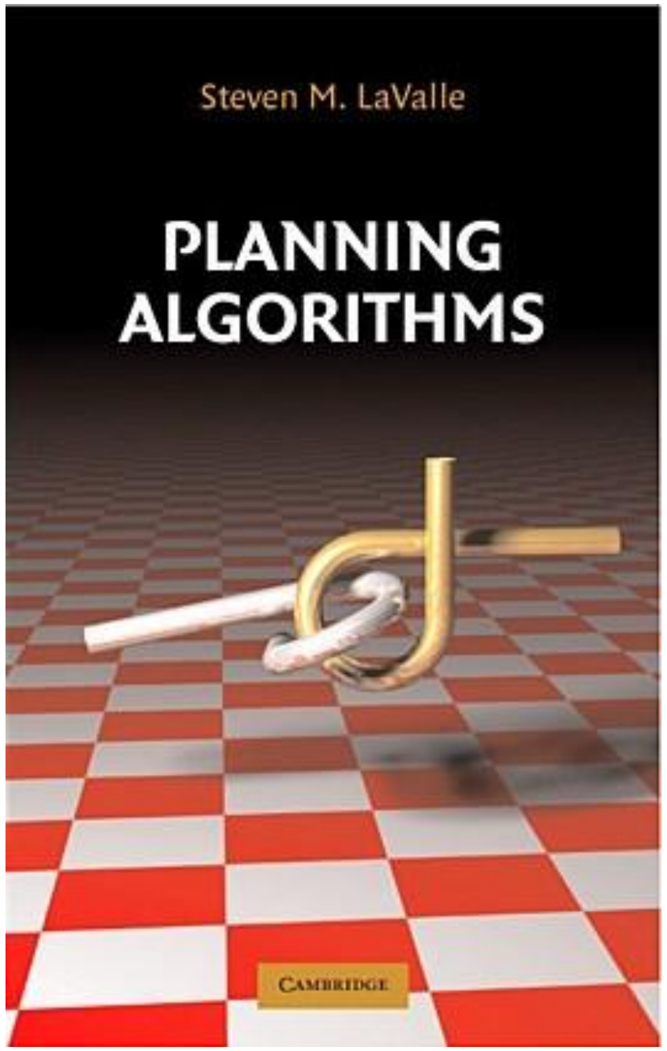
C-space and RRTs

Textbooks



http://planning.cs.uiuc.edu

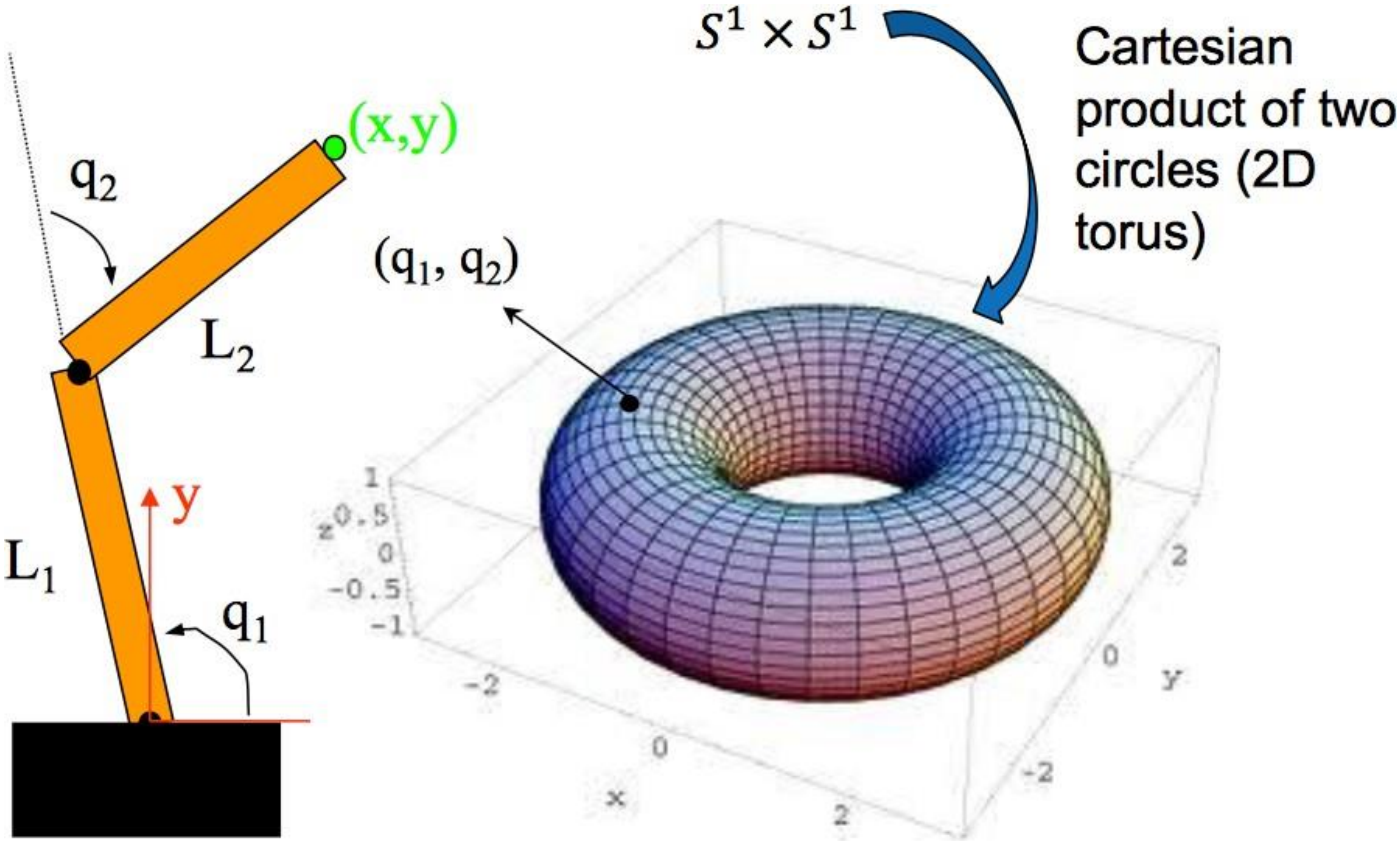
Configuration

- A *configuration* is a complete specification of every point of the robot
- C-space is the space of all possible configurations

- Some examples:
  - a point robot that can translate in 2D: (x, y)
  - a robot that can rotate and translate: (x, y, theta)

Example

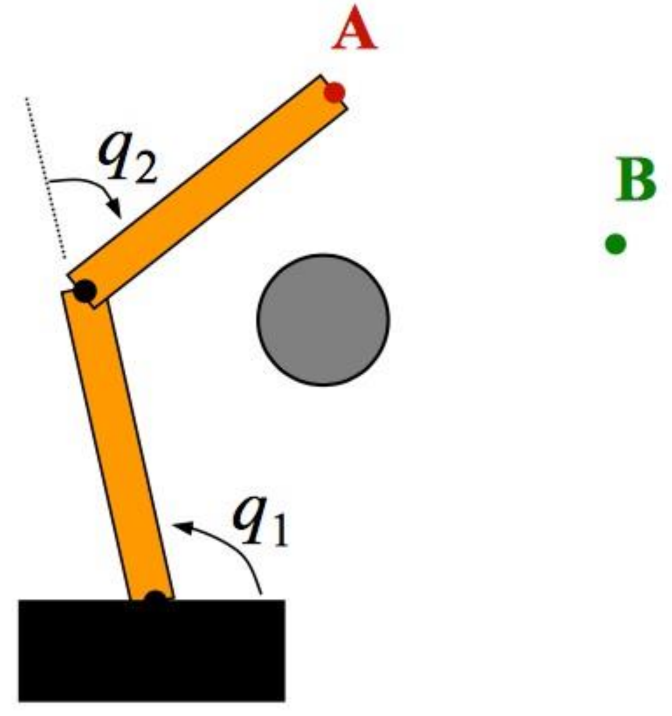
Two Link Manipulator



Topology of Configuration Space

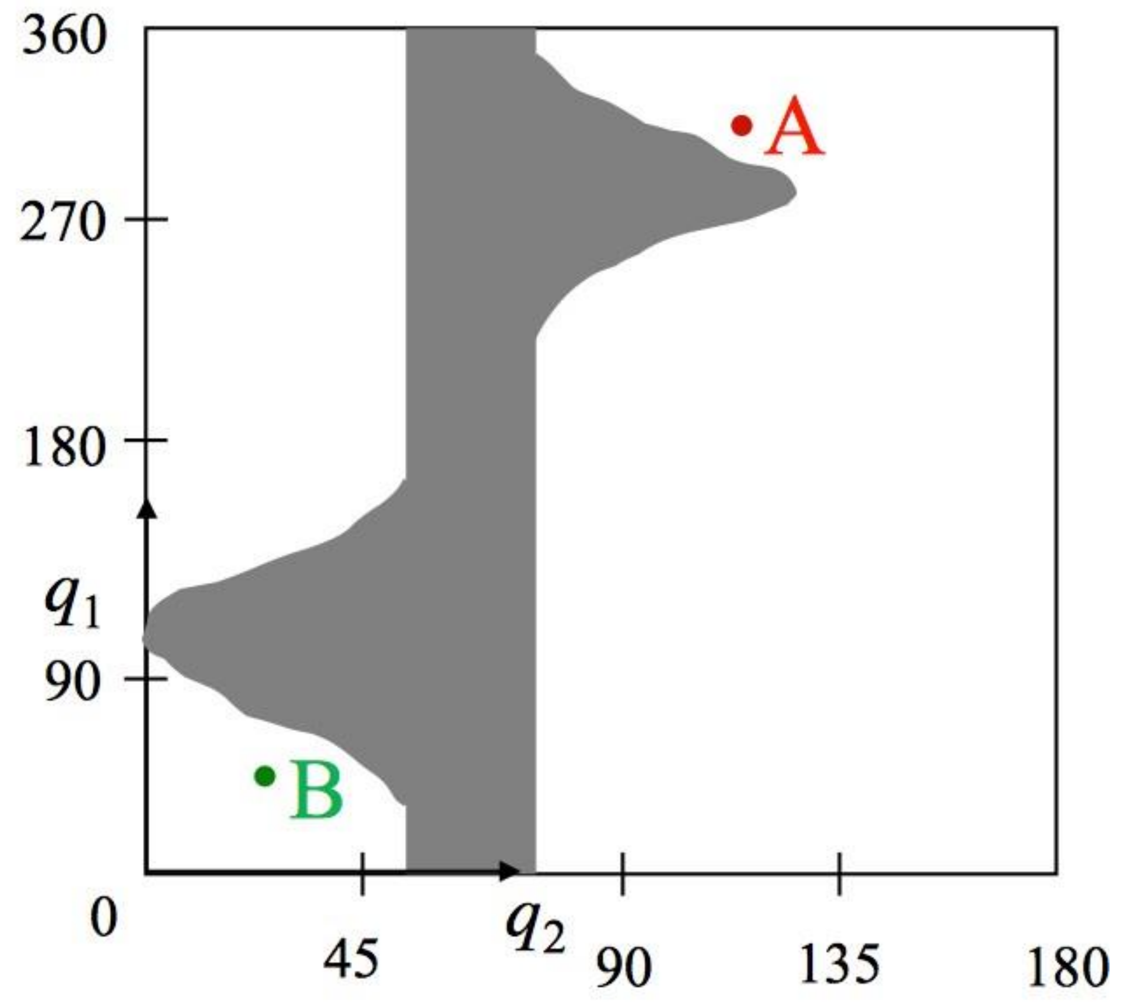
C-space Obstacles

Reference *configuration*



An obstacle in the robot's workspace

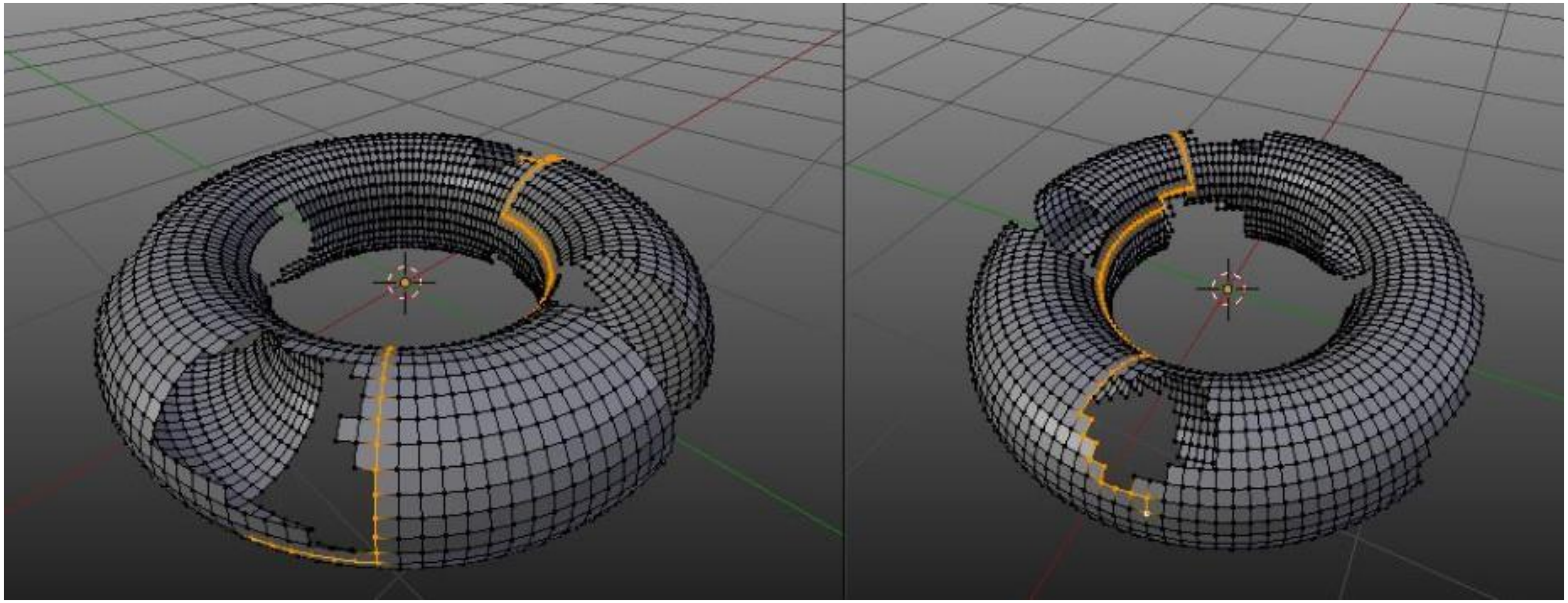
How do we get from **A** to **B** ?



The C-space representation of this obstacle…

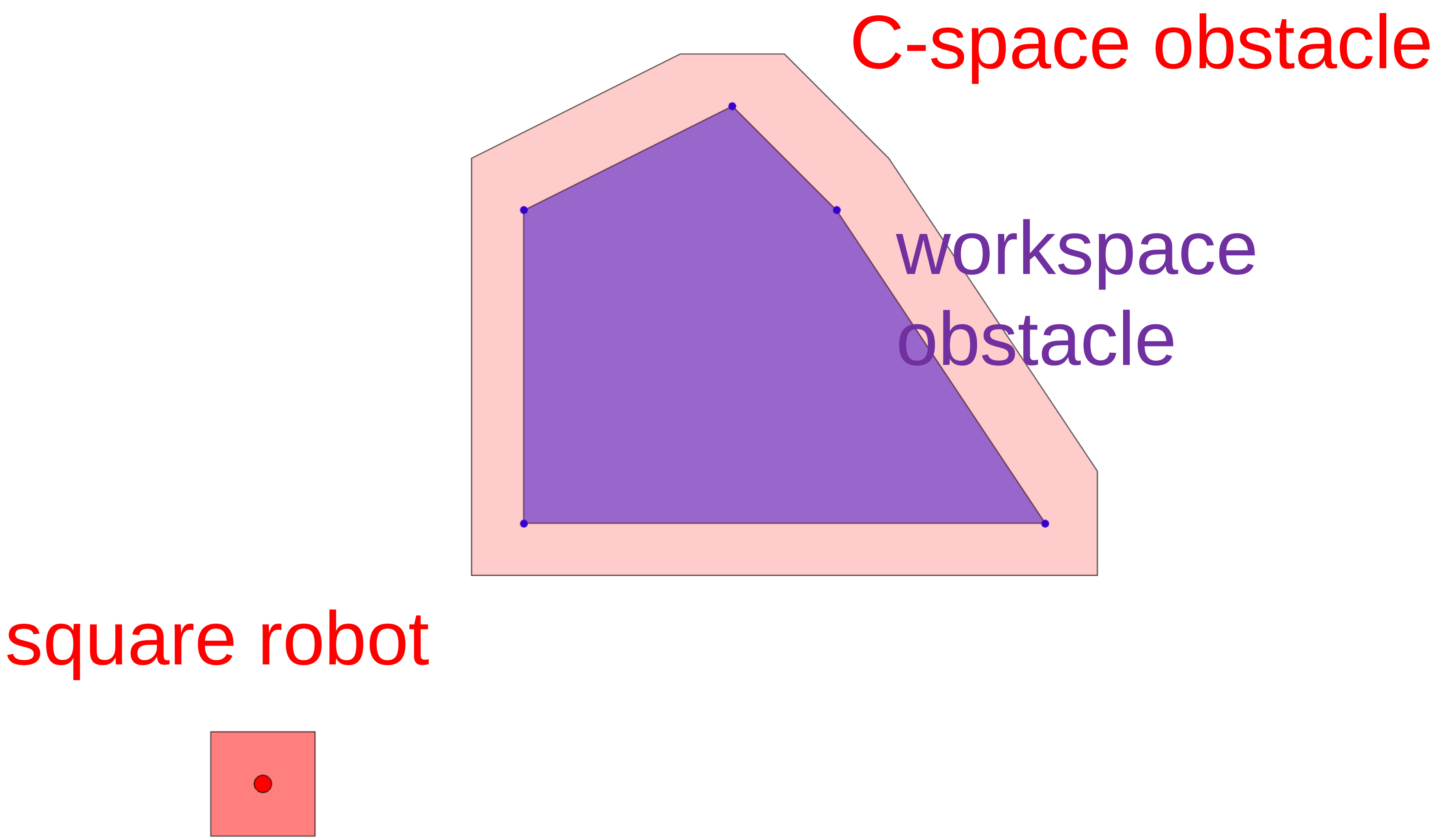
Punching holes in the donut

- We plan for shortest paths in the C-space and not in the workspace.
- *Naïve idea*: draw a grid in the C-space
- No vertices/edges inside C-space obstacles



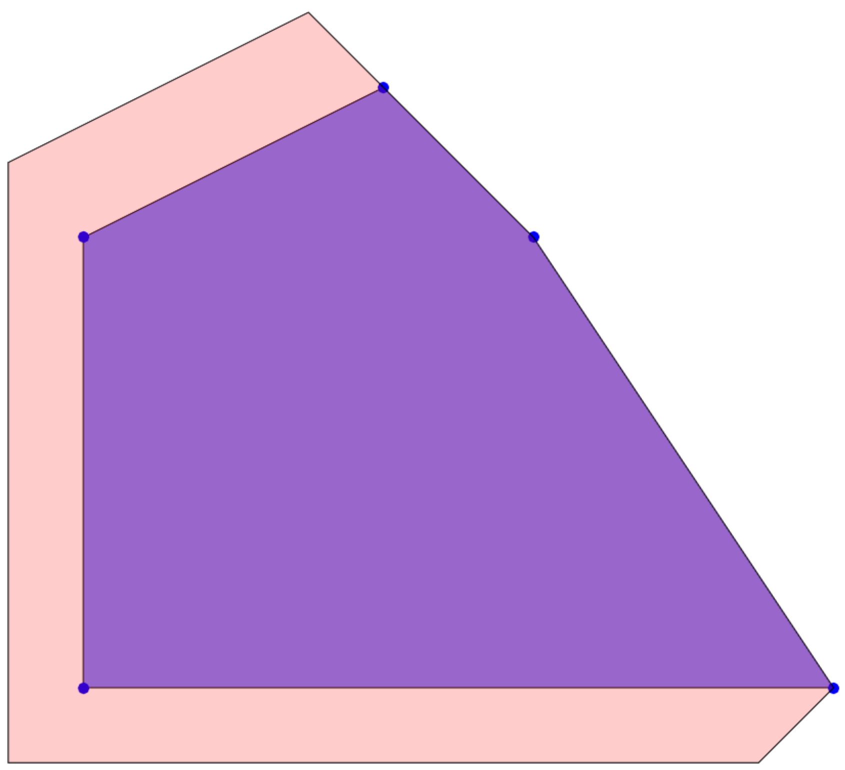
2D robot that can only translate

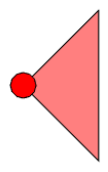
- If robot and obstacles are both polygonal, we can compute C-space obstacles by taking Minkowski difference



Non-Symmetric Robots

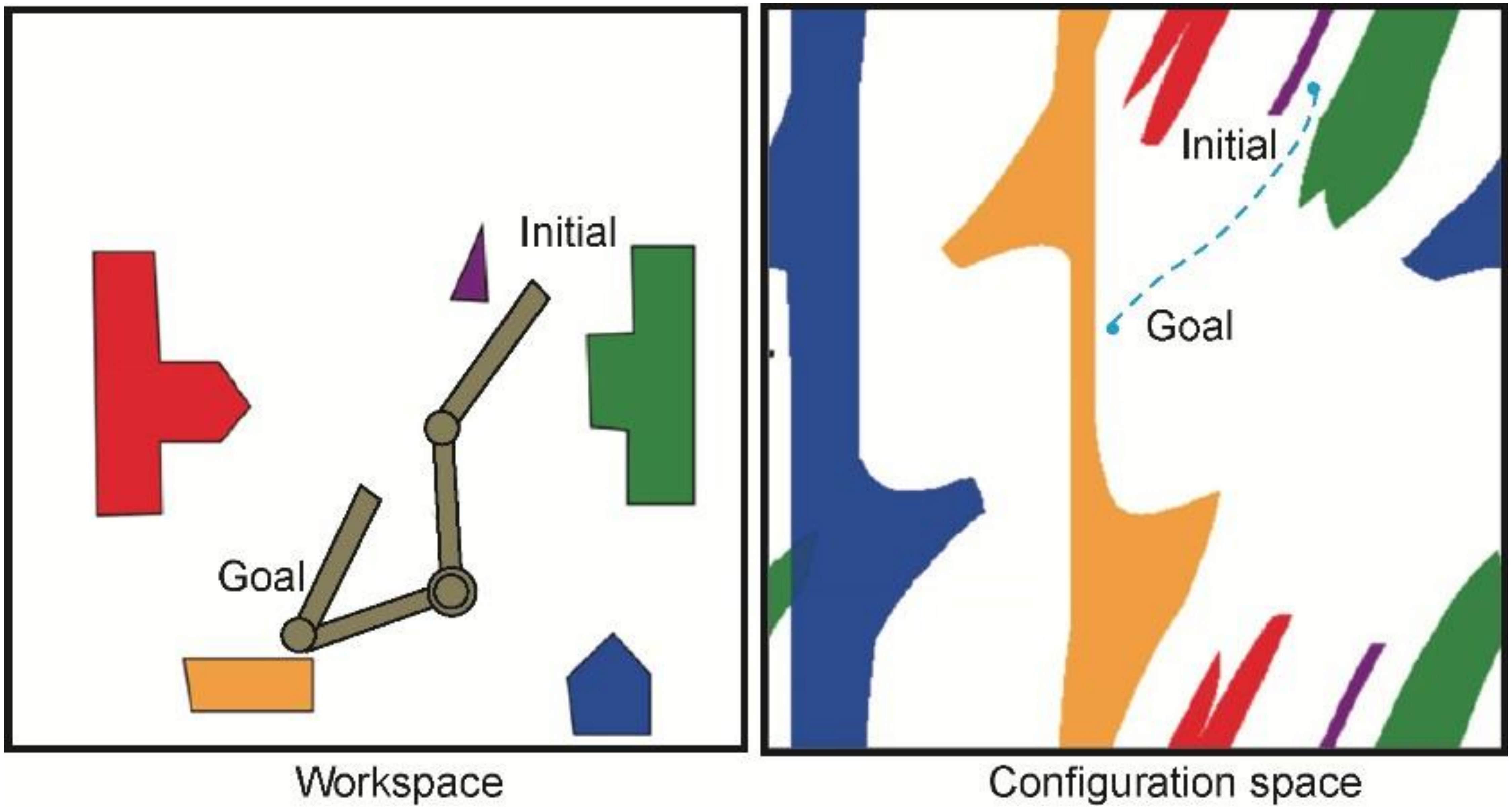
- If the robot is not symmetric about the origin, we should take the Minkowski difference and not the sum!
- That is "flip" the robot and then take Minkowski sum





C-space obstacles can become complicated quickly!

[Pan and Manocha, '15]



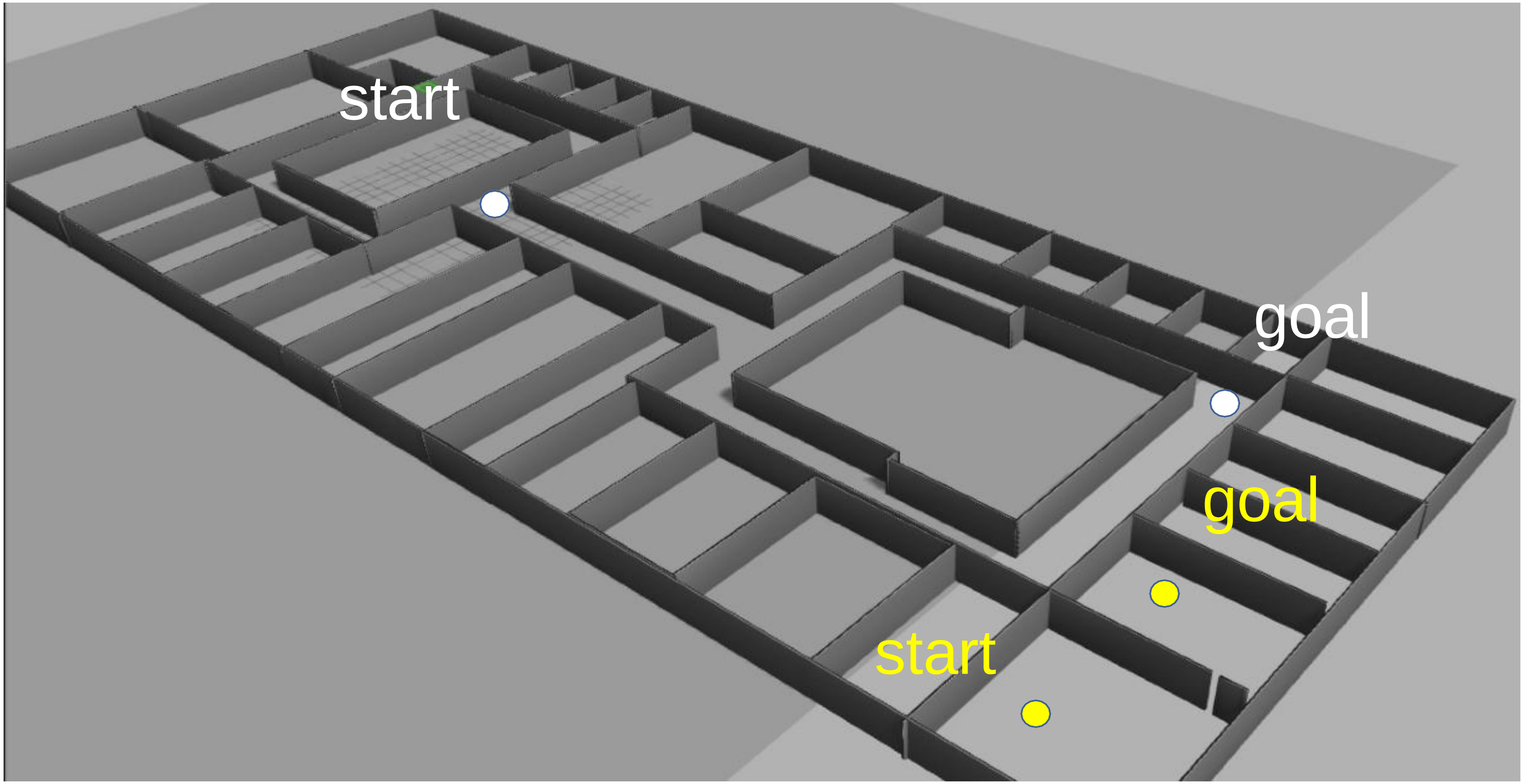
Computing exact C-space obstacles is challenging.

Solution v1

- *Idea*: checking whether a specific configuration is in collision is easier
- Let’s assume we have a *fast* collision checker
  - black-box for now
- Don’t construct the C-space obstacles
- Discretize the C-space
- For each vertex, check if it is in collision, else add it to the graph

Discretizing the C-space

What is a good grid resolution? Too small and we may miss solutions, too large makes it computationally expensive.



Discretizing the C-space

- Recall that we are planning in the C-space and not the workspace
- The robot is a point in C-space
- C-space can be very complex and “narrow openings”
- Finding one path, let alone, the optimal path is challenging

Solution v2

- Design a scheme that produces *sequences* of discrete samples, x_i, in the C-space

Solution v2

- Design a scheme that produces *sequences* of discrete samples, x_i, in the C-space
- Check if x_i is a collision-free configuration
- If x_i is in C_{free} then add it to the graph
  - add edges existing vertices (*more details later*)
- If x_i is not in C_{free}, then discard it

- Check if we have found a path, else repeat

Rapidly Exploring Dense Trees (RDTs)

- One of the most popular techniques
- Introduced by LaValle in ‘98
  - many, many, many extensions and variants

RDTs vs RRTs vs PRMs

- Many versions of the same idea
  - *with different guarantees*

- Rapidly exploring Random Trees
  - randomly sample the C-space
  - graph built will be a tree
- Rapidly exploring Dense Trees
  - any sequence of samples (not necessarily random)
- Probabilistic Roadmaps
  - build a roadmap instead of a tree
  - useful for multiple queries with diff. start and goals

And many, many variants

- articulated robots
- kinematics, dynamics, differential constraints

- http://msl.cs.uiuc.edu/rrt/gallery.html (circa 2000) sampled config.

SIMPLE_RDT(q_0)

$\mathcal{G}$.init(q_0);

for $i = 1$ **to** k **do**

  $\mathcal{G}$.add_vertex($\alpha(i)$);

starting configuration

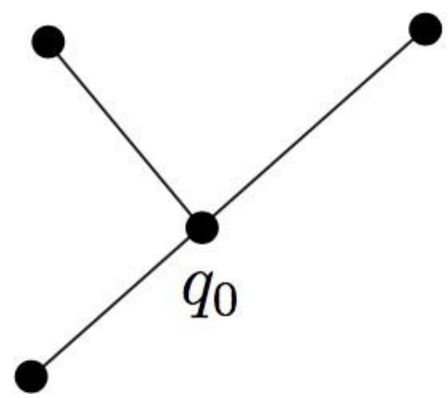
  $q_n \leftarrow$ NEAREST($S(\mathcal{G}), \alpha(i)$);

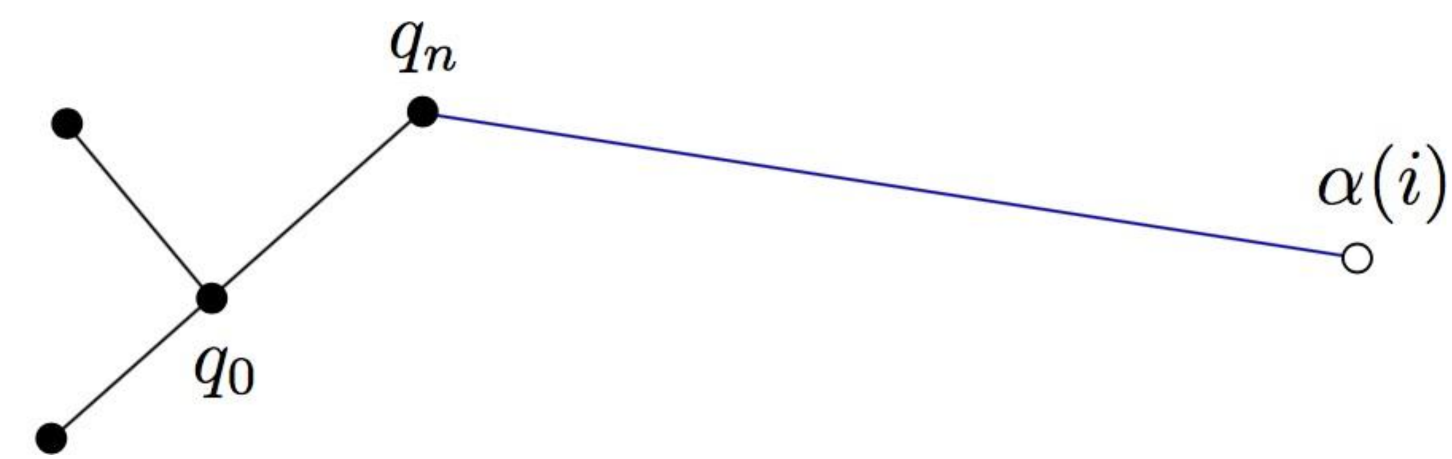
  $\mathcal{G}$.add_edge($q_n, \alpha(i)$);

SIMPLE_RDT(q_0)

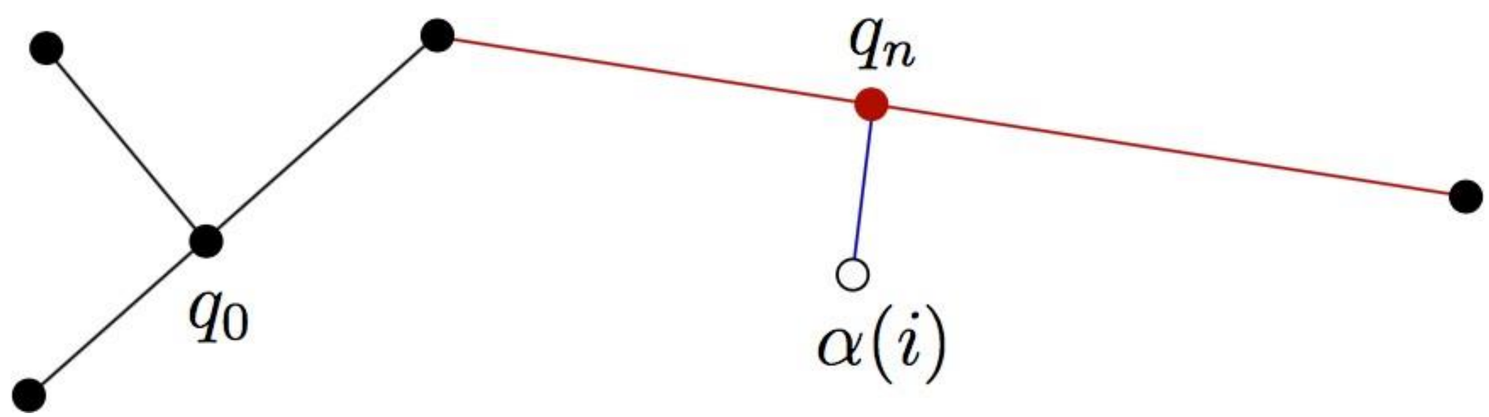
$\mathcal{G}$.init(q_0);
for $i = 1$ **to** k **do**
    $\mathcal{G}$.add_vertex($\alpha(i)$);
    $q_n \leftarrow$ NEAREST($S(\mathcal{G}), \alpha(i)$);
    $\mathcal{G}$.add_edge($q_n, \alpha(i)$);

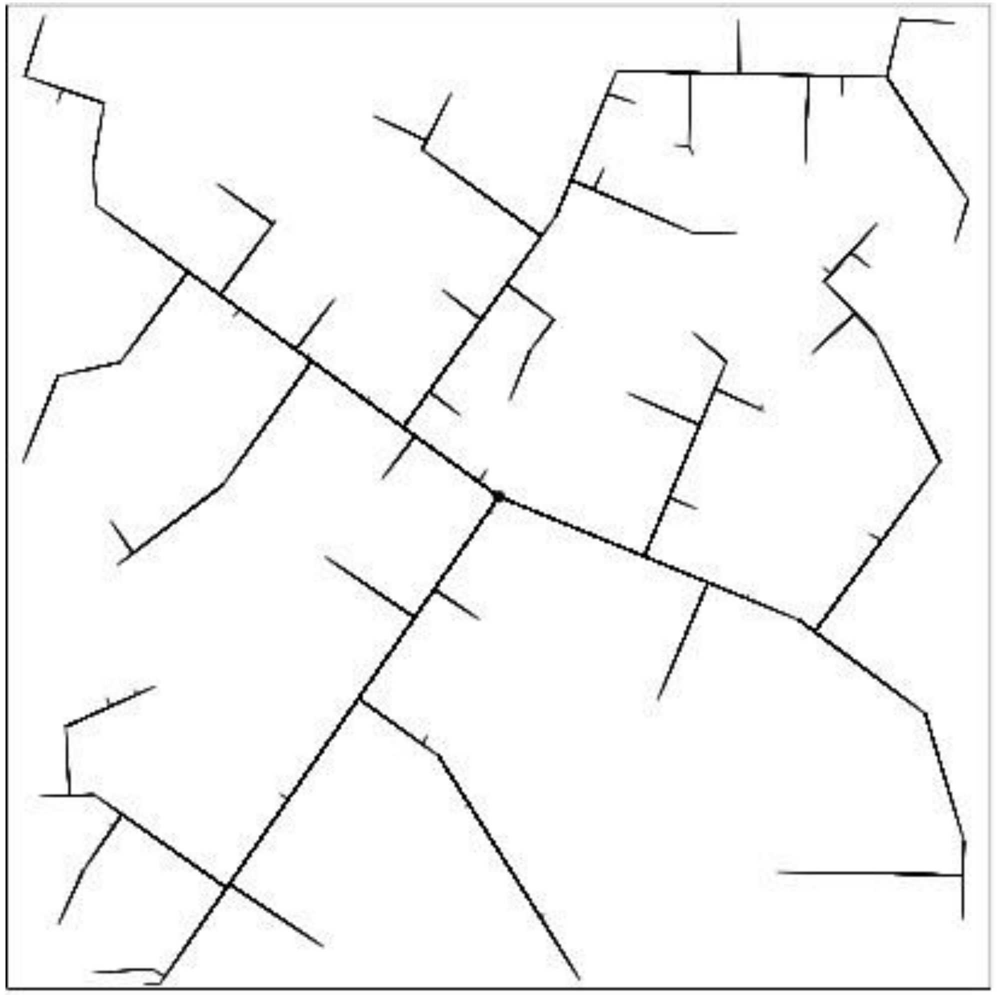
assume no obstacles



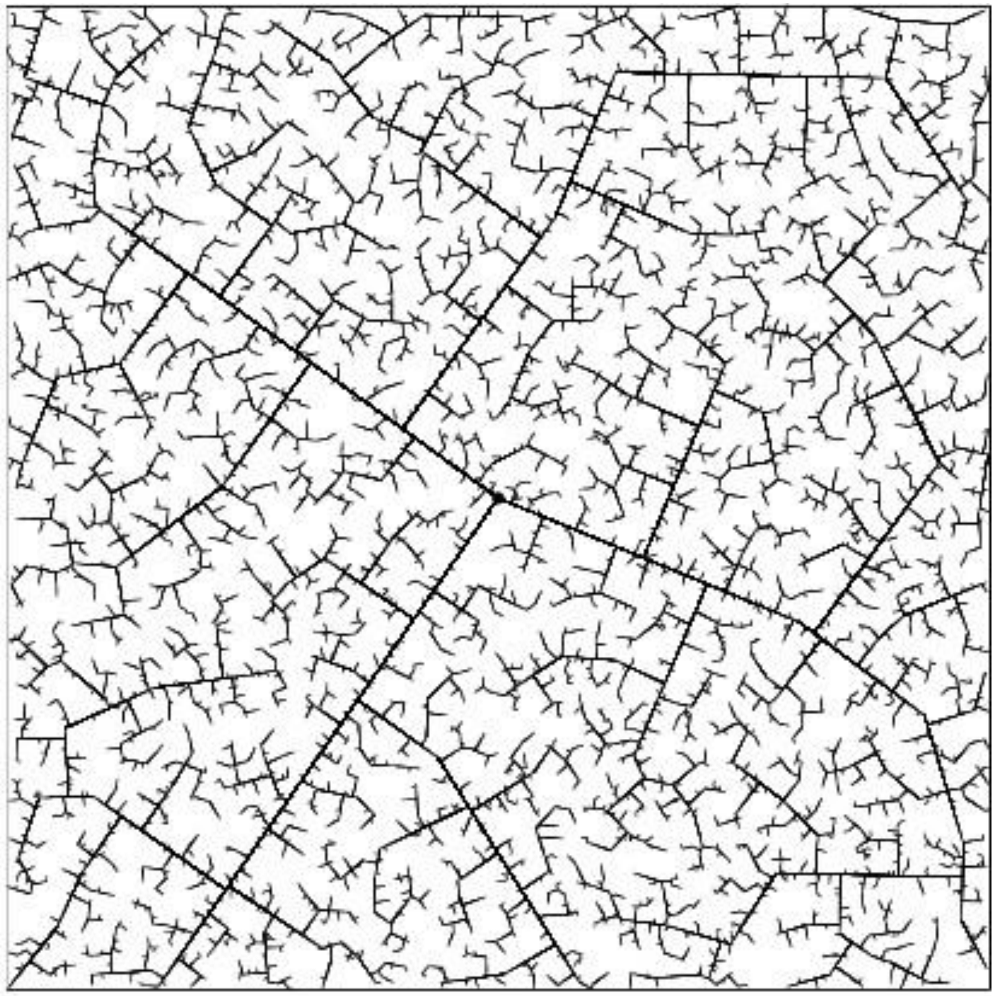


if nearest point lies on an existing edge, then split that edge





45 iterations



2345 iterations

recall that there is *no* parameter involved

```
RDT(q_0)
  G.init(q_0);
  for i = 1 to k do
      q_n ← NEAREST(S, α(i));
      q_s ← STOPPING-CONFIGURATION(q_n, α(i));
      if q_s ≠ q_n then
          G.add_vertex(q_s);
          G.add_edge(q_n, q_s);
```

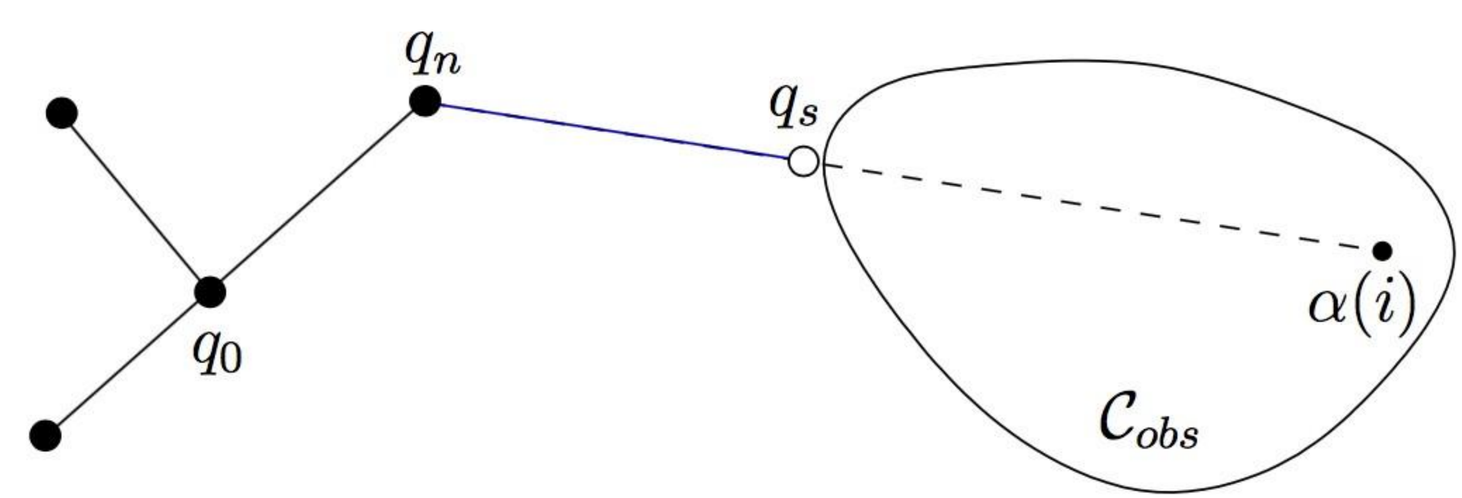


Figure 5.20: If there is an obstacle, the edge travels up to the obstacle boundary, as far as allowed by the collision detection algorithm.

What about the goal?

- So far, RDT is only building a tree
- Occasionally add the goal configuration and see if it gets connected to the tree
  - say every 100th iteration

RDT(q_0)

$\mathcal{G}$.init(q_0);
for $i = 1$ **to** k **do**
  $q_n \leftarrow$ NEAREST($S, \alpha(i)$);
  $q_s \leftarrow$ STOPPING-CONFIGURATION(q_n,$\alpha(i)$);
  if $q_s \neq q_n$ **then**
    $\mathcal{G}$.add_vertex(q_s);
    $\mathcal{G}$.add_edge(q_n, q_s);

Remember

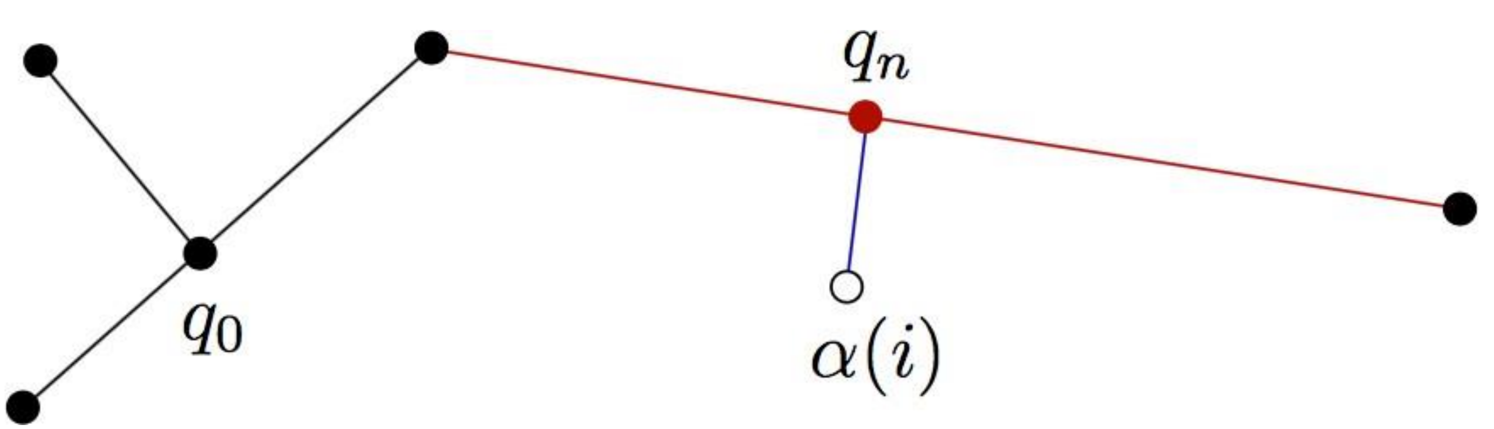
- We are in the C-space
- A vertex in the graph is a specific configuration
- How to find the *nearest* configuration?
  - What is the *distance* function?

RDT(q_0)

$\mathcal{G}$.init(q_0);
for $i = 1$ **to** k **do**
    $q_n \leftarrow$ NEAREST$(S, \alpha(i))$;
    $q_s \leftarrow$ STOPPING-CONFIGURATION$(q_n, \alpha(i))$;
    if $q_s \neq q_n$ **then**
        $\mathcal{G}$.add_vertex(q_s);
        $\mathcal{G}$.add_edge(q_n, q_s);

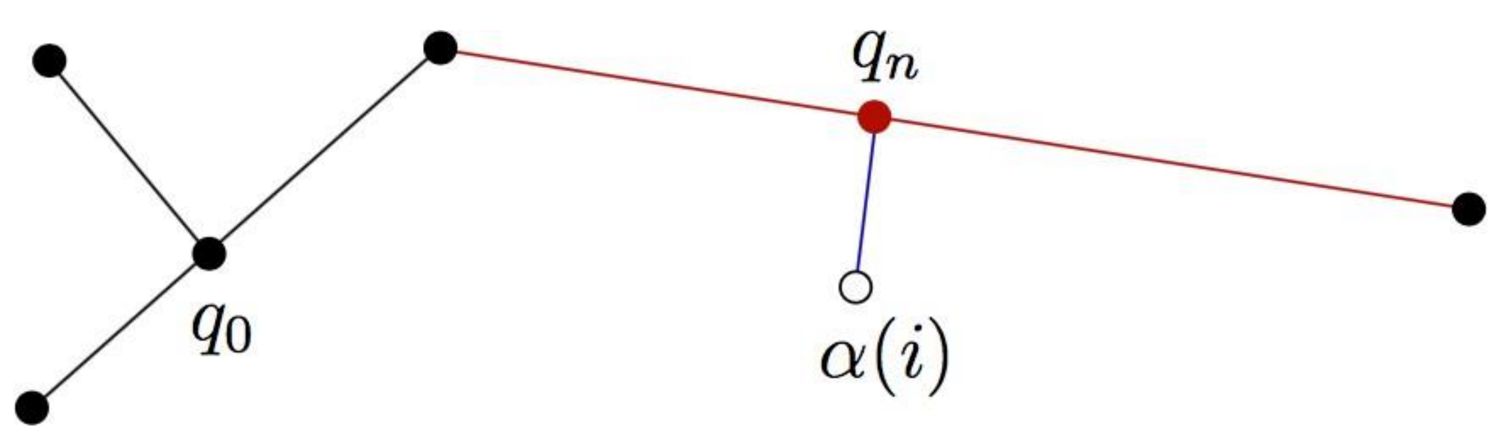
Distances in C-space

- The C-space is not necessarily Euclidean space
- Need to define appropriate *distances*



Distances in C-space

- The C-space is not necessarily Euclidean space
- Need to define appropriate *distances*
- Each *edge* represents a path in C-space
  - An edge means that there is a *collision-free path* between two configurations



Practical Solutions

- Use a step size parameter
- Move in steps and check collision of a configuration
- Often, a *local steering* function is used instead*

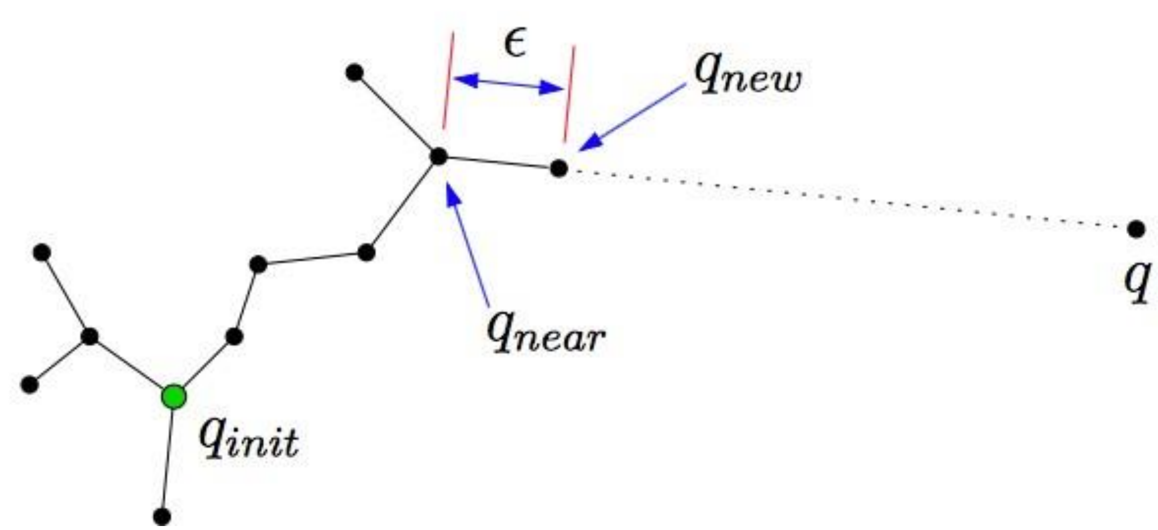
BUILD_RRT(q_{init})

$\mathcal{T}$.init(q_{init});
for $k = 1$ **to** K **do**
  $q_{rand} \leftarrow$ RANDOM_CONFIG();
  EXTEND($\mathcal{T}, q_{rand}$);
Return $\mathcal{T}$

EXTEND($\mathcal{T}, q$)

$q_{near} \leftarrow$ NEAREST_NEIGHBOR($q, \mathcal{T}$);
if NEW_CONFIG(q, q_{near}, q_{new}) **then**
  $\mathcal{T}$.add_vertex(q_{new});
  $\mathcal{T}$.add_edge(q_{near}, q_{new});
  if $q_{new} = q$ **then**
    Return *Reached*;
  else
    Return *Advanced*;
Return *Trapped*;

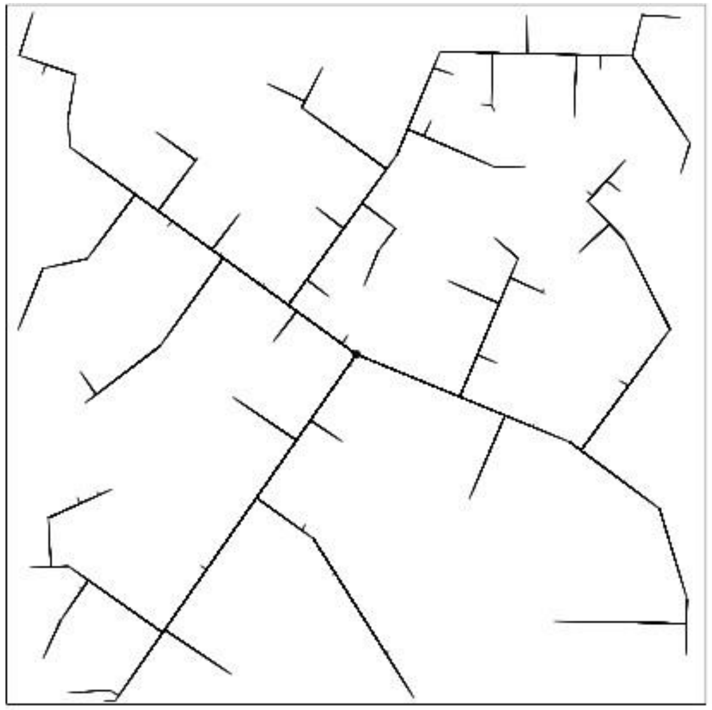
Figure 2: The basic RRT construction algorithm.



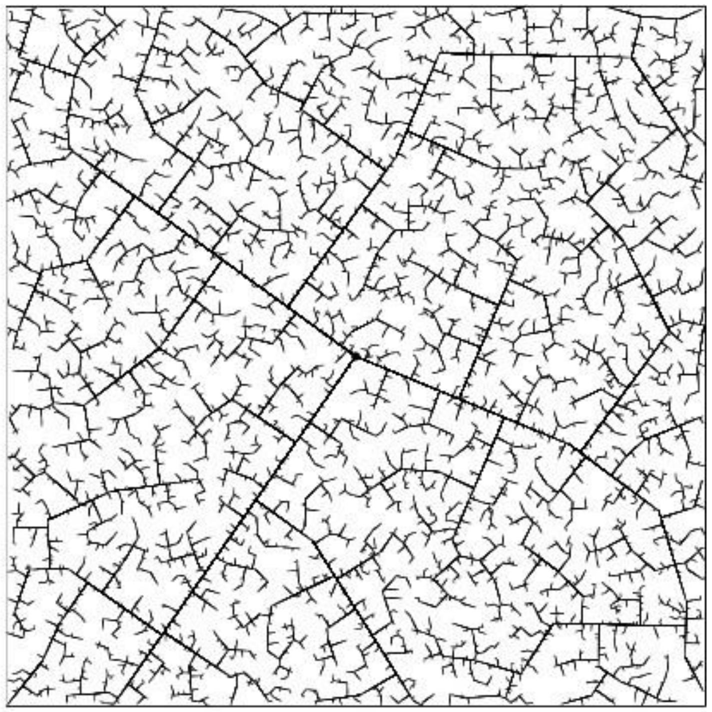
https://personalrobotics.ri.cmu.edu/files/courses/papers/Kuffner00-rrtconnect.pdf

Tree "grows" from the start config. due to step size parameter

without step size

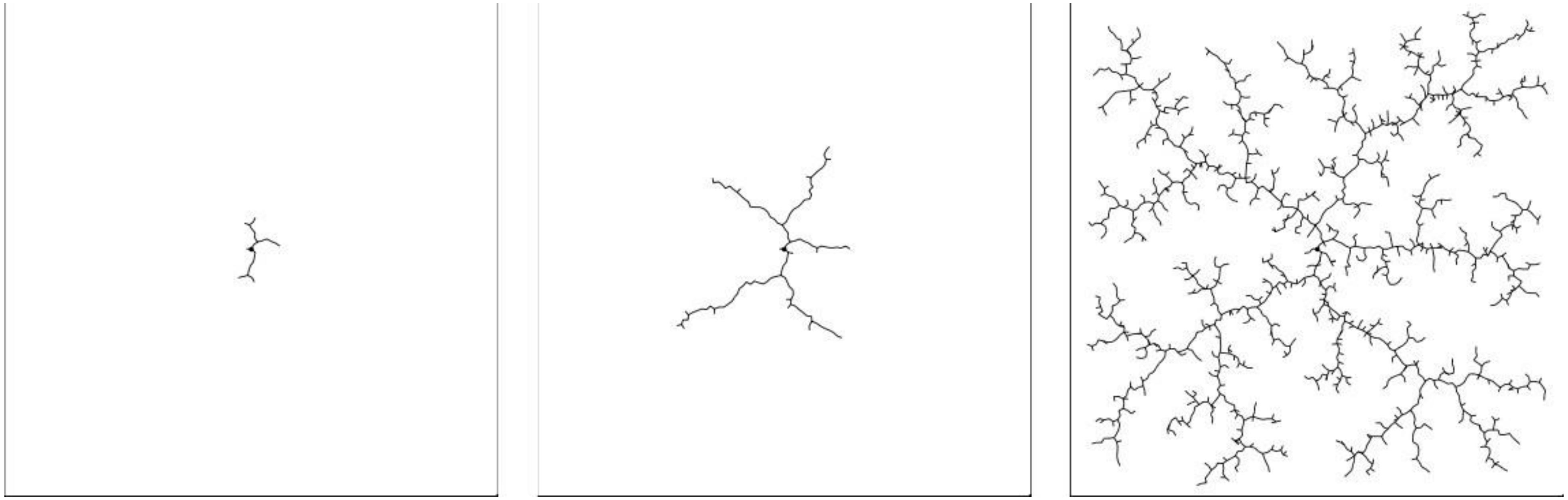


45 iterations



2345 iterations

with step size



- Is RRT complete?

Guarantees?

Guarantees?

- Is RRT complete?
  - No but it is *probabilistically complete* – if there exists a path, the probability that RRT will not find the path decays to zero as the number of samples approaches infinity.

Guarantees?

- Is RRT complete?
  - No but it is *probabilistically complete* – if there exists a path, the probability that RRT will not find the path decays to zero as the number of samples approaches infinity.
  - Probability of failure decreases *exponentially*.
  - *Not all variants are probabilistically complete.*

- Is RRT optimal?

Guarantees?

- Is RRT optimal?
  - No. In fact, emphatically no!
  - "… *the probability that the RRT converges to an optimum solution, as the number of samples approaches infinity, is zero under some reasonable technical assumptions*."

https://arxiv.org/pdf/1005.0416.pdf

Guarantees?

- Is RRT optimal?
  - No. In fact, emphatically no!
  - "… *the probability that the RRT converges to an optimum solution, as the number of samples approaches infinity, is zero under some reasonable technical assumptions*."
- The main reason is that in RRT once we build a tree, we never modify the tree.
- RRT* is RRT + rewiring of the tree
  - Optimal!

https://arxiv.org/pdf/1005.0416.pdf

Why are RRTs so popular?

- Once you define the C-space and implement following subroutines, the actual algorithm is very *simple*:
  - random configuration generator
  - nearest neighbor
  - collision checker
- In *practice*, it works rather *well*.
- Can add *heuristics* on top, *e.g.*, bias the random config. generator towards goal configuration

EXTEND() is the real crux

- EXTEND() takes the tree and *extends it closer to the given random configuration*
- We say the Euclidean case where it's a straight line
- All kinematic (and dynamic) constraints can be handled within EXTEND()

BUILD_RRT(q_{init})
$\mathcal{T}$.init(q_{init});
for $k = 1$ **to** K **do**
  $q_{rand} \leftarrow$ RANDOM_CONFIG();
  EXTEND($\mathcal{T}, q_{rand}$);
Return $\mathcal{T}$

EXTEND($\mathcal{T}, q$)
$q_{near} \leftarrow$ NEAREST_NEIGHBOR($q, \mathcal{T}$);
if NEW_CONFIG(q, q_{near}, q_{new}) **then**
  $\mathcal{T}$.add_vertex(q_{new});
  $\mathcal{T}$.add_edge(q_{near}, q_{new});
  if $q_{new} = q$ **then**
    Return *Reached*;
  else
    Return *Advanced*;
Return *Trapped*;