Proceedings of the Fifteenth International Symposium on Combinatorial Search (SoCS 2022)

Scaling-Up Generalized Planning as Heuristic Search with Landmarks

Javier Segovia-Aguas[1], Sergio Jiménez Celorrio[2], Laura Sebastiá [2] and Anders Jonsson[1]

[1]Universitat Pompeu Fabra
[2]VRAIN - Valencian Research Institute for Artificial Intelligence, Universitat Politècnica de València
javier.segovia@upf.edu, serjice@dsic.upv.es, anders.jonsson@upf.edu, lsebastia@dsic.upv.es

Abstract

Landmarks are one of the most effective search heuristics for classical planning, but largely ignored in generalized planning. Generalized planning (GP) is usually addressed as a combinatorial search in a given space of algorithmic solutions, where candidate solutions are evaluated w.r.t. the instances they solve. This type of solution evaluation ignores any sub-goal information that is not explicit in the representation of the planning instances, causing plateaus in the space of candidate generalized plans. Furthermore, node expansion in GP is a run-time bottleneck since it requires evaluating every child node over the entire batch of classical planning instances in a GP problem. In this paper we define a landmark counting heuristic for GP (that considers sub-goal information that is not explicitly represented in the planning instances), and a novel heuristic search algorithm for GP (that we call PGP) and that progressively processes subsets of the planning instances of a GP problem. Our two orthogonal contributions are analyzed in an ablation study, showing that both improve the state-of-the-art in *GP as heuristic search*, and that both benefit from each other when used in combination.

Introduction

Generalized planning (GP) addresses the computation of algorithmic solutions that are valid for a set of classical planning instances from a given domain (Winner and Veloso 2003; Hu and Levesque 2011; Srivastava, Immerman, and Zilberstein 2011; Srivastava 2011; Hu and De Giacomo 2011; Belle and Levesque 2016; Illanes and McIlraith 2019; Segovia-Aguas, Jiménez, and Jonsson 2019; Francès, Bonet, and Geffner 2021). In the worst case, each classical planning instance may require a completely different solution but in practice, many planning domains are known to have polynomial algorithmic solutions (Helmert 2006a; Fern, Khardon, and Tadepalli 2011). GP is however a challenging computation task; specifying an algorithmic solution for a set of classical planning instances often requires features that are not explicitly represented in those instances and hence, they must be discovered (Bonet and Geffner 2021).

GP is typically addressed as a combinatorial search in a space of algorithmic solutions, where candidate solutions are evaluated w.r.t. the instances they solve. Recently, heuristic search in the solution space of *planning programs* for GP has shown to be effective when guided by goal-oriented heuristic functions (Segovia-Aguas, Jiménez, and Jonsson 2021). However the used heuristics ignore sub-goal information, and often cause large search plateaus. In addition, each candidate solution is evaluated over the entire batch of classical planning instances of the GP problem, increasing the likelihood of search getting stuck in plateaus.

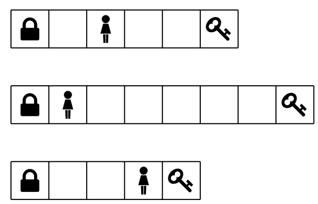

Figure 1: Three initial states of classical planning instances where an agent must open a lock in a $1 \times N$ corridor.

Figure 1 shows three initial states that correspond to three classical planning instances where an agent must open a lock in a $1 \times N$ corridor. The actions available for the agent are: *move* (one cell) right or left, *pick-up* or *drop* the key, and *open* the lock with the key. A generalized plan that solves these three instances, and that generalizes no matter the initial agent location or corridor length, can be formulated as: *move right until reaching the end of the corridor, pick up the key, move left until reaching the beginning of the corridor, and finally, open the lock with the key*. Please note that the only information provided by the goals of the previous instances is whether the lock is open. Relevant sub-goal information is however automatically deducible from the problem representation (Hoffmann, Porteous, and Sebastia 2004). For example the *landmarks* in Figure 2, indicating that the agent must reach all cells of the corridor and hold the key, can automatically be extracted from the representation of the first classical planning instance illustrated in Figure 1.

This paper introduces two orthogonal contributions:

- A *new heuristic search algorithm for GP*, that improves the performance of computing generalized plans by progressively evaluating them on subsets of the input planning instances.
- The adaptation of the *landmark graph* from classical

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

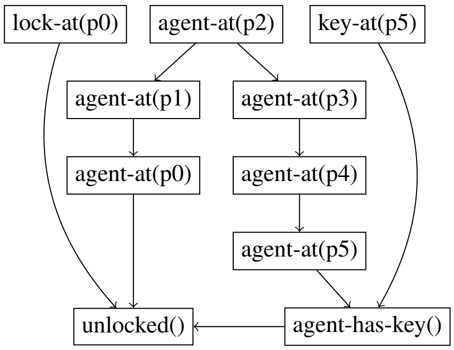

Figure 2: Example of a partial *landmark graph* for the first classical planning instance illustrated in Figure 1.

planning to GP, and the definition of a *landmark counting heuristic* for GP, that considers sub-goal information that is not explicitly represented in the planning instances.

The performance of these two orthogonal contributions is analyzed in an ablation study, showing that both outperform BFGP, the state-of-the-art in GP as heuristic search (Segovia-Aguas, Jiménez, and Jonsson 2021), and that both benefit from each other when used in combination.

Background

Classical Planning

In this work we consider the STRIPS fragment of the Planning Domain Definition Language (PDDL) for classical planning (Haslum 2019), that compactly defines a planning problem as $P = \langle \mathcal{D}, \mathcal{I} \rangle$, where $\mathcal{D}$ is the planning *domain* and $\mathcal{I}$ is an *instance*. The **domain**[1] $\mathcal{D} = \langle \mathcal{F}, \mathcal{A} \rangle$ consists of a set of FOL *predicates* $\mathcal{F}$, each of the form $p(x_1, \ldots, x_k)$ where p denotes a k-ary predicate symbol and x_i, $1 \leq i \leq k$, are variables; and a set of *action schemes* $\mathcal{A}$, where each $\alpha \in \mathcal{A}$ is defined as $\alpha = \langle par_\alpha, pre_\alpha, \mathit{eff}_\alpha \rangle$ with par_α denoting its *parameters* (arguments), and pre_α and eff_α are sets of atoms defined over variables in par_α that stand for the *preconditions* and the *effects* of the action schema α. The **instance** is defined as $\mathcal{I} = \langle \Omega, I, G \rangle$, where Ω is the finite set of world *objects*, I is the *initial state*, and G is the *goal* condition, a partial state that compactly represents the subset of goals states S_G.

A *state* consists of all ground atoms $p(o_1, \ldots, o_k)$, with k-ary predicate symbols p and instance objects $o_i \in \Omega$ for $1 \leq i \leq k$, interpreted either *true* or *false*; a *partial state* is a subset of all ground atoms. The set of *ground actions* A is computed substituting the parameters par_α of each action schema $\alpha \in \mathcal{A}$ with a tuple of objects $\overrightarrow{o}$ of the same size as the action parameters, i.e. a ground action $a \in A$ from an action schema $\alpha \in \mathcal{A}$ is $a = \alpha[\overrightarrow{o}]$ s.t. $|\overrightarrow{o}| = |par_\alpha|$; hence, pre_a and eff_a are partial states after grounding pre_α and eff_α atoms over objects $\overrightarrow{o}$. A ground action a is *applicable* iff its preconditions hold in the current state s, i.e. iff $pre_a \subseteq s$. Let us first split the action effects into *positive* and *negative* that respectively interpret ground atoms to true and false after applying action a, i.e. $\mathit{eff}_a = \mathit{eff}^+_a \cup \mathit{eff}^-_a$. The successor state $s' = a(s)$ is built removing the negative effects, and then adding the positive action effects, i.e. $s' = (s \setminus \mathit{eff}^-_a) \cup \mathit{eff}^+_a$.

A *solution* to P is a sequence of actions, or *sequential plan*, $\pi = \langle a_1, \ldots, a_m \rangle$, such that applied in the initial state $s_0 = I$ it induces a trajectory $\tau = \langle s_0, a_1, s_1, \ldots, a_m, s_m \rangle$ where each action $a_i(s_{i-1})$ is applicable, and the goal condition holds in the last state, i.e. $G \subseteq s_m$.

Example. The classical planning instances, with initial states illustrated in Figure 1, can be formulated with ground atoms $\{$`lock-at(`p_0`)`, `key-at(`p_{N-1}`)`, `agent-at(`p_i`)`, `agent-has-key`, `unlocked`, `adjacent(`p_i`,`p_{i+1}`)`$\}$, where N is the corridor length and $\Omega = \{p_i | 0 \leq i < N\}$ is the set of objects representing the different corridor locations, and four action schemes $\mathcal{A}$ $=\{$`move(`x_1`,`x_2`)`, `pickup-key(`x`)`, `drop-key(`x`)`, `open-lock(`x`)`$\}$. The initial states of the three classical planning instances can then be represented as I_1=$\{$`lock-at(`p_0`)`, `key-at(`p_5`)`, `agent-at(`p_2`)`$\}$, I_2=$\{$`lock-at(`p_0`)`, `key-at(`p_7`)`, `agent-at(`p_1`)`$\}$, I_3=$\{$`lock-at(`p_0`)`, `key-at(`p_4`)`, `agent-at(`p_3`)`$\}$, completed with the corresponding `adjacent(`p_i`,`p_{i+1}`)` ground atoms, that are static. The goal condition is the same for the three instances: $G_1 = G_2 = G_3 = \{$`unlocked`$\}$.

Landmarks in Classical Planning

Fact landmarks were introduced for classical planning by Porteous, Sebastia, and Hoffmann (2001) as a subgoaling mechanism, which later was adopted for the first landmark-based heuristic by Richter, Helmert, and Westphal (2008). In this article we refer to *fact landmarks* as *landmarks*.

Definition 1 (Landmark)**.** *A ground atom $p(o_1, \ldots, o_k)$ is a landmark of a classical planning problem P iff for every sequential plan $\pi = \langle a_1, \ldots, a_m \rangle$ that solves P, the ground atom $p(o_1, \ldots, o_k)$ holds for some state s_i with $0 \leq i \leq m$ of the induced trajectory $\tau = \langle s_0, a_1, s_1, ..., a_m, s_m \rangle$.*

According to Definition 1, all the facts appearing in the initial state and goals of a classical planning problem are *landmarks* (resp. considering time steps $i = 0$ and $i = m$). Landmarks can also be formulae over the state variables and actions. For instance, *disjunctive landmarks* indicate that, for every sequential plan that solves P, one of the atoms in a given disjunction holds at some state s_i, $0 \leq i \leq m$, in the induced trajectory τ. Landmarks can be (partially) ordered according to the time step where they must be achieved (Hoffmann, Porteous, and Sebastia 2004; Richter, Helmert, and Westphal 2008; Karpas and Domshlak 2009).

In this paper we focus on *fact*, *disjunctive landmarks*, and their orderings, while considering as future work other formalisms such as *action landmarks* (Karpas and Domshlak 2009; Helmert and Domshlak 2009; Büchner, Keller, and Helmert 2021). The two kinds of orderings we extract between landmarks are *natural orderings*, i.e. a landmark is true some time before another landmark, and *greedy necessary orderings*, i.e. one landmark is always true one step be-

[1] We assume domain constants to be included in the set of instance objects, and reformulate $\mathcal{A}$ accordingly if required.

fore another landmark becomes true for the first time (Hoffmann, Porteous, and Sebastia 2004).

Given a classical planning problem P, its corresponding *landmark graph*, $LG = \langle LM, O\rangle$, is a directed graph that comprises the set of landmarks LM, and the set of orderings O between these landmarks. For instance, Figure 2 illustrates the landmark graph of the first instance introduced in Figure 1. For clarity, static atoms indicating the adjacency of two corridor cells and natural orderings are omitted.

Generalized Planning with Planning Programs

This work builds on top of the inductive formalism for GP, where a GP problem is a finite set of classical planning instances from a given domain (Jiménez, Segovia-Aguas, and Jonsson 2019). In more detail, we build on top of the *GP as heuristic search* approach (Segovia-Aguas, Jiménez, and Jonsson 2021), which represents generalized plans as *planning programs*.

Definition 2 (GP problem)**.** *A* GP problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ *is a finite and non-empty set of T classical planning problems $P_1 = \langle \mathcal{D}, \mathcal{I}_1\rangle, \ldots, P_T = \langle \mathcal{D}, \mathcal{I}_T\rangle$ that belong to the same domain D, and where each instance $\mathcal{I}_t$, $1 \leq t \leq T$, may actually differ in the set of ground atoms and actions, initial state, or goals.*

Unlike sequential plans, *planning programs* include a control flow construct which allows the compact representation of solutions to classical and GP problems (Segovia-Aguas, Jiménez, and Jonsson 2019). Formally a *planning program* is a sequence of n instructions $\Pi = \langle w_0, \ldots, w_{n-1}\rangle$, where each instruction $w_i \in \Pi$ is associated with a *program line* $0 \leq i < n$, and is either:

- A *planning action* $w_i \in A$.
- A *goto instruction* $w_i = \mathsf{go}(i', y)$, where i' is a program line and y is a proposition.
- A *termination instruction* $w_i = \mathsf{end}$. The last instruction of a planning program is always a termination instruction, i.e. $w_{n-1} = \mathsf{end}$.

The execution model for a planning program is a *program state* (s, i), i.e. a pair of a planning state $s \in S$ and program counter $0 \leq i < n$. Given a program state (s, i), the execution of a programmed instruction w_i is defined as:

- If $w_i \in A$, and w_i is applicable in s, the new program state is $(s', i{+}1)$, where $s' = w_i(s)$ is the *successor* state. If w_i is not applicable the new program state is $(s, i+1)$, i.e. the planning state is unmodified.
- If $w_i = \mathsf{go}(i', y)$, the new program state is (s, i') if y holds in s, and $(s, i+1)$ otherwise. The proposition y can actually be the result of an arbitrary expression on the state variables, e.g. a state *feature* (Lotinac et al. 2016).
- If $w_i = \mathsf{end}$, program execution terminates.

To execute a planning program Π on a classical planning problem $P = \langle \mathcal{D}, \mathcal{I}\rangle$, the initial program state is set to $(I, 0)$, i.e. the initial state of P and the first program line of Π. A program Π *solves* P iff the execution terminates in a program state (s, i) that satisfies the goal condition, i.e. $w_i = \mathsf{end}$ and $G \subseteq s$. Otherwise the execution of the program fails. The two possible sources of failure of the execution of a planning program Π on a classical planning problem P are then:

1. *Incorrect program*, i.e. execution terminates in a program state (s, i) that does not satisfy the goal condition, i.e. $(w_i = \mathsf{end}) \wedge (s \notin S_G)$.
2. *Infinite program*, i.e. execution enters into an infinite loop that never reaches an end instruction. This can be easily detected whenever a program state is duplicated.

Definition 3 (GP solution)**.** *A* generalized plan Π *is a solution to a GP problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ iff, for every classical planning problem $P_t \in \mathcal{P}$, $1 \leq t \leq T$, the sequential plan that results from executing Π on P_t, i.e. $exec(\Pi, P_t) = \langle a_1, \ldots, a_m\rangle$, solves P_t.*

Planning Programs for STRIPS Domains

To build *planning programs* that generalize over the instances of a STRIPS domain, we introduce the notion of *pointer* over the world objects, and redefine planning programs accordingly.

Definition 4 (Pointer)**.** *A* pointer $z \in Z$ *is a bound variable, with finite domain $D_z = [0..|\Omega|)$, that indexes an object of a planning instance $\mathcal{I}$.*

We redefine planning programs, so *planning actions* $w_i \in A$ are not ground actions over the instance objects, but action schemes $\mathcal{A}$ instantiated over *pointers* in Z, i.e. $\alpha[\overrightarrow{z}] \in A_Z$. The execution model of planning programs is updated accordingly; instructions $w_i = \alpha[\overrightarrow{z}] \in A_Z$ first map every pointer to its indexed object in constant time, which turns the instruction into a ground action $a = \alpha[\overrightarrow{o}] \in A$, from which the standard execution model applies. Figure 3 illustrates the relation between (i) an action schema; (ii) its instantiation over pointers; and (iii) its instantiation over objects. Pointers may be typed to address the subset of objects of the same type, although we also refer to them as pointers in the article for short.

In addition, the set of instructions of planning programs is extended with the set of *primitive pointer operations* that comprises: $\{\mathtt{inc}(z_1), \mathtt{dec}(z_1), \mathtt{clear}(z_1), \mathtt{set}(z_1, z_2) \mid z_1, z_2 \in Z\}$ over the pointers in Z, and $\{\mathtt{test}_p(\overrightarrow{z}) \mid \overrightarrow{z} \in Z^{ar(p)}\}$ over the lists of pointers in $Z^{ar(p)}$ for each predicate symbol $p \in \mathcal{F}$ in a given planning domain $\mathcal{D}$. Respectively these primitive instructions *increment/decrement* a pointer by one, set a pointer to zero, and *set* the value of a pointer z_2 to another pointer z_1. Instruction $\mathtt{inc}(z_1)$ is applicable iff $z_1 < |\Omega| - 1$, and $\mathtt{dec}(z_1)$ is applicable iff $z_1 > 0$, while the remaining instructions are always applicable. The $\mathtt{test}_p(\overrightarrow{z})$ instruction returns the interpretation of $p(\overrightarrow{z})$ at the current state which, similarly to planning actions, requires first to map $\overrightarrow{z}$ to the corresponding indexed objects $\overrightarrow{o}$ s.t. $p(\overrightarrow{o})$ is a ground atom.

Last the *goto* instructions of planning programs are restricted to be conditioned by a single Boolean y_z that, playing the role of a *zero* FLAGS register (Dandamudi 2005), is dedicated to store the outcome of the last executed primitive operation over pointers. The FLAG y_z allows to keeping the solution space tractable. Formally, it is defined as:

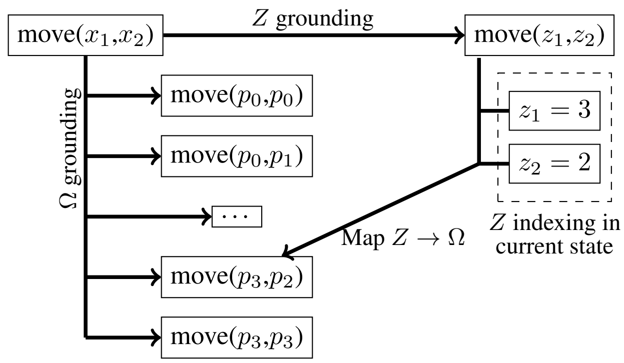

Figure 3: Action schema *move*(x_1,x_2) $\in \mathcal{A}$, where x_1 and x_2 are free variables, for moving the agent in the domain of Figure 1. The action *move*(z_1,z_2) $\in \mathcal{A}_{\mathcal{Z}}$ ground over the pointers $Z = \{z_1, z_2\}$, where z_1 and z_2 are bound variables in $[0, |\Omega|)$, that are respectively indexing objects p_3 and p_2. The set of actions $\{move(p_0, p_0), move(p_0, p_1), \ldots, move(p_3, p_2), move(p_3, p_3)\}$ ground over the set of objects $\Omega = \{p_0, p_1, p_2, p_3\}$.

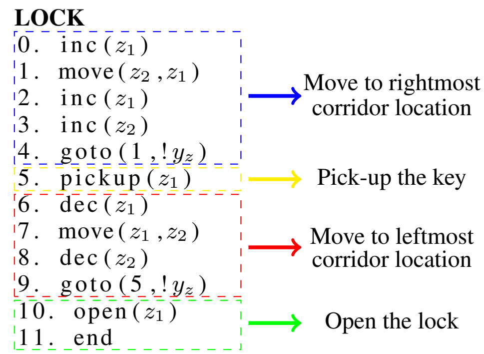

Figure 4: Planning program that solves instances illustrated in Figure 1. Note that y_z captures the outcome of the last executed primitive pointer instruction, and that a pointer increase makes $y_z = False$ until it becomes inapplicable where $y_z = True$, while a decrease instruction makes $y_z = True$ when the pointer decreases from 1 to 0 or if it is inapplicable, otherwise $y_z = False$.

$$y_z \equiv \begin{cases} False, & \text{if applicable } w_i = inc(z_1), \\ (z_1 == 1), & \text{if applicable } w_i = dec(z_1), \\ True, & \text{if } w_i = clear(z_1), \\ (z_2 == 0), & \text{if } w_i = set(z_1, z_2), \\ \neg p(\overrightarrow{z}), & \text{if } w_i = test_p(\overrightarrow{z}), \\ True, & \text{if inapplicable } w_i. \end{cases}$$

Object typing is naturally supported by pointers, specializing them to the number of objects of a particular type. Pointers can trivially be mapped to a mutually exclusive set of PDDL propositional variables. Likewise primitive pointer operations, and goto instructions, can be coded as PDDL actions with conditional effects.

Example. Figure 4 shows a planning program with $n = 12$ program lines, and pointers $Z = \{z_1, z_2\}$, that solves the three planning instances above, and that is computed by our PGP algorithm. Program lines 1, 5, 7 and 10 contain planning actions in A_Z, where pointers index the corridor locations. The program lines 4 and 9 contain *goto* instructions A_{go} that branch the program execution flow according to the value of the zero flag y_z. The remaining lines contain primitive pointers instructions, that operate over pointers, and update the zero flag y_z accordingly. The last instruction is an `end` instruction. The program of Figure 4 leverages the fact that, in the three classical planning instances given as input, adjacent locations are named with consecutive numbers.

Pointers are always initialized to zero. Therefore, the execution of the planning program of Figure 4 on the first classical planning instance illustrated in Figure 1 produces the following sequential plan $\pi = \langle move(p_0, p_1), move(p_1, p_2), move(p_2, p_3), move(p_3, p_4), pickup(p_4), move(p_4, p_3), move(p_3, p_2), move(p_2, p_1), move(p_1, p_0), open(p_0)\rangle$. Next we detail the execution of the first five program lines, which produces the sequence of ground actions to reach the rightmost location of the corridor; pointers z_1 and z_2 are initialized to zero, so they initially index the same object p_0. Program line 0 increments the value of pointer z_1, so it indexes p_1 and hence, the execution of program line 1 corresponds to the execution of the ground action $move(p_0, p_1)$. Lines 2 and 3 increment the two pointers, so z_1 indexes p_2 while z_2 indexes p_1. Line 4 indicates that the block of program lines $[1-4]$ is repeated until pointer z_2 can no longer be incremented.

Progressive Generalized Planning

This section describes our *Progressive heuristic search algorithm for Generalized Planning* (PGP). This algorithm adapts a *Best-First Search* (BFS), in the solution space of the possible planning programs that can be built with n program lines and $|Z|$ pointers, so that it progressively processes the full batch of classical planning instances of a GP problem.

Progressive Best-First Search for GP

The input to our PGP algorithm is a GP problem $\mathcal{P}_{n,Z} = \{P_1, \ldots, P_T\}$. PGP outputs a planning program Π that solves $\{P_1, \ldots, P_T\}$, or it reports unsolvability within the maximum number of n program lines and $|Z|$ pointers. Briefly, PGP keeps a subset of the classical planning instances called the *active instances*, that initially contains only the first classical planning instance of the GP problem. When PGP finds a program that solves the full set of *active instances*, it validates that program on the remaining instances of the GP problem, and augments the set of *active instances* with the first instance for which the program fails. The procedure is repeated until PGP finds a program that solves all the instances in the GP problem. PGP can be understood as a variant of *counterexample-guided search* (Seipp and Helmert 2018).

Algorithm 1 shows the pseudo-code of PGP. In more detail, in Lines 1-2, the algorithm **initializes** the subset of *ac-*

Algorithm 1: PGP

```
Data: A GP problem P_{n,Z}
Result: A planning program Π that solves P_{n,Z} (or unsolvable)
1  active ← {P_1} ;
2  open ← insertProgram(∅, Π_empty, active);
3  while open ≠ ∅ do
4  |  Π ← extractBestProgram(open) ;
5  |  childs ← expandProgram(Π, P_{n,Z}) ;
6  |  for Π' ∈ childs do
7  |  |  if isSolution(Π',active) then
8  |  |  |  if isSolution(Π',P_{n,Z}) then
9  |  |  |  |  return Π';
10 |  |  |  P_fail ← getFirstFailed(Π',P_{n,Z});
11 |  |  |  active ← active ∪ {P_fail};
12 |  |  |  reevaluateQueue(open,active);
13 |  |  if not isDeadEnd(Π', active) then
14 |  |  |  open ← insertProgram(open, Π', active);
15 |  end
16 end
17 return unsolvable;
```

tive instances with the first planning instance $P_1 \in \mathcal{P}_{n,Z}$, and inserts an empty planning program Π_{empty} with n undefined program lines, into the empty *open* priority queue. The main loop runs in Lines 3-16, while there are programs to expand, and it returns a GP solution if found (Line 9), otherwise it returns $unsolvable$ on Line 17. In every iteration, the best-evaluated program Π is **selected** and removed from the $open$ priority queue (Line 4), and **expanded** (Line 5). For each child node Π', PGP checks whether Π' solves the $active$ instances (Line 7). If Π' also solves $\mathcal{P}_{n,Z}$ then it is returned as the GP solution (Lines 8-9), otherwise the first instance P_{fail} where Π' fails is added to the $active$ set, and the $open$ queue is **reevaluated** (Lines 10-12) w.r.t. the problems in the augmented *active* set. In Lines 13-14, the child Π' is evaluated over the set of $active$ instances and **inserted** in the $open$ queue if it is not a **deadend** for the $active$ set.

PGP is a *frontier search* algorithm meaning that, to reduce memory requirements, it stores only the *open list* of generated nodes but not the *closed list* of expanded nodes (Korf et al. 2005). With regard to node expansion, let Π be the partially specified program that corresponds to the best-evaluated node extracted by PGP from the open list (Line 4). PGP generates one successor for each program that results from programming the first undefined line of Π. This node expansion procedure guarantees that duplicate successors are not generated and it keeps the branching factor tractable; at a given undefined program line PGP can only program (i) a planning action; (ii) a primitive pointer instruction; or (iii) a goto instruction. The maximum depth of the PGP search tree is the number of program lines n, since only an undefined line can be programmed.

Theorem 1 (Termination)**.** *The execution of* PGP *on a GP problem, always terminates.*

Proof. The only possible cause for non-termination would be that PGP could generate duplicate search nodes, allowing the infinite re-opening of an already discarded program. By the definition of the PGP expansion procedure, it does not generate duplicate successors; a child node always has one more line programmed than its parent. □

Theorem 2 (Completeness)**.** *Given a GP problem, and a maximum number of program lines n and pointers $|Z|$, if there is a solution planning program within these bounds then PGP can compute it.*

Proof. PGP only discards a search node when its corresponding partially specified planning program fails to solve a planning instance. This is precisely the definition for not being a GP solution. Further, any planning program that could be built programming the remaining undefined program lines would also fail to solve that same instance. □

Theorem 3 (Soundness)**.** *If the execution of* PGP *on a GP problem outputs a generalized plan Π, then this means that Π is a solution for that GP problem.*

Proof. PGP runs until (i) the open list is empty, which means that search space is exhausted without finding a solution and no generalized plan is output; or (ii) PGP extracted from the open list a planning program whose execution solves all the instances $P_t \in \mathcal{P}_{n,Z}$. This is precisely the definition of a solution for a GP problem. □

Landmarks in Generalized Planning

This section defines a landmark counting heuristic for guiding a combinatorial search in our GP solution-space.

The Landmark Graph with Pointer Assignments

For each classical planning problem $P_t \in \mathcal{P}_{n,Z}$, we enrich its corresponding landmark graph LG_t with: (i) *pointer landmarks*, that indicate pointer assignments that must be achieved by a solution to P_t; and (ii) orderings between *pointer landmarks* and the regular landmarks, computed by the LAMA algorithm.

First we compute, for every classical planning problem $P_t \in \mathcal{P}_{n,Z}$, $1 \leq t \leq T$, its corresponding *landmark graph* $LG_t = \langle LM_t, O_t \rangle$, as a pre-processing step. We implement the back-chaining LAMA algorithm for finding landmarks and orderings between landmarks (Richter and Westphal 2010). Briefly we start from a set of known landmarks, and find new landmarks that hold in any plan before an already known landmark may become true. The procedure stops when no more new landmarks are discovered. In more detail, given a landmark q, if all actions that achieve q for the first time share a precondition p, this means that p is also a landmark, and that there is a *greedy necessary* ordering $p \rightarrow_{gn} q$ between them. The algorithm discovers *disjunctive landmarks* too; when q is a landmark, and all actions that first achieve q have p_1, or p_2, ..., or p_k as a precondition, this means that $p_1 \vee p_2 \vee \ldots \vee p_k$ is also a landmark. We also implement the algorithm for adding *additional orderings from restricted Relaxed Planning Graphs* (RPGs) to discover natural orderings between landmarks (Richter and Westphal 2010).

Definition 5 (Pointer landmarks)**.** *Given a classical planning problem P and a set of pointers Z, we say that the assignments $\bigwedge_j \bigvee_i (z_i = o_j)$ are* pointer landmarks *iff the ground atom $p(\overrightarrow{o})$ is a landmark for P, $z_i \in Z$, and $o_j \in \overrightarrow{o}$.*

In other words given a landmark $p(\overrightarrow{o})$, a *pointer landmark* indicates that at least one pointer must point to each object in $\overrightarrow{o}$. *Pointer landmarks*, and their corresponding orderings, are computed as follows. For every *greedy necessary* order $p \rightarrow_{gn} q$ in the landmark graph LG_t computed by the LAMA algorithm, if an action $a \equiv \alpha(\overrightarrow{o})$ is an achiever for q, then we have that the assignments $\bigwedge_j \bigvee_i (z_i = o_j)$ are pointer landmarks, and furthermore, $\bigwedge_j \bigvee_i (z_i = o_j) \rightarrow_{gn} q$ is a *greedy necessary* ordering where $\{z_i\} \subseteq Z$ are the pointers of the same type of the corresponding object $o_j \in \overrightarrow{o}$. For example given the first instance illustrated in Figure 1 and the set of pointers $Z = \{z_1, z_2\}$, then the corresponding *landmark graph* of Figure 2 is enriched with the disjunctive landmark $(z_1 = 5 \vee z_2 = 5)$ and the greedy necessary order $(z_1 = 5 \vee z_2 = 5) \rightarrow_{gn}$ `agent-has-key`, among other orderings. These particular pointer landmarks, and their *greedy necessary* ordering, are created since the actions `pickup`(z_1) or `pickup`(z_2) for $z_1 = 5$ or $z_2 = 5$, are first achievers of the `agent-has-key` landmark.

The Landmark Counting Heuristic for GP

The *landmark graph* extracted from a given classical planning instance can be used to guide a forward search in the space of states reachable from the initial state of that instance (Richter and Westphal 2010; Büchner, Keller, and Helmert 2021). For example, to implement the evaluation function $f_{LM}(s, \pi)$ of the LAMA planner, that computes the number of landmarks that have not been achieved on the path from the initial state to the state $s \in S$, given by the sequence of planning actions π. This evaluation function is formalized as:

$$f_{LM}(s, \pi) = |(LM \setminus Reached(s, \pi)) \cup ReqAgain(s, \pi)|,$$

where LM is the set of landmarks discovered with the previously described mechanism, $Reached(s, \pi) \subseteq LM$ is the subset of reached landmarks; a landmark p is first reached in state s if all predecessors of p in the corresponding landmark graph have been reached, and $p \in s$. Once a landmark is reached in a state s, it remains reached in all successor states. Last, $ReqAgain(s, \pi) \subseteq Reached(s, \pi)$ is the subset of landmarks that must be achieved again that comprises: goals p that are false in s, and greedy necessary predecessors p of some landmark q that has not been reached yet. Note that $f_{LM}(s, \pi)$ is not a heuristic function in the standard sense, since its value is path-dependent, but it works well for classical planning.

Next we show how we leverage landmarks to inform a combinatorial search in our GP solution-space. When a program execution terminates because an unspecified program line is reached, we retrieve the last state reached, and estimate how far this state is from a goal state with the following heuristic function:

- $f_{LM}(\Pi, \mathcal{P}_{n,Z}) = \sum_t f_{LM}(\Pi, P_t)$ for each $P_t \in \mathcal{P}_{n,Z}$, where $f_{LM}(\Pi, P_t) = f_{LM}(s, \pi)$ such that:
  - $\pi = exec(\Pi, P_t)$ is the sequential plan that results from executing Π on P_t,
  - s is the last state reached after executing Π on P_t,
  - $f_{LM}(s, \pi)$ is the landmark counting heuristic defined above.

Note that, when used in combination with our PGP search algorithm, the $f_{LM}(\Pi, \mathcal{P}_{n,Z})$ heuristic is evaluated only over the $active$ set, i.e. $f_{LM}(\Pi, active)$, instead of over the full GP problem $\mathcal{P}_{n,Z}$ which allows to saving computations.

A potential issue with our landmark counting heuristic is that aggregating each $f_{LM}(\Pi, P_t)$ could make instances with more landmarks bias the search. This issue could be mitigated normalizing the heuristic values with the total number of landmarks; however, we have not observed this issue to have an impact in the experiments.

Evaluation

We compare on eight STRIPS domains the performance of PGP(f_{GC}), our algorithm guided by our landmark counting heuristic for GP, w.r.t the *GP as heuristic search* approach (Segovia-Aguas, Jiménez, and Jonsson 2021) that serves as a baseline. We also report an ablation study of our two orthogonal contributions. All experiments were performed using 10 random input instances of increasing difficulty per domain, in an Ubuntu 20.04 LTS, with AMD® Ryzen 7 3700x 8-core processor and 32GB of RAM, with a 1 hour time bound [2].

Benchmarks. The *Baking* domain, where an agent follows the steps to bake a set of cakes. *Corridor*, where an agent moves from an arbitrary initial location to a destination location in a corridor. *Gripper*, where a robot must pick all balls from room A and drop them in room B. *Intrusion*, where an attacker performs a number of actions to steal data from some host computers. *Lock*, the domain illustrated in Figure 1. *Ontable*, in which towers of blocks are placed on the table. *Spanner*, where an agent must pick up spanners to tighten the loose nuts at the end of a corridor (spanners can only be used once and the agent cannot go back, introducing dead-ends). *Visitall*, where starting from the bottom-left corner of a square grid, an agent must visit all grid cells.

Synthesis and ablation study. In the first experiment we use as a baseline the *GP as heuristic search* approach (Segovia-Aguas, Jiménez, and Jonsson 2021) that implements a Best-First Search (BFS) guided by its best single heuristic; a Euclidean distance (which actually acts as a *counter of unachieved goals* in propositional domains) and that we denote as f_{GC}. Table 1 reports the best solutions found in terms of the number of required program lines n and pointers $|Z|$. The results show that BFS(f_{GC}) works well for *Visitall* and *Gripper*, where tracking the achieved problem goals provides a monotonic measure of

[2] Repository at https://github.com/aig-upf/pgp-landmarks.

	$n, \|Z\|$	Baseline- BFS(f_{GC})		Contribution 1- BFS(f_{LM})		Contribution 2- PGP(f_{GC})		Contri. 1&2- PGP(f_{LM})	
		T/M	Ex/Ev	T/M	Ex/Ev	T/M	Ex/Ev	T/M	Ex/Ev
Baking	13, 6	TE/TE	TE/TE	TE/TE	TE/TE	ME/ME	ME/ME	**98/46**	**48K/1.4M**
Corridor	11, 2	101/**27**	**6K**/120K	45/34	7K/**114K**	**12**/50	**6K**/120K	16/57	**6K**/121K
Gripper	8, 4	5/**15**	**3K/63K**	43/20	19K/339K	**1**/28	**3K/63K**	8/20	19K/336K
Intrusion	9, 1	94/284	73K/1.3M	**0/18**	**8/190**	24/877	74K/1.3M	**0/18**	**8/190**
Lock	12, 2	TE/TE	TE/TE	TE/TE	TE/TE	ME/ME	ME/ME	**3/46**	**1K/30K**
Ontable	11, 3	TE/TE	TE/TE	TE/TE	TE/TE	**25**/196	10K/366K	323/**98**	**3K/113K**
Spanner	12, 5	TE/TE	TE/TE	TE/TE	TE/TE	TE/TE	TE/TE	**172/604**	**25K/705K**
Visitall	7, 2	**0/7**	50/511	7/17	**27/249**	**0/7**	88/900	**0**/17	49/446

Table 1: Number of program lines n and pointers $|Z|$, total time (T) in seconds, memory peak (M) in MB, expanded (Ex) and evaluated (Ev) nodes (K is 10^3 and M is 10^6). TE and ME stand for time and memory exceeded. Best results in bold.

search progress; however, it becomes insufficient when explicit goals do not provide such information (e.g. in *Baking*, *Intrusion*, *Lock* or *Spanner*). The benefits of combining our two orthogonal contributions are represented by $PGP(f_{LM})$, where our landmark counting heuristic f_{LM} is used to inform our PGP algorithm. $PGP(f_{LM})$ solves all domains within five minutes (approx.) including the preprocessing time of all the landmark graphs. Figure 5 shows the generalized plans computed by $PGP(f_{LM})$; the solution for the *lock* domain was already shown in Figure 4. We successfully validated all these solutions on large instances.

Our two orthogonal contributions are also evaluated separately in an *ablation study*, where we either ablate the contributed PGP algorithm which is the case for BFS(f_{LM}), or ablate the contributed heuristic f_{LM} which is the case for PGP(f_{GC}). BFS(f_{LM}), has the same coverage as the baseline. Its main drawback is that aggregating landmarks over all input instances may bias search towards certain regions of the space of planning programs where no solution generalized plan exists, e.g. consuming the first program line with an instruction that reaches a new landmark but that must be actually used some lines after invalidating the rest of the program. On the other hand, PGP(f_{GC}) outperforms the baseline; its heuristic is evaluated only in the subset of $active$ instances, improving the coverage and total time of the baseline, but still it suffers from large plateaus due to the poorly informed heuristic.

As a rule of thumb, f_{LM} is not better than f_{GC} in domains where the explicit problem goals are already providing an informative notion of search progress; the *Gripper* domain is a representative example of this. On the other hand, PGP exploits the fact that, in many domains, computing a succinct solution that generalizes to a small set of instances is generalizing to unseen problems. Therefore, PGP will perform worse than BFS when the programs that successfully solve the problems of the *active* set successively fail to generalize to the remaining problems.

Synthesis with f_1 for tie breaking. In Segovia-Aguas, Jiménez, and Jonsson (2021), f_{GC} is also used in combination with the structural evaluation function $f_1(\Pi)$, that counts the number of goto instructions in a planning program Π, and that is used for tie breaking. In this experiment we evaluate this same tie breaking with our contributions, i.e. $PGP(f_{LM}, f_1)$, and compare it with the best original setting in Segovia-Aguas, Jiménez, and Jonsson (2021), i.e. BFS(f_{GC},f_1) in our subset of propositional domains. Results in Table 2 show that $PGP(f_{LM}, f_1)$ also outperforms the *state-of-the-art* in *GP as heuristic search*, BFS(f_{GC},f_1), in almost all domains.

	BFS(f_{GC},f_1)		PGP(f_{LM},f_1)	
	T/M	Ex/Ev	T/M	Ex/Ev
Baking	TE/TE	TE/TE	**63/35**	**30K/937K**
Corridor	49/**16**	**3K**/67K	**8**/43	**3K/62K**
Gripper	**4/14**	**3K/59K**	8/18	19K/336K
Intrusion	55/183	51K/896K	**0/18**	**8/190**
Lock	TE/TE	TE/TE	**3/45**	**1K/26K**
Ontable	TE/TE	TE/TE	**312/97**	**3K/110K**
Spanner	TE/TE	TE/TE	**168/604**	**24K/670K**
Visitall	**0/7**	**45**/496	**0**/17	53/**458**

Table 2: Synthesis with BFS(f_{GC},f_1) and PGP(f_{LM},f_1), with the same input settings and metrics from Table 1.

Related Work

Our GP approach is related to previous work that computes *generalized heuristics* for guiding state-space search on new classical planning instances of a given domain (Francès et al. 2019; Ståhlberg, Francès, and Seipp 2021; Karia and Srivastava 2021). However, $PGP(f_{LM})$ does not aim learning a generalized heuristic but instead, it leverages the classical planning landmark machinery, that was originally conceived for state space search. We believe that our approach opens the door to incorporating into GP other successful techniques coming from classical planning, e.g. *helpful actions/preferred operators*.

Most of the previous work on GP compute generalized plans that solve, at once, the entire set of classical planning instances given as input. $PGP(f_{LM})$ implements a progressive approach that, one by one, processes the full batch of classical planning instances in a GP problem. Remarkably our progressive approach overcomes the main drawback of *bottom-up/online* approaches for GP (Winner and Veloso 2003; Srivastava, Immerman, and Zilberstein 2011), which suffer from the complexity of merging a new individual solution with the previously found solutions.

First-order logic (FOL) policies that specify a strategy for solving planning instances have also shown to generalize

GRIPPER

0. pick(z_b, z_{r1}, z_g)
1. inc(z_{r2})
2. move(z_{r1}, z_{r2})
3. drop(z_b, z_{r2}, z_g)
4. move(z_{r2}, z_{r1})
5. inc(z_b)
6. goto(0, $\neg y_z$)
7. end

INTRUSION

0. recon(z_h)
1. break-into(z_h)
2. clean(z_h)
3. gain-root(z_h)
4. download-files(z_h)
5. steal-data(z_h)
6. inc(z_h)
7. goto(0, $\neg y_z$)
8. end

ONTABLE (BLOCKS)

0. unstack(z_{o1}, z_{o2})
1. inc(z_{o2})
2. goto(0, $\neg y_z$)
3. put-down(z_{o1})
4. clear(z_{o2})
5. inc(z_{o1})
6. goto(0, $\neg y_z$)
7. clear(z_{o1})
8. inc(z_{o3})
9. goto(0, $\neg y_z$)
10. end

VISITALL

0. visit(z_i, z_j)
1. inc(z_i)
2. goto(0, $\neg y_z$)
3. clear(z_i)
4. inc(z_j)
5. goto(0, $\neg y_z$)
6. end

BAKING

0. puteggingpan(z_e, z_p)
1. putflourinpan(z_f, z_p)
2. mix(z_e, z_f, z_p)
3. putpaninoven(z_p, z_o)
4. bakecake(z_o, z_p, z_c)
5. removepanfromoven(z_p, z_o)
6. cleanpan(z_p, z_s)
7. inc(z_c)
8. inc(z_e)
9. inc(z_f)
10. inc(z_s)
11. goto(0, $\neg y_z$)
12. end

CORRIDOR

0. inc(z_1)
1. move(z_2, z_1)
2. inc(z_1)
3. inc(z_2)
4. goto(1, $\neg y_z$)
5. move(z_1, z_2)
6. set(z_1, z_2)
7. dec(z_2)
8. test(goal-at(z_1))
9. goto(4, y_z)
10. end

SPANNER

0. pickup_spanner(z_{l1}, z_s, z_m)
1. tighten_nut(z_{l1}, z_s, z_m, z_n)
2. inc(z_n)
3. inc(z_s)
4. goto(0, $\neg y_z$)
5. inc(z_{l1})
6. walk(z_{l2}, z_{l1}, z_m)
7. clear(z_n)
8. clear(z_s)
9. inc(z_{l1})
10. goto(0, $\neg y_z$)
11. end

Figure 5: Generalized plans computed by $PGP(f_{LM})$. The parameters of non-goto instructions are *pointers*. Goto instructions have two parameters; the destination line, and a condition over the zero flag (either y_z or $\neg y_z$).

to planning domains (Khardon 1999; Martin and Geffner 2004). Validating FOL policies over large instances (with large numbers of objects) is difficult because of the complexity of variable matching; our approach has constant variable matching complexity since *planning programs* have no free variables. On the other hand, generalized polices are able to solve problems when actions are not deterministic (Belle and Levesque 2016), which connects GP to more general notions of planning, such as MDPs and POMDPs (Kolobov 2012). Unlike related work that focus on the computation of generalized policies, our GP approach does not require knowing the full state space of the input instances (Francès, Bonet, and Geffner 2021), which may easily be too large to be fully specified, or reformulating actions w.r.t. a pool of features (Bonet, Francès, and Geffner 2019).

Deep Reinforcement Learning (DRL) is also used to learn policies (Sutton and Barto 2018), represented with *Deep Neural Networks* (DNNs), that solve sequential decision-making problems, even when symbolic representations of states and actions are not available (Mnih 2015). Off-the-shelf tools for learning DNNs have also been successfully applied to learn black-box generalized policies, and heuristics, from PDDL representations (Bueno 2019; Garg, Bajpai, and Mausam 2020; Toyer et al. 2020). DNNs are suitable for black box decision-making, but they are difficult to interpret; DNNs represent knowledge as millions of coupled parameters, so it becomes difficult to identify the piece of knowledge responsible for solving a particular task, as well as to understand whether this piece of knowledge will be useful for unseen problems. Last but not least, *model-based DRL* approaches have exhibited good performance in several domains (Hafner et al. 2020), but the learned world models are again represented as NNs; solutions are then difficult to interpret and their generalization capacity in the presence of new objects is not evaluated.

Conclusions

We presented $PGP(f_{LM})$, a novel heuristic search approach to GP that progressively processes the classical planning instances of a GP problem, and that leverages a landmark counting heuristic to search in the space of planning programs. $PGP(f_{LM})$ allows to transfer landmark counting heuristics, originally conceived for state-space, to the solution-space search of GP. There is still room for improving our f_{LM} heuristic for GP; the information captured by our landmark graphs could be augmented exploiting cyclic dependencies (Büchner, Keller, and Helmert 2021), considering the remaining number of programmable lines (Marzal, Sebastia, and Onaindia 2014), or leveraging different relaxations of the planning instances (Keyder, Richter, and Helmert 2010). Besides landmarks, heuristic planners implement complementary ideas such as *helpful actions/preferred operators* (Hoffmann and Nebel 2001), *multi-queue best-first search* for multiple heuristics combination (Helmert 2006b), or *novelty-based exploration* (Lipovetzky and Geffner 2012). A promising future research direction is to incorporate into the *GP as heuristic search* approach all these techniques that have proved successful for classical planning.

Acknowledgements

Javier Segovia-Aguas is supported by TAILOR, a project funded by EU H2020 research and innovation programme no. 952215. Sergio Jiménez is supported by Spanish grants RYC-2015-18009 and TIN2017-88476-C2-1-R. Anders Jonsson is partially supported by Spanish grants PID2019-108141GB-I00. Laura Sebastia is supported by TIN2017-88476-C2-1-R and TAILOR-952215.

References

Belle, V.; and Levesque, H. J. 2016. Foundations for generalized planning in unbounded stochastic domains. In *KR*.

Bonet, B.; Francès, G.; and Geffner, H. 2019. Learning features and abstract actions for computing generalized plans. In *AAAI*.

Bonet, B.; and Geffner, H. 2021. General policies, representations, and planning width. In *AAAI*.

Büchner, C.; Keller, T.; and Helmert, M. 2021. Exploiting Cyclic Dependencies in Landmark Heuristics. In *ICAPS*.

Bueno, T. P. *et al.*. 2019. Deep reactive policies for planning in stochastic nonlinear domains. In *AAAI*.

Dandamudi, S. P. 2005. Installing and using nasm. *Guide to Assembly Language Programming in Linux*.

Fern, A.; Khardon, R.; and Tadepalli, P. 2011. The first learning track of the international planning competition. *Machine Learning*, 84(1-2): 81–107.

Francès, G.; Bonet, B.; and Geffner, H. 2021. Learning General Policies from Small Examples Without Supervision. In *AAAI*.

Francès, G.; Corrêa, A. B.; Geissmann, C.; and Pommerening, F. 2019. Generalized potential heuristics for classical planning. In *IJCAI*.

Garg, S.; Bajpai, A.; and Mausam. 2020. Symbolic network: generalized neural policies for relational MDPs. In *ICML*.

Hafner, D.; Lillicrap, T. P.; Norouzi, M.; and Ba, J. 2020. Mastering Atari with Discrete World Models. In *ICLR*.

Haslum, P. *et al.*. 2019. An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2): 1–187.

Helmert, M. 2006a. New Complexity Results for Classical Planning Benchmarks. In *ICAPS*.

Helmert, M. 2006b. The Fast Downward Planning System. *JAIR*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, critical paths and abstractions: what's the difference anyway? In *ICAPS*.

Hoffmann, J.; and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14: 253–302.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *JAIR*, 22: 215–278.

Hu, Y.; and De Giacomo, G. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *IJCAI*.

Hu, Y.; and Levesque, H. J. 2011. A Correctness Result for Reasoning about One-Dimensional Planning Problems. In *IJCAI*.

Illanes, L.; and McIlraith, S. A. 2019. Generalized planning via abstraction: arbitrary numbers of objects. In *AAAI*.

Jiménez, S.; Segovia-Aguas, J.; and Jonsson, A. 2019. A review of generalized planning. *KER*, 34.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *AAAI*.

Karpas, E.; and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *IJCAI*.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and Complete Landmarks for And/Or Graphs. In *ECAI*.

Khardon, R. 1999. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1-2): 125–148.

Kolobov, A. 2012. Planning with Markov decision processes: An AI perspective. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1): 1–210.

Korf, R. E.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM*, 52(5).

Lipovetzky, N.; and Geffner, H. 2012. Width and serialization of classical planning problems. In *ECAI*.

Lotinac, D.; Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2016. Automatic generation of high-level state features for generalized planning. In *IJCAI*.

Martin, M.; and Geffner, H. 2004. Learning generalized policies from planning examples using concept languages. *Applied Intelligence*, 20: 9–19.

Marzal, E.; Sebastia, L.; and Onaindia, E. 2014. On the use of temporal landmarks for planning with deadlines. In *ICAPS*.

Mnih, V. *et al.*. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *ECP*.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks Revisited. In *AAAI*.

Richter, S.; and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR*, 39: 127–177.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2019. Computing programs for generalized planning using a classical planner. *Artificial Intelligence*.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2021. Generalized Planning as Heuristic Search. In *ICAPS*.

Seipp, J.; and Helmert, M. 2018. Counterexample-guided Cartesian abstraction refinement for classical planning. *JAIR*, 62: 535–577.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011. A new representation and associated algorithms for generalized planning. *Artificial Intelligence*, 175(2): 615–647.

Srivastava, S. *et al.*. 2011. Directed Search for Generalized Plans Using Classical Planners. In *ICAPS*.

Ståhlberg, S.; Francès, G.; and Seipp, J. 2021. Learning Generalized Unsolvability Heuristics for Classical Planning. In *IJCAI*.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Toyer, S.; Thiébaux, S.; Trevizan, F.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *JAIR*, 68: 1–68.

Winner, E.; and Veloso, M. 2003. DISTILL: Learning Domain-Specific Planners by Example. In *ICML*.