Computing Programs, Controllers and Grammars with classical planning

Sergio Jiménez

Department of Computing and Information Systems
University of Melbourne

2017

Introduction

Figure 1: Sorting a specific list of numbers.

Introduction

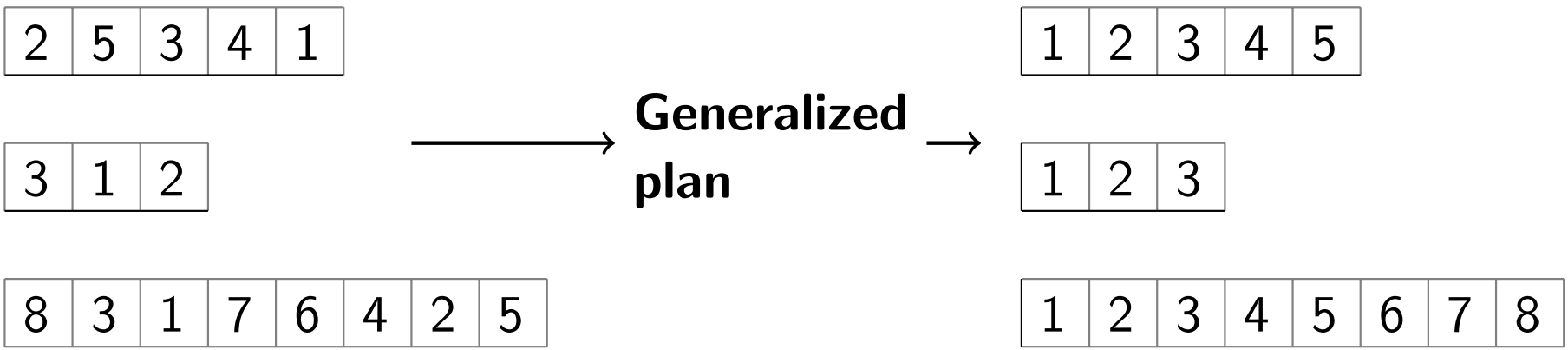


Figure 2: Sorting different lists of numbers with variable size.

Introduction

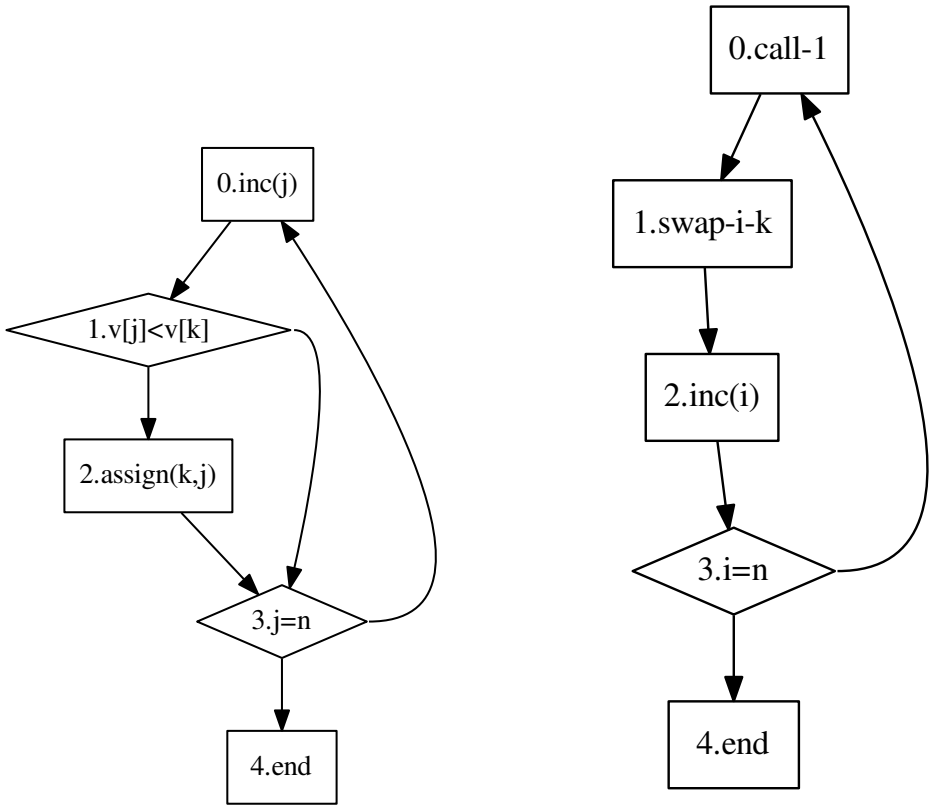


Figure 3: Generalized plan that corresponds to the *select-sort* algorithm (auxiliary procedure left, and main procedure, right).

Classical planning

- A **classical planning task** is a tuple $P = \langle F, A, I, G \rangle$
  - F set of fluents (propositional state variables)
  - A set of actions
  - I initial state
  - G goal condition
- A **classical plan** $\pi = \langle a_1, \ldots, a_n \rangle$ *solves* P iff its application starting from I satisfies G

Classical planning with conditional effects

- **Actions** $a \in A$
  (1) **Preconditions**, $\text{pre}(a)$. *Applicable* in state s iff $\text{pre}(a) \subseteq s$
  (2) **Conditional effects**, $C \rhd E \in \text{cond}(a)$. Two sets of literals: the *condition* C and the *effect* E
    - *Triggered effects*, depend on the state s where a is applied

$$\text{eff}(s, a) = \bigcup_{C \rhd E \in \text{cond}(a), C \subseteq s} E,$$

    - *Resulting state*, computed applying the triggered effects

$$\theta(s, a) = (s \setminus \neg\text{eff}(s, a)) \cup \text{eff}(s, a)$$

Generalized planning

A **generalized planning task** is a set of individual classical planning tasks with conditional effects $\mathcal{P} = \{P_1, \ldots, P_\tau\}$

- Each $P_t = \langle F, A, I_t, G_t \rangle$, $1 \leq t \leq \tau$
  - Share the same planning frame $\Phi = \langle F, A \rangle$
  - **Differ** only in the **initial state** and **goals**
- A generalized plan Π solves $\mathcal{P}$ iff **solves every** $P_t \in \mathcal{P}$

Representing generalized plan as programs

Given planning frame $\Phi = \langle F, A \rangle$, a **program Π is a numbered list of instructions**. Each *instruction* is either:

- An action $a \in A$
- **Conditional goto**, $goto(i', !f)$
  - i' is the target program line
  - $f \in F$ is the jump condition
- **Termination**, marking the end of the program

Representing generalized plan as programs

```
0. dec(x)
1. goto(0,!(x=0))
2. dec(y)
3. goto(2,!(y=0))
4. end
```

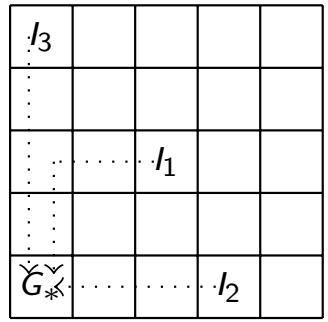


Figure 4: Generalized plan for navigating to cell $(0, 0)$ in a grid.

Representing generalized plan as programs

Execution of program Π on planning problem $P = \langle F, A, I, G \rangle$:

1. Set $s = I$ and $pc = 0$ (*current state* and *program counter*)
2. Update s and pc with instruction w on line pc:
   - 2.1 If $w \in A$ then: $s = \theta(s, w)$ and $pc = pc + 1$
   - 2.2 If $w = goto(i', !f)$ then: $pc = i'$ if $\neg f \in s$ else $pc = pc + 1$
   - 2.3 If w is a termination instruction execution **ends**

Representing generalized plan as programs

1. Π **solves** P, iff Π ends and G holds in s.
2. Π **solves** $\mathcal{P} = \{P_1, \ldots, P_\tau\}$, iff Π solves every $P_t \in \mathcal{P}$
3. Π **fails** on P if:
   - 3.1 Execution ends but goals do not hold
   - 3.2 The precondition of an instruction $a \in A$ does not hold
   - 3.3 Execution enters an infinite loop

A classical planner can detect these success/failure conditions

Computing programs with classical planning

Compiling classical planning problem $P = \langle F, A, I, G \rangle$ into another classical planning problem $P_n = \langle F_n, A_n, I_n, G_n \rangle$:

- n bounds the **max** number of **program lines**
- F_n extends F with fluents for coding
  - **program counter**, $\{pc = 0, \ldots, pc = n\}$
  - **program lines**, $\{\text{ins}_{i,w}\}$ where $0 \leq i \leq n$, $w \in A \cup goto(i', !f) \cup \{\text{end}, \text{nil}\}$
- A_n replaces A with actions for **Programming** and **Executing** instruction w at program line i
  - Program_{w_i} and Execute_{w_i}
- I_n extends I with $pc = 0$ and $\{\text{ins}_{i,nil}\}$ (**empty program lines**)
- G_n extends G with fluent *done* added by Execute_{end_i}

Computing programs with classical planning

```
*0. nil          0. dec(x)
 1. nil          1. goto(0,!(x=0))
 2. nil          2. dec(y)
 3. nil          3. goto(2,!(y=0))
 4. end          4. end
```

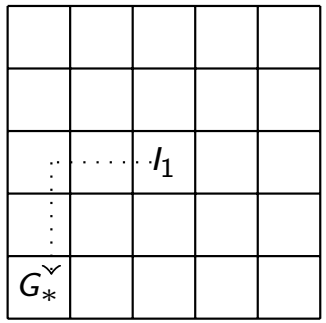


Figure 5: Empty program and generalized plan for navigating to cell $(0, 0)$ in a grid.

Computing programs with classical planning

Compiling generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_\tau\}$ into classical planning problem $P_n = \langle F_n, A_n, I_n, G_n \rangle$:

- F_n includes $\{\text{test}_t\}_{1 \leq t \leq \tau}$ to indicate the current problem
- A_n includes actions $\text{Execute}_{end_{i,t}}$ applicable when $G_t \subseteq s$
  1. Reset the program counter, $pc = 0$
  2. Reset the state $s = I_{t+1}$ (if $t < \tau$) or adds *done* (if $t = \tau$)
- I_n extends I_1 with $pc = 0$ and $\{\text{ins}_{i,nil}\}$

Computing programs with classical planning

```
*0. nil          0. dec(x)
 1. nil          1. goto(0,!(x=0))
 2. nil          2. dec(y)
 3. nil          3. goto(2,!(y=0))
 4. end          4. end
```

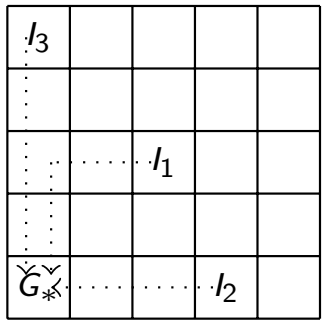


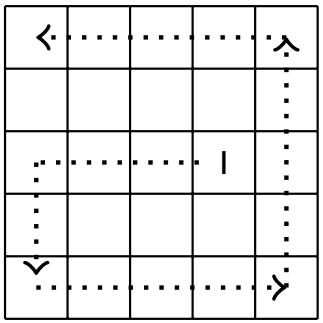
Figure 6: Empty program and generalized plan for navigating to cell $(0, 0)$ in a grid.

Computing programs with classical planning

Limitations of the approach

- May require *informative* examples to generalize
- May require high-level state features [Lotinac et al., 2016]
- May require too long programs [Segovia-Aguas et al., 2016a]

Generalized plans with a call stack



```
main:   0. call(Π1)
        1. call(Π2)
        2. call(Π3)
        3. call(Π4)
        4. end
```

```
Π2:     0. inc(x)
        1. goto(0,!(x=n))
        2. end

Π3:     0. inc(y)
        1. goto(0,!(y=n))
        2. end
```

```
Π4:     0. dec(x)
        1. goto(0,!(x=0))
        2. end
```

(a) (b)

Figure 7: Program for visiting the corners of a $n \times n$ grid using four auxiliary procedures (Π_1 as defined by the program in Figure 4).

Generalized plans with a call stack

We recursively define a **call stack** $\Omega \oplus (j, i)$:

- Ω is a call stack with finite size $|\Omega| = \ell$
- (j, i) is the element at the top of the stack
  - j is the **procedure** Π_j, $0 \leq j \leq m$
  - i is the **line**, $0 \leq i \leq |\Pi_j|$

Generalized plans with a call stack

Execution of program Π on planning problem $P = \langle F, A, I, G \rangle$:

1. Set $s = I$ and $\Omega = (0, 0)$ (*current state* and *stack*)
2. Update s and $\Omega \oplus (j, i)$ with instruction w:
   - 2.1 If $w \in A$, then $s = \theta(s, w)$ and $\Omega = \Omega \oplus (j, i + 1))$
   - 2.2 If $w = goto(i', !f)$ then: $\Omega = \Omega \oplus (j, i'))$ if $\neg f \in s$ else $\Omega = \Omega \oplus (j, i + 1))$
   - 2.3 If $w =$ `call(j')`, $\Omega = \Omega \oplus (j, i + 1) \oplus (j', 0))$ (recursive call when $j = j'$).
   - 2.4 If w is an end instruction pops the top element from the stack. Execution terminates if the stack becomes empty

Some experimental results

Domain	Procs	Solution	Lines	Instances	Time(s)	Total t.(s)	Plan length
Blocks	2	NP	4,3	5,5	2,2	4	46,61
Blocks	1	OP	6	5	85	85	73
Fibonacci	2	NP	3,3	2,5	2,177	179	12,129
Fibonacci	1	OP	5	2	3570	3570	56
Grid	3	NP	5,5,2	2,2,4	611,631,2	1244	43,43,213
Grid	1	ME	10	5	-	-	-
Gripper	3	NP	3,3,3	2,2,2	1,1,2	4	12,12,54
Gripper	1	OP	4	2	1	1	77
Hall-A	5	NP	5,5,5,3,5	2,2,2,2,2	3,7,3,1,4	18	44,40,40,73,155
Hall-A	1	ME	14	2	-	-	-
List	1	R	5	6	5	5	120
Reverse	1	OP	4	2	22	22	38
Sorting	2	NP	4,4	4,3	14,16	30	73,188
Sorting	1	ME	7	4	-	-	-
Summatory	1	OP	3	2	1	1	26
Tree/DFS	1	RP	6	1	165	165	51
Visitall	3	NP	4,2,4	2,2,2	1,1,5	7	47,18,238
Visitall	1	ME	7	2	-	-	-
Visual-Marker	3	NP	4,2,4	4,2,5	2,1,10	13	82,18,205
Visual-Marker	1	ME	8	2	-	-	-

Table 1: Procedures (1: only main; >1: DCK as auxiliary procedures were used), solution kind (NP=Nested Procedures, OP=One Procedure, R=Recursivity, RP=Recursivity with Parameters, ME=Memory error) and per each procedure: program lines, instances used to generate each procedure, planning time and plan length.

Hierarchical Finite State Machines

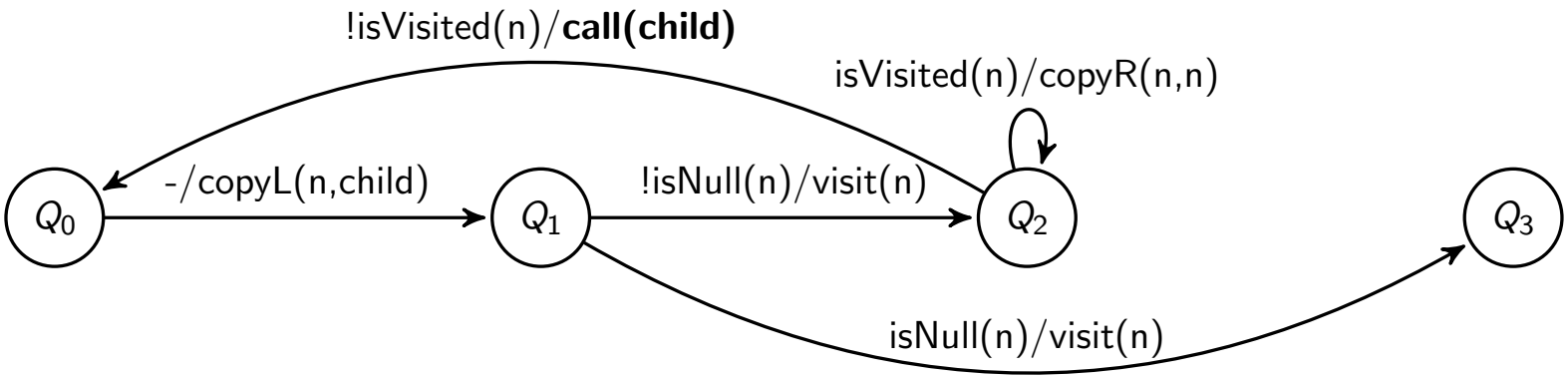


Figure 8: Hierarchical FSC $C[n]$ that traverses a binary tree. The lone parameter of the controller $[n]$ represents the current tree node.

Context Free Grammars

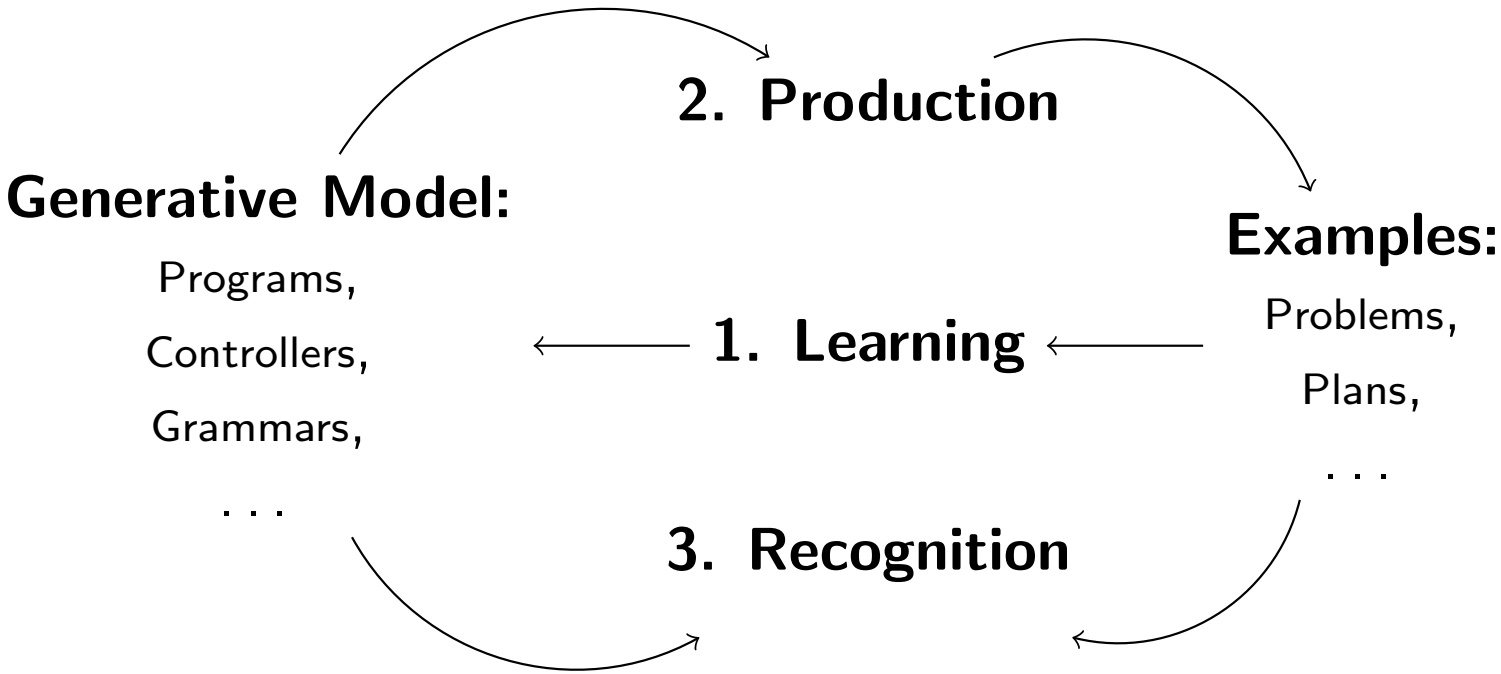
```
S -> aSa
S -> bSb
S -> ϵ
```

Π_0:
```
0. choose(1|5|8)
1. parse_a
2. call(Π0)
3. parse_a
4. end
5. parse_b
6. call(Π0)
7. parse_b
8. end
```

Figure 9: CFG and the corresponding planning program.

Conclusions

- **Learning** different **generative models** from **small** sets of examples with **classical planning**



Bibliography I


Jimenez Celorrio, S. and Jonsson, A. (2015).
Computing plans with control flow and procedures using a classical planner.
In *Eighth Annual Symposium on Combinatorial Search*.


Lotinac, D., Segovia-Aguas, J., Jiménez, S., and Jonsson, A. (2016).
Automatic generation of high-level state features for generalized planning.
In *International Joint Conference on Artificial Intelligence*.


Segovia-Aguas, J., Jiménez, S., and Jonsson, A. (2016a).
Generalized planning with procedural domain control knowledge.
In *International Conference on Automated Planning and Scheduling*.


Segovia-Aguas, J., Jiménez, S., and Jonsson, A. (2016b).
Hierarchical finite state controllers for generalized planning.
In *International Joint Conference on Artificial Intelligence*.


Segovia-Aguas, J., Jiménez, S., and Jonsson, A. (2017).
Generating context-free grammars using classical planning.
In *International Joint Conference on Artificial Intelligence*.