{in\text{-}city} = \mathit{at}_G \circ \mathit{in\text{-}city}) \\ \mathit{wrong\_city} &:= \mathit{package} \sqcap \neg \mathit{right\_city} \\ \mathit{in\_truck} &:= \exists \mathit{in}.\mathit{truck} \\ \mathit{loaded} &:= \mathit{wrong\_loc} \sqcap \mathit{in\_truck} \\ \mathit{wrong\_city\_at\_airplane} &:= \mathit{wrong\_city} \sqcap \mathit{at\_airplane} \end{aligned}$$

as concepts and $|\mathit{loaded}|$ and $|\mathit{wrong\_city\_at\_airplane}|$ as features as a direct response to some of the difficult states the system reported. That took the slack variables down to 8. I could see plainly from the reported states, that all I needed to add was

$$\mathit{right\_city\_truck\_ready} := \mathit{right\_city} \sqcap \mathit{wrong\_loc} \sqcap \mathit{at\_truck}$$

to obtain a descending heuristic on these two small states.

FRED: And did you?

AMELIA: Yes. And the system even found a descending heuristic on a slightly larger task as well. Namely,

$$\begin{aligned} &25 \cdot f_{FF} + |\mathit{waiting\_packages}| + 2 \cdot |\mathit{loaded}| \\ &-24 \cdot |\mathit{wrong\_city\_at\_airplane}| + 2 \cdot |\mathit{right\_city\_truck\_ready}|. \end{aligned}$$

FRED: That all sounds very promising so far.

AMELIA: It definitely looked that way. I hoped that these four features combined with the FF heuristic would be enough to make for a descending heuristic on the entire domain.

FRED: But they weren't?

AMELIA: Unfortunately, it didn't result in a direct path towards the goal for a larger task. As a consequence, I saw myself forced to use more tasks, which made the optimization process a lot slower.

FRED: I see how that would get bothersome quickly. Did you know you can specify a sample rate to significantly reduce the amount of states used for the optimization?

AMELIA: I found out about that out of necessity. I think I might still have been stuck at that stage if I hadn't. I had to scale up the problems by quite a bit, but I eventually found a state where loading a package into an airplane resulted in progress towards the goal but did not decrease the heuristic value. So then I added

$$\begin{aligned} \mathit{in\_airplane} &:= \exists \mathit{in}.\mathit{airplane} \\ \mathit{boarded} &:= \mathit{wrong\_city} \sqcap \mathit{in\_airplane} \end{aligned}$$

as concepts and $|\mathit{boarded}|$ to the feature pool.

FRED: I take it that didn't result in a descending heuristic either?

AMELIA: It didn't. In fact, the amount of expanded states increased for the largest task in the suite. And that's where I'm stuck right now. I have been including more and bigger tasks into the optimization process but am failing to find an example of it failing to be descending. I finally decided that perhaps, the FF heuristic and the features are not actually complementing each other nicely and started an optimization with just the features I have found, but that has been running for a few hours now.

FRED: What sort of output are you getting from the tool?

AMELIA: The output suggests that the slack will be somewhere between 1 and 5, which makes it doubly frustrating that it also suggest the process is running out of memory. If it finished running, I'm sure I would get some useful states out of it.

FRED: It doesn't actually need to finish running.

AMELIA: Come again?

FRED: You can interrupt the MIP solver and it will still report difficult states depending on the best solution it found so far. [AMELIA *presses Ctrl + C on her keyboard and her screen displays some states with a non-zero slack variable.*]

AMELIA: That certainly would have been useful knowledge a few hours ago.

FRED: Well, you know it now. You can also set a time limit for the MIP solver if you are not around to interrupt it manually. Is the output at all useful to you?

AMELIA: Absolutely. [AMELIA *points at her screen.*] From this state I can see that I'll need to disambiguate whether a package is waiting for a truck in the wrong city or in the right city.

FRED: Wow, you've gotten very quick at interpreting the output.

AMELIA: I hope so, but I also expected something along these lines. While the program was running, I contemplated the features I had so far and figured out and wrote down what I think is a descending heuristic based on them. [AMELIA *pulls up a text file.*]

Logistics.cpt

```
real_at = at or (in:at).

right_city = package and (real_at:in_city == at_g:in_city).
wrong_city = package and not right_city.
right_location = package and (real_at == at_g).
wrong_location = package and not right_location.

at_airport = real_at some airport.

at_truck = at some at- some truck.
at_airplane = at some at- some airplane.

in_truck = in some truck.
in_airplane = in some airplane.
in_vehicle = in_truck or in_airplane.

wrong_city_waiting = wrong_city and not at_airport
    and not at_truck and not in_truck.
wrong_city_truck_ready = wrong_city and not at_airport
    and at_truck.
wrong_city_loaded = wrong_city and not at_airport
    and in_truck.
in_truck_at_airport = wrong_city and at_airport
    and in_truck.
at_right_airport = wrong_city and at_airport
    and not at_airplane and not in_vehicle.
at_right_airplane = wrong_city and at_airplane.
boarded = wrong_city and in_airplane.
landed = right_city and in_airplane.
right_city_waiting = right_city and wrong_location
    and not at_truck and not in_vehicle.
right_city_truck_ready = right_city and wrong_location
    and at_truck.
```

```
    right_city_loaded = right_city and wrong_location
        and in_truck.
    arrived = right_location and in_truck.
    delivered = right_location and not in_vehicle.

    return {
        #wrong_city_waiting,
        #wrong_city_truck_ready,
        #wrong_city_loaded,
        #in_truck_at_airport,
        #at_right_airport,
        #at_right_airplane,
        #boarded,
        #landed,
        #right_city_waiting,
        #right_city_truck_ready,
        #right_city_loaded,
        #arrived,
        #delivered
    }.
```

Amelia: It consists of a number of concepts that correspond to mutually exclusive states a package can be in. A package moving down the list of features is always progress towards the goal, so assigning them weights 0 through 12 should result in a descending heuristic.

Fred: Wait, if you figured this out already, why were you still running the program?

Amelia: For one thing, I was hoping it might help me discover a different solution. Mostly, I wanted to see if I could have stumbled upon this solution if I hadn't thought it up by simply following the procedure.

Fred: I see. It's encouraging to know that it works, especially now that you know you can interrupt the program and still get usable results.

Amelia: Yes, I feel confident applying this approach to a more challenging problem after this experience. I am a bit worried that I didn't see any continuous improvements when trying out the heuristics I found along the way on larger instances though.

Fred: I wouldn't be so quick to jump to that conclusion. We should probably run some experiments to see if adding more features improves performance or not.

Method	Number of Features 1	2	3	4	5	6	7	8	9	10	11	12
blind	3.38											
slack	3.32	3.61	4.08	2.00	2.54	5.59	2.06	8.48	11.01	13.50	3.66	22.97
h^*	3.32	3.61	4.08	4.45	4.45	5.60	4.42	7.65	7.65	10.40	9.66	15.01
FF	23.09											
FF-slack	23.09	23.10	23.19	23.17	23.45	23.40	23.28	23.18	23.23	22.84	23.12	23.23
FF-h^*	23.09	23.09	23.09	23.09	23.09	23.09	22.49	22.49	22.49	22.49	22.49	22.49

Table 1: Evaluation scores of Logistics heuristics.

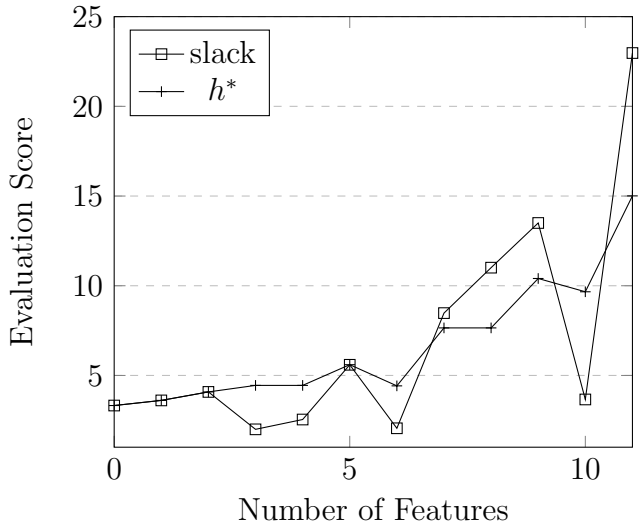


Figure 2: Evaluation scores of Logistics heuristics.

Discussion. When working on the Logistics domain, Amelia and Fred independently discover the same descending heuristic for the Logistics domain Francès et al. (2019a) discovered. It should of course be stressed that this is a fictitious conversation and we were aware of the existence of this heuristic when we used this tool on the Logistics domain. That being said, we believe the tool significantly lowers the time and planning knowledge required. For a domain like Logistics the output of the tool genuinely provides actionable outputs along every step of the way as it is depicted in the dialogue.

We performed the experiment Fred alludes to at the end of the conversation. We learned four different heuristics based on 12 different sets of features. We started with just $|\mathit{wrong\_city\_waiting}|$ and added the other features excluding $|\mathit{delivered}|$ one by one in the order they are listed in the Logistics.cpt. The heuristics were trained on all states of three tasks with 2 packages, a random sample of 10% of all tasks of two states with 3 packages, a random

sample of 5% of all states of two tasks with 4 packages, and a random sample of 2% of all states of one task with 5 packages. For each set of features we learned the heuristic using both $\mathcal{M}_{slack}$ and $\mathcal{M}^*$, once using just the features and once including the heuristic feature h_{FF}.

The results can be found in Table 1, where the former are named slack and h^* and the latter FF-slack and FF-h^*. The heuristics which aimed to add to the FF heuristic did not differ meaningfully from it in terms of state evaluations. The evaluation scores for the heuristics which did not use the FF heuristic as a feature can also be seen in Figure 2. Both approaches show a clear positive trend with more features added, but the heuristics found by $\mathcal{M}_{slack}$ appear to be much more volatile than those found by $\mathcal{M}^*$.

These results suggest that there is value in both variants of the learning step. Another advantage of $\mathcal{M}^*$ is that it is much quicker since it is a strictly linear program, whereas $\mathcal{M}_{slack}$ requires indicator constraints. Solving $\mathcal{M}_{slack}$ optimally for the full feature set and a task with 3 packages and 2,744 states takes 37.72 seconds, solving $\mathcal{M}^*$ instead takes only 3.78 seconds. When using sampling to reduce the state space to approximately 20% reduces the times to 0.25 seconds and 0.05 seconds respectively. Despite this drastic speed-up, $\mathcal{M}_{slack}$ still found a heuristic which provably descends on the entire domain when trained on the sample of 20% of the states.

5.3 TERMES

The TERMES domain models a small robot that must build structures much larger than itself out of uniform blocks. The blocks can only be placed on the floor or be stacked directly on top of each other, and no overhangs are possible. This means that the state of the construction area can be fully represented by specifying the number of blocks at each space. The robot can move from one space to a neighboring space as long as the height difference is no larger than one block. The robot can carry at most one block at a time. When the robot is carrying a block, it can place it down at a neighboring space that has the same height as the space the robot is on. Likewise, the robot can pick up a block from a neighboring space if the height of that space is one more than that of its current location and the robot is not currently carrying a block. A specific location functions as a depot, where the robot can destroy a block it is carrying and create a block and start carrying it if it is currently not carrying a block.

FRED: TERMES is quite a bit harder than the domains we have looked at so far, so I'm interested to see what we can achieve with this tool.

AMELIA: It really is.

FRED: To help us out a bit, I have written a custom state renderer which we can pass to the tool. It turns the output from this into this [FRED *shows* AMELIA *his screen.*]

```
at        height                  2  2
pos_0_1   (pos_0_0, n2)           0r 1
          (pos_0_1, n0)
          (pos_1_0, n2)
          (pos_1_1, n1)
```

AMELIA: That is definitely a lot easier to read. As for features, I think we should probably start with something simple like a feature which counts towers which are the correct height. Something like

$$\mathit{correct\_height} := \mathit{position} \sqcap \mathit{height} = \mathit{height}_G.$$

FRED: Good idea, that should already improve things a bit. It's possible it will be somewhat counterproductive when towers need to be built to serve as ramps for other towers, but it should be usable. [FRED *runs the optimizer.*] That's unexpected, the optimizer assigned a weight of 1 to the feature.

AMELIA: That is weird. Making a tower the right height generally is progress, so it should really have a negative weight. I think this is the worst possible heuristic it could have found using this feature.

FRED: Perhaps there is a bug in the tool?

AMELIA: Could be. Is there a way to restrict the weights?

FRED: I could change the bounds in the source code, that shouldn't be too hard.

AMELIA: Could you restrict the weights to negative numbers and run it again?

FRED: Of course. [FRED *edits a file and runs the tool again. It takes less long this time.*] Now it found a solution with a negative weight and the sum of the slack values is exactly the same. [FRED *runs a search algorithm.*] But the negative weight makes for pretty good performance and the heuristic with a positive weight is as terrible as one would expect.

AMELIA: That is strange.

FRED: In a way it makes sense since the tool is just looking for a heuristic that descends on as many states as possible. A heuristic that descends in exactly the wrong direction is a pretty good solution.

AMELIA: That doesn't make much sense to me. If it doesn't matter in which direction it descends, how can we expect it to find a good heuristic at all?

FRED: The goals don't have to have a neighbor with a lower heuristic value, so they can act as sinks.

AMELIA: That explains why this isn't working then. The problem we're learning from requires the robot to build a single tower with a height of 2; the last move it makes to reach a goal must always be destroying a block at a depot. Since the feature we defined can't distinguish between states where the robot is holding a block and those where it isn't, the feature can't distinguish between any goal state and its predecessors.

FRED: That makes a lot of sense. The goals don't really act as sinks, so the direction in which the heuristic descends is irrelevant.

AMELIA: I wonder if behavior similar to this shows up often and it's usually difficult to identify the problem simply because it is obfuscated. That could potentially be disastrous in a domain like this where the effects of an action are highly context-dependent.

FRED: If that's true, it would explain the results of our experiments of the Logistics domain somewhat. Be that as it may, how about we try to alleviate this problem by adding a new feature?

AMELIA: We don't have to solve it by adding a feature, we could also add a task where the last action can be placing a block. Any task which requires a tower with height one to be built should work.

FRED: That's quite ingenious, how about we try both?

AMELIA: Good idea. Simply $|has\_block|$ should do the trick. *has_block* is a nullary predicate, so it will either be the top concept or the bottom concept. [FRED *runs the tool again.*]

AMELIA: That seems to have worked. The feature now has a negative weight.

AMELIA: Looking at the output, it seems like one way to improve the heuristic would be to have a feature that counts up how many blocks each tower has too much or is missing. Although I'm not sure how to write a feature like that.

FRED: I don't think that's possible. The maximum total number of missing blocks does not scale linearly with the number of objects The multiplication feature is the only way to express concepts which are not linear in the number of objects and it is too coarse to express the exact number of blocks.

AMELIA: Are you sure that it is impossible?

FRED: Absolutely. It can only express composite numbers and the number of blocks might be a prime number.

AMELIA: We could manually specify features for towers which are one block off, two blocks off and so on. If we define $pred := succ^{-1}$ we can define

$$one\_too\_high := position \sqcap height \circ succ = height_G$$
$$one\_too\_low := position \sqcap height \circ pred = height_G$$

and similar concepts for different numbers. The first component of a member of the *succ* relation is the successor of the second component, that makes these two

FRED: The only tasks we are learning from have a maximum height of three, so there is no reason to add more than three of these. I don't expect this will generalize very well.

AMELIA: I don't either.

FRED: But we can run the tool regardless and see what we get back. I think this will likely take a while, so we can consider what other features we might want to add while the program is busy.

AMELIA: What might be helpful is a feature that involves the position of the robot. So far we've ignored that entirely and most reported states differ only by the position of the robot.

FRED: Maybe we can measure the distance of the robot to something? Like we did for the VisitAll domain.

AMELIA: That is likely not very helpful since how good a certain position depends on whether the robot is carrying a block or not. If it is, it should move towards a position where it should place a block, if it doesn't it should move towards a position where it should take one.

FRED: I think we can express a conditional like that. Something along the lines of

$$dist(at \sqcap has\_block, traversable, constructable)$$
$$dist(at \sqcap \neg has\_block, traversable, deconstructable)$$

would work. The role *traversable* needs to represent two neighbors with a height difference of at most one and *constructable* and *deconstructable* concepts where a bock should be placed or removed respectively.

AMELIA: *traversable* seems easy enough to define, but *constructable* and *deconstructable* seem very hard and I don't like how the usefulness of these distance features depend so much on the accuracy of these two concepts.

FRED: It's definitely not ideal. Does

$$\begin{aligned} \mathit{traversable} := & \exists x \in \mathit{numb}.(\\ & \quad \mathit{neighbor}[\exists \mathit{height}.\{x\}, \exists \mathit{height}.\{x\}] \\ & \sqcap \mathit{neighbor}[\exists \mathit{height} \circ \mathit{succ}.\{x\}, \exists \mathit{height}.\{x\}] \\ & \sqcap \mathit{neighbor}[\exists \mathit{height}.\{x\}, \exists \mathit{height} \circ \mathit{succ}.\{x\}]) \end{aligned}$$

seem right to you?

AMELIA: That's more complex than I thought it would be, but that does seem right. I think *constructable* and *deconstructable* will be especially hard because it won't be sufficient to consider the goal height and current height of the towers.

FRED: I see what you mean. If we did that we might encourage the robot to deconstruct ramps that it needs to build taller towers. Or rather, those situations might lead to the feature not being useful at all. Kumar et al. (2014) discuss an interesting concept in their attempt to create a domain-specific algorithm for TERMES. They consider towers to cast a shadow in all directions. If a position is n spaces removed from a tower with height $h > n$, then the shadow of that tower is $h - n$ at that position.

AMELIA: And then we could use that height to capture whether it might be useful to exceed the target height of a tower?

FRED: Exactly, you most definitely never want to exceed the maximum of the target height and the highest shadow cast by another tower, so that should be a useful feature.

AMELIA: I think we will only be able to implement this up to a fixed distance.

FRED: You're right, I don't see how we could express this accurately for a tower of arbitrary size, but we could, for instance, consider shadows up to a distance of at most three, which allows us to deal with towers with a maximum height of four and it should still be better than no feature at all for higher towers.

AMELIA: Right first we need to find towers that should cast a shadow

and the starting height of those shadows. How about

$$\begin{aligned} \mathit{too\_high} &:= \mathit{height} \sqcap (\mathit{height}_G \circ \mathit{pred}^+) \\ \mathit{too\_low} &:= (\mathit{height} \circ \mathit{pred}^+) \sqcap \mathit{height}_G \\ \mathit{initial\_shadow\_height} &:= \mathit{too\_high} \sqcup \mathit{too\_low}? \end{aligned}$$

FRED: So that contains all towers that do not have the goal height and associates it with the maximum of its current height and its goal height?

AMELIA: Getting the shadow height for the neighbors of these towers is then quite straightforward.

$$\begin{aligned} \mathit{shadow\_1\_height} &:= \mathit{neighbor} \circ \mathit{initial\_shadow\_height} \circ \mathit{succ} \\ \mathit{shadow\_2\_height} &:= \mathit{shadow\_1\_height} \\ &\quad \sqcup \mathit{neighbor} \circ \mathit{shadow\_1\_height} \circ \mathit{succ} \\ \mathit{shadow\_3\_height} &:= \mathit{shadow\_2\_height} \\ &\quad \sqcup \mathit{neighbor} \circ \mathit{shadow\_2\_height} \circ \mathit{succ} \end{aligned}$$

FRED: Won't those roles contain multiple heights for the same position if more than one of its neighbors have the incorrect height?

AMELIA: I think you're right about that, but we can filter out everything but the highest height for each position.

$$\begin{aligned} \mathit{height\_aux} &:= \mathit{height}_G \sqcup \mathit{shadow\_3\_height} \\ \mathit{max\_height} &:= \exists p \in \mathit{position}.(\\ &\quad \mathit{height\_aux}[\{p\}, \neg\exists \mathit{pred}^+ \circ \mathit{height\_aux}^{-1}.\{p\}]) \end{aligned}$$

FRED: And then we can define *one_over_max_height*, *one_under_max_height* and so on similarly to how we defined *one_too_high*.

AMELIA: Exactly, though I don't think that's enough to use for the distance feature, we need to make sure a block can actually be placed or removed.

FRED: The concepts

$$\begin{aligned} \mathit{constructable} &:= \mathit{under\_max\_height} \sqcap \\ &\quad \exists \mathit{height} \sqcap (\mathit{neighbor} \circ \mathit{height}).\mathit{numb} \\ \mathit{deconstructable} &:= \mathit{over\_max\_height} \sqcap \\ &\quad \exists \mathit{height} \sqcap (\mathit{neighbor} \circ \mathit{height} \circ \mathit{succ}).\mathit{numb} \end{aligned}$$

ensure that the tower has a neighbor that is the same height if a block needs to be placed and a neighbor that is one block lower if a block needs to be removed.

AMELIA: A final feature I can think to add is one that counts the towers which are the wrong height and can not currently be reached by the robot.

$$\begin{aligned} \mathit{wrong\_height\_unreachable} := \mathit{position} \sqcap \neg \mathit{correct\_height} \\ \sqcap \neg \exists \mathit{traverasable}^{+}.\mathit{at} \end{aligned}$$

FRED: Yes, it definitely seems like a good idea to discourage that situation from arising. I think we should probably test what sort of heuristic we can get from this now.

Discussion. In this exchange, Amelia and Fred encountered some of the limits of what can be expressed by the concept language. The partial restriction, role intersection and union as well as the existential quantifier are needed a few times to express certain features. Ultimately, they discover that the multiplication feature is not sufficient to express some useful features of the TERMES domain and have to resort to a kludge which breaks down when the tasks get too big. It would be possible to express the number of missing and excessive blocks if the cardinality feature were expanded to roles. However, we suspect the shadow height could still not be expressed for an arbitrarily high maximum tower height.

Table 2 shows the evaluation score of various heuristics based on the features found by Amelia and Fred. We added multiple features at once in the following order:

1. $|\mathit{correct\_height}|$ and $|(\exists \mathit{too\_low}.\mathit{numb}) \sqcap \mathit{has\_block}|$
2. $|\exists \mathit{too\_low}.\mathit{numb}|$ and $|\exists \mathit{too\_high}.\mathit{numb}|$
3. $|\mathit{one\_too\_low}|$, $|\mathit{two\_too\_low}|$, and $|\mathit{three\_too\_low}|$
4. $|\mathit{one\_over\_max\_height}|$, $|\mathit{two\_over\_max\_height}|$, $|\mathit{three\_over\_max\_height}|$, $|\mathit{one\_under\_max\_height}|$, $|\mathit{two\_under\_max\_height}|$, and $|\mathit{three\_under\_max\_height}|$
5. $\mathit{dist}(\mathit{at} \sqcap \mathit{has\_block}, \mathit{traversable}, \mathit{constructable})$ and $\mathit{dist}(\mathit{at} \sqcap \neg \mathit{has\_block}, \mathit{traversable}, \mathit{deconstructable})$
6. $|\mathit{wrong\_height\_unreachable}|$
7. $|\mathit{constructable}|$ and $|\mathit{deconstructable}|$

Method	Number of Features 2	4	7	13	15	16	18
blind	0.05						
slack	4.83	1.03	0.00	0.31	0.00	0.00	0.00
h^*	5.11	1.92	3.75	0.37	0.00	0.00	0.00
FF	9.58						
FF-slack	8.20	9.23	10.72	4.36	4.46	3.65	2.70
FF-h^*	8.99	8.74	9.92	7.95	7.41	4.64	3.64

Table 2: Evaluation scores of TERMES heuristics.

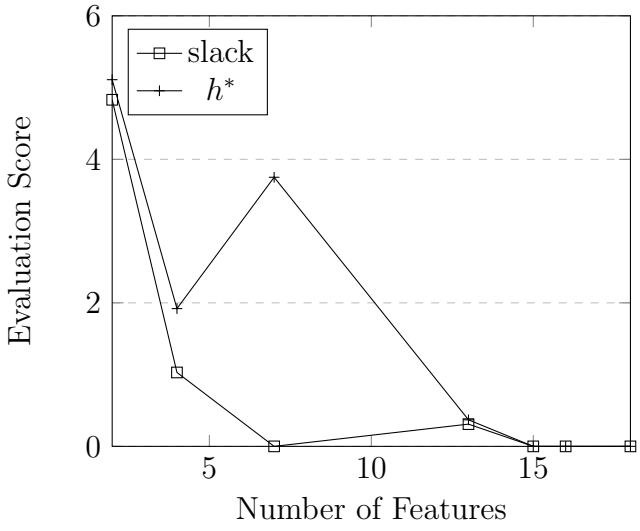
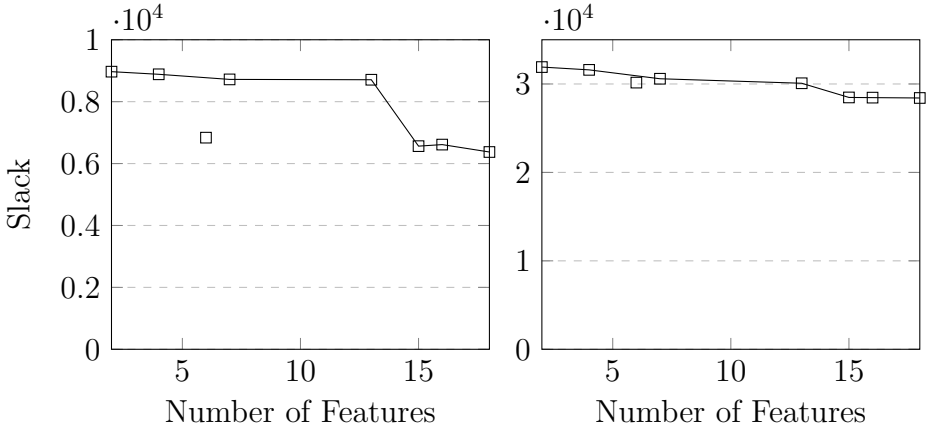


Figure 3: Evaluation scores of TERMES heuristics.



(a) Heuristics found by $\mathcal{M}_{slack}$. (b) Heuristics found by $\mathcal{M}^*$.

Figure 4: Total slack value of Logistics heuristics found by two different mixed integer programs.

The heuristics were trained on the full state space of four small TERMES tasks. As can be plainly seen, while adding some features allows a significant improvement over blind search in this regard and a modest improvement over the FF heuristic with the right set of features, adding more eventually greatly diminishes the performance of all heuristics. The heuristics which attempt to improve on the FF heuristic quickly become markedly worse than the FF heuristic itself and the heuristics using only the features become worse than a blind heuristic, solving not a single task within the time limit. In both cases, the heuristic found by $\mathcal{M}^*$ deteriorates less quickly. It is possible that this discrepancy is caused by problems similar to the one Amelia and Fred discovered when they used only the first feature. This is somewhat supported by the differences between the unsuccessful runs with the two heuristics. The smallest task was solved by a blind search with a plan length of 36 in just over a second. The heuristic found by $\mathcal{M}^*$ using the first five sets of features failed to improve upon the heuristic value it found after just under a minute at a search depth of 136. By contrast, the heuristic found by $\mathcal{M}_{slack}$ reached its lowest heuristic value in just 16 seconds at a search depth of 300. This conjures up an image of a guide who very confidently leads you in a myriad of wrong directions until they get stuck entirely. We saw this same tendency in different runs, but more a more in-depth study would need to be conducted to make any definite statements on this matter.

Perhaps more interesting than the difference between the two approaches is the similarity. The fact that they both perform worse when adding more features beyond a certain point suggests that they either do not approximate

the right kind of heuristic, or that they approximate this kind of heuristic in the wrong way. Wilt and Ruml (2016) showed that simply approaching h^* is not a guarantee for improved performance in sub-optimal planning, even if it is approached very well. As can be seen in Figure 4, the total slack value did generally decrease with more features, but in both cases there is still considerable room for improvement. Unsurprisingly, $\mathcal{M}^*$ especially did not approach h^* well at all. For both approaches, adding the two distance features caused the biggest drop of the slack value, but made the performance of the heuristic worse. We also tried adding these features as the third set of features instead of the fifth set. This resulted in lower slack values, as depicted by the lonely data points in Figure 4. For $\mathcal{M}_{slack}$ this had no measurable impact on the performance since neither the original third heuristic nor this alternative one solved any tasks. For $\mathcal{M}^*$ this reduced the score down to 1.22 from 3.75 original third heuristic.

Finally, it is also noteworthy that the total slack value actually increased when we added the penultimate set of features. This is due to the fact that the solver hit the time limit for all but the smallest $\mathcal{M}_{slack}$ program and there is no guarantee that any of the heuristics it found are optimal. By contrast, the solver found the optimal solution for $\mathcal{M}_{slack}$ every single time.

A set of features may exist which would have resulted in a heuristic with fantastic performance. It is also possible that this set of features could plausibly be found by a human with the help of this tool. However, this tool is intended to be used by fallible humans who can not be expected to find the perfect features unaided.

5.4 Sokoban

The Sokoban domain is modeled after a game of the same name. In this game, agents must push stones onto predefined goal spaces. An agent can only push one stone at a time, and can only push a stone away from itself when it occupies a neighboring position and moves to the original position of the stone by doing so. Agents can move free of cost to unoccupied neighboring spaces, but pushing a stone in any direction has a cost of 1. The goal is for every stone to occupy a goal space with a minimal amount of stone pushes.

For the International Planning Competition 2018 Sokoban was encoded using a ternary predicate *MOVE-DIR* in such a way that the interpretation of *MOVE-DIR*$(p1, p2, dir)$ is that *p1* is connected *p2* in the direction *dir*. This provides a very good argument for allowing n-ary roles since it eliminates the requirement to compile away these roles and separates the concerns of modeling a domain and specifying knowledge about it.

We implemented the following concepts and roles and features:

$$\begin{aligned}
\mathit{connected} &:= \exists \mathit{dir} \in \mathit{direction}.\mathit{MOVE\text{-}DIR}[\top, \top, \mathit{dir}] \\
\mathit{stone_location} &:= \exists \mathit{at}^{-1}.\mathit{stone} \\
\mathit{non_goal_stone} &:= \mathit{stone_location} \sqcap \neg \mathit{IS\text{-}GOAL} \\
\mathit{free_stone} &:= \mathit{stone_location} \sqcap (\geq 2\ \mathit{connected}.\mathit{clear}) \\
\mathit{player_location} &:= \exists \mathit{at}^{-1}.\mathit{player} \\
\mathit{freeable} &:= \mathit{clear} \sqcup \mathit{free_stone} \sqcup \mathit{player_location} \\
\mathit{free_move_dir} &:= \mathit{MOVE\text{-}DIR}[\mathit{freeable}, \mathit{freeable}, \mathit{dir}] \\
\mathit{push_dir} &:= \exists \mathit{dir} \in \mathit{direction}.(\\
&\qquad \mathit{free_move_dir}^{-1} \circ \mathit{free_move_dir} \circ \mathit{free_move_dir})
\end{aligned}$$

and the following features based on those concepts and roles:

$$\begin{gathered}
|\mathit{non_goal_stone}| \\
\mathit{dist}(\mathit{player_location}, \mathit{connected}, \mathit{non_goal_stone}) \\
\mathit{dist}(\mathit{non_goal_stone}, \mathit{push_dir}, \mathit{is_goal} \sqcap \neg \mathit{stone_location}) \cdot |\mathit{location}| \\
|\mathit{stone_location} \sqcap \mathit{stone_location}| \cdot |\mathit{location}| \cdot |\mathit{location}|.
\end{gathered}$$

The role *push_dir* shows another reason why the existential quantifier and partial restriction are valuable additions to the concept language and *free_stone* shows an example of a qualified cardinality restriction. It could also be used to detect corridors with a concept like $(= 2\ \mathit{connected}.\mathit{location})$ as well as rooms with $(> n\ \mathit{neighborhood}.\mathit{location})$ for some natural number n and some role *neighborhood* which should associate each location with some the locations around it. Disappointingly, we were not able to get a useful heuristic out of these features. However, this is not particularly surprising considering the problems we faced with the TERMES domain.

Culberson (1999) showed that Sokoban is PSPACE-complete and Junghanns and Schaeffer (2001) demonstrated that adding sophisticated, domain-specific knowledge can provide an orders-of-magnitude improvement to search efficiency for Sokoban. So it is possible that with a better learning step this problem could be tackled, but that would probably require a considerable effort.

6 Future Work

There are many ways left to improve the tool created for this thesis. We have already mentioned the biggest issue, namely that $\mathcal{M}^*$ and $\mathcal{M}_{slack}$ do

not appear to result in heuristics that are good for satisficing planning when the features supplied to them are not good enough. It would certainly be worth it to study exactly what type of combination of features makes the resulting heuristic perform worse. Additionally, there are several avenues available to improve the learning portion of this system. A better MIP may exist which would alleviate the problems we have encountered with our approach and a better understanding of what exactly causes these problems would certainly help there. Rovner (2020) presents one such MIP based on the Goal Distance Rank Correlation introduced by Wilt and Ruml (2016) in the setting of potential heuristics for non-generalized planning. Alternatively, one could attempt to learn a heuristic which explicitly maximizes the Goal Distance Rank Correlation. This is not possible with linear programming but could likely be achieved by using a neural network for the learning. This would have the added advantage that the range of heuristics that can be expressed in terms of a set of features greatly increases. As a result, there could be less pressure on the expert to provide the perfect features. The downside to this approach would be a significant loss of the explicability of the heuristic.

Another potential way to make more of the imperfect features supplied by the user is to augment the set of features with combinations of the supplied concepts and roles and features. As an example, in the VisitAll domain this could have resulted in the system discovering that the multiplication of the number of nodes and the distance to the nearest unvisited node is useful before the user does.

To increase the productivity of those working with the tool, it would be beneficial if it were possible to explore the state spaces and try out concepts on states. Instead of simply printing a few difficult states after finding a heuristic, the tool could enter an interactive mode where the user can specify new features and concepts as well as query for states and look at the heuristics arising from different solutions the solver found.

The concept language and features presented in this thesis could also use more work. There are a few features that we considered but did not end up evaluating. We will list all three of them here for posterity. They are the maximum distance between two concepts along a role, the weighted minimum distance between two concepts along a role where the weight of an edge is determined by the distance between the two individuals along a second role, and the role cardinality. It is also thinkable that concept languages are not the right tool for this task and it would be better to allow users to specify features in a Turing complete scripting language.

Finally, it would also be invaluable to use the states encountered during actual searches to improve the heuristic.

7 Conclusion

The main results of this thesis are a tool which makes specifying domain knowledge in the form of concept languages significantly easier. We demonstrated that someone without much planning expertise can plausibly find a general solution to an entire domain of problems using this tool. We also considered some extensions to the concept language that has most often been used in planning. Most notably, we introduced a novel way to incorporate n-ary predicates into concept languages.

Unfortunately, we were not able to use this tool to find a good heuristic for any domain classical planning typically struggles with. But we did set a baseline that still has multiple possible avenues for improvement in the future.

References

Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, chapter 3, pages 135–180. Elsevier, 2007.

Blai Bonet, Guillem Francès, and Héctor Geffner. Learning features and abstract actions for computing generalized plans. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*, pages 2703 – 2710, 2016.

Joseph C. Culberson. Sokoban is PSPACE-complete. *International Conference on Fun with Algorithms*, pages 65–76, 1999.

Rik de Graaff. Downward guide. 2020. doi: 10.5281/zenodo.3900929. URL `https://doi.org/10.5281/zenodo.3900929`.

Guillem Francès, Augusto B. Corrêa, Cedric Geissmann, and Florian Pommerening. Generalized potential heuristics for classical planning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI 2019)*, pages 5554–5561, 2019a.

Guillem Francès, Augusto B. Corrêa, Cedric Geissmann, and Florian Pommerening. Generalized potential heuristics for classical planning: additional material. In *Technical Report CS-2019-003*. University of Basel, Departement of Mathematics and Computer Science, 2019b.

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. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

Jörg Hoffmann. FF: The fast-forward planning system. *AI magazine*, 22(3): 57–62, 2001.

Bernhard Hollunder and Franz Baader. Qualifying number restrictions in concept languages. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, KR 1991, page 335–346. Morgan Kaufmann Publishers Inc., 1991.

Andreas Junghanns and Jonathan Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1–2):219–251, 2001.

TK Satish Kumar, Sangmook Johnny Jung, and Sven Koenig. A tree-based algorithm for construction robots. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*, pages 481 – 489, 2014.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – the planning domain definition language. *Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control*, 1998.

Drew M McDermott. The 1998 AI planning systems competition. *AI magazine*, 21(2):35–55, 2000.

Alexander Rovner. Potential heuristics for satisficing planning. Master's thesis, University of Basel, 2020.

Jendrik Seipp, Florian Pommerening, Gabriele Röger, and Malte Helmert. Correlation complexity of classical planning domains. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pages 3242–3250, 2016.

Christopher Wilt and Wheeler Ruml. Effective heuristics for suboptimal best-first search. *Journal of Artificial Intelligence Research*, 57:273–306, 2016.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

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

Author – Autor
Rik de Graaff

Matriculation Number – Matrikelnummer
17-057-043

Title of Work – Titel der Arbeit
Concept Languages as Expert Input for
Generalized Planning

Type of Work – Typ der Arbeit
Bachelor's 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,

Signature – Unterschrift