Generalized planning as heuristic search: A new planning search-space that leverages pointers over objects

Javier Segovia-Aguas a,*, Sergio Jiménez b, Anders Jonsson a

a Universitat Pompeu Fabra, Spain
b Universitat Politècnica de València, Spain

ARTICLE INFO

Keywords:
Generalized planning
Classical planning
Heuristic search
Planning and learning
Domain-specific control knowledge
Program synthesis
Programming by example

ABSTRACT

Planning as heuristic search is one of the most successful approaches to classical planning but unfortunately, it does not trivially extend to Generalized Planning (GP); GP aims to compute algorithmic solutions that are valid for a set of classical planning instances from a given domain, even if these instances differ in their number of objects, the initial and goal configuration of these objects and hence, in the number (and possible values) of the state variables. State-space search, as it is implemented by heuristic planners, becomes then impractical for GP. In this paper we adapt the planning as heuristic search paradigm to the generalization requirements of GP, and present the first native heuristic search approach to GP. First, the paper introduces a new pointer-based solution space for GP that is independent of the number of classical planning instances in a GP problem and the size of those instances (i.e. the number of objects, state variables and their domain sizes). Second, the paper defines an upgraded version of our GP algorithm, called Best-First Generalized Planning (BFGP), that implements a best-first search in our pointer-based solution space for GP. Lastly, the paper defines a set of evaluation and heuristic functions for BFGP that assess the structural complexity of the candidate GP solutions, as well as their fitness to a given input set of classical planning instances. The computation of these evaluation and heuristic functions does not require grounding states or actions in advance. Therefore our GP as heuristic search approach can handle large sets of state variables with large numerical domains, e.g. integers.

1. Introduction

Automated Planning (AP) is a model-based approach to the control of autonomous systems. In more detail, AP explores model-based computations to generate sequences of actions (a.k.a. plans) that accomplish defined objectives across various domains. In AP, plans are typically generated by taking into account a model encompassing initial conditions, available actions, and the goals to be accomplished. There is a wide palette of different AP models that deal with partial state observability, non-deterministic state transitions, durative actions, or temporarily extended goals [1,2]. Classical planning is the simplest AP model, which assumes that the system dynamics can be modeled as a finite, deterministic, and fully observable, transition system. In this kind of transition systems, classical planning studies the synthesis of sequences of actions that are able to transform an initial state into a state where a set of given goals is satisfied.

* Corresponding author.

E-mail addresses: javier.segovia@upf.edu (J. Segovia-Aguas), serjice@dsic.upv.es (S. Jiménez), anders.jonsson@upf.edu (A. Jonsson).

https://doi.org/10.1016/j.artint.2024.104097
Received 22 July 2021; Received in revised form 30 January 2024; Accepted 3 February 2024
Available online 15 February 2024
0004-3702/© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).
We illustrate the notion of a classical planning problem using the *blocksworld*, a popular classical planning domain, which consists of a set of blocks, a table, and a robot hand [3]. The robot hand can be empty or holding one block, and a block can be on top of another block or on the table. Different classical planning problems can be defined in the *blocksworld*, by changing the number of blocks, and their initial or goal configuration. A well-known *blocksworld* problem, since it is small but non-trivial, is the *Sussman anomaly* [4] (the leftmost problem shown in Fig. 1); in the *Sussman anomaly* there are three blocks that we labeled as \( b_0, b_1, \) and \( b_2 \). Initially block \( b_1 \) is on the table, \( b_2 \) on top of \( b_0 \), and \( b_0 \) on the table, and the goal is to stack the three blocks such that \( b_0 \) is on top of \( b_1 \), which in turn is on top of \( b_2 \). Fig. 1 shows three different classical planning problems from *blocksworld*. Each problem has a different number of blocks. The figure shows the blocks configuration for the initial state (left side) and goals (right side) of each problem. The leftmost classical planning problem corresponds to the *Sussman anomaly* problem.

Heuristic search is one of the most successful approaches to classical planning [5–8]. Winners of the International Planning Competition are often heuristic planners [9], and the workshop on Heuristics and Search for Domain-Independent Planning is one of the discussion forums with the longest tradition at the International Conference on Automated Planning and Scheduling (ICAPS), the major conference for research on AP. Briefly, the planning as heuristic search approach addresses the synthesis of sequential plans as a combinatorial search in the space of the states reachable from the initial state. This combinatorial search is usually implemented as a forward search, guided by heuristics that are automatically extracted from the representation of the planning problem. In the last decade a wide landscape of effective search algorithms, and heuristic functions, have been developed for classical planning [10–14]. Most of these heuristic search techniques are based on the notion of relaxed plan, a solution to a relaxation of the classical planning problem; the cost of the relaxed plan is an informative and cheap estimate of the actual cost-to-go for many different classical planning problems.

Generalized planning (GP) addresses the representation and computation of solutions that are valid for a set of classical planning instances from a given domain [15–24]. In the worst case each classical planning instance may require a completely different solution but in practice, many classical planning domains are known to have algorithmic solutions [25,26]. In other words, one can compute a single compact general solution that exploits some common structure of the different classical planning instances in a given domain. Generalized plans are then algorithmic solutions that supplement sequences of planning actions with control-flow. For example, a generalized plan that solves any classical planning instance from the *blocksworld* domain [3], no matter the actual number, or identity of the blocks, and no matter the initial and goal configuration of the blocks, can be compactly specified as follows: “*put all the blocks on the table and then, in a proper order, move each block to its goal placement*”.

Unfortunately search algorithms and heuristic functions from classical planning cannot be directly applied to GP. The computation of relaxed plans, as it is implemented by off-the-shelf heuristic planners, requires a pre-processing step for grounding states and actions. On the other hand, GP solutions must be able to generalize to (possibly infinite) sets of classical planning instances, with different sets of objects (i.e. state variables with different domain sizes and/or different number of state variables) as well as with different initial states and goal configuration for the objects. These particular generalization requirements of GP make it impractical to ground states and actions and hence, to directly apply the state-space search or the cost-to-go estimates of heuristic planners. What is more, the knowledge represented in a given set of classical planning instances may not be enough to specify an algorithmic solution that solves them all. For example, the classical planning instances from the *blocksworld* do not include representation features for explicitly specifying whether “*all blocks are on the table*”, or for specifying “*the proper order for moving the blocks to their goal placements*”. A key challenge in GP is then, given a set of planning instances, to automatically discover the representation features required for computing a compact and general solution for those instances.

Because of the algorithmic kind of generalized plans, GP is a promising research direction to help bridge the current gap between automated planning and programming [27]. Unfortunately, most of the work on GP inherits the STRIPS representation, in which states are represented using exclusively Boolean variables (i.e. propositions that specify the properties and relations of the world objects), and state-transitions correspond to actions for object manipulations. In this work we introduce a novel pointer-based representation for GP problems and solutions, that allows us to adapt the planning as heuristic search paradigm to GP. Our pointer-based representation is closer to common programming languages like Python, C++ or Java, while it also naturally applies to the STRIPS problems traditionally addressed by the AP community. Given a GP problem that comprises a finite set of classical planning instances from a given domain, our GP as heuristic search approach implements a combinatorial search to synthesize a program that solves the full set of input instances. With respect to previous work, our heuristic search approach to GP introduces the following contributions:

1. A pointer-based representation for GP problems and solutions. Our representation formalism is closer to common programming languages, while it also applies to object-centered representations (like STRIPS) traditionally used in AP.
2. A tractable solution-space for GP. We leverage the computational models of the Random-Access Machine [28] and the Intel x86 FLAGS register [29] to define a compact and general solution search space for GP. Remarkably our new search space for GP is independent of the number of input planning instances in a GP problem, and the size of these instances (i.e. the number of objects, state variables, and their domain sizes).

3. A heuristic search algorithm for GP with grounding-free evaluation/heuristic functions. We present the BFGP algorithm that implements a Best-First Search in our GP solution-space. We also define evaluation and heuristic functions for guiding BFGP. These functions assess the structural complexity of the candidate GP solutions, as well as their fitness to an input set of classical planning instances. Evaluating these functions does not require to ground states/actions in advance, so they apply to GP problems where state variables have large domains (e.g. integers).

4. A translator from the STRIPS fragment of PDDL to our pointer-based representation for GP. We automate the representation change from PDDL to pointer-based, and show several solutions to planning domains from the International Planning Competition [30] which are validated on large random instances.

A preliminary description of our GP as heuristic search approach previously appeared as ICAPS and IJCAI conference papers [31, 32]. In this work we extend the seminal ideas presented in the conference papers, and provide a more exhaustive evaluation of our GP as heuristic search approach. Compared to the conference papers, the present paper includes the following novel material:

- We formalize the notion of pointers over the set of objects of a planning problem, and introduce a pointer-based formalization for states, state-constraints, and planning action schemes. We also introduce a pointer-based formalization for classical planning problems and their solutions. We show that our pointer-based formalization naturally applies to the STRIPS planning tasks traditionally addressed in AP.
- We introduce the notion of a partially specified planning program, that refers to the sketch of an algorithmic planning solution, and that enables a better formalization of our GP search algorithm and heuristics functions.
- We provide new theoretical results of our heuristic search algorithm for GP, that include termination, soundness, completeness, and complexity proofs. We also implemented new evaluation functions for guiding our GP as heuristic search approach, and extended the empirical evaluation, including more results at a wider landscape of planning domains, to characterize better the performance of our GP as heuristic search approach.

The paper is structured as follows: Section 2 reviews previous research work related to computing policies and generalization in planning, and it pinpoints the main differences with our approach. Section 3 provides the planning models we rely on in this work (namely the classical planning model and the GP model) and it also presents the formalism we leverage for the representation of our GP solutions (i.e. planning programs and the Random Access Machine). Section 4 shows how to extend the classical planning model with a set of pointers over objects, and the corresponding primitive operations for manipulating these pointers. This extension allows us to define, in an agnostic manner, a set of features and a set of actions for computing GP solutions that can solve any instance from a given planning domain. Section 5 describes our GP as heuristic search approach; the section provides details on our solution space, evaluation functions, and heuristic search algorithm for GP. Section 6 presents the empirical evaluation of our GP as heuristic search approach and comparisons with the classical planning compilation for GP, that serves as a baseline. Finally, Section 7 wraps-up our work and discusses open issues and future work.

2. Related work

Here we first review previous work on GP according to the following three dimensions: problem representation, solution representation, and computational approach. Then, we connect the research work on GP with other relevant areas in AI, such as program synthesis, deep learning, and (deep) reinforcement learning. Last, we discuss the features that distinguish our GP as heuristic search approach from the reviewed related work.

Regarding problem representation, there are two different approaches for the specification of the set of classical planning instances that are comprised in a GP problem. The explicit approach, that enumerates every classical planning instance in a GP problem [33], and the implicit approach, that defines the constraints that hold for the set of classical planning instances of a GP problem. The implicit approach is of interest because it can compactly specify infinite sets of classical planning instances (e.g. the infinite set of the classical planning instances that belong to the blocks world domain) [34,35,20]. In addition to the set of classical planning instances, extra background knowledge can be specified with the aim of reducing the space of GP solutions. For instance, plan traces demonstrating how to solve some of the input instances [36–38], the full state space [22], the particular subset of state features that can be used for computing a generalized plan [39,40], negative examples that specify undesired behavior for the targeted GP solutions [41,22], or state invariants that any state in a given domain must satisfy [42].

With respect to solution representation, different formalisms appeared in the AP literature to represent solutions that are valid for a set of classical planning instances; sequential plans are used in conformant planning [43,44], conditional tree-like plans are used in contingent planning [45,2], or policies are used in FOND planning, as well as in MDP/POMDP planning [46]. In all these planning settings, a set of different classical planning instances, with different initial states, can be implicitly represented as a disjunctive formulae over the state variables. Different goals can also be considered by coding them as part of the state representation, e.g. using static state variables [33]. Further, the notion of feature in GP is related to the notion of state observation in the POMDP formalism, where observations depend on the current state and the action just taken [47]. With this regard it can be understood that GP solvers
The connection between AP and programming representations is not exclusive from our GP approach; programs of different kinds are proposed as an alternative to traditional planning action languages [50–52,27]. **GOLOG** programs have been also used to represent planning solutions, that could branch and loop, and that could contain non-deterministic parts [53]. Furthermore the semantics compatibility between **GOLOG** and PDDL [50] can be exploited and a PDDL planner can be embedded [54] to address the sub-problems that are combinatorial in nature. Since the early days of AI, hierarchies, LTL formulae, and policies, are also used to specify sketches of general planning solutions [55]. In the AP literature these solution sketches are often called domain-specific control knowledge, since they are traditionally used to control the planning process, and they apply to the entire set of classical planning instances that belong to a given domain [56,57,36,37]. Last but not least, algorithmic solutions, represented either as lifted policies, finite automata, or as programs with control-flow constructs for branching and looping, are used to represent GP solutions [34,58,39,59–61,15,33,21,24].

Regarding the computation of generalized plans, there are two main approaches to GP. The top-down/ offline approach considers the entire set of classical planning instances in a given GP problem as a single batch, and computes a solution plan that is valid for the full batch at once. A common approach for the offline computation of generalized plans is compiling the GP problem into another form of problem solving, and using an off-the-shelf solver to work out the compiled problem. For example, GP problems have been compiled into classical planning problems [61,33], conformant planning problems [39], LTL synthesis problems [62], POND planning problems [63,20], MAXSAT problems [22], or ASP problems [64]. The compilation approach is appealing because it leverages the latest advances of other well-founded scientific communities, with robust and scalable solvers. In addition, the computational complexity of some of these tasks is theoretically characterized with respect to structural features of the input problems, which may provide insights on the difficulty of the addressed GP problem. A weak point of the compilation approach is however the size of the compiled problems to be solved; solvers are usually sensitive to the size of the input problems. On the other hand, the bottom-up/online approach incrementalizes the set of classical planning instances in a GP problem [15,17,65]. Given a classical planning instance, a solution to that instance is computed and then, the solution is merged with solutions computed for the previous instances. The online approach is then appealing for handling GP problems that comprise large sets of classical planning instances. The main drawback of online approaches is dealing with the over-fitting produced by the individual processing of the different classical planning instances in a GP problem.

As noted by previous work on GP, the aims of GP are connected to program synthesis [33,20,62,31]. Program synthesis is a task traditionally studied by the computer-aided verification community [66], and that aims to compute programs such that they satisfy a given correctness specification [67–69]. Program synthesis follows the functional programming paradigm. This means that a program is a function composition, where each function in the composition maps its input parameters into a single output, and where looping is implemented with recursion. Work on program synthesis is classified according to how the correctness specification of a program is formulated. The programming by example (PBE) paradigm specifies the desired program behavior with a finite and non-empty set of ground input/output examples. This approach is related to the explicit representation of GP problems; a ground input/output example can be understood as the initial/goal state pair that represents a classical planning instance, and the instruction set of the functional programming language can be understood as the available actions for transforming an initial state into a goal state. Program synthesis also allows the implicit representation of the input correctness specifications, e.g. using fist-order formulae specified in SMTLIB,1 the formal language for SAT-Modulo Theories (SMT) [70]. The mainstream approach for program synthesis is to specify a formal grammar that incrementally enumerates the space of possible programs, and to leverage the satisfiability machinery of SMT solvers to validate whether a candidate program is actually a solution [71]. With this regard, work on theorem proving is also related to program synthesis, specially since SMT solvers allow the representation and satisfaction of first-order logic formulae [72]. Lastly, another popular trend in program synthesis is Programming by sketches that addresses program synthesis in the particular setting where a partially specified solution is provided as input [73].

Besides computational methods for formal verification and logic satisfaction, optimization methods (that are predominant in Machine Learning [74]) have also been applied to the computation of planning solutions that generalize. For instance, off-the-shelf Deep Learning (DL) tools, have been successfully applied to the computation of generalized policies for classical and probabilistic planning domains [75–77]. Generalized policies are a powerful solution representation formalism whose applicability goes beyond classical planning; generalized policies can represent planning solutions that deal with non-deterministic actions [78]. Further, generalized policies can represent solutions to planning problems whose aim is not the satisfaction of a given goal condition but the optimization of a given utility function [79]. The aims of GP are also related to Reinforcement Learning (RL) [80]; while the cited DL approaches can be viewed as off-line optimization approaches to GP, the RL paradigm can be viewed as an online optimization approach to GP. RL methods incrementally compute policies, by iteratively addressing a set of sequential decision-making episodes. In RL learning experience is however not given beforehand (learning experience is collected by the autonomous exploration of the state space), and RL assumes that there is an explicit notion of reward function (which helps to guide exploration towards the most promising portions of the state-space). Note that DL and DRL approaches learn policies, without requiring a symbolic representation of the state and the action space. This means that it is possible to compute (deep) policies that generalize from raw sensor data (e.g. sequences of images) [81,82]. The main disadvantage of computing solutions represented as deep policies is that they are black-

---

1 https://smtlib.cs.uiowa.edu/.
box models that lack transparency and explanation capacity, which makes it difficult to interpret the produced solutions. This is a strong requirement in application areas that require humans in the loop, such as health, law, or defense \[83\].

With regard to the reviewed related work, our **GP as heuristic planning** approach is framed as follows:

- **Numeric state variables.** Previous work on GP mainly followed the object-centered STRIPS representation. Addressing programming tasks with such representation is unpractical since it may require to encode all values in the domain of a state variable as objects. Other approaches, such as Qualitative Numeric Planning (QNP) \([48,49]\), handle large numeric state variables qualitatively with propositions to denote whether a variable is equal to zero. In this work we handle GP problems with integer state variables, which allow to naturally address diverse programming tasks as if they were GP problems.

- **Explicit problem representation.** In this work, a GP problem comprises the explicit enumeration of a finite set of classical planning instances to be solved. Interestingly our experimental results show that, in several domains, solving a small set of a few randomly generated classical planning instances, is enough to obtain a solution that generalizes to the infinite set of problems that belong to a given domain.

- **No background knowledge.** Our approach does not require any additional help such as state invariants, plans/traces/demonstrations, negative examples, or the specification of the subset of features to appear in the generalized plans. With this regard, we leverage the computational models of the Random-Access Machine \([28]\) and the Intel x86 FLAGS register \([29]\) to produce an agnostic set of state features that is shared for the different classical planning instances of a given domain (no matter their actual number of objects).

- **Generalized plans represented as structured programs.** Structured programming provides a white-box modeling paradigm that is widely popular. In this work we focus on generalized plans represented as structured programs, with control flow constructs for branching and looping the program execution flow. The application of a generalized plan on a particular classical planning instance is then a deterministic matching-function process, which makes it easier to define effective evaluation and heuristic functions. Further, the asymptotic complexity of structured programs can be assessed from their structure, which is helpful to establish preferences on the possible generalized plans.

- **Off-line satisfiability approach.** This work follows an off-line approach to GP that aims to compute, at once, a generalized plan that solves all the classical planning instances that are given as input. Because many heuristic search algorithms are easily extended to online versions, we believe that our GP as heuristic search approach is a stepping stone towards online approaches that can deal with larger sets of classical planning instances.

- **Native heuristic search for GP.** By **native** heuristic search, we mean that we defined a search space, evaluation/heuristic functions, and a search algorithm, that are specially targeted to GP. Our **GP as heuristic search** approach is related to an existing classical planning compilation for GP \([33]\). Our approach overcomes however the main drawback of the compilation whose search space grows exponentially with the number and domain size of the state variables; in practice, this drawback limited the applicability of the compilation to planning instances of small size since the performance of off-the-shelf classical planners is sensitive to the size of the input instances. Our experiments support this claim, and show that our BFGP algorithm significantly reduces the CPU-time required to compute and validate generalized plans, compared to the classical planning compilation approach to GP \([33]\).

### 3. Background

This section introduces the necessary notation to formalize our **GP as heuristic search** approach. First, the section formalizes the **classical planning** model and the **generalized planning** model. Then the section formalizes planning programs, our formalism for the compact representation of planning solutions. Lastly the section formalizes the **Random Access Machine**, given that our **GP as heuristic planning** approach borrows several mechanisms from this abstract computation machine.

#### 3.1. Classical planning

Our formalization of the classical planning model is similar to the abstract planning framework called **Finite Functional Planning**, introduced for the theoretical analysis of planning languages \([84]\). Let \(X\) be a set of **state variables**, where each variable \(x \in X\) has a domain \(D_x\). A **proposition** is a state variable \(x \in X\) with domain \(D_x = \{0, 1\}\), where \(x = 0\) and \(x = 1\) are interpreted as **false** and **true** assignments, respectively. A **state** \(s\) is a total assignment of values to the set of state variables, i.e. \(s = (x_0 = v_0, \ldots, x_N = v_N)\), such that \(\forall 0 \leq i < N. v_i \in D_{x_i}\) and where \(N\) is the number of state variables in \(X\). For a subset of the state variables \(X' \subseteq X\), let \(D[X'] = \{x \in D_x | x \in X'\}\) denote its joint domain. The state space is denoted as \(S = D[X]\). 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 that is defined by the values assigned by \(s\) to the subset of state variables in \(X'\). The projection of \(s\) onto \(X'\) defines the subset \(\{s | s \in S, s_{X'} \subseteq s\}\) of the states that are consistent with the corresponding partial state. Last, let us define a **state-constraint** as a Boolean function \(C : S \rightarrow \{0, 1\}\) over the state variables, that implicitly defines the subset of states \(S_C = \{s | C(s) = 1\}\) that are consistent with that constraint.

Let \(A\) be a set of **deterministic actions** such that each action \(a \in A\) is characterized by two functions; an **applicability function** \(\rho_a : S \rightarrow \{0, 1\}\) and a **successor function** \(\theta_a : S \rightarrow S\). An action \(a \in A\) is applicable in a given state \(s \in S\) iff \(\rho_a(s) = 1\). The execution of an applicable action \(a \in A\), in a state \(s \in S\) results in the successor state \(s' = \theta_a(s)\). Please note that this definition of deterministic actions generalizes actions with **conditional effects** \([85]\), common in GP since their state-dependent outcomes allow the adaptation of generalized plans to different classical planning instances.
A classical planning instance is a tuple \( P = (X, A, I, G) \), 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 constraint on the value of the state variables that induces the subset of goal states \( S_G = \{ s \mid s \models G, s \in S \} \). Given a classical planning instance \( P \), a plan is an action sequence \( \pi = (a_1, \ldots, a_m) \) whose execution induces a trajectory \( \tau = (s_0, a_1, s_1, \ldots, a_m, s_m) \) such that, for each \( 1 \leq i \leq m \), \( a_i \) is applicable in \( s_{i-1} \) and results in the successor \( s_i = \theta_{a_i}(s_{i-1}) \). A plan \( \pi \) solves \( P \) if and only if the execution of \( \pi \) in \( s_0 \), where \( s_0 = I \), finishes in a goal state, i.e. \( s_m \in S_G \).

Planning languages, such as PDDL [86], can compactly represent the infinite set of classical planning instances of a given domain using a finite set of functions and action schemes. Given a finite set of objects \( \Omega \), and a finite set of functions \( \Phi \) defined over that set of objects, we assume that each state variable \( x \in X \) stands for a function interpretation \( x \equiv \phi(\overline{x}) \), where \( \phi \in \Phi \) is a function with arity \( \text{arity}(\phi) \), and \( \overline{x} \in \Omega^{\text{arity}(\phi)} \) is a vector of objects comprised in the Cartesian product space of \( \Omega^{\text{arity}(\phi)} \); objects and function signatures can be typed so that the set of possible function interpretations is constrained. Functions in \( \Phi \) can be Boolean e.g. to represent PDDL predicates, or numeric e.g. to represent PDDL numeric fluents. Likewise, given a set of action schemes \( \Xi \), we assume that each action \( a \in A \) is built from an action schema \( \xi \in \Xi \) by substituting each variable in the action scheme with an object from \( \Omega \). An action scheme \( \xi \in \Xi \) is a tuple \( \xi = (\text{name}(\xi), \text{par}(\xi), \text{pre}(\xi), \text{eff}(\xi)) \) where:

- \( \text{name}(\xi) \) is the identifier of the action schema,
- \( \text{par}(\xi) \) is the list of free variables, again these variables can be typed so they can only be substituted by objects of the same type,
- \( \text{pre}(\xi) \) is a conjunction of Boolean formulae, where each formula is a logical evaluation, i.e. \( =, \neq, >, \leq, \geq \), between two function symbols \( \phi_1, \phi_2 \in \Phi \) defined over \( \text{par}(\xi) \), or a function symbol \( \phi \) also defined over \( \text{par}(\xi) \) and a value \( v \), that compactly represents the subset of states where the corresponding ground actions are applicable, and
- \( \text{eff}(\xi) \) is a set of logical assignments, where each function symbol \( \phi \) gets the value either from another function symbol \( \phi' \) (both defined over \( \text{par}(\xi) \)) or from a constant value \( v \), that compactly represents the updates of the state variables caused by the application of the corresponding ground actions.

### 3.2. Generalized planning

Generalized planning is an umbrella term that refers to more general notions of planning [21]. This work builds on top of the inductive formalism for GP, where a GP problem is a finite set of classical planning instances that belong to the same domain [19,62].

#### Definition 1 (GP problem)

A GP problem is a non-empty set \( P = \{P_1, \ldots, P_T\} \) of \( T \) classical planning instances from a given domain \( D \).

Each instance \( P_t \in P, 1 \leq t \leq T \), may actually differ in the set of state variables \( X_t \), actions \( A_t \), initial state \( I_t \), and goals \( G_t \), but the corresponding set of state variables \( X_t \) is induced from the common set of functions \( \Phi \). Likewise, the set of actions \( A_t \) is induced from the common set of action schemes \( \Xi \), when grounded over the particular set of states \( \Omega_t \) of the instance.

The aim of GP is to compute algorithmic planning solutions, a.k.a. generalized plans, which work for the full input set of planning problems. There are diverse representations for GP solutions, ranging from generalized polices [34,58], to finite state controllers [39,59], formal grammars [60], hierarchies [87,61], or programs [15,33]. Each representation has its own expressiveness capacity, as well as its own validation complexity and computational diversity. In spite of this representation diversity, we can define a common condition under which a generalized plan is considered a solution to a GP problem.

#### Definition 2 (GP solution)

A generalized plan \( \Pi \) solves a GP problem \( P = \{P_1, \ldots, P_T\} \) iff, for every classical planning instance \( P_t \in P, 1 \leq t \leq T \), the execution of \( \Pi \) on \( P_t \), denoted as \( \text{exec}(\Pi, P_t) = (a_1, \ldots, a_m) \), induces a classical plan that solves \( P_t \).

**Example.** Fig. 2 shows the initial state and goal of two classical planning instances, \( P_1 = (X, A, I_1, G_1) \) and \( P_2 = (X, A, I_2, G_2) \), for sorting two six-element lists. In this particular example the two instances share the same set of state variables \( X = \{x_i \equiv \text{vector}(a_i); 0 \leq i \leq 5\} \) that is built with the one-arity function \( \Phi = \{\text{vector}\} \) and the set of objects \( \Omega_1 = \Omega_2 = \{0_2, \ldots, 0_5\} \), and where \( \forall_{x \in X} D_x = N_0 \). The two classical planning instances also share the set of deterministic actions \( A \), with \( \frac{6 \times 5}{2} = 15 \) actions \( \text{swap}(a_1, a_2) \), that swap the content of two list positions \( i < j \), and that are induced from the single action scheme \( \Xi = \{\text{swap}(x, y)\} \). An example solution plan for \( P_1 \) is \( \pi_1 = (\text{swap}(0_5, 0_4), \text{swap}(0_1, 0_2), \text{swap}(0_1, 0_3)) \) while \( \pi_2 = (\text{swap}(0_5, 0_4), \text{swap}(0_1, 0_3)) \) is an example of a sequential plan that solves \( P_2 \). Note that \( P = \{P_1, P_2\} \) is a GP problem since it comprises two classical planning instances that are built using the same set of functions \( \Phi \) and action schemes \( \Xi \). Fig. 3 shows an example of a generalized plan that solves \( P \), and that is represented as a sorting network [88]. The sorting network is illustrated using two different types of items (namely the wires and the comparators). For each state variable, there is a wire that carries the value of that variable from left to right in the network. On the other hand, comparators connect two different wires, corresponding to a pair of variables \( (x_i, x_j) \), such that \( i < j \). When a pair of values traveling through a pair of wires \((i, j)\), encounters a comparator, then the comparator applies the action \( \text{swap}(a_1, a_2) \) iff \( \text{vector}(a_1) \geq \text{vector}(a_2) \), which in turn is \( x_i \geq x_j \). The sorting network of Fig. 3 can actually solve any instance for sorting the content of any six-element list, no matter its initial content. This solution is however not valid for sorting lists with different lengths. In this paper we will show how to represent, and compute, planning solutions that leverage indirect memory addressing to generalize no matter the number of objects, and corresponding state variables.
### 3.3. Planning programs

In this work we represent planning solutions as planning programs \[33\]. Unlike sequential plans, planning programs include a control flow construct which allows the compact representation of solutions to classical and GP problems. Formally a planning program \( \Pi \) is a sequence of \( n \) instructions, where each instruction \( \Pi[i] \) is associated with a program line \( 0 \leq i < n \), and it is either:

- A planning action \( \Pi[i] \in A \).
- A goto instruction \( \Pi[i] = go(i', \gamma) \), where \( i' \) is a program line \( 0 \leq i' < i \) or \( i + 1 < i' < n \), and \( \gamma \) is a proposition.
- A termination instruction \( \Pi[i] = \text{end} \). The last instruction of a planning program is always a termination instruction, i.e. \( \Pi[n - 1] = \text{end} \).

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 counter \( 0 \leq i < n \). Given a program state \((s, i)\), the execution of a programmed instruction \( \Pi[i] \) is defined as:

- If \( \Pi[i] \in A \), the new program state is \((s', i + 1)\), where \( s' = \theta_{\Pi[i]}(s) \) is the successor when applying \( \Pi[i] \) in \( s \).
- If \( \Pi[i] = go(i', \gamma) \), the new program state is \((s, i + 1)\) if \( \gamma \) holds in \( s \), and \((s, i')\) otherwise.\(^2\) Proposition \( \gamma \) can be the result of an arbitrary expression on state variables, e.g. a state feature \[89\].
- If \( \Pi[i] = \text{end} \), program execution terminates.

To execute a planning program \( \Pi \) on a classical planning instance \( P = (X, A, I, G) \), the initial program state is set to \((I, 0)\), i.e. the initial state of \( P \) and the first program line of \( \Pi \). A program \( \Pi \) solves \( P \) iff the execution terminates in a program state \((s, i)\) that satisfies the goal condition, i.e. \( \Pi[i] = \text{end} \land s \in S_G \). Otherwise, the execution of the program fails. If a planning program fails to solve the planning instance, the only possible sources of failure are:

1. **Inapplicable program**, i.e. executing action \( \Pi[i] \in A \) fails in program state \((s, i)\) since \( \Pi[i] \) is not applicable in \( s \).
2. **Incorrect program**, i.e. execution terminates in a program state \((s, i)\) that does not satisfy the goal condition, i.e. \((\Pi[i] = \text{end}) \land (s \notin S_G) \).
3. **Infinite program**, i.e. execution enters into an infinite loop that never reaches an end instruction.

In this work we model instructions \( \Pi[i] \in A \) as if they were always applicable but that their effects only update the current state iff the preconditions of the action hold in the current planning state. Formally, when executing \( \Pi[i] \) in \((s, i)\), the new program state is \((s', i + 1)\) iff \( \Pi[i] \) is applicable, otherwise it is \((s, i + 1)\). Therefore, in this work the execution of a program on a classical planning instance will never return an inapplicable program, and only incorrect or infinite program are possible sources of failure. This particular action modeling is common in reinforcement learning \[80\], and in conformant planning \[44\], because it delivers compact solutions that apply to sets of different problems (typically with different initial states).

\(^2\) We adopt the convention of jumping to line \( i' \) whenever \( \gamma \) is false, following the semantics of JMPZ instructions in the Random-Access Machine that jump when a register equals zero.
3.4. The Random-access machine

The Random-Access Machine (RAM) is an abstract computation machine, in the class of the register machines, that is polynomially equivalent to a Turing machine [90]. The RAM enhances a multiple-register counter machine [91] with indirect memory addressing; indirect memory addressing is useful for coding RAM programs that access an unbounded number of registers, no matter how many there are. A register in a RAM machine is then a memory location with both an address i.e. a unique identifier that works as a natural number (that we denote as \( r \)), and a content i.e. a single natural number (that we denote as \([r]\)).

A RAM program \( P \) is a finite sequence of \( n \) instructions, where each program instruction \( P[i] \), is associated with a program line \( 0 \leq i < n \). The execution of a RAM program starts at its first program instruction \( P[0] \). The execution of program instruction \( P[i] \) updates the RAM registers and the current program line. Diverse base instructions sets, that are Turing complete, can be defined. We focus on the three base sets of RAM instructions:

- **Base1.** \( \{\text{inc}(r), \text{dec}(r), \text{jmpz}(r,i), \text{halt}() \mid r \in R \} \). Respectively, these instructions increment/decrement a register by one, jump to program line \( 0 \leq i < n \) if the content of a register \( r \) is zero (i.e. if \([r] == 0\)), or end the program execution.
- **Base2.** \( \{\text{inc}(r_1), \text{clear}(r_1), \text{jmpz}(r_1, r_2, i), \text{halt}() \mid r_1, r_2 \in R \} \). In this set the value of a register cannot be decremented but instead, it can be set to zero with a clear instruction. In addition, jump instructions go to program line \( 0 \leq i < n \) if the content of two given registers is the same (i.e. if \([r_1] == [r_2]\)).
- **Base3.** \( \{\text{inc}(r_1), \text{set}(r_1, r_2), \text{jmpz}(r_1, r_2, i), \text{halt}() \mid r_1, r_2 \in R \} \). This set comprises no instruction to decrement, or clear, a register but instead, it includes an instruction to set a register to the value of another register.

The three base sets are equivalent [90]; one can build the instructions of one base set with instructions of another base set. Further, the expansive instruction set (that contains the instructions of Base 1,2 and 3) does not alter the expressiveness of the individual Base sets, since each of them already is Turing complete. The choice of the set of RAM instructions depends on the convenience of the programmer for the problem being addressed.

4. Planning with a random-access machine

The synthesis of effective features for a planning domain is a challenging research question investigated since the early days of AP [92]. Furthermore, the set of ground actions for the different problems of a given domain, is usually different since it depends on the number of objects in the problem; e.g. back to the sorting example illustrated in Figs. 2 and 3, classical planning problems for sorting a vector of length six induced \( \frac{6 \times 5}{2} \) swap \((o_j, o_j)\), \( i < j \) actions, while instances for sorting a vector of length seven would induce a set of \( \frac{7 \times 6}{2} \) swap \((o_j, o_j)\) actions. This section extends the classical planning model with a set of pointers, defined over the objects of a classical planning instance, and with the primitive instructions for manipulating those pointers; the extension allows the agnostic definition of a set of states features, and a set of actions, that are shared by the different instances of a classical planning domain (no matter their actual number of objects).

First, the section shows how to compactly represent a transition system using pointers. Then the section shows that the pointer-based representation naturally applies to the STRIPS formalism. Last, the section formalizes our extension of the classical planning model with a RAM machine, that produces the aimed set of state features and actions that are shared by all the instances of a classical planning domain. Those sets of shared state features and actions are later leveraged (at Section 5) for the computation of GP solutions, that generalize no matter the number of world objects.

4.1. Representing transition systems with pointers over objects

A transition system can be graphically represented as a directed graph and hence formalized as a pair \((S, \rightarrow)\), where \( S \) is a set of states, and \( \rightarrow \) denotes a relation of state transitions \( S \times S \). Transition systems differ from finite automata since the sets of states and transitions are not necessarily finite. Further a transition system does not necessary define a start/initial state or a subset of final/goal states. Transitions between states may be labeled, and the same label may appear on more than one transition. A prominent example are transition systems that correspond to a classical planning problem [1,2], where state transitions are labeled with actions (i.e. between two states \( s, s' \in S \), there exists a transition \((s \rightarrow s')\) iff the execution of action \( a \) in state \( s \) produces the state \( s' \)). Given a state \( s \) and an action label \( a \), if there exists only a single tuple \((s, a, s') \) in \( \rightarrow \) then the transition is said to be deterministic. In this work we restrict ourselves to deterministic transitions systems, i.e. transition systems such that all their transitions are deterministic.

**States.** WLOG we assume that the states of a transition system are factored; given a set of world objects \( \Omega \), a state is factored into a finite set of variables \( X \) s.t. each variable \( x \in X \) either represents a property of a world object, or a relation over \( k \) world objects. Formally \( x \equiv \phi(a_1, \ldots, a_k) \), where \( \phi \) is a \( k \)-ary function in \( \mathbb{N} \), and \( \{a_i\}_{i=1}^k \) are objects in \( \Omega \). For instance in the example illustrated by Figs. 2 and 3, given the one-arity function vector and the six-objects set \( \Omega = \{o_0, o_1, o_2, o_3, o_4, o_5\} \), each state variable \( x_i \in X \) is defined as \( x_i \equiv \text{vector}(a_i) \).

---

3. When the set of labels is a singleton, the transition system is essentially unlabeled, so the simpler definition that omits labels applies.
4.2. \textbf{Pointers and state constraints.} \textit{Pointers} are variables for indexing the objects of a transition system. In combination with function symbols, pointers are useful to define \textit{state constraints} that produce not only compact, but \textit{general} representations of a possibly infinite set of states. By \textit{general} we mean that a constraint represents a set of states that share some common structure, no matter the actual number of objects. Fig. 4 shows the Boolean function \texttt{constraint\_all\_sorted} that implements a \textit{global constraint} for checking whether the content of the vector of state variables is sorted in increasing order; the \texttt{constraint\_all\_sorted} function is procedurally defined, leveraging a single pointer \(z\), and it applies to any number of objects, and to any domain size of the corresponding state variables.

\textbf{Action schemes.} Action schemes compactly specify a (possibly infinite) set of transitions that share a common structure; action schemes generalize over any number, or identity of world objects. They do not refer to specific objects but instead, they leverage parameters to indirectly refer to the different world objects. Next we show that \textit{pointers over objects} enable the compact and general definition of (possibly infinite) sets of state transitions via \textit{action schemes}.

\textbf{Definition 3 (Pointer).} Given a set of objects \(\Omega\), with \(|\Omega|\) denoting the number of objects in the set, we define a pointer as a finite-domain variable \(z\) whose domain is \(D_z = \{0, \ldots, |\Omega| - 1\}\).

\begin{verbatim}
Bool constraint_all_sorted () {
    For (Pointer z:=1; z<|\Omega|; z++) {
        If ( vector(z-1) > vector(z) )
            Return False;
    }
    Return True;
}

Fig. 4. Boolean function \texttt{constraint\_all\_sorted} that checks whether the vector of state variables is sorted in increasing order. The constraint is implemented leveraging the single pointer \(z\) over the objects in \(\Omega\); \(\text{vector}(z)\) is interpreted as \(\text{vector}(x_i) \equiv x_i \in X\).
\end{verbatim}

\begin{verbatim}
Bool schema_swap (Pointer z1, Pointer z2) {
    If (z1>0 and z2>0 and z1<|\Omega| and z2<|\Omega| and z1<z2) {
        Variable aux;
        aux := vector(z1);
        vector(z1) := vector(z2);
        vector(z2) := aux;
        Return True;
    }
    Return False;
}

Fig. 5. Pointer-based representation of the \texttt{swap} action schema. When applicable, the \texttt{swap} action schema exchanges the value of the state variables indexed by its two parameters, the pointers \(z_1\) and \(z_2\).
\end{verbatim}

\begin{verbatim}
Definition 4 (Action schema with pointers). Given a set of \(X\) state variables, an \textit{action schema} with pointers is a tuple \((\text{name}, \text{params}, \text{pre}, \text{eff})\) where:
\begin{itemize}
    \item \text{name} is the label that uniquely identifies the action schema.
    \item \text{params} is a finite set of pointers \(Z\) defined over the set \(\Omega\) of objects.
    \item \text{pre} is a state constraint where state variables are indirectly addressed via the function symbols and the pointers in \text{params}, i.e. \(x \equiv \phi(\bar{z})\) such that \(\phi \in \Phi\) and \(\bar{z} \in Z^{\Phi}(\phi)\). The \text{pre} state constraint implicitly represents the subset of states where the action schema is applicable.
    \item \text{eff} is a partial assignment of the state variables where a subset of the state variables is indirectly addressed via the function symbols and the pointers in \text{params}. The \text{eff} partial assignment implicitly represents the successor state that results from the execution of the action schema at a given state.
\end{itemize}

Fig. 5 illustrates our pointer-based definition of an action schema; when applicable, the \texttt{swap} action schema exchanges the value of the state variables indexed by its two parameters (the pointers \(z_1\) and \(z_2\)). The state variables are \textit{global}, so they can be accessed from any action schema. The \texttt{swap} action schema is succinct, because it compactly defines an infinite set of different state transitions that share a common structure. The \texttt{swap} action schema is also general, because it applies to any sorting instance, no matter the length of the vector of state variables or the domain size of those variables. What is more, the execution of the \texttt{swap} action schema is a deterministic matching-free process since the input pointers do always index a single object in \(\Omega\).

4.2. \textbf{Representing STRIPS problems with pointers over objects}

Since the early 1970s, the \texttt{STRIPS} formalism is widely used for research in \textit{automated planning} [93]. Even today, \texttt{STRIPS} is an essential fragment of \texttt{PDDL} [86], the input language of the \textit{International Planning Competition}, and most planners support the \texttt{STRIPS} representation. Here we show that our pointer-based representation naturally applies to object-centered classical planning
formalisms, such as STRIPS. In fact, our pointer-based representation can be understood as an instantiation of F-STRIPS [94], where the single level of indirection of pointers over objects is enough to represent STRIPS problems with constant memory access.

STRIPS compactly represents the set of states of a transition system using a finite set of objects, and a finite set of first-order logic (FOL) predicates, that indicate properties of the objects and their relations. Likewise, STRIPS compactly represents the space of possible state transitions using FOL operators, which are defined as a tuple \( op = (name(op), arg\text{\_}set(op), pre(op), \text{eff}^{-}(op), \text{eff}^{+}(op)) \) where, \( name(op) \) is a unique identifier of the operator, \( arg\text{\_}set(op) \) is a set of variable symbols specifying the arguments of the operator, and \( pre(op), \text{eff}^{-}(op), \text{eff}^{+}(op) \) are sets of FOL predicates, with variables exclusively taken from \( arg\text{\_}set(op) \), and that respectively specify the preconditions, negative effects and positive effects. The representation of a STRIPS problem is completed by specifying an initial state, that defines the initial situation of the objects, and the target set of goal states, typically specified as a partial state.

**State representation.** When applying our pointer-based formalism to a STRIPS problem, each state variable \( x \in X \) has domain \( D_x = \{0,1\} \), and it is built as a FO STRIPS predicate \( \phi \in \Phi \) grounded by a vector of objects \( \vec{o} \in \Omega^{|\phi|} \). Fig. 6 shows the representation of a blockworld state using the STRIPS formalism, as well as using our formalism. In this state there are three blocks, \( \Omega = \{b0,b1,b2\} \), that are stacked in a single tower. Predicates \( \text{clear}(?x), \text{holding}(?x) \), and \( \text{ontable}(?x) \), are encoded as three different Boolean functions that map each vector of objects to either 0 or 1 in the current state. Omitted state variables are assumed to be zero valued. Our vector \( X \) of state variables is the result of unifying the predicates and object tuple valuations into a vector. The length of the vector of state variables is then upper bounded by \( |X| \leq \sum_{k \geq 2} n_k |\Omega|^k \), where \( n_k \) is the number of first-order predicates with arity \( k \).

For instance, the \( X \) vector contains at most \( |\Omega|^2 + 3|\Omega| + 1 \) state variables for the blockworld domain.\(^4\)

**Action representation.** Given a FO STRIPS operator \( op = (name(op), arg\text{\_}set(op), pre(op), \text{eff}^{-}(op), \text{eff}^{+}(op)) \), our pointer-based formalism produces its corresponding pointer-based action scheme \( (name, \text{params}, \text{pre}, \text{eff}) \):

- The name of the action schema is \( name(op) \), the name of the given FO STRIPS operator.
- For each argument in \( arg\text{\_}set(op) \), the action schema has a pointer that indexes an object \( o \in \Omega \).
- The set \( pre(op) \) is transformed into a conjunctive arithmetic-logic expression with conditions of two kinds: (i) conditions asserting that each pointer of the action schema is within its domain and (ii), for each precondition in \( pre(op) \) a condition asserting that the state variable addressed by the pointers content equals to some specific value of its domain.
- Each negative effect in \( \text{eff}^{-}(op) \) is transformed into an indirect variable assignment that sets the corresponding state variable to 0. Likewise, each positive effect in \( \text{eff}^{+}(op) \) is transformed into an indirect variable assignment that sets the corresponding state variable to 1.

Next we show the grammar that formalizes our pointer-based representation of STRIPS action schemes,

\[
\Pi ::= if(CONDITIONS)[ASSIGNMENTS]
\]

\( \text{Return False;} \)

\( CONDITIONS ::= (z > 0 \text{ and } z < |\Omega|) \text{ and } CONDITIONS | (p(z_1,\ldots,z_k) == 0) \text{ and } CONDITIONS | (p(z_1,\ldots,z_k) == 1) \text{ and } CONDITIONS | (True) \)

\( ASSIGNMENTS ::= p(z_1,\ldots,z_k) := 0; \text{ ASSIGNMENTS | p(z_1,\ldots,z_k) := 1; \text{ ASSIGNMENTS | Return True;} \)

where \( CONDITIONS \) includes assertions over predicates \( p(z_1,\ldots,z_k) \) instantiated with the action arguments (i.e. the pointers), and represents the operator preconditions (\( == \) denotes the equality operator, \( := \) indicates an assignment, and a semicolon denotes

---

\(^4\) State-invariants [95] can be leveraged to save space for the memory allocation of the state variables, e.g. in the blockworld one block cannot be on top of two different blocks simultaneously.
\begin{verbatim}
(defun unstack (z1 z2)
  (if (and (z1>=0) (z2>=0) (z1<| and z2<|)
    (clear z2) := 1;
    (handempty z1) := 1;
    (on z1 z2) := 0;
    Return True;
  Return False;
)

(defun init()
  (clear b0) := 1; on(b0, b1) := 1; on(b1, b2) := 1; ontable(b2) := 1;
)

(defun goals()
  (ontable(b0)=1 and ontable(b1)=1 and ontable(b2)=1);
)
\end{verbatim}

Fig. 8. The \texttt{unstack} action schema from \texttt{blockworld} defined with two pointers \((z1\text{ and } z2)\).

Fig. 9. The \texttt{init} and \texttt{goals} procedures for representing the STRIPS planning problem of unstacking the three-block tower of Fig. 6.

the end of a program instruction). ASSIGNMENTS is a conjunction of assignments representing the operator positive/negative effects; in more detail \(p(z_1,...,z_k) := 1\) denotes a \textit{positive} effect while \(p(z_1,...,z_k) := 0\) denotes a \textit{negative} effect. Fig. 8 shows our pointer-based definition for the \texttt{unstack} action schema from the blockworld that implements the corresponding operator represented in the STRIPS fragment of PDDL of Fig. 7. The action schema of Fig. 8 is implemented using two pointers \((z1\text{ and } z2)\), and it applies to any blockworld instance, no matter the number of blocks or their identity.

\textbf{Problem representation.} We complete our pointer-based representation of a STRIPS problem with the \texttt{init} and \texttt{goals} procedures: the \texttt{init procedure} is a write-only procedure, that implements a total variable assignment of the state variables for specifying the initial state of the STRIPS problem. The \texttt{goals procedure} is a read-only Boolean procedure, that encodes the stateconstraint that specifies the subset of goal states. Fig. 9 shows the \texttt{init} and \texttt{goals} procedures for the planning problem of unstacking the 3-block tower of Fig. 6. The content of the \texttt{init} and \texttt{goals} procedures is inductively formalized as follows:

\textit{INIT} := (p(o_1,...,o_k) := 1); \textit{INIT};

\textit{GOALS} := Return(CONDITIONS);

\textit{CONDITIONS} := (p(o_1,...,o_k) == 0) and \textit{CONDITIONS} |

\hspace{2em} (p(o_1,...,o_k) == 1) and \textit{CONDITIONS} |

\hspace{2em} (True)

\textbf{Solution representation.} Our pointer-based representation of a sequential plan \(\pi\) comprises instructions for: (i), invoking the Boolean function that encodes an action scheme and (ii), incrementing/decrementing the value of a pointer. Formally:

\begin{verbatim}
\pi := STATEMENTS;

STATEMENTS := a(z_1,...,z_k); STATEMENTS |

z++; STATEMENTS |

z--; STATEMENTS |

;
\end{verbatim}

where \(a(z_1,...,z_k)\) is an action scheme instantiated with a set of pointers \([z_1,...,z_k] \subseteq Z\), and \((z++, z- | z \in Z)\) are the instructions to \textit{increment/decrement} a pointer \(z \in Z\). Pointers are always initialized to zero.
void ONTACLE-SEQUENTIAL-PLAN () {
    int z1=0, z2=0;
    z2++;
    act_unstack(z1, z2);
    act_putdown(z1);
    z1++;
    z2++;
    act_unstack(z1, z2);
    act_putdown(z1);
}

Fig. 10. Pointer-based representation of the sequential plan π = (unstack(b₀,b₁), putdown(b₀), unstack(b₁,b₂), putdown(b₁)) for unstacking the three-block tower of Fig. 6.

Fig. 11. Relation between the action scheme unstack(?,?,?) (i), the action unstack(z₁,z₂) instantiated with pointers (z₁, z₂), and (ii), the ground actions instantiated with the set of three blocks Ω = {b₀, b₁, b₂}. Pointers z₁ and z₂ are bound variables in [0, ..., |Ω|], that currently are indexing blocks b₀ and b₁, respectively.

Fig. 10 illustrates our pointer-based representation of the four-action sequential plan π = (unstack(b₀,b₁), putdown(b₀), unstack(b₁,b₂), putdown(b₁)) for unstacking the three-block tower of Fig. 6. In more detail, the ONTACLE-SEQUENTIAL-PLAN() program leverages two pointers, Z = {z₁, z₂}, that are initialized to zero so they initially point to the first object (block b₀ in this case). After executing the first z₂++ instruction, z₂ points to the second block, b₁, while z₁ still points to block b₀. This means that the first act_unstack(z₁, z₂) instruction of the program in Fig. 10 is actually executing the ground action unstack(b₀, b₁), which corresponds to the first step of plan π. Likewise, the first act_putdown(z₁) program instruction executes the ground action putdown(b₀), i.e. the second step of the sequential plan π. The second act_unstack(z₁, z₂) program instruction is executing the ground action unstack(b₁, b₂), since both z₁ and z₂ are increased just before that instruction is executed. Finally, the second act_putdown(z₁) executes the ground action putdown(b₁), which is the fourth and last step of the sequential plan π.

Fig. 11 illustrates the particular relation between an action scheme (i), its corresponding action instantiated over pointers and (ii), the ground actions instantiated over the set of world objects Ω. In Fig. 11, the pointers z₁ and z₂ are bound variables in [0, ..., |Ω|], that currently are indexing blocks b₀ and b₁, respectively.

Beyond STRIPS. Our pointer-based representation supports object typing by simply specializing pointers to the subset of objects of a particular type [86]. Furthermore, our pointer-based representation naturally supports classical planning with numeric state variables, as defined in PDDL2.1 [96]. To support the representation of numeric state variables, the vector of state variables store integers instead of Boolean values. Goals and action preconditions can then include assertions over numeric state variables, and action effects can include assignments of the numeric state variables. For example, distance(b₀,b₁)=7 can be used to indicate that the physical distance between blocks b₀ and b₁ is of seven units. Likewise distance(z₁,z₂) > distance(z₂,z₃) can be used to indicate that the distance between the blocks indexed by pointers z₁ and z₂ is larger than the distance between the blocks pointed by z₂ and z₃.

4.3. Extending the classical planning model with a RAM

Now we are ready to leverage our pointer-based representation, with the notion of Random-Access Machine (RAM), to extend the classical planning model. The extension produces an agnostic set of state features, and a set of actions, that are shared for the different classical planning instances of a given domain (no matter their actual number of objects).

Given a classical planning instance \( P = (X, A, I, G) \), s.t. the state variables and actions are generated with the set of functions \( \Phi \) and action schemes \( \Xi \) of a given domain, grounded with a set of objects \( \Omega \). Then, the classical planning instance extended with a RAM machine of \(|Z| + 2\) registers (i.e. \(|Z|\) pointers that reference the planning objects, plus two dedicated FLAGS registers namely the zero and carry flags), is defined as \( P'_Z = (X'_Z, A'_Z, I'_Z, G) \) where:

- The new set of state variables \( X'_Z \) comprises:
  - The state variables \( X \) of the original planning instance, such that each state variable \( x_i \in X \) is \( x_i = \phi(\overline{a}) \) with \( \phi \in \Phi \) and \( \overline{a} \in \Omega^{\Phi}(\phi) \), as defined above.
  - Two Boolean variables \( Y = \{y_z, y_c\} \), that play the role of the zero and carry FLAGS registers, respectively.
  - The pointers \( Z \), a set of extra state variables s.t. each \( z \in Z \) has finite domain \( D_z = [0, \ldots, |\Omega| - 1] \).
A set of derived state variables $X_Z = \{ \phi(\bar{z}) \mid \phi \in \Phi, \bar{z} \in Z^{ar(\phi)} \}$ whose value is given by the interpretations of the functions of the domain with the corresponding pointers.  

- The new set of actions $A'_Z$ will represent the set of actions that is shared for the different classical planning instances in a given domain, and it includes:
  
  - The planning actions $A'$ that result from reformulating each action scheme $\xi \in \Xi$ into its corresponding pointer-based version. The reformulation is a two-step procedure that requires $Z$ to contain, at least, as many pointers as the largest arity of an scheme in $\Xi$: (i), each parameter in $par(\xi)$ is replaced with a pointer in $Z$ and (ii), preconditions and effects are rewritten to refer to these pointers.
  
  - The RAM actions that implement the following sets of RAM instructions $\{inc(z_1), dec(z_1), cmp(z_1, z_2), \text{set}(z_1, z_2) \mid z_1, z_2 \in Z \}$ over the pointers in $Z$, and $\{ \text{test}(\phi(\bar{z_1})), cmp(\phi(\bar{z_1}), \phi(\bar{z_2})) \mid \bar{z}_1, \bar{z}_2 \in Z^{ar(\phi)} \}$ over the lists of pointers in $Z^{ar(\phi)}$ for each function symbol $\phi \in \Phi$. Respectively, these RAM instructions increment/decrement a pointer by one while keeping the values within the pointer domain, compare two pointers, set the value of a pointer $z_2$ to another pointer $z_1$, and compare the value of $\phi(\bar{z}_1)$ greater than zero, and compare the value of $\phi(\bar{z}_1)$ and $\phi(\bar{z}_2)$. The $\text{cmp}(\phi(\bar{z}_1), \phi(\bar{z}_2))$ instructions are only required for numeric functions\textsuperscript{5} to compare whether $\phi(\bar{z}_1)$ is greater, equal or smaller than $\phi(\bar{z}_2)$. The outcome of executing a RAM action is captured in a value, here denoted as $ret$:

$$
\begin{align*}
\text{inc}(z_1) \Rightarrow ret &:= \begin{cases} 
z_1 + 1 & \text{if } z_1 + 1 < |\Omega| \\ 0 & \text{Otherwise} \end{cases} \\
\text{dec}(z_1) \Rightarrow ret &:= z_1 - 1, \\
\text{cmp}(z_1, z_2) \Rightarrow ret &:= z_1 - z_2, \\
\text{set}(z_1, z_2) \Rightarrow ret &:= z_2, \\
\text{test}(\phi(\bar{z}_1)) \Rightarrow ret &:= \phi(\bar{z}_1), \\
\text{cmp}(\phi(\bar{z}_1), \phi(\bar{z}_2)) \Rightarrow ret &:= \phi(\bar{z}_1) - \phi(\bar{z}_2).
\end{align*}
$$

Once a RAM action is executed, its returned value is used to update the Boolean FLAGS $Y = \{y_z, y_c\}$ as defined, i.e. $y_z := (\text{ret} = 0)$ and $y_c := (\text{ret} > 0)$. FLAGS are dedicated to capture the outcome of RAM instructions. The combination of both FLAGS registers can then represent any outcome of a three-way comparison \[97\].

- The new initial state $I'_Z$ is the initial state of the original planning instance, but extended with all pointers set to zero and the two FLAGS set to $\text{False}$. The goals are the same as those of the original planning instance.

The number of pointers $|Z|$ is a parameter that indicates how many pointers are used in the extension. At least $Z$ must contain as many pointers as the largest arity of the functions $\Phi$ and action schemes $\Xi$ of the given domain. Therefore $\{Y \cup Z \cup X_Z\}$, becomes a subset of state variables shared by all the instances that belong to the same domain, no matter their number of objects. Likewise $A'_Z$ becomes a set of actions that is shared by all the instances that belong to the same domain, no matter their actual number of objects.

Example. Here we extend the classical planning instance $P_1 = \langle X, A, I_1, G_1 \rangle$ (illustrated in Fig. 2) using a RAM with $Z = \{i, j\}$ two pointers. According to this extension, our pointer-based representation of the sequential plan $\pi_1 = \langle \text{swap}(a_0, a_5), \text{swap}(a_0, a_2), \text{swap}(a_1, a_0) \rangle$ is the following sequence of thirteen actions $\pi'_1 = \langle \text{inc}(j)^5, \text{swap}(i, j), \text{inc}(i), \text{dec}(j)^3, \text{swap}(i, j), \text{inc}(j), \text{swap}(i, j) \rangle$; where superscripts refer to the number of times that an instruction is sequentially repeated, and where $\text{swap}(i, j)$ refers to the pointer-based action schema defined in Fig. 5. Likewise, our pointer-based version of the sequential plan $\pi_2 = \langle \text{swap}(a_0, a_2), \text{swap}(a_0, a_3) \rangle$, that solves the classical planning problem $P_2$ in Fig. 2, is the ten-action sequence $\pi'_2 = \langle \text{inc}(j)^7, \text{swap}(i, j), \text{inc}(i)^5, \text{inc}(j)^3, \text{swap}(i, j) \rangle$.

4.3.1. Theoretical properties

Our extension of a classical planning problem with a RAM machine preserves the solution space of the original problem. Sequential plans in the extended planning model may however be longer (e.g. the pointer-based version of plan $\pi_1$ from the previous example required thirteen steps while the original sequential plan only had three steps). As a rule of thumb, an increment of the original plan length happens when pointers must be incremented (or decremented) multiple times to access the corresponding objects before the corresponding action is executed.

Theorem 1. Given a classical planning instance $P = \langle X, A, I, G \rangle$, its extension $P'_Z$, using a RAM machine with $Z$ pointers s.t. $|Z| \geq \max_{a \in A} ar(a)$, preserves the solution space of $P$.

Proof. $\Rightarrow$: Let $\pi = \langle a_1, \ldots, a_n \rangle$ be a plan that solves $P$, an equivalent plan $\pi'$ that solves $P'_Z$ is built as follows; for each action $a_i \in \pi$, $A'$ contains a pointer-based action schema $d'_i$ that replaces each parameter in $par(a_i)$ with a pointer $z \in Z$. For each such pointer $z$, the plan repeatedly applies RAM actions $\text{inc}(z)$ or $\text{dec}(z)$ until they reference the associated vector of objects $\bar{o}$, and then it

\[\text{Compare instructions are syntactic sugar for Boolean functions since they can be implemented composing test instructions.}\]
applies $a'_i$. The resulting plan $\pi'$ has exactly the same effect as $\pi$ on the original planning state variables in $X$, and since the goal condition of $P'_Z$ is the same as that of $P$, it follows that $\pi'$ solves $P'_Z$.

$:=$ Let $\pi' = \langle a'_1, \ldots, a'_{n'} \rangle$ be a plan that solves $P'_Z$. Identify each action in $A'$ among those of $\pi'$, and execute $\pi'$ to identify the assignment of objects to pointers when applying each action in $A'$. Construct a plan $\pi$ corresponding to the subsequence of actions in $A'$ from $\pi'$, replacing each action schema $a'_i \in A'$ by an original action $a_i \in A$ and choosing as parameters of $a_i$ the objects referenced by the pointers of $a'_i$ at the moment of execution. Hence $a_i$ has the same effect as $a'_i$ on the state variables in $X$, implying that $\pi$ has the same effect as $\pi'$ on $X$. Since the goal condition of $P$ is the same as that of $P'_Z$, it follows that $\pi$ solves $P$. \hfill $\square$

The execution of a plan corresponding to a classical planning problem extended with a RAM machine is a deterministic matching-free process that does not require explicit action grounding; the plan execution itself determines the values of the pointers that feed the action schemes.

**Theorem 2.** The new set of actions $A'_Z$ is independent of the number of objects, state variables, and their domain size.

**Proof.** The number of actions of a classical planning instance, extended with a RAM of $|Z|$ pointers, is

$$|A'_Z| = 2|Z|^2 + \sum_{\phi \in \Phi} |Z|^{2\ar(\phi)} + |A'|.$$  \hfill (1)

This number exclusively depends on the number of pointers in $Z$, on the arity of the functions in $\Phi$, and on the arity of the action schemes in $\Xi$. First, the increment/decrement instructions induce $2|Z|$ actions, the set instructions over pointers induce $|Z|^2 - |Z|$ actions, and comparison instructions of pointers induce $|Z|^2 - |Z|$ actions; comparison instructions can compare two pointers but for symmetry breaking, we only consider the single parameter ordering $(z_i, z_j)$ where $i < j$, i.e. we consider cmp $(z_1, z_2)$ but not cmp $(z_2, z_1)$. Second, test instructions are defined over each function symbol and list of pointers with the same size as its arity, inducing $\sum_{\phi} |Z|^{\ar(\phi)}$ actions, and comparison of predicates with pointers induce $\sum_{\phi} (|Z|^{2\ar(\phi)} - |Z|^{\ar(\phi)})$ actions. Therefore, the total number of RAM instructions is $2|Z|^2 + 2(|Z|^2 - |Z|) + \sum_{\phi} (|Z|^{\ar(\phi)} + |Z|^{2\ar(\phi)} - |Z|^{\ar(\phi)}) = 2|Z|^2 + \sum_{\phi} |Z|^{2\ar(\phi)}$ which only depends on the number of pointers in $Z$ and the arity of each function symbol $\phi$. Last, as defined by our abstraction procedure, the number of actions in $A'$ is given by the number of parameters of the actions schemes $\Xi$ and the number of pointers in $Z$ to replace those parameters. This means that the size of $A'$ is upper bounded by $|A'| \leq \sum_{\phi \in \Xi} |Z|^{\ar(\phi)}$. Last, it follows that $A'_Z$, whose size is given by $|A'_Z| = 2|Z|^2 + \sum_{\phi} |Z|^{2\ar(\phi)} + |A'|$, it is also independent of the number of objects $\Omega$, state variables in $X$ and their domain size. \hfill $\square$

5. Generalized planning as heuristic search

This section presents our GP as heuristic search approach; first the section details the search space of our GP as heuristic search approach and then, the section explains the particular details of our heuristic search algorithm for GP, called BFGP.

5.1. The search space

Here we characterize the branching factor of the space of planning programs introduced in Section 3.3, and we show that its size depends on the domain of the planning state variables (which unfortunately may be unbound). Then we define a tractable GP search space by conditioning the branching and looping of planning programs with the FLAGS registers of the RAM model. We show that our new search space for GP is independent of the number of input planning instances in a GP problem, and the size of these instances (i.e. the number of objects, state variables, and their domain sizes). This enables the definition of a heuristic search approach for GP, capable of managing large sets of state variables with large numerical domains, such as integers.

5.1.1. Planning programs conditioned over valued state variables

The branching and looping of the planning programs introduced in Section 3.3 are conditioned over valued state variables [33], i.e. go to some line $i$ if state variable $x$ has value $v$. The space of this kind of solutions can be characterized by a binary encoding; given a set of state variables $X$, a set of actions $A$, a maximum number of program lines $n$ such that the last instruction is $\Pi_{n-1} = \text{end}$, and defining the propositions of goto instructions as $(x = v)$ atoms where $x \in X$ and $v \in D_x$, the space of possible planning programs is encoded with three bit-vectors:

1. The action vector of length $(n - 1) \times |A|$, indicating whether an action $a \in A$ is programmed on line $0 \leq i < n - 1$.
2. The transition vector of length $(n - 1) \times (n - 2)$, indicating whether a go($i'$, *) instruction is programmed on line $0 \leq i < n - 1$.
3. The proposition vector of length $(n - 1) \times \sum_{x \in X} |D_x|$, indicating whether a go(*, $(x = v)$) instruction is programmed on line $0 \leq i < n - 1$.

**Definition 5** Partially specified planning program $\Pi$ is a planning program s.t. the content of some of its program lines may be undefined, i.e. $\exists i \in [0, n)$ s.t. $\Pi[i] = \text{< empty >}$. 

14
A partially specified planning program is encoded as the concatenation of the previous three bit-vectors and the length of the resulting bit-vector is:

\[(n-1)\left(|A| + (n - 2) + \sum_{x \in X} |D_x|\right).
\] (2)

The previous binary encoding allows us to quantify the similarity of two partially specified planning programs (e.g. the Hamming distance of their corresponding bit-vector representation) and more importantly, to systematically enumerate the space of all possible planning programs with a maximum of \(n\) lines. Let us define the empty program as the partial specification of a planning program whose instructions are all undefined (i.e. all bits of its bit-vector representation are set to False). Starting from the empty program, we can enumerate the entire set of possible planning programs with two search operators:

- program\((i,a)\), that programs an action \(a \in A\) at line \(i\) of a program
- program\((i,i',x,v)\), that programs a goto\((i',\langle x = v \rangle)\) instructions at line \(i\) of a program.

These two search operators are only applicable when \(\Pi[i] = \langle\ \text{empty}\rangle\) (meaning that \(i\) is an undefined program line i.e. in the bit-vector representation the bits corresponding to the encoding of the program line \(i\) are set to False). Given the bit-vector representation of a partially specified planning program, the application of the program\((i,a)\) or program\((i,i',x,v)\) search operators set the corresponding bits to True. With this regard, the partially specified planning program of a given search node is at Hamming distance 1 from its parent, when programming a planning action with program\((i,a)\), or at Hamming distance 2, when programming a goto instruction with program\((i,i',x,v)\). In fact, this is the search space leveraged by the classical planning compilation approach for computing planning programs with an off-the-shelf classical planner [33]. Equation (2) reveals that the number of planning programs with \(n\) lines depends on the number of grounded actions \(|A|\), and the number of state variables \(x \in X\) and their domain \(D_x\). This dependence causes an scalability issue, limiting the applicability of the cited compilation to planning instances of contained size. In the worst case, the domain of state variable might be infinite, e.g. a numeric state variable, hence the search space might be intractable.

5.1.2. Planning programs conditioned over a feature language

We overcome the intractability of the previous solution space by conditioning the branching and looping of planning programs with a finite feature language. In more detail, we leverage our extension of the classical planning model with a RAM, since it produces a minimalist but general set of features for the classical planning instances of a given domain.

Definition 6 (The feature language). We define the feature language as the four possible joint values of the two Boolean variables \(Y = \{y_x, y_c\}\), and we denote this language as \(\mathcal{L} = \{\neg y_x \wedge \neg y_c, y_x \wedge \neg y_c, \neg y_x \wedge y_c, y_x \wedge y_c\}\).

Our feature language \(\mathcal{L}\) is minimalist, it only contains four elements. We say that \(\mathcal{L}\) is general because it is independent of the number of objects (and hence, of the number of state variables and their domain). Features in \(\mathcal{L}\) are a function of (i) the state variables and (ii) the last executed instruction. The value of FLAGS \(Y = \{y_x, y_c\}\) depend on the last executed instruction, and considering that only RAM instructions update the variables in \(Y\), we have a space of \(2^{|Y|} \times 2^{|Z|} + \sum_{i} |Z|^{2^{|\text{vars}(\phi)|}}\) state observations implemented with only \(|Y|\) Boolean variables. The four joint values of \(\{y_x, y_c\}\) can model then a large space of observations, e.g. \(0 = 0, \neq 0, <0, >0, \leq 0, \geq 0\) as well as relations \(=, \neq, <, >, \leq, \geq\) on pairs of state variables.

Definition 7 (The GP with a RAM search space). Given a GP problem \(P\), that is built extending a set \(\{P_1, \ldots, P_F\}\) of classical planning instances from a given domain with a RAM of \(|Z|\) pointers. Our GP search space is the set of partially specified planning programs that can be built with \(n\) program lines, the common set of planning actions \(A'_Z\), and goto instructions that are exclusively conditioned on a feature in \(\mathcal{L}\).

According to Definition 7, we represent GP solutions as planning programs where goto instructions can exclusively be conditioned on a feature in \(\mathcal{L}\). Limiting the conditions of goto instructions to any of the four features in \(\mathcal{L}\) bounds the number of planning programs. Although the \(Y = \{y_x, y_c\}\) flags have four possible value combinations, the fourth case \((y_x \wedge y_c) \in \mathcal{L}\) can never happen as a result of a comparison; this fourth case is however useful for representing unconditional goto. The proposition vector required to encode a planning program becomes now a vector of only \((n - 1) \times 4\) bits (one bit for each of the four features in \(\mathcal{L}\)). Equation (2) simplifies then to:

\[(n-1)\left(|A'_Z| + (n - 2) + 4\right).
\] (3)

Equation (3) shows that the size of our new solution space for GP is independent of the number of objects and hence of the number of original state variables and their domain size; Theorem 2 already showed that \(A'_Z\) no longer grows with the number of objects. This novel GP solution space can now scale to planning problems where state variables have large domains (e.g. integers) and that have a large number of state variables.
Theorem 3. The space of planning programs that exclusively branch and loop over the features in \( L \) preserves the solution space of planning programs that branch and loop over valued state variables.

Proof. Given a GP problem \( P \) and a planning program \( \Pi \) conditioned over valued state variables that solves \( P \). An equivalent planning program, that exclusively branches over any of the features in \( L \), is built replacing each goto\((i, ! (x=v))\) instruction in \( \Pi \), where \( x \equiv \phi(\bar{s}) \) s.t. \( \phi \in \Phi \) and \( \bar{s} \in \Omega^{arity(\phi)} \), by a finite block of instructions that: (i) increments/decrements a vector of auxiliary pointers \( \bar{z}_{\text{aux}} \), with size \( arity(\phi) \), until they index objects \( \bar{s} \), (ii) given auxiliary static state variables for each possible value, i.e. \( Y_{\bar{s} \in D_{\bar{s}}} \), and a dedicated object for each new variable \( o_{\bar{s}} \), such that \( x \equiv \phi(\bar{s}, o_{\bar{s}}) \), increments/decrements another auxiliary pointer \( \bar{z}_{\text{static}} \) in a function \( \phi_{\text{static}}(\bar{z}_{\text{static}}) \) until it reaches object \( o_{\bar{s}} \), (iii) compares the content of these two state variables with a \( \text{cmp}(\bar{z}_{\text{aux}}, \phi_{\text{static}}(\bar{z}_{\text{static}})) \) instruction and (iv), jumps to target line \( j \) when the state variables differ in their content with a goto\((i, ! (y_{\bar{s}} \land \neg y_{\bar{s}}))\) instruction.

Example. Fig. 12 shows two examples of planning programs that were synthesized by our BFGP algorithm searching in our tractable GP solution-space: (left) a generalized plan for reversing a list, and (right) a generalized plan for sorting a list. Note that goto instructions are exclusively conditioned on a feature in \( L \), and that both planning programs are solutions for an infinite set of classical planning problems; they generalize with a swap action schema of arity 2 and a vector function symbol of arity 1, no matter the number of objects \( \Omega \) and no matter the state variables content, i.e. \( x \equiv \text{vector}(o_{i}) \) such that \( o_{i} \in \Omega, x \in X \) and \( D_{x} \subseteq \mathbb{N}_{0} \). In the planning program for reversing a list (left), line 0 sets pointer \( j \) to the last element of the list. Line 1 swaps in the vector the element pointed by \( i \) (initially set to zero) and the element pointed by \( j \), then pointer \( j \) is decremented, pointer \( i \) is incremented, and this sequence of instructions is repeated until the condition on line 5 becomes false, i.e. when \( j < i \), which means that reversing the list is finished. The planning program for sorting a list (right) is actually an implementation of the selection-sort algorithm. In this program, pointers \( j \) and \( i \) are used for inner (lines 5-7) and outer (lines 8-11) loops respectively, and \( min \) to point to the minimum value in the inner loop (lines 3-4); \( \neg y_{\bar{s}} \land \neg y_{\bar{s}} \) on line 2 represents whether the content of vector\((j)\) is less than the content of vector\((min)\), while \( y_{\bar{s}} \land \neg y_{\bar{s}} \) on line 7 represents whether \( j = \text{length} \) (resp. \( i = \text{length} \) on line 11). Note that object ordering affects the actual sequential plan that is eventually produced by the execution of the generalized plan but object ordering does not affect the correctness/completeness of the generalized plan. This is a common feature of structured programs e.g. a SelectionSort program is sound and complete but the actual number of executed swap instructions depends on the input list to be sorted.

5.2. The search algorithm

Given a GP problem, our heuristic search algorithm (called BFGP) implements a Best-First Search (BFS) in our solution space of planning programs with \( n \) program lines, and a RAM machine with \( |Z| \) pointers. Algorithm 1 shows the BFGP pseudo-code; 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 [98]. The BFGP algorithm runs as follows:

1. Initialization. The empty program is the root node of the search-tree developed by BFGP. This means that initially, the empty program \( \Pi_{\text{empty}} \) is evaluated by the evaluation functions \( f \in \mathcal{F} \) and then inserted into an Open\(\mathcal{F} \) list, which is implemented as a priority queue.

2. Selection. While Open\(\mathcal{F} \) is not empty, extractBestProgram gets the best partial program \( \Pi \) from Open\(\mathcal{F} \). A program \( \Pi \) is better than another program \( \Pi' \) iff exists a prefix of \( f \) values for \( \Pi \) that is smaller than the same prefix for \( \Pi' \), e.g. given \( \mathcal{F} = (f_{2}, f_{1}) \), \( \Pi \) is better than \( \Pi' \) if \( f_{3}(\Pi, P) < f_{3}(\Pi', P) \), or if they tie for \( f_{3} \) but \( f_{1}(\Pi) < f_{1}(\Pi') \). In case both programs tie for all \( f \in \mathcal{F} \), older programs (those inserted earlier in Open\(\mathcal{F} \)) will be considered better, hence extracted first from Open\(\mathcal{F} \).

3. Expansion. Once the best partial program \( \Pi \) is extracted from Open\(\mathcal{F} \), the expandProgram procedure computes all children programs of \( \Pi \) that are syntactically valid for a given set of pointers \( Z \) and bounded number of program lines \( n \). In more detail, BFGP expands \( \Pi \) by generating one successor node \( \Pi' \) for each program that result from programing the maximum undefined program line that is reached after executing \( \Pi \) on all the instances in \( P \). Given a partially specified program \( \Pi \), only its
max\_{P \in P} f_4(\Pi, P) \) line is programmable. BFGP implements this particular node expansion procedure because it guarantees that duplicate successors are not generated in the BFGP search-tree. In addition, this node expansion procedure induces a tractable branching factor of \(|A_Z'| + (n - 2) \times 4\); at a given program line BFGP can only program a planning action in \(A_Z'\) or a goto instruction that can jump to \(n - 2\) different destination program lines, and that is conditioned by any of the four different features in \(L\). The depth of the search tree developed by the BFGP algorithm is the number of program lines \(n\), since only an undefined line can be programmed.

4. Insertion. Before a new search node is inserted into the open list, the corresponding program \(\Pi'\) is executed on the classical planning instances in \(P\). This execution can result in the three following different outcomes:

(a) \(\Pi'\) is a solution for \(P\). If the execution of \(\Pi'\) solves all the instances \(P_i \in P\), then search ends, and \(\Pi'\) will be returned as a valid solution for the GP problem \(P\).
(b) \(\Pi'\) fails to solve \(P\). If the execution of \(\Pi'\) on a given instance \(P_i \in P\) fails, this means that the search node corresponding to the partially planning program \(\Pi'\) is a dead-end. The search node will be discarded, so \(\Pi'\) is not inserted into the open list. The source of failure could be either because a terminal instruction is executed but the goal condition does not hold, or because an infinite loop is detected.
(c) \(\Pi'\) may still be a solution for \(P\). This means that the execution of \(\Pi'\) on some of the classical planning instances in \(P\) reached an undefined program line (\(\Pi'\) might solve some of the instances in \(P\)). As a consequence \(\Pi'\) is inserted into \(\text{Open}_P\) by calling \text{insertProgram} at its corresponding position according to its evaluation over \(F\) functions.

Algorithm 1: Best-First Generalized Planning (BFGP).

Data: The GP problem \(P\), pointers \(Z\), program lines \(n\), a list of evaluation functions \(F\)
Result: A generalized plan \(\Pi\) that solves \(P\)

\(\text{Open}_P \leftarrow \{\Pi_{\text{open}}\} ; \)

while \(\text{Open}_P \neq \emptyset\) do

\(\Pi \leftarrow \text{extractBestProgram}(\text{Open}_P) ; \)

\(\text{ChildrenPrograms} \leftarrow \text{expandProgram}(\Pi, Z, n) ; \)

for \(\Pi' \in \text{ChildrenPrograms}\) do

if \(\text{isDeadEnd}(\Pi', P)\) then

\(\text{continue} \)

end

if \(\text{isGoal}(\Pi', P)\) then

\(\text{return}(\Pi') \)

end

\(\text{Open}_P \leftarrow \text{insertProgram}(\text{Open}_P, \Pi') ; \)

end

return \emptyset // no solution found

Example. Let us recover from the previous example (Fig. 2) the GP problem \(P = \{P_1, P_2\}\), and the partially specified program \(\Pi = 0,\text{swap}(1, j) \ 1,\text{inc}(1) \ 2,\text{dec}(j) \ 3,\text{goto}(2, ! (y_z \land \neg y_v)) \ 4,\text{empty}\) 5.end, where lines [0, 3] are programmed and only line 4 is unspecified. Imagine now that BFGP extracts this program from the open list because it has the best evaluation value. In this case, the execution of \(\Pi\) on the classical planning instances \(P_1\) and \(P_2\), implemented by the node evaluation procedure, ended in both instances at the undefined program line 4. This means that the only programmable line is 4. Assuming that two pointers are available (i.e. \(Z = [i, j]\)) we can program any of the following twelve actions in line 4. \(\{\text{inc}(i), \text{inc}(j), \text{dec}(i), \text{dec}(j), \text{cmp}(i, j), \text{set}(i, j), \text{set}(j, i), \text{test}(\text{vector}(i)), \text{test}(\text{vector}(j)), \text{cmp}(\text{vector}(i), \text{vector}(j)), \text{swap}(i, j), \text{swap}(j, i)\}\). A goto can only be programmed after a RAM action, which is not the case of line 4, since line 3 has another goto instruction.\(^6\) In other words the search node corresponding to the partially specified program from the previous example would have twelve children that could be added to the open list.

5.2.1. The evaluation functions

Here we present the evaluation and heuristic functions that guide the BFGP algorithm. The functions that range from \(f_1\) to \(f_6\) were introduced in prior work [31], while \(f_7, f_8\) and \(f_9\) are introduced for first time in this article. Here we define two different families of evaluation functions, that exploit two different sources of information, to guide a combinatorial search in our GP solution space of partially specified planning programs:

- **Program structure.** Given a partially specified planning program \(\Pi\), we define a set of evaluation functions \(f(\Pi)\), that establish preferences/priors on the structure of the aimed generalized plans. For instance, following the Occam’s razor principle a structural function can prefer generalized plans of simpler complexity or it can prefer generalized plans with more programmed lines so execution failures can be detected earlier in the search.

- \(f_1(\Pi)\), the number of goto instructions in \(\Pi\).

\(^6\) In the hypothetical case that previous line 3. would contain a RAM action, a goto instruction for jumping to lines [0, 3] conditioned by the corresponding four features in \(L\) could also be programmed at line 4.
- $f_5(\Pi)$, the number of undefined program lines in $\Pi$.
- $f_3(\Pi)$, the maximum number of occurrences that any instruction $w \in \alpha'_\Pi$ is programmed in $\Pi$, i.e.,

$$f_3(\Pi) = \max_{w \in \alpha'_\Pi} \sum_{i \in \Pi} 1(w == w_i).$$

- $f_7(\Pi)$, the max number of nested goto instructions in $\Pi$. A goto instruction jumps from an origin program line to a destination program line. We say that a goto instruction is nested when it appears within the origin and destination lines of another goto instruction.

**Fitness to the input instances.** Given a partially specified planning program $\Pi$ and a GP problem $P = \{P_1, \ldots, P_T\}$, we define a set of evaluation functions $f(\Pi, P)$ that assess the performance of $\Pi$ on $P$, executing $\Pi$ on each of the classical planning instances $P_i \in P$, $1 \leq i \leq T$. Section 3 defined the execution of a planning program on a classical planning instance as a deterministic procedure that terminates either succeeding to solve that instance or failing it. Likewise the execution of a partially specified planning program is a deterministic procedure that introduces a new termination case, reaching an unspecified program line. When the program execution terminates because an unspecified program line is reached, $f(\Pi, P)$ functions can be used to assess the cost of that program execution, as well as to estimate how far is the program from solving the given GP problem.

- $f_4(\Pi, P) = n - \max_{P \in P} f_4(\Pi, P_i)$, where $f_4(\Pi, P)$ returns the number of the undefined program line eventually reached after executing $\Pi$ on the classical planning instance $P_i \in P$.
- $f_5(\Pi, P) = \sum_{P \in P} f_5(\Pi, P_i)$, where

$$f_5(\Pi, P_i) = \sum_{x \in X_i} (v_x - G_i(x))^2.$$  

Here, $v_x \in D_x$ is the value eventually reached, for the state variable $x \in X_i$, after executing $\Pi$ on the classical planning instance $P_i \in P$, and $G_i(x)$ is the value for this same variable as specified in the goals of $P_i$. Note that for Boolean variables the squared difference becomes a simple difference. This means that for STRIPS planning problems, where all the state variables are Boolean, $f_2(\Pi, P)$ is actually a counter of how many atomic goals in $G_i$ are still not true.

- $f_6(\Pi, P) = \sum_{P \in P} |\text{exec}(\Pi, P_i)|$, where $\text{exec}(\Pi, P_i)$ is the sequence of actions induced from executing the planning program $\Pi$ on the planning instance $P_i$.
- $f_7(\Pi, P) = f_5(\Pi, P) + f_6(\Pi, P)$ is the sum of an estimation to the goal and the total accumulated cost, akin to an evaluation function for $A^\ast$ searching algorithm.
- $f_8(\Pi, P) = W \cdot f_5(\Pi, P) + f_6(\Pi, P)$ is similar to $f_8$ but the estimation to the goal is multiplied by a factor $W$, which is set to 5 by default, akin to an evaluation function for $W A^\ast$ searching algorithm.

All these functions are **evaluation functions** (i.e. smaller values are preferred). The structural functions $f_1(\Pi)$, $f_2(\Pi)$, $f_3(\Pi)$ and $f_7(\Pi)$, are all computed in linear time by traversing the bit-vector representation of $\Pi$. On the other hand, the computational complexity of the three empirical functions $f_4(\Pi, P)$, $f_5(\Pi, P)$, $f_6(\Pi, P)$, $f_8(\Pi, P)$ and $f_9(\Pi, P)$ is given by the complexity of the partially specified program $\Pi$. **Performance-based** functions accumulate a set of $T$ costs (one for each classical planning instance in the GP problem) that could actually be combined with different aggregation functions, e.g. sum, max, average, weighted average, etc. Functions $f_4(\Pi, P)$ and $f_5(\Pi, P)$ are the only cost-to-go heuristic functions; they provide an estimate on how far is a partially specified planning program from solving a GP problem. With this regard, $f_5(\Pi, P)$ requires that the goal condition of the classical planning instances in a GP problem is specified as a partial state. On the other hand $f_5(\Pi, P)$ does not impose requirements on the structure of the goal condition, so they can even be a black-box Boolean procedure over the state variables.

**Example.** We illustrate how our evaluation functions work on the following partially specified program $\Pi = 0 . \text{swap}(i, j)$ 1 . inc(i) 2 . dec(j) 3 . goto(2, ! (y_2 \land y_3)) 4 . <empty> S . end, where only line 4 is not programmed yet. The value of the evaluation functions for this partially specified program is $f_1(\Pi) = 1$, $f_2(\Pi) = 5 - 4 = 1$, $f_3(\Pi) = 0$, $f_4(\Pi) = 1$. Given the GP problem $P = \{P_1, P_2\}$ that comprises the two classical planning instances illustrated in Fig. 2, and pointers $i$ and $j$ starting at the first and last object indexes, respectively, we can compute $f_4$ and $f_5$ to evaluate how far $\Pi$ is from solving the GP problem of sorting lists, the accumulated cost $f_6$, and evaluation functions $f_8$ and $f_9$ that combine heuristic-like functions with accumulated cost. In this case $f_4(\Pi, P) = 5 - 4 = 1$, $f_5(\Pi, P) = 32$, $f_6(\Pi, P) = 14 + 14 = 28$, $f_8(\Pi, P) = 32 + 28 = 60$ and $f_9(\Pi, P) = 32 + 5 \cdot 28 = 172$.

### 5.2.2. Theoretical properties

**Theorem 4 (Termination).** Given a generalized planning problem $P$, a finite set of pointers $Z$, and a finite number of program lines $n$, the execution of the BFGP algorithm always terminates.

**Proof.** By definition of the expansion procedure of the BFGP algorithm (i), only unspecified lines can be programmed and (ii), children programs always have one more line programmed than their parent. This means that BFGP increases the number of programmed lines, until all lines are programmed. When all lines are programmed BFGP necessarily terminates, either by succeeding to solve $P$, or by failing to solve some of the classical planning instances in $P$. The only possible cause for the non-termination of the BFGP algorithm would be that BFGP could generate duplicate search nodes, allowing the infinite re-opening of an already discarded
node. By definition of the expansion procedure of the BFGP algorithm, the re-opening of an already discarded node is impossible; BFGP only allows programming the maximum undefined program line that is reached after the execution of that program on all the instances in $P$. □

**Theorem 5 (Completeness).** Given a GP problem $P$, a maximum number of pointers $|Z|$, and maximum number of program lines $n$, if there is a planning program $\Pi$ within these bounds that solves $P$, then the BFGP algorithm can compute it.

**Proof.** The BFGP algorithm implements a complete enumeration of the entire space of planning programs with a maximum number of pointers $|Z|$ and maximum number of program lines $n$ except: (i), a search node was identified as a dead-end or (ii), the ancestor of a search node was identified as a dead-end. BFGP is safe because it only discards a search node when its corresponding partially specified planning program failed to solve an input planning instance (which is actually the definition for not being a GP solution). Furthermore, if a partially specified planning program failed to solve an input planning instance, any planning program that can be built programming the remaining undefined program lines will also fail to solve that problem. □

**Theorem 6 (Soundness).** If the execution of the BFGP algorithm on a GP problem $P$ outputs a generalized plan $\Pi$, this means that $\Pi$ is a solution for $P$.

**Proof.** The BFGP algorithm 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), BFGP extracted from the open list a planning program whose execution, in all the classical planning instances $P_i \in P$, resulted successful. This is actually the definition of a solution for a GP problem. □

**Theorem 7 (Time and Memory).** The time and memory consumption of the BFGP algorithm are characterized by the big-Oh expression $O((|A'_Z| + (n-2) \times 4)^n)$.

**Proof.** The BFGP algorithm is an implementation of a BFS, whose memory and time complexity are characterized as $O(b^d)$, where $b$ denotes the branching factor and $d$ denotes the depth of the corresponding search tree. The branching factor of the search tree induced by the BFGP algorithm is the number of different instructions that can be programmed at an undefined program line, which is $b \leq |A'_Z| + (n-2) \times 4$; gotos can only be programmed after RAM operations. The depth of the search tree induced by the BFGP is given by the maximum number of program lines $n$. □

BFGP may be incomplete in the sense that either the bound $n$ on the maximum number of program lines, or the maximum number of pointers available $|Z|$, may be too small to accommodate a solution to a given GP problem. With respect to solution quality BFGP does not guarantee that the computed planning programs are optimal. BFGP can however be run in anytime mode and use $f_6(\Pi, P)$ to rank GP solutions according to their execution cost in the classical planning instances that are comprised in the given GP problem (e.g. to prefer a sorting planning program with smaller computational complexity).

6. Evaluation

This section evaluates the empirical performance of our GP as heuristic search approach. All experiments are performed in an Ubuntu 20.04 LTS, with AMD® Ryzen 7 3700x 8-core processor × 16 and 32 GB of RAM.7

6.1. Benchmarks

We report results in nine different domains; three propositional domains and six numeric domains. In the propositional domains the functions $\Phi$ that induce the state variables are Boolean. In the numeric domains these functions are positive numeric functions. Next, we provide more details on each of the nine domains:

- **Corridor**, an agent moves from an arbitrary initial location to a destination location in a corridor.
- **Gripper**, a robot must pick all balls from room A and drop them in room B.
- **Visitall**, starting from the bottom-left corner of a squared grid, an agent must visit all cells.
- **Fibonacci**, compute the $n^{th}$ term of the Fibonacci sequence.
- **Find**, counts the number of occurrences of a specific value in a list.
- **Reverse**, for reversing the content of a list.
- **Select**, find the minimum value of a list.
- **Sorting**, for sorting the values of a vector.
- **Triangular Sum**, compute the $n^{th}$ triangular number.

---

7 The source code, evaluation scripts, and used benchmarks can be found at: https://github.com/jsego/bfgp-pp.
Corridor, gripper and visitall are propositional, the remaining six domains are numeric. For each domain, we build a GP problem that comprises ten randomly generated classical planning instances; in the case of the corridor domain, instances go from corridors of length 3 to 12; in gripper, instances have from 2 to 11 balls in room A to be dropped in room B; in visitall instances are squared grids ranging from 2 × 2 to 11 × 11 cells; Fibonacci and triangular sum instances ranging from the 2nd to the 11th number in the sequence; and the remaining domains, find, reverse, select and sorting have instances with vectors from size 2 to 11 that are initialized with random content. The result of arithmetical operations in these domains is bounded to 105 in the synthesis of GP solutions, and to 106 in the validation of GP solutions.

All domains include the base set of RAM instructions \{inc(z_1), dec(z_1), cmp(z_1, z_2), set(z_1, z_2) \mid z_1, z_2 \in Z\}, such that \(z_1\) and \(z_2\) are pointers of the same type, and the RAM instructions \{test(\phi(z_T)), cmp(\phi(z_T), \phi(z_T')) \mid z_T, z_T' \in Z^\phi(\phi)\}, for each function \(\phi \in \Phi\) and where function parameters and pointers also are of the same type. We remind the reader that compare instructions are only defined for numeric functions. In addition, each domain contains the regular planning action schemes that do not affect the FLAGS. We also recall that planning actions are modeled as they are always executable, but that their effects only update the current state if their preconditions hold in the current state. Otherwise the execution of an action has no effect.

- **Propositional domains.** The corridor domain needs two planning action schemes, move_left(z_1, z_2) and move_right(z_1, z_2), to move from location at \(z_1\) to location at \(z_2\), which must be exactly one location to the left or the right of \(z_1\), respectively. The gripper domain includes the following three action schemes; move(z_1, z_2) to denote the robot is moving from the room pointed by \(z_1\) to the one pointed by \(z_2\), pick(z_1, z_2, z_3) to pick the ball pointed by \(z_1\), at room pointed by \(z_2\), and with the gripper pointed by \(z_3\), and drop(z_1, z_2, z_3), to drop ball \(z_1\) at room \(z_2\) with gripper \(z_3\). Visitall needs four action schema to move in the four cardinal directions of grid, i.e. move_left(z_1, z_2, z_3) to move one to the left from column \(z_1\) to \(z_2\), at row \(z_3\) (resp. for move_right), and move_up(z_1, z_2, z_3) to move once up from row \(z_1\) to \(z_2\), at column \(z_3\) (resp. for move_down); every move visits the destination cell.

- **Numeric domains.** The triangular sum and Fibonacci domains include the action schemes vector_inc(z_1) and vector_dec(z_1), to increase and decrease by one the content of the vector at \(z_1\), and the action scheme vector_add(z_1, z_2) for adding the content of the vector at \(z_2\) to the content at \(z_1\). Select only requires one action schema to mark a specific vector index at \(z_1\) as selected, i.e. select(z_1). Find includes the accumulate(z_1) action schema for counting the number of occurrences of the target element. Reverse and Sorting include the swap(z_1, z_2) action schema to swap the vector values addressed by \(z_1\) and \(z_2\).

### 6.2. Synthesis and validation of GP solutions

Here we present the first experiments to evaluate the performance of the BFGP algorithm. First, we assess every evaluation/heuristic function \(f_i\) by running BFGP(\(f_i\)). Then we show the solutions generated by the best configuration, BFGP(\(f_5\)). Last, the synthesized solutions are validated w.r.t. test sets of larger instances (i.e. larger number of objects).

#### 6.2.1. Performance of BFGP(\(f_i\))

Table 1 details the results of the first synthesis experiment where the BFGP algorithm uses each of our nine different evaluation/heuristic functions (the computation bounds are 3, 600 seconds of CPU time and 32GB of memory and best results are shown in bold). Regarding structure-based functions \(f_2\) dominates in all domains and metrics (except in the reverse domain where \(f_1\) is faster and \(f_1\) expands fewer nodes) and it also has the highest coverage solving 7 out of 9 domains (\(f_1\), \(f_2\) and \(f_7\) have lower coverage failing in the same four domains, namely corridor, gripper, sorting and visitall). Regarding performance-based functions, \(f_5\) is the clear winner with the best scores in more than half of domains and with full coverage over the benchmarks, followed by \(f_1\) and \(f_5\) that cover 6 domains but improving the metrics of \(f_1\) in several domains, e.g., \(f_5\) has the lowest memory consumption in all domains and expands and/or evaluates fewer nodes in Fibonacci, reverse and triangular sum.

Table 2 summarizes the results from Table 1, grouping results by domains and averaging the metrics by the total number of functions that solved each domain. There are 5 domains which are solved by all the nine evaluation/heuristic functions. In the rest of domains, there are at least 5 or more functions that do not solve them, i.e. gripper is only solved by \(f_2\), \(f_4\), \(f_5\) and \(f_6\); corridor and visitall are the least solved domains (only \(f_5\) solved them); and sorting which is solved by \(f_2\) and \(f_5\) but it is the hardest in terms of each metric average.

#### 6.2.2. The synthesized solutions

Fig. 13 shows the programs computed by BFGP(\(f_5\)). In Corridor there are two pointers, \(l_1\) and \(l_2\), that start pointing to the first location; the solution moves the agent to the rightmost cell, then one by one to the left while not at goal cell. In Fibonacci, pointers \(i\) and \(j\) are used to compute the \(n\)-th Fibonacci number, where \(i\) addresses the \(F_i\) number to which \(F_{i-1}\) and \(F_{i-2}\) are added using \(j\) as an auxiliary pointer; and finishes when \(i\) reaches the \(n\)-th element (the last one). In Find, there is a pointer \(i\) to iterate over a vector, accumulating for each vector location the occurrences of a target value.

The Gripper solution uses one pointer for balls (\(b_1\)), two for rooms (\(r_1\) and \(r_2\)) and one for grippers (\(g_1\)); for each ball \(b_1\), the agent will pick it up from room \(r_1\) (always room A) using gripper \(g_1\) (always left gripper), sets \(r_2\) to room B, moves from A to B, drop ball \(b_1\) at room B, goes back to room A, and continues with the next ball. The Reverse domain uses two pointers \(i\) and \(j\) and finds a

---

8. For reproducibility reasons we fix the random seed to generate the classical planning instances in the GP problems.
Table 1
We report the number of program lines $n$, and pointers $|Z|$ per domain, and for each evaluation/heuristic function, CPU time (secs), memory peak (MBs), and the numbers of expanded and evaluated nodes. TE stands for Time-Exceeded (>1h of CPU time). Best results in bold.

| Domain       | $n$, $|Z|$ | $f_1$ | $f_2$ | $f_3$ |
|--------------|------------|-------|-------|-------|
| Corridor     | 10, 2  | TE   | -    | -     | -    | 86   | 41   | -    | 16.8K | 78.1K | TE   | -    | -     |
| Fibonacci    | 7, 2   | 167  | 164  | 42.6K | 373.8K | 0    | 4    | 38.0K | 38.1K | 0     | 4    | 38.0K | 38.1K |
| Find         | 4, 1   | 0    | 4    | 9     | 31    | 1,988| 5    | 1.1M  | 1.1M  | 80    | 68   | 88.8K | 164.0K |
| Gripper      | 8, 4   | TE   | -    | -     | -     | 5,2  | 1.1M | 1.1M  | TE    | -     | -    | -     |
| Reverse      | 7, 2   | 82.6K | 148.8K | 0    | 5    | 207 | 238 | 1    | 6    | 408   | 3.0K |
| Select       | 7, 2   | 354.3K | 361.3K | 0     | 5    | 207 | 238 | 1  | 6   | 408   | 3.0K |
| Sorting      | 9, 2   | TE   | -    | -     | -     | 0.8K | 3.5M | 3.5M  | TE    | -     | -    | -     |
| T. Sum       | 5, 2   | 1    | 6    | 373   | 2.7K  | 1    | 5    | 207   | 238   | 1    | 6    | 408   | 3.0K |
| Vistall      | 13, 4  | TE   | -    | -     | -     | 160.2| 60.2 | 96.0K | 177.3K | 883.1 | 4.9 | 730.6K | 730.6K |
| Average      |        | 160.2| 60.2 | 96.0K | 177.3K | 883.1| 4.9  | 730.6K | 730.6K | 334.2 | 90.4 | 188.9K | 246.4K |

Table 2
We report for each domain, the time (secs), memory peak (MBs), and expanded and evaluated nodes averaged by the number of functions that solved the domain in Table 1.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Time</th>
<th>Mem.</th>
<th>Exp.</th>
<th>Eval.</th>
<th>#f, Solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corridor</td>
<td>86.0</td>
<td>41.0</td>
<td>16.8K</td>
<td>78.1K</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>442.6</td>
<td>227.9</td>
<td>0.2M</td>
<td>0.6M</td>
<td>9</td>
</tr>
<tr>
<td>Find</td>
<td>0.0</td>
<td>4.0</td>
<td>21.6</td>
<td>31</td>
<td>9</td>
</tr>
<tr>
<td>Gripper</td>
<td>1,011.3</td>
<td>22.0</td>
<td>1.1M</td>
<td>1.2M</td>
<td>4</td>
</tr>
<tr>
<td>Reverse</td>
<td>144.1</td>
<td>56.6</td>
<td>165.5K</td>
<td>186.0K</td>
<td>9</td>
</tr>
<tr>
<td>Select</td>
<td>436.8</td>
<td>83.1</td>
<td>289.4K</td>
<td>322.4K</td>
<td>9</td>
</tr>
<tr>
<td>Sorting</td>
<td>2,805.5</td>
<td>1,244</td>
<td>2.2M</td>
<td>4.9M</td>
<td>2</td>
</tr>
<tr>
<td>T. Sum</td>
<td>0.8</td>
<td>5.8</td>
<td>348.7</td>
<td>2.2K</td>
<td>9</td>
</tr>
<tr>
<td>Vistall</td>
<td>998.0</td>
<td>67.0</td>
<td>116.6K</td>
<td>122.7K</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f_4</th>
<th>f_5</th>
<th>f_6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f_4</td>
<td>f_5</td>
</tr>
<tr>
<td></td>
<td>f_6</td>
<td>f_7</td>
</tr>
<tr>
<td></td>
<td>f_8</td>
<td>f_9</td>
</tr>
</tbody>
</table>

solution with $O(n^2)$ complexity of a vector of size $n$; it moves all values from $j$ to $n-1$ indexes once to the right and places the last element in the $j$-th location, using $i$ as an auxiliary pointer; then increases $j$ by one until it reaches the end of the vector. The Select domain has two pointers $i$ and $j$; it iterates over the vector with pointer $i$, and assigns $i$ to $j$ every time the value pointed by $i$ is smaller than the one pointed by $j$, at the end $j$ will point to the smallest value which will be selected.

The Sorting solution is succinct but easy to interpret; while visiting each vector index in increasing order with $i$ and $j$ pointers s.t. $j = i - 1$, if $i$'th value is smaller than $j$'th it moves the value backwards in the vector until it is relatively sorted, then proceeds searching for the next pair of adjacent and unsorted values, until all values are sorted. In Triangular Sum, given a vector initialized as $v_{i,d} = d x$, each index is visited in increasing order with pointers $i$ and $j$ such that $j = i - 1$, executing for each one $v_i \leftarrow v_i + v_j$. The last domain, Vistall, has two pointers $r1$ and $r2$ for rows, and two more $c1$ and $c2$ to iterate over columns. Since the agent starts
**Fig. 13.** Solutions computed by BFGP($f_j$).

**Fig. 14.** Alternative solution to the Gripper domain. This program is interpreted as moving $r_1$ to room A if both pointers are initially in room B, otherwise move $r_2$ to room B, then for each ball run a pick action, move to room B, drop it, move back to room A, until all balls have been moved.

In the bottom-left corner, the strategy consists of visiting the grid row by row, going first to the right, once up, then all to the left, until all rows are visited.

BFGP implements an *inductive* approach to GP, this means that the computed generalized plans are synthesized with the only requirement of satisfying the formal specification given by the finite set of input examples. Therefore, in domains where input examples follow a clear distribution, it may then be exploited by BFGP (e.g. the locations of a corridor or grid, or the indexes of a vector, which are naturally expressed in a specific order). However, the *object ordering* does not affect the correctness/completeness of the approach, e.g. the *Sorting* program is able to sort any input list no matter its size or content, or the *Blocks Ontable* program (Fig. 15) puts all blocks onto the table no matter the block names, their ordering, or the initial towers settings, with the cost of an extra iteration. In the particular case of *Gripper*, rooms are constants in the domain, for that reason they always appear in the same order. In case we shuffle the rooms order for each instance, we would either need more lines to solve *Gripper* (i.e. solve the problem for each possible permutation), or add the goal information in the initial state, as it is done in *Corridor* and *Grid*, to keep the solution short. In the latter, the *planning program* of Fig. 14 solves all *Gripper* instances with no particular order in rooms.

### 6.2.3. Validation of the synthesized solutions

Here we validate the BFGP($f_5$) solutions of Fig. 13 with a larger and harder set of instances. Table 3 reports the CPU time, and peak memory, yield when running the solutions synthesized by BFGP($f_5$) on a validation set. All the solutions synthesized by BFGP($f_5$) were successfully validated. 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 validates terminating programs with less memory and much faster [41].

In the validation set, each state variable from the planning domain is bounded by 10^9, instead of 10^12 which was the synthesis bound. Corridor is validated over 100 instances with corridors of length \( n \in [13, 112] \) and random initial and goal locations. Gripper is validated over 1,000 instances of \( n \in [12, 1011] \) balls that are initially in room A and need to be moved to room B. Fibonacci has a validation set of 33 instances, ranging from the 12th Fibonacci term to the 44th, i.e. the integer 701,408,733 (the 45th number would overflow the validation bound). The solutions for Select, and Find domains, are validated on 100 instances each, with vector sizes ranging from 100 to 1,090, and random integer elements bounded by 10^11. Similarly, Reverse and Sorting have 100 validation instances with vectors of random integers, but their sizes range from 12 to 111. The solution for Triangular Sum is validated over 1,000 instances, the last one corresponding to the 1,011th term in the sequence, i.e. the integer 511,566. In Visitall, there are 50 validation instances with squared grids ranging from 12 x 12 to 61 x 61.

### Table 3

<table>
<thead>
<tr>
<th>Domain</th>
<th>Instances</th>
<th>Time_(\text{mem} )</th>
<th>Mem_(\text{mem} )</th>
<th>Time</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corridor</td>
<td>100</td>
<td>0.42</td>
<td>11.4MB</td>
<td>0.13</td>
<td>10.8MB</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>33</td>
<td>0.03</td>
<td>5.8MB</td>
<td>0.02</td>
<td>5.6MB</td>
</tr>
<tr>
<td>Find</td>
<td>100</td>
<td>9.84</td>
<td>70.1MB</td>
<td>0.69</td>
<td>48.5MB</td>
</tr>
<tr>
<td>Gripper</td>
<td>1,000</td>
<td>153.53</td>
<td>1,007.6MB</td>
<td>16.88</td>
<td>968.2MB</td>
</tr>
<tr>
<td>Reverse</td>
<td>100</td>
<td>6.67</td>
<td>30.2MB</td>
<td>1.40</td>
<td>12.5MB</td>
</tr>
<tr>
<td>Select</td>
<td>100</td>
<td>18.39</td>
<td>163.8MB</td>
<td>1.38</td>
<td>113.2MB</td>
</tr>
<tr>
<td>Triangular Sum</td>
<td>1,000</td>
<td>72.69</td>
<td>706.2MB</td>
<td>9.59</td>
<td>685.7MB</td>
</tr>
<tr>
<td>Visitall</td>
<td>50</td>
<td>44.48</td>
<td>403.3MB</td>
<td>3.83</td>
<td>79.7MB</td>
</tr>
</tbody>
</table>

### Table 4

For each domain we report, CPU time (secs), memory peak (MBs), num. of expanded and evaluated nodes of BFGP\( (f_i, f_j) \), BFGP\( (f_s, f_j) \) and BFGP\( (f_s) \). Best results in bold.

<table>
<thead>
<tr>
<th>Domain</th>
<th>BFGP( (f_i, f_j) )</th>
<th>Time</th>
<th>Mem.</th>
<th>Exp.</th>
<th>Eval.</th>
<th>BFGP( (f_s, f_j) )</th>
<th>Time</th>
<th>Mem.</th>
<th>Exp.</th>
<th>Eval.</th>
<th>BFGP( (f_s) )</th>
<th>Time</th>
<th>Mem.</th>
<th>Exp.</th>
<th>Eval.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corridor</td>
<td></td>
<td>38</td>
<td>25</td>
<td>7.8K</td>
<td>44.2K</td>
<td></td>
<td>38</td>
<td>25</td>
<td>7.8K</td>
<td>44.2K</td>
<td></td>
<td>86</td>
<td>41</td>
<td>16.8K</td>
<td>78.1K</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1,227</td>
<td>139</td>
<td>579.7K</td>
<td>718.2K</td>
<td>457</td>
<td>139</td>
<td>293.8K</td>
<td>431.2K</td>
<td>187</td>
<td>189</td>
<td>68.5K</td>
<td>457.7K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Find</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>4</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gripper</td>
<td>19</td>
<td>37</td>
<td>3.2K</td>
<td>69.5K</td>
<td>20</td>
<td>37</td>
<td>3.4K</td>
<td>69.6K</td>
<td>20</td>
<td>39</td>
<td>3.6K</td>
<td>74.9K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reverse</td>
<td>125</td>
<td>61</td>
<td>134.6K</td>
<td>163.8K</td>
<td>201</td>
<td>61</td>
<td>219.9K</td>
<td>220.9K</td>
<td>186</td>
<td>81</td>
<td>220.1K</td>
<td>221.0K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select</td>
<td>65</td>
<td>64</td>
<td>23.6K</td>
<td>152.9K</td>
<td>65</td>
<td>64</td>
<td>23.6K</td>
<td>152.9K</td>
<td>78</td>
<td>81</td>
<td>29.0K</td>
<td>196.3K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorting</td>
<td>1,137</td>
<td>1,604</td>
<td>516.4K</td>
<td>3.9M</td>
<td>1,215</td>
<td>1,596</td>
<td>555.2K</td>
<td>4.0M</td>
<td>2,054</td>
<td>2,483</td>
<td>988.9K</td>
<td>6.3M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T. Sum</td>
<td>1</td>
<td>6</td>
<td>304</td>
<td>2.2K</td>
<td>1</td>
<td>6</td>
<td>304</td>
<td>2.2K</td>
<td>1</td>
<td>6</td>
<td>343</td>
<td>2.3K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visitall</td>
<td>276</td>
<td>30</td>
<td>32.3K</td>
<td>42.8K</td>
<td>969</td>
<td>43</td>
<td>116.6K</td>
<td>121.6K</td>
<td>998</td>
<td>67</td>
<td>116.6K</td>
<td>122.7K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>320.9</td>
<td>218.9</td>
<td>144.2K</td>
<td>566.0K</td>
<td>329.6</td>
<td>219.4</td>
<td>135.6K</td>
<td>560.3K</td>
<td>401.1</td>
<td>332.3</td>
<td>160.4K</td>
<td>828.1K</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.3. Performance of BFGP with function combinations

The base performance of BFGP with a single evaluation/heuristic function is improved combining both structural and cost-to-go information; we can guide the search of BFGP with a cost-to-go heuristic function and break ties with a structural evaluation function, and vice versa. Thus, we run all configurations of BFGP\( (f_i, f_j) \) and BFGP\( (f_s, f_j) \) such that \( f_i \in \{f_1, f_2, f_3, f_5\} \) and \( f_j \in \{f_4, f_5, f_6, f_7, f_9\} \), and select the configuration that solves all domains with the best average time. There are 40 BFGP\( (f_1, f_2) / \text{BFGP}(f_3, f_5) \) configurations, but only BFGP\( (f_3, f_5) \) and BFGP\( (f_5, f_3) \) are able to solve all domains. The performance of these two configurations is then compared against BFGP\( (f_3) \), since it is the only single evaluation/heuristic function that solve all domains in the previous experiment. Table 4 summarizes that comparison, showing that BFGP\( (f_3) \) is improved in every domain either by BFGP\( (f_3, f_5) \) or BFGP\( (f_5, f_3) \). Furthermore, BFGP\( (f_3, f_5) \) has the best average performance in time and memory, shortly followed by BFGP\( (f_5, f_3) \) that has the best average in node expansions and evaluations, empirically proving that goal-oriented functions might be benefited when combined with structural-based functions.

6.4. Comparative synthesis performance with related GP solvers

The comparative analysis of GP solvers performance is complex mainly because of their different assumptions, problem specification, feature language, hyper-parameters, and different solution representations (non-/deterministic, boolean/numeric, ...) [21]. In this regard, we have chosen the two closest GP solvers in terms of solution representation, i.e. Planning Programs (PP) [33] and Hierarchical Finite State Controllers (HFSC) [61]. Both approaches are compilation-based, where the input is a classical planning domain with some instances, and an upper bound in the number of program lines \( n \) for PP, or in the number of controller states \( |Q| \) for
HFSC. Both approaches produce a single domain and instance that can be solved with an off-the-shelf classical planner. Solutions to the produced instances are computed following a top-down strategy, with the LAMA-2011 \cite{99} planner (first solution setting) of the Fast-Downward \cite{8} system, that produce sequences of interleaved programming and executing actions, from which the programs, controllers and classical plans can be induced.

Section 5 already discussed the relation between the search spaces of PP (worst case intractable, e.g. integer representation) and BFGP (tractable with bounded size). PP and HFSC are proven to have equivalent solutions in the other representation \cite{61}, so the theory proven in Section 5 actually applies to both. A priori, there are some strengths and weaknesses that were already previously identified for PP and HFSC \cite{61,33}:

- **Strengths:**
  - GP problems can be solved with an off-the-shelf classical planner,
  - classical planners are biased towards shorter solution plans (which may produce succinct generalized plans),
  - good performance for small bounds and problems with small number of objects.

- **Weaknesses:**
  - Computation sensitive to the order of the input instances and their number of objects,
  - scalability dramatically drops down when the number of input instances grows,
  - require hand-coding features as derived predicates in several domains or extra knowledge to compute generalized plans with deterministic behavior, e.g. in gripper domain the balls need to be serialized to know which ball is the next one to be picked,
  - the synthesized generalized plans can over-fit because the feature language includes all fluents in the set of classical planning instances given as input.

Compared to PP and HFSC, the strengths and weakness of BFGP are:

- **Strengths:**
  - The computation of generalized plans is not affected by the order of input instances,
  - the scalability of BFGP decreases smoothly and continuously with the number of objects,
  - solutions do not over-fit because of the feature language (over-fitting is only due to poorly sampled input instances),
  - domains do not require extra knowledge (BFGP implements an agnostic feature generation).

- **Weaknesses:**
  - BFGP is not using off-the-shelf classical planning machinery, although most of them could be adapted \cite{65},
  - generalized plans are longer because of pointer manipulations, which may reduce performance in simple instances,
  - BFGP may require to increase the default number of input pointers.

The experimental setting for comparing GP solvers uses the same $|P| = 10$ random input instances as in BFGP synthesis experiments. Since we observed that PP and HFSC were limited by this number, we have however proceed on removing the largest instance from the pool of instances, until one of them solved the problem without over-fitting. In addition, we encoded extra knowledge in the PP and HFSC domains, when necessary, with the form of derived predicates to guarantee generalization, which in turn helped to compute shorter generalized plans, e.g. in corridor domain it requires to know whether the agent is at the goal or in the rightmost location; and in gripper it requires to serialize the balls and know whether no more balls are in room A; and so on. The comparison of the three GP solvers is shown in Table 5; even with the advantage given to PP and HFSC, with the hand-crafted extra knowledge in their domains and the reduction of input instances (yielding to shortest number of executed instructions in half of domains i.e. $\sum |II|$, BFGP outperforms PP and HFSC in many aspects, i.e. full coverage over the benchmarks, best computation time in 6 out of 9 domains, and shortest classical plans in all domains i.e. $\sum |I|$, with less assumptions.

| Domain          | $|P|$ | $|I|$  | $|II|$ | $|I|$  | $|II|$ | $|I|$  | $|II|$ |
|-----------------|------|-------|-------|-------|-------|-------|-------|
| PP              | 32   | 3     | 2     | 1     | 2     | 1     | 1     |
| HFSC            | 32   | 3     | 2     | 1     | 2     | 1     | 1     |
| BFGP ($f_3$, $f_4$) | 32   | 3     | 2     | 1     | 2     | 1     | 1     |

Table 5

Computing CPU time (secs) for solving domains in the GP compilation approaches, i.e. PP \cite{33} and HFSC \cite{61}, and $BFGP(f_3, f_4)$. Accumulated size of the classical plans, $\sum |I|$, and number of instructions, $\sum |II|$. |
6.5. Validation of GP solutions in more complex domains

Here we present several GP benchmarks, with known polynomial time solutions, that are too complex for our current BFGP algorithm (within the given time and memory bounds). Our aim is showing that our approach is expressive enough to represent solutions to GP problems coming from IPC planning domains, noise-free supervised classification tasks, and numeric domains. These solutions are succinctly represented as GP plans, instead of long sequences of grounded actions for large problems, and validated efficiently without being affected by the grounding methods of current off-the-shelf classical planners.

- **Blocks Ontable**, towers of blocks where all blocks must be placed on the table.
- **Grid**, an agent has to move from an arbitrary location to a destination one in a 2D grid.
- **Miconic**, is an elevator problem where passengers at origin, wait for the elevator to enter, and then served at their destination floor.
- **Michalski Trains**, is a classic of relational supervised machine learning. A binary noise-free classification task with 10 trains that either go east or west, and multiple features such as the number of wheels, wagons, or their shape for each train among others. The goal is to learn the features that classify all trains in the right direction.
- **Satellite**, consists of taking images of different targets with instruments that are boarded in satellites. In addition, instruments need to be calibrated and in specific modes for taking each image; and each satellite has only power for one instrument at a time, so it needs to switch the current instrument off, switch on the next and calibrate it, before using a new instrument for taking images.
- **Sieve of Eratosthenes**, is a method to find prime numbers up to a certain bound using only additive and iterative mechanisms.
- **Spanner**, consists of tighten all loose nuts at the end of a corridor, with the picked spanners along the corridor. Spanners can only be used once, and when the agent moves to the next room it can not go back, so if there are unpicked spanners in visited rooms the task could become unsolvable.

Fig. 15 shows the hand-coded solutions for these benchmarks. In **Blocks Ontable**, given \( n \) blocks the complexity of the solution is cubic, i.e. \( O(n^3) \), where it searches \( n \) times, every \( o1 \) block that is on top of an \( o2 \) block, then unstack and put \( o1 \) down on the table. In **Spanner**, an agent picks up all available spanners in location \( l2 \), walks to the next \( l1 \) location and repeats the process until it reaches the last location (the gate), collecting all spanners on its path; once in the gate, it tightens each loose nut with a spanner. The solution to **Michalski Trains** is summarized as, each train \( l1 \) will go east if it has a car which is closed and short, otherwise it will go west. In **Sieve of Eratosthenes** all numbers are initially classified as primes, and it should decide whether they are not; so it iterates over \( i \) and uses \( j \) and \( k \) as auxiliary pointers, where the first acts as a counter that ranges from 0 to \( i \), and second adds up to the next multiple of \( i \), i.e. \( k \% i = 0 \); then every \( k \)-th number will be set to no prime, \( i \) is increased by one and the process repeats until \( i \) reaches the last element. In **Grid**, the agent moves to the top-right corner, then it moves left while not at the goal column, then down while not at the goal row, visiting the resulting coordinate. In **Miconic**, the elevator moves to the uppermost floor, then for every floor down, it boards and departs all available passengers, once in the bottom floor, it moves upwards serving the remaining passengers that are in the elevator until reaching the uppermost floor again. The last domain, **Satellite**, is the most complex because it requires iteration over multiple variable types, i.e. satellites, instruments, modes and directions. The solution to this domain consists of switching off all instruments and turning all satellites to the first direction; then for each satellite, the \( i1 \) instrument is switched on, calibrated with its calibration target direction \( d2 \), and used to take images of every direction \( d2 \) in every mode \( m1 \); once it finishes, the satellite turns to the first direction \( d1 \) again, switches off the current instrument, and continues with the next one, until all satellites have used all their instruments.

We get some main take away lessons from the analysis of Fig. 15 solutions; solutions have common high-level structures, that either iterate over all combinations of variable types (i.e. **Blocks**, **Miconic**, **Satellite**, ...), or build complex logic queries (i.e. **Michalski Trains**) which in some cases require the goal information to be coded in the initial state (i.e. **Grid** with **goal_column** and **goal_row**). This suggests that planning programs may be synthesized more efficiently using predefined structures (such as **FOR** or **IF-THEN-ELSE**) constructs [100]) although this is out of the scope of this paper.

Table 6 shows the validation results in complex domains, where validation without infinite detection again scales better. All domains are successfully validated (except **Blocks Ontable** and **Satellite** with infinite detection mode that get a time- and memory-exceeded, respectively). **Blocks Ontable** can be solved with 13 lines and 3 pointers, and the validation set consists of 100 instances that range from 12 to 111 blocks. **Grid** requires 21 lines of code and 4 pointers, and it is validated with 50 instances with grids between 12 \( \times \) 12 and 111 \( \times \) 111 size. **Miconic** needs 25 lines and 3 pointers, and 100 instances that validates from 12 floors and 18 passengers to 111 floors and 166 passengers. **Michalski Trains** uses 15 lines and 6 pointers to classify all the trains in the unique classical task with 10 trains and their features. **Satellite** is by difference the most complex in terms of required lines and pointers, which are 43 and 5, respectively. Its validation set consists of 20 instances, starting with 11 satellites, 22 instruments and modes, and 44 directions, and finishing with 30 satellites, 60 instruments and modes and 120 directions. **Sieve of Eratosthenes** requires 16 lines and 3 pointers to classify either as prime or non-prime, all the numbers comprised in the first 114 natural numbers. **Spanner**, uses 14 lines and 5 pointers to solve all 100 instances of the validation set, that range from 24 spanners and nuts and a corridor with 12 locations, to 222 spanners and nuts and a corridor with 111 locations.
7. Conclusions

The paper presented an innovative solution space for GP that enables the definition of a heuristic search approach to GP. This novel solution space for GP is independent of the number of input planning instances in a GP problem, and the size of these instances (i.e. the number of objects, state variables, and their domain sizes). Therefore our GP as heuristic search approach can handle large sets of state variables with large numerical domains, e.g. integers.
We believe that this work is a step-forward towards building stronger connections between the areas of automated planning and programming. The work presented a formalization of classical planning as a vector transformation task, which is a common programming task. According to this formalism, computing a sequential plan for this task is computing a composition of vector transformation operations. Likewise computing a generalized plan is computing an algorithmic expression of the vector transformations. With this regard, the BFGP algorithm starts from an empty program, but nothing prevents us from starting search from a partially specified generalized plan to develop new online approaches that scale up better. In fact, local search approaches have already shown successful for both planning and program synthesis.

Our cost-to-go heuristics are still less informed than current state-of-the-art heuristics for classical planning; note that our heuristics only consider goals that are explicitly provided in the problem representation. A clear example is the Euclidean distance, and that for STRIPS planning problems is actually a goal counter. We believe that better estimates may be obtained by building on top of the powerful techniques of modern planning heuristics. In more detail, a promising approach for the development of more informative heuristics for GP is to consider sub-goals, that are not explicitly given in the problem representation. For instance sets of sub-goals can be discovered as a pre-processing step, without grounding, regarding the set of relevant atoms that are traversed by the polynomial IW(1) algorithm, when achieving individual goals. Besides landmarks, heuristic planners implement complementary ideas such as helpful actions or multiple queues for combining different heuristics, or novel-based exploration. Incorporating those classical planning technologies into the GP as heuristic search approach is a promising research direction. In addition to more informative heuristics, we are interested in more expressive solution representations, where not only goal-agnostic (e.g. Blocks Ontable in Fig. 6) and goal-oriented (e.g. Grid in Fig. 6) generalized plans are computed, but also solutions which include distance functions that measure progress towards (sub-)goals; distance functions are known to be required to represent compact algorithmic solutions to polynomially-approximable (poly-APX) domains.

Since we are approaching GP as a classic tree search, a wide landscape of effective techniques, coming from combinatorial search and classical planning, could help to improve the base performance of our approach. We mention some of the more promising ones. Exploration in search can be more effective when adding one open list per evaluation function and more sophisticated mechanisms could be implemented for handling closed nodes. For instance, delayed duplicate detection could be implemented to manage large closed lists with magnetic disk memory. Further, once a search node is canceled (e.g. because $f_j(I, P_i)$ identified that the planning program fails on a given instance), any program equivalent to this node should also be canceled, e.g. any program that can be built with transpositions of the causally-independent instructions. Given that the depth of the search-tree is bounded, techniques coming from SAT/CSP/SMTs, such a non-chronological backtracking, limited discrepancy search, or taboo search, might also result effective to improve our approach. SATPLAN planners exploit multiple-thread computing to parallelize search in solution spaces with different bounds. This same idea could be applied to multiple searches for GP solutions with different program sizes.

Last but not least, another interesting research direction is the extension of our GP as heuristic search approach for computing generalized plans starting from different input settings. For instance, the computation of generalized plans from a set of plan traces that demonstrates how to solve several planning problems. We are also interested on exploring the application of our GP as heuristic search approach to planning problems that are not goal-oriented, where the objective is to maximize a given utility function. In this particular setting, ideas from approximated policy iteration, and reinforcement learning, could be incorporated to our framework. With this regard, we are exploring the extension of our approach to GP problems that include real state variables. We believe that we can address this kind of GP problems by introducing the notion of precision for the comparison of real variables, and redefining accordingly our mechanism for the update of the FLAGS registers.

| Domain                  | $n_i|Z|$ | $|P|$ | $Time_{pe}$ | $Mem_{pe}$ | $Time$ | $Mem$ |
|-------------------------|--------|------|------------|------------|--------|-------|
| Blocks Ontable          | 13, 3  | 100  | TE         | –          | 148.59 | 17 MB |
| Grid                    | 21, 4  | 50   | 0.67       | 14 MB      | 0.15   | 10 MB |
| Miconic                 | 25, 3  | 100  | 2,248.59   | 13,129 MB  | 12.68  | 166 MB|
| Michalski Trains        | 15, 6  | 1    | 0.04       | 7 MB       | 0.00   | 4 MB  |
| Satellite               | 43, 5  | 20   | –          | ME         | 89.83  | 51 MB |
| Sieve of Erathostenes   | 16, 3  | 90   | 9.60       | 39 MB      | 0.43   | 13 MB |
| Spanner                 | 14, 5  | 100  | 326.79     | 1,512 MB   | 5.10   | 61 MB |

---

**Table 6**

Validation of complex domains, CPU time (secs) and memory peak for program validation, with/without infinite program detection. TE and ME stands for time (1 h) and memory exceeded, respectively. Best results in bold.

---

CRediT authorship contribution statement

**Javier Segovia-Aguas**: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Sergio Jiménez**: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Software, Supervision, Validation, Writing – original draft, Writing – revision.

---

*We have already explored this general scope of our GP approach to synthesize complex action models from examples [101].*
Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A link to the source code is included in the article.

Acknowledgements

This work has been co-funded by MCIN/AEI /10.13039/501100011033 under the Maria de Maeztu Units of Excellence Programme (CEX2020-001195-M), TAILOR (H2020 #952215) and AIPLAN4EU (H2020 #101016442) projects. Javier Segovia-Aguas is also supported by AGAUR SGR and the Spanish grant PID2019-108141 GB-I00. Sergio Jiménez is supported by the Spanish MINECO project PID2021-127647NB-C22. Anders Jonsson is partially supported by Spanish grant PID2019-108141 GB-I00.

References

J. Segovia-Aguas, S. Jiménez and A. Jonsson

Artificial Intelligence 330 (2024) 104097