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 de Barcelona
[2]Florida Universitària
[3]Universitat Politècnica de València

July 23, 2022

Outline

1. Motivation and contributions
2. Classical Planning and Generalized Planning
3. A C++ Representation for Planning
4. Generalized Planning with C++
5. Evaluation
6. Conclusions and Future Work

Outline

1. Motivation and contributions
2. Classical Planning and Generalized Planning
3. A C++ Representation for Planning
4. Generalized Planning with C++
5. Evaluation
6. Conclusions and Future Work

Motivation and Contributions

There are some long-standing problems in **Generalized Planning** (GP) such as *computing solutions* efficiently; proofs of *partial/total correctness*; scalability over new problems ...

Motivation and Contributions

There are some long-standing problems in **Generalized Planning** (GP) such as *computing solutions* efficiently; proofs of *partial/total correctness*; scalability over new problems ...

1. How can we automatically **prove solutions** are **terminating**? First, with a *representation* that allow terminating solutions, and second by imposing *syntactic constraints* over the solution space

Motivation and Contributions

There are some long-standing problems in **Generalized Planning** (GP) such as *computing solutions* efficiently; proofs of *partial/total correctness*; scalability over new problems ...

1. How can we automatically **prove solutions** are **terminating**? First, with a *representation* that allow terminating solutions, and second by imposing *syntactic constraints* over the solution space
2. How can we estimate the **asymptotic complexity** of solving a planning instance? With solution representations that only *iterate over the set of objects*

Motivation and Contributions

There are some long-standing problems in **Generalized Planning** (GP) such as *computing solutions* efficiently; proofs of *partial/total correctness*; scalability over new problems ...

1. How can we automatically **prove solutions** are **terminating**? First, with a *representation* that allow terminating solutions, and second by imposing *syntactic constraints* over the solution space
2. How can we estimate the **asymptotic complexity** of solving a planning instance? With solution representations that only *iterate over the set of objects*
3. How to **improve runtime** and **memory** requirements in the GP as heuristic search approach? Since solutions are syntactically terminating, there is *no need to check infinite executions*

Motivation and Contributions

There are some long-standing problems in **Generalized Planning** (GP) such as *computing solutions* efficiently; proofs of *partial/total correctness*; scalability over new problems ...

1. How can we automatically **prove solutions** are **terminating**? First, with a *representation* that allow terminating solutions, and second by imposing *syntactic constraints* over the solution space
2. How can we estimate the **asymptotic complexity** of solving a planning instance? With solution representations that only *iterate over the set of objects*
3. How to **improve runtime** and **memory** requirements in the GP as heuristic search approach? Since solutions are syntactically terminating, there is *no need to check infinite executions*
4. How can we **handle large planning instances**? Translating a *solution* to an *efficient high-level programming language*, e.g. C++

Outline

1. Motivation and contributions
2. **Classical Planning and Generalized Planning**
3. A C++ Representation for Planning
4. Generalized Planning with C++
5. Evaluation
6. Conclusions and Future Work

Classical Planning

Classical Planning **Domain** $\mathcal{D} = \langle \mathcal{F}, \mathcal{A} \rangle$

where $\mathcal{F}$ is a set of FOL *predicates*, each of the form $p(x_1, \ldots, x_k)$, and $\mathcal{A}$ is a set of *action schemes*; each $\alpha \in \mathcal{A}$ is defined as $\alpha = \langle par_\alpha, pre_\alpha, eff_\alpha \rangle$ with par_α denoting its *parameters*, and pre_α and eff_α are sets of atoms defined over variables in par_α that stand for the *preconditions* and the (positive/negative) *effects* of the action schema α

Classical Planning **Instance** $\mathcal{I} = \langle \Omega, I, G \rangle$

where Ω is the finite set of world *objects*, I is the *initial state*, and G is the *goal* condition (a partial state that compactly represents the subset of goals states S_G)

Classical Planning **Solution** π

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

Generalized Planning (GP)

GP Problem $\mathcal{P}$

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

GP Solution Π

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

Outline

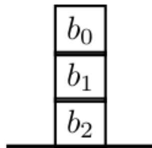
1. Motivation and contributions
2. Classical Planning and Generalized Planning
3. **A C++ Representation for Planning**
4. Generalized Planning with C++
5. Evaluation
6. Conclusions and Future Work

Classical Planning Problems as C++ Programs

State

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). Thus, a **state** is defined as the set of predicate map containers.

Example (State)




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

Classical Planning Problems as C++ Programs

Actions

Each $\alpha \in \mathcal{A}$ is represented with a C++ Boolean function. The arguments of the function are par_α still, but acting as *indexes*. An *index* $z \in Z$ is a finite domain variable ranging the number of objects i.e., $D_z = [0, |\Omega|)$.

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

Classical Planning Problems as C++ Programs

Example (Action)

```
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;
}
```

Classical Planning Problems as C++ Programs

Problem

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:

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

Classical Planning Problems as C++ Programs

Example (Problem)

```
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));
}
```

Sequential Plans as C++ Programs

Sequential Plan

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.

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

Sequential Plans as C++ Programs

Example (Sequential Plan)

```
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);
}
```

Outline

1. Motivation and contributions
2. Classical Planning and Generalized Planning
3. A C++ Representation for Planning
4. **Generalized Planning with C++**
5. Evaluation
6. Conclusions and Future Work

Representing Generalized Plans with C++

Generalized Plan

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

Representing Generalized Plans with C++

Example (Generalized Plan)

```
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);
   }}}}
```

Representing Generalized Plans with C++

Example (Generalized Plan)

```
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);
   }}}}
```

Theorem

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

Synthesis of C++ Programs as Heuristic Search

BFGP++

It is the **algorithm** to synthesize generalized plans, represented as C++ programs that are syntactically terminating. Each **node** in the search is a **partial program** of up to n lines and $|Z|$ indexes over objects.

- **Syntactic constraints**: avoid to update same indexes inside a *For* instruction
- **Prunning rules**: i) do not program *conditionals* or *loops* in the last line; ii) do not program *loops* that only iterate over 1 object
- **Evaluation functions** (the lower the better):
  - $f_{euclidean}(\Pi, \mathcal{P}) = \sum_{P_t \in \mathcal{P}} \sum_{v \in G_t} (s_t[v] - G_t[v])^2$
  - $f_{min(\#loops)}(\Pi)$
  - $f_{max(\#loops)}(\Pi) = -f_{min(\#loops)}(\Pi)$

Outline

1. Motivation and contributions
2. Classical Planning and Generalized Planning
3. A C++ Representation for Planning
4. Generalized Planning with C++
5. **Evaluation**
6. Conclusions and Future Work

Evaluation - Synthesis

Domains	$n, \lvert Z \rvert$	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**

Evaluation - Validation

Domains	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

Outline

1. Motivation and contributions
2. Classical Planning and Generalized Planning
3. A C++ Representation for Planning
4. Generalized Planning with C++
5. Evaluation
6. **Conclusions and Future Work**

Conclusions and Future Work

To wrap up

- A **novel C++ representation** for GP problems and their solutions
- BFGP++ **algorithm** that implements a heuristic search in the space of candidate C++ generalized plans, which naturally models STRIPS domains and outperforms previous GP as heuristic search approach [Segovia-Aguas et al., ICAPS 2021]
- BFGP++ **skips** the costly **check of infinite programs** (terminating by definition)
- **Generalized plans** are translatable **to high-level programming languages**, i.e. C++, and successfully validated in large instances with several thousands of objects

Conclusions and Future Work

To wrap up

- A **novel C++ representation** for GP problems and their solutions
- BFGP++ **algorithm** that implements a heuristic search in the space of candidate C++ generalized plans, which naturally models STRIPS domains and outperforms previous GP as heuristic search approach [Segovia-Aguas et al., ICAPS 2021]
- BFGP++ **skips** the costly **check of infinite programs** (terminating by definition)
- **Generalized plans** are translatable **to high-level programming languages**, i.e. C++, and successfully validated in large instances with several thousands of objects

Future Work

- Better informed *cost-to-go* heuristics, i.e. landmarks [Segovia-Aguas et al., SoCS 2022]
- Automatically proving total correctness
- Getting rid of the algorithm hyperparameters
- Optimal asymptotic complexity for (un)bounded generalized plans

Conclusions and Future Work

To wrap up

- A **novel C++ representation** for GP problems and their solutions
- BFGP++ **algorithm** that implements a heuristic search in the space of candidate C++ generalized plans, which naturally models STRIPS domains and outperforms previous GP as heuristic search approach [Segovia-Aguas et al., ICAPS 2021]
- BFGP++ **skips** the costly **check of infinite programs** (terminating by definition)
- **Generalized plans** are translatable **to high-level programming languages**, i.e. C++, and successfully validated in large instances with several thousands of objects

Future Work

- Better informed *cost-to-go* heuristics, i.e. landmarks [Segovia-Aguas et al., SoCS 2022]
- Automatically proving total correctness
- Getting rid of the algorithm hyperparameters
- Optimal asymptotic complexity for (un)bounded generalized plans

Coffee break!