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

Javier Segovia-Aguas\textsuperscript{1}, Sergio Jiménez Celorrio\textsuperscript{2} and Anders Jonsson\textsuperscript{1}

\textsuperscript{1}Dept. Information and Communication Technologies, Universitat Pompeu Fabra, Spain
\textsuperscript{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 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 \(s|_{X'} = (x_i = v_i)_{x_i \in X'}\) 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 \(\text{par}(a) \subseteq X\), called parameters, and is characterized by two functions: an applicability function \(\rho_a : D[\text{par}(a)] \to \{0, 1\}\), and a successor function \(\theta_a : D[\text{par}(a)] \to D[\text{par}(a)]\). Action \(a\) is applicable in a
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 \( D_z = \{0, \ldots, |X|\} \). 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 \( \ast z \) the content of that pointer, our GP as heuristic search implements the following primitive operations over pointers:

- \( \text{inc}(z_1) \) returns \( z_1 + 1 \),
- \( \text{dec}(z_1) \) returns \( z_1 - 1 \),
- \( \text{cmp}(z_1, z_2) \) returns \( z_1 - z_2 \),
- \( \text{cmp}(\ast z_1, \ast z_2) \) returns \( \ast z_1 - \ast z_2 \),
- \( \text{set}(z_1, z_2) \) returns \( z_2 \),
- \( y_i := (\text{res} = 0) \),
- \( y_i := (\text{res} > 0) \).

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 classical planning instances \( \mathcal{P} = \{P^1_Z, \ldots, P^T_Z\} \) 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, \( P^i_Z = \langle X^i_Z, A^i_Z, I^i_Z, G^i \rangle, \ldots, P^T_Z = \langle X^T_Z, A^T_Z, I^T_Z, G^T \rangle \) where \( \forall i, (Y \cup Z) \subseteq X^i_Z, 1 \leq i \leq T \).

Representations of GP solutions range from programs [Winner and Veloso, 2003; Jimenez and Johnson, 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 \( \text{exec}(\Pi, P) = \langle a_1, \ldots, a_m \rangle \) as the sequential plan produced by the execution of a generalized plan \( \Pi \) on a classical planning instance \( P \).

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

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 \( w_i = \text{go}(i', y) \), where \( i' \) is a program line \( 0 \leq i' < i \) or \( i + 1 < i' < n \), and \( y \) is a proposition.
- A termination instruction \( w_i = \text{end} \). The last instruction of a program \( \Pi \) is always \( w_{n-1} = \text{end} \).

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 \( s \).
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 \( \Pi \) 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 \( \mathcal{A}_Z \), 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 \( =, \neq, <, >, \leq, \geq \) 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 \( \Pi \) solves all the instances \( P_t \in \mathcal{P} \), then search ends, and \( \Pi \) 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 \( \Pi \):
  - \( f_1(\Pi) \), the number of goto instructions in \( \Pi \).
  - \( f_2(\Pi) \), number of undefined program lines in \( \Pi \).
  - \( f_3(\Pi) \), the number of repeated actions in \( \Pi \).

- **The program performance.** These functions assess the performance of \( \Pi \) 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 \( \Pi \) on an instance \( P_t \in \mathcal{P} \) fails, this means that the search node corresponding to the planning program \( \Pi \) 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 \( \Pi \) on all the instances in \( \mathcal{P} \).
  - \( h_5(\Pi, \mathcal{P}) = \sum_{P_t \in \mathcal{P}} \sum_{v \in \mathcal{G}_t} (s_t[v] - G_t[v])^2 \). This function accumulates, for each instance \( P_t \in \mathcal{P} \),

---

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.
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 $\text{exec}(\Pi, P_t)$ 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 count.

$$-f_6(\Pi, P) = \sum_{\pi_i \in \Pi} |\text{exec}(\Pi, \pi_i)|,$$

where $\text{exec}(\Pi, P_t)$ is the sequence of actions induced from executing the planning program $\Pi$ on the planning instance $P_t$.

All these functions are cost functions (i.e., smaller values are preferred). Functions $h_4(\Pi, P)$ and $h_5(\Pi, P)$ are cost-to-go heuristics; they provide an estimate on how far a program is from solving the given GP problem. Functions $h_4(\Pi, P)$, $h_5(\Pi, P)$, and $f_6(\Pi, P)$ 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/heuristics 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 $f_5$ 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


