Representation and Synthesis of C++ Programs for Generalized Planning

Javier Segovia-Aguas[1] , **Yolanda E-Martín**[2] and **Sergio Jiménez**[3]
[1]Universitat Pompeu Fabra
[2]Florida Universitària
[3]VRAIN - Valencian Research Institute for Artificial Intelligence, Universitat Politècnica de València
javier.segovia@upf.edu, yescudero@florida-uni.es, serjice@dsic.upv.es

Abstract

The paper introduces a novel representation for *Generalized Planning* (GP) problems, and their solutions, as C++ programs. Our C++ representation allows to formally proving the termination of generalized plans, and to specifying their *asymptotic complexity* w.r.t. the number of world objects. Characterizing the complexity of C++ generalized plans enables the application of a combinatorial search that enumerates the space of possible GP solutions in order of complexity. Experimental results show that our implementation of this approach, which we call BFGP++, outperforms the previous *GP as heuristic search* approach for the computation of generalized plans represented as compiler-styled programs. Last but not least, the execution of a C++ program on a classical planning instance is a deterministic grounding-free and search-free process, so our C++ representation allows us to automatically validate the computed solutions on large test instances of thousands of objects, where off-the-shelf classical planners get stuck either in the pre-processing or in the search.

1 Introduction

Automated planning has not achieved the level of integration with common programming languages, like C, JAVA, or PYTHON, that is achieved by other forms of problem solving such as *constraint satisfaction* or *operational research* [Schulte *et al.*, 2010; Prud'homme *et al.*, 2014; Perron and Furnon, 2019]. An important reason is the low-level representations traditionally handled in planning [Geffner, 2003; Rintanen, 2015]. Since the early 70's, STRIPS is the most popular representation language for research in automated planning [Fikes and Nilsson, 1971]. Even today, STRIPS is an essential fragment of PDDL [Haslum *et al.*, 2019], the input language of the *International Planning Competition*, and most planners support the STRIPS features. In spite of its popularity, the STRIPS representation is too low-level for many interesting applications [Smith *et al.*, 2008; Rintanen, 2015]; STRIPS limits representation to Boolean state variables, and Boolean constraints, and focuses computation on plans represented as sequences of ground actions.

Recent advances in *planning as heuristic search* [Francès, 2017; Say *et al.*, 2017; Scala *et al.*, 2020], and in *planning as satisfiability* [Bryce *et al.*, 2015; Scala *et al.*, 2016; Rintanen, 2017], show that handling more expressive problem representations does not necessarily increase planning complexity. On the other hand, advances in *generalized planning* (GP) are producing effective algorithmic solution representations for (possibly infinite) sets of planning instances that share common structure [Schmid and Wysotzki, 2000; Winner and Veloso, 2003; Hu and Levesque, 2011; Srivastava *et al.*, 2011a; Srivastava *et al.*, 2011b; Hu and De Giacomo, 2011; Schmid and Kitzelmann, 2011; Belle and Levesque, 2016; Illanes and McIlraith, 2019; Jiménez *et al.*, 2019; Segovia-Aguas *et al.*, 2019; Francès *et al.*, 2021].

This paper introduces a novel C++ representation for GP. The contribution of the paper is three-fold:

1. *Proving termination and characterizing complexity*. Our representation of GP solutions allows to formally proving the termination of generalized plans represented as C++ programs. In addition, our C++ representation reveals the *asymptotic complexity* of generalized plans w.r.t the number of world objects. This is a relevant topic beyond GP, since it allows defining formal upper-bounds on the complexity of the (possibly infinite) set of instances of a classical planning domain.
2. *Improving the GP as heuristic search approach*. By definition, any generalized plan built by BFGP++ is terminating. BFGP++ skips the costly check of infinite executions for candidate solutions and hence, it outperforms the previous *GP as heuristic search* approach [Segovia-Aguas *et al.*, 2021; Segovia-Aguas *et al.*, 2022a].
3. *Validation of GP solutions at large instances*. The generalized plans produced by BFGP++ are compilable with standard programming tools, such as GCC g++, and efficiently validated in large instances (with thousands of objects), where off-the-shelf planners get stuck either in the pre-processing or in the search.

2 Preliminaries

2.1 Classical Planning

Following the formalization by Bonet and Geffner 2021, we define a *classical planning problem* as a pair $P = \langle \mathcal{D}, \mathcal{I} \rangle$,

where $\mathcal{D}$ is a first-order *planning domain* and $\mathcal{I}$ is the information of the classical planning *instance*. The *domain* contains the set of predicate symbols Ψ and the action schemes with preconditions and effects, given by atoms $p(x_1, \ldots, x_k)$, or their negations, where $p \in \Psi$ is a predicate symbol and each x_i is a variable symbol representing an argument of the action scheme. A classical planning *instance* is a tuple $\mathcal{I} = \langle \Omega, I, G \rangle$ where Ω is the finite set of world objects. Last, I and G respectively are the *initial* and *goal* configurations of the world objects, and they are defined using ground atoms $p(o_1, \ldots, o_k)$, or their negation.

The *set of states*, $S(P)$, associated with a classical planning problem P, are the possible sets of ground atoms. The initial state is $s_0 = I$, and the subset of goal states $S_G \subseteq S(P)$, contains all the states $s_g \in S(P)$ s.t. $G \subseteq s_g$. The *state graph* associated with a classical planning problem P has as nodes the states $S(P)$. Edges of this graph are defined as follows: for each pair of states $s \in S(P)$ and $s' \in S(P)$, the graph has a directed edge (s, s') iff there is a ground action a that is applicable in s (i.e. whose preconditions hold in s) and whose effects transform the state s into $s' = f(s, a)$.

A *solution* to a classical planning problem P is a sequential plan $\pi = \langle a_1, \ldots, a_m \rangle$ such that $s_0 = I$, the ground actions a_i are applicable in states s_{i-1}, they produce successors $s_i = f(s_{i-1}, a_i)$, and the goal condition holds in the last reached state, i.e. $G \subseteq s_m$.

2.2 Generalized Planning

This work builds on top of the inductive formalism for GP, where a GP problem is a set of classical planning instances that belong to the same domain $\mathcal{D}$. In other words, they are all represented with the same predicates and actions schemes, but they may differ in the number of objects, and the initial/-goal configuration of these objects.

Definition 1 (GP problem)**.** *A* GP problem *is a non-empty set $\mathcal{P} = \{P_1, \ldots, P_T\}$ of T classical planning instances from a given domain $\mathcal{D}$.*

The aim of GP is to compute algorithmic planning solutions, a.k.a. generalized plans, that work for the given set of planning problems. In this paper we focus on the computation of GP solutions represented as C++ programs.

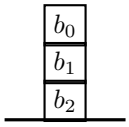
Definition 2 (GP solution)**.** *A* generalized plan Π *solves a GP problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ iff, for every classical planning instance $P_t \in \mathcal{P}$, $1 \leq t \leq T$, the execution of Π on P_t, denoted as $exec(\Pi, P_t) = \langle a_1, \ldots, a_m \rangle$, induces a classical plan that solves P_t.*

3 A C++ Representation for Planning

We start explaining our C++ representation for propositional classical planning and then we show that it naturally extends to numeric planning and to GP. Our novel representation can actually be implemented with any structured programming language that supports `If` conditionals and `For` loops, as well as `Vectors` (to store arrays that can change in size) and `Map` containers (to store key-value pairs with unique keys); the paper exemplifies our representation with the *C++* language, but other common programming languages, such as *Python* or *Java*, could also be used.

```
map<vector<int>, bool> pred_clear;
map<vector<int>, bool> pred_handempty;
map<vector<int>, bool> pred_holding;
map<vector<int>, bool> pred_on;
map<vector<int>, bool> pred_ontable;
```

Figure 1: C++ declaration of the *blocksworld* first-order predicates.




```
pred_clear[{0}]=1;
pred_handempty[{}]=1;
pred_on[{0,1}]=1; pred_on[{1,2}]=1;
pred_ontable[{2}]=1;
```

Figure 2: Example of a three-block state from *blocksworld* (left), and its corresponding C++ representation (right).

3.1 Classical planning problems as C++ programs

We exemplify our C++ representation in the classic *blocksworld* [Slaney and Thiébaux, 2001], that consists of a set of blocks, a table, and a robot hand. The domain defines five first-order predicates, namely *clear(?x)*, *handempty()*, *holding(?x)*, *on(?x,?y)*, and *ontable(?x)*. A block that has nothing on it is clear, the robot hand can be empty or holding one block, and a block can be on top of another block or on the table. The domain defines four action schemes, *stack(?x,?y)*, *unstack(?x,?y)*, *pick-up(?x)* and *put-down(?x)* for stacking (or unstacking) a block on top of another block and putting down (or picking up) a block onto the table.

States. Each first-order predicate $p(x_1, \ldots, x_k) \in \Psi$ is represented as a C++ *map container* s.t. the key for indexing the map is a k-integer vector (where k is the predicate arity). The map stores a Boolean for each of the corresponding ground predicates $p(o_1, \ldots, o_k)$ holding in the current state; we follow the closed-world assumption so our C++ state representation stores a maximum of $\sum_{k \geq 0} n_k |\Omega|^k$ Boolean, where n_k is the number of first-order predicates with arity k, and $|\Omega|$ is the number of world objects[1]. Figure 1 shows the C++ declaration of the blocksword predicates, while Figure 2 shows our C++ representation of a *blocksworld* state, where there are three blocks that are stacked in a single tower.

Actions. Each action scheme is represented with a C++ Boolean function. The arguments of the function are those of the action scheme, but in our C++ representation they act as *indexes* to address the maps that are encoding the state. Formally, an *index* $z \in Z$ is a finite domain variable ranging the number of objects i.e., $D_z = [0, |\Omega|)$. We inductively define our C++ representation of an action scheme with the following grammar:

[1] $\sum_{k \geq 0} n_k |\Omega|^k$ is also the number of propositions that result from grounding a STRIPS classical planning instance.

$$
\begin{aligned}
Action &:= if(Condition_z(s))\{Effect(s)\}\ return\ false;\\
Condition_z(s) &:= (p(z_1,\ldots,z_k) == 0)\&\&\ Condition_z(s) \mid\\
&\quad (p(z_1,\ldots,z_k) == 1)\&\&\ Condition_z(s) \mid\\
&\quad true\\
Effect(s) &:= (p(z_1,\ldots,z_k) = 0);\ Effect(s) \mid\\
&\quad (p(z_1,\ldots,z_k) = 1);\ Effect(s) \mid\\
&\quad return\ true;
\end{aligned}
$$

where $Condition_z(s)$ is a conjunction of assertions over predicates $p(z_1,\ldots,z_k)$ instantiated with the action arguments, $==$ denotes the equality operator, $\&\&$ is the logical AND operator, $=$ indicates a value assignment, and ; denotes the end of an instruction. Likewise *Effect(s)* is a conjunction of assignments representing the positive/negative effects of an action scheme; $(p(z_1,\ldots,z_k) = 1)$ denotes a *positive* effect while $(p(z_1,\ldots,z_k) = 0)$ denotes a *negative* effect.

Figure 4 shows our C++ representation of the `unstack` action scheme from the blocksworld (Figure 3), that compactly represents a set of state transitions, and that applies to any *blocksword* instance, no matter the number of blocks. Note that we represent actions as if they were always applicable, but action effects only update the state iff the action preconditions hold in the current state. This action modeling, common in RL, facilitates the specification of compact algorithm-like solutions, and it preserves the original branching factor (successor states equal to their parents are ignored).

```
(:action unstack
  :parameters (?x ?y)
  :precondition (and (on ?x ?y)(clear ?x)(handempty))
  :effect (and (holding ?x) (clear ?y)
               (not (clear ?x)) (not (handempty))
               (not (on ?x ?y)))))
```

Figure 3: PDDL representation of the `unstack` action scheme.

```
bool act_unstack(int x, int y){
  if(pred_on[{x,y}]==1  && pred_clear[{x}]==1
&& pred_handempty[{}]==1){
    pred_holding[{x}] = 1; pred_clear[{y}] = 1;
    pred_clear[{x}] = 0; pred_handempty[{}] = 0;
    pred_on[{x,y}] = 0;
    return true;
  }
  return false;
}
```

Figure 4: Our C++ representation of the `unstack` action scheme.

Problems. Our C++ representation of a propositional planning problem is completed with the functions for representing the *initial state* and the *goals*. These functions are formalized as follows:

$$
\begin{aligned}
Init &:= (p(o_1,\ldots,o_k) = 1);\ Init \mid\\
&\quad ;\\
Goals &:= return(Condition_o(s));\\
Condition_o(s) &:= (p(o_1,\ldots,o_k) == 0)\&\&\ Condition_o(s) \mid\\
&\quad (p(o_1,\ldots,o_k) == 1)\&\&\ Condition_o(s) \mid\\
&\quad true
\end{aligned}
$$

The *init function* is a write-only *void function* that initializes the C++ maps with the assignments for representing the ground atoms $p(o_1,\ldots,o_k)$, that hold in the initial state. The *goal function* is a read-only *Boolean function* that encodes, as a partial state, the subset of goal states. Figure 5 shows the *init* and *goal* C++ functions for representing the problem of unstacking the three-block tower of Figure 2.

```
void init() {
 pred_clear[{0}]=1;
 pred_handempty[{}]=1;
 pred_on[{0,1}]=1; pred_on[{1,2}]=1;
 pred_ontable[{2}]=1;
}

bool goals() {
  return ((pred_ontable[{0}]==1) &&
          (pred_ontable[{1}]==1) &&
          (pred_ontable[{2}]==1));
}
```

Figure 5: The *init* and *goal* C++ functions representing the planning problem of unstacking the three-block tower of Figure 2.

3.2 Sequential plans as C++ programs

Our C++ representation of a sequential plan π uses programming instructions for: (i), invoking the *C++ Boolean function* that encodes an action scheme and (ii), incrementing/decrementing the value of an index. Formally:

$$
\begin{aligned}
\pi &:= Statement(s)\\
Statement(s) &:= a(z_1,\ldots,z_k); Statement(s) \mid\\
&\quad z++; Statement(s) \mid\\
&\quad z--; Statement(s) \mid\\
&\quad ;
\end{aligned}
$$

where $a(z_1,\ldots,z_k)$ is an action scheme instantiated with a subset of indexes $\{z_1,\ldots,z_k\} \subseteq Z$, and $\{$`z++`, `z--` $\mid z \in Z\}$ are the instructions to increment/decrement an index $z \in Z$. Indexes in Z are always initialized to zero. Figure 6 illustrates the relation between an action scheme (i), its corresponding action instantiated over indexes in Z and (ii), the corresponding ground actions instantiated over the objects in Ω.

Figure 7 illustrates our C++ representation of the four-action sequential plan $\pi = \langle$`unstack(b0,b1)`, `putdown(b0)`, `unstack(b1,b2)`, `putdown(b1)`$\rangle$ for unstacking the three-block tower of Figure 2. The `void ONTABLE-SEQUENTIAL()` program leverages two indexes, $Z = \{z_1, z_2\}$, that are initialized to zero so they initially point to the first object (block b_0 in this case). After executing the first z_2`++` instruction, z_2 points to the second block, b_1, while z_1 still points to block b_0. This means that the first

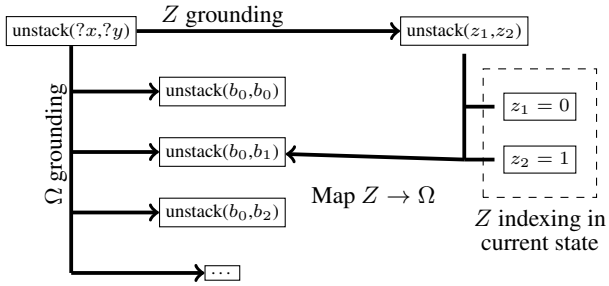


Figure 6: Relation between the action scheme $unstack(?x, ?y)$ (i), the action $unstack(z_1, z_2)$ instantiated with indexes (z_1, z_2), and (ii), the ground actions instantiated with the set of three blocks $\Omega = \{b_0, b_1, b_2\}$. Indexes z_1 and z_2 are bound variables in $[0, \ldots, |\Omega|)$ and currently, they are indexing blocks b_0 and b_1, respectively.

`act_unstack(z1, z2)` instruction of the program in Figure 7 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 z_1 and z_2 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.

```
void ONTABLE-SEQUENTIAL (){
    int z1=0, z2=0;
    z2++;
    act_unstack(z1,z2);
    act_putdown(z1);
    z1++;
    z2++;
    act_unstack(z1,z2);
    act_putdown(z1);
}
```

Figure 7: C++ representation of a *sequential plan* for unstacking and putting onto the table the three-block tower of Figure 2.

Theorem 1. *Provided a number of indexes $|Z|$ as large as the largest arity of an action, our C++ representation for sequential plans preserves the original solution space.*

Proof. Any sequential plan π can be rewritten as an equivalent C++ procedure that first initializes indexes in Z to zero and then, for each ground action $a \in \pi$, it repeatedly applies `z++/z--` instructions until the indexes address the objects of the corresponding ground action a. □

3.3 Beyond propositional planning

Our C++ representation naturally supports planning with numeric state variables, that can participate into the representation of a classical planning problem as defined in PDDL2.1 [Fox and Long, 2003]. To support the representation of numeric state variables, our *C++ maps* store *integers* instead of Boolean. Likewise goals and action preconditions can include assertions over the numeric state variables, and action effects can include assignments of the numeric state variables. For instance, the set of numeric state variables that indicate the physical distance between two blocks is declared in our C++ fragment as `map< vector<int>, int> distance`; with this regard, $distance[\{0, 1\}] = 7$ indicates that the distance between blocks b_0 and b_1 is of seven units, and $distance[\{z1, z2\}] > distance[\{z2, z3\}]$ indicates that the distance between the blocks pointed by indexes $z1$ and $z2$ is larger than the distance between the blocks pointed by $z2$ and $z3$. Object *typing* is also naturally supported by our C++ representation specializing map indexes to the number of objects of a particular type.

4 Generalized Planning with C++

This section details our approach for computing generalized plans represented as C++ programs.

4.1 Representing generalized plans with C++

Given that a GP problem is just a set of classical planning problems from a given domain, and that the C++ representation for classical planning is detailed above, we directly define here our C++ representation of generalized plans. Our C++ representation of a generalized plan Π extends our representation of a sequential plan π with two control-flow constructs (`If` conditionals and `For` loops):

$$
\begin{aligned}
\Pi &:= ExtdStmnt(s) \\
ExtdStmnt(s) &:= If; ExtdStmnt(s) \mid For; ExtdStmnt(s) \mid \\
&\quad Statement(s); ExtdStmnt(s) \mid; \\
If &:= if(Condition)\{ExtdStmnt(s)\} \\
Condition &:= (p(z_1, \ldots, z_k) == 0) \mid (p(z_1, \ldots, z_k) \neq 0) \mid \\
&\quad (z_1 > z_2) \mid (z_1 == z_2) \mid (z_1 < z_2) \mid \\
&\quad (p(z_1, \ldots, z_k) > p(z'_1, \ldots, z'_k)) \mid \\
&\quad (p(z_1, \ldots, z_k) == p(z'_1, \ldots, z'_k)) \mid \\
&\quad (p(z_1, \ldots, z_k) < p(z'_1, \ldots, z'_k)) \mid \\
For &:= for(z = 0; z < |\Omega|; z++)\{ExtdStmnt(s)\}| \\
&\quad for(z = |\Omega| - 1; z \geq 0; z--)\{ExtdStmnt(s)\}
\end{aligned}
$$

where `ExtdStmnt(s)` extends `Statement(s)`, as defined for sequential plans, with `If` conditional and `For` loop instructions. The `Condition` of an `If` instruction is restricted to: (i), checking whether a $p(z_1, \ldots, z_k)$ predicate instantiated with indexes in Z equals to zero, (ii), the three *three-way comparison* [Browning and Sutherland, 2020] of two different indexes in Z and (iii), the three-way comparison of two predicates instantiated with indexes in Z (or two numeric fluents in the case of a numeric domain). Last, we restrict `For` loops to exclusively iterate over the domain $[0, |\Omega|)$ of an index $z \in Z$. Like in our C++ representation for sequential plans, in a C++ generalized plan the set of indexes Z are always initialized to zero.

Figure 8 shows an example of a generalized plan, represented as a C++ program and computed by BFGP++, for unstacking any number of towers from the blocksworld, no matter the actual number of blocks $|\Omega|$. Please note that object ordering affects to the sequential plan produced by the execution of the generalized plan but it does not affect the

```
void ONTABLE (){
 int z1=0, z2=0, z3=0;
 for(z1=0; z1<|Ω|; z1++){
  for(z2=0; z2<|Ω|; z2++){
   for(z3=0; z3<|Ω|; z3++){
    act_put_down(z2);
    act_unstack(z2,z3);
   }}}}
```

Figure 8: Generalized plan, represented as a C++ program and computed by BFGP++, for unstacking any number of towers from the blocksworld, no matter the number of blocks $|\Omega|$.

correctness/completeness of the generalized plan[2]. As a matter of fact, the program of Figure 8 leverages three indexes $Z = \{z_1, z_2, z_3\}$ to be robust to any block ordering.

From C++ programs to sequential plans

Given a C++ generalized plan Π, and a classical planning problem P, the sequential plan $exec(\Pi, P)$ is built executing Π on P. This is a deterministic search-free procedure where no instantiation is required (there are no *free variables*). The program execution may however produce actions whose execution does not modify the planning state, i.e. the current state does not meet the precondition of those actions, so action effects are not applied. These actions are automatically discarded since identifying them is straightforward; the execution of the corresponding C++ function $a(z_1, \ldots, z_k)$ returns `false`.

We illustrate the building of a sequential plan from a C++ generalized plan with the execution of the program of Figure 8 on the initial state of Figure 2. This execution produces the following sequence of ground actions repeated thrice: $3 \times \langle$putdown(b0), unstack(b0,b0), putdown(b0), **unstack(b0,b1)**, **putdown(b0)**, unstack(b0,b2), **putdown(b1)**, unstack(b1,b0), putdown(b1), unstack(b1,b1), putdown(b1), **unstack(b1,b2)**, putdown(b2), unstack(b2,b0), putdown(b2), unstack(b2,b1), putdown(b2), unstack(b2,b2)$\rangle$. Only the four actions in bold update the state, i.e. they are applicable when they are executed; in the first repetition unstack(b0,b1), putdown(b0) and unstack(b1,b2), are applicable; while putdown(b1), becomes only applicable in the second out of the three repetitions. The sequence of these four ground actions actually is the sequential plan for unstacking the three-block tower of Figure 2.

Termination and complexity of generalized plans

Enabling C++ programs with `For` loops introduces a new possible source of failure of a program execution in a given classical planning instance; the program execution could enter into an *infinite loop*. To formally guarantee the termination of our C++ generalized plans, we restrict ourselves to C++ programs s.t. *ExtdStmnt(s)* in the body of a `For` loop do not include instructions that modify the index iterated by that loop [Hoare, 1969].

Theorem 2. *A generalized plan Π, represented in our C++ fragment is always terminating.*

[2]This also occurs with regular programs e.g. a *SelectionSort* program is sound and complete but the number of `swap` instructions executed by the program depends on the input list to be sorted.

Proof. The grammar used for the production of a C++ generalized plan Π comprises sequential statements, `If` conditionals, and `For` loops that iterate over the finite domain $[0, |\Omega|)$ of an index. Since these rules guarantee the program is well structured, and given that sequential statements and `If` conditionals can only advance the program execution, then `For` loops would be the only possible cause of non-termination. Since we do not allow to programming instructions inside a `For` loop that modify the index iterated by the loop, we have that (nested) `For` loops always modify indexes in a unique direction, which is terminating because the domain of an index is finite by definition. □

Besides providing a formal termination proof for C++ generalized plans, restricting loops to iterations over the domain $[0, |\Omega|)$ of an index also reveals the *asymptotic complexity* of generalized plans w.r.t. the number of objects. We leverage the *big-O notation* to formulate an upper-bound on the complexity of generalized plans and, as a consequence, on the difficulty of the problems solved by those plans [Skiena, 1998]. If there is a C++ generalized plan Π that solves a given classical planning problem P, it means that the length of the sequential plan produced by the execution of Π on P is upper-bounded by a polynomial of the number of objects $|\Omega|$; the degree of that polynomial is given by the maximum number of nested *for loops* in the C++ program. As $|\Omega|$ grows larger this term will come to dominate, so that all other terms, and the coefficients, can be ignored.

Definition 3 (Asymptotic complexity of a generalized plan)**.** *The asymptotic complexity of a generalized plan represented by a C++ program is defined as the joint product of the ranges of the indexes iterated by the largest set of nested loops.*

For instance the generalized plan of Figure 8, with three nested loops, and where each loop range is $[0, |\Omega|)$, has asymptotic complexity $O(|\Omega|^3)$. This is a worst-case upper-bound; the program of Figure 8 exhibits worst-case complexity $O(|\Omega|^3)$ but we showed that the produced sequential plan for the three-block instance of Figure 2 contained four actions.

4.2 Synthesis of C++ programs as heuristic search

BFGP++ is our approach to the synthesis of generalized plans, represented as C++ programs. BFGP++ implements a Best-First Search (BFS) in the space of possible programs that can be built with the grammar of Section 4.1. Since this search space is unbound, we bound it with two input parameters: a maximum number program lines n, and a maximum number of *indexes* $|Z|$ that can be used by a program. Next, we provide more details on the implementation of BFGP++.

The search space

Each node of the BFGP++ search space corresponds to a *partially specified program*. By partially specified program we mean that some of its n program lines may be undefined, because they are not programmed yet. Starting from the *empty program* (i.e. the partially specified program whose n lines are all undefined), we enumerate the space of possible programs with a search operator that programs a possible

programming instruction (according to the grammar of Section 4.1), at an undefined program line $0 \leq i < n$. This search operator is only applicable when program line i is undefined. Initially $i := 0$, and after line i is programmed $i := i + 1$. For instance, we can build the generalized plan of Figure 8 with the index set $Z = \{z_1, z_2, z_3\}$, starting from the empty program, and following the successive application of the following six grammar rules (indexes in Z are always initialized to zero):

1. $for(z1 = 0; z1 < |\Omega|; z1{+}{+})\{ExtdStmnt(s)\}$
2. $for(z2 = 0; z2 < |\Omega|; z2{+}{+})\{ExtdStmnt(s)\}$
3. $for(z3 = 0; z3 < |\Omega|; z3{+}{+})\{ExtdStmnt(s)\}$
4. $act\_putdown(z2); ExtdStmnt(s)$
5. $act\_unstack(z2, z3); ExtdStmnt(s)$
6. ;

Pruning rules

To keep the search space tractable, BFGP++ implements the following pruning rules that reduce the search space but preserve the solution space:

- We do not allow programming a *conditional if* or a *for loop* at the last program line; it is meaningless to program control-flow structures with empty body.
- We do not allow programming *for loops* that iterate over an object type that contains, at most, a single object for all the instances of the GP problem.

In addition, we implement a simple but effective symmetry breaking mechanism to safely prune the programming of *for loops* that correspond to *symmetries* (permutations of the order of the *for loops*) of already expanded programs. We use a *tabu list*, that is initially empty and, every time a node is expanded, we store an abstraction of its programmed *for loops*; the remaining program instructions, and the precise identity of the indexes iterated by the *for loops* are ignored. For instance, the generalized plan of Figure 8 is abstracted by {(block,++,1,8), (block,++,2,7), (block,++,3,6)}, since this program contains three *for loops* and where, `block` indicates the index type, ++ indicates the kind of the *for loop* (++increasing/−−decreasing), and each pair of numbers indicates the first and last program lines of each *for loop*.

The evaluation functions

BFGP++ implements a *Best-First Search* (BFS) in the previous solution space. To reduce memory requirements, we store only the open list of generated nodes but not the closed list of expanded nodes [Korf *et al.*, 2005]. We consider the following evaluation functions for sorting the open list:

- $f_{euclidean}(\Pi, \mathcal{P}) = \sum_{P_t \in \mathcal{P}} \sum_{v \in G_t} (s_t[v] - G_t[v])^2$. This function accumulates, for each classical planning problem $P_t \in \mathcal{P}$ in a GP problem, the *euclidean distance* of state s_t to the goal state variables G_t. The state s_t is obtained applying the sequence of actions $exec(\Pi, P_t)$ to the initial state I_t of that problem $P_t \in \mathcal{P}$.
- $f_{min(\#loops)}(\Pi)$ is the number of loop instructions in Π.

Both $f_{euclidean}$ and $f_{min(\#loops)}$ were proposed in Segovia-Aguas *et al.* (2021), named h_5 and f_1 respectively. In more detail, the configuration with the best reported performance was $(f_{euclidean}, f_{min(\#loops)})$, i.e. sorting the open list with $f_{euclidean}$, and breaking ties with the $f_{min(\#loops)}$ function. In this paper we introduce a third alternative function $f_{max(\#loops)}(\Pi) = -f_{min(\#loops)}(\Pi)$, which aims maximizing the number of loops in a program. We discovered that prioritizing by $f_{euclidean}$ and breaking ties with our new function $f_{max(\#loops)}$, instead of $f_{min(\#loops)}$, performs much better at a wide range of challenging GP problems. Section 5 provides details on the obtained results.

Properties of BFGP++

Our BFGP++ algorithm for the synthesis of C++ generalized plans is terminating; termination follows from a terminating searching algorithm and evaluation functions. Regarding the former, BFGP++ implements a *frontier BFS* which is known to be terminating at finite search spaces [Korf *et al.*, 2005], we recall that the search space of BFGP++ is finite since n and $|Z|$ are bounded. Regarding the evaluation functions, $f_{min(\#loops)}$ and $f_{max(\#loops)}$, they terminate in n steps, where n is the number of lines of the program to evaluate, and $f_{euclidean}$ terminates iff the program executions terminate, which immediately follows from Theorem 2. Further, BFGP++ is sound, since it only outputs a program when it is able to solve all the planning instances given as input in the GP problem (Definition 2). Last, BFGP++ is complete provided that there exists a GP solution within the given number of program lines n and indexes $|Z|$.

5 Evaluation

Next we report results on the synthesis (and validation) of C++ generalized plans with BFGP++, and compared them w.r.t. the *GP as heuristic search* approach [Segovia-Aguas *et al.*, 2021; Segovia-Aguas *et al.*, 2022a], the state-of-the-art for the computation of generalized plans represented as programs. All experiments are performed in an Ubuntu 20.04 LTS, with AMD® Ryzen 7 3700x 8-core processor × 16 and 32GB of RAM, with a 1 hour time bound. We also report the computed solutions, and their asymptotic complexity w.r.t the number of world objects.

We address the full set of domains from Segovia-Aguas *et al.* (2021), all numeric domains, and we also address several STRIPS domains (marked with a *). Next we briefly describe the domains: **Blocks* (ontable)**, put all blocks onto the table from any number of towers and blocks setting. **Corridor***, move from any initial location in a corridor to any other arbitrary location. **Fibonacci**, given the first numbers of the Fibonacci sequence, compute the n^{th} value. **Find**, count the number of occurrences of a specific value in a vector. **Floyd***, given an input graph, compute a path that connects two distant nodes. **Gripper***, move all balls from room A to the adjacent room B. **Intrusion***, steal data from a set of hosts. **Reverse**, reverse the content of a given vector. **Select**, search for the minimum value in a vector. **Sorting**, sort in increasing order all the values in a vector. **Spanner***, collect the spanners in a directed corridor to tighten all loose nuts. **Triangular Sum**, given the first numbers of the triangular sum, compute the n^{th} value. **Visitall***, starting at the bottom left corner of a squared grid visit all locations in the grid.

	$n, \|Z\|$	GP as heuristic search (2021)				BFGP++			
		Time	Mem.	Exp.	Eval.	Time	Mem.	Exp.	Eval.
Blocks (ontable)	9, 3	TE	TE	TE	TE	**0.08**	**5**	**9**	**347**
Corridor	11, 2	TE	TE	TE	TE	**701.90**	**27**	**661.4K**	**2.5M**
Fibonacci	7, 2	32.05	83	242.1K	1.7M	**1.61**	**5**	**2.4K**	**19.1K**
Find	6, 3	0.98	7	13.9K	38.4K	**0.14**	**5**	**1.3K**	**3.9K**
Floyd	8, 3	TE	TE	TE	TE	**0.22**	**5**	**4**	**138**
Gripper	8, 4	TE	TE	TE	TE	**105.00**	**206**	**83.2K**	**1.0M**
Intrusion	9, 1	TE	TE	TE	TE	**521.24**	**874**	**411.8K**	**3.4M**
Reverse	7, 2	9.75	31	99.8K	344.7K	**0.23**	**4**	**626**	**2.8K**
Select	7, 2	9.52	32	96.3K	331.6K	**0.28**	**4**	**737**	**4.4K**
Sorting	8, 2	129.72	335	1.2M	4.1M	**0.02**	**4**	**52**	**245**
Spanner	12, 5	TE	TE	TE	TE	**0.91**	**5**	**14**	**367**
Triangular Sum	5, 2	0.15	5	1.4K	9.9K	**0.01**	**4**	**9**	**96**
Visitall	15, 4	TE	TE	TE	TE	**1.67**	**6**	**117**	**2.7K**

Table 1: Number n of program lines and $|Z|$ indexes. For each synthesis configuration we report CPU time (secs), memory peak (MBs), and number of expanded and evaluated nodes. Best results in bold. TE stands for Time-Exceeded (>1h of CPU).

Synthesis of C++ generalized plans

For the synthesis experiments, we use ten random instances of increasing difficulty per domain and compare BFGP++, that leverages the function combination $(f_{euclidean}, f_{max(\#loops)})$, with the best configuration of the *GP as heuristic search* approach [Segovia-Aguas *et al.*, 2021; Segovia-Aguas *et al.*, 2022a], that leverages the evaluation functions $(f_{euclidean}, f_{min(\#loops)})$ and serves as a baseline.

Table 1 summarizes the obtained results, when computing the best solutions found in terms of number of required program lines and pointers, and it shows that BFGP++ outperforms the baseline over all the domains. One of the main performance issues in the synthesis of planning programs is to guarantee that a given (partially specified) program is terminating [Segovia-Aguas *et al.*, 2020]; in Segovia-Aguas *et al.* (2021), search nodes corresponding to infinite programs were checked and discarded in execution time, which was costly and had no guarantees beyond the given set of input instances. Alternatively, the candidate C++ generalized plans considered by BFGP++ are, by definition, terminating for any input instance. Therefore BFGP++ skips the costly check of infinite programs.

Note that the baseline fails to solve the STRIPS domains within the given computation bounds. In STRIPS domains, $f_{euclidean}$ is a simple *goal counting* heuristic. The function combination of BFGP++ $f_{euclidean}, f_{max(\#loops)}$ breaks ties prioritizing C++ programs of higher asymptotic complexity. This means that, if a C++ generalized plan does not exist within $n-1$ lines, but it does exist for n lines, it will probably contain as many nested loops as possible. This is not however a *rule of thumb*, as shown by the *Corridor* domain, which is the domain that took the largest time to be solved; requiring two consecutive (not nested) `For` loops, which are hard to identify by our current evaluation functions.

Validation of C++ generalized plans

The generalized plans synthesized by BFGP++ are compilable with the *GCC g++* compiler, and they are all successfully validated on twenty large random instances of increasing size. Table 2 reports the size of the largest instance, and the average and total validation times (including compilation time). This experiment shows that representing generalized plans as C++ programs is a scalable approach to deal with large classical planning instances, of thousands of objects, where off-the-shelf classical planners get stuck even in the pre-processing.

Next we report the computed programs and their asymptotic complexity. Please note that the reported validation times correlate to the revealed program complexity.

Floyd. Indexes increase up to $N = |\Omega|$, which denotes the number of graph nodes, of the input instances. The asymptotic complexity of the generalized plan is $O(N^3)$.

```
void FLOYD(){
 int z1=0, z2=0, z3=0;
 for(z1=0; z1<N; z1++){
  for(z2=0; z2<N; z2++){
   for(z3=0; z3<N; z3++){
    act_connect(z1,z2,z3);
}}}}
```

Corridor. This generalized plan requires the composition of two *for loops*, the first one to move to the rightmost location, and a second one to move back until reaching the goal location. In the worst case it iterates twice over the set of locations N, and its asymptotic complexity is $O(N)$.

```
void CORRIDOR(){
 int l1 = 0, l2 = 0;
 for(l1=0; l1<N; l1++){
   act_move(l2,l1);
   l2 = l1;
 }
 for(l1=N-1; l1>=0; l1--){
   if(pred_goal_at[{l1}] == 0 ){
     if (l2>0) l2--;
   }
   act_move(l1,l2);
 }}
```

Fibonacci and Triangular Sum. Both are numeric domains, where an input vector is given with the first elements of the sequence to compute. They compute every a-th value iteratively until reaching the last N value. Their asymptotic complexity is $O(N)$.

```
void FIBONACCI(){
 int a=0, b=0;
 for(a=0; a<N; a++){
  act_vector_add(a,b);
  if (b>0) b--;
  act_vector_add(a,b);
  b = a;}}
```

```
void TRIANGULARSUM(){
  int a=0, b=0;
  for(a=0; a<N; a++){
   act_vector_add(a,b);
   b = a;
  }}
```

Find, Reverse, Select and Sorting. Four numeric domains for vector manipulation. N is the size of the input vector, the first three domains run in $O(N)$. *Sorting* has two nested loops so its complexity is $O(N^2)$.

```
void FIND(){
 int i=0, t=0, a=0;
 for(i=0; i<N; i++){
  if(p_v[{i}]==p_v[{t}]){
     act_accumulate(a);
  }}}

void SELECT(){
 int a=0, b=0;
 for(a=0; a<N; a++){
  if(p_v[{a}]<p_v[{b}]){
    b=a; }}
 act_select(b);}

void REVERSE(){
 int i=0, j=0;
 for(i=N-1; i>=0; i--){
  if(i>j){
   swap(p_v[{i}],p_v[{j}]);
  }
  if(j<N-1) j++; }}

void SORTING(){
 int i=0, j=0;
 for(i=0; i<N; i++){
  for(j=0; j<N; j++){
   if(p_v[{i}]<p_v[{j}]){
     swap(p_v[{i}],p_v[{j}]);}}}}
```

Gripper and Intrusion. These two domains have generalized plans where a sequence of planning actions is repeated over the finite set of objects: the number of balls NB for *Gripper*, and the number of hosts NH for *Intrusion*. Their complexity is $O(NB)$ and $O(NH)$, respectively.

```
void GRIPPER(){
 int r1=0, r2=0, b1=0, g1=0;
 for(b1=0; b1<NB; b1++){
   act_pick(b1,r1,g1);
   if (r2<NROOMS-1) r2++;
   act_move(r1,r2);
   act_drop(b1,r2,g1);
   act_move(r2,r1); }}

void INTRUSION(){
 int h1=0;
 for(h1=0; h1<NH; h1++){
   act_recon(h1);
   act_break_into(h1);
   act_clean(h1);
   act_gain_root(h1);
   act_download_files(h1);
   act_steal_data(h1);  }}
```

Spanner. The complexity of this generalized plan is $O(NLOC^2 \cdot NNUTS \cdot NSPAN)$, where $NLOC$ is the number of locations, $NNUTS$ is the number of nuts, and $NSPAN$ is the number of spanners.

```
void SPANNER(){
 int l1=0, l2=0, m1=0, n1=0, s1=0;
 for(l1=0; l1<NLOC; l1++){
  for(l2=0; l2<NLOC; l2++){
   for(n1=0; n1<NNUTS; n1++){
    for(s1=0; s1<NSPAN; s1++){
     act_pickup_spanner(l1,s1,m1);
     act_tighten_nut(l1,s1,m1,n1);
     act_walk(l1,l2,m1); }}}}
```

Visitall. The complexity of this generalized plan is $O(NROWS^2 \cdot NCOLS^2)$, where $NROWS$ is the number of grid rows and $NCOLS$ the number of grid columns.

```
void VISITALL(){
 int r1=0, r2=0, y1=0, y2=0;
 for(r1=0; r1<NROWS; r1++){
  for(r2=0; r2<NROWS; r2++){
   for(c1=0; c1<NCOLS; c1++){
    for(c2=NCOLS-1; c2>=0; c2--){
     act_right(c1,c2);
     act_visit(r1,c2);
     if(p_at_row[{r2}]==0){
      act_up(r1,r2);
      act_left(c2,c1); }}}}}}
```

6 Conclusions

We presented a novel C++ representation for GP problems, and their solutions. We also introduced BFGP++ that implements a heuristic search in the space of candidate C++ generalized plans, which naturally models STRIPS domains, and that outperforms the previous *GP as heuristic search* approach [Segovia-Aguas *et al.*, 2021; Segovia-Aguas *et al.*, 2022a]; since our C++ programs are terminating by definition, BFGP++ can skip the costly check of infinite programs. In addition we showed that our C++ generalized plans are compilable with standard programming tools (GCC g++) and that they are successfully validated in large instances, with several thousands of objects, where off-the-shelf classical planners get stuck in the pre-processing. Our *cost-to-go heuristic* is less informed than current heuristics for classical planning; our research agenda is to obtain better estimates building on top of modern heuristics for classical planning [Segovia-Aguas *et al.*, 2022b]

	Max. input	Avg. instance time (s)	Total time (s)
Blocks (ont.)	1,000 blocks	33.592±37.888	673.427
Corridor	5,001 locs.	0.010± 0.005	1.495
Fibonacci	44th num.	0.001±0.000	1.285
Find	5,001 elems.	0.004±0.002	1.352
Floyd	872 vertices	30.502±38.561	611.308
Gripper	5,001 balls	0.011± 0.005	1.702
Intrusion	1,001 hosts	0.003±0.001	1.568
Reverse	5,001 elems.	0.007±0.003	1.416
Select	5,001 elems.	0.010±0.004	1.485
Sorting	5,001 elems.	1.337±1.200	28.029
Spanner	212 spanners	20.609±26.363	413.798
T. Sum	5,001st num.	0.008±0.004	1.458
Visitall	100 × 100 grid	6.738±7.474	136.325

Table 2: Size of the largest instance, avg. instance time with standard deviation and total validation time including compilation time (both in seconds).

Our C++ representation for GP can be viewed as an instantiation of *F*-STRIPS [Geffner, 2000]; we show that the single level of indirection of indexes over objects is enough to represent GP problems, and their solutions, with constant memory access; in *F*-STRIPS function symbols can be recursively nested, so the actual complexity of state queries in *F*-STRIPS depends on the depth of this nesting. Prior work characterized the complexity of planning tasks following a related but different approach; analysing the behaviour of *state-space heuristics* for classical planning [Hoffmann, 2005; Helmert, 2008; Seipp *et al.*, 2016]. When generalized plans are represented as *generalized policies* their complexity has also been characterized w.r.t. the features in the policy rules [Bonet and Geffner, 2021].

References

[Belle and Levesque, 2016] Vaishak Belle and Hector J Levesque. Foundations for generalized planning in unbounded stochastic domains. In *KR*, 2016.

[Bonet and Geffner, 2021] Blai Bonet and Hector Geffner. General policies, representations, and planning width. In *AAAI*, 2021.

[Browning and Sutherland, 2020] J Burton Browning and Bruce Sutherland. Working with numbers. In *C++ 20 Recipes*, pages 115–145. Springer, 2020.

[Bryce *et al.*, 2015] Daniel Bryce, Sicun Gao, David J Musliner, and Robert P Goldman. Smt-based nonlinear pddl+ planning. In *AAAI*, 2015.

[Fikes and Nilsson, 1971] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 1971.

[Fox and Long, 2003] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *JAIR*, 20:61–124, 2003.

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

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

[Geffner, 2000] Héctor Geffner. Functional strips: a more flexible language for planning and problem solving. In *Logic-based artificial intelligence*. 2000.

[Geffner, 2003] HA Geffner. Pddl 2.1: Representation vs. computation. *JAIR*, 20:139–144, 2003.

[Haslum *et al.*, 2019] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187, 2019.

[Helmert, 2008] Malte Helmert. *Understanding planning tasks: domain complexity and heuristic decomposition*, volume 4929. Springer, 2008.

[Hoare, 1969] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoffmann, 2005] Jörg Hoffmann. Where'ignoring delete lists' works: Local search topology in planning benchmarks. *JAIR*, 24:685–758, 2005.

[Hu and De Giacomo, 2011] Yuxiao Hu and Giuseppe De Giacomo. Generalized planning: Synthesizing plans that work for multiple environments. In *IJCAI*, 2011.

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

[Illanes and McIlraith, 2019] León Illanes and Sheila A McIlraith. Generalized planning via abstraction: arbitrary numbers of objects. In *AAAI*, 2019.

[Jiménez *et al.*, 2019] Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. A review of generalized planning. *KER*, 34:e5, 2019.

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

[Perron and Furnon, 2019] Laurent Perron and Vincent Furnon. Or-tools, 2019.

[Prud'homme *et al.*, 2014] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. Choco documentation. *INRIA Rennes, LINA CNRS UMR*, 2014.

[Rintanen, 2015] Jussi Rintanen. Impact of modeling languages on the theory and practice in planning research. In *AAAI*, 2015.

[Rintanen, 2017] Jussi Rintanen. Temporal planning with clock-based smt encodings. In *IJCAI*, 2017.

[Say *et al.*, 2017] Buser Say, Ga Wu, Yu Qing Zhou, and Scott Sanner. Nonlinear hybrid planning with deep net learned transition models and mixed-integer linear programming. In *IJCAI*, 2017.

[Scala *et al.*, 2016] Enrico Scala, Miguel Ramirez, Patrik Haslum, Sylvie Thiebaux, et al. Numeric planning with disjunctive global constraints via smt. In *ICAPS*, 2016.

[Scala *et al.*, 2020] Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramirez. Subgoaling techniques for satisficing and optimal numeric planning. *JAIR*, 2020.

[Schmid and Kitzelmann, 2011] Ute Schmid and Emanuel Kitzelmann. Inductive rule learning on the knowledge level. *Cognitive Systems Research*, 12(3-4):237–248, 2011.

[Schmid and Wysotzki, 2000] Ute Schmid and Fritz Wysotzki. Applying inductive program synthesis to macro learning. In *AIPS*, pages 371–378, 2000.

[Schulte *et al.*, 2010] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. Modeling and programming with gecode. 2010.

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

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

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

[Segovia-Aguas *et al.*, 2022a] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Computing programs for generalized planning as heuristic search. In *IJCAI*, 2022.

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

[Seipp *et al.*, 2016] Jendrik Seipp, Florian Pommerening, Gabriele Röger, and Malte Helmert. Correlation complexity of classical planning domains. In *IJCAI*, 2016.

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

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

[Smith *et al.*, 2008] David E Smith, Jeremy Frank, and William Cushing. The anml language. In *KEPS*, 2008.

[Srivastava *et al.*, 2011a] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. A new representation and associated algorithms for generalized planning. *Artificial Intelligence*, 175(2):615–647, 2011.

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

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