

07-380 AI & ML II

Classical Planning II
and Motion Planning



Instructors: Pat Virtue & Mohammad Salameh

Slide credits: CMU AI

Today

Solving Planning Problems:

- ✓ State Space Search
 - Planning Graphs and GraphPlan
 - Planning Heuristics
 - State Space Search (yes, back here)

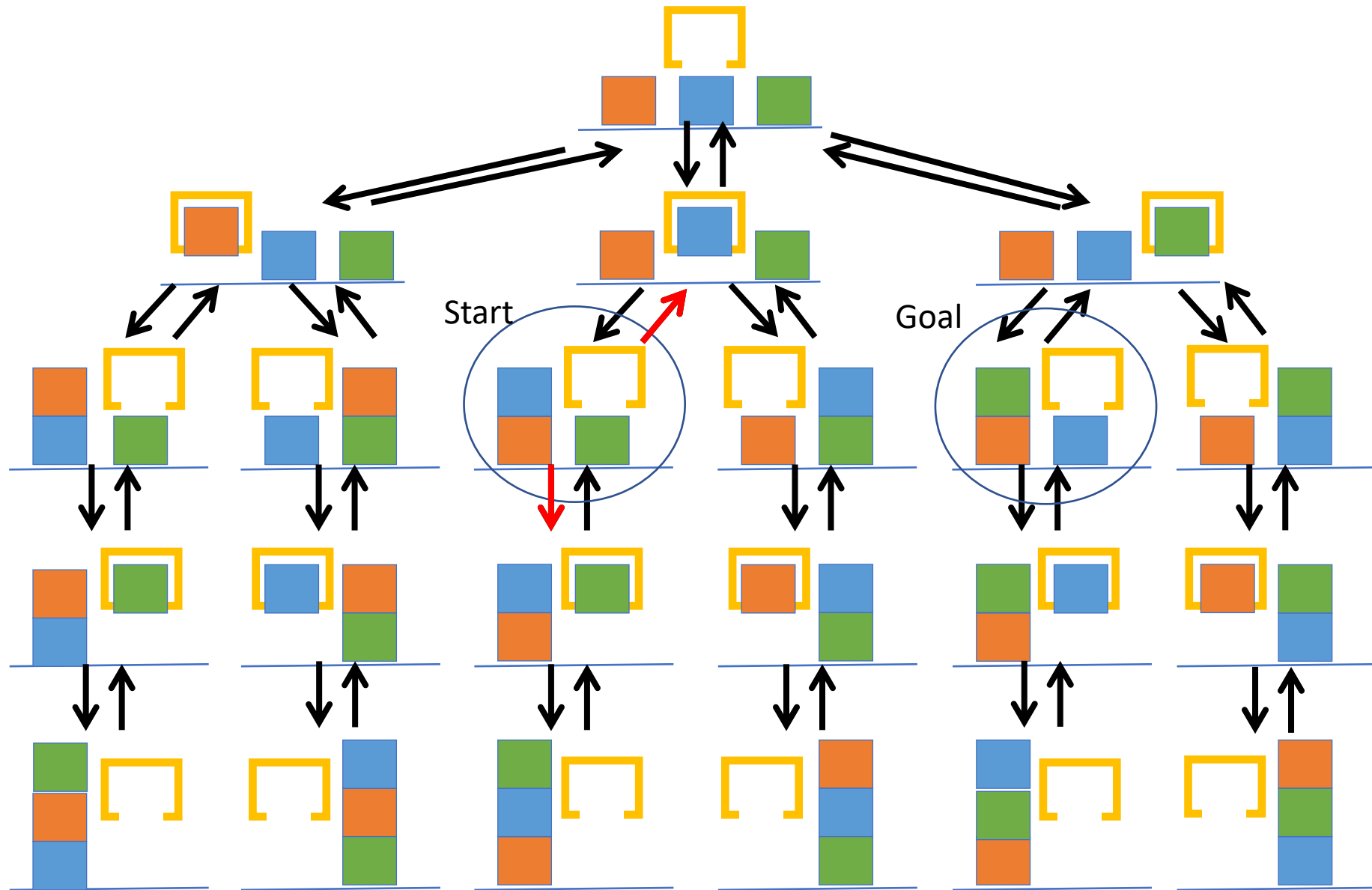
Planning in Continuous State Space

- Motion Planning and Configuration Space
- RRT (Rapidly-Exploring Random Trees)
- RRT*

Solving Planning Problems

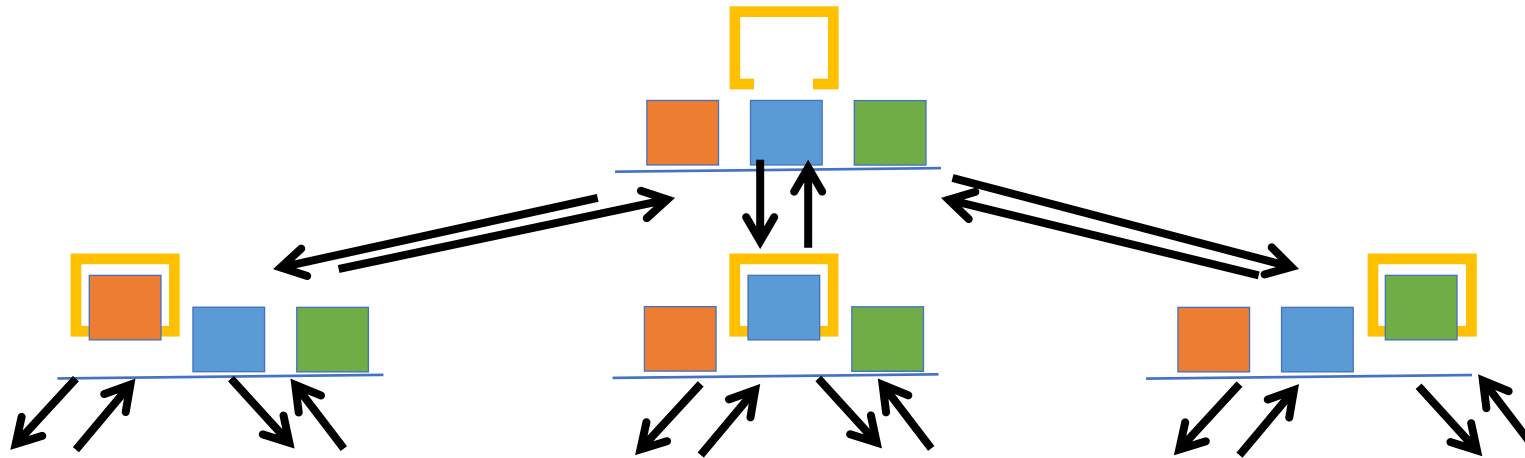
State-space Search

Search with a State Space Graph



Size of the State-Space Tree

A state-space tree's size is exponential in the number of predicates



Can we reduce the size of the graph we use for planning?

Solving Planning Problems

Planning Graphs & Graph Plan

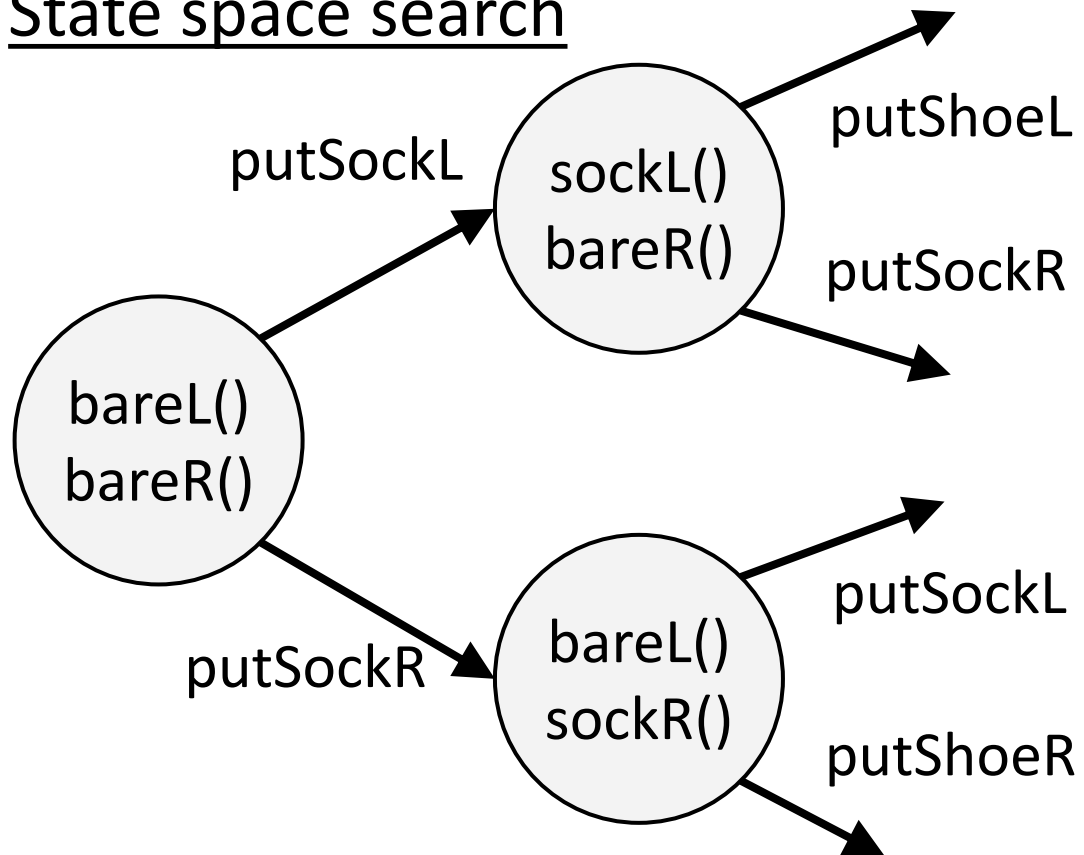
Planning Graph

- 1) Allow actions to be simultaneous
- 2) Always add predicates (don't delete)

GraphPlan is a **relaxation** of other classical planning search techniques

The GraphPlan search graph space is linear in the number of predicates

State space search



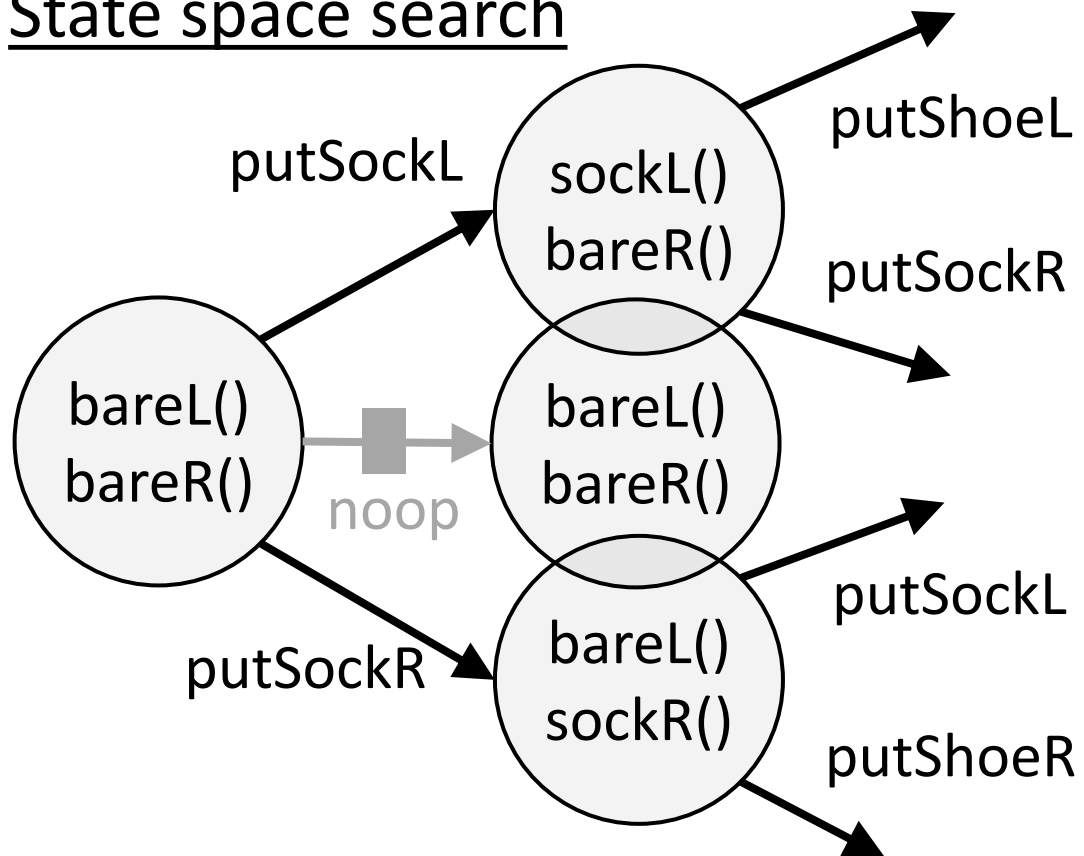
GraphPlan

- 1) Allow actions to be simultaneous
- 2) Always add predicates (don't delete)

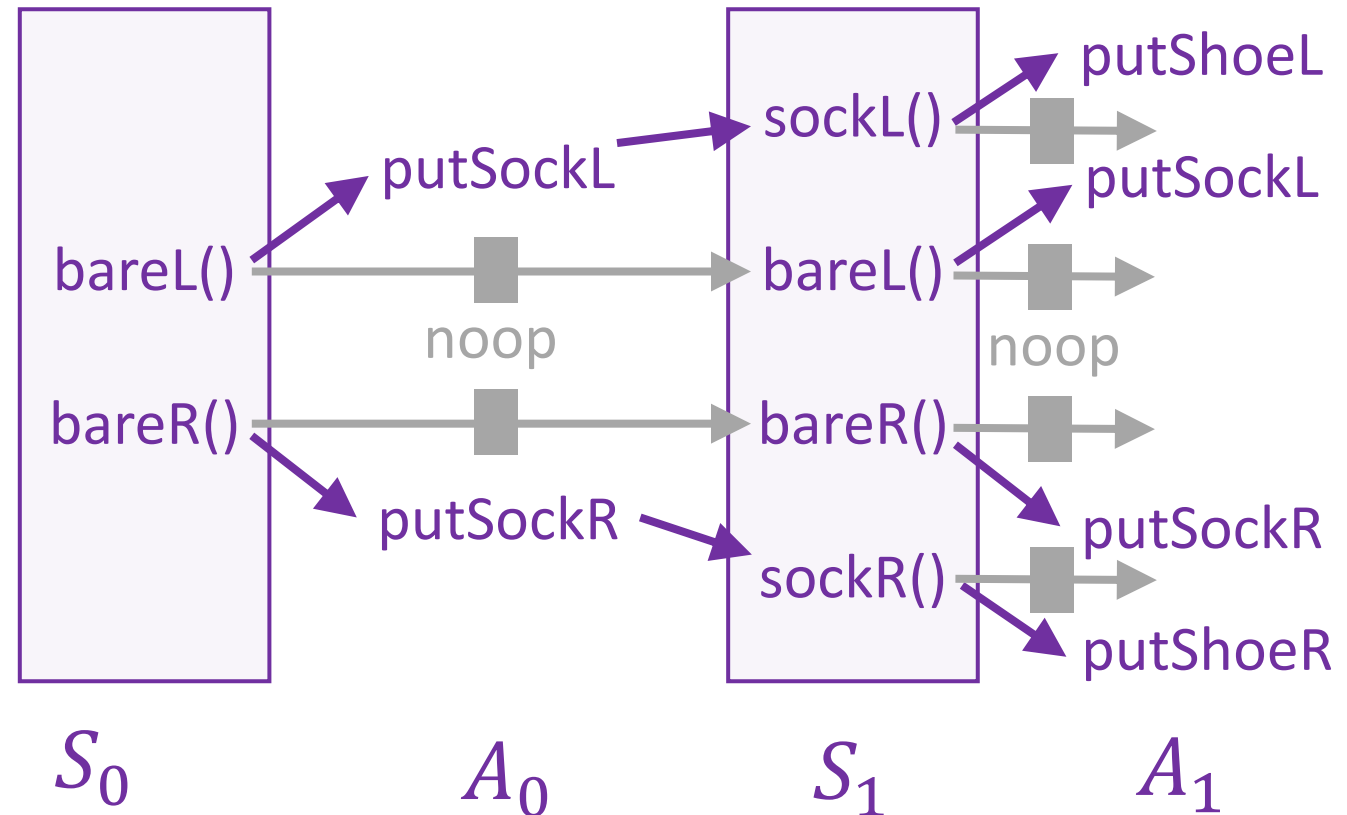
GraphPlan is a **relaxation** of other classical planning search techniques

The **planning graph** is linear in the number of predicates

State space search



Planning graph



Building a Planning Graph

Initialize S_0 with all predicates in the start state



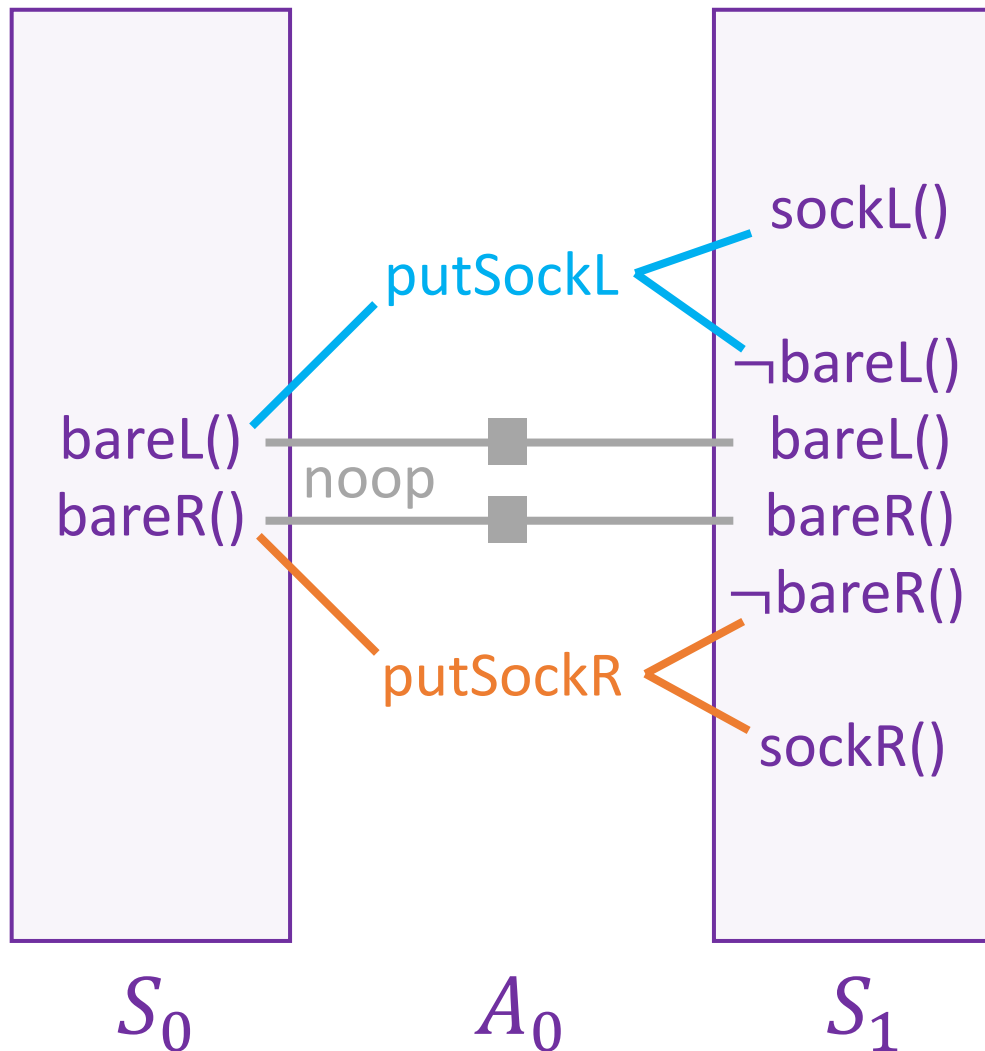
S_0

A_0

S_1

Building a Planning Graph

Extend graph to S_1 with all actions in A_0 that can be taken from S_0



For now, we are just assuming that any actions (including no-ops) can be taken individually in any order to get us to the next state

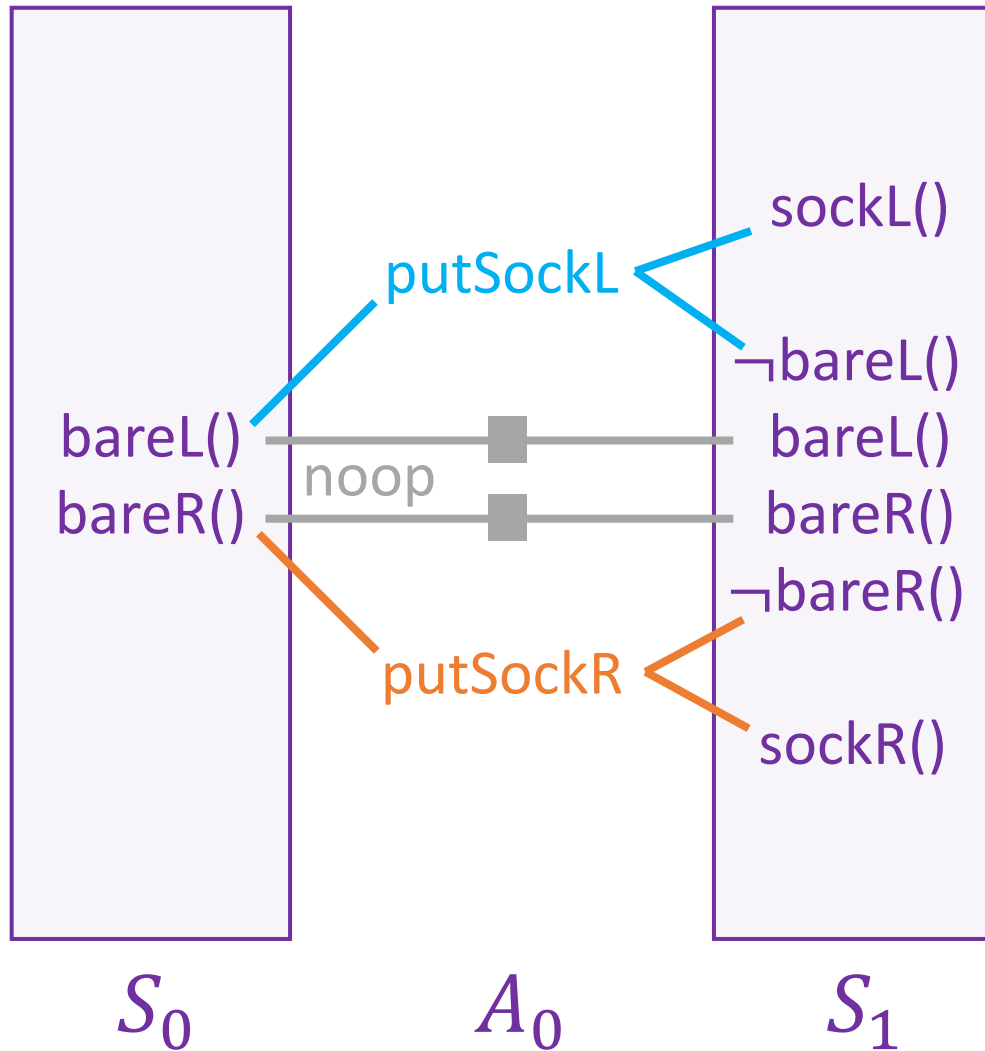
→ **This certainly isn't always true**

E.g., we can't put our socks on and then take the no-ops that require bare feet
(More on these exclusion checks later)

Building a Planning Graph

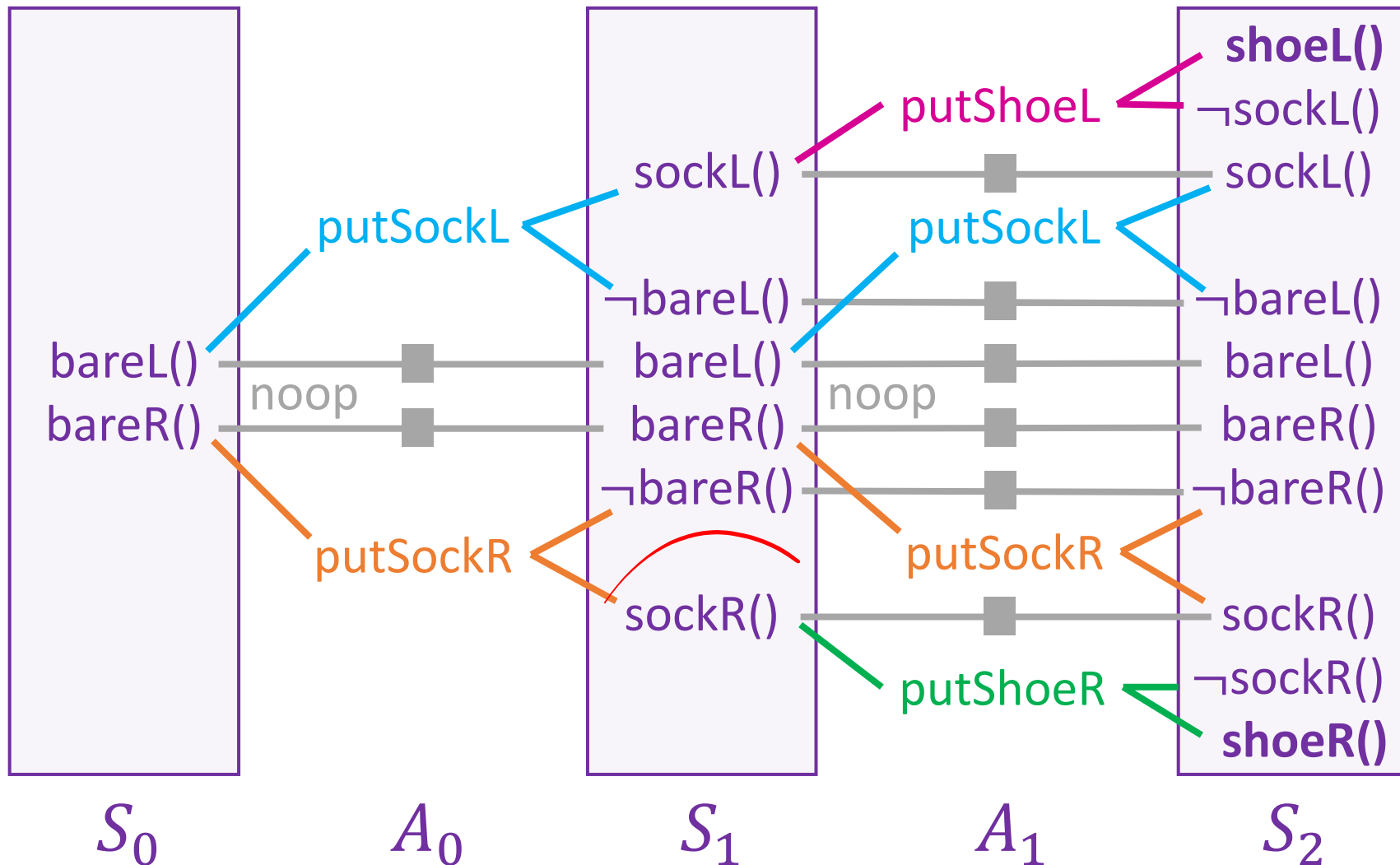
Search for solution. Does S_1 contain all goal predicates, shoeL(), shoeR()?

Nope, not yet



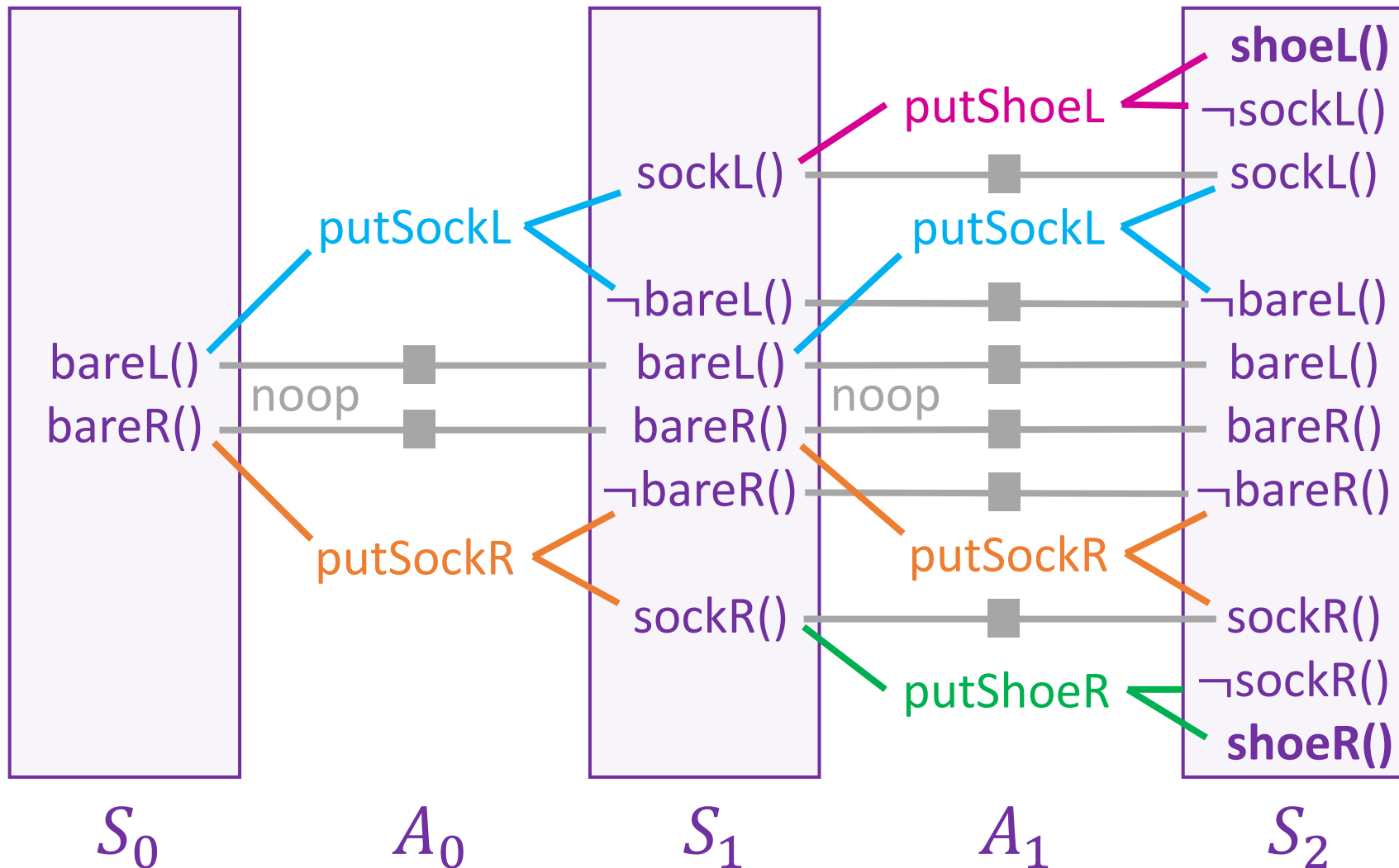
Building a Planning Graph

Extend graph to S_2 with all actions in A_1 that can be taken from S_1



Building a Planning Graph

Search for solution. Does S_2 contain all goal predicates, **shoeL()**, **shoeR()**?



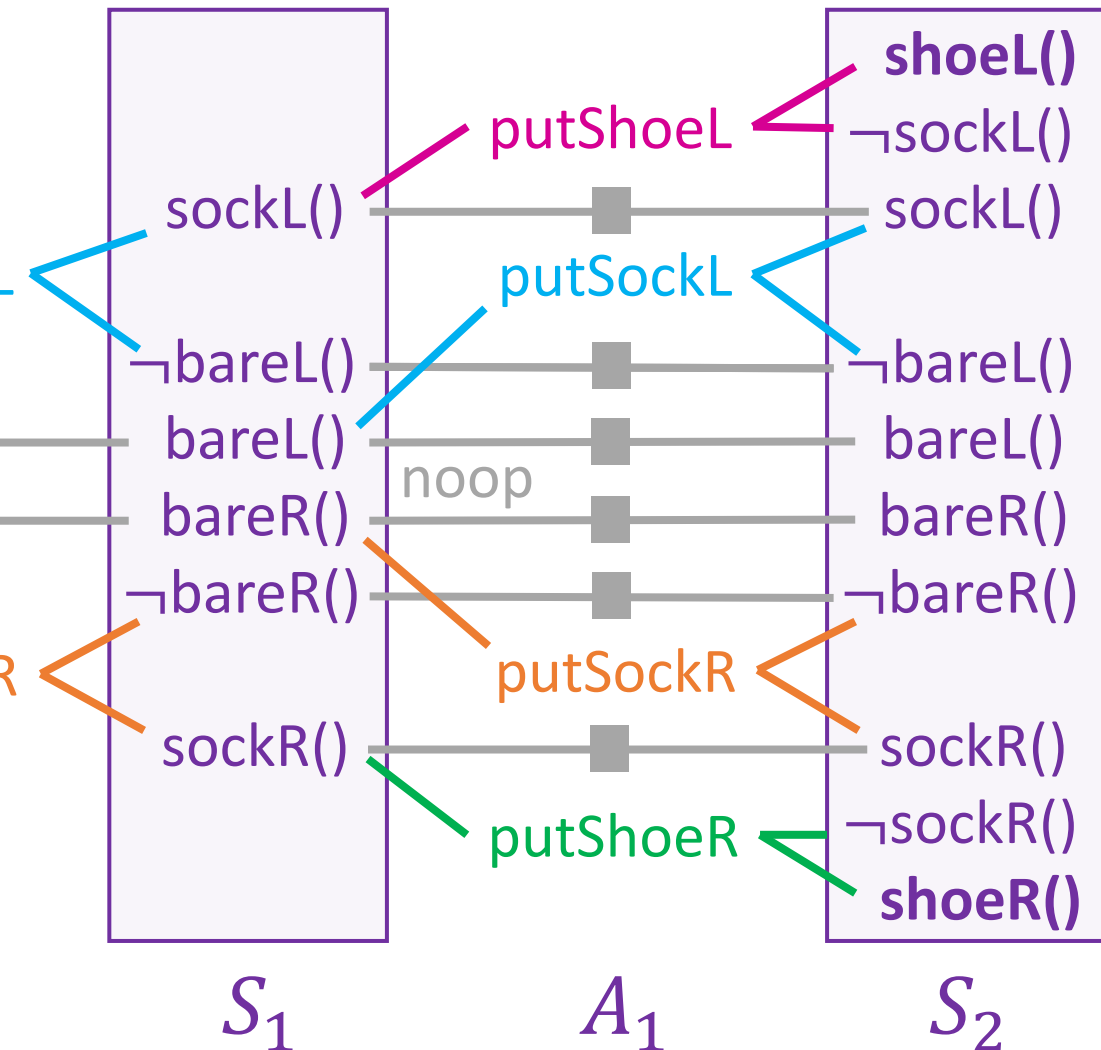
Yes, but
....is this ok??

Maybe ☺

We need to check
that we can
actually take the
subset of actions
that lead us the
goal predicates

Building a Planning Graph

Search for solution. Does S_2 contain all goal predicates, **shoeL()**, **shoeR()**?

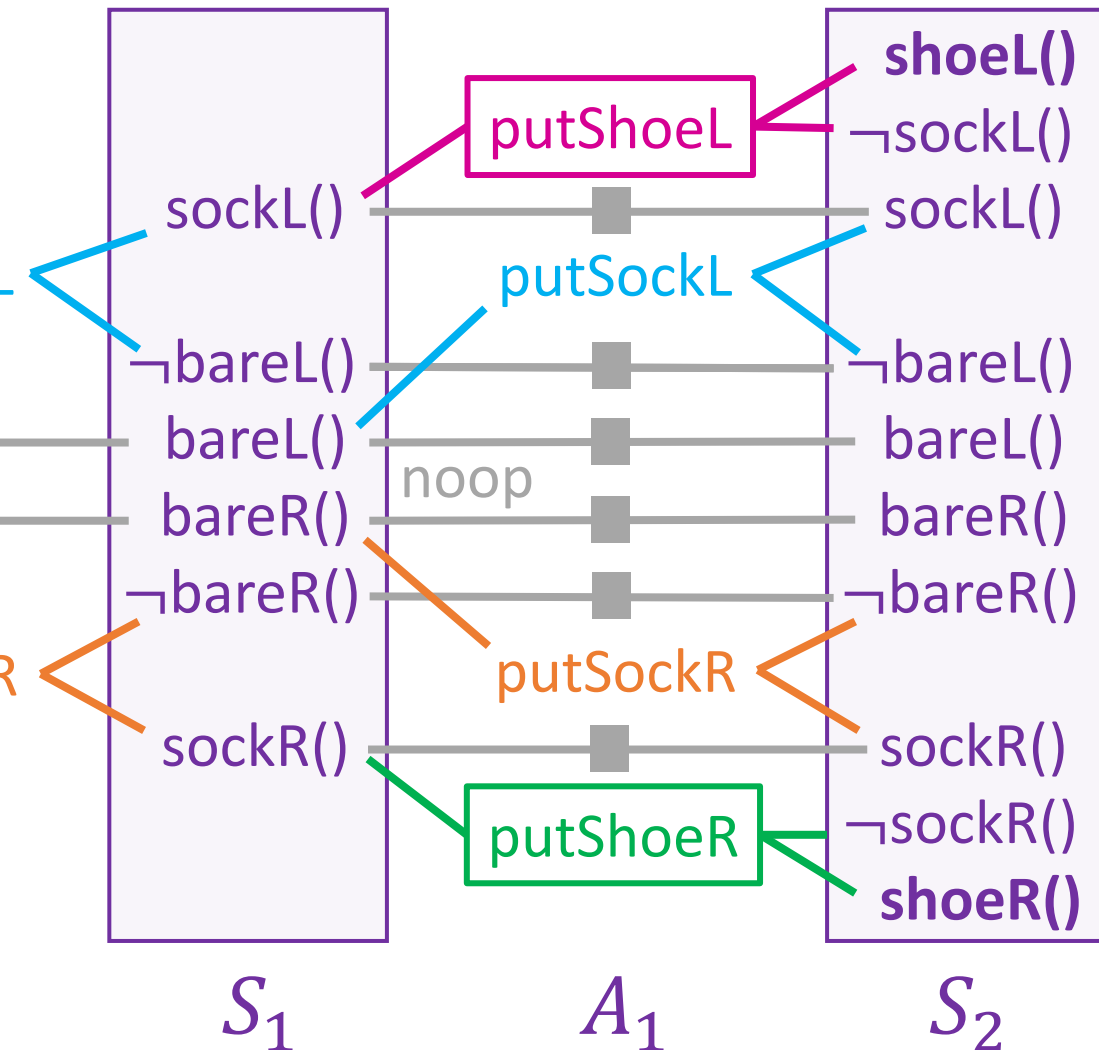


Are the goal predicates ok?

- Do they directly contradict each other (**negation**)?
- Are the actions that produced them ok (**consistent support**)?
 - We'll need to check **putShoeL** and **putShoeR**
 - Can we really do both of these actions in either order?

Building a Planning Graph

Search for solution. Does S_2 contain all goal predicates, **shoeL()**, **shoeR()**?



Are the actions leading to the goals okay (not **exclusive**)?

Actions A and B are **exclusive (mutex)** at action-level i , if:

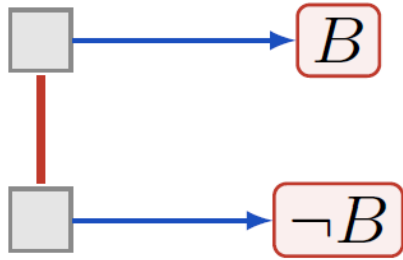
Interference: one action effect deletes or negates a **precondition** of the other

Inconsistency: one action effect deletes or negates the **effect** of the other

Competing Needs: the actions have preconditions that are mutex in prev. predicate-level

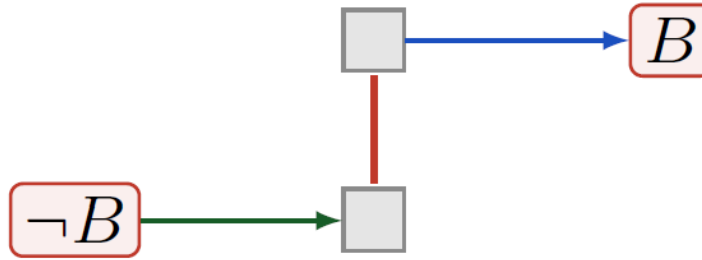
Planning Graph Mutexes

Inconsistency: one action effect deletes or negates the **effect** of the other



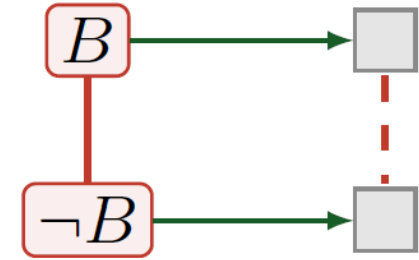
inconsistent effects

Interference: one action effect deletes or negates a **precondition** of the other



interference

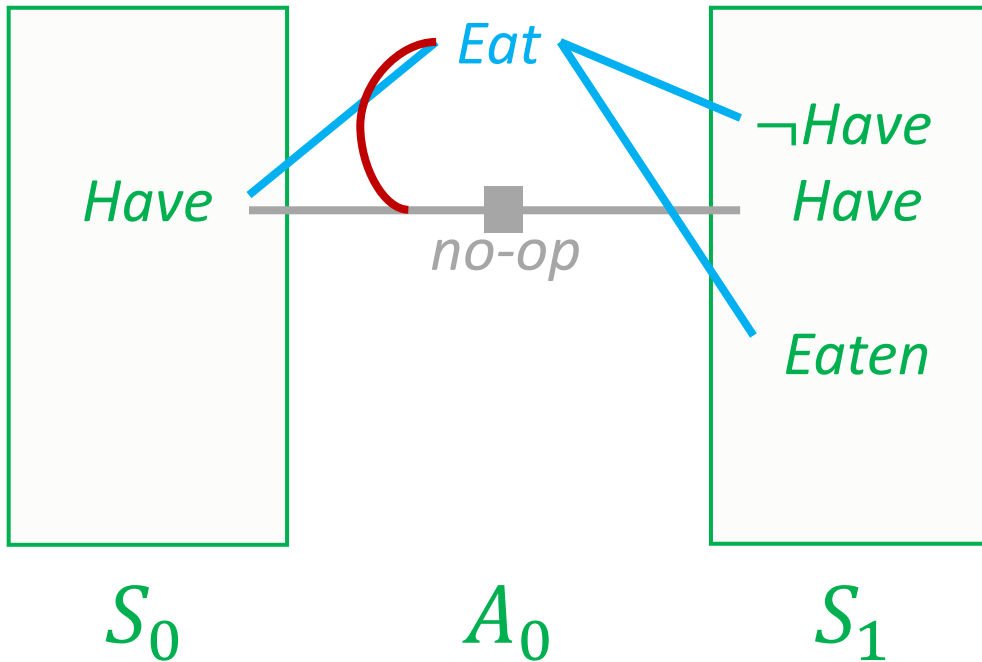
Competing Needs: the actions have preconditions that are mutex in prev. predicate-level



competing needs

Planning Graph Mutexes

Can you have your cake and eat it too?



In A_0 , *Eat* and *no-op*(*Have*)

have **inconsistent effects**

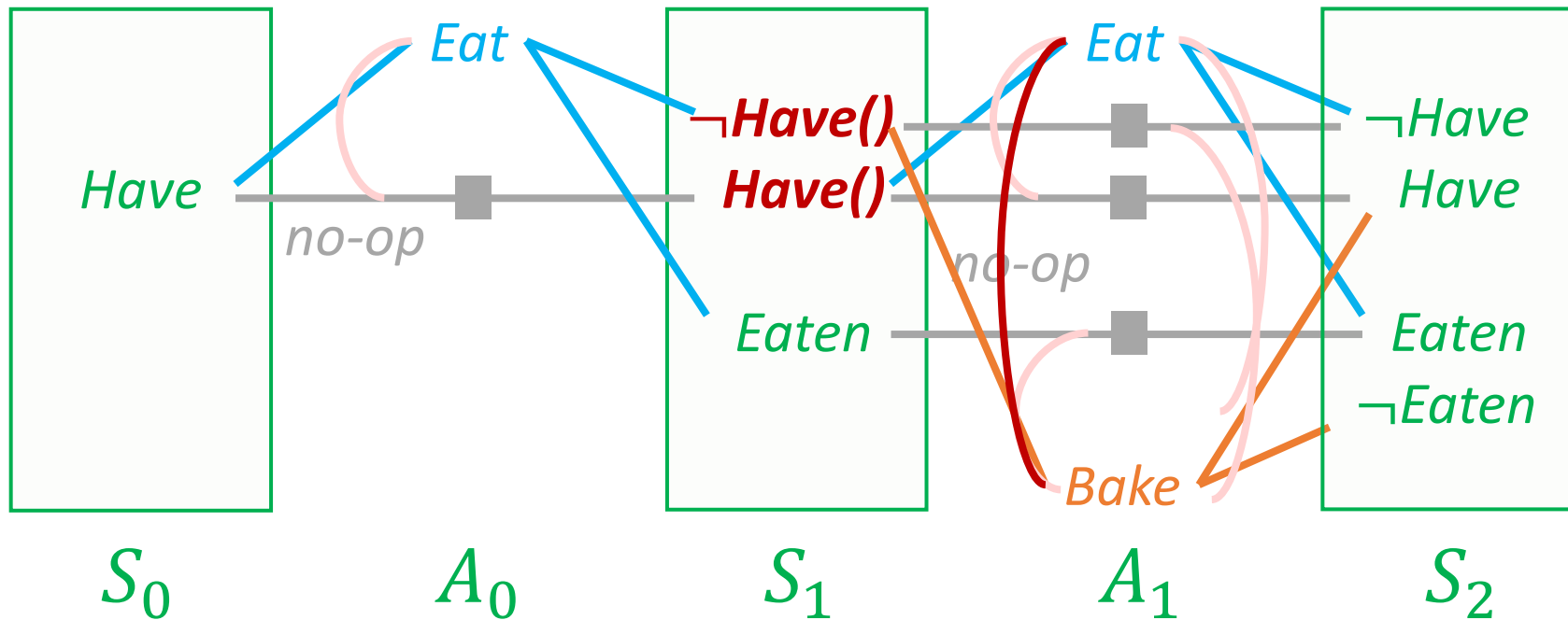
and are **inconsistent**

← *Eat* add $\neg Have$, the *no-op* adds *Have*

← *Eat* deletes the *no-op*'s precondition

Planning Graph Mutexes

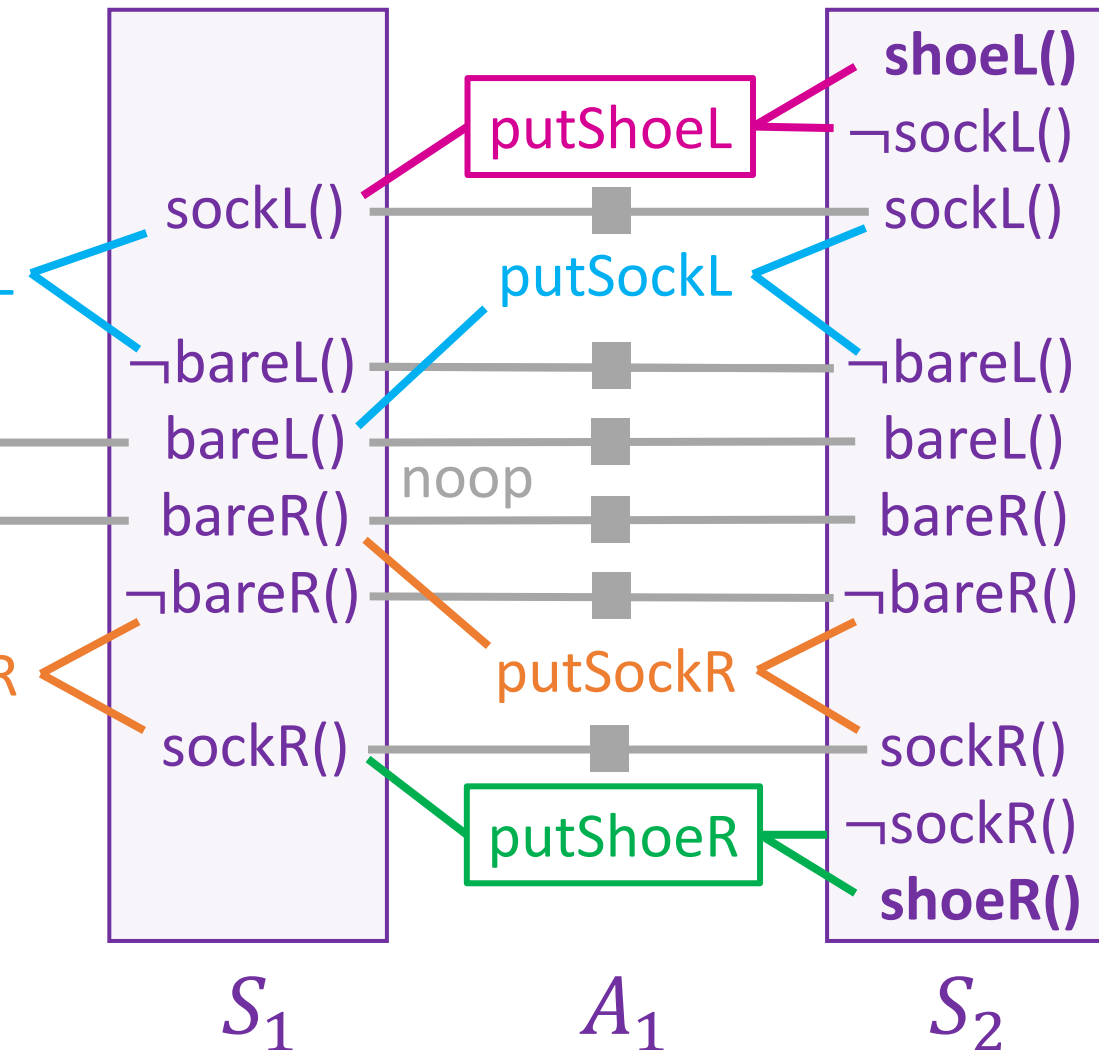
Can you have your cake and eat it too?



In A_1 , *Eat* and *Bake* have
competing needs $\leftarrow$ *Eat* needs *Have* and
Bake needs $\neg \text{Have}$

Building a Planning Graph

Search for solution. Does S_2 contain all goal predicates, **shoeL()**, **shoeR()**?



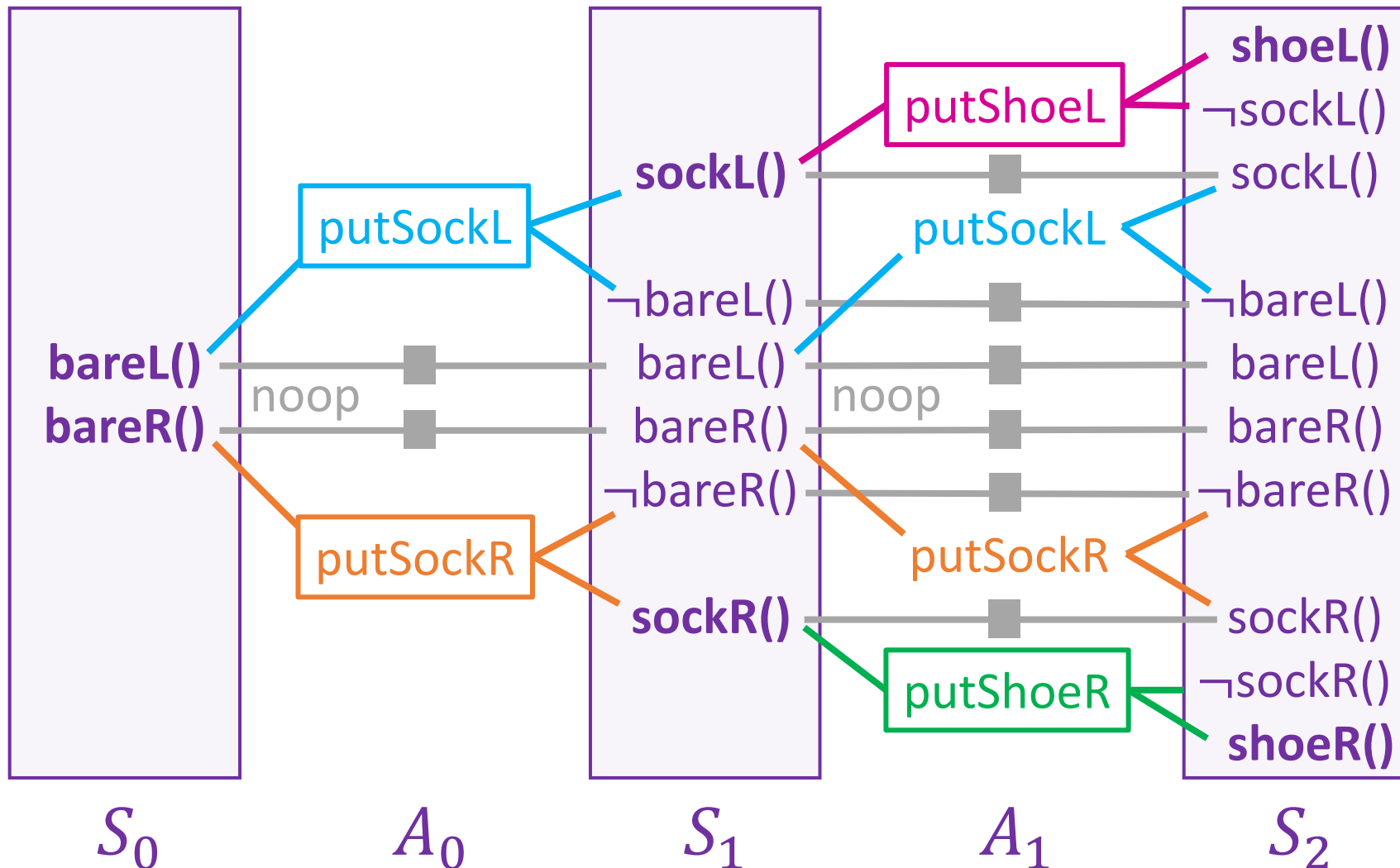
Are the actions leading to the goals okay (not **exclusive**)?

If yes, we need to check the S_1 pre-condition predicates of those actions: **sockL()** and **sockR()**

... and, then check the actions in A_0 that led us to that set of predicates...

Building a Planning Graph

Search backwards for solution of non-exclusive predicates and actions



Solution found!

We can do
 putSockL and
 putSockR
in any order, then

We can do
 putShoeL and
 putShoeR
in any order

GraphPlan High Level Algorithm

Initialize first predicate layer with predicates from **initial state**

Loop

Extend the GraphPlan graph by adding an action level and then a predicate level

If graph has **leveled off** (no new predicates added from previous level):
Return NO SOLUTION

If **all predicates in the goal are present** in the added predicate level:

Search for a possible plan in the planning graph
(see solution algorithm)

If plan found, return with that plan

GraphPlan Big Picture

Construct a planning graph as an approximation of the state space graph

- Num layers limited by num possible predicates and num possible actions
- Polynomial space and construction time

The approximation

We do not delete any predicates that were EVER true since the start of the search. The **planning graph** computes the **possibly reachable** states although they aren't necessarily **feasible**

The Bad News

GraphPlan backward search is exponential. We may have to try all the combinations of actions at each action layer

But... the planning graph is still **super useful** for more modern techniques

Today

Solving Planning Problems:

- ✓ State Space Search
- ✓ Planning Graphs and GraphPlan
 - Planning Heuristics
 - State Space Search (yes, back here)

Planning in Continuous State Space

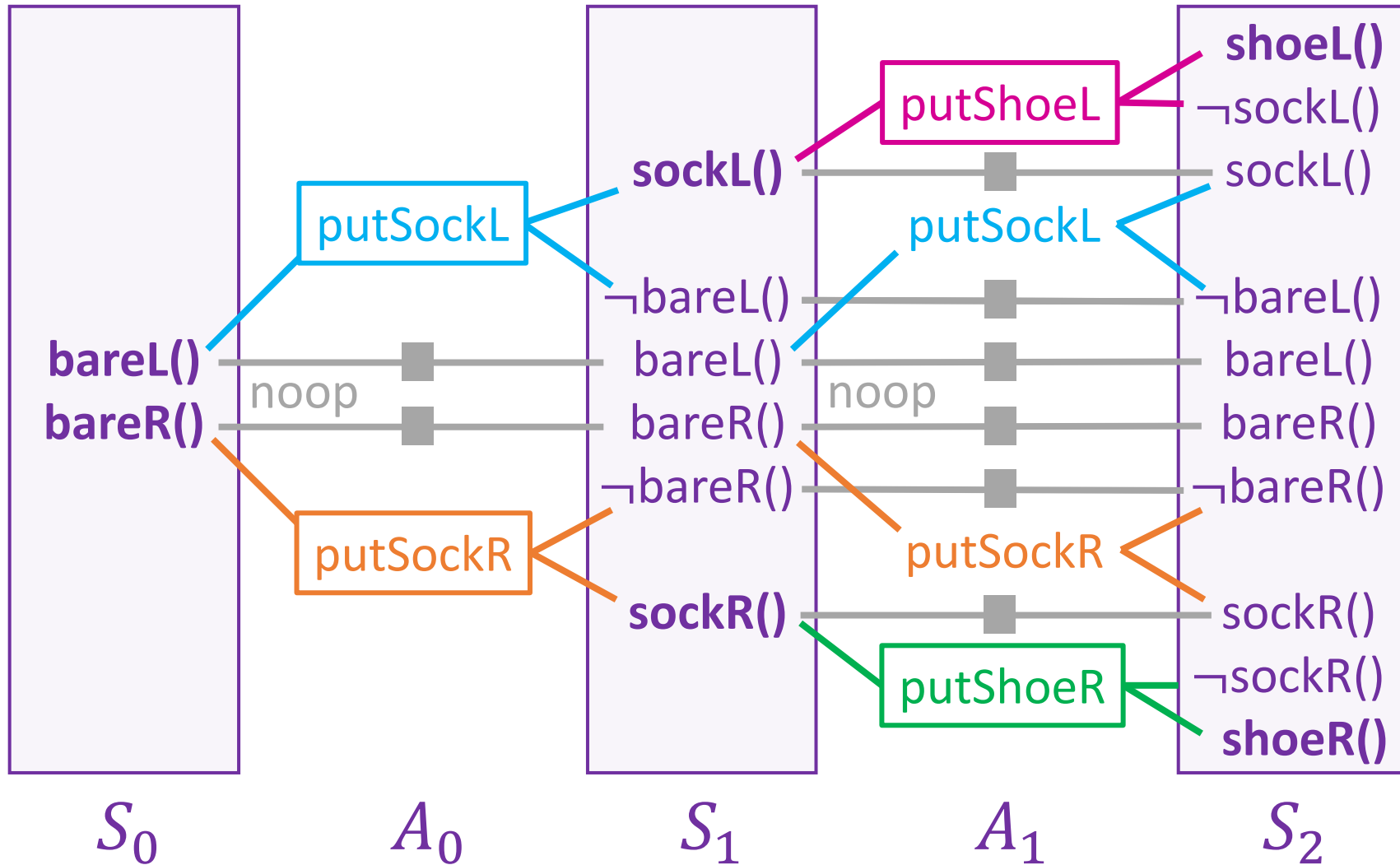
- Motion Planning and Configuration Space
- RRT (Rapidly-Exploring Random Trees)
- RRT*

Solving Planning Problems

Planning Heuristics

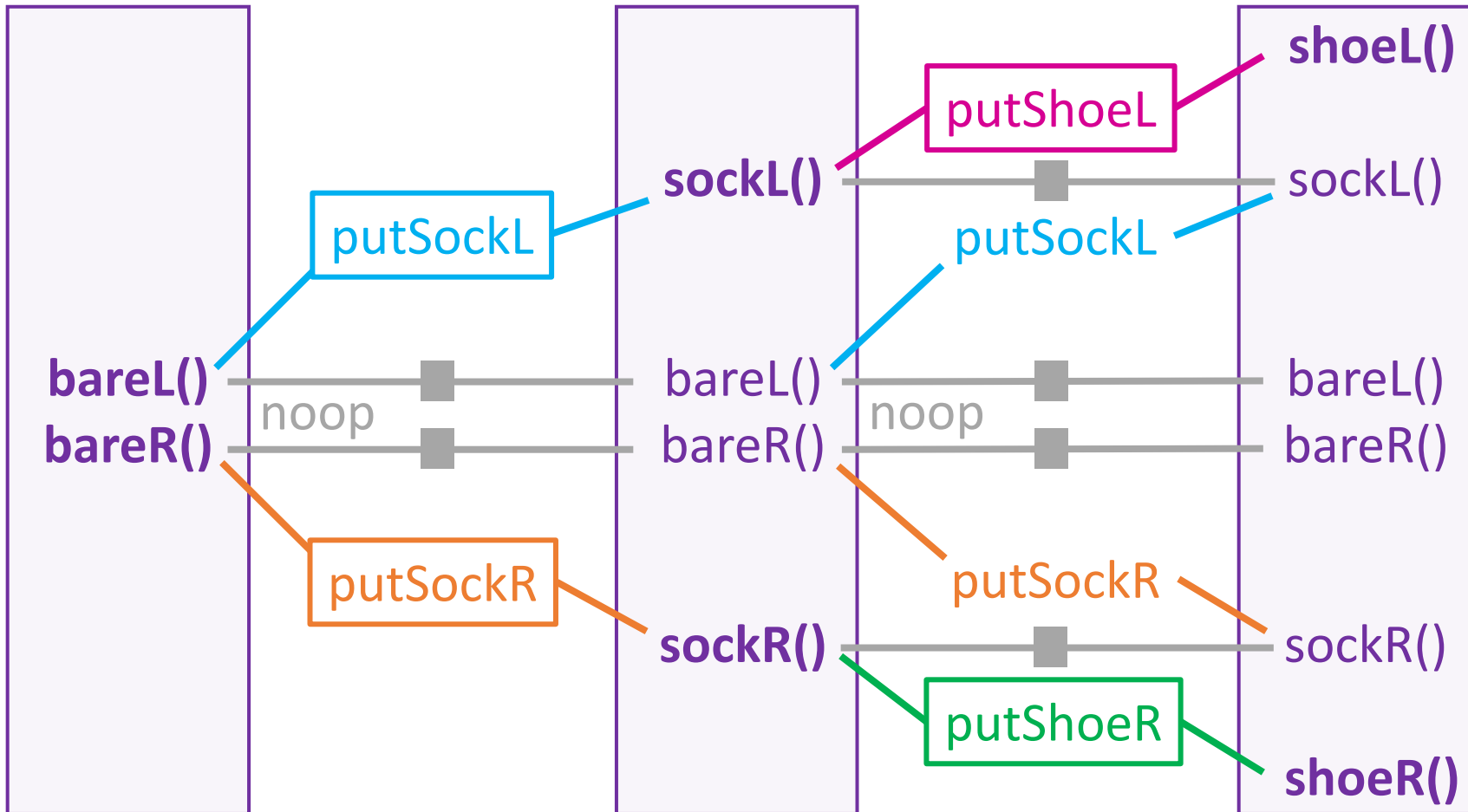
Delete-Relaxed Planning Graph

Key idea: Don't add the delete effects



(Delete-)Relaxed Planning Graph

Key idea: Don't add the delete effects (cheating even more than GraphPlan!)

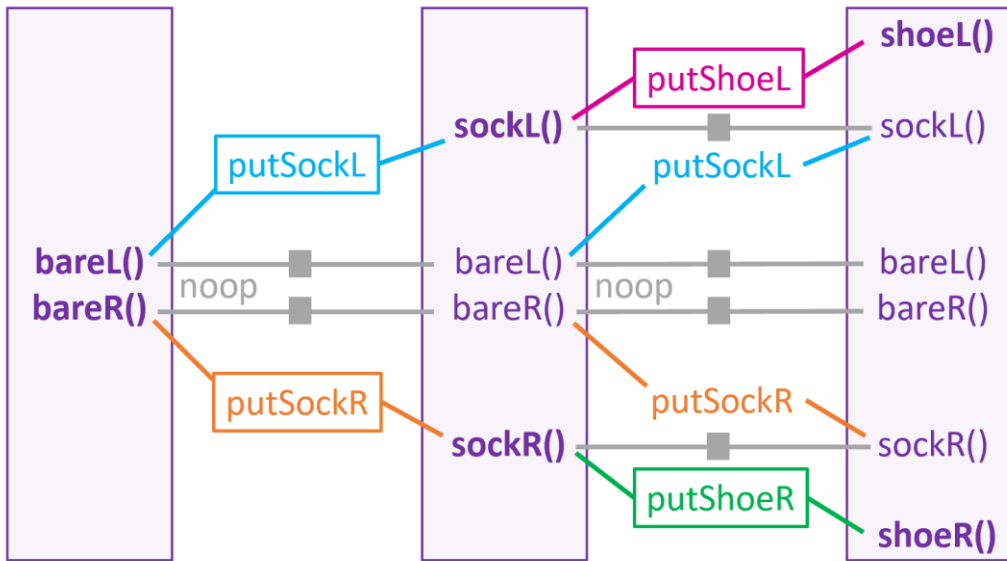


Resulting **mutexes** go away too!!

- Super fast to find a solution to this relaxed problem
- Not great for an actual plan, but
- Gives us a really good heuristic!

Planning Graph Heuristics

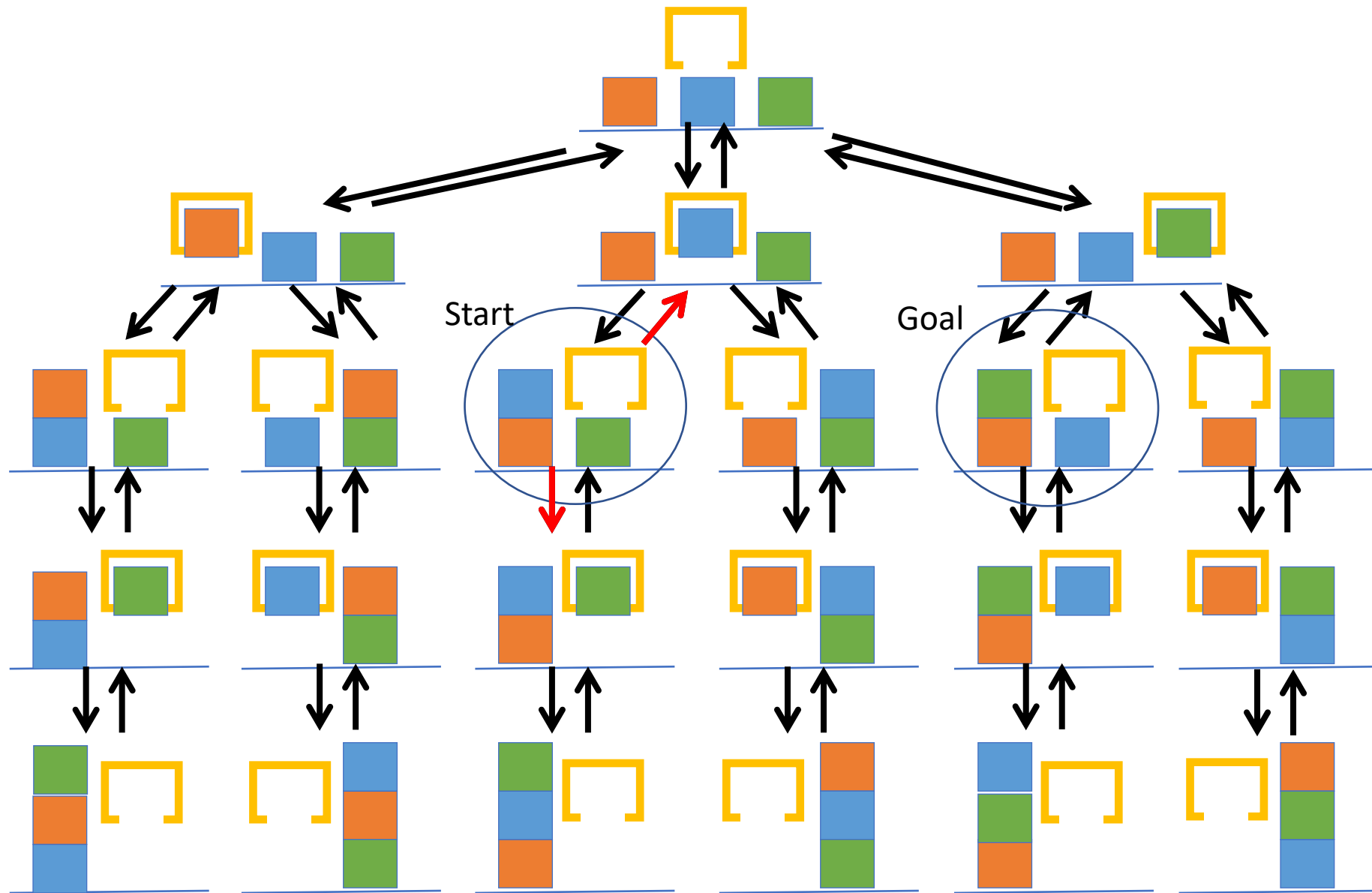
The Fast Forward planning technique used planning graph heuristics to speed up planning search in **state space graphs**!



Solving Planning Problems

Fast Downward

Back to Searching with a State Space Graph



Modern Planning Workhorse: Fast Downward

The Fast Downward planning technique uses planning graph heuristics to speed up as well as a few other tricks while searching in state space graphs!

Today

Solving Planning Problems:

- ✓ State Space Search
 - Planning Graphs and GraphPlan
 - Planning Heuristics
 - State Space Search (yes, back here)

Planning in Continuous State Space

- Motion Planning and Configuration Space
- RRT (Rapidly-Exploring Random Trees)
- RRT*

Planning in Continuous State Space

Motion Planning

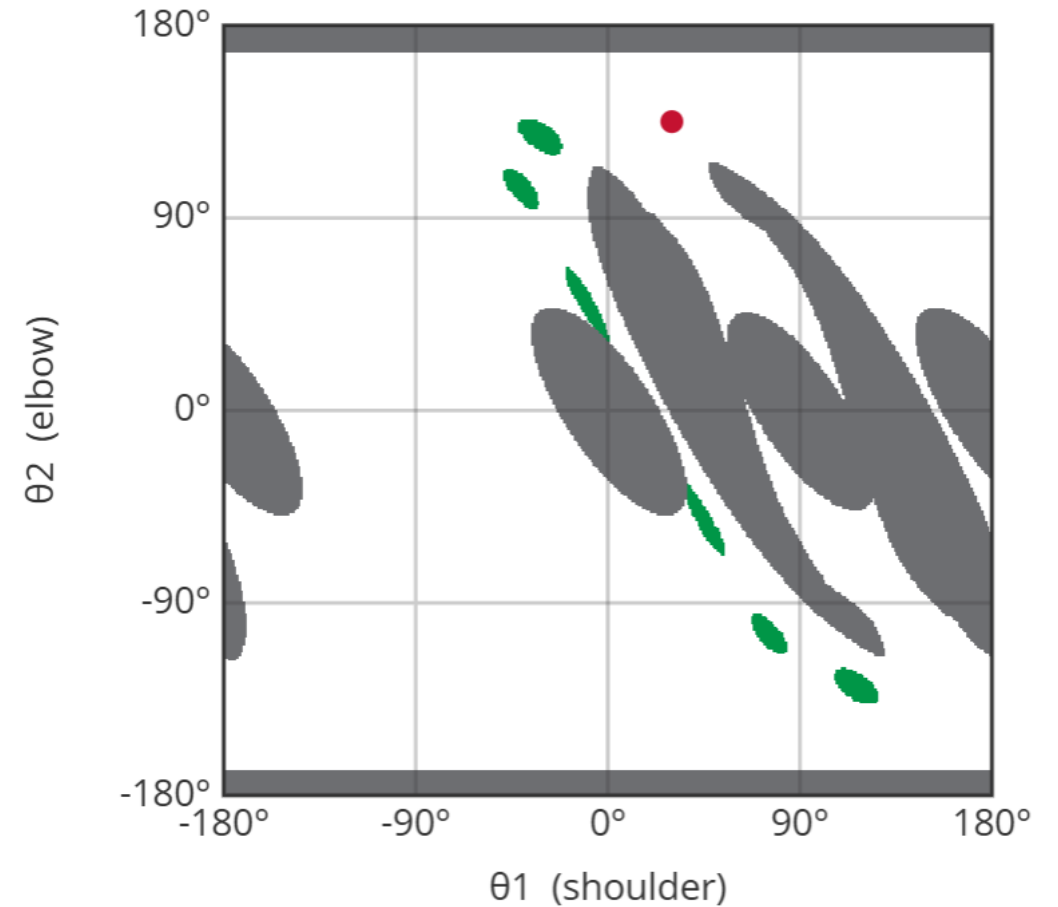
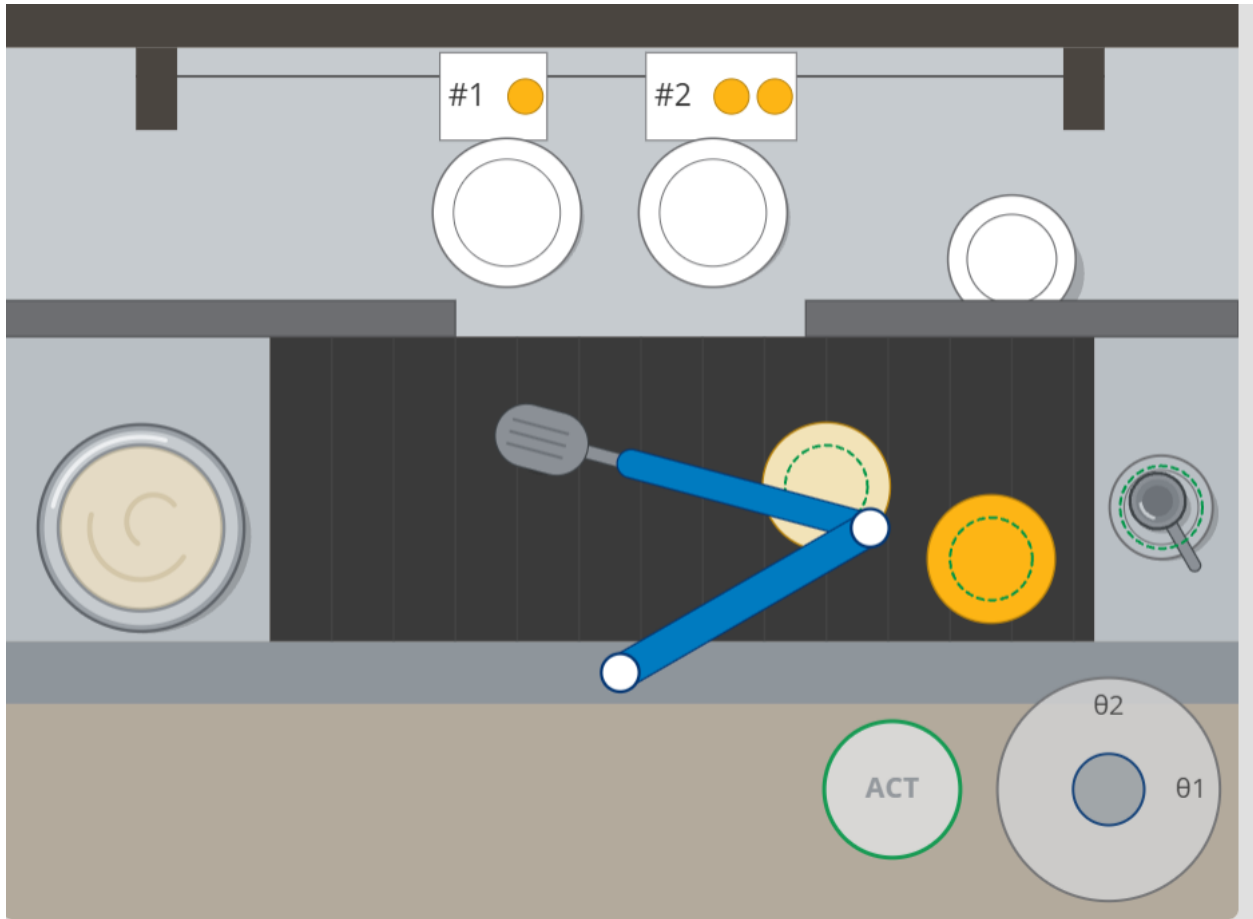
Among Us

How do we write an Among Us agent to move from one location to another?



Robot Cook Environment

Demo: Robot Cook

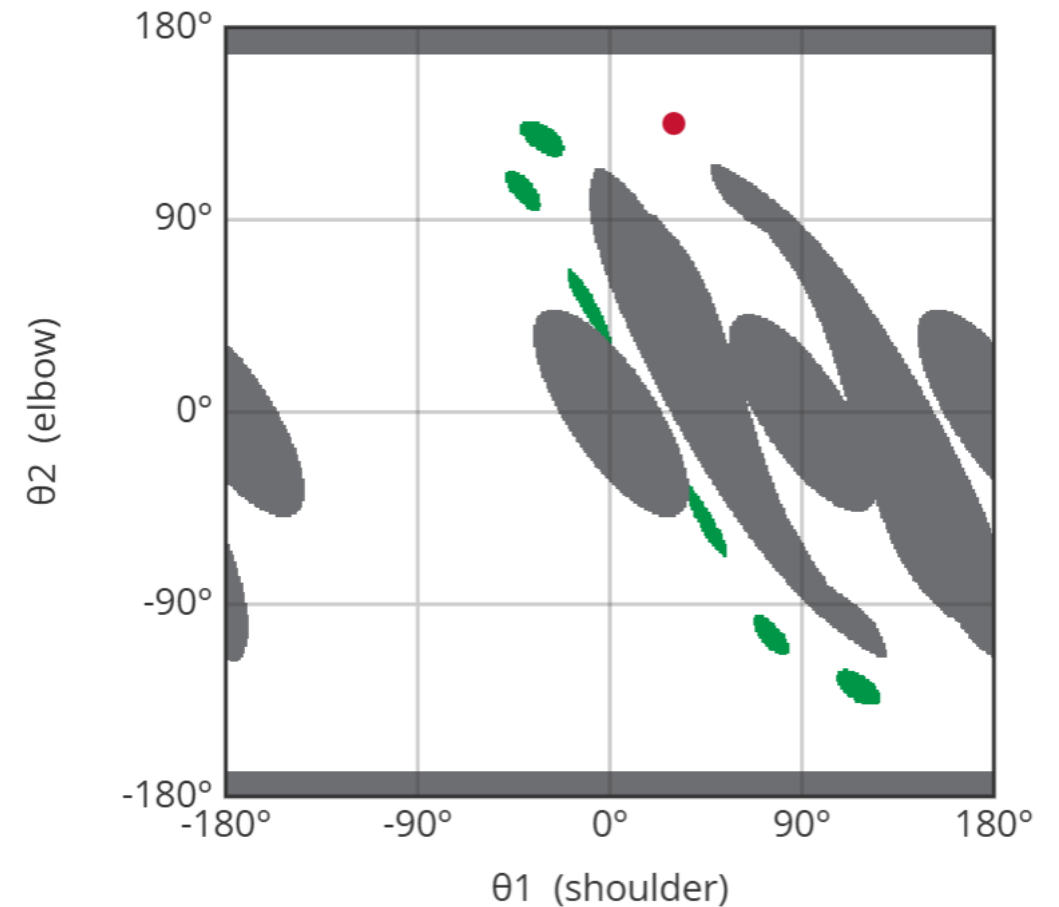
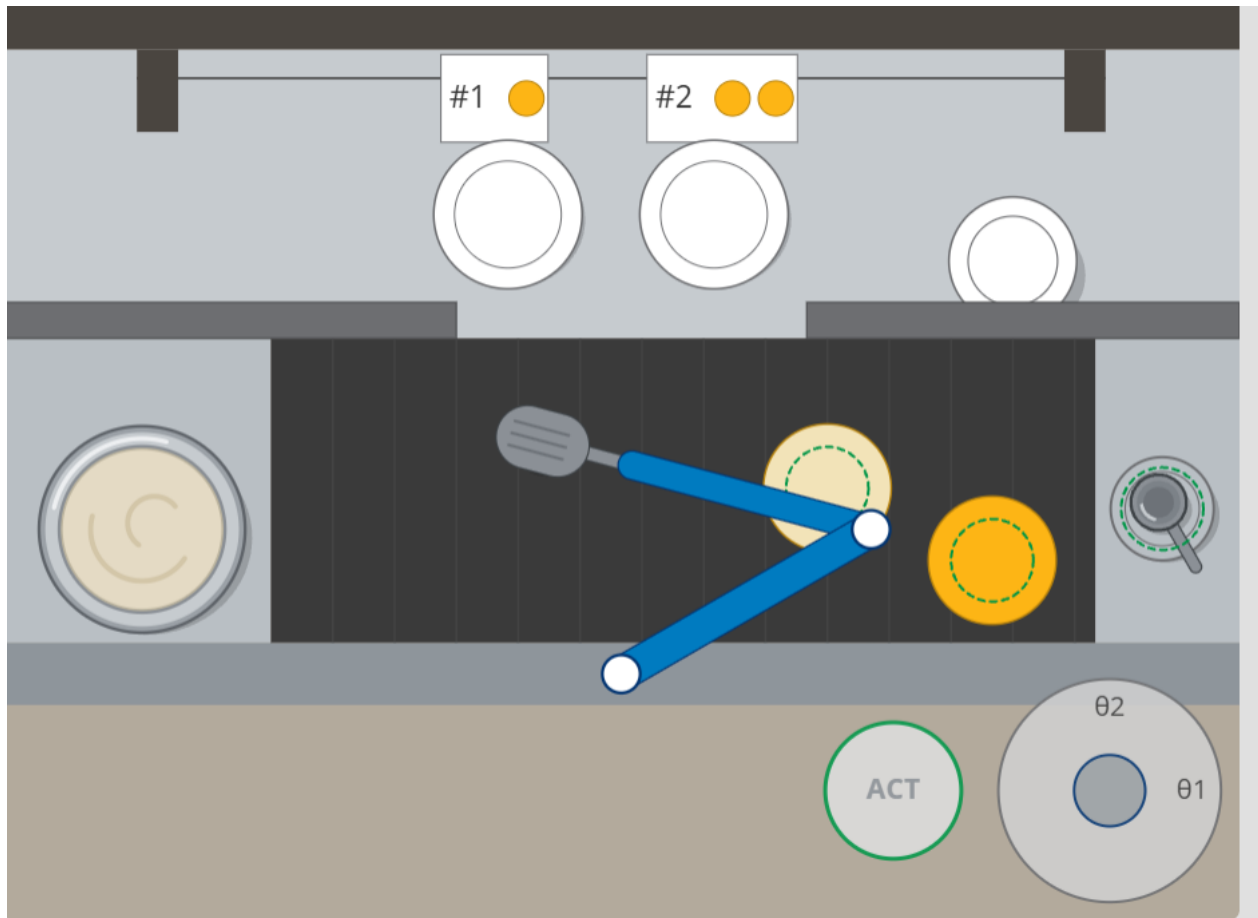


Demo: [Robot Cook](#)

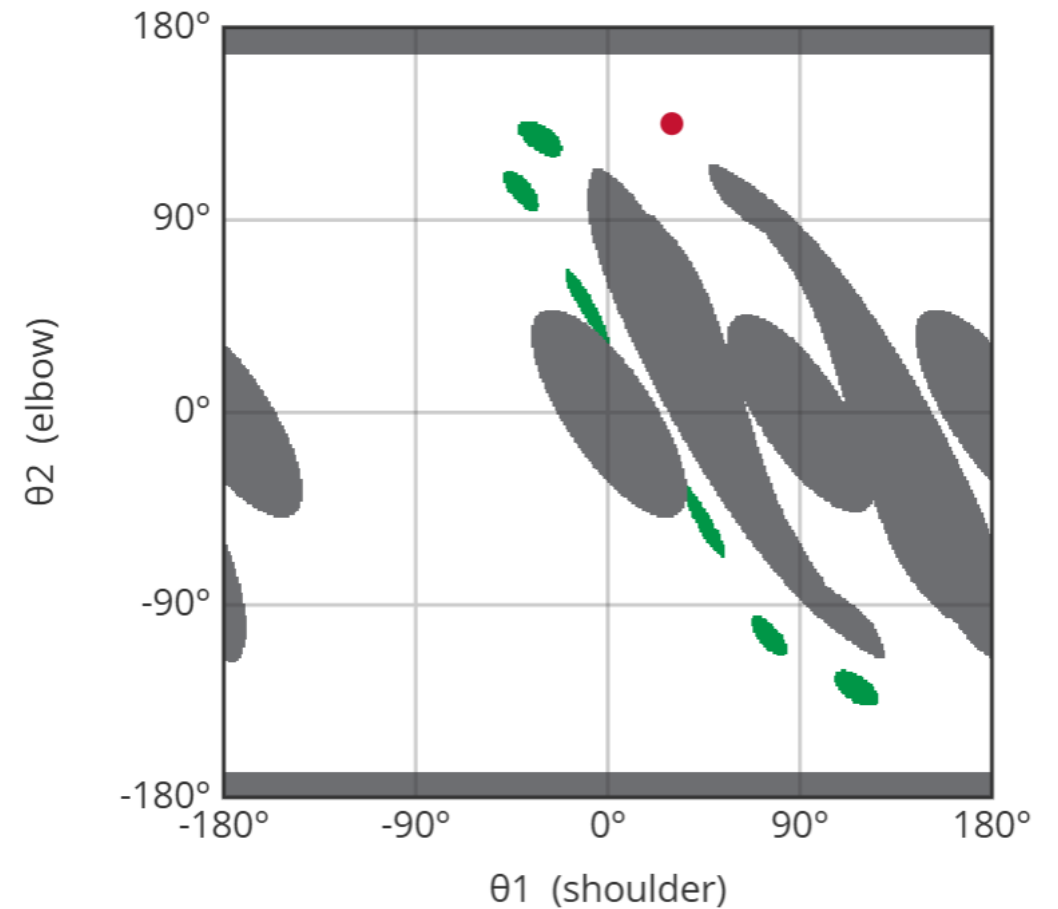
Poll 1 & 2

Poll 1: What is this particular gray thing?

Poll 2: What action is available at this particular green thing?



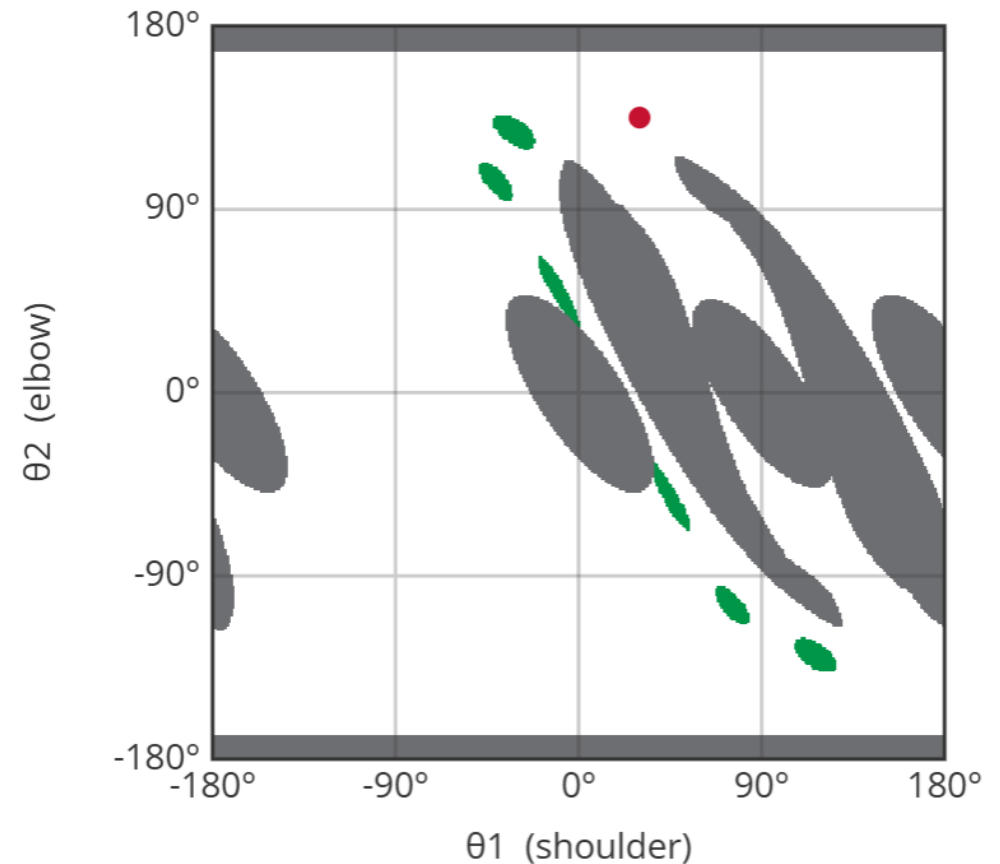
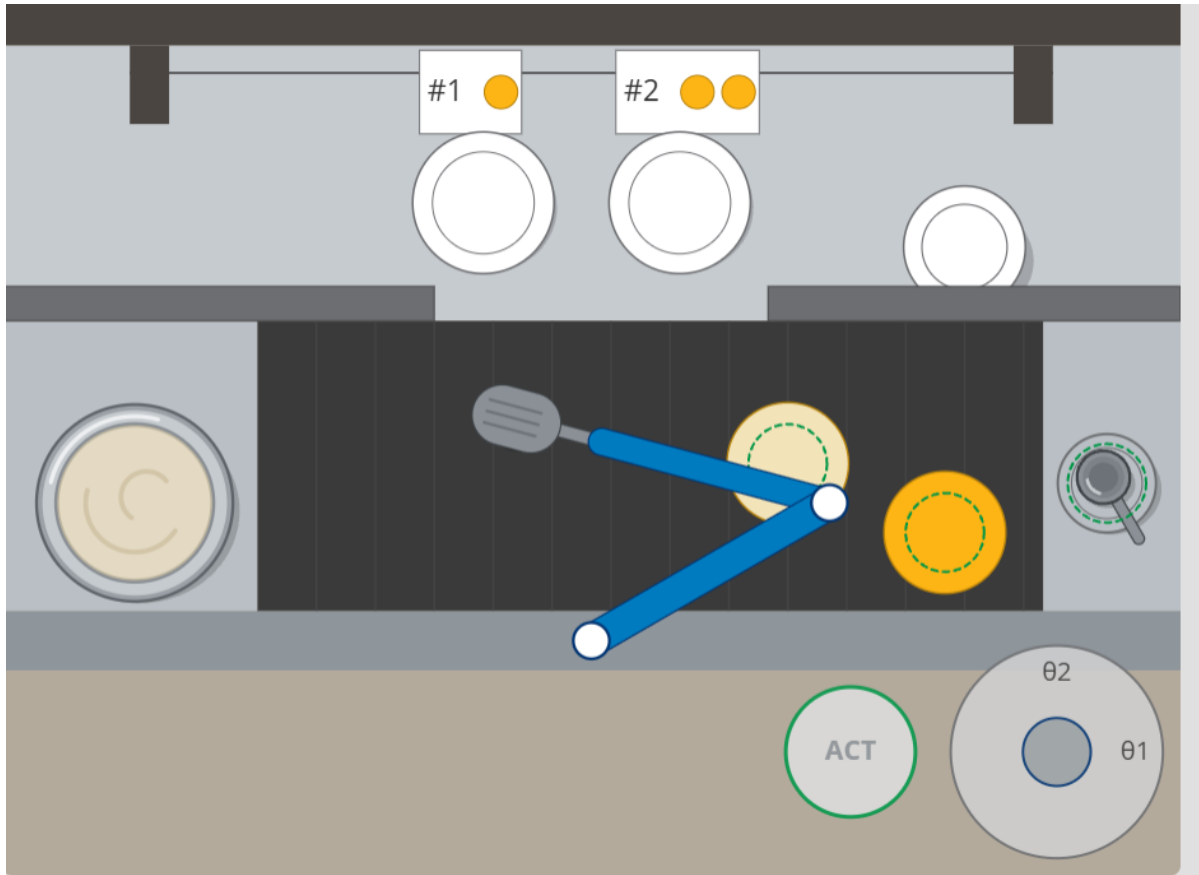
Question



Configuration Space

Physical space vs configuration space (C-space)

Physical state, s , vs a configuration, q



Demo: [Piano Mover](#)

Poll 3

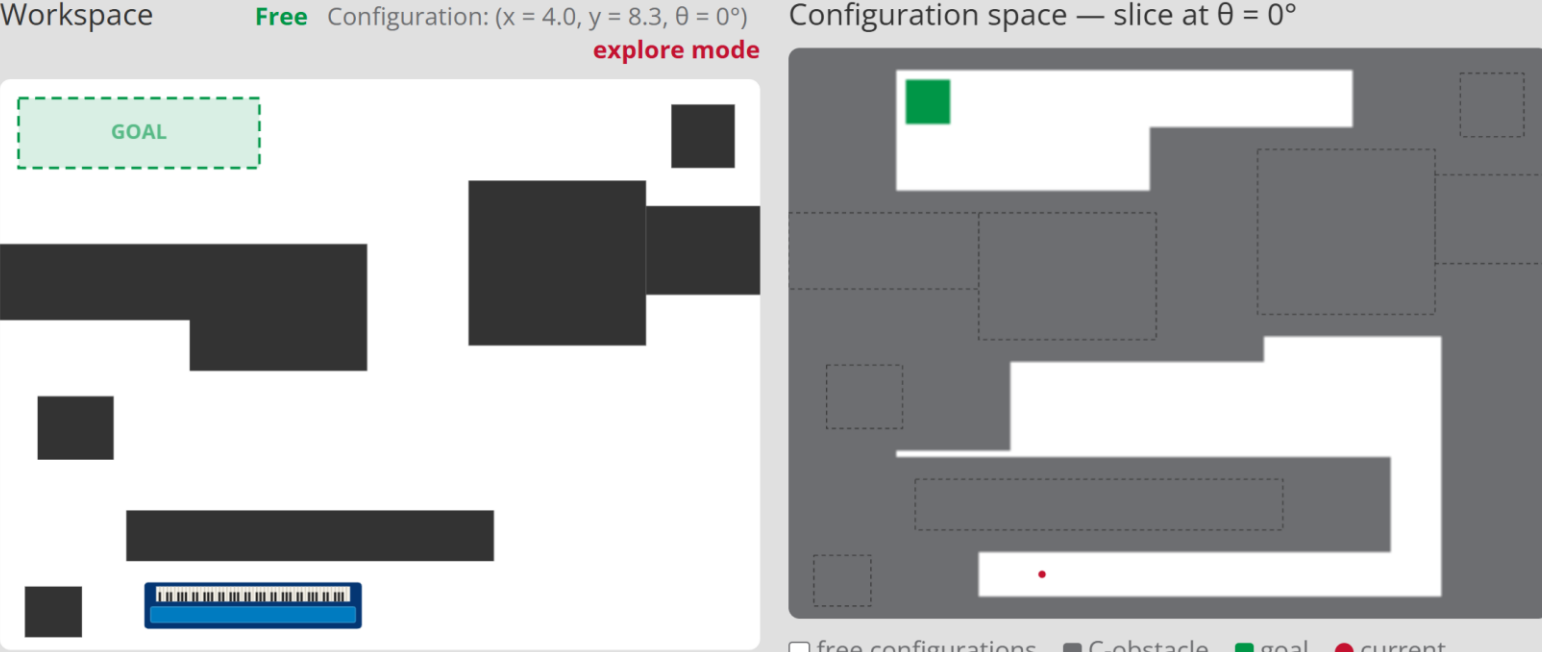
How many dimensions is the C-space in the Piano Mover environment

Piano Mover Configuration Space

Move a piano into the corner of a cluttered room and watch its configuration space update live.

Workspace **Free** Configuration: (x = 4.0, y = 8.3, $\theta = 0^\circ$) **explore mode**

Configuration space — slice at $\theta = 0^\circ$



Drag the piano; it stops at walls and furniture. Rotate with the θ slider, the mouse wheel, or **Q/E**. Arrow keys nudge it. Park it in the green goal spot.

θ :  0° ☒ Trace path **CLEAR TRACE** **RESET PIANO**

☐ free configurations ☐ C-obstacle ☒ goal ☒ current configuration — drag it!

Motion Planning

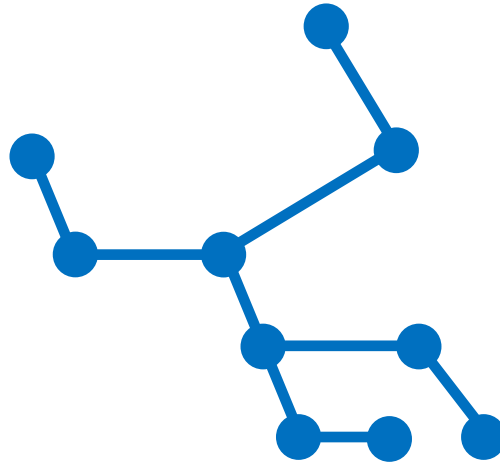
RRT

RRT (Rapidly-Exploring Random Trees)

🎵 *Catch a falling star, and
put it in your pocket;
never let it fade away 🎵*

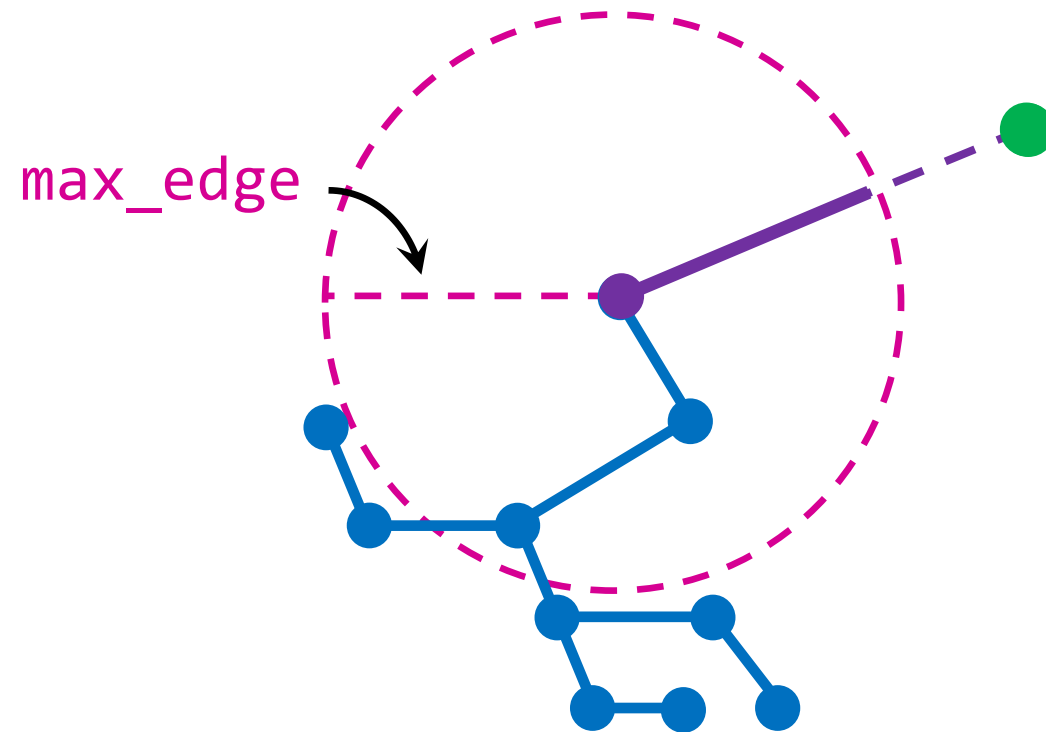
RRT (Rapidly-Exploring Random Trees)

🎵 *Pick a random point, and
find its nearest neighbor;
never let it get-too-far away* 🎵



RRT (Rapidly-Exploring Random Trees)

♪ Pick a *random point*, and
find its *nearest neighbor*;
never let it *get-too-far* away ♪

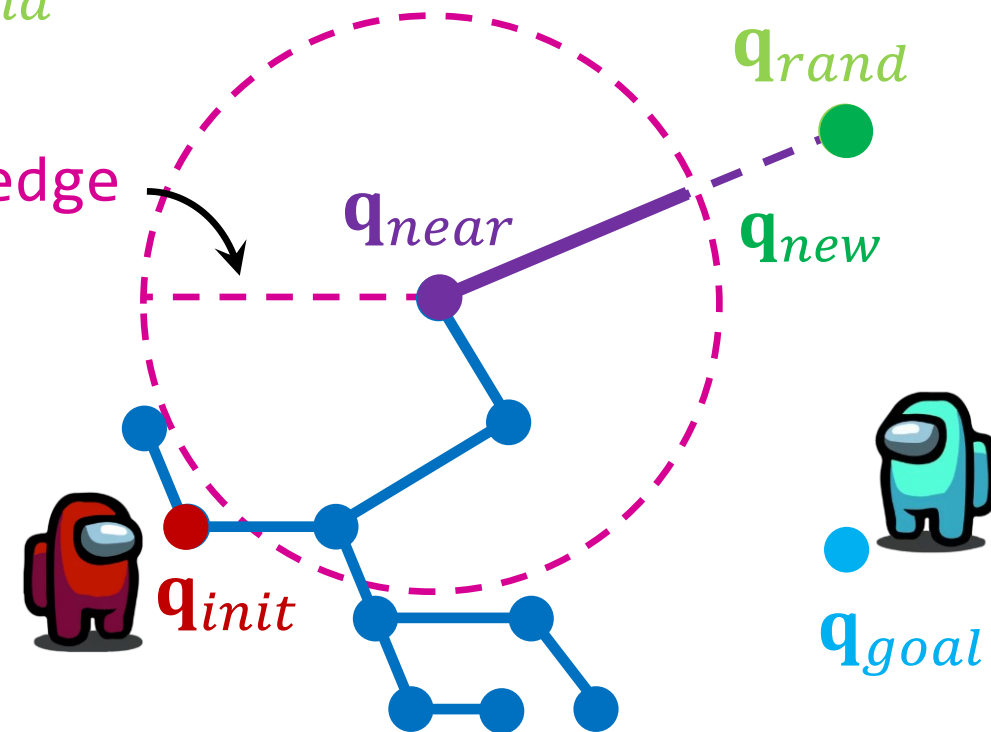


RRT (Rapidly-Exploring Random Trees)

Initialize tree with node containing $\mathbf{q}_{init}$

While a $\mathbf{q}_{goal}$ has not been reached

- ➔ 1. Sample $\mathbf{q}_{rand}$ from configuration space
2. Find $\mathbf{q}_{near}$, the node in tree closest to $\mathbf{q}_{rand}$
3. If $dist(\mathbf{q}_{near}, \mathbf{q}_{rand}) \leq max_edge$
 - $\mathbf{q}_{new} \leftarrow \mathbf{q}_{rand}$else
 - $\mathbf{q}_{new} \leftarrow$ compute $\mathbf{q}$ in direction of $\mathbf{q}_{rand}$ but only max_edge away
4. Add $\mathbf{q}_{new}$ to tree with parent $\mathbf{q}_{near}$

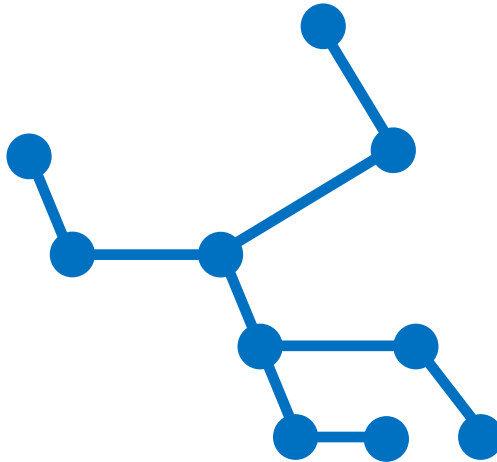


RRT (Rapidly-Exploring Random Trees)

What about collisions!

If $\mathbf{q}_{rand}$ is infeasible, either

- Reject it and continue to sample a new $\mathbf{q}_{rand}$
- Or go ahead and find $\mathbf{q}_{new}$ from the (infeasible) $\mathbf{q}_{rand}$



RRT (Rapidly-Exploring Random Trees)

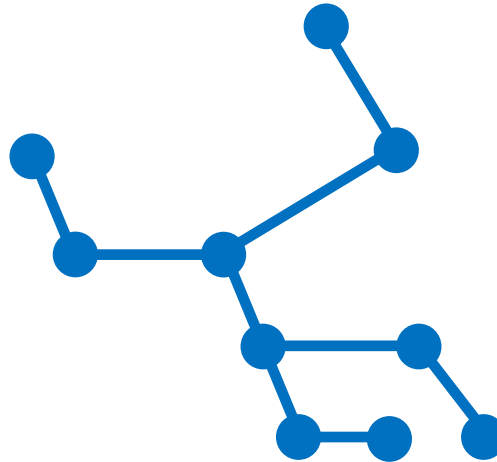
What about collisions!

If $\mathbf{q}_{rand}$ is infeasible, either:

- Reject it and continue to sample a new $\mathbf{q}_{rand}$
- Or go ahead and find $\mathbf{q}_{new}$ from the (infeasible) $\mathbf{q}_{rand}$

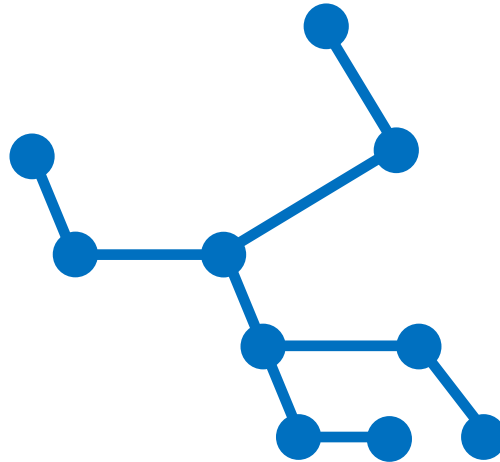
If $\mathbf{q}_{new}$ is infeasible just:

- Reject it and continue to sample a new $\mathbf{q}_{rand}$



RRT (Rapidly-Exploring Random Trees)

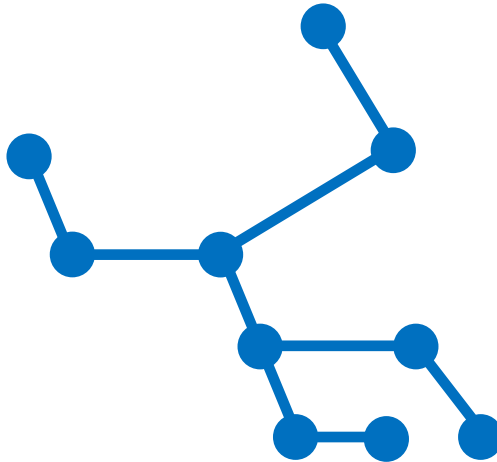
Check the whole line segment before adding a new config to the tree!



RRT (Rapidly-Exploring Random Trees)

Don't forget about **limits** of agent movement per step

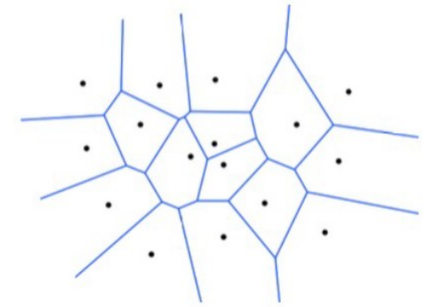
- We'll need to convert the RRT path to actions
- May need to chop up segment and
- Convert config pairs to actions



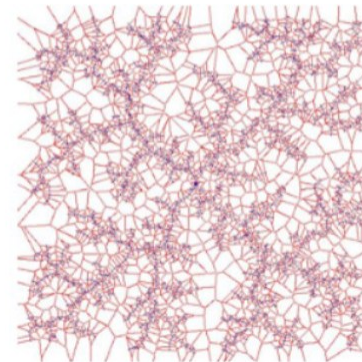
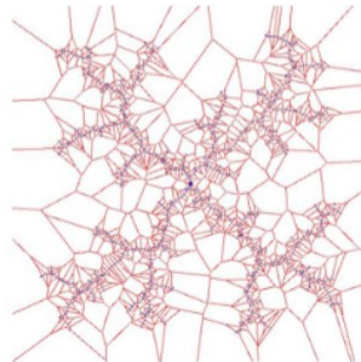
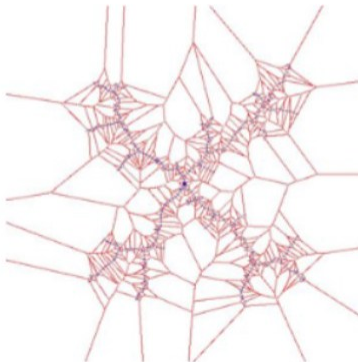
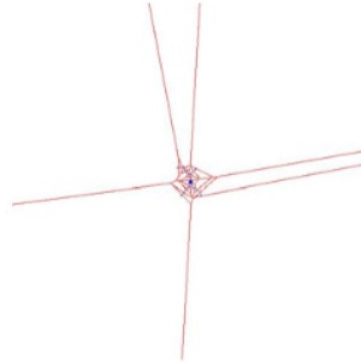
RRT Properties

Exploring the Frontier!

The 2D Voronoi diagram induced by a set of seed sites: Subdivision of the plane in cells, where each cell corresponds to the region where one site is the closest site for all the points in the region.



The largest Voronoi regions belong to the states on the frontier of the search. Since sampling is done uniformly, the probability of expanding an existing state is proportional to the size of its Voronoi region, such that the tree preferentially expands towards these large, so far unsearched areas.



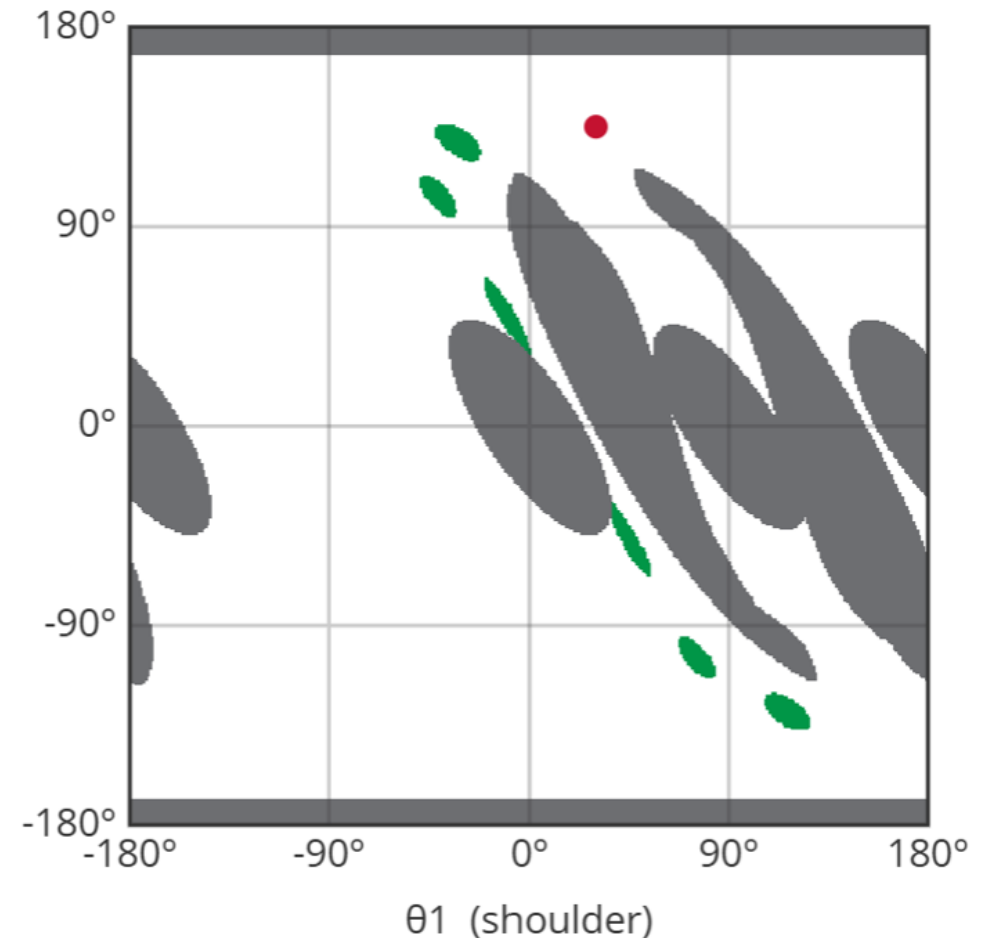
RRT Limitations: Completeness

RRT can be probabilistically complete

(We'll get there eventually)

Improvement

- With some probability let $\mathbf{q}_{rand}$ be the goal



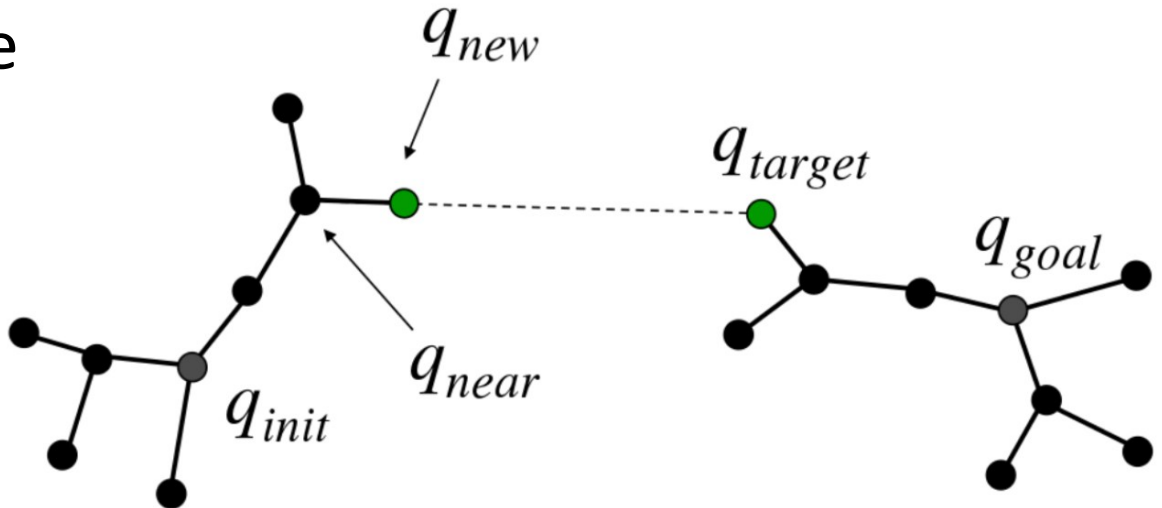
RRT Limitations: Completeness

RRT can be probabilistically complete

(We'll get there eventually)

Improvement

- With some probability let $\mathbf{q}_{rand}$ be the goal
- RRT-Connect: Grow a tree from the goal too



RRT Limitations: Optimality

A node's parent is fixed the moment the node is created, and it is never reconsidered.

- New samples can extend the tree, but they cannot repair a detour taken 500 iterations ago
- So the solution is determined by the early, sparse phase of the tree, when the samples were least informative.
- Karaman and Frazzoli, 2011: *RRT converges to a suboptimal solution with probability 1*

The fix is not more samples. It is to allow the tree to be rewritten

→ RRT *

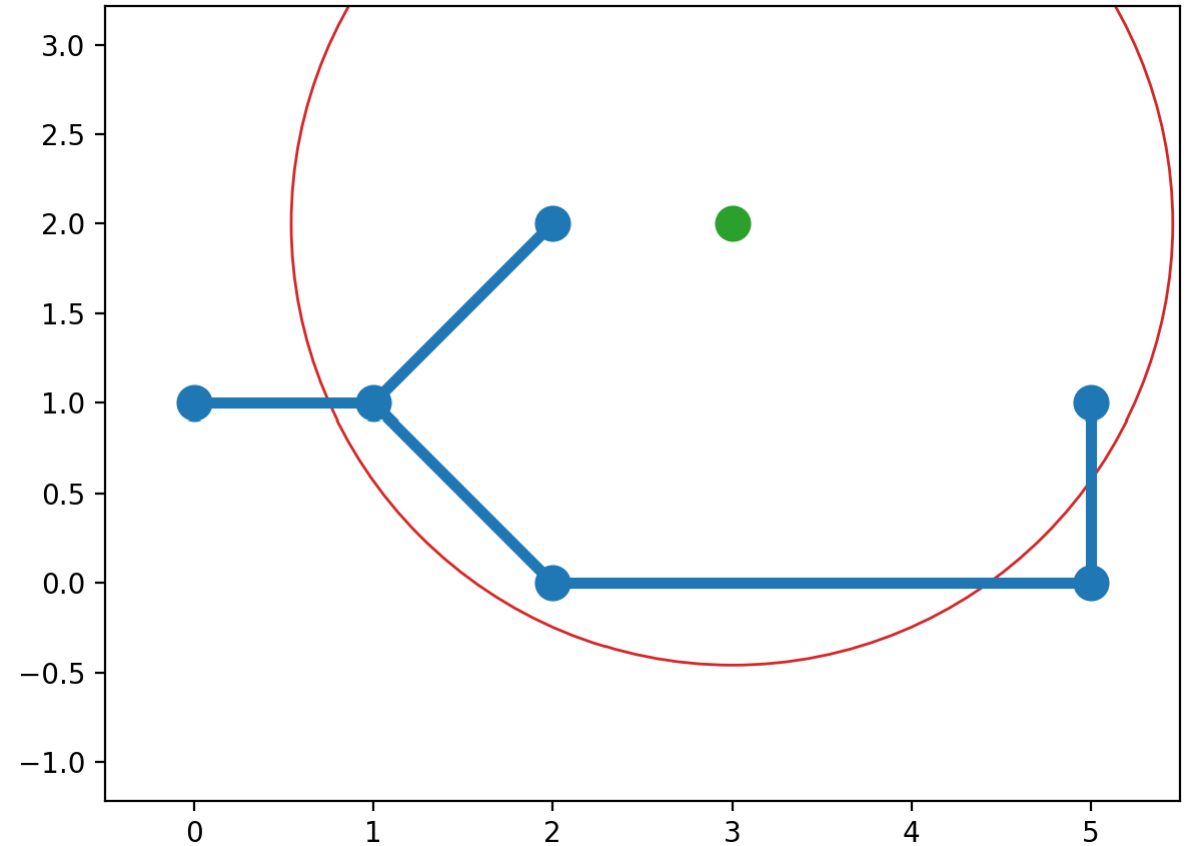
Motion Planning

RRT*

RRT*

Two tricks using the tree **path cost**:

- 1) Find a better parent
- 2) Rewire neighborhood



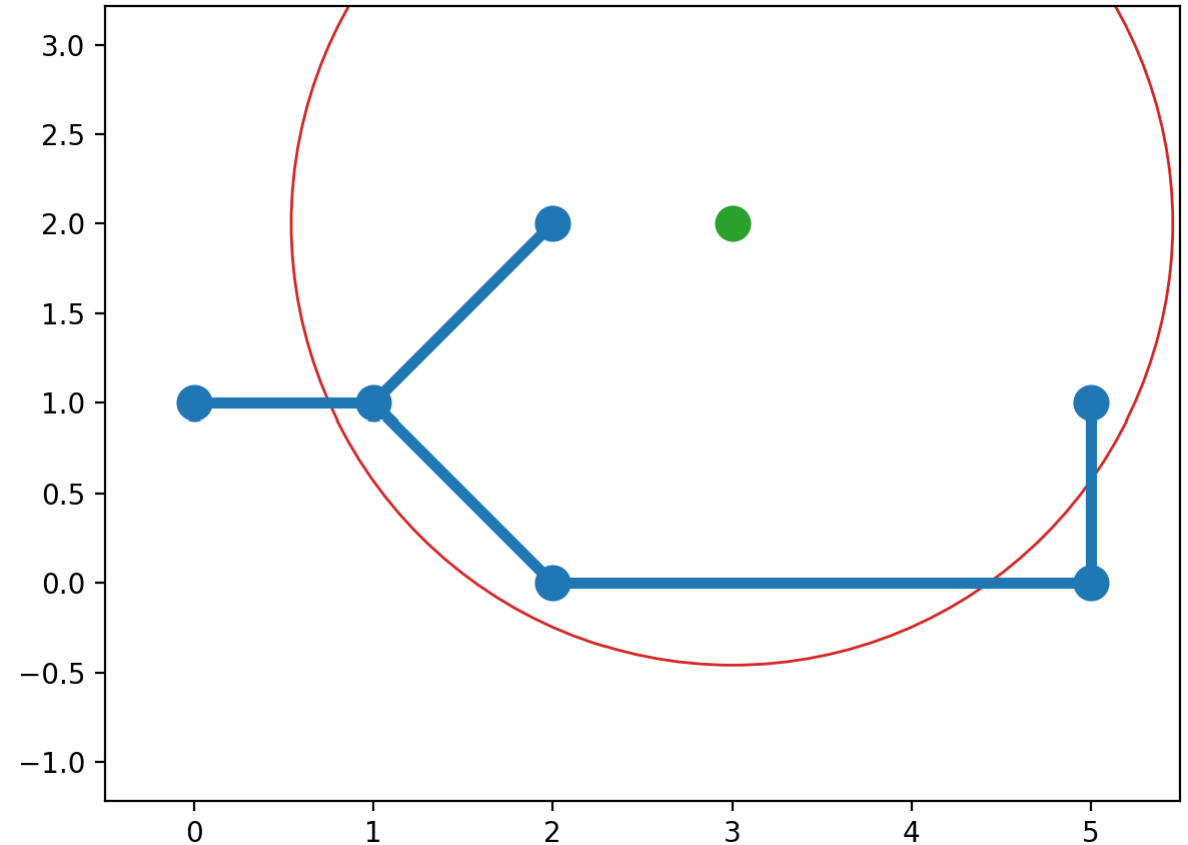
RRT*

Two tricks using the tree **path cost**: 1) Find a better parent

Instead of connecting $\mathbf{q}_{new}$ to the one nearest node in the tree ($\mathbf{q}_{near}$),

- Find **all nearest nodes** in the tree within a **radius R** of $\mathbf{q}_{new}$
- Choose $\mathbf{q}_{parent}$ of $\mathbf{q}_{new}$ to be the **neighbor** that gives the lowest **path cost** to $\mathbf{q}_{new}$:

$$cost(\mathbf{q}_{parent}) + c(\mathbf{q}_{parent}, \mathbf{q}_{new})$$



RRT*

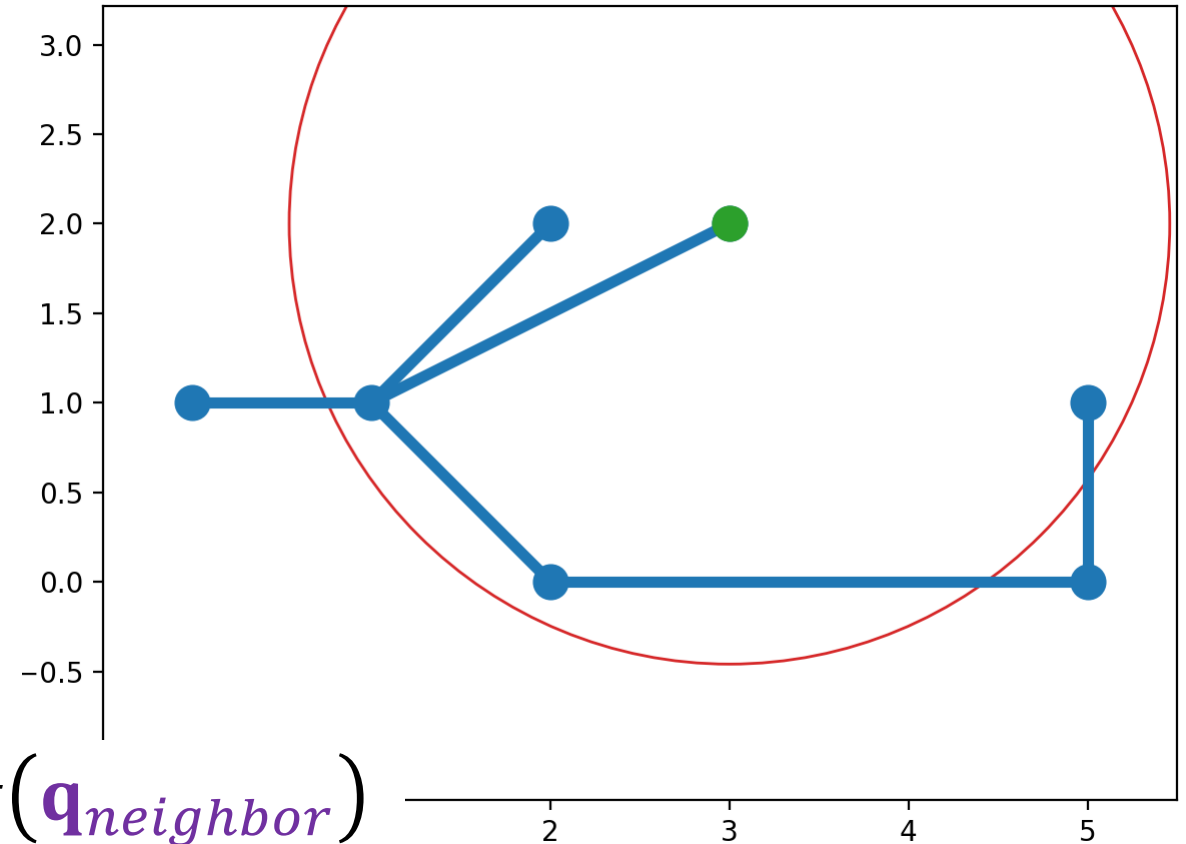
Two tricks using the tree **path cost**: 2) Rewire the neighborhood to see if the new node is a better parent

After attaching q_{new} to the best neighbor,

For each $q_{neighbor}$ within a **radius R** of q_{new} :

- Change parent of $q_{neighbor}$ to q_{new} if it provides a better **path cost** to $q_{neighbor}$, i.e., if:

$$cost(q_{new}) + c(q_{new}, q_{neighbor}) < cost(q_{neighbor})$$



RRT*

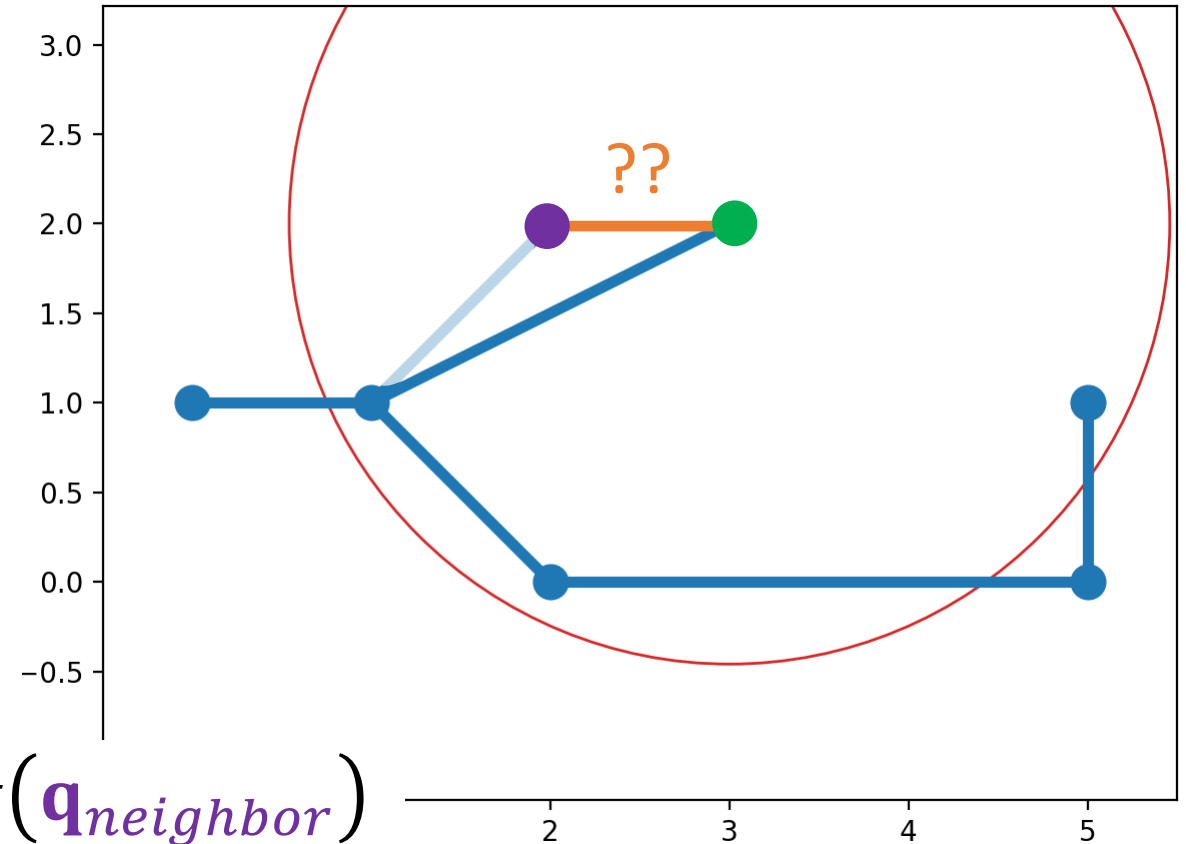
Two tricks using the tree **path cost**: 2) Rewire the neighborhood to see if the new node is a better parent

After attaching $\mathbf{q}_{new}$ to the best neighbor,

For each $\mathbf{q}_{neighbor}$ within a **radius R** of $\mathbf{q}_{new}$:

- Change parent of $\mathbf{q}_{neighbor}$ to $\mathbf{q}_{new}$ if it provides a better **path cost** to $\mathbf{q}_{neighbor}$, i.e., if:

$$cost(\mathbf{q}_{new}) + c(\mathbf{q}_{new}, \mathbf{q}_{neighbor}) < cost(\mathbf{q}_{neighbor})$$



RRT*

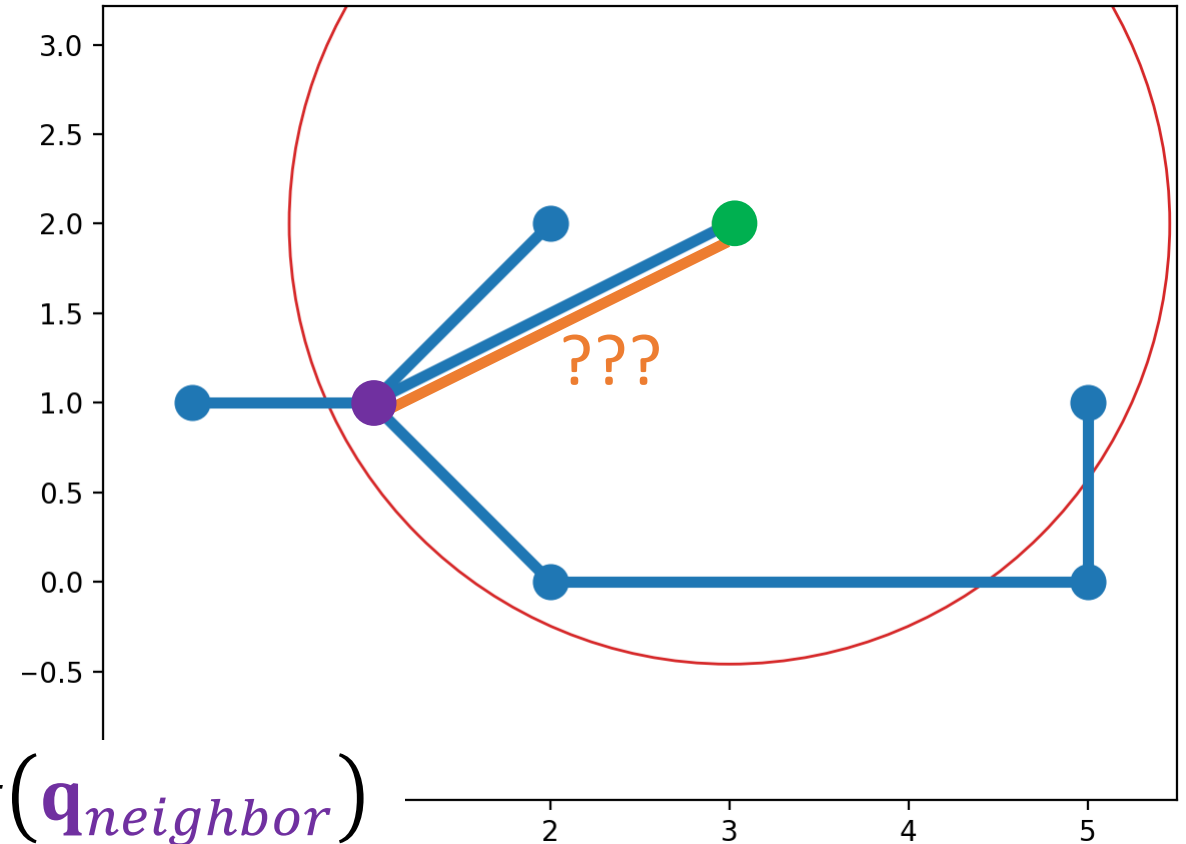
Two tricks using the tree **path cost**: 2) Rewire the neighborhood to see if the new node is a better parent

After attaching q_{new} to the best neighbor,

For each $q_{neighbor}$ within a **radius R** of q_{new} :

- Change parent of $q_{neighbor}$ to q_{new} if it provides a better **path cost** to $q_{neighbor}$, i.e., if:

$$cost(q_{new}) + c(q_{new}, q_{neighbor}) < cost(q_{neighbor})$$



RRT*

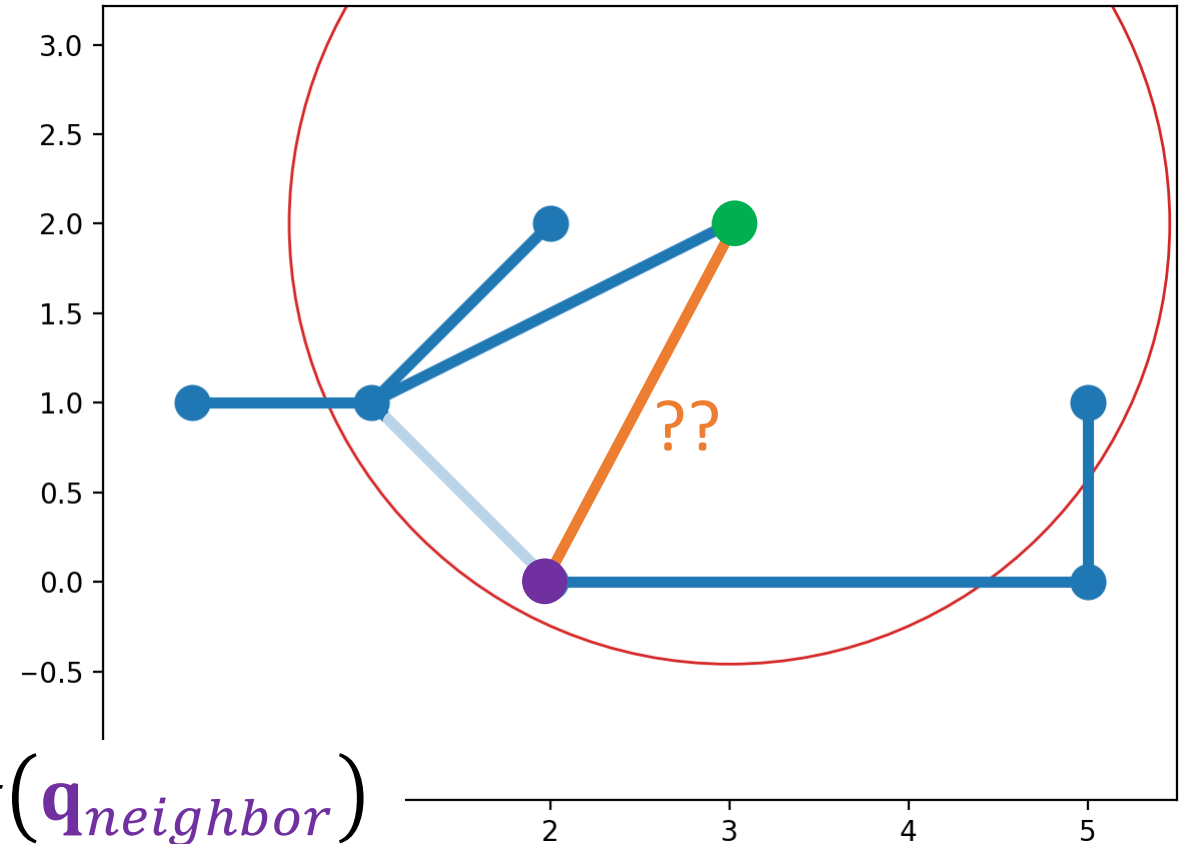
Two tricks using the tree **path cost**: 2) Rewire the neighborhood to see if the new node is a better parent

After attaching $\mathbf{q}_{new}$ to the best neighbor,

For each $\mathbf{q}_{neighbor}$ within a **radius R** of $\mathbf{q}_{new}$:

- Change parent of $\mathbf{q}_{neighbor}$ to $\mathbf{q}_{new}$ if it provides a better **path cost** to $\mathbf{q}_{neighbor}$, i.e., if:

$$cost(\mathbf{q}_{new}) + c(\mathbf{q}_{new}, \mathbf{q}_{neighbor}) < cost(\mathbf{q}_{neighbor})$$



RRT*

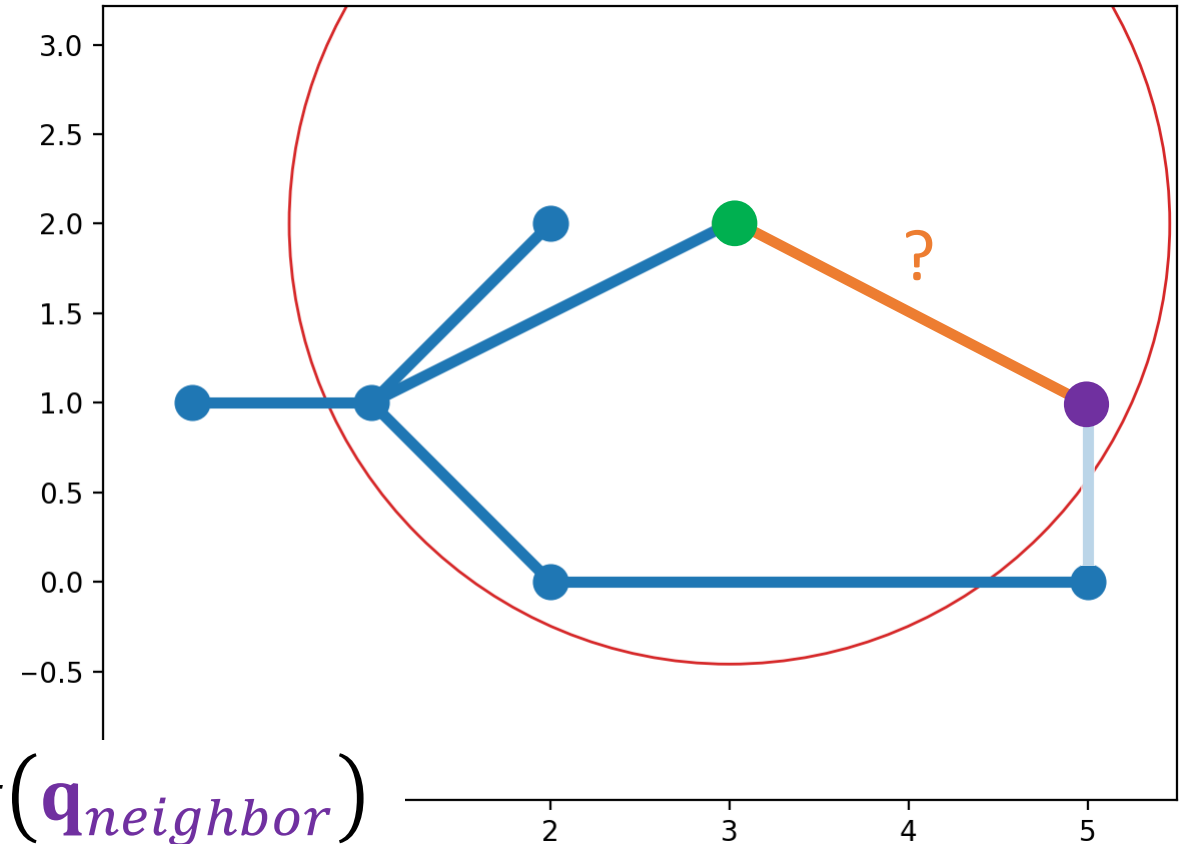
Two tricks using the tree **path cost**: 2) Rewire the neighborhood to see if the new node is a better parent

After attaching q_{new} to the best neighbor,

For each $q_{neighbor}$ within a **radius R** of q_{new} :

- Change parent of $q_{neighbor}$ to q_{new} if it provides a better **path cost** to $q_{neighbor}$, i.e., if:

$$cost(q_{new}) + c(q_{new}, q_{neighbor}) < cost(q_{neighbor})$$



Beyond RRT

Lots of extensions!

[Wikipedia list](#) →

- Rapidly exploring [random graph](#) (RRG) and RRT*,^{[5][6][7]} a variant of RRT that converges towards an optimal solution
- LQR-RRT,^[8] a [kinodynamic](#) variant for complex or [underactuated](#) dynamics
- RRT*,^[9] a family of RRT-based planners aiming to generate solutions for high-dimensional systems in real-time, by progressively searching in lower-dimensional subspaces.
- RRT*-Smart,^[10] a method for [accelerating the convergence rate](#) of RRT* by using path optimization (in a similar fashion to [Theta*](#)) and intelligent sampling (by biasing sampling towards path vertices, which – after path optimization – are likely to be close to obstacles)
- A*-RRT and A*-RRT*,^[11] a two-phase [motion planning](#) method that uses a [graph search algorithm](#) to search for an initial feasible path in a low-dimensional space (not considering the complete state space) in a first phase, avoiding hazardous areas and preferring low-risk routes, which is then used to focus the RRT* search in the continuous high-dimensional space in a second phase
- RRT*FN,^{[12][13][14]} RRT* with a fixed number of nodes, which randomly removes a leaf node in the tree in every iteration
- RRT*-AR,^[15] sampling-based alternate routes planning
- Informed RRT*,^{[16][17]} improves the convergence speed of RRT* by introducing a [heuristic](#), similar to the way in which [A*](#) improves upon [Dijkstra's algorithm](#)
- Real-Time RRT* (RT-RRT*),^[18] a variant of RRT* and informed RRT* that uses an online tree rewiring strategy that allows the tree root to move with the agent without discarding previously sampled paths, in order to obtain [real-time](#) path-planning in a dynamic environment such as a computer game
- RRT^X and RRT[#],^{[19][20]} optimization of RRT* for dynamic environments
- Theta*-RRT^[21] a two-phase [motion planning](#) method similar to A*-RRT* that uses a hierarchical combination of [any-angle search](#)

Summary: Motion Planning

Dealing with continuous configuration space

1) Chop into discrete configuration space or 2) sample the configuration space (e.g., RRT)

RRT

Grow a tree by randomly sample points from configuration space and adding short, feasible segments from the existing tree toward the sampled config.

Stop when you've connected $\mathbf{q}_{init}$ with $\mathbf{q}_{goal}$.

- Not optimal

RRT*

Consider minimal path cost as samples are added to the RRT, rewiring parents as needed

- Converges to optimal path (path to goal keeps getting better until optimal)