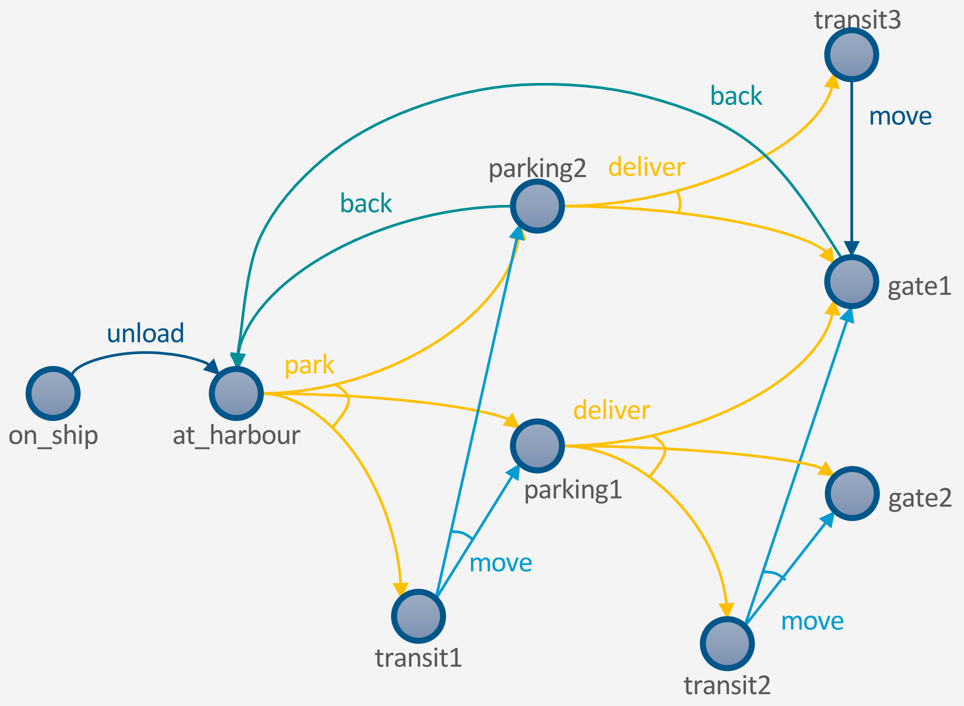
Universität Münster

Automated Planning and Acting

Nondeterministic Models



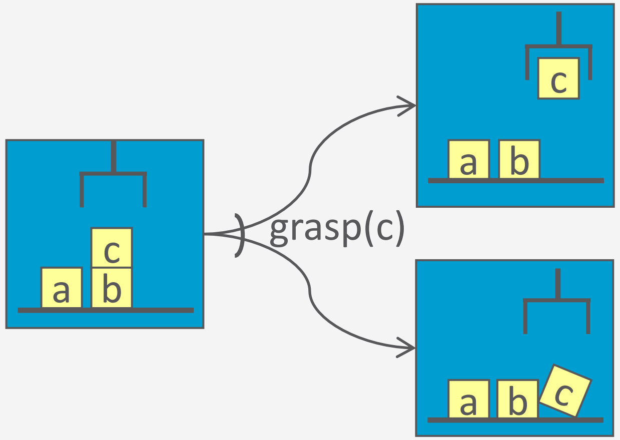
living.knowledge

Tanya Braun
Research Group Data Science, Computer Science Department

Content: Planning and Acting

Motivation

- We have assumed action a in state s has just one possible outcome
  - $\gamma(s, a)$
- Often more than one possible outcome
  - Unintended outcomes
  - Exogenous events
  - Inherent uncertainty



Outline per the Book

5.2 Planning Problem
- Planning domains
- Plans as policies
- Planning problems and solutions

5.3 And/Or Graph Search
- Planning by forward search

5.5 Determinisation Techniques
- Guided planning for safe solutions
- Planning for safe solutions by determinisation

5.6 Online Approaches
- Lookahead
- Lookahead by determinisation
- Lookahead with a bounded number of steps

Nondeterministic Planning Domains

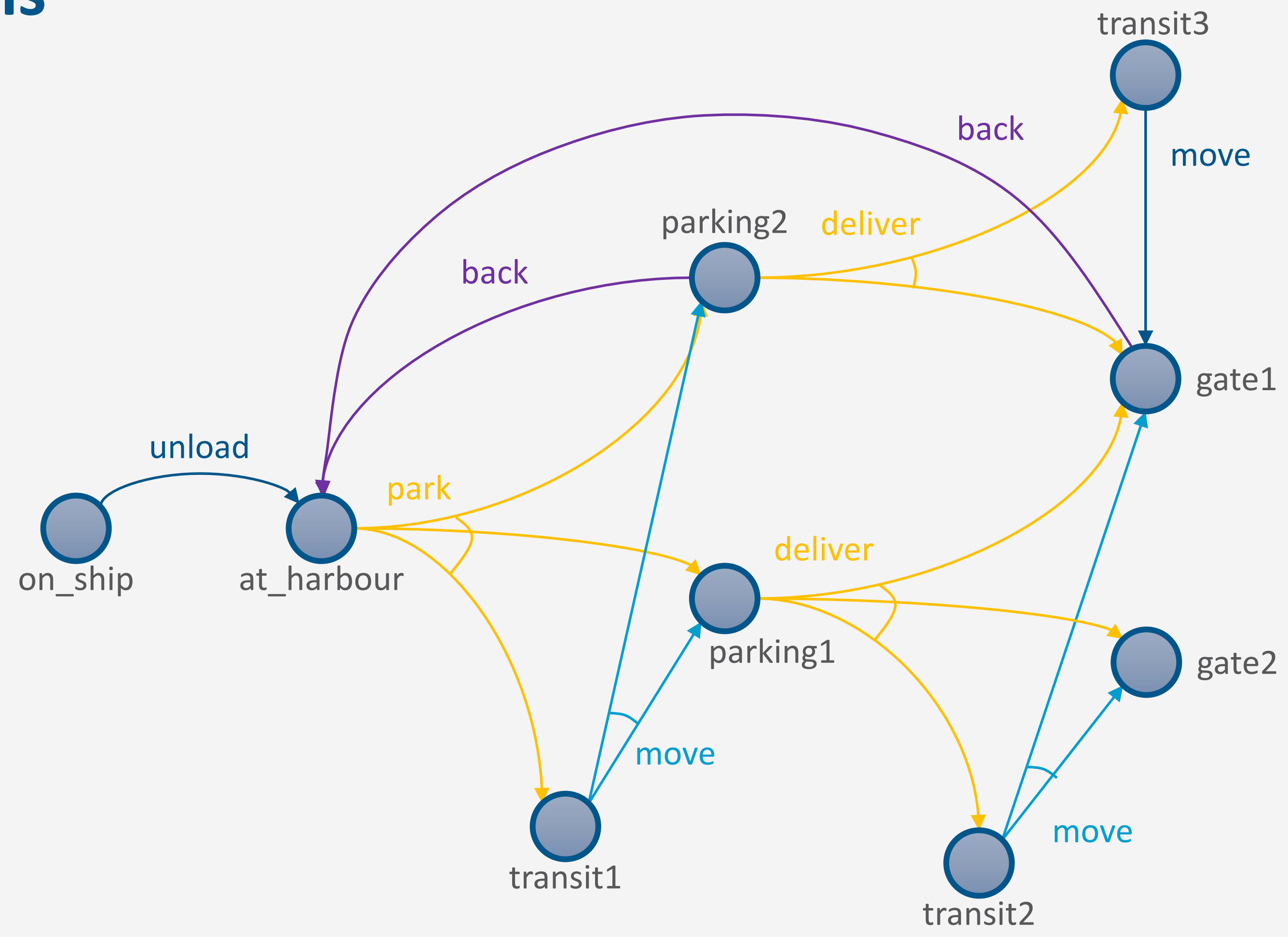
- Planning domain: 3-tuple (S, A, γ)
  - S and A – finite sets of states and actions
  - $\gamma : S \times A \rightarrow 2^S$
- $\gamma(s,a)$ = {all possible "next states" after applying action a in state s}
- a is applicable in state s iff $\gamma(s,a) \neq \emptyset$
- $Applicable(s) = \{\text{all actions applicable in } s\} = \{a \in A | \gamma(s,a) \neq \emptyset\}$
- One possible action representation:
  - n mutually exclusive "effects" lists

- Problem: n may be combinatorically large
  - Suppose *a* can cause any possible combination of effects $e_1, e_2, \ldots, e_k$
  - Need $eff_1, eff_2, \ldots, eff_{2^k \triangleq n}$ effect lists
    - One for each possible combination of $e_1, e_2, \ldots, e_k$
  - *Section 5.4: a way to alleviate this*
- For now, ignore most of that
  - states, actions $\Leftrightarrow$ nodes, edges in a graph

$a(z_1, \ldots, z_k)$
    pre: $p_1, \ldots, p_m$
    eff$_1$: $e_{11}, e_{12}, \ldots$
    eff$_2$: $e_{21}, e_{22}, \ldots$
    $\vdots$
    eff$_n$: $e_{n1}, e_{n2}, \ldots$

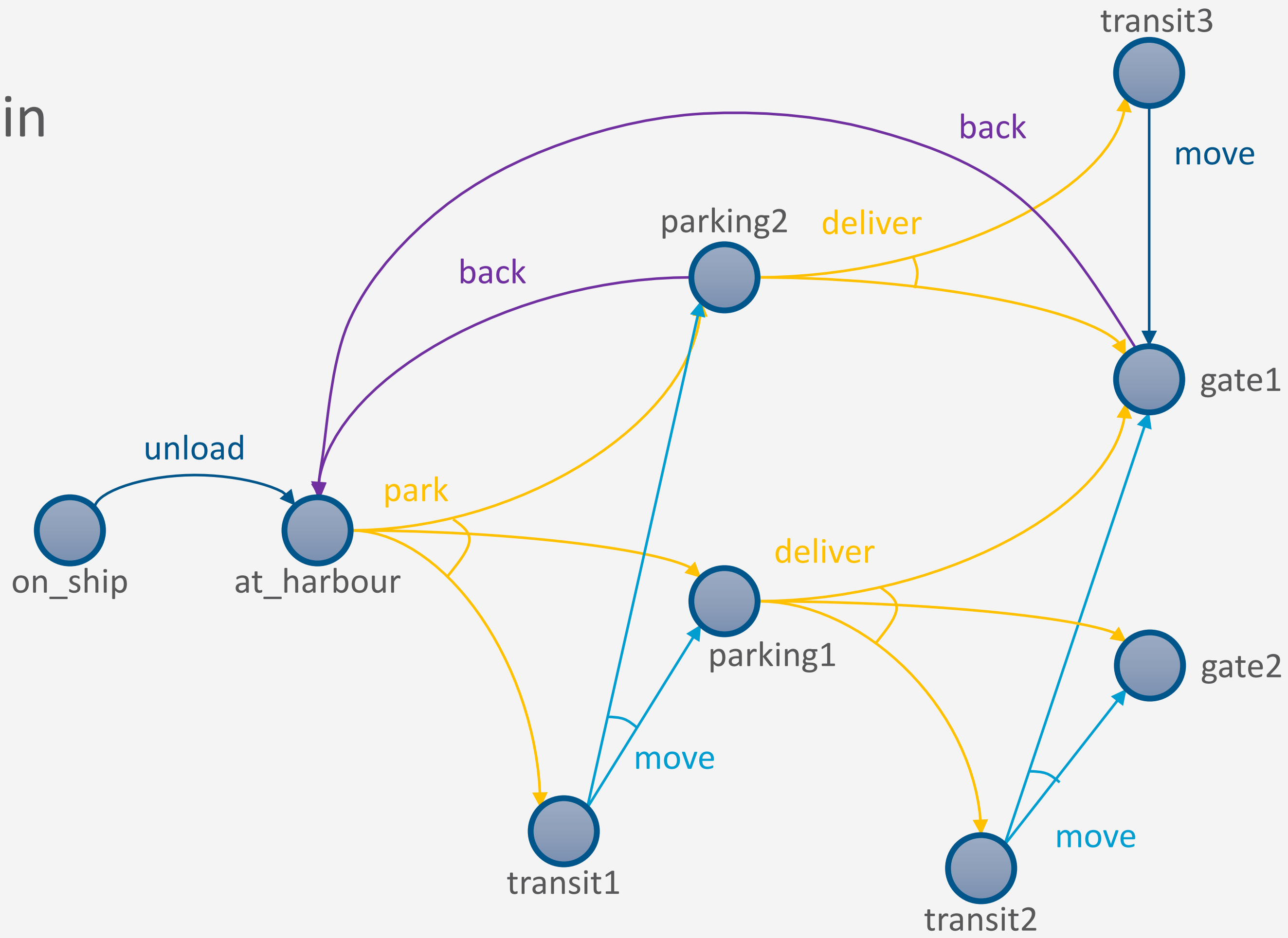
Nondeterministic Planning Domains

- For deterministic planning problems, search space was a graph
- Now it's an AND/OR graph
  - OR branch:
    - Several applicable actions, which one to choose?
  - AND branch:
    - Multiple possible outcomes
    - Must handle all of them
- Analogy to PSP
  - *OR* branch ⇔ resolver selection
  - *AND* branch ⇔ flaw selection



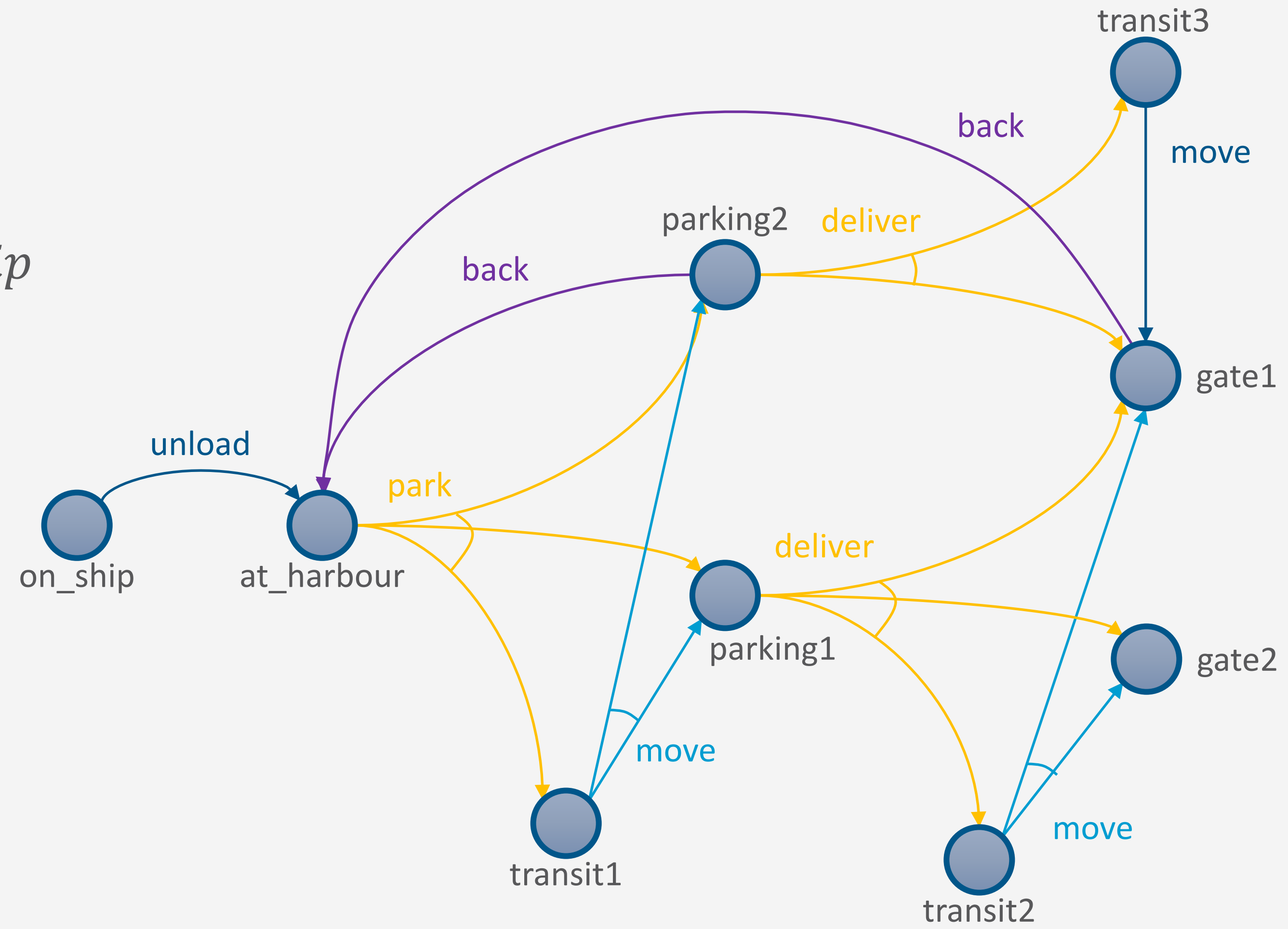
Example

- Very simple harbor management domain
  - Unload a single item from a ship
  - Move it around a harbor



Example

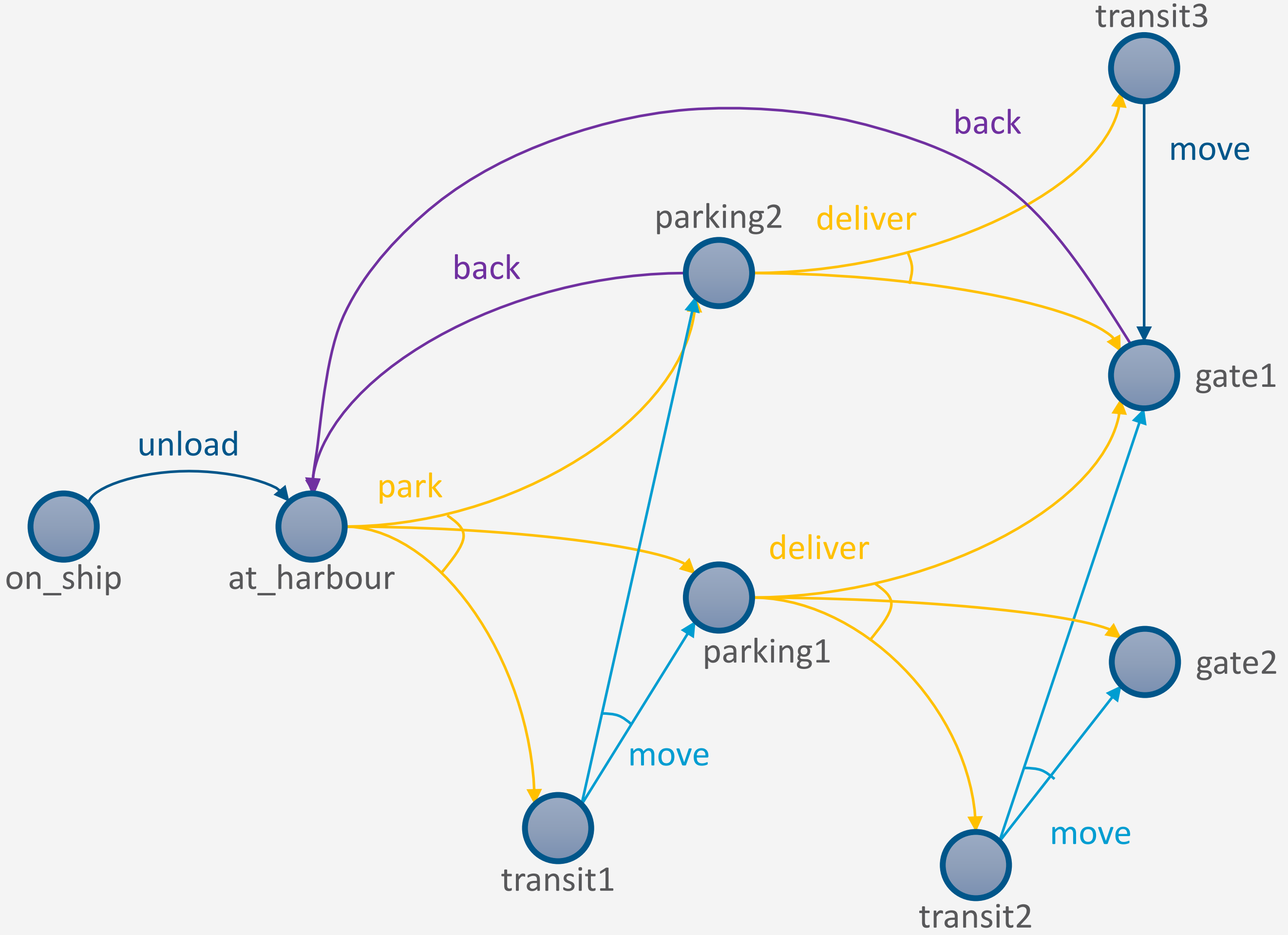
- One state variable: $pos(item)$
  - Simplified names for states
    - For $\{pos(item) = on\_ship\}$ write on_ship
- Five actions
  - Deterministic:
    - $unload$
    - $back$
    - ($move$ in $transit3$)
  - Nondeterministic:
    - $park$,
    - $move$,
    - $deliver$



Actions

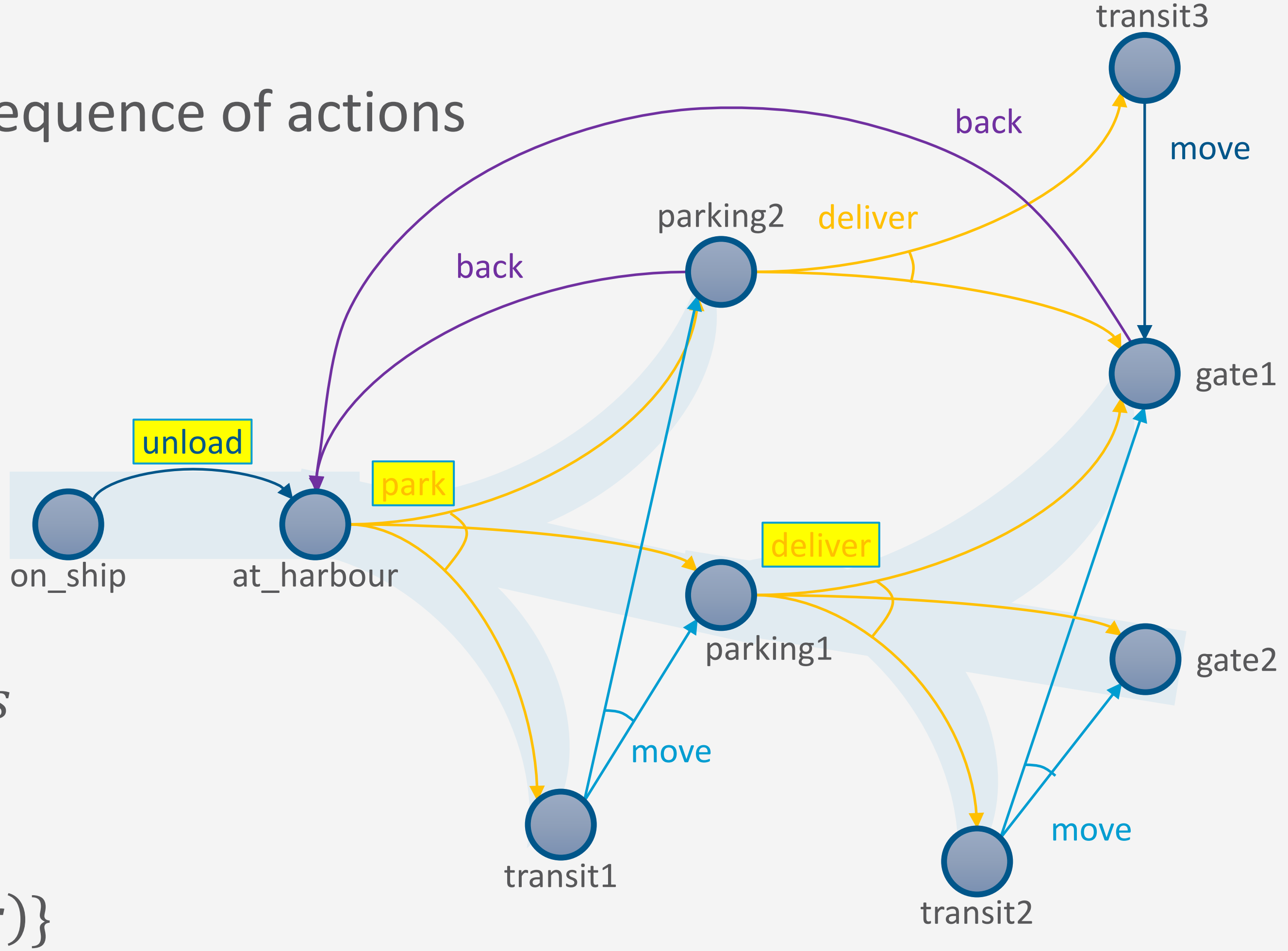
- Action example:
  - $park$

    pre: $pos(item) = at\_harbor$
    eff$_1$: $pos(item) \leftarrow parking1$
    eff$_2$: $pos(item) \leftarrow parking2$
    eff$_3$: $pos(item) \leftarrow transit1$
- Three possible outcomes
  - Put item in $parking1$ or $parking2$ if one of them has space or
  - in $transit1$ if there is no parking space



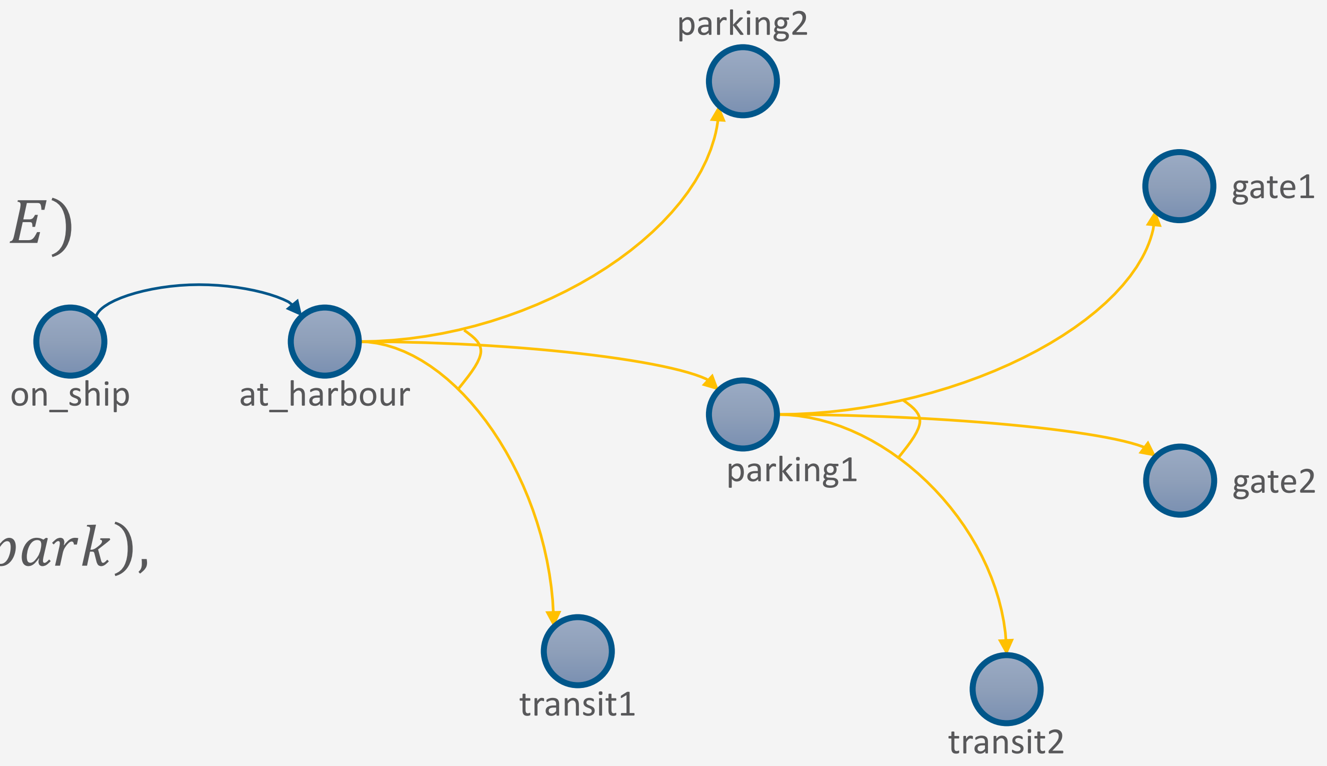
~~Plans~~ Policies

- Need something more general than a sequence of actions
  - After park, what do we do next?
- Policy: a *partial* function $\pi : S \mapsto A$
  - i.e., $\mathrm{dom}(\pi) \subseteq S$
    - Domain: values for which π defined
  - For every $s \in \mathrm{dom}(\pi)$, require $\pi(s) \in Applicable(s)$
- Meaning:
  - Perform $\pi(s)$ whenever we are in state s
- Example
  - $\pi_1 = \{(on\_ship, unload), (at\_harbor, park), (parking1, deliver)\}$



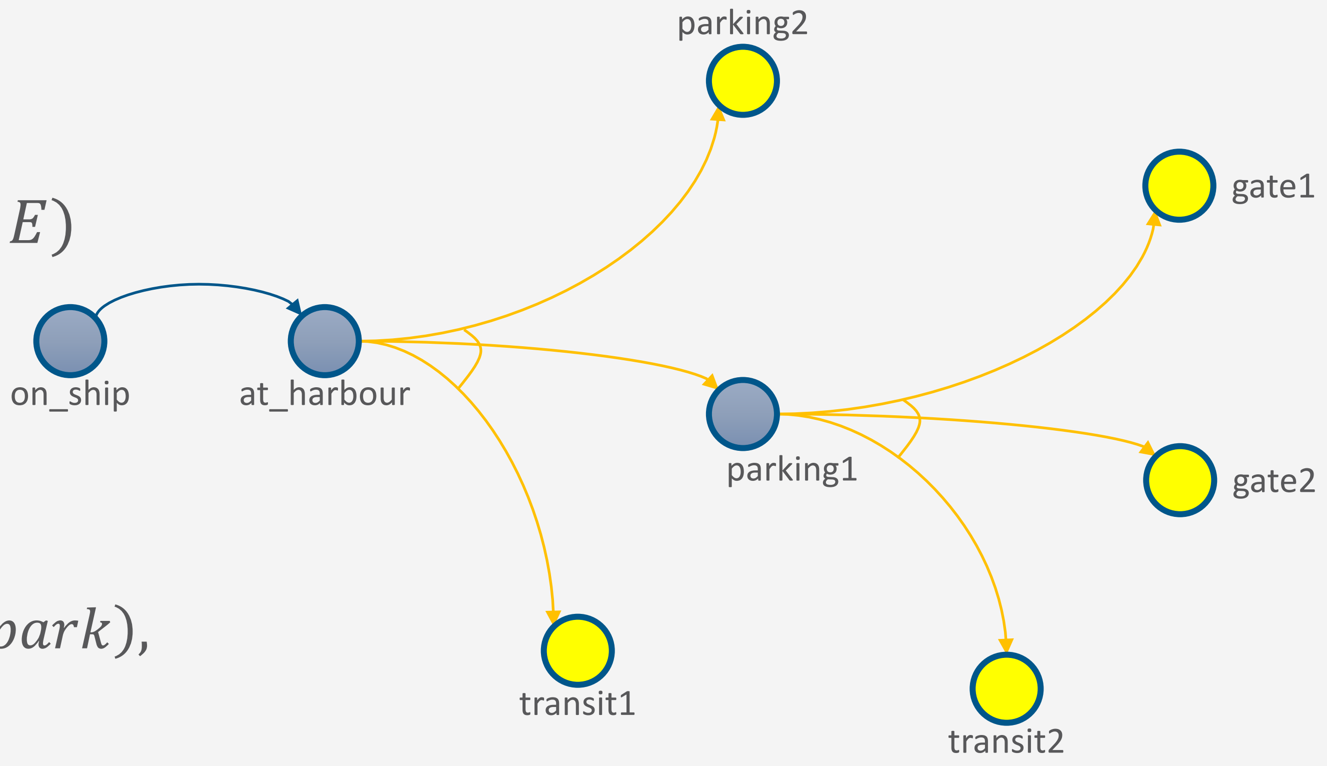
Definitions Over Policies

- Transitive closure $\hat{\gamma}(s,\pi)$ = {all states reachable from s using π}
  - $\hat{\gamma}(s,\pi) = S_0 \cup S_1 \cup S_2 \cup \cdots$
    - $S_0 = \{s\}$
    - $S_{i+1} = \bigcup\{\gamma(s,\pi(s)) \mid s \in S_i\}, i \geq 0$
- Reachability graph $Graph(s,\pi) = (V,E)$
  - $V = \hat{\gamma}(s,\pi)$
  - $E = \{(s',s'') | s' \in V, s'' \in \gamma(s',\pi(s'))\}$
- Example
  - $\pi_1 = \{(on\_ship, unload), (at\_harbor, park), (parking1, deliver)\}$
  - $Graph(on\_ship, \pi_1)$



Definitions Over Policies

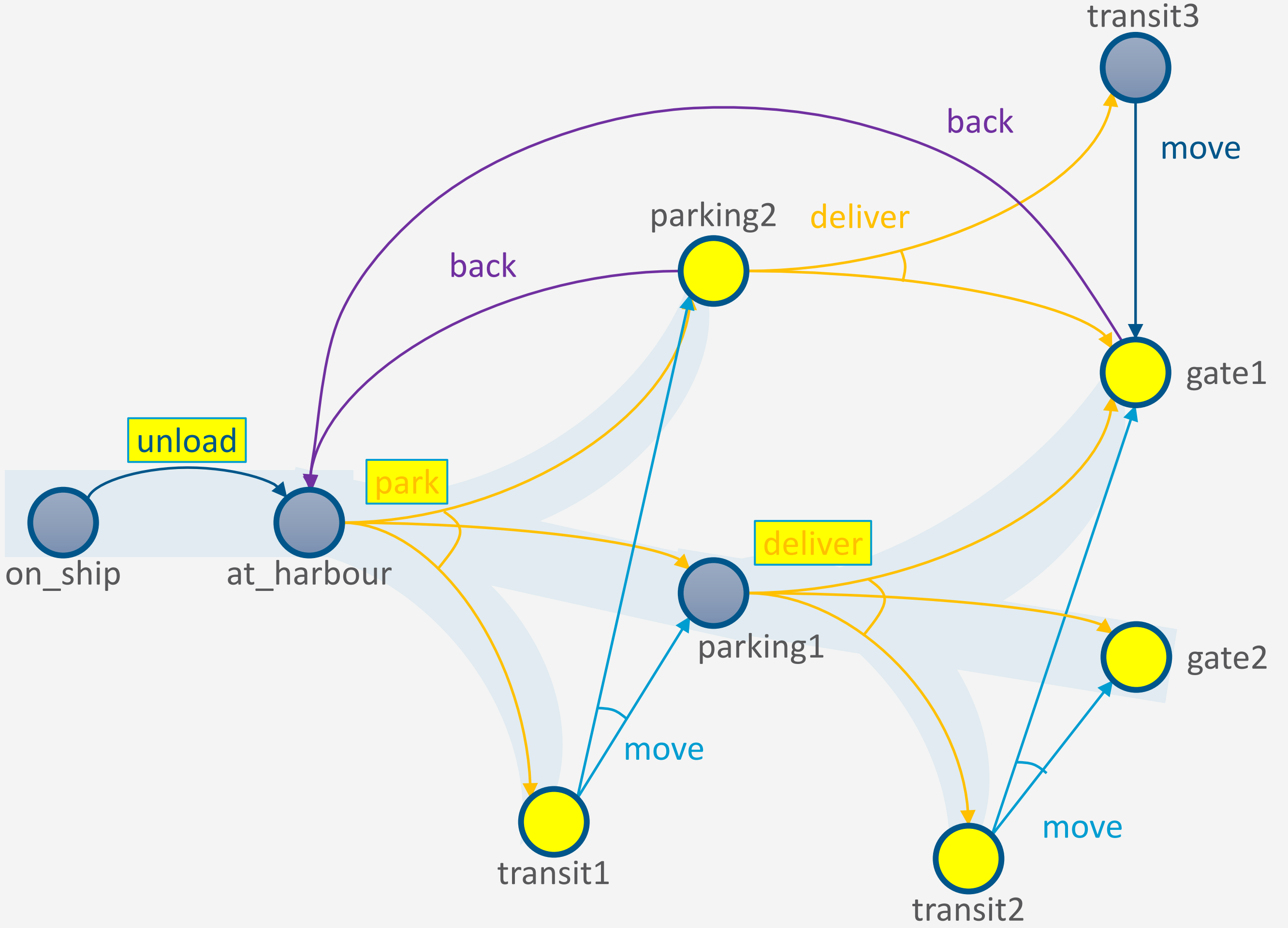
- Transitive closure $\hat{\gamma}(s,\pi)$ = {all states reachable from s using π}
  - $\hat{\gamma}(s,\pi) = S_0 \cup S_1 \cup S_2 \cup \cdots$
    - $S_0 = \{s\}$
    - $S_{i+1} = \bigcup\{\gamma(s,\pi(s)) \mid s \in S_i\}, i \geq 0$
- Reachability graph $Graph(s,\pi) = (V,E)$
  - $V = \hat{\gamma}(s,\pi)$
  - $E = \{(s',s'') | s' \in V, s'' \in \gamma(s',\pi(s'))\}$
- $leaves(s,\pi) = \hat{\gamma}(s,\pi) \setminus Dom(\pi)$
- Example:
  - $\pi_1 = \{(on\_ship, unload), (at\_harbor, park), (parking1, deliver)\}$
  - $leaves(on\_ship, \pi_1)$ in bright yellow



Performing a Policy

```
PerformPolicy(π)
   s ← observe current state
   while s ∈ Dom(π) do
      perform action π(s)
      s ← observe current state
```

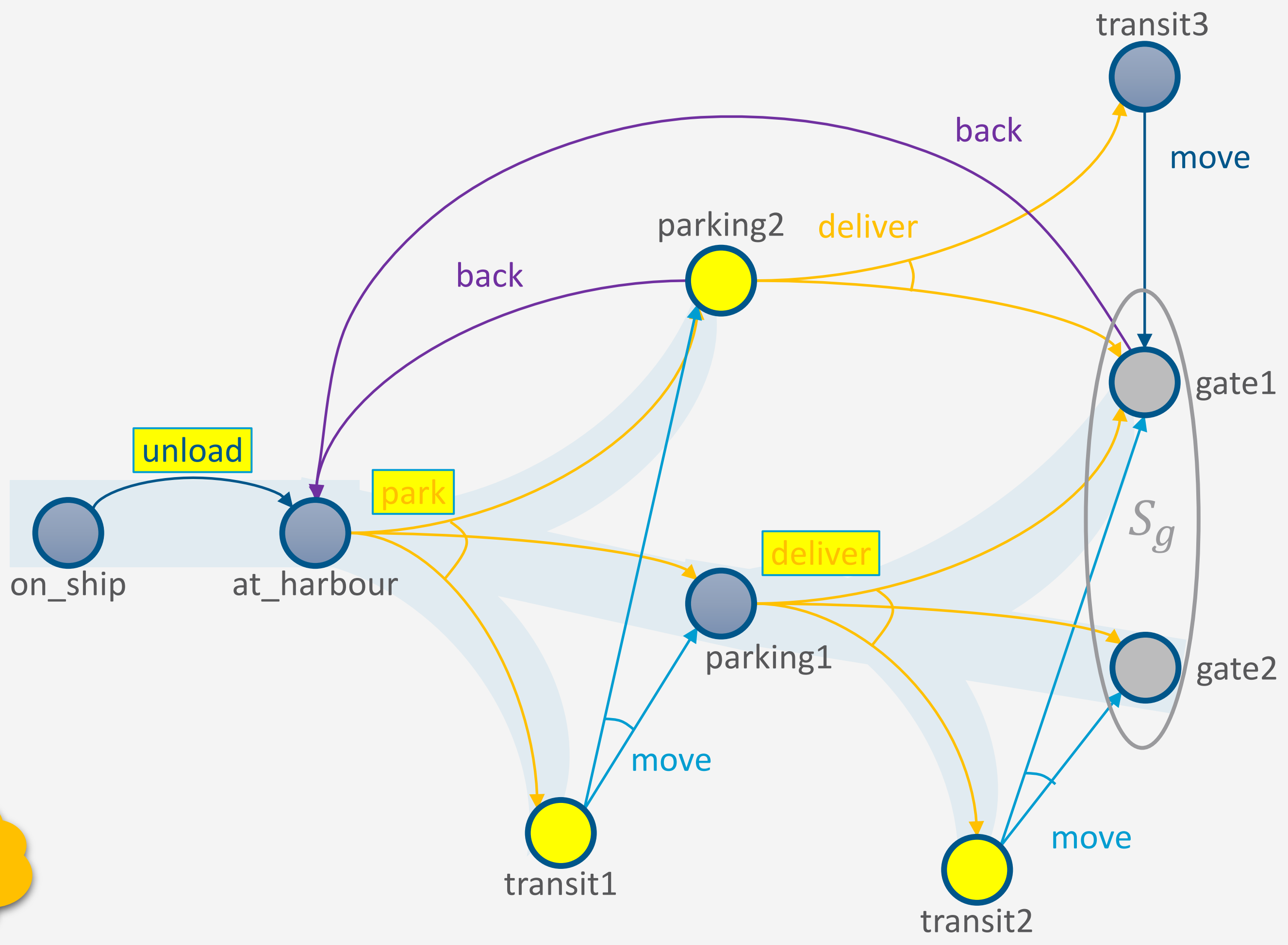
- $\pi_1 = \{(on\_ship, unload), (at\_harbor, park), (parking1, deliver)\}$



Planning Problems and Solutions

- Planning problem $P = (\Sigma, s_0, S_g)$
  - Planning domain $\Sigma = (S, A, \gamma)$
  - Initial state $s_0 \in S$
  - Set of goal states $S_g \subseteq S$ (shown in grey)
- π is a solution if at least one execution ends at a goal
  - $leaves(s_0, \pi) \cap S_g \neq \emptyset$
- Example
  - $\pi_1 = \{(on\_ship, unload), (at\_harbor, park), (parking1, deliver)\}$

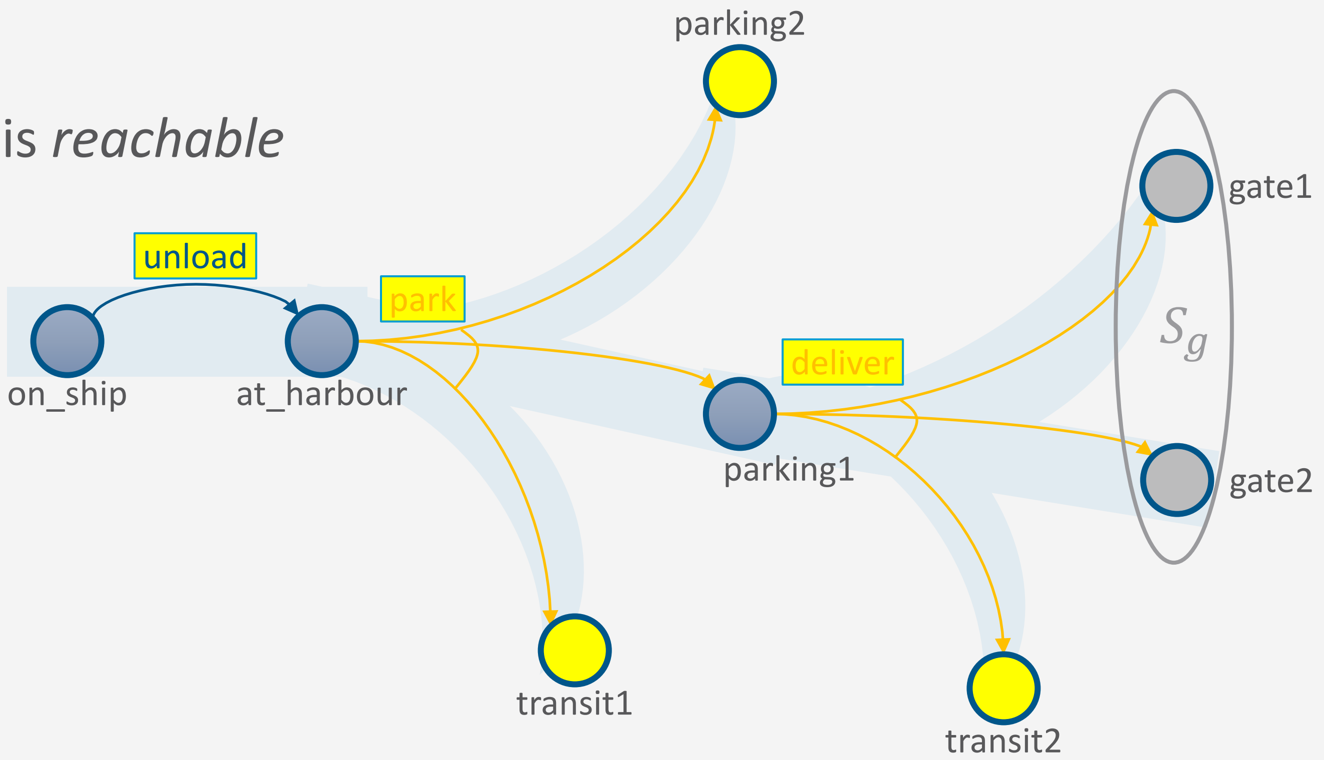
Is π_1 a solution?



Safe Solutions

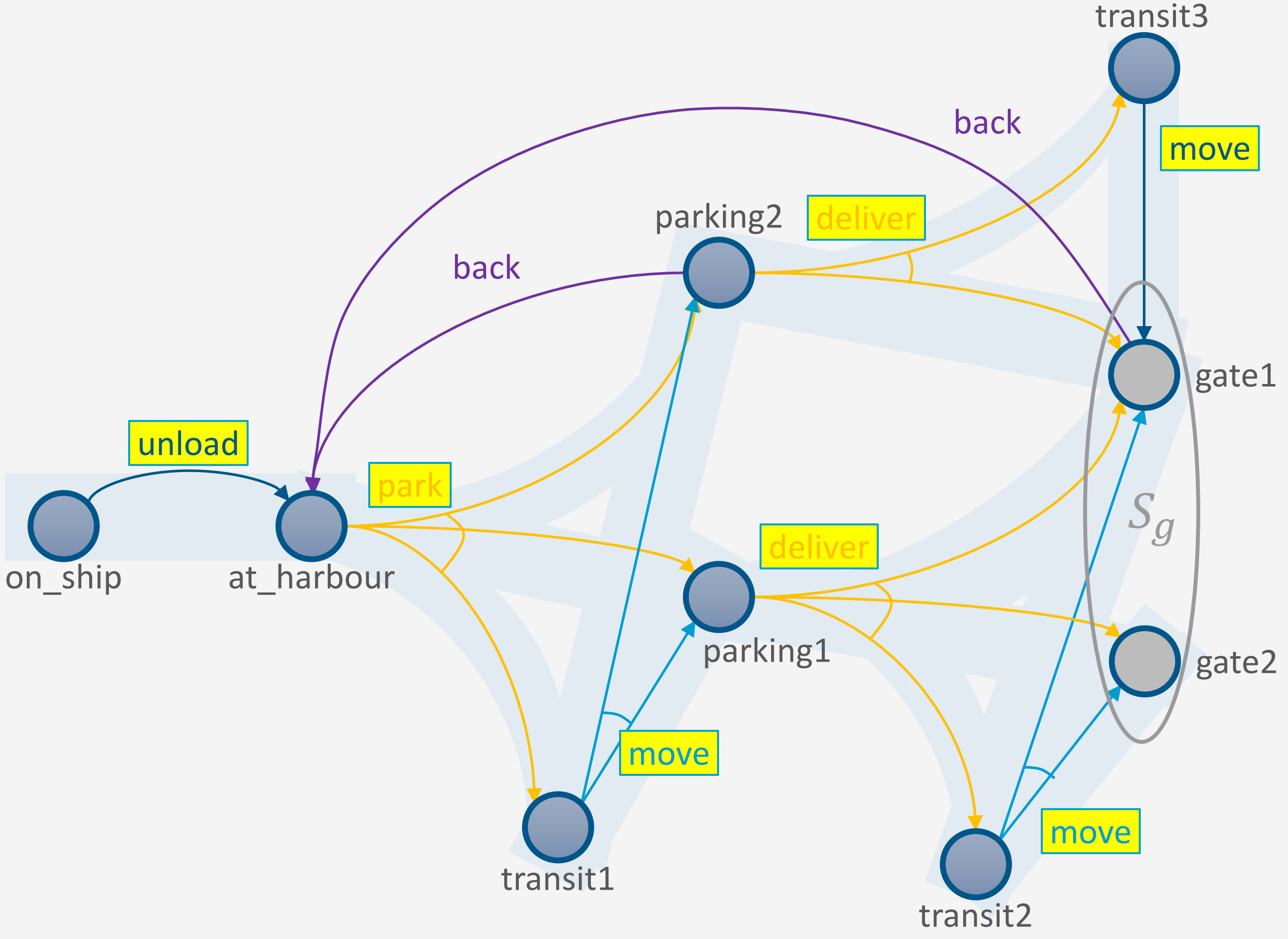
- A solution π is safe if
$$\forall s \in \hat{\gamma}(s_0, \pi),$$
$$leaves(s, \pi) \cap S_g \neq \emptyset$$
  - At every node of $Graph(s_0, \pi)$, the goal is *reachable*
- Otherwise, *unsafe*
- Example
  - $\pi_1 = \{(on\_ship, unload),$
    $(at\_harbor, park),$
    $(parking1, deliver)\}$

Is π_1 safe?



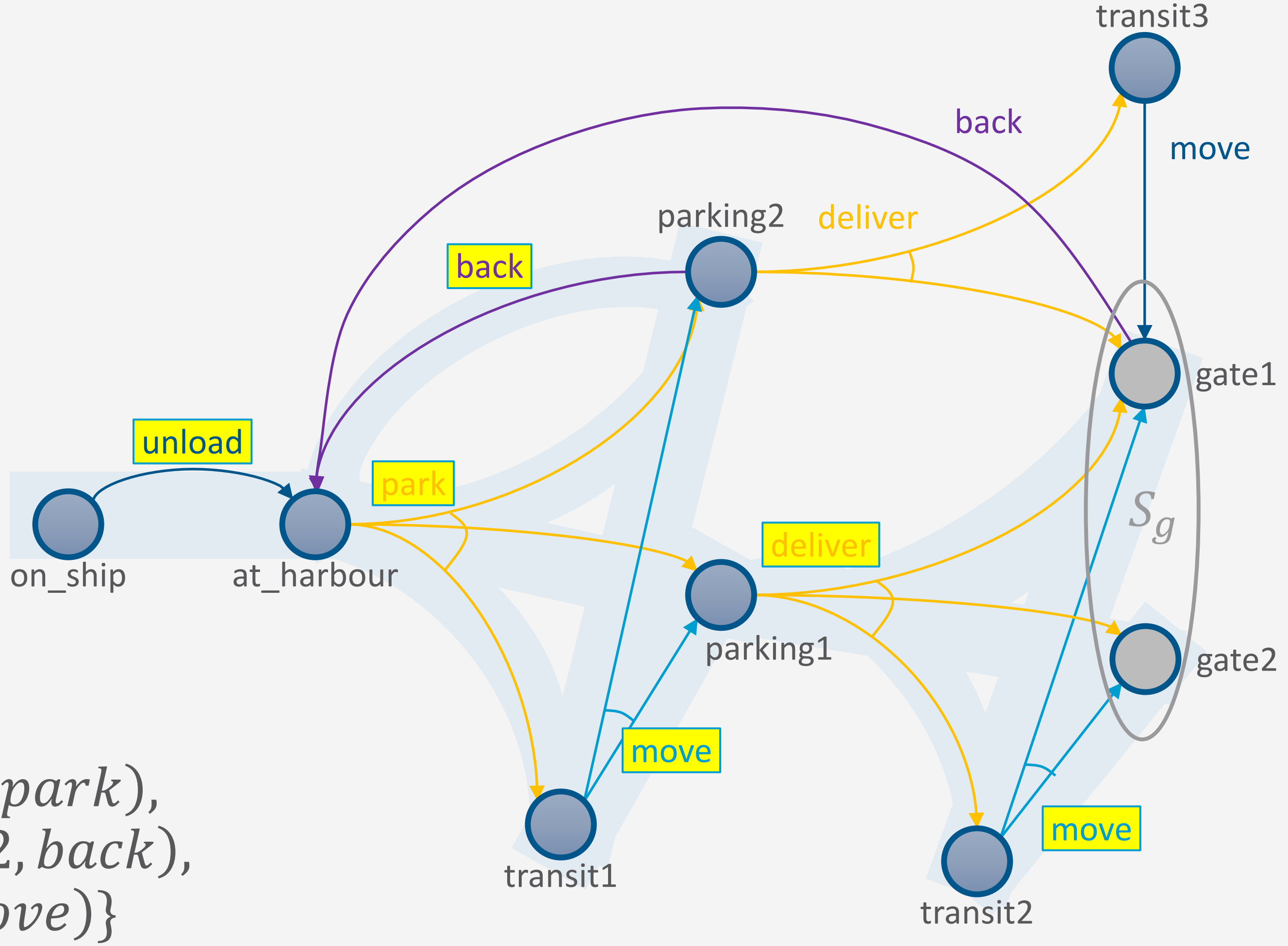
Safe Solutions

- $\pi_2 = \{(on\_ship, unload),$
  $(at\_harbor, park),$
  $(parking1, deliver),$
  $(parking2, deliver),$
  $(transit1, move),$
  $(transit2, move),$
  $(transit3, move)\}$
- Acyclic safe solution
  - $Graph(s_0, \pi)$ is acyclic and
  - $leaves(s_0, \pi) \subseteq S_g$
- Guaranteed to reach a goal

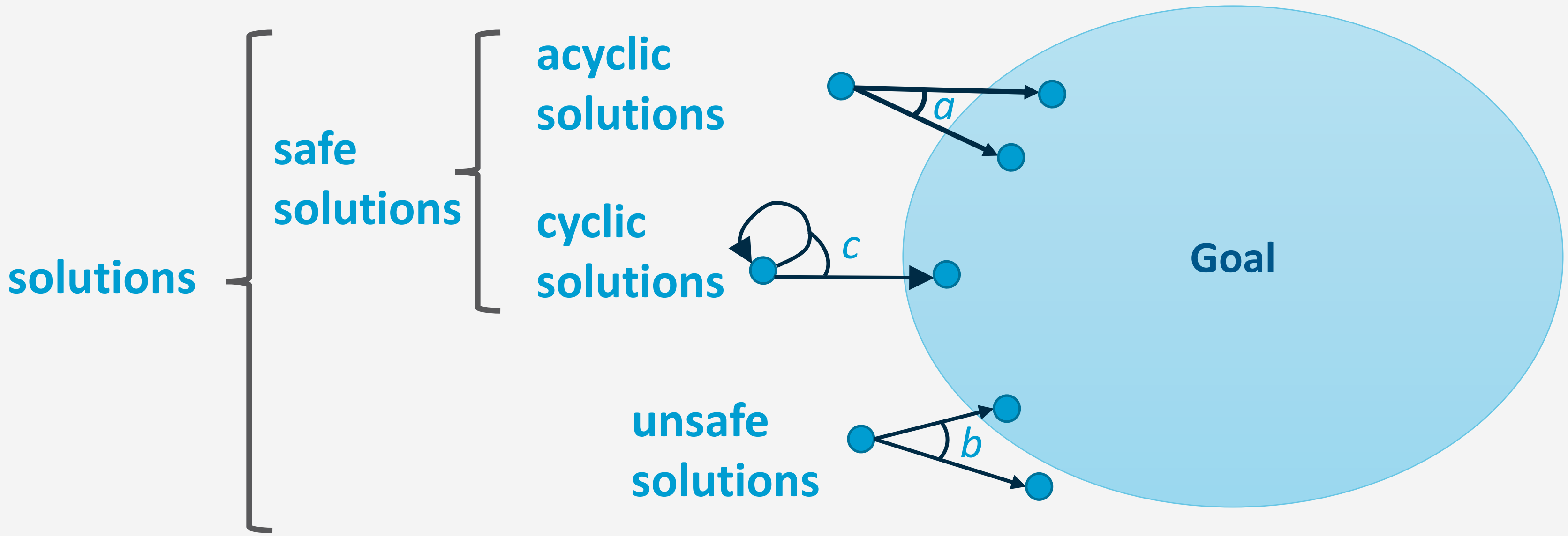


Safe Solutions

- Cyclic safe solution
  - $Graph(s_0, \pi)$ is cyclic,
  - $leaves(s_0, \pi) \subseteq S_g$, and
  - $\forall s \in \hat{\gamma}(s_0, \pi),$
    $leaves(s, \pi) \cap S_g \neq \emptyset$
- At every state, there is an execution path that ends at a goal
- Will never get caught in a dead end
- Example
  - $\pi_3 = \{(on\_ship, unload), (at\_harbor, park), (parking1, deliver), (parking2, back), (transit1, move), (transit2, move)\}$



Kinds of Solutions



Intermediate Summary

- Planning Problems
  - Planning domains
  - Plans as policies
  - Planning problems and solutions
    - Types of solutions: safe, unsafe, acyclic, cyclic

Outline per the Book

5.2 Planning Problem

- Planning domains
- Plans as policies
- Planning problems and solutions

5.3 And/Or Graph Search

- Planning by forward search

5.5 Determinisation Techniques

- Guided planning for safe solutions
- Planning for safe solutions by determinisation

5.6 Online Approaches

- Lookahead
- Lookahead by determinisation
- Lookahead with a bounded number of steps

Finding (Unsafe) Solutions

- Input: planning problem (Σ, s_0, S_g)

```
Find-Solution(Σ,s₀,S_g)
    s ← s₀
    π ← ∅
    Visited ← {s₀}
    loop
        if s ∈ S_g then
            return π
        A′ ← Applicable(s)
        if A′ = ∅ then
            return failure
        nondeterministically choose a ∈ A′
        nondeterministically choose s′ ∈ γ(s,a)
        if s′ ∈ Visited then
            return failure
        π(s) ← a
        Visited ← Visited ∪ {s′}
        s ← s′
```

Decide which state to plan for

Cycle-checking

```
Forward-search(Σ,s₀,g)
    s ← s₀
    π ← ⟨⟩
    loop
        if s satisfies g then
            return π
        A′ ←{a ∈ A | a is applicable in s}
        if A′ = ∅ then
            return failure
        nondeterministically choose a ∈ A′
        s ← γ(s,a)
        π ← π.a
```

For comparison: Forward-search with *deterministic* models

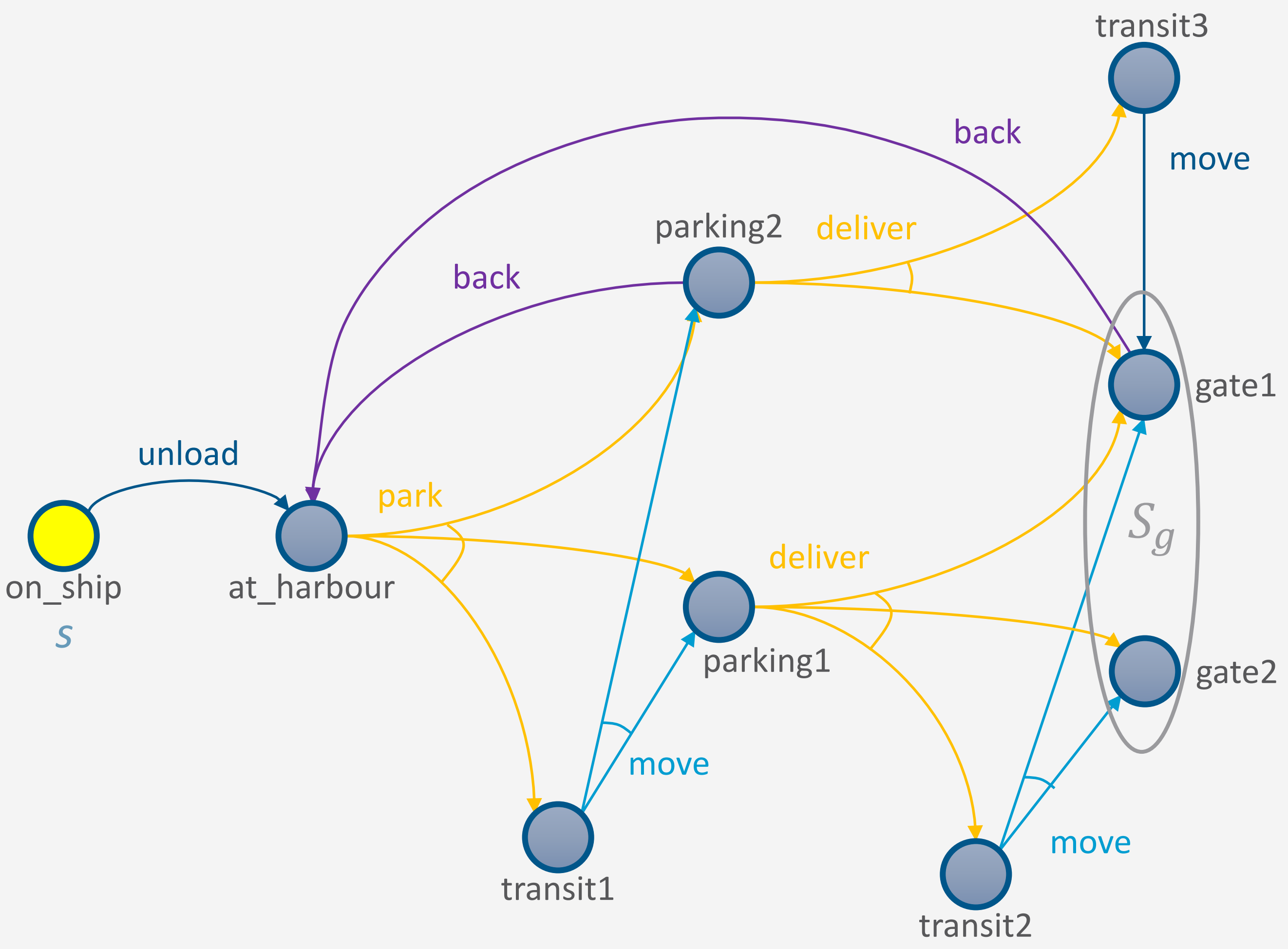
Example

```
Find-Solution(Σ,s0,Sg)
    s ← s0
    π ← ∅
    Visited ← {s0}
    ...
```

s = on_ship

$\pi = \{\}$

$Visited = \{\text{on\_ship}\}$

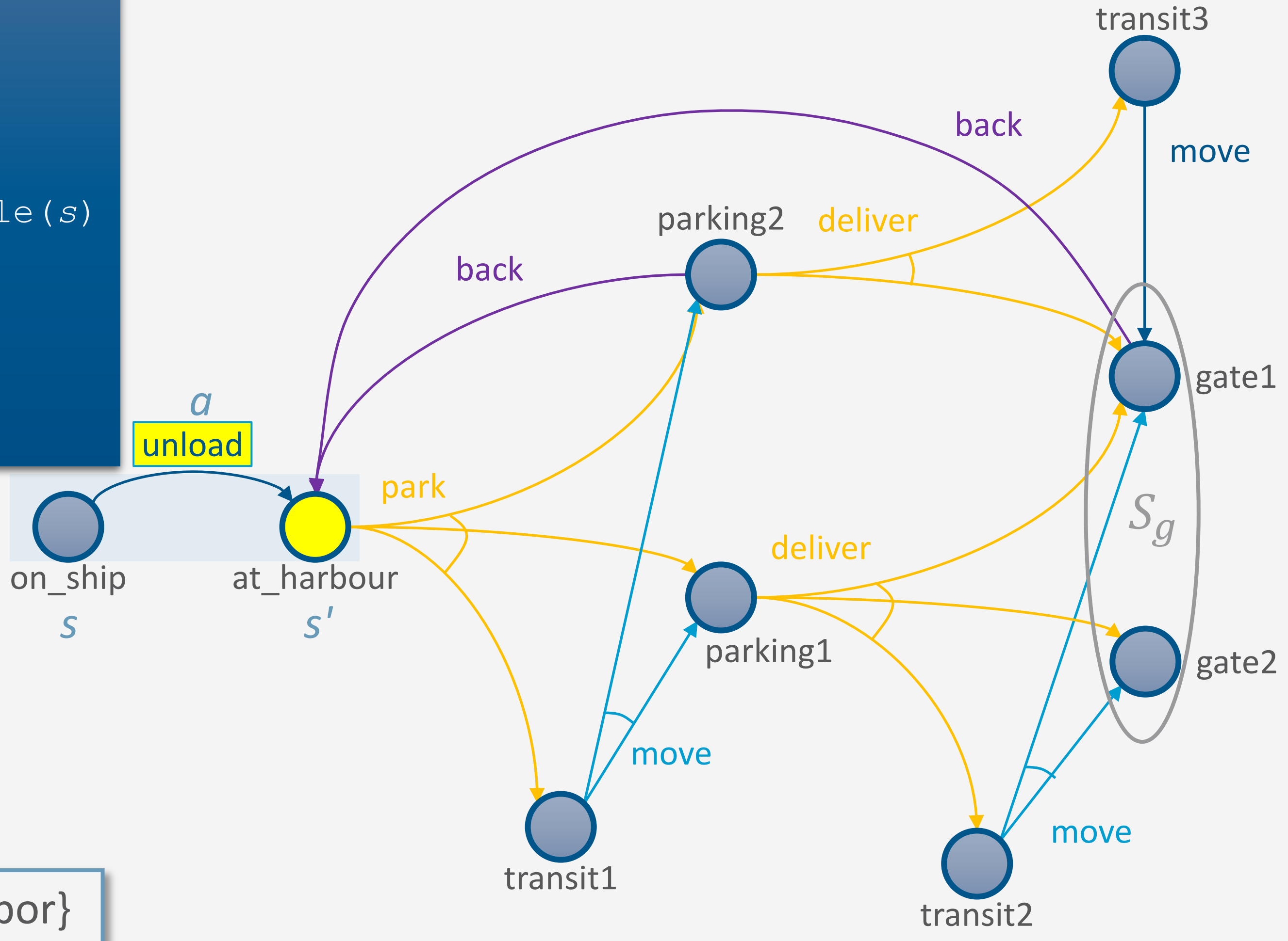


```
Find-Solution(Σ,s0,Sg)
    ...
    loop
        if s ∈ Sg then
            return π
        ...
        nondeterministically choose a ∈ Applicable(s)
        nondeterministically choose s′ ∈ γ(s,a)
        ...
        π(s) ← a
        Visited ← Visited ∪ {s′}
        s ← s′
```

s = on_ship, a = unload
$\gamma(s,a)$ = {at_harbor}
s' = at_harbor

π = {(on_ship, unload)}

Visited = {on_ship, at_harbor}

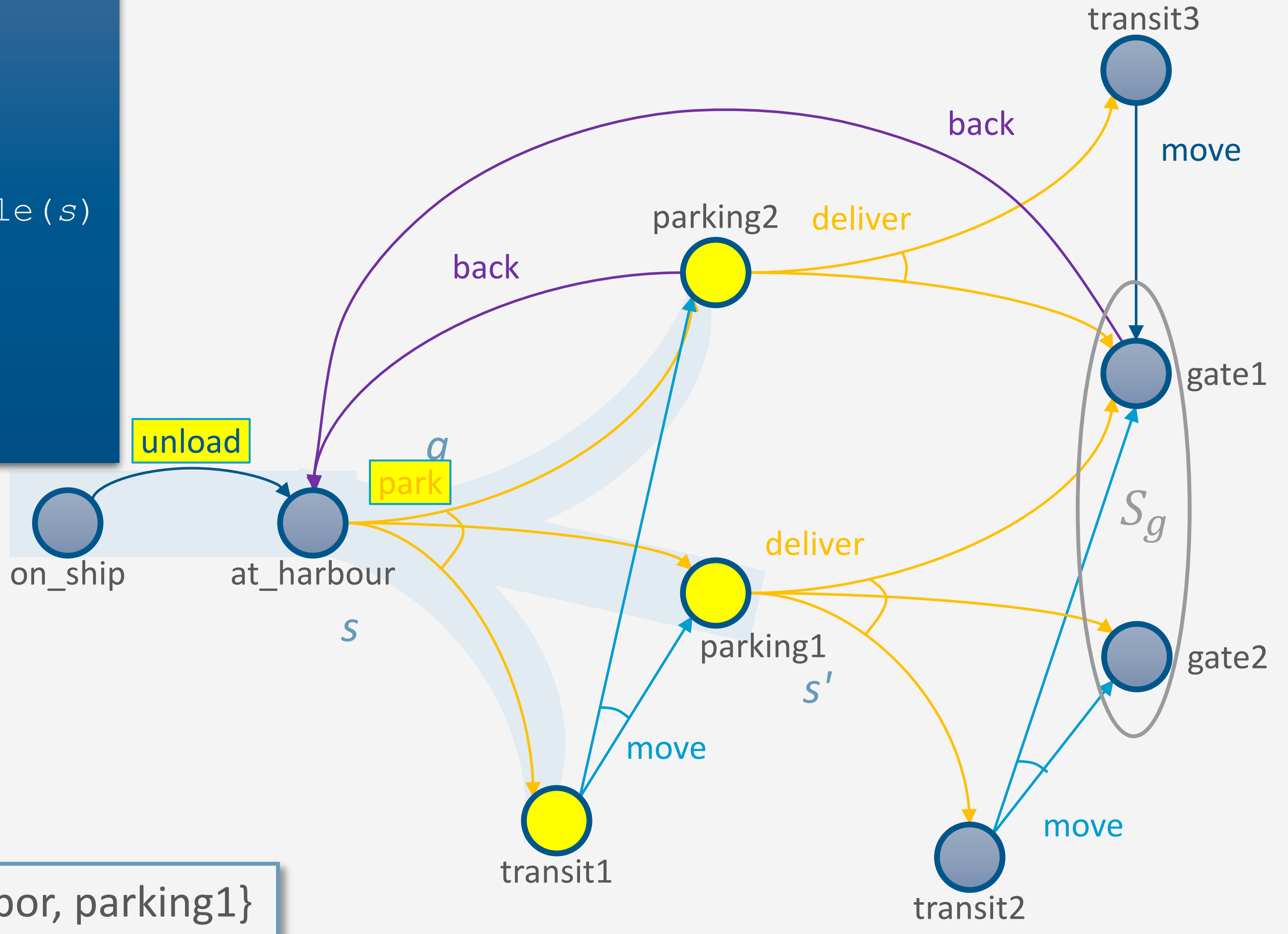


```
Find-Solution(Σ,s0,Sg)
    ...
    loop
        if s ∈ Sg then
            return π
        ...
        nondeterministically choose a ∈ Applicable(s)
        nondeterministically choose s′ ∈ γ(s,a)
        ...
        π(s) ← a
        Visited ← Visited ∪ {s′}
        s ← s′
```

s = at_harbor, a = park
$\gamma(s,a)$ = {parking1, parking2, transit1}
s' = parking1

π = {(on_ship, unload), (at_harbor, park)}

Visited = {on_ship, at_harbor, parking1}

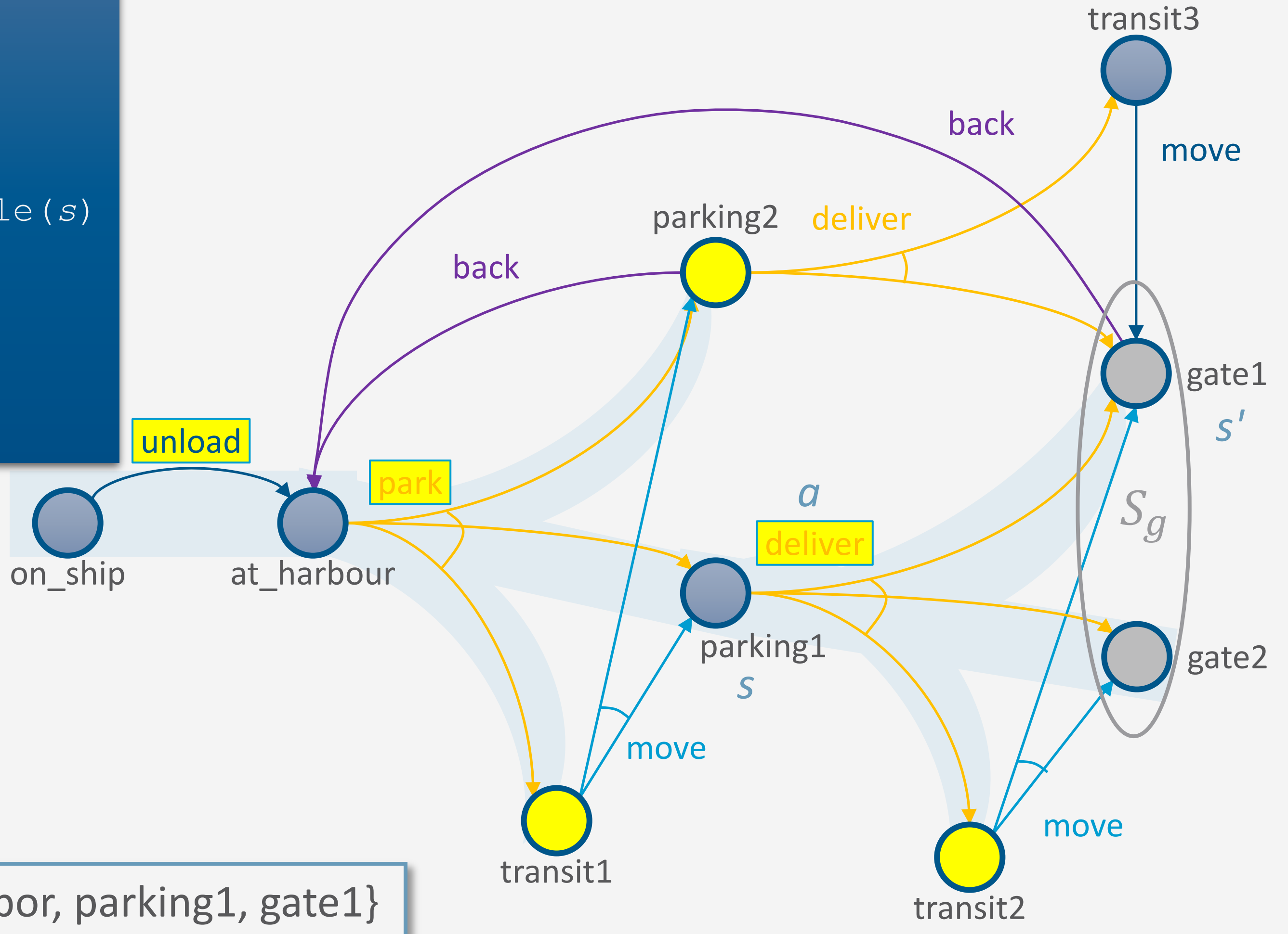


```
Find-Solution(Σ,s0,Sg)
    ...
    loop
        if s ∈ Sg then
            return π
        ...
        nondeterministically choose a ∈ Applicable(s)
        nondeterministically choose s′ ∈ γ(s,a)
        ...
        π(s) ← a
        Visited ← Visited ∪ {s′}
        s ← s′
```

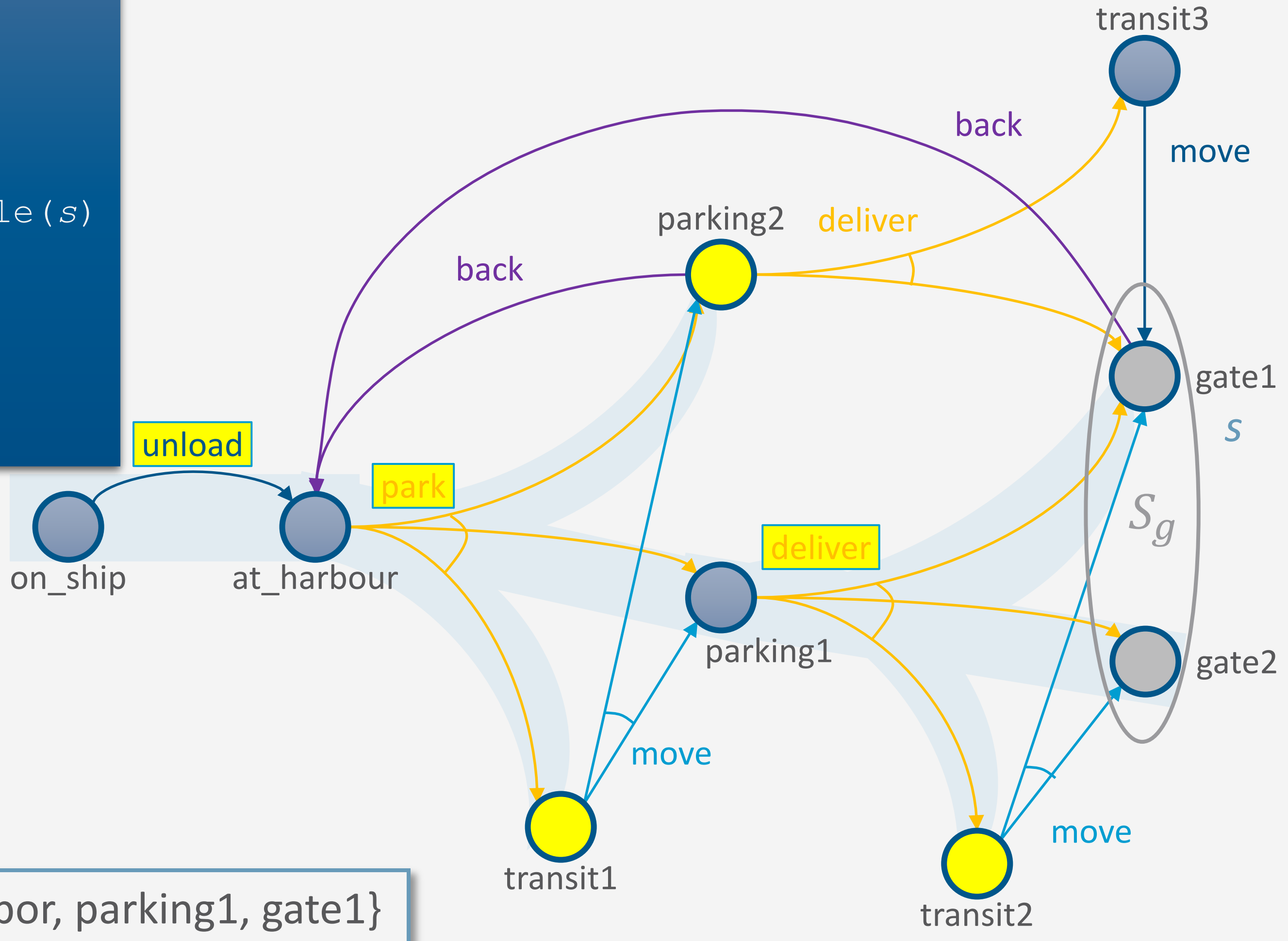
s = parking1, a = deliver
$\gamma(s,a)$ = {gate1, gate2, transit2}
s' = gate1

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver)}

Visited = {on_ship, at_harbor, parking1, gate1}



```
Find-Solution(Σ,s0,Sg)
    ...
    loop
        if s ∈ Sg then
            return π
        ...
        nondeterministically choose a ∈ Applicable(s)
        nondeterministically choose s' ∈ γ(s,a)
        ...
        π(s) ← a
        Visited ← Visited ∪ {s'}
        s ← s'
```



s = gate1

gate1 is a goal, so return *π*

π = {(on_ship, unload), (at_harbor, park), (parking1, deliver)}

Visited = {on_ship, at_harbor, parking1, gate1}

Finding Acyclic Safe Solutions

- Check for cycles
  - For each $s' \in (\gamma(s,a) \cap Dom(\pi))$
    - Is $s' \in \hat{\gamma}(s',\pi)$?
  - Formally, has−loops$(\pi, s, Frontier)$ iff $\exists s' \in (\gamma(s,a) \cap \mathrm{dom}(\pi)) : s' \in \hat{\gamma}(s',\pi)$
    - I.e., a state s' is reachable from itself

```
Find-Acyclic-Solution(Σ,s0,Sg)
    π ← ∅
    Frontier ← {s0}
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        if Applicable(s) = ∅ then
            return failure
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-loops(π,s,Frontier) then
            return failure
    return π
```

Keep track of unexpanded states, like in A*

Add all outcomes that π does not already handle

Cycle-checking

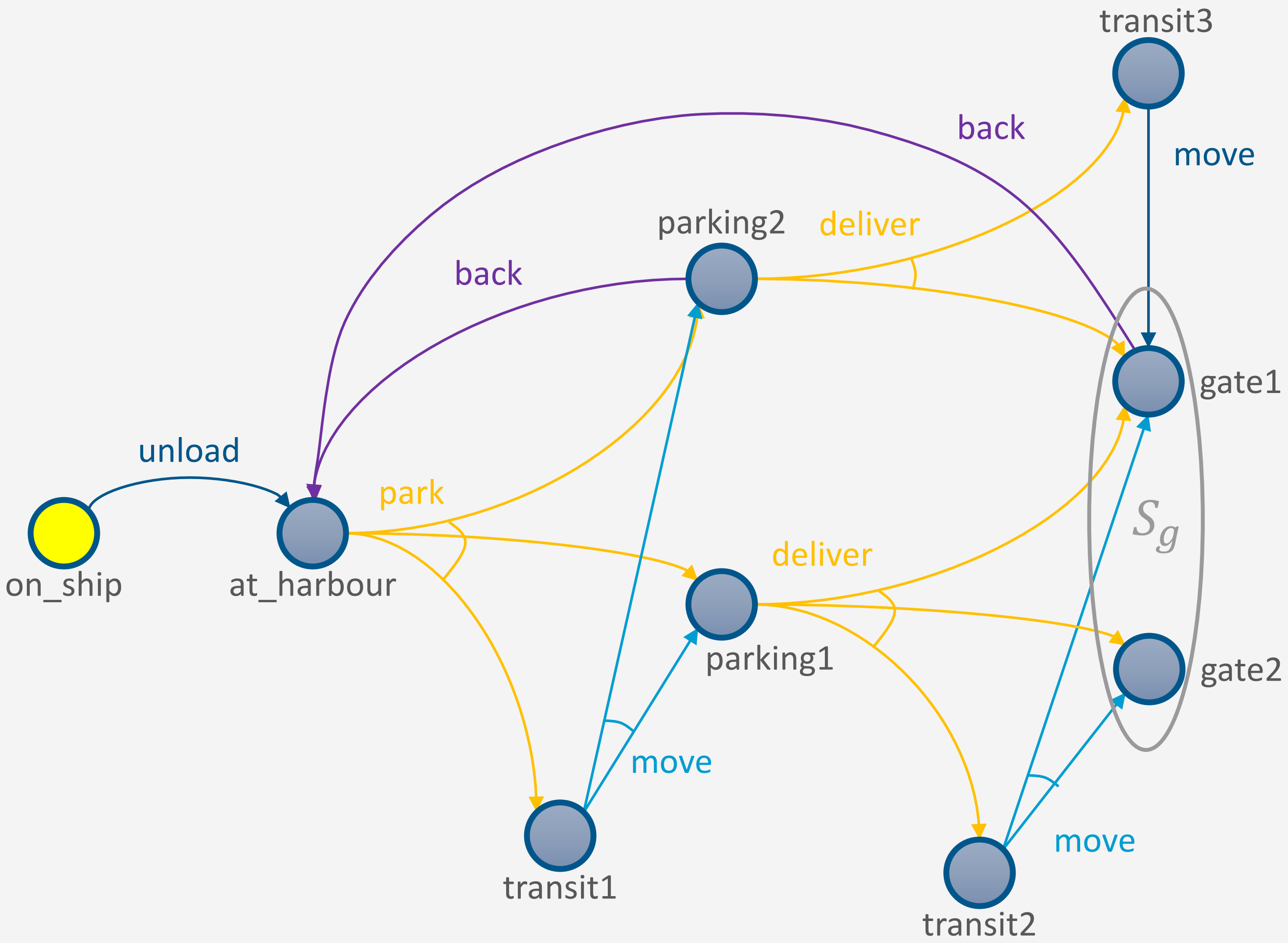
Input
- Planning problem (Σ, s_0, S_g)

Example

```
Find-Acyclic-Solution(Σ,s0,Sg)
    π ← ∅
    Frontier ← {s0}
    for every s ∈ Frontier \ Sg do
        ...
```

$Frontier \setminus S_g$ = {on_ship}

$\pi = \{\}$



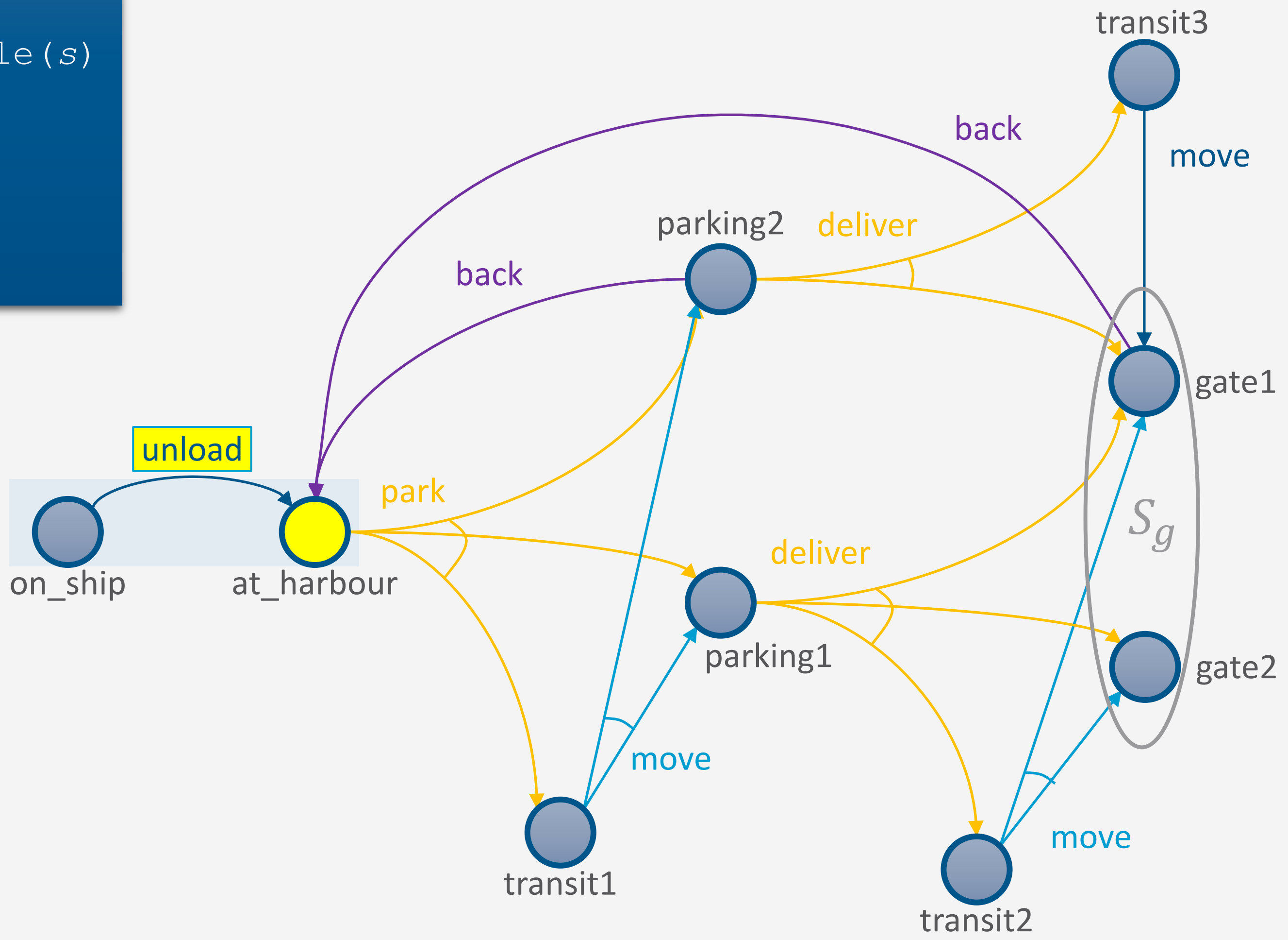
```
Find-Acyclic-Solution(Σ,s_0,S_g)
    ...
    for every s ∈ Frontier \ S_g do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-loops(π,s,Frontier) then
            return failure
    return π
```

s = on_ship

$Frontier \setminus S_g$ = {at_harbor}

π = {(on_ship, unload)}

Nondeterministic



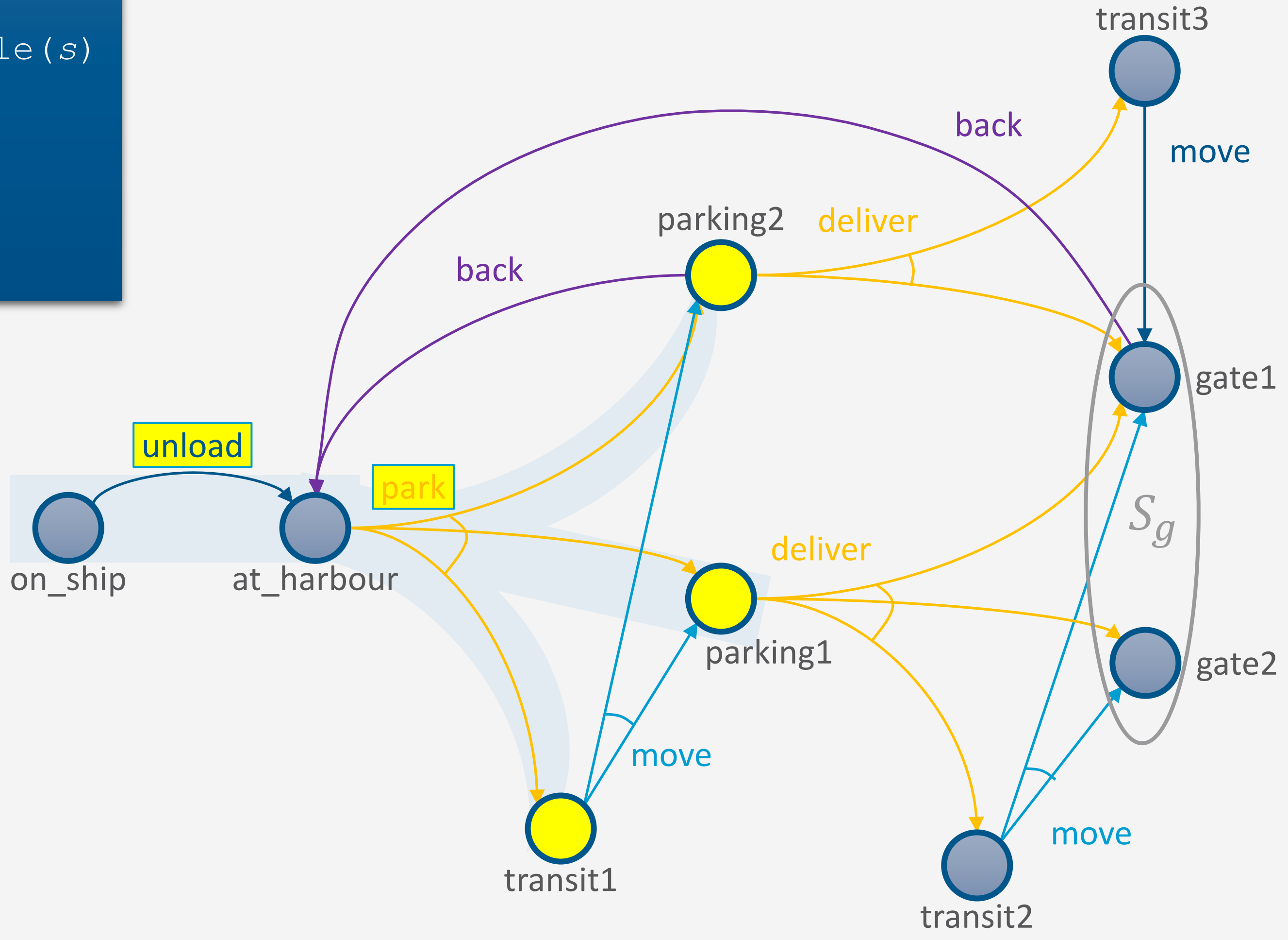
```
Find-Acyclic-Solution(Σ,s0,Sg)
    ...
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-loops(π,s,Frontier) then
            return failure
    return π
```

Nondeterministic

s = at_harbor

Frontier \ S_g = {parking1, parking2, transit1}

π = {(on_ship, unload), (at_harbor, park)}



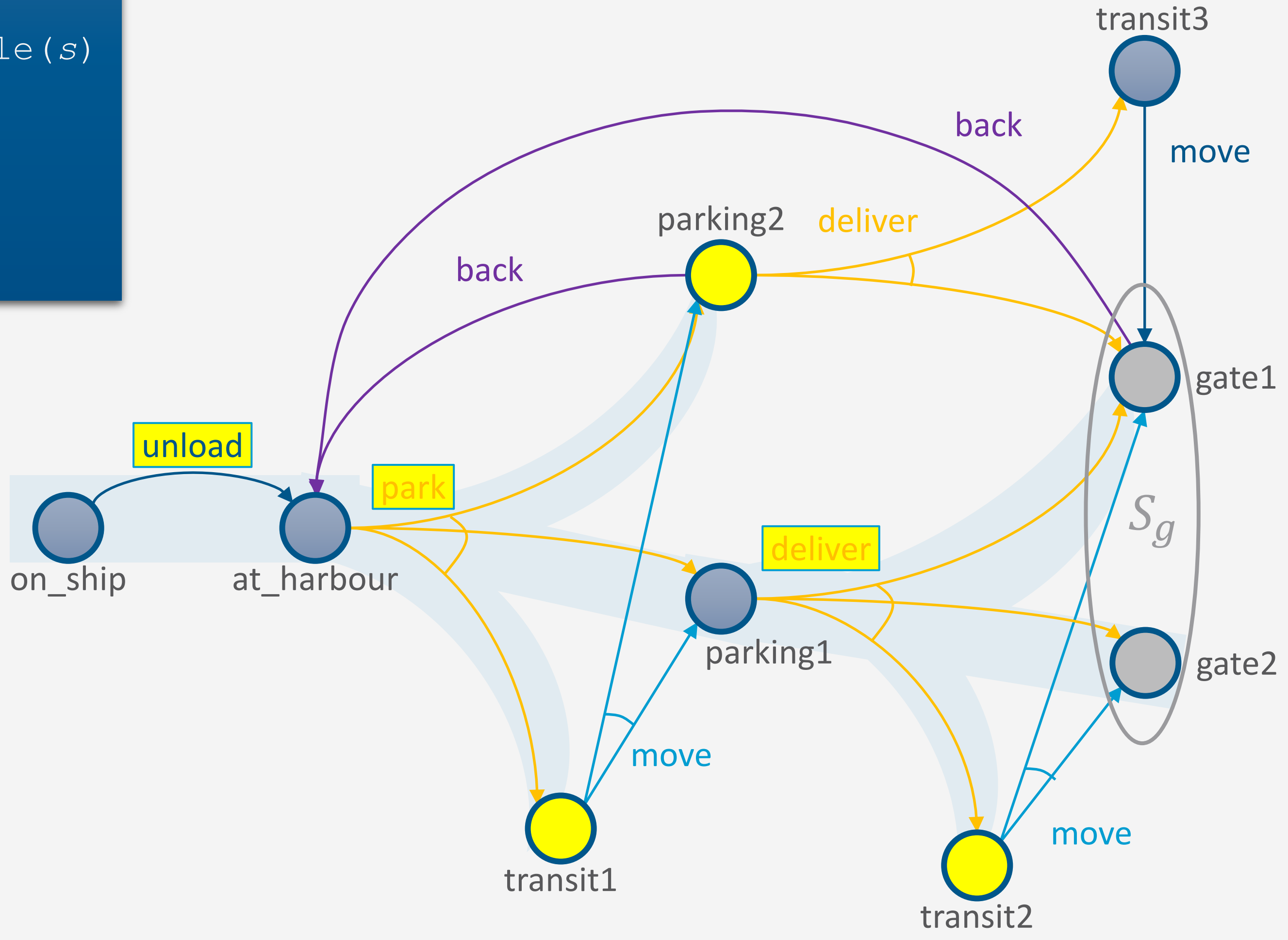
```
Find-Acyclic-Solution(Σ,s_0,S_g)
    ...
    for every s ∈ Frontier \ S_g do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-loops(π,s,Frontier) then
            return failure
    return π
```

Nondeterministic

s = parking1

Frontier \ S_g = {parking2, transit1, transit2}

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver)}



```
Find-Acyclic-Solution(Σ,s0,Sg)
    ...
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-loops(π,s,Frontier) then
            return failure
    return π
```

s = parking2

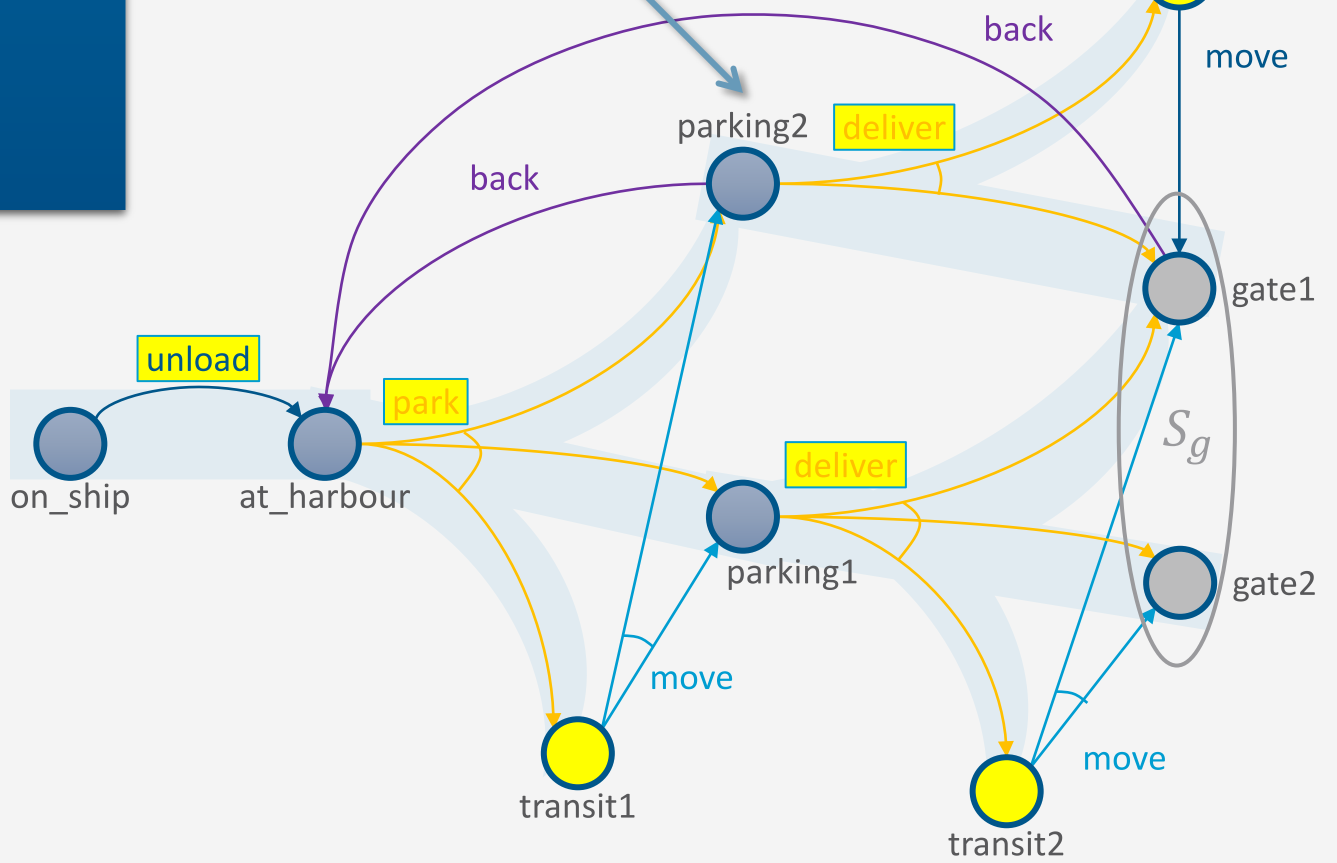
Frontier \ S_g = {transit1, transit2, transit3}

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver)}

Nondeterministic

nondeterministically choose *back* or *deliver*

- *back* ⇒ cycle, so return *failure*
- *deliver* ⇒ no cycle, so continue



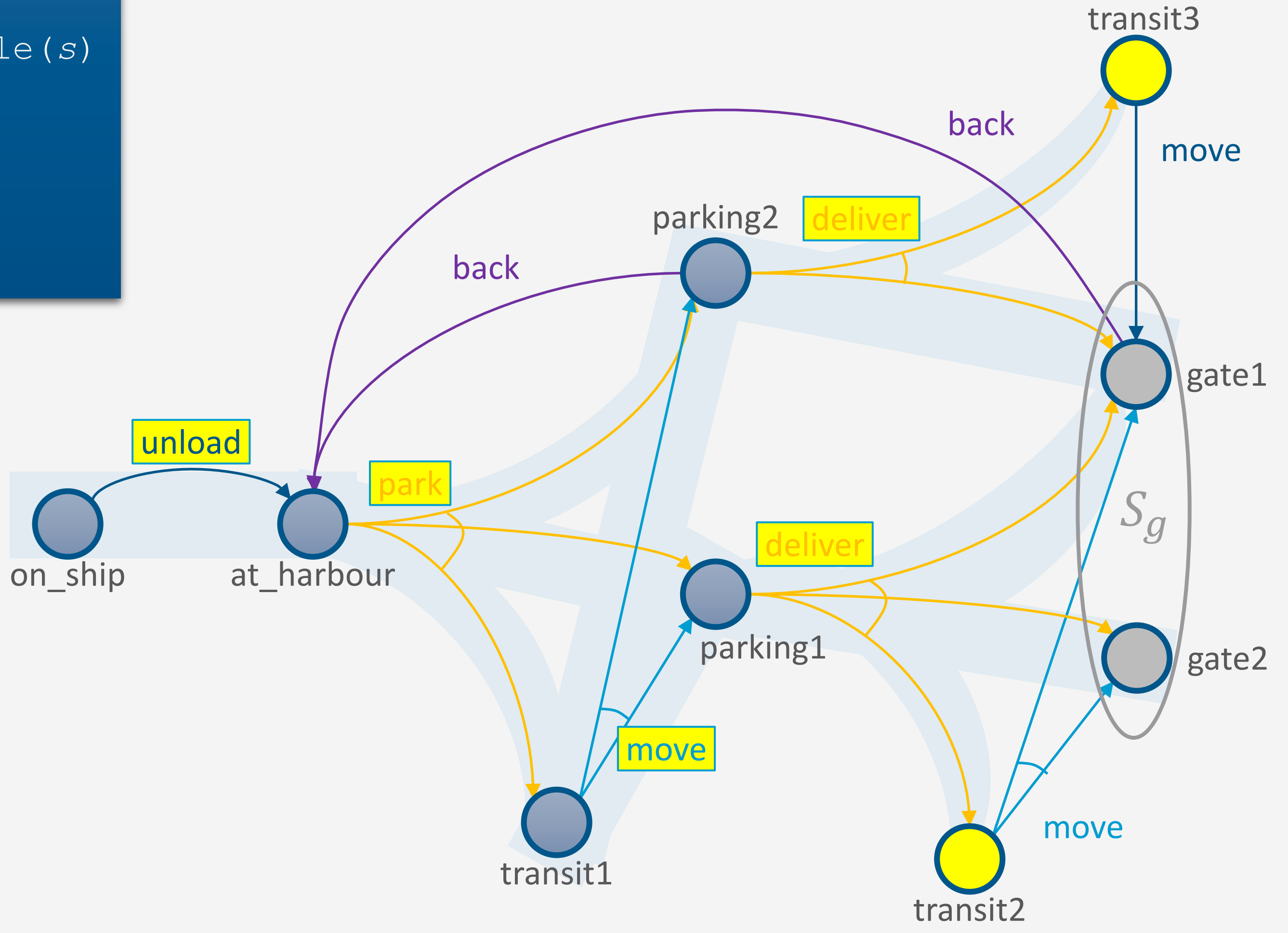
```
Find-Acyclic-Solution(Σ,s0,Sg)
    ...
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-loops(π,s,Frontier) then
            return failure
    return π
```

Nondeterministic

s = transit1

$Frontier \setminus S_g$ = {transit2, transit3}

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(transit1, move)}



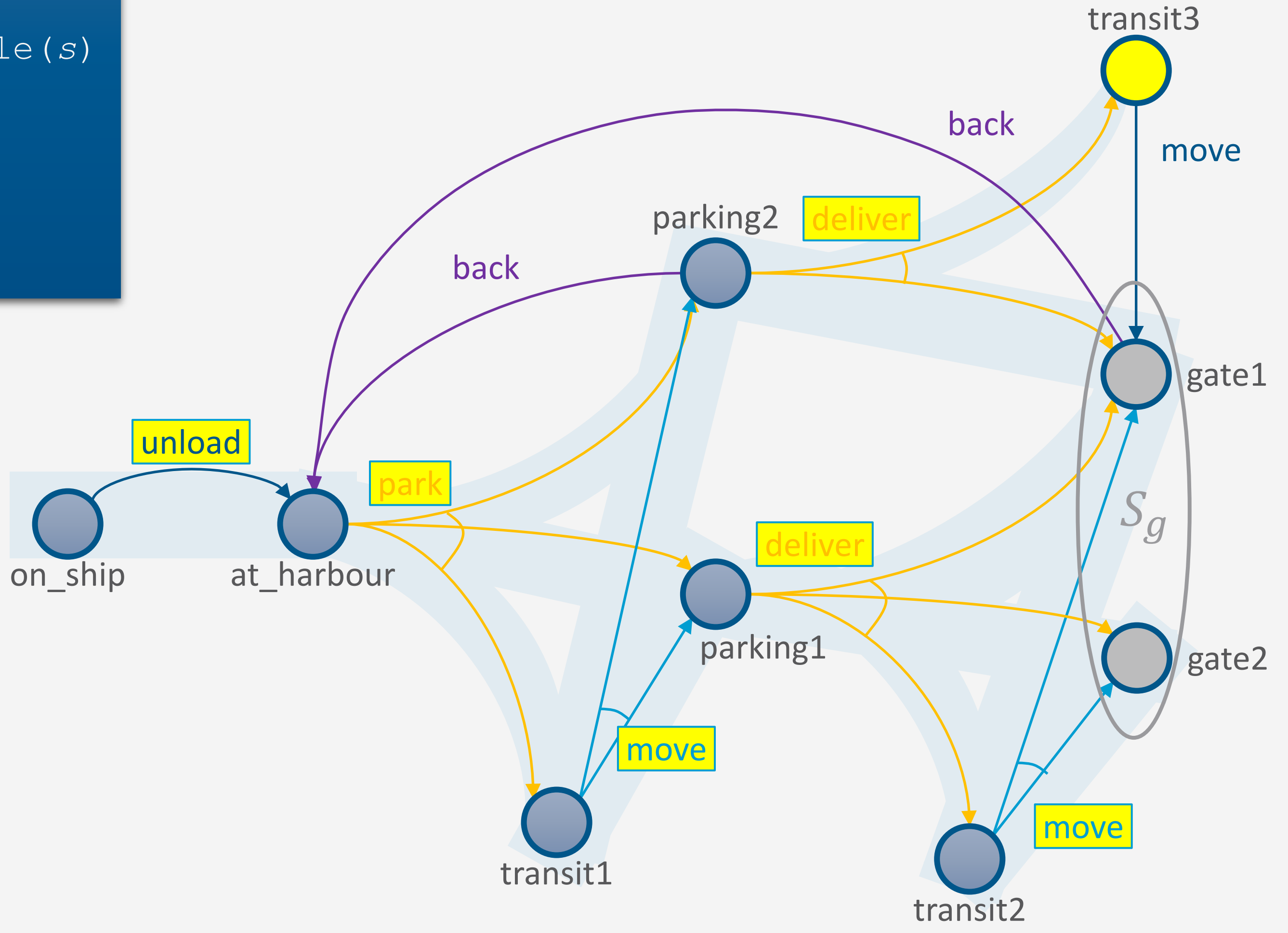
```
Find-Acyclic-Solution(Σ,s0,Sg)
    ...
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ Dom(π))
        if has-loops(π,s,Frontier) then
            return failure
    return π
```

Nondeterministic

s = transit2

$Frontier \setminus S_g$ = {transit3}

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(transit1, move),
(transit2, move)}



```
Find-Acyclic-Solution(Σ,s0,Sg)
   ...
   for every s ∈ Frontier \ Sg do
      Frontier ← Frontier \ {s}
      ...
      nondeterministically choose a ∈ Applicable(s)
      π ← π ∪ (s,a)
      Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
      if has-loops(π,s,Frontier) then
         return failure
   return π
```

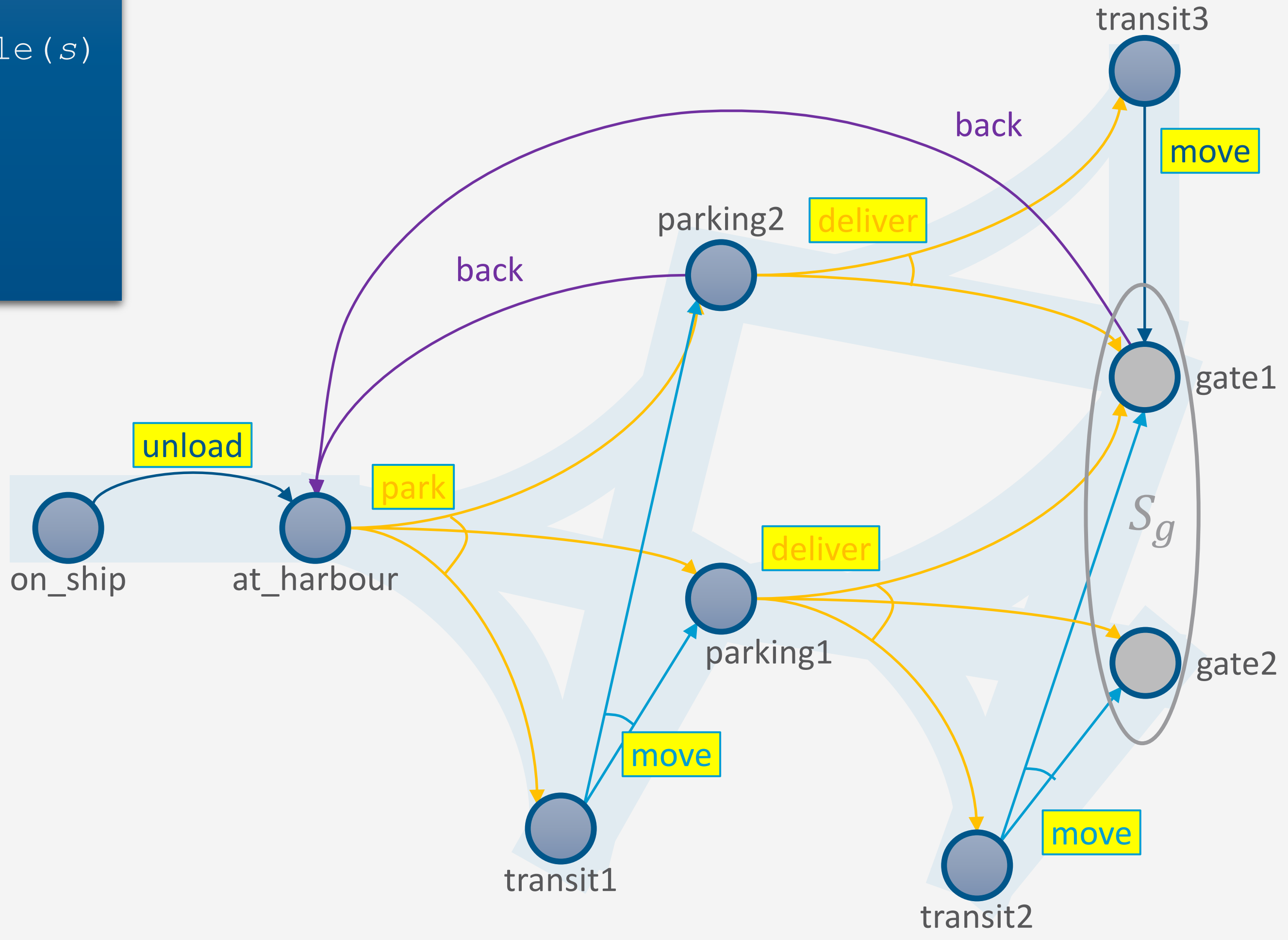
Nondeterministic

s = transit3

$Frontier \setminus S_g = \emptyset$

Found a solution, so return π

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(transit1, move),
(transit2, move),
(transit3, move)}



Finding Safe Solutions

- Same as `Find-Acyclic-Solution` except for cycle-checking
- `has-unsafe-loops` instead of `has-loops`
- Check if π contains any cycles that cannot be escaped:
  - For each $s' \in \left(\gamma(s,a) \cap \mathrm{dom}(\pi)\right)$
    - Is $\hat{\gamma}(s',\pi) \cap Frontier = \emptyset$?
  - Formally, $\mathrm{has-unsafe-loops}(\pi, s, Frontier)$ iff
    $$\exists s' \in \left(\gamma(s,a) \cap \mathrm{dom}(\pi)\right) : \hat{\gamma}(s',\pi) \cap Frontier = \emptyset$$

```
Find-Safe-Solution(Σ,s0,Sg)
    π ← ∅
    Frontier ← {s0}
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        if Applicable(s) = ∅ then
            return failure
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-unsafe-loops(π,s,Frontier) then
            return failure
    return π
```

Different cycle-checking

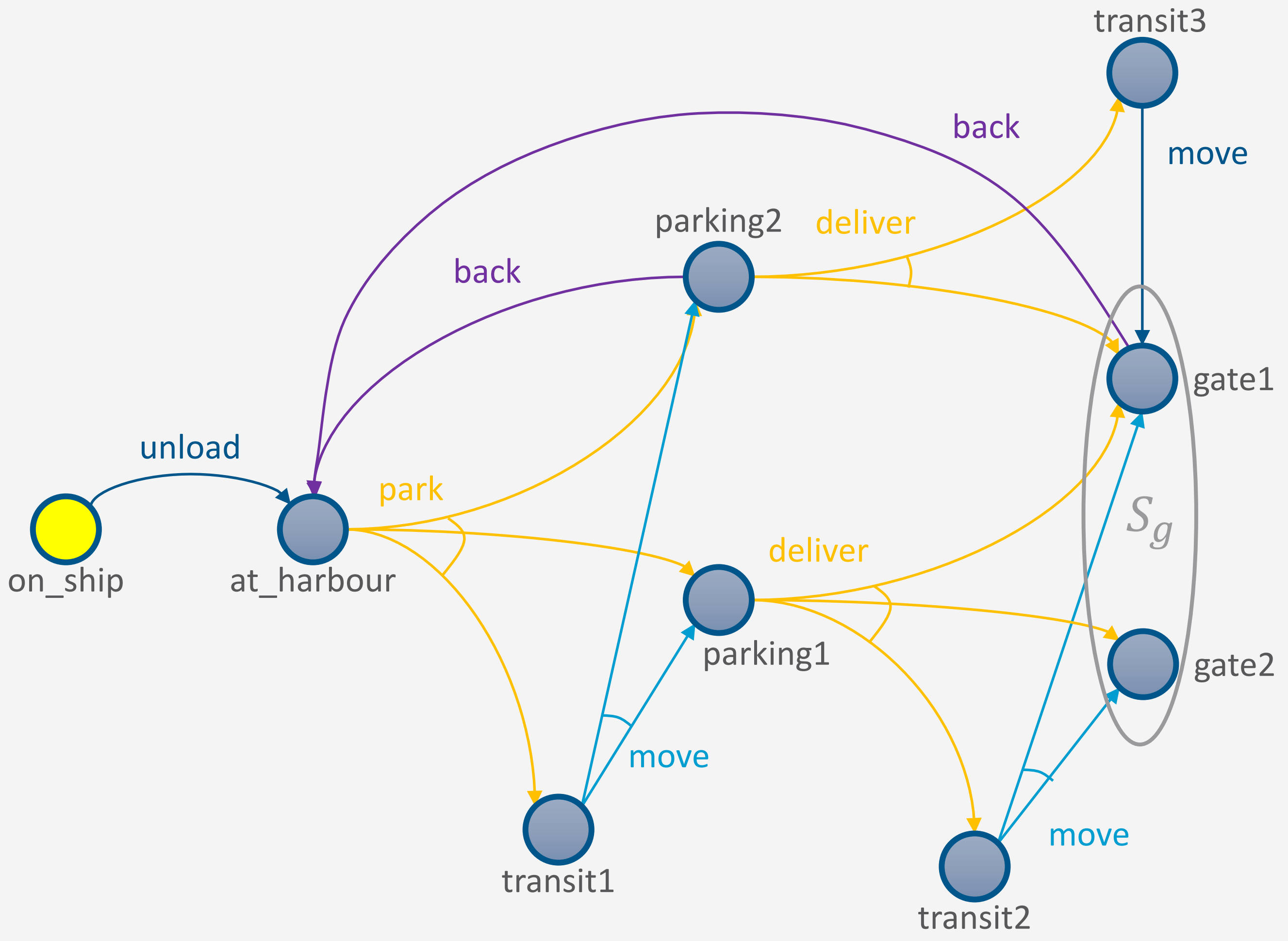
Input
- Planning problem (Σ, s_0, S_g)

Example

```
Find-Safe-Solution(Σ,s_0,S_g)
    π ← ∅
    Frontier ← {s_0}
    for every s ∈ Frontier \ S_g do
        ...
```

$Frontier \setminus S_g$ = {on_ship}

$\pi = \{\}$



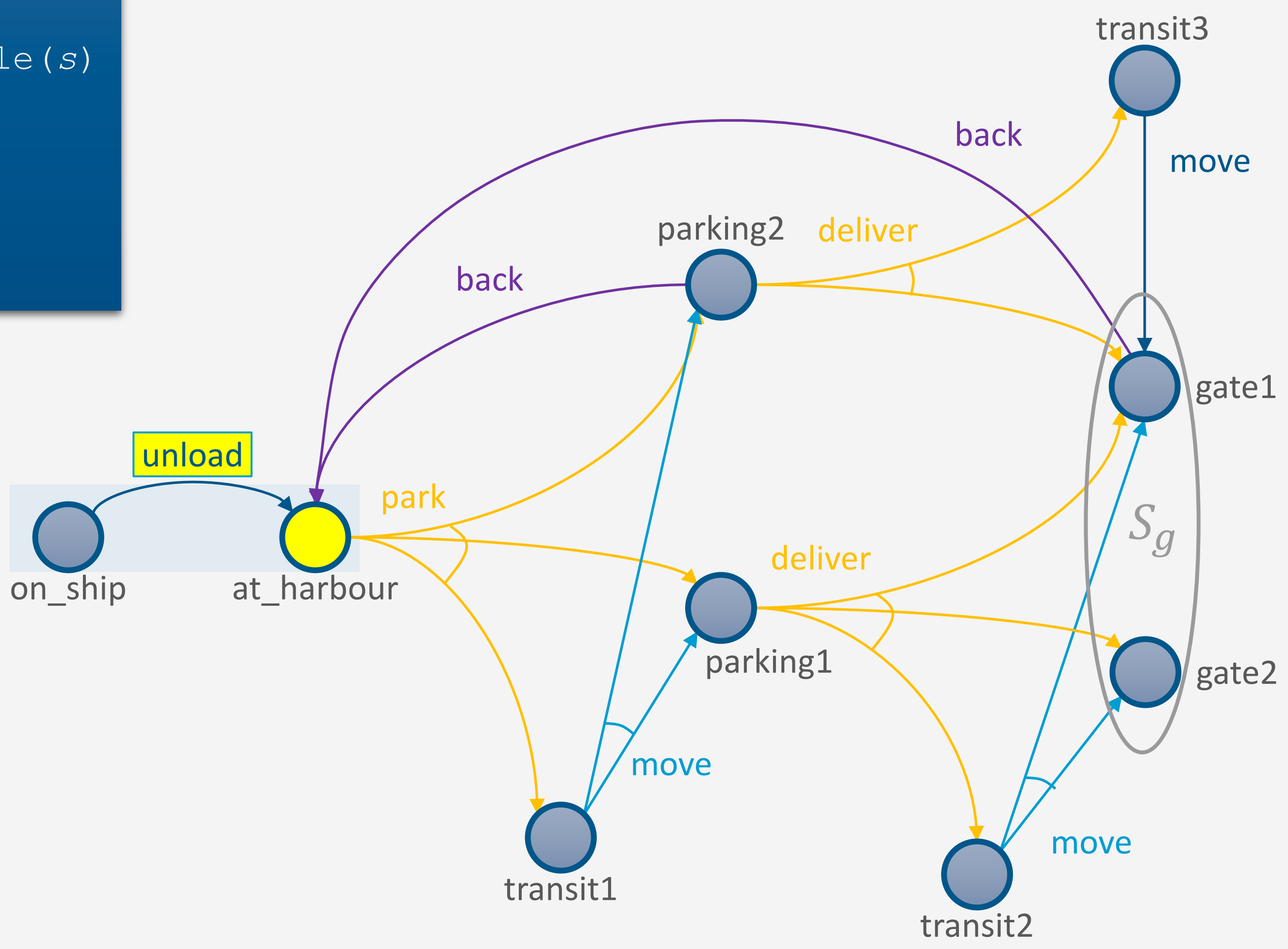
```
Find-Safe-Solution(Σ,s0,Sg)
    ...
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-unsafe-loops(π,s,Frontier) then
            return failure
    return π
```

Nondeterministic

s = on_ship

$Frontier \setminus S_g$ = {at_harbor}

π = {(on_ship, unload)}



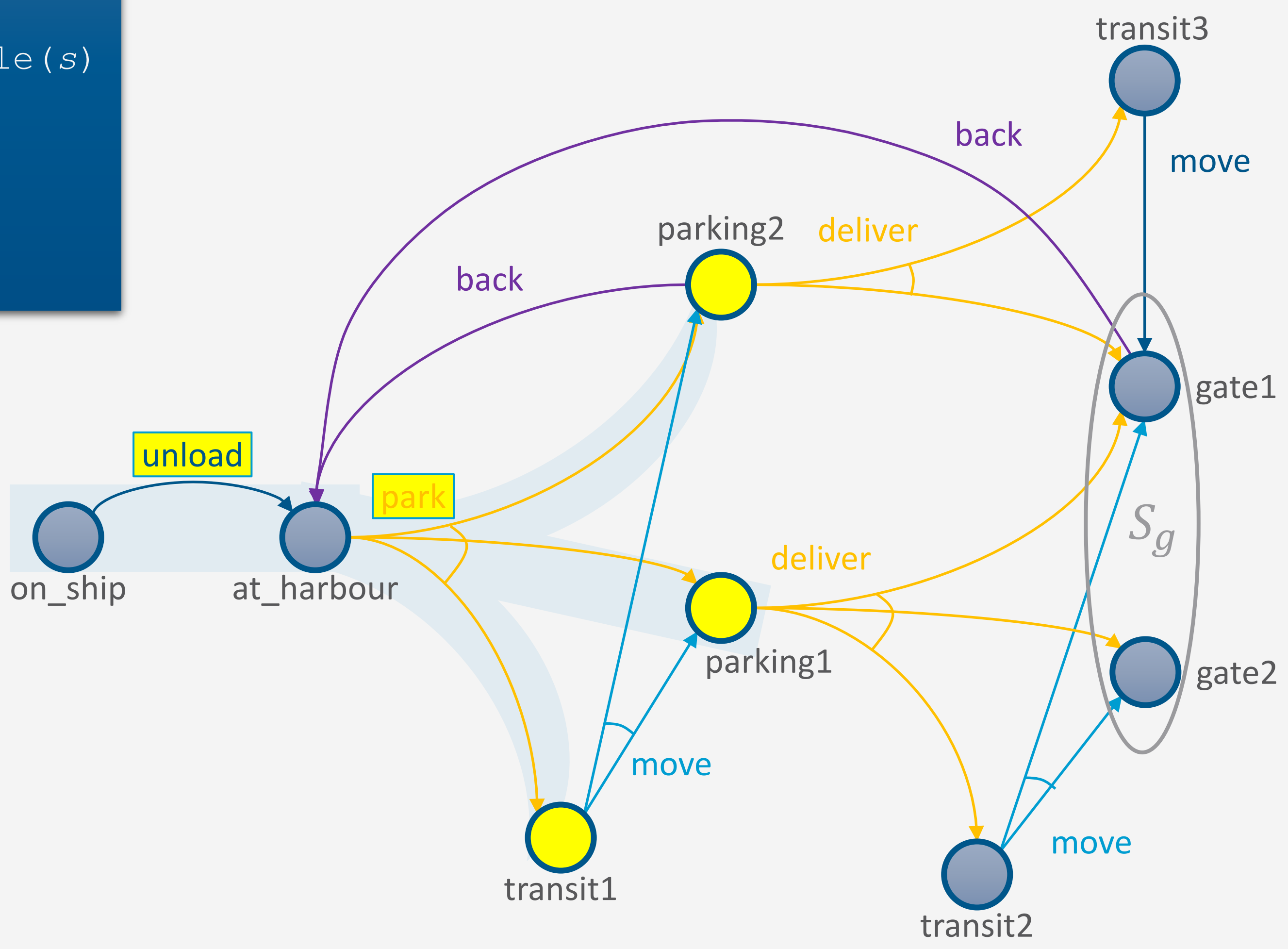
```
Find-Safe-Solution(Σ,s0,Sg)
    ...
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-unsafe-loops(π,s,Frontier) then
            return failure
    return π
```

s = at_harbor

$Frontier \setminus S_g$ = {parking1, parking2, transit1}

π = {(on_ship, unload),
(at_harbor, park)}

Nondeterministic



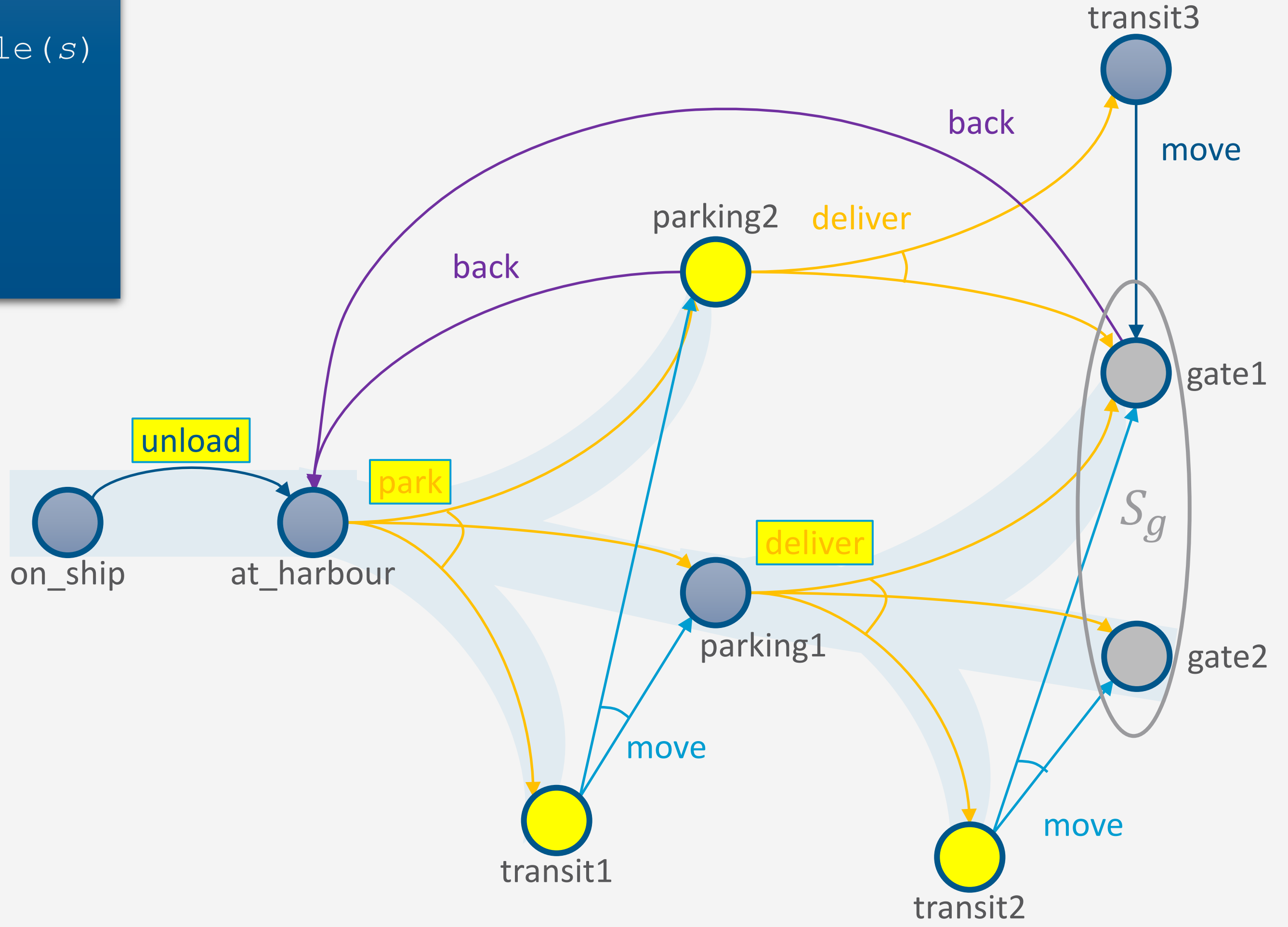
```
Find-Safe-Solution(Σ,s0,Sg)
    ...
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-unsafe-loops(π,s,Frontier) then
            return failure
    return π
```

s = parking1

$Frontier \setminus S_g$ = {parking2, transit1, transit2}

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver)}

Nondeterministic



```
Find-Safe-Solution(Σ,s0,Sg)
    ...
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-unsafe-loops(π,s,Frontier) then
            return failure
    return π
```

s = parking2

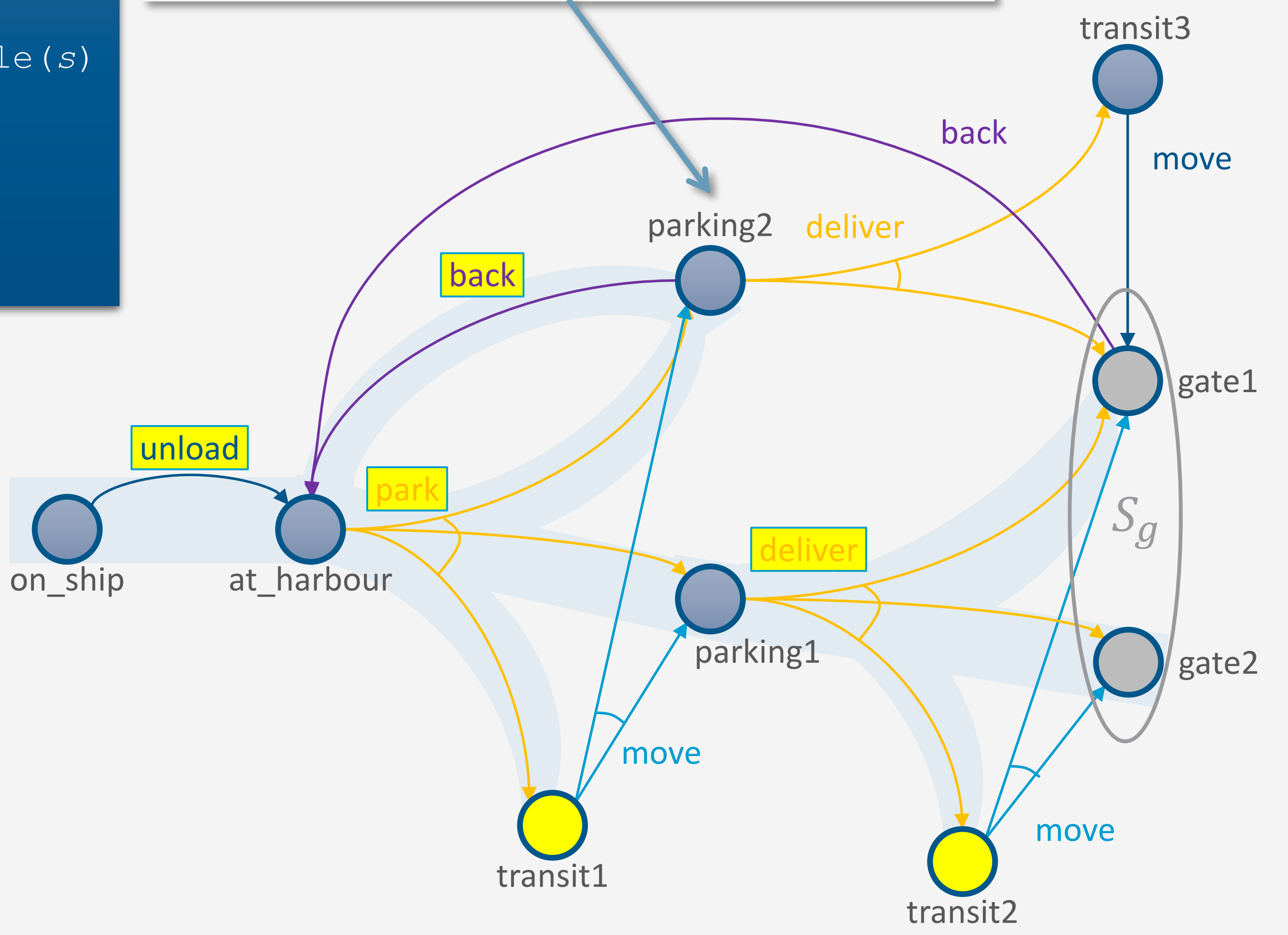
Frontier \ S_g = {transit1, transit2}

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, back)}

Nondeterministic

nondeterministically choose *back* or *deliver*

- *back* is okay: escapable cycle



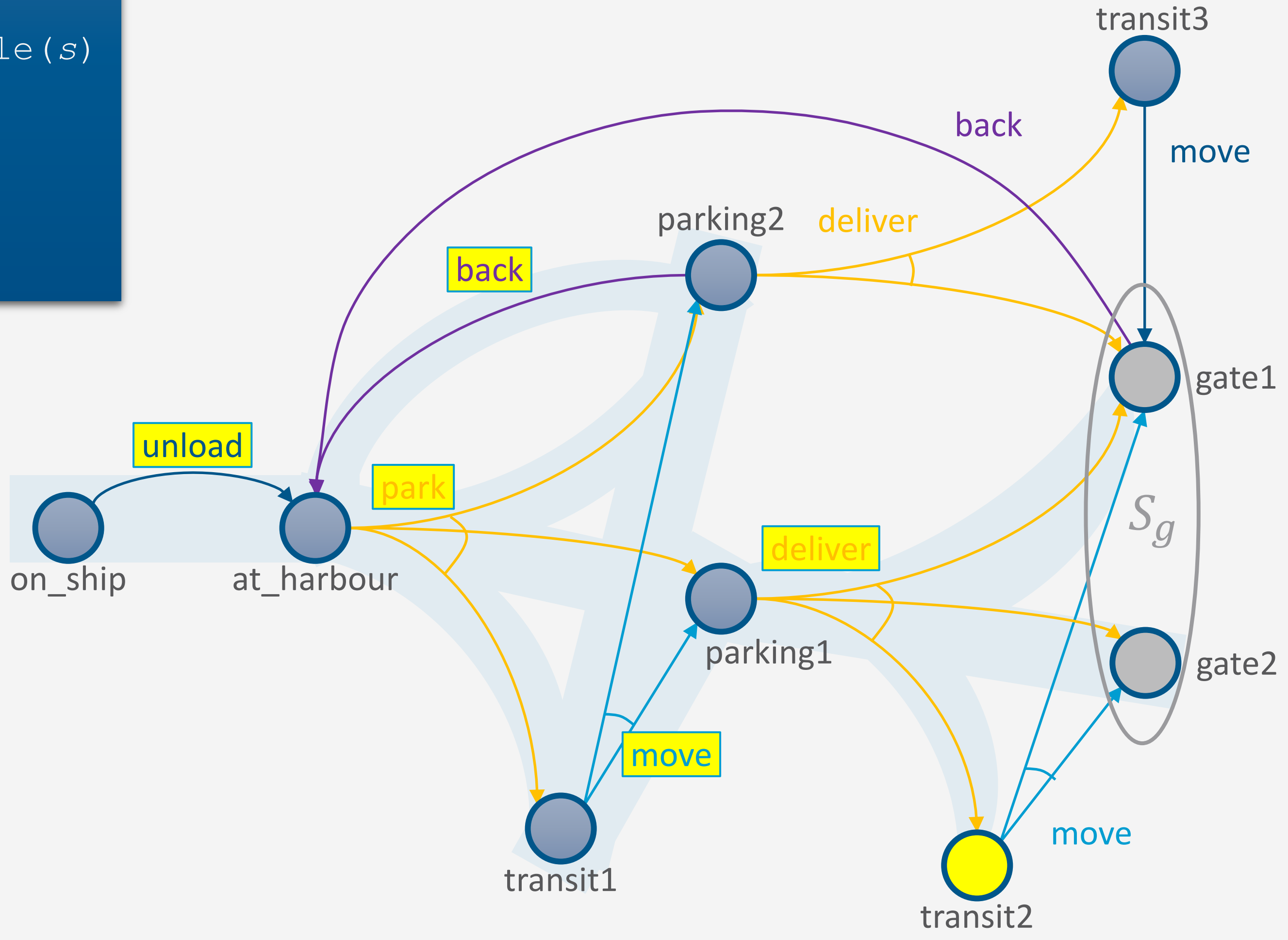
```
Find-Safe-Solution(Σ,s0,Sg)
    ...
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-unsafe-loops(π,s,Frontier) then
            return failure
    return π
```

Nondeterministic

s = transit1

$Frontier \setminus S_g$ = {transit2}

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, back),
(transit1, move)}



```
Find-Safe-Solution(Σ,s0,Sg)
    ...
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        ...
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ dom(π))
        if has-unsafe-loops(π,s,Frontier) then
            return failure
    return π
```

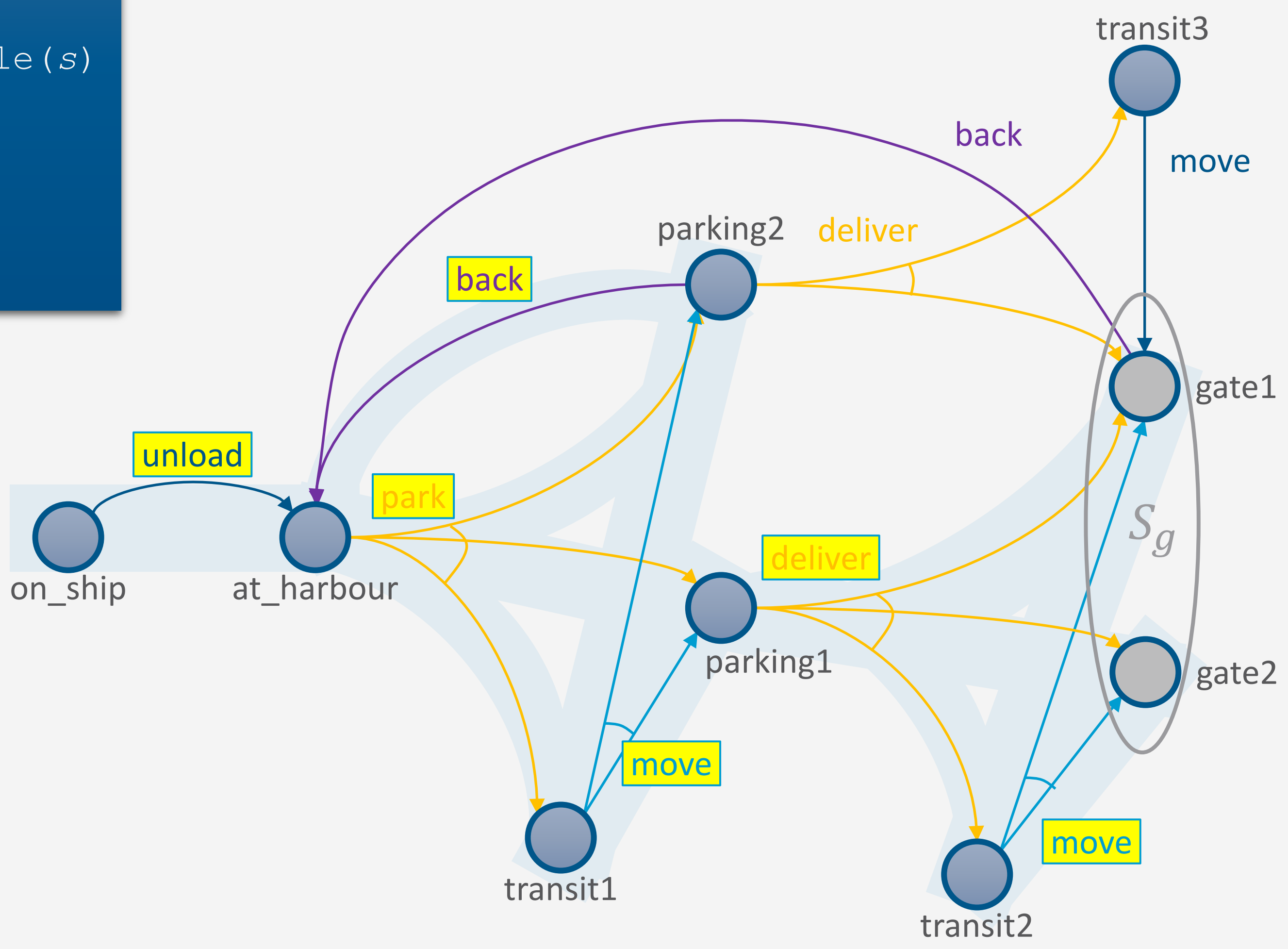
Nondeterministic

s = transit2

$Frontier \setminus S_g = \emptyset$

Found a solution, so return π

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, back),
(transit1, move),
(transit2, move)}



Intermediate Summary

- And/Or Graph Search
  - Analogue to forward search in deterministic models
  - Algorithms for each type of solution
    - Unsafe
    - Cyclic safe
    - Acyclic safe

Outline per the Book

5.2 Planning Problem

- Planning domains
- Plans as policies
- Planning problems and solutions

5.3 And/Or Graph Search

- Planning by forward search

5.5 Determinisation Techniques

- Guided planning for safe solutions
- Planning for safe solutions by determinisation

5.6 Online Approaches

- Lookahead
- Lookahead by determinisation
- Lookahead with a bounded number of steps

Guided-Find-Safe-Solution

- Motivation:
  - Much easier to find solutions if they do not have to be safe
  - Find-Safe-Solution needs plans for all possible outcomes of actions
  - Find-Solution only needs a plan for one of them
- Idea:
  - loop
    - Find a solution π
    - Look at each leaf node of π
      - If the leaf node is not a goal,
        find a solution and
        incorporate it into π

Guided-Find-Safe-Solution

- Input: Planning problem (Σ, s_0, S_g)

π is a solution. Return the part that is reachable from s_0.

Choose any leaf s that is not a goal. Find a solution π' for s.

For each (s, a) in π', add to π unless π already has an action at s.

s is unsolvable. For each (s', a) that can produce s, modify π and Σ so we will never use a at s'

```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    if s0 ∈ Sg then
        return ∅
    if Applicable(s0) = ∅ then
        return failure
    π ← ∅
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        arbitrarily select s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```

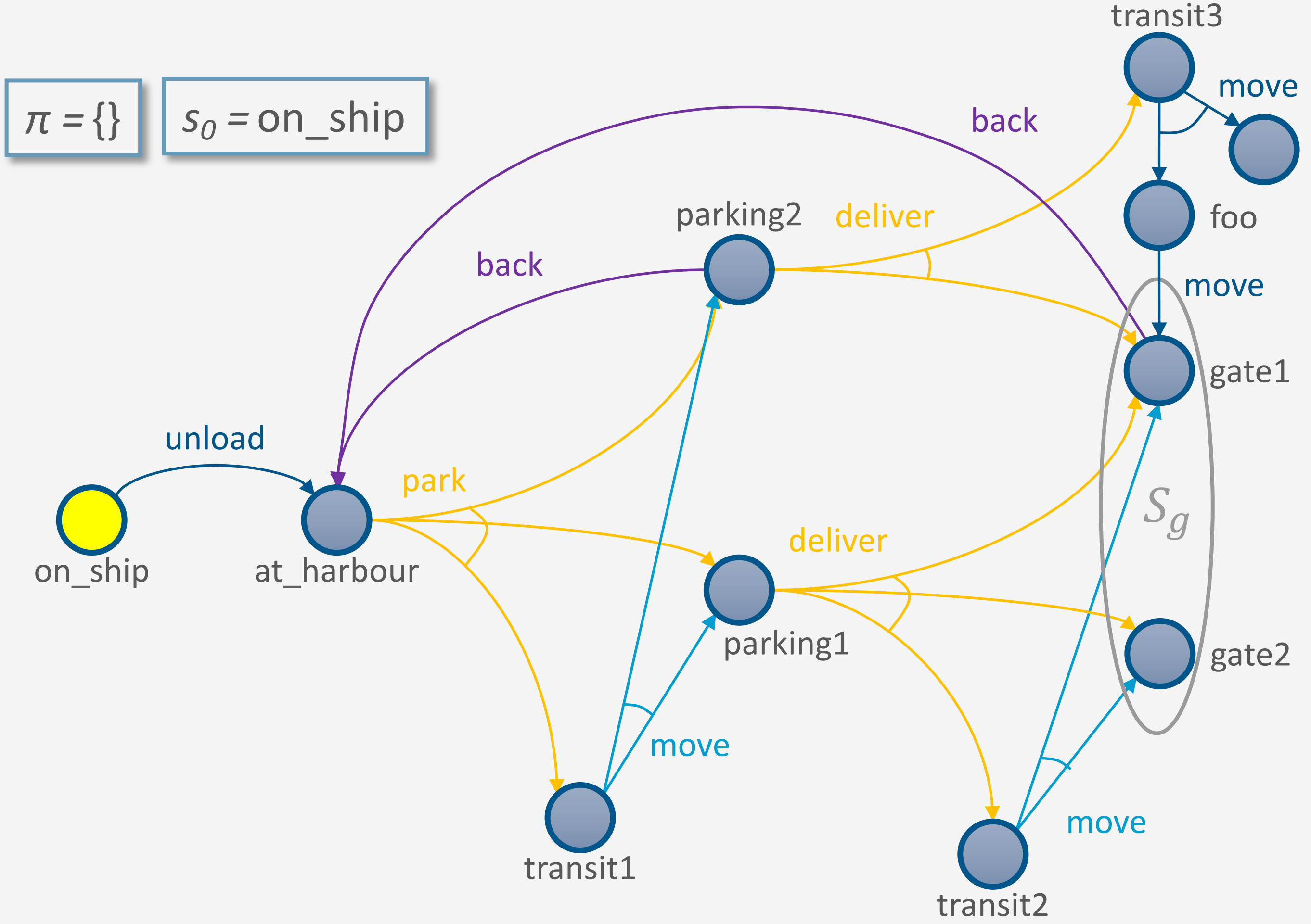
⇐ not in the book (applies to: else if s = s0 then / return failure)

Example

```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    ...
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
             π ← π ∪ {(s,a) ∈ π′ | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s′,a s.t. s ∈ γ(s′,a) do
                π ← π \ {(s′,a)}
                make a not applicable in s′
```

$\pi = \{\}$

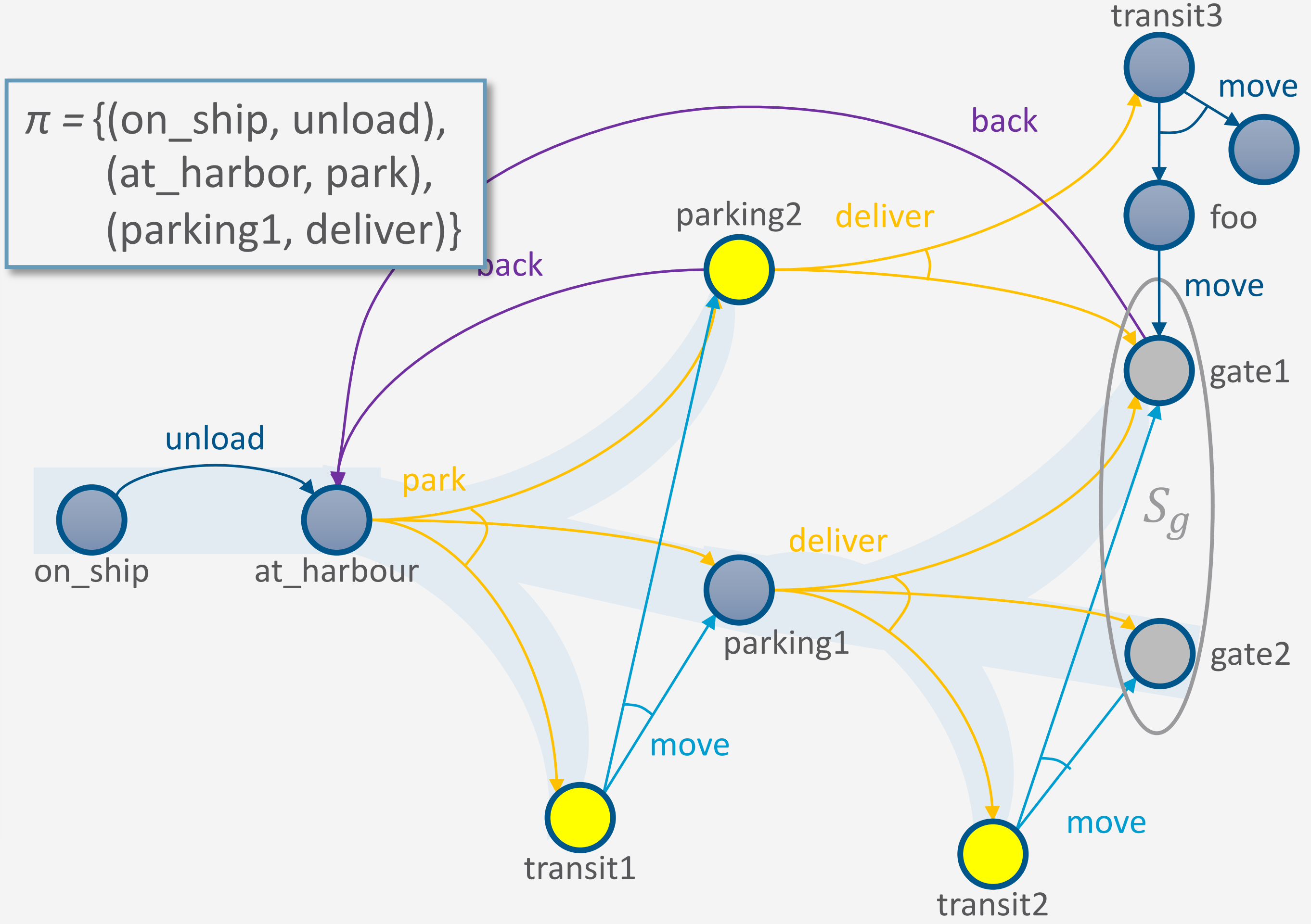
$s_0 = \text{on\_ship}$



Example

```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    ...
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```

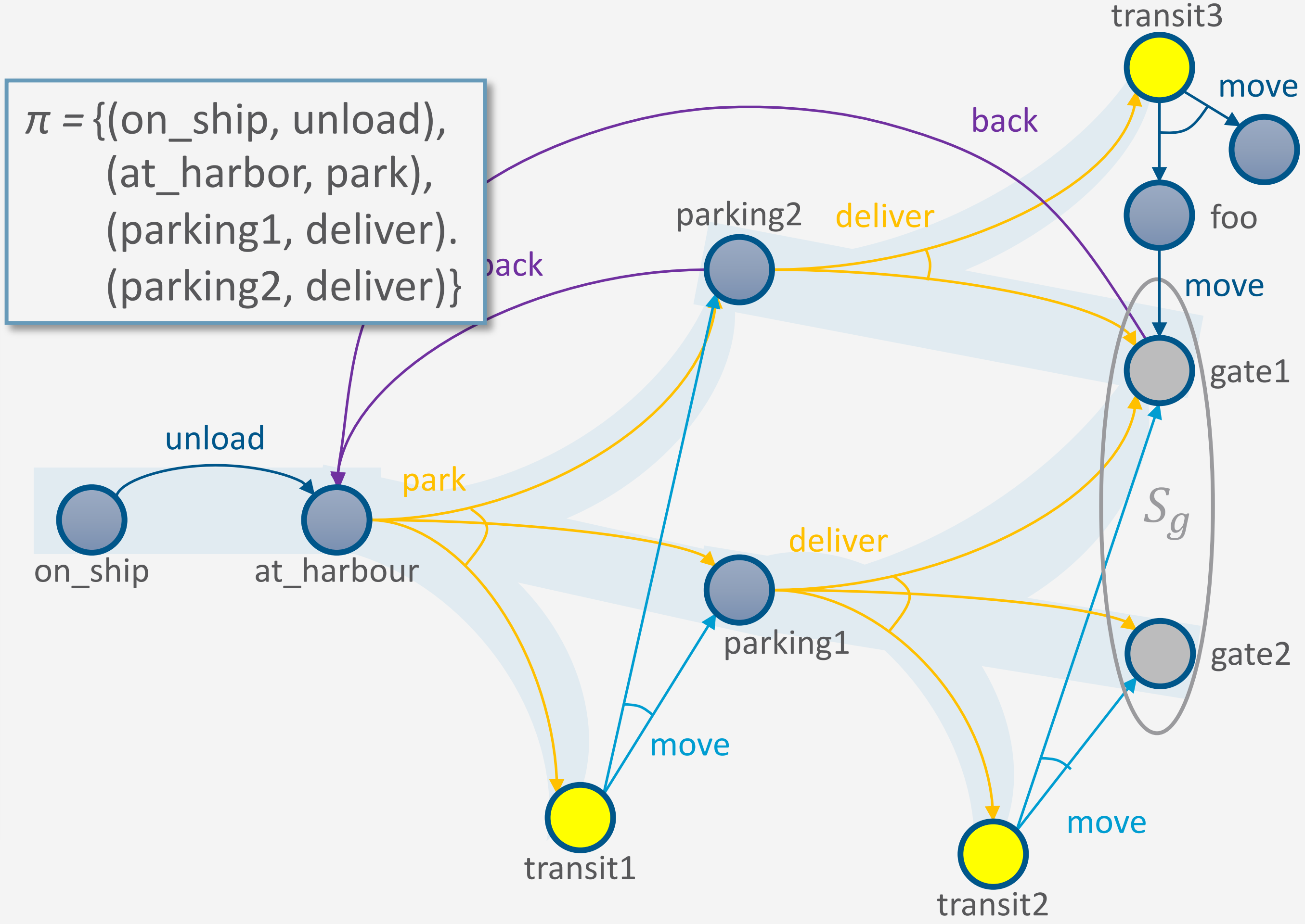
π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver)}



Example

```
Guided-Find-Safe-Solution(Σ,s₀,S_g)
    ...
    loop
        Q ← leaves(s₀,π) \ S_g
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s₀,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,S_g)
        if π' ≠ failure then
             π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s₀ then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```

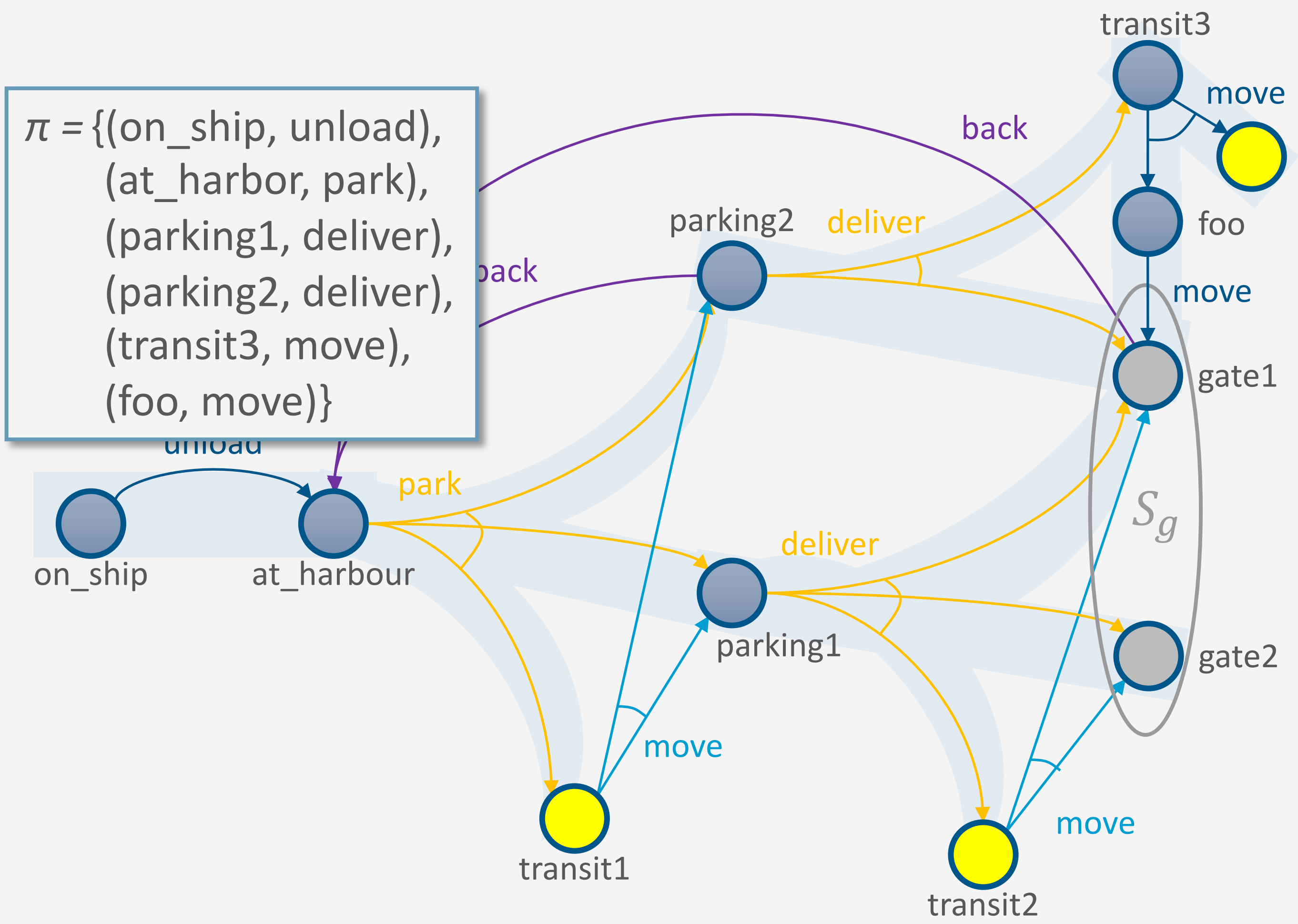
π = {(on_ship, unload), (at_harbor, park), (parking1, deliver). (parking2, deliver)}



Example

```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    ...
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
             π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```

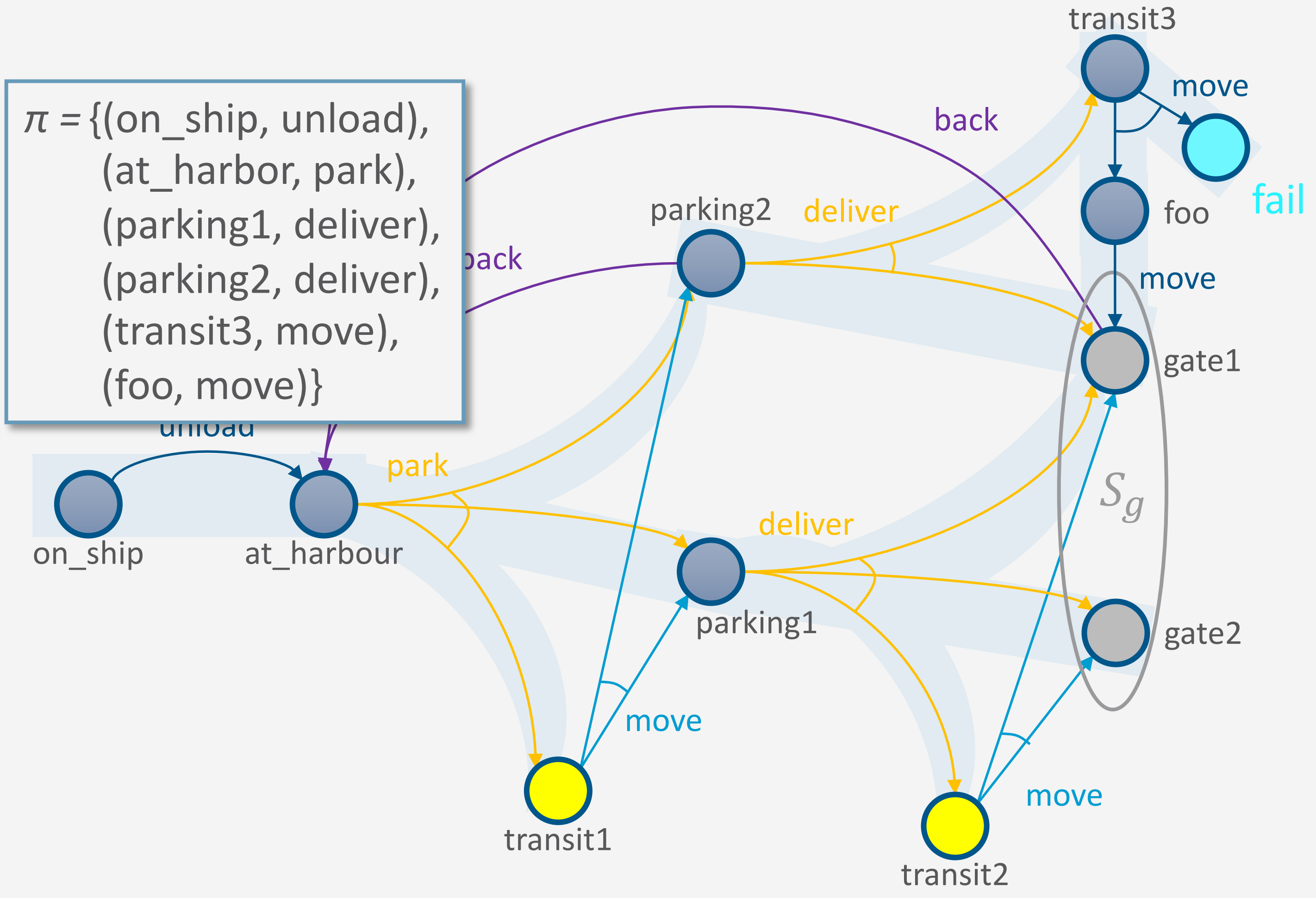
π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(transit3, move),
(foo, move)}



Example

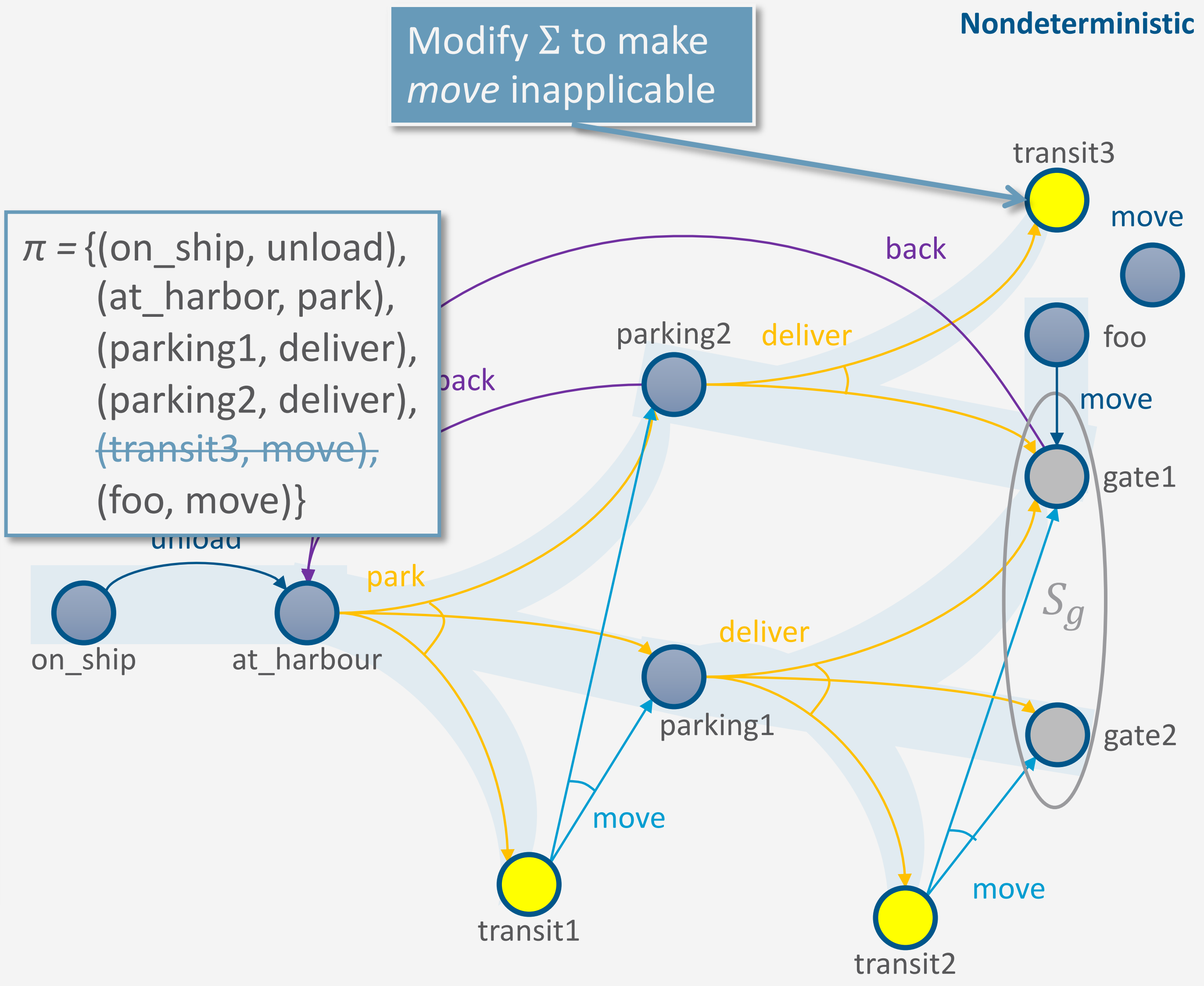
```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    ...
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
             π ← π ∪ {(s,a) ∈ π′ | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s′,a s.t. s ∈ γ(s′,a) do
                π ← π \ {(s′,a)}
                make a not applicable in s′
```

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(transit3, move),
(foo, move)}



Example

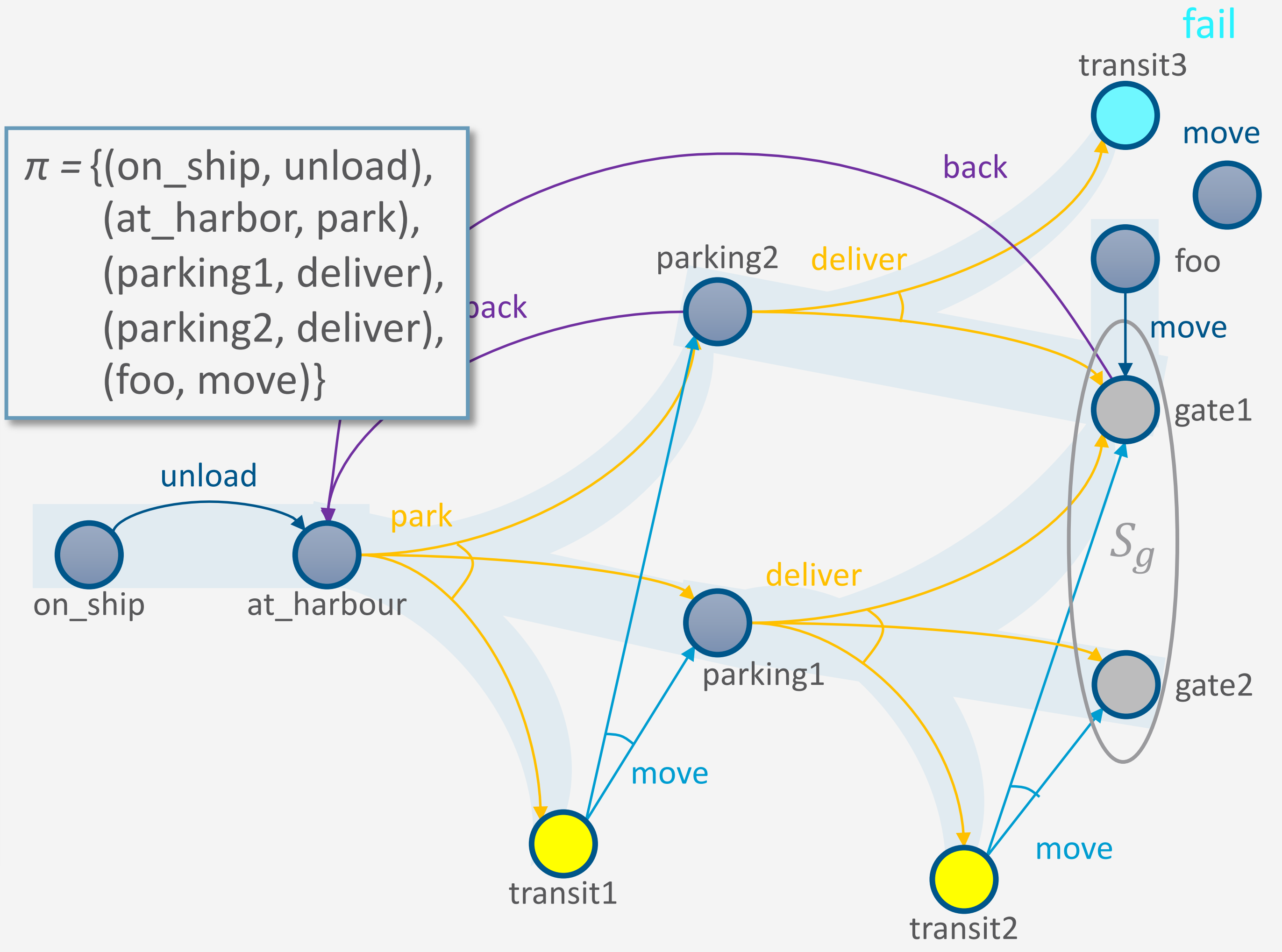
```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    ...
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
             π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```



Example

```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    ...
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(foo, move)}

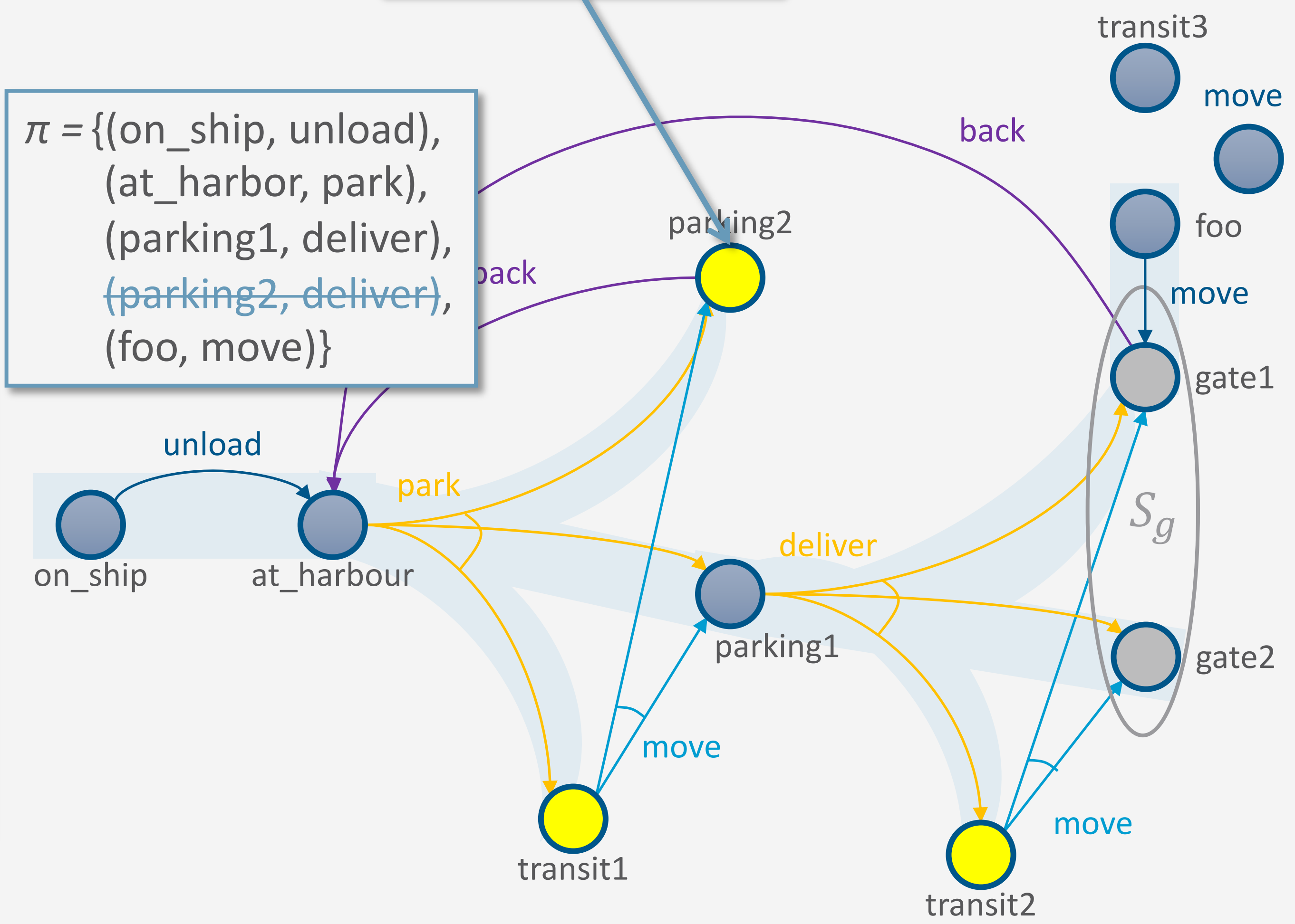


Example

```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    ...
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = Ø then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
             π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```

Modify Σ to make *deliver* inapplicable

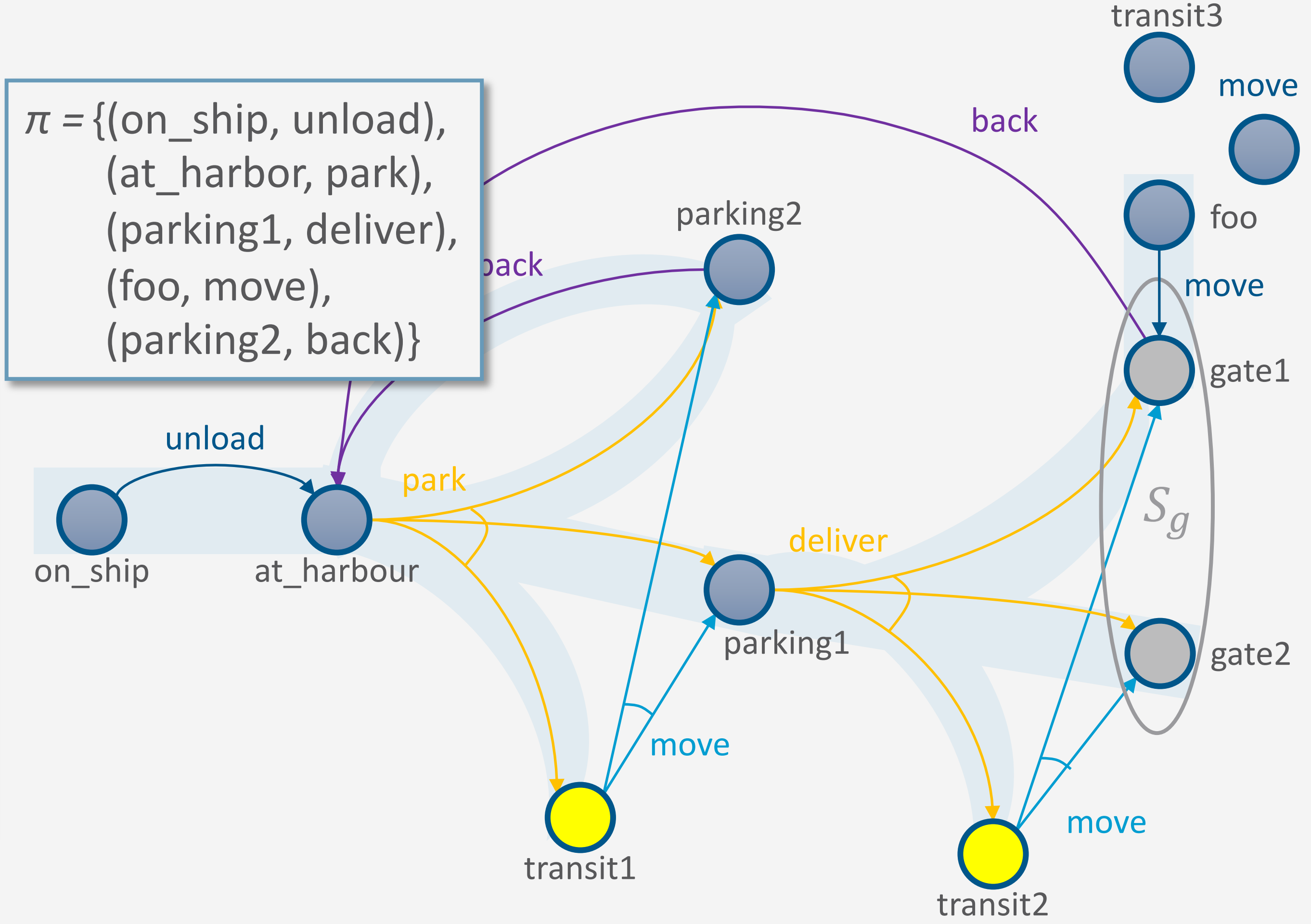
π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
~~(parking2, deliver)~~,
(foo, move)}



Example

```
Guided-Find-Safe-Solution(Σ,s₀,S_g)
    ...
    loop
        Q ← leaves(s₀,π) \ S_g
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s₀,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,S_g)
        if π' ≠ failure then
             π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s₀ then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```

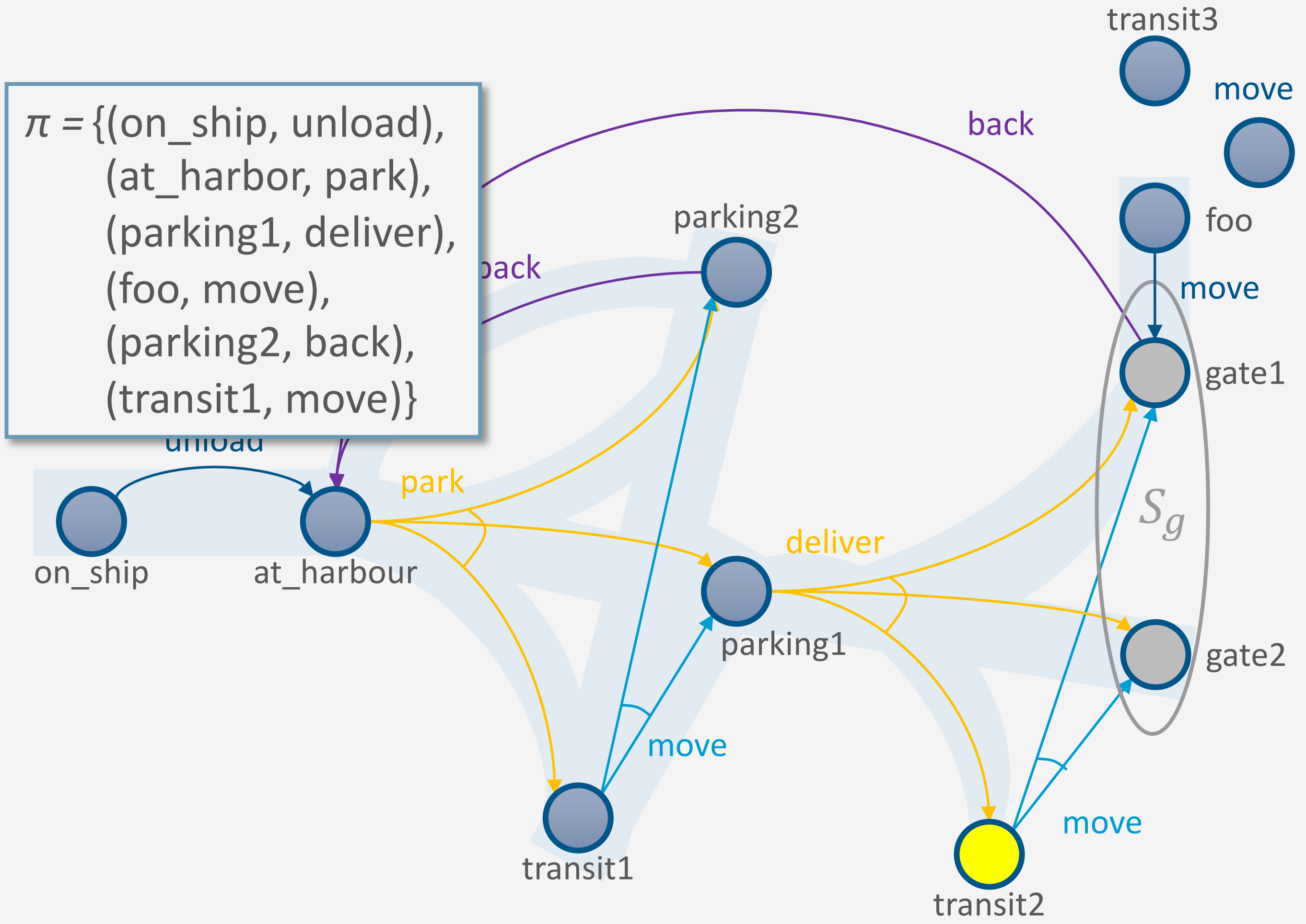
π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(foo, move),
(parking2, back)}



Example

```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    ...
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
             π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```

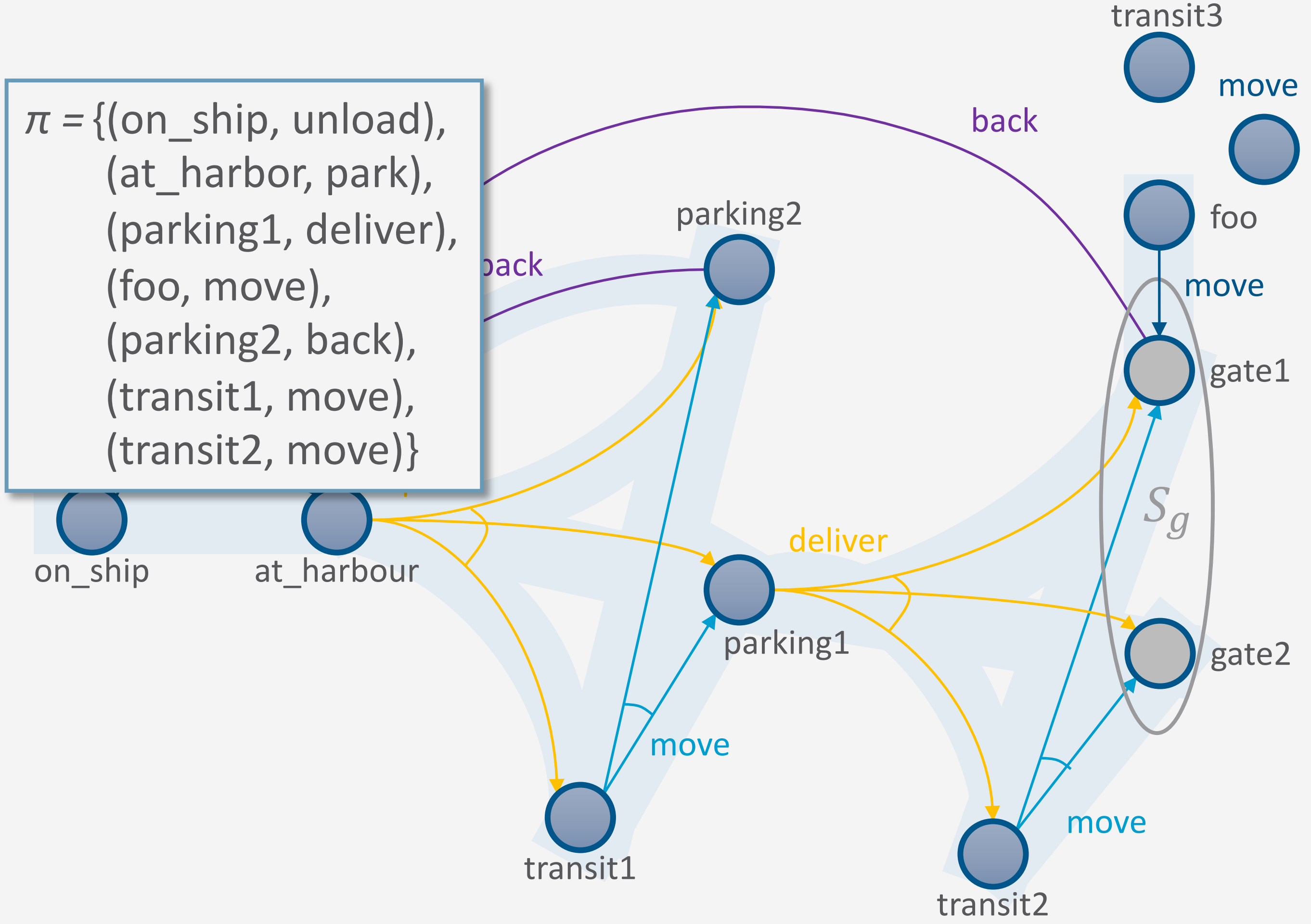
π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(foo, move),
(parking2, back),
(transit1, move)}



Example

```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    ...
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
             π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(foo, move),
(parking2, back),
(transit1, move),
(transit2, move)}

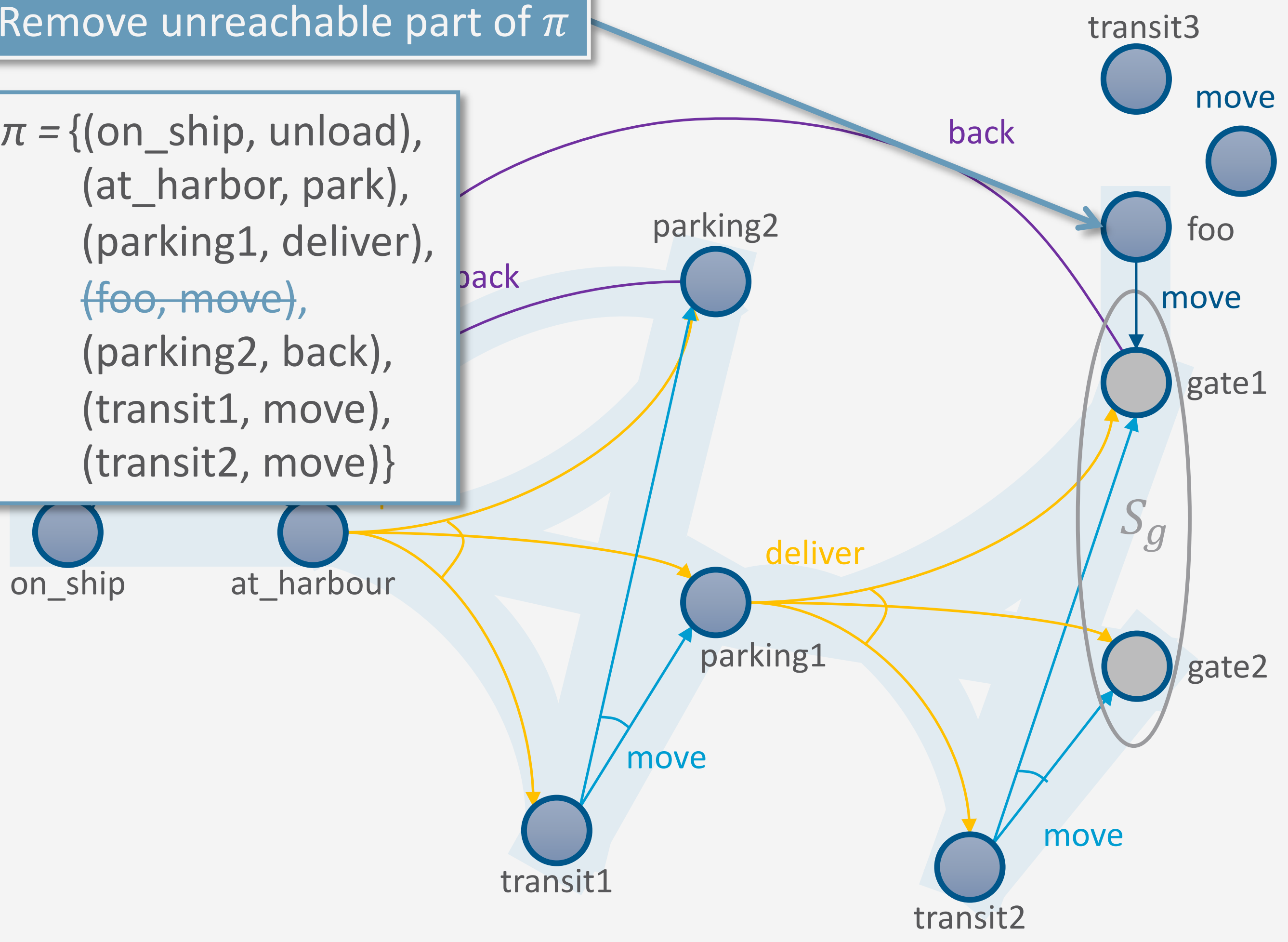


Example

```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    ...
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
             π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```

Remove unreachable part of π

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
~~(foo, move),~~
(parking2, back),
(transit1, move),
(transit2, move)}



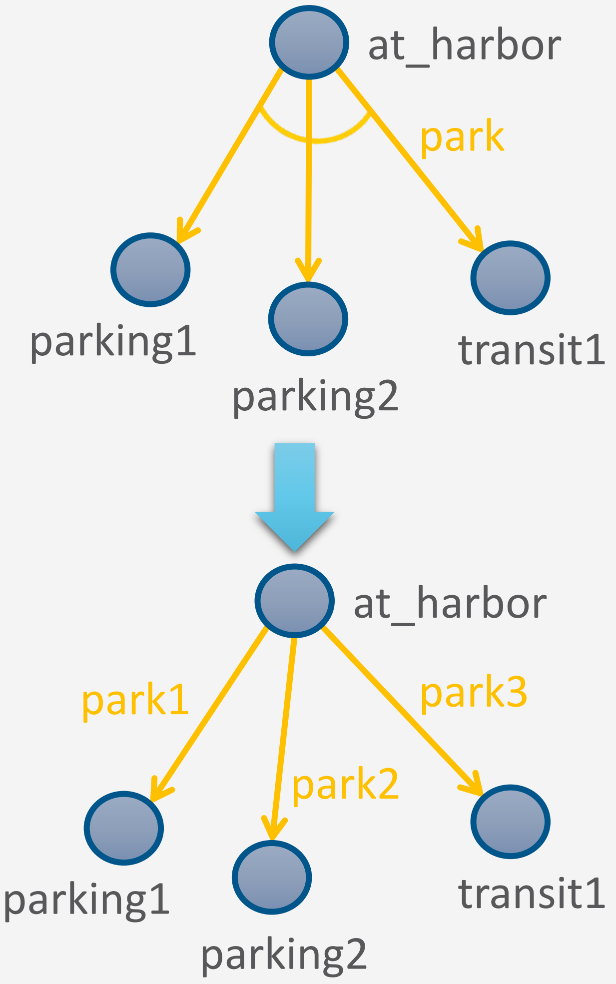
Determinisation

- How to implement it?
  - Need implementation of `Find-Solution`
  - Need it to be very efficient
    - Called many times
- Idea: instead, use a classical planner
  - Any algorithm from Ch. 2
  - Efficient algorithms, search heuristics
- For that, determinise actions

```
Guided-Find-Safe-Solution(Σ,s0,Sg)
    if s0 ∈ Sg then
        return ∅
    if Applicable(s0) = ∅ then
        return failure
    π ← ∅
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        π' ← Find-Solution(Σ,s,Sg)
        if π' ≠ failure then
            π ← π ∪ {(s,a) ∈ π' | s ∉ Dom(π)}
        else if s = s0 then
            return failure
        else
            for every s',a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make a not applicable in s'
```

Determinisation

- Convert nondeterministic actions into something a classical planner can use
- Determinise
  - Suppose a_i has K possible outcomes
    - K deterministic actions $a_i^k, k \in \{1, \dots, K\}$, one for each outcome
  - Given nondeterministic domain $\Sigma = (S, A, \gamma)$
    - Determinised domain $\Sigma_d = (S, A_d, \gamma_d)$ with
      - $A_d = \bigcup_{a_i \in A, a_i \text{ deterministic}} \{a_i\} \cup \bigcup_{a_i \in A, a_i \text{ nondeterministic}} \bigcup_{k=1}^{K} \{a_i^k\}$
      - γ_d defined as γ with determinised inputs s, a_i^k yielding a state with effects according to k
- Classical planner returns a plan $p = \langle a_1, a_2, \dots, a_n \rangle$
  - If p is acyclic, can convert it to a policy



Determinisation

- Nondeterministic planning problem $P = (\Sigma, s_0, S_g)$
- Determinisation $P_d = (\Sigma_d, s_0, S_g)$
  - As on previous slide
- Classical planner returns a solution for P_d
  - A plan $p = \langle a_1, a_2, \dots, a_n \rangle$
- If p is acyclic, can convert it to an (unsafe) solution for P
  - $\{(s_0, \boldsymbol{a}_1), (s_1, \boldsymbol{a}_2), \dots, (s_{n-1}, \boldsymbol{a}_n)\}$

  where
  - each $\boldsymbol{a}_i$ is the nondeterministic action whose determinisation includes a_i
    - Function det2nondet returns exactly this
  - each $s_i \in \gamma_d(s_{i-1}, a_i)$

```
Plan2policy(p=⟨a1,...,an⟩,s)
    π ← ∅
    for i from 1 to n do
        π ← π ∪ {s,det2nondet(ai)}
        s ← γd(s,ai)
    return π
```

Determinisation

- Input: Planning problem (Σ, s_0, S_g)

Same as Guided-Find-Safe-Solution

Any classical planner that does not return cyclic plans.

Convert p' to a policy. Add each (s, a) to π unless π already has an action for s.

s is unsolvable. For each (s', a) that can produce s, modify π and Σ_d such that we will never use a at s'.

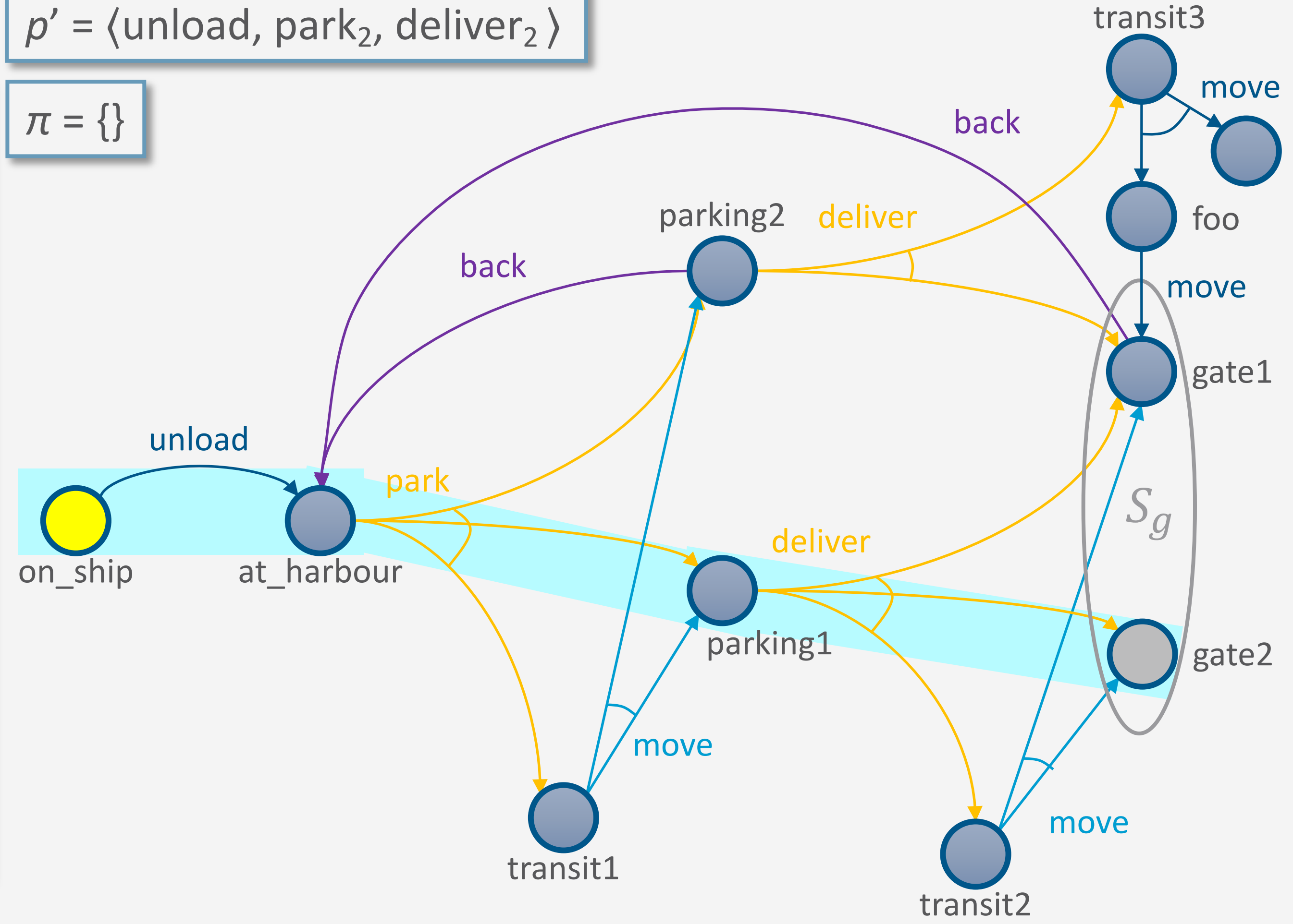
```
Find-Safe-Solution-by-Determinisation(Σ,s0,Sg)
    if s0 ∈ Sg then
        return ∅
    if Applicable(s0) = ∅ then
        return failure
    π ← ∅
    Σd ← mk-deterministic(Σ)
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σd,s,Sg)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make the actions in the
                        determinisation not
                        applicable in s'
```

Example

```
Find-Safe-Solution-by-Determinisation(Σ,s₀,S_g)
    ...
    Σ_d ← mk-deterministic(Σ)
    loop
        Q ← leaves(s₀,π) \ S_g
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s₀,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σ_d,s,S_g)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle \text{unload}, \text{park}_2, \text{deliver}_2 \rangle$

$\pi = \{\}$

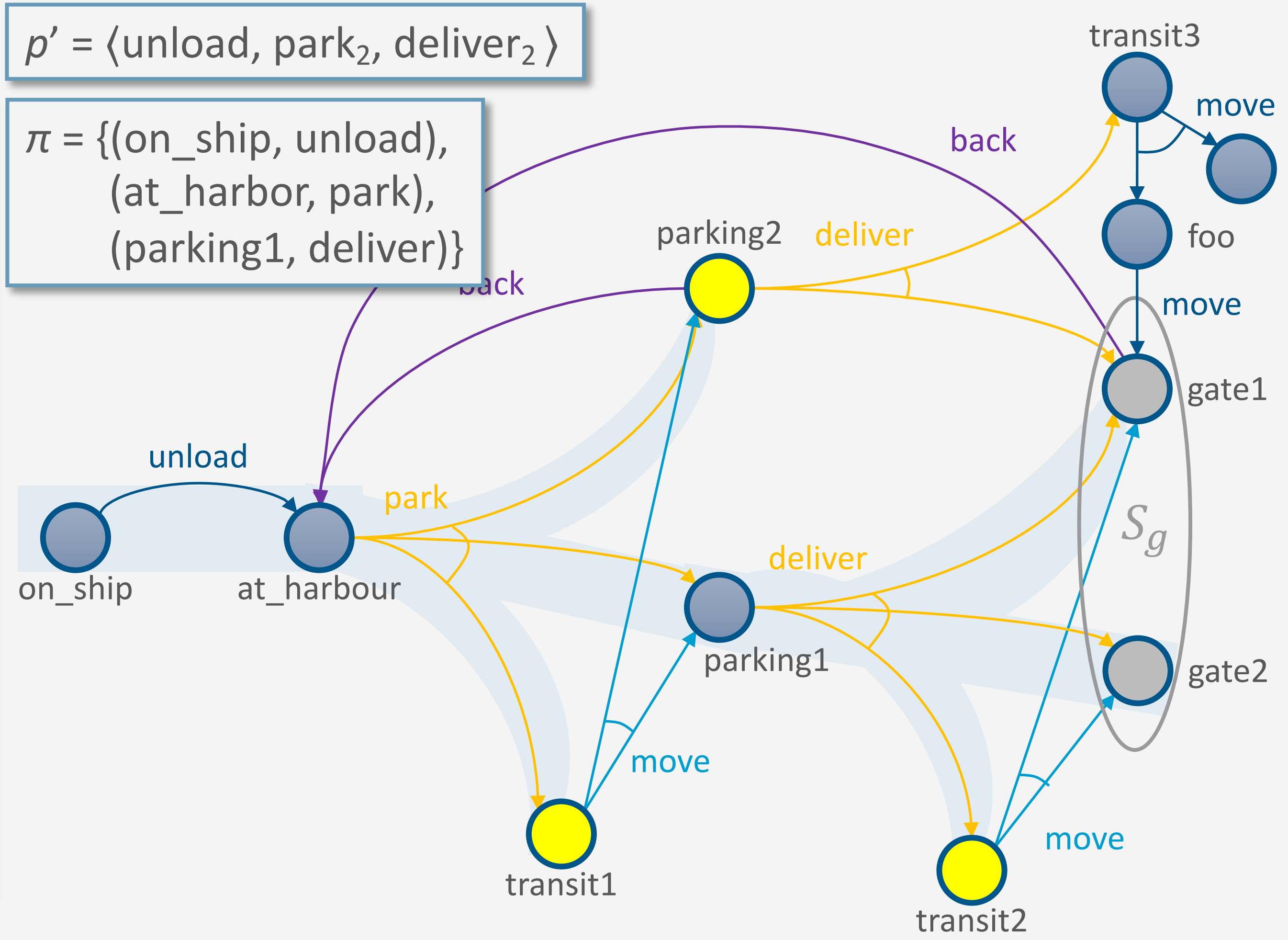


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s0,Sg)
    ...
    Σd ← mk-deterministic(Σ)
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σd,s,Sg)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle$unload, park$_2$, deliver$_2\rangle$

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver)}

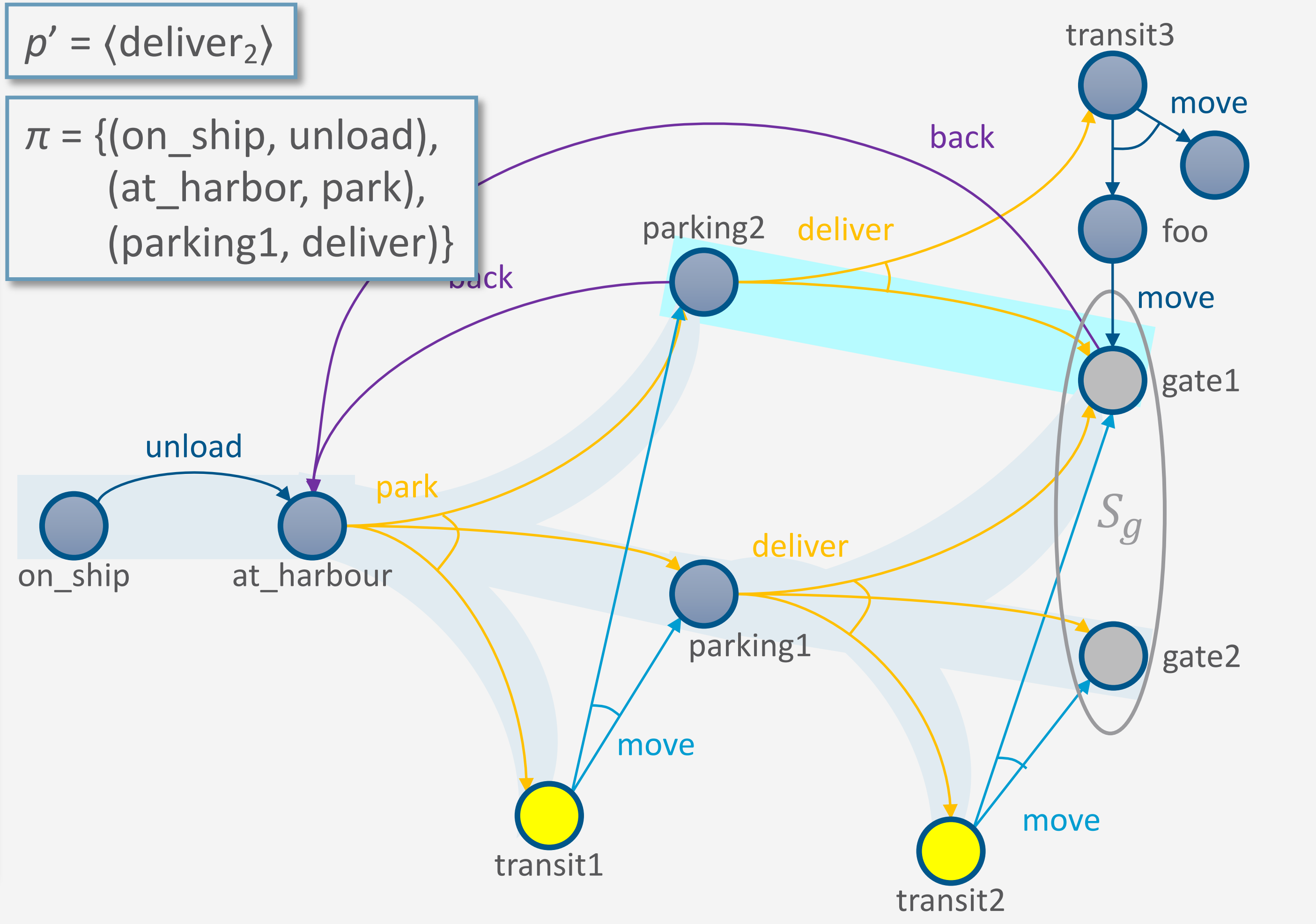


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s0,Sg)
    ...
    Σd ← mk-deterministic(Σ)
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σd,s,Sg)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle \text{deliver}_2 \rangle$

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver)}

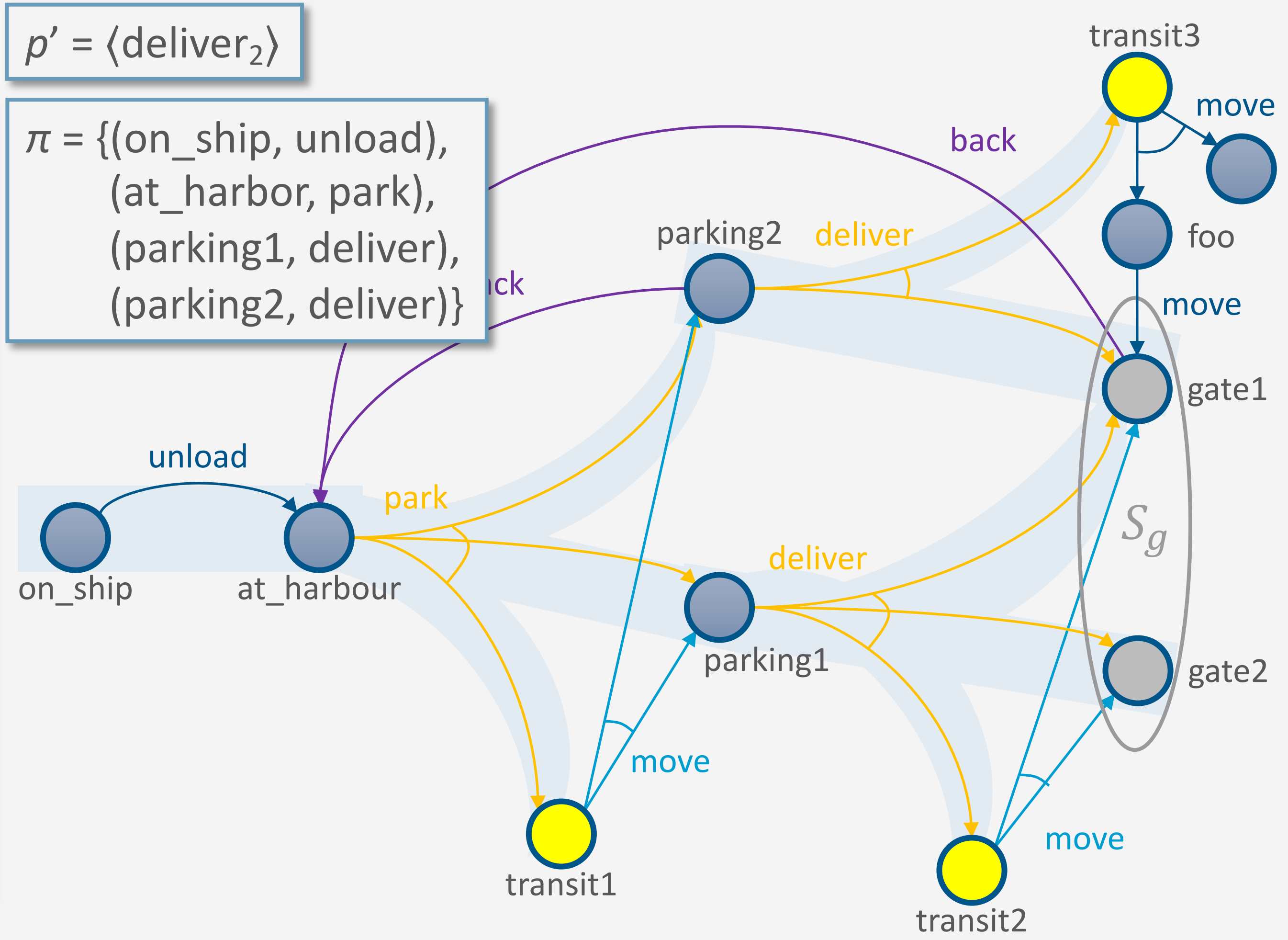


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s₀,S_g)
    ...
    Σ_d ← mk-deterministic(Σ)
    loop
        Q ← leaves(s₀,π) \ S_g
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s₀,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σ_d,s,S_g)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle \text{deliver}_2 \rangle$

$\pi = \{(\text{on\_ship, unload}), (\text{at\_harbor, park}), (\text{parking1, deliver}), (\text{parking2, deliver})\}$

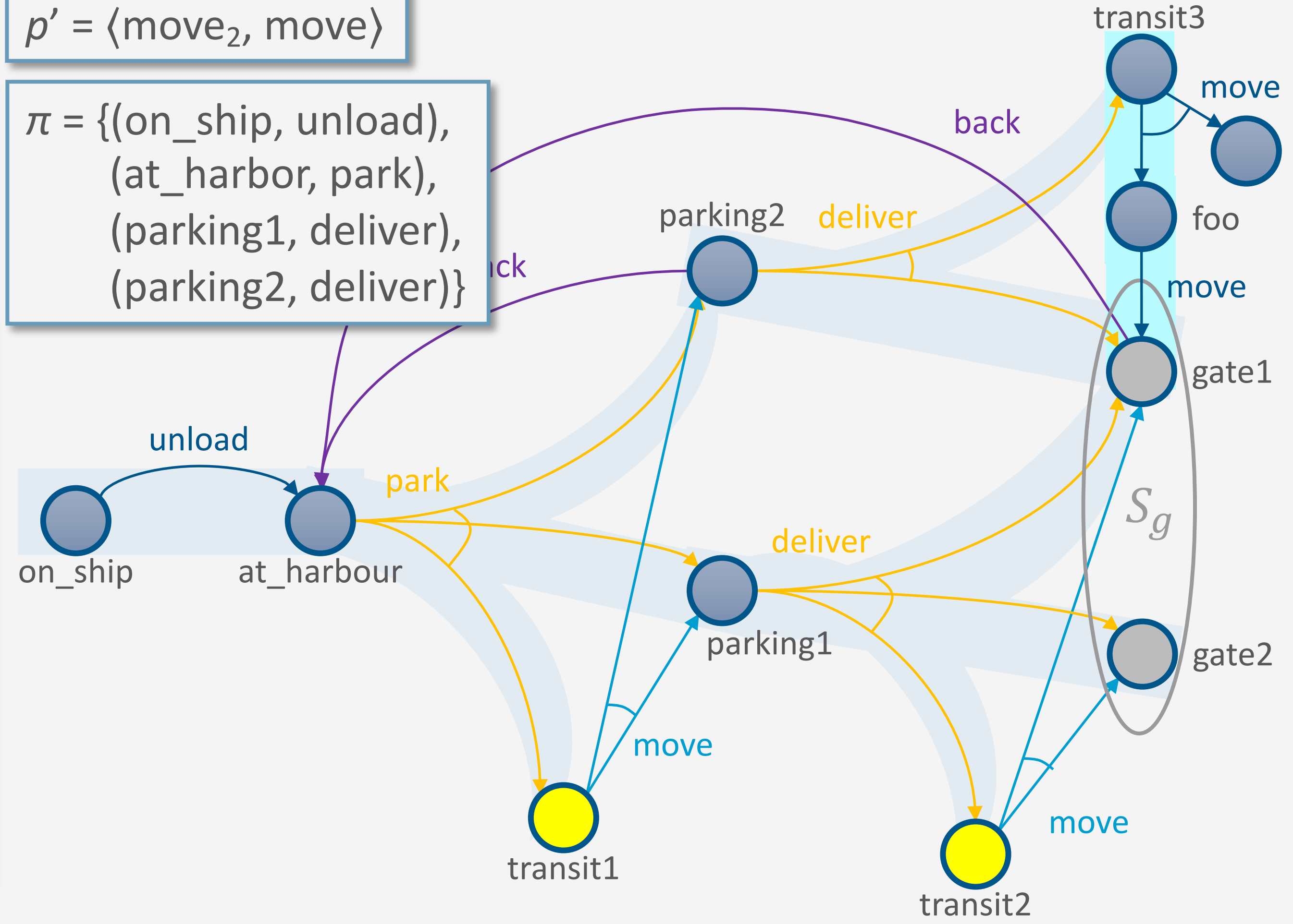


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s_0,S_g)
    ...
    Σ_d ← mk-deterministic(Σ)
    loop
        Q ← leaves(s_0,π) \ S_g
        if Q = Ø then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s_0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σ_d,s,S_g)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle move_2, move \rangle$

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver)}

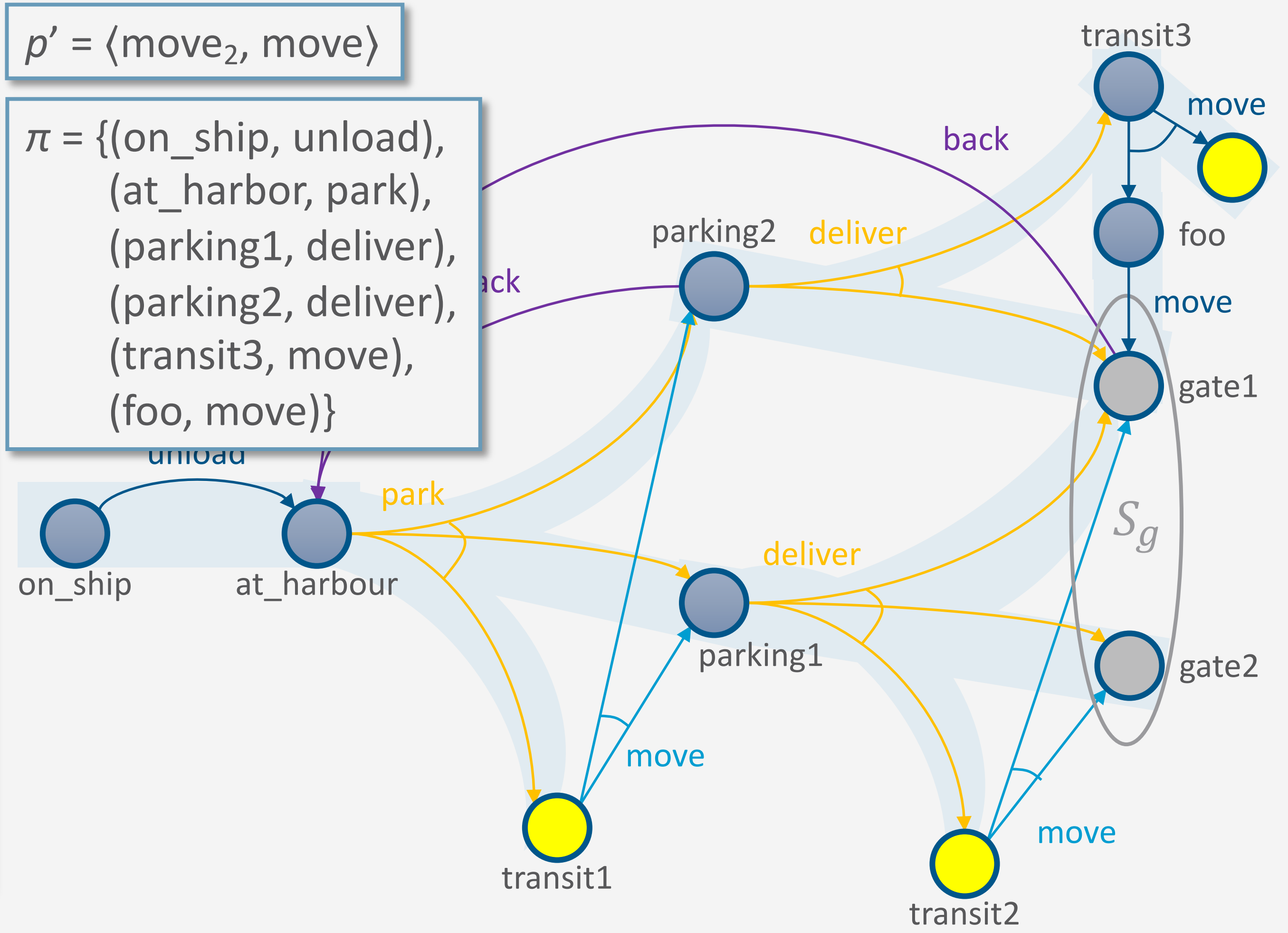


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s0,Sg)
    ...
    Σd ← mk-deterministic(Σ)
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σd,s,Sg)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle move_2, move \rangle$

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(transit3, move),
(foo, move)}

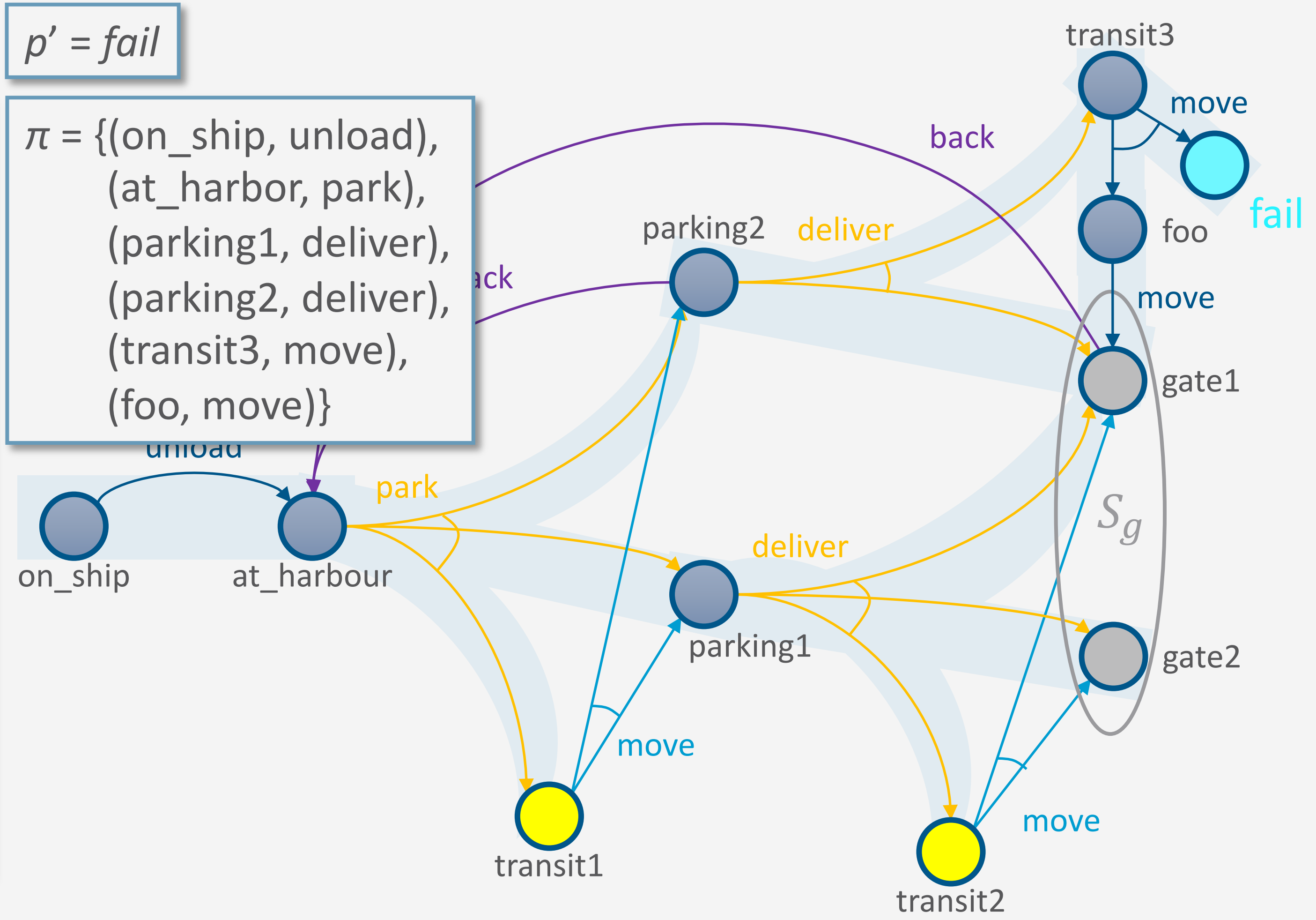


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s_0,S_g)
    ...
    Σ_d ← mk-deterministic(Σ)
    loop
        Q ← leaves(s_0,π) \ S_g
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s_0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σ_d,s,S_g)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

p' = fail

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(transit3, move),
(foo, move)}



Example

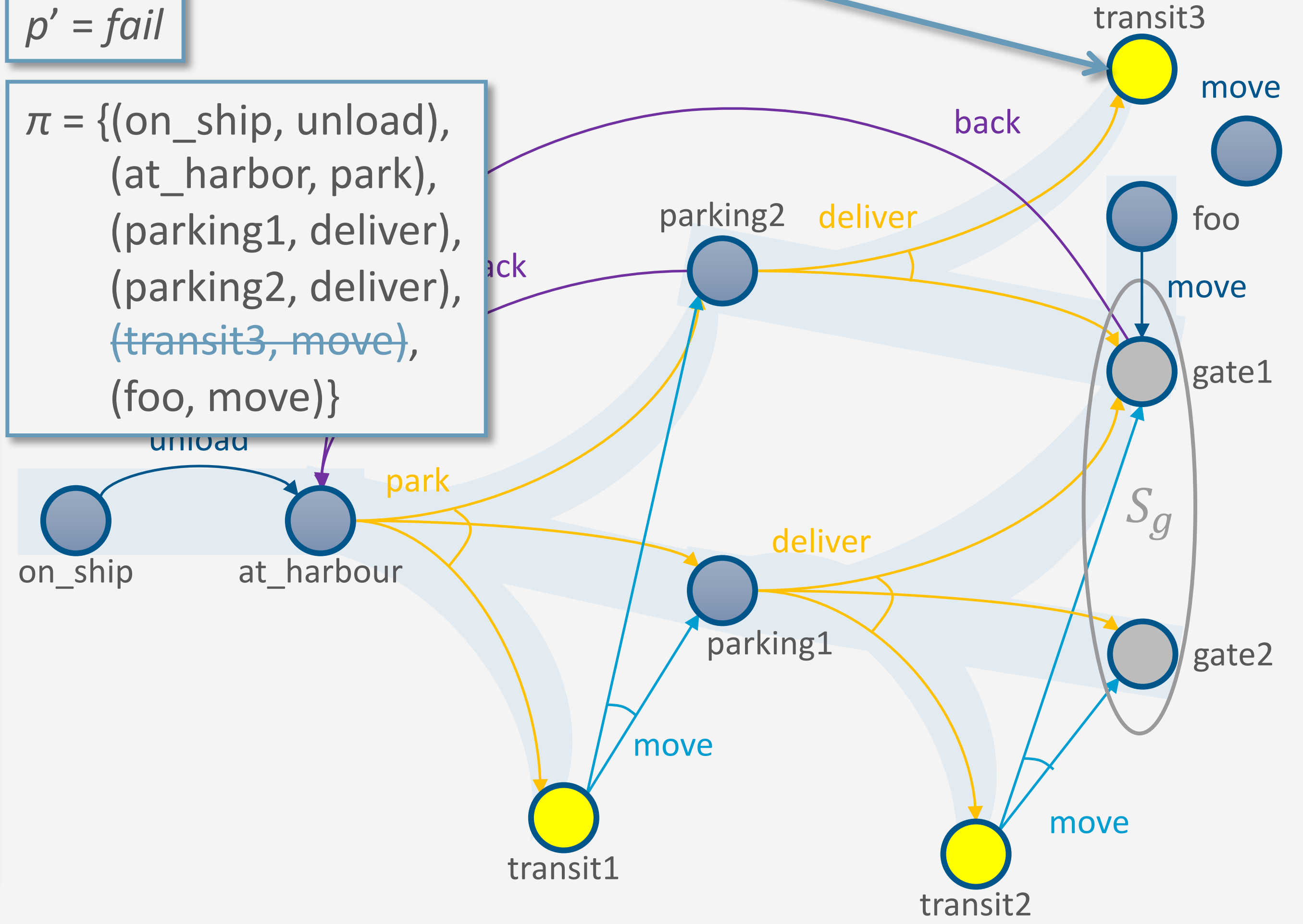
```
Find-Safe-Solution-by-Determinisation(Σ,s_0,S_g)
    ...
    Σ_d ← mk-deterministic(Σ)
    loop
        Q ← leaves(s_0,π) \ S_g
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s_0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σ_d,s,S_g)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

Modify Σ_d to make *move* inapplicable

Nondeterministic

p' = fail

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
~~(transit3, move)~~,
(foo, move)}

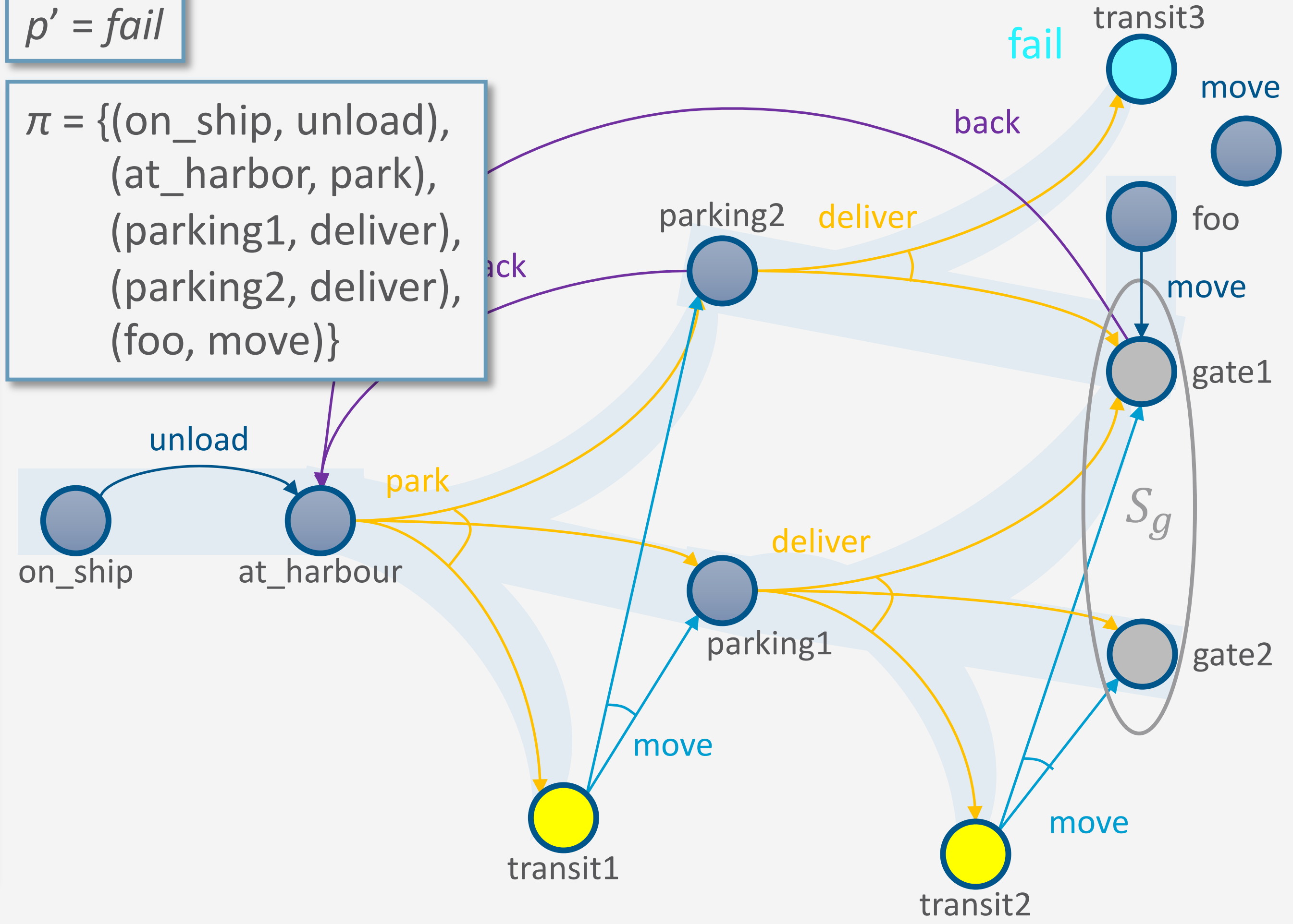


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s0,Sg)
    ...
    Σd ← mk-deterministic(Σ)
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σd,s,Sg)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = fail$

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(foo, move)}



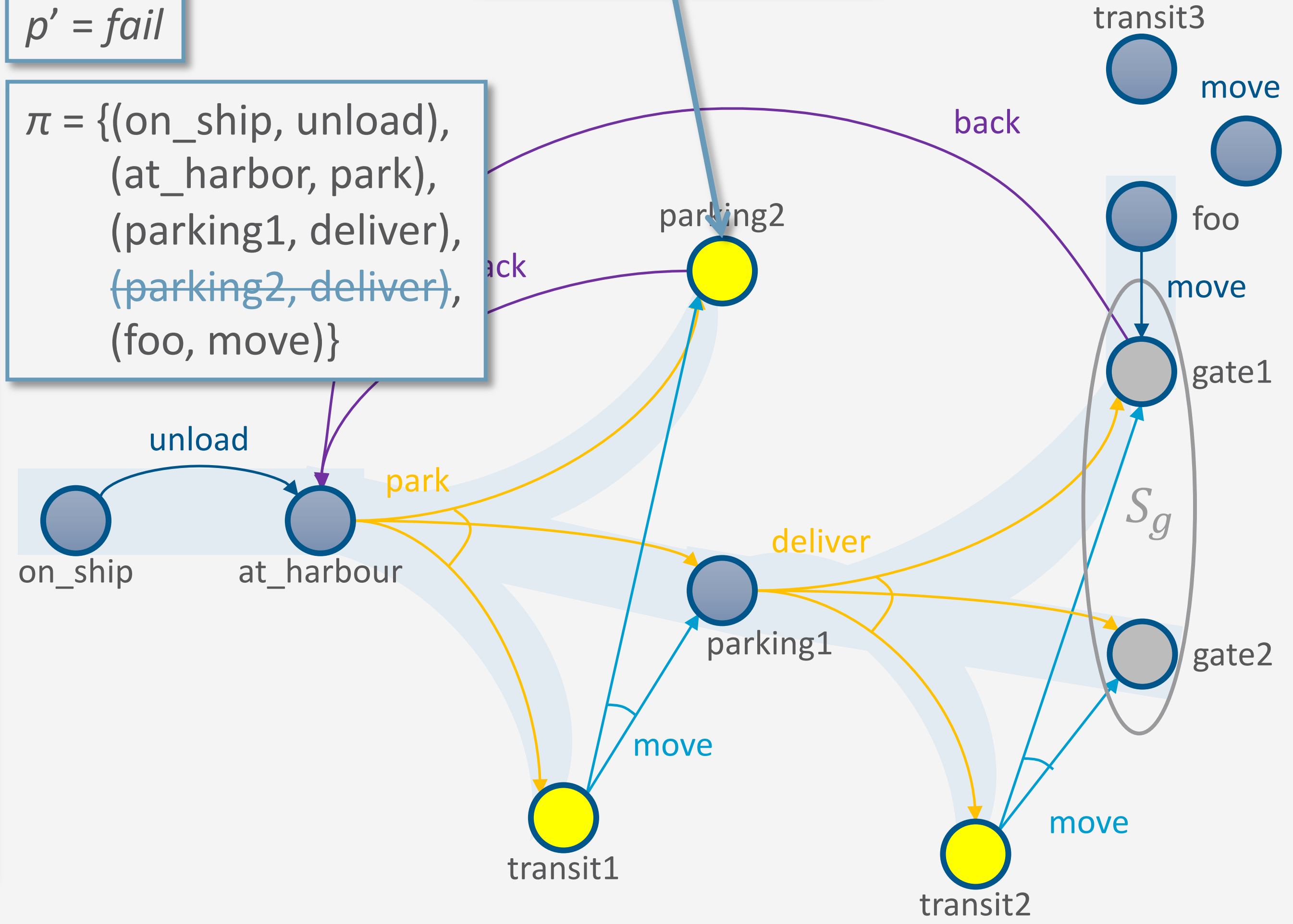
Example

```
Find-Safe-Solution-by-Determinisation(Σ,s0,Sg)
   ...
   Σd ← mk-deterministic(Σ)
   loop
      Q ← leaves(s0,π) \ Sg
      if Q = ∅ then
         π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
         return π
      select arbitrarily s ∈ Q
      p' ← Forward-search(Σd,s,Sg)
      if p' ≠ fail then
         π ← Plan2policy(p',s)
         π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
      else if … else
         for every s', a s.t. s ∈ γ(s',a) do
            π ← π \ {(s',a)}
            make actions in determinisation
               not applicable in s'
```

p' = *fail*

Modify Σ_d to make *deliver* inapplicable

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
~~(parking2, deliver)~~,
(foo, move)}

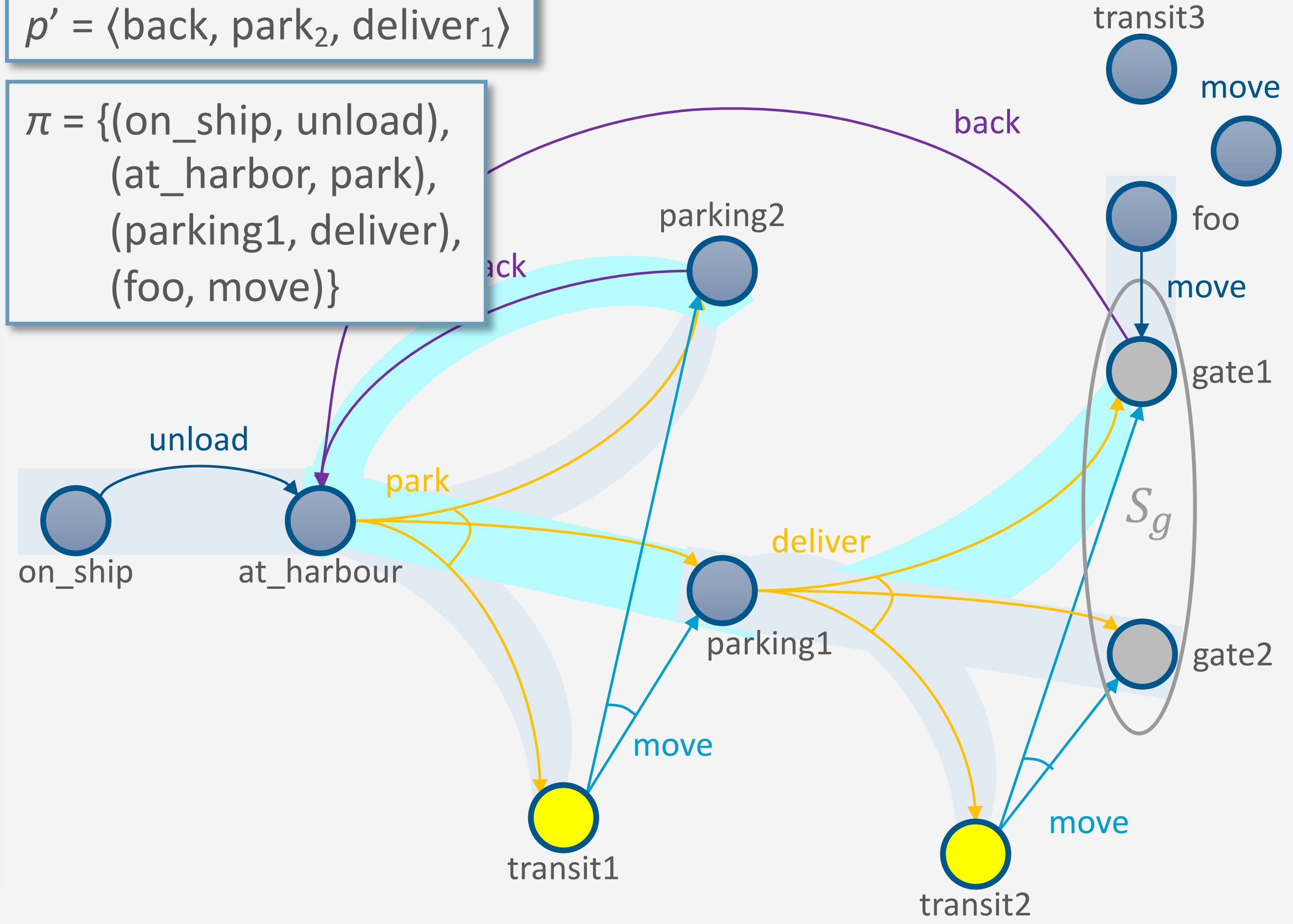


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s0,Sg)
    ...
    Σd ← mk-deterministic(Σ)
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σd,s,Sg)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle$back, park$_2$, deliver$_1\rangle$

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(foo, move)}

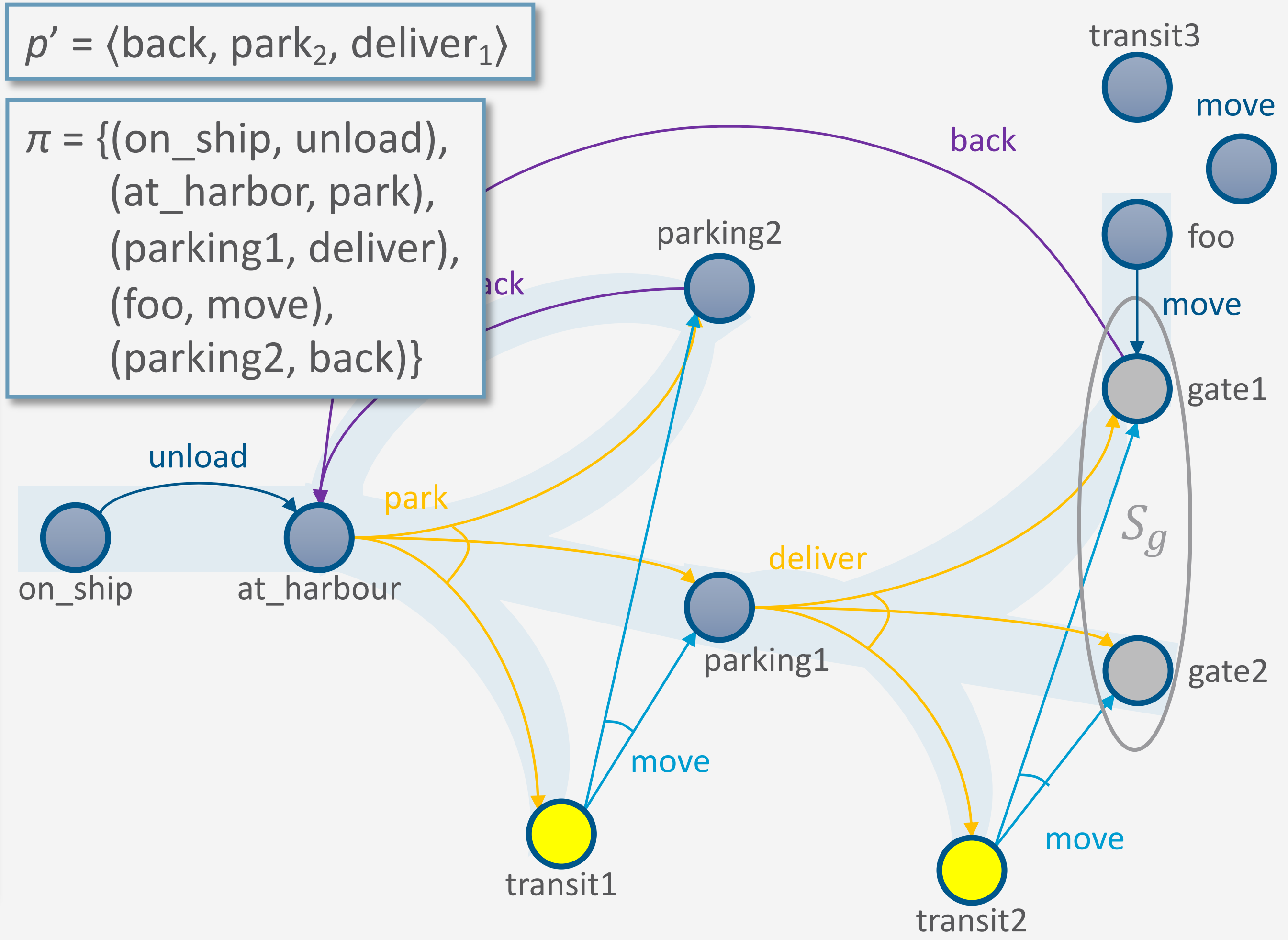


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s0,Sg)
    ...
    Σd ← mk-deterministic(Σ)
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σd,s,Sg)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle \text{back}, \text{park}_2, \text{deliver}_1 \rangle$

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(foo, move),
(parking2, back)}

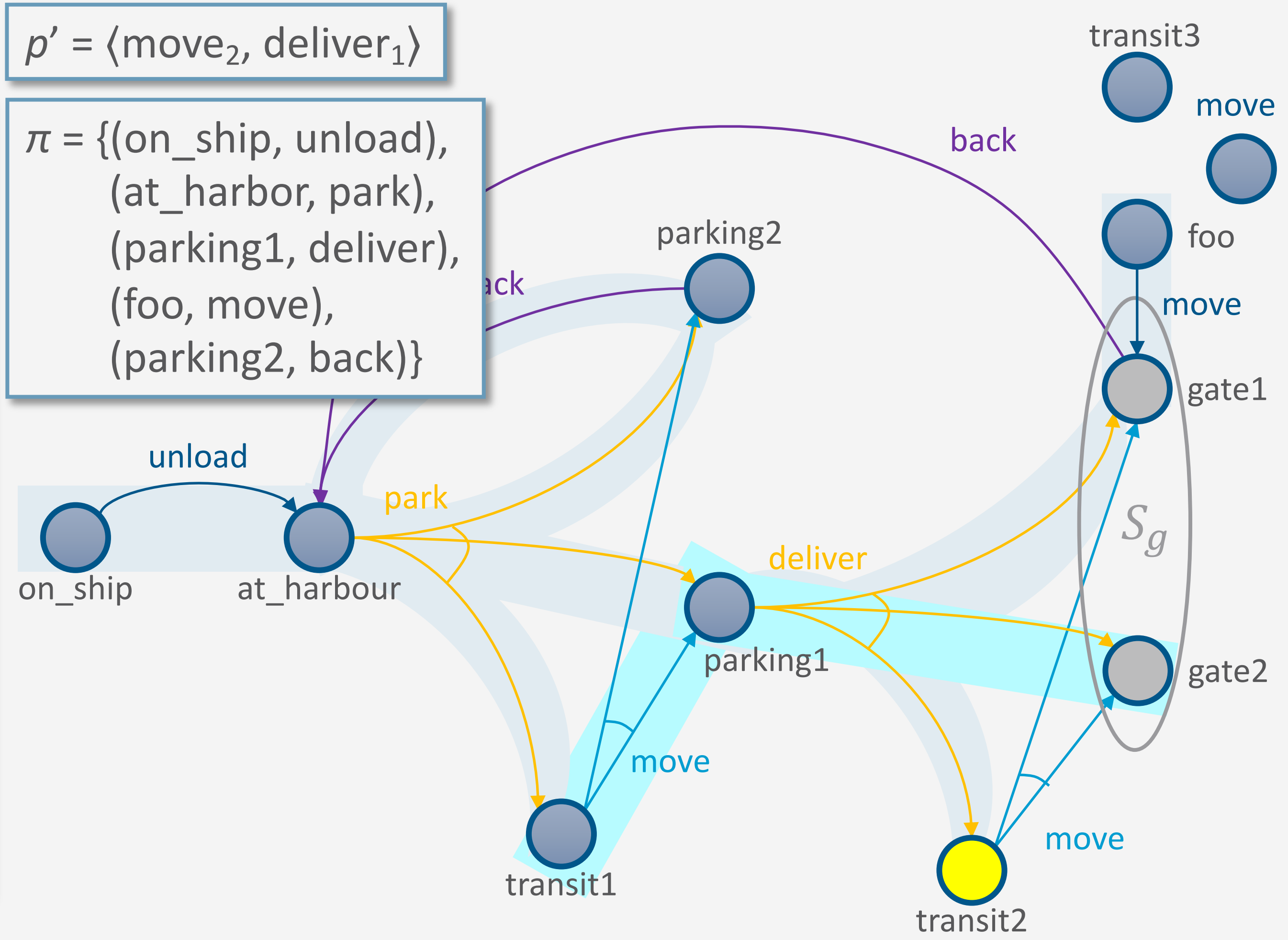


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s₀,S_g)
    ...
    Σ_d ← mk-deterministic(Σ)
    loop
        Q ← leaves(s₀,π) \ S_g
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s₀,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σ_d,s,S_g)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle move_2, deliver_1 \rangle$

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(foo, move),
(parking2, back)}

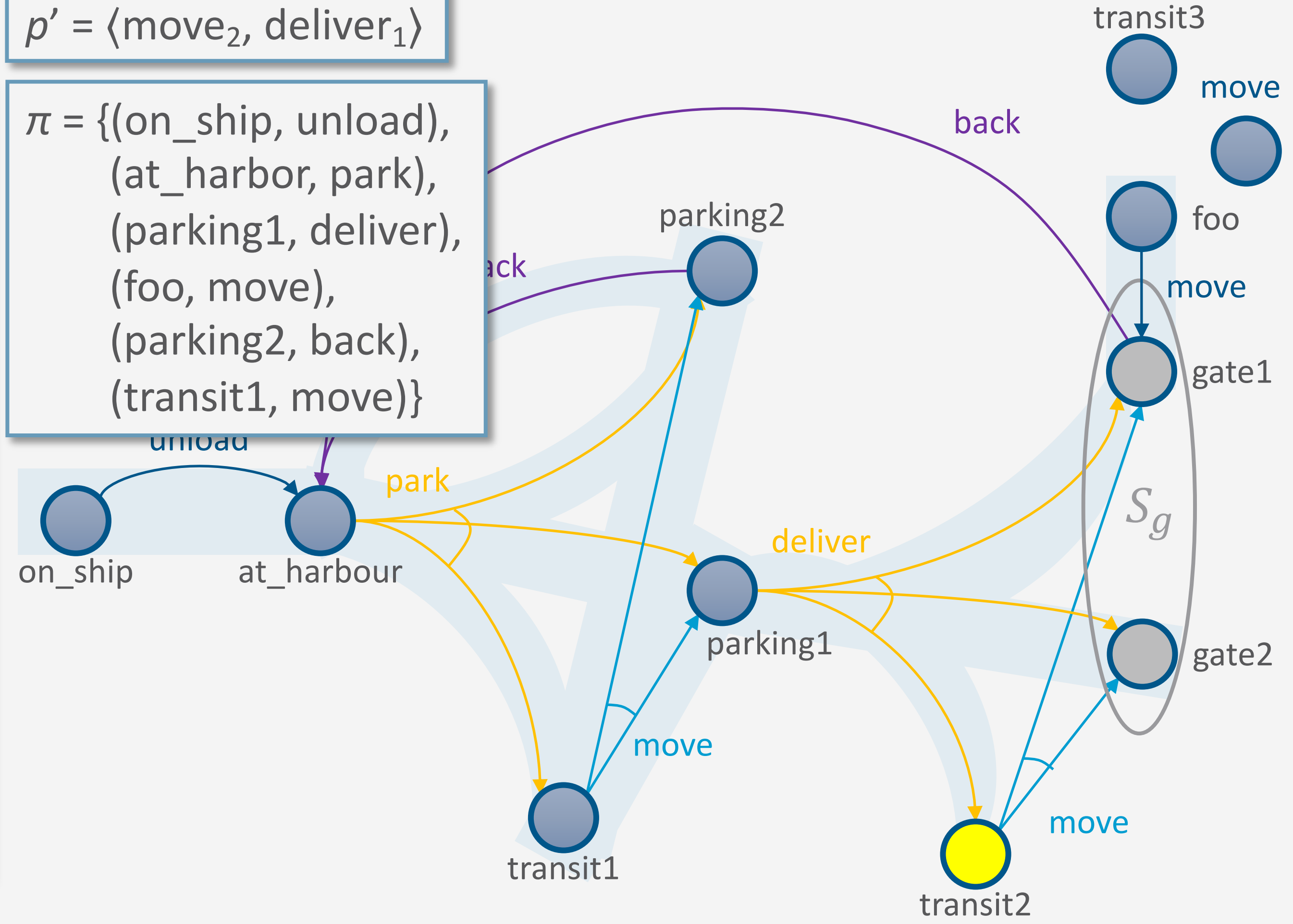


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s₀,S_g)
    ...
    Σ_d ← mk-deterministic(Σ)
    loop
        Q ← leaves(s₀,π) \ S_g
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s₀,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σ_d,s,S_g)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle move_2, deliver_1 \rangle$

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(foo, move),
(parking2, back),
(transit1, move)}

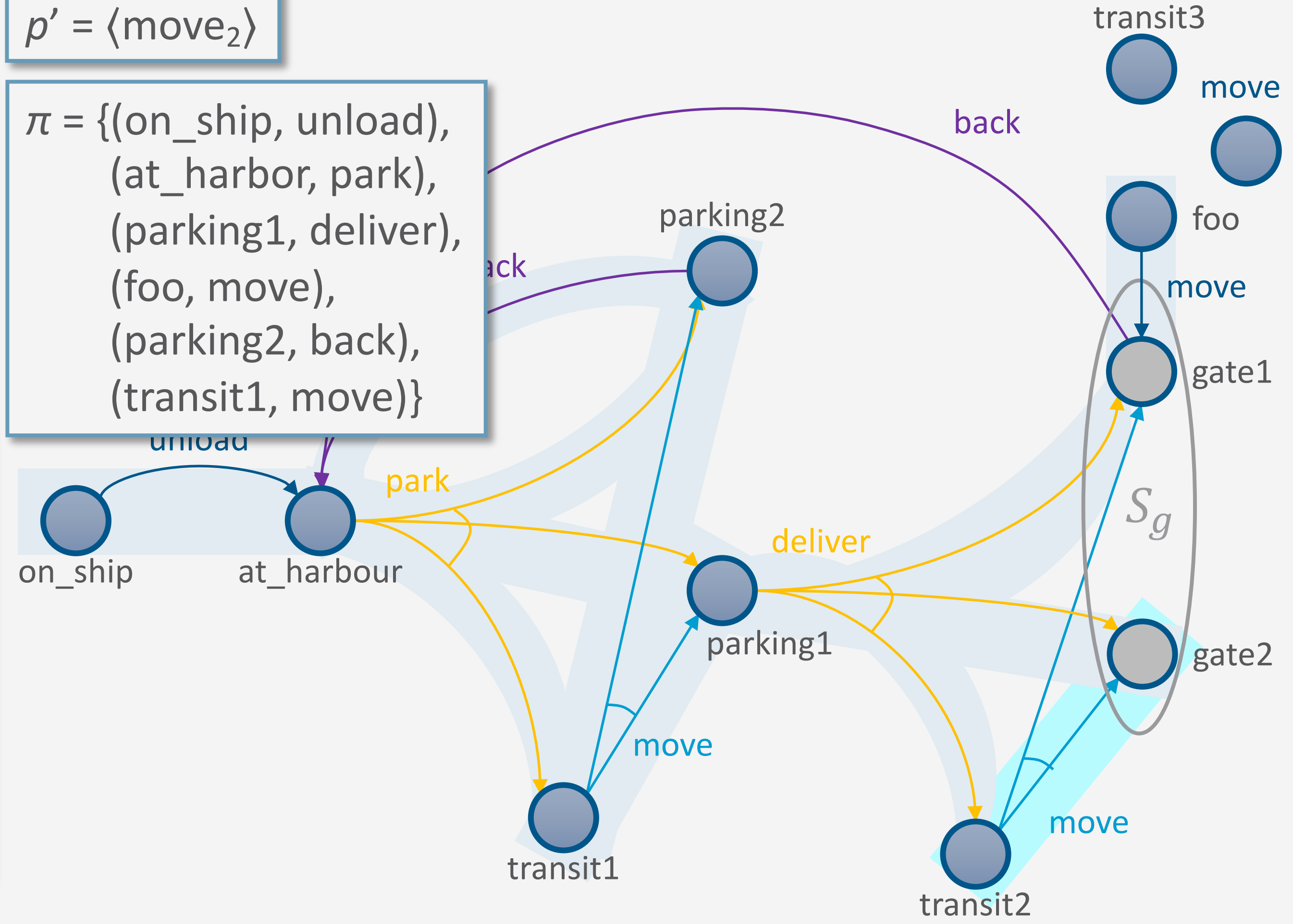


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s_0,S_g)
    ...
    Σ_d ← mk-deterministic(Σ)
    loop
        Q ← leaves(s_0,π) \ S_g
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s_0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σ_d,s,S_g)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle \text{move}_2 \rangle$

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(foo, move),
(parking2, back),
(transit1, move)}

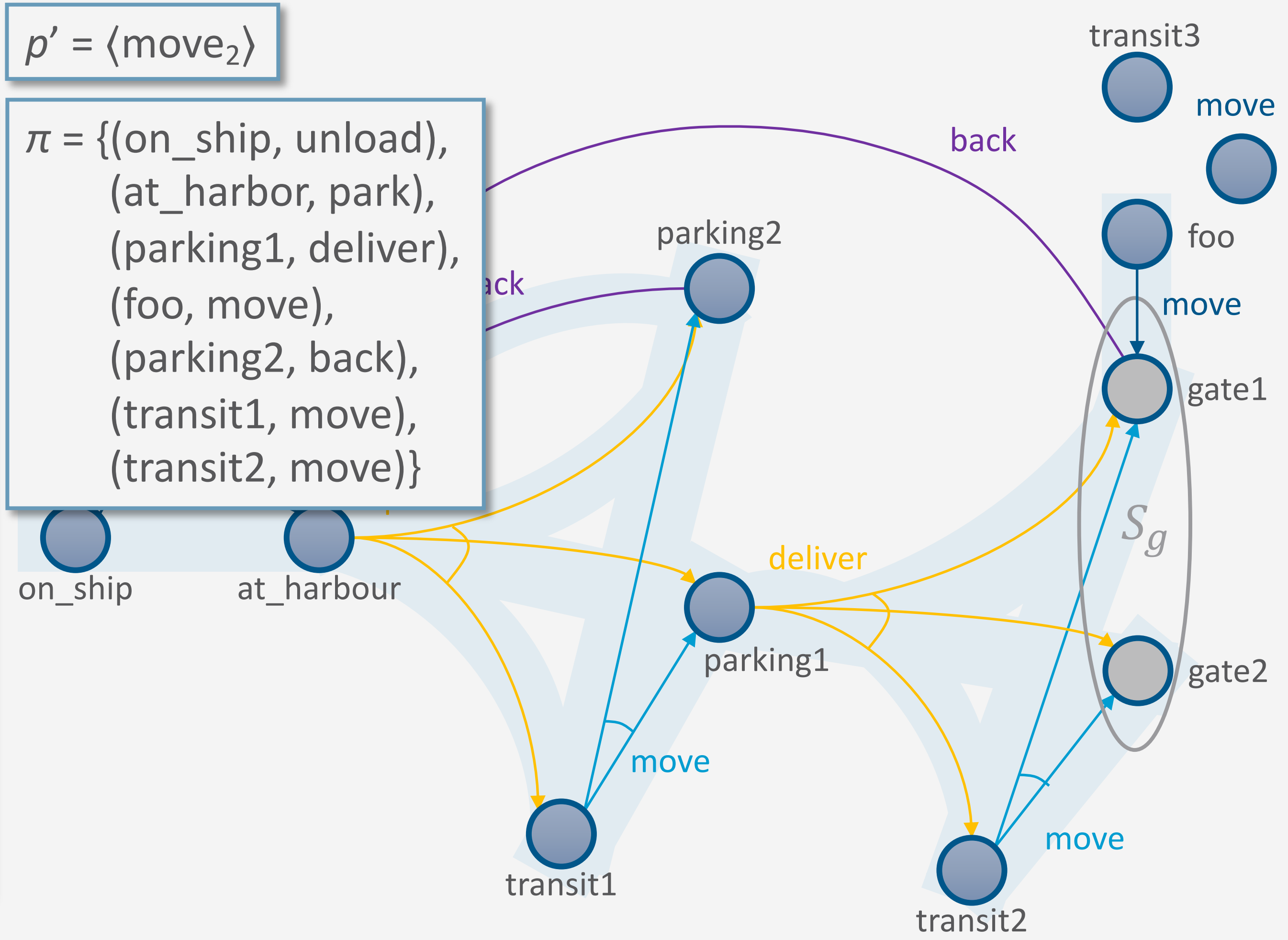


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s₀,S_g)
    ...
    Σ_d ← mk-deterministic(Σ)
    loop
        Q ← leaves(s₀,π) \ S_g
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s₀,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σ_d,s,S_g)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if … else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make actions in determinisation
                    not applicable in s'
```

$p' = \langle move_2 \rangle$

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(foo, move),
(parking2, back),
(transit1, move),
(transit2, move)}

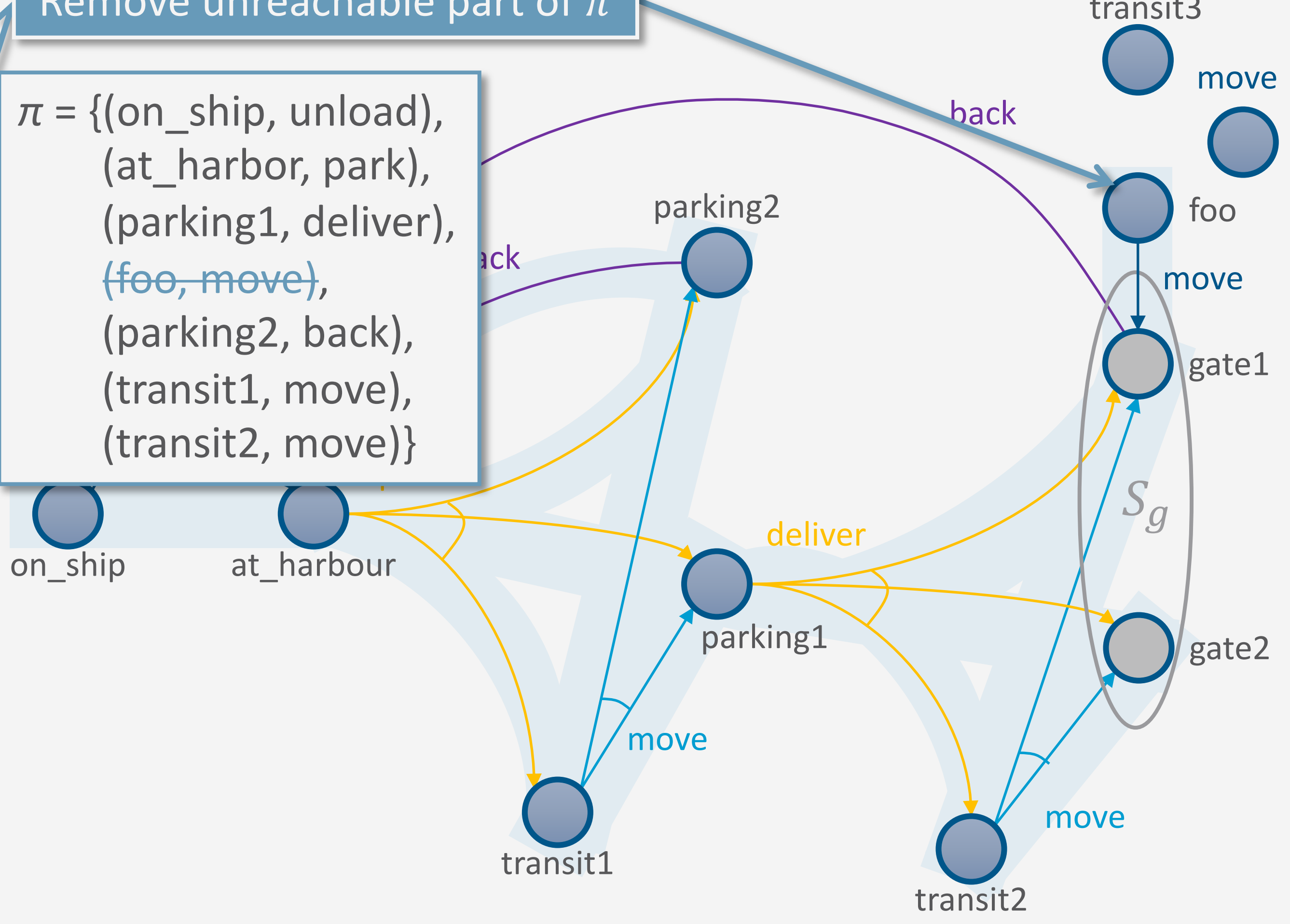


Example

```
Find-Safe-Solution-by-Determinisation(Σ,s0,Sg)
    ...
    Σd ← mk-deterministic(Σ)
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        p′ ← Forward-search(Σd,s,Sg)
        if p′ ≠ fail then
            π ← Plan2policy(p′,s)
            π ← π ∪ {(s,a) ∈ π′ | s ∉ dom(π)}
        else if … else
            for every s′, a s.t. s ∈ γ(s′,a) do
                π ← π \ {(s′,a)}
                make actions in determinisation
                    not applicable in s′
```

Remove unreachable part of π

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
~~(foo, move)~~,
(parking2, back),
(transit1, move),
(transit2, move)}



Making Actions Inapplicable

- Modify Σ_d to make actions inapplicable: Exponential time in worst-case
- Better: table of bad state-action pairs
  - For every (s', a) s.t. $s \in \gamma(s', a)$, $Bad[s'] \leftarrow Bad[s'] \cup determinization(a)$
- Modify classical planner to take the table as an argument
  - If s is current state, only choose actions in $Applicable(s) \setminus Bad(s)$

```
Find-Safe-Solution-by-Determinisation(Σ,s0,Sg)
    if s0 ∈ Sg then
        return ∅
    if Applicable(s0) = ∅ then
        return failure
    π ← ∅
    Σd ← mk-deterministic(Σ)
    loop
        Q ← leaves(s0,π) \ Sg
        if Q = ∅ then
            π ← π \ {(s,a) ∈ π | s ∉ γ̂(s0,π)}
            return π
        select arbitrarily s ∈ Q
        p' ← Forward-search(Σd,s,Sg)
        if p' ≠ fail then
            π ← Plan2policy(p',s)
            π ← π ∪ {(s,a) ∈ π' | s ∉ dom(π)}
        else if s = s0 then
            return failure
        else
            for every s', a s.t. s ∈ γ(s',a) do
                π ← π \ {(s',a)}
                make the actions in the
                        determinisation not
                        applicable in s'
```

Intermediate Summary

- Determinisation Techniques
  - Guided-find-safe-solution
    - Call find-solution to get an unsafe solution
    - Call find-solution additional times on the leaves
  - Find-safe-solution-by-determinization
    - Use determinized actions
    - Call classical planner rather than find-solution
    - If dead-ends are encountered, modify actions that lead to them

Outline per the Book

5.2 Planning Problem

- Planning domains
- Plans as policies
- Planning problems and solutions

5.3 And/Or Graph Search

- Planning by forward search

5.5 Determinisation Techniques

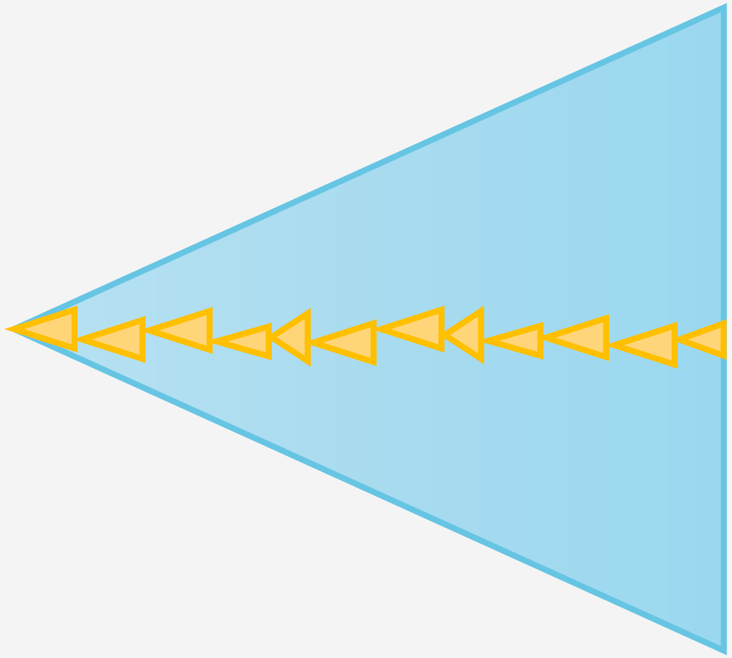
- Guided planning for safe solutions
- Planning for safe solutions by determinisation

5.6 Online Approaches

- Lookahead
- Lookahead by determinisation
- Lookahead with a bounded number of steps

Online Approaches

- Motivation
  1. Planning models are approximate – execution seldom works out as planned
  2. Large problems may require too much planning time
- 2nd motivation even more stronger in nondeterministic domains
  - Nondeterminism makes planning exponentially harder
    - Exponentially more time, exponentially larger policies



Offline vs. Runtime Search Spaces

Online Approaches

- Need to identify good actions without exploring entire search space (partial planning)
  - Can be done using heuristic estimates
- Some domains are safely explorable
  - Safe to create partial plans, because goal states are reachable from all situations
- Other domains contain dead-ends, partial planning will not guarantee success
  - Can get trapped in dead ends that we would have detected if we had planned fully
    - No applicable actions
      - Robot goes down a steep incline and cannot come back up
    - Applicable actions, but caught in a loop
      - Robot goes into a collection of rooms from which there is no exit
  - However, partial planning can still make success more likely

Lookahead-Partial-Plan

- Adaptation of Run-Lazy-Lookahead (Ch. 2)
- Lookahead is any planning algorithm that returns a policy π
  - π may be partial solution, or unsafe solution
  - Lookahead-Partial-Plan executes π as far as it will go, then calls Lookahead again
  - θ context-dependent vector of parameters to restrict in some way the search for a solution

```
Lookahead-Partial-Plan(Σ,s0,Sg,θ)
    s ← s0
    while s ∉ Sg and Applicable(s) ≠ ∅ do
        π ← Lookahead(s,θ)
        if π = ∅ then
            return failure
        else
            perform partial plan π
            s ← observe current state
```

Inputs:

- Planning problem (Σ, s_0, S_g)
- Vector of parameters θ
- *Same for next versions*

FS-Replan

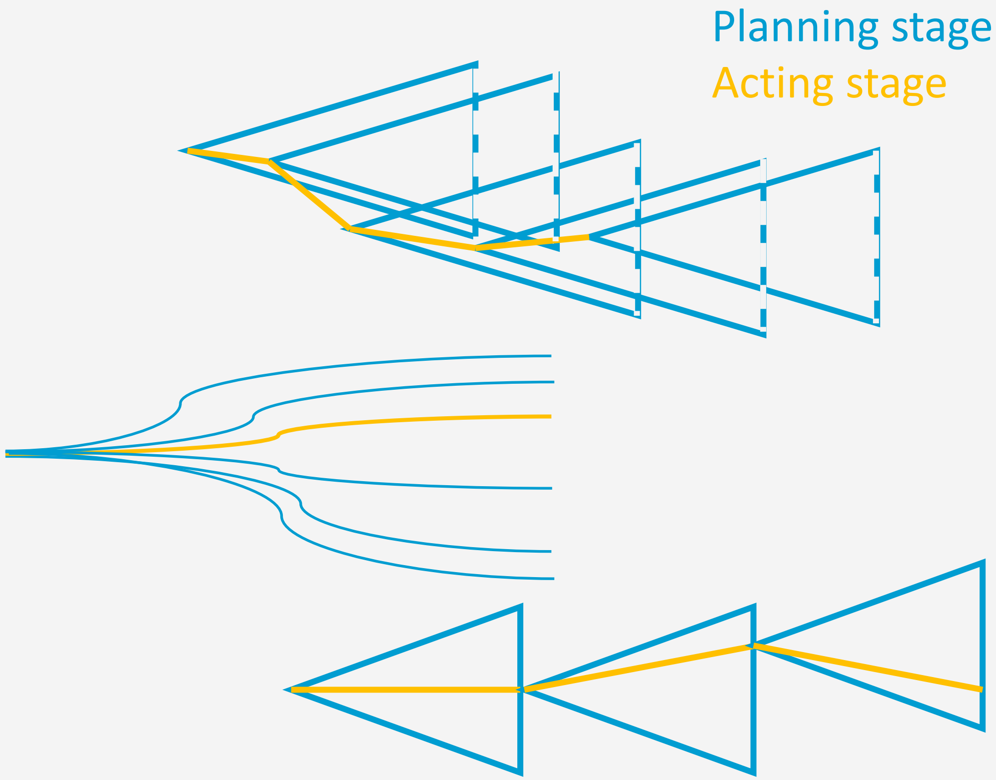
- Adaptation of Run-Lookahead (Ch. 2)
- Calls Forward-Search (Ch. 2) on determinised domain, converts to a policy
  - Unsafe solution
- Generalisation:
  - Lookahead can be any planning algorithm that returns a policy π

```
FS-Replan(Σ,s,S_g)
    π_d ← ∅
    while s ∉ S_g and Applicable(s) ≠ ∅ do
        if π_d undefined for s then
            π_d ← Plan2policy(Forward-search(Σ_d,s,S_g),s)
            if π_d = failure then
                return failure
        perform action π_d(s)
        s ← observe resulting state
```

```
Generalised-FS-Replan(Σ,s,S_g,θ)
    π_d ← ∅
    while s ∉ S_g and Applicable(s) ≠ ∅ do
        if π_d undefined for s then
            π_d ← Lookahead(s,θ)
            if π_d = failure then
                return failure
        perform action π_d(s)
        s ← observe resulting state
```

Possibilities for Lookahead

- Lookahead could be one of the algorithms we discussed earlier
  - Find-Safe-Solution
  - Find-Acyclic-Solution
  - Guided-Find-Safe-Solution
  - Find-Safe-Solution-by-Determinization
- What if it does not have time to run to completion?
  - Can use the same techniques, we discussed earlier
    - Receding horizon
    - Sampling
    - Subgoaling
    - Iterative Deepening



Possibilities for Lookahead (cont'd)

- Full horizon, limited breadth:
  - Look for solution that works for *some* of the outcomes
  - E.g., modify *Find-Acyclic-Solution* to examine i outcomes of every action
- Iterative broadening:
  - For $i = 1$, increase i by 1 until time runs out
    - Look for a solution that handles i outcomes per action

```
T ← i elements of γ(s,a)\Dom(π)
Frontier ← Frontier ∪ T
```

```
Find-Acyclic-Solution(Σ,s0,Sg)
    π ← ∅
    Frontier ← {s0}
    for every s ∈ Frontier \ Sg do
        Frontier ← Frontier \ {s}
        if Applicable(s) = ∅ then
            return failure
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s,a)
        Frontier ← Frontier ∪ (γ(s,a) \ Dom(π))
        if has-loops(π,s,Frontier) then
            return failure
    return π
```

Input
- Planning problem (Σ, s_0, S_g)

MinMax Learning Real Time A* (MinMax LRTA*)

- Lookahead with a bounded number of steps
- Input: Planning problem (Σ, s_0, S_g)
- Loop
  - Choose an action a that (according to a heuristics h) has optimal worst-case cost
    - Update $h(s)$ to use a's worst-case cost
    - Perform a

Looks ahead 1 step; can be modified to look ahead k steps

Assumes each action has cost 1; can easily be modified to use cost $\neq 1$ by replacing 1 with $c(s)$

Min-Max-LRTA*(Σ, s_0, S_g)

$s \leftarrow s_0$

while $s \notin S_g$ and Applicable(s) $\neq \emptyset$ **do**

  $a \leftarrow \mathrm{argmin}_{a \in \mathrm{Applicable}(s)} \max_{s' \in \gamma(s,a)} h(s')$

  $h(s) \leftarrow \max\{h(s), 1 + \max_{s' \in \gamma(s,a)} h(s')\}$

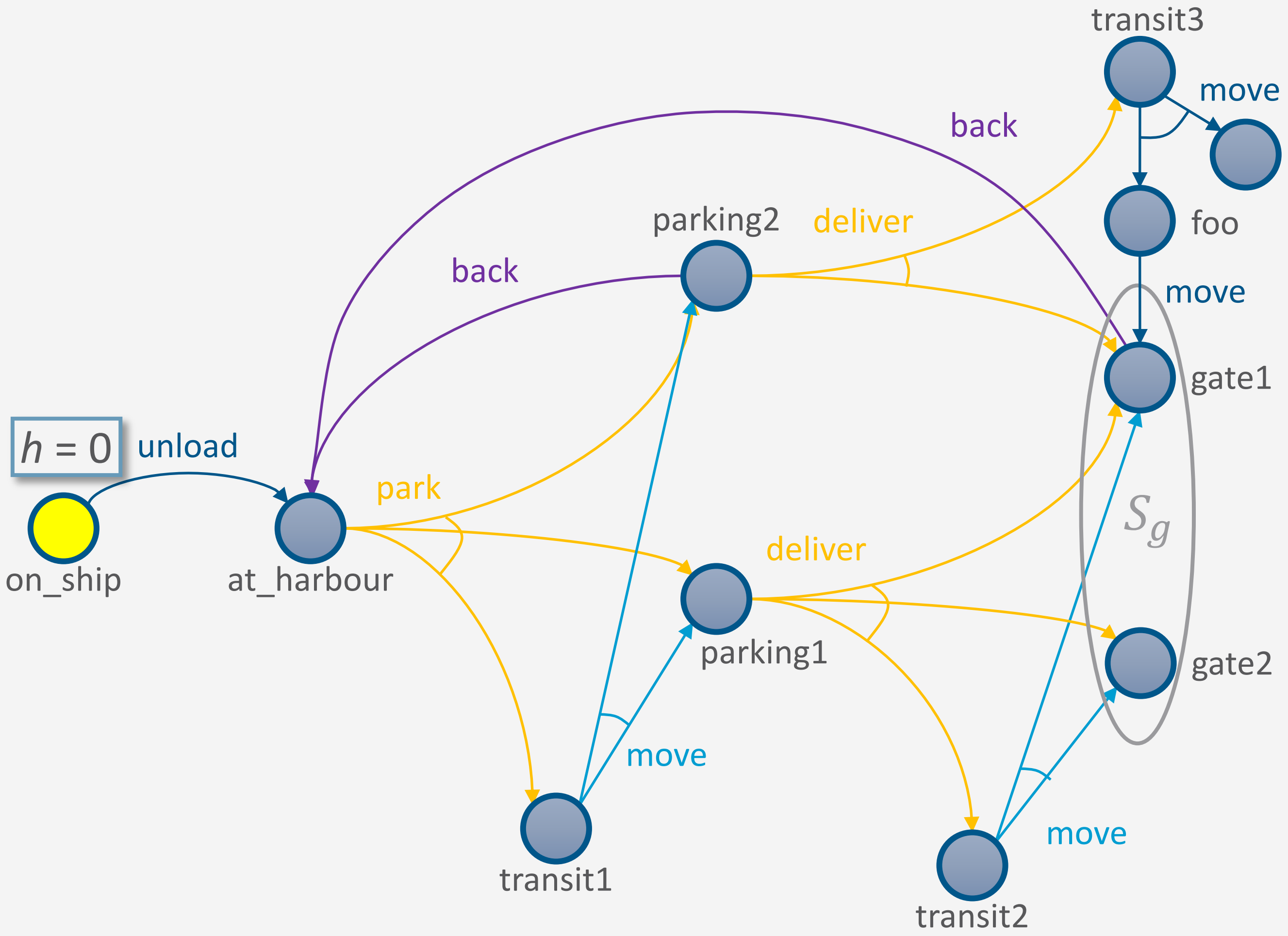
  perform action a

  $s \leftarrow$ the current state

Example

```
Min-Max-LRTA*(Σ,s₀,S_g)
    s ← s₀
    while s ∉ S_g and Applicable(s) ≠ ∅ do
        a ← argmin_{a∈Applicable(s)} max_{s'∈γ(s,a)} h(s')
        h(s) ← max{h(s),1 + max_{s'∈γ(s,a)} h(s')}
        perform action a
        s ← the current state
```

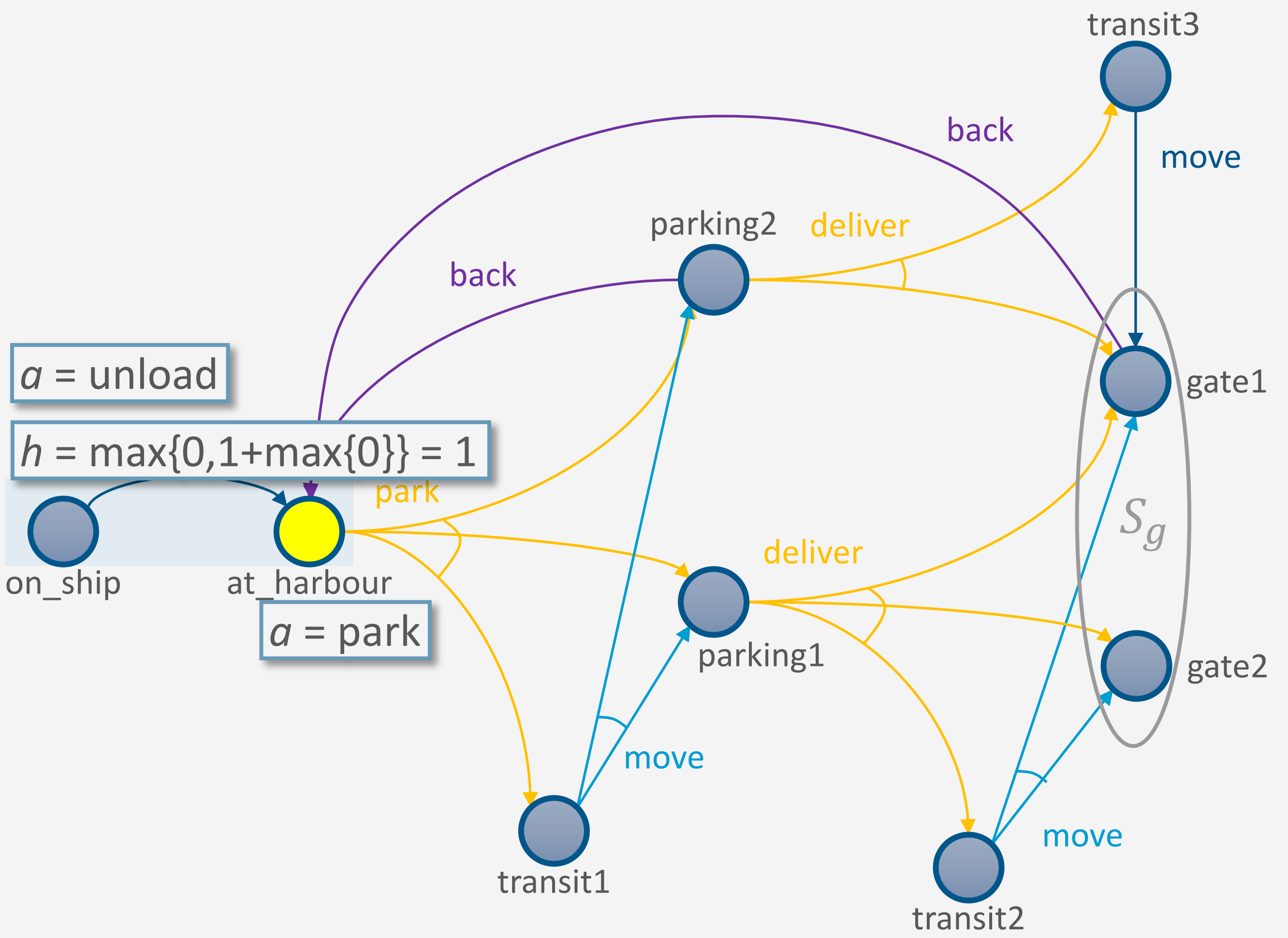
Suppose that initially, $h(s) = 0$ for every state s



Example

```
Min-Max-LRTA*(Σ,s₀,S_g)
    s ← s₀
    while s ∉ S_g and Applicable(s) ≠ ∅ do
        a ← argmin_{a∈Applicable(s)} max_{s′∈γ(s,a)} h(s′)
        h(s) ← max{h(s),1 + max_{s′∈γ(s,a)} h(s′)}
        perform action a
        s ← the current state
```

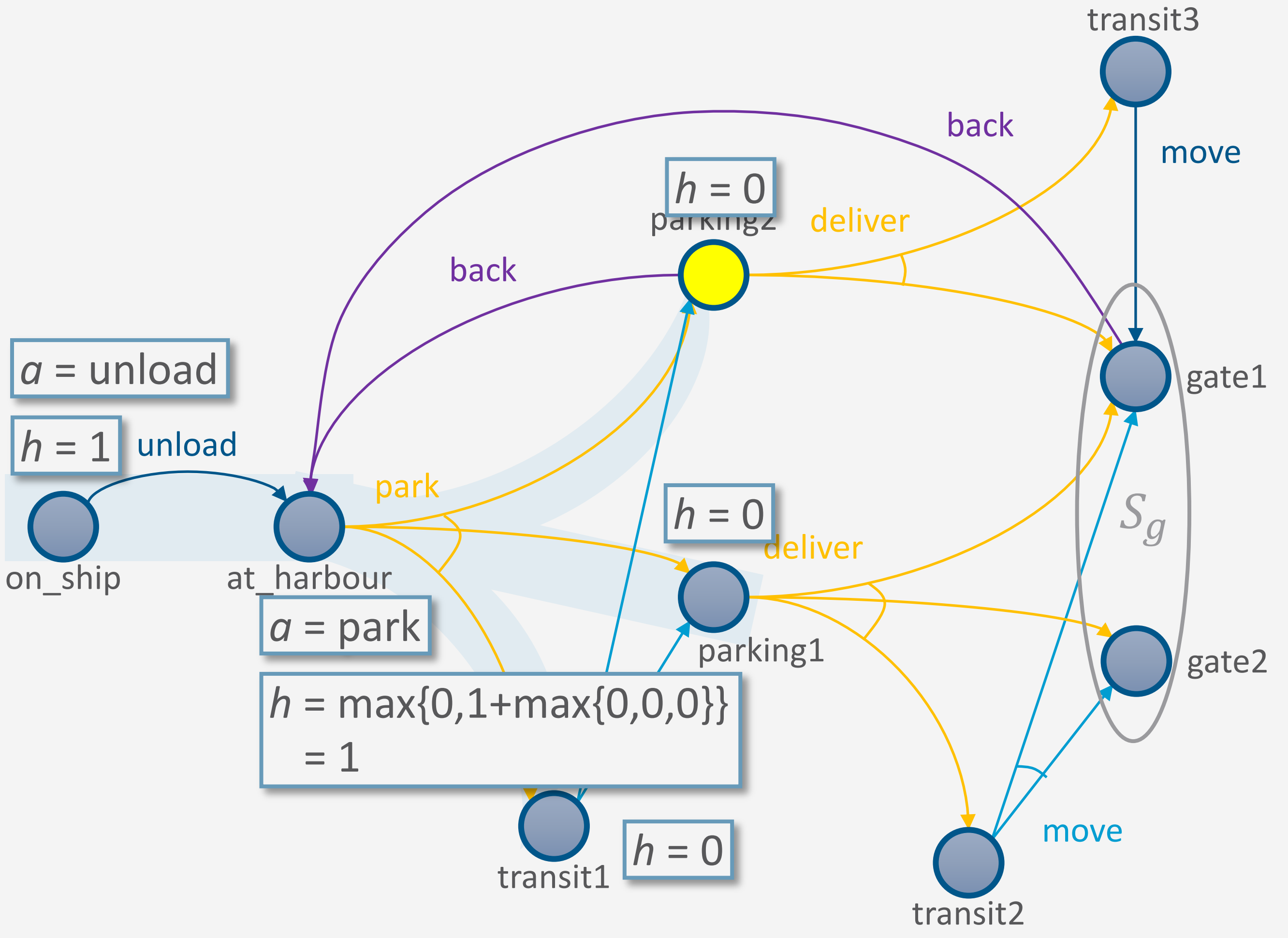
Suppose that initially,
$h(s) = 0$ for every state s



Example

```
Min-Max-LRTA*(Σ,s0,Sg)
    s ← s0
    while s ∉ Sg and Applicable(s) ≠ ∅ do
        a ← argmin_{a∈Applicable(s)} max_{s′∈γ(s,a)} h(s′)
        h(s) ← max{h(s),1 + max_{s′∈γ(s,a)} h(s′)}
        perform action a
        s ← the current state
```

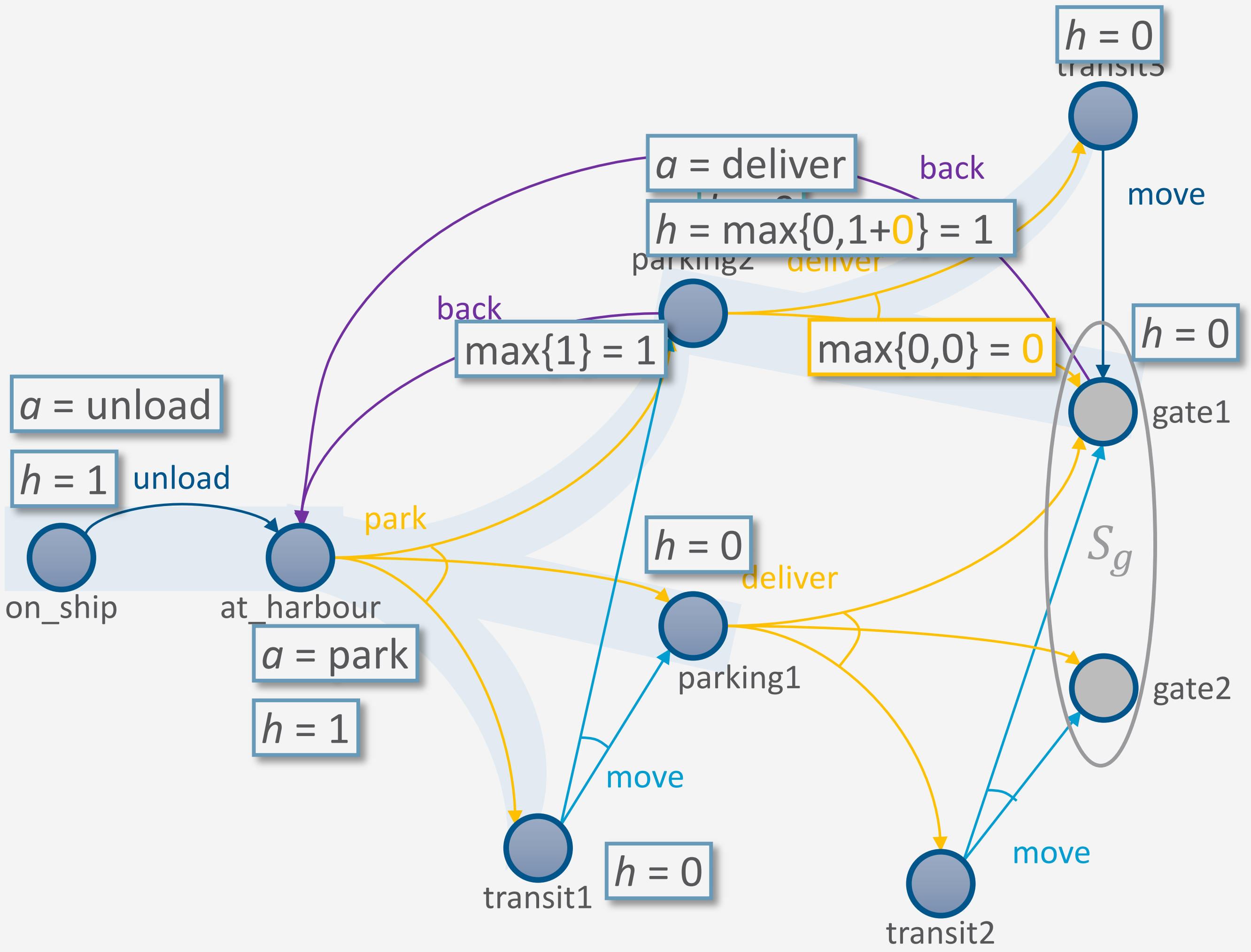
Suppose that initially, $h(s) = 0$ for every state s



Example

```
Min-Max-LRTA*(Σ, s0, Sg)
    s ← s0
    while s ∉ Sg and Applicable(s) ≠ ∅ do
        a ← argmin_{a∈Applicable(s)} max_{s'∈γ(s,a)} h(s')
        h(s) ← max{h(s), 1 + max_{s'∈γ(s,a)} h(s')}
        perform action a
        s ← the current state
```

Suppose that initially,
$h(s) = 0$ for every state s



Safely Explorable Domains

- Safely explorable domain
  - For every state s, at least one goal state is reachable from s
    - No dead ends

What about picking a random action?

- In a safely explorable domain,
  - Using Lookahead-Partial-Plan or FS-Replan
    - Lookahead never returns failure
    - Then we will eventually reach a goal
  - Using MinMax LRTA*
    - Algorithm is guaranteed to terminate and generate a solution

Intermediate Summary

- Online approaches
  - Lookahead-partial-plan
    - Adaptation of Run-Lazy-Lookahead
  - FS-replan
    - Adaptation of Run-Lookahead
- Ways to do the lookahead
  - Full breadth with limited depth: Iterative deepening
  - Full depth with limited breadth: Iterative broadening
- Min-Max-LRTA*
- Convergence in safely explorable domains

Can also adapt *Run-Concurrent-Lookahead*

Can put bounds on both depth and breadth

Outline per the Book

5.2 Planning Problem

- Planning domains
- Plans as policies
- Planning problems and solutions

5.3 And/Or Graph Search

- Planning by forward search

5.5 Determinisation Techniques

- Guided planning for safe solutions
- Planning for safe solutions by determinisation

5.6 Online Approaches

- Lookahead
- Lookahead by determinisation
- Lookahead with a bounded number of steps

$\Longrightarrow$ Next: Probabilistic Models