Path Planning For Autonomous Systems

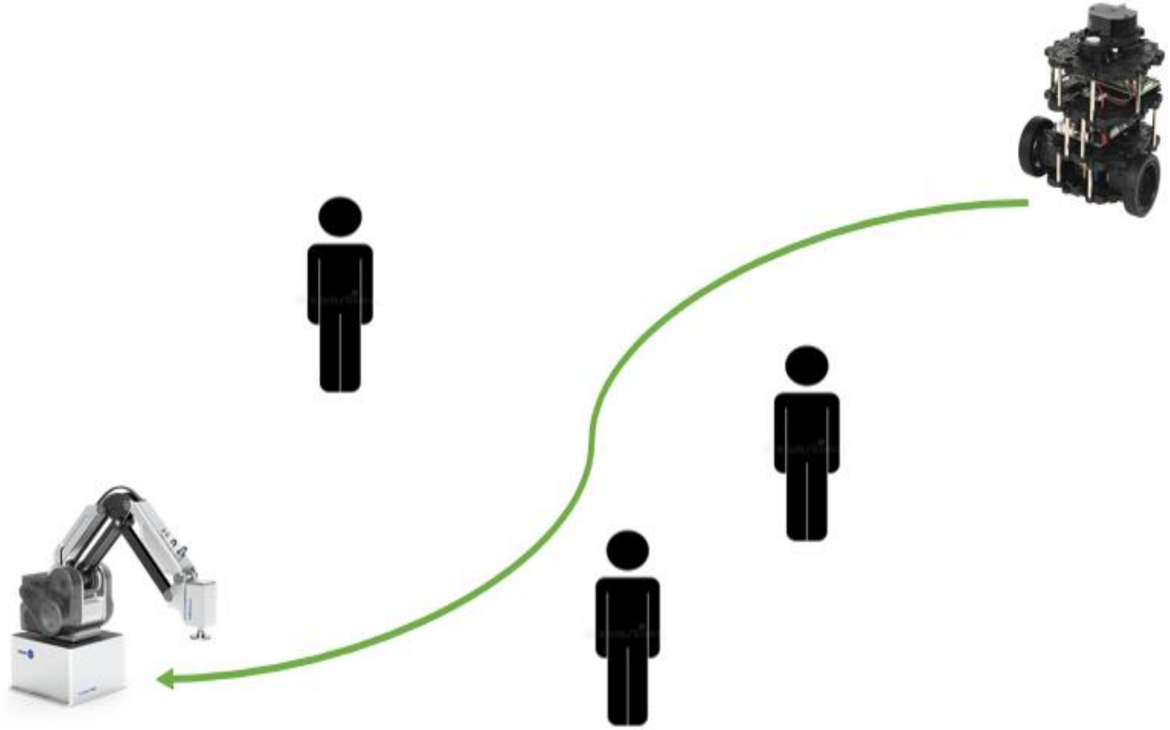
Mukund Mitra, Pradipta Biswas

Mukund Mitra
PhD Scholar, IISc Bangalore
Contact: mukundmitra@iisc.ac.in

Pradipta Biswas
Assoc. professor, IISc Bangalore
Contact: pradipta@iisc.ac.in

Definition: Path Planning

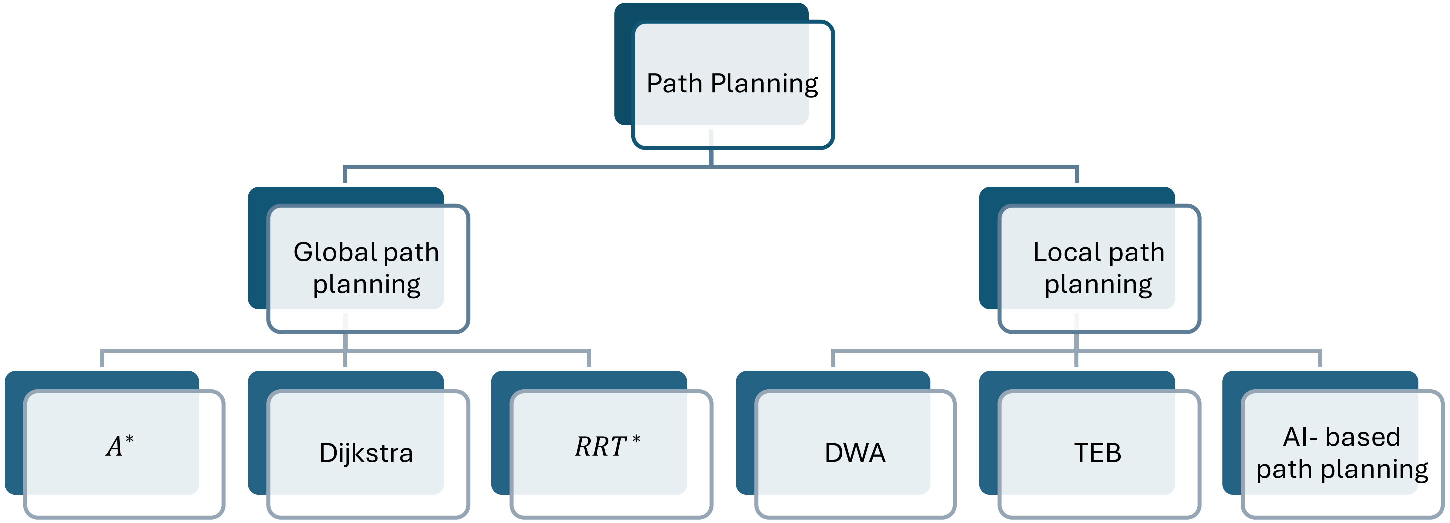
- Finding a continuous path connecting start and goal
- Mobile robots, unmanned aerial vehicles, and autonomous vehicles
- safe, efficient, collision-free, and least-cost travel paths from an origin to a destination



Applications

- Warehouse applications
- Manufacturing
- Safety and patrolling
- Autonomous driving

Classification



Path Planning

Global path planning

Local path planning

A^*

Dijkstra

RRT^*

DWA

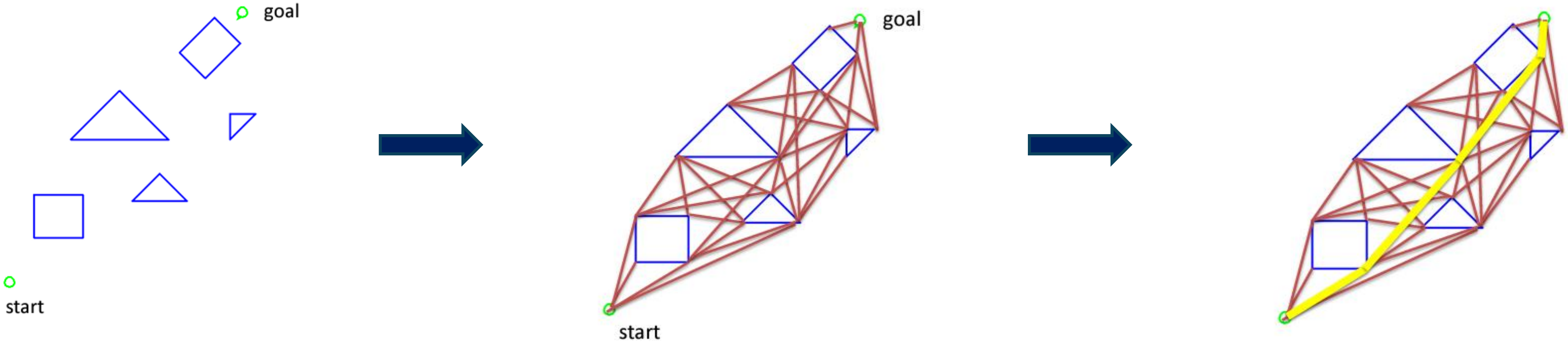
TEB

AI- based path planning

- Miscellaneous:
  - Coverage path planning
  - Potential-field based planning
  - Human-aware path planning

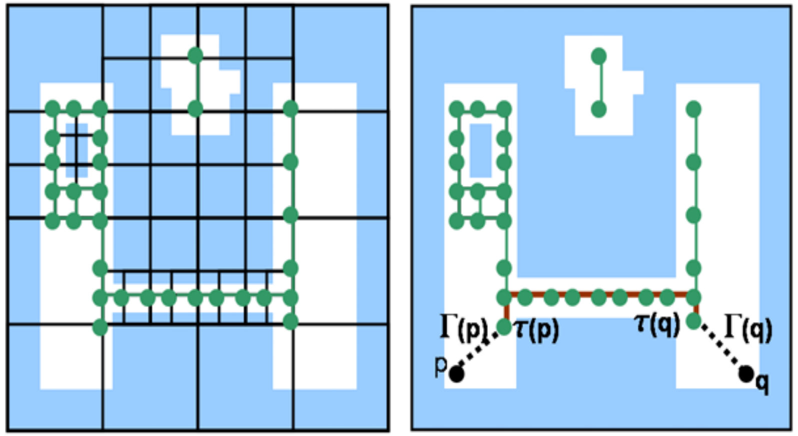
Visibility Graph

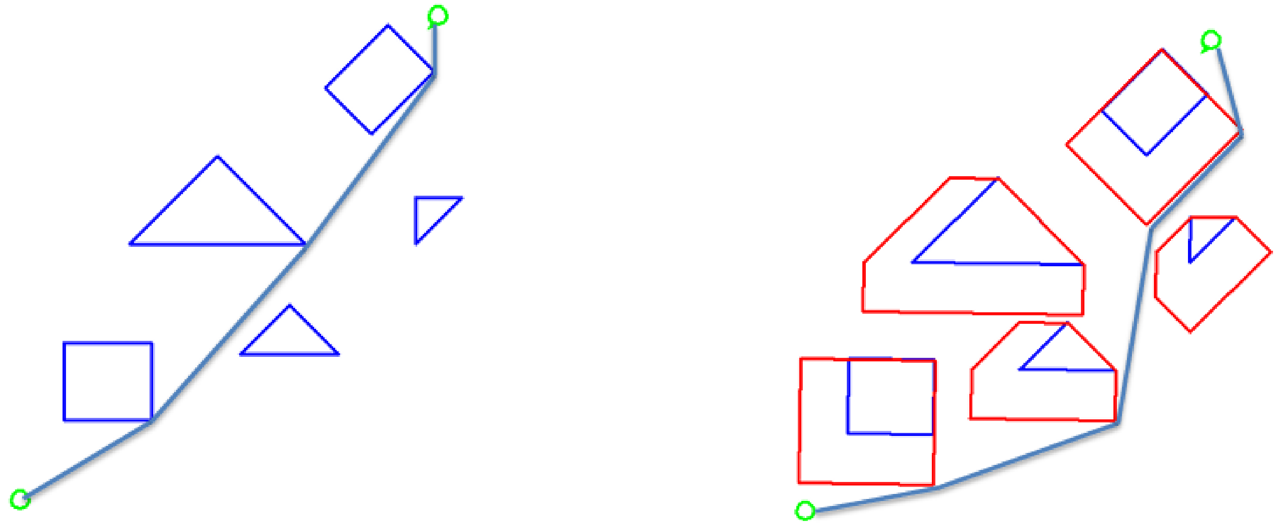
- For graph-based algorithms like A^*, Dijkstra
- Assume the robot is a point in 2D planar space
- Assume obstacles are 2D polygons.
- Create a visibility graph:
    - Nodes are start point, goal point, vertices of obstacles
    - Connect all visible nodes, that is, a straight line unobstructed path between any two nodes.
    - Include all edges of polynomial obstacles.
- Implement any graph search algorithm like A star, Dijkstra from start node to goal node



Minkowski Sum

- What if the size of real workspace is very large?
  - Do Mapping
- Robot is 3D with some volume, may collide with the obstacles
  - Inflate the obstacles and then implement the algorithms, Minkowski sum





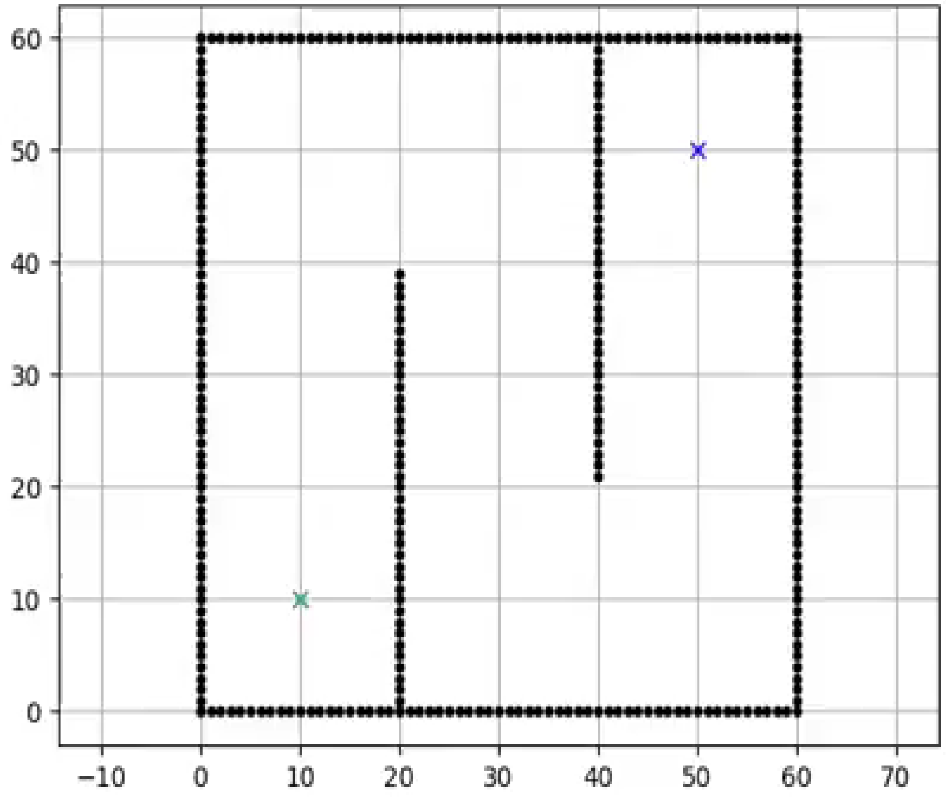
$$P \oplus Q = \{x \mid x = p + q, p \in P, q \in Q\}$$

$$\mathcal{CO}_i = \{\mathcal{Z} \in \mathcal{C} : \mathcal{R}(\mathcal{Z}) \cap \mathcal{O}_i \neq \phi\}$$

$$\mathcal{CO} = \mathcal{O} \oplus (-\mathcal{R})$$

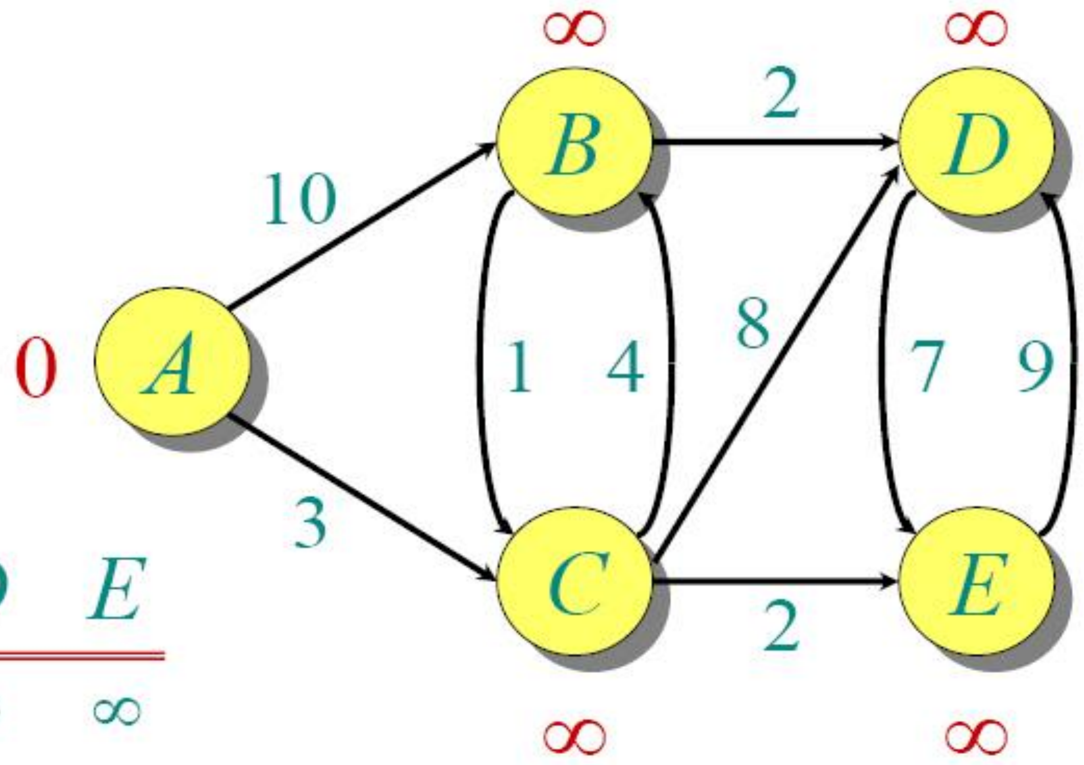
Dijkstra’s Algorithm

- A solution to the single-source shortest path problem in graph theory
- Works on both directed and undirected graphs. All edges must have nonnegative weights.
- Approach: Greedy
- Input: Weighted graph G={E,V} and source vertex v∈V, such that all edge weights are nonnegative
- Output: Shortest paths from a given source vertex v∈V to all other vertices



Dijkstra’s Algorithm

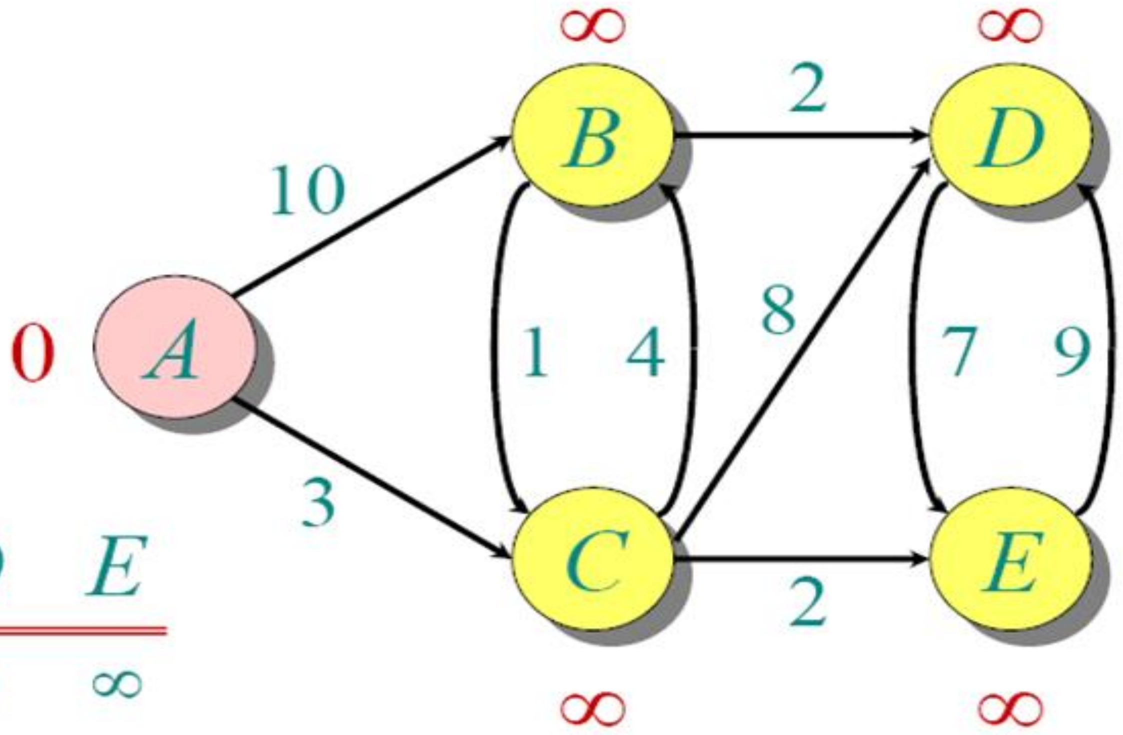
Initialize:



Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞

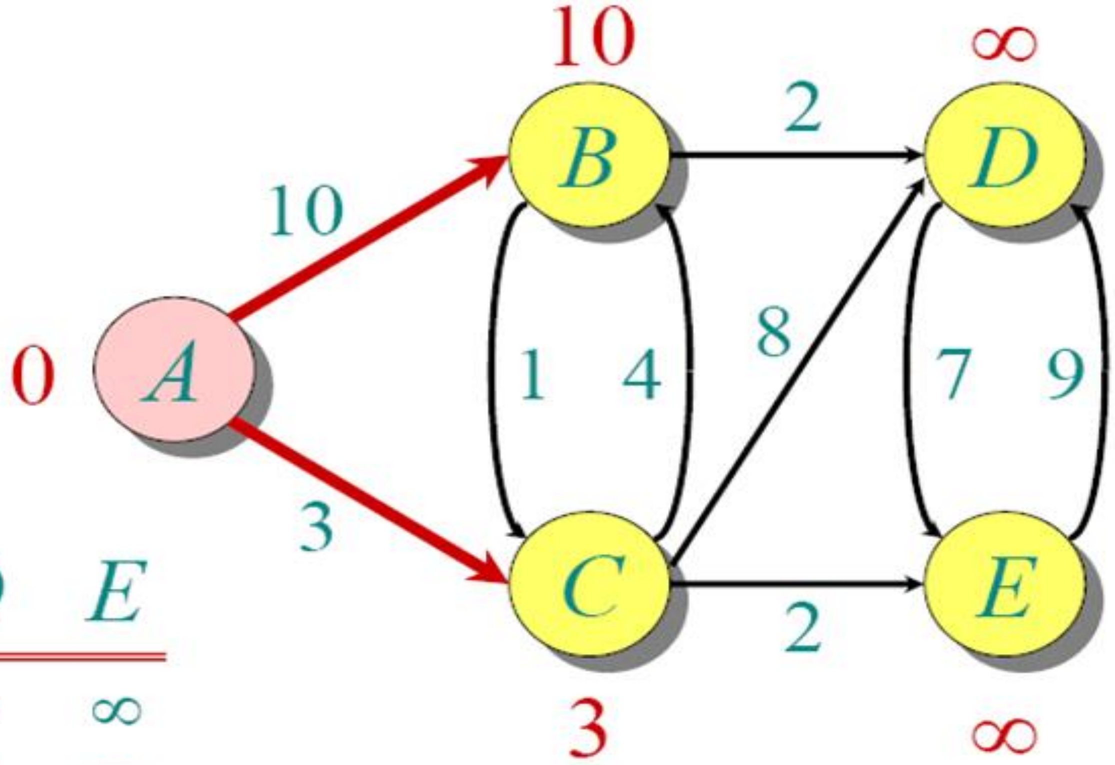
S: {}

Dijkstra’s Algorithm



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

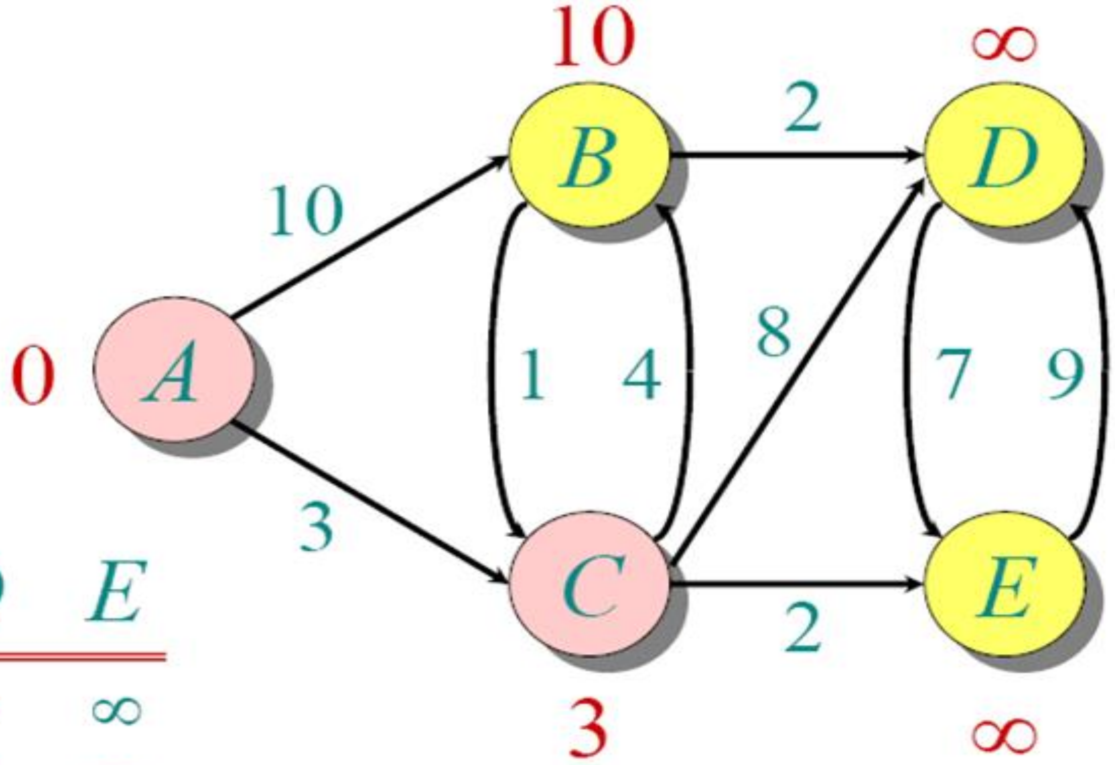
Dijkstra’s Algorithm



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

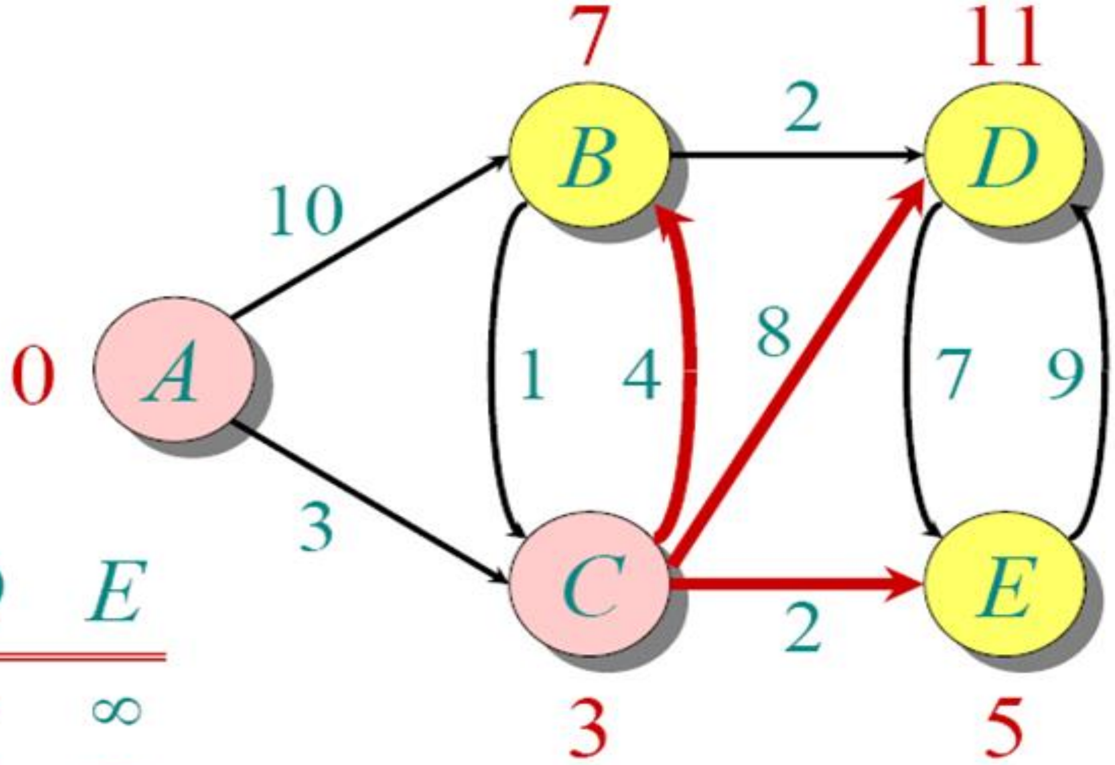
Dijkstra’s Algorithm



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

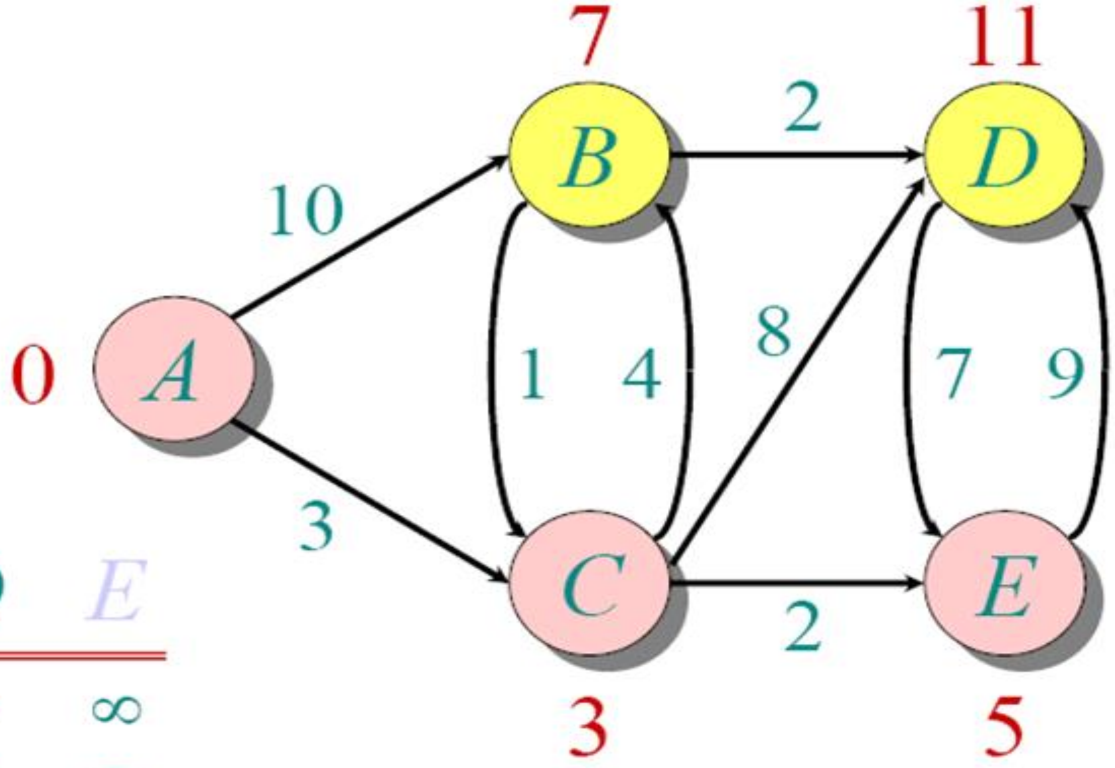
Dijkstra’s Algorithm



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }

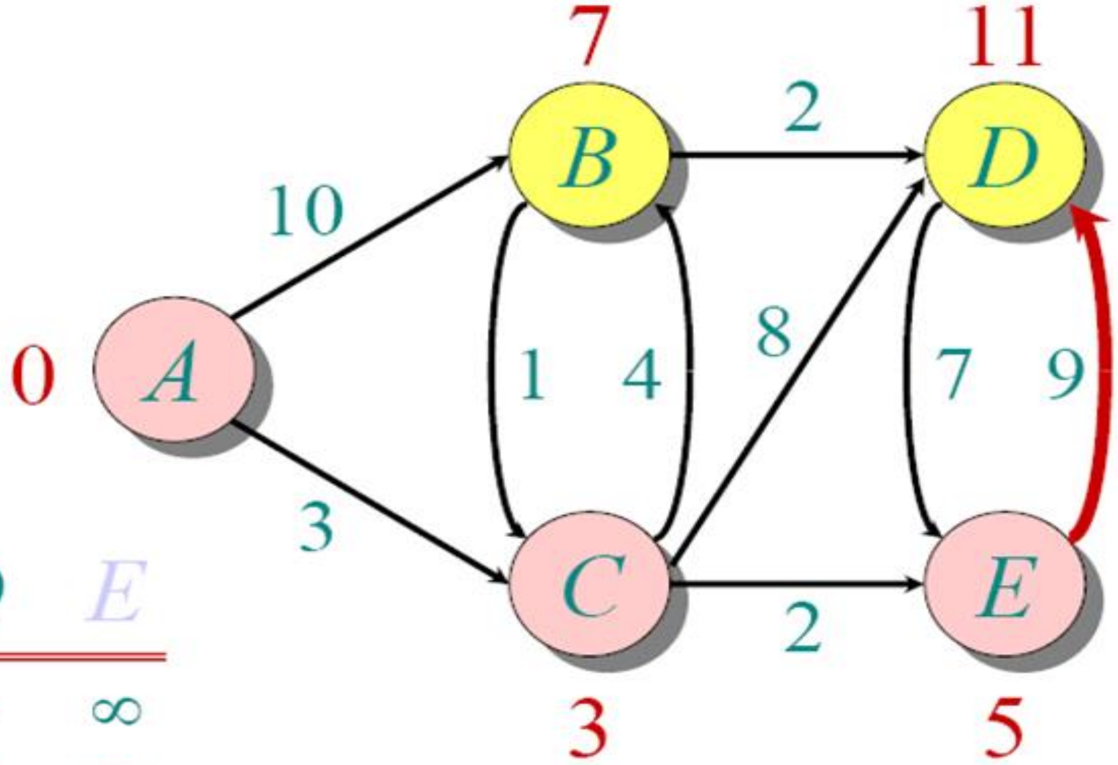
Dijkstra’s Algorithm



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

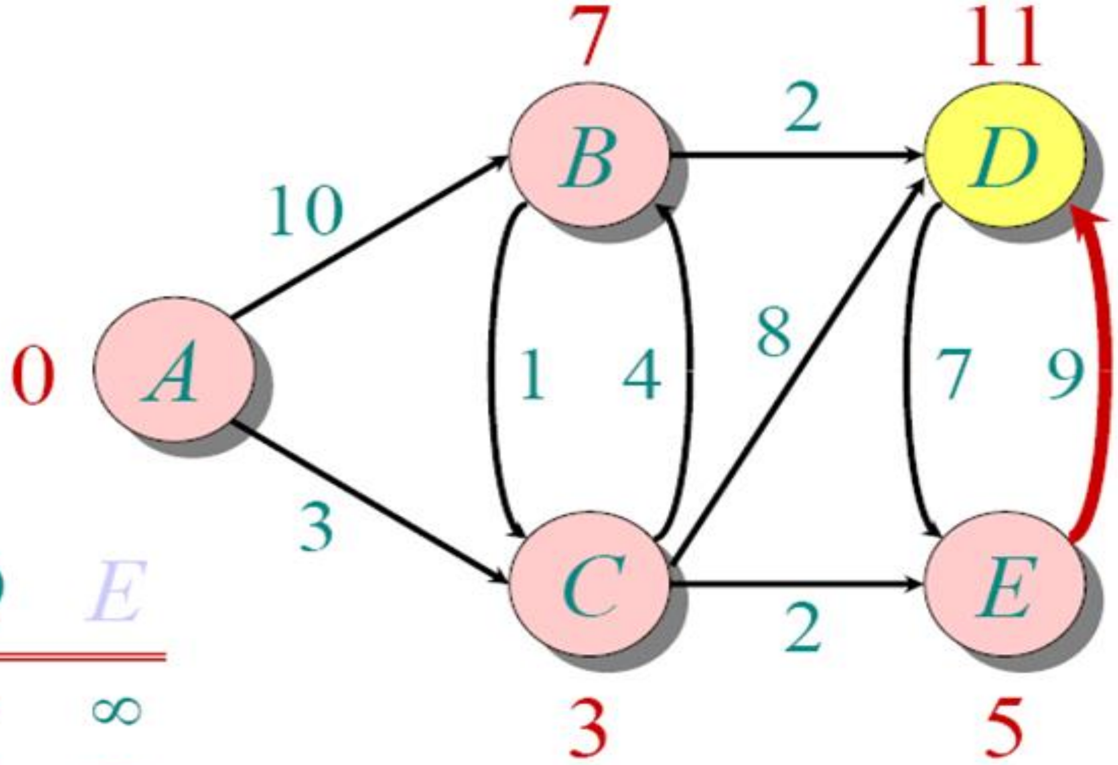
Dijkstra’s Algorithm



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{ A, C, E \}$

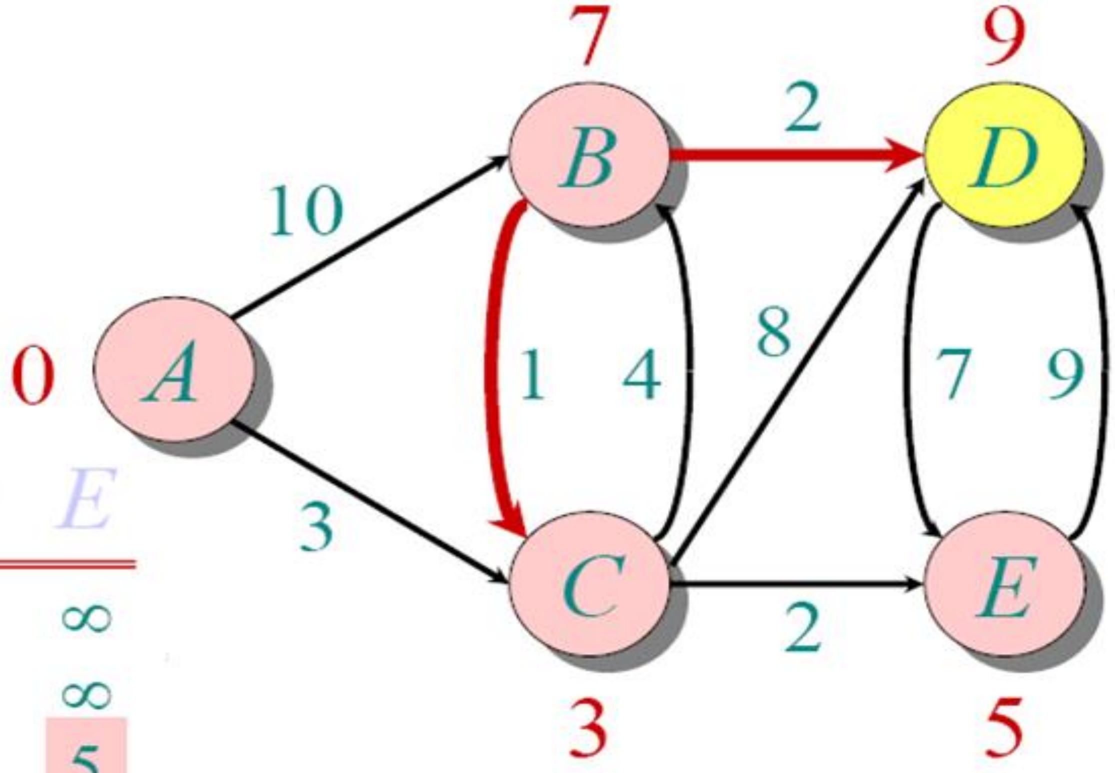
Dijkstra’s Algorithm



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Dijkstra’s Algorithm

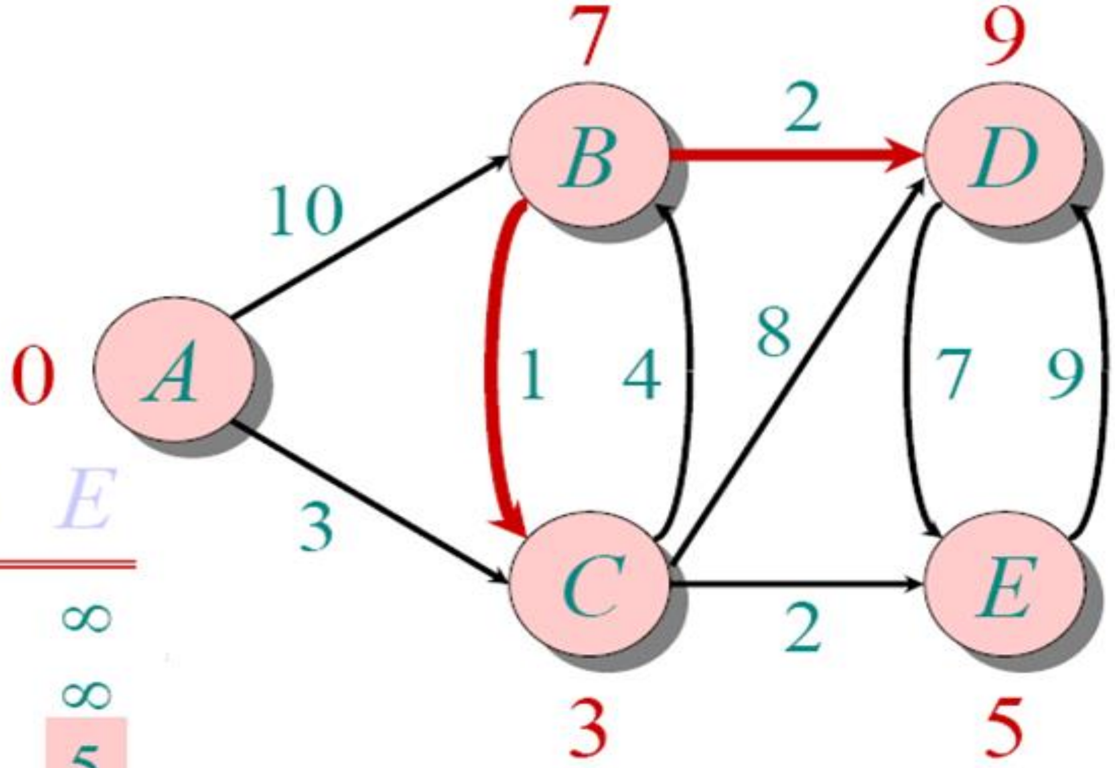


Q:

A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	

S: { *A*, *C*, *E*, *B* }

Dijkstra’s Algorithm

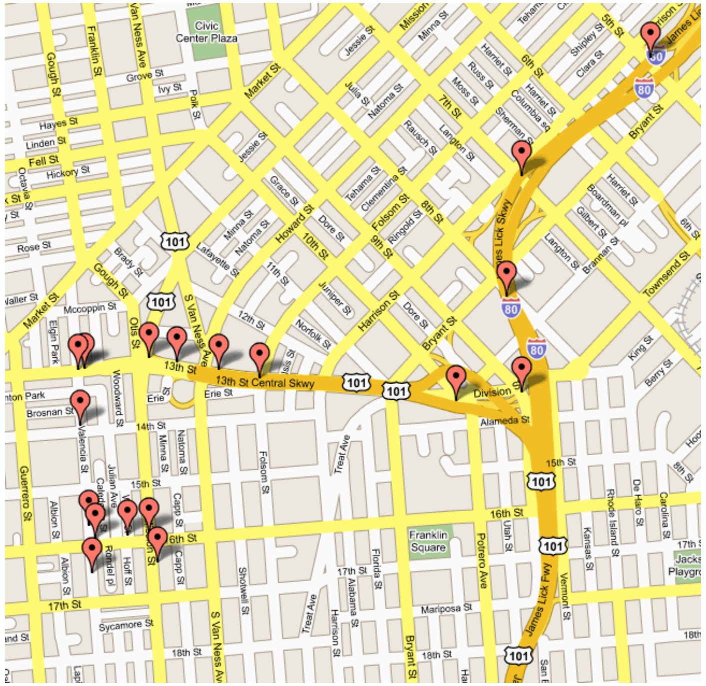


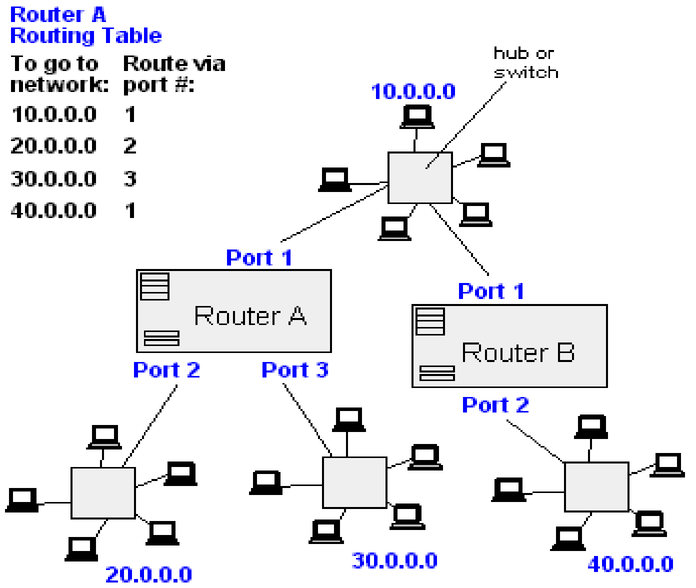
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B, D }

Dijkstra’s Algorithm: Application

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems





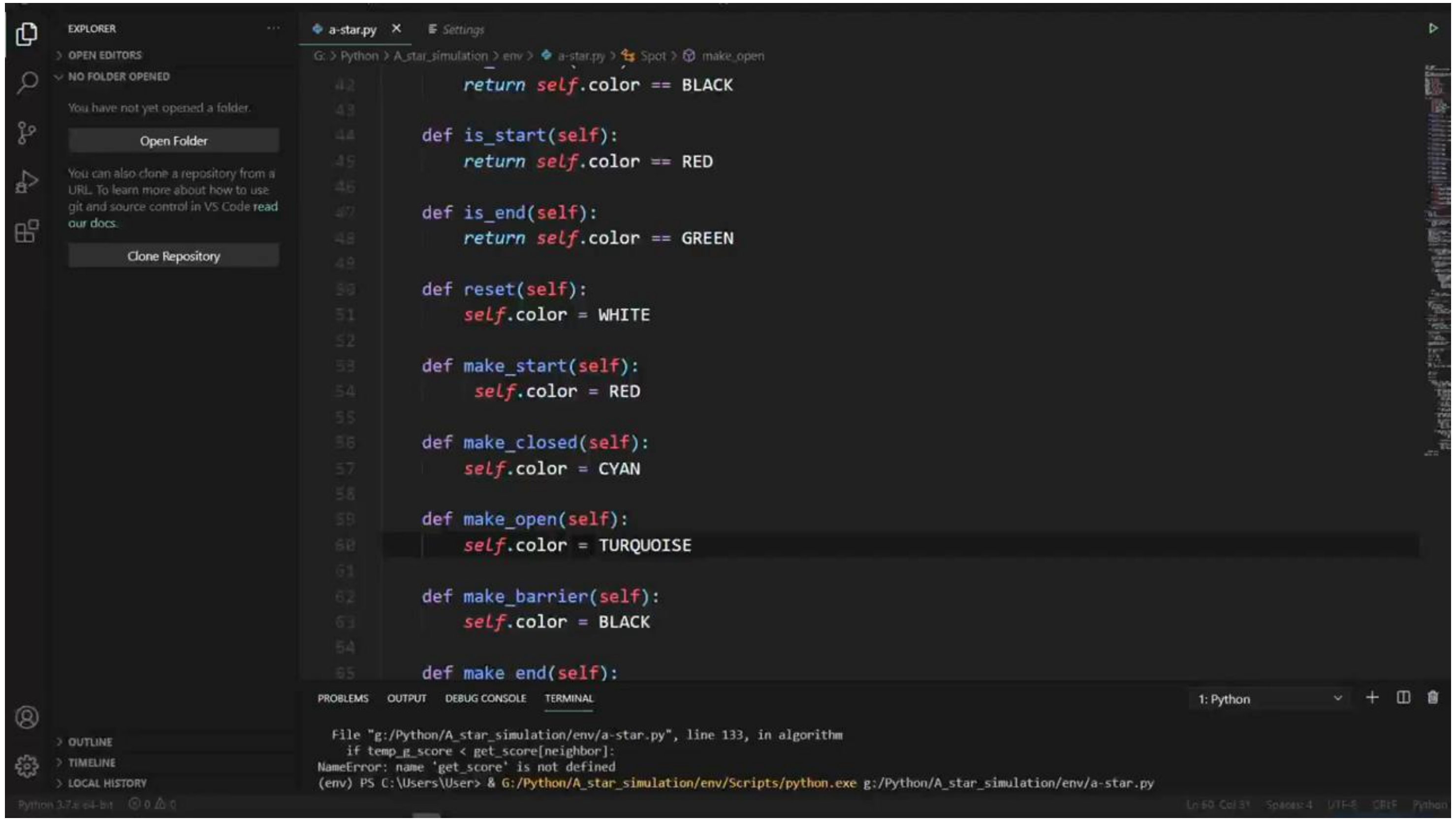
From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.

Router A Routing Table

To go to network:	Route via port #:
10.0.0.0	1
20.0.0.0	2
30.0.0.0	3
40.0.0.0	1

A^* Algorithm

- Popular graph traversal path planning algorithm
- Similar to Dijkstra’s, except that it guides its search towards the most promising states, saving a significant amount of computation time
- Uses the least expensive path and expands it using the function shown below:
    - $f(n) = g(n) + h(n)$
- Applications:
    - Manufacturing industries
    - Manipulators and mobile robots
    - Social navigation



You have not yet opened a folder.

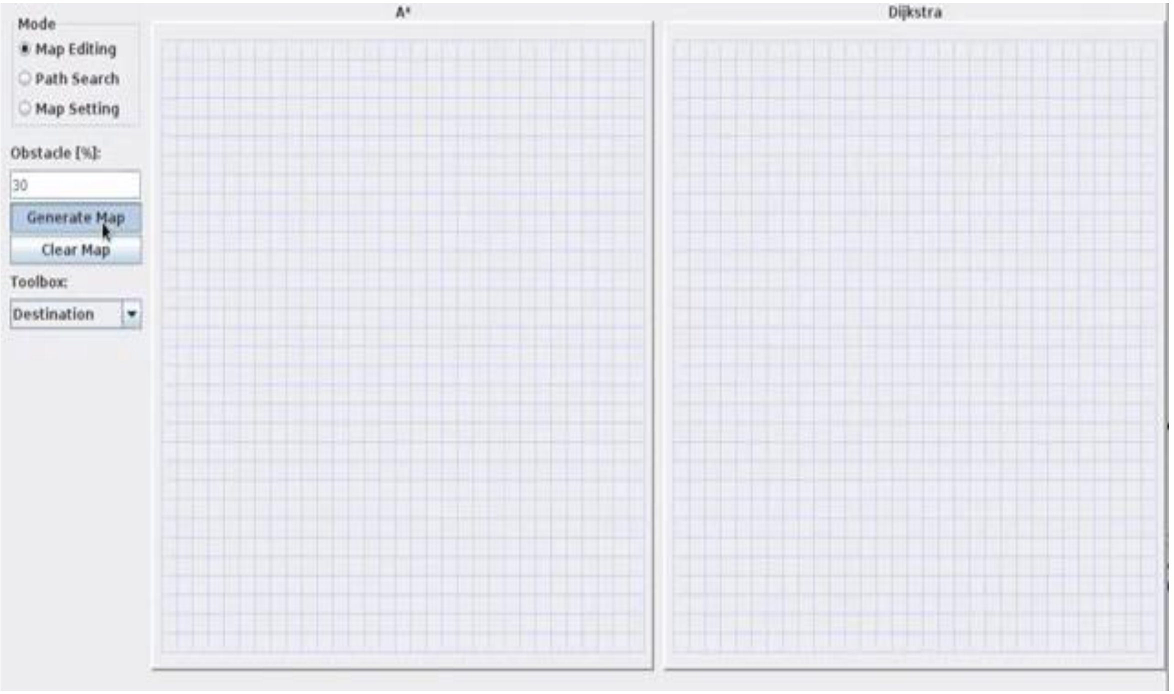
You can also clone a repository from a URL. To learn more about how to use git and source control in VS Code read our docs.

```python
        return self.color == BLACK

    def is_start(self):
        return self.color == RED

    def is_end(self):
        return self.color == GREEN

    def reset(self):
        self.color = WHITE

    def make_start(self):
         self.color = RED

    def make_closed(self):
        self.color = CYAN

    def make_open(self):
        self.color = TURQUOISE

    def make_barrier(self):
        self.color = BLACK

    def make end(self):
```

```
  File "g:/Python/A_star_simulation/env/a-star.py", line 133, in algorithm
    if temp_g_score < get_score[neighbor]:
NameError: name 'get_score' is not defined
(env) PS C:\Users\User> & G:/Python/A_star_simulation/env/Scripts/python.exe g:/Python/A_star_simulation/env/a-star.py
```

A^* Vs Dijkstra

S.No.	Parameters	A^*	Dijkstra
1.	Search algorithm	Best first search/directed search	Greedy best first search/blind search
2.	Time complexity	O(n log n)	O(n^2)
3.	Heuristic function	f(n) = g(n)+h(n)	f(n) = g(n)
4.	Rate of convergence	Faster	Slower



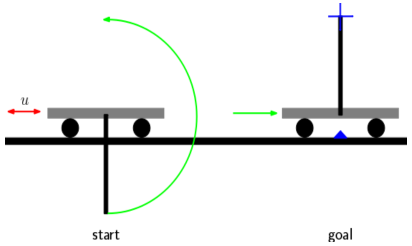
Mode
● Map Editing
○ Path Search
○ Map Setting

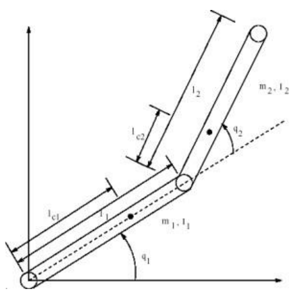
Obstacle [%]: 30

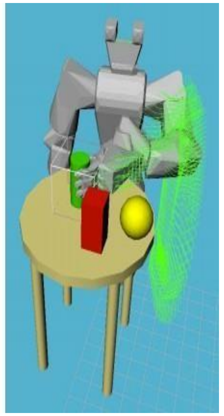
Toolbox: Destination

Motion Planning

- Problem
  - Given start state X_S, goal state X_G
  - Asked for: a sequence of control inputs that leads from start to goal
- Why tricky?
  - Need to avoid obstacles
  - For systems with underactuated dynamics: can't simply move along any coordinate at will
  - E.g., car, helicopter, airplane, but also robot manipulator hitting joint limits







Solve by Nonlinear Optimization for Control?

- Could try by, for example, the following formulation:

$$
\begin{array}{ll}
\min_{u,x} & (x_T - x_G)^\top (x_T - x_G) \\
\text{s.t.} & x_{t+1} = f(x_t, u_t) \quad \forall t \\
 & u_t \in \mathcal{U}_t \\
 & x_t \in \mathcal{X}_t \\
 & x_0 = x_S
\end{array}
$$

- Or, with constraints:

$$
\begin{array}{ll}
\min_{u,x} & \|u\| \\
\text{s.t.} & x_{t+1} = f(x_t, u_t) \quad \forall t \\
 & u_t \in \mathcal{U}_t \\
 & x_t \in \mathcal{X}_t \\
 & x_0 = x_S \\
 & X_T = x_G
\end{array}
$$

- For more complicated problems with longer horizons, often get stuck in local maxima that don’t reach the goal

Motion Planning: Outline

- Configuration Space
- Probabilistic Roadmap
  - Boundary Value Problem
  - Sampling
  - Collision checking
- Rapidly-exploring Random Trees (RRTs)
- Smoothing

Configuration Space (C-Space)

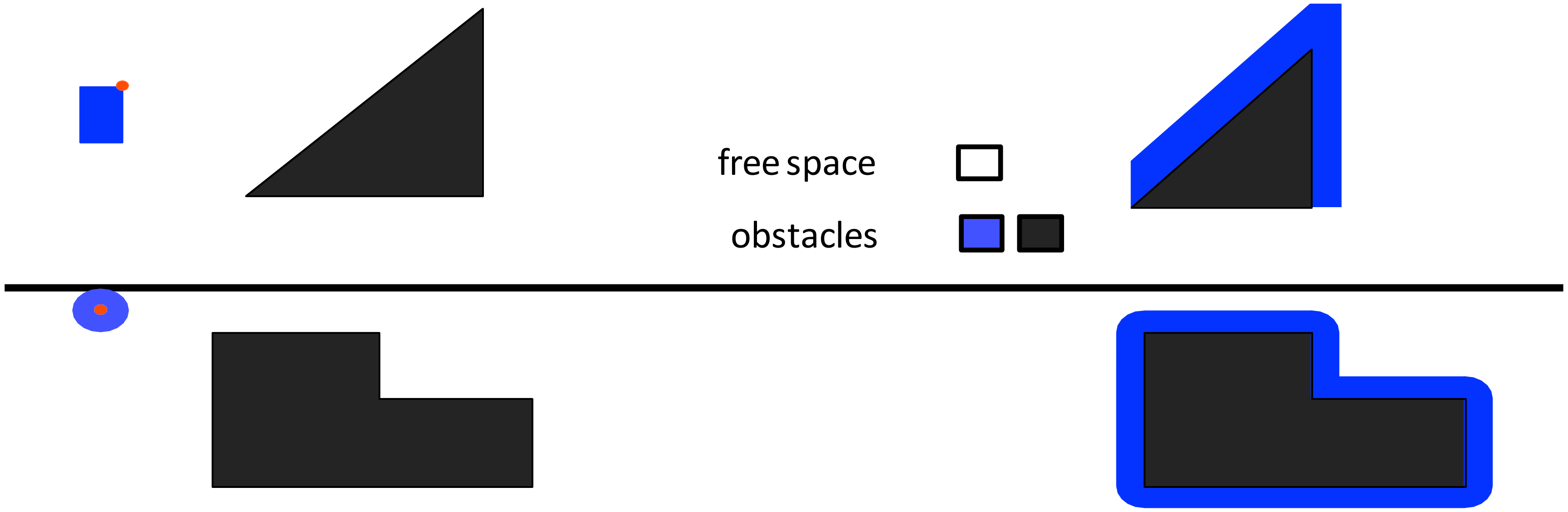
= { x | x is a pose of the robot}

○ obstacles → configuration space obstacles

Workspace

Configuration Space

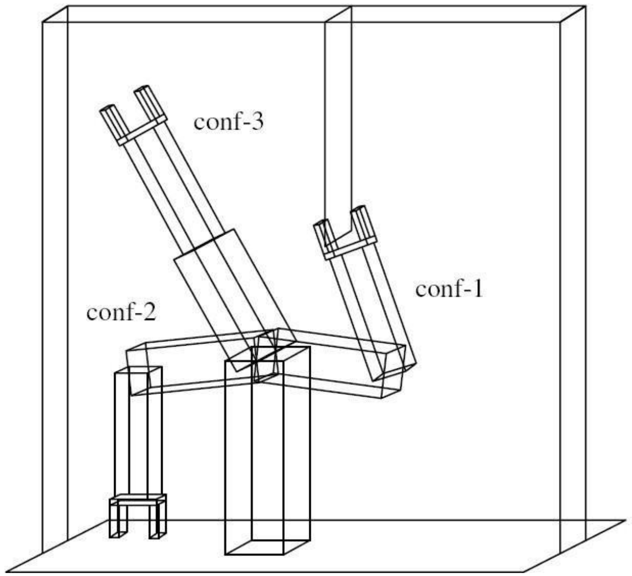
(2 DOF: translation only, no rotation)

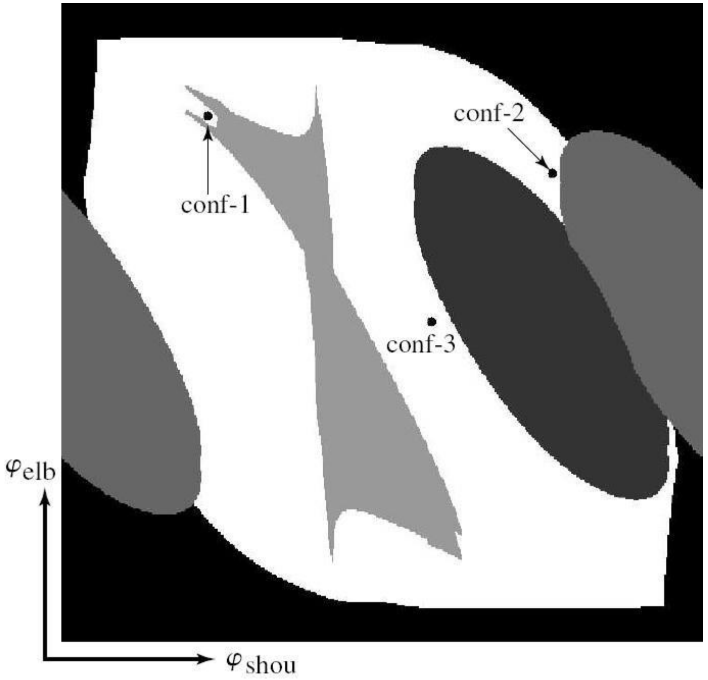


free space ☐

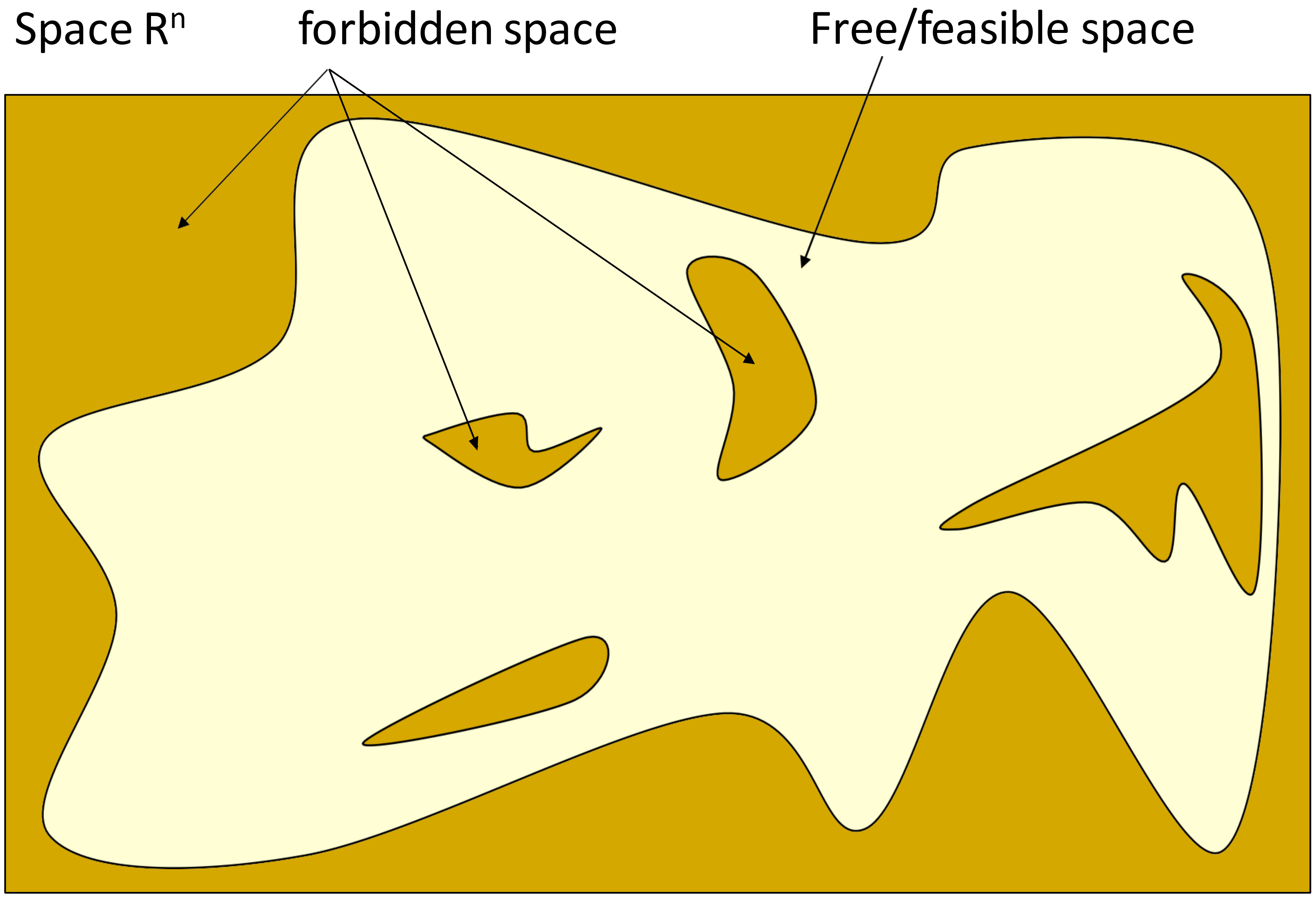
obstacles 🟦 ■

Motion planning





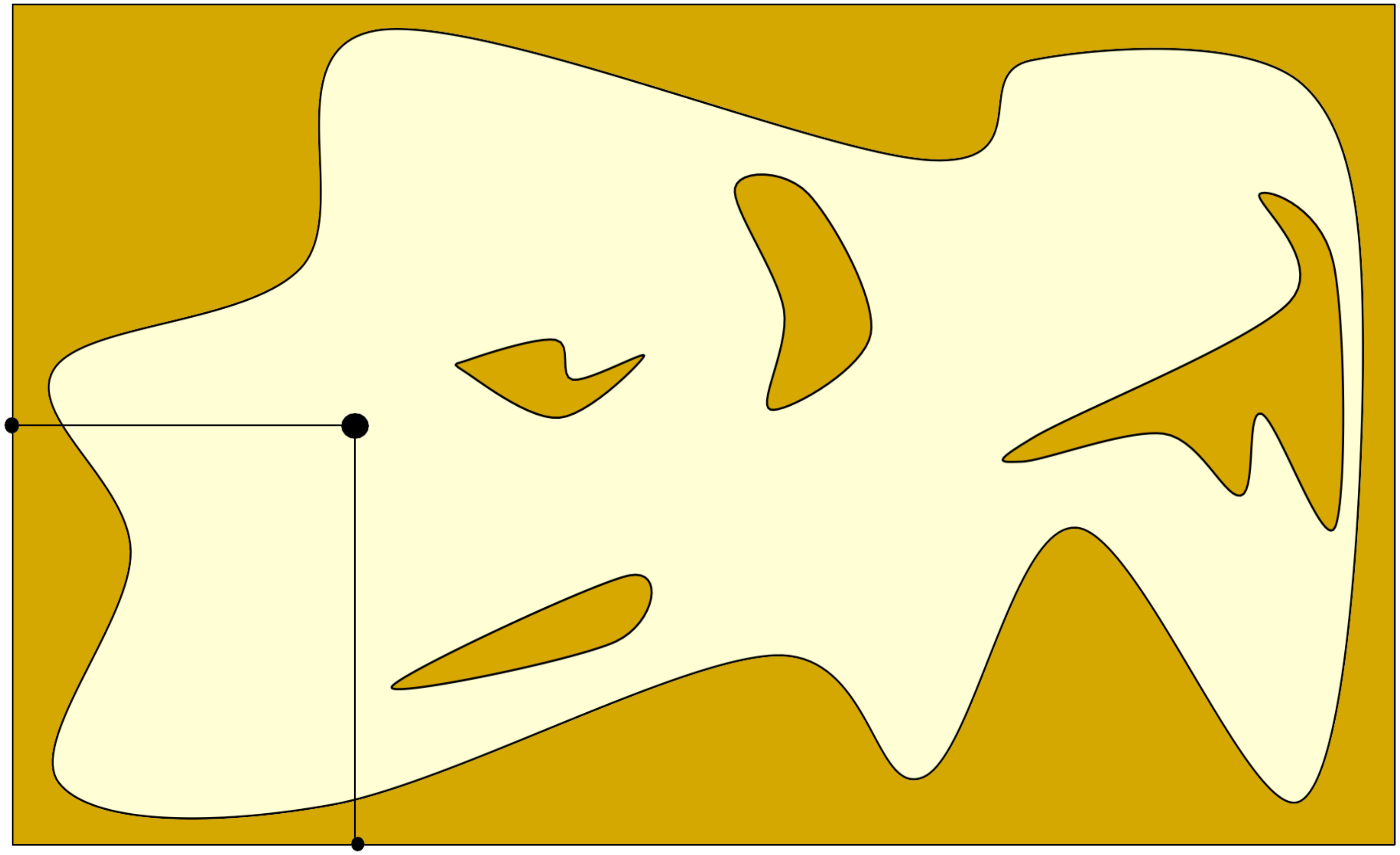
Probabilistic Roadmap (PRM)



Space R^n forbidden space Free/feasible space

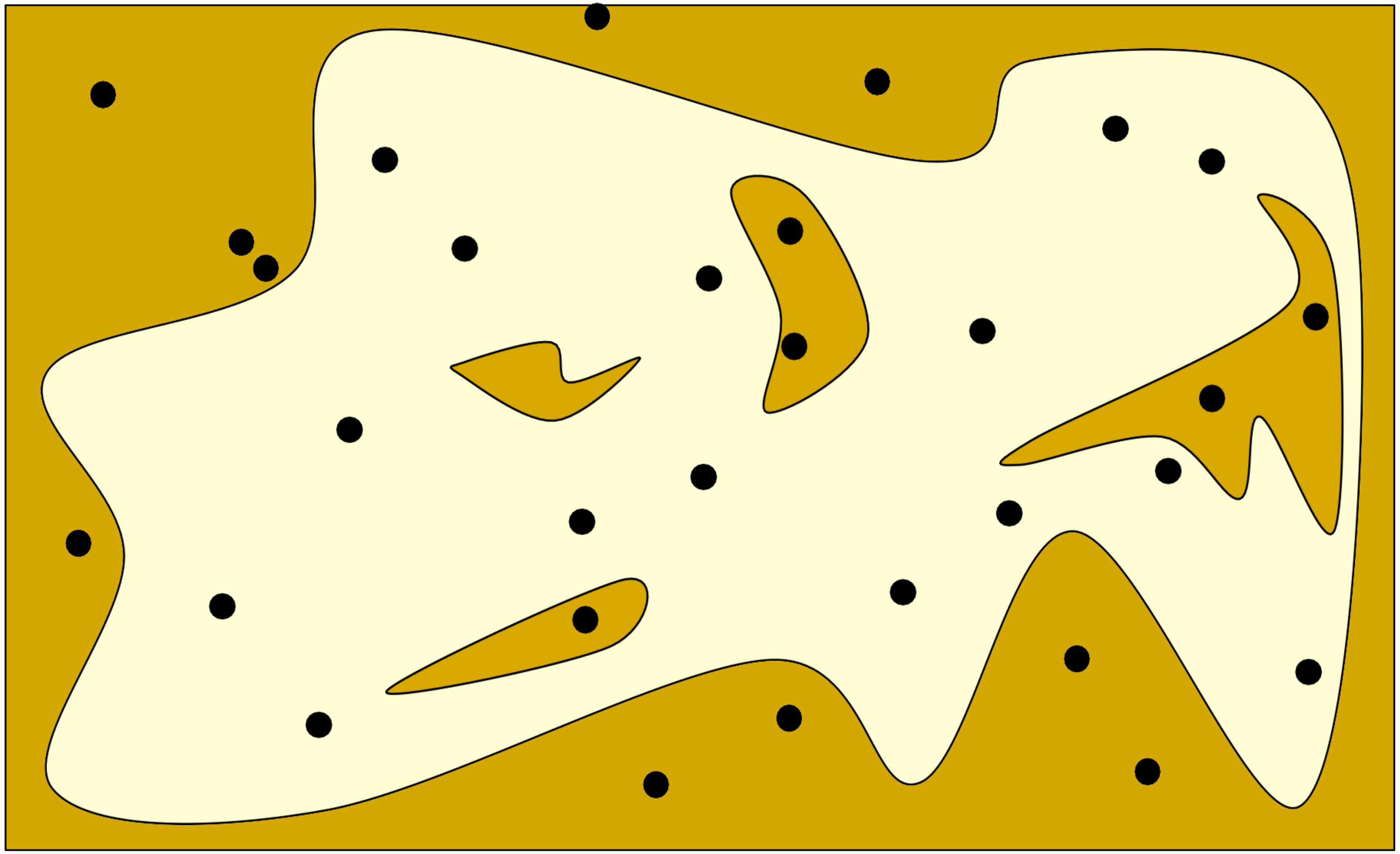
Probabilistic Roadmap (PRM)

Configurations are sampled by picking coordinates at random



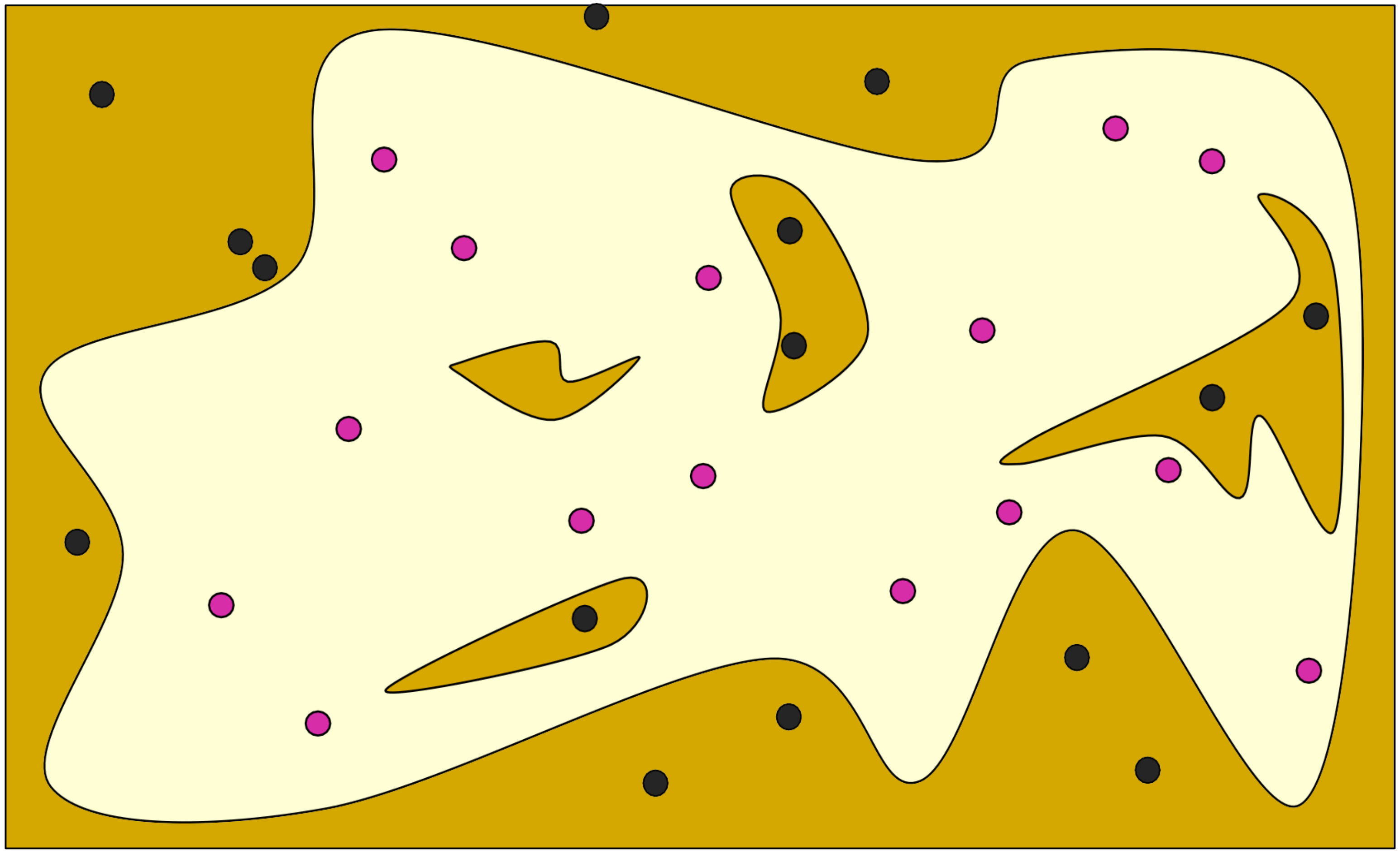
Probabilistic Roadmap (PRM)

Configurations are sampled by picking coordinates at random



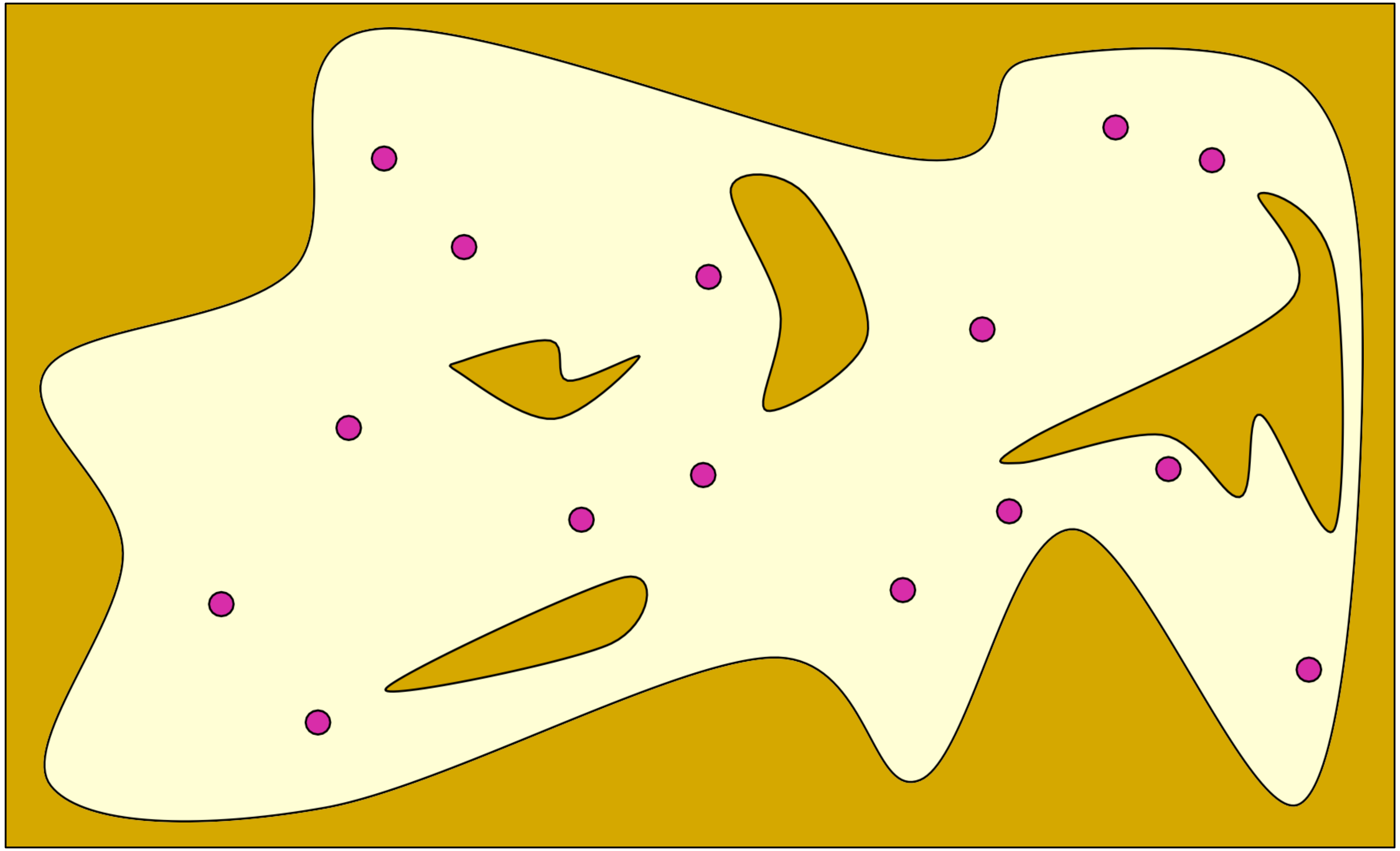
Probabilistic Roadmap (PRM)

Sampled configurations are tested for collision



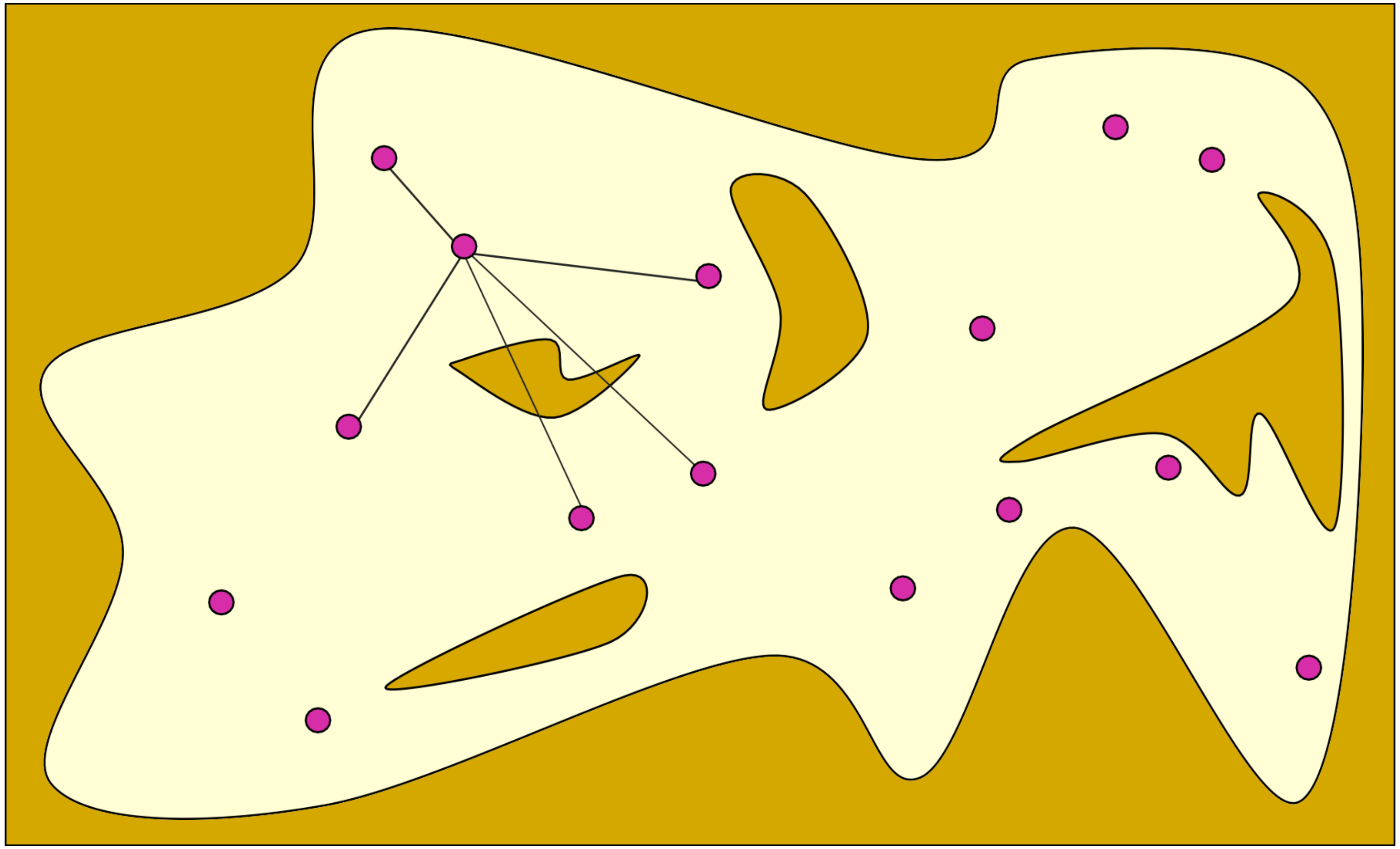
Probabilistic Roadmap (PRM)

The collision-free configurations are retained as milestones



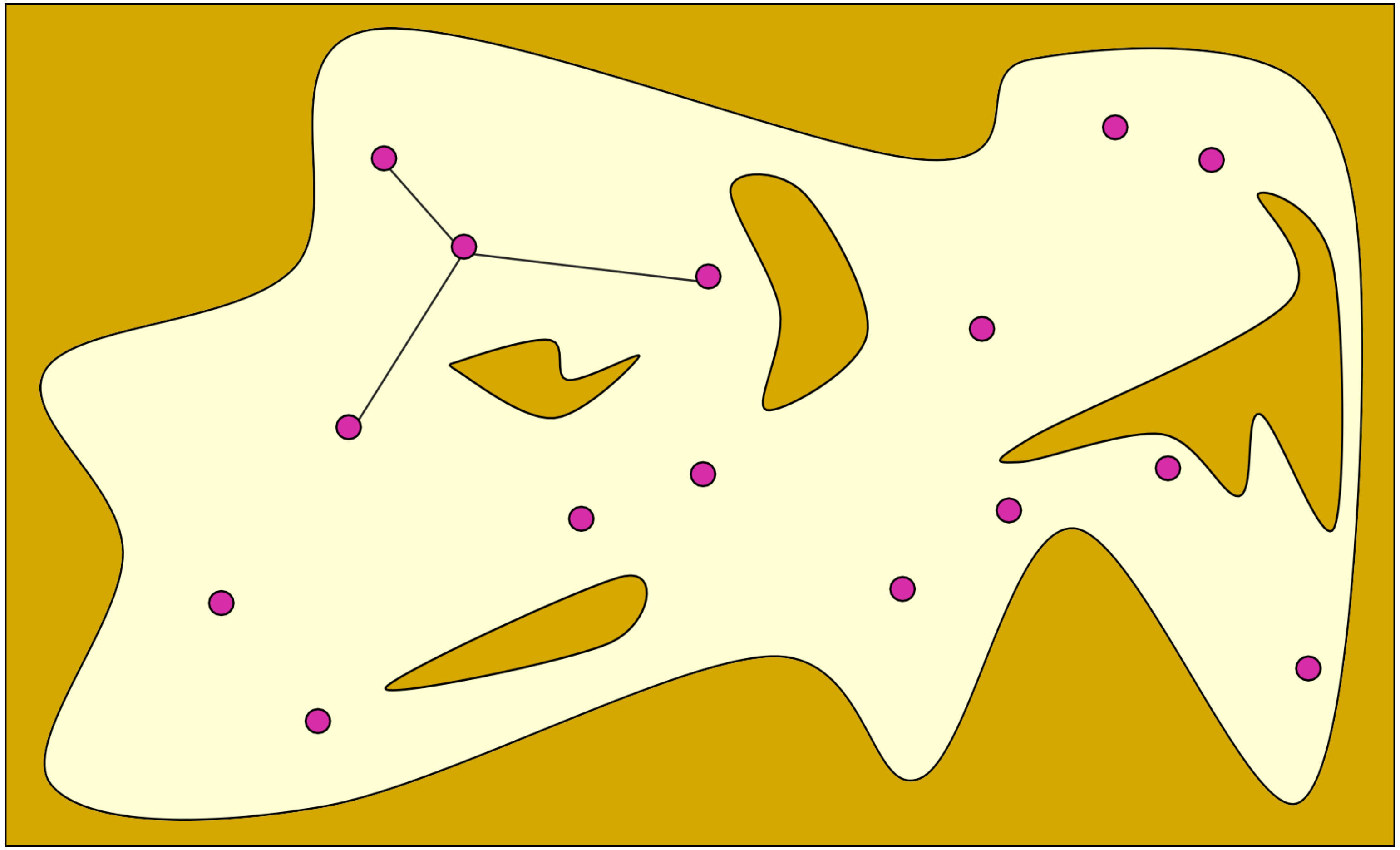
Probabilistic Roadmap (PRM)

Each milestone is linked by straight paths to its nearest neighbors



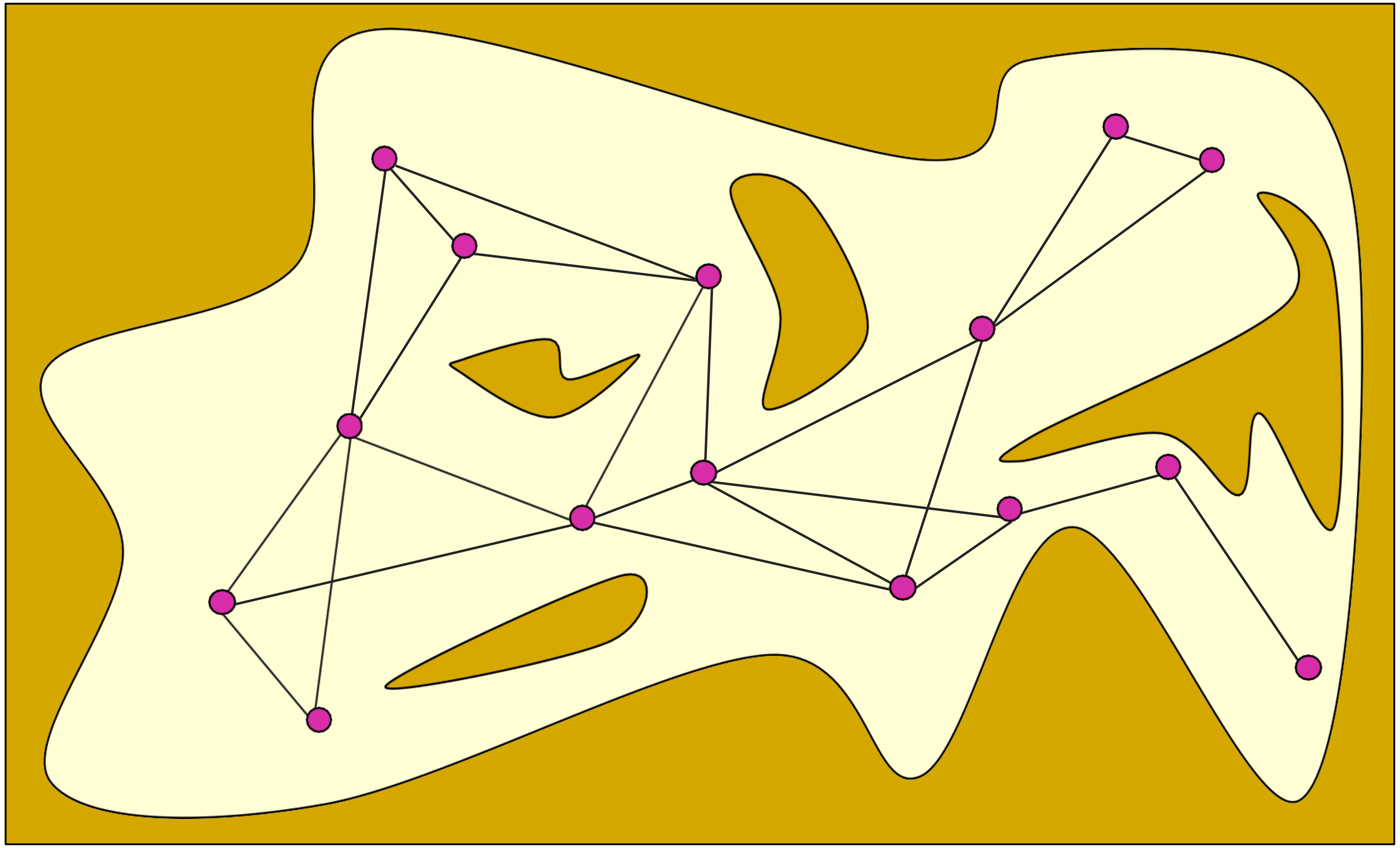
Probabilistic Roadmap (PRM)

Each milestone is linked by straight paths to its nearest neighbors



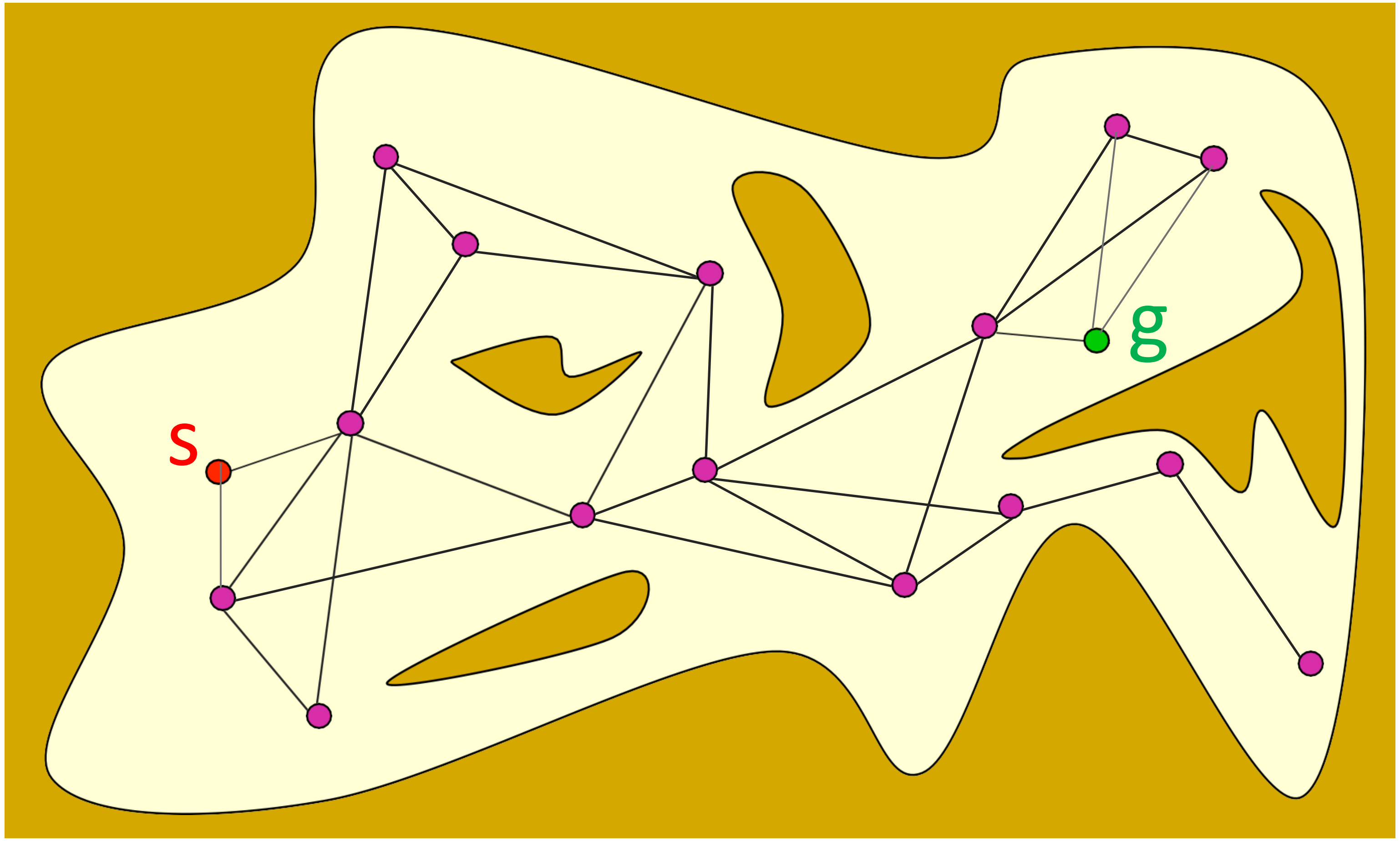
Probabilistic Roadmap (PRM)

The collision-free links are retained as local paths to form the PRM



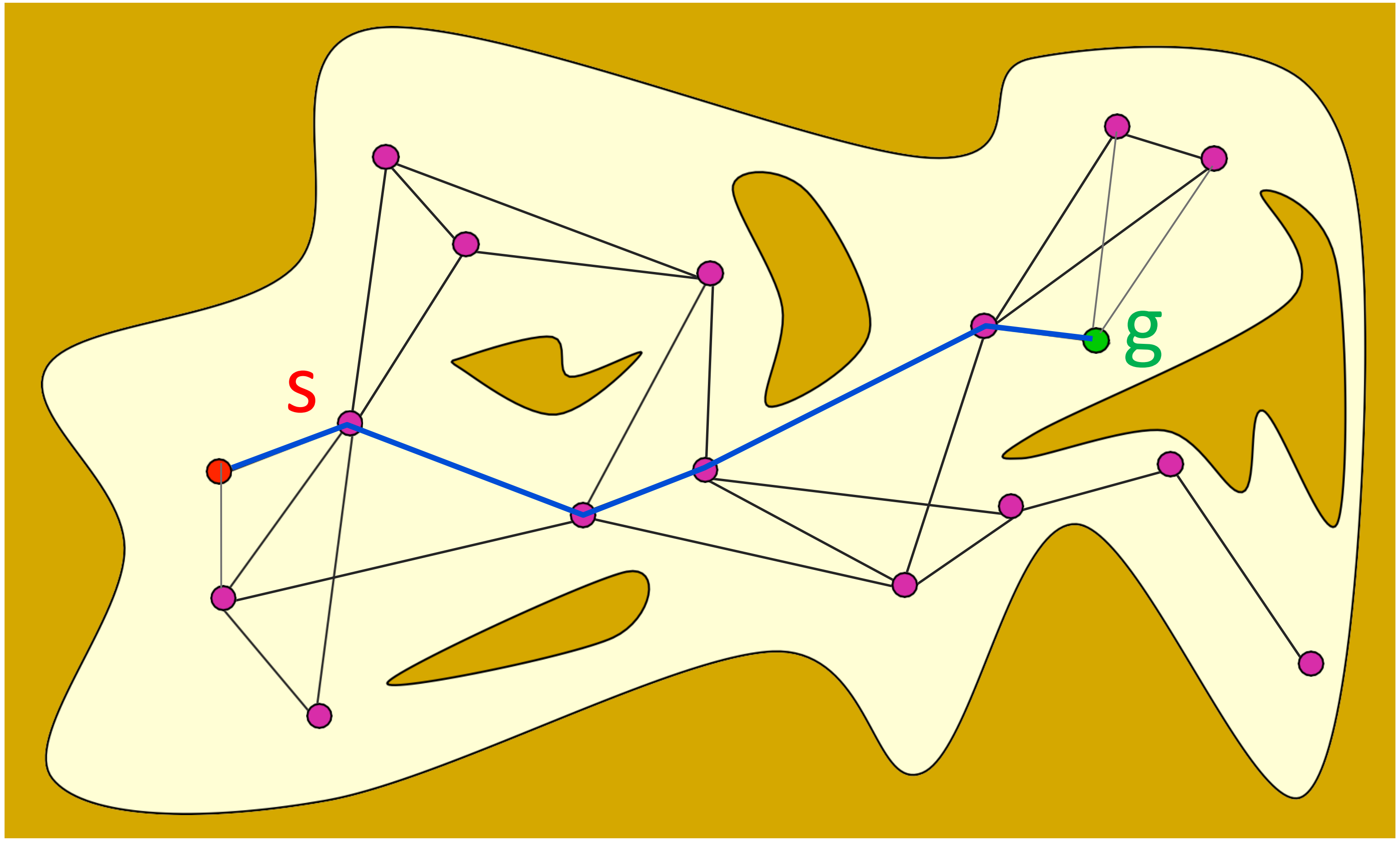
Probabilistic Roadmap (PRM)

The start and goal configurations are included as milestones



Probabilistic Roadmap (PRM)

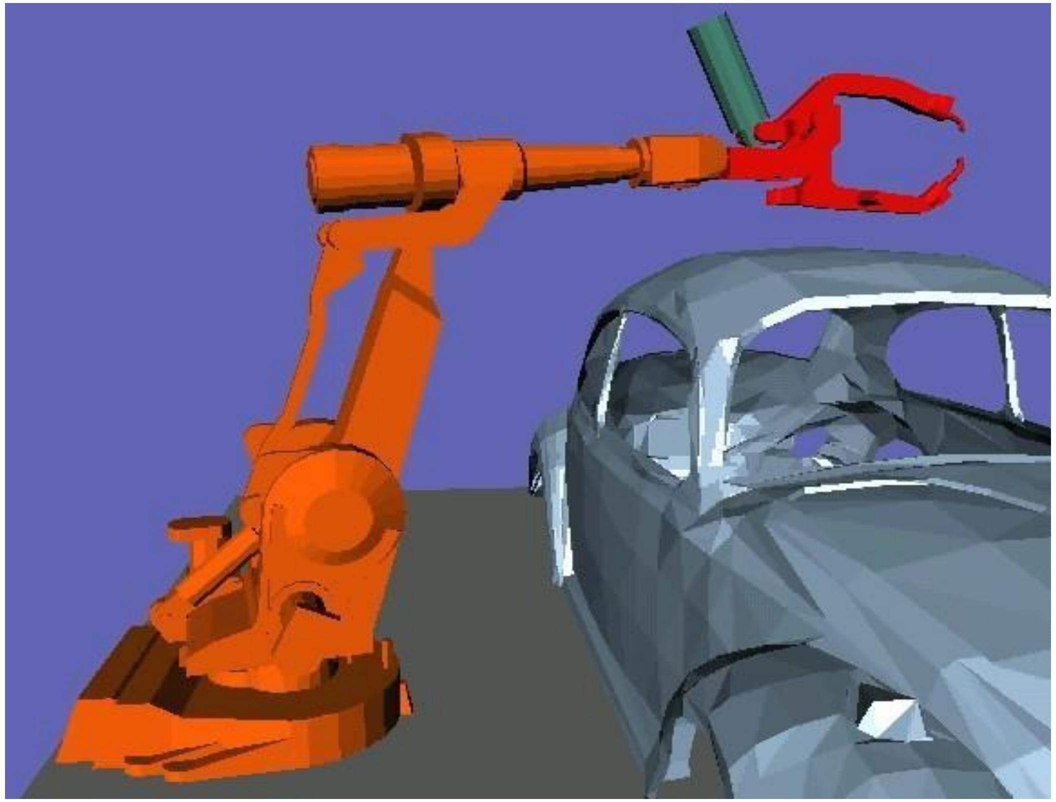
The start and goal configurations are included as milestones



Probabilistic Roadmap

- Initialize set of points with xS and xG
- Randomly sample points in configuration space
- Connect nearby points if they can be reached from each other
- Find path from X_s to X_G in the graph
  - alternatively: keep track of connected components incrementally, and declare success when X_s and X_G are in same connected component

Example



PRM: Challenges

- Connecting neighboring points: Generally requires solving a Boundary Value Problem:

$$\begin{array}{ll} \min_{u,x} & \|u\| \\ \text{s.t.} & x_{t+1} = f(x_t, u_t) \quad \forall t \\ & u_t \in \mathcal{U}_t \\ & x_t \in \mathcal{X}_t \\ & x_0 = x_S \\ & X_T = x_G \end{array}$$

Typically solved without collision checking; later verified if valid by collision checking

- Collision checking:
  - Often takes majority of time in applications (see Lavalle)

PRM’s Pros and Cons

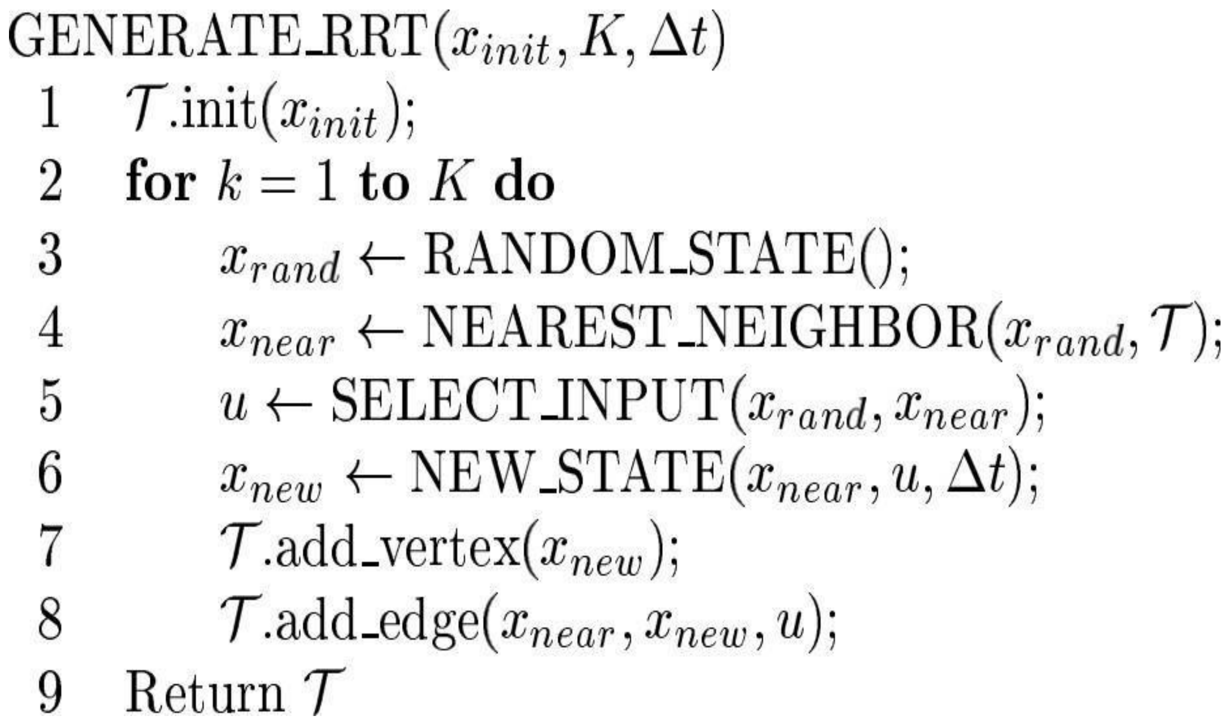
- Pro:
  - Probabilistically complete: i.e., with probability one, if run for long enough the graph will contain a solution path if one exists

- Cons:
  - Required to solve boundary value problem
  - Build graph over state space but no particular focus on generating a path

Rapidly exploring Random Trees

- Basic idea:
  - Build up a tree through generating “next states” in the tree by executing random controls
  - However: not exactly above to ensure good coverage

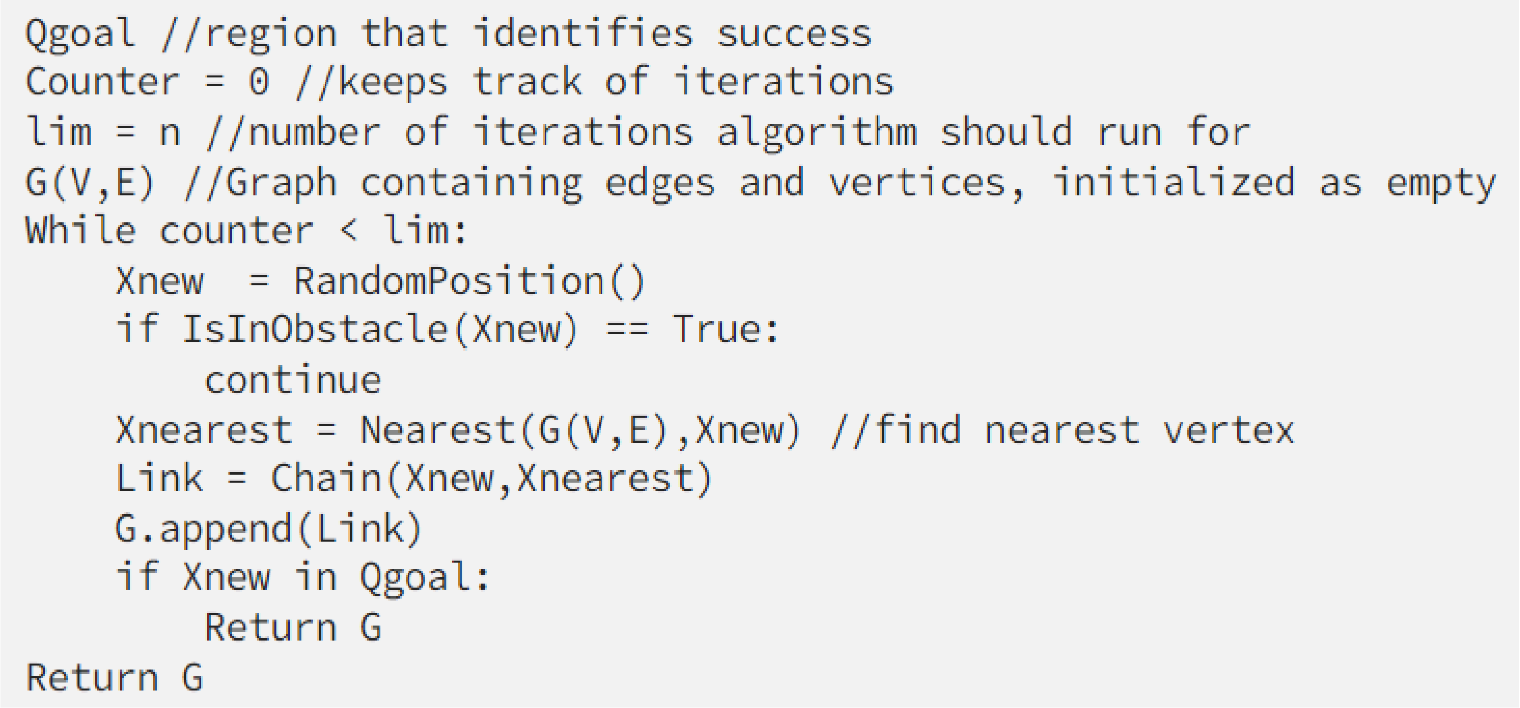
Rapidly-exploring Random Trees (RRT)



```
GENERATE_RRT($x_{init}, K, \Delta t$)
    $\mathcal{T}$.init($x_{init}$);
    for $k = 1$ to $K$ do
        $x_{rand} \leftarrow$ RANDOM_STATE();
        $x_{near} \leftarrow$ NEAREST_NEIGHBOR($x_{rand}, \mathcal{T}$);
        $u \leftarrow$ SELECT_INPUT($x_{rand}, x_{near}$);
        $x_{new} \leftarrow$ NEW_STATE($x_{near}, u, \Delta t$);
        $\mathcal{T}$.add_vertex($x_{new}$);
        $\mathcal{T}$.add_edge($x_{near}, x_{new}, u$);
    Return $\mathcal{T}$
```

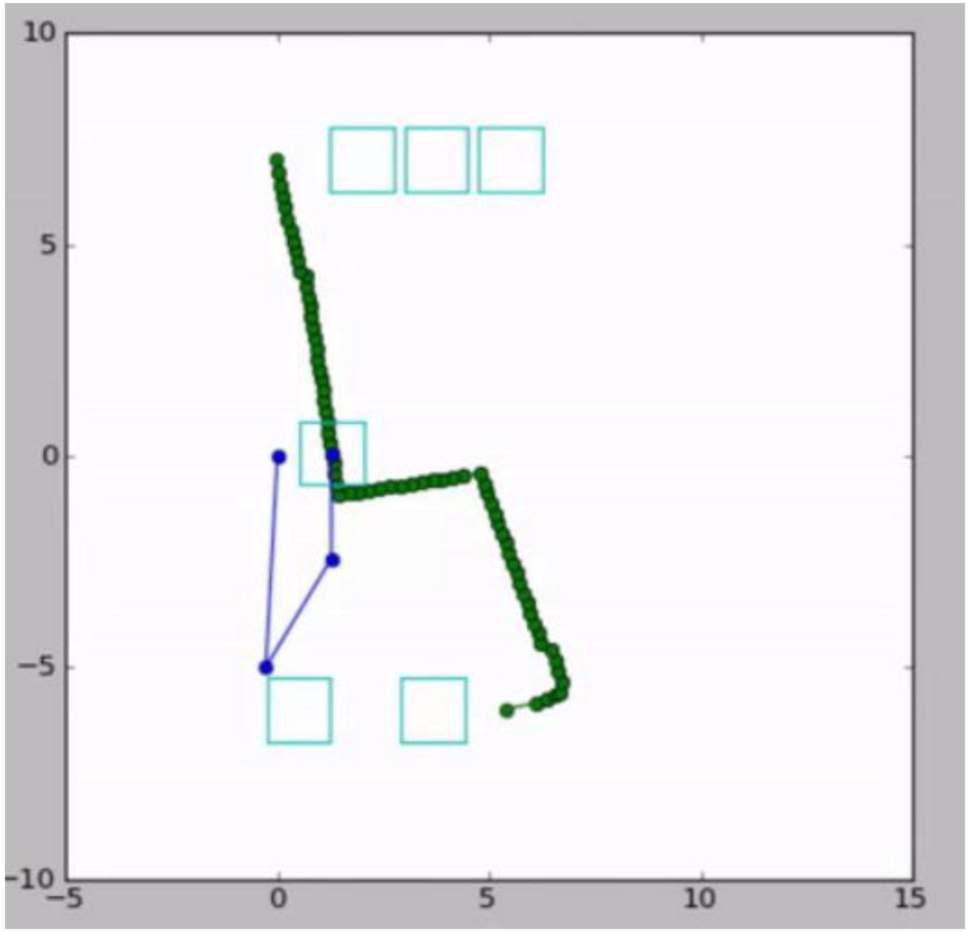
- RANDOM_STATE(): often uniformly at random over space with probability 99%, and the goal state with probability 1%, this ensures it attempts to connect to goal semi-regularly

RRT Pseudo code

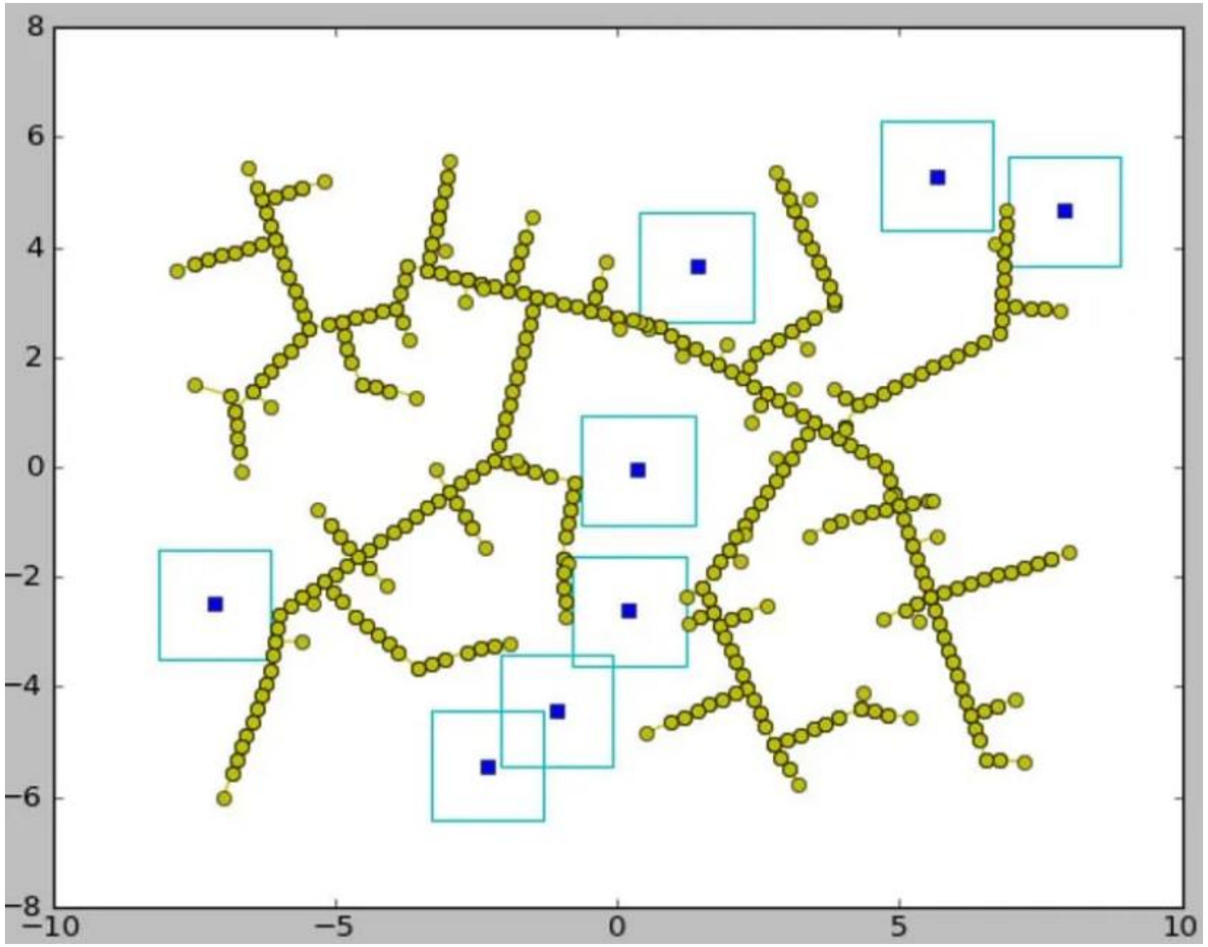


```
Qgoal //region that identifies success
Counter = 0 //keeps track of iterations
lim = n //number of iterations algorithm should run for
G(V,E) //Graph containing edges and vertices, initialized as empty
While counter < lim:
    Xnew  = RandomPosition()
    if IsInObstacle(Xnew) == True:
        continue
    Xnearest = Nearest(G(V,E),Xnew) //find nearest vertex
    Link = Chain(Xnew,Xnearest)
    G.append(Link)
    if Xnew in Qgoal:
        Return G
Return G
```

RRT Path: Example



RRT Graph: Example



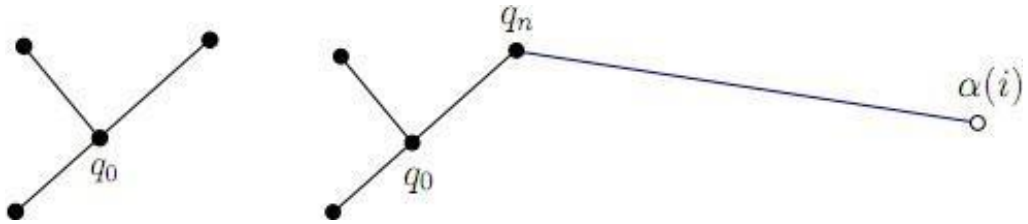
RRT Practicalities

- NEAREST_NEIGHBOR(x_{rand}, T): need to find (approximate) nearest neighbor efficiently
  - KD Trees data structure

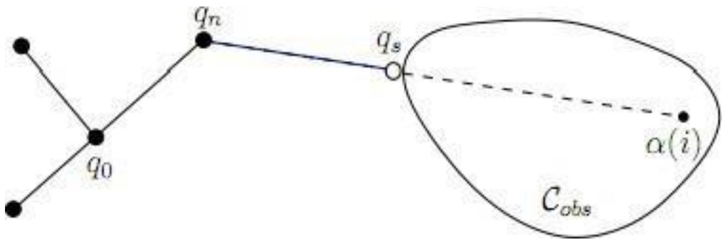
- SELECT_INPUT(x_{rand}, x_{near})
  - Two point boundary value problem
    - If too hard to solve, often just select best out of a set of control sequences. This set could be random, or some well chosen set of primitives

RRT Extension

- No obstacles, holonomic:



- With obstacles, holonomic:



- Non-holonomic: approximately (sometimes as approximate as picking best of a few random control sequences) solve two-point boundary value problem

Growing RRT

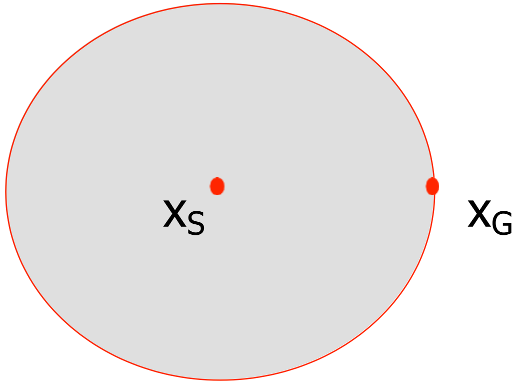
45 iterations

390 iterations

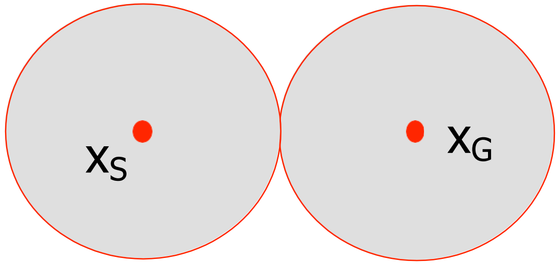
Demo: http://en.wikipedia.org/wiki/File:Rapidly-exploring_Random_Tree_(RRT)_500x373.gif

Bi-directional RRT

- Volume swept out by unidirectional RRT:



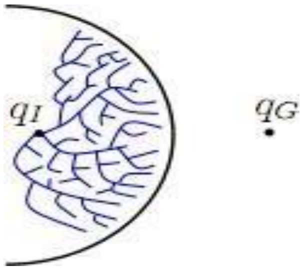
- Volume swept out by bi-directional RRT:



- Difference becomes even more pronounced in higher dimensions

Resolution-Complete RRT (RC-RRT)

- Issue: nearest points chosen for expansion are (too) often the ones stuck behind an obstacle



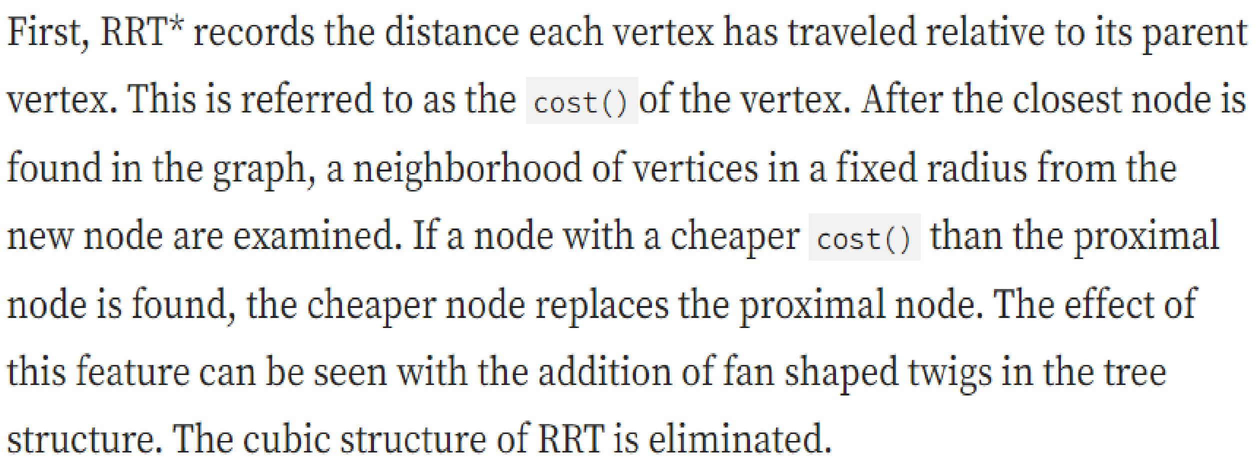
- RC-RRT solution:
  - Choose a maximum number of times, m, you are willing to try to expand each node
  - For each node in the tree, keep track of its Constraint Violation Frequency (CVF)
  - Initialize CVF to zero when node is added to tree
  - Whenever an expansion from the node is unsuccessful (e.g., per hitting an obstacle):
    - Increase CVF of that node by 1
    - Increase CVF of its parent node by 1/m, its grandparent 1/m2, …
  - When a node is selected for expansion, skip over it with probability CVF/m

RRT*

Two differences from RRT:

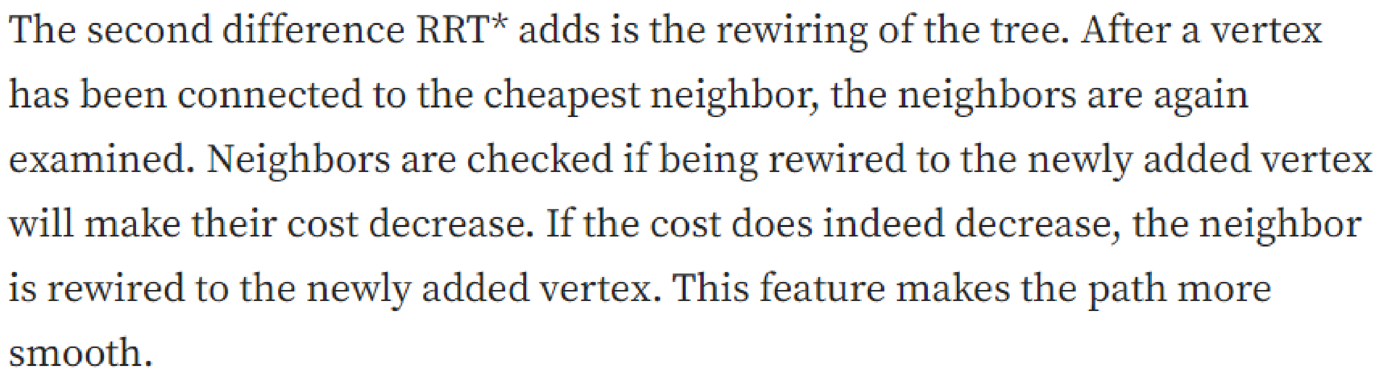
- Records the distance each vertex has traversed relative to its parent vertex
- Rewiring of the tree

RRT*



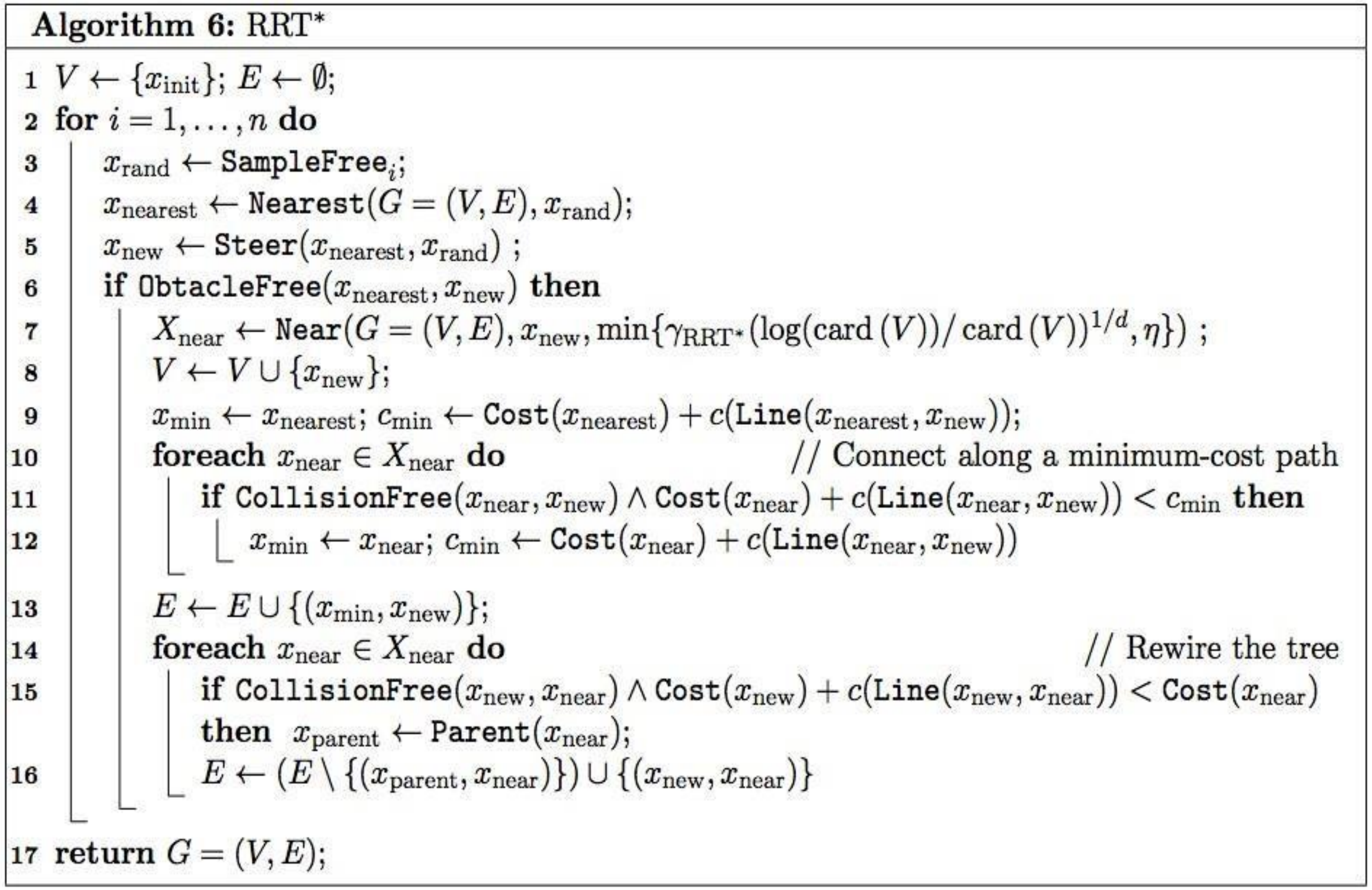
First, RRT* records the distance each vertex has traveled relative to its parent vertex. This is referred to as the `cost()` of the vertex. After the closest node is found in the graph, a neighborhood of vertices in a fixed radius from the new node are examined. If a node with a cheaper `cost()` than the proximal node is found, the cheaper node replaces the proximal node. The effect of this feature can be seen with the addition of fan shaped twigs in the tree structure. The cubic structure of RRT is eliminated.

RRT*



The second difference RRT* adds is the rewiring of the tree. After a vertex has been connected to the cheapest neighbor, the neighbors are again examined. Neighbors are checked if being rewired to the newly added vertex will make their cost decrease. If the cost does indeed decrease, the neighbor is rewired to the newly added vertex. This feature makes the path more smooth.

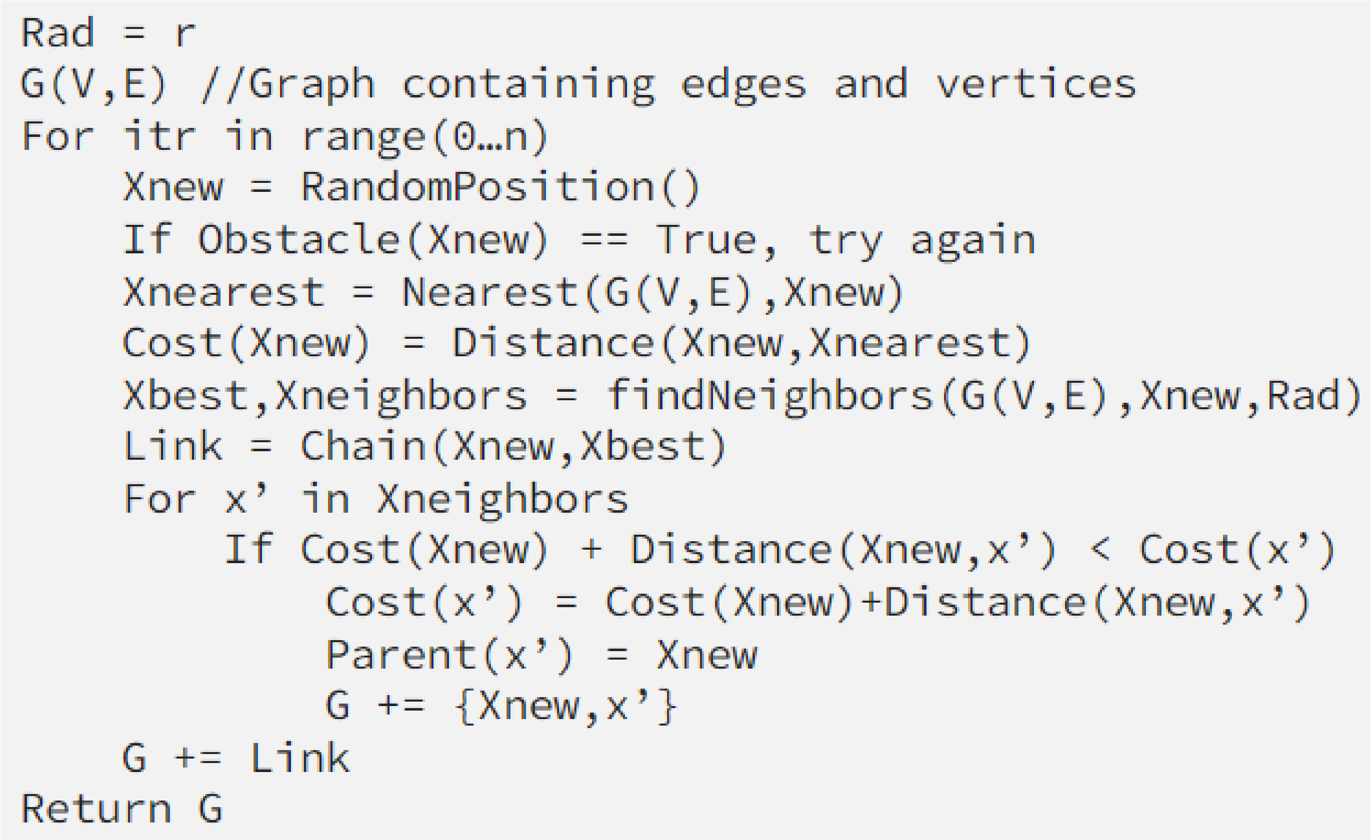
RRT*



Algorithm 6: RRT*

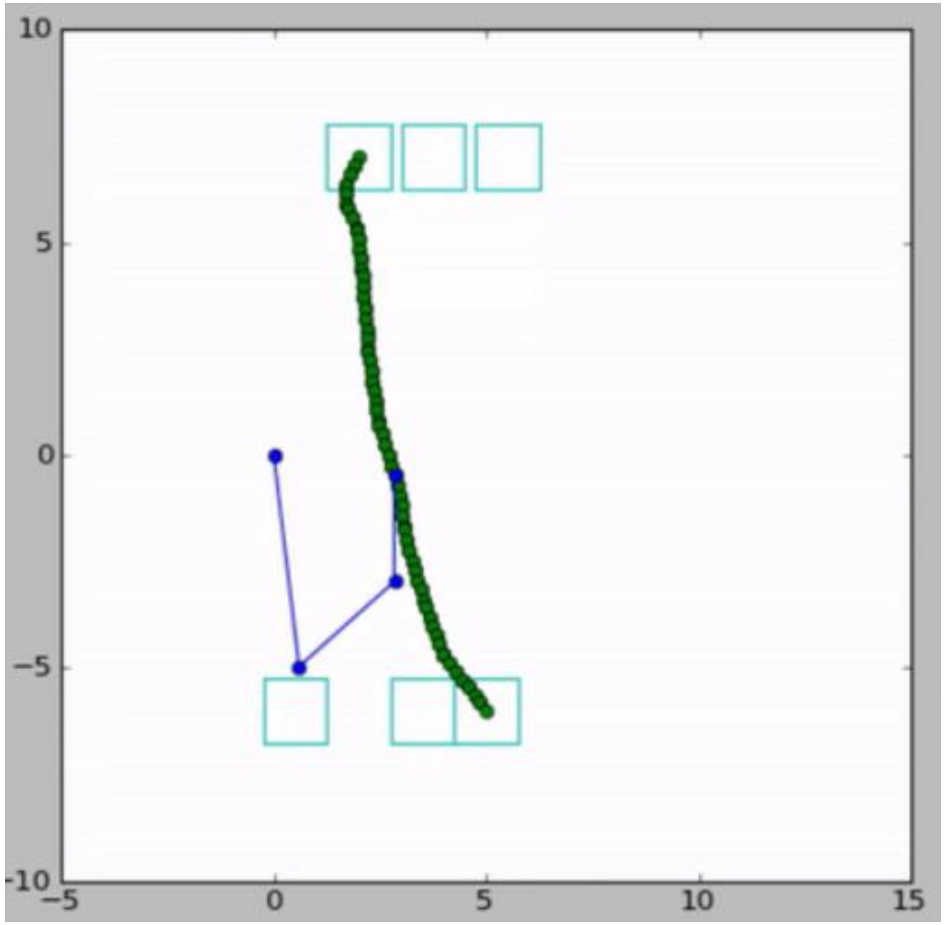
```
$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ do
    $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
    $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
    $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$ ;
    if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ then
        $X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$ ;
        $V \leftarrow V \cup \{x_{\text{new}}\}$;
        $x_{\text{min}} \leftarrow x_{\text{nearest}}$; $c_{\text{min}} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
        foreach $x_{\text{near}} \in X_{\text{near}}$ do                // Connect along a minimum-cost path
            if $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\text{min}}$ then
                $x_{\text{min}} \leftarrow x_{\text{near}}$; $c_{\text{min}} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
        $E \leftarrow E \cup \{(x_{\text{min}}, x_{\text{new}})\}$;
        foreach $x_{\text{near}} \in X_{\text{near}}$ do                // Rewire the tree
            if $\texttt{CollisionFree}(x_{\text{new}}, x_{\text{near}}) \wedge \texttt{Cost}(x_{\text{new}}) + c(\texttt{Line}(x_{\text{new}}, x_{\text{near}})) < \texttt{Cost}(x_{\text{near}})$
            then $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
            $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
return $G = (V, E)$;
```

RRT* Pseudo code



```
Rad = r
G(V,E) //Graph containing edges and vertices
For itr in range(0…n)
    Xnew = RandomPosition()
    If Obstacle(Xnew) == True, try again
    Xnearest = Nearest(G(V,E),Xnew)
    Cost(Xnew) = Distance(Xnew,Xnearest)
    Xbest,Xneighbors = findNeighbors(G(V,E),Xnew,Rad)
    Link = Chain(Xnew,Xbest)
    For x’ in Xneighbors
        If Cost(Xnew) + Distance(Xnew,x’) < Cost(x’)
            Cost(x’) = Cost(Xnew)+Distance(Xnew,x’)
            Parent(x’) = Xnew
            G += {Xnew,x’}
    G += Link
Return G
```

RRT* Path: Example



RRT* Path: Example

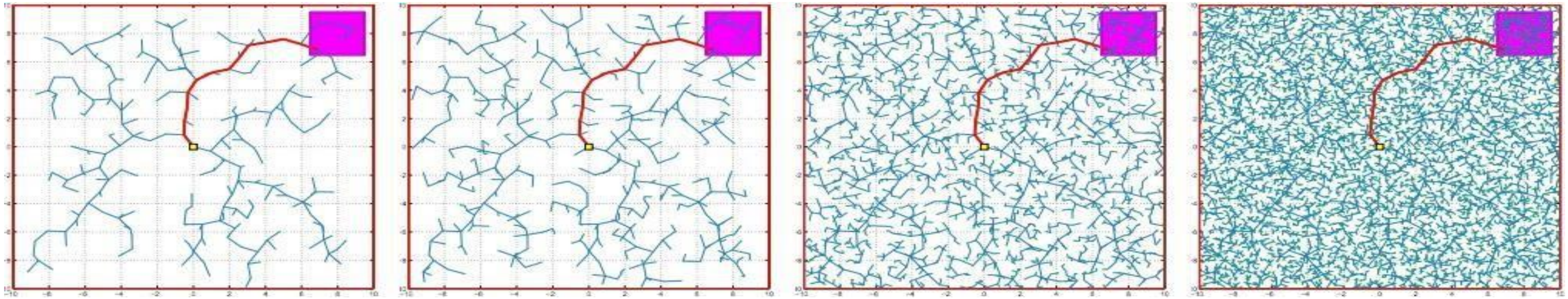
- Asymptotically optimal
- Main idea:
  - Swap new point in as parent for nearby vertices who can be reached along shorter path through new point than through their original (current) parent

RRT*: Limitations

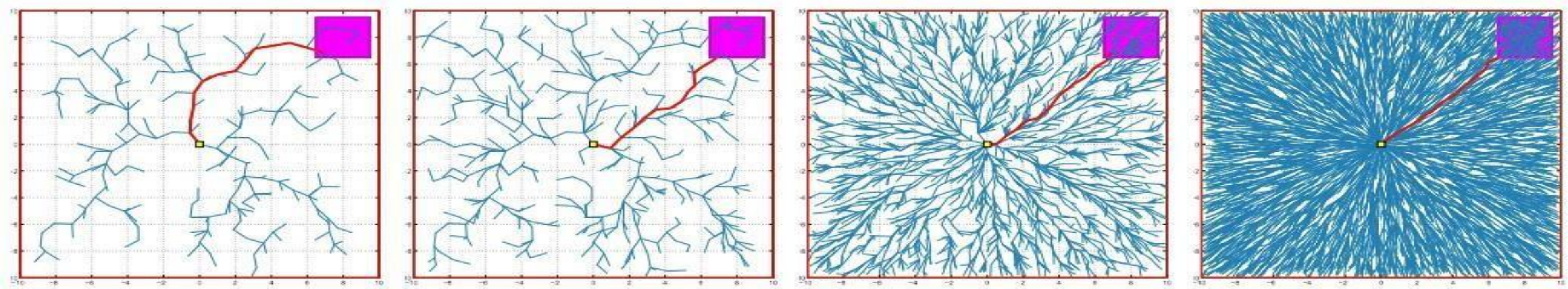
- Time?
    - ~x8 times more time-consuming than RRT

RRT*

RRT

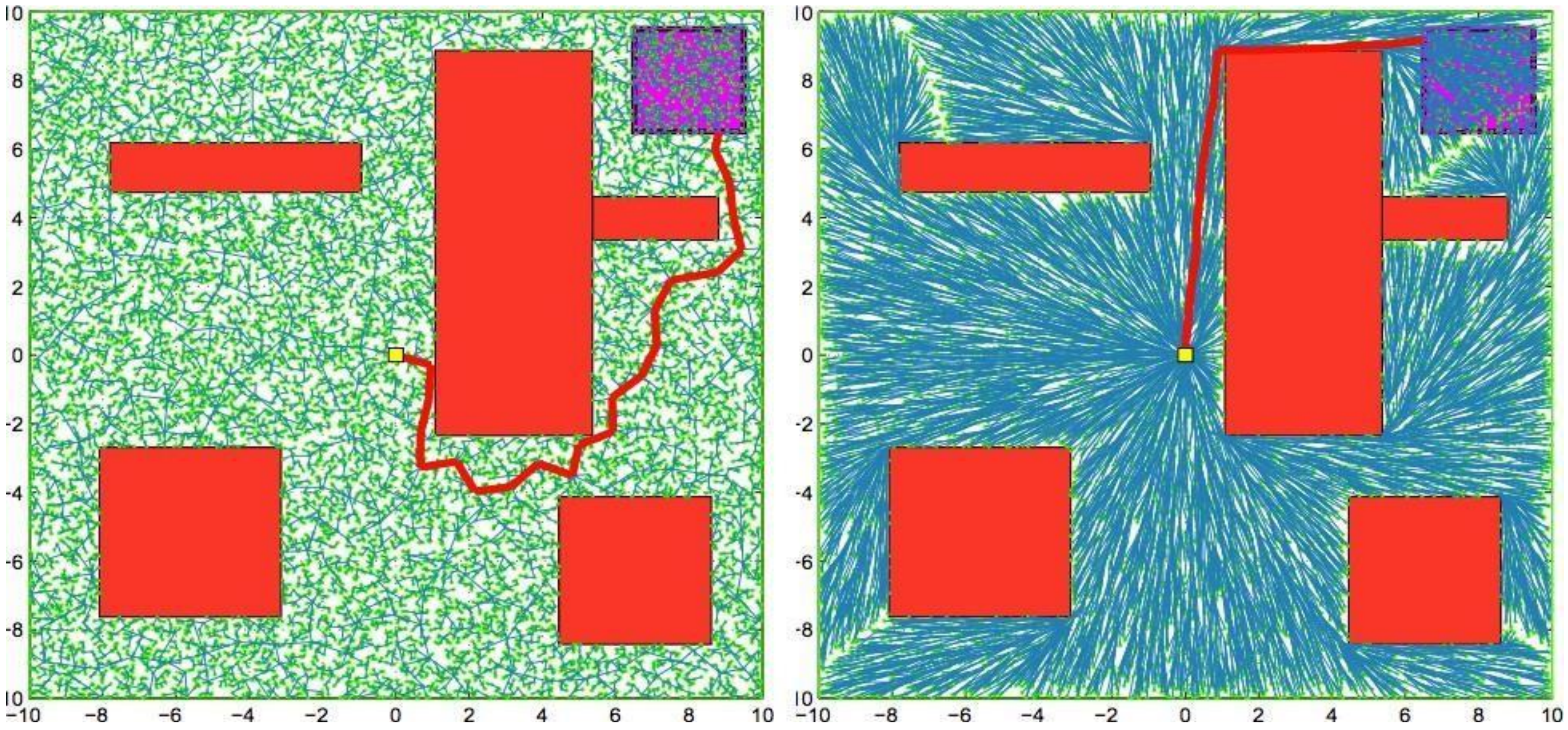


RRT*



Source: Karaman and Frazzoli

RRT*



RRT

RRT*

Source: Karaman and Frazzoli

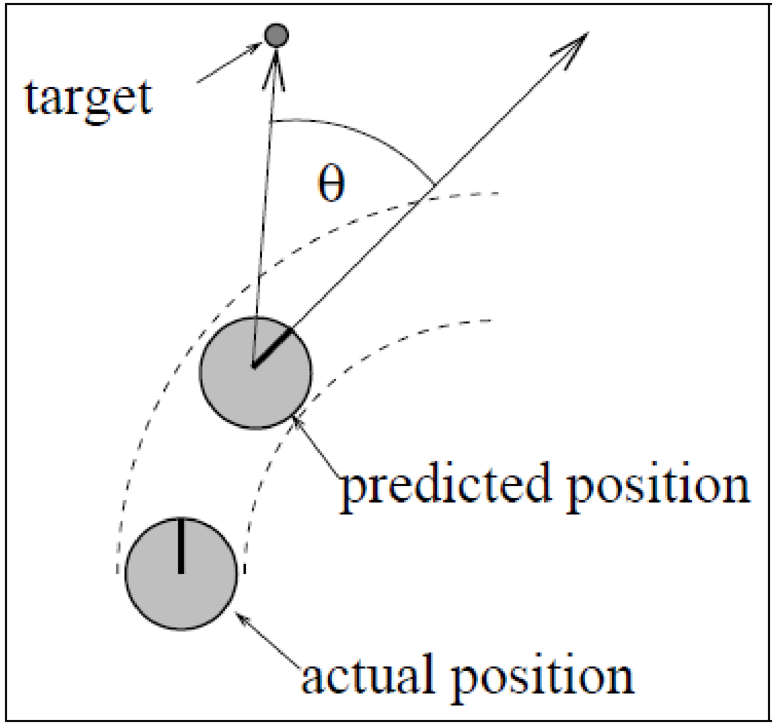
Smoothing

- Randomized motion planners tend to find not so great paths for execution: very jagged, often much longer than necessary
- In practice: do smoothing before using the path

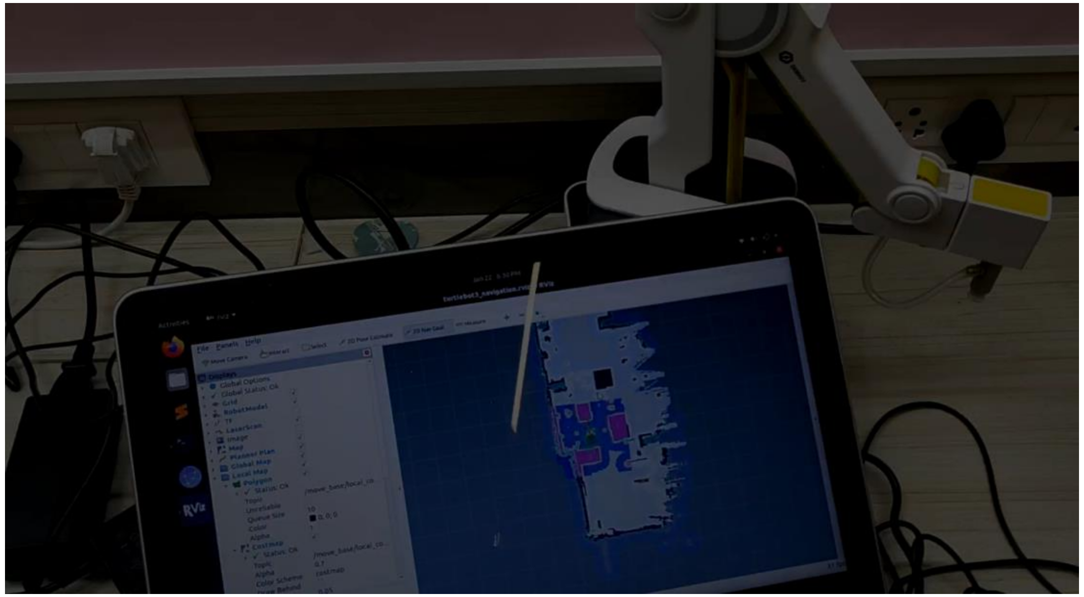
- Shortcutting:
  - along the found path, pick two vertices xt1, xt2 and try to connect them directly (skipping over all intermediate vertices)
  - Nonlinear optimization for optimal control
    - Allows to specify an objective function that includes smoothness in state, control, small control inputs, etc.

DWA: Dynamic-Window Approach

- Discretize and maximize an objective function given by:
  - $G(v,\omega) = \sigma\big(\alpha.heading(v,\omega) + \beta.dist(v,\omega) + \gamma.velocity(v,\omega)\big)$
- Heading (v,ω) → alignment of robot with that of direction of target.
- Dist (v,ω) → distance to the closest obstacle if the corresponding (v,ω) were chosen
- Velocity (v,ω) returns the 'v'
- Other planners include TEB(timed elastic band) local planner, learning based path planner, etc.



DWA: Implementation



DWA

- Maximizing solely the clearance *(dist)* and *velocity* => no incentive to move towards goal
- Maximizing only heading, robot will not move around the obstacles
- Using all three components, robot will move around obstacles as fast as it can
- Local approaches are better for obstacle avoidance
- Low computational complexity
- Sometimes the robot gets stuck in local optimum