Planning in Factored Spaces

Tom Silver

Robot Planning Meets Machine Learning

Princeton University

Fall 2025

Recap and Preview

Previously:

- Planning in finite "tabular" state and action spaces
- Careful treatment of uncertainty in transitions and observations
- Offline planning and online planning

Now:

- Planning in finite "factored" state and action spaces
- No more uncertainty
- Online planning only

Our focus turns to leveraging *structure in the problem space*

Later:

- Planning in continuous state and action spaces

Classical Planning Problem Setting

A *classical planning problem* is:

1. A finite state space $\mathcal{S}$
2. A finite action space $\mathcal{A}$
3. An initiable action function $I: \mathcal{S} \times \mathcal{A} \rightarrow \{\mathrm{T}, \mathrm{F}\}$
4. A transition function $F: \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$
5. A cost function $C: \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}_{\geq 0}$
6. An initial state $s_0 \in \mathcal{S}$
7. A goal function $G: \mathcal{S} \rightarrow \{\mathrm{T}, \mathrm{F}\}$

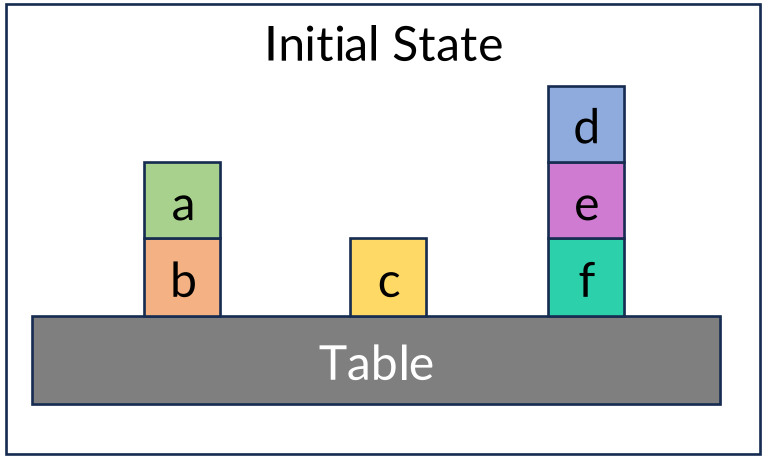
Deterministic! Can be partial

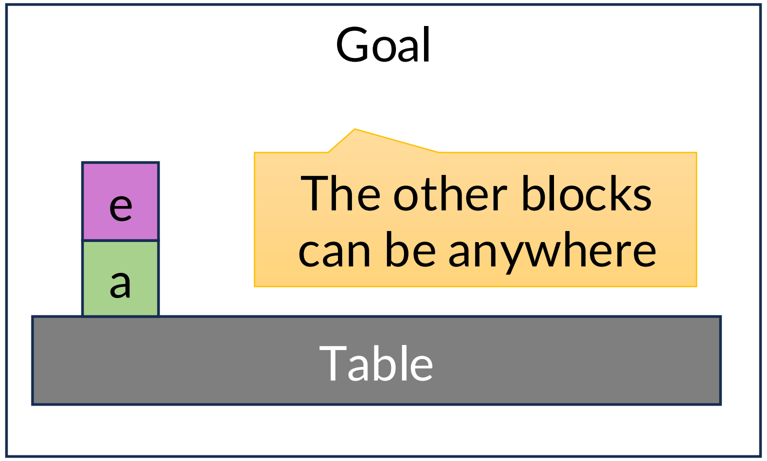
Lower better. Could do rewards instead; just a convention.

Equivalent to a set of states

Example: Blocks World

- **States:** each block is either on the table or on some other block
- **Actions:** picking or placing a block
- **Initiation:** can only pick and place on “clear” blocks
- **Transition function:** as you’d expect
- **Cost function:** always 1
- **Initial state:** e.g., see right
- **Goal function:** e.g., see right





Definition of a Solution (Plan)

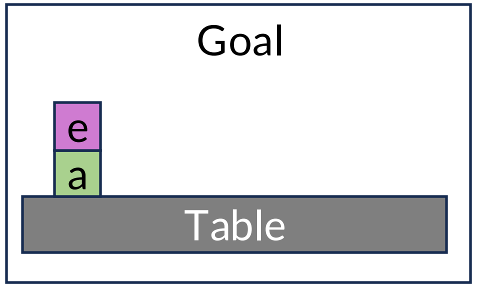
A *solution* ψ to a classical planning problem is a sequence of states $s_0, s_1, \ldots, s_T$ and actions $a_0, a_1, \ldots, a_{T-1}$ such that

1. Each action is initiable: $I(s_t, a_t) = \text{True}$
2. Transitions are valid: $T(s_t, a_t) = \text{s}_{t+1}$
3. The goal is achieved: $G(s_T) = \text{True}$

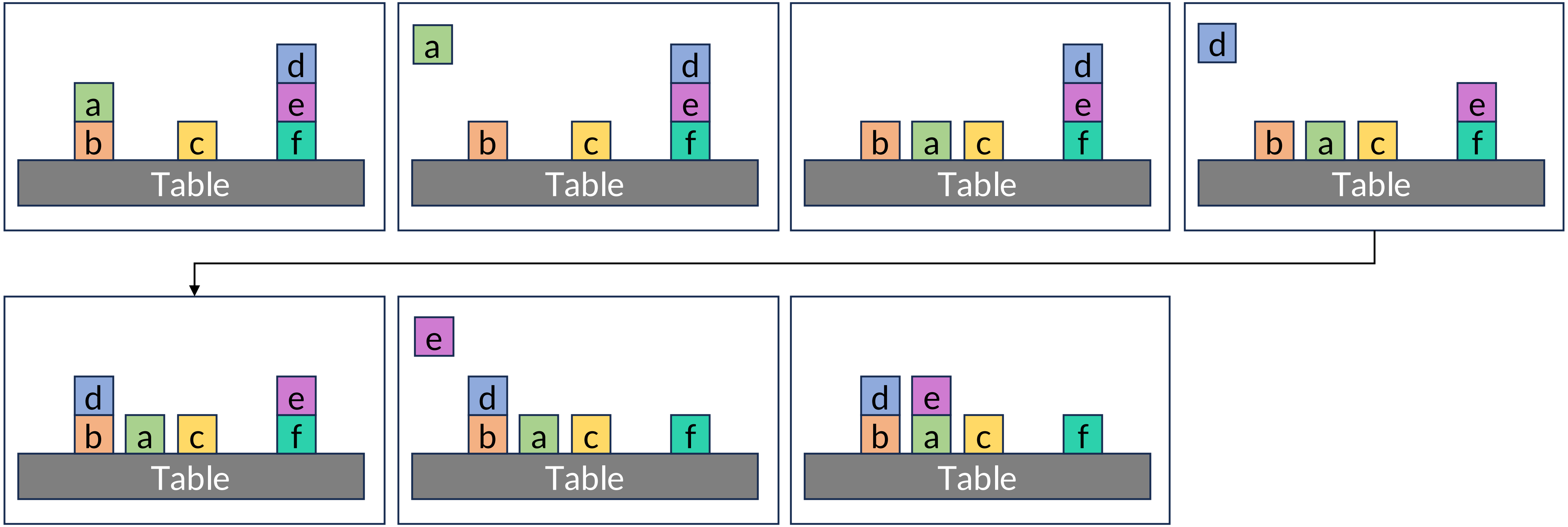
The *cost of a solution* ψ is $C(\psi) \triangleq \sum_t C(s_t, a_t, s_{t+1})$

A solution ψ^* is *optimal* if it minimizes costs: $C(\psi^*) = \min_{\psi} C(\psi)$

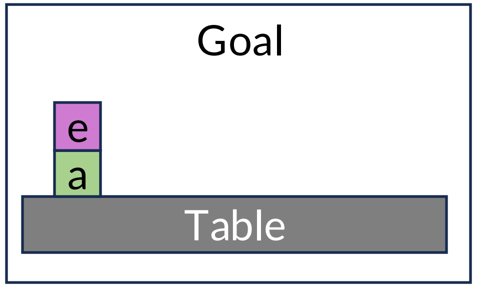
Example: Blocks World



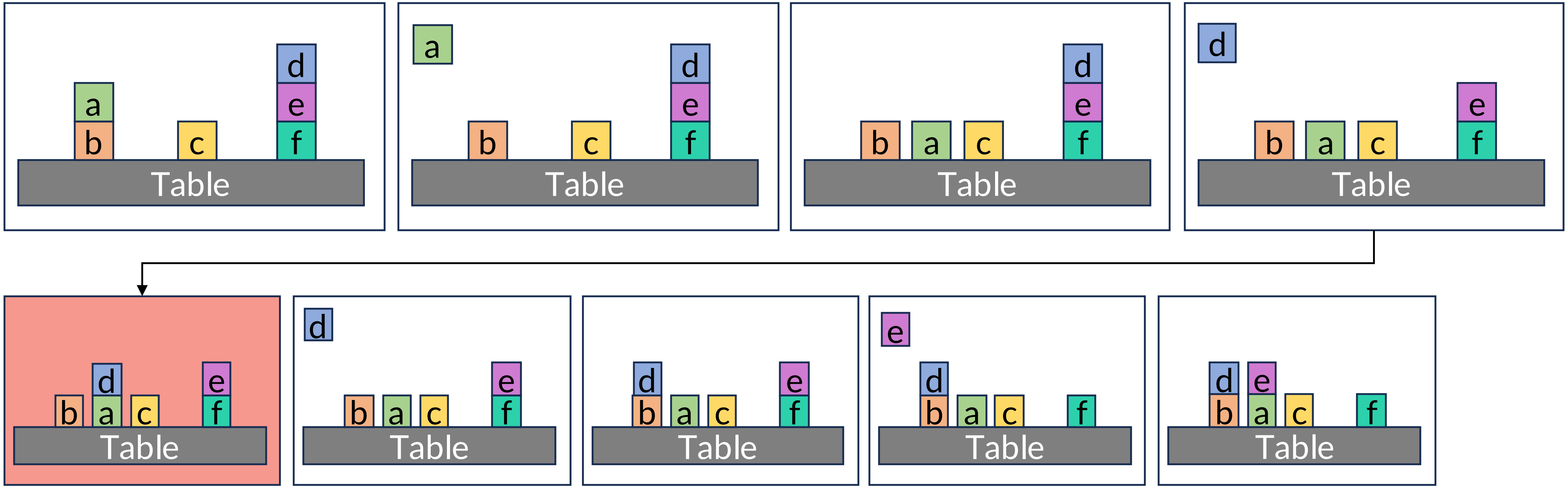
An optimal plan



Example: Blocks World



A suboptimal plan



A Stupidest Possible Algorithm

Randomly sample applicable actions until the goal is reached.

A Stupidest Possible Algorithm

Randomly sample applicable actions until the goal is reached.

What is this planner?

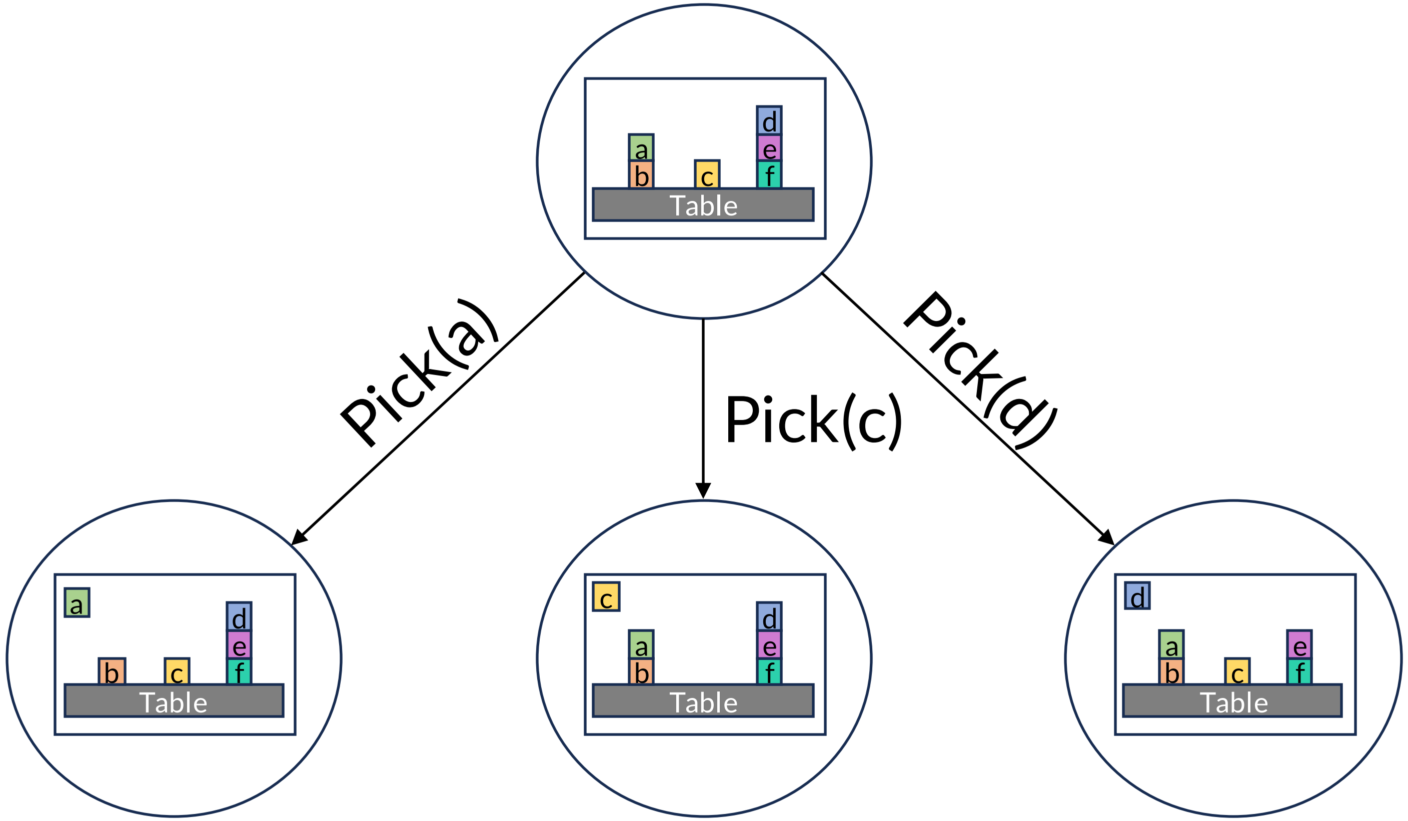
Definitions:

A planner is *sound* if its output is guaranteed to be a solution.

A planner is *complete* if it is guaranteed to return an output eventually.

A sound planner is *optimal* if its output is guaranteed to be optimal.
Otherwise, the planner is *satisficing*.

A Better Approach: Graph Search

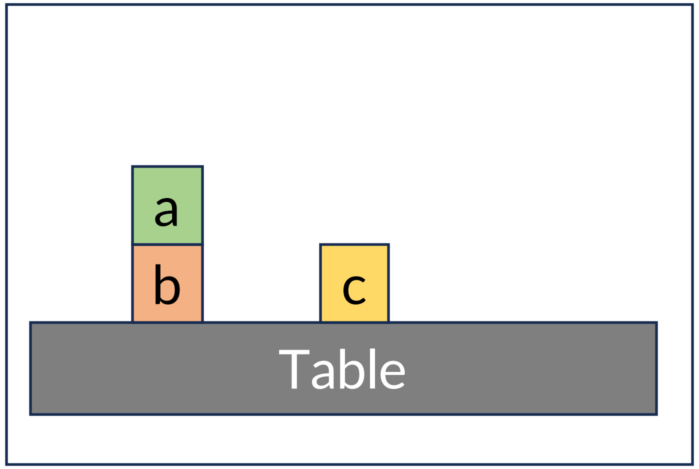


Graph Search

GraphSearch($s_0, \mathcal{A}, I, F, C, G$)

```
// priority queue of nodes
initialize queue = []
root = Node($s_0$, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {$s_0 \mapsto 0$}
while queue is not empty
    pop node from queue
    $s, c$ = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < $c$ : continue
    if $G(s)$ : return node.extractPlan()
    for $a \in \mathcal{A}$ s.t. $I(s, a)$
        $s' = F(s, a)$
        $c' = c + C(s, a, s')$
        if bestPathCost[s’] < $c'$ : continue
        bestPathCost[s’] = $c'$
        child = Node($s'$, $c'$, parent=node)
        push child onto queue with PRIORITY(child)
```

Goal: b on a



GraphSearch(s_0, $\mathcal{A}$, I, F, C, G)

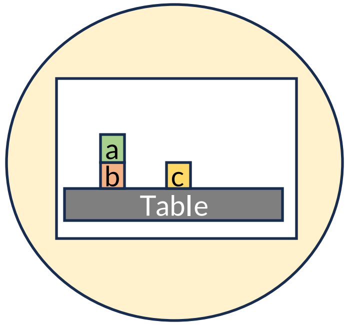
// priority queue of nodes
initialize queue $=$ []
root $=$ Node(s_0, 0, parent=null)
// priority differs between algorithms
push root onto queue with priority(root)
// state $\rightarrow$ best known path cost
initialize bestPathCost $= \{s_0 \mapsto 0\}$
while queue is not empty
  pop node from queue
  $s, c =$ node.state, node.pathCost
  // skip if we already found a better path
  if bestPathCost[s] $< c$: **continue**
  if $G(s)$: **return** node.extractPlan()
  for $a \in \mathcal{A}$ s.t. $I(s, a)$
    $s' = F(s, a)$
    $c' = c + C(s, a, s')$
    if bestPathCost[s'] $< c'$: **continue**
    bestPathCost[s'] $= c'$
    child $=$ Node(s', c', parent=node)
    push child onto queue with priority(child)

queue

GraphSearch(s_0, $\mathcal{A}$, I, F, C, G)

```
// priority queue of nodes
initialize queue = []
root = Node(s_0, 0, parent=null)
// priority differs between algorithms
push root onto queue with priority(root)
// state → best known path cost
initialize bestPathCost = {s_0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ 𝒜 s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = Node(s′, c′, parent=node)
        push child onto queue with priority(child)
```

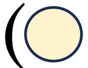
queue



GraphSearch(s_0, $\mathcal{A}$, I, F, C, G)

```
// priority queue of nodes
initialize queue = []
root = Node(s0, 0, parent=null)
// priority differs between algorithms
push root onto queue with priority(root)
// state → best known path cost
initialize bestPathCost = {s0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = Node(s′, c′, parent=node)
        push child onto queue with priority(child)
```

queue

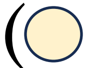
(, 0)

For now, let's say priority = path cost

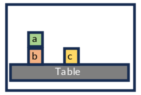
GRAPHSEARCH(s_0, $\mathcal{A}$, I, F, C, G)

```
// priority queue of nodes
initialize queue = []
root = NODE(s_0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {s_0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s' = F(s, a)
        c' = c + C(s, a, s')
        if bestPathCost[s'] < c' : continue
        bestPathCost[s'] = c'
        child = NODE(s', c', parent=node)
        push child onto queue with PRIORITY(child)
```

queue

(, 0)

bestPathCost



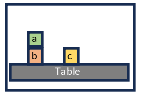
$\rightarrow 0$

GraphSearch(s_0, $\mathcal{A}$, I, F, C, G)

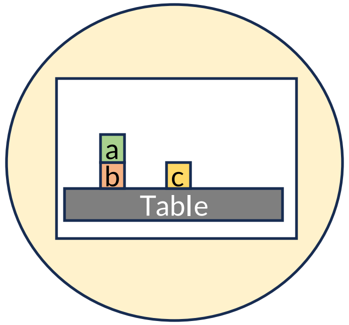
// priority queue of nodes
initialize queue $=$ []
root $=$ Node(s_0, 0, parent=null)
// priority differs between algorithms
push root onto queue with priority(root)
// state $\rightarrow$ best known path cost
initialize bestPathCost $= \{s_0 \mapsto 0\}$
while queue is not empty
    pop node from queue
    $s, c =$ node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] $< c$: **continue**
    if $G(s)$: **return** node.extractPlan()
    for $a \in \mathcal{A}$ s.t. $I(s, a)$
        $s' = F(s, a)$
        $c' = c + C(s, a, s')$
        if bestPathCost[s’] $< c'$: **continue**
        bestPathCost[s’] $= c'$
        child $=$ Node(s', c', parent=node)
        push child onto queue with priority(child)

queue

bestPathCost



$\to 0$

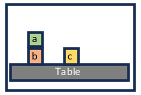


GraphSearch(s_0, $\mathcal{A}$, I, F, C, G)

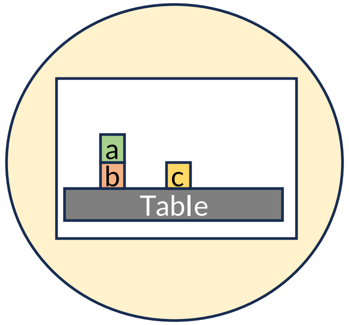
```
// priority queue of nodes
initialize queue = []
root = Node(s0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {s0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = Node(s′, c′, parent=node)
        push child onto queue with PRIORITY(child)
```

queue

bestPathCost



→ 0

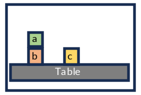


GraphSearch(s_0, $\mathcal{A}$, I, F, C, G)

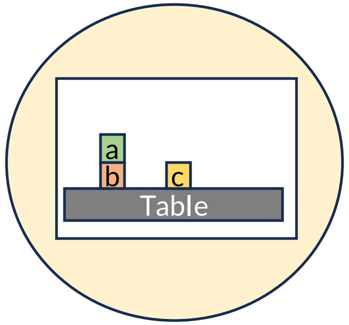
```
// priority queue of nodes
initialize queue = []
root = Node(s0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {s0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = Node(s′, c′, parent=node)
        push child onto queue with PRIORITY(child)
```

queue

bestPathCost



$\rightarrow 0$

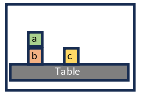


GRAPHSEARCH$(s_0, \mathcal{A}, I, F, C, G)$

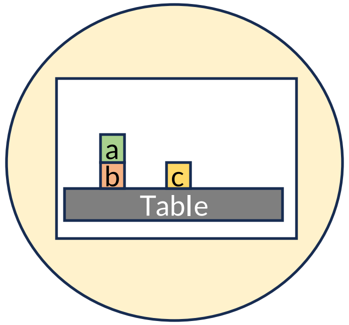
// priority queue of nodes
initialize queue $=$ []
root $=$ NODE$(s_0, 0,$ parent=null$)$
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state $\rightarrow$ best known path cost
initialize bestPathCost $= \{s_0 \mapsto 0\}$
while queue is not empty
    pop node from queue
    $s, c =$ node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] $< c$: **continue**
    if $G(s)$: **return** node.extractPlan()
    for $a \in \mathcal{A}$ s.t. $I(s, a)$
        $s' = F(s, a)$
        $c' = c + C(s, a, s')$
        if bestPathCost[s'] $< c'$: **continue**
        bestPathCost[s'] $= c'$
        child $=$ NODE$(s', c',$ parent=node$)$
        push child onto queue with PRIORITY(child)

queue

bestPathCost



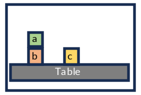
→ 0



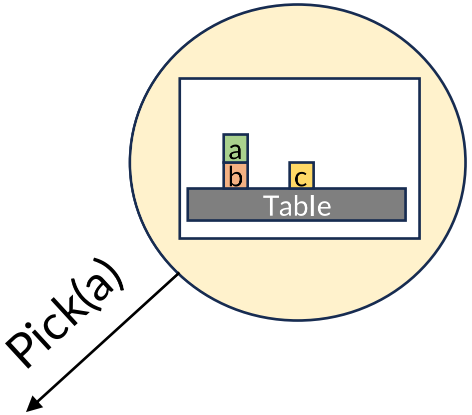

```
GraphSearch(s0, A, I, F, C, G)
// priority queue of nodes
initialize queue = []
root = Node(s0, 0, parent=null)
// priority differs between algorithms
push root onto queue with priority(root)
// state → best known path cost
initialize bestPathCost = {s0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = Node(s′, c′, parent=node)
        push child onto queue with priority(child)
```

queue

bestPathCost



$\rightarrow 0$

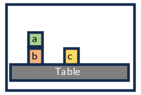


GRAPHSEARCH$(s_0, \mathcal{A}, I, F, C, G)$

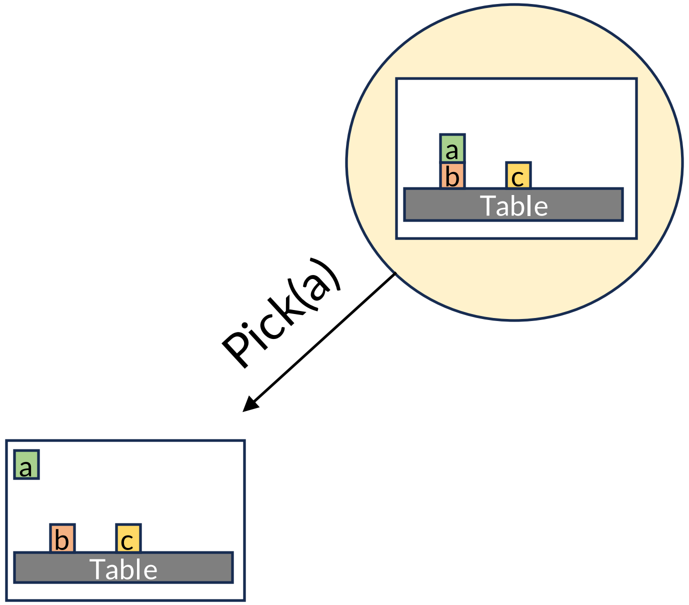
```
// priority queue of nodes
initialize queue = []
root = NODE(s_0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {s_0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = NODE(s′, c′, parent=node)
        push child onto queue with PRIORITY(child)
```

queue

bestPathCost



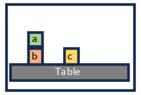
$\rightarrow 0$



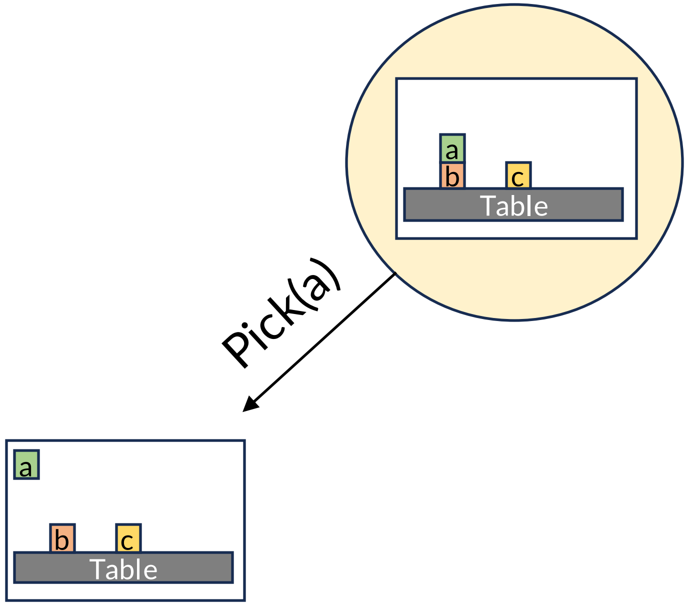

```
GraphSearch(s0, A, I, F, C, G)
// priority queue of nodes
initialize queue = []
root = Node(s0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {s0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = Node(s′, c′, parent=node)
        push child onto queue with PRIORITY(child)
```

queue

bestPathCost



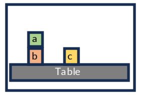
$\rightarrow 0$

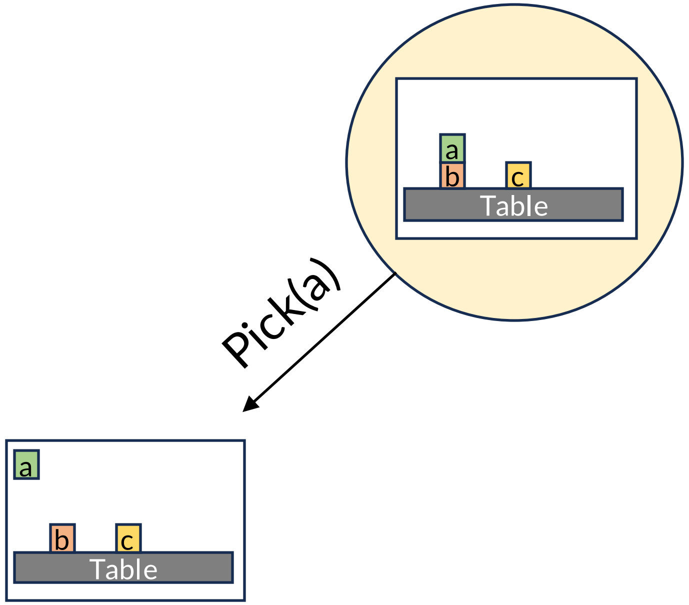



```
GraphSearch(s_0, 𝒜, I, F, C, G)
// priority queue of nodes
initialize queue = []
root = Node(s_0, 0, parent=null)
// priority differs between algorithms
push root onto queue with priority(root)
// state → best known path cost
initialize bestPathCost = {s_0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ 𝒜 s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s'] < c′ : continue
        bestPathCost[s'] = c′
        child = Node(s′, c′, parent=node)
        push child onto queue with priority(child)
```

queue

bestPathCost

 → 0

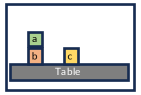


GraphSearch(s_0, $\mathcal{A}$, I, F, C, G)

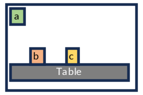
```
// priority queue of nodes
initialize queue = []
root = Node(s_0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {s_0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s' = F(s, a)
        c' = c + C(s, a, s')
        if bestPathCost[s'] < c' : continue
        bestPathCost[s'] = c'
        child = Node(s', c', parent=node)
        push child onto queue with PRIORITY(child)
```

queue

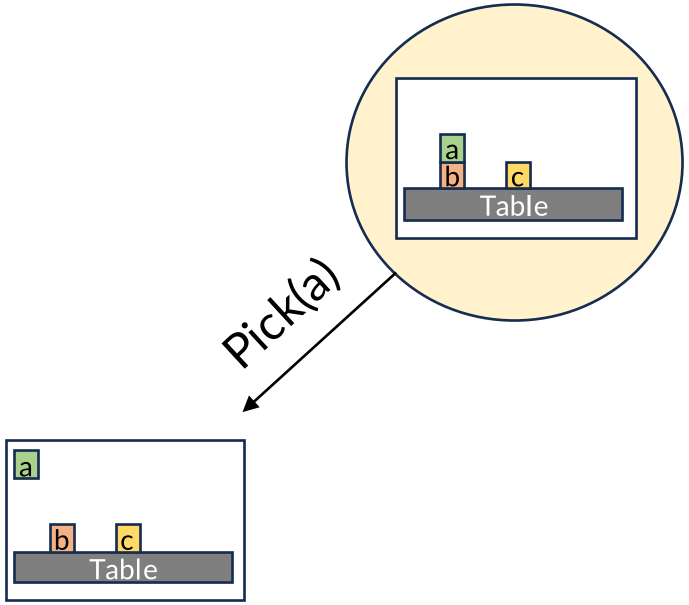
bestPathCost



$\rightarrow 0$



$\rightarrow 1$

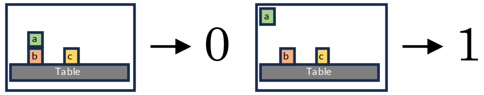


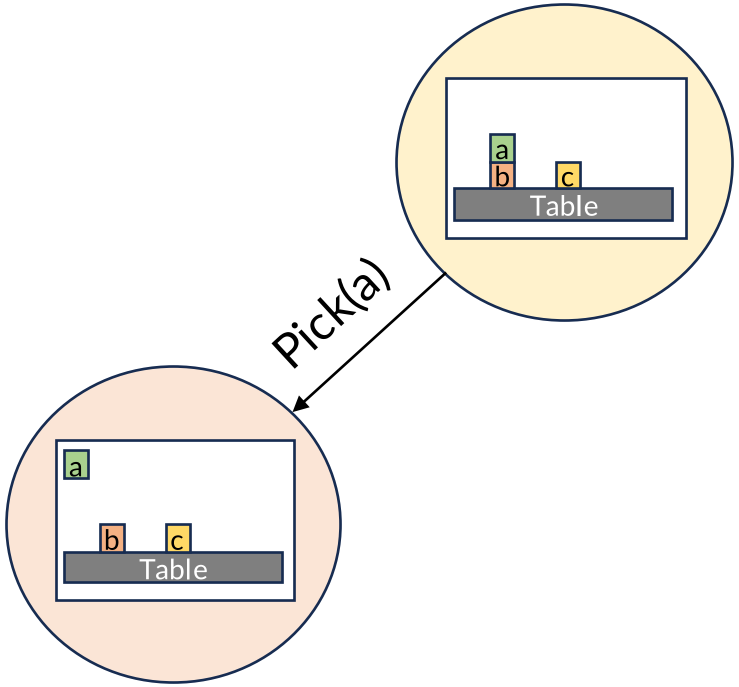
GRAPHSEARCH$(s_0, \mathcal{A}, I, F, C, G)$

// priority queue of nodes
initialize queue $=$ []
root $=$ NODE(s_0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state $\rightarrow$ best known path cost
initialize bestPathCost $= \{s_0 \mapsto 0\}$
while queue is not empty
    pop node from queue
    $s, c =$ node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] $< c$: **continue**
    if $G(s)$: **return** node.extractPlan()
    for $a \in \mathcal{A}$ s.t. $I(s, a)$
        $s' = F(s, a)$
        $c' = c + C(s, a, s')$
        if bestPathCost[s'] $< c'$: **continue**
        $\boxed{\text{bestPathCost[s']} = c'}$
        child $=$ NODE(s', c', parent=node)
        push child onto queue with PRIORITY(child)

queue

bestPathCost

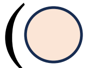




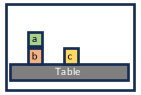
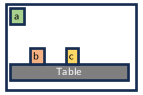
$\textsc{GraphSearch}(s_0, \mathcal{A}, I, F, C, G)$

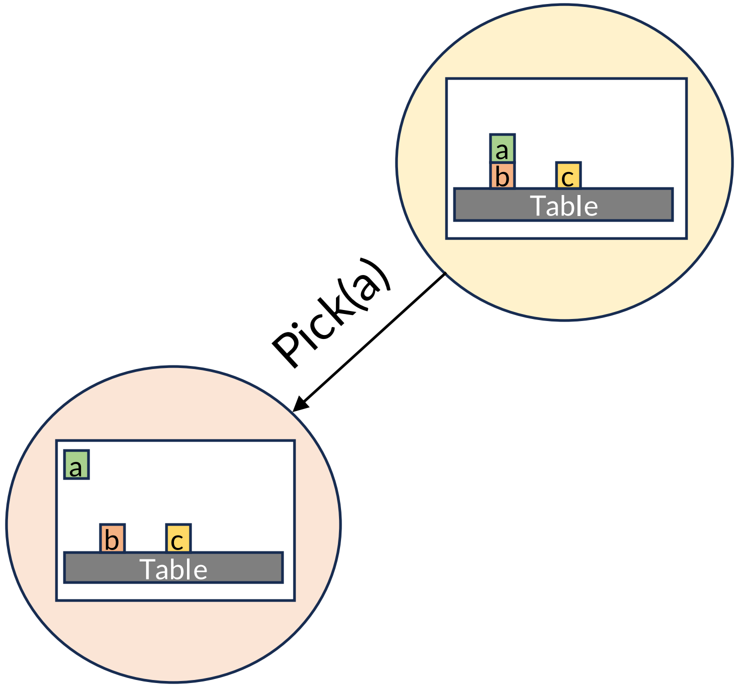
// priority queue of nodes
initialize queue $=$ []
root $=$ $\textsc{Node}(s_0, 0,$ parent=null$)$
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state $\rightarrow$ best known path cost
initialize bestPathCost $= \{s_0 \mapsto 0\}$
while queue is not empty
    pop node from queue
    $s, c =$ node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] $< c$: **continue**
    if $G(s)$: **return** node.extractPlan()
    for $a \in \mathcal{A}$ s.t. $I(s, a)$
        $s' = F(s, a)$
        $c' = c + C(s, a, s')$
        if bestPathCost[s'] $< c'$: **continue**
        bestPathCost[s'] $= c'$
        child $=$ $\textsc{Node}(s', c',$ parent=node$)$
        push child onto queue with PRIORITY(child)

queue

(, 1)

bestPathCost

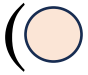
 → 0  → 1



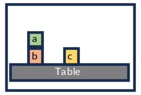
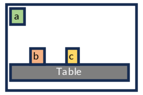
GraphSearch(s_0, $\mathcal{A}$, I, F, C, G)

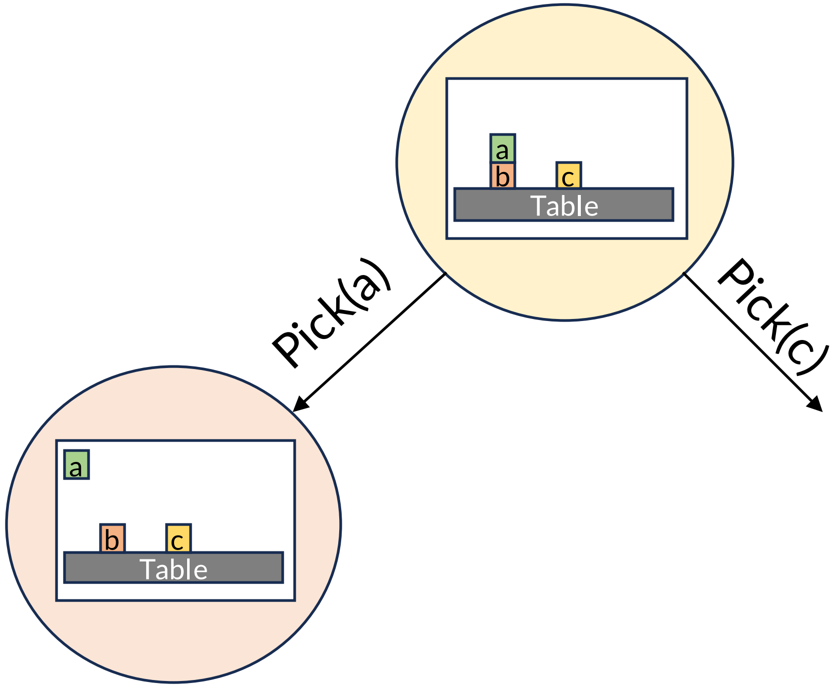
```
// priority queue of nodes
initialize queue = []
root = Node(s0, 0, parent=null)
// priority differs between algorithms
push root onto queue with priority(root)
// state → best known path cost
initialize bestPathCost = {s0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = Node(s′, c′, parent=node)
        push child onto queue with priority(child)
```

queue

(, 1)

bestPathCost

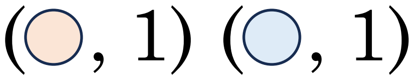
 → 0  → 1



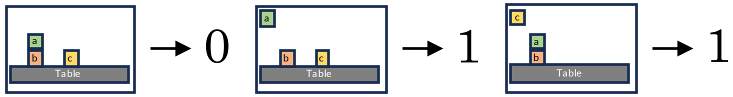
$\text{GRAPHSEARCH}(s_0, \mathcal{A}, I, F, C, G)$

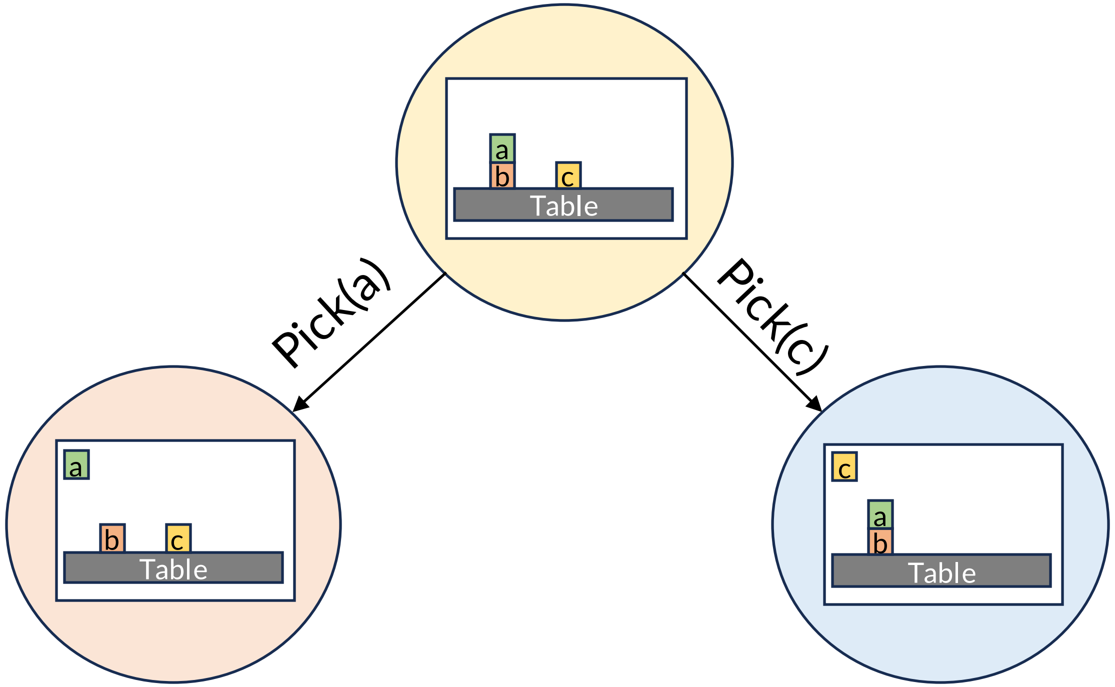
// priority queue of nodes
initialize queue $=$ []
root $=$ $\text{NODE}(s_0, 0,$ parent=null$)$
// PRIORITY differs between algorithms
push root **onto** queue **with** PRIORITY(root)
// state $\rightarrow$ best known path cost
initialize bestPathCost $= \{s_0 \mapsto 0\}$
while queue **is not empty**
  pop node **from** queue
  $s, c =$ node.state, node.pathCost
  // skip if we already found a better path
  if bestPathCost[s] $< c$: **continue**
  if $G(s)$: **return** node.extractPlan()
  $\boxed{\textbf{for } a \in \mathcal{A} \text{ s.t. } I(s, a)}$
    $s' = F(s, a)$
    $c' = c + C(s, a, s')$
    if bestPathCost[s'] $< c'$: **continue**
    bestPathCost[s'] $= c'$
    child $= \text{NODE}(s', c',$ parent=node$)$
    push child **onto** queue **with** PRIORITY(child)

queue



bestPathCost

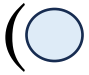




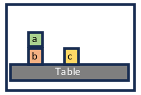
GraphSearch($s_0, \mathcal{A}, I, F, C, G$)

```
// priority queue of nodes
initialize queue = []
root = Node(s_0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {s_0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = Node(s′, c′, parent=node)
        push child onto queue with PRIORITY(child)
```

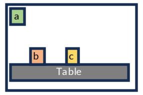
queue

(, 1)

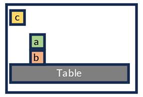
bestPathCost



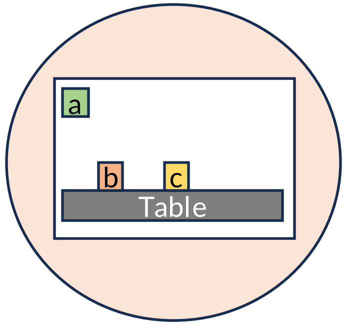
→ 0



→ 1

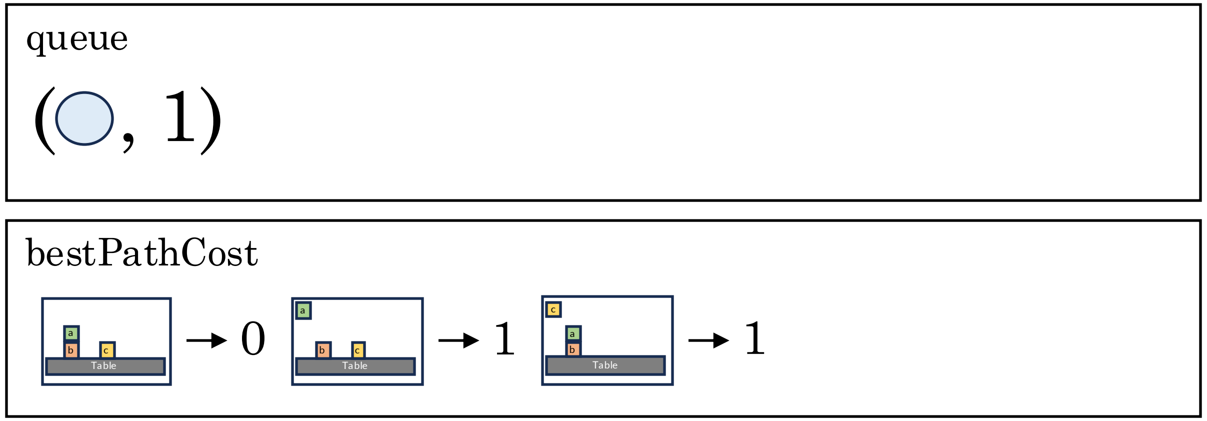


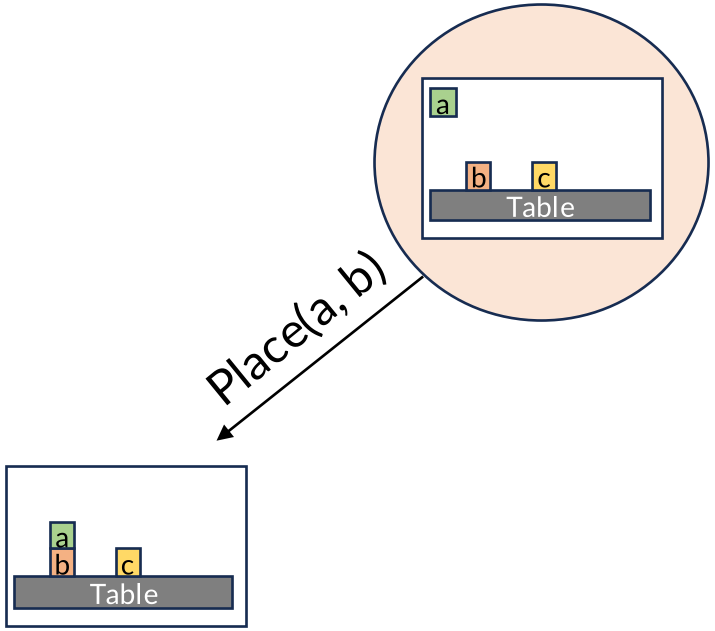
→ 1



GraphSearch($s_0, \mathcal{A}, I, F, C, G$)

// priority queue of nodes
initialize queue $=$ []
root $=$ Node(s_0, 0, parent=null)
// priority differs between algorithms
push root onto queue with priority(root)
// state $\rightarrow$ best known path cost
initialize bestPathCost $= \{s_0 \mapsto 0\}$
while queue is not empty
  pop node from queue
  $s, c =$ node.state, node.pathCost
  // skip if we already found a better path
  if bestPathCost[s] $< c$: **continue**
  if $G(s)$: **return** node.extractPlan()
  for $a \in \mathcal{A}$ s.t. $I(s, a)$
    $s' = F(s, a)$
    $c' = c + C(s, a, s')$
    if bestPathCost[s'] $< c'$: **continue**
    bestPathCost[s'] $= c'$
    child $=$ Node(s', c', parent=node)
    push child onto queue with priority(child)

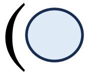




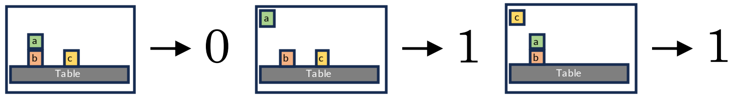
GRAPHSEARCH$(s_0, \mathcal{A}, I, F, C, G)$

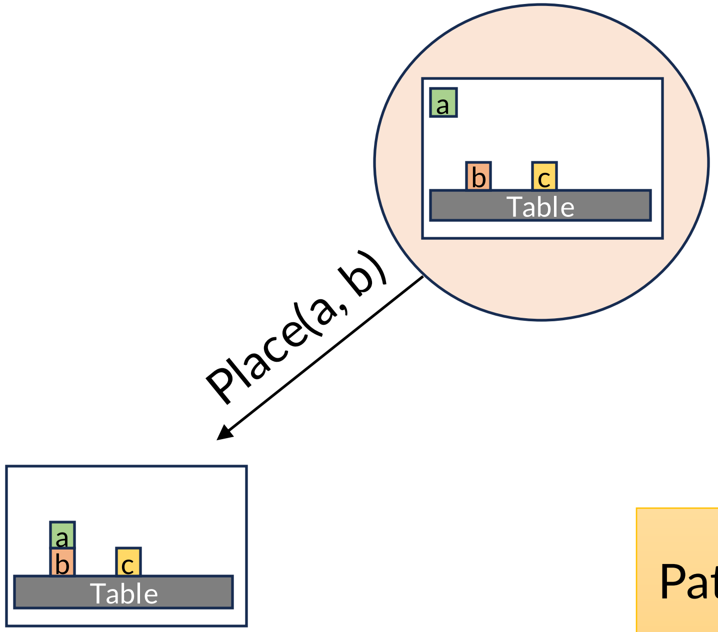
```
// priority queue of nodes
initialize queue = []
root = NODE(s0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {s0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = NODE(s′, c′, parent=node)
        push child onto queue with PRIORITY(child)
```

queue

(, 1)

bestPathCost

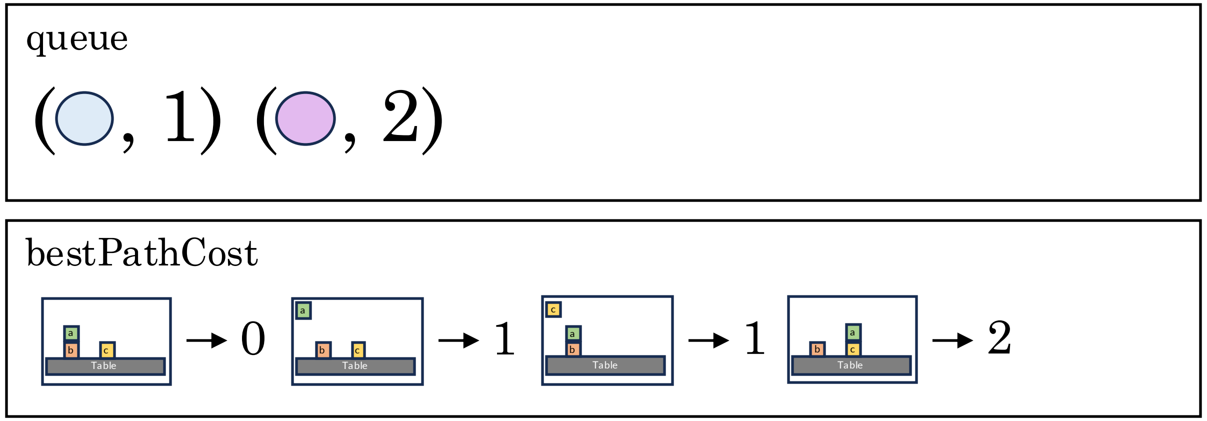


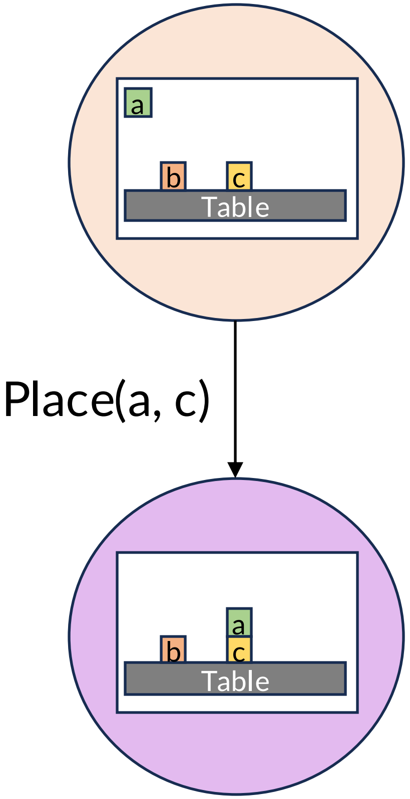


Path cost is 2, which is worse than 0

GRAPHSEARCH(s_0, $\mathcal{A}$, I, F, C, G)

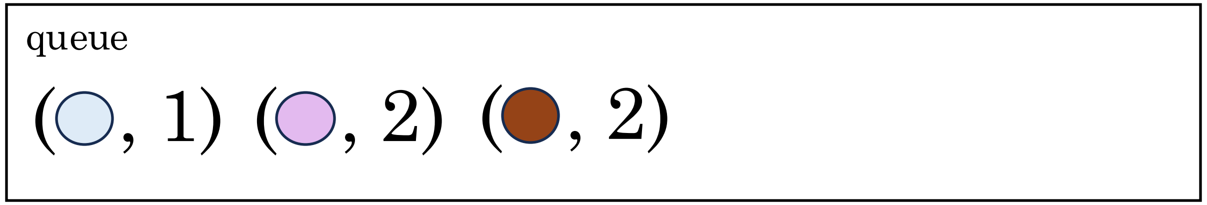
```
// priority queue of nodes
initialize queue = []
root = NODE(s0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {s0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s' = F(s, a)
        c' = c + C(s, a, s')
        if bestPathCost[s'] < c' : continue
        bestPathCost[s'] = c'
        child = NODE(s', c', parent=node)
        push child onto queue with PRIORITY(child)
```

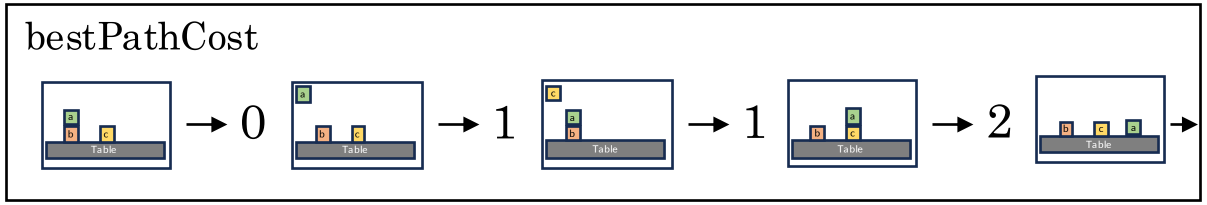


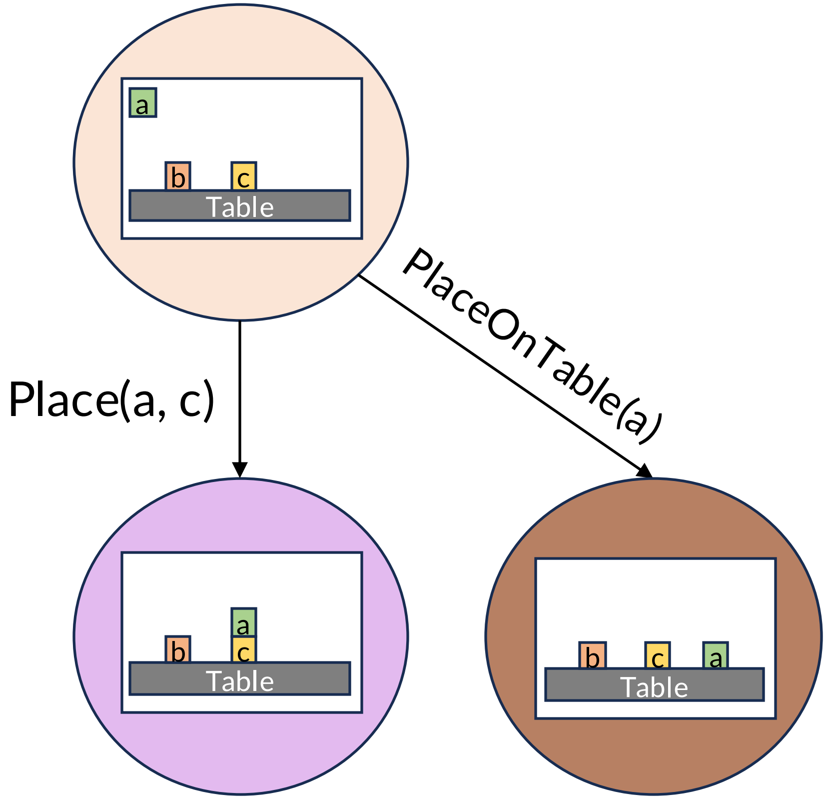


GraphSearch(s_0, $\mathcal{A}$, I, F, C, G)

```
// priority queue of nodes
initialize queue = []
root = Node(s_0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {s_0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ 𝒜 s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = Node(s′, c′, parent=node)
        push child onto queue with PRIORITY(child)
```







GRAPHSEARCH(s_0, $\mathcal{A}$, I, F, C, G)

```
// priority queue of nodes
initialize queue = []
root = NODE(s0, 0, parent=null)
// PRIORITY differs between algorithms
push root onto queue with PRIORITY(root)
// state → best known path cost
initialize bestPathCost = {s0 ↦ 0}
while queue is not empty
    pop node from queue
    s, c = node.state, node.pathCost
    // skip if we already found a better path
    if bestPathCost[s] < c : continue
    if G(s) : return node.extractPlan()
    for a ∈ A s.t. I(s, a)
        s′ = F(s, a)
        c′ = c + C(s, a, s′)
        if bestPathCost[s’] < c′ : continue
        bestPathCost[s’] = c′
        child = NODE(s′, c′, parent=node)
        push child onto queue with PRIORITY(child)
```

Heuristics and Value Functions

As in MDP land, we can define value functions:

$$V^*(s) = \begin{cases} 0 & \text{if } G(s) \\ \min\limits_{\substack{a:I(s,a) \\ s'=F(s,a)}} C(s,a,s') + V^*(s') & \text{o.w.} \end{cases}$$

"Cost-to-go"

Heuristics and Value Functions

As in MDP land, we can define value functions:

$$V^*(s) = \begin{cases} 0 & \text{if } G(s) \\ \min\limits_{\substack{a:I(s,a) \\ s'=F(s,a)}} C(s,a,s') + V^*(s') & \text{o.w.} \end{cases}$$

A **heuristic** $\hat{V}(s)$ is an approximate value function.

Same as MDP land

Heuristics and Value Functions

As in MDP land, we can define value functions:

$$V^*(s) = \begin{cases} 0 & \text{if } G(s) \\ \min\limits_{\substack{a : I(s,a) \\ s' = F(s,a)}} C(s, a, s') + V^*(s') & \text{o.w.} \end{cases}$$

A **heuristic** $\hat{V}(s)$ is an approximate value function.

A heuristic is **admissible** if it never overestimates the cost-to-go:

For all $s \in \mathcal{S}, \hat{V}(s) \leq V^*(s)$.

Graph Search Variations

Algorithm	Priority Function	Optimal?	Notes
Uniform cost search	path cost	Yes	If costs are 1, this is breadth-first search. Like Dijkstra's, but returns shortest path to goal, not shortest paths to all states
Greedy best-first search (GBFS)	**heuristic**(state)	No	Good choice for fast satisficing planning
A* search	path cost + **heuristic**(state)	Depends	Optimal if heuristic is *admissible* (never overestimates cost-to-go)
Depth first search	negative path cost	No	Can be more memory-efficient if implemented as a special case

Where do Heuristics Come From?

1. Hand-designed based on understanding of the problem

2. Learned from data (later in the course)

3. **Automatically derived from the problem representation**

Factored Classical Planning Problems

Consider a classical planning problem where:

States are *factored* into n Boolean features:

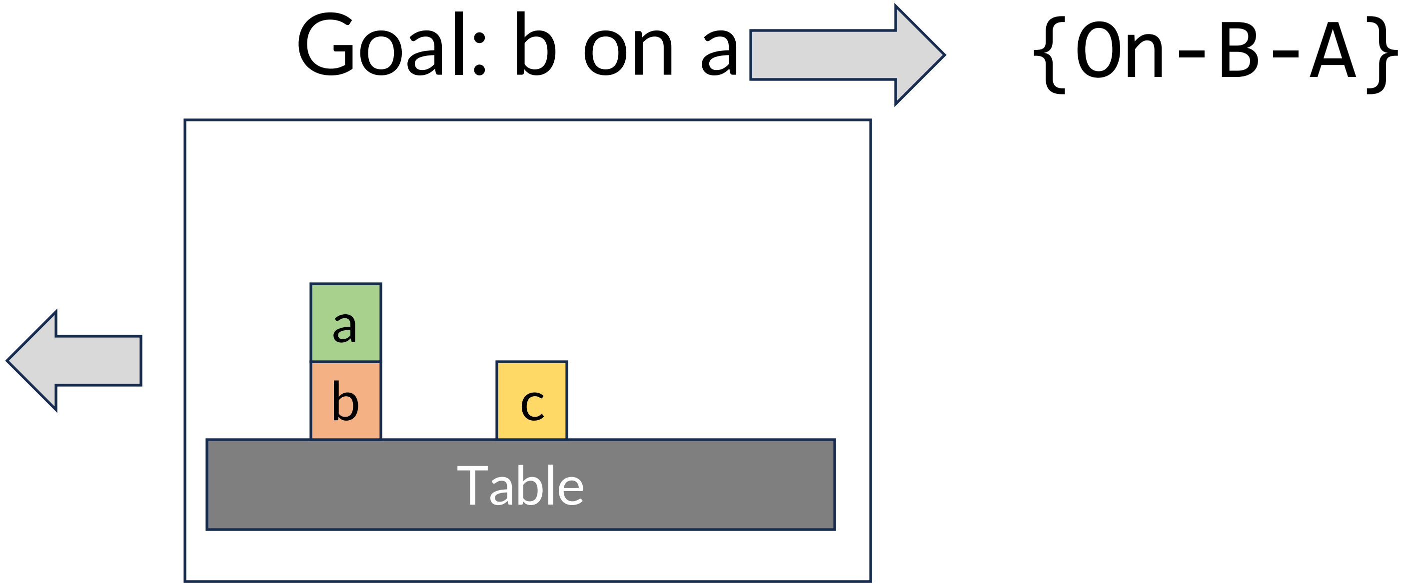
$$\mathcal{S} = \{\mathrm{T}, \mathrm{F}\}^n$$

The goal is to “activate” features $\{i_1, \ldots, i_m\}$ (for $1 \leq i_j \leq n$):

$$G(s) = s[i_1] \wedge \cdots \wedge s[i_m]$$

Blocks World Example

Feature	Value
On-A-B	True
On-A-C	False
On-B-A	False
On-B-C	False
On-C-A	False
On-C-B	False
OnTable-A	False
OnTable-B	True
OnTable-C	True
Holding-A	False
Holding-B	False
Holding-C	False
HandEmpty	True

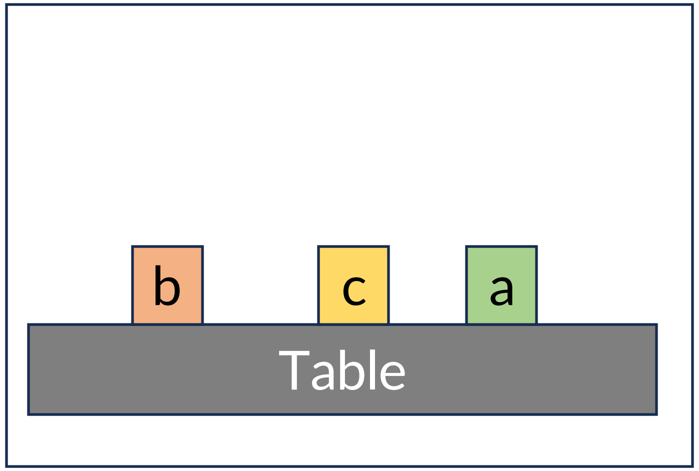


Blocks World Example

Feature	Value
`On-A-B`	**`False`**
`On-A-C`	`False`
`On-B-A`	`False`
`On-B-C`	`False`
`On-C-A`	`False`
`On-C-B`	`False`
`OnTable-A`	**`True`**
`OnTable-B`	`True`
`OnTable-C`	`True`
`Holding-A`	`False`
`Holding-B`	`False`
`Holding-C`	`False`
`HandEmpty`	`True`

Goal: a on b & b on c

`{On-A-B, On-B-C}`



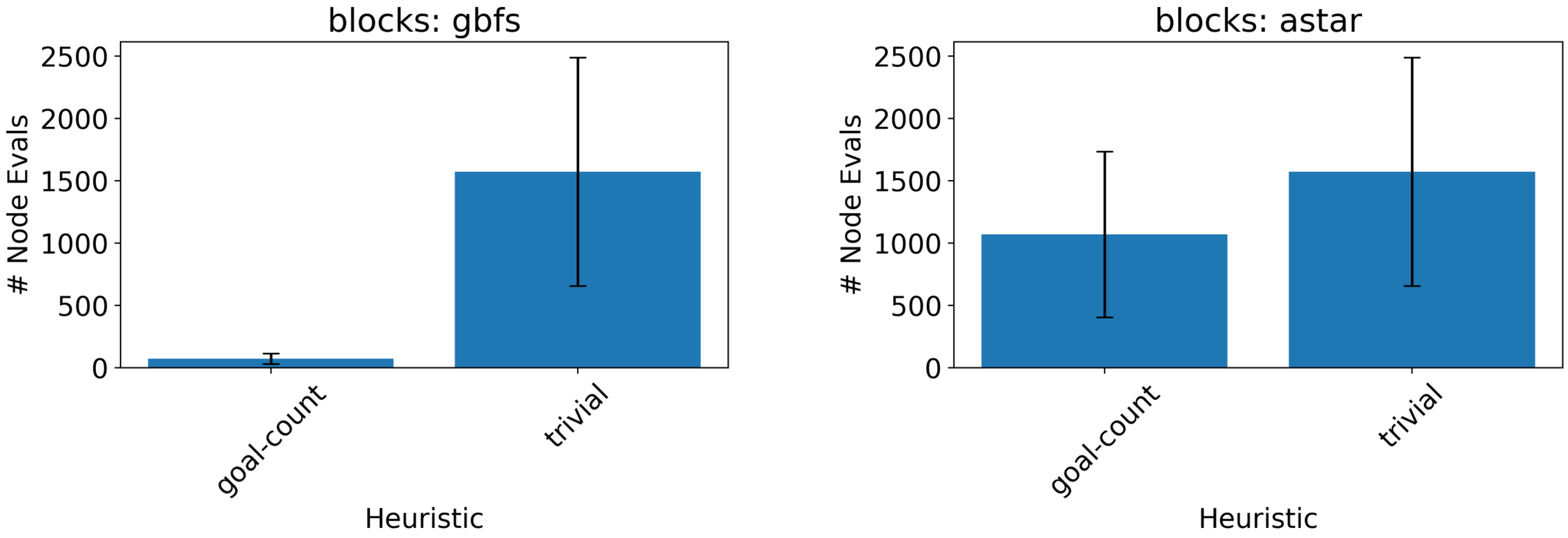
Goal-Count: Our First Problem-Derived Heuristic

The goal-count heuristic counts the number of goal features that are not yet activated:

$$V_{GC}(s) \triangleq |\{i : \neg s[i] \land i \in G\}|$$

Assuming all transition costs are 1, is V_{GC} admissible?

Goal-Count Can Help!



Limitations of Goal-Count

1. Very sparse

2. Can be “misleading”

Examples?

Factoring Further: Actions + Transitions

A (STRIPS / PDDL) *operator* has:

1. Preconditions
2. Add effects
3. Delete effects

Each is a set of features

Notation: ω is an operator, Ω is the set of all operators

<u>Pick-A-from-C:</u>

Preconditions: {HandEmpty, On-A-C, Clear-A}

Add effects: {Holding-A, Clear-C}

Delete effects: {HandEmpty, On-A-C, Clear-A}

Factored Classical Planning Problems

A *factored classical planning problem* is:

1. A finite state space $\mathcal{S} = 2^{\{1,\ldots,n\}}$ Set of all true features
2. A finite action space $\mathcal{A} = \Omega$ Actions = operators
3. An initiable action function $I(s, \omega) = \mathrm{pre}(\omega) \subseteq s$ Preconditions hold
4. A transition function $F(s, \omega) = \big(\mathrm{s} - \mathrm{del}(\omega)\big) \cup \mathrm{add}(\omega)$ Effects
5. A cost function $C(s, \omega, s') = 1$ For simplicity
6. An initial state $s_0 \in \mathcal{S}$
7. A goal function $G(s) = \mathrm{g} \subseteq s$ g is another feature set

Lifted Operators

It is often convenient to define operators with **parameters**: placeholders for objects

Objects can also be **typed**

Preprocessing: *ground* all lifted operators with all combinations of objects (obeying types)

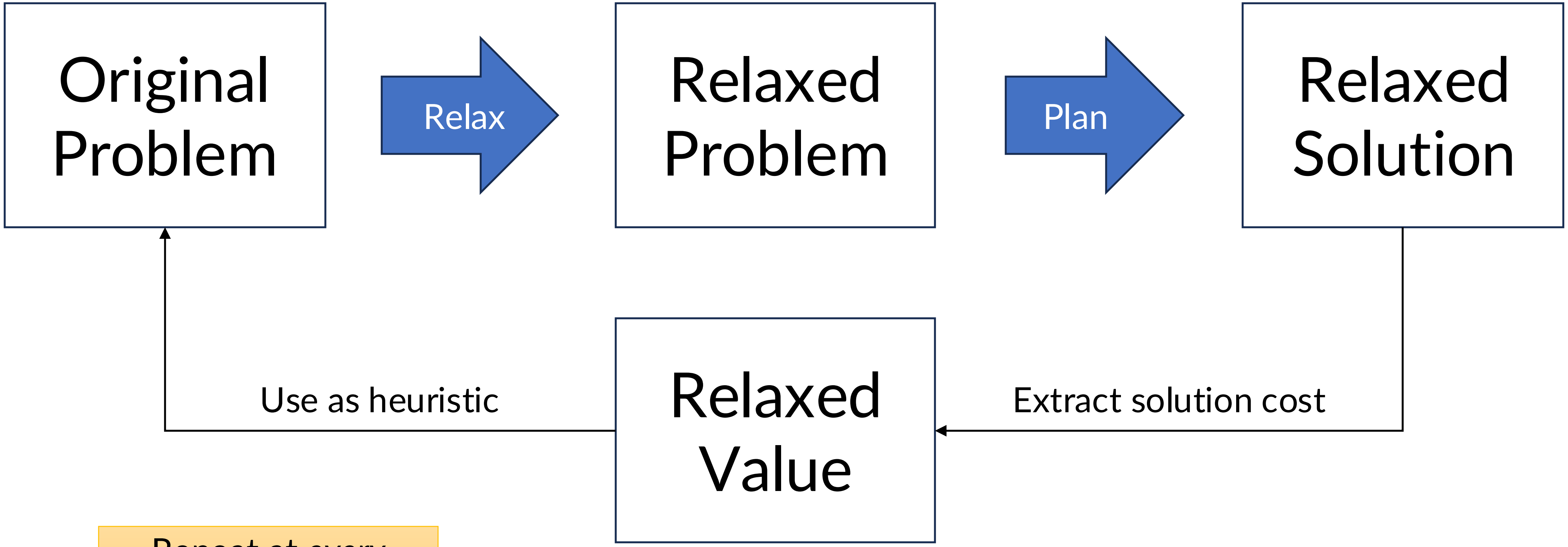
Pick(?x, ?y):

Preconditions: {HandEmpty(),
On(?x, ?y),
Clear(?x)}

Add effects: {Holding(?x), Clear(?y)}

Delete effects: {HandEmpty(),
On(?x, ?y)
Clear(?x)}

A Recipe for Heuristic Generation



Repeat at *every state*

Delete Relaxation

Pick(?x, ?y):

Preconditions: {HandEmpty(),
On(?x, ?y),
Clear(?x)}

Add effects: {Holding(?x),
Clear(?y)}

Delete effects: {HandEmpty(),
On(?x, ?y)
Clear(?x)}

Relax

Pick(?x, ?y):

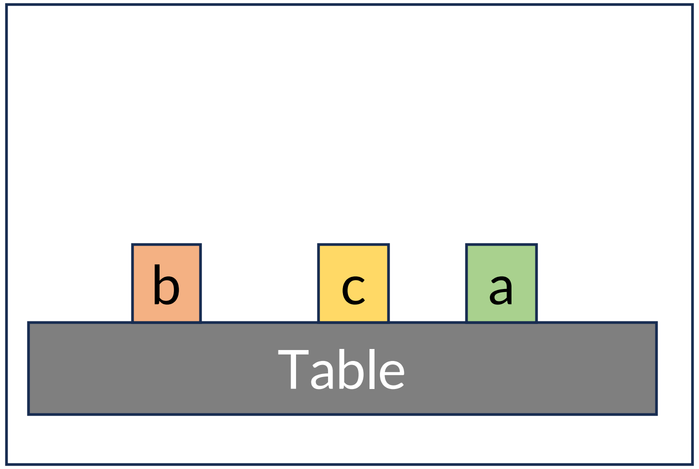
Preconditions: {HandEmpty(),
On(?x, ?y),
Clear(?x)}

Add effects: {Holding(?x),
Clear(?y)}

Delete effects: {}

Delete-Relax: Our Second Problem-Derived Heuristic

The delete-relax heuristic $V_{DR}(s)$ is the optimal cost of the **relaxed** planning problem with initial state s.

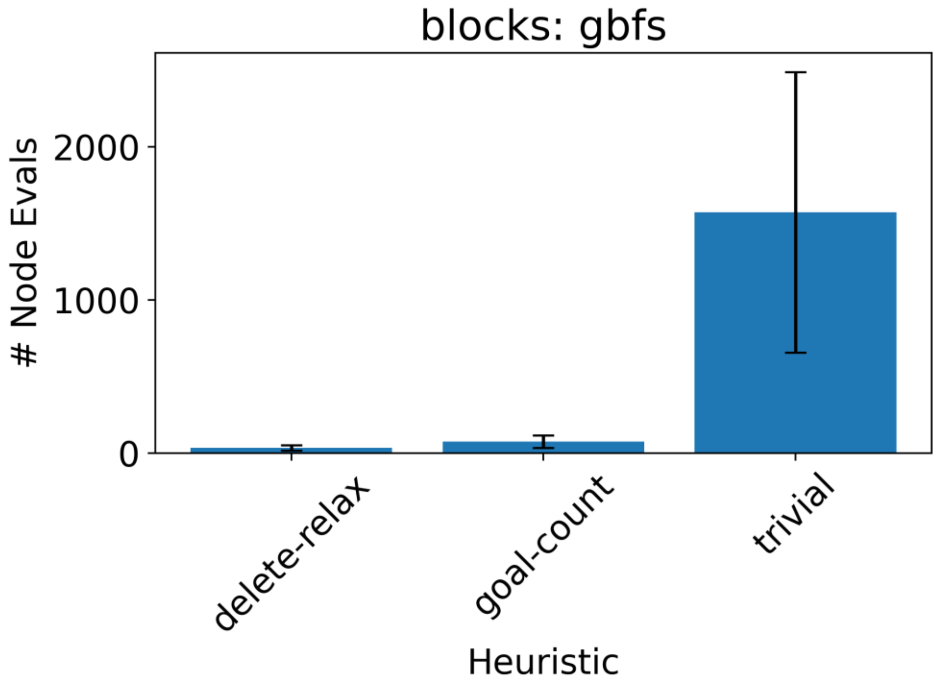


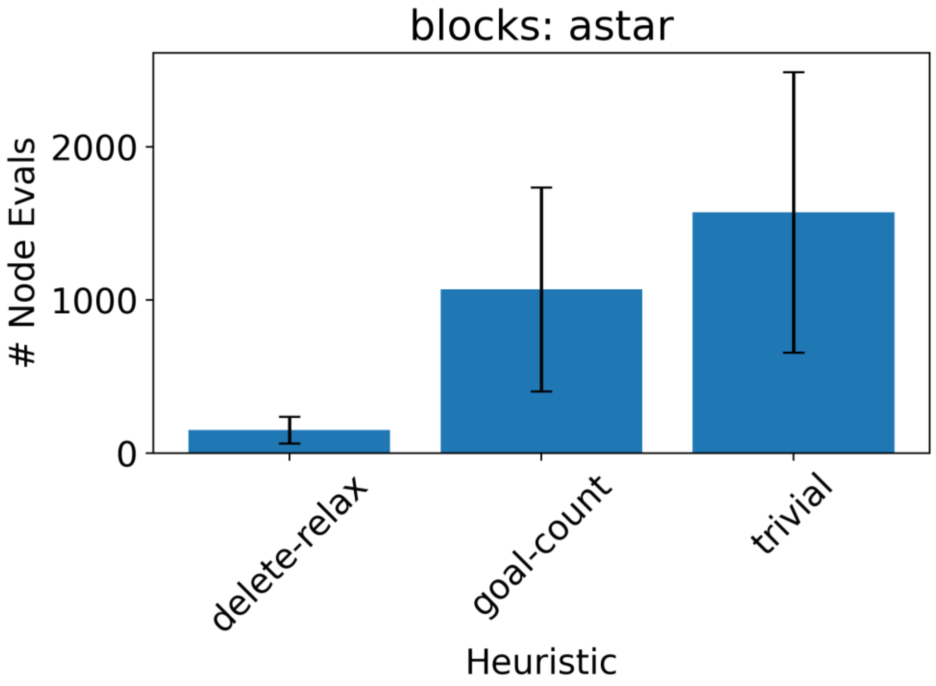
Goal: holding(a) & holding(b)

What is $V_{DR}(s)$?
What is $V^*(s)$?

Is V_{DR} admissible?

Delete Relaxation Can Help!

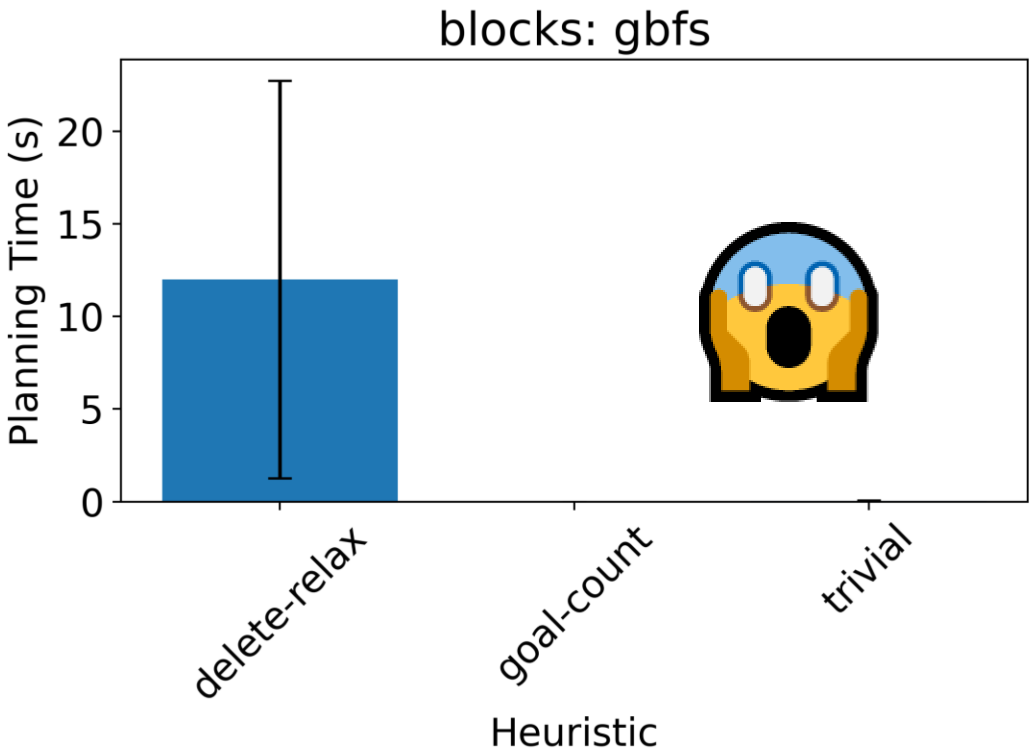


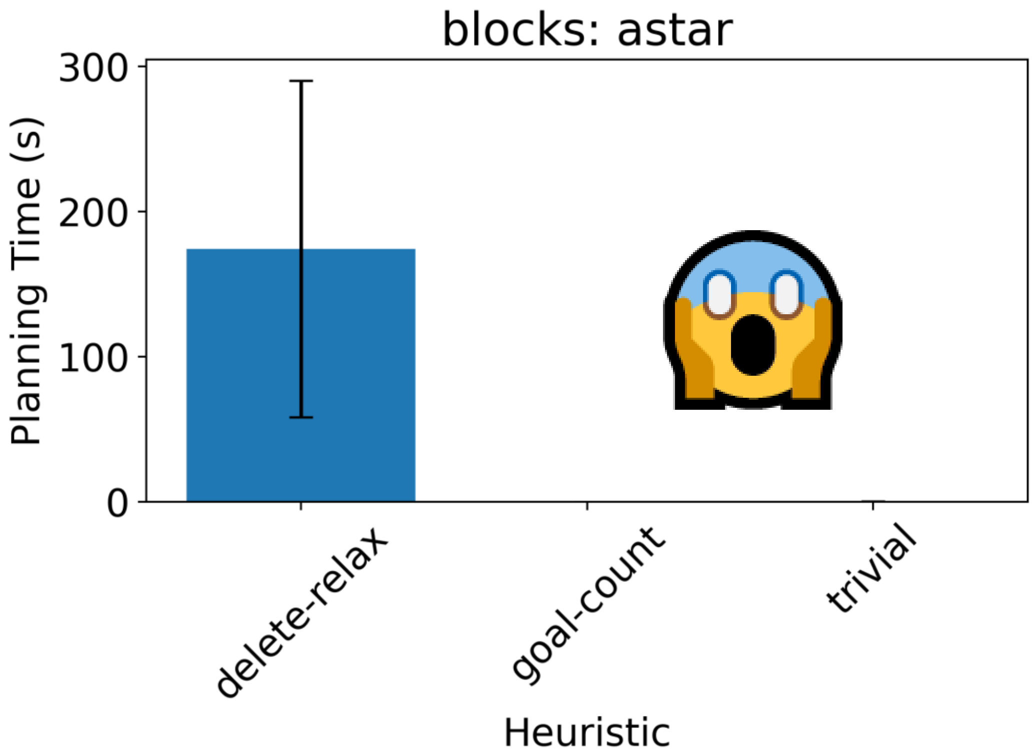


But these plots are extremely misleading. Why?

More Revealing Plots

Solving delete-relaxed problems exactly is formally hard (NP-complete)





hFF: A Better Delete Relaxation Heuristic

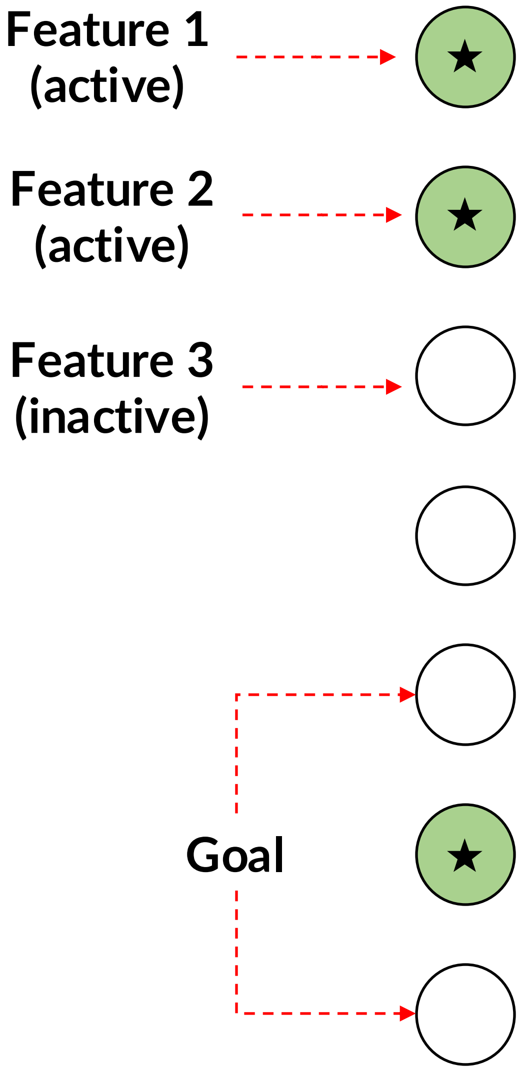
Construct a *non-optimal* relaxed plan in a particular way:

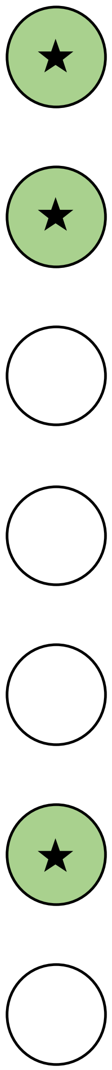
<u>Forward pass:</u>

1. Imagine we could execute *all* initiable actions simultaneously
2. Aggregate the next states into superset of all active features
3. Repeat (1) and (2) until convergence (or goal is active)

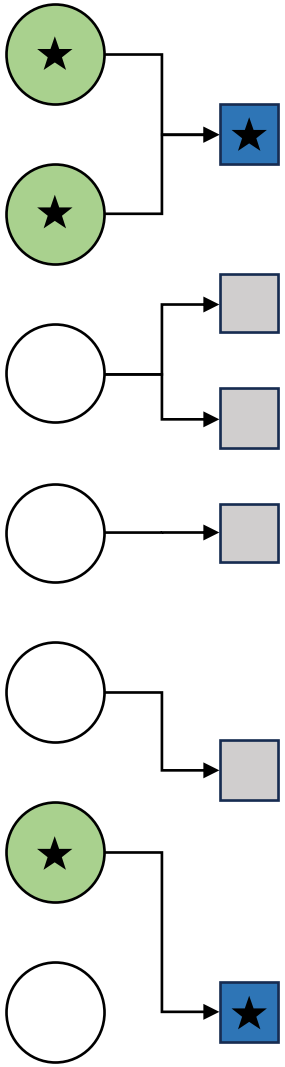
<u>Backward pass:</u>

1. Build a relaxed plan by selecting “necessary” actions

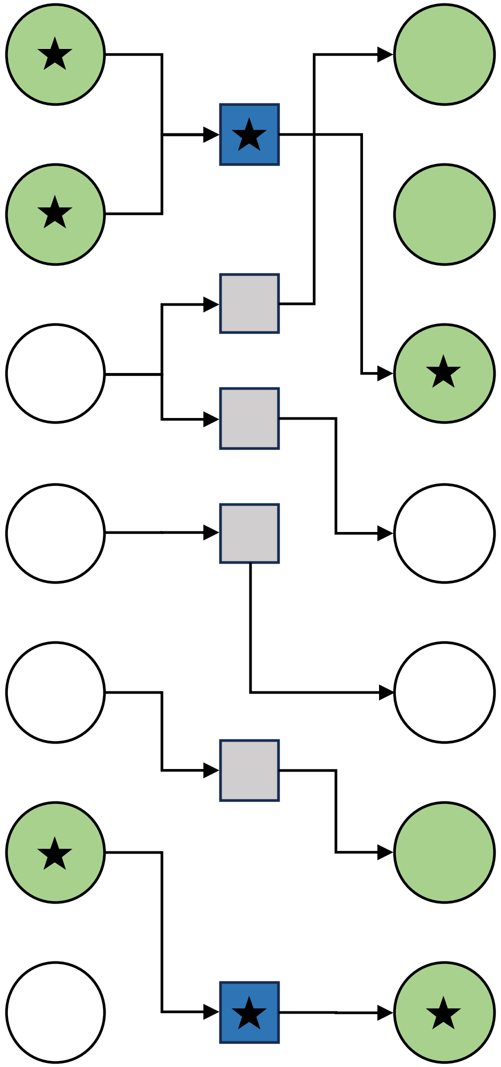

Feature 1 (active)
Feature 2 (active)
Feature 3 (inactive)
Goal

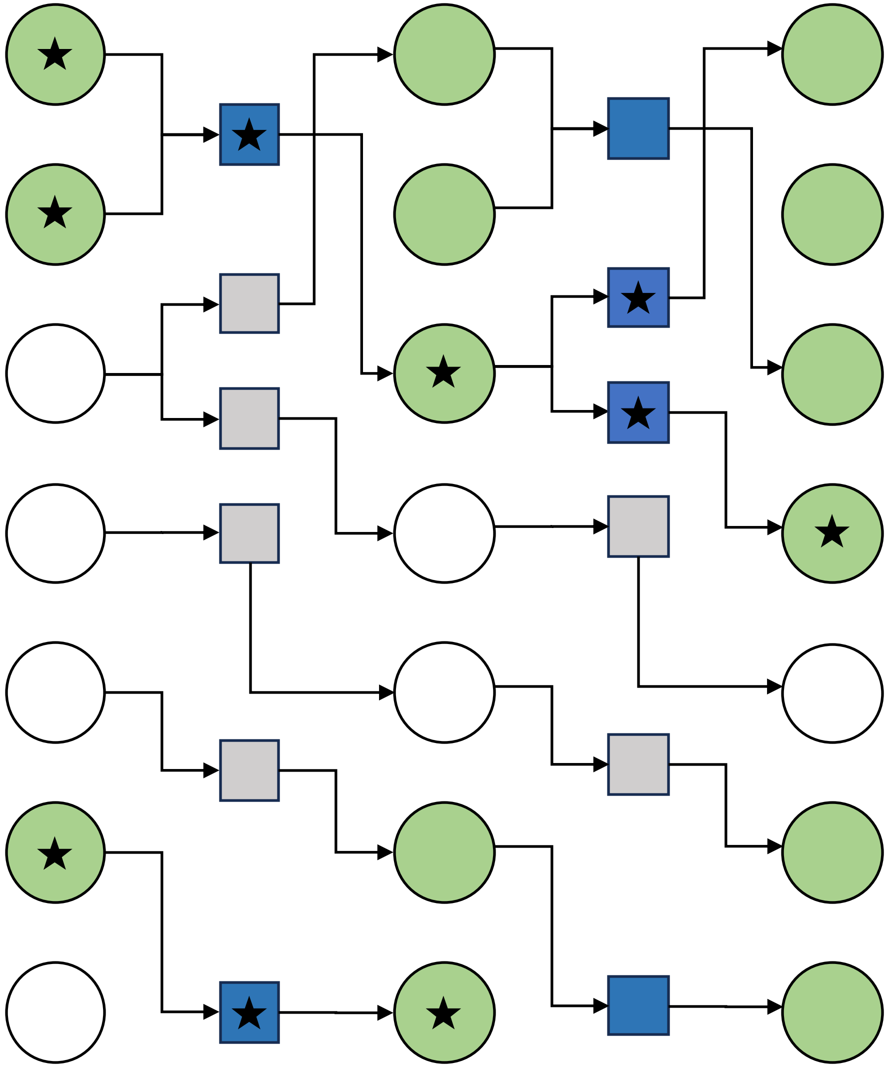


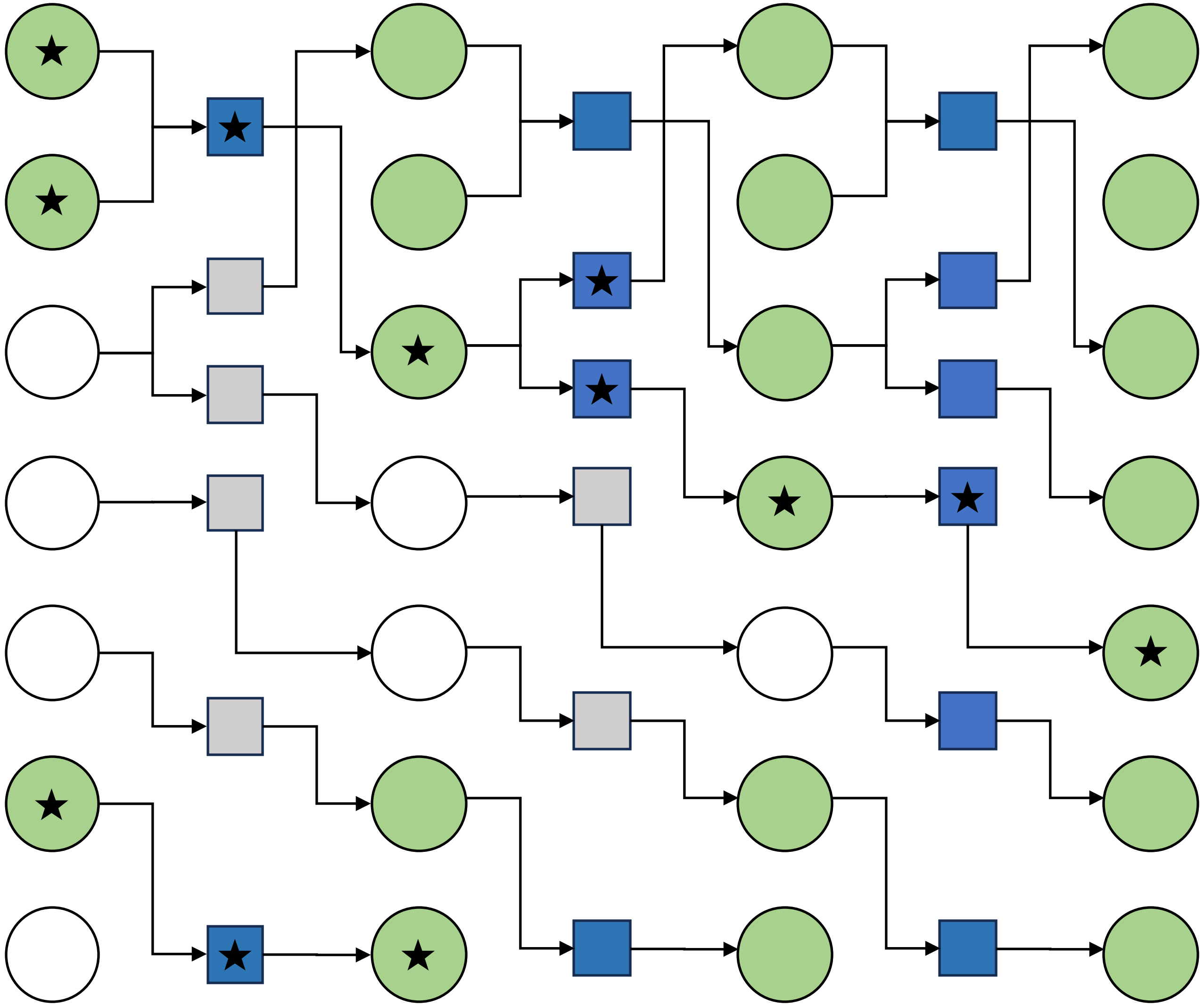
Operators



Operators

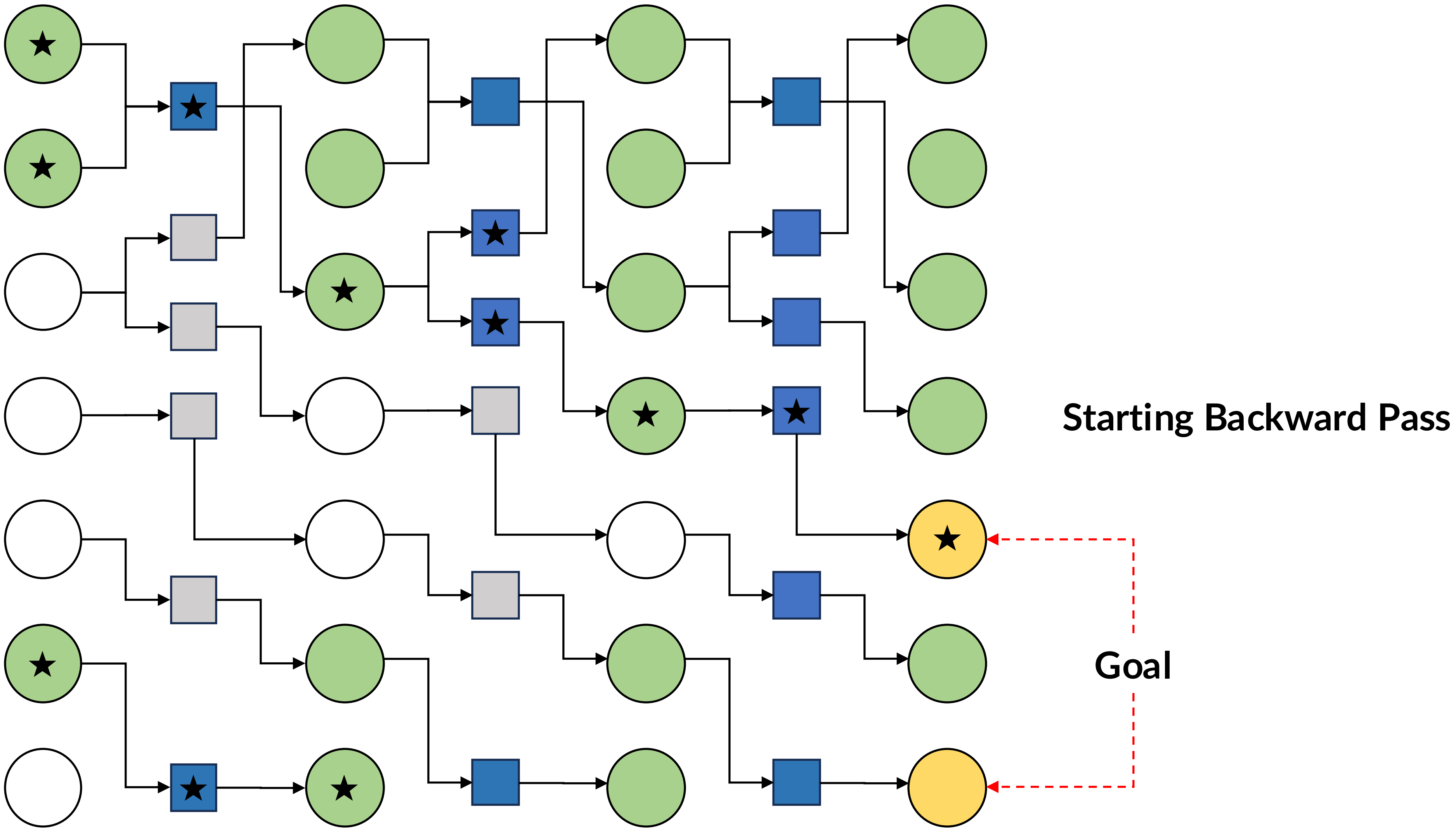


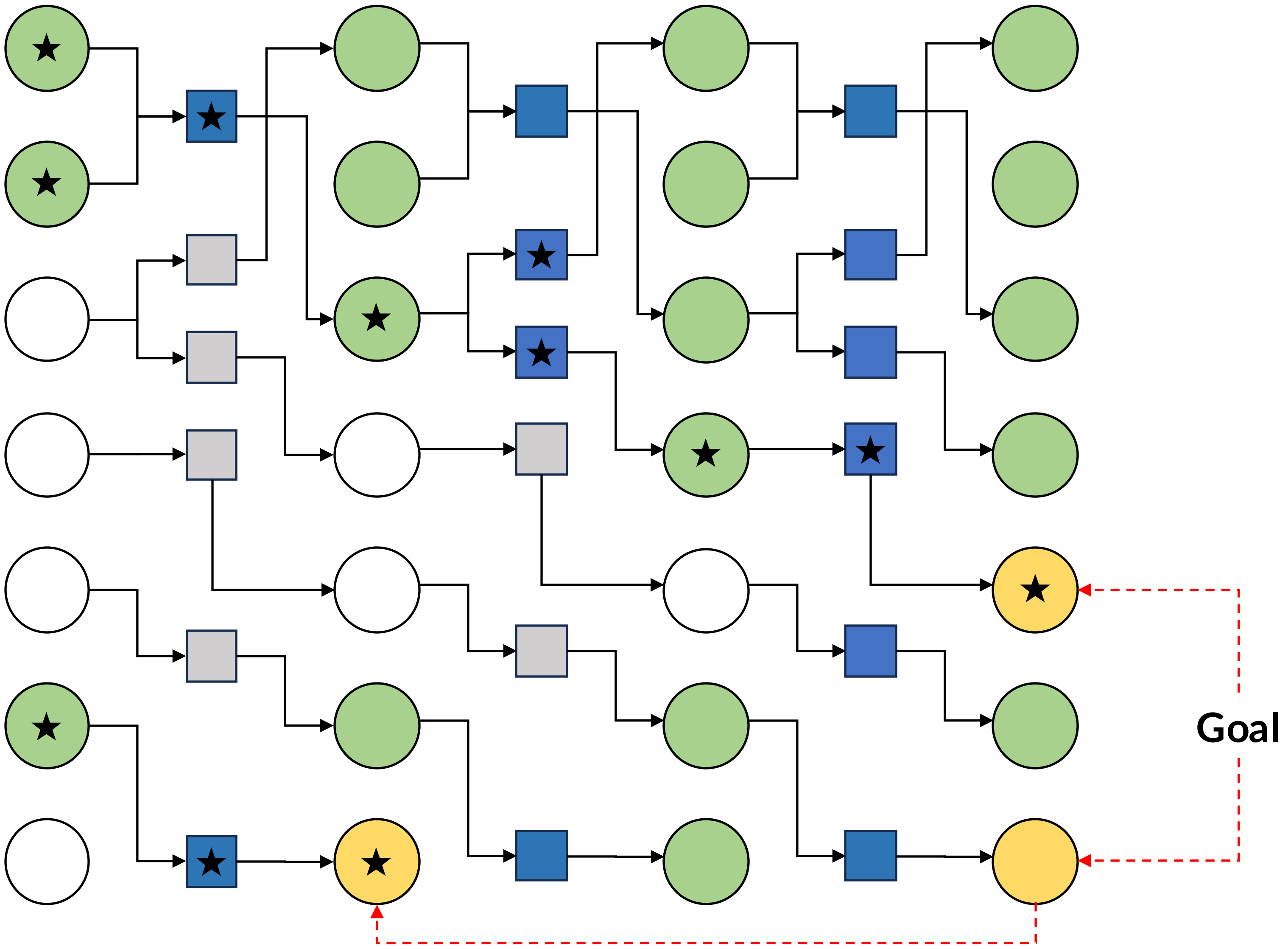


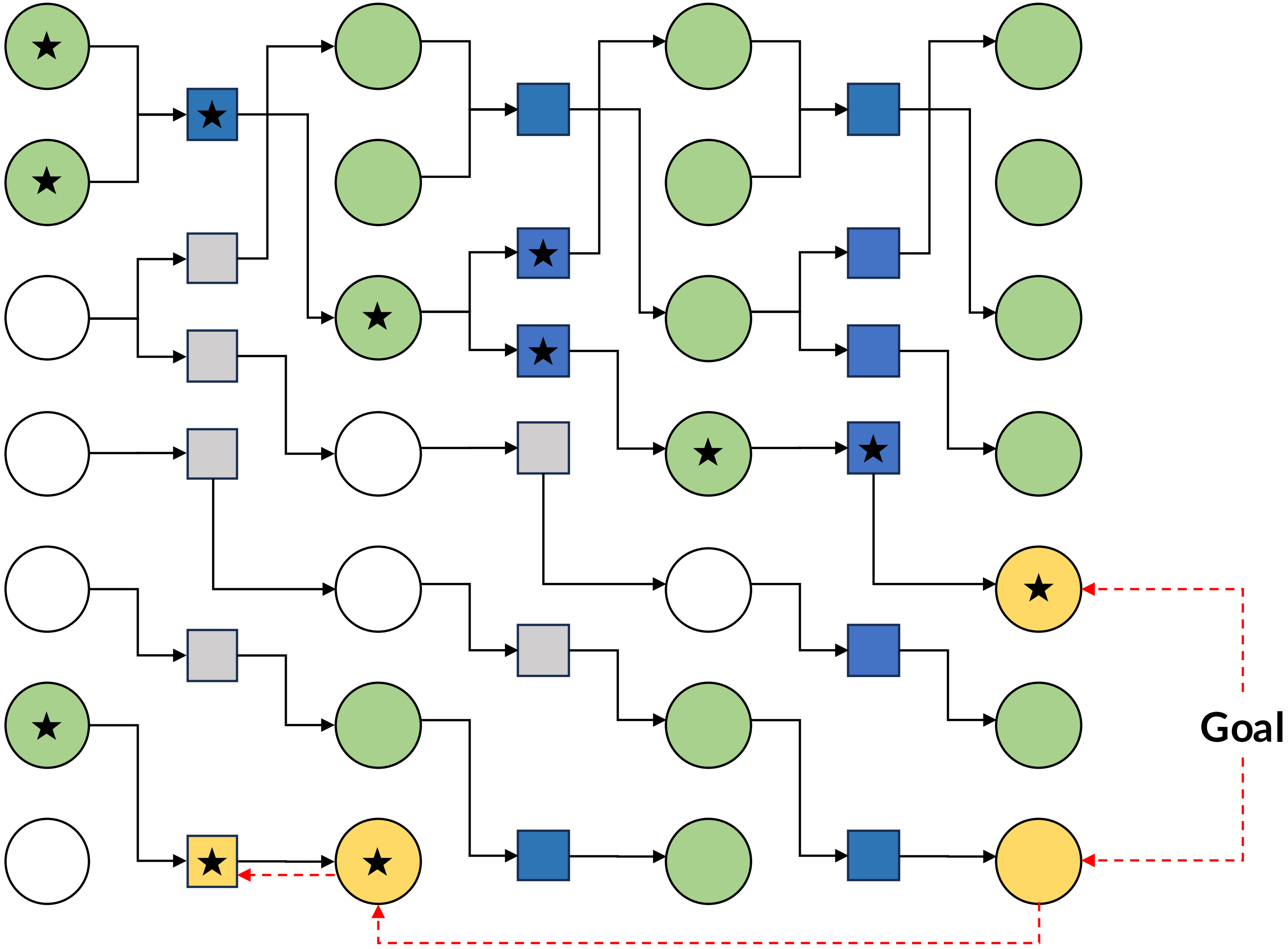


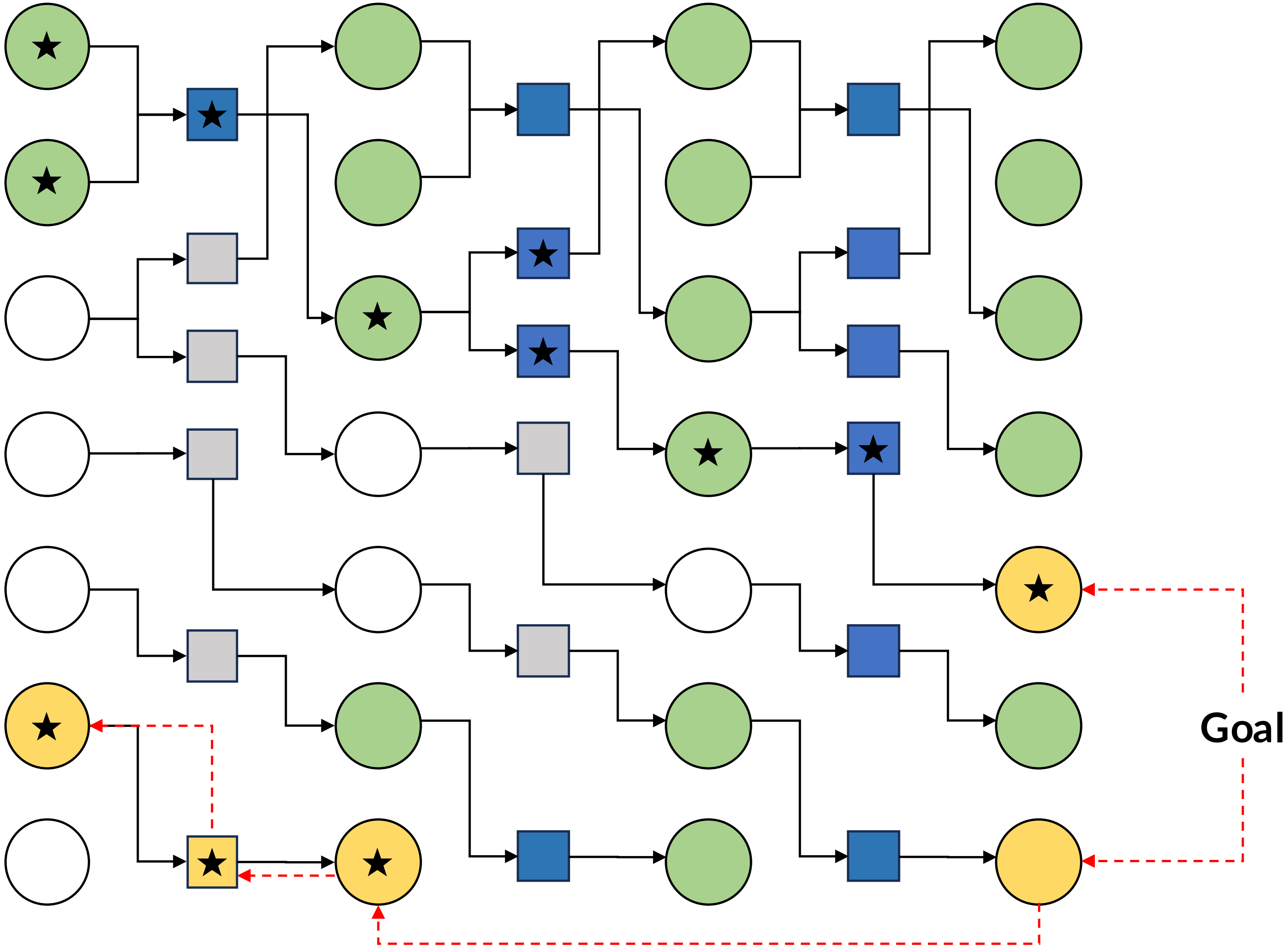
Convergence

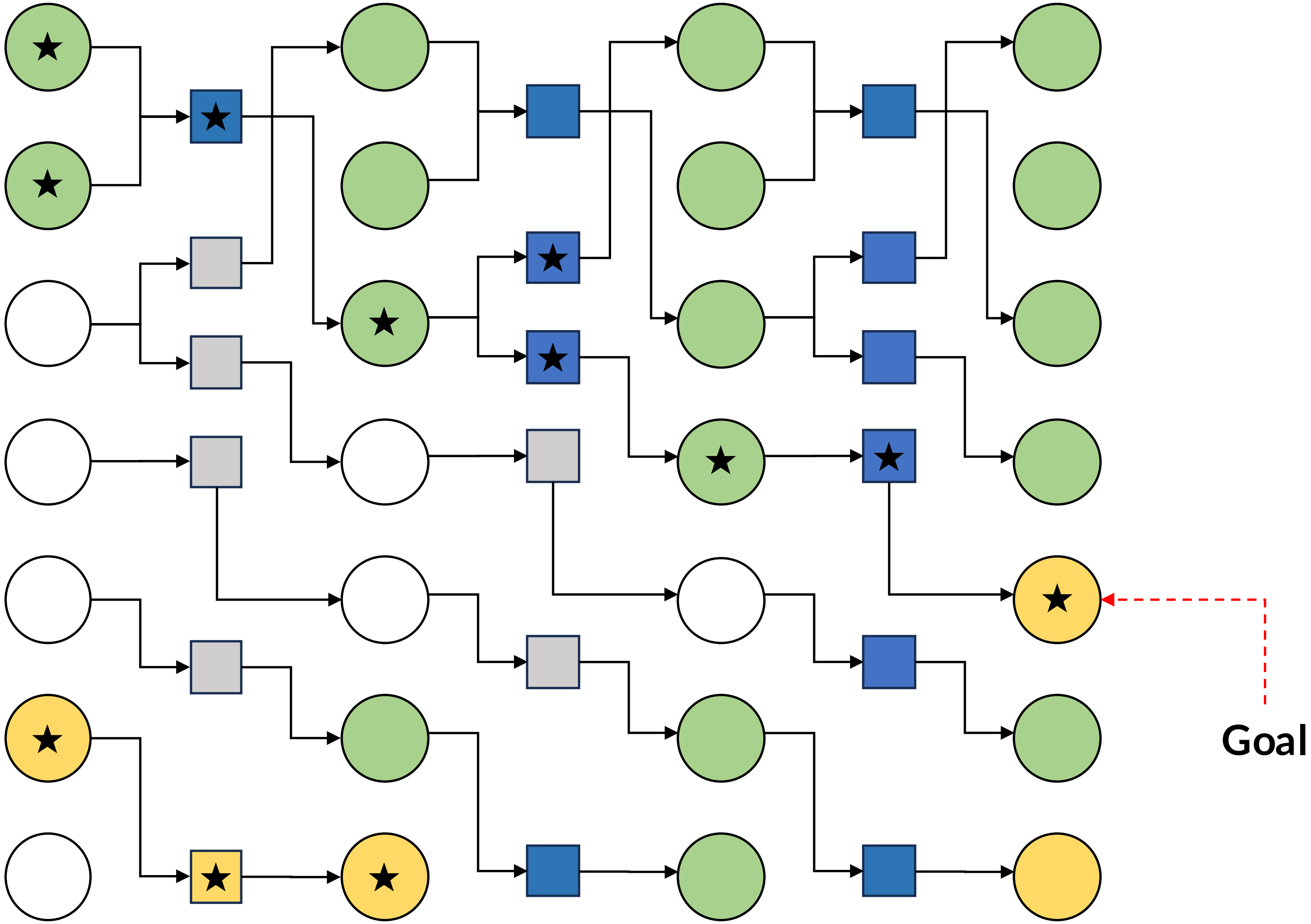
Forward Pass Complete

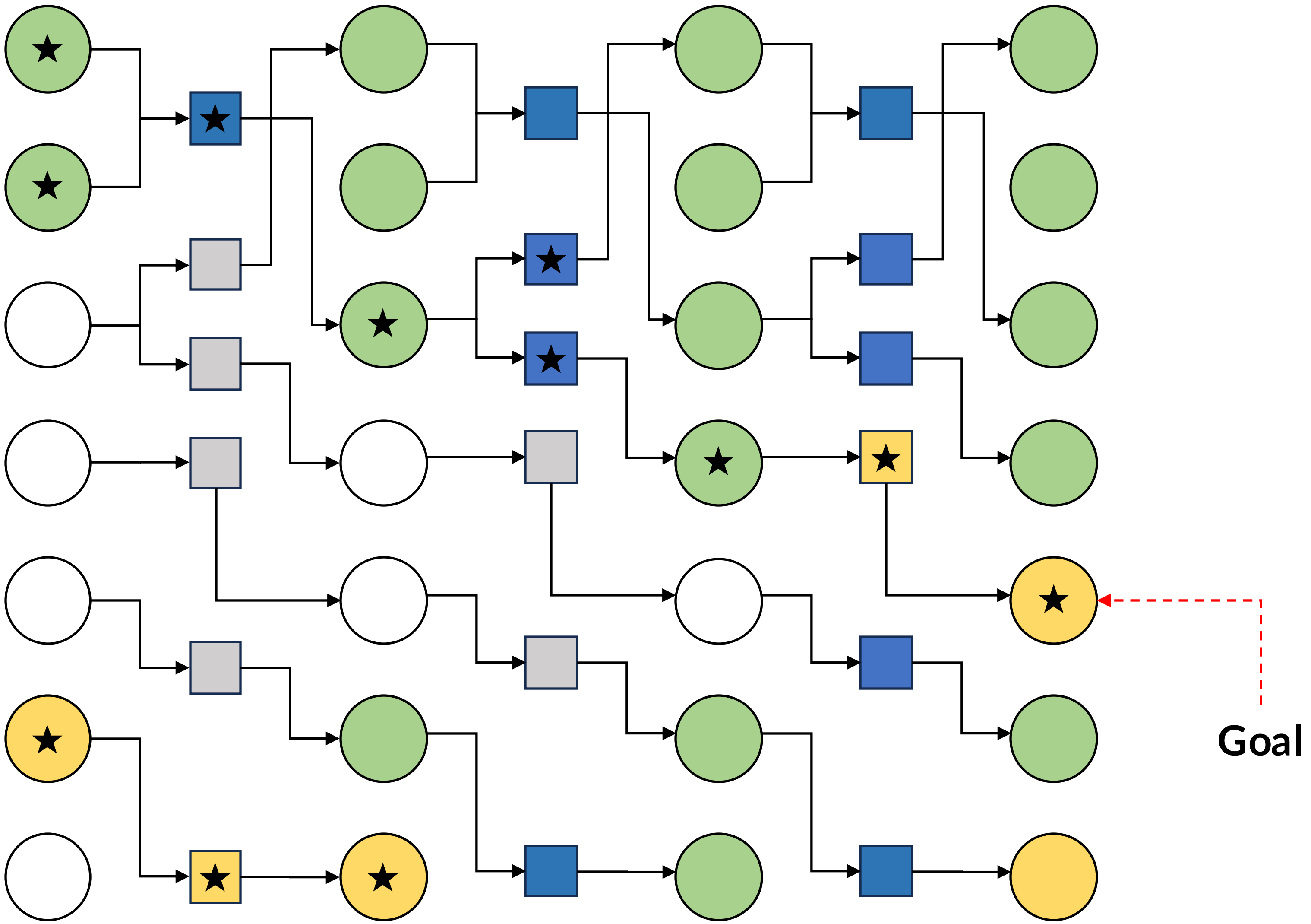

Starting Backward Pass
Goal

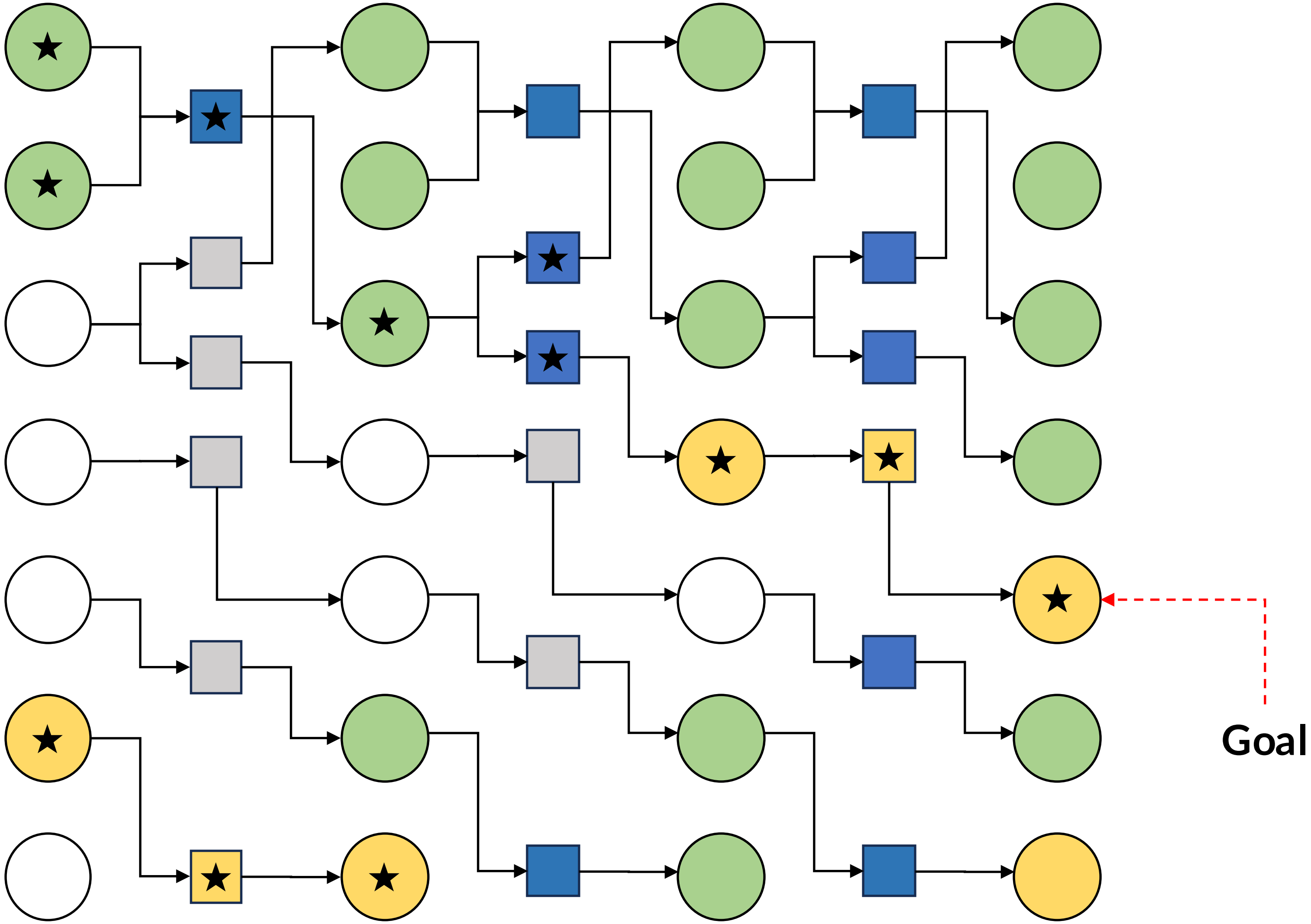

Goal

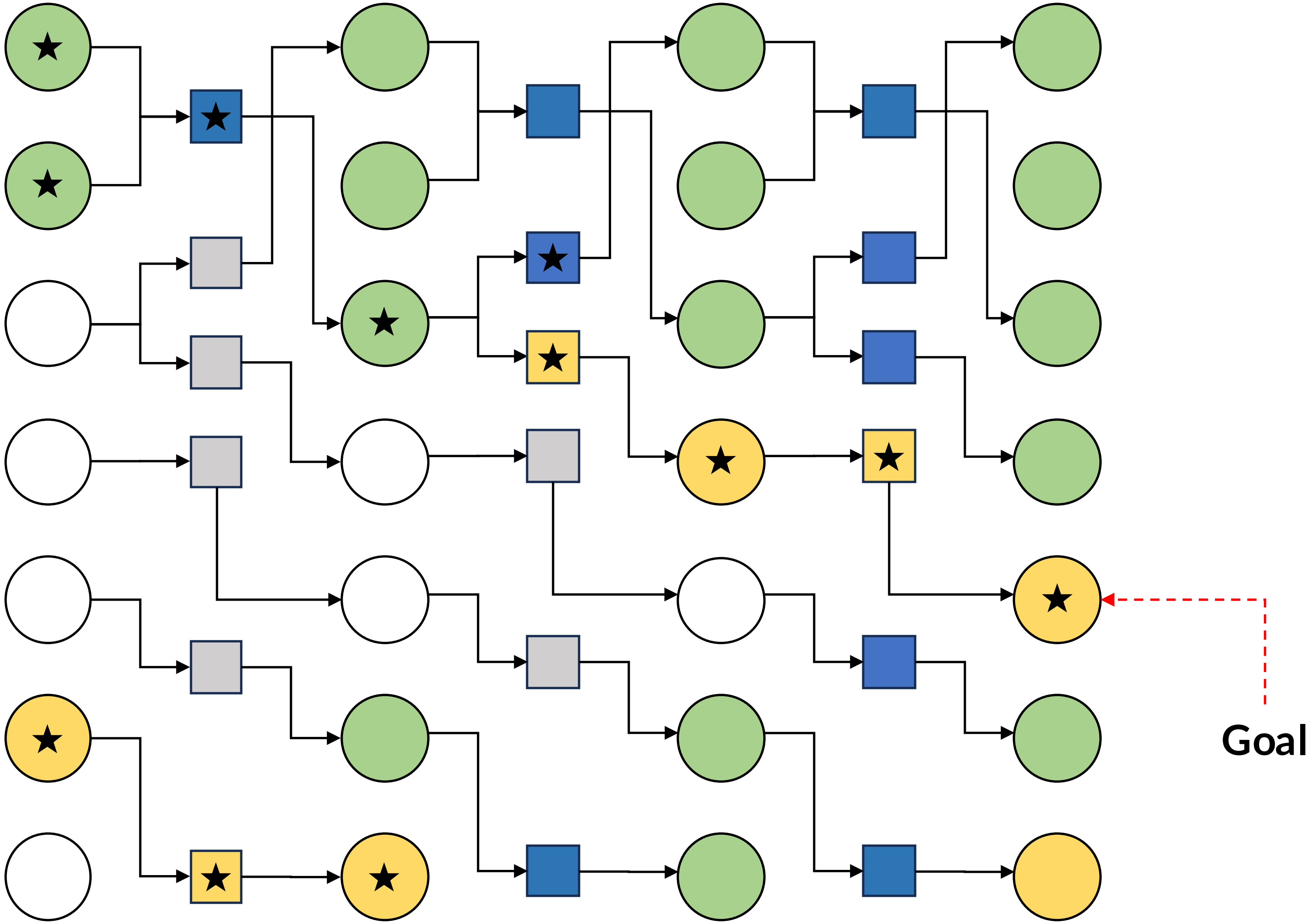

Goal

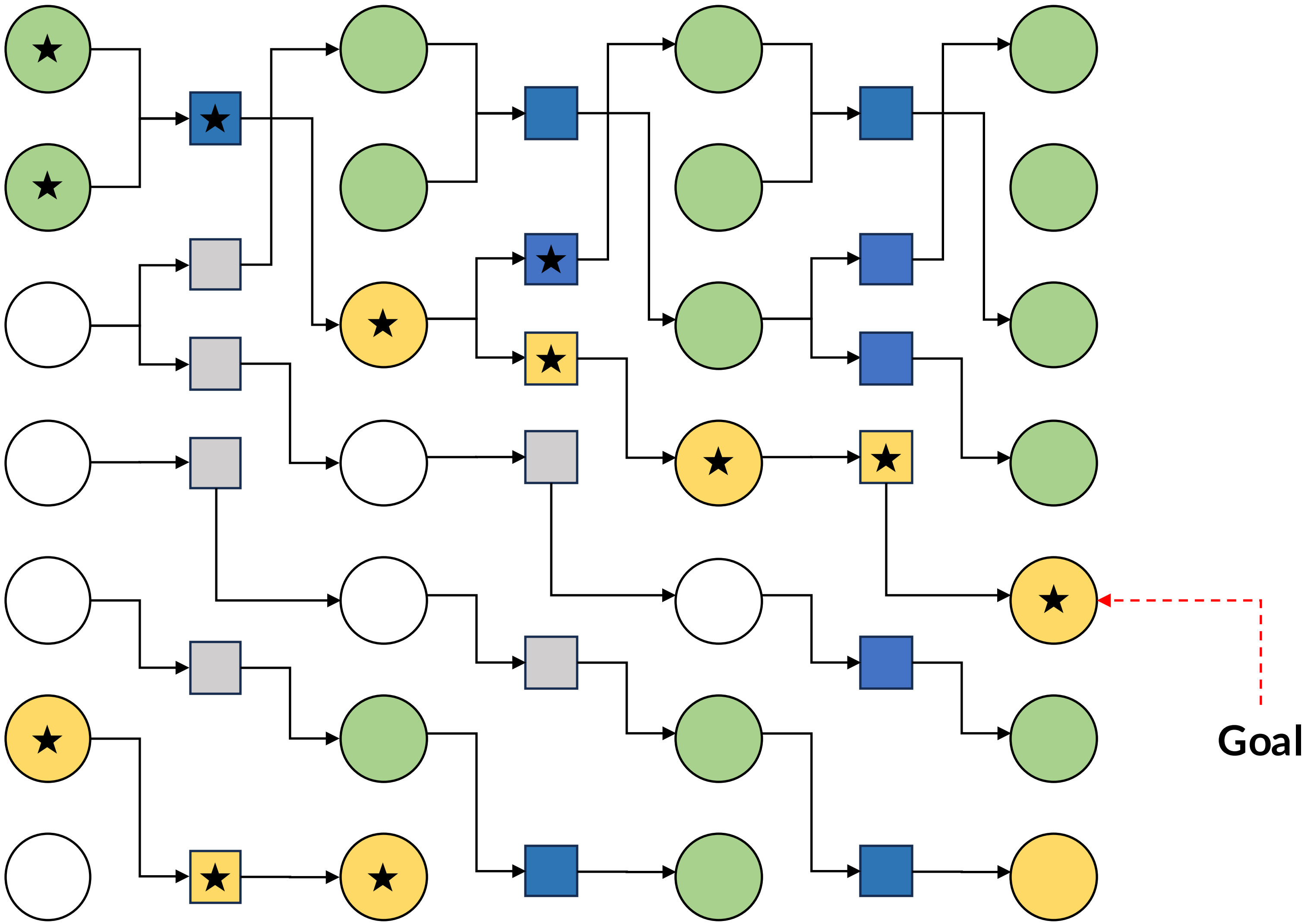

Goal

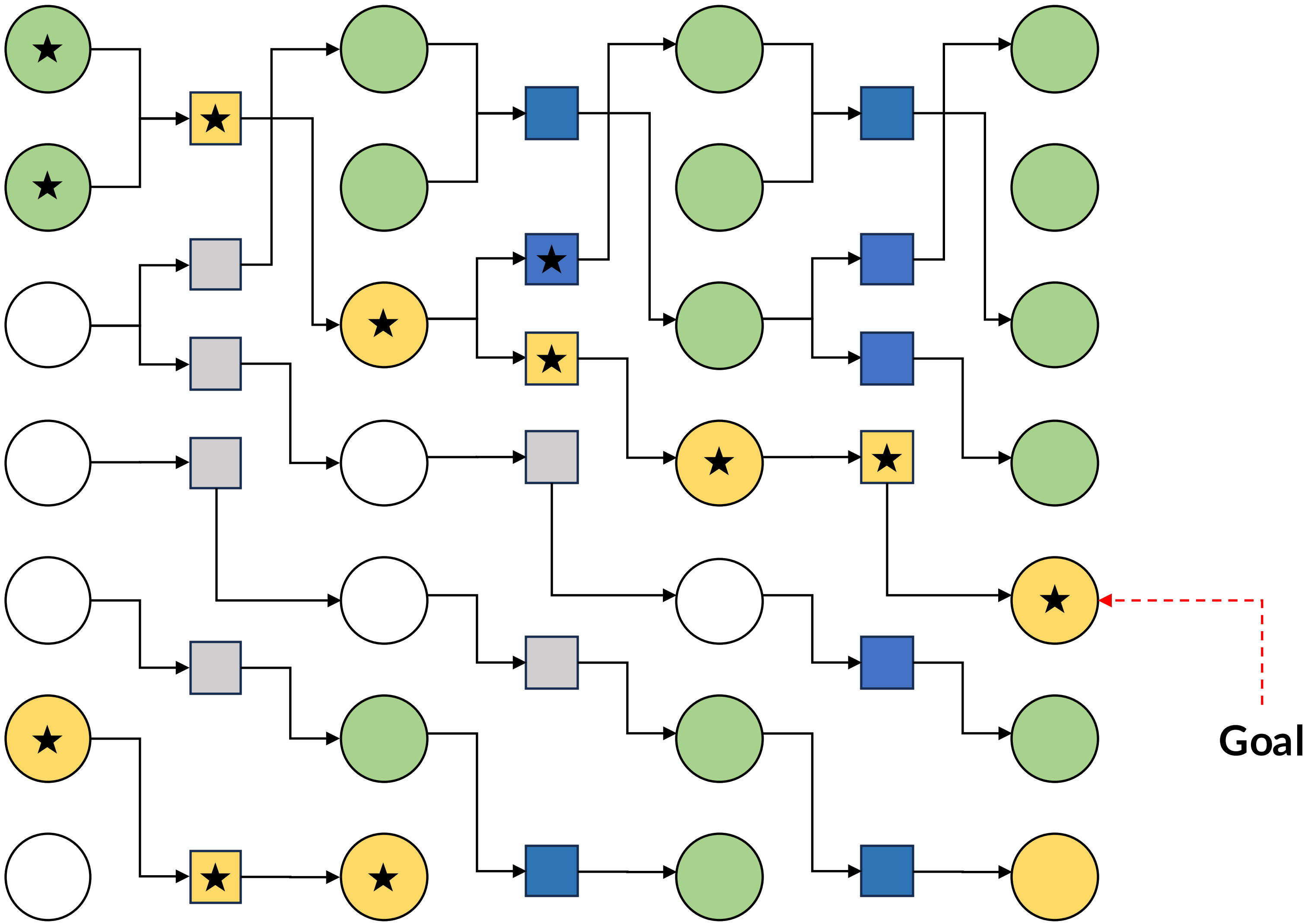

Goal


Goal

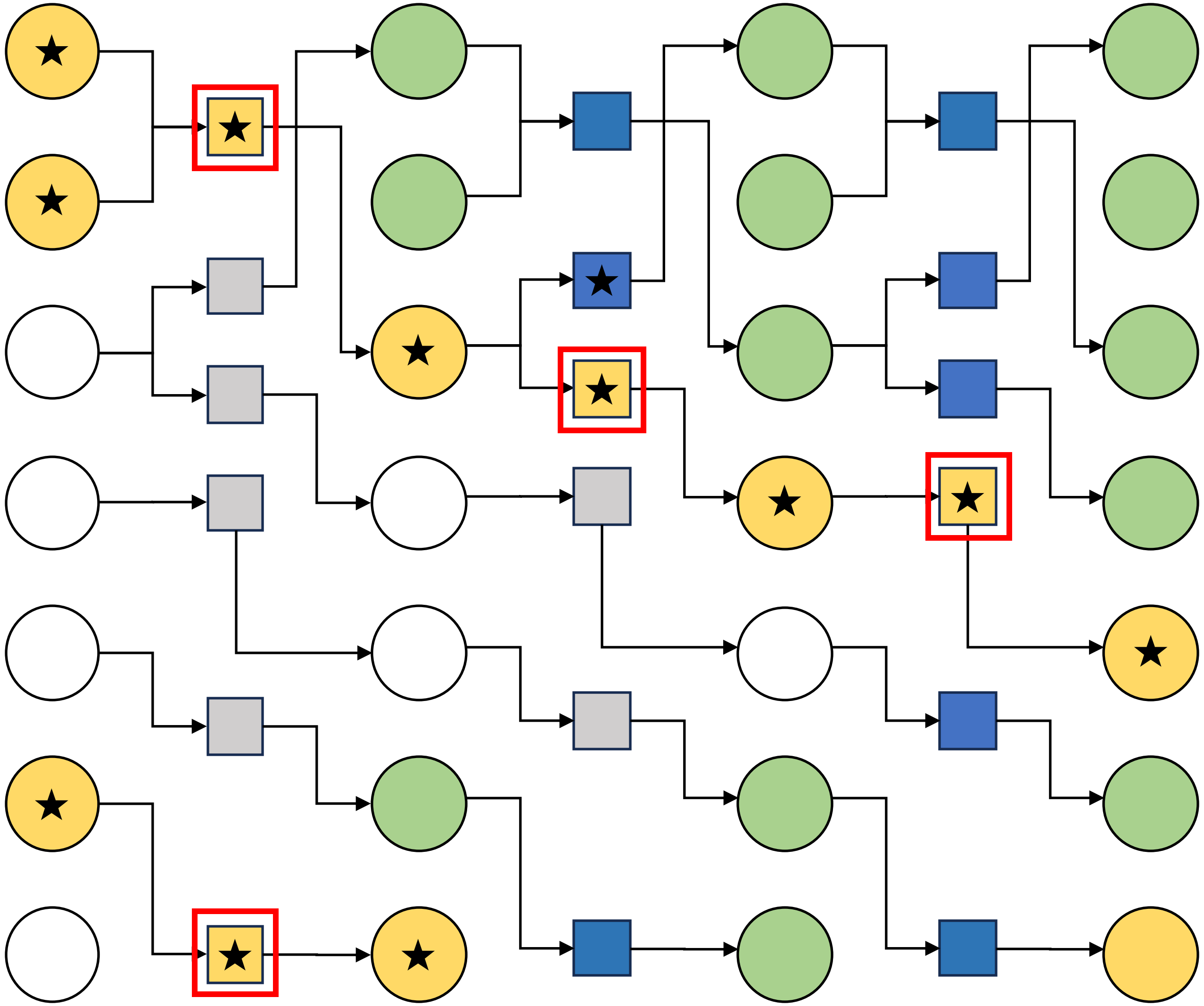

Goal


Goal


Goal

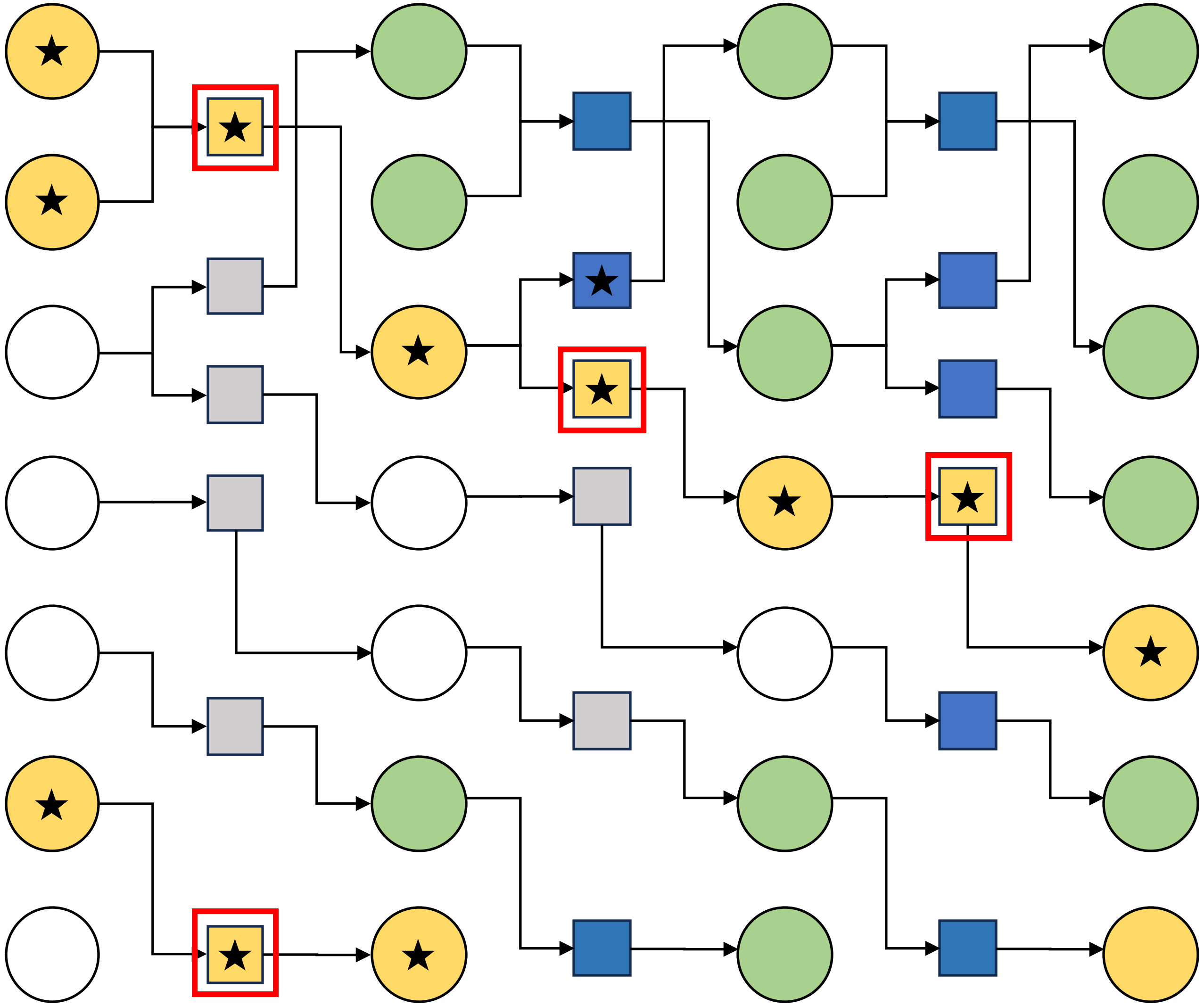

Goal

Relaxed Plan



Relaxed Plan

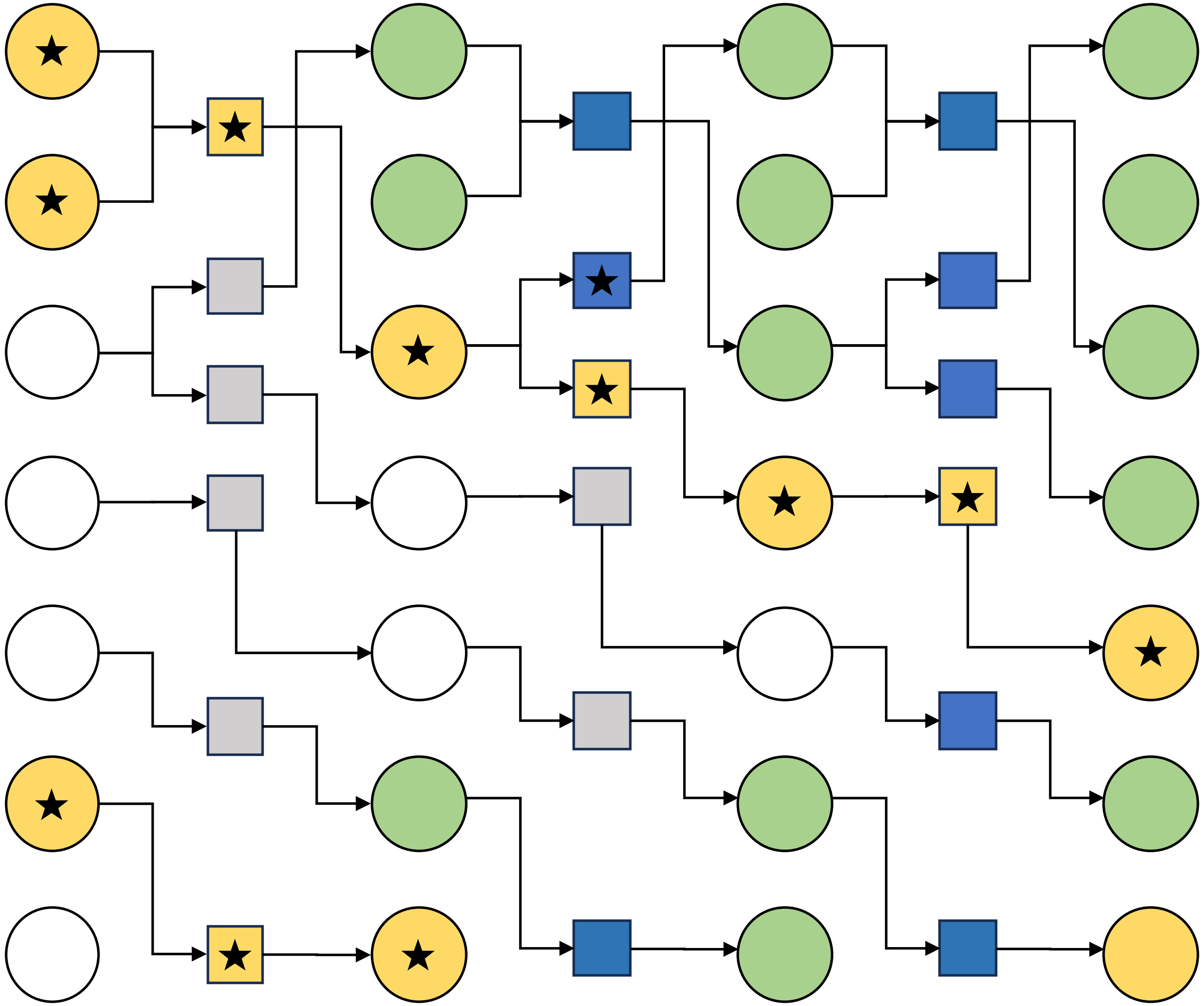
$V_{hFF} = 4$



Terminology: Relaxed Planning Graph

Also useful for testing *reachability*

Also useful for *pruning actions*



hFF Can Really Help!