ELEC 3210
Introduction to Mobile Robotics
Lecture 17

(Machine Learning and Infomation Processing for Robotics)

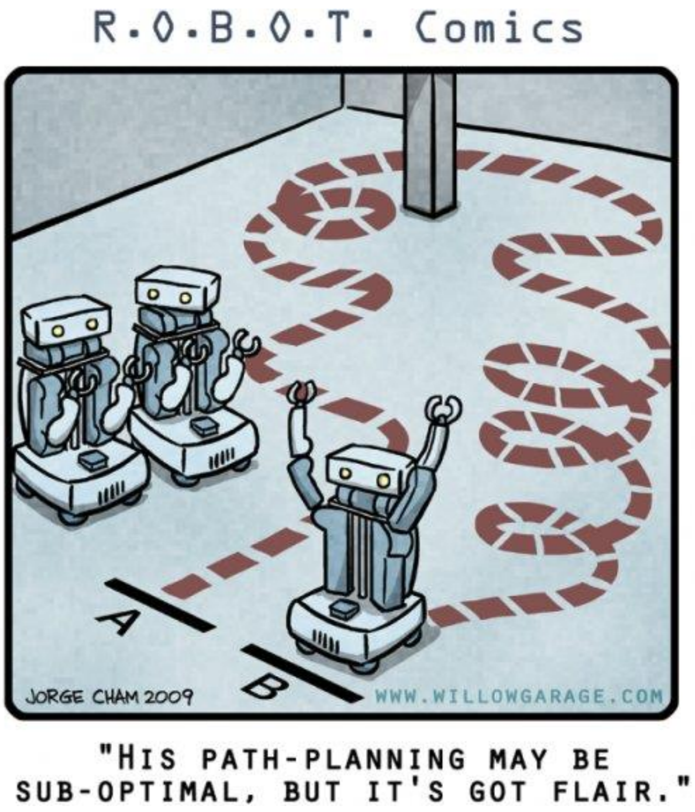
Huan YIN

Research Assistant Professor, Dept. of ECE

eehyin@ust.hk

Recap L16


R.O.B.O.T. Comics
A
B
JORGE CHAM 2009
WWW.WILLOWGARAGE.COM
"HIS PATH-PLANNING MAY BE SUB-OPTIMAL, BUT IT'S GOT FLAIR."

Recap L16

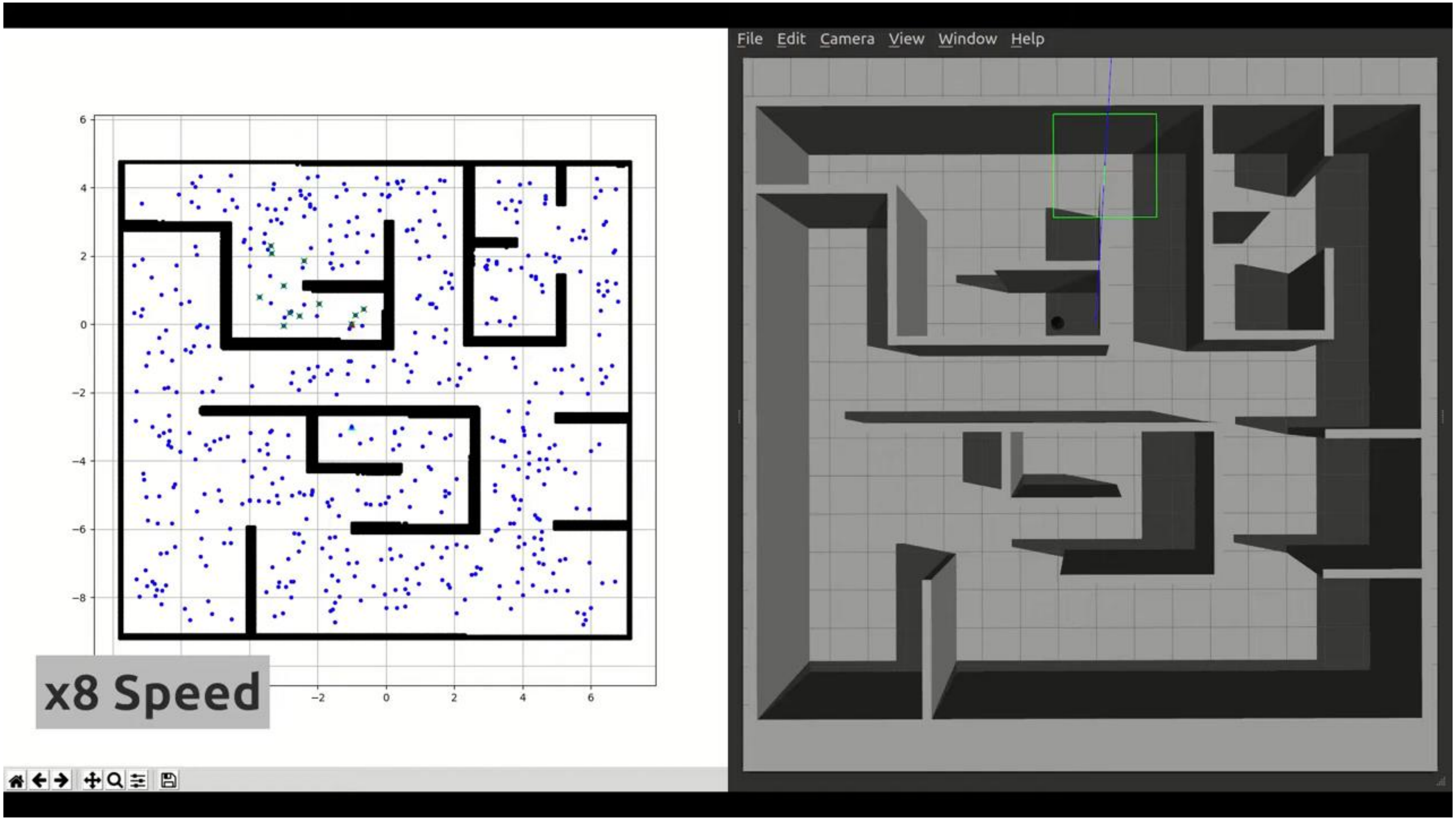
- Basic concepts of Motion Planning
- Planning as graph search problem
- How to construct the graph?

- Combinatorial Planning
  - Resolution Completeness
  - Visibility Graph, Voronoi Diagram, Cell Decomposition
- Sampling-based Planning
  - Probabilistic Completeness
  - Probabilistic road maps (PRM)
  - Rapidly exploring random tree (RRT)

Courtesy:

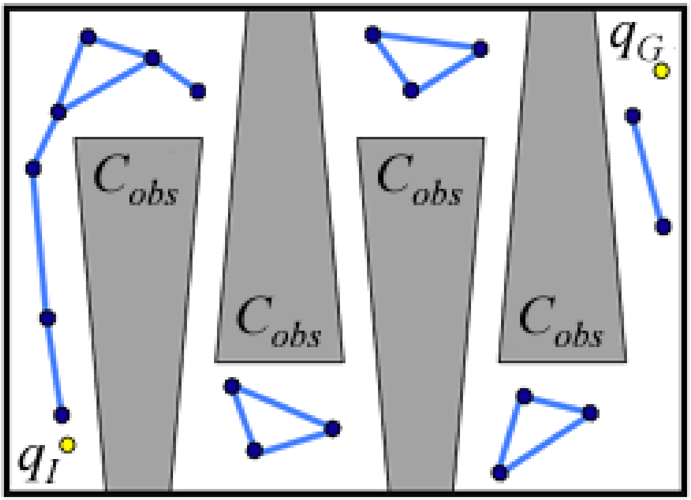
RRT

Probabilistic road maps (PRM)



Probabilistically Complete

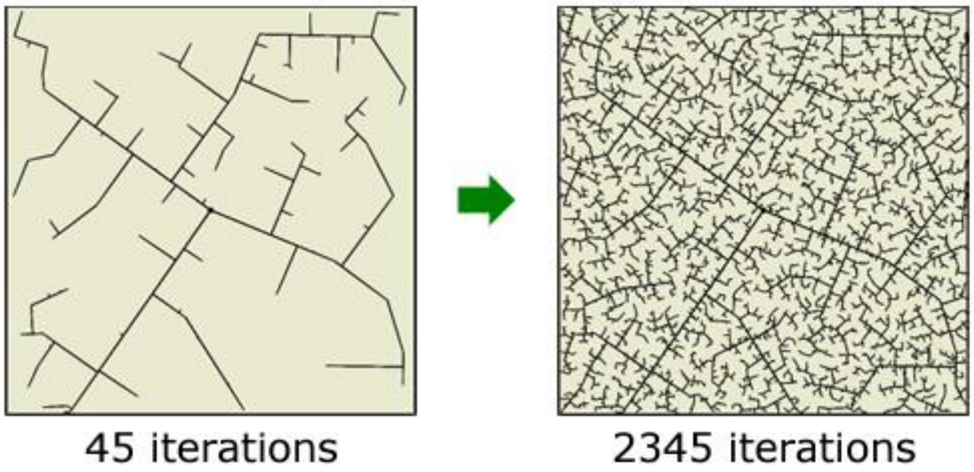
- Do not work well for some problems, narrow passages



Courtesy: Wolfram Burgard

Rapidly Exploring Random Trees

- **Idea:** aggressively probe and explore the C-space by expanding incrementally from an initial configuration
- The explored territory is marked by a tree rooted at the initial



Courtesy: Wolfram Burgard

RRT

- The algorithm: Given C and q_0

Algorithm 1: RRT

G.init(q_0)
repeat
    $q_{rand} \rightarrow$ RANDOM_CONFIG($\mathcal{C}$)
    $q_{near} \leftarrow$ NEAREST(G, q_{rand})
    G.add_edge(q_{near}, q_{rand})
until *condition*

Sample from a bounded region centered around q_0

Courtesy: Wolfram Burgard

RRT

- The algorithm: Given C and q_0

Algorithm 1: RRT

G.init(q_0)
repeat
    $q_{rand} \rightarrow$ RANDOM_CONFIG($\mathcal{C}$)
    $q_{near} \leftarrow$ NEAREST(G, q_{rand})
    G.add_edge(q_{near}, q_{rand})
until *condition*

Finds closest vertex in G using a distance function

Courtesy: Wolfram Burgard

RRT

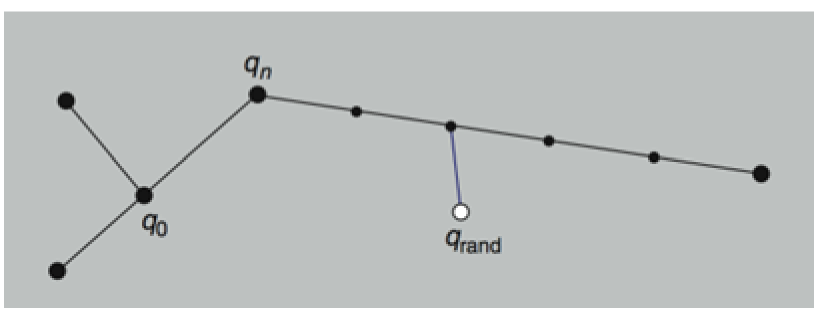
- The algorithm: Given C and q_0

Algorithm 1: RRT

G.init(q_0)
repeat
  $q_{rand} \rightarrow$ RANDOM_CONFIG($\mathcal{C}$)
  $q_{near} \leftarrow$ NEAREST(G, q_{rand})
  G.add_edge(q_{near}, q_{rand})
until *condition*

Several stategies to find q_{near} given the closest vertex on G:

- take closest vertex
- Check intermediate points at regular intervals and split edge at q_{near}



Courtesy: Wolfram Burgard

RRT

- The algorithm: Given C and q_0

Algorithm 1: RRT

G.init(q_0)
repeat
  $q_{rand} \rightarrow$ RANDOM_CONFIG($\mathcal{C}$)
  $q_{near} \leftarrow$ NEAREST(G, q_{rand})
  G.add_edge(q_{near}, q_{rand})
until *condition*

Connect nearest point with random point that travels from q_{near} to q_{rand}

- No collision: add edge
- Collision: new vertex is q_i, as close as possible to C_{obs}

Courtesy: Wolfram Burgard

RRT

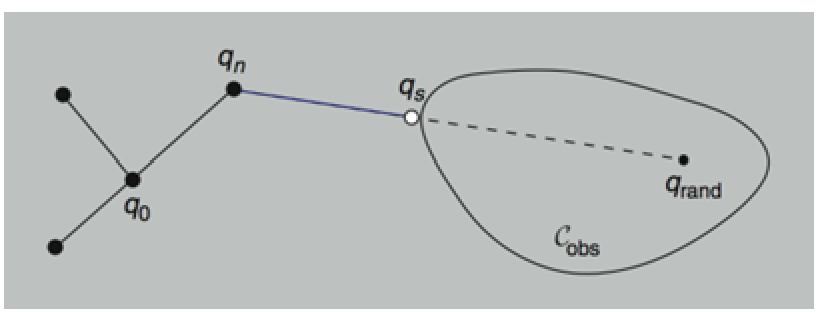
- The algorithm: Given C and q_0

Algorithm 1: RRT

G.init(q_0)
repeat
  $q_{rand} \rightarrow$ RANDOM_CONFIG($\mathcal{C}$)
  $q_{near} \leftarrow$ NEAREST(G, q_{rand})
  G.add_edge(q_{near}, q_{rand})
until *condition*

Connect nearest point with random point that travels from q_{near} to q_{rand}

- No collision: add edge
- Collision: new vertex is q_i, as close as possible to C_{obs}



Courtesy: Wolfram Burgard

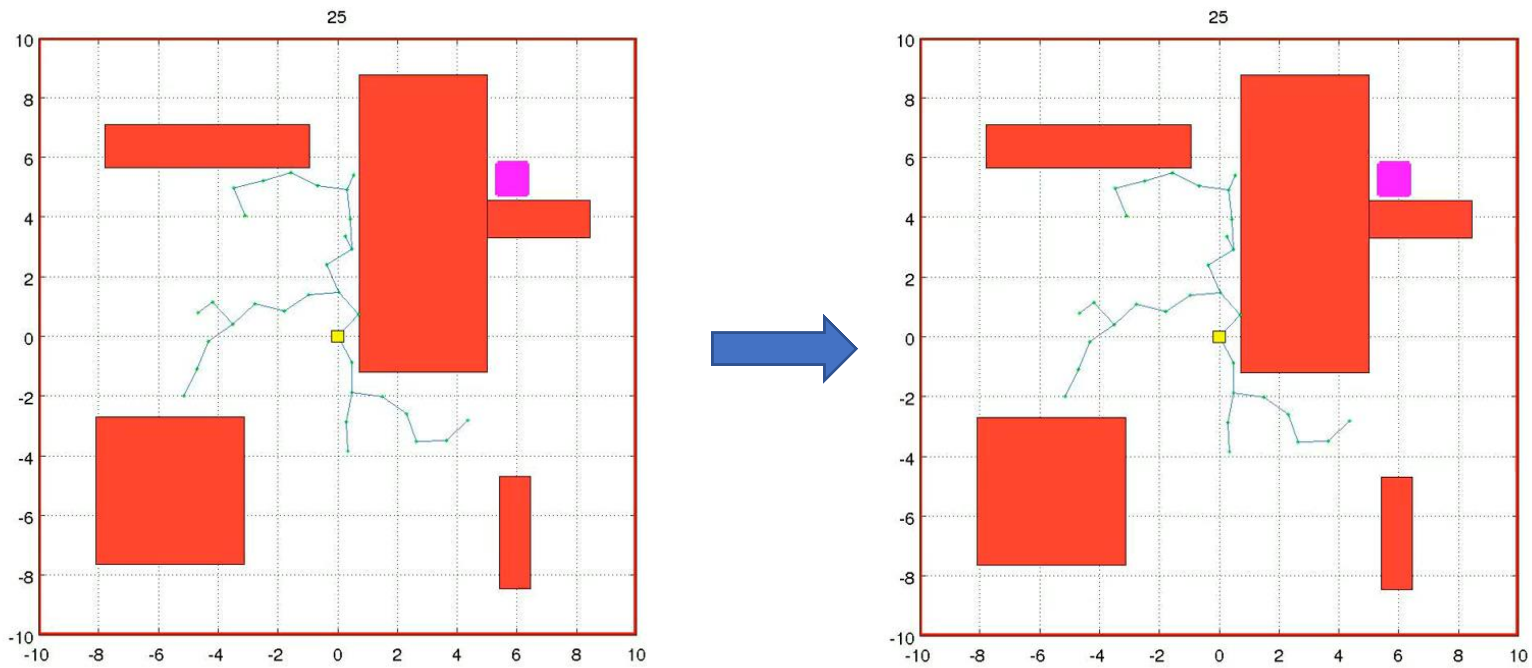
RRT

- RRT is exploring the space, explore until the final configuration is reached
- Can add bias to the goal when expanding randomly
- Pros:
  - Balance between greedy search and exploration
  - Easy to implement
- Cons
  - Metric sensivity
  - Unknown rate of convergence

Courtesy: Steven LaValle

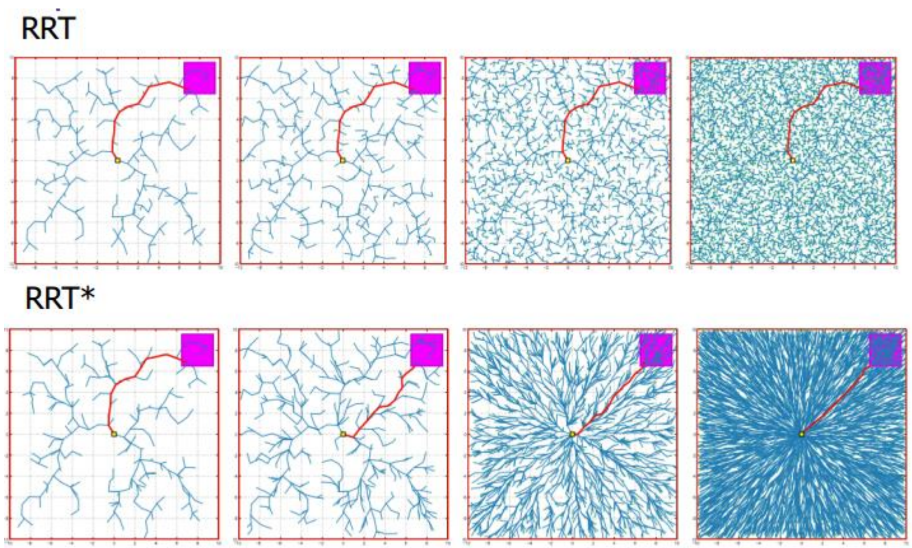
RRT*

- 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.

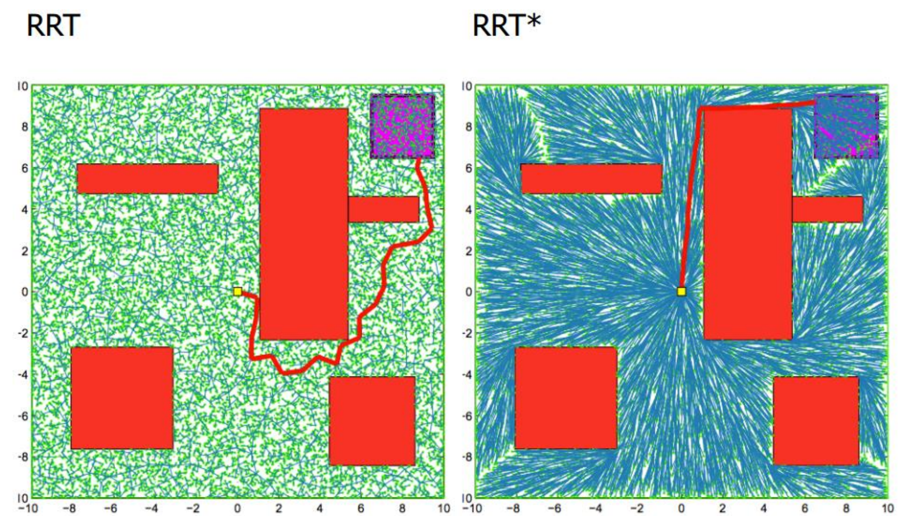


Courtesy: Shaojie Shen

RRT*



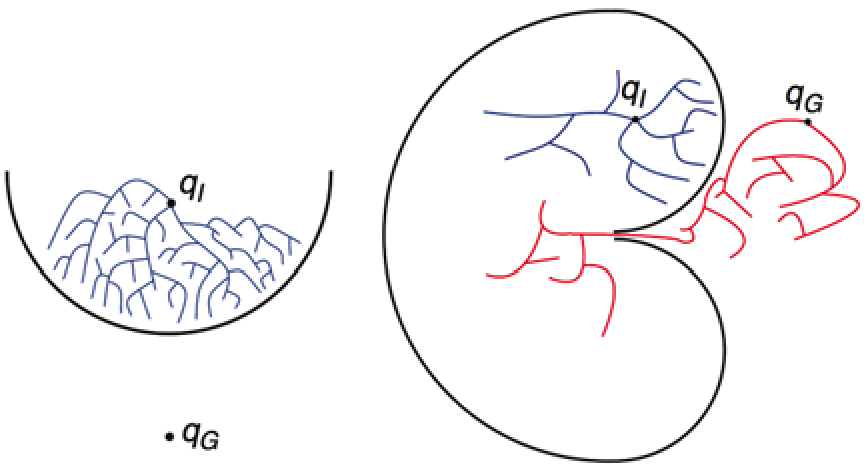
Source: Karaman and Frazzoli



Courtesy: Shaojie Shen

Bi-RRT

- Some problems require more effective methods: bidirectional search
- Grow two RRTs
- In every other step, try to extend each tree towards the newest vertex of the other tree



Filling a well　　A bug trap

Courtesy: Wolfram Burgard

RRT Page

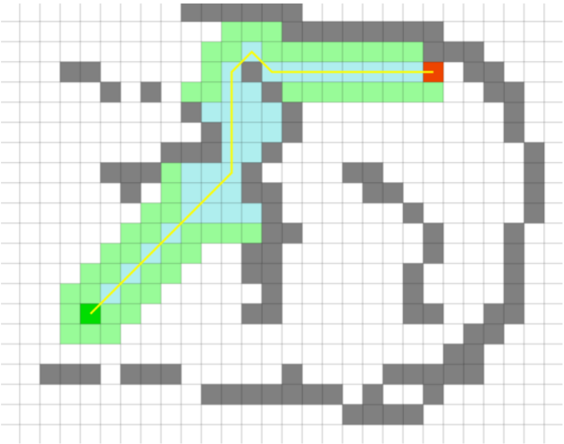
- https://lavalle.pl/rrt/

Courtesy: Steven LaValle

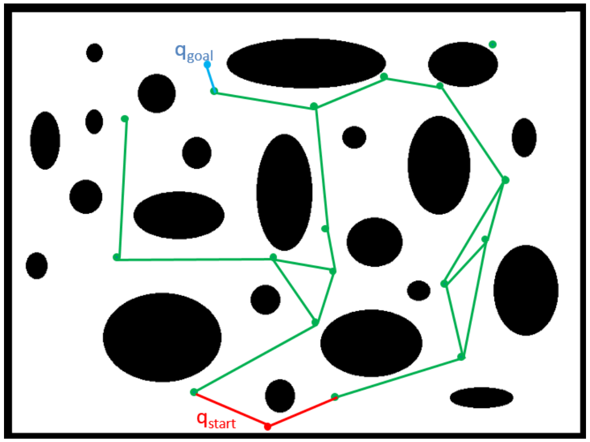
Graph Search

Search-based Method

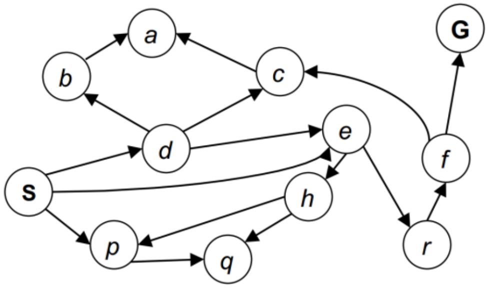
- State space graph: a mathematical representation of a search algorithm
  - For every search problem, there's a corresponding state space graph
  - Connectivity between nodes in the graph is represented by (directed or undirected) edges



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



The graph generated by probabilistic roadmap (PRM)

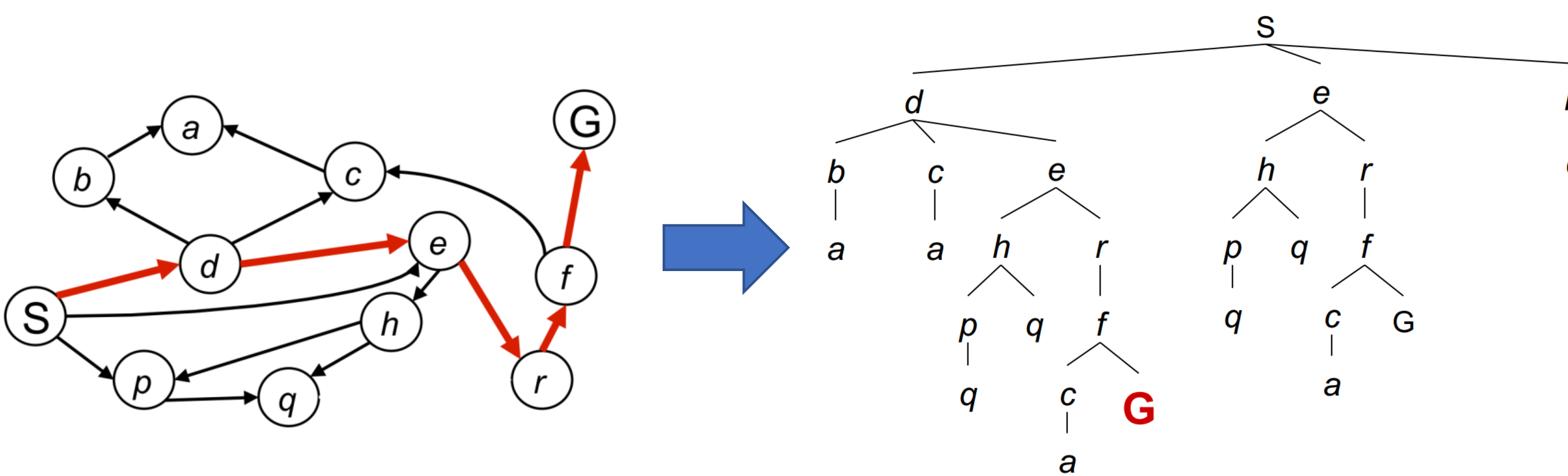


Ridiculously tiny search graph for a tiny search problem

Courtesy: Shaojie Shen

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.



Courtesy: Shaojie Shen

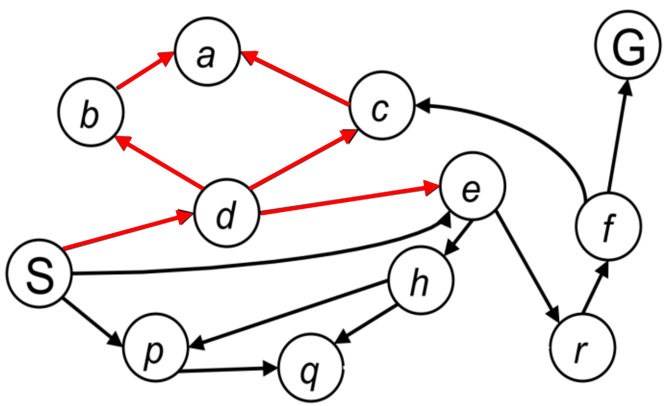
How to Construct a Search Tree?

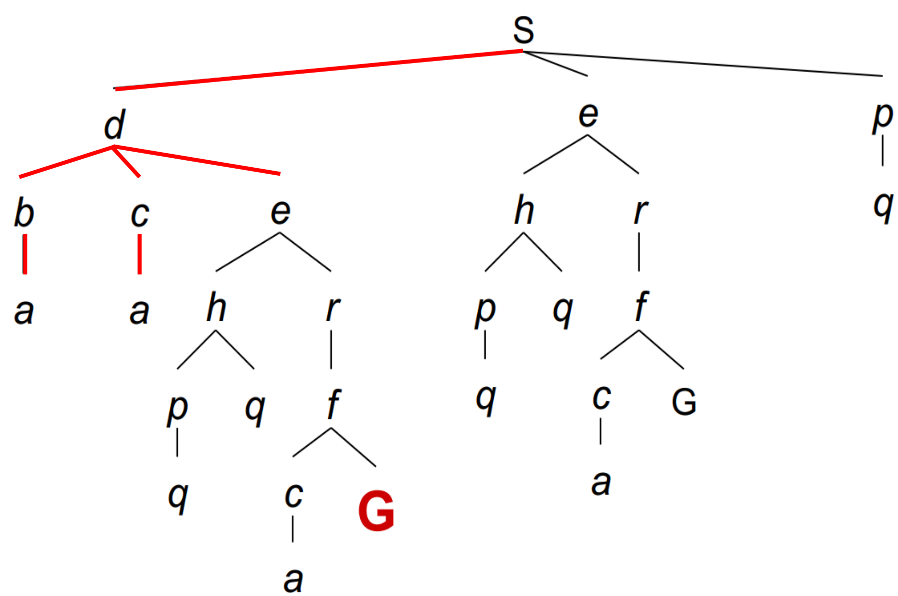
- Maintain a container to store all the nodes to be visited
  - Intuition: When we "discover" a node, we store it in our "memory". We can only visit one node at a time, but we can teleport to any node that we discover before.
- The container is initialized with the start state X_S
- Loop
  - Remove a 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

Courtesy: Shaojie Shen

Depth First Search (DFS)

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

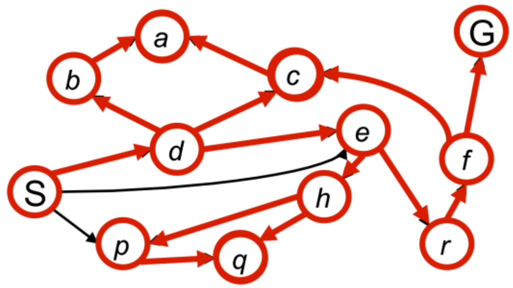


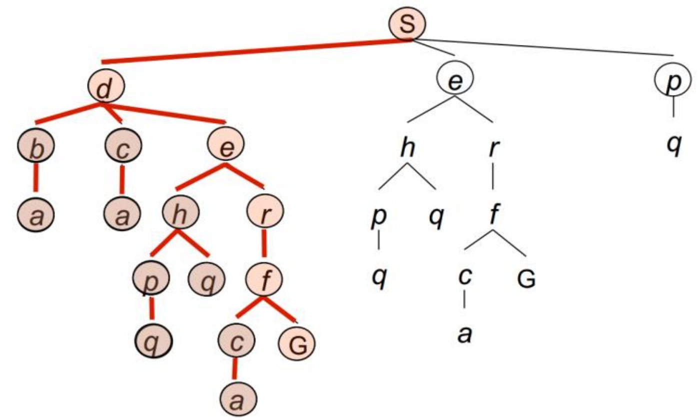


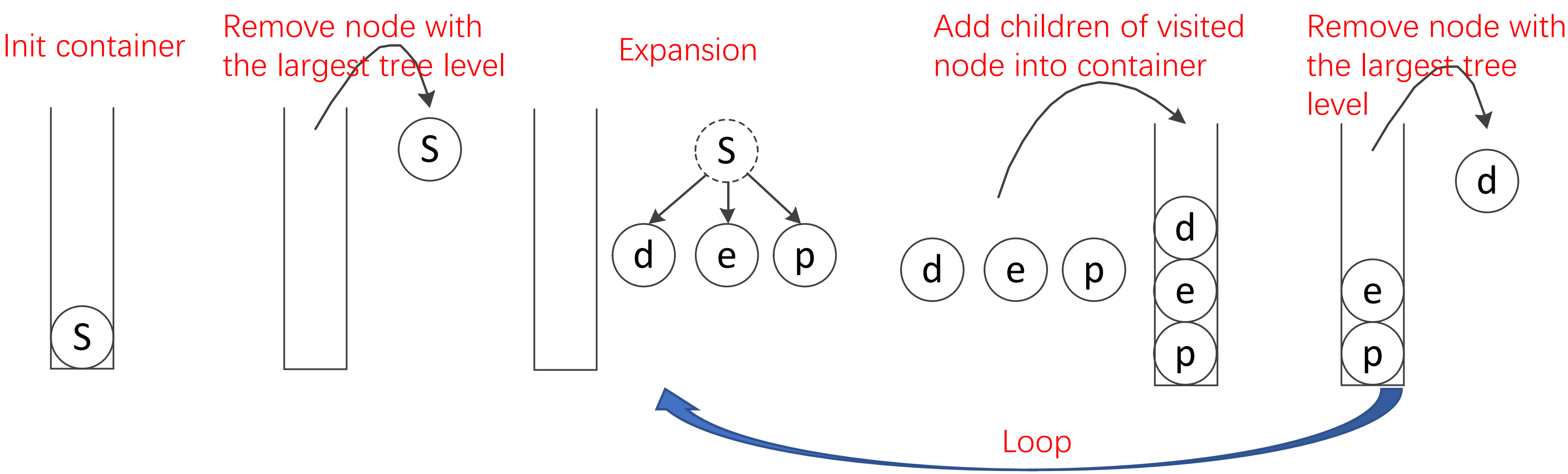
Courtesy: Shaojie Shen

Depth First Search (DFS)

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



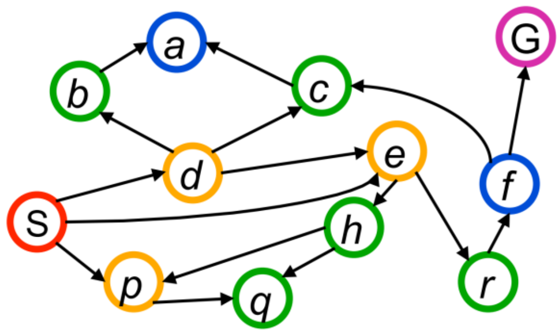


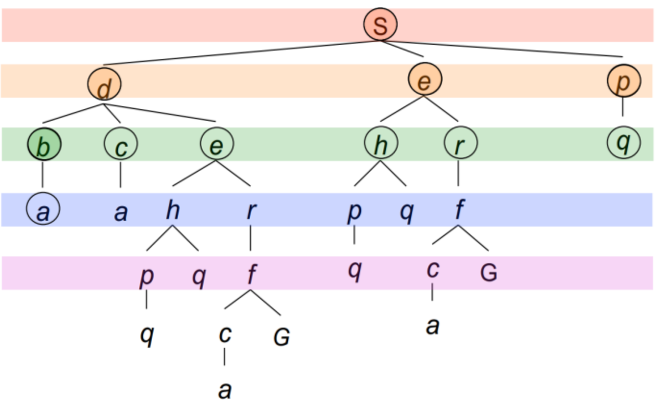


Courtesy: Shaojie Shen

Breadth First Search (BFS)

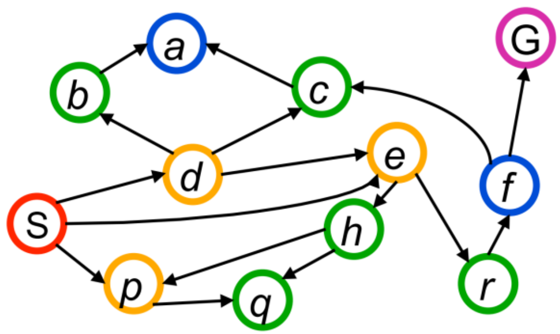
- Strategy: remove (visit) the shallowest node in the container

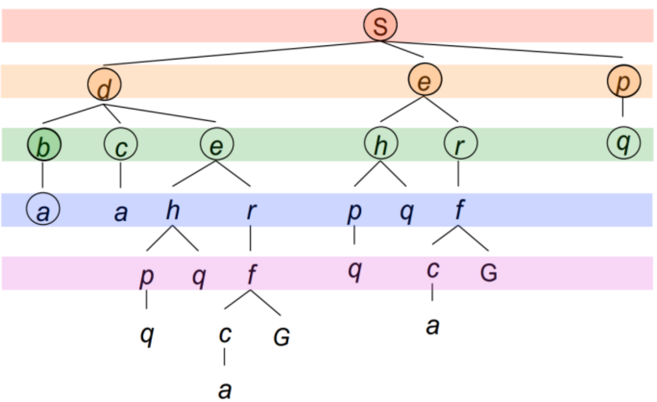




Breadth First Search (BFS)

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





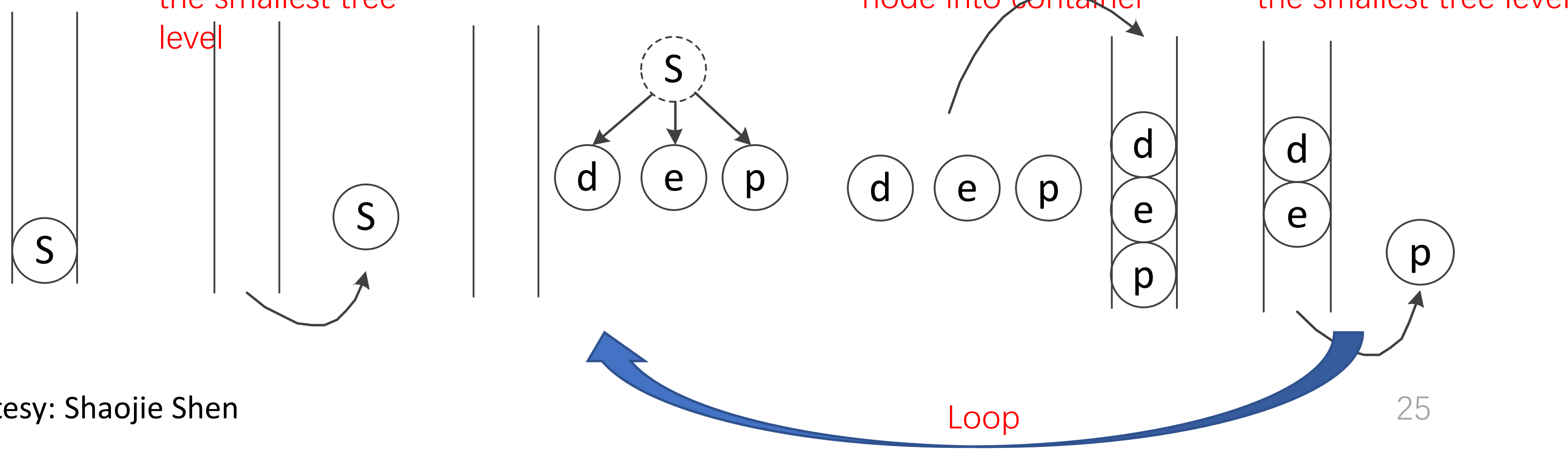
Init container

Remove node with the smallest tree level

Expansion

Add children of visited node into container

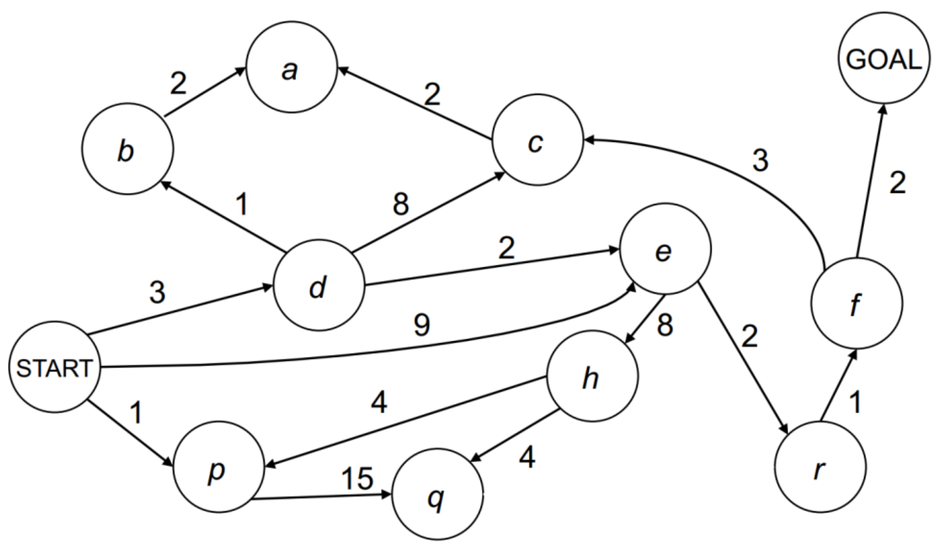
Remove node with the smallest tree level



Courtesy: Shaojie Shen

Costs on Actions

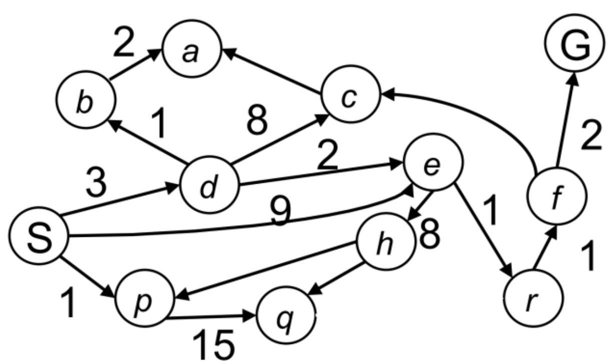
- A practical search problem has a cost "C" from a node to its neighbor
  - Length, time, energy, etc.
- When all weight are 1, BFS finds the least-cost path with minimal steps
- For general cases, how to find the least-cost path as soon as possible?

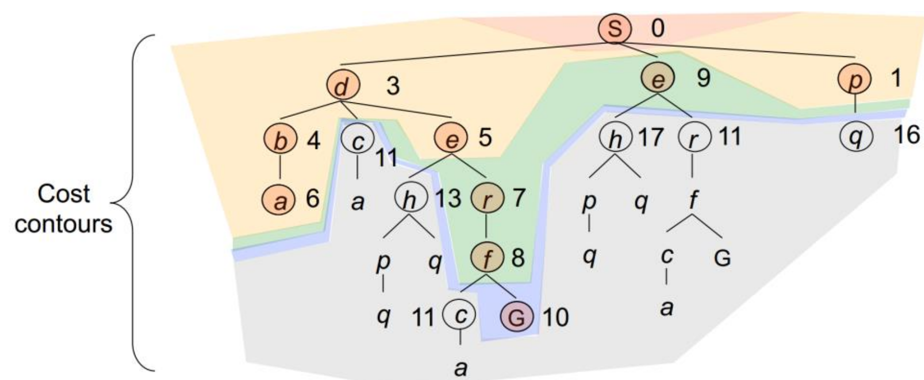


Courtesy: Shaojie Shen

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





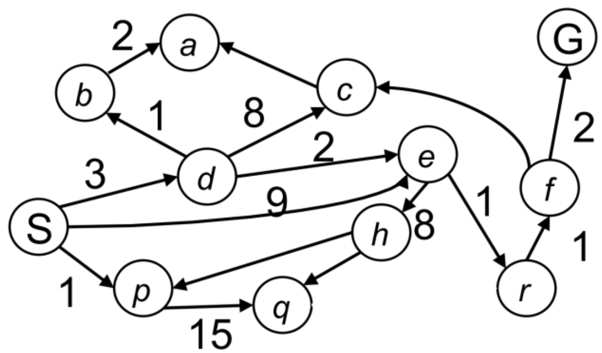
Courtesy: Shaojie Shen

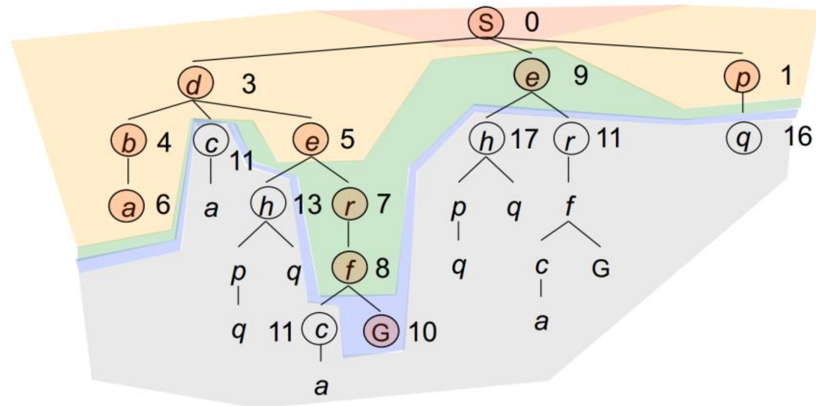
Dijkstra's Algorithm

- 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

Courtesy: Shaojie Shen

Dijkstra's Algorithm





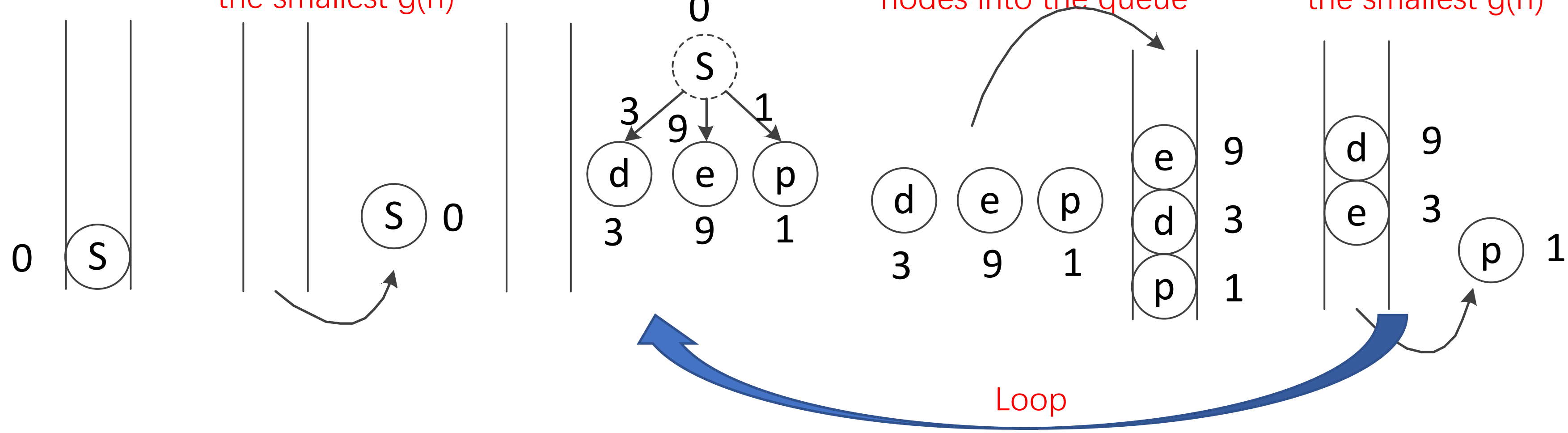
Init. queue

Remove node with the smallest g(n)

Expansion

Push neighbors of visited nodes into the queue

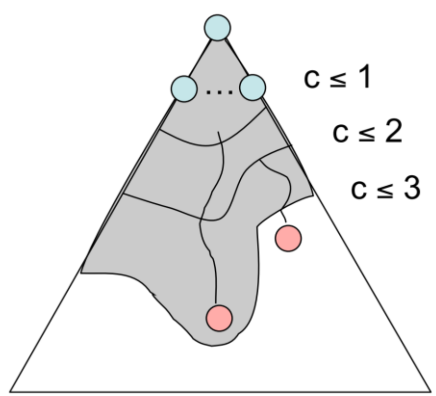
Remove node with the smallest g(n)

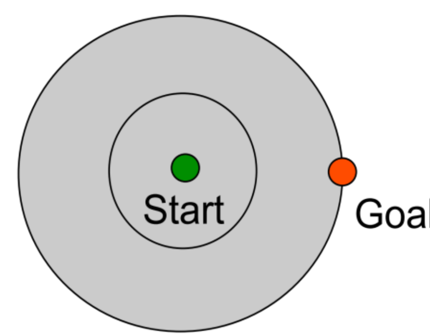


Courtesy: Shaojie Shen

Dijkstra’s Algorithm

- Pros
  - Complete and optimal
- Coons
  - 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

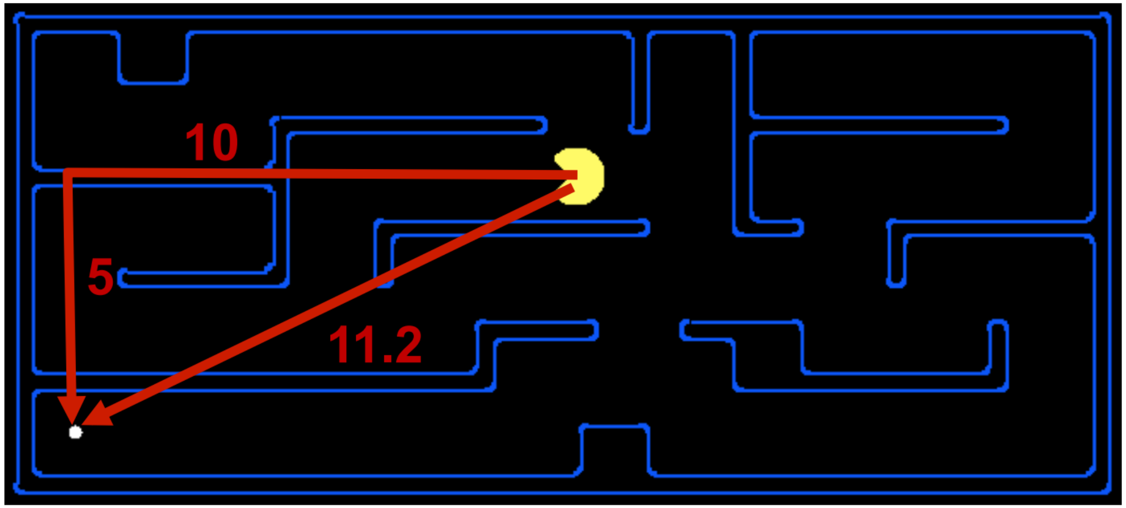




Courtesy: Shaojie Shen

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



Courtesy: Shaojie Shen

A*: 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

Courtesy: Shaojie Shen

A* Algorithm

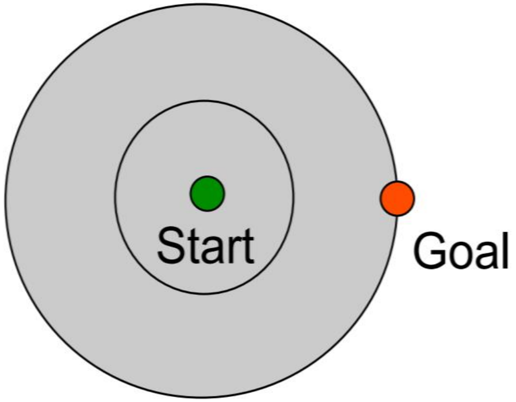
- Maintain a priority queue to store all the nodes to be visited
- The heuristic function h(n) for all nodes are pre-defined
- The priority queue is initialized with the start state X_S
- Assign $g(X_S)=0$, and $g(n)$=infinite for all other nodes in the graph
- Loop
  - If the queue is empty, return 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

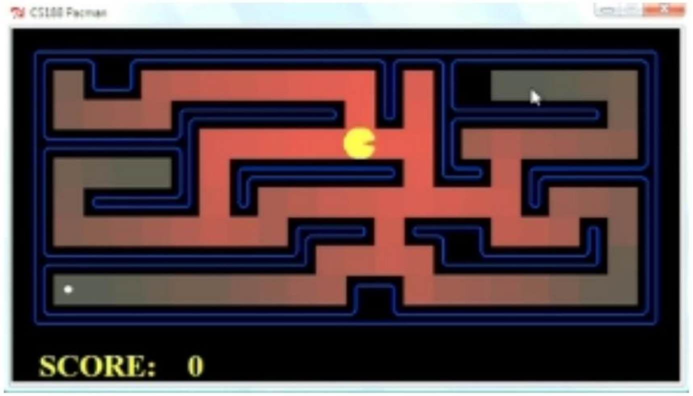
Only difference comparing to Dijkstra's algorithm

Courtesy: Shaojie Shen

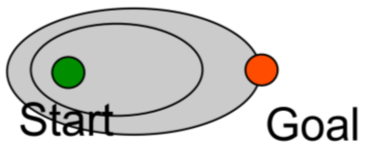
Dijkstra's VS A*

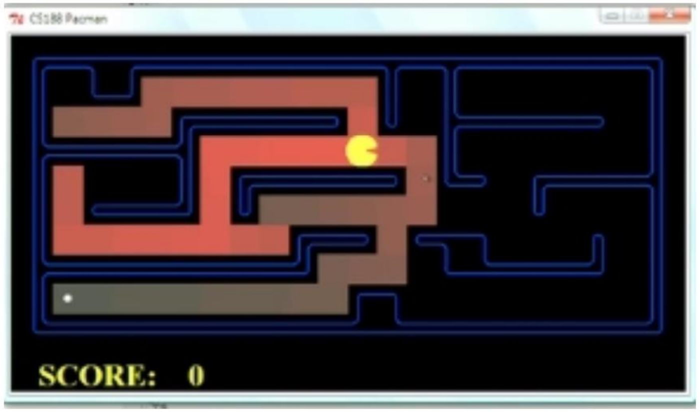
- Dijkstra's algorithm visits in all directions





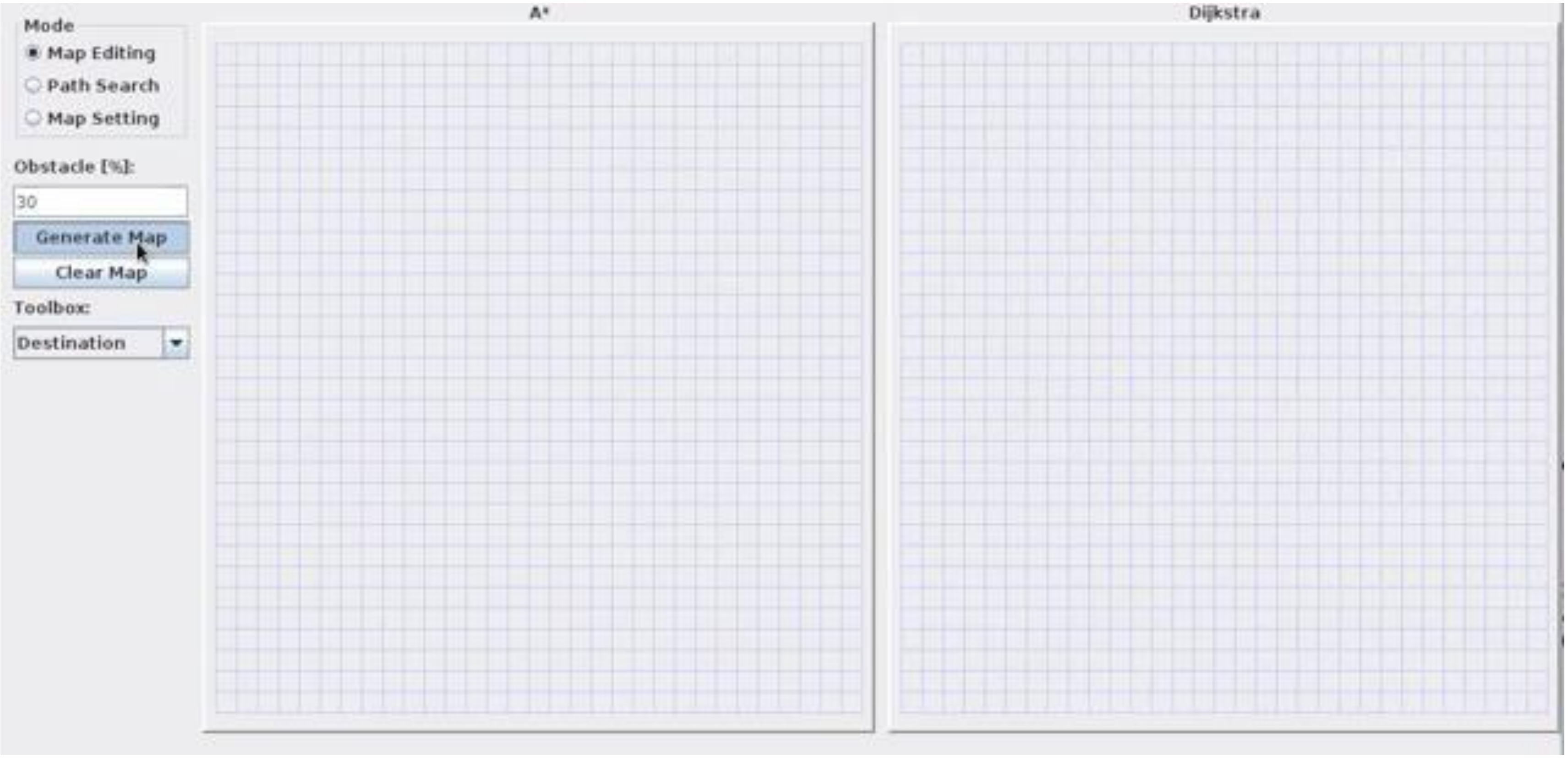
- A* visits mainly towards the goal, but does not hedge its bets to ensure optimality





Courtesy: Shaojie Shen

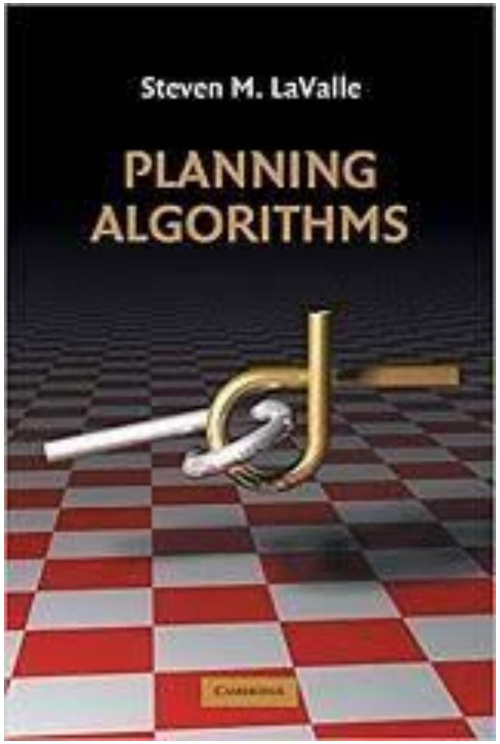
Dijkstra's VS A*



Courtesy: YouTube

Reading Resources

- If you are really interested in planning
- LaValle, Steven M. Planning algorithms. Cambridge university press, 2006.



Next Lecture

- Trajectory planning
  - Guest Lecture by Haokun Wang
  - Aerial Robot (the famous one in HKUST Robotics Institute) ☺