Chicago, USA
2024

Hybrid Declarative-Imperative Representations for Hybrid Discrete-Continuous Decision-Making

Jiayuan Mao

Joshua B. Tenenbaum

Tomás Lozano-Pérez

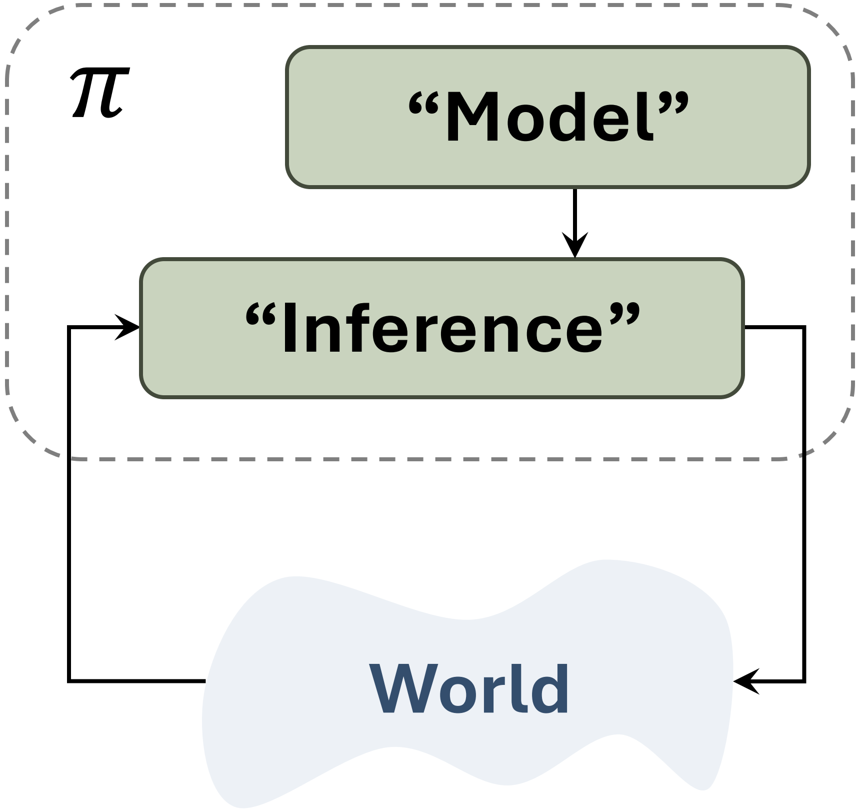
Leslie Pack Kaelbling

Massachusetts Institute of Technology

jiayuanm.com
jiayuanm@mit.edu

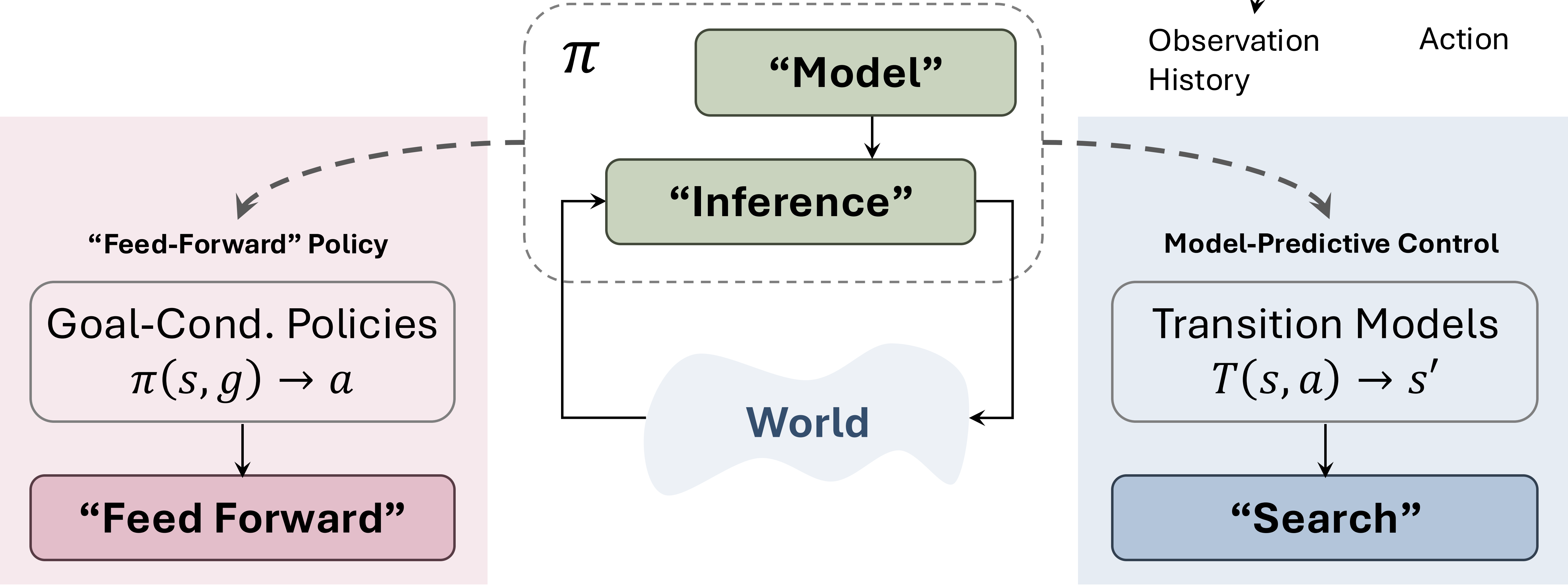
Structures of the “Robot Brain”

$$\pi : \underbrace{(o, a)^*}_{\text{Observation History}} \rightarrow \underbrace{a}_{\text{Action}}$$



Structures of the “Robot Brain”

$$\pi: \underbrace{(o, a)^*}_{\text{Observation History}} \to \underset{\text{Action}}{a}$$



π

“Model”

“Inference”

World

“Feed-Forward” Policy

Goal-Cond. Policies

$\pi(s, g) \to a$

“Feed Forward”

Model-Predictive Control

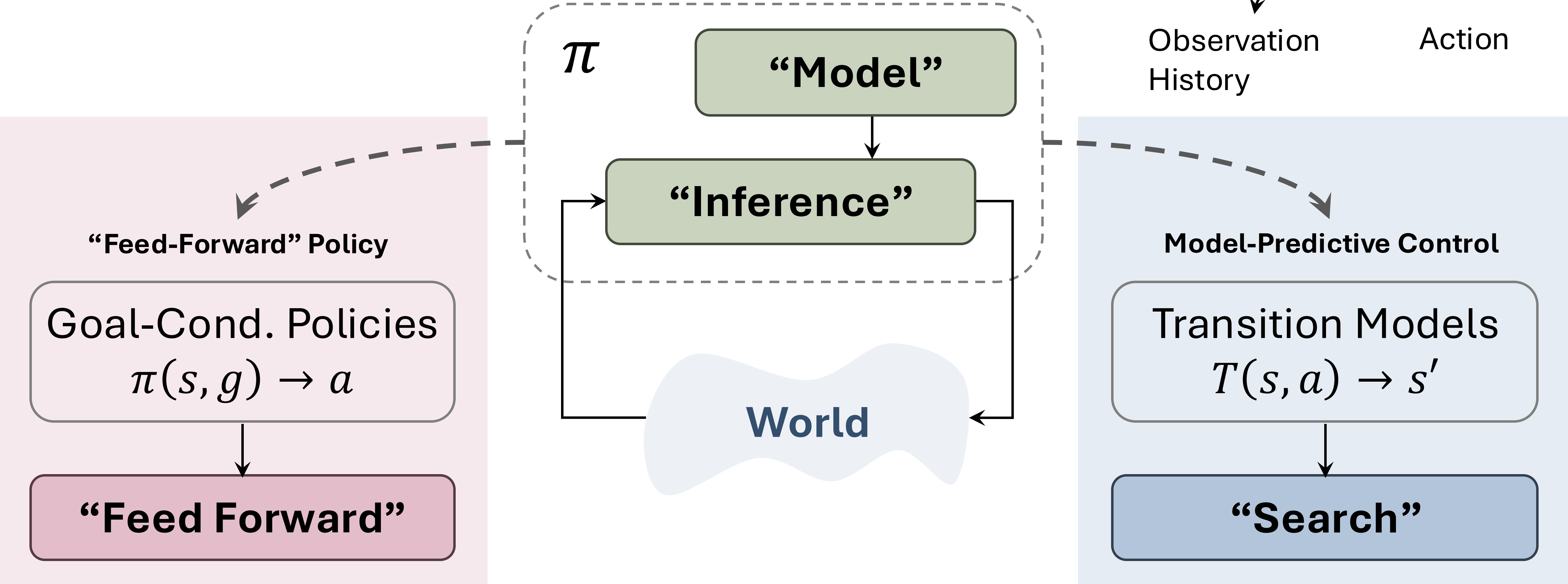
Transition Models

$T(s, a) \to s'$

“Search”

Observations: “We Need Both”

$$\pi: \underbrace{(o,a)^*}_{\text{Observation History}} \to \underset{\text{Action}}{a}$$



π

“Model”

“Inference”

World

“Feed-Forward” Policy

Goal-Cond. Policies

$\pi(s,g) \to a$

“Feed Forward”

Model-Predictive Control

Transition Models

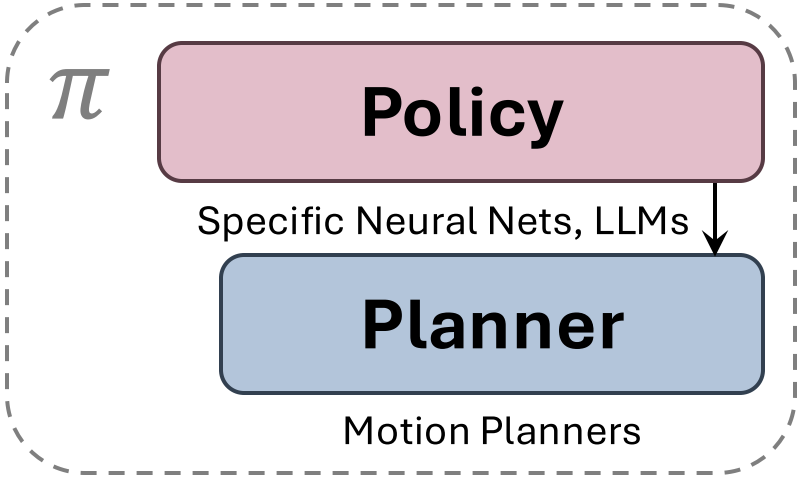
$T(s,a) \to s'$

“Search”

The choice between policy and model depends on the context and the task

Many times, they need to be combined

A Broad Class of “Hybrid” Systems



π

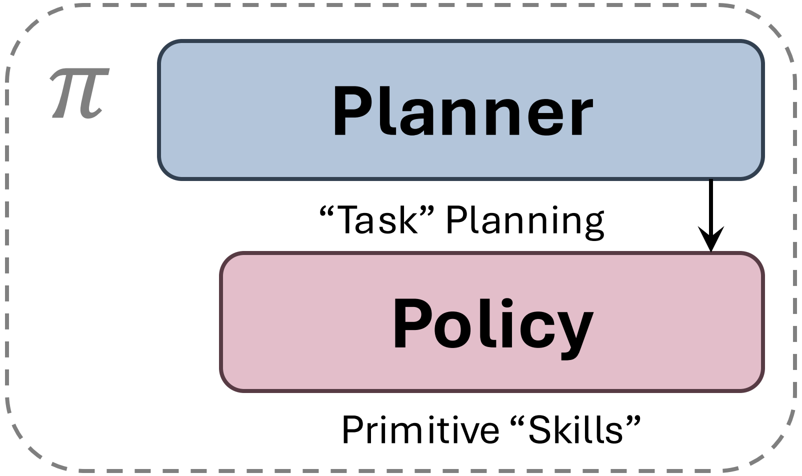
Policy

Specific Neural Nets, LLMs

Planner

Motion Planners

e.g.,
Code-As-Policy
[Liang et al. 2022]



π

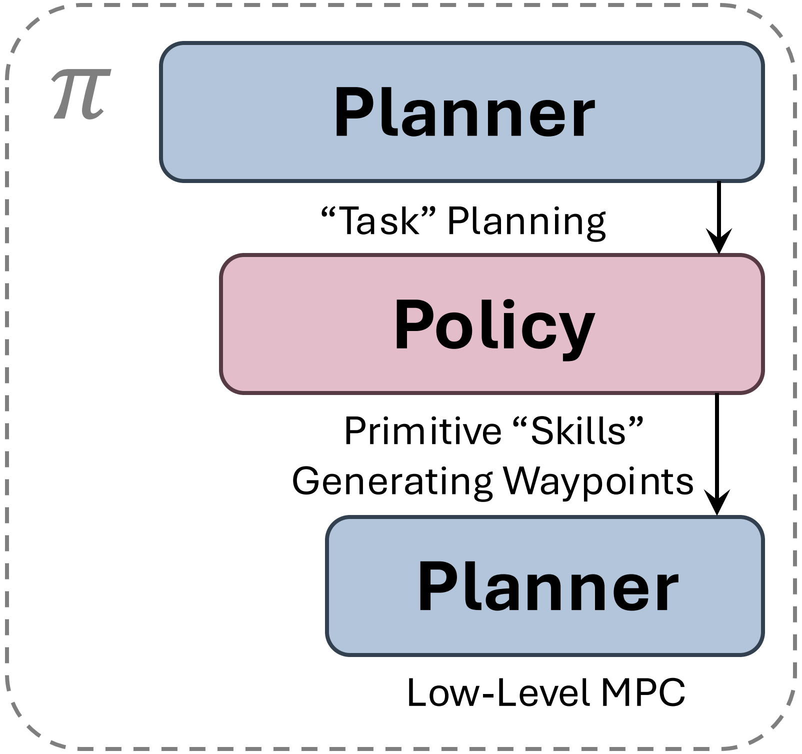
Planner

“Task” Planning

Policy

Primitive “Skills”

e.g.,
Text2Motion
[Lin et al. 2023]



π

Planner

“Task” Planning

Policy

Primitive “Skills”
Generating Waypoints

Planner

Low-Level MPC

Open Research Questions

Theory

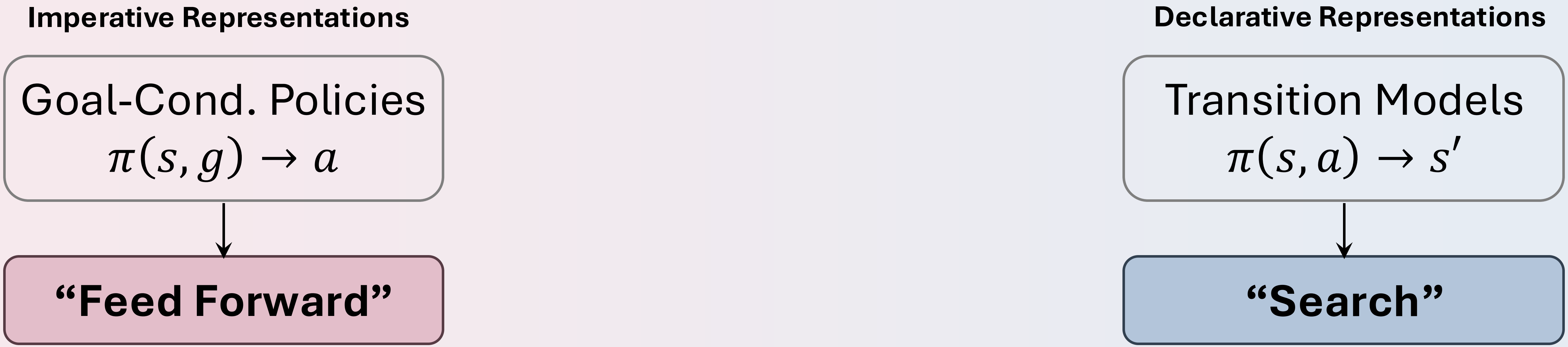
How to mathematically describe all these combinations and their trade-offs (e.g. the complexity)?

Practice

How to flexibly mix-and-match all these modules to build efficient and effective systems?

The Continuous Spectrum of Hybrid Systems

Today’s talk: a unified theory, starting with a “programming language”



Imperative Representations

Goal-Cond. Policies

$\pi(s, g) \rightarrow a$

“Feed Forward”

Declarative Representations

Transition Models

$\pi(s, a) \rightarrow s'$

“Search”

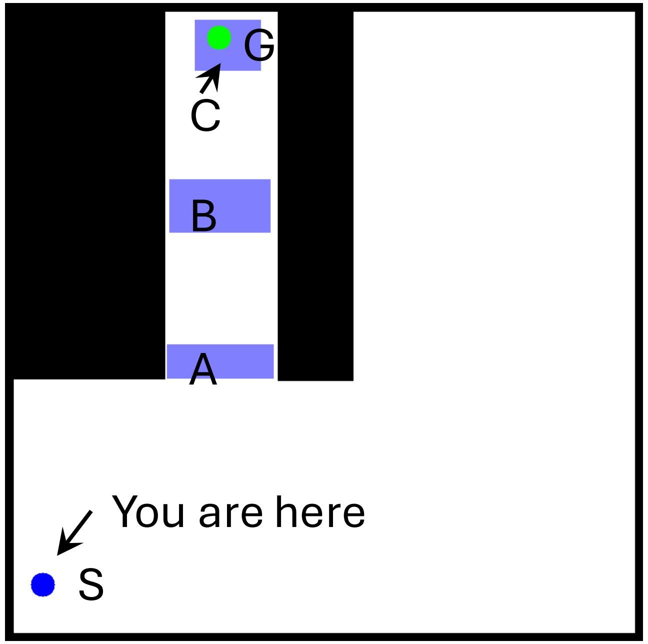
NAMO in the Crow Description Language: Basic Primitives

State: the state is represented as a set of objects and relational features

```
object A, B, C: object
feature shape_of(o: object) -> vector
feature pos_of(o: object) -> vector
```

Primitive Action: parameterized “low-level” controllers

```
controller move_path(t: list[vector])
controller attach(o: object)
```



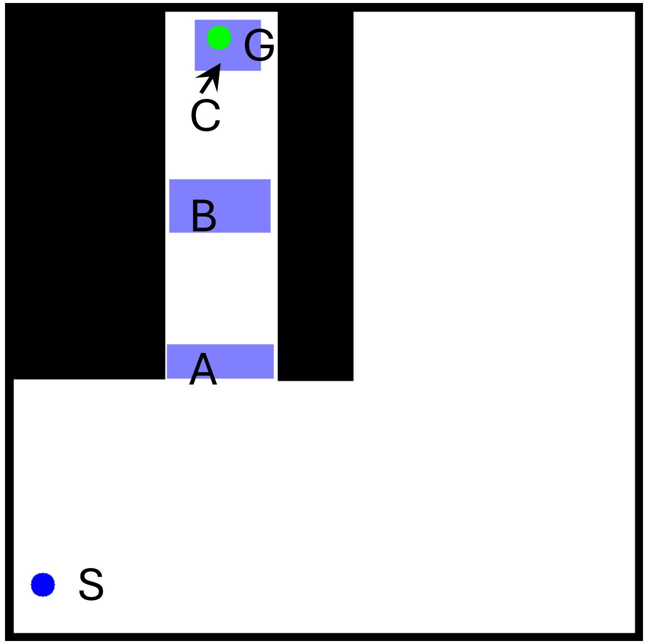
“Navigation Among Movable Obstacles”
Reif and Sharir, 1985
Wilfong, 1988
Stilman and Kuffner, 2005

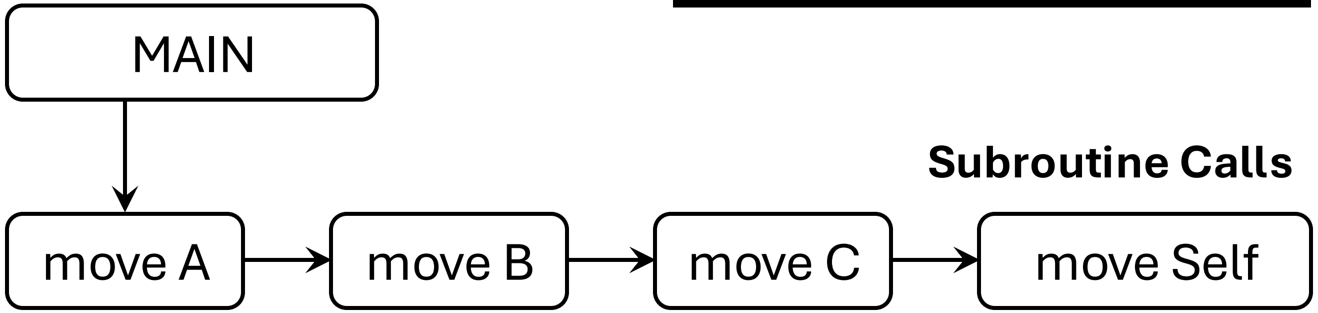
Directly Programmed Solution

Imperative

```
global_goal: agent_pos() == (270, 50)

behavior goto_v0(G):
  goal: agent_pos() == G
  body:
    achieve pos_of(A) == (500, 100)
    achieve pos_of(B) == (500, 300)
    achieve pos_of(C) == (500, 500)
    let path = find_path(agent_pos(), G)
    do move_path(path)
```





MAIN

move A → move B → move C → move Self

Subroutine Calls

Like “Behavior Trees”

Mateas and Stern. 2002. “A Behavior Language for story-based believable agents”
Bagnell et al. 2012. “An Integrated System for Autonomous Robotics Manipulation”
Colledanchise and Ögren. 2018 “Behavior Trees in Robotics and AI”

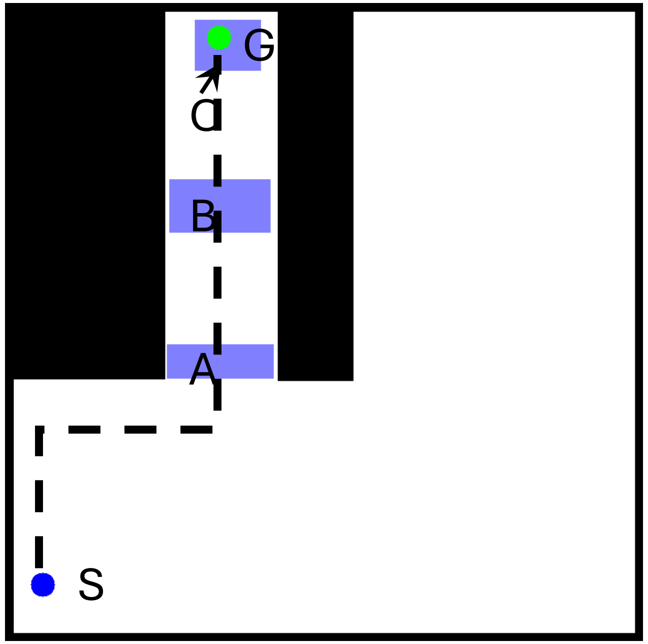
Adding (Continuous) Variable Bindings

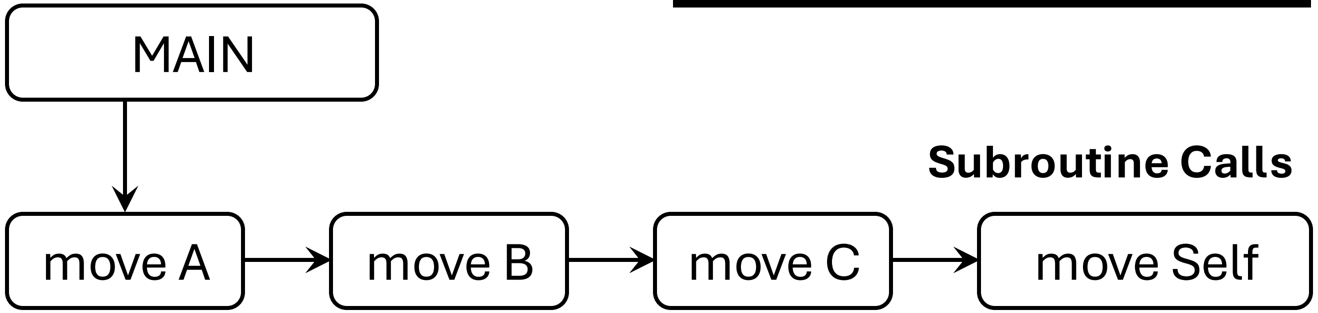
Imperative **+Variable**

```
global_goal: agent_pos() == (270, 50)

behavior goto_v1(G: vector):
  goal: agent_pos() == G
  body:
    bind path = find_path(agent_pos(), G)
    achieve not_blocking(A, path)
    achieve not_blocking(B, path)
    achieve not_blocking(C, path)
    do move_path(path)


behavior move_away(x: object, path):
  goal: not_blocking(x, p)
  body:
    bind new_p: valid_pos(x, new_p)
    ...
```





Subroutine Calls

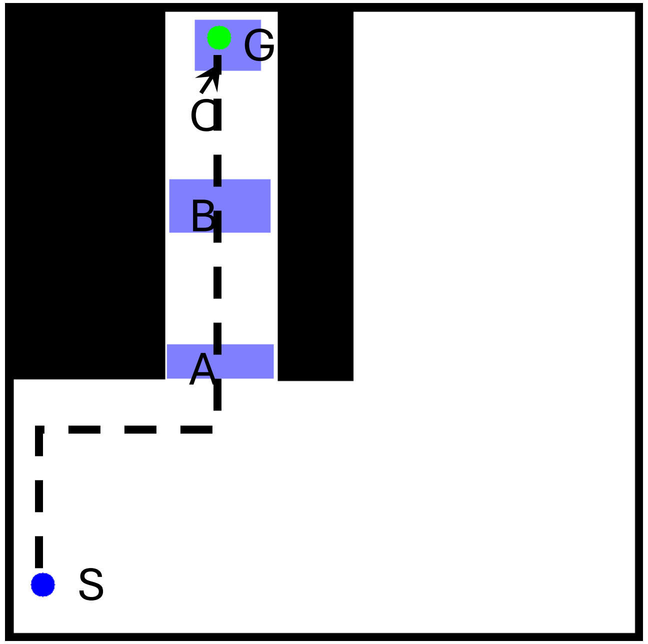
Adding Flexible Ordering

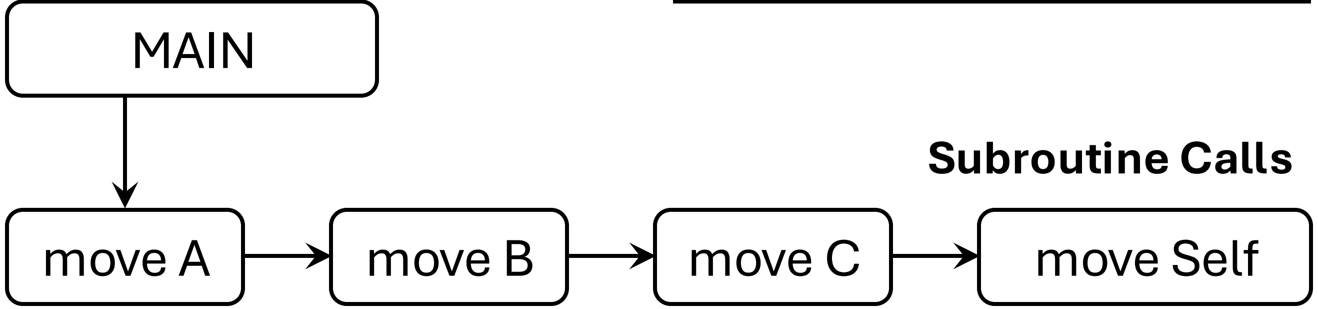
Imperative **+Variables** **+Ordering**

```
global_goal: agent_pos() == (270, 50)

behavior goto_v1(G: vector):
  goal: agent_pos() == G
  body:
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)

behavior move_away(x: object, path):
  goal: not_blocking(x, p)
  body:
    bind new_p: valid_pos(x, new_p)
    ...
```





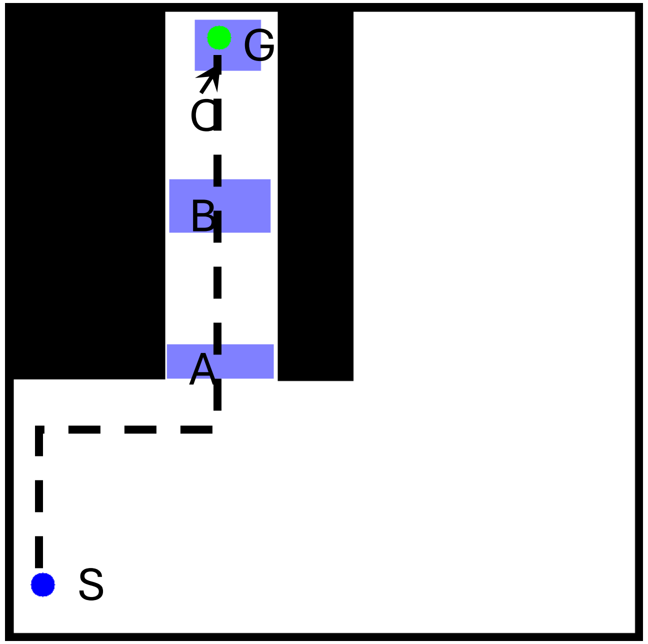
Adding Flexible Ordering

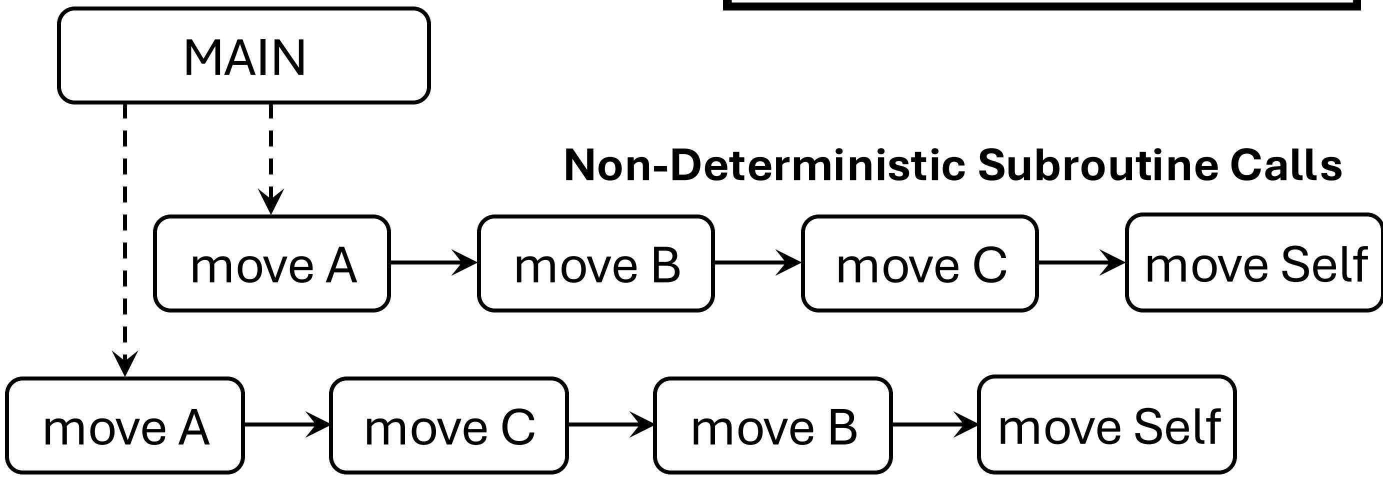
Imperative **+Variables** **+Ordering**

```
global_goal: agent_pos() == (270, 50)

behavior goto_v1(G: vector):
  goal: agent_pos() == G
  body:
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)

behavior move_away(x: object, path):
  goal: not_blocking(x, p)
  body:
    bind new_p: valid_pos(x, new_p)
    ...
```




Non-Deterministic Subroutine Calls

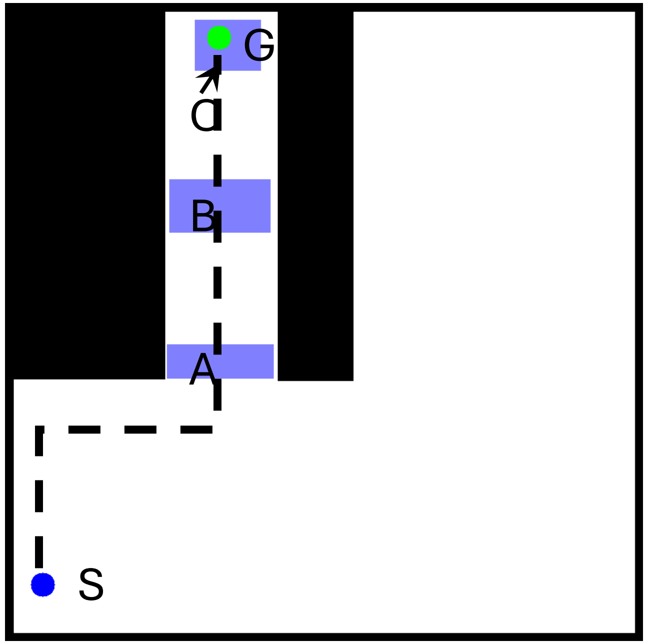
Adding Flexible Ordering

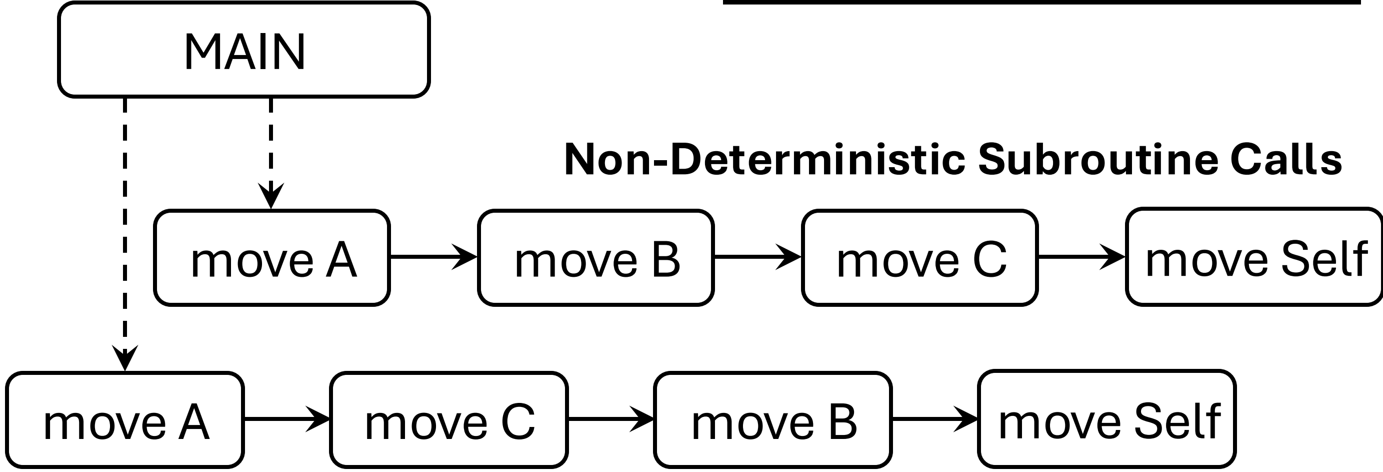
Imperative **+Variables** **+Ordering**

```
global_goal: agent_pos() == (270, 50)

behavior goto_v1(G: vector):
  goal: agent_pos() == G
  body:
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)

behavior move_away(x: object, path):
  goal: not_blocking(x, p)
  body:
    assert reachable(x)
    bind new_p: valid_pos(x, new_p)
    ...
  eff: pos_of(x) = new_p
```




Non-Deterministic Subroutine Calls

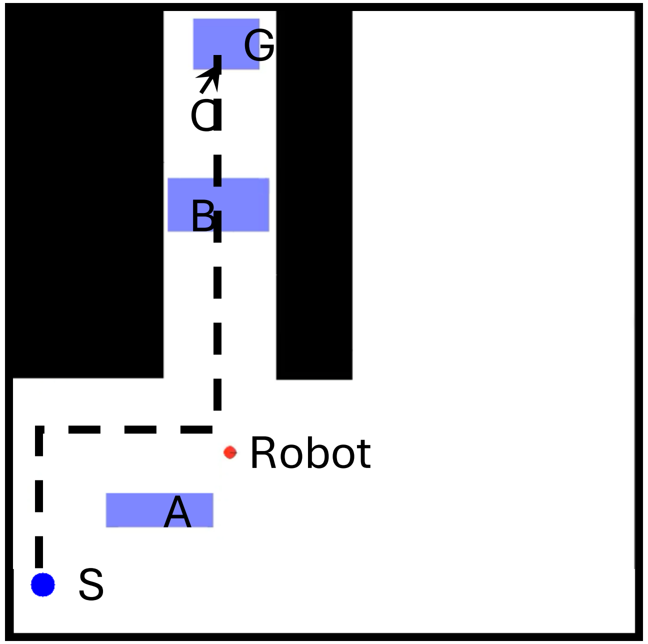
Adding Flexible Ordering

Imperative **+Variables** **+Ordering**

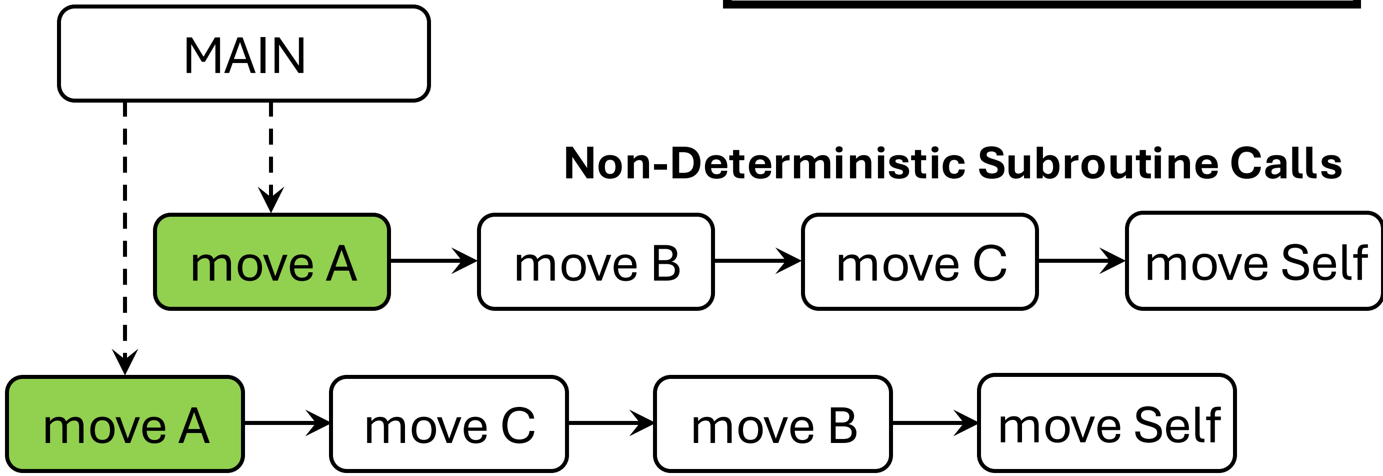
```
global_goal: agent_pos() == (270, 50)

behavior goto_v1(G: vector):
  goal: agent_pos() == G
  body:
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)

behavior move_away(x: object, path):
  goal: not_blocking(x, p)
  body:
    assert reachable(x)
    bind new_p: valid_pos(x, new_p)
    ...
  eff: pos_of(x) = new_p
```



Non-Deterministic Subroutine Calls



MAIN

move A → move B → move C → move Self

move A → move C → move B → move Self

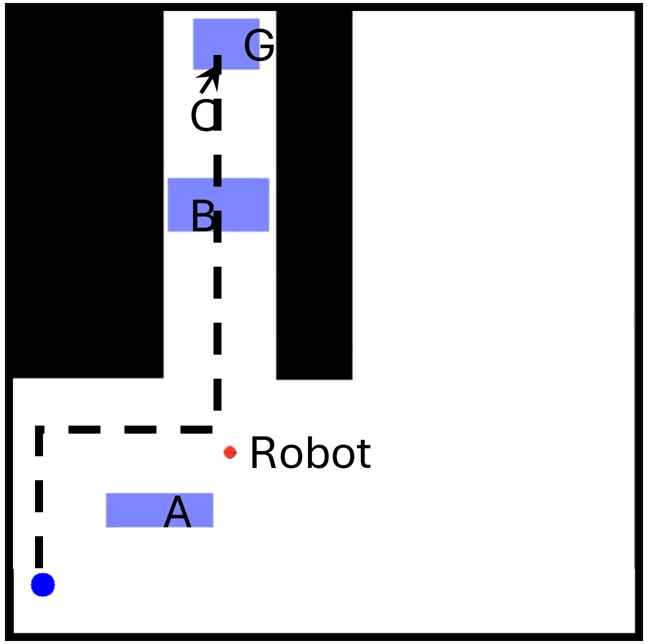
Adding Flexible Ordering

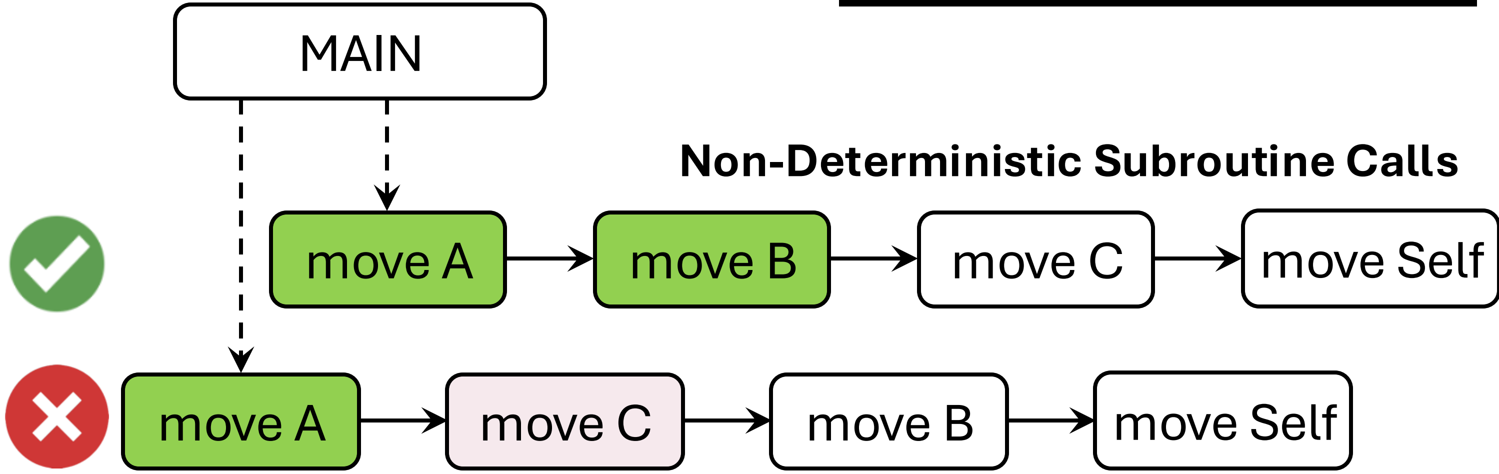
Imperative ● **+Variables** ● **+Ordering** ●

```
global_goal: agent_pos() == (270, 50)

behavior goto_v1(G: vector):
  goal: agent_pos() == G
  body:
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)

behavior move_away(x: object, path):
  goal: not_blocking(x, p)
  body:
    assert reachable(x)
    bind new_p: valid_pos(x, new_p)
    ...
  eff: pos_of(x) = new_p
```




Non-Deterministic Subroutine Calls

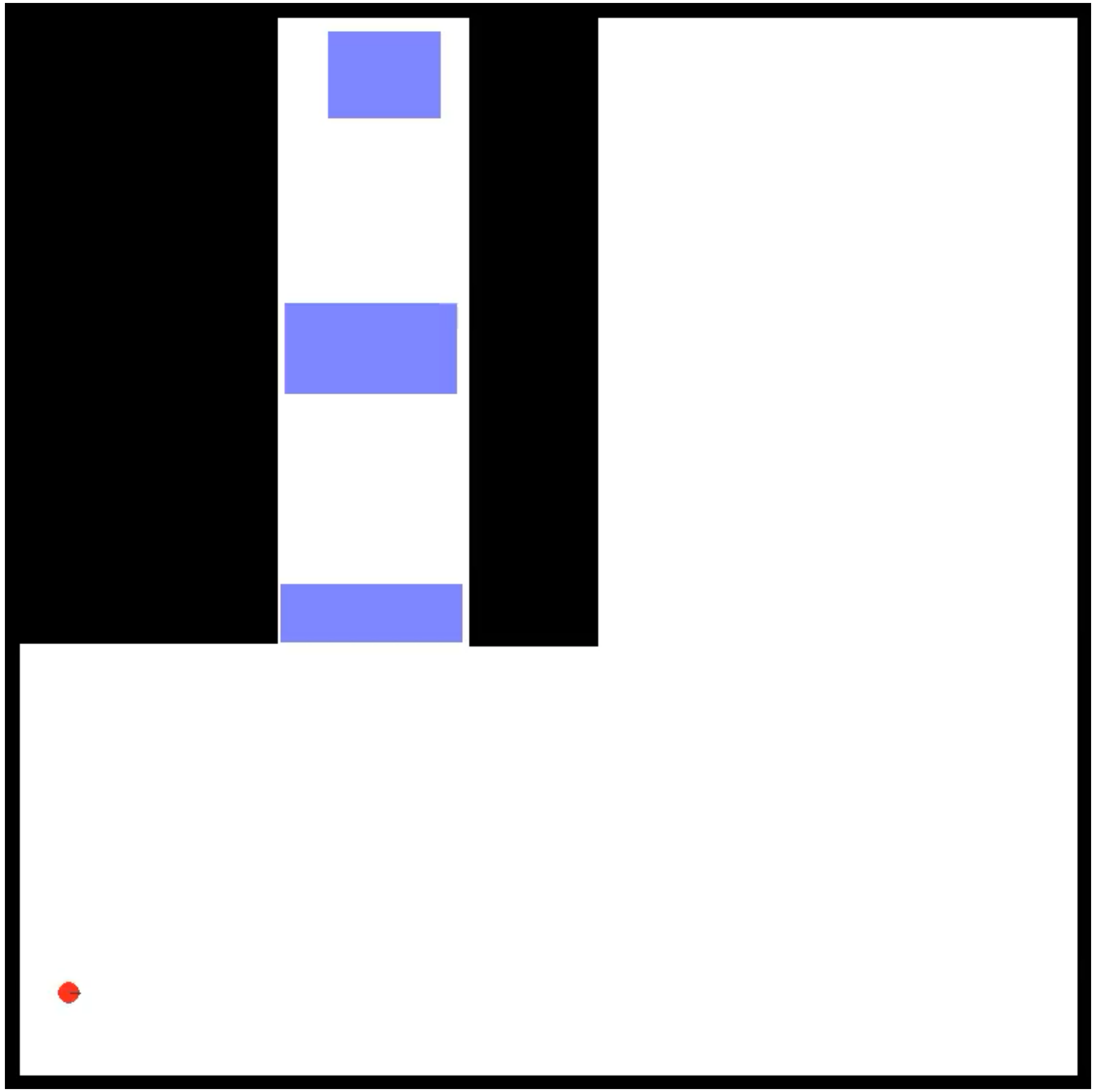
Adding Flexible Ordering

Imperative **+Variables** **+Ordering**

```
global_goal: agent_pos() == (270, 50)

behavior goto_v1(G: vector):
  goal: agent_pos() == G
  body:
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)

behavior move_away(x: object, path):
  goal: not_blocking(x, p)
  body:
    assert reachable(x)
    bind new_p: valid_pos(x, new_p)
    ...
  eff: pos_of(x) = new_p
```



The Spectrum Between Imperative and Declarative

Imperative **+Variables** **+Ordering**

Insight 1: Behaviors = Generators of “non-deterministic subroutine calls”

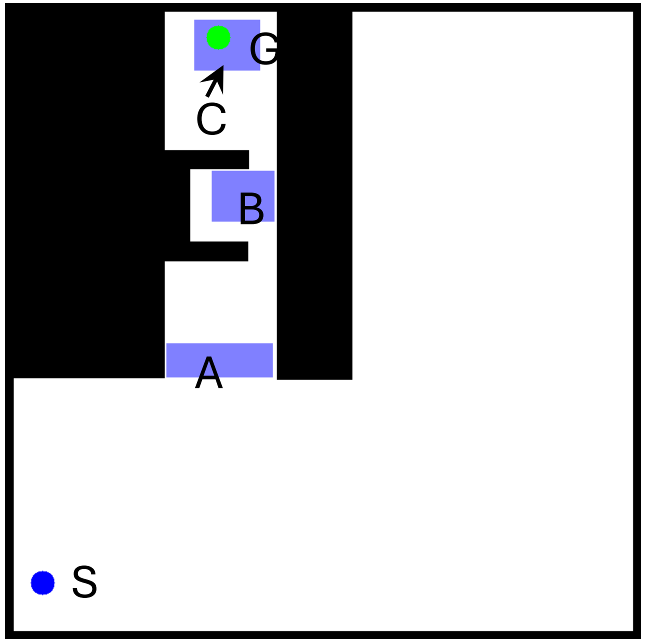
+ Verifiers based on causal models

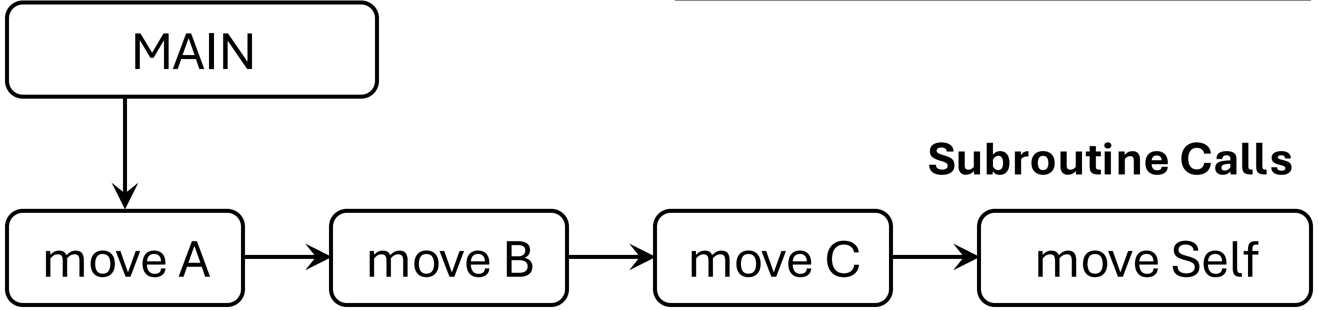
Adding Flexible Ordering

Imperative **+Variables** **+Ordering**

```
global_goal: agent_pos() == (270, 50)

behavior goto_v1(G: vector):
  goal: agent_pos() == G
  body:
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)
```



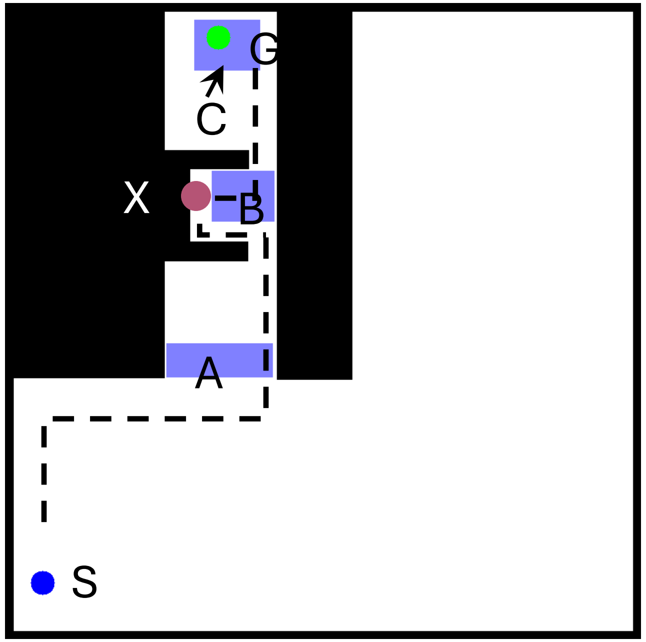


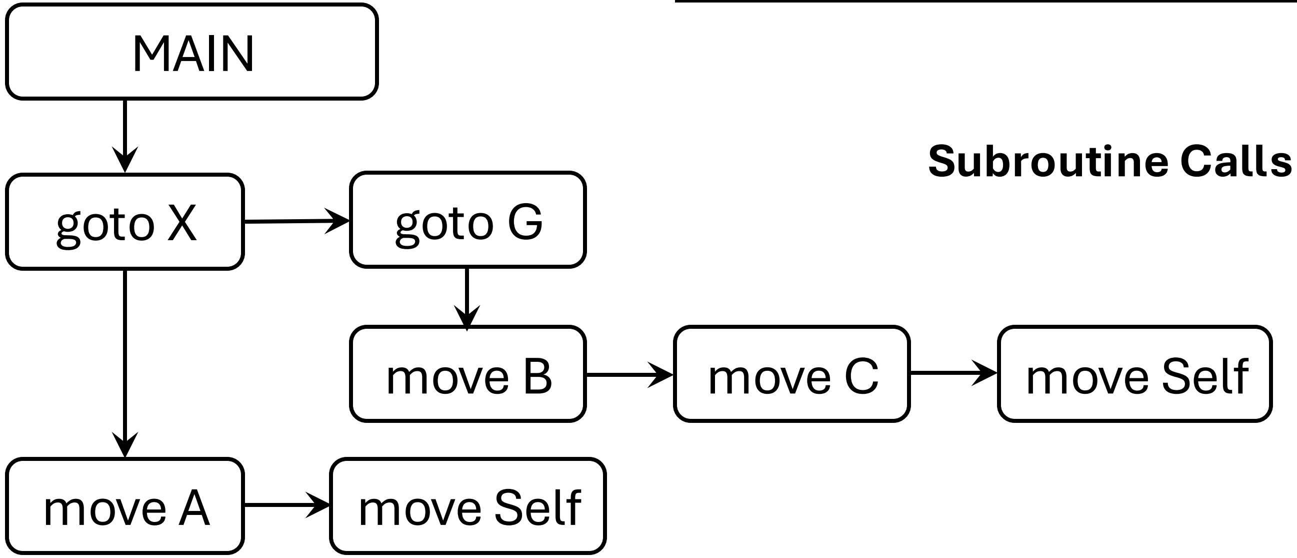
Adding More Recursive Subroutines

Imperative **+Variables** **+Ordering**

```
global_goal: agent_pos() == (270, 50)

behavior goto_v2(G: vector):
  goal: agent_pos() == G
  body:
    bind waypoint: vector
    achieve agent_pos() == waypoint
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)
```




MAIN

goto X

goto G

move B

move C

move Self

move A

move Self

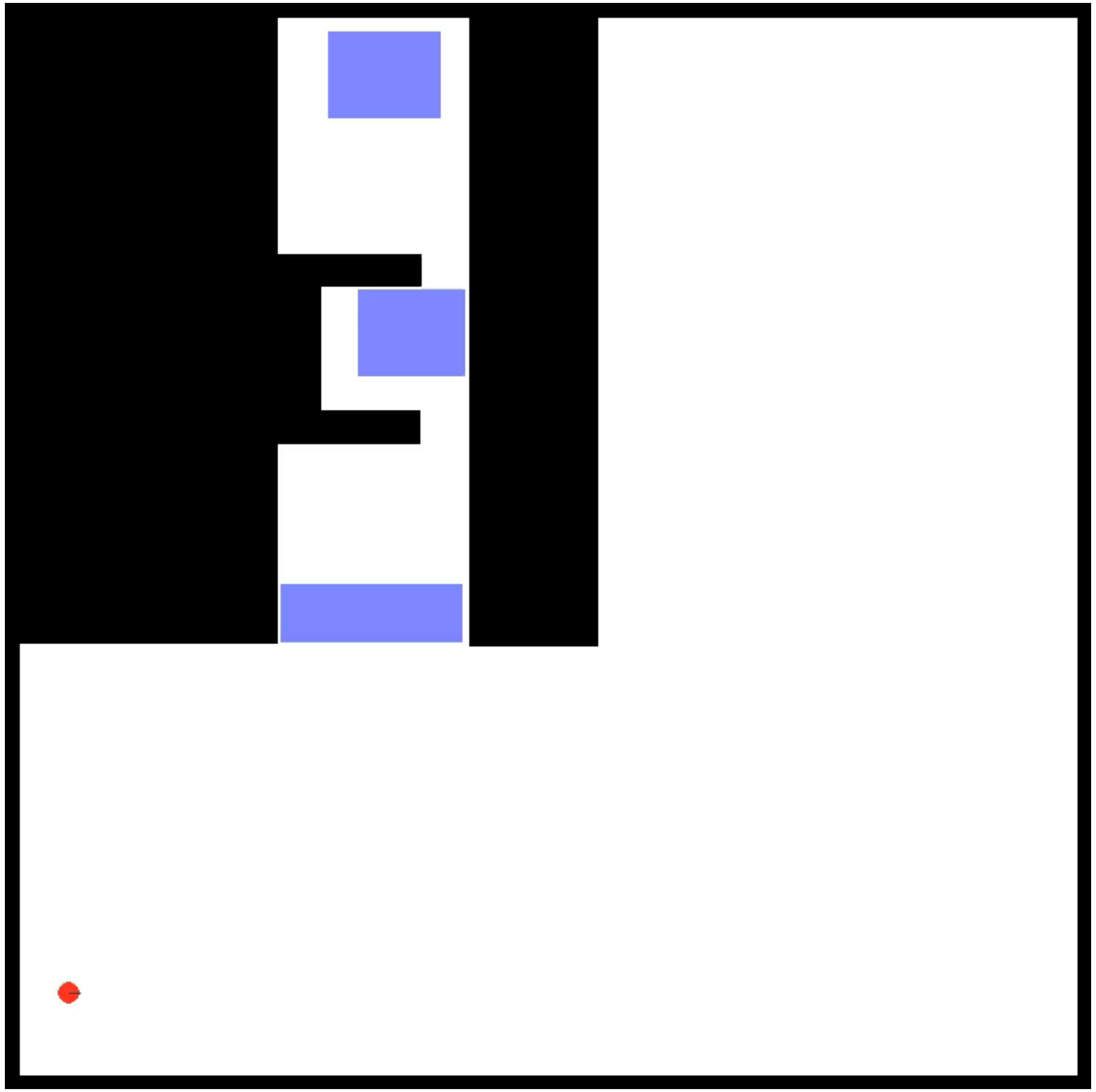
Subroutine Calls

Adding More Recursive Subroutines

Imperative **+Variables** **+Ordering**

```
global_goal: agent_pos() == (270, 50)

behavior goto_v2(G: vector):
  goal: agent_pos() == G
  body:
    bind waypoint: vector
    achieve agent_pos() == waypoint
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)
```

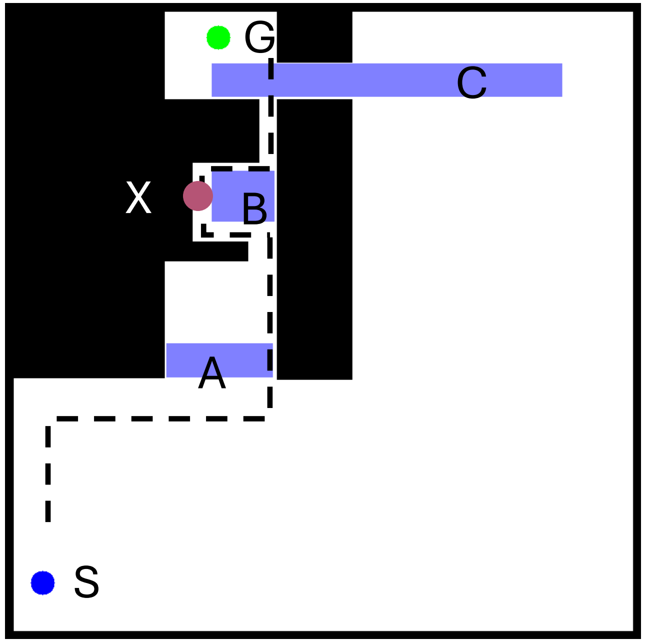


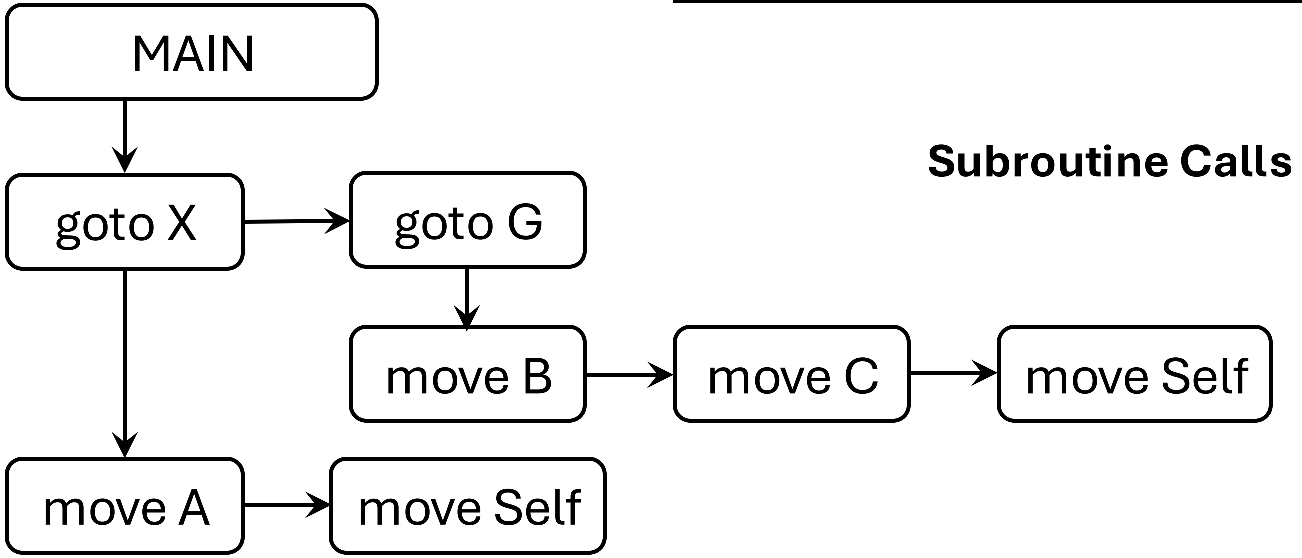
Adding Flexible Promotion

Imperative **+Variables** **+Ordering** **+Promotion**

```
global_goal: agent_pos() == (270, 50)

behavior goto_v2(G: vector):
  goal: agent_pos() == G
  body:
    bind waypoint: vector
    achieve agent_pos() == waypoint
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)
```





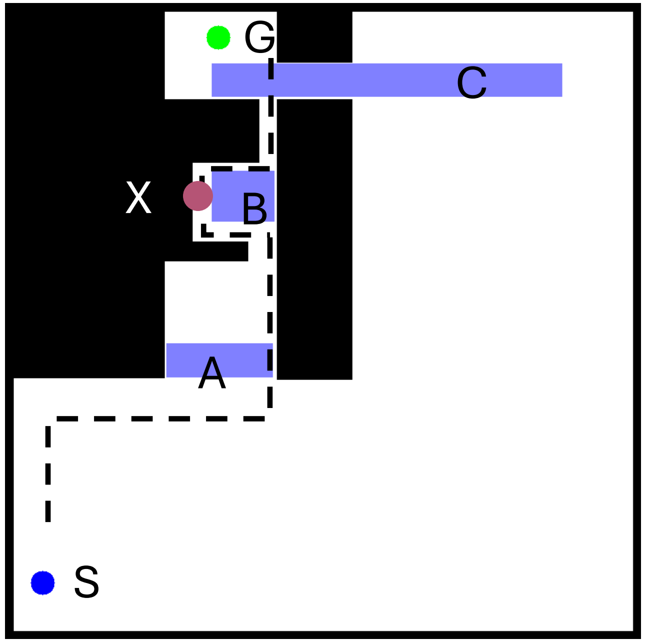
Subroutine Calls

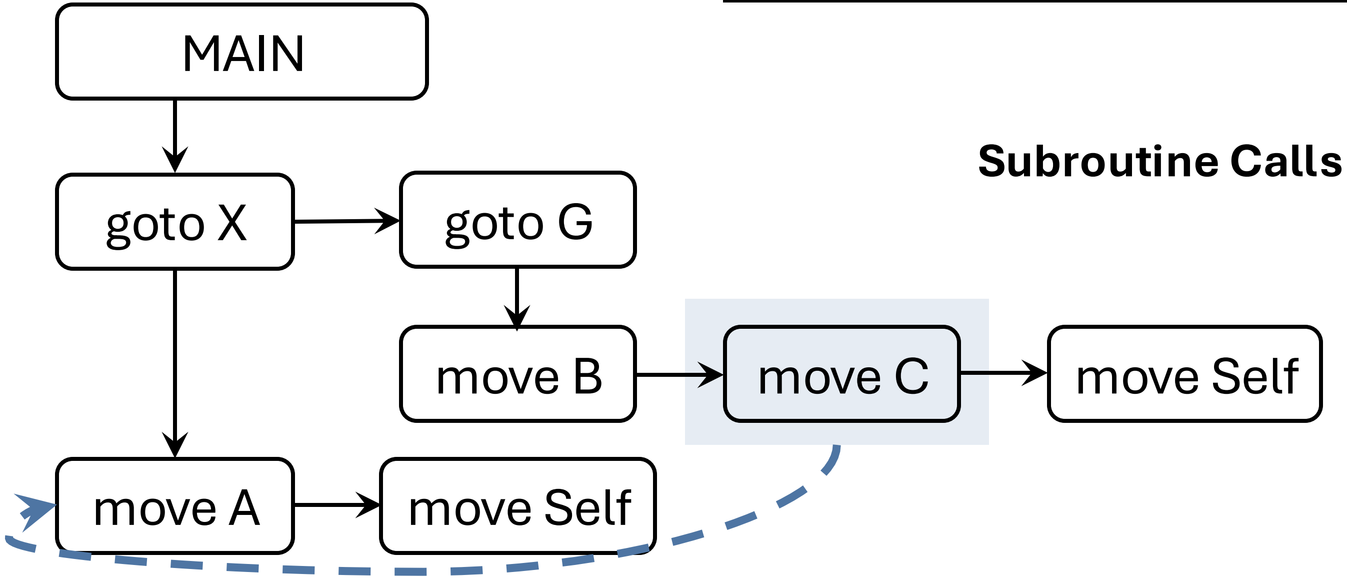
Adding Flexible Promotion

Imperative **+Variables** **+Ordering** **+Promotion**

```
global_goal: agent_pos() == (270, 50)

behavior goto_v2(G: vector):
  goal: agent_pos() == G
  body:
    bind waypoint: vector
    achieve agent_pos() == waypoint
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)
```





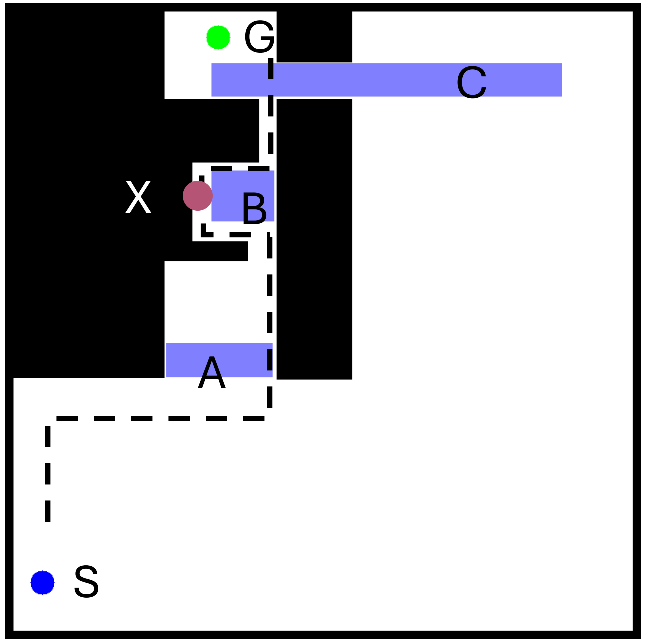
Subroutine Calls

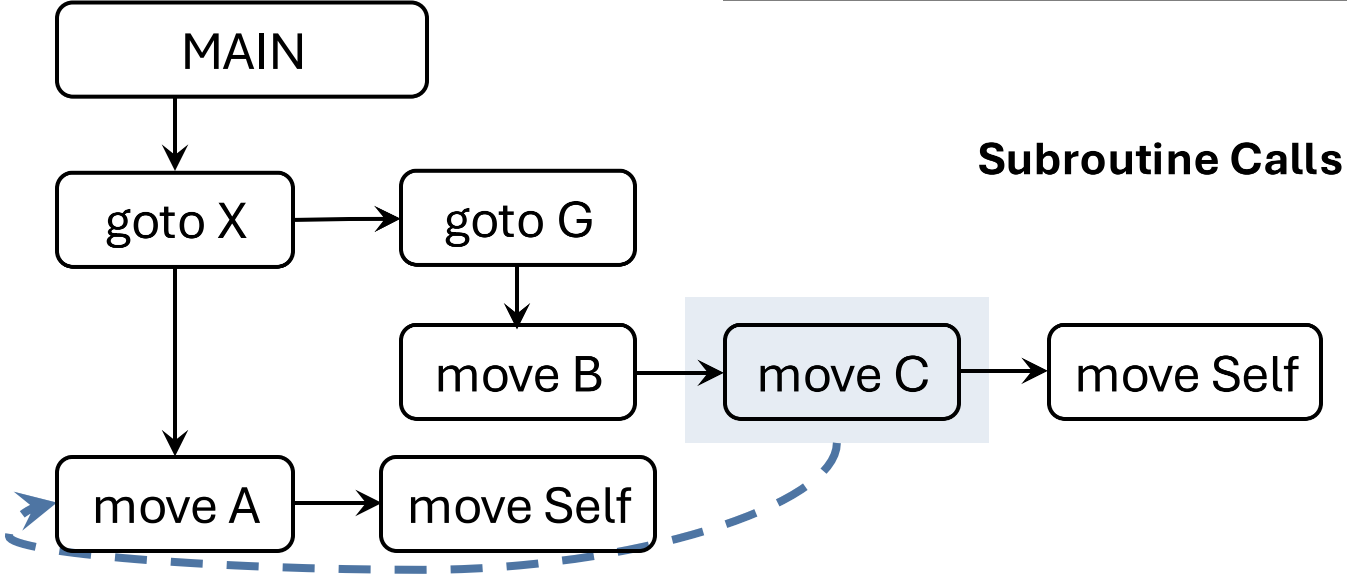
Adding Flexible Promotion

Imperative **+Variables** **+Ordering** **+Promotion**

```
global_goal: agent_pos() == (270, 50)

behavior goto_v3(G: vector):
  goal: agent_pos() == G
  body:
    bind waypoint: vector
    achieve agent_pos() == waypoint
    bind path = find_path(agent_pos(), G)
    promotable unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)
```

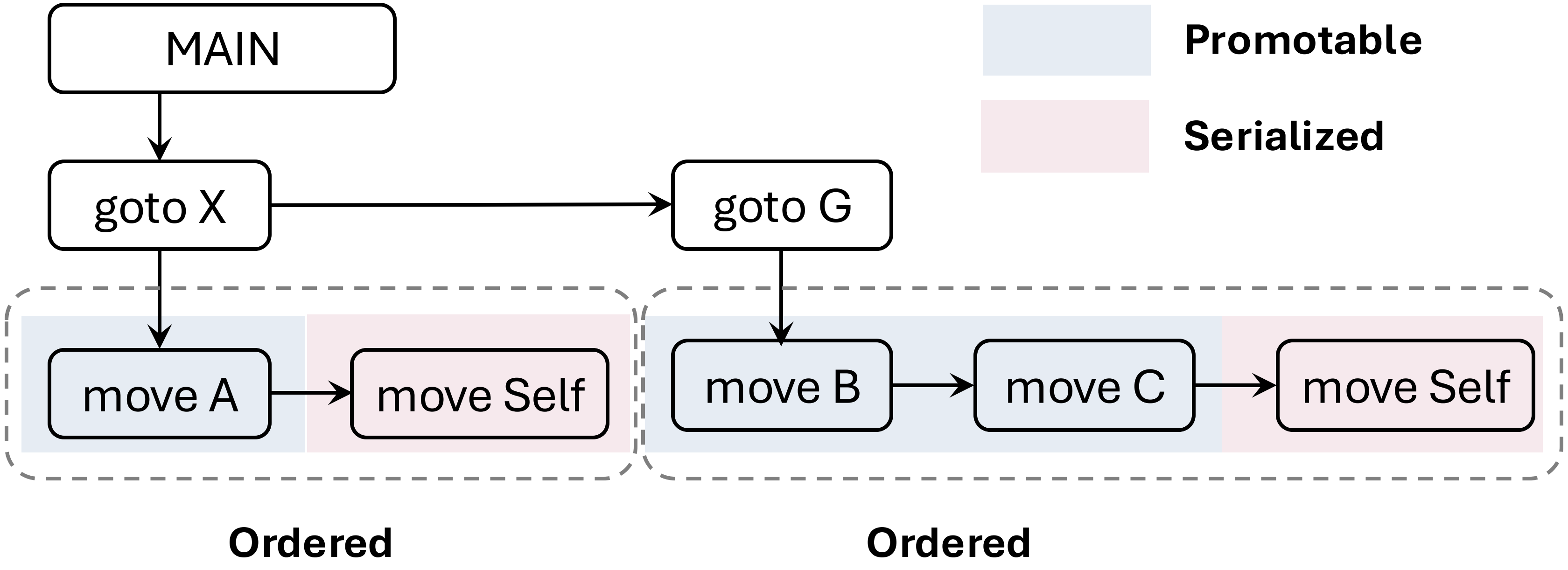




Subroutine Calls

Adding Flexible Promotion

Imperative **+Variables** **+Ordering** **+Promotion**



MAIN

goto X

goto G

move A

move Self

move B

move C

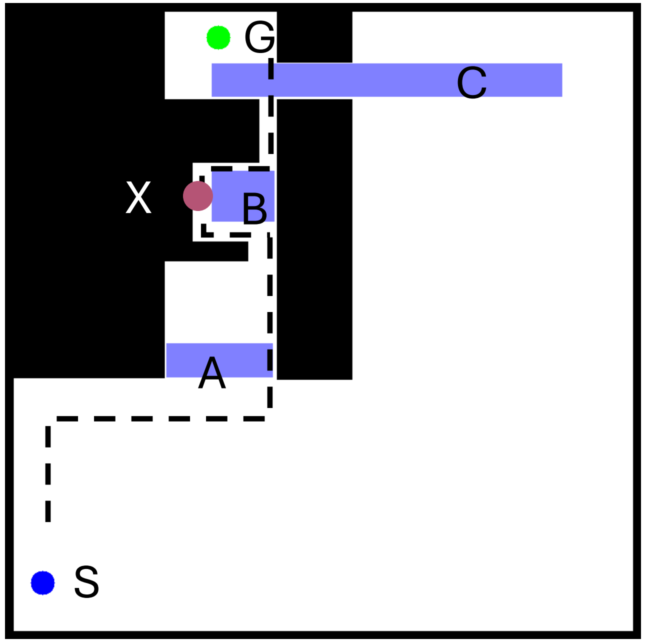
move Self

Promotable

Serialized

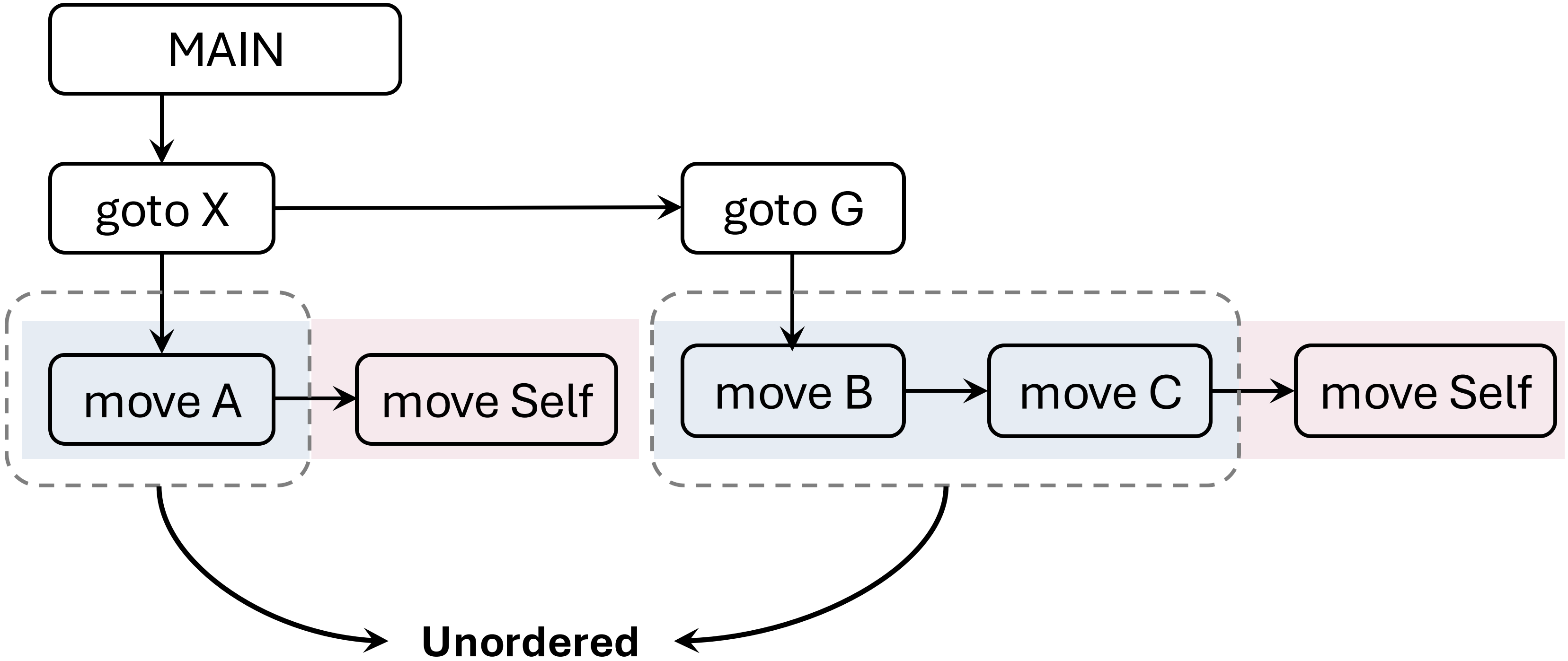
Ordered

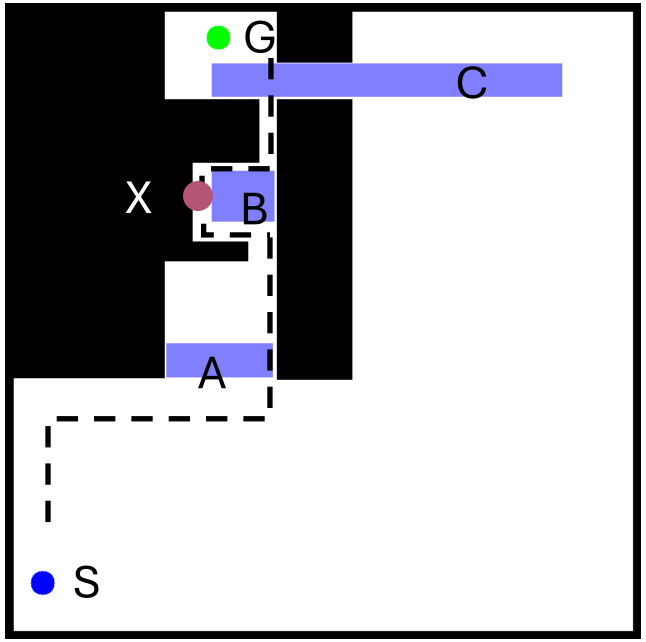
Ordered



Adding Flexible Promotion

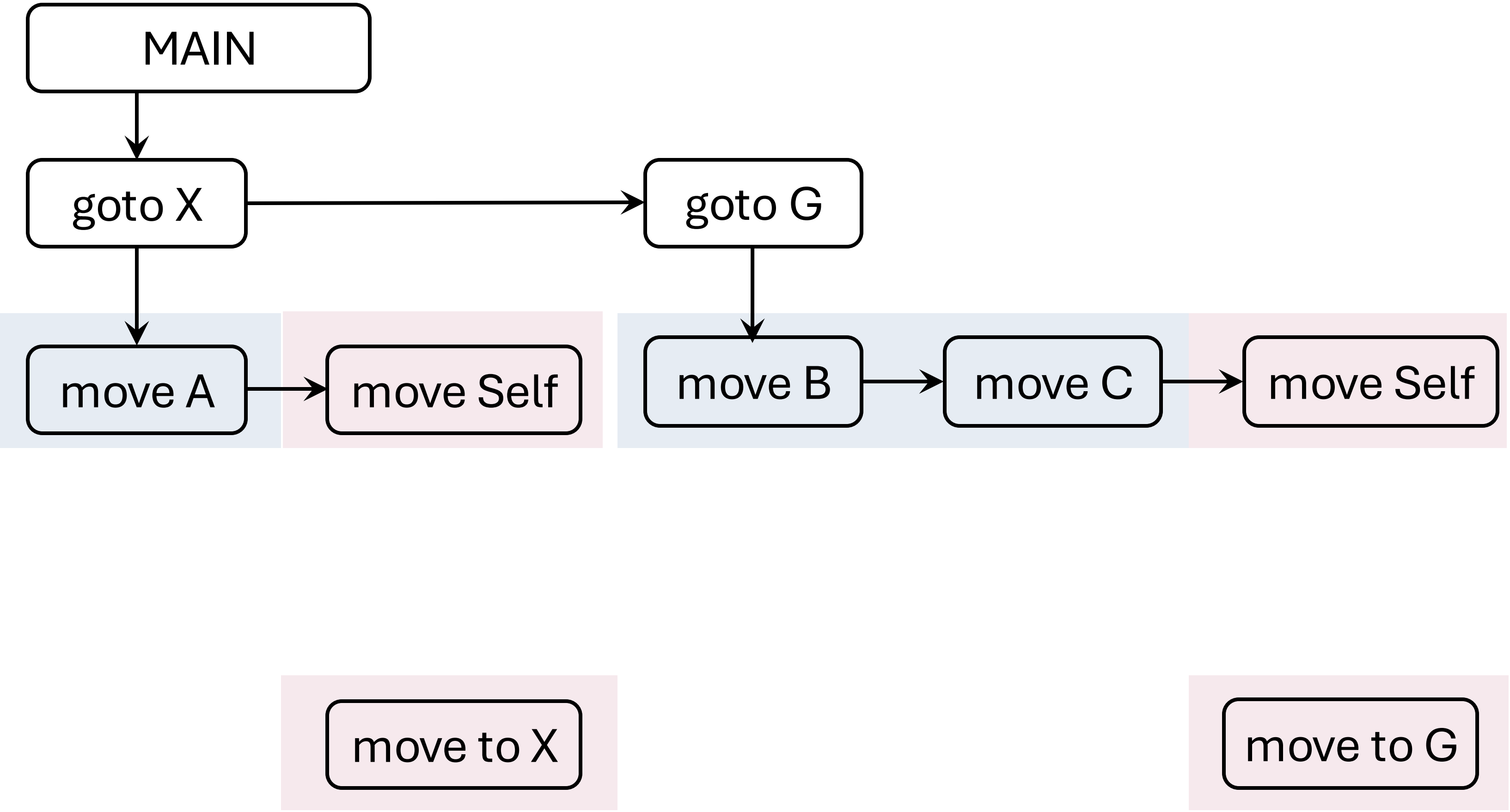
Imperative **+Variables** **+Ordering** **+Promotion**

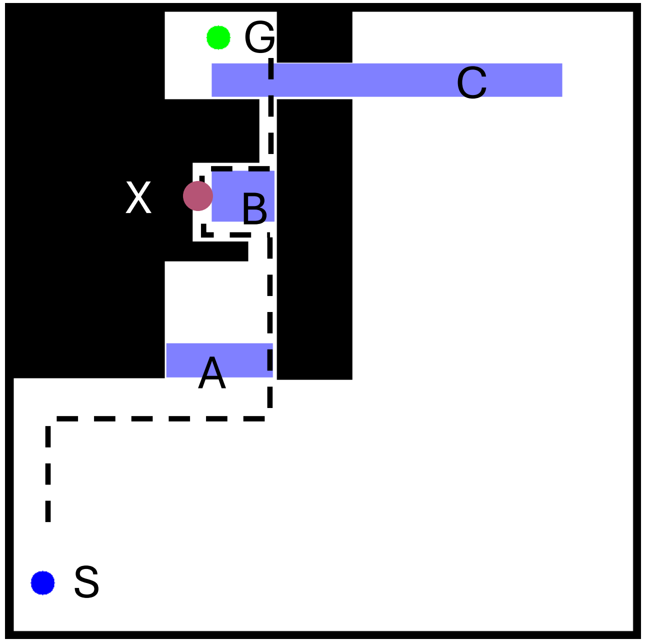




Adding Flexible Promotion

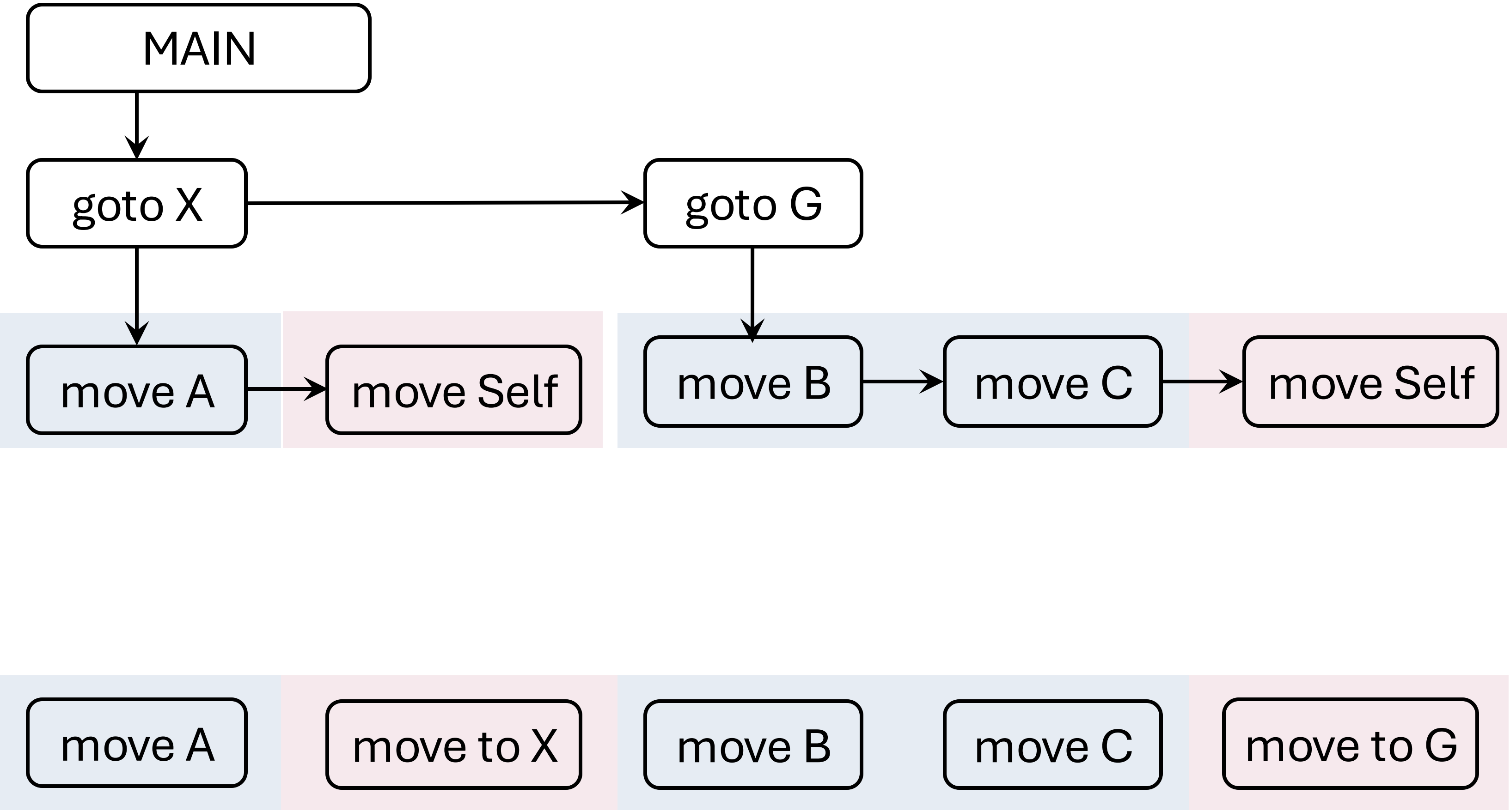
Imperative **+Variables** **+Ordering** **+Promotion**





Adding Flexible Promotion

Imperative +Variables +Ordering +Promotion


MAIN

goto X

goto G

move A

move Self

move B

move C

move Self

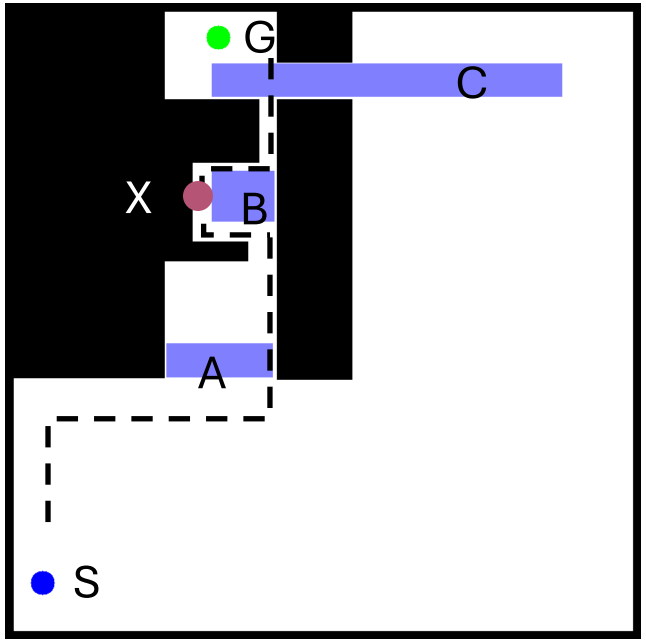
move A

move to X

move B

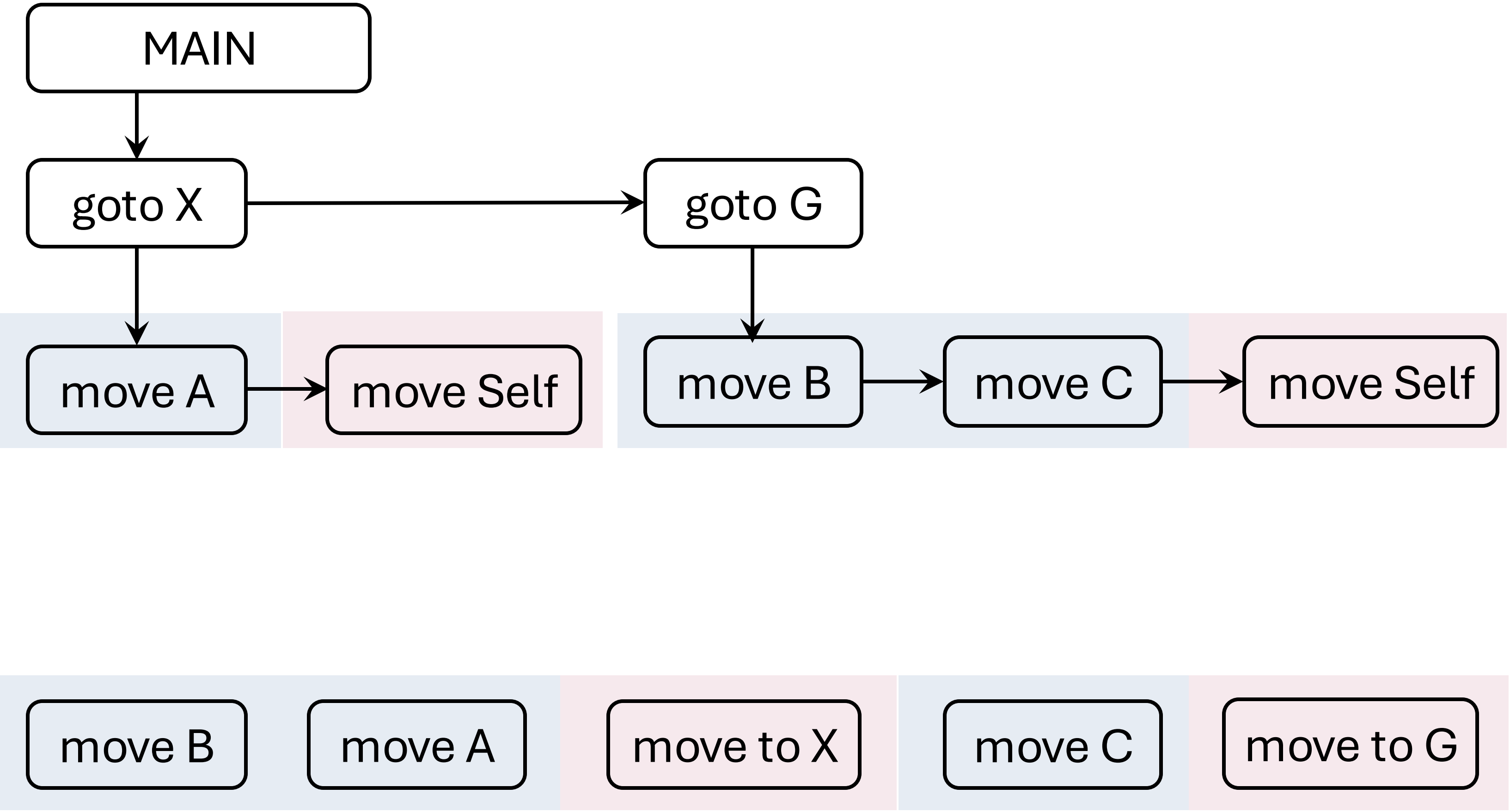
move C

move to G



Adding Flexible Promotion

Imperative **+Variables** **+Ordering** **+Promotion**



MAIN

goto X

goto G

move A

move Self

move B

move C

move Self

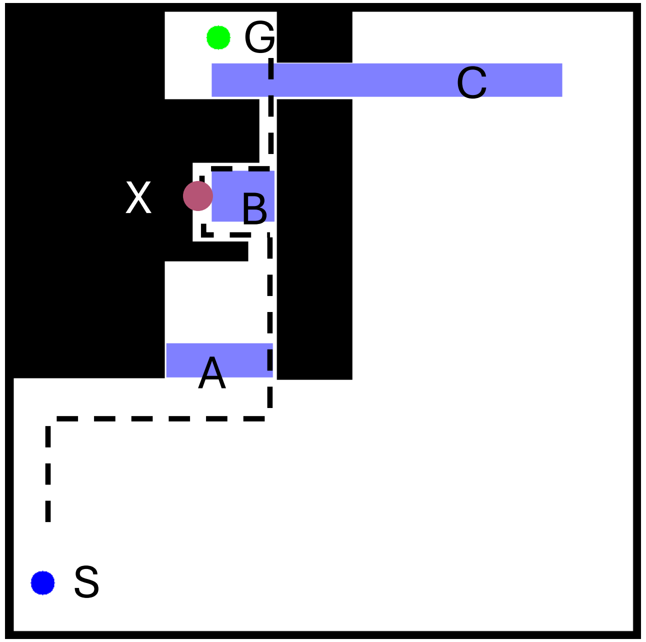
move B

move A

move to X

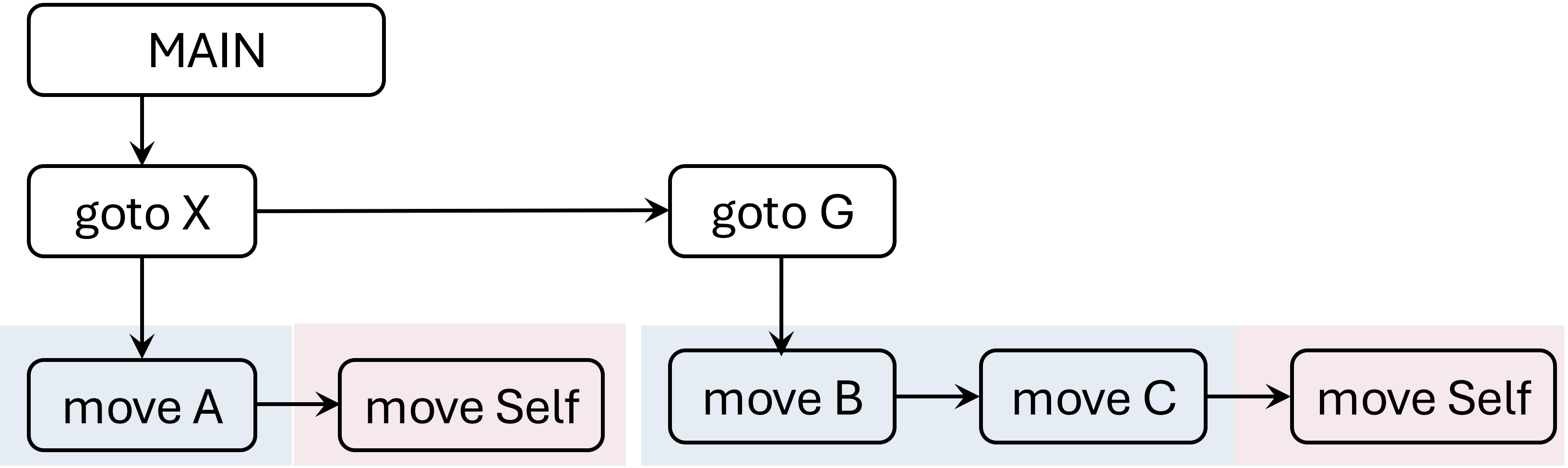
move C

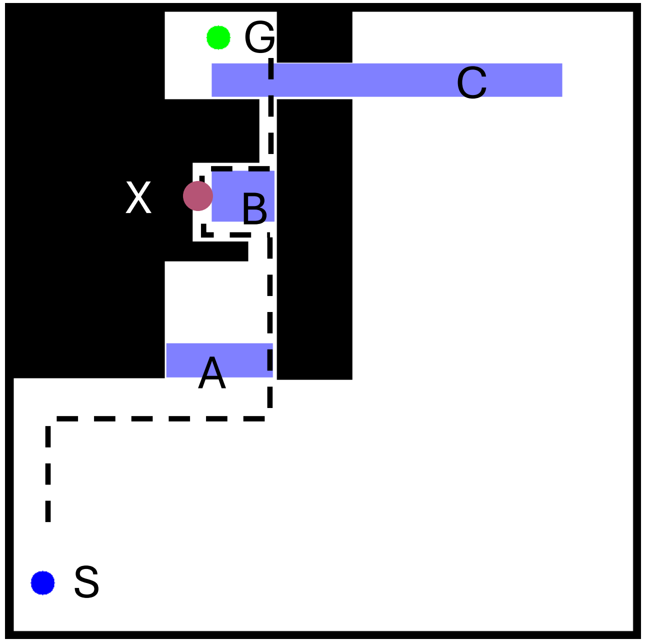
move to G

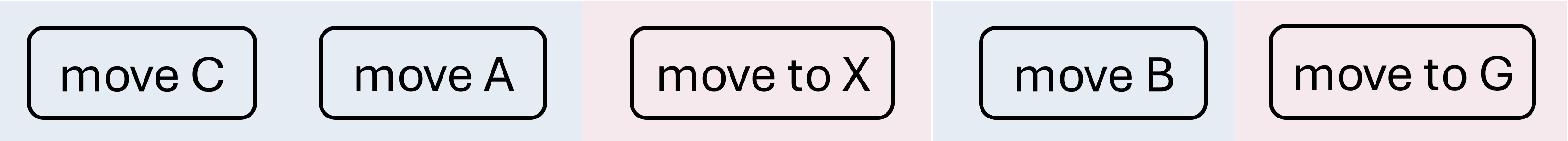


Adding Flexible Promotion

Imperative **+Variables** **+Ordering** **+Promotion**

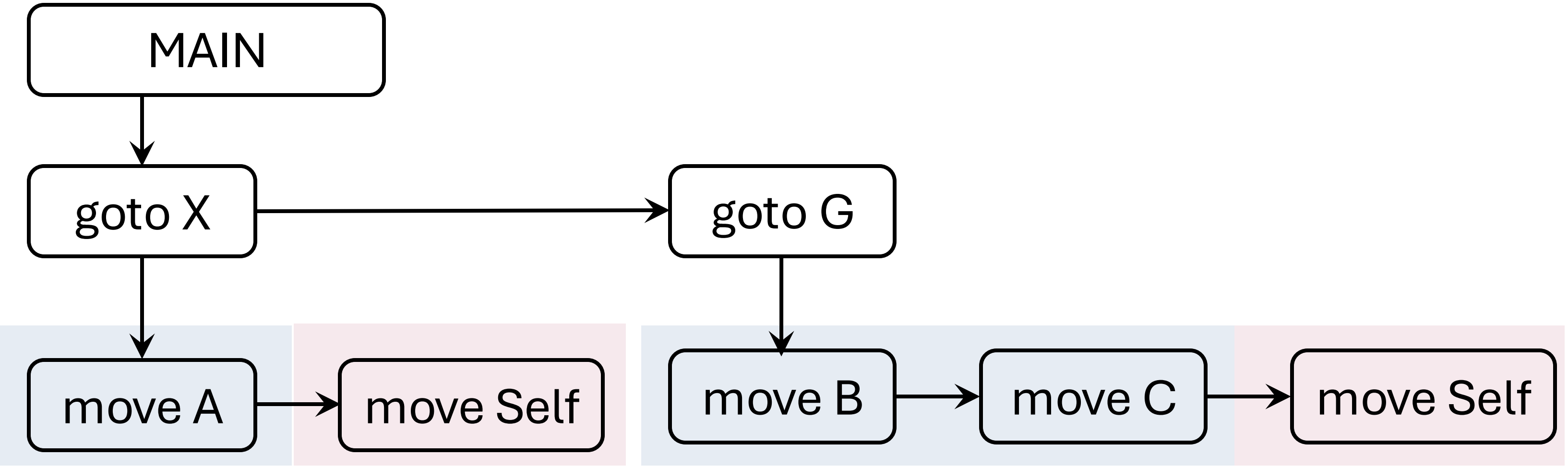


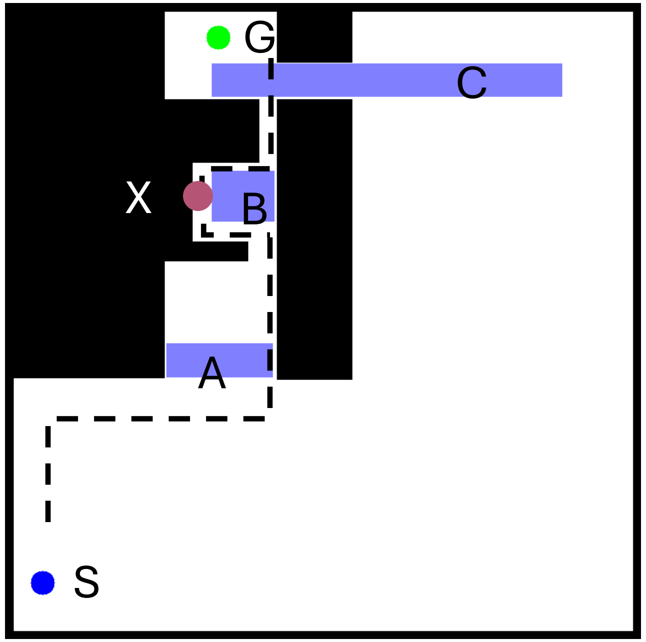


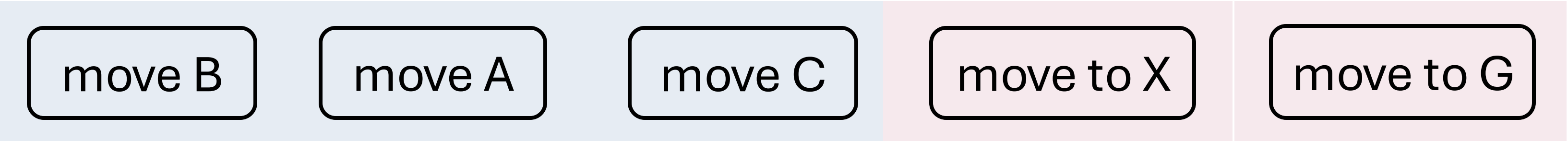


Adding Flexible Promotion

Imperative **+Variables** **+Ordering** **+Promotion**

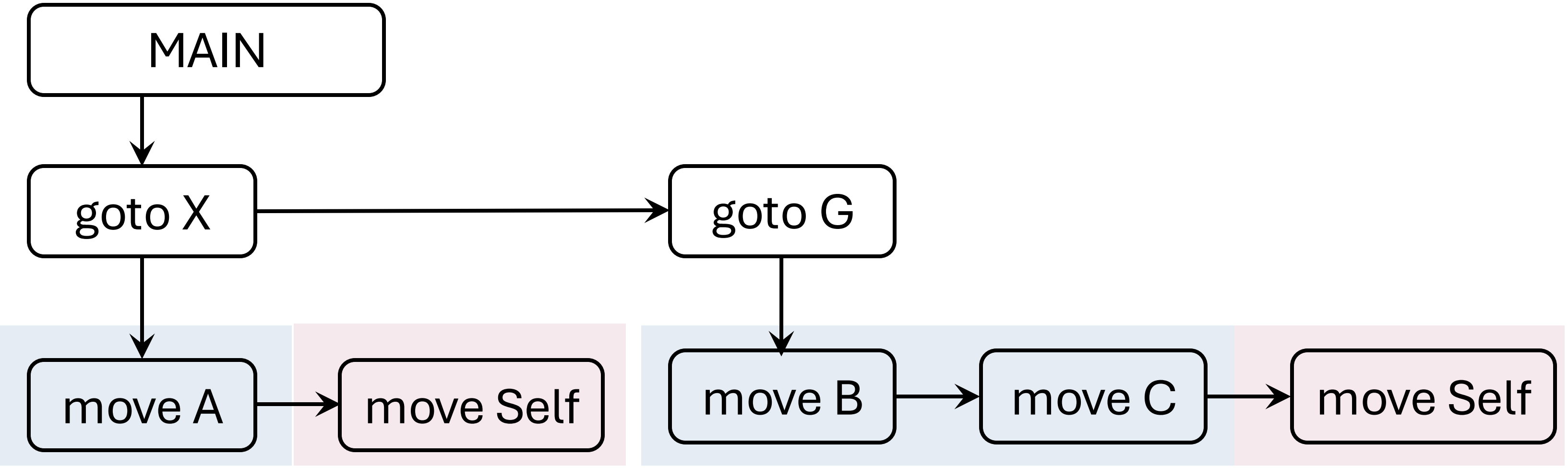


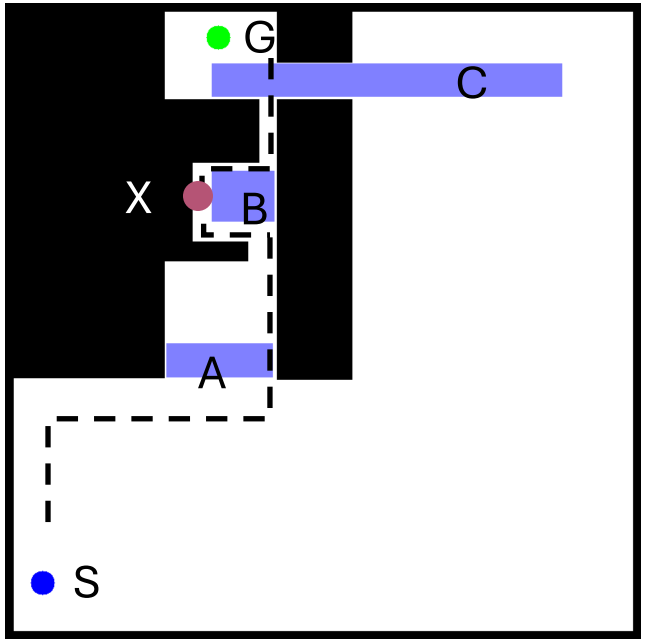


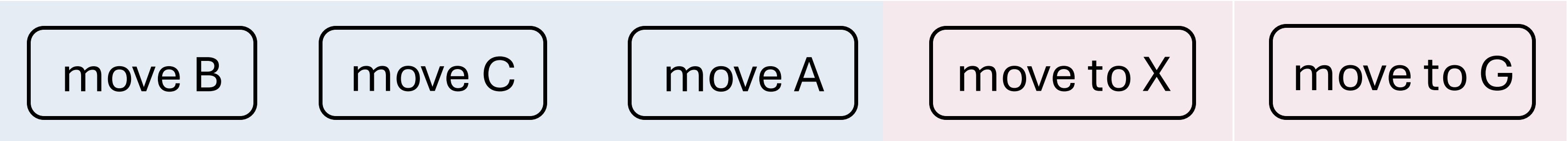


Adding Flexible Promotion

Imperative **+Variables** **+Ordering** **+Promotion**





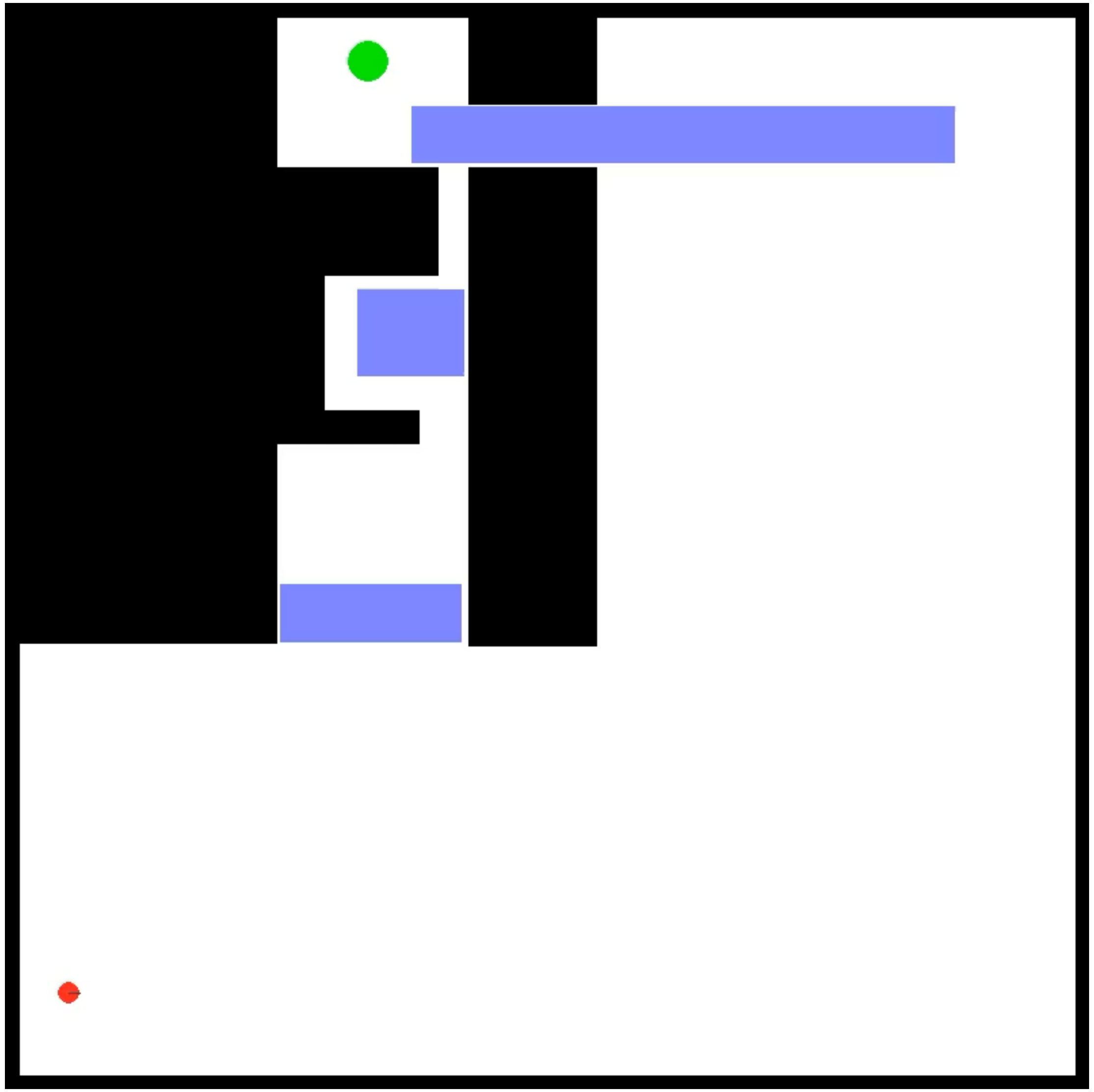


Adding Flexible Promotion

Imperative ● **+Variables** ● **+Ordering** ● **+Promotion** ●

```
global_goal: agent_pos() == (270, 50)

behavior goto_v3(G: vector):
  goal: agent_pos() == G
  body:
    bind waypoint: vector
    achieve agent_pos() == waypoint
    bind path = find_path(agent_pos(), G)
    promotable unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)
```



The Spectrum Between Imperative and Declarative

Imperative **+Variables** **+Ordering** **+Promotion**

Insight 1: Behaviors = Generators of “non-deterministic subroutine calls”
+ Verifies using causal models: pre- and post-conditions

Insight 2: Declarative = Imperative + Variable + Ordering + Promotion

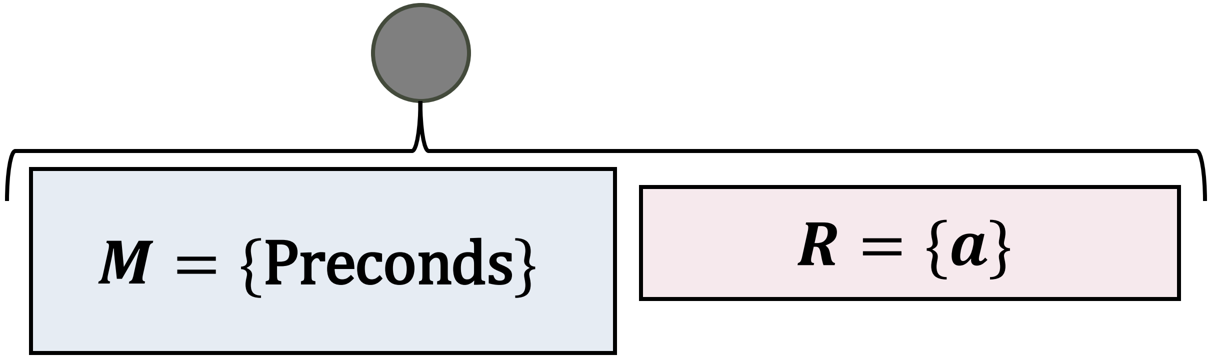
Specifically, if you only denote:

- the variables needed
- the preconditions they need to satisfy
- no ordering information about how preconditions should be achieved

You get full declarative modeling

Reformulate Classical Formulations by “Language Feats.”

	+Variables Binding	+Ordering	+Promotion
PDDL	❌ Discrete Only	❗ Always	❗ Always



$\boldsymbol{M} = \{\textbf{Preconds}\}$

$\boldsymbol{R} = \{\boldsymbol{a}\}$

❌ Do not support

❗ Support, but you can’t configure

✅ Support, and configurable

Reformulate Classical Formulations by “Language Feats.”

	+Variable Binding	+Ordering	+Promotion
PDDL	❌ Discrete Only	❗ Always	❗ Always
HTN/HGN	❌ Discrete Only	❌	❗ Always
GOLOG	❌ Discrete Only	✅	❗ Manual Interleaving
PDDLStream	✅	❗ Always	❗ Always
CDL (Ours)	✅	✅	✅

❌ Do not support ❗ Support, but you can’t configure ✅ Support, and configurable

Reformulate Classical Formulations by “Language Feats.”

	+Variable Binding	+Ordering	+Promotion
PDDL	✗ Discrete Only	! Always	! Always
HTN/HGN			
GOLOG			
PDDLStream			
CDL (Ours)	✓	✓	✓

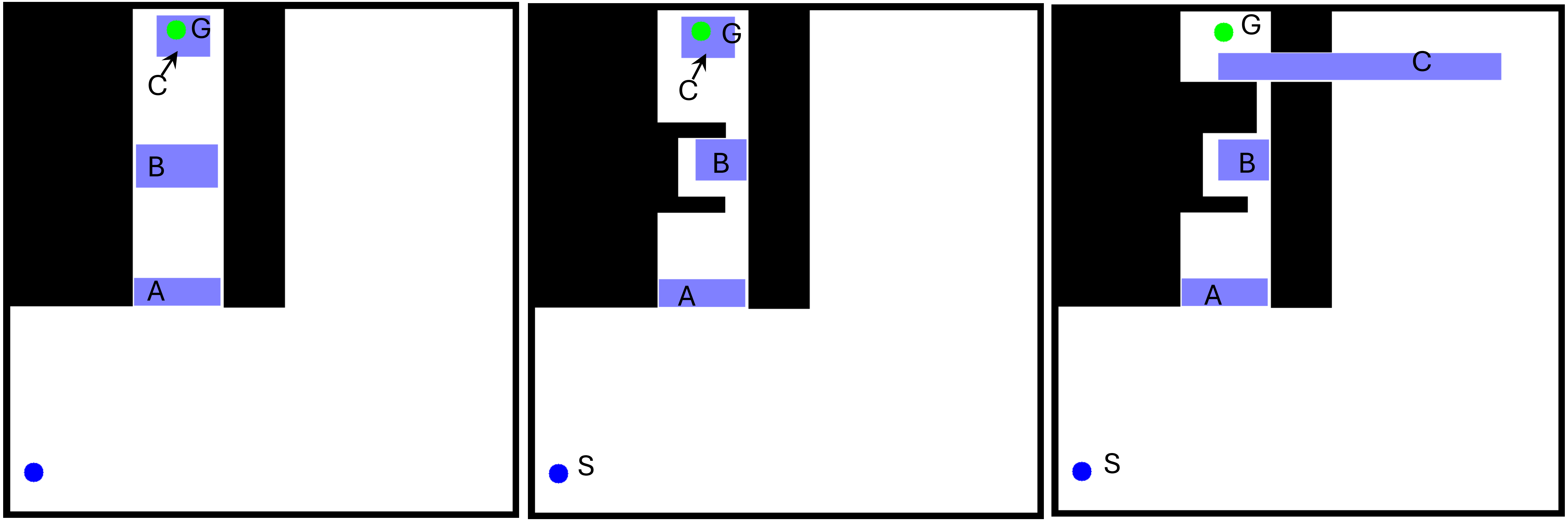
The Crow Planner:

Consumes the flexible representations

- Sound
- Probabilistically resolution complete

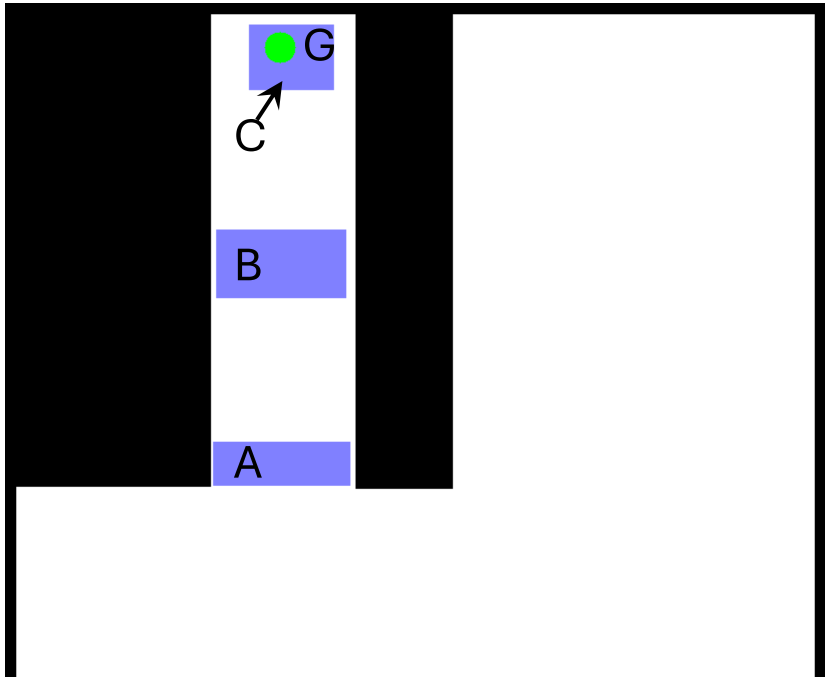
✗ Do not support ! Support, but you can’t configure ✓ Support, and configurable

Application: Context-Specific Solution Strategies

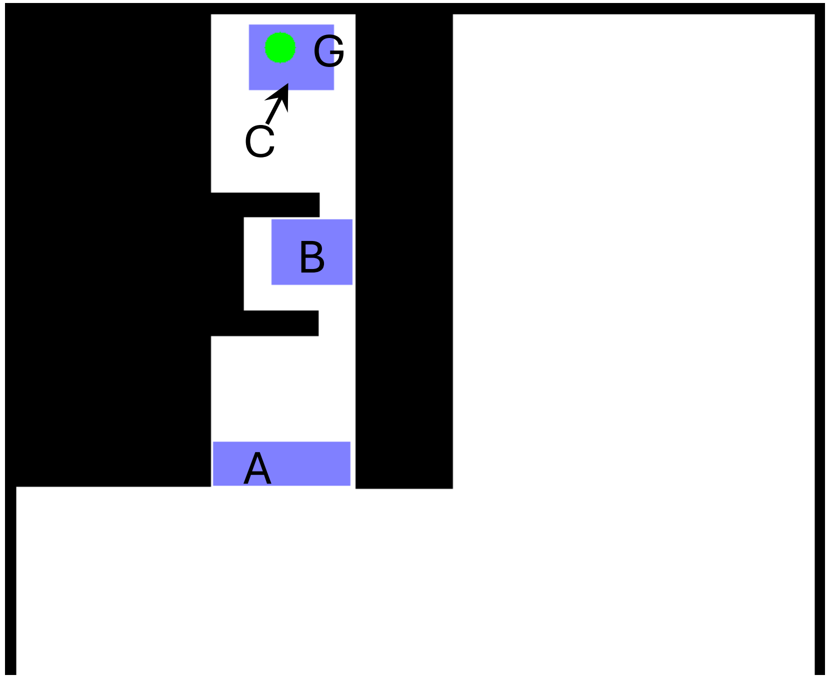


Mid Level: Closely related to the *LP1* class in Stilman and Kuffner 2005, *“disconnected spaces can be connected by moving a single obstacle”*

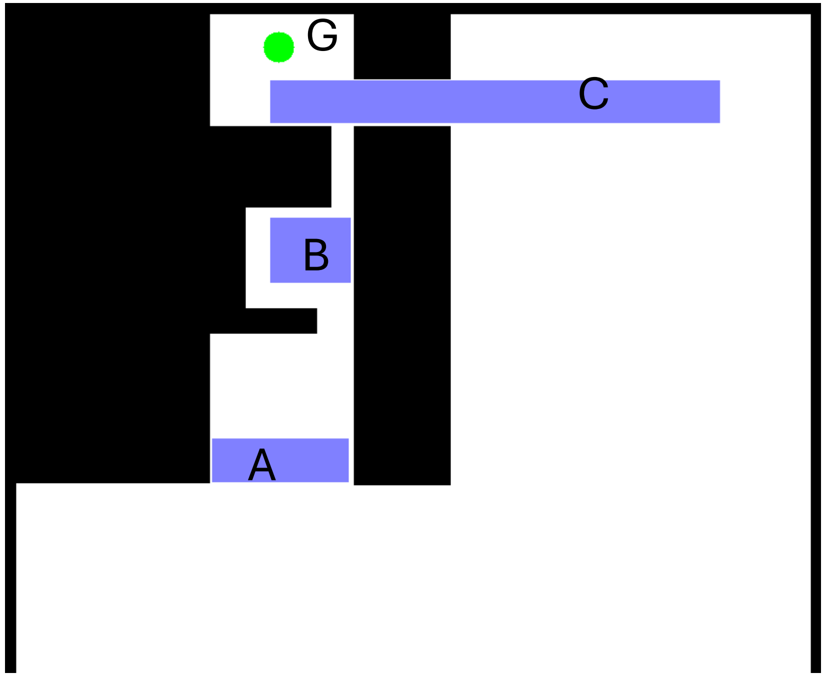
Application: Context-Specific Solution Strategies



```
behavior goto_v1(G: vector):
  goal: agent_pos() == G
  body:
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)
```

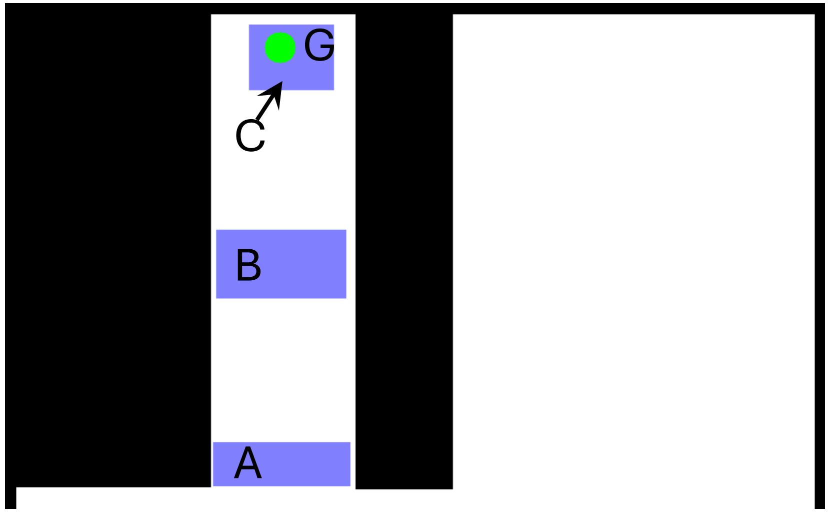


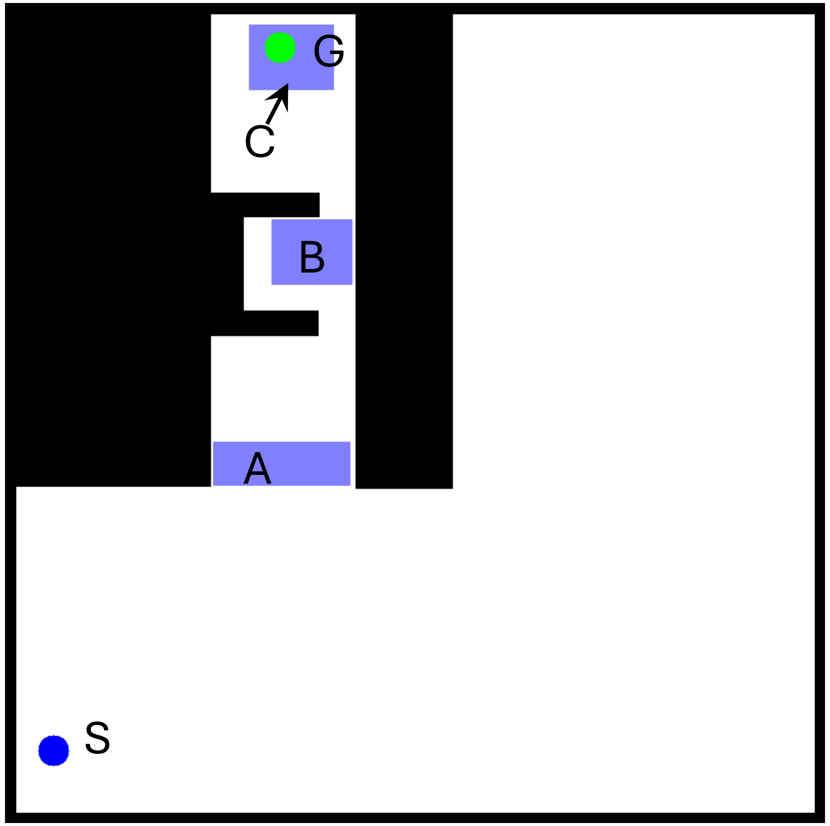
```
behavior goto_v2(G: vector):
  goal: agent_pos() == G
  body:
    bind waypoint: vector
    achieve agent_pos() == waypoint
    bind path = find_path(agent_pos(), G)
    unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)
```

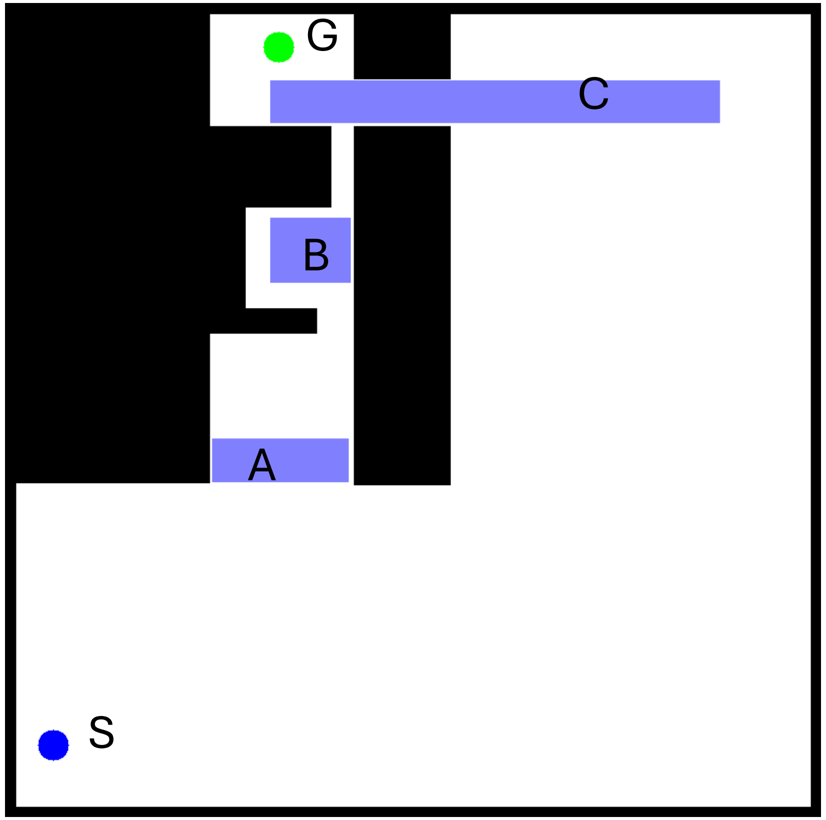


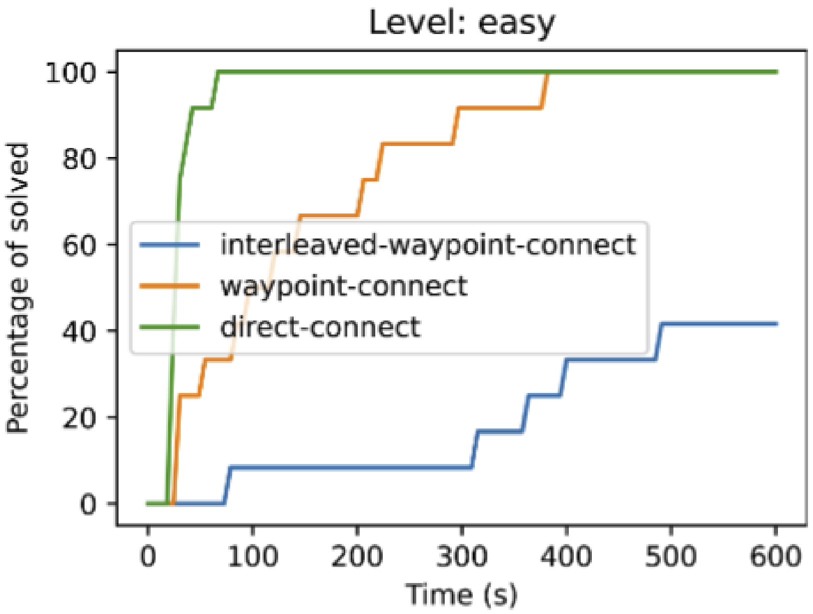
```
behavior goto_v3(G: vector):
  goal: agent_pos() == G
  body:
    bind waypoint: vector
    achieve agent_pos() == waypoint
    bind path = find_path(agent_pos(), G)
    promotable unordered:
      achieve not_blocking(A, path)
      achieve not_blocking(B, path)
      achieve not_blocking(C, path)
    do move_path(path)
```

Context-Specific Strategies Improves Efficiency

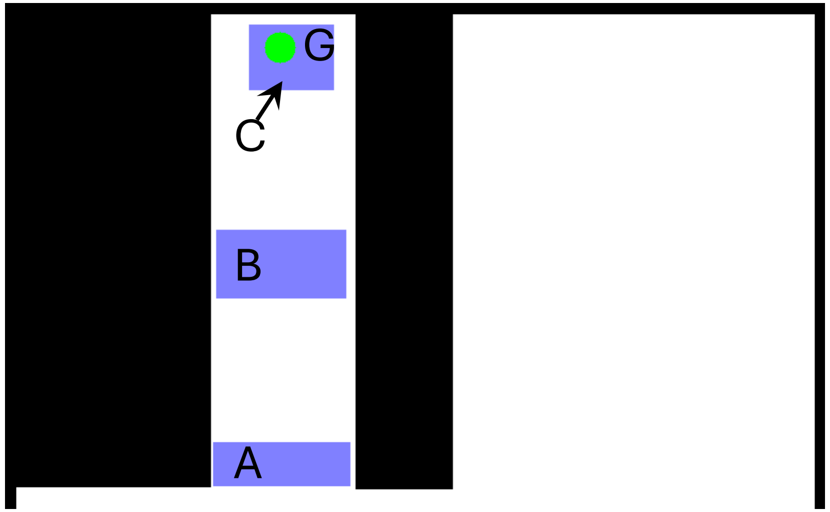


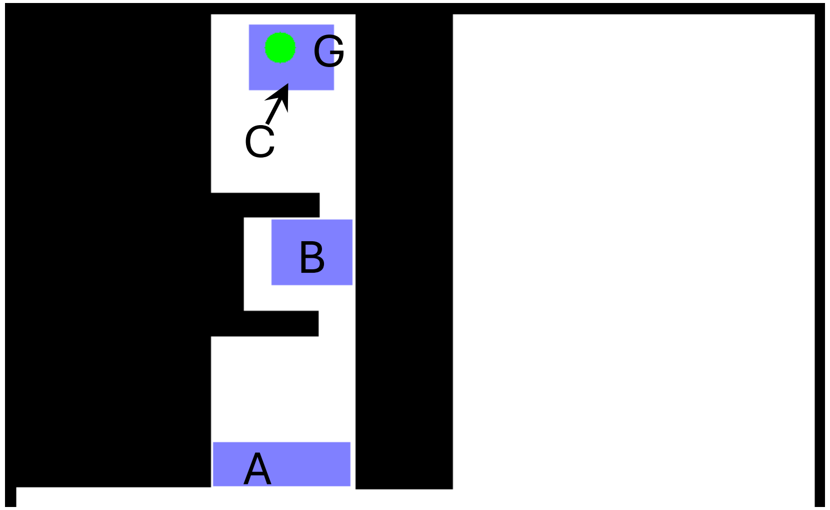


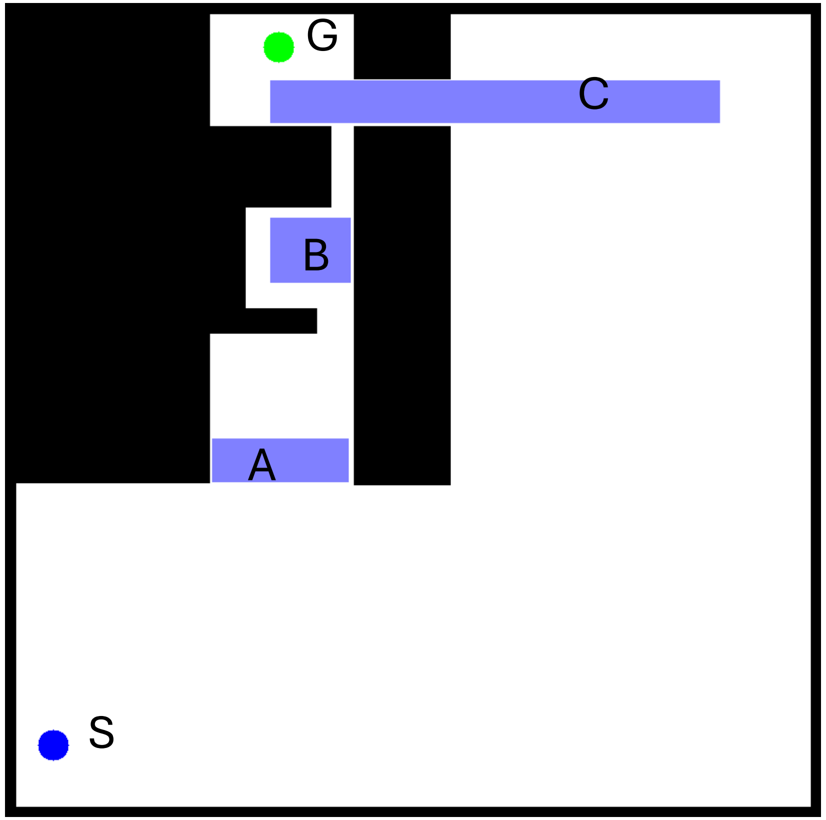


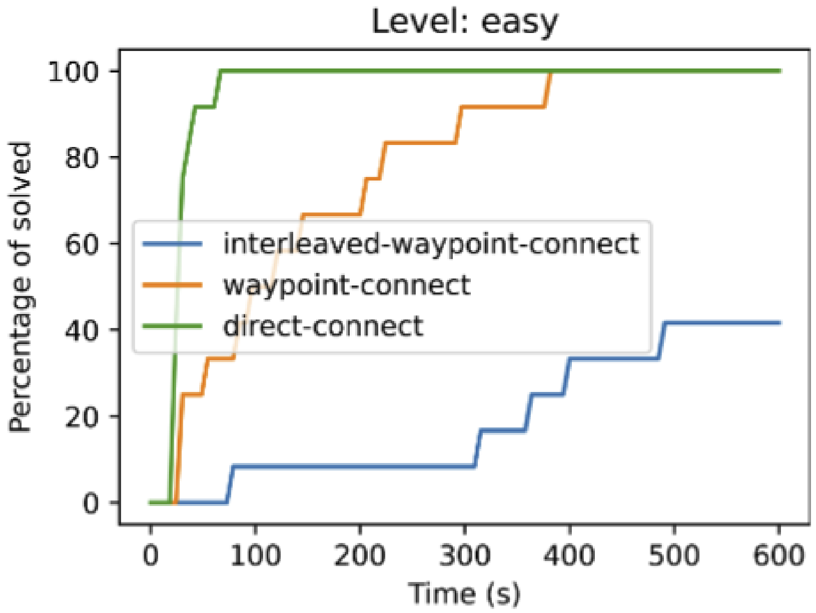


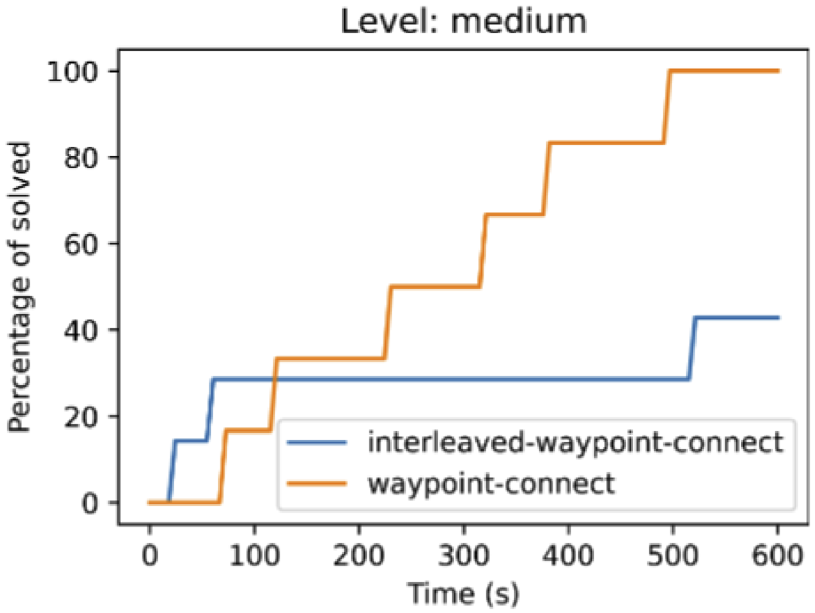
Context-Specific Strategies Improves Efficiency



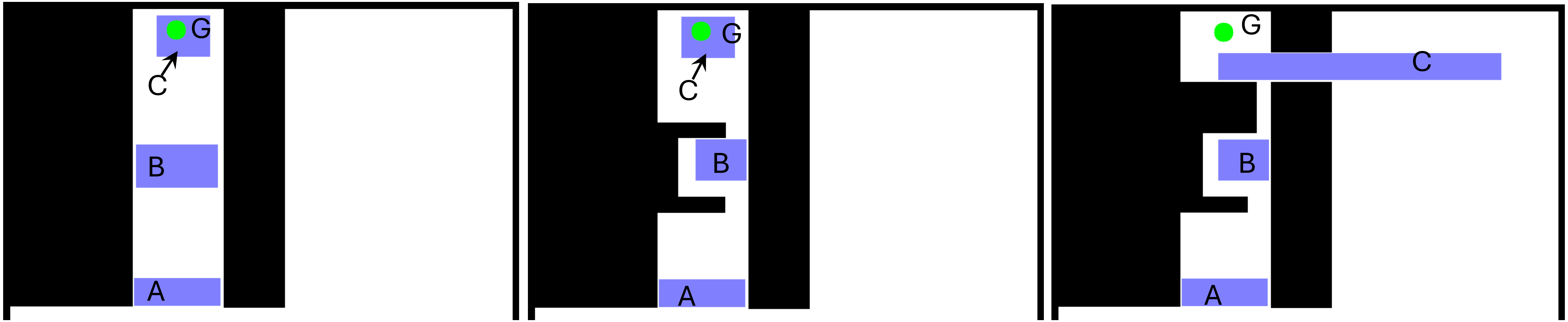


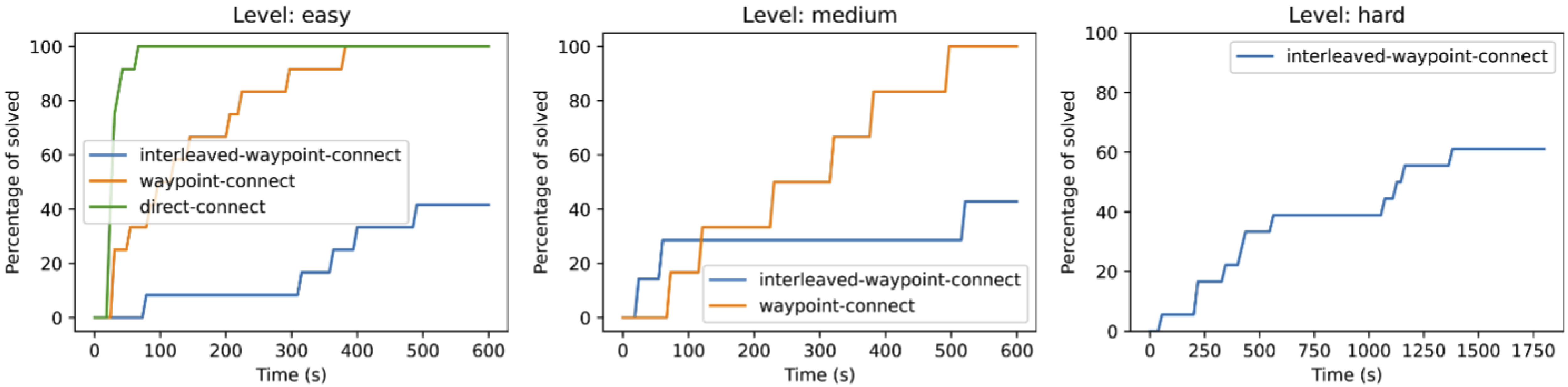






Context-Specific Strategies Improves Efficiency





Theory: Planning Complexity of Problems

Promotable Section M

Serialized Section R

Theorem (very informally): under serializability assumptions over R, the planning complexity is bounded by $n^{O(k)}$, where n is the number of objects, k is the maximum number of subgoals that would accumulate in M

Intuition: k defines how easy it is to "serialize" a problem

- **NAMO:** k is the number of obstacles that have "dependencies"

Closely Related to "Width" in Symbolic Planning and Neural Network Expressivity

Lipovetzky and Geffner. 2012. "Width and serialization of classical planning problems"

Mao et al. 2023. "What Planning Problem Can A Relational Neural Network Solve?"

Dirty Laundry

Theory

- The bound is not tight because it treats all objects “uniformly”
- Ultimately, what we really want to is to identify the “kernel” of the problem

Practice

- Although we support description of different solution strategies compactly,
- we do not know which one to apply
- Actually, this can be as hard as solving the original problem

Conclusion

Principle: Using program semantics to characterize flexibilities in behaviors

We provide a new framework for “how to plan more efficiently”

- **Theory:** characterize the hardness of a problem
- **Practice:** a framework for mix-and-match representations

Next: learning how to reason more efficiently

- learning to select the best strategy in context
- learning to form new strategies, by reasoning about different types of flexibilities