Concept Languages as Expert Input for Generalized Planning: Preliminary Results

Rik de Graaff, Augusto B. Corrêa, Florian Pommerening
University of Basel, Switzerland
{rik.degraaff, augusto.blaascorrea, florian.pommerening}@unibas.ch

Abstract

Planning is an attractive approach to solving problems because of its generality and its ease of use. A domain expert often has knowledge that could improve the performance of such a solution but this knowledge may be vague or not directly actionable. For example, an expert could be aware that a certain property is important but not have a good way of using this knowledge to speed up the search for a solution. In this paper we evaluate concept languages as a possible input language for injecting 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.

Introduction

The oft-repeated maxim "physics, not advice" (McDermott 2000) states that the 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 the tasks. This is sensible because the person specifying the domain might have incorrect ideas about the best solution strategies. It is a separation of concerns: specifying a planning domain and solving it are different problems requiring different expertise. For this reason planning algorithms require no domain knowledge apart from a model of the world.

Search algorithms, such as A* (Hart, Nilsson, and Raphael 1968) and greedy best-first search (Doran and Michie 1966), in combination with informative domain-independent heuristics are considered the state of the art in optimal and satisficing planning (Katz et al. 2018; Seipp and Röger 2018). However, this "planning-as-heuristic-search" (Bonet and Geffner 2001) paradigm usually cannot compete with domain-specific algorithms precisely because of their generality. Domain-independent heuristics by definition cannot include domain-specific knowledge. A domain expert has such knowledge but might have no way to incorporate it into an algorithm. The knowledge can be vague (e.g., "the number of blocks above a misplaced block is important") or even misleading for a specific algorithm. In this paper, we explore the middle ground between planning and domain-specific solutions by allowing experts to specify domain knowledge in an iterative process.

We allow the user to *insert advice* for a planner in the form of concepts. A concept is a formula in a description logic that describes a set of objects that the expert considers important in this domain. The description is general in the sense that a concept can be evaluated in any task of the domain. Concepts can be treated as advice by incorporating them in a heuristic function that is then used in the search. Francès et al. (2019) defined *generalized potential heuristics* based on such concepts in the area of generalized planning. These heuristics are represented as a weighted sum of concept-based features. Once the features and weights are fixed, the heuristic can be used in any task of the domain. Francès et al. found that features often represent aspects that have a clear meaning to humans and these heuristics can be interpreted in natural language. They proposed a way to automatically discover useful features but this does not scale to larger domains due to the vast search space of candidate heuristics.

In this work, we explore how useful concept languages are as a means to express expert knowledge of a domain. Using the concepts in a generalized potential heuristic also allows us to deal with vague or misleading knowledge, as the user only specifies the concepts, while the system then finds appropriate weights. An analogy to this is evaluating a chess position, where the expert would specify that the number of pawns and rooks is important and the system would then figure out to value rooks 5 times higher than pawns.

We start from the same concept language used by Francès et al. (2019) and introduce new extensions. We also use a different way of learning the weights: their method tries to find weights that give strong theoretical guarantees, which might not exist in a practical domain. We instead look for weights that approximate these guarantees as close as possible. If the features added by the expert are not sufficient to find good weights, we present them with feedback to help them analyze the problem and find missing features.

Classical Planning

We explain our work in the context of *classical planning*. However, we only make light assumptions on the planning model, so it should be easy to adapt our methods to a more general setting. We discuss this in the last section.

The *domain* of a classical planning task is a tuple $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$, where $\mathcal{P}$ is a set of *predicates*, $\mathcal{A}$ is a set of *action* *schemas*, $\mathcal{C}$ is a set of *constants*. An action schema $a[X] \in \mathcal{A}$ is represented as a set of parameters X, a precondition $pre(a[X])$, an effect $\mathit{eff}(a[X])$, and a cost $cost(a[X]) \in \mathbb{R}_{\geq 0}$. The precondition $pre(a[X])$ is a first-order logical formula over the predicates in $\mathcal{P}$ applied to $X \cup \mathcal{C}$, and the effect $\mathit{eff}(a[X])$ is a conjunction over such predicates and their negations.

A planning task Π in a domain $\mathcal{D}$ is a tuple $\langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{O}$ is a set of objects, $\mathcal{I}$ is the *initial state*, and $\mathcal{G}$ is the *goal condition*. A predicate $P \in \mathcal{P}$ applied to objects in $\mathcal{C} \cup \mathcal{O}$ is an atom and a (partial) truth assignment to all atoms is a (partial) state. The initial state $\mathcal{I}$ is a state and the the goal condition $\mathcal{G}$ is a partial state. A state s with $\mathcal{G} \subseteq s$ is a *goal state*. We denote the set of all states by S.

Action schemas $a[X] \in \mathcal{A}$ can be grounded by replacing the parameters X in $pre(a[X])$ and $\mathit{eff}(a[X])$ of action schema a with objects from $\mathcal{O}$. The result of grounding an action schema with is a (ground) *action* a with no parameters and the same cost as the action schema. If the precondition of an action is satisfied in a state, we say the action is *applicable*. If an action a is applicable in a state s then applying it leads to the successor state $s[a]$ which assigns all positive literals in $\mathit{eff}(a)$ to true, all negative literals in $\mathit{eff}(a)$ to false, and all remaining atoms to the value they had in s. The set of successor states of a state s is $succ(s) = \{s[a] \mid a \text{ is applicable in } s\}$.

A sequence of actions $\langle a_1, \dots, a_n \rangle$ is a *path* from s_0 to s_n if there are states $\langle s_1, \dots, s_{n-1} \rangle$ where a_i is applicable in s_{i-1} and $s_{i-1}[a_i] = s_i$ for all $i \leq n$. Its cost is $\sum_{i=1}^{n} cost(a_i)$. 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* is a function $h : S \to \mathbb{R}$. The perfect heuristic h^* maps each state to the cost of an optimal plan. A *generalized heuristic* is a function that is defined for all states of all tasks in a given domain. A heuristic h is *descending* on a state s if s has at least one successor s' where $h(s') \leq h(s) - 1$. The heuristic function 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 heuristic is descending and dead-end avoiding if it is descending and dead-end avoiding on every alive state. A heuristic with this property guides standard greedy algorithms directly to a goal (Francès et al. 2019).

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, Horrocks, and Sattler 2007). Concept languages deal with *individuals*; classes of individuals which are called *concepts*; and binary relationships between individuals which are called *roles*. A concept language is defined by which *constructors* for concepts and roles it allows. Similar to Francès et al. (2019), we consider the standard language $\mathcal{SOIQ}$ with the added constructors of role-value-map, role union and intersection. However, we also extend the language used by Francès et al. (2019) with a non-standard constructor to deal with higher-arity relations between individuals.

Syntax and Semantics Concepts and roles are defined inductively and are interpreted with a model $\cdot^{\mathcal{M}}$ relative to a universe of individuals Δ. A set of *named individuals*, *named concepts* and *named roles* form the basis of the inductive definition. The model $\cdot^{\mathcal{M}}$ maps named individuals a to individuals $a^{\mathcal{M}} \in \Delta$, named concepts c to subsets of individuals $c^{\mathcal{M}} \subseteq \Delta$, and named roles r to relations between objects $r^{\mathcal{M}} \subseteq \Delta^2$. The remaining concepts and roles are built and interpreted as follows from existing concepts C, C', roles R, R', named individuals $a_1, \dots a_n$, natural numbers n, and comparison operators $\sim \in \{=, >, <, \geq, \leq\}$:

$$\top^{\mathcal{M}} = \Delta, \bot^{\mathcal{M}} = \emptyset, (\neg C)^{\mathcal{M}} = \Delta \backslash C^{\mathcal{M}},$$
$$(C \sqcup C')^{\mathcal{M}} = C^{\mathcal{M}} \cup C'^{\mathcal{M}}, (C \sqcap C')^{\mathcal{M}} = C^{\mathcal{M}} \cap C'^{\mathcal{M}},$$
$$(R \sqcup R')^{\mathcal{M}} = R^{\mathcal{M}} \cup R'^{\mathcal{M}}, (R \sqcap R')^{\mathcal{M}} = R^{\mathcal{M}} \cap R'^{\mathcal{M}},$$
$$(\exists R.C)^{\mathcal{M}} = \{x \mid \exists y.(x,y) \in R^{\mathcal{M}} \wedge y \in C^{\mathcal{M}}\},$$
$$(\forall R.C)^{\mathcal{M}} = \{x \mid \forall y.(x,y) \in R^{\mathcal{M}} \to y \in C^{\mathcal{M}}\},$$
$$\{a_1, \dots, a_n\}^{\mathcal{M}} = \{a_1^{\mathcal{M}}, .., a_n^{\mathcal{M}}\},$$
$$(R = R')^{\mathcal{M}} = \{x \mid (x,y) \in R^{\mathcal{M}} \leftrightarrow (x,y) \in R'^{\mathcal{M}}\},$$
$$(R^{-1})^{\mathcal{M}} = \{(x,y) \mid (y,x) \in R^{\mathcal{M}}\},$$
$$R^{+\mathcal{M}} = \{(x_0, x_n) \mid \exists x_1, \dots x_{n-1}. \; (x_{i-1}, x_i) \in R^{\mathcal{M}} \text{ for all } i \in \{1, \dots, n\}\},$$
$$(R \circ R')^{\mathcal{M}} = \{(x,y) \mid \exists z.(x,z) \in R^{\mathcal{M}} \wedge (z,y) \in R'^{\mathcal{M}}\}.$$
$$(\sim n \; R.C)^{\mathcal{M}} = \{x \mid \#\{y \mid (x,y) \in R^{\mathcal{M}} \wedge y \in C^{\mathcal{M}}\} \sim n\}.$$

For a more detailed description, we refer to the book by Baader, Horrocks, and Sattler (2007). As an example, let *package* be a named concept that is mapped to all packages in a logistics problem, and *in* be a role that is mapped to the relation of which object is in which truck. Then, the concept $(\mathit{package} \sqcap \exists \mathit{in}.\top)$ represents all packages loaded in some truck. If, additionally, *at* is a role relating trucks to locations and *connected* is a role encoding which locations are connected by a road, then $\forall \mathit{at}.\exists \mathit{connected}^{+}.\{A\}$ describes the set of all trucks that can drive to location A. We can combine these two concepts so $(\mathit{package} \sqcap \exists \mathit{in}.\forall \mathit{at}.\exists \mathit{connected}^{+}.\{A\})$ describes all packages that can be delivered to location A.

Extensions

Next, we introduce new extensions to the concept language. These extensions make the concept language more expressive, at the cost of the computational guarantees for the language $\mathcal{SOIQ}$. However, in our work, we are only concerned with evaluating concepts and roles in a finite model which remains efficiently possible.

Higher-arity roles In our applications there might be relations between objects that are not binary. Francès et al.

(2019) dealt with n-ary roles by replacing them with multiple binary roles but as an input language for expert knowledge, it is more natural to talk about such roles directly. We thus allow named roles of any arity in our concept language, where an n-ary role R is interpreted as $R^{\mathcal{M}} \subseteq \Delta^n$. The above definitions all assume binary roles, so we introduce a projection operator $\cdot^{\bar{\pi}(m)}$ that projects away position $m \leq n$ of an n-ary role R.

$$\begin{aligned}(R^{\bar{\pi}(m)})^{\mathcal{M}} =& \{(x_1, \dots, x_{m-1}, x_{m+1}, \dots, x_n) \\ & \mid \exists x_m.(x_1, ..., x_n) \in R^{\mathcal{M}}\}.\end{aligned}$$

Projection *to* a subset I of indices (denoted $\pi(I)$) can be seen as syntactic sugar, that repeatedly projects away all indices outside of I.

In addition to projecting roles to a lower arity, we allow to restrict roles to entries where objects in a given position m have to be in a given concept C. We call this operation *atomic selection*:

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

We define *general selection* as syntactic sugar: if R is an n-ary relation and $C_1, \dots, C_n$ are concepts, then $R[C_1, \dots, C_n]^{\mathcal{M}} = ((\cdots(R^{1 \in C_1})\cdots)^{n \in C_n})^{\mathcal{M}}$. In our previous logistics example, if *red* is a concept containing all red trucks, then $\mathit{in}[\mathit{package}, \mathit{red}]^{\pi(\{1\})}$ represents the set of all packages in red trucks. As we can see in this example, unary roles can be treated as concepts.

Quantifiers for Roles We also introduce the *universal quantifier* and the *existential quantifier* as

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

where $\cdot^{\mathcal{M}(a=x)}$ denotes the model which is identical to $\cdot^{\mathcal{M}}$ with the addition of the new named individual a such that $a^{\mathcal{M}} = x$.

Concept Languages of Planning Tasks

We focus on planning tasks represented in the STRIPS fragment of PDDL (McDermott 2000). Extending the technique to a larger PDDL fragment would be straight-forward as we only make light assumptions on representation of the states and the goal. Like Francès et al. (2019) we define the concept language of a planning domain $\langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$ as the language where the named individuals are $\mathcal{C}$, and the named concepts are the unary predicates in $\mathcal{P}$. In contrast to them, we do not replace higher-arity predicates in $\mathcal{P}$ by binary ones but represent all of them explicitly as higher-arity named roles. In a given task $\langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, we consider the set of individuals $\Delta = \mathcal{O} \cup \mathcal{C}$ in all states. For every state s we define the model $\cdot^{\mathcal{M}(s)}$ with $a^{\mathcal{M}(s)} = a$ for all named individuals a, $C^{\mathcal{M}(s)} = \{x \mid s \models C(x)\}$ for every named concept C, and $R^{\mathcal{M}(s)} = \{(x_1, \dots, x_n) \mid s \models R(x_1, \dots, x_n)\}$ for every named role R. The exception are nullary concepts where we define $R^{\mathcal{M}(s)} = \Delta$ if $s^{\mathcal{M}(R)}$ and $R^{\mathcal{M}(s)} = \emptyset$ otherwise.

We also use *goal concepts* C_G and *goal roles* R_G for all named concepts C and roles R. We define them slightly more general than Francès et al. to enable their use in larger PDDL fragments:

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

Generalized Potential Heuristics

Francès et al. introduced *generalized potential heuristics* as a weighted sum over features which map states to integers.

Definition 1 *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*

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

We use the *cardinality* and *distance features* defined by them and add *product* and *heuristic features* as further ways to express task-independent properties.

Definition 2 *Let s be a state, C and C' be concepts, R a role, and h a generalized heuristic.*

- *The value of the* cardinality feature $|C|$ *in s is* $|C^{\mathcal{M}(s)}|$.
- *The value of the* minimal distance feature $\mathit{dist}(C, R, C')$ *in s is the smalles n such that there are $x_0, \dots x_n$ 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 $i \leq n$. If no such individuals $x_0, \dots x_n$ exist, the value is* 0.
- *The value of the* heuristic feature f_h *in s is $h(s)$.*
- *For two features f_1 and f_2, the value of the* product feature $f_1 \cdot f_2$ *in s is $f_1(s) \cdot f_2(s)$.*

Note that heuristic features allow to incorporate any other heuristic into the mix which might already have good performance on the domain. The remaining features can then be used to mitigate weaknesses of the heuristic.

The product feature solves a problem that Francès et al. pointed out: cardinality and distance features can only return values up to $|\Delta|$ which means that generalized potential heuristics over them are limited to values that are linear in the number of objects in a task. With product features, we can express arbitrarily high values.

Expressing Expert Knowledge

Our main contribution is a system that allows domain experts to express concept-based features that represent relevant properties of the domain. The high level overview of this system is illustrated in Figure 1. We explain the details of this process below.

The expert does not need to be familiar with the internal of the planning system they use to solve the problems. Instead, they only need to know which aspects of the domain are important and have a working knowledge of the domain model and description logics to a degree where they can express

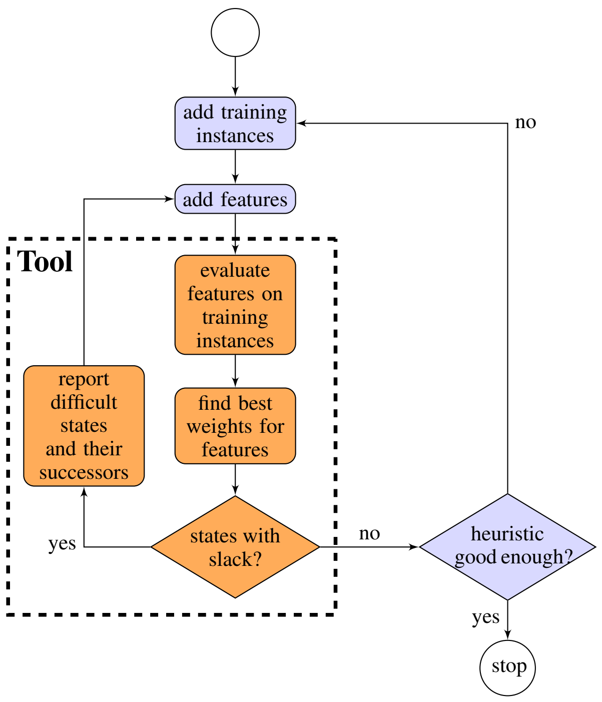
add training instances
no
add features
Tool
evaluate features on training instances
find best weights for features
report difficult states and their successors
states with slack?
yes
no
heuristic good enough?
yes
stop

Figure 1: Overview of the iterative process. Tasks inside the box are performed by the tool, while tasks outside the box are performed by the domain expert.

these aspects as features. The knowledge passed to the planner in this way should be thought of as hints ("This property is interesting") not directives ("You have to use this action").

The concepts used in our features express sets of objects with a certain property rather than the property itself but features than map these sets to numbers. For example, in a logistics domain, the number of undelivered packages is relevant and could be expressed with a cardinality feature $|package \sqcap \neg(at = at_G)|$. We explicitly do not require the expert to set any weights for these features, that is, we assume the expert has an idea about *what* is important but does not want to specify a full heuristic. Instead, we use a mixed integer program (MIP) to find good weights for the features that the expert specified. This process is described in the following section. It also leaves room for features that turn out to be less useful than the expert thought or could even guide the search in the wrong direction. The best weight for such features might be 0 (disabling the feature) or even negative.

Coming up with good features is not an easy task, even for someone familiar with the domain. In case the initial set of features is not sufficient to work in all situations, our system will detect this and present the user with a state where the heuristic is not informative enough to pick a good successor (we will describe this in more detail after discussing the MIP). The expert can then analyze this situation and add new concepts and features to the pool that cover this situation. This iterative process gives the MIP more choices for finding a useful heuristic for the domain so the average performance of the heuristic should improve over time. The user can stop the process at any time and use the discovered heuristic on all future tasks of the domain.

Francès et al. (2019) also learned weights for a generalized potential heuristic automatically. The difference is that they systematically generated all features up to a given formula size while we use a smaller pool of hand-crafted features. While their approach is more general, it has issues with scaling as a complex concept can only be considered if the feature pool is large enough to contain all concepts of its size. For example, although Francès et al. show a descending generalized potential heuristic for the Logistics domain, the features used were so complex that the systematic feature generation could not scale up to it. Yet, these features represent very intuitive knowledge about the domain and it is reasonable to expect that a user could come up with them by themselves. In fact, Francès et al. came up with this heuristic hand-crafting the features and testing it manually. This is also the main goal of our system: To allow users to choose a set of hand-crafted features in order to avoid the issues with automatic feature generation and to test it by themselves. In this way, we can use much larger concepts without increasing the solving time significantly.

Learning Heuristics

Francès et al. used a MIP to learn weights for a set of features such that the resulting potential heuristic is descending and dead-end avoiding. If such a heuristic can be found, it will guide a greedy search to a goal state without backtracking. However, in general such a heuristic might not exist for a given domain. Here we prefer practical applicability in all domains over strong guarantees in some, so instead of looking for a descending and dead-end avoiding heuristic, we approximate one. Our assumption is that a heuristic that is descending in most states has a high rank correlation with the perfect heuristic (Wilt and Ruml 2015) and thus guides a search to a goal state quickly. Alternatively, we can skip the step of approximating a heuristic that correlates with the perfect heuristic and approximate the perfect heuristic directly.

In both cases we start from a set of alive states $\mathcal{S}$ from a domain $\mathcal{D}$ and a set of *candidate features* $\mathcal{F}$ for the same domain. To find the set $\mathcal{S}$, we completely expand the state space of several small instances.

Approximating a descending and dead-end avoiding heuristic Francès et al. (2019) use a MIP over variables w_f for every feature $f \in \mathcal{F}$ to describe the weight for f. Their constraints enforce that the corresponding heuristic is descending and dead-end avoiding, while their objective function minimizes the complexity of the selected features. If no such heuristic exists, the MIP is infeasible. In our case, we want the MIP to always find some heuristic but do not care about the complexity of the selected features (the assumption is that the amount of user-provided features is small enough to use them all). We thus use *slack variables* $u_{s,s'}, v_{s,s'}$ for every state s and every transition $s \rightarrow s'$ from an alive state s to a state $s' \in succ(s)$. We then minimize the total slack on all constraints. Apart from this, the MIP is identical to the one used by Francès et al. (2019). We denote it by $\mathcal{M}_{slack}(\mathcal{S}, \mathcal{F})$. As a shorthand notation, we write

$h(s)$ to represent $\sum_{f\in\mathcal{F}} w_f \cdot f(s)$, where $f(s)$ represents the value of the feature f in the state s.

$$\text{Minimize} \sum u_{s,s'} + \sum v_{s,s'} \text{ s.t.}$$
$$\bigvee_{s'\in succ(s)} h(s) + u_{s,s'} \geq h(s') + 1 \text{ for all } s \in \mathcal{S}$$
$$h(s') + v_{s,s'} \geq h(s)$$
$$\text{for all } s \in \mathcal{S}, s' \in succ(s), s' \text{ unsolvable}$$
$$u_{s,s'} \geq 0 \text{ for all } s \in \mathcal{S}, s' \in succ(s)$$
$$v_{s,s'} \geq 0 \text{ for all } s \in \mathcal{S}, s' \in succ(s), s' \text{ unsolvable.}$$

With a slack of 0, the first constraint ensures that the heuristic is descending while the second ensures that it avoids dead ends. By minimizing the total slack $\mathcal{M}_{slack}(\mathcal{S},\mathcal{F})$ intuitively computes the generalized potential heuristic which is closest to being descending and dead-end avoiding using all the features provided.

Approximating the perfect heuristic Instead of trying to approximate a descending and dead-end avoiding heuristic, we can also approximate the perfect heuristic h^* which can be easily computed on the small tasks we use to generate $\mathcal{S}$. Our heuristic only has to match h^* along transitions from an alive state to a solvable successor as long as all transitions to unsolvable states are less attractive than a transition to a solvable successor. We can obtain weights for such a heuristic by solving the mixed integer program $\mathcal{M}^*(\mathcal{S},\mathcal{F})$ which uses the same set of variables as $\mathcal{M}_{slack}(\mathcal{S},\mathcal{F})$.

$$\text{Minimize} \sum |u_{s,s'}| + \sum v_{s,s'} \text{ s.t.}$$
$$h(s) - h(s') + u_{s,s'} = h^*(s) - h^*(s')$$
$$\text{for all } s \in \mathcal{S}, s' \in succ(s), s' \text{ is solvable}$$
$$h(s') + v_{s,s'} \geq \max_{\substack{t\in succ(s)\cup\{s\} \\ t \text{ solvable}}} h(t)$$
$$\text{for all } s \in \mathcal{S}, s' \in succ(s), s' \text{ is unsolvable}$$
$$v_{s,s'} \geq 0 \text{ for all } s \in \mathcal{S}, s' \in succ(s), s' \text{ unsolvable}$$

The first 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 constraint deals with unsolvable successors of solvable states. Since the value of the features, and thus the heuristic, can only be a finite number, the heuristic cannot indicate an unsolvable state by an infinite value. Instead, we strengthen the requirement for dead-end avoiding heuristics and compare the value of the heuristic for the unsolvable state to that for its predecessor and all of its solvable siblings.

Neither of these two mixed integer programs is linear because they use disjunctions, maximizations and absolute values. However, both can be solved by modern MIP solvers after some rewriting as described by Francès et al. (2019). In particular, $\mathcal{M}^*(\mathcal{S},\mathcal{F})$ can be rewritten as a linear program without integer variables so it can be solved more efficiently.

In some domains, it might happen that the set $\mathcal{S}$ is too large even when computed from very small instances. This produces a very large MIP which takes considerably more time to solve. To reduce this problem, we can randomly sample a number of states from $\mathcal{S}$ and use this sample to produce the MIP. This strategy was also done by Francès et al. (2019) to reduce the time needed to solve the MIP.

Improving the Heuristic Iteratively

When solving $\mathcal{M}_{slack}(\mathcal{S},\mathcal{F})$ and $\mathcal{M}^*(\mathcal{S},\mathcal{F})$, we can identify states which the heuristic is uninformed by checking the slack variables with the highest values. We can present the domain expert with these states and their successors as well as the feature values and heuristic value for these states. This 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 to alleviate the problem.

In case the highest-valued slack variable is some $u_{s,s'} > 0$ this means that the heuristic value does not sufficiently decrease along transitions starting from s. The state s is alive, so there has to be an optimal plan for s. A good heuristic should decrease along this plan (although this is not necessary as the heuristic could decrease along a different plan as well). The expert should then try to express the reason why the successor of s on this plan is better than s in terms of a concept-based feature.

Alternatively, if the highest-valued slack variable is some $v_{s,s'} > 0$, the heuristic would mislead the search into a dead end by not sufficiently separating the unsolvable state s' from the solvable state s. In this case the expert should look at the reason why the transition from s to s' makes solving the task impossible and express this with a concept-based feature.

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 $\mathcal{F}$, the expert can solve the mixed integer programs again with the new set of features. Heuristic features can be added to exploit both human intuition about a domain and information state of the art heuristics are good at detecting. The information provided by both is automatically weighted and mixed by the MIP finding appropriate weights. Using multiplication features it is also possible to multiply a heuristic with a (possibly empty) concept cardinality and thus use the heuristic only for states where it provides good estimates.

When the resulting heuristic captures everything in the sample states $\mathcal{S}$ reasonably well, the expert can add additional tasks, expanding $\mathcal{S}$. 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.

Use Cases

Next, we illustrate how our proposed iterative process can work in practice by discussing its use on three domains. To do so, we implemented the overall workflow as a Python tool. The user can give a domain and an initial set of features to the tool. Features can be formulated in a Manchester-like syntax[1]. The user then selects which MIP to solve to find weights for the specified features. We use the Python API[2] of CPLEX 12.10 for defining and solving the MIPs. To generate the states S used in the MIPs, the user also specifies a set of small training tasks. We use a modified version of Fast Downward 19.12 (Helmert 2006) to completely explore the state space of these tasks and sample state from the state spaces at a user-specified rate. Our version of Fast Downward evaluates the features for each state and returns a list of states together with the value of each feature in each state. We had to modify the translation phase (Helmert 2009) of Fast Downward to keep the static information in the finite-domain representation. Otherwise, we would not be able to evaluate concepts that refer to static predicates.

After the MIP finds an optimal solution (or reaches the time limit) it verifies if there are any states where the associate slack variables are not 0. If this is the case, it means that the heuristic is not descending or perfect. The top 5 states with highest value for their slack variables are displayed to the user together with their successors. The user then can add more features and repeat the process. Once the user is satisfied with the heuristic, they can export it to a file and use it with our modified version of Fast Downward for all tasks in the domain. The source code of our tool is available online (de Graaff 2020).

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. We evaluated the quality of the final heuristic by using it in a greedy search on a set of larger test tasks. This experiment was run on Intel Xeon E5-2660 CPUs with a time limit of 5 minutes for each task. To judge the heuristic quality, we measure the number of evaluations required by the greedy. These numbers are usually dominated by larger tasks, so we instead report an evaluation score between 0 (for not solving the task) and 1 (for solving the task in a single evaluation) where intermediate values are scaled logarithmically.

VisitAll

In the domain VisitAll, an agent has to visit every node of an undirected graph. The agent can move along an edge of the graph with a cost of one. As the graph is undirected, there are no dead ends in the domain. We used the smallest VisitAll task as the only training instance for this domain.

A first guess to what might be important in this domain is the number of unvisited nodes. We can express this with the feature $|\neg visited|$ and use it to initialize our tool. There are no descending heuristics with only this feature, so both of the MIPs find a solution with positive slack values. The states with the highest slack have in common that the agent is in a position where all neighboring nodes are already visited. A good action to take in this situation is to move to a neighboring node in a way that reduces the distance to an unvisited node. The distance feature $dist(at\text{-}robot, connected, \neg visited)$ expresses this quantity and can be added to the candidate pool.

After the second feature is added, both MIPs find a solution $k|\neg visited| + dist(at\text{-}robot, connected, \neg visited)$ for some weight k. As discussed by Francès et al. (2019), this is a descending heuristic for VisitAll if k is lower than the diameter of the graph. Our tool is not able to determine this, as adding larger state spaces to the training instances will only produce a heuristic with a higher value for k. However, the domain expert could notice this problem and add the feature $|\top| \cdot |\neg visited|$. This feature multiplies the number of all nodes with the number of unvisited nodes, so it effectively uses a sufficiently high weight for the first feature. The sum of this feature and the distance feature (both weighted with 1) is a descending heuristic for all VisitAll tasks.

Even though this heuristic is descending and will not require backtracking in the search, it may be suboptimal and thus produce suboptimal plans. We compare its evaluation score with the evaluation score of the FF heuristic (Hoffmann and Nebel 2001) in Figure 2. The FF heuristic has no access to domain-specific knowledge and shows a lower performance as expected.

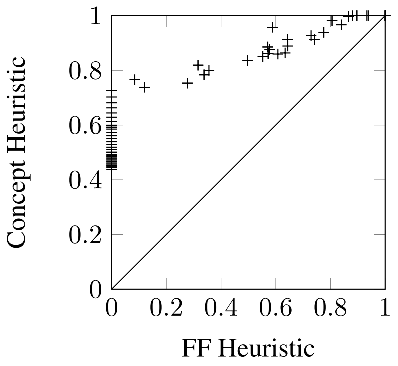

Figure 2: Evaluation scores of eager greedy search with concept heuristic and the FF heuristic for VisitAll tasks.

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.

As the state spaces of small Logistics tasks are already too large, we had to use a sample of the state spaces for the learning phase. The learning phase used 9 tasks of which three had 2 packages and two each had 3, 4, and 5 packages. We then used all of the states from the tasks with 2 packages and 10%/5%/2% from the tasks 3/4/5 packages, respectively. The Logistics domain is the only domain where we had to use sampling to be able to learn a heuristic.

We used a concept that described all delivered packages to initialize the procedure and then iteratively analyzed the

[1]https://www.w3.org/TR/owl2-manchester-syntax/

[2]https://pypi.org/project/cplex/

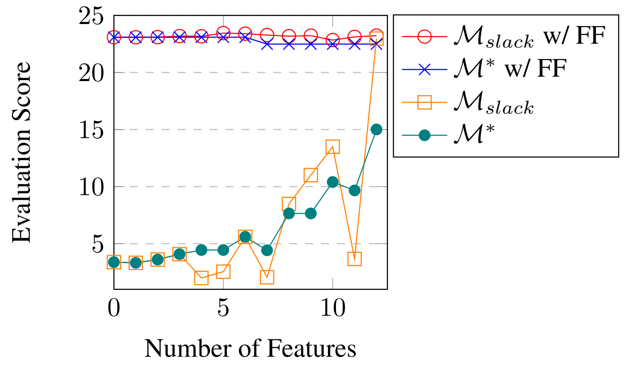

Figure 3: Evaluation scores of Logistics heuristics.

states where the heuristic required high slack. Similar to the domain VisitAll, this process helped to identify new features. The concepts we ended up with describe the situations a package can be in: from being on the ground in the wrong city to being on the way to the airport, in the plane, or in a truck in the correct city. These concepts directly correspond to the ones described by Francès et al. (2019) (P_1–P_{12} in their Table 1). It is worth pointing out that while Francès et al. describe an automated framework to discover these heuristics, it did not scale to the feature complexity required for Logistics so these features are also hand-crafted in their paper. Our tool adds support for this process and enables a user to discover the heuristic feature by feature.

We also attempted to start with the FF heuristic as a heuristic feature and use the reported states to add features until we found a dead-end avoiding and descending heuristic. We were unable to identify any set of features from which such a heuristic could be constructed except those we already mentioned above which are sufficiently expressive even without the FF heuristic.

While working with this domain, we noticed that adding features and thereby reducing the total slack does not always result in a better heuristic. To investigate this, we evaluate how the heuristic quality changes when adding features. Using the features we mentioned above, we measure the heuristic quality when just using the highest-weighted feature, just the two highest-weighted, and so on. For each set of features we learned weights with both $\mathcal{M}_{slack}$ and $\mathcal{M}^*$, once using just the features and once including the heuristic feature h^{FF}. The results are shown in Figure 3. We can clearly see how well FF performs on this domain. Additional features do not improve its performance significantly. For the configurations without the FF heuristic feature we can see a relatively steady increase in performance on average. That is, adding a useful feature will generally improve the heuristic. This is visible for both $\mathcal{M}_{slack}$ and $\mathcal{M}^*$. However, both models have cases where adding a feature harms the heuristic performance. This is more visible with $\mathcal{M}_{slack}$ where performance sharply drops when adding the second-to-last feature. There seem to be weight functions with low total slack that still guide the search in the wrong direction. Neither learning strategy dominates the other which suggests that there is value in both MIP variants.

TERMES

The TERMES domain models a robot that must build structures 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. It can place or pick up a block if the placed block is on the same height as the robot. A specific location is an infinite depot, where the robot can always pick up or deposit blocks.

To find a heuristic for TERMES, we started from the two simple features and then added features iteratively based on the reported states. The final heuristic we used has a total of 18 features added over 7 iterations:

1. Our initial set of features contains a feature counting the number of spaces with correct height ($|position \sqcap (height = height_G)|$) and a feature counting the number of spaces with missing blocks ($|(\exists too\_low.numb) \sqcap has\_block|$). In the second feature, the role too_low is again constructed from three named roles describing height, goal height, and the predecessor function. If the robot is holding a block (has_block), then the feature evaluates to the number of spaces where the height is lower than the goal, otherwise it is 0).[3]
2. We then added one feature to count the number of blocks with height too low ($\exists too\_low.numb$) and one feature counting the number of blocks with height too high ($\exists too\_high.numb$).
3. In the next iteration, we added three new features. These features count the number of spaces that are exactly one, two, or three blocks too low compared to the goal. These features mention specific cases because we do not know how to express the number of missing blocks at a location in a more general way. Hence, we encoded the specific cases for the most common cases.
4. We added six features that have a similar interpretation as the ones added in the previous iteration. These count the number of extra blocks or missing blocks compared to the maximal height a tower may require (either its goal height or the height it may require as a "ladder" to reach another tower).
5. We then added two distance features. The first feature encodes the idea that a robot carrying a block should move to a space where a block still needs to be placed. The second feature encodes that a robot not carrying a block should move a space where a block should be removed.
6. The next feature counts the number of towers which are the wrong height and cannot currently be reached by the

[3]We use the fact that nullary atoms have intepretation Δ when true in a state, as has_block is a nullary predicate in TERMES.

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 1: Evaluation scores of TERMES heuristics. For each method, the best evaluation score is highlighted.

robot. This should discourage the search from exploring such situations.

7. The last two features count the number of towers where a block should be added or removed and where a neighboring tower of the appropriate height to remove or add the block exists.

The training set used contained the four smallest TERMES tasks used in the IPC 2018. These tasks had a maximum height of three and, because of that, we encoded the features up to three extra/missing blocks from the goal height.

Table 1 shows the evaluation score of all methods at each iteration when we added new features. While adding some features allows a significant improvement over blind search and a modest improvement over the FF heuristic, adding more eventually diminishes the performance of all methods. The methods using FF as a heuristic feature become worse than the FF heuristic itself. The one without the heuristic feature become worse than a blind search, not solving a single task within the time limit. In both cases, the heuristic found by $\mathcal{M}_{\text{slack}}$ deteriorates quicker than the one found by $\mathcal{M}^*$. Our assumption that the discovered weights would disable a misleading feature turned out to be false in this case.

One of the reasons for this behavior is because our training set is not representative enough. In our training instances, the final action of any plan is never to place a block. This adds bias to the learning and affects the performance on heuristics that do not follow this structure, leading to an uninformed heuristic in tasks that are not represented in the training set.

Both $\mathcal{M}_{slack}$ and $\mathcal{M}^*$ perform worse when adding more features beyond a certain point. This suggests that they either do not approximate the right kind of heuristic, or that the approximation is not useful to improve performance. Wilt and Ruml (2016) showed that simply approximating h^* is not a guarantee for improved performance in sub-optimal planning, even if it is approximated very closely. In fact, for both MIP methods, the total slack value did generally decrease with more features, but in both cases there is still considerable room for improvement. This is shown in Figure 4. After we added the 18th feature, CPLEX also reached the 30 minutes time limit for all configurations except for $\mathcal{M}_{slack}$ without the FF heuristic feature.

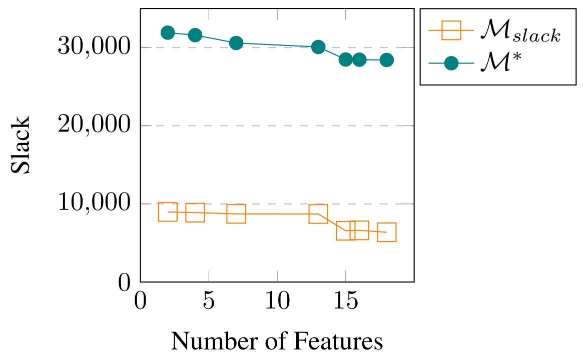

Figure 4: Total slack value of TERMES heuristics found by $\mathcal{M}_{slack}$ and $\mathcal{M}^*$.

Overall, the TERMES domain is much more challenging than the previous two. It shows that the choice of training instances plays an important role in the learning step. A way to improve performance could be to generate the training tasks in a more targeted way.

Conclusion and Future Work

Our main contribution is a tool which allows a domain expert without expertise in automated planning to specify knowledge in the form of concept-based features. These features are then used to learn a generalized potential heuristic for the domain in question. The user does not need to specify a weight function or relationship between the features, as the learning process is automated and only depends on a set of input features and training instances. Concept-based features have the advantage of being interpretable and were already successfully used in generalized planning (Francès et al. 2019). We extended this language so higher-arity predicates do not have to be compiled away and the domain expert can use them in their natural form.

As an initial evaluation of the approach we tried the tool on three domains. The results on domains that have simple solution strategies like VisitAll and Logistics are encouraging. In those cases we saw that the reported states help to identify weak points in the feature pool and fix them by adding new features. In a more complex domain like TERMES, we could not replicate this and saw that there are cases were adding a feature harms the heuristic.

One potential way to make more of the hand-crafted set of features is to augment it with systematically generated features as used by Francès et al. (2019). 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 did. Additionally, there are many possibilities to improve the learning part of this system. We saw that solving the MIP sometimes became the bottleneck of this system. A different relaxation of the property we are trying to learn might speed up the process or learn more useful heuristics from the same data. Rovner (2020) presents a MIP based on the Goal Distance Rank Correlation (Wilt and Ruml 2016) in the setting of potential heuristics for classical planning which would be an interesting choice. Alternatively, one could attempt to learn a heuristic which explicitly maximizes the Goal Distance Rank Correlation using a neural network. The downside to this approach would be a significant loss of the explainability of the heuristic.

Although we focused on classical planning, our tool could be extended to other fragments of planning. Our methods can deal with any planning fragments that use a first-order factored representation and that can be solved with explicit state-space search. For example, it should be possible to adapt the method to probabilistic planning (e.g., Trevizan, Thiébaux, and Haslum 2017), temporal planning (e.g., Fox and Long 2003), FOND planning (e.g., Erol, Hendler, and Nau 1994), and lifted planning (e.g., Corrêa et al. 2020). However, it has not yet been tested if generalized potential heuristics are effective in these areas.

Acknowledgments

This work was supported by the Swiss National Science Foundation (SNSF) as part of the project "Certified Correctness and Guaranteed Performance for Domain-Independent Planning" (CCGP-Plan). Moreover, this research was partially supported by TAILOR, a project funded by the EU Horizon 2020 research and innovation programme under grant agreement no. 952215. We thank Malte Helmert for his input on an earlier version of this work.

References

Baader, F.; Horrocks, I.; and Sattler, U. 2007. Description Logics. In van Harmelen, F.; Lifschitz, V.; and Porter, B., eds., *Handbook of Knowledge Representation*, chapter 3, 135–180. Elsevier.

Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence* 129(1): 5–33.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation using Query Optimization Techniques. In Beck, J. C.; Karpas, E.; and Sohrabi, S., eds., *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*, 80–89. AAAI Press.

de Graaff, R. 2020. Downward Guide. https://doi.org/10.5281/zenodo.3900929.

Doran, J. E.; and Michie, D. 1966. Experiments with the Graph Traverser program. *Proceedings of the Royal Society A* 294: 235–259.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN Planning: Complexity and Expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI 1994)*, 1123–1128. AAAI Press.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research* 20: 61–124.

Francès, G.; Corrêa, A. B.; Geissmann, C.; and Pommerening, F. 2019. Generalized Potential Heuristics for Classical Planning. In Kraus, S., ed., *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 5554–5561. IJCAI.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2): 100–107.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26: 191–246.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence* 173: 503–535.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14: 253–302.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online Planner Selection for Cost-Optimal Planning. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 57–64.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 21(2): 35–55.

Rovner, A. 2020. *Potential Heuristics for Satisficing Planning*. Master's thesis, University of Basel.

Seipp, J.; and Röger, G. 2018. Fast Downward Stone Soup 2018. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 80–82.

Trevizan, F. W.; Thiébaux, S.; and Haslum, P. 2017. Occupation Measure Heuristics for Probabilistic Planning. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 306–315. AAAI Press.

Wilt, C.; and Ruml, W. 2015. Building a Heuristic for Greedy Search. In Lelis, L.; and Stern, R., eds., *Proceedings of the Eighth Annual Symposium on Combinatorial Search (SoCS 2015)*, 131–139. AAAI Press.

Wilt, C.; and Ruml, W. 2016. Effective Heuristics for Suboptimal Best-First Search. *Journal of Artificial Intelligence Research* 57: 273–306.