Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19)

Generalized Potential Heuristics for Classical Planning

Guillem Francès, **Augusto B. Corrêa**, **Cedric Geissmann** and **Florian Pommerening**
University of Basel, Basel, Switzerland
{guillem.frances,augusto.blaascorrea,cedric.geissmann,florian.pommerening}@unibas.ch

Abstract

Generalized planning aims at computing solutions that work for all instances of the same domain. In this paper, we show that several interesting planning domains possess compact generalized heuristics that can guide a greedy search *in guaranteed polynomial time* to the goal, and which work for *any instance of the domain*. These heuristics are weighted sums of state features that capture the number of objects satisfying a certain first-order logic property in any given state. These features have a meaningful interpretation and generalize naturally to the whole domain. Additionally, we present an approach based on mixed integer linear programming to compute such heuristics automatically from the observation of small training instances. We develop two variations of the approach that progressively refine the heuristic as new states are encountered. We illustrate the approach empirically on a number of standard domains, where we show that the generated heuristics will correctly generalize to all possible instances.

1 Introduction

Domain-independent planners scale up nowadays to problems with very large state spaces, but the solutions they compute typically work only on a single problem. Generalized planning, in contrast, aims at computing more general solutions that work for potentially infinite classes of problems with similar underlying structure [Levesque, 2005; Srivastava *et al.*, 2008; Bonet *et al.*, 2009; Hu and De Giacomo, 2011]. Once a generalized solution is found, instantiating it for a concrete problem of the class usually requires little effort. A possible generalized solution for the problem of clearing a particular block b in Blocksworld, for instance, would somehow represent the idea that blocks above b need to be picked up and put away repeatedly until b is clear.

Recently, Bonet *et al.* [2019] have shown how to learn an abstraction over a first-order logical language from sample instances, and how to use a fully-observable non-deterministic planner to obtain a generalized policy from that abstraction [Bonet *et al.*, 2017]. Building on their work, we instead try to find generalized heuristics that capture greedy strategies in planning domains. For that, we use the same feature space as Bonet *et al.* [2019], where features encode how many objects in a state of the problem satisfy some property, such as "the number of blocks above block b". These features are defined in the first-order language of the domain description and are meaningful and easily interpretable [Fox *et al.*, 2017]. We show that simple linear combinations of these features exist in many domains which are *descending and dead-end avoiding* heuristics as defined by Seipp *et al.* [2016]. Such heuristics can be seen both as a *strong measures of progress* as defined by Parmar [2002], and as a generalization of *potential heuristics* [Pommerening *et al.*, 2015], and can be used in a greedy search to reach the goal in a polynomial number of steps. The key difference with standard potential heuristics is that our heuristics are instance-independent: once found, they can be used in any instance of the domain.

We start by providing some background and formalizing generalized potential heuristics (Sections 2–3). In Section 4 we demonstrate that generalized descending and dead-end avoiding potential heuristics exist for many standard domains. In Section 5 we show how to learn these heuristics automatically from small training instances using a mixed integer program (MIP). We present empirical results for a number of domains in Section 6, and conclude by discussing related work and the implications of this work in Sections 7 and 8.

2 Background

We start by introducing the formal basis and notation of planning tasks, generalized planning, and concept languages.

2.1 Classical Planning Formalism

We consider deterministic planning tasks that compactly represent a state space plus a first-order vocabulary with an interpretation over states. The state space of a task is a tuple $\langle S, L, T, s_I, S_G \rangle$, where S is a set of *states*, L a set of *labels*, $T \subseteq S \times L \times S$ the *transition relation*, $s_I \in S$ the *initial state*, and $S_G \subseteq S$ the set of *goal states*. It is deterministic: for a state s and label l there is at most one transition $\langle s, l, s' \rangle \in T$.

The first-order vocabulary of a task is a tuple $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P} \rangle$, where $\mathcal{V}$ is a set of variables, $\mathcal{F}$ a set of function symbols and $\mathcal{P}$ a set of predicate symbols. Each predicate and function symbol has an arity in $\mathbb{N}_0^+$. We assume w.l.o.g. that $\mathcal{P}$ contains only unary and binary predicates, and $\mathcal{F}$ contains only nullary functions, i.e., constants. Function symbols

with arity $n > 0$ can be compiled into $(n+1)$-ary predicate symbols and predicate symbols with arity $n > 2$ can be replaced by multiple binary predicates. Each state $s \in S$ in the planning task represents a first-order interpretation $\mathcal{I}(s)$ that assigns a truth value to any formula in the language induced by Σ. We write $s \models \varphi$ if formula φ is true under $\mathcal{I}(s)$.

Classical planning tasks represented in PDDL [McDermott, 2000] fit our definition by using the set of PDDL objects as constants and all possible sets of ground atoms as the set of states. More expressive formalisms such as Functional STRIPS [Geffner, 2000] also fit our definition after compiling away function symbols.

The set of *successors* of state s consists of all states s' such that $\langle s, l, s' \rangle \in T$ for some label l. A *path* between states s_0 and s_n is a sequence of labels $\langle l_1, \ldots, l_n \rangle$ such that there are states $s_1, \ldots, s_{n-1}$ with $\langle s_{i-1}, l_i, s_i \rangle \in T$ for $1 \leq i \leq n$. A *plan* is a path from the initial state s_I to some goal state. A state is called *reachable* if there is a path to it from s_I, and it is called *solvable* if there is a path from it to a goal state. A state is *alive* if it is solvable, reachable and not a goal state.

2.2 Generalized Planning

Generalized planning is interested in finding solutions that work for a whole class of planning tasks [Srivastava *et al.*, 2008]. We call a set of planning tasks all using the same set of predicate symbols a *generalized planning domain*. The planning tasks in a domain can still have different sets of constants and thus different state spaces. In PDDL, there is a syntactic distinction between objects that are used in all tasks of a domain and those that are used in one task only. While both are constants from a first-order logic point of view, we refer to the former as *PDDL-level constants* to distinguish them.

In Blocksworld, for instance, all instances use predicates *on*, *clear*, etc., but have different constants for the blocks being used. If Blocksworld was encoded with a constant representing the table, this constant would be a PDDL-level constant, while the ones representing blocks would not.

For the challenge of finding a general solution to a domain to be feasible, we assume that its instances share a common underlying structure.

2.3 Concept Languages

Concept Languages or description logics are a family of knowledge representation formalisms based on tractable subsets of first-order logic [Baader *et al.*, 2017]. They build on the notions of *concepts*, classes of objects that share some property, and *roles*, relations between these objects. Several description languages have been studied in the literature, each with different expressivity. We here use the standard language $\mathcal{SOI}$ with equality role-value-maps.

Syntax. Formally, *complex* concepts and roles are defined inductively from sets of given (application-dependent) *primitive* concepts and roles, which are, respectively, unary and binary first-order predicates. Any primitive concept is a concept, and any primitive role is a role. The *universal concept* $\top$ and the *bottom concept* $\bot$ are also concepts. Let C and C' be concepts, and R and R' roles. The *negation* $\neg C$, the *union* $C \sqcup C'$, the *intersection* $C \sqcap C'$, the *existential restriction* $\exists R.C$, the *universal restriction* $\forall R.C$, and the *role-value-map* $R = R'$ are also concepts, and if $a_1, \ldots a_n$ are constants, the *nominal* $\{a_1, \ldots a_n\}$ is a concept. The *inverse* role R^{-1}, the (non-reflexive) *transitive closure* role R^+, and the *composition* role $R \circ R'$ are also roles.

Semantics. The semantics of the above constructs are defined relative to a given *universe* Δ. A model $\cdot^{\mathcal{M}}$ maps each constant to an element of Δ, each primitive concept to a subset of Δ, and each primitive role to a subset of $\Delta \times \Delta$. The definition of $\mathcal{M}$ extends to complex concepts and roles:

$$
\begin{aligned}
\top^{\mathcal{M}} &= \Delta, \qquad \bot^{\mathcal{M}} = \emptyset, \\
(\neg C)^{\mathcal{M}} &= \Delta \setminus C^{\mathcal{M}}, \qquad \{a_1, \ldots a_n\}^{\mathcal{M}} = \{a_1^{\mathcal{M}}, \ldots a_n^{\mathcal{M}}\}, \\
(C \sqcup C')^{\mathcal{M}} &= C^{\mathcal{M}} \cup C'^{\mathcal{M}}, \qquad (C \sqcap C')^{\mathcal{M}} = C^{\mathcal{M}} \cap C'^{\mathcal{M}}, \\
(\exists R.C)^{\mathcal{M}} &= \{a \mid \exists b : (a,b) \in R^{\mathcal{M}} \wedge b \in C^{\mathcal{M}}\}, \\
(\forall R.C)^{\mathcal{M}} &= \{a \mid \forall b : (a,b) \in R^{\mathcal{M}} \rightarrow b \in C^{\mathcal{M}}\}, \\
(R = R')^{\mathcal{M}} &= \{a \mid \forall b : (a,b) \in R^{\mathcal{M}} \leftrightarrow (a,b) \in R'^{\mathcal{M}}\}, \\
(R^{-1})^{\mathcal{M}} &= \{(b,a) \mid (a,b) \in R^{\mathcal{M}}\}, \\
(R \circ R')^{\mathcal{M}} &= \{(a,c) \mid \exists b : (a,b) \in R^{\mathcal{M}} \wedge (b,c) \in R'^{\mathcal{M}}\}, \\
(R^+)^{\mathcal{M}} &= \{(a_0, a_n) \mid \exists a_1, \ldots, a_{n-1} : \\
&\qquad (a_{i-1}, a_i) \in R^{\mathcal{M}} \text{ for all } 1 \leq i \leq n\}.
\end{aligned}
$$

Complexity. We define the *complexity* $\mathcal{K}(X)$ of a concept or role X as the size of its syntax tree, e.g., $\mathcal{K}(\exists R.C) = 1 + \mathcal{K}(R) + \mathcal{K}(C)$. The complexity of $\top$ and $\bot$ is 0, and that of any primitive concept or role is 1.

2.4 Concept Languages in Planning Tasks

We limit ourselves to tasks whose goals are expressed as conjunctions of ground atoms, and whose predicates have arity at most 2. We take the unary predicates of the planning task to be primitive concepts, and its binary predicates to be primitive roles, in a universe that consists exactly of all constants in the task, which we consider to denote themselves. As a result, each state s of the planning task uniquely determines a model $\mathcal{M}(s)$ for the concept language of the task where $C^{\mathcal{M}(s)} = \{a \mid \mathcal{I}(s) \models C(a)\}$ for primitive concepts C, and $R^{\mathcal{M}(s)} = \{(a,b) \mid \mathcal{I}(s) \models R(a,b)\}$ for primitive roles R.

To ensure that concepts can be interpreted in all instances of a domain, we restrict nominal concepts to PDDL-level constants, which are by definition common to all instances. To support goal-oriented reasoning, we follow Martin and Geffner [2004] and introduce *goal concepts* C_G and *goal roles* R_G for every primitive concept C and role R that is used in the goal formula of the task. The denotation of these elements is fixed in all states s, and given by the atoms in the goal formula.

In the Blocksworld encoding discussed above, the table constant would give rise to a nominal concept $\{table\}$. The universe Δ would consist of all blocks plus the table. The concept *clear* would then represent the set of all blocks that are clear; the concept $\exists on.\{table\}$, all blocks that lie on the

table, and the concept $clear_G$, the set of all blocks $b \in \Delta$ such that $clear(b)$ is part of the goal conjunction.

3 Generalized Potential Heuristics

Potential heuristics, weighted sums over conjunctive state features, have enjoyed some recent success in classical planning [Pommerening *et al.*, 2015; Seipp *et al.*, 2015]. Here, we generalize the idea to features mapping states to integers.

Definition 1. *Let S be a set of states 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 state $s \in S$ is*

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

Note that S can contain states from multiple planning tasks and that a potential heuristic is well-defined on any state of any task of a domain if all its features are. Following Bonet *et al.* [2019], we use two types of features:

Definition 2. *Let C and C' be concepts and R a role from the concept language of a planning domain, and s a state of some task in the domain. The value of the* cardinality feature $|C|$ *in s is $|C^{\mathcal{M}(s)}|$. The value of the* distance feature *dist(C, R, C') is* $\min\{n \mid \exists a_0, \ldots, a_n$ *s.t.* $a_0 \in C^{\mathcal{M}(s)}, a_n \in C'^{\mathcal{M}(s)}$, *and* $(a_{i-1}, a_i) \in R^{\mathcal{M}(s)}$ *for* $1 \leq i \leq n\}$. *When no such R-path exists between the denotations of concepts C and C', we define the value of dist(C, R, C') to be zero.*

For example, feature $|\exists carry.\top|$ maps each state in Gripper to the number of carried balls. In the VisitAll domain, *dist*$(at\text{-}robot, connected, place \sqcap \neg visited)$ maps to the smallest distance between the robot and an unvisited place.

Potential heuristics with concept-based features compactly represent a heuristic function for the whole domain. However, not all such heuristics are useful. We now investigate a useful subclass with strong theoretical guarantees.

3.1 Descending and Dead-End Avoiding Heuristics

Seipp *et al.* [2016] study descending and dead-end avoiding heuristics. A heuristic function h is *descending* if every alive state s has a successor $s' \in succ(s)$ with lower heuristic value $h(s') < h(s)$, and it is *dead-end avoiding* if any successor $s' \in succ(s)$ of an alive state s such that $h(s') < h(s)$ is solvable. Descending, dead-end avoiding heuristics are relevant because they guide standard greedy algorithms such as hill-climbing or greedy best-first search to a goal state (if possible) or detect that the task is unsolvable (if not) without backtracking. If the heuristic values are integers, the number of steps the algorithms take is bounded by the difference between the heuristic value of the initial state and the smallest heuristic value [Seipp *et al.*, 2016]. In our case, heuristic values can be real numbers but it is easy to see that the result generalizes for steepest ascent hill-climbing with a slightly stricter definition of *descending* heuristics: for our heuristics, we require that every alive state s has a successor s' with $h(s') + 1 \leq h(s)$. Steepest ascent hill-climbing improves the heuristic value by at least 1 in each step and thus has the same guarantee as with integer-valued heuristics.

The values of our features as defined above are always upper-bounded by $|\Delta|$. Since heuristic values are linear combinations of such values and the weights are fixed for the whole domain, all heuristic values are in $O(|\Delta|)$. The difference between two heuristic values and thus the number of steps a greedy algorithm takes are also limited by $O(|\Delta|)$. Thus, a potential heuristic using concept-based features that is descending and dead-end avoiding on all instances of a domain can be used to solve every instance in polynomial time.

3.2 Limitations

While there are interesting domains that can be solved with concept-based potential functions, there are also some theoretical limitations of the approach. First, since all heuristic values are finite, we cannot represent infinite heuristic values and thus cannot detect unsolvable instances. Despite this, unsolvable states can be avoided as long as the initial state is solvable, as we demonstrate in the domain Spanner below. Next, the functions that can be expressed depend on the predicates available in the domain. If a relation between objects is implicitly encoded (e.g., in the available actions), we cannot express it. We will see this limitation in the domain Gripper below. Finally, the downside of guaranteeing that greedy algorithms run in $O(|\Delta|)$ is that the plans we discover must have a length in $O(|\Delta|)$. With sufficiently high weights, a subset of tasks might be solved, but we cannot hope to find a general solution in domains with super-linear plans, as we will see in the domain VisitAll below.

4 Examples

We now show examples of descending and dead-end avoiding potential functions for several classical planning domains. For simplicity, all used concepts and their intended meanings are listed in Table 1. The somewhat tedious proofs that the described functions are indeed descending and dead-end avoiding can be found in a technical report [Francès *et al.*, 2019].

Spanner

In the domain Spanner, an agent has to pick up spanners on its way to a gate with nuts that have to be tightened. Spanners can only be used once, and the path to the gate is one-way. In the standard encoding, the function

$$|S_1| + |S_2| + dist(S_3, link, S_4) + 2|S_5|$$

is descending and dead-end avoiding. Its value decreases when spanners are picked up (S_1), nuts are tightened (S_2), or the agent approaches the gate ($dist(S_3, link, S_4)$). Unsolvable states are avoided through a penalty for leaving a position before picking up all spanners (S_5). Interestingly, there is an alternative encoding with no distance features that uses a cardinality feature counting the cells behind the agent.

Gripper

In the domain Gripper, a robot with two grippers has to transport some balls between two rooms. We use a generalization of the standard domain with several rooms, robots, and grippers per robot. The goal remains to transport all balls into a certain room. The domain is dead-end-free, and the following function is descending:

$$8|G_1| + 4|G_2| - 2|G_3| - |G_4|$$

		Spanner	$\mathcal{K}(C)$
S_1	$spanner \sqcap (\exists at.\top)$	Spanners which have not been picked up	4
S_2	$nut \sqcap (\neg tightened)$	Untightened nuts	4
S_3	$\exists at^{-1}.man$	Locations occupied by some agent	4
S_4	$\exists at^{-1}.S_2$	Locations with an untightened nut	7
S_5	$spanner \sqcap \exists at.\exists (link^+)^{-1}.S_3$	Spanners in a cell no longer reachable by any agent	12
		Gripper	$\mathcal{K}(C)$
G_1	$\neg(at = at_G) \sqcap \exists at.\top$	Balls that are in a room different from the target room	7
G_2	$\exists carry.\top$	Balls carried by some robot	2
G_3	$\exists carry.\exists gripper.\exists at\text{-}robby.\exists at_G^{-1}.\top$	Balls carried by some robot while robot is in the target room	9
G_4	$\exists at\text{-}robby.((\exists at^{-1}.\top) \sqcap \neg(\exists at_G^{-1}.\top))$	Robots in non-empty rooms other than the target room	10
		Blocksworld	$\mathcal{K}(C)$
B_1	$ontable_G \sqcap ontable$	Blocks that are correctly placed on the table	3
B_2	$(\exists on_G.\top) \sqcap (on = on_G)$	Blocks that are placed on their target block	6
B_3	$\neg(ontable_G \sqcup \exists on_G.\top)$	Blocks that are not mentioned in the goal	5
B_4	$B_1 \sqcup B_2 \sqcup B_3$	Blocks where the block (or table) below is consistent with the goal	16
B_5	$\forall on_G^{-1}.(on = on_G)$	Blocks where the block above is consistent with the goal	6
B_6	$B_4 \sqcap \forall on^+.(B_4 \sqcap B_5)$	Blocks that are well-placed	43
B_7	$holding \sqcap \exists on_G.(clear \sqcap B_6)$	Blocks held while their target block is clear and well-placed	49
		VisitAll	$\mathcal{K}(C)$
V_1	$place \sqcap \neg visited$	Places not yet visited	4
		Logistics	$\mathcal{K}(C)$
L_1	$package \sqcap (at \circ in\text{-}city = at_G \circ in\text{-}city)$	Packages on the ground in the correct city	9
L_2	$package \sqcap (in \circ at \circ in\text{-}city = at_G \circ in\text{-}city)$	Packages in a vehicle in the correct city	11
L_3	$package \sqcap \neg(at \circ in\text{-}city = at_G \circ in\text{-}city)$	Packages on the ground in the wrong city	10
L_4	$package \sqcap \neg(in \circ at \circ in\text{-}city = at_G \circ in\text{-}city)$	Packages in a vehicle in the wrong city	12
P_1	$package \sqcap (\exists in.truck) \sqcap (in \circ at = at_G)$	Packages in a truck at the right location for that package	11
P_2	$L_2 \sqcap (\exists in.truck) \sqcap \neg(in \circ at = at_G)$	Packages in a truck in the right city but wrong location	22
P_3	$L_1 \sqcap (\exists at.\exists at^{-1}.truck) \sqcap \neg(at = at_G)$	Packages on the ground in the right city with an available truck	21
P_4	$L_1 \sqcap (\exists at.\neg\exists at^{-1}.truck) \sqcap \neg(at = at_G)$	Packages on the ground in the right city without an available truck	22
P_5	$L_2 \sqcap (\exists in.airplane)$	Packages in an airplane in the right city	15
P_6	$L_4 \sqcap (\exists in.airplane)$	Packages in an airplane in the wrong city	18
P_7	$L_3 \sqcap (\exists at.\exists at^{-1}.airplane)$	Packages on the ground in an airport of the wrong city with an available plane	17
P_8	$L_3 \sqcap (\exists at.(airport \sqcap \neg\exists at^{-1}.airplane))$	Packages on the ground in an airport of the wrong city without an available plane	20
P_9	$L_4 \sqcap (\exists in.(truck \sqcap \exists at.airport))$	Packages in a truck at the airport of the wrong city	20
P_{10}	$L_4 \sqcap (\exists in.\exists at.\neg airport)$	Packages in a truck at a non-airport location of the wrong city	19
P_{11}	$L_3 \sqcap (\exists at.\neg airport \sqcap \exists at^{-1}.truck)$	Packages on the ground of a non-airport location of the wrong city with an available truck	20
P_{12}	$L_3 \sqcap (\exists at.\neg airport \sqcap \neg\exists at^{-1}.truck)$	Packages on the ground of a non-airport location of the wrong city without an available truck	21

Table 1: Useful concepts and their complexities for selected classical planning domains.

Interestingly, in the default encoding no predicate relates robots to grippers, as these are all *implicitly* assumed to belong to the only robot. Hence, we cannot express a concept like G_3 that considers the carried balls and the location of the robot at the same time. This could be addressed with a simple extension to our concept language, but there is a trade-off between expressivity and tractability of the set of features.

Blocksworld

In the domain Blocksworld, the goal is to stack and unstack blocks to achieve a target configuration. In contrast to Seipp *et al.* [2016], we can represent the notion of *well-placed blocks* [Slaney and Thiébaux, 2001] and thus do not require exponential and instance-dependent weights:

$$-4|B_6| - |holding| - 2|ontable| - 2|B_7|$$

The function rewards placing a block in its final position (B_6), unstacking a misplaced block (*holding*), dropping it (*ontable*) and picking it back up once its target is available (B_7).

VisitAll

In the domain VisitAll, an agent has to visit all nodes of an undirected graph. For graphs with a diameter of at most k, the following function is descending:

$$k|V_1| + dist(at\text{-}robot, connected, V_1)$$

Moving closer to the closest unvisited location reduces the value by 1. Reaching it decreases the cardinality feature by 1 but can increase the distance feature by up to k.

VisitAll shows the limitation to plans with a linear length: to generalize to all graphs, the weight k would have to depend on the instance. A possible solution is to allow more general features, for example one that multiplies the cardinalities of two concepts. Replacing $k|V_1|$ with $|V_1 \times place|$ would yield a potential function that is descending in all VisitAll instances.

Logistics

In the domain Logistics, packages need to be transported between different locations in different cities. Each city has a single airport, trucks move within cities, and planes between airports. The heuristic $\sum_{i=1}^{12} i|P_i|$ is descending: concepts P_i encode mutually exclusive states that a package can be in and there is always an action that only moves packages from concepts P_i to concepts P_j with $j \leq i$. Logistics is also representative for other domains, e.g., the domain Miconic that

controls an elevator to deliver passengers can be seen as a Logistics domain with a single city and truck.

5 Learning the Heuristics from Sample Tasks

As we have seen, simple descending and dead-end avoiding potential heuristics exist for several standard domains. We next propose a way to learn them automatically. Our approach takes as input a set $\mathcal{F}$ of *candidate features* and a set of states $\mathcal{S}$. It then computes a generalized potential heuristic with features from $\mathcal{F}$ that is descending and dead-end avoiding *over all states in* $\mathcal{S}$, and further minimizes a measure of heuristic complexity. Following Bonet *et al.* [2019], for $\mathcal{F}$ we consider all concept-based features up to a certain complexity, where the complexity $\mathcal{K}(f)$ of a feature f is the total complexity of its concepts and roles (see Section 2.3). In turn, for $\mathcal{S}$ we consider all reachable states from a few *training instances*, which we assume small enough so that their reachable state space can be fully expanded. States in $\mathcal{S}$ can thus be labeled with information on whether they are alive or unsolvable. We denote by $\mathcal{S}_A$ the subset of alive states in $\mathcal{S}$, and by $\mathcal{T}$ the set of transitions starting in an alive state.

From $\mathcal{S}$ and $\mathcal{F}$ we define the mixed integer linear program $\mathcal{M}(\mathcal{S}, \mathcal{F})$, with variables $w_f \in \mathbb{Z}$ for each $f \in \mathcal{F}$, as follows:

$$\min_{w} \sum_{f \in \mathcal{F}} [w_f \neq 0] \mathcal{K}(f) \quad \text{subject to}$$

$$\bigvee_{s' \in succ(s)} h(s') + 1 \leq h(s) \qquad \text{for } s \in \mathcal{S}_A \tag{1}$$

$$h(s') \geq h(s) \qquad \text{for } (s, s') \in \mathcal{T}, s' \text{ unsolvable}, \tag{2}$$

where $h(s)$ is used as shorthand for the linear expression $\sum_{f \in \mathcal{F}} w_f f(s)$. Note that $f(s)$ here are integer constants. Constraints (1) encode the condition of being descending, while constraints (2) encode that of being dead-end avoiding. Because simpler features can be expected to generalize better, we minimize the total complexity of selected features.

In order to deal with the nonlinearity of constraints (1), we use *indicator constraints*. Full details can be found in the technical report [Francès *et al.*, 2019]. The resulting MIP has $O(|\mathcal{F}| + |\mathcal{T}|)$ variables and $O(|\mathcal{F}| + |\mathcal{T}|)$ constraints.

Theorem 1. *Any solution to the MIP $\mathcal{M}(\mathcal{S}, \mathcal{F})$ induces a generalized potential heuristic with features from $\mathcal{F}$ that is descending and dead-end avoiding on $\mathcal{S}$.*

Many weights of such a solution will typically be zero, in which case we consider the corresponding feature as unused.

5.1 Incremental Constraint Generation

The above MIP $\mathcal{M}(\mathcal{S}, \mathcal{F})$ has size linear in $\mathcal{S}$, the number of reachable states of the training instances, which somewhat limits the approach. Often, however, many of the states in $\mathcal{S}$ are intuitively redundant; for our approach to work, it would be enough to have a few states that are *representative* of the whole domain dynamics. We can not easily know in advance which states are representative, but we can use the above observation to generate the constraints in $\mathcal{M}(\mathcal{S}, \mathcal{F})$ incrementally within a *constraint generation loop*.

For this, we assume that $\mathcal{S}$ contains labeled states from fully-explored training instances. We randomly sample a subset of states $\mathcal{S}^0 \subseteq \mathcal{S}$. The solutions to the MIP $\mathcal{M}(\mathcal{S}^0, \mathcal{F})$ induce generalized potential heuristics h that are descending and dead-end avoiding *over the states in* $\mathcal{S}^0$. We can check if h is also descending and dead-end avoiding over the rest of states in $\mathcal{S}$. If it is, we are done. Otherwise, we put the first χ states where it is not into a set of *flaws* $\mathcal{S}^0_\chi$, and restart the process by solving the MIP $\mathcal{M}(\mathcal{S}^1, \mathcal{F})$, where $\mathcal{S}^1 = \mathcal{S}^0 \cup \mathcal{S}^0_\chi$. Since the size of $\mathcal{S}^i$ increases monotonically with i, the process eventually converges to the computation of $\mathcal{M}(\mathcal{S}, \mathcal{F})$, but often we find a suitable heuristic before that. If $\mathcal{M}(\mathcal{S}^i, \mathcal{F})$ turns out to be unsolvable at some iteration i, then no generalized heuristic with the properties we seek exist *when using only features from* $\mathcal{F}$, in which case we can increase the complexity bound used to generate the set $\mathcal{F}$ and start over.

5.2 Refinement of the Heuristic for Unseen States

Assume that we have a generalized potential heuristic h that is descending and dead-end avoiding over all states $\mathcal{S}$ in our training set, computed with the MIP $\mathcal{M}(\mathcal{S}, \mathcal{F})$. It might of course be the case that h does not generalize to other instances of the generalized problem. This will be quickly apparent, since a (steepest-ascent) hill-climbing search will reach a local minimum $s \notin S_G$ in polynomial time. When that happens, we would like to refine h to make it behave correctly in states like s, but we will typically not have information on which of the states in the instance are unsolvable. If the path followed by the hill-climbing search is $s_I = s_0, s_1, \ldots, s_n = s$, however, the reason for the heuristic failure must be one of the following:

(a) the problem is unsolvable,

(b) for some pair (s_i, s_{i+1}), s_i is solvable, s_{i+1} is not, and $h(s_{i+1}) < h(s_i)$,

(c) s is solvable, but none of its successors has smaller h-value.

We here assume that we deal with solvable instances only, since we view detecting unsolvability as an orthogonal problem.[1] In that case, upon finding a local minimum we can refine h by adding to the original MIP $\mathcal{M}(\mathcal{S}, \mathcal{F})$ the following *refinement constraint* that tackles cases (b) and (c):

$$\left(\bigvee_{i=0}^{n-1} h(s_i) \leq h(s_{i+1}) \right) \vee \left(\bigvee_{s' \in succ(s)} h(s') + 1 \leq h(s) \right),$$

using again indicator constraints to represent the disjunction.

If the new MIP has a solution, we can then restart the search from s_I. When new flaws are found, we add refinement constraints for them iteratively until we are able to reach a goal state. Otherwise, if the MIP has no solution, then the set $\mathcal{F}$ of candidate features is not expressive enough to represent a heuristic that generalizes to the instance. Again, we can restart the process with a larger set of candidate concepts.

[1] A way to represent both finite and infinite heuristic values with finite potential heuristics is to use two functions: one that represents the finite values and one that recognizes unsolvable states [Corrêa and Pommerening, 2019]. Different methods are useful for these two cases, and here we only focus on the first one.

	G	M	S	V
# of training instances	8	12	11	9
# of iterations	2.0	2.7	1.0	1.7
$\|\mathcal{F}\|$	469	2105	904	330
# of MIP variables	2017	7273	3381	1039
# of MIP constraints	2238	7331	3370	1190
Complexity of h	8 (18)	6 (14)	8 (20)	5 (8)
# of features in h	5	4	5	3
Total time	8h	32m	178s	87s
Total MIP time	7.4h	26m	6.8s	2.1s

Table 2: Results of the incremental constraint generation approach on Gripper (**G**), Miconic (**M**), Spanner (**S**) and VisitAll (**V**). Number of iterations of the constraint generation loop, MIP size and running times averaged over 3 runs. Heuristic complexity k_1 (k_2) denotes heuristic with maximum complexity feature k_1 and total aggregated complexity k_2. "Number of features" stands for the number of non-zero weights in the function h. MIP dimensions and size $|\mathcal{F}|$ of the pool of candidate features reported for the last iteration. Total MIP time aggregates MIP solution times on each iteration.

6 Empirical Results

We have implemented a prototype of the above approach in Python, building on the feature generator by Bonet *et al.* [2019] and on the Pyperplan planner.[2] The experiments below run on Intel Xeon E3-1275 CPUs using CPLEX v12.8 as a MIP solver. No run used more than 6 GB of memory. Source code and benchmarks are available online.[3]

We use the standard encodings of Miconic, Spanner, and VisitAll, and a generalization of Gripper to an arbitrary number of robots, grippers, and rooms, as described in Section 4. For each domain, we use training instances with a (reachable) state space of at most 15000 states. To keep the set of candidate concepts small, we limit role composition to primitive roles. In our experiments, allowing nested composition increased the number of concepts and the chance of overfitting without resulting in simpler heuristics. In all the experiments we use the incremental method described in Section 5.1, as it showed consistently better performance. We set a maximum complexity of $k = 8$ when generating cardinality features and of $k = 5$ for distance features. The initial set of states $\mathcal{S}_0$ has 100 randomly sampled states along with their successors, plus all states in an arbitrary optimal plan. The generation of candidate features takes a few minutes in all cases.

Our main result is that for the four domains above, our approach finds a potential heuristic that not only is descending and dead-end avoiding over all training instances, but which can be (manually) proven to generalize to all instances of the class, which means that when used within a steepest-ascent hill-climbing, it will be able to solve any such instance in polynomial time. The only exception is VisitAll, where we empirically confirm what we discussed in Section 4: we can only generalize to instances with graph diameter bounded by the largest diameter used in the training instances.

Table 2 gives an overview of the whole learning process. In the four domains, the constraint generation loop converges quickly to a final solution, never requiring more than 3 iterations. The number of features in the set of candidate features $\mathcal{F}$ varies significantly, being a function of the number of unary and binary predicates as well as the logical structure of the problem encoding. In terms of runtime, Gripper by far requires the most time, most of which is spent solving the MIP in each of the two iterations. We have not further investigated the reason why CPLEX takes so long in this case. VisitAll is the only domain where distance features appear necessary.

Some of the computed heuristic functions have lower complexity than the ones discussed in Section 4, but they capture similar greedy mechanisms. In Spanner, for instance, representing the distance to the gate turns out to be simpler by using a transitive closure that counts the number of empty cells behind the agent than by using a distance feature. To illustrate, the following is the heuristic computed automatically from 12 small training instances in the Miconic domain:

$$\begin{aligned} &- 3|\mathit{boarded}| - 5|\mathit{served}| \\ &+ 2|(\forall \mathit{origin}^{-1}.\mathit{boarded}) \sqcap \mathit{lift\text{-}at}| \\ &+ 1|(\forall \mathit{destin}^{-1}.\mathit{served}) \sqcap \mathit{lift\text{-}at}|. \end{aligned}$$

The first two features reward boarding passengers into the elevator and serving them to their destination. The third rewards the elevator for moving away from a floor F once it has boarded all passengers that start their trip in F, and the last rewards the elevator for moving away from F once it has served all passengers that want to go to F. The function differs from the idea we presented for the more general Logistics domain in Section 4, as the concepts here do not all represent mutually exclusive sets of passengers. This illustrates a general trend with the heuristics computed by our method: they often differ from the ones we had in mind and have lower overall complexity. Additionally, different MIP solutions inducing different generalized heuristics, all with equal overall complexity, can be found by our method.

Our prototype is not yet able to scale up to domains such as Blocksworld or Logistics. Although there exist descending and dead-end avoiding generalized potential heuristics for these domains (Section 4), the complexity of some of their features seems too high. For example, the number of features below the complexity bound of 49 in Blocksworld is in the order of 2 million. While this is out of scope now, solving a MIP with millions of variables is conceivably possible with sufficient computing resources. Better methods of feature selection can also help our method to scale up more robustly.

7 Related Work

Our work relates to a number of previous research threads. Parmar [2002] studies *measures of progress* from the first-order framework of situation calculus. A predicate p is a measure of progress if in every possible state there is some action that monotonically increases the extension of p. If such a predicate can be succinctly represented, then a policy that always selects a p-increasing action is an efficient planning algorithm. Descending potential heuristics are measures of progress in dead-end-free domains. Parmar introduces measures of progress for different types of domains. In particular,

[2] https://bitbucket.org/malte/pyperplan/

[3] https://doi.org/10.5281/zenodo.3236083

tiered measures of progress underlie our heuristic for Logistics. Her discussion is limited to dead-end free domains, and leaves open the question of how to find measures of progress automatically, which is the focus of our work. This question is also tackled by Yoon *et al.* [2005], who use first-order features based on taxonomic syntax to *approximate* measures of progress. In their case, these measures are lexicographic ensembles of heuristics which they learn greedily from a number of *sample plans*. This is similar in spirit to our work, but we are interested in learning *exact* measures of progress over the whole state space of the problem, not over some sample trajectories that need to be externally chosen, and we aim at minimizing the complexity of the learnt heuristic, which should help generalization. When our approach succeeds, we have learnt a (provably correct) polynomial search algorithm for all instances of the planning domain at hand.

Seipp *et al.* [2016] study descending, dead-end avoiding potential heuristics over conjunctive state features, but the features and weights in their heuristics are instance-dependent, and they do not address the question of how to obtain these heuristics automatically. Our contribution can be seen as a possible answer to such a question based on first-order features and learning from observing small instances.

Concept languages have been used to learn classical planning policies before. Martin and Geffner [2004] learn simple rule-based policies in the form of lists of concept-action pairs, with a focus on capturing optimal behavior in Blocksworld. The taxonomic syntax used by Yoon *et al.* [2005], similar to concept languages, is also used in other works by the same authors [Yoon *et al.*, 2006; Yoon *et al.*, 2008].

This work is also related to recent research in generalized planning [Levesque, 2005; Srivastava *et al.*, 2008; Hu and De Giacomo, 2011; Bonet and Geffner, 2018], in particular to the approach taken by Bonet *et al.* [2019], who learn a generalized abstraction from training instances based on the same type of first-order features that we use here. Their work however requires that the generalized problem can be abstracted into a Qualitative Numerical Problem (QNP), where domain dynamics need to be expressible through (indeterminate) increments and decrements of numerical variables X and preconditions of the form $X > 0$ and $X = 0$. This inductively-learned QNP is then mapped into a fully-observable non-deterministic problem to obtain a generalized policy with certain correctness guarantees [Bonet and Geffner, 2018]. Our approach, which is in contrast more direct, can reason with precise quantitative change (e.g., favor some transition iff feature f_i increases more than f_2), and can handle domains such as Spanner, which do not have an obvious QNP abstraction. On the other hand, we cannot capture things like looping policies, which Bonet *et al.* can.

Finally, our work is also related to the line of research by Toyer *et al.* [2018], who use a neural network-based approach for computing generalized policies for stochastic planning domains. A major difference with the work that we present here is that the logical grounding of our approach lends itself to better interpretability and, more relevantly, to the possibility of reasoning, manually or automatically, about the theoretical correctness of the heuristics that we learn.

8 Discussion

We have presented a way to automatically compute descending and dead-end avoiding potential heuristics that are well-defined on all instances of a generalized planning domain. The presented approach of course has a number of limitations, the most obvious being that not all problems have a greedy solution strategy that can be expressed as a compact generalized potential heuristic. Because of our choice of features, heuristic values are linear in the number of objects, and so is the length of the plans we can compute, which leaves out many interesting problems that do not have plans of that type. Our work can be seen however as a preliminary step towards the automatic synthesis of solvers with polynomial performance guarantees over all instances of a generalized planning problem. An important next step for this would be to deductively prove the correctness of the computed heuristics from the domain model, in the spirit of [Levesque, 2005].

The heuristic functions that we defined manually are easily interpretable by a human. The ones that our approach discovers automatically are sometimes slightly less obvious. It is not clear how to optimize for interpretability, but in general it seems that optimizing the total complexity of the heuristic yields heuristics which are not too hard to interpret. The extent to which these heuristics and their interpretability relate to the logical invariants that are implicit in the domain representation would deserve further examination.

We have also seen a trade-off in the choice of the underlying concept language and of the complexity bounds. More expressive concepts allow some properties to be represented more concisely, but a richer language aggravates the combinatorial explosion of the set of candidate features to be explored and increases the size of the mixed integer linear program that we use. At the same time, for a fixed set of training instances, increasing the allowed complexity of concepts increases the chances of overfitting. Such problems can be mitigated by using our incremental constraint generation approach, which can increase the complexity bound progressively as the current bound is proven too low to contain a heuristic with the desired properties. An interesting research direction would be to devise techniques to explore the pool of candidate features more selectively.

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). The authors thank Blai Bonet for his contribution to the codebase used in the experiments.

References

[Baader *et al.*, 2017] Franz Baader, Ian Horrocks, Carsten Lutz, and Uli Sattler. *Introduction to Description Logic*. Cambridge University Press, 2017.

[Bonet and Geffner, 2018] Blai Bonet and Héctor Geffner. Features, projections, and representation change for generalized planning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, pages 4667–4673. IJCAI, 2018.

[Bonet *et al.*, 2009] Blai Bonet, Héctor Palacios, and Héctor Geffner. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 34–41. AAAI Press, 2009.

[Bonet *et al.*, 2017] Blai Bonet, Giuseppe De Giacomo, Héctor Geffner, and Sasha Rubin. Generalized planning: Non-deterministic abstractions and trajectory constraints. In Carles Sierra, editor, *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, pages 873–879. IJCAI, 2017.

[Bonet *et al.*, 2019] 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)*. AAAI Press, 2019.

[Corrêa and Pommerening, 2019] Augusto B. Corrêa and Florian Pommerening. An empirical study of perfect potential heuristics. In Nir Lipovetzky, Eva Onaindia, and David E. Smith, editors, *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*. AAAI Press, 2019.

[Fox *et al.*, 2017] Maria Fox, Derek Long, and Daniele Magazzeni. Explainable planning. In *IJCAI-17 Workshop on Explainable AI (XAI)*, pages 24–30, 2017.

[Francès *et al.*, 2019] Guillem Francès, Augusto B. Corrêa, Cedric Geissmann, and Florian Pommerening. Generalized potential heuristics for classical planning: Additional material. Technical Report CS-2019-003, University of Basel, Department of Mathematics and Computer Science, 2019.

[Geffner, 2000] Héctor Geffner. Functional Strips: A more flexible language for planning and problem solving. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, volume 597 of *Kluwer International Series In Engineering And Computer Science*, chapter 9, pages 187–209. Kluwer, Dordrecht, 2000.

[Hu and De Giacomo, 2011] Yuxiao Hu and Giuseppe De Giacomo. Generalized planning: Synthesizing plans that work for multiple environments. In Toby Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 918–923. AAAI Press, 2011.

[Levesque, 2005] Hector Levesque. Planning with loops. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 509–515. Professional Book Center, 2005.

[Martin and Geffner, 2004] Mario Martin and Héctor Geffner. Learning generalized policies from planning examples using concept languages. *Applied Intelligence*, 20(1):9–19, 2004.

[McDermott, 2000] Drew McDermott. The 1998 AI Planning Systems competition. *AI Magazine*, 21(2):35–55, 2000.

[Parmar, 2002] Aarati Parmar. A logical measure of progress for planning. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI 2002)*, pages 498–505. AAAI Press, 2002.

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

[Seipp *et al.*, 2015] Jendrik Seipp, Florian Pommerening, and Malte Helmert. New optimization functions for potential heuristics. In Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein, editors, *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, pages 193–201. AAAI Press, 2015.

[Seipp *et al.*, 2016] Jendrik Seipp, Florian Pommerening, Gabriele Röger, and Malte Helmert. Correlation complexity of classical planning domains. In Subbarao Kambhampati, editor, *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pages 3242–3250. AAAI Press, 2016.

[Slaney and Thiébaux, 2001] John Slaney and Sylvie Thiébaux. Blocks World revisited. *Artificial Intelligence*, 125(1–2):119–153, 2001.

[Srivastava *et al.*, 2008] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. Learning generalized plans using abstract counting. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 991–997. AAAI Press, 2008.

[Toyer *et al.*, 2018] Sam Toyer, Felipe Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, pages 6294–6301. AAAI Press, 2018.

[Yoon *et al.*, 2005] Sung Wook Yoon, Alan Fern, and Robert Givan. Learning measures of progress for planning domains. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005)*, pages 1217–1222. AAAI Press, 2005.

[Yoon *et al.*, 2006] Sung Wook Yoon, Alan Fern, and Robert Givan. Learning heuristic functions from relaxed plans. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*, pages 162–170. AAAI Press, 2006.

[Yoon *et al.*, 2008] Sung Wook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *Journal of Machine Learning Research*, 9:683–718, 2008.