Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)
Sister Conferences Best Papers Track

Computing Programs for Generalized Planning as Heuristic Search (Extended Abstract)*

Javier Segovia-Aguas[1] , **Sergio Jiménez Celorrio**[2] and **Anders Jonsson**[1]
[1]Dept. Information and Communication Technologies, Universitat Pompeu Fabra, Spain
[2]Valencian Research Institute for Artificial Intelligence, Universitat Politècnica de València, Spain
javier.segovia@upf.edu, serjice@dsic.upv.es, anders.jonsson@upf.edu

Abstract

Although *heuristic search* is one of the most successful approaches to classical planning, this planning paradigm does not apply straightforwardly to *Generalized Planning* (GP). This paper adapts the *planning as heuristic search* paradigm to the particularities of GP, and presents the first native heuristic search approach to GP. First, the paper defines a program-based solution space for GP that is independent of the number of planning instances in a GP problem, and the size of these instances. Second, the paper defines the BFGP algorithm for GP, that implements a best-first search in our program-based solution space, and that is guided by different evaluation and heuristic functions.

1 Introduction

Heuristic search is one of the most successful approaches to classical planning [Bonet and Geffner, 2001; Hoffmann, 2003; Helmert, 2006; Lipovetzky and Geffner, 2017]. Unfortunately, it is not straightforward to adopt state-of-the-art search algorithms and heuristics from classical planning to *Generalized Planning* (GP). The *planning as heuristic search* approach traditionally addresses the computation of sequential plans implementing a grounded state-space search. GP requires however reasoning about the synthesis of algorithm-like solutions that, in addition to action sequences, contain branching and looping constructs [Winner and Veloso, 2003; Hu and Levesque, 2011; Siddharth *et al.*, 2011; Hu and De Giacomo, 2013; Segovia-Aguas *et al.*, 2016; Illanes and McIlraith, 2019; Francès *et al.*, 2021]. Furthermore, GP aims to synthesize solutions that generalize to a (possibly infinite) set of planning instances. The domain of the state variables may then be large, making unfeasible the grounding traditionally implemented by of off-the-shelf classical planners.

This paper adapts the *planning as heuristic search* paradigm to the particularities of GP, and presents the first native heuristic search approach to GP. Given a GP problem, that comprises an input set of classical planning instances from a given domain, our *GP as heuristic search* approach computes an algorithm-like plan that solves the full set of input instances. The contribution of the paper is two-fold:

- *A tractable solution space for GP*. We leverage the computational models of the *Random-Access Machine* [Skiena, 1998] and the *Intel x86* FLAGS register [Dandamudi, 2005] to define an innovative program-based solution space that is independent of the number of input planning instances in a GP problem, and the size of these instances (i.e. the number of state variables and their domain size).
- *A heuristic search algorithm for GP*. We present the BFGP algorithm that implements a best-first search in our solution space for GP. We also define several evaluation and heuristic functions to guide BFGP; evaluating these functions does not require to ground states/actions in advance, so they allow to addressing GP problems where state variables have large domains (e.g. integers).

The paper is structured as follows. First we formalize the classical planning model. Then we show how to extend this model with a *Random-Access Machine* (RAM) and formalize GP with *planning programs*, our representation formalism for GP solutions. Last, we describe the implementation of our *GP as heuristic search* approach and report results on its empirical performance. More details on the *GP as heuristic search* approach can be found in Segovia-Aguas *et al.* [2021].

2 Classical Planning

Let X be a set of *state variables*, each $x \in X$ with domain D_x. A *state* is a total assignment of values to the set of state variables. For a variable subset $X' \subseteq X$, let $D[X'] = \times_{x \in X'} D_x$ denote its joint domain. The state space is then $S = D[X]$. Given a state $s \in S$ and a subset of variables $X' \subseteq X$, let $s_{|X'} = \langle x_i = v_i \rangle_{x_i \in X'}$ be the *projection* of s onto X' i.e. the partial state defined by the values that s assigns to the variables in X'. The *projection* of s onto X' defines the subset of states $\{s \mid s \in S, s_{|X'} \subseteq s\}$ that are consistent with the corresponding partial state.

Let A be a set of deterministic actions. An action $a \in A$ has an associated set of variables $par(a) \subseteq X$, called *parameters*, and is characterized by two functions: an *applicability function* $\rho_a : D[par(a)] \rightarrow \{0, 1\}$, and a *successor function* $\theta_a : D[par(a)] \rightarrow D[par(a)]$. Action a is applicable in a

---

*This is an extended abstract of the "Generalized Planning as Heuristic Search" paper that appeared at ICAPS 2021 conference.

given state s iff $\rho_a(s_{|par(a)})$ equals 1, and results in a *successor* state $s' = s \oplus a$, that is built replacing the values that s assigns to variables in $par(a)$ with the values specified by $\theta_a(s_{|par(a)})$.

A *classical planning instance* is a tuple $P = \langle X, A, I, G\rangle$, where X is a set of state variables, A is a set of actions, $I \in S$ is an initial state, and G is a goal condition on the state variables that induces the subset of *goal states* $S_G = \{s \mid s \vDash G, s \in S\}$. Given P, a *plan* is an action sequence $\pi = \langle a_1, \ldots, a_m\rangle$ whose execution induces a *trajectory* $\tau = \langle s_0, a_1, s_1, \ldots, a_m, s_m\rangle$ such that, for each $1 \leq i \leq m$, a_i is applicable in s_{i-1} and results in the successor state $s_i = s_{i-1} \oplus a_i$. A plan π *solves* P if the execution of π in $s_0 = I$ finishes in a goal state, i.e. $s_m \in S_G$.

3 Generalized Planning as Heuristic Search

This work builds on top of the inductive formalism for GP, where a GP problem is a set of classical planning instances with a common structure. Here we describe our heuristic search approach to GP.

3.1 Classical Planning with a RAM

To define a tractable solution space for GP, that is independent of the number (and domain size) of the planning state variables, we extend the classical planning model with: (i), a set of pointers over the state variables (ii), their primitive operations and (iii), two Boolean (the *zero* and *carry* FLAGS) to store the result of the primitive operations over pointers.

Formally a *pointer* is a finite domain variable $z \in Z$ with domain $D_z = [0..|X|)$. To formalize the primitive operations over pointers we leverage the notion of the RAM; the RAM is an abstract computation machine, that is polynomially equivalent to a Turing machine, and that enhances a multiple-register *counter machine* with indirect memory addressing [Boolos *et al.*, 2002]. The indirect memory addressing of the RAM enables the definition of programs that access an unbounded number of state variables. Let $z \in Z$ be a pointer over the state variables, and $*z$ the content of that pointer, our *GP as heuristic search* implements the following primitive operations over pointers: $\{\texttt{inc}(z_1), \texttt{dec}(z_1), \texttt{cmp}(z_1, z_2), \texttt{cmp}(*z_1, *z_2), \texttt{set}(z_1, z_2) \mid z_1, z_2 \in Z\}$. Respectively, these primitive operations increment/decrement a pointer, compare two pointers (or their content), and set the value of a pointer z_1 to another pointer z_2. Each primitive operation also updates two Boolean $Y = \{y_z, y_c\}$, the *zero* and *carry* FLAGS, according to the result (denoted here by `res`) of that primitive operation:

$$\begin{aligned}
inc(z_1) &\implies res := z_1 + 1,\\
dec(z_1) &\implies res := z_1 - 1,\\
cmp(z_1, z_2) &\implies res := z_1 - z_2,\\
cmp(*z_1, *z_2) &\implies res := *z_1 - *z_2,\\
set(z_1, z_2) &\implies res := z_2,\\
y_z &:= (res == 0),\\
y_c &:= (res > 0).
\end{aligned}$$

Given a classical planning instance $P = \langle X, A, I, G\rangle$, its extension with a RAM of $|Z|$ pointers and two FLAGS is the classical planning instance $P_Z = \langle X_Z, A_Z, I_Z, G\rangle$, where the set of the state variables is extended with the FLAGS and the pointers, $X_Z = X \cup Y \cup Z$. The set of actions A_Z comprises the primitive pointer operations and the original actions A, but replacing their parameters by pointers in Z. The initial state I_Z is extended to set the FLAGS to False, and the pointers to zero (by default). The goals of P_Z are the same as those of the original instance. An extended instance P_Z preserves the solution space of the original instance P [Segovia-Aguas *et al.*, 2021].

3.2 Generalized Planning with a RAM

A GP problem is a set of classical planning instances with a common structure. In this work the common structure is given by the RAM extension; it provides a set of different classical planning instances with a common set of FLAGS, pointers, and actions defined over those pointers.

Definition 1 (GP problem)**.** *A GP problem is a set of T classical planning instances $\mathcal{P} = \{P_Z^1, \ldots, P_Z^T\}$ that share the same subset of state variables $\{Y \cup Z\}$, and actions A_Z, but may differ in their state variables, initial state, and goals. Formally, $P_Z^1 = \langle X_Z^1, A_Z, I_Z^1, G_1\rangle, \ldots, P_Z^T = \langle X_Z^T, A_Z, I_Z^T, G_T\rangle$ where $\forall_t\{Y \cup Z\} \subset X_Z^t$, $1 \leq t \leq T$.*

Representations of GP solutions range from *programs* [Winner and Veloso, 2003; Jimenez and Jonsson, 2015; Segovia-Aguas *et al.*, 2019] and *generalized policies* [Martín and Geffner, 2004], to *finite state controllers* [Bonet *et al.*, 2010; Segovia-Aguas *et al.*, 2019] or *formal grammars* and *hierarchies* [Nau *et al.*, 2003; Segovia-Aguas *et al.*, 2017]. Each representation has its own expressiveness capacity, as well as its own computation complexity. We can however define a common condition under which a generalized plan is considered a solution to a GP problem [Jiménez *et al.*, 2019]. First, let us define $exec(\Pi, P) = \langle a_1, \ldots, a_m\rangle$ as the sequential plan produced by the execution of a generalized plan Π, on a classical planning instance P.

Definition 2 (GP solution)**.** *A generalized plan Π solves a GP problem $\mathcal{P}$ iff for every classical planning instance $P_t \in \mathcal{P}$, $1 \leq t \leq T$, it holds that $exec(\Pi, P_t)$ solves P_t.*

In this work we represent GP solutions as *planning programs* [Segovia-Aguas *et al.*, 2019]. 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 $0 \leq i' < i$ or $i + 1 < i' < n$, and y is a proposition.
- A *termination instruction* $w_i = \mathsf{end}$. The last instruction of a program Π is always $w_{n-1} = \mathsf{end}$.

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 line $0 \leq i < n$. Given a program state (s, i), the execution of a programmed instruction w_i is defined as:

- If $w_i \in A$, the new program state is $(s', i + 1)$, where $s' = s \oplus w_i$ is the *successor* when applying w_i in s.

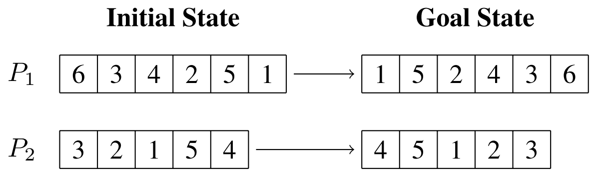

Figure 1: Classical planning instances for reversing the content of two lists by swapping their elements.

```
REVERSE
0. swap($z_1$,$z_2$)
1. inc($z_1$)
2. dec($z_2$)
3. cmp($z_2$,$z_1$)
4. goto(0,$\neg(\neg y_z \wedge \neg y_c)$)
5. end
```

Figure 2: *Generalized plan* for reversing a list, no matter its length.

- If $w_i = \mathsf{go}(i', !y)$, the new program state is $(s, i+1)$ if y holds in s, and (s, i') otherwise[1]. Proposition y can be the result of an arbitrary expression on the state variables [Lotinac *et al.*, 2016].
- If $w_i = \mathsf{end}$, program execution terminates.

To execute a planning program Π on a planning instance P, the initial program state is set to $(I, 0)$, i.e. the initial state of P and the first program line of Π. A planning 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 $s \in S_G$.

Example. Figure 1 shows the initial state and goals of two classical planning instances, $P_1 = \langle X, A, I_1, G_1 \rangle$ and $P_2 = \langle X, A, I_2, G_2 \rangle$, for reversing two lists. Both instances can be defined with a set of state variables $X = \{x_i\}_1^l$, where an integer is assigned to every x_i and l is the list length, and a set of $swap(x_i, x_j)$ actions that swap the content of two state variables. An example solution plan for P_1 is $\pi_1 = \langle swap(x_1, x_6), swap(x_2, x_5), swap(x_3, x_4) \rangle$ while $\pi_2 = \langle swap(x_1, x_5), swap(x_2, x_4) \rangle$ is a solution plan for P_2. This set of two classical planning instances can be extended with a RAM, and the resulting set $\mathcal{P} = \{P_Z^1, P_Z^2\}$ is an example GP problem. Figure 2 shows the generalized plan, represented as a planning program with six lines and two pointers $Z = \{z_1, z_2\}$. Initially, the first list element is pointed by z_1 (default) and the last element by z_2. The elements pointed by z_1 and z_2 are swapped in Line 0, and the pointers are moved forward and backward once respectively (Lines 1-2). Then, z_2 and z_1 are compared in Line 3, and the FLAGS are updated as follows: $y_z = ((z_2 - z_1) == 0)$ and $y_c = ((z_2 - z_1) > 0)$. This is repeated until $\neg(\neg y_z \wedge \neg y_c)$ is falsified, which only occurs when z_2 is smaller than z_1, thus reversing any list, no matter its length or content.

3.3 The BFGP Algorithm for GP

Given a GP problem $\mathcal{P} = \{P_1, \ldots, P_T\}$, a number of program lines n, and a number of pointers $|Z|$, the BFGP algorithm outputs a planning program Π that solves every classical planning instance $P_t \in \mathcal{P}$, $1 \leq t \leq T$. Otherwise BFGP reports that there is no solution within the given number of program lines and pointers.

Search space. BFGP searches in the space of planning programs with n program lines, and $|Z|$ pointers, that can be built with the shared set of actions A_Z, and goto instructions that are exclusively conditioned on the value of FLAGS $Y = \{y_z, y_c\}$. Since only the primitive operations over pointers update FLAGS Y, we have an observation space of $2^{|Y|} \times 2|Z|^2$ state observations implemented with only $|Y|$ Boolean variables. The four joint values of $\{y_z, y_c\}$ model then a large space of observations, including $= 0, \neq 0, < 0, > 0, \leq 0, \geq 0$ as well as relations $=, \neq, <, >, \leq, \geq$ on variable pairs.

Search algorithm. BFGP implements a Best First Search (BFS) that starts with an empty planning program. To generate a tractable set of successor nodes, child nodes in the search tree are restricted to planning programs that result from programming the PC^{MAX} line (i.e. the maximum line reached after executing the current program on the classical planning instances in $\mathcal{P}$). This procedure for successor generation guarantees that duplicate successors are not generated. BFGP is a *frontier search* algorithm, meaning that, to reduce memory requirements, BFGP stores only the open list of generated nodes, but not the closed list of expanded nodes [Korf *et al.*, 2005]. BFS sequentially expands the best node in a priority queue (aka *open list*) sorted by an evaluation/heuristic function. If the planning program Π solves all the instances $P_t \in \mathcal{P}$, then search ends, and Π is a valid solution for the GP problem $\mathcal{P}$.

Evaluation functions. BFGP exploits two different sources of information to guide the search in the space of candidate planning programs:

- *The program structure*. These are evaluation functions computed in linear time in the size of program Π.
  - $f_1(\Pi)$, the number of *goto* instructions in Π.
  - $f_2(\Pi)$, number of *undefined* program lines in Π.
  - $f_3(\Pi)$, the number of repeated actions in Π.
- *The program performance*. These functions assess the performance of Π executing it on each of the classical planning instances $P_t \in \mathcal{P}$, $1 \leq t \leq T$; the execution of a planning program on a classical planning instance is a deterministic procedure that requires no variable instantiation. If the execution of Π on an instance $P_t \in \mathcal{P}$ fails, this means that the search node corresponding to the planning program Π is a dead-end, and hence it is not added to the open list:
  - $h_4(\Pi, \mathcal{P}) = n - PC^{MAX}$, where PC^{MAX} is the maximum program line that is eventually reached after executing Π on all the instances in $\mathcal{P}$.
  - $h_5(\Pi, \mathcal{P}) = \sum_{P_t \in \mathcal{P}} \sum_{v \in G_t} (s_t[v] - G_t[v])^2$. This function accumulates, for each instance $P_t \in \mathcal{P}$,

---

[1] We adopt the convention of jumping to line i' whenever y is *false*, inspired by jump instructions in the *Random-Access Machine* that jump when a register equals zero.

Domain	$n, \|Z\|$	f_1				f_2				f_3			
		Time	Mem.	Exp.	Eval.	Time	Mem.	Exp.	Eval.	Time	Mem.	Exp.	Eval.
T. Sum	5, 2	0.24	4.2	4.8K	5.8K	0.30	**3.8**	6.4K	6.4K	0.13	3.9	2.2K	2.8K
Corridor	7, 2	3.04	6.6	12.4K	26.7K	**0.41**	**3.8**	**2.2K**	**2.2K**	3.66	4.2	24.2K	25.9K
Reverse	7, 3	82	61	0.28M	0.57M	181	**4.0**	0.95M	0.95M	170	23	0.75M	0.84M
Select	7, 3	198	110	0.83M	1.10M	27	**3.9**	0.12M	0.12M	76.49	11	0.34M	0.38M
Find	7, 3	195	175	0.50M	1.36M	271	**4.0**	1.46M	1.46M	**86**	12	**0.41M**	**0.45M**
Fibonacci	8, 3	**496**	922	**2.48M**	**6.79M**	1,082	**3.9**	11.8M	11.8M	TO	-	-	-
Gripper	8, 4	TO	-	-	-	3,439	**4.1**	19.9M	19.9M	TO	-	-	-
Sorting	9, 3	TO	-	-	-	TO	-	-	-	**3,143**	**711**	**19.5M**	**22.9M**
Average		162	213	0.68M	1.64M	714	3.9	4.89M	4.89M	580	128	3.51M	4.09M

Domain	$n, \|Z\|$	h_4				h_5				f_6			
		Time	Mem.	Exp.	Eval.	Time	Mem.	Exp.	Eval.	Time	Mem.	Exp.	Eval.
T. Sum	5, 2	0.10	**3.8**	1.6K	1.6K	**0.09**	3.9	**1.2K**	**1.4K**	0.45	4.8	7.4K	7.4K
Corridor	7, 2	1.09	3.9	5.3K	5.4K	5.29	4.1	30.3K	31.3K	6.16	7.6	35.3K	35.3K
Reverse	7, 3	205	4.2	0.88M	0.88M	**1.46**	4.2	**4.9K**	**6.3K**	369	230	1.57M	1.65M
Select	7, 3	**0.80**	**3.9**	**3.0K**	**4.2K**	94	5.7	0.34M	0.35M	255	155	1.06M	1.14M
Find	7, 3	415	4.4	1.76M	1.76M	140	7.0	0.58M	0.59M	423	244	1.76M	1.77M
Fibonacci	8, 3	TO	-	-	-	1,500	120	11.3M	11.8M	TO	-	-	-
Gripper	8, 4	TO	-	-	-	**83**	5.5	**0.34M**	**0.35M**	TO	-	-	-
Sorting	9, 3	TO	-	-	-	TO	-	-	-	TO	-	-	-
Average		125	4.0	0.53M	0.53M	260	21	1.80M	1.88M	211	128	0.89M	0.92M

Table 1: We report the number of program lines n, and pointers $|Z|$ per domain, and for each evaluation/heuristic function, CPU (secs), memory peak (MBs), and the numbers of expanded and evaluated nodes. TO stands for Time-Out (>1h of CPU). Best results in bold.

Dom.	BFGP(f_1, h_5) / BFGP(h_5, f_1)			
	T.	M.	Exp.	Eval.
T. Sum	0.2/**0.1**	4.3/**3.8**	2.8K/**1.1K**	4.8K/**1.4K**
Corr.	**3.2**/4.5	6.6/**5.9**	**6.5K**/26.0K	**21.6K**/27.5K
Rev.	63/**1.4**	52/**4.7**	81.9K/**3.7K**	0.3M/**7.7K**
Sel.	203/**80**	110/**7.0**	0.6M/**0.3M**	0.9M/**0.3M**
Find	313/**162**	176/**14**	0.9M/**0.7M**	1.5M/**0.7M**
Fibo.	528/**22**	828/**32**	1.4M/**75K**	5.3M/**0.2M**
Grip.	TO/**6.9**	**-/10**	**-/5.8K**	**-/37.3K**
Sort.	TO/**713**	**-/730**	**-/4.4M**	**-/4.5M**

Table 2: CPU time (secs), memory peak (MBs), num. of expanded and evaluated nodes. Best results in bold.

Dom.	Inst.	Time_∞	Mem_∞	Time	Mem
T. Sum	44,709	1,066.74	53MB	**574.08**	**47MB**
Corr.	1,000	0.23	5.0MB	**0.15**	**4.7MB**
Rev.	50	37.96	5.2GB	**2.70**	**0.3GB**
Sel.	50	144.75	19.6GB	**2.29**	**33MB**
Find	50	114.55	19.6GB	**2.12**	**33MB**
Fibo.	33	**0.00**	4.2MB	**0.00**	**3.9MB**
Grip.	1,000	2.71	0.1GB	**1.65**	**0.1GB**
Sort.	20	272.06	15.2GB	**52.04**	**3.8MB**

Table 3: Validation set, CPU-time (secs) and memory peak for program validation, with/out *infinite program* detection.

the *euclidean distance* of state s_t to the goal state variables G_t. The state s_t is obtained applying the sequence of actions $exec(\Pi, P_t)$ to the initial state I_t of that problem $P_t \in \mathcal{P}$. Computing $h_5(\Pi, P_t)$ requires that goals are specified as a partial state. Note that for Boolean variables the squared difference becomes a simple goal counter.

- $f_6(\Pi, \mathcal{P}) = \sum_{P_t \in \mathcal{P}} |exec(\Pi, P_t)|$, where $exec(\Pi, P_t)$ is the sequence of actions induced from executing the planning program Π on the planning instance P_t.

All these functions are *cost functions* (i.e. smaller values are preferred). Functions $h_4(\Pi, \mathcal{P})$ and $h_5(\Pi, \mathcal{P})$ are cost-to-go *heuristics*; they provide an estimate on how far a program is from solving the given GP problem. Functions $h_4(\Pi, \mathcal{P})$, $h_5(\Pi, \mathcal{P})$, and $f_6(\Pi, \mathcal{P})$ aggregate several costs that could be expressed as a combination of different functions, e.g. *sum*, *max*, average, weighted average, etc.

4 Evaluation

We evaluated the performance of our *GP as heuristic search* approach in eight domains [Segovia-Aguas *et al.*, 2021]. Experiments performed in an Ubuntu 20.04 LTS, with AMD® Ryzen 7 3700x 8-core processor × 16 and 32GB of RAM.

Table 1 summarizes the results of BFGP with our six different evaluation/heuristic functions (best results in bold): i) f_2 and h_5 exhibited the best coverage; ii) there is no clear dominance of a *structure* evaluation function, f_2 has the best memory consumption while f_3 is the only structural function that solves *Sorting*; iii) the *performance*-based function h_5 dominates h_4 and f_6. Interestingly, the base performance of BFGP with a single evaluation/heuristic function is improved guiding BFGP with a cost-to-go heuristic function and breaking ties with a structural evaluation function (or vice versa). Table 2 shows the performance of $BFGP(f_1, h_5)$ and its reversed configuration $BFGP(h_5, f_1)$ which actually resulted in the overall best configuration solving all domains.

The solutions synthesized by BFGP were successfully validated. Table 3 reports the CPU time, and peak memory, yield when running the solutions synthesized by $BFGP(h_5, f_1)$ on a validation set. The largest CPU-times and memory peaks correspond to the configuration that implements the detection of *infinite programs*, which requires saving states to detect whether they are revisited during execution. Skipping this mechanism allows to validate non-infinite programs faster [Segovia-Aguas *et al.*, 2020].

5 Conclusion

We presented the first native heuristic search approach for GP that leverages solutions represented as automated planning programs. We believe this work builds a stronger connection between the two closely related areas of planning and *program synthesis* [Gulwani *et al.*, 2017; Alur *et al.*, 2018]. A wide landscape of effective techniques, coming from heuristic search and classical planning, promise to improve the base performance of our approach [Segovia-Aguas *et al.*, 2022]. For instance, better estimates may be obtained by building on top of better informed planning heuristics [Francès, 2017].

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 supported by Spanish grant PID2019-108141GB-I00.

References

[Alur *et al.*, 2018] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Communications of the ACM*, 61(12):84–93, 2018.

[Bonet and Geffner, 2001] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[Bonet *et al.*, 2010] Blai Bonet, Hector Palacios, and Hector Geffner. Automatic derivation of finite-state machines for behavior control. In *AAAI*, volume 24, 2010.

[Boolos *et al.*, 2002] George S Boolos, John P Burgess, and Richard C Jeffrey. *Computability and logic*. Cambridge university press, 2002.

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

[Francès *et al.*, 2021] Guillem Francès, Blai Bonet, and Hector Geffner. Learning general planning policies from small examples without supervision. In *AAAI*, 2021.

[Francès, 2017] Guillem Francès. *Effective planning with expressive languages*. PhD thesis, Universitat Pompeu Fabra, 2017.

[Gulwani *et al.*, 2017] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends in Programming Languages*, 2017.

[Helmert, 2006] Malte Helmert. The Fast Downward Planning System. *JAIR*, 26:191–246, 2006.

[Hoffmann, 2003] Jörg Hoffmann. The metric-ff planning system: Translating "ignoring delete lists" to numeric state variables. *JAIR*, 20:291–341, 2003.

[Hu and De Giacomo, 2013] Yuxiao Hu and Giuseppe De Giacomo. A generic technique for synthesizing bounded finite-state controllers. In *ICAPS*, 2013.

[Hu and Levesque, 2011] Yuxiao Hu and Hector J. Levesque. A correctness result for reasoning about one-dimensional planning problems. In *IJCAI*, pages 2638–2643, 2011.

[Illanes and McIlraith, 2019] León Illanes and Sheila A McIlraith. Generalized planning via abstraction: arbitrary numbers of objects. In *AAAI*, volume 33, pages 7610–7618, 2019.

[Jimenez and Jonsson, 2015] Sergio Jimenez and Anders Jonsson. Computing plans with control flow and procedures using a classical planner. In *International Symposium on Combinatorial Search*, 2015.

[Jiménez *et al.*, 2019] Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. A review of generalized planning. *KER*, 34:1–28, 2019.

[Korf *et al.*, 2005] Richard E Korf, Weixiong Zhang, Ignacio Thayer, and Heath Hohwald. Frontier search. *Journal of the ACM*, 52(5):715–748, 2005.

[Lipovetzky and Geffner, 2017] Nir Lipovetzky and Hector Geffner. Best-first width search: Exploration and exploitation in classical planning. In *AAAI*, volume 31, 2017.

[Lotinac *et al.*, 2016] Damir Lotinac, Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Automatic generation of high-level state features for generalized planning. In *IJCAI*, pages 3199–3205, 2016.

[Martín and Geffner, 2004] Mario Martín and Hector Geffner. Learning generalized policies from planning examples using concept languages. *Applied Intelligence*, 20:9–19, 2004.

[Nau *et al.*, 2003] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *JAIR*, 20:379–404, 2003.

[Segovia-Aguas *et al.*, 2016] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generalized planning with procedural domain control knowledge. In *ICAPS*, 2016.

[Segovia-Aguas *et al.*, 2017] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generating context-free grammars using classical planning. In *IJCAI*, 2017.

[Segovia-Aguas *et al.*, 2019] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Computing programs for generalized planning using a classical planner. *Artificial Intelligence*, 272:52–85, 2019.

[Segovia-Aguas *et al.*, 2020] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generalized planning with positive and negative examples. In *AAAI*, volume 34, pages 9949–9956, 2020.

[Segovia-Aguas *et al.*, 2021] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generalized planning as heuristic search. In *ICAPS*, volume 31, pages 569–577, 2021.

[Segovia-Aguas *et al.*, 2022] Javier Segovia-Aguas, Sergio Jiménez, Anders Jonsson, and Laura Sebastiá. Scaling-up generalized planning as heuristic search with landmarks. *arXiv preprint arXiv:2205.04850*, 2022.

[Siddharth *et al.*, 2011] Srivastava Siddharth, Immerman Neil, Zilberstein Shlomo, and Zhang Tianjiao. Directed search for generalized plans using classical planners. In *ICAPS*, volume 21, 2011.

[Skiena, 1998] Steven S Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.

[Winner and Veloso, 2003] Elly Winner and Manuela Veloso. Distill: Learning domain-specific planners by example. In *ICML*, pages 800–807, 2003.