香港科技大學 THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

電子及計算機工程學系 DEPARTMENT OF ELECTRONIC & COMPUTER ENGINEERING

Introduction to Aerial Robotics
Lecture 4

Shaojie Shen

Associate Professor

Dept. of ECE, HKUST

25 Feb 2025

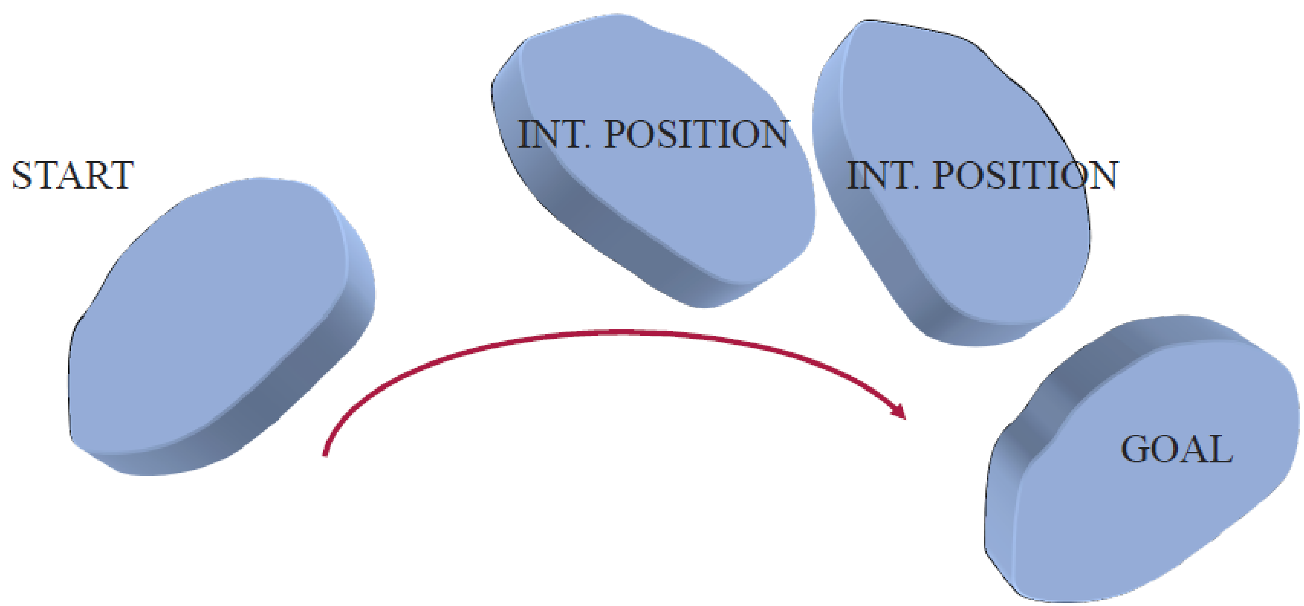
Outline

- Continuation on Trajectory Generation
- Path Planning

Trajectory Generation

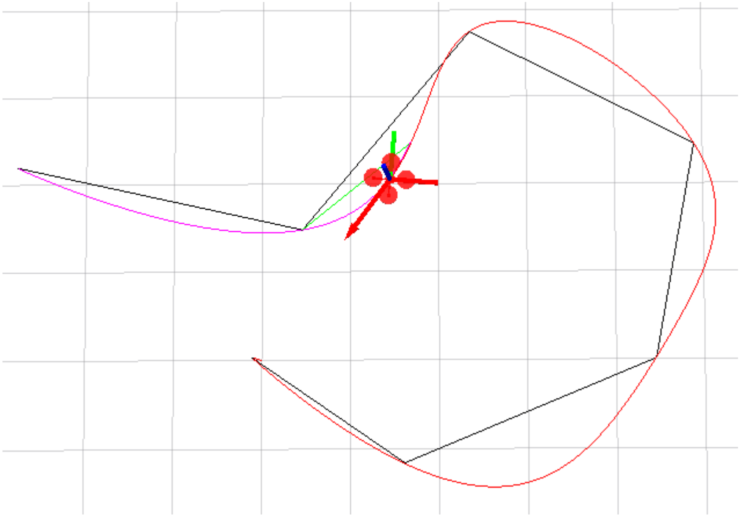
Smooth 3D Trajectories

- Smooth trajectory is beneficial for autonomous flight
  - Smooth trajectories respect the continuous nature of aerial robots
  - The robot should not stop at turns



Smooth 3D Trajectories

- General setup
  - Start, goal positions (orientations)
  - Waypoint positions (orientations)
    - Waypoints can be found by path planning (A*, RRT*, etc)
  - Smoothness criterion
    - Generally translates into minimizing rate of change of "input"



Differential Flatness

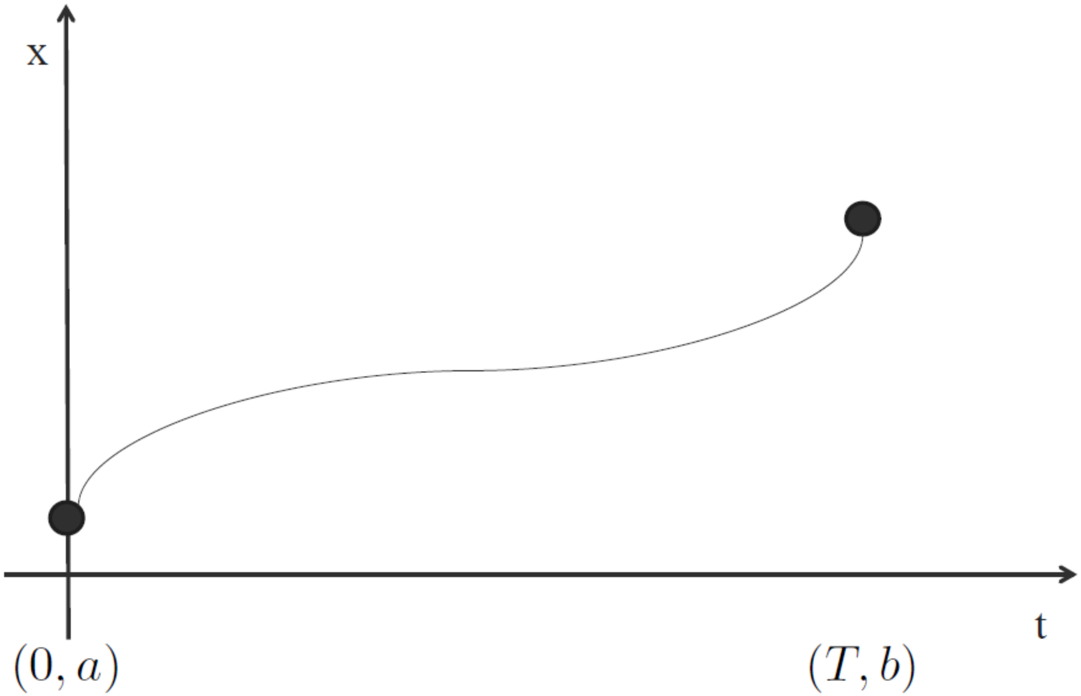
- The states and the inputs of a quadrotor can be written as algebraic functions of four carefully selected flat outputs and their derivatives
  - Enables automated generation of trajectories
  - Any smooth trajectory in the space of flat outputs (with reasonably bounded derivatives) can be followed by the under-actuated quadrotor
  - A possible choice:
    - $\boldsymbol{\sigma} = [x, y, z, \psi]^T$
  - Trajectory in the space of flat outputs:
    - $\boldsymbol{\sigma}(t) = [T_0, T_M] \to \mathbb{R}^3 \times SO(2)$

Polynomial Trajectories

- Flat outputs:
  - $\boldsymbol{\sigma} = [x, y, z, \psi]^T$
- Trajectory in the space of flat outputs:
  - $\boldsymbol{\sigma}(t) = [T_0, T_M] \to \mathbb{R}^3 \times SO(2)$
- Polynomial functions can be used to specify trajectories in the space of flat outputs
  - Easy determination of smoothness criterion with polynomial orders
  - Easy and closed form calculation of derivatives
  - Decoupled trajectory generation in three dimensions

Smooth 1D Trajectory

- Design a trajectory $x(t)$ such that:
  - $x(0) = a$
  - $x(T) = b$



Smooth 1D Trajectory

- 5^{th} order polynomial trajectory:
  - $x(t) = c_5 t^5 + c_4 t^4 + c_3 t^3 + c_2 t^2 + c_1 t + c_0$
- Boundary conditions

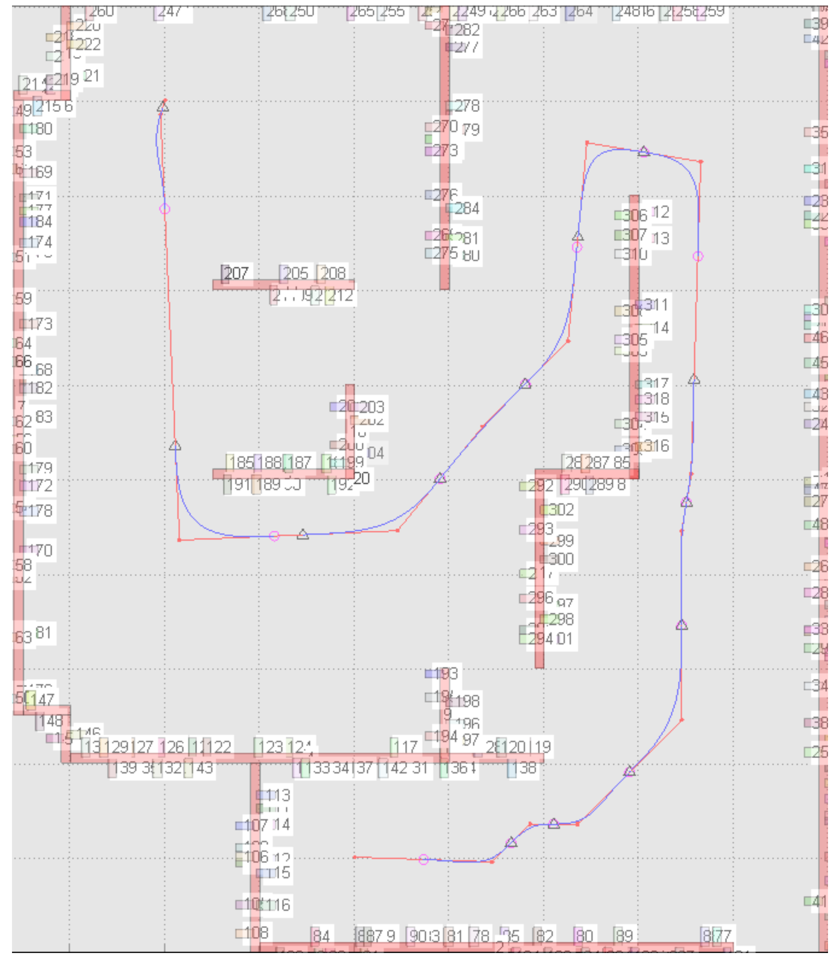
	Position	Velocity	Acceleration
t = 0	a	0	0
t = T	b	0	0

- Solve:

$$\begin{bmatrix} a \\ b \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ T^5 & T^4 & T^3 & T^2 & T & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 5T^4 & 4T^3 & 3T^2 & 2T & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 20T^3 & 12T^2 & 6T & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix}$$

Smooth Multi-Segment Trajectory

- Given waypoints to a desired goal
- Smooth the corners of straight line segments
- Preferred constant velocity motion at v
- Preferred zero acceleration
- Requires special handling of short segments



Smooth 1D Trajectory

- Generate each 5^{th} order polynomial independently:
  - $x(t) = c_5 t^5 + c_4 t^4 + c_3 t^3 + c_2 t^2 + c_1 t + c_0$
- Boundary conditions

	Position	Velocity	Acceleration
t = 0	a	v_0	0
t = T	b	v_T	0

- Solve:

$$\begin{bmatrix} a \\ b \\ v_0 \\ v_T \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ T^5 & T^4 & T^3 & T^2 & T & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 5T^4 & 4T^3 & 3T^2 & 2T & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 20T^3 & 12T^2 & 6T & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix}$$

Optimization-based Trajectory Generation

- Explicitly minimize certain derivatives in the space of flat outputs
- Quadrotor dynamics

Derivative	Translation	Rotation	Thrust
0	Position		
1	Velocity		
2	Acceleration	Rotation	
3	Jerk	Angular Velocity	
4	Snap	Angular Acceleration	Differential Thrust
5	Crackle	Angular Jerk	Change in Thrust

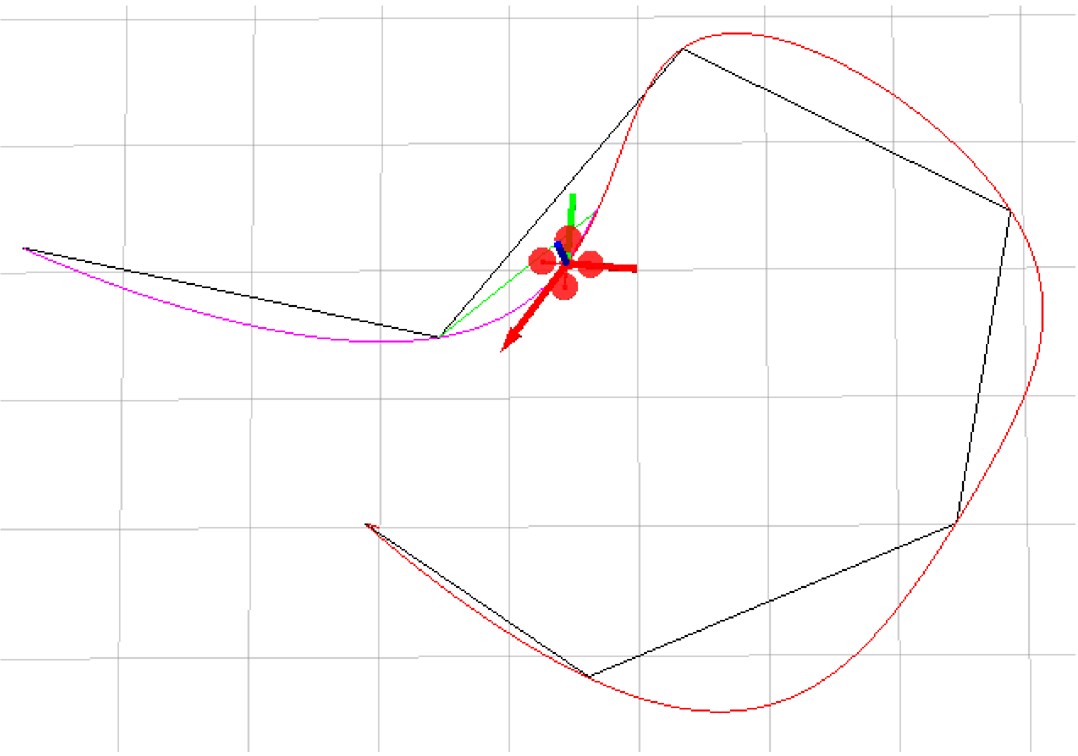
Optimization-based Trajectory Generation

- Explicitly minimize certain derivatives in the space of flat outputs
  - Minimum jerk: minimize angular velocity, good for visual tracking
  - Minimum snap: minimize differential thrust, saves energy

Derivative	Translation	Rotation	Thrust
0	Position		
1	Velocity		
2	Acceleration	Rotation	
3	Jerk	Angular Velocity	
4	Snap	Angular Acceleration	Differential Thrust
5	Crackle	Angular Jerk	Change in Thrust

Minimum Snap Trajectory Generation

- Multi-segment minimum snap trajectory



Minimum Snap Trajectory Generation

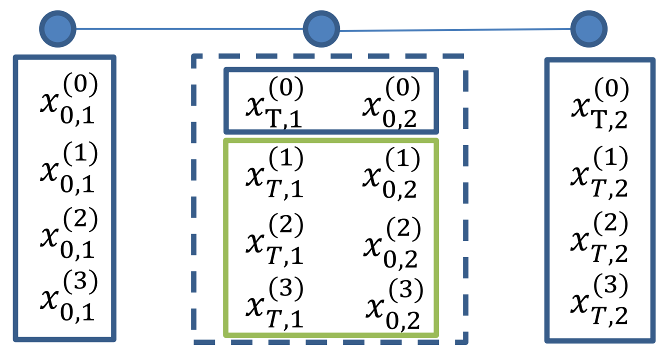
- Formulation – segment durations must be known!

$$f(t) = \begin{cases} f_1(t) \doteq \sum_{i=0}^{N} p_{1,i}(t - T_0)^i & T_0 \le t \le T_1 \\ f_2(t) \doteq \sum_{i=0}^{N} p_{2,i}(t - T_1)^i & T_1 \le t \le T_2 \\ \vdots & \vdots \\ f_M(t) \doteq \sum_{i=0}^{N} p_{M,i}(t - T_{M-1})^i & T_{M-1} \le t \le T_M \end{cases}$$

Subject to:

Derivative constraints: $\begin{cases} f_j^{(k)}(T_{j-1}) = x_{0,j}^{(k)} \\ f_j^{(k)}(T_j) = x_{T,j}^{(k)} \end{cases}$

Continuity constraints: $f_j^{(k)}(T_j) = f_{j+1}^{(k)}(T_j)$



- Minimum degree polynomial to ensure smoothness for one-segment trajectory:
  - Minimum jerk: N = 2 x 3 - 1 = 5
  - Minimum snap: N = 2 x 4 - 1 = 7

Minimum Snap Trajectory Generation

- Cost function for one polynomial segment:

$$f(t) = \sum_i p_i t^i$$

$$\Rightarrow f^{(4)}(t) = \sum_{i \geq 4} i(i-1)(i-2)(i-3)t^{i-4} p_i$$

$$\Rightarrow \left(f^{(4)}(t)\right)^2 = \sum_{i \geq 4, j \geq 4} i(i-1)(i-2)(i-3)j(j-1)(j-2)(j-3)t^{i+j-8} p_i p_j$$

$$\Rightarrow J(T) = \int_0^T \left(f^4(t)\right)^2 dt = \sum_{i \geq 4, j \geq 4} \frac{i(i-1)(i-2)(i-3)j(j-1)(j-2)(j-3)}{i+j-7} T^{i+j-7} p_i p_j$$

$$\Rightarrow J(T) = \int_0^T \left(f^4(t)\right)^2 dt = \begin{bmatrix} \vdots \\ p_i \\ \vdots \end{bmatrix}^T \begin{bmatrix} & \vdots & \\ \cdots & \frac{i(i-1)(i-2)(i-3)j(j-1)(j-2)(j-3)}{i+j-7} T^{i+j-7} & \cdots \\ & \vdots & \end{bmatrix} \begin{bmatrix} \vdots \\ p_j \\ \vdots \end{bmatrix}$$

$$\Rightarrow \boxed{J_k(T) = \mathbf{p}_k^T \mathbf{Q}_k \mathbf{p}_k}$$

Minimize this!

Minimum Snap Trajectory Generation

- Derivative constraint for one polynomial segment
  - Also models waypoint constraint (0^{th} order derivative)

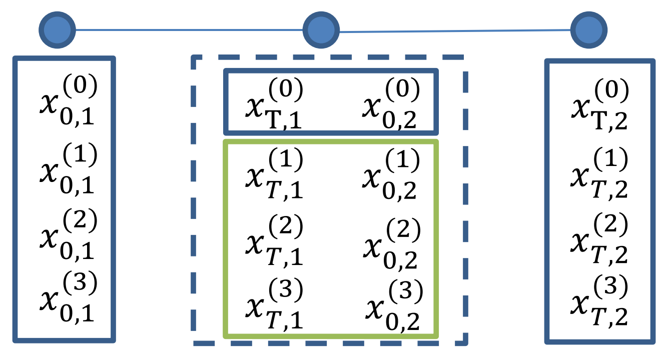
$$f_j^{(k)}(T_j) = x_j^{(k)}$$

$$\Rightarrow \sum_{i \geq k} \frac{i!}{(i-k)!} T_j^{i-k} p_{j,i} = x_{T,j}^{(k)}$$

$$\Rightarrow \begin{bmatrix} \cdots & \frac{i!}{(i-k)!} T_j^{i-k} & \cdots \end{bmatrix} \begin{bmatrix} \vdots \\ p_{j,i} \\ \vdots \end{bmatrix} = x_{T,j}^{(k)}$$

$$\Rightarrow \begin{bmatrix} \cdots & \frac{i!}{(i-k)!} T_{j-1}^{i-k} & \cdots \\ \cdots & \frac{i!}{(i-k)!} T_j^{i-k} & \cdots \end{bmatrix} \begin{bmatrix} \vdots \\ p_{j,i} \\ \vdots \end{bmatrix} = \begin{bmatrix} x_{0,j}^{(k)} \\ x_{T,j}^{(k)} \end{bmatrix}$$

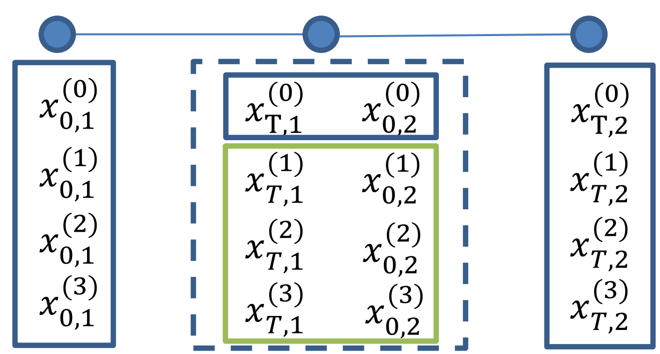
$$\Rightarrow \mathbf{A}_j \mathbf{p}_j = \mathbf{d}_j$$



Minimum Snap Trajectory Generation

- Continuity constraint between two segments:
  - Ensures continuity between trajectory segments when no specific derivatives are given

$$
\begin{aligned}
& f_j^{(k)}(T_j) = f_{j+1}^{(k)}(T_j) \\
\Rightarrow & \sum_{i \geq k} \frac{i!}{(i-k)!} T_j^{i-k} p_{j,i} - \sum_{l \geq k} \frac{l!}{(l-k)!} T_j^{l-k} p_{j+1,l} = 0 \\
\Rightarrow & \begin{bmatrix} \cdots & \frac{i!}{(i-k)!} T_j^{i-k} & \cdots & -\frac{l!}{(l-k)!} T_j^{l-k} & \cdots \end{bmatrix} \begin{bmatrix} \vdots \\ p_{j,i} \\ \vdots \\ p_{j+1,l} \\ \vdots \end{bmatrix} = 0 \\
\Rightarrow & \begin{bmatrix} \mathbf{A}_j & -\mathbf{A}_{j+1} \end{bmatrix} \begin{bmatrix} \mathbf{p}_j \\ \mathbf{p}_{j+1} \end{bmatrix} = 0
\end{aligned}
$$



Minimum Snap Trajectory Generation

- Constrained quadratic programming (QP) formulation:

$$\min \quad \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{bmatrix}^T \begin{bmatrix} \mathbf{Q}_1 & & \\ & \ddots & \\ & & \mathbf{Q}_M \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{bmatrix}$$

$$\mathrm{s.t.} \quad \mathbf{A}_{\mathrm{eq}} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{bmatrix} = \mathbf{d}_{eq}$$

Minimum Snap Trajectory Generation

- Direct optimization of polynomial trajectories is numerically unstable
- A change of variable that instead optimizes segment endpoint derivatives is preferred
- We have $\boldsymbol{M}_{\mathrm{j}}\mathbf{p}_j = \mathbf{d}_j$, where $\boldsymbol{M}_{\mathrm{j}}$ is a mapping matrix that maps polynomial coefficients to derivatives

$$J = \begin{bmatrix} \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_M \end{bmatrix}^T \begin{bmatrix} \boldsymbol{M}_1 & & \\ & \ddots & \\ & & \boldsymbol{M}_M \end{bmatrix}^{-T} \begin{bmatrix} \mathbf{Q}_1 & & \\ & \ddots & \\ & & \mathbf{Q}_M \end{bmatrix} \begin{bmatrix} \boldsymbol{M}_1 & & \\ & \ddots & \\ & & \boldsymbol{M}_M \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_M \end{bmatrix}$$

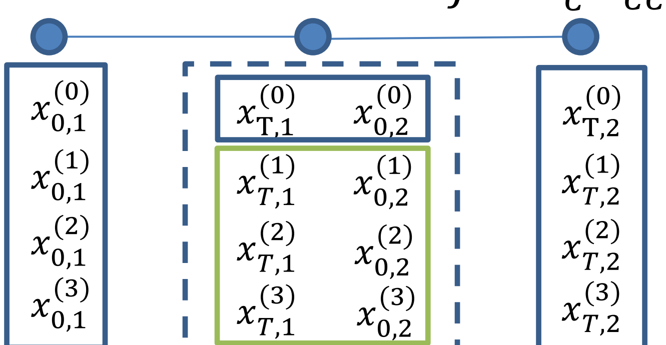
Minimum Snap Trajectory Generation

- Use a selection matrix $\mathbf{C}$ to separate fixed / constrained ($\mathbf{d}_C$) and free / unknown ($\mathbf{d}_U$) variables
  - Free variables : derivatives unspecified, only enforced by continuity constraints

$$J = \begin{bmatrix} \mathbf{d}_C \\ \mathbf{d}_U \end{bmatrix}^T \underbrace{\mathbf{C}\boldsymbol{M}^{-T}\mathbf{Q}\boldsymbol{M}^{-1}\mathbf{C}^T}_{\mathbf{R}} \begin{bmatrix} \mathbf{d}_C \\ \mathbf{d}_U \end{bmatrix} = \begin{bmatrix} \mathbf{d}_C \\ \mathbf{d}_U \end{bmatrix}^T \begin{bmatrix} \mathbf{R}_{CC} & \mathbf{R}_{CU} \\ \mathbf{R}_{UC} & \mathbf{R}_{UU} \end{bmatrix} \begin{bmatrix} \mathbf{d}_C \\ \mathbf{d}_U \end{bmatrix}$$

- Turned into an unconstrained quadratic programming that can be solved in closed form:

$$J = \mathbf{d}_C^T \mathbf{R}_{CC} \mathbf{d}_C + \mathbf{d}_C^T \mathbf{R}_{CU} \mathbf{d}_U + \mathbf{d}_U^T \mathbf{R}_{UC} \mathbf{d}_C + \mathbf{d}_U^T \mathbf{R}_{UU} \mathbf{d}_U$$



$$\mathbf{d}_U^* = -\mathbf{R}_{UU}^{-1} \mathbf{R}_{CU}^T \mathbf{d}_C$$

Minimum Snap Trajectory Generation

$d_{0,j}^{(k)}$

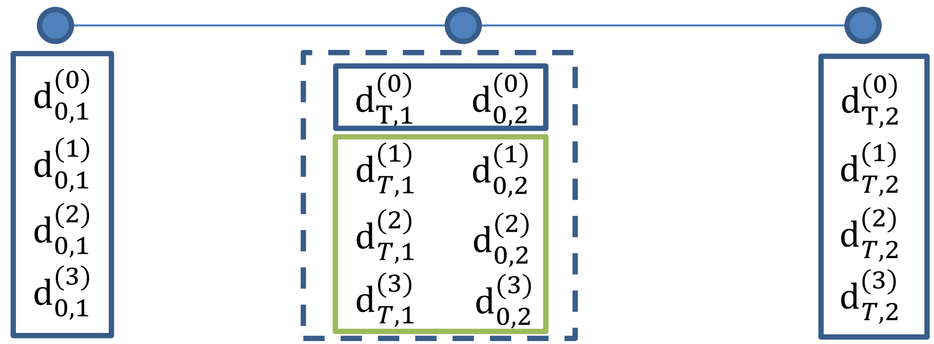
Index of derivative

Index of segment

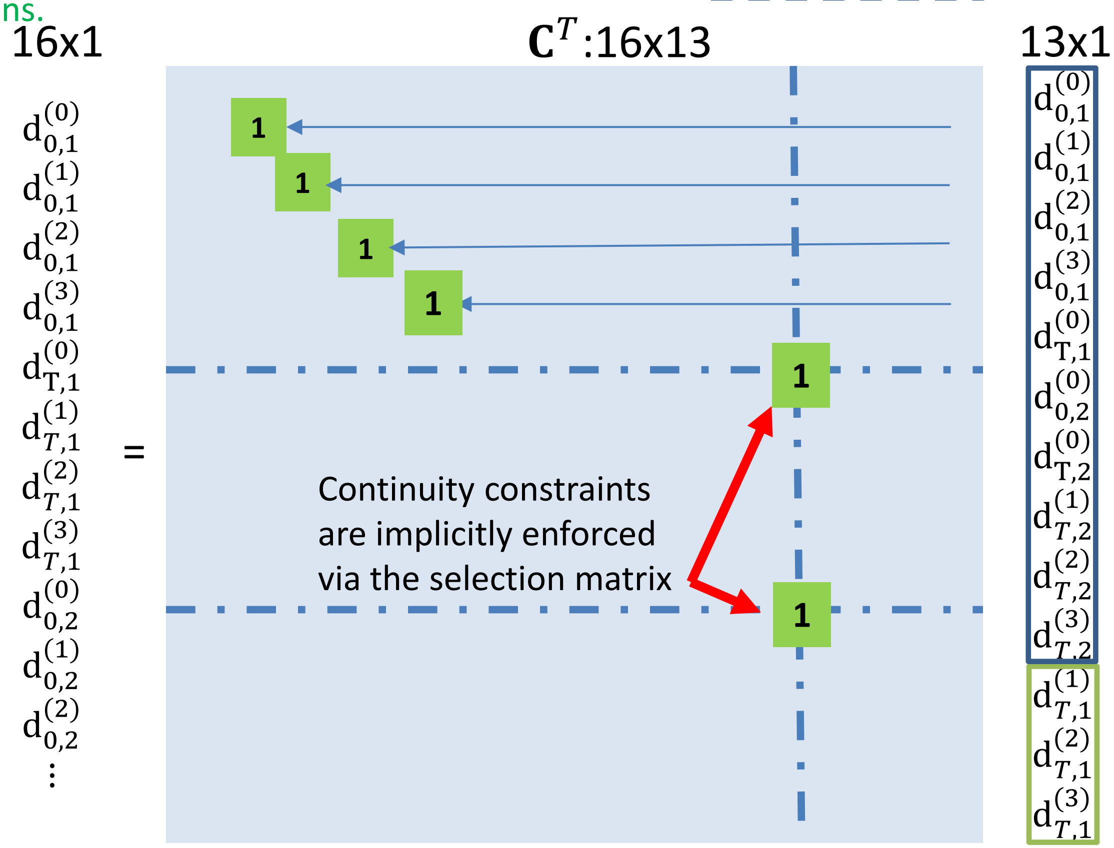
Time index

Fixed / constrained derivatives: fixed start, goal state, and intermediate positions

Free / unknown derivatives: all derivatives at intermediate connections.

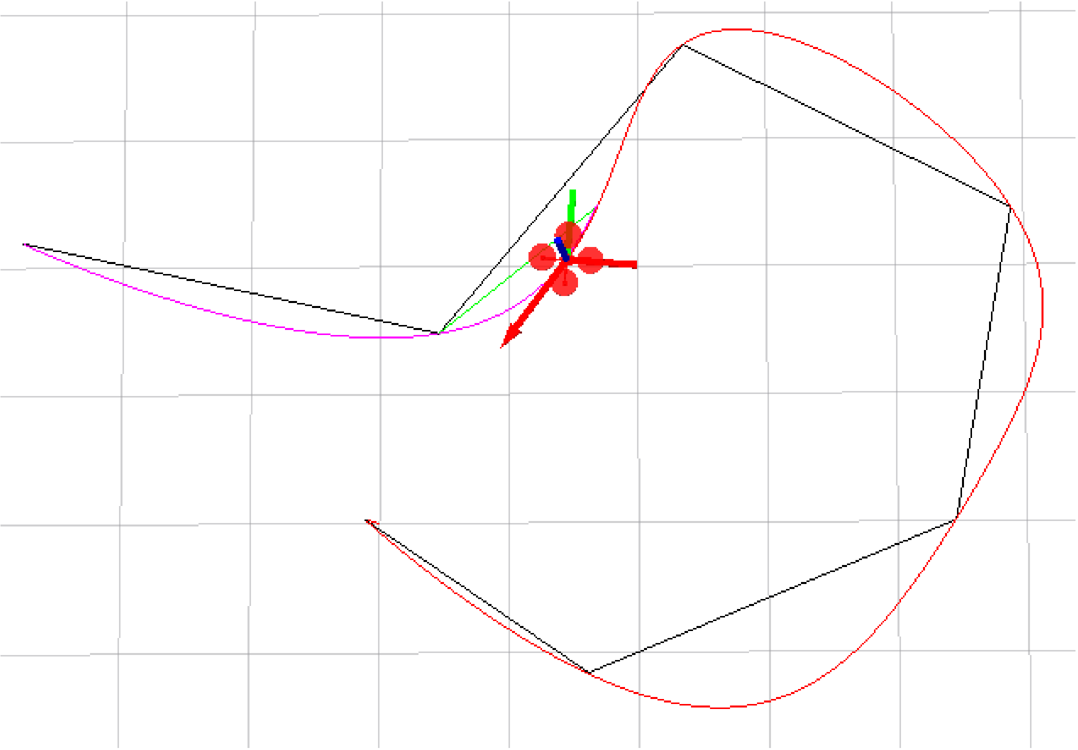


$$\begin{bmatrix} \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_M \end{bmatrix} = \mathbf{C}^T \begin{bmatrix} \mathbf{d}_C \\ \mathbf{d}_U \end{bmatrix}$$



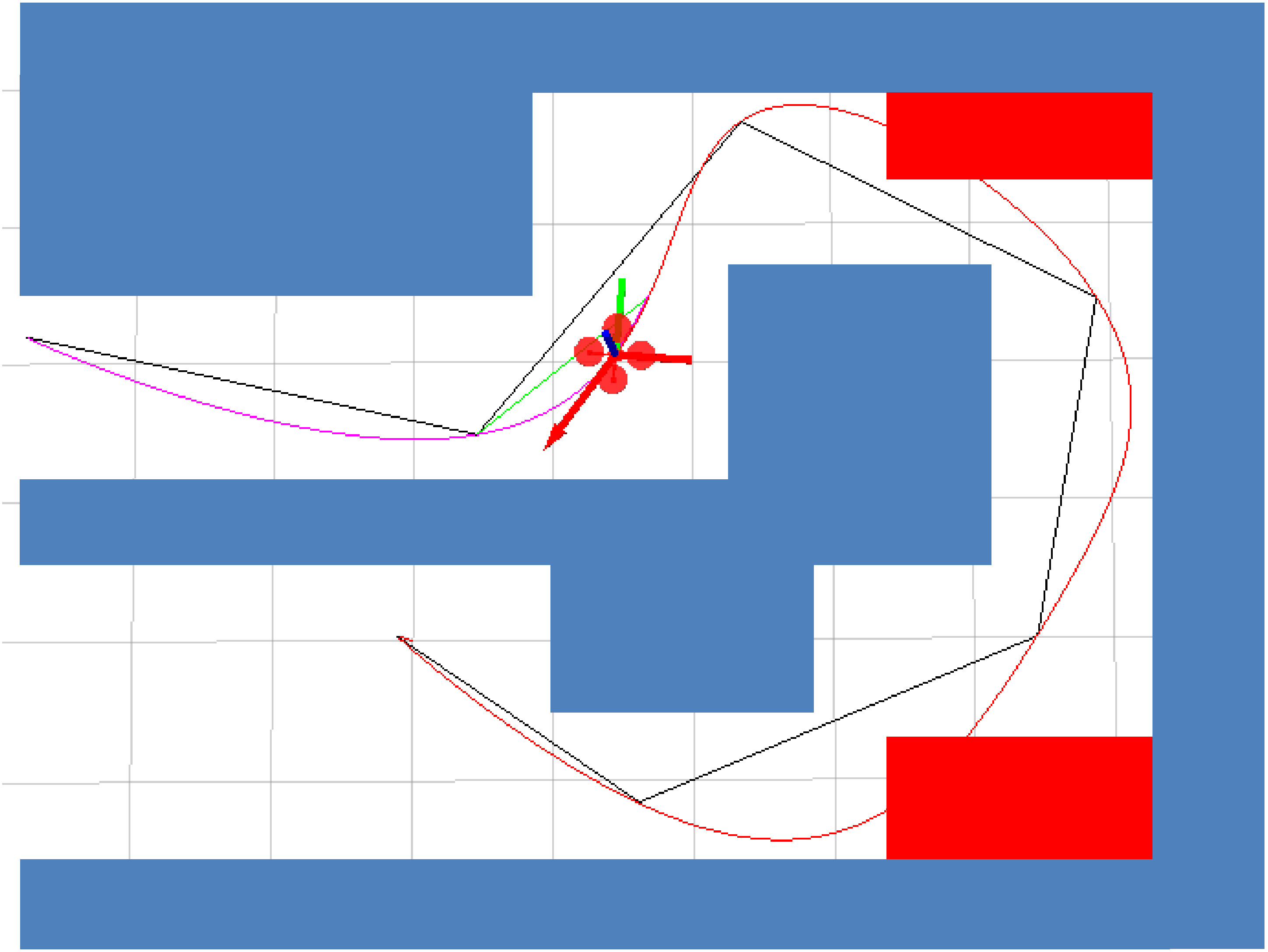
Minimum Snap Trajectory Generation

- Final trajectory

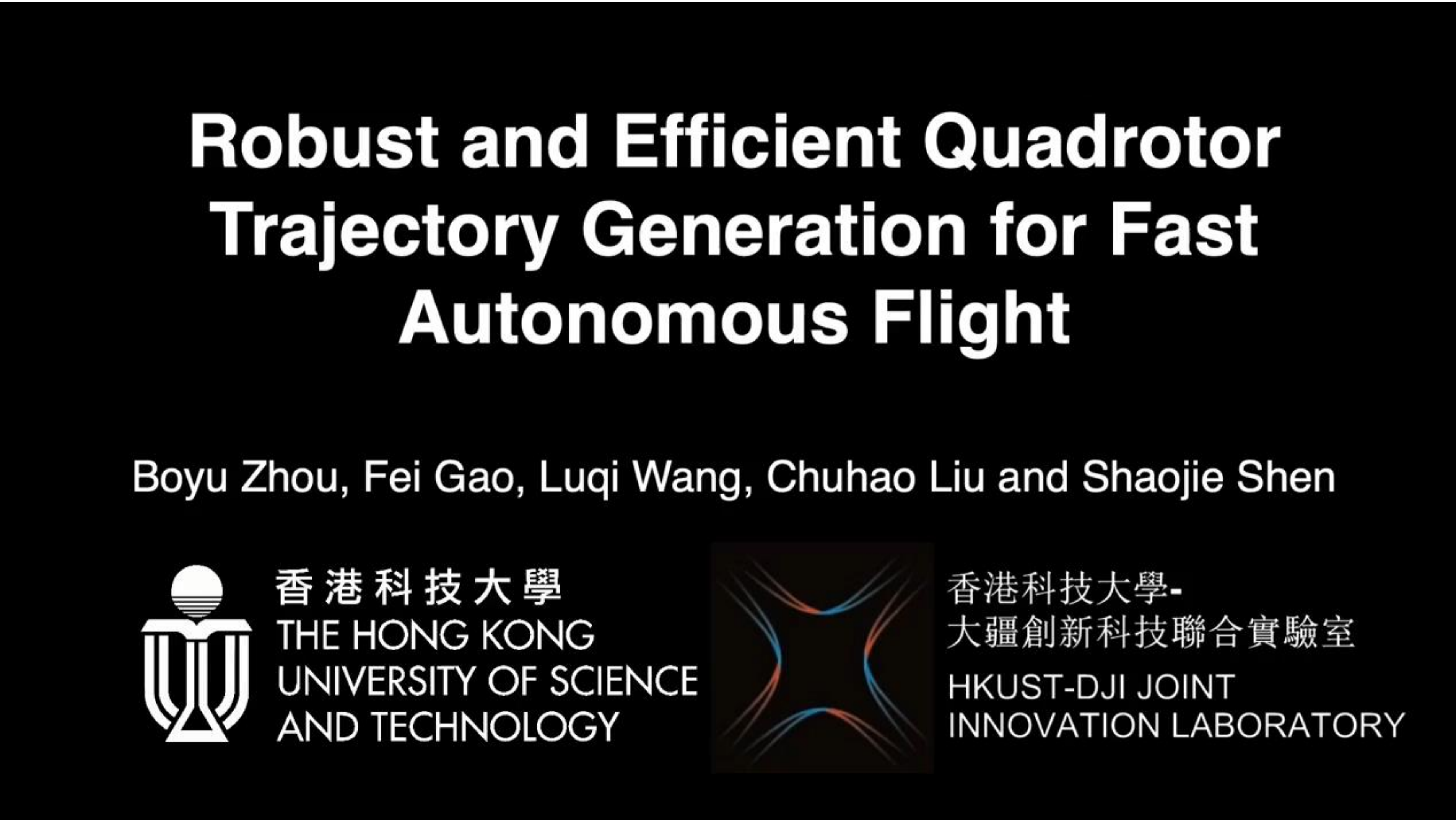


Minimum Snap Trajectory Generation

How to Ensure Collision-Free Trajectories?



Extension: Kinodynamic Search, Fast Collision Avoidance

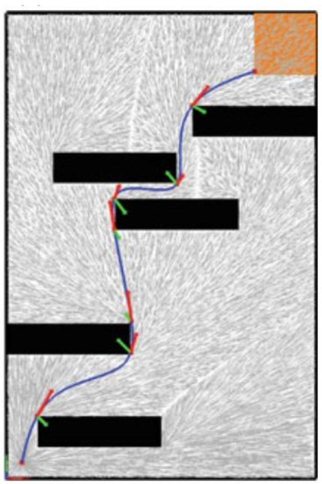


Extension: Temporal Optimization, Teach and Repeat

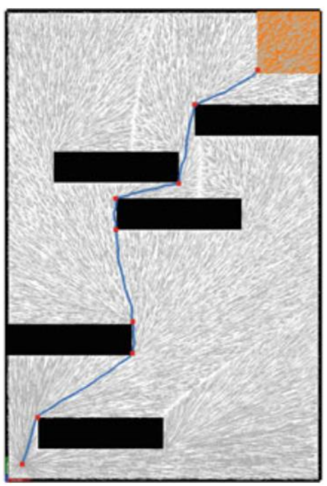
Path Planning

Motivation

- Why we need path planning?
  - Fundamental problem in robotics - finding collision-free route from A to B
- Hierarchical approach (path planning + trajectory generation)
  - Low complexity solution
    - Path planning can be more efficient since it's in a much lower dimension state space.



We already know how to fit the polynomial for given waypoints

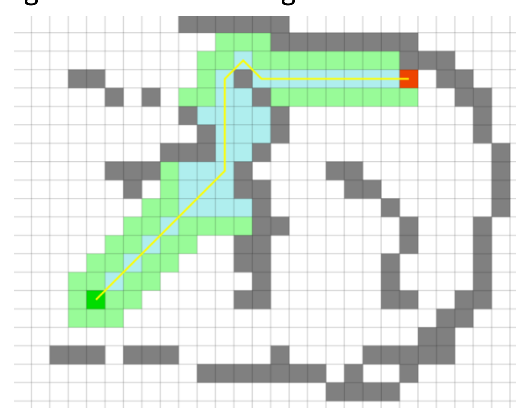


Then how to get these collision-free waypoints? → the role of path planning

Outline

- Configuration space obstacle
- Search-based methods
  - General graph search: DFS, BFS
  - A* search
- Sampling-based methods
  - Probabilistic roadmap (PRM)
  - Rapidly exploring random tree (RRT)

Grid-based graph:
use grid as vertices and grid connections as edges

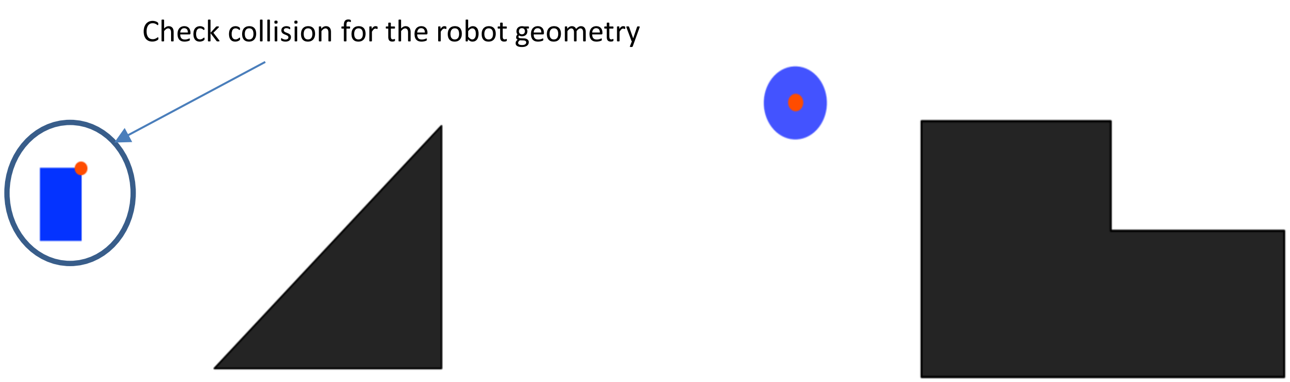


RRT example
See more examples at http://ompl.kavrakilab.org/gallery.html

A* search example
Try more search-based method at http://qiao.github.io/PathFinding.js/visual/

Workspace Space Obstacle

- Planning in *workspace*
  - Robot has different shape and size
  - Collision detection requires knowing the robot geometry - time consuming and hard



(1) Rectangular mobile robot

(2) Circular mobile robot

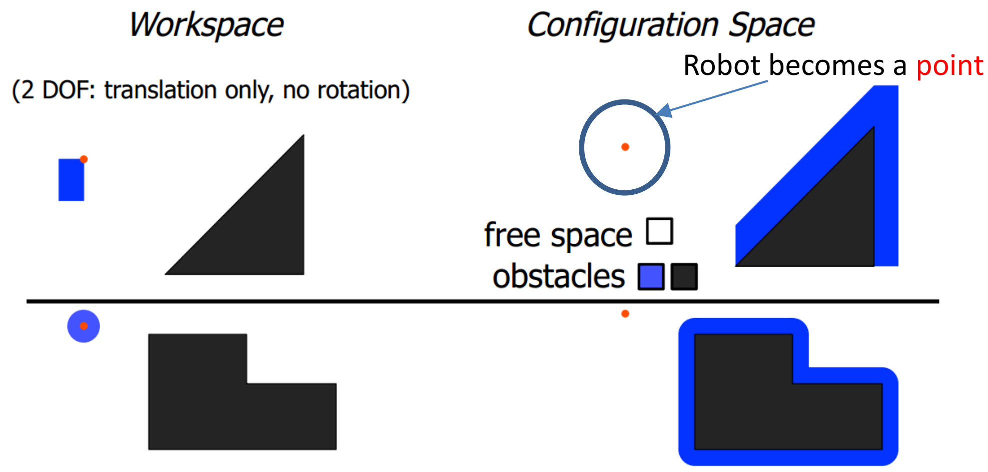
Configuration Space

- **Robot configuration**: a specification of the positions of all points of the robot
- **Robot degree of freedom (DOF):** The minimum number n of real-valued coordinates needed to represent the robot **configuration**
- **Robot configuration space**: a n-dim space containing all possible robot configurations, denoted as C-space
- Each robot pose is a point in the C-space
- Examples

	Configuration	C-space	DOF
Rigid rotation	$\boldsymbol{R}$	$SO(3)$	3
Rigid motion	$g = (\boldsymbol{R}, \boldsymbol{p})$	$SE(3)$	6
Flat outputs	$\boldsymbol{\sigma} = (\psi, \boldsymbol{p})$	$SO(2) \times \mathbb{R}^3$	4

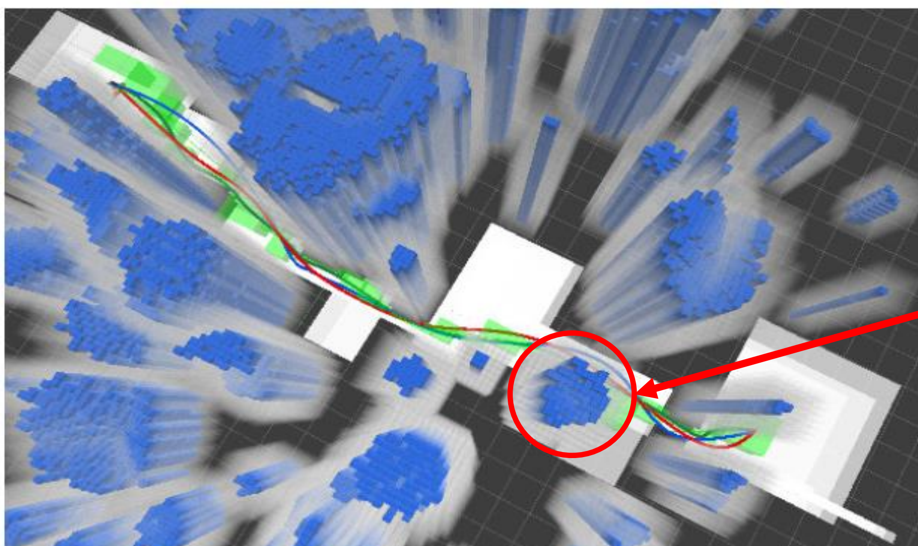
Configuration Space Obstacle

- Planning in *configuration space*: C-space
  - Robot is represented by a point in C-space, e.g. position (a point in $\mathbb{R}^3$), pose (a point in $SE(3)$) , etc.
  - Obstacles need to be represented in configuration space (one-time work prior to motion planning), called configuration space obstacle, or C-obstacle
  - C-space = (C-obstacle) ∪ (C-free)
  - The path planning is finding a path between start point q_{start} and goal point q_{goal} within C-free



Workspace and Configuration Space Obstacles

- In *workspace*
  - Robot has shape and size (i.e. hard for motion planning)
- In *configuration space*: C-space
  - Robot is a point (i.e. easy for motion planning)
  - Obstacle are represented in C-space prior to motion planning
- Representing an obstacle in C-space can be extremely complicated. So approximated (but more conservative) representations are used in practice.



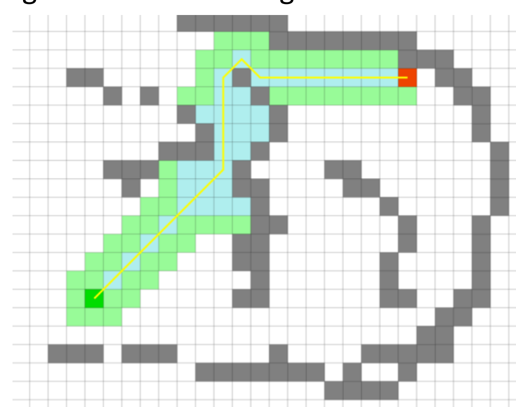
If we model the robot conservatively as a ball with radius δ_r,
then the C-space can be constructed by inflating obstacle at all directions by δ_r.

Outline

- Configuration space obstacle
- Search-based methods
  - General graph search: DFS, BFS
  - A* search
- Sampling-based methods
  - Probabilistic roadmap (PRM)
  - Rapidly exploring random tree (RRT)

Grid-based graph:
use grid as vertices and grid connections as edges

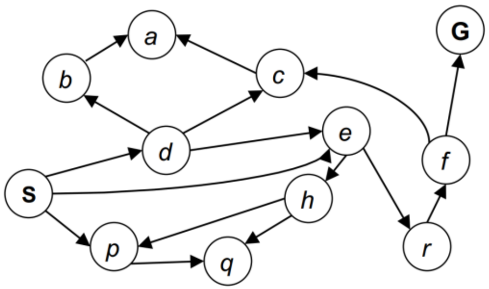
RRT example
See more examples at http://ompl.kavrakilab.org/gallery.html



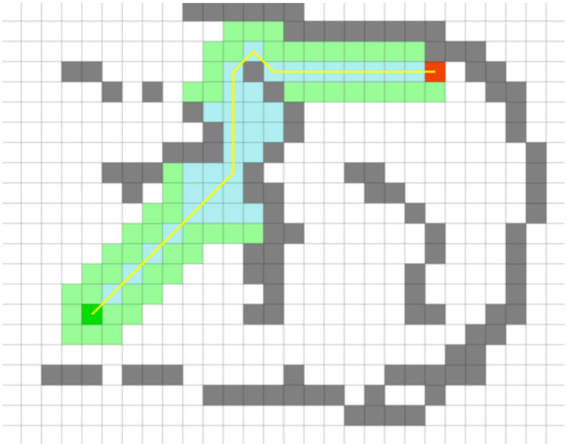
A* search example
Try more search-based method at http://qiao.github.io/PathFinding.js/visual/

Search-based Method

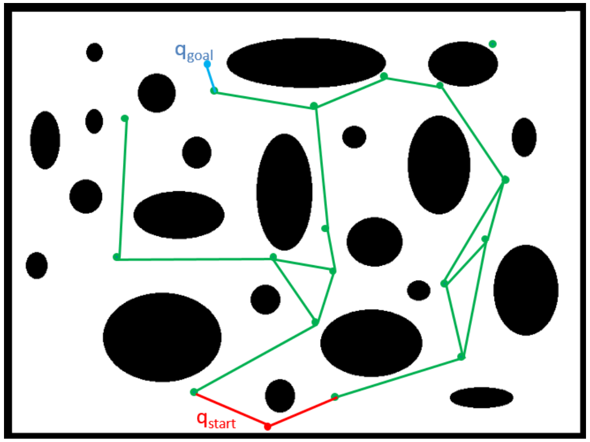
- State space graph: a mathematical representation of a search algorithm
  - For every search problem, there's a corresponding state space graph
  - Connectivity between nodes in the graph is represented by (directed or undirected) edges



Ridiculously tiny search graph for a tiny search problem



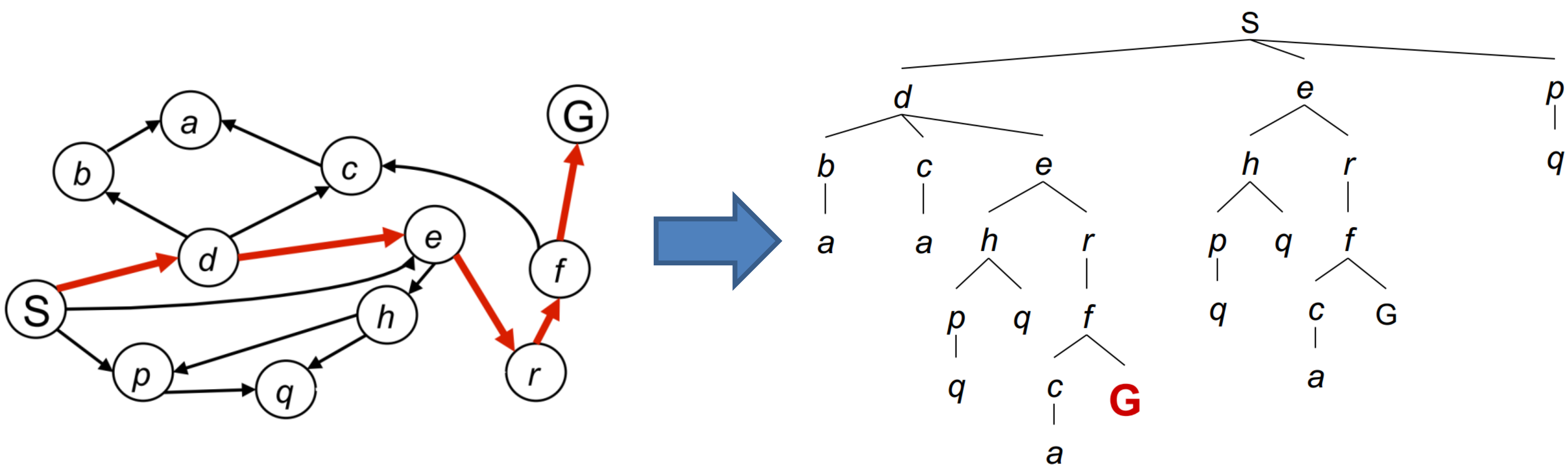
Grid-based graph: use grid as vertices and grid connections as edges



The graph generated by probabilistic roadmap (PRM)

From Graph to Search Tree

- The search always start from start state X_S
  - Searching the graph produces a search tree, this is a “what if” tree of plans and outcomes
  - Back-tracing a node in the search tree gives us a path from the start state to that node
  - For many problems we can never actually build the whole tree, too large or inefficient – we only want to reach the goal node asap.

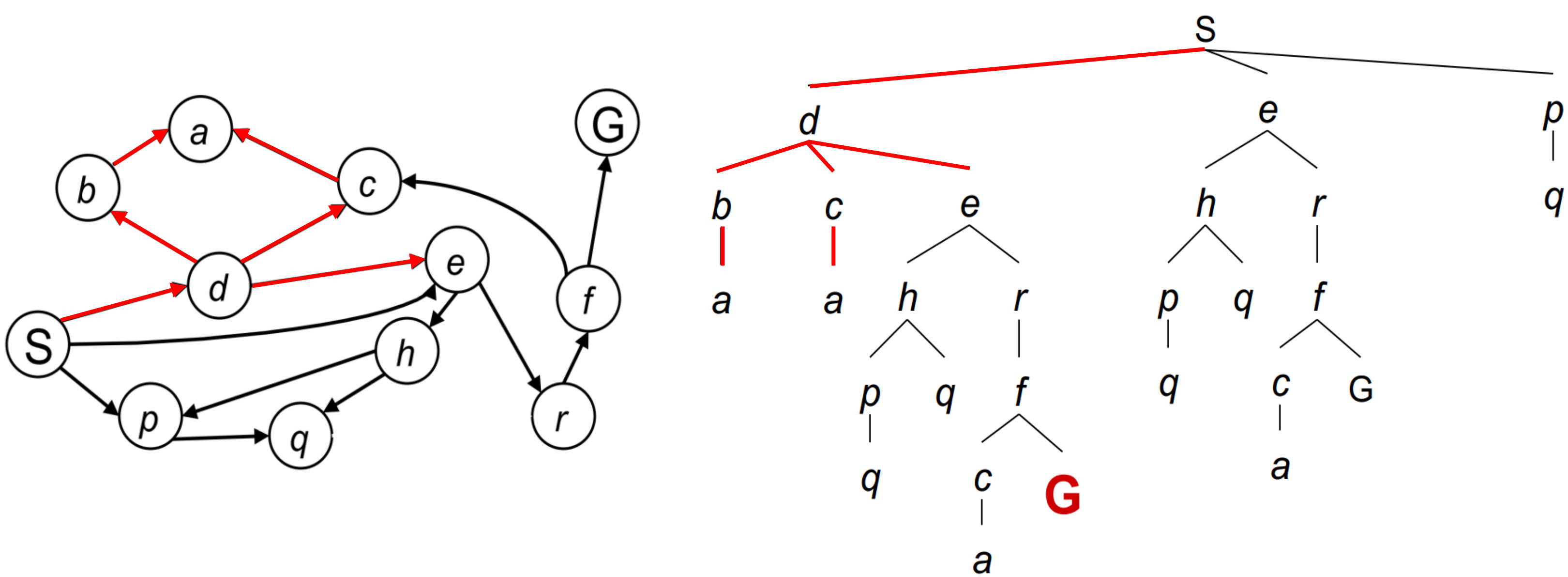


How to Construct a Search Tree?

- Maintain a container to store all the nodes to be visited
  - Intuition: When we "discover" a node, we store it in our "memory". We can only visit one node at a time, but we can teleport to any node that we discover before.
- The container is initialized with the start state X_S
- Loop
  - Remove a node from the container according to some pre-defined score function
    - Visit a node
  - Expansion: Obtain all neighbors of the node, and push them into the container
    - Discover all its neighbors
- End Loop
- Question 1: When to end the loop?
  - Possible option: End the loop when the container is empty
- Question 2: What if the graph is cyclic?
  - When a node is removed (visited) from the container, it should never be added back to the container again
- Question 3: In what way to remove the right node such that the goal state can be reached as soon as possible, which results in less expansion of the graph node.

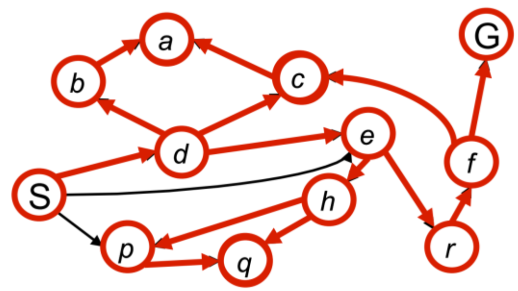
Depth First Search (DFS)

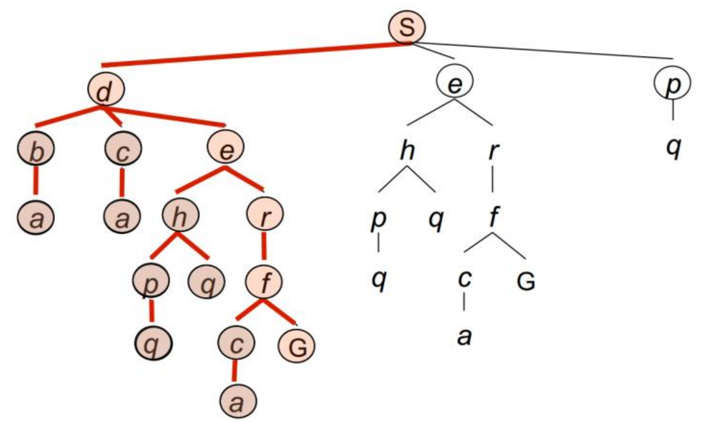
- Strategy: remove (visit) the deepest node in the container



Depth First Search (DFS)

- Implementation: maintain a last in first out (LIFO) container (i.e. stack)





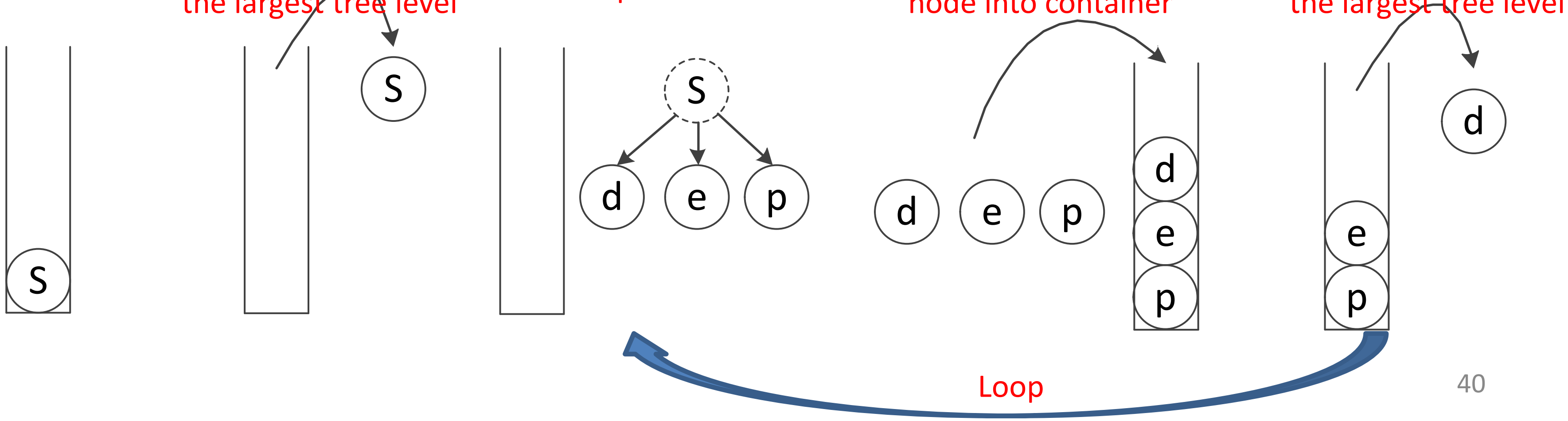
Init container

Remove node with the largest tree level

Expansion

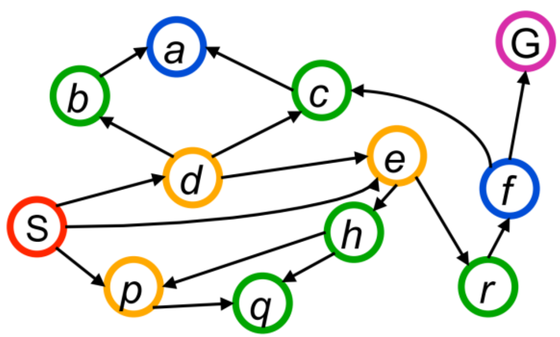
Add children of visited node into container

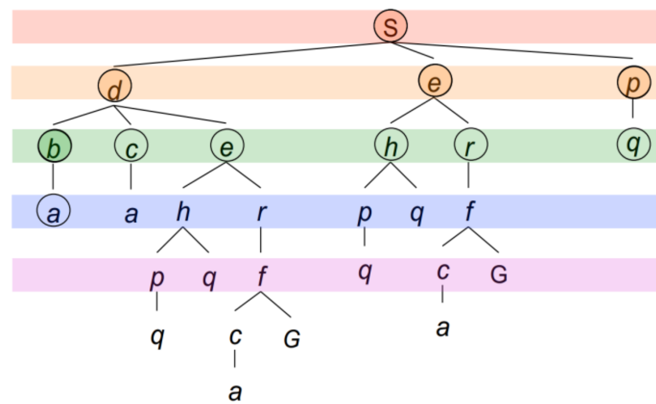
Remove node with the largest tree level



Breadth First Search (BFS)

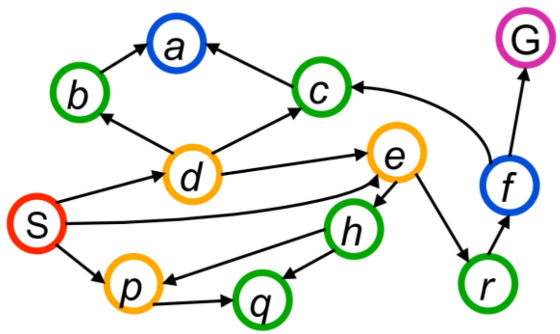
- Strategy: remove (visit) the shallowest node in the container

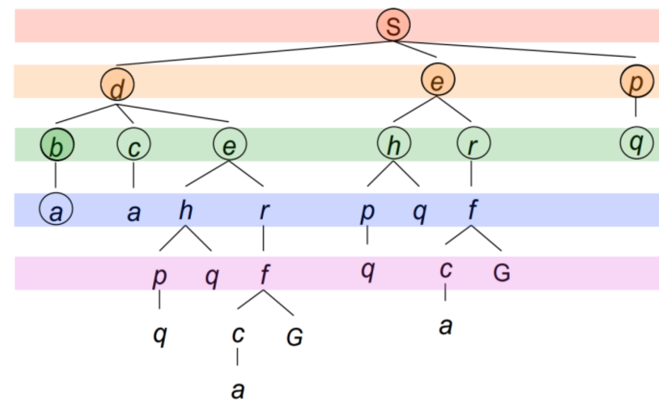


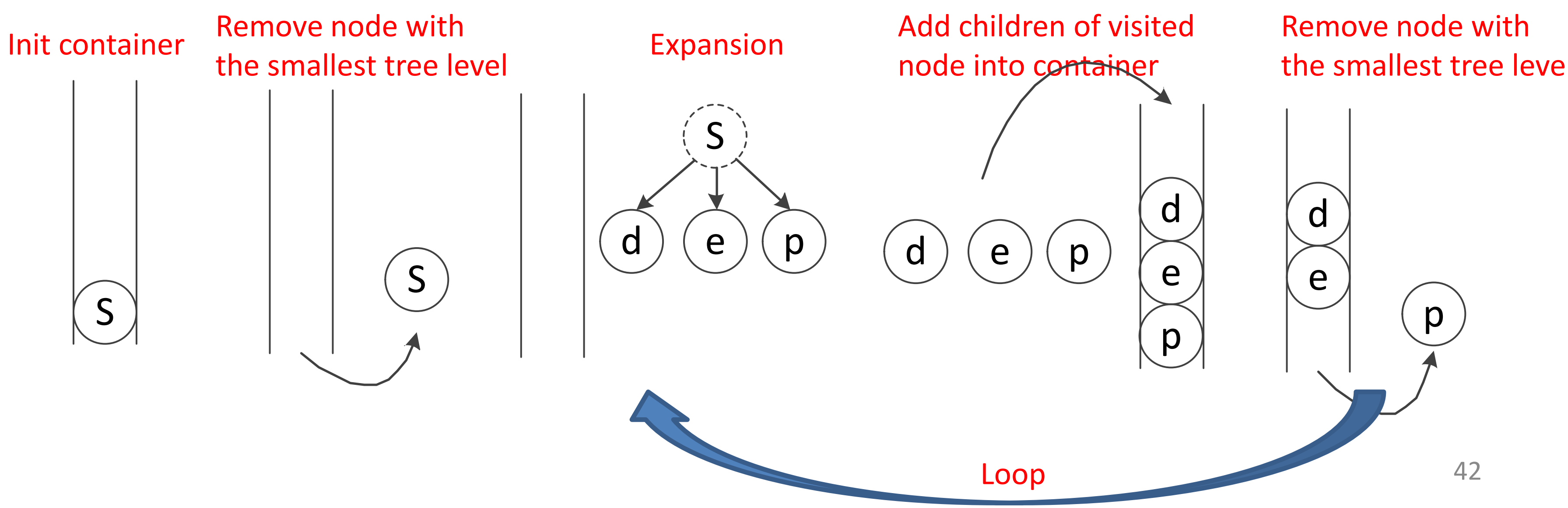


Breadth First Search (BFS)

- Implementation: maintain a first in first out (FIFO) container (i.e. queue)

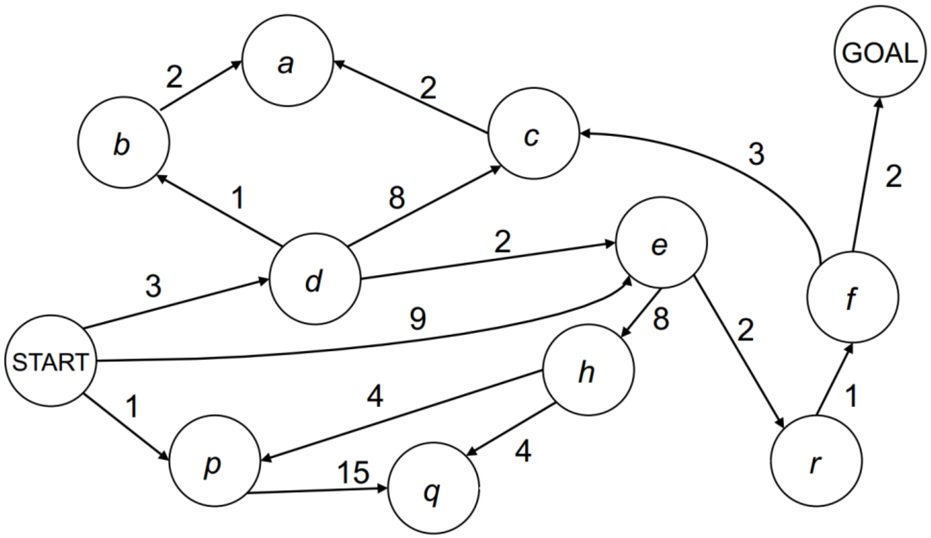






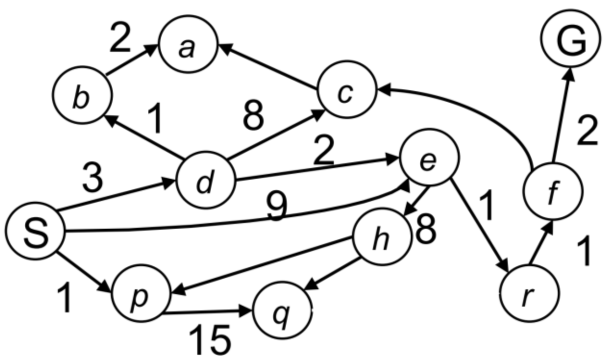
Costs on Actions

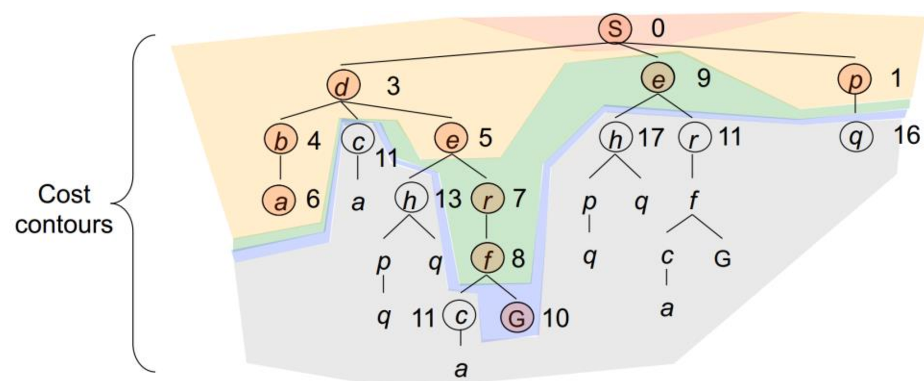
- A practical search problem has a cost "C" from a node to its neighbor
  - Length, time, energy, etc.
- When all weight are 1, BFS finds the least-cost path with minimal steps
- For general cases, how to find the least-cost path as soon as possible?



Dijkstra’s Algorithm

- Strategy: remove (visit) the node with cheapest accumulated cost g(n)
  - g(n): The current best estimates of the accumulated cost from the start state to node “n”
  - Update the accumulated costs g(m) for all unvisited neighbors “m” of node “n”
  - A node that has been visited is guaranteed to have the smallest cost from the start state

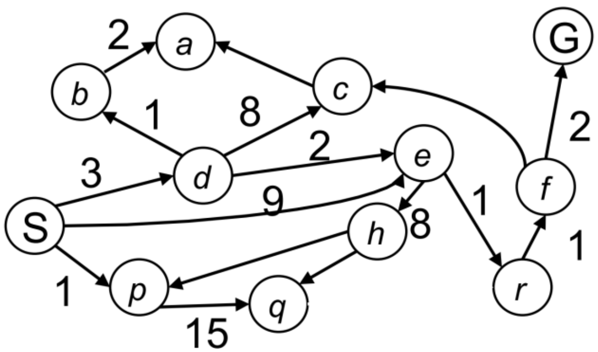


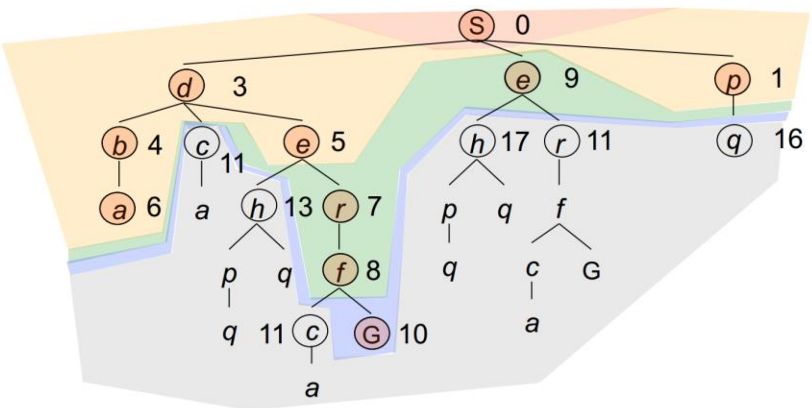


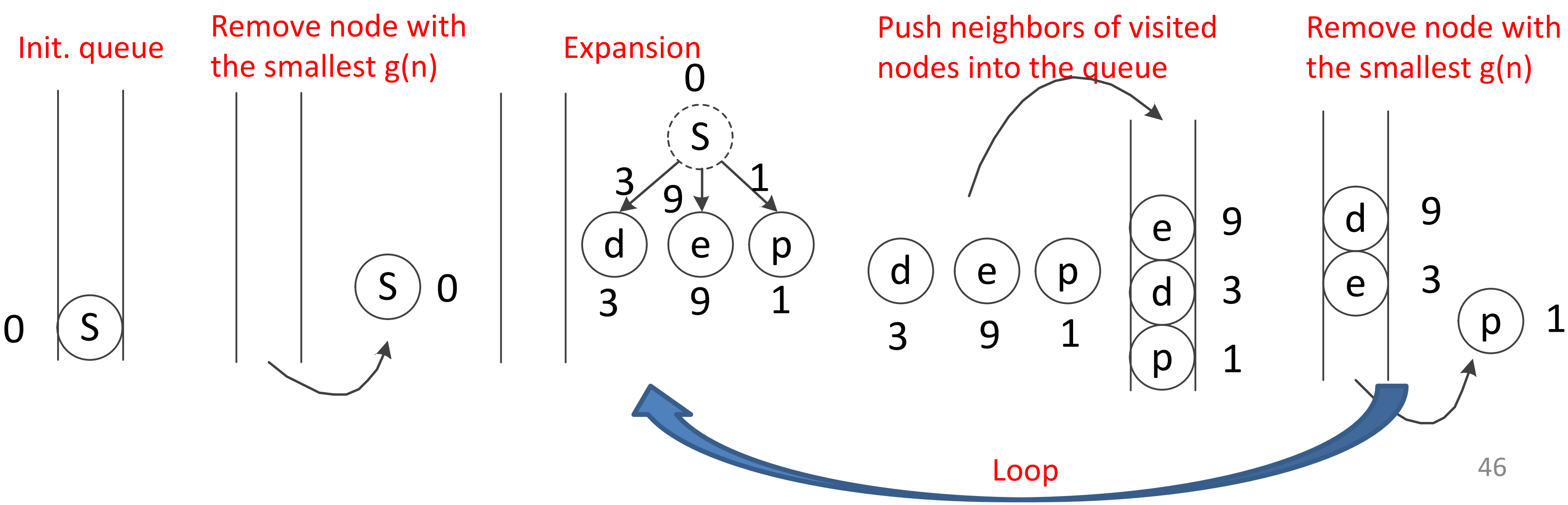
Dijkstra's Algorithm

- Maintain a priority queue to store all the nodes to be visited
- The priority queue is initialized with the start state X_S
- Assign $g(X_S)=0$, and g(n)=infinite for all other nodes in the graph
- Loop
  - If the queue is empty, return FALSE; break;
  - Remove the node "n" with the lowest g(n) from the priority queue
  - Mark node "n" as visited
  - If the node "n" is the goal state, return TRUE; break;
  - For all unvisited neighbors "m" of node "n"
    - If g(m) = infinite
      - Push node "m" into the queue
    - If $g(m) > g(n) + C_{nm}$
      - $g(m)= g(n) + C_{nm}$
  - end
- End Loop

Dijkstra’s Algorithm

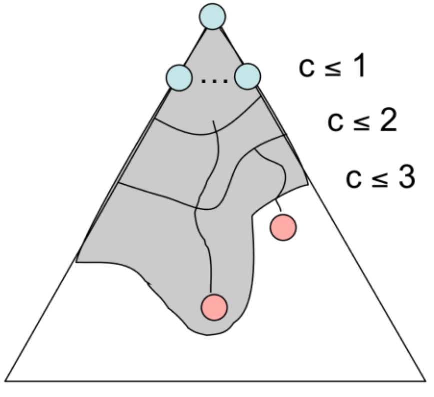


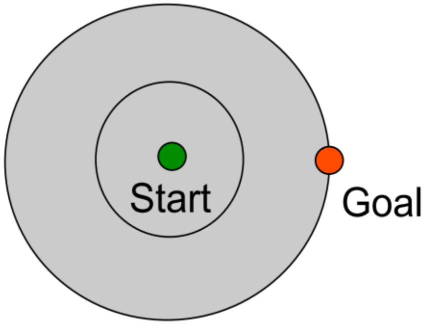




Issues of Dijkstra's Algorithm

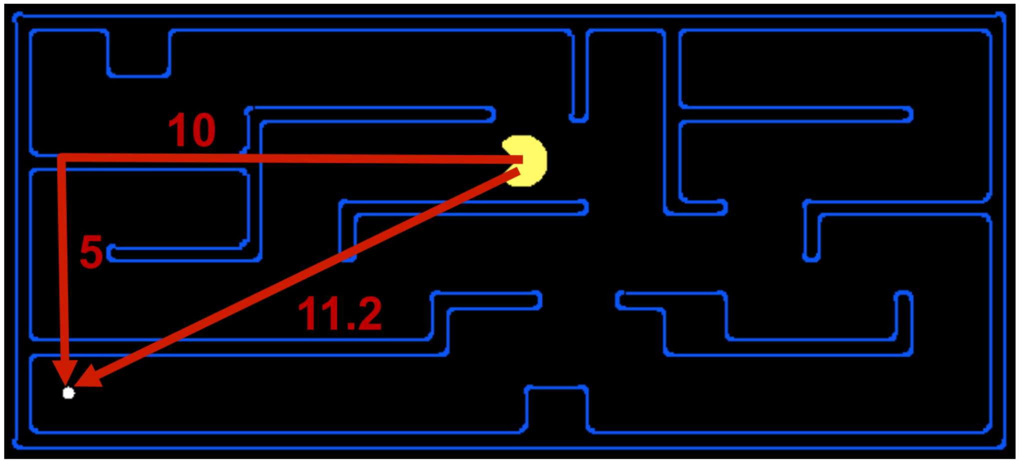
- The good:
  - Complete and optimal
- The bad:
  - Can only see the cost *accumulated so far* (i.e. the uniform cost), thus exploring next state in every "direction"
  - No information about goal location





Search Heuristics

- Overcome the shortcomings of uniform cost search by inferring the least cost to goal (i.e. goal cost)
- Designed for particular search problem
- Examples: Manhattan distance VS. Euclidean distance



A* Search: Combining Dijkstra's and a Heuristic

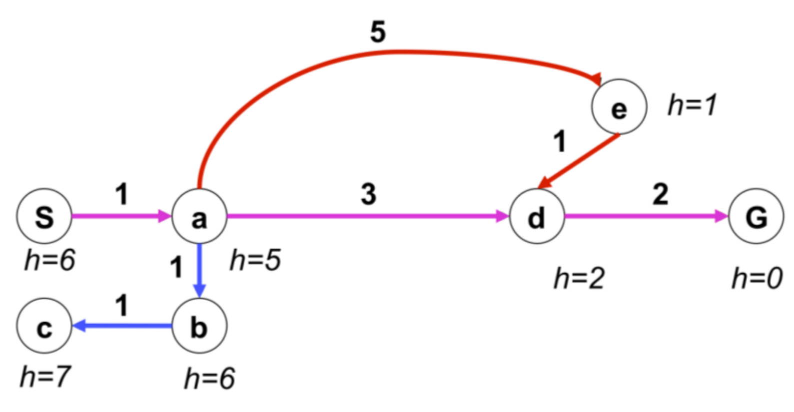
- Accumulated cost
  - g(n): The current best estimates of the accumulated cost from the start state to node "n"
- Heuristic
  - h(n): The estimated least cost from node n to goal state (i.e. goal cost)
- The least estimated cost from start state to goal state passing through node "n" is f(n) = g(n) + h(n)
- Strategy: remove (visit) the node with cheapest f(n) = g(n) + h(n)
  - Update the accumulated costs g(m) for all unvisited neighbors "m" of node "n"
  - A node that has been visited is guaranteed to have the smallest cost from the start state

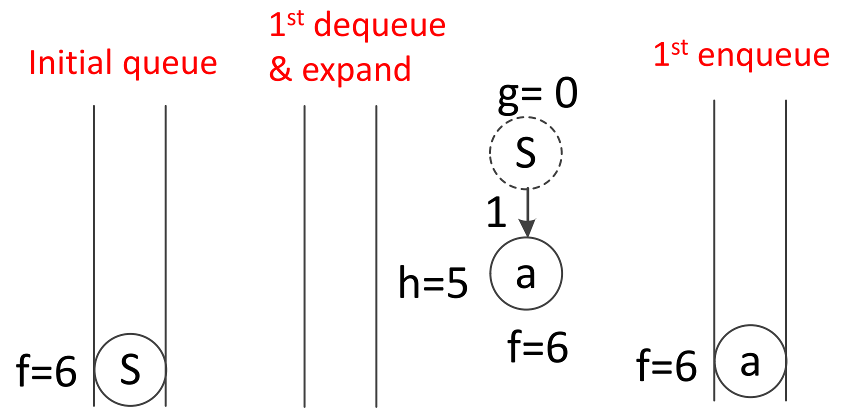
A* Algorithm

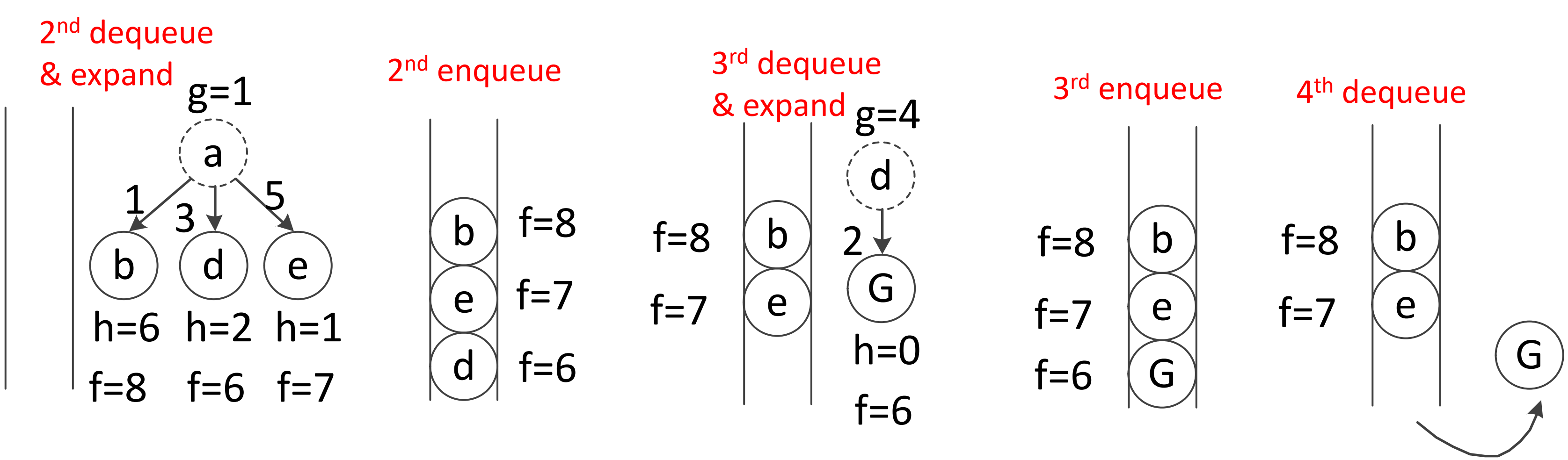
- Maintain a priority queue to store all the nodes to be visited
- The heuristic function h(n) for all nodes are pre-defined
- The priority queue is initialized with the start state X_S
- Assign $g(X_S)=0$, and g(n)=infinite for all other nodes in the graph
- Loop
  - If the queue is empty, return FALSE; break;
  - Remove the node "n" with the lowest $f(n)=g(n)+h(n)$ from the priority queue
  - Mark node "n" as visited
  - If the node "n" is the goal state, return TRUE; break;
  - For all unvisited neighbors "m" of node "n"
    - If g(m) = infinite
      - Push node "m" into the queue
    - If $g(m) > g(n) + C_{nm}$
      - $g(m)= g(n) + C_{nm}$
  - end
- End Loop

Only difference comparing to Dijkstra's algorithm

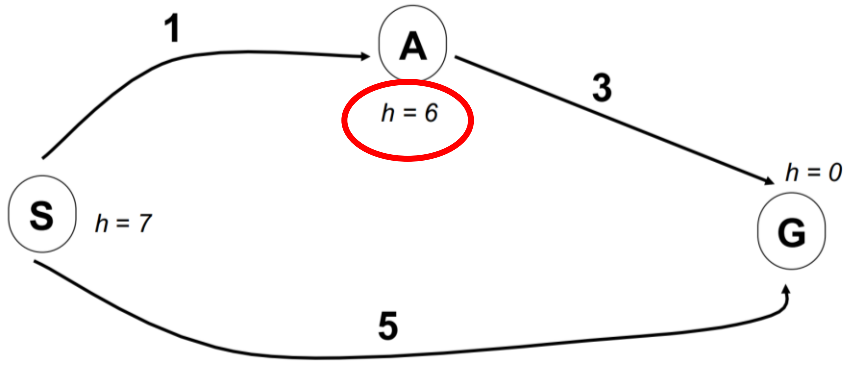
A* Search Example







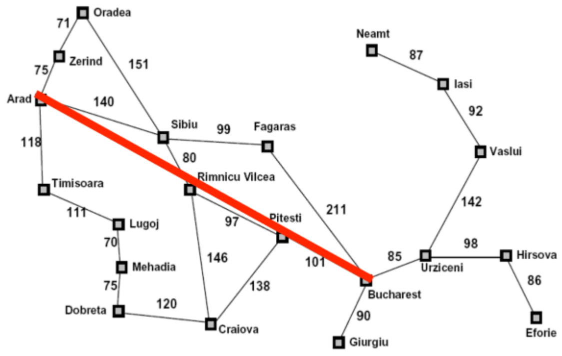
Is A* Optimal?

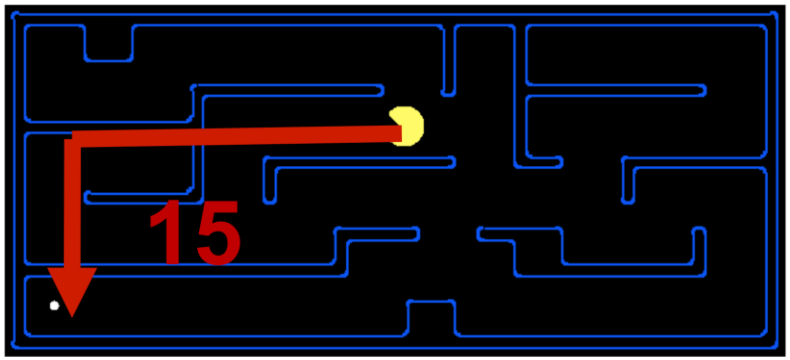


- What went wrong?
- For node A: actual least cost to goal (i.e. goal cost) < estimated least cost to goal (i.e. heuristic)
- We need the estimate to be less than actual least cost to goal (i.e. goal cost) for all nodes!

Admissible Heuristics

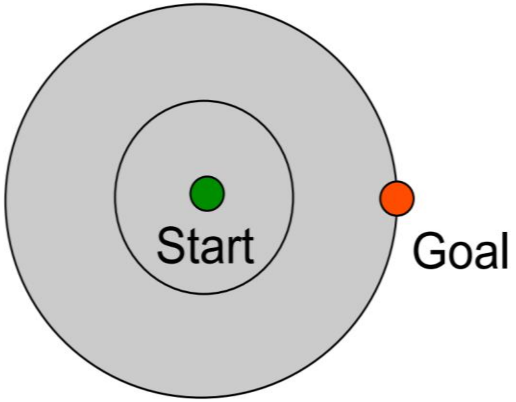
- A Heuristic h is admissible (optimistic) if:
  - h(n) < h*(n) for all node "n", where h*(n) is the true least cost to goal from node "n"
- If the heuristic is admissible, the A* search is optimal
- Coming up with admissible heuristics is most of what's involved in using A* in practice.
- Example:

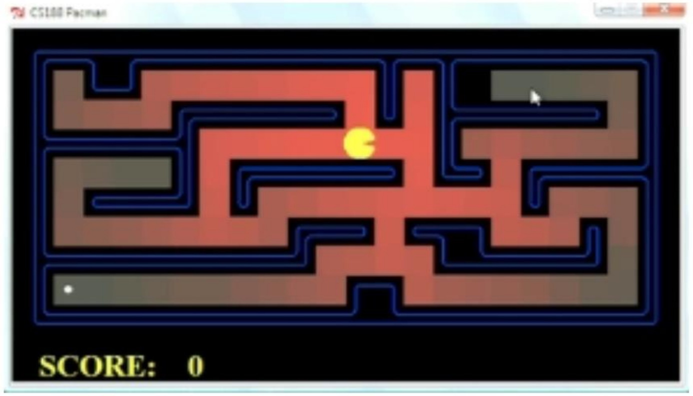




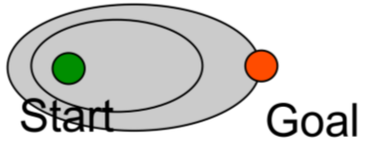
Dijkstra’s VS A*

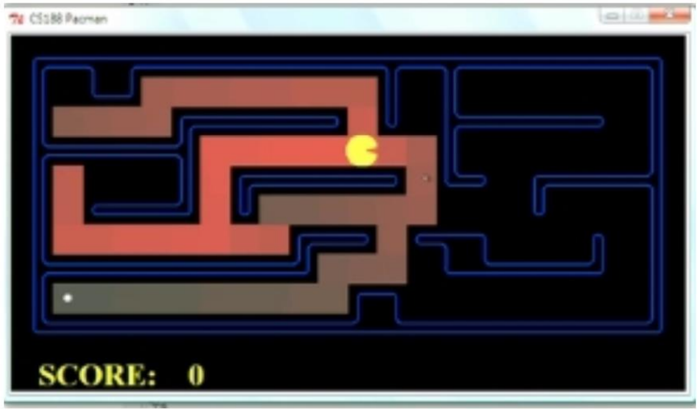
- Dijkstra’s algorithm visits in all directions





- A* visits mainly towards the goal, but does not hedge its bets to ensure optimality



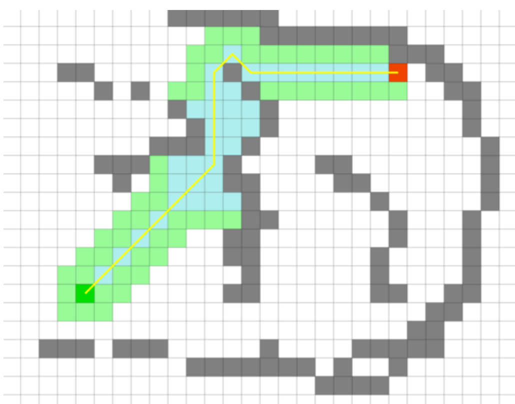


Outline

- Configuration space obstacle
- Search-based methods
  - General graph search: DFS, BFS
  - A* search
- Sampling-based methods
  - Probabilistic roadmap (PRM)
  - Rapidly exploring random tree (RRT)

Grid-based graph:
use grid as vertices and grid connections as edges

RRT example
See more examples at http://ompl.kavrakilab.org/gallery.html



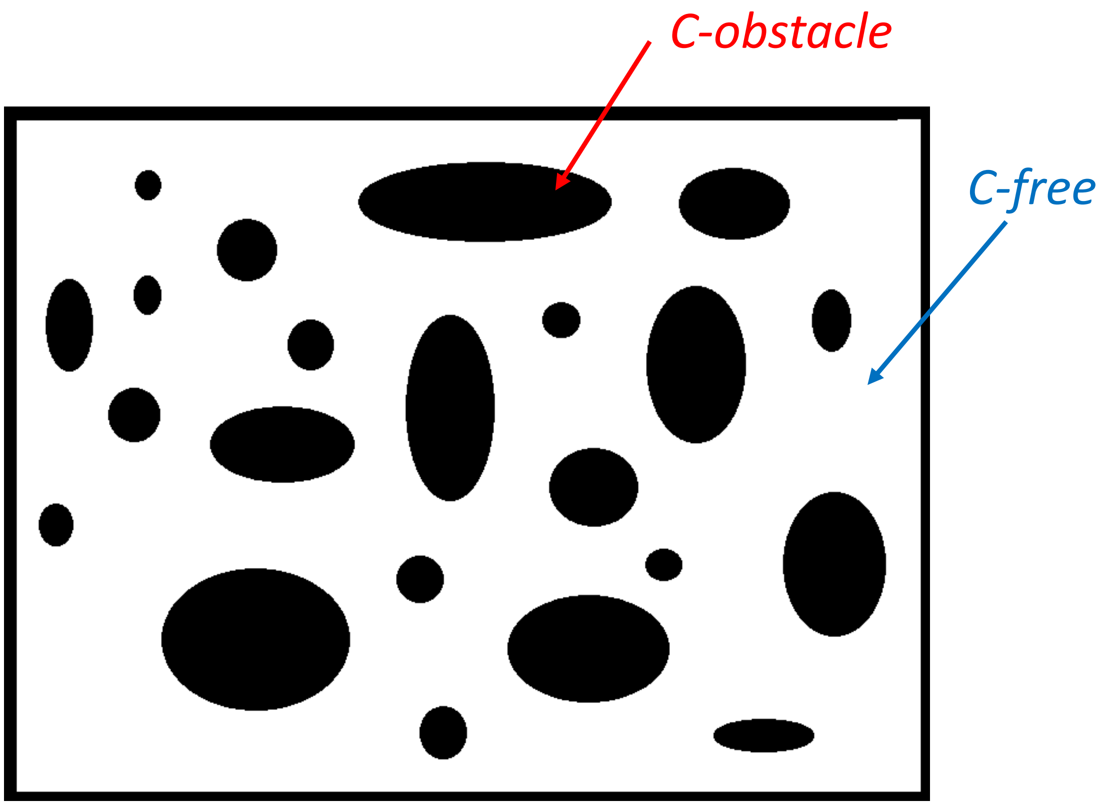
A* search example
Try more search-based method at http://qiao.github.io/PathFinding.js/visual/

Probabilistic Roadmap (PRM)

- Basic Idea
  - Build a **graph** to characterizes the free configuration space in probabilistic manner, and then use graph search algorithm to find a path
- Algorithm
  - Initialize set of points with q_{start} and q_{goal}
  - Randomly sample points in configuration space
  - Connect nearby points if they can be reached from each other
  - Find path from q_{start} to q_{goal} in the graph
- Step by step illustration as follows

Probabilistic Roadmap (PRM)

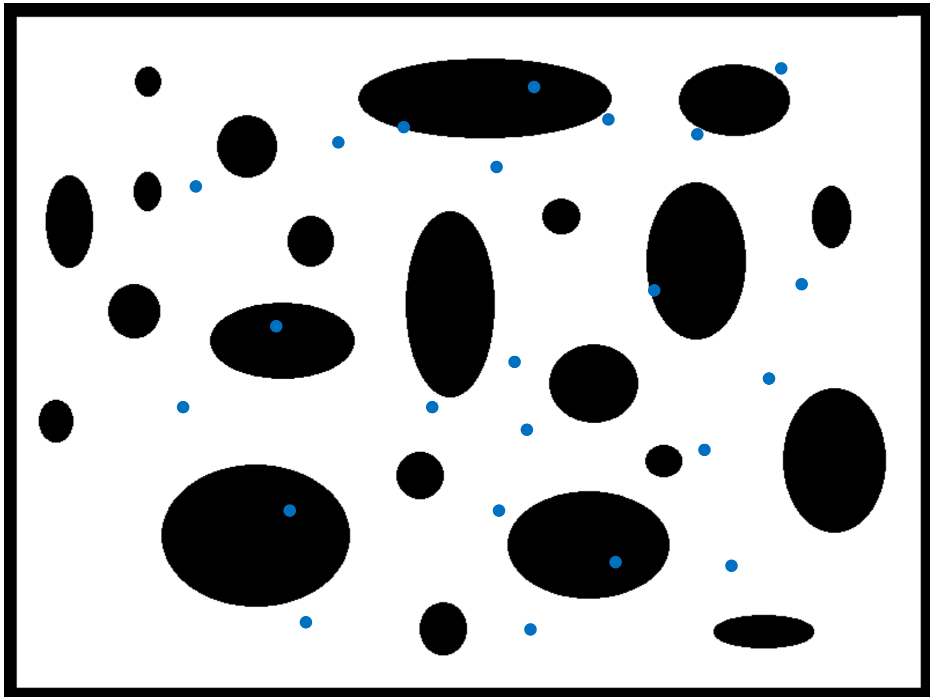
- Free space and obstacle space



C-space

Probabilistic Roadmap (PRM)

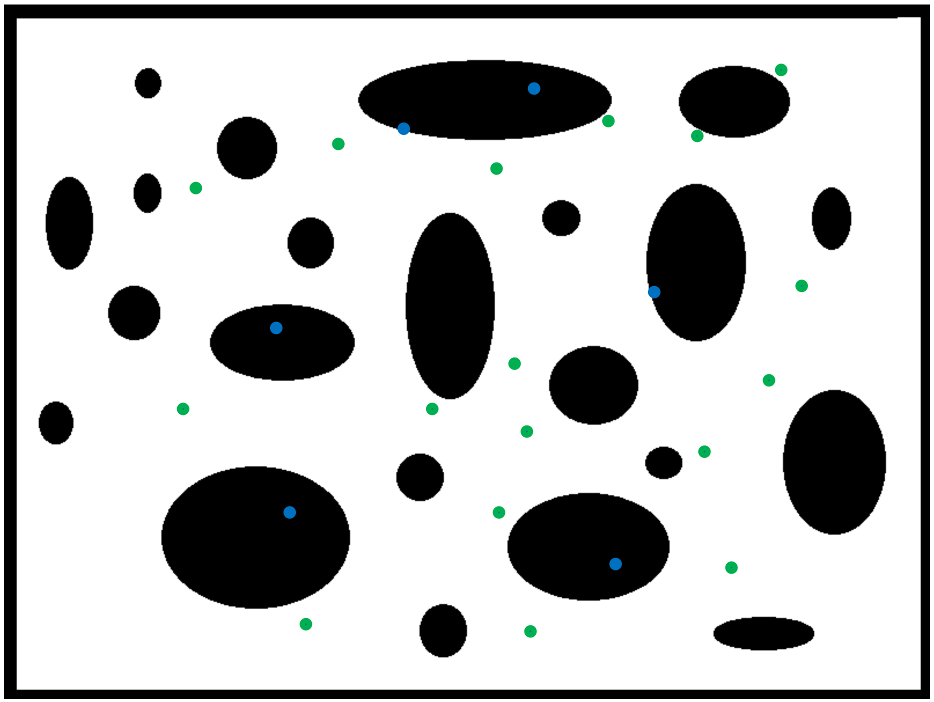
- Configurations are sampled by picking each coordinate at random.



C-space

Probabilistic Roadmap (PRM)

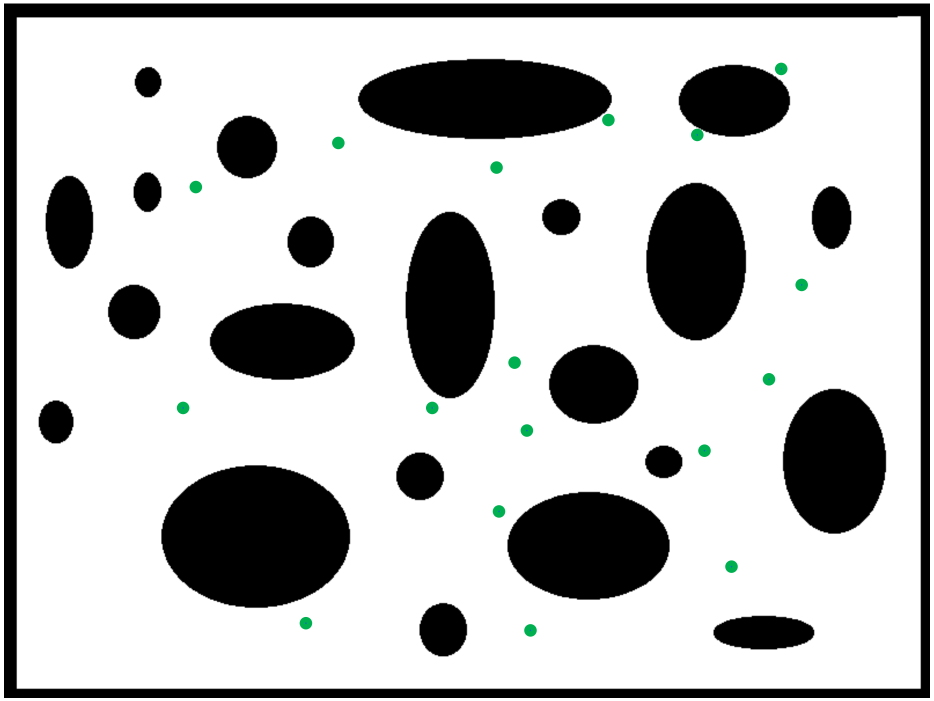
- Sampled configurations are tested for collision.



C-space

Probabilistic Roadmap (PRM)

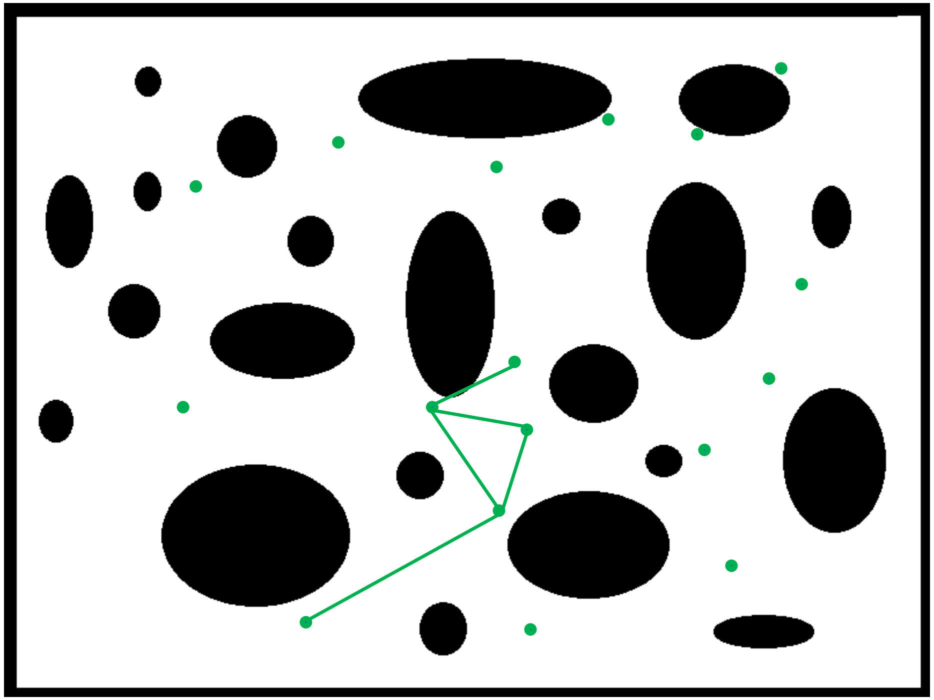
- The collision-free configurations are retained as milestones.



C-space

Probabilistic Roadmap (PRM)

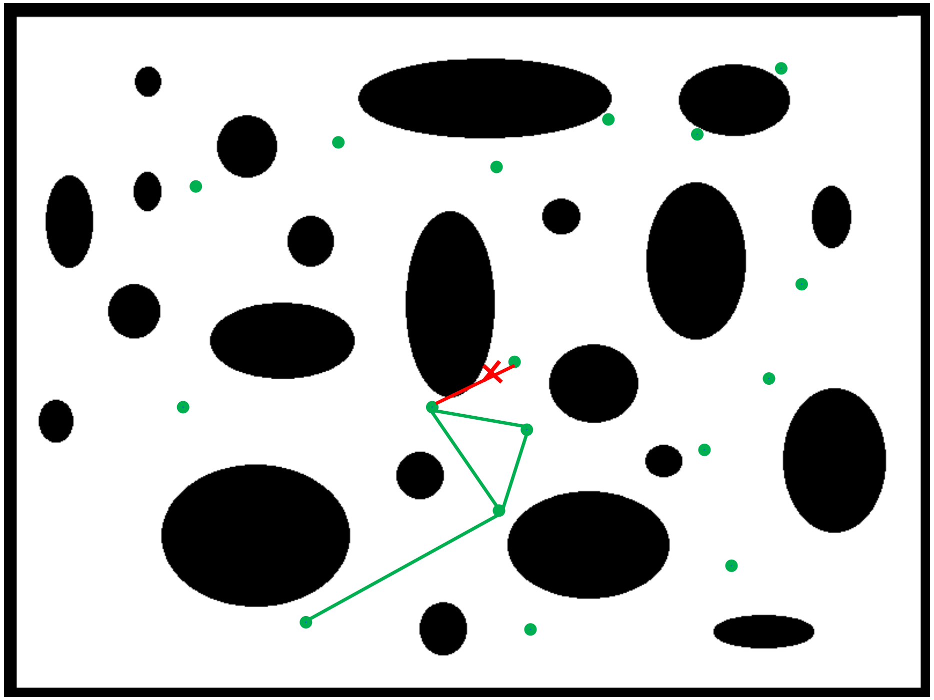
- Each milestone is linked by straight paths to its nearest neighbors.



C-space

Probabilistic Roadmap (PRM)

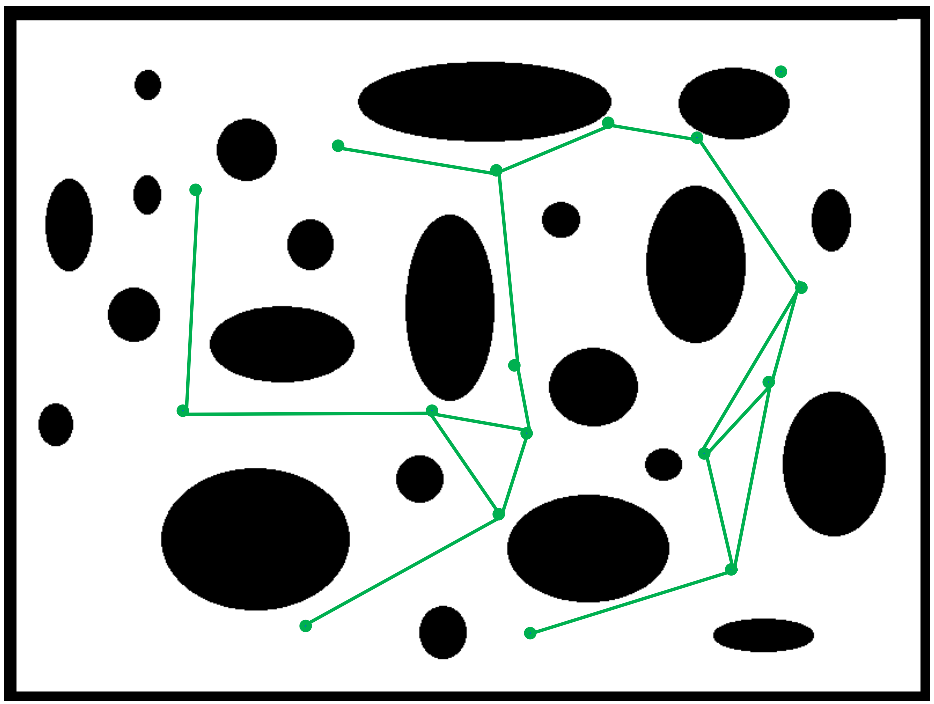
- Eliminate collision links.



C-space

Probabilistic Roadmap (PRM)

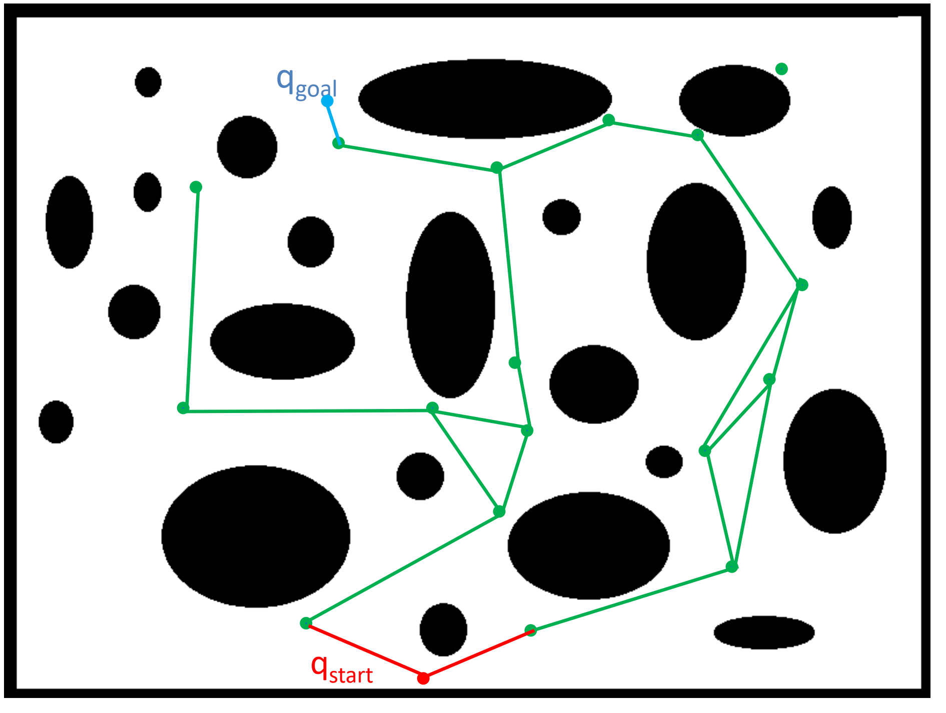
- The collision-free links are retained as local paths to form the PRM.



C-space

Probabilistic Roadmap (PRM)

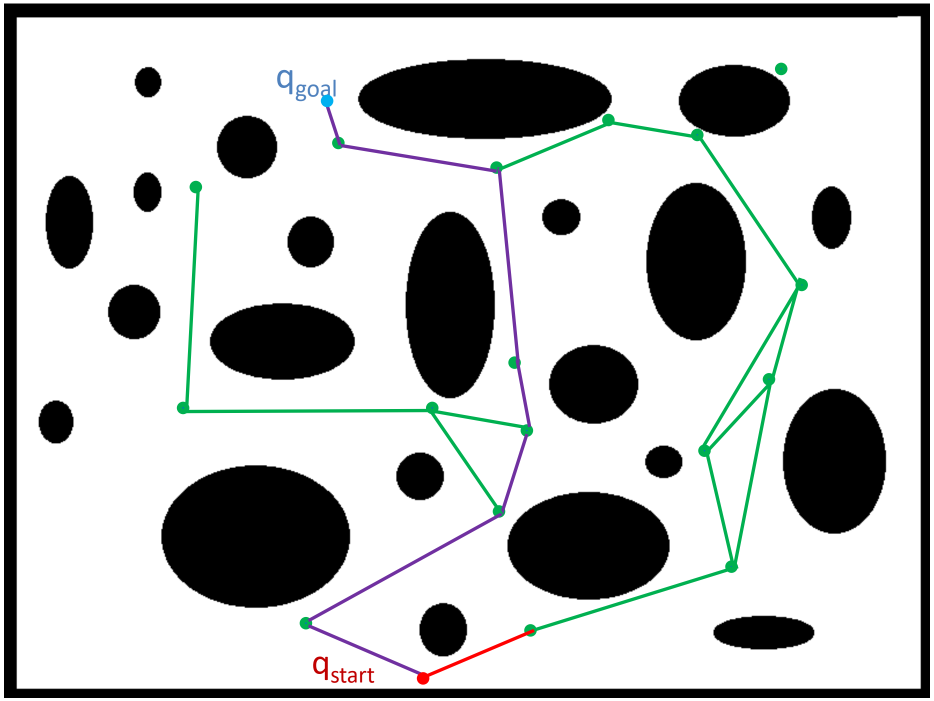
- Connect the start and goal point to the roadmap.



C-space

Probabilistic Roadmap (PRM)

- Search the roadmap for a path from start to goal point (e.g. A* algorithm).



C-space

PRM’s Pros and Cons

- Pros:
  - Probabilistically complete: i.e., with probability one, if run for long enough the graph will contain a solution path if one exists.
  - Can cope with high-dimensional system
- Cons:
  - Collision detection takes majority of time
  - Suboptimal solution if only limited samples are given
  - Build graph over C-space but no particular focus on generating a path

Rapidly exploring Random Trees (RRT)

- Basic Idea
  - Starting from the start configuration q_{start}, build up a **tree** through generating "next configuration"
- Algorithm

Algorithm BuildRRT

Input: Start configuration q_{start}, number of vertices in RRT K

Output: RRT T

L1: G.init(q_{start})

L2: **for** k = 1 **to** K

L3: $q_{rand} \leftarrow$ RAND_CONF();

L4: $q_{near} \leftarrow$ NEAREST_VERTEX(q_{rand}, T);

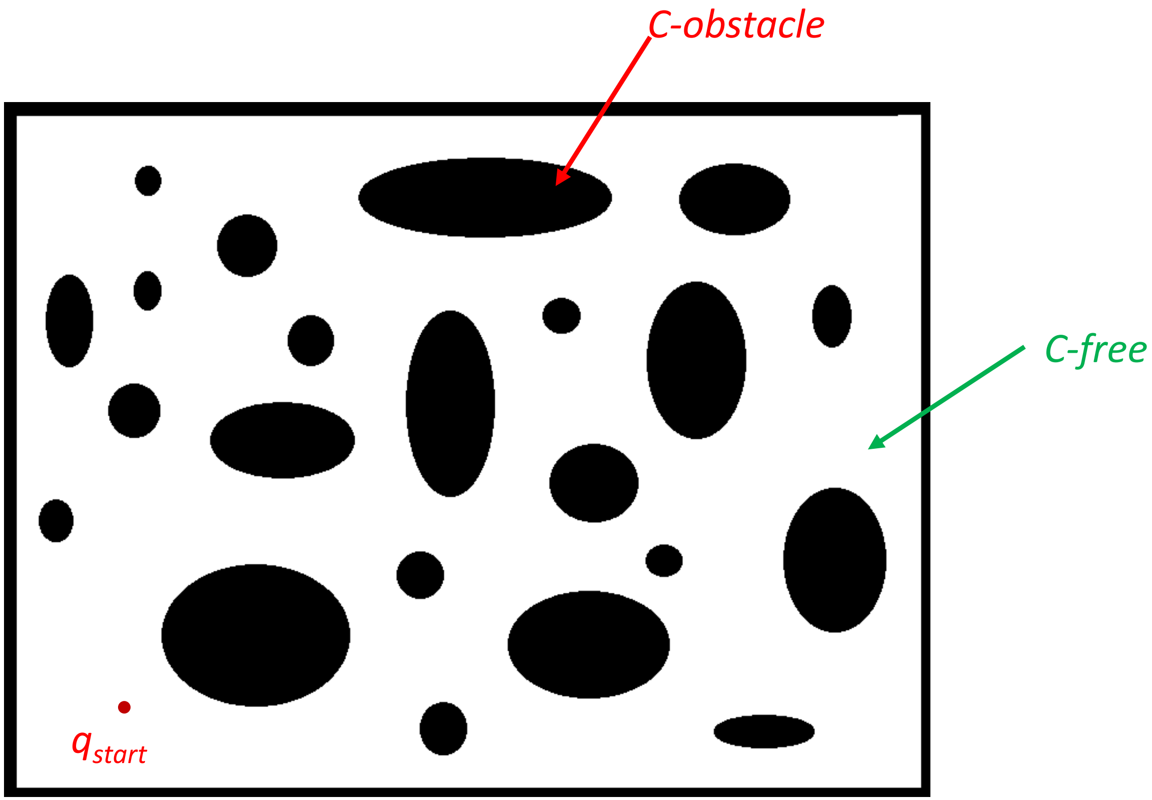
L5: $q_{new} \leftarrow$ NEW_CONF(q_{near}, q_{rand});

L6: T.add_vertex(q_{new}); T.add_edge(q_{near}, q_{new})

L7: **return** G

Rapidly exploring Random Trees (RRT)

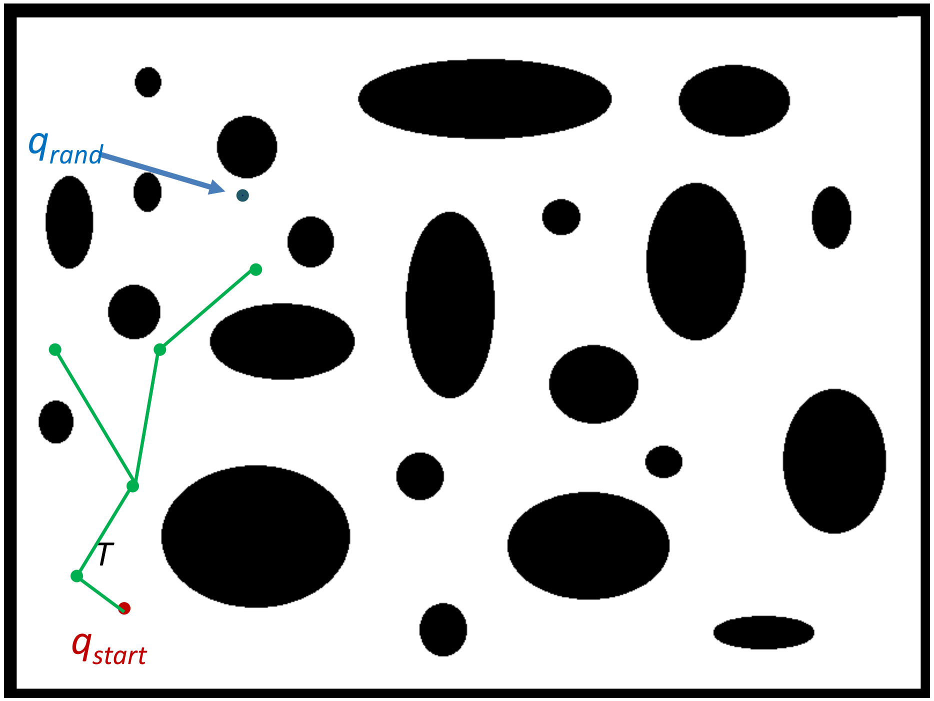
- L1: T.init(q_{start}); Initialize



C-space

Rapidly exploring Random Trees (RRT)

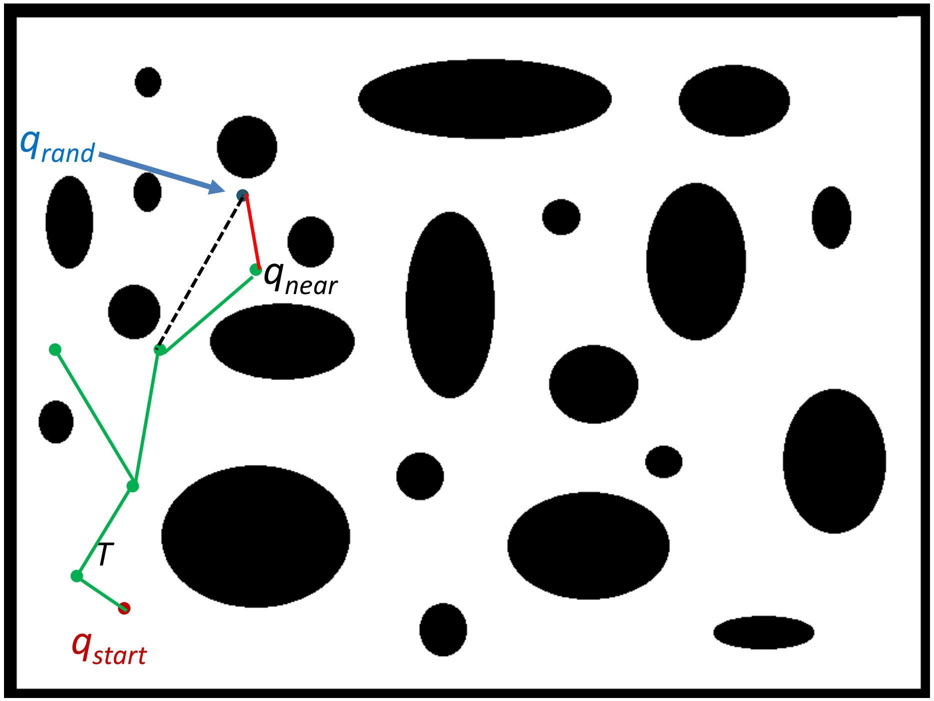
- L3: $q_{rand} \leftarrow$ RAND_CONF(); generate a random configuration
  - q_{rand} is sampled from a uniform distribution on C-space



C-space

Rapidly exploring Random Trees (RRT)

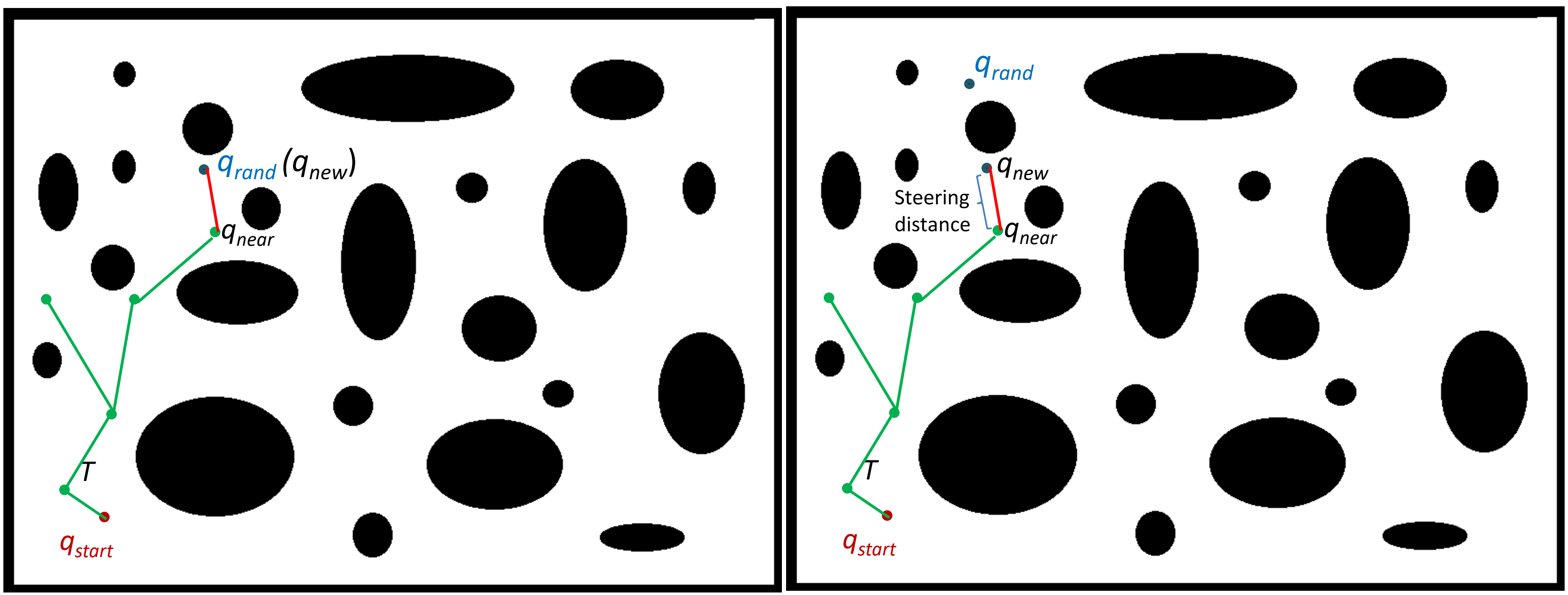
- L4: $q_{near} \leftarrow$ NEAREST_VERTEX(q_{rand}, T); find the nearest configuration
  - Define the proper distance



C-space

Rapidly exploring Random Trees (RRT)

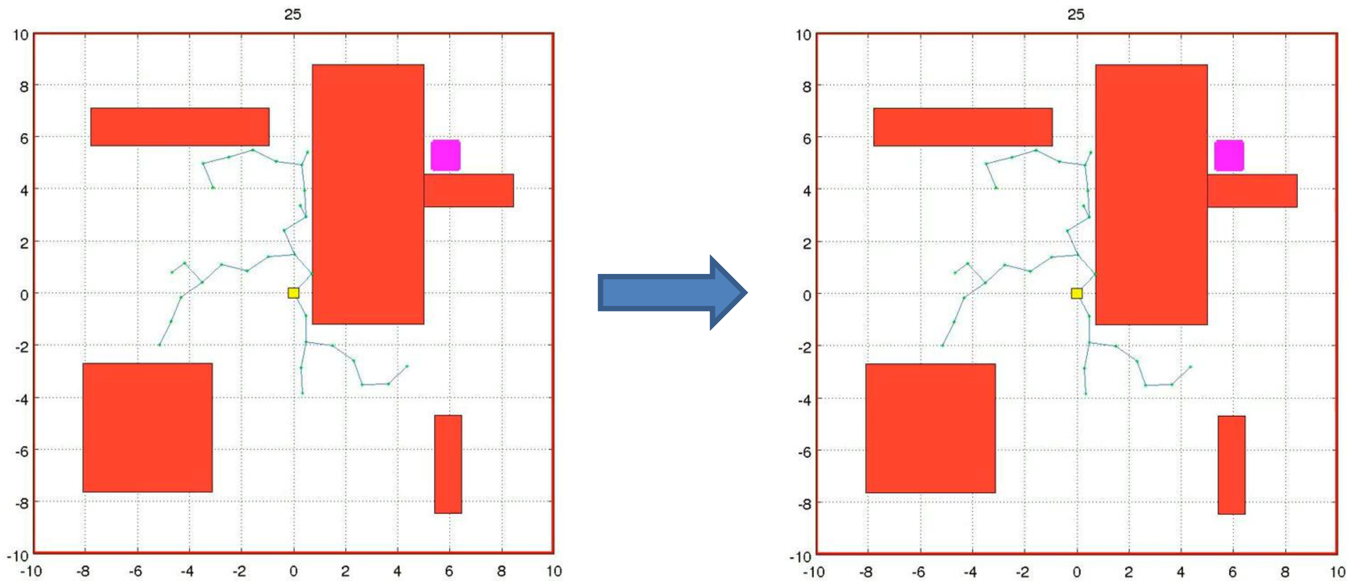
- L5: $q_{new} \leftarrow$ NEW_CONF(q_{near}, q_{rand}); generate a new configuration
- L6: T.add_vertex(q_{new}); T.add_edge(q_{near}, q_{new}); add the new configuration



C-space C-space

RRT*

- Basic Idea
  - RRT is simple, but is prone to be probabilistic incomplete.
  - Add **rewire** function: swap new point in as parent for nearby vertices who can be reached along shorter path through new point than through their original (current) path
  - RRT* is asymptotically optimal.



[Karaman, Sertac, and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning." *The international journal of robotics research* 30.7 (2011): 846-894.]

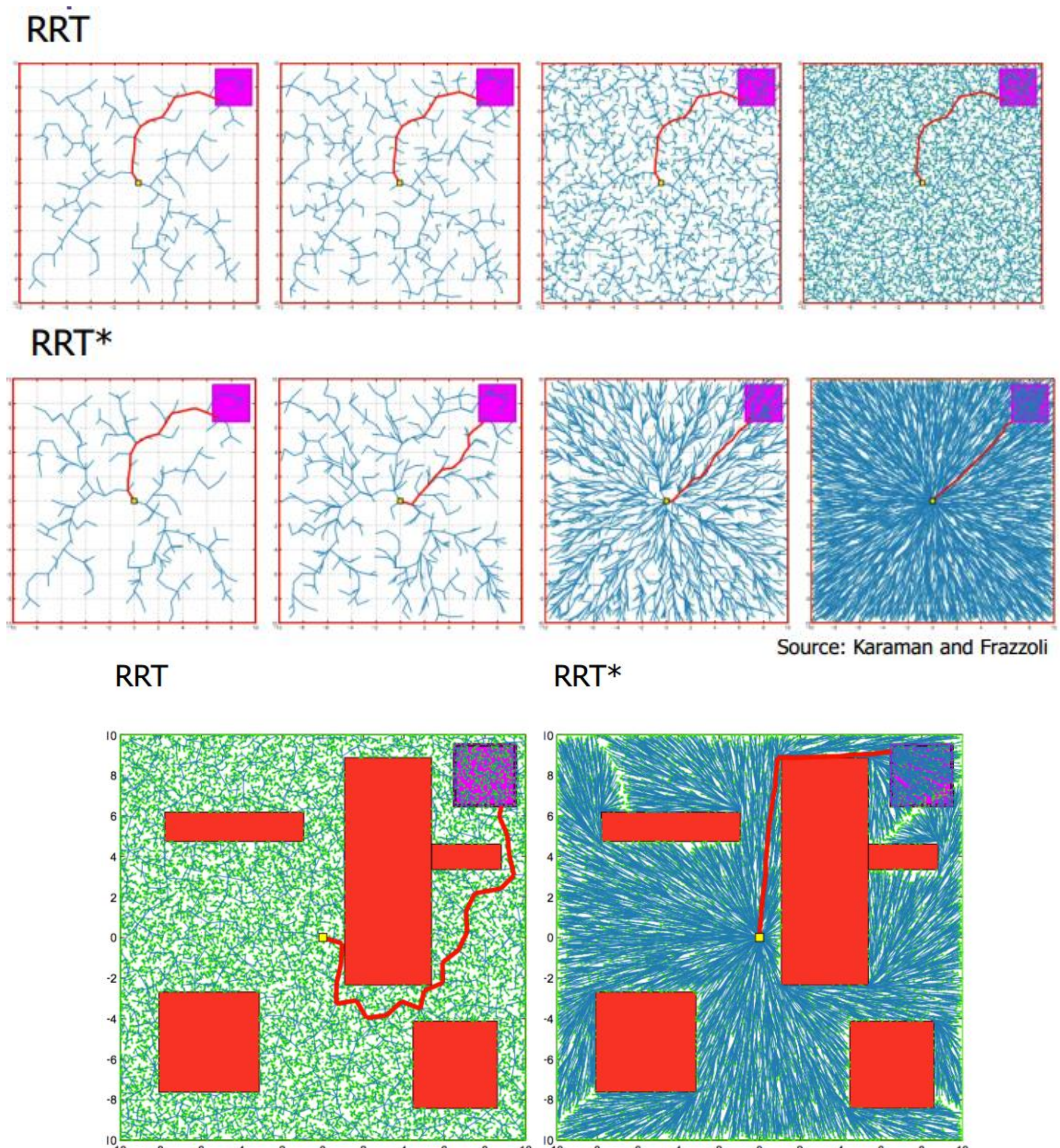
RRT*

- Algorithm

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**
    $X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$;
    $V \leftarrow V \cup \{x_{\text{new}}\}$;
    $x_{\min} \leftarrow x_{\text{nearest}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Connect along a minimum-cost path
      if $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
        $x_{\min} \leftarrow x_{\text{near}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
    $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$;
    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* vs RRT



Logistics

- Project 1, phase 1 due this Friday (02/28)
- Project 1, phase 2 is released (02/25)
  - Due next Friday: 03/07