Path Planning Algorithms

EEL 4930/5934: Autonomous Robots

Spring 2023

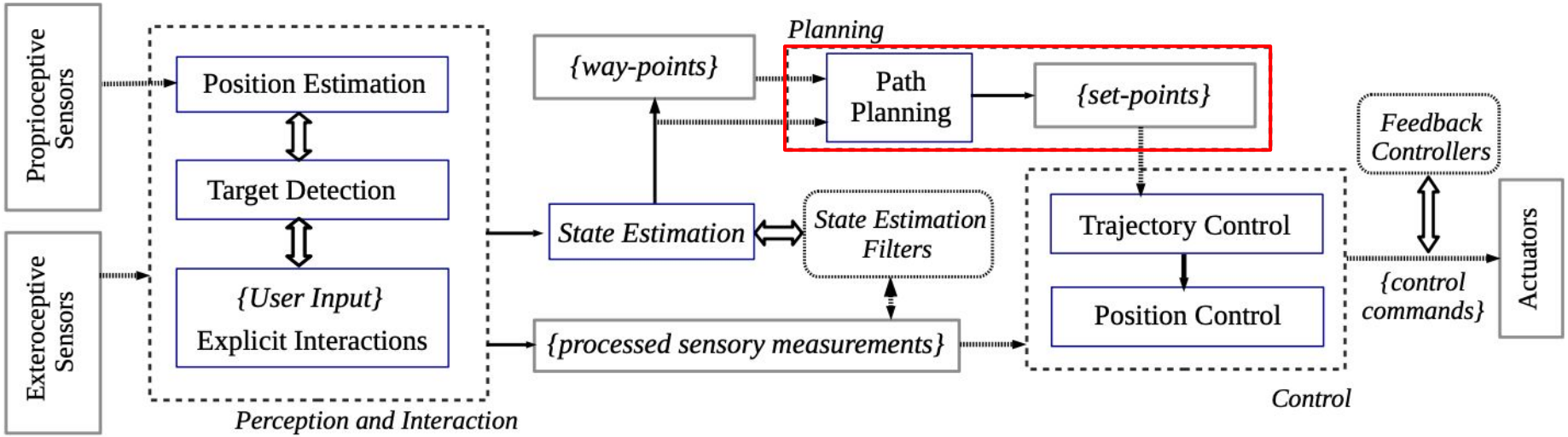
Md Jahidul Islam

Lecture 10

ECE | Florida
Electrical & Computer Engineering

UF | UNIVERSITY of FLORIDA

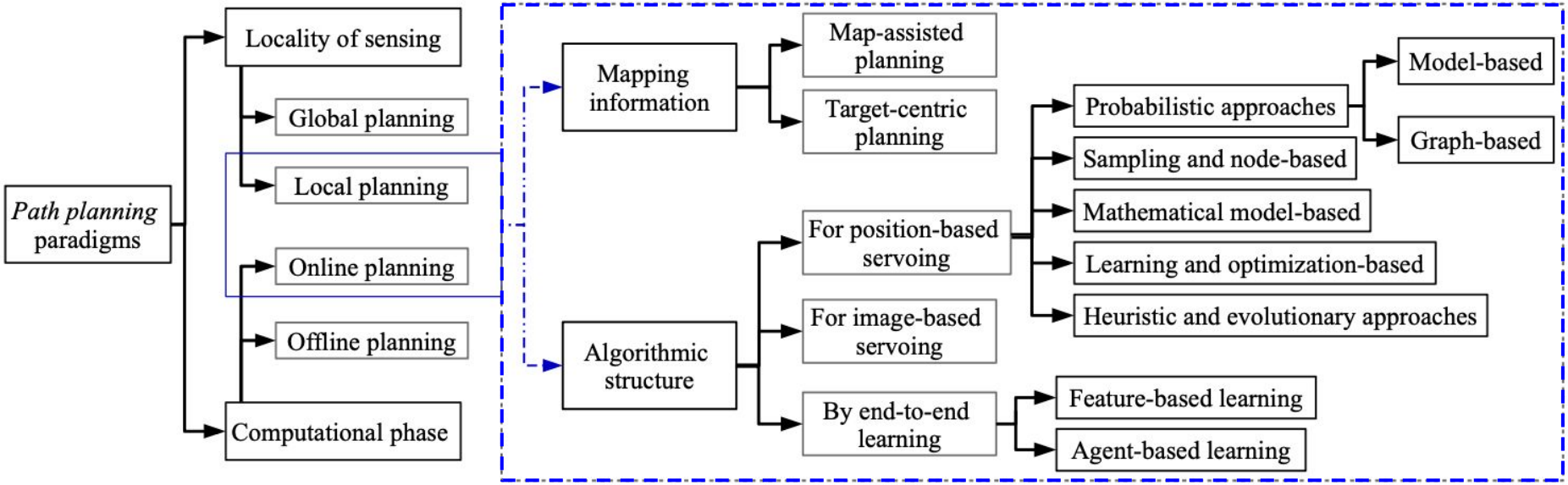
Path Planning



Pipeline Example: autonomous target following by mobile robots

Path Planners

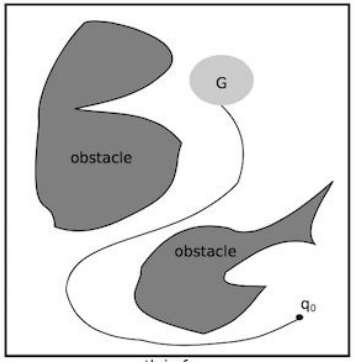
Good for autonomous target following

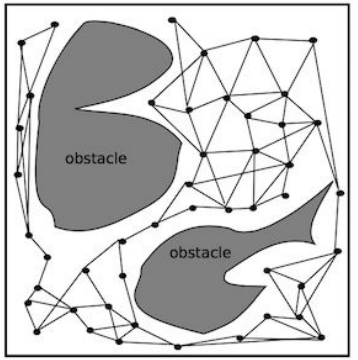


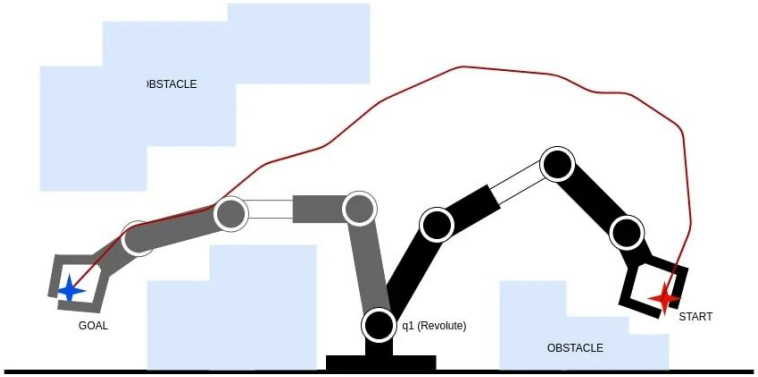
Map-based Planning

When a map is available:

- Map representation
  - Data structure to ‘describe’ the environment
  - Can be fully / partially observable
  - Discretize or simplify for computation
- Path planning then becomes a ‘search problem’
  - Finding source-destination paths
  - Finding optimal (eg, shortest) path
  - Ensuring safe and effective robot navigation
    - Avoid obstacles, dynamic agents
    - Plan for human-robot interaction





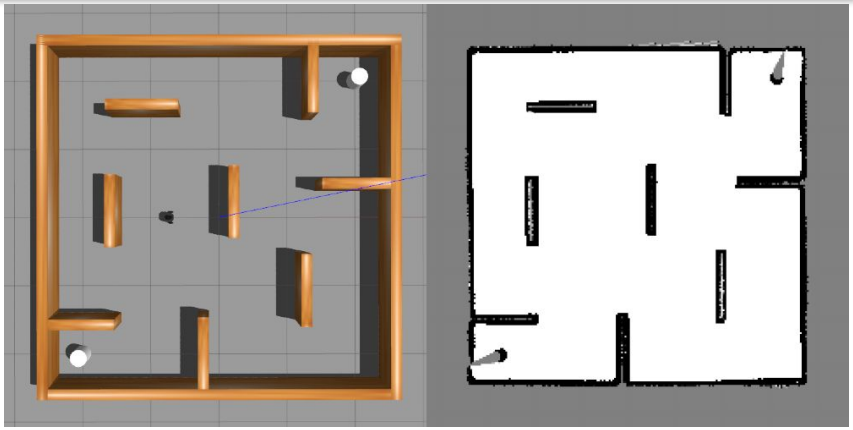


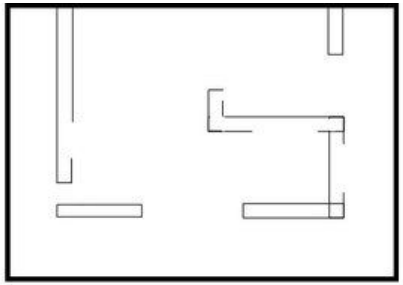
Ref: Probabilistic Robotics (Chapter 9)

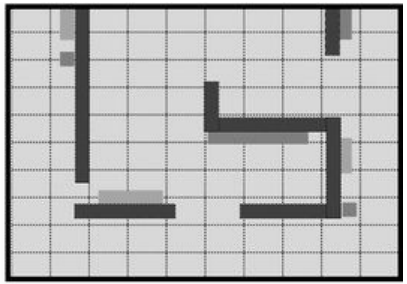
Roadmaps / Grids / Graphs based Methods

Most simplified form of planners:

- Discretizes a map (LIDAR scans, point clouds)
  - Into finite spaces (cells, grids, nodes)
  - Assigns numbers or reference points
  - Represents the ‘environment’ as a graph, tree, occupancy grid, roadmaps, etc.
- Utilizes the graph theory literature for planning
  - Source-destination search
  - Optimal path search
- Good for simple navigation with static obstacles
- ROS has lots of good libraries to explore!
- **We will cover:** BFS, DFS, Dijkstra, A*, Bug0/Bug1, RRT, RRT*, and PRMs







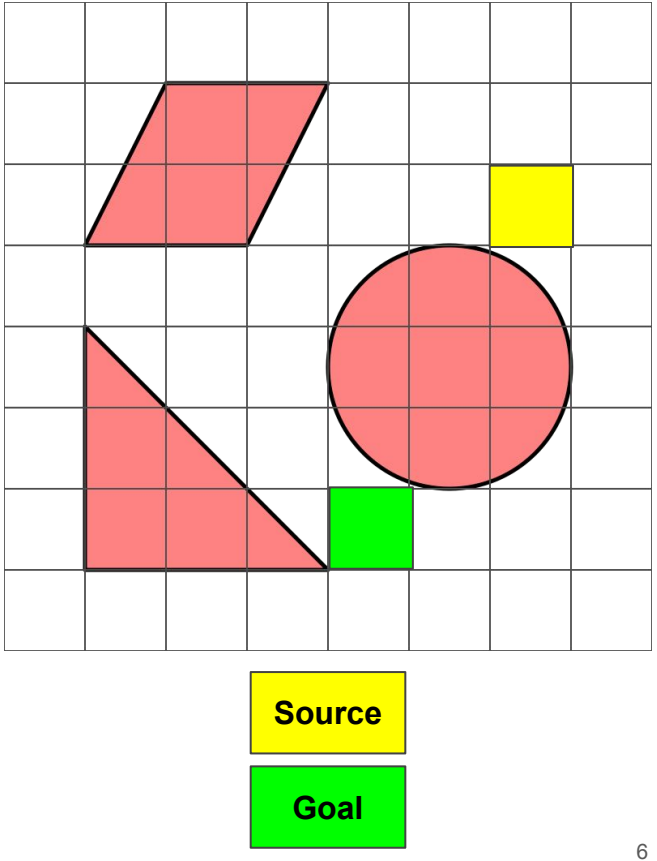
BFS and DFS Algorithms

BFS: Breadth First Search

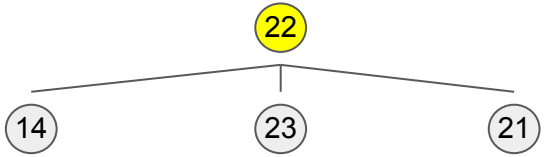
- Search a tree, one level at a time
- Uses FIFO (first in first out) queue for implementation
- Algorithmic characteristics:
  - Complete (finds solution if there is one)
  - Optimal if cost is increasing with path depth

DFS: Depth First Search

- Search a tree, keep expanding one child at a time
- Uses LIFO (last in first out) stack for implementation
- Algorithmic characteristics
  - NOT Complete if infinite depth
  - NOT Optimal



BFS Algorithm

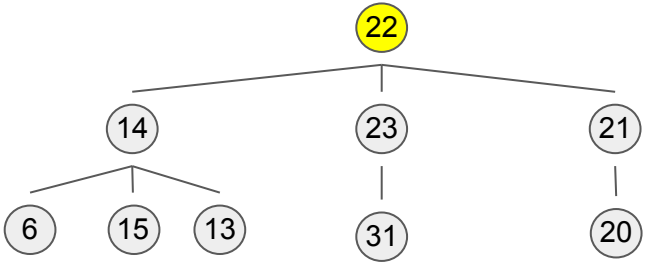


Q = [22]
Q = [14, 23, 21]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Order: up, right, left, down

BFS Algorithm

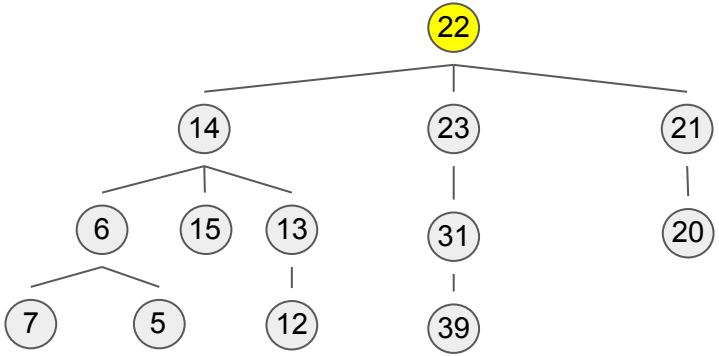


Q = [22]
Q = [14, 23, 21]
Q = [23, 21, 6, 15, 13]
Q = [21, 6, 15, 13, 31]
Q = [6, 15, 13, 31, 20]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Order: up, right, left, down

BFS Algorithm

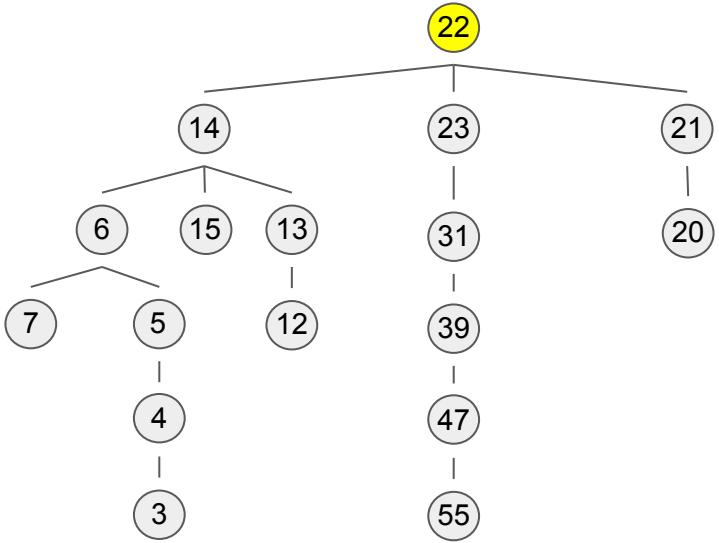


Q = [22]
Q = [14, 23, 21]
Q = [23, 21, 6, 15, 13]
Q = [21, 6, 15, 13, 31]
Q = [6, 15, 13, 31, 20]
Q = [15, 13, 31, 20, 7, 5]
Q = [13, 31, 20, 7, 5]
Q = [31, 20, 7, 5, 12]
Q = [20, 7, 5, 12, 39]
Q = [7, 5, 12, 39]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Order: up, right, left, down

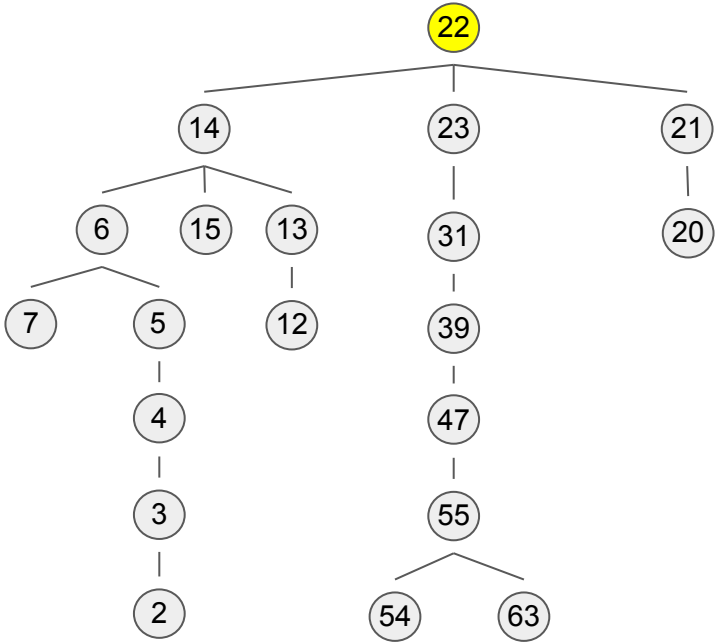
BFS Algorithm



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Order: up, right, left, down

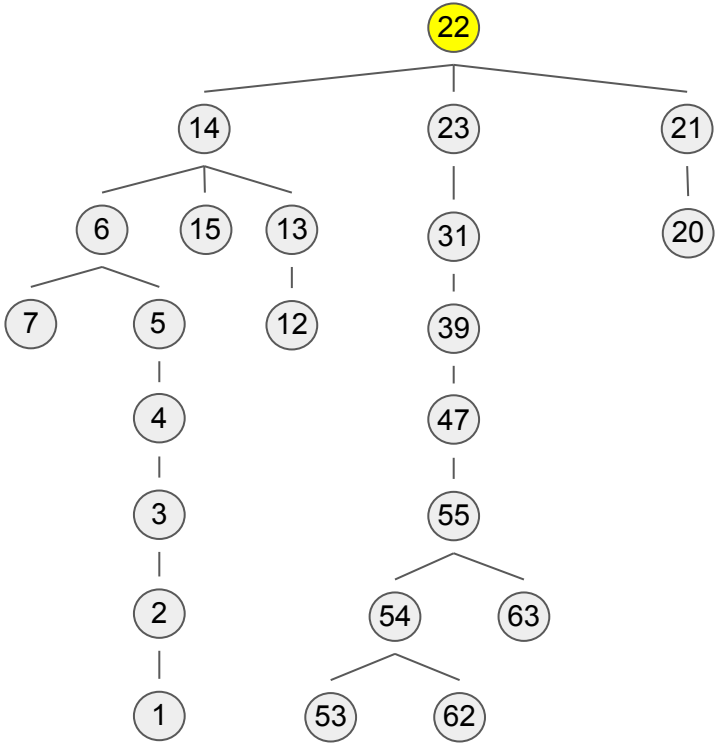
BFS Algorithm



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Order: up, right, left, down

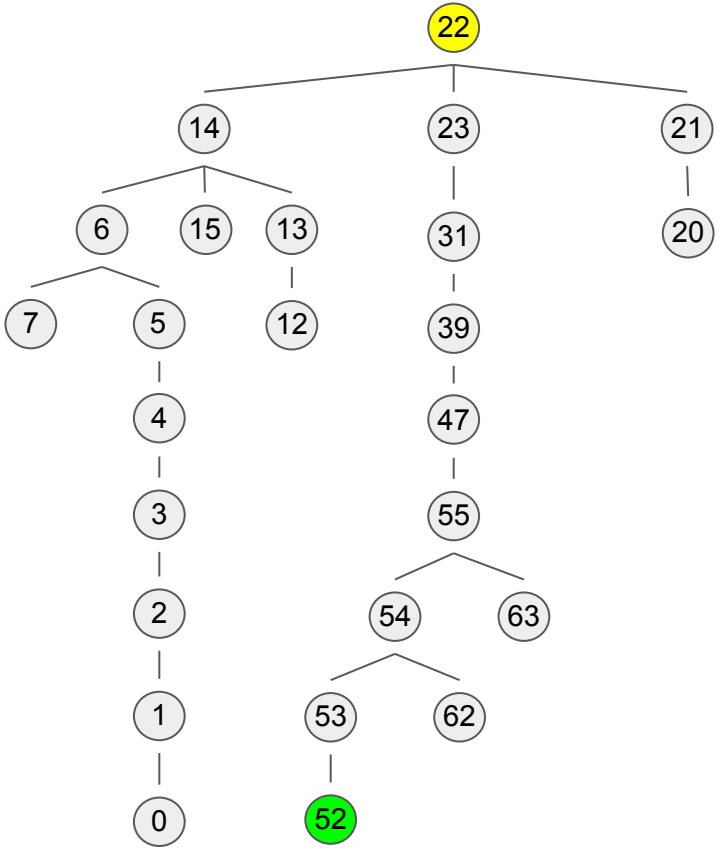
BFS Algorithm



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Order: up, right, left, down

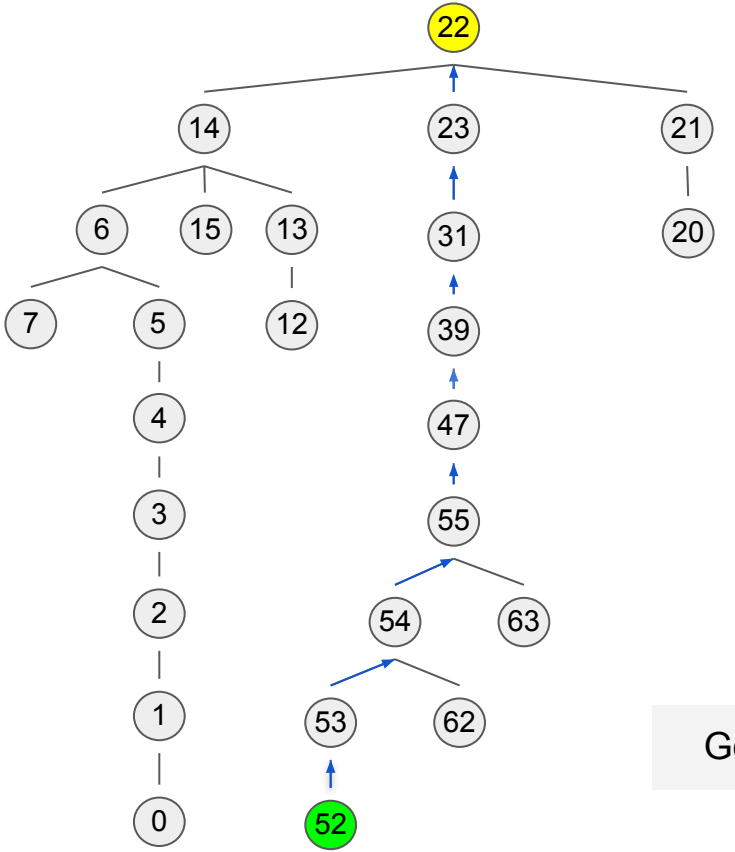
BFS Algorithm



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Order: up, right, left, down

BFS Algorithm

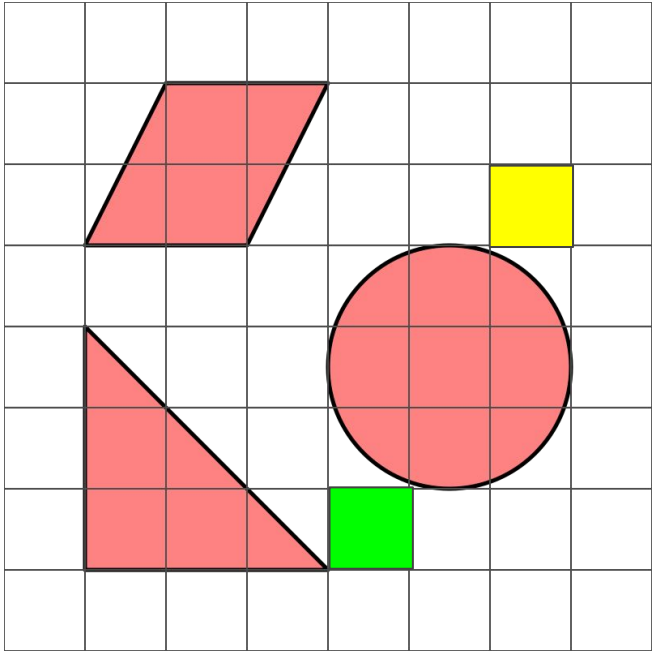


0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

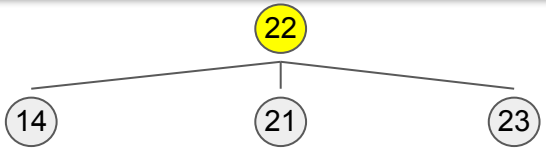
Going backward from the end node can lead us to the optimal path

BFS vs DFS

BFS: Breadth First Search	**DFS**: Depth First Search
BFS builds the tree level by level	DFS builds the tree subtree by subtree
Uses queue: FIFO (First In First Out)	Uses stack: LIFO (Last In First Out)
Suitable for searching vertices closer to the given source	Suitable when there are solutions away from source
There is no concept of backtracking	Uses the idea of backtracking subtree
Complete and optimal	Not complete and not optimal
Needs more memory and run-time	Needs less memory and relatively faster
Runtime: O(V E) when using adjacency list and $O(V^2)$ with Adjacency matrix	Runtime: O(V+E) when using adjacency list and $O(V^2)$ with Adjacency matrix



DFS Algorithm

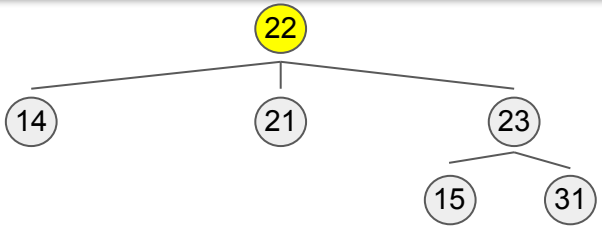


S = [22]
S = [14, 21, 23]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Order: up, left, down, right

DFS Algorithm

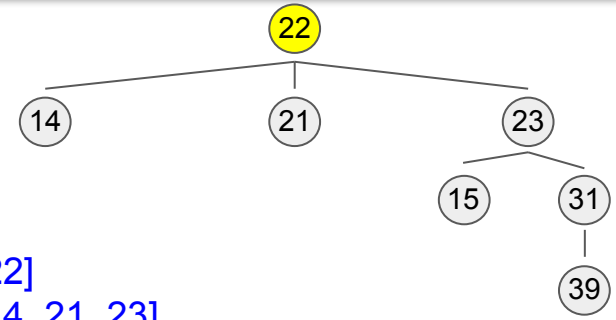


S = [22]
S = [14, 21, 23]
S = [14, 21, 15, 31]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Order: up, left, down, right

DFS Algorithm

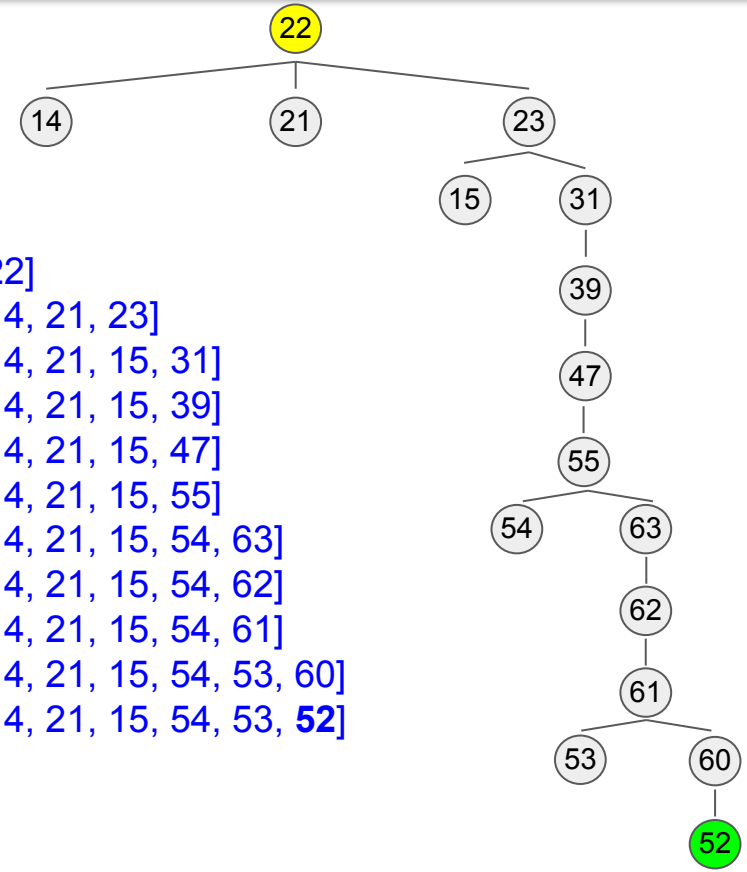


S = [22]
S = [14, 21, 23]
S = [14, 21, 15, 31]
S = [14, 21, 15, 39]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Order: up, left, down, right

DFS Algorithm

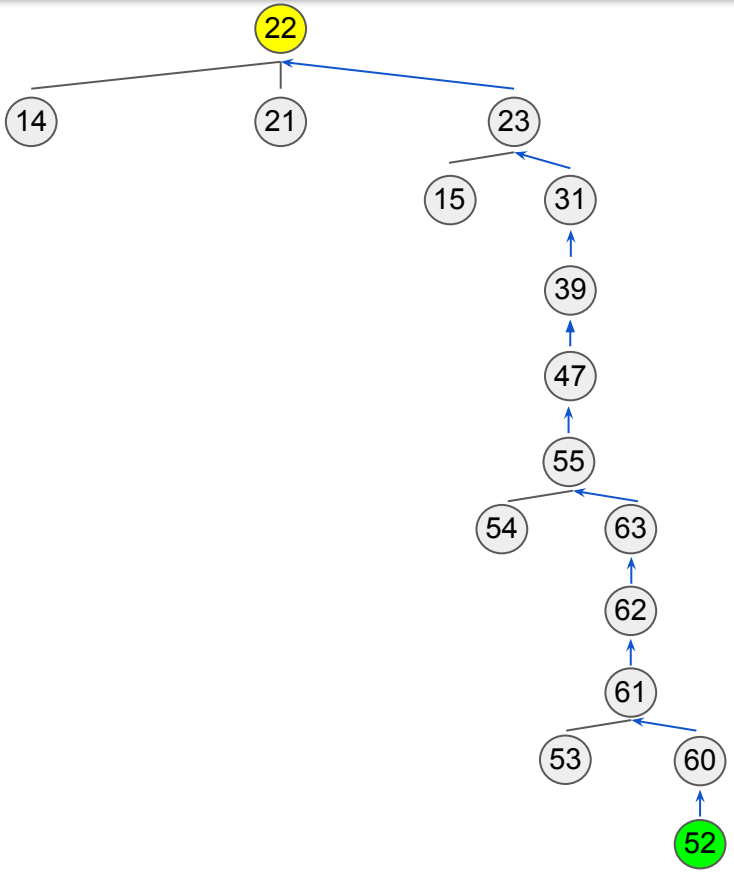


S = [22]
S = [14, 21, 23]
S = [14, 21, 15, 31]
S = [14, 21, 15, 39]
S = [14, 21, 15, 47]
S = [14, 21, 15, 55]
S = [14, 21, 15, 54, 63]
S = [14, 21, 15, 54, 62]
S = [14, 21, 15, 54, 61]
S = [14, 21, 15, 54, 53, 60]
S = [14, 21, 15, 54, 53, **52**]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Order: up, left, down, right

DFS Algorithm



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

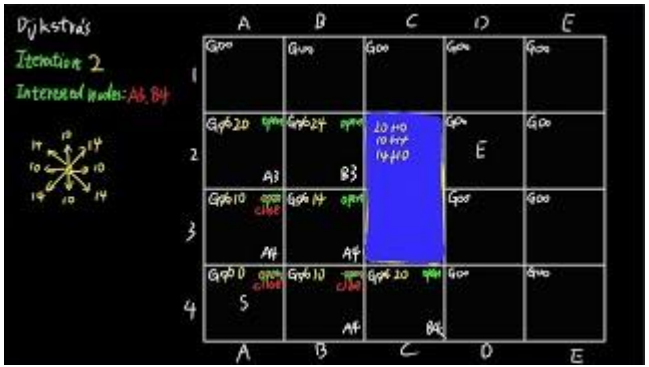
A solution, but not necessarily the optimal path

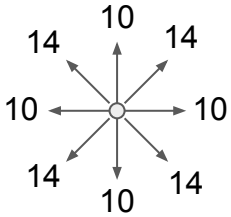
Dijkstra and A*

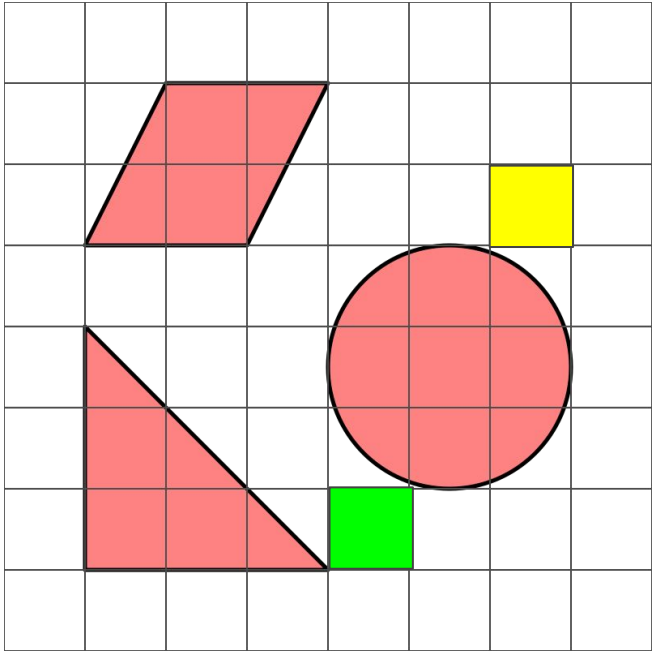
Dijkstra: searches the single-source shortest path

- Optimal and complete, but not always fast
  - Start node is assigned a distance of zero
  - Other node's distance are set to infinity
  - Compute ***g(n)***: path cost from the start node to n

A*: uses heuristics considering not only how far a state is from the start but also how close it is to the goal







Dijkstra Algorithm

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Dijkstra Algorithm

0	1	2	3	4	5	6	7
8	9	10	11	12	14 13	10 14	14 15
16	17	18	19	20	10 21	22	10 23
24	25	26	27	28	29	30	14 31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Computed *g(.)* cost for all child nodes: in top left corner of a cell

Dijkstra Algorithm

0	1	2	3	28 4	24 5	20 6	24 7
8	9	10	11	24 12	14 13	10 14	14 15
16	17	18	19	20 20	10 21	22	10 23
24	25	26	27	28	29	30	14 31
32	33	34	35	36	37	38	24 39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Expand all the child nodes and compute their *g(.)* cost

Dijkstra Algorithm

0	1	2	38 3	28 4	24 5	20 6	24 7
8	9	10	11	24 12	14 13	10 14	14 15
16	17	18	19	20 20	10 21	22	10 23
24	25	26	34 27	28	29	30	14 31
32	33	34	35	36	37	38	24 39
40	41	42	43	44	45	46	34 47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Expand all the child nodes and compute their *g(.)* cost

Dijkstra Algorithm

0	1	48 2	38 3	28 4	24 5	20 6	24 7
8	9	10	11	24 12	14 13	10 14	14 15
16	17	18	19	20 20	10 21	22	10 23
24	25	44 26	34 27	28	29	30	14 31
32	33	48 34	44 35	36	37	38	24 39
40	41	42	43	44	45	46	34 47
48	49	50	51	52	53	48 54	44 55
56	57	58	59	60	61	62	63

Expand all the child nodes and compute their *g(.)* cost

Dijkstra Algorithm

Expand all the child nodes and compute their *g(.)* cost

Dijkstra Algorithm

68 0	58 1	48 2	38 3	28 4	24 5	20 6	24 7
72 8	9	10	11	24 12	14 13	10 14	14 15
68 16	17	18	19	20 20	10 21	22	10 23
64 24	54 25	44 26	34 27	28	29	30	14 31
68 32	33	48 34	44 35	36	37	38	24 39
40	41	42	54 43	44	45	46	34 47
48	49	50	51	68 52	58 53	48 54	44 55
56	57	58	59	72 60	62 61	58 62	54 63

We reached the goal node, so stop subtree expansion

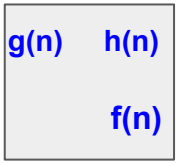
Dijkstra Algorithm

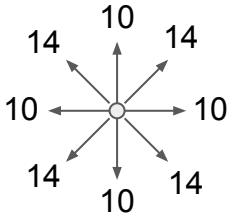
Going backward from the end node can lead us to the optimal paths

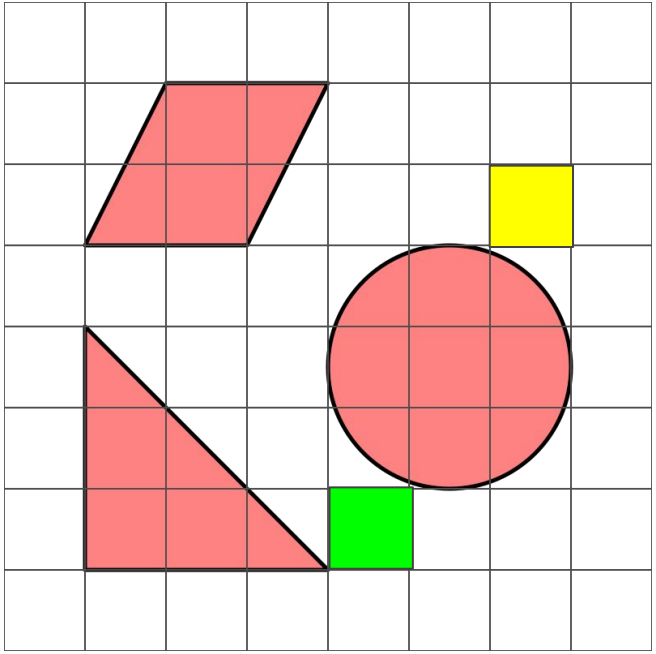
A* Search Algorithm

A* Search:

- Uses heuristics to find the “best” node to expand
- Optimal and complete
- Evaluate a node *n* for expansion based on the function: ***f(n) = g(n) + h(n)***
  - ***g(n)***: path cost from the start node to n
  - ***h(n)*****:** estimated cost of the cheapest path from node *n* to the goal node (heuristic)







A* Search Algorithm

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

A* Search Algorithm

Expand all the child nodes
- Compute ***h(.)*** cost: top-right corner
- compute ***g(.)*** cost: top-left corner
- ***f(.) = g(.) + h(.)***: bottom-right corner

Select the cell with the minimum ***f(.)*** cost

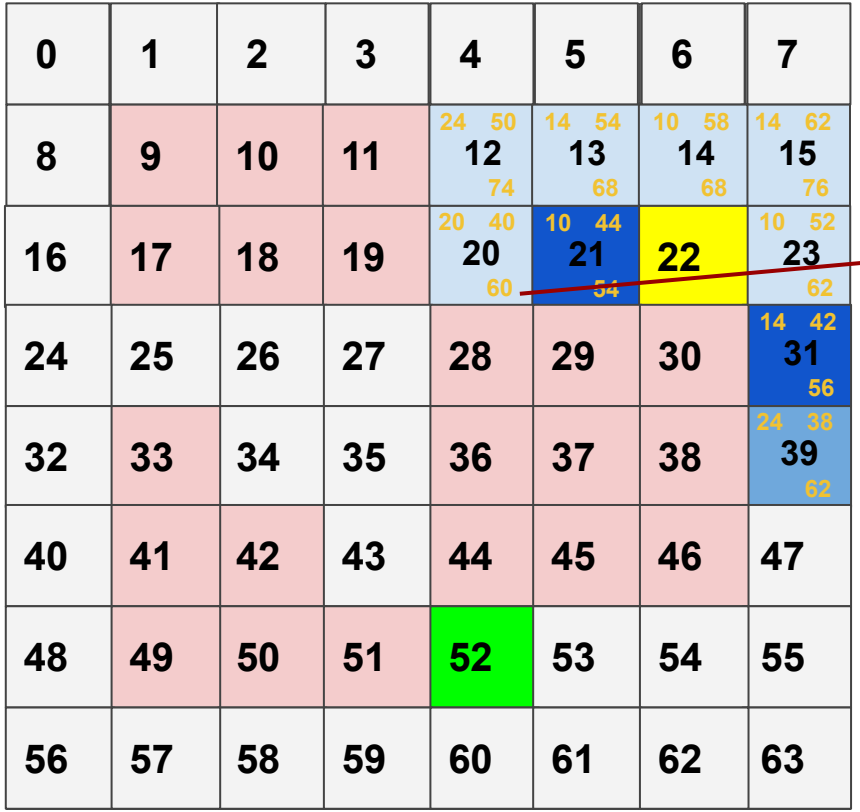
A* Search Algorithm

Expand all the child nodes
- Compute ***h(.)*** cost: top-right corner
- compute ***g(.)*** cost: top-left corner
- ***f(.) = g(.) + h(.)***: bottom-right corner

Select the cell with the minimum ***f(.)***

A* Search Algorithm

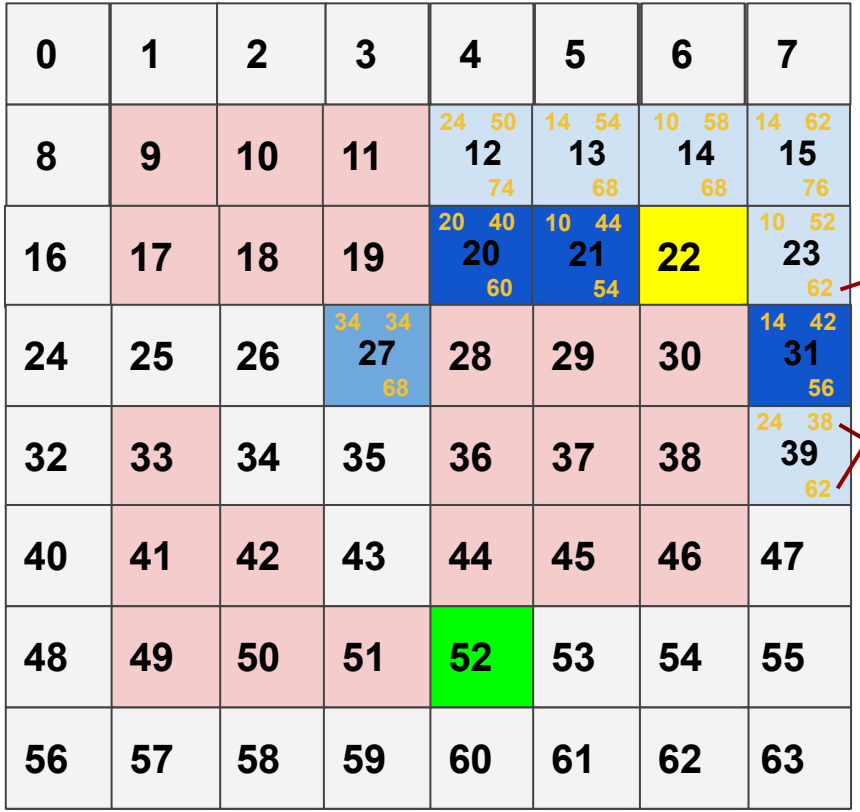
Expand all the child nodes
- Compute ***h(.)*** cost: top-right corner
- compute ***g(.)*** cost: top-left corner
- ***f(.) = g(.) + h(.)***: bottom-right corner



Select the cell with the minimum ***f(.)***

A* Search Algorithm

Expand all the child nodes
- Compute ***h(.)*** cost: top-right corner
- compute ***g(.)*** cost: top-left corner
- ***f(.) = g(.) + h(.)***: bottom-right corner



If there are two cells with the minimum *f(.)*

Select the one with the minimum ***h(.)*** cost

A* Search Algorithm

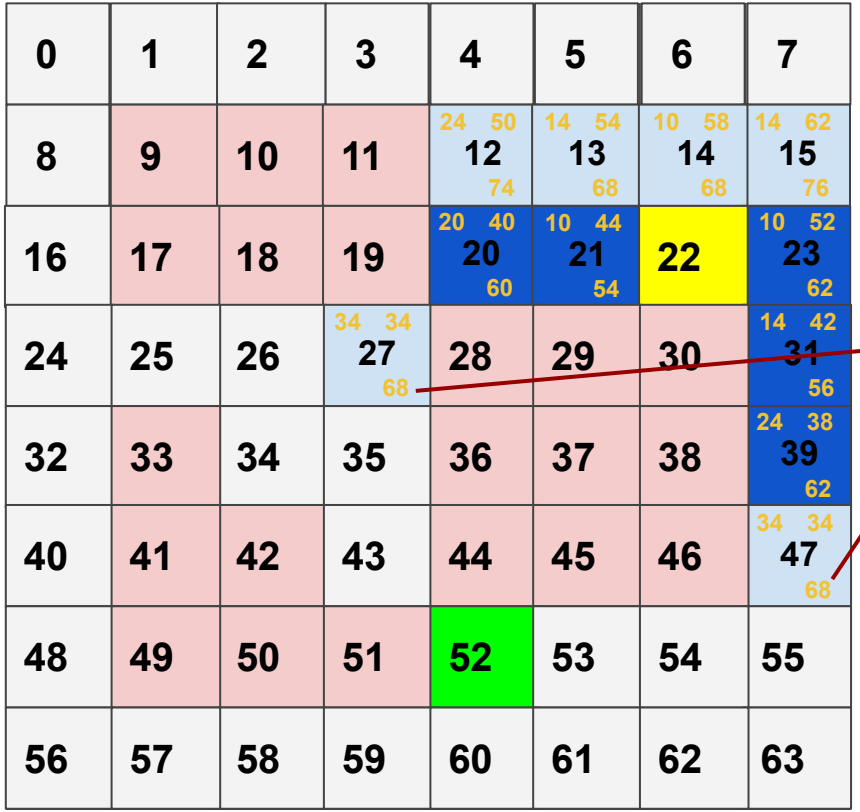
Expand all the child nodes
- Compute ***h(.)*** cost: top-right corner
- compute ***g(.)*** cost: top-left corner
- ***f(.) = g(.) + h(.)***: bottom-right corner

0	1	2	3	4	5	6	7
8	9	10	11	24 50 12 74	14 54 13 68	10 58 14 68	14 62 15 76
16	17	18	19	20 40 20 60	10 44 21 54	22	10 52 23 62
24	25	26	34 34 27 68	28	29	30	14 42 31 56
32	33	34	35	36	37	38	24 38 39 62
40	41	42	43	44	45	46	34 34 47 68
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Select the cell with the minimum ***f(.)***

A* Search Algorithm

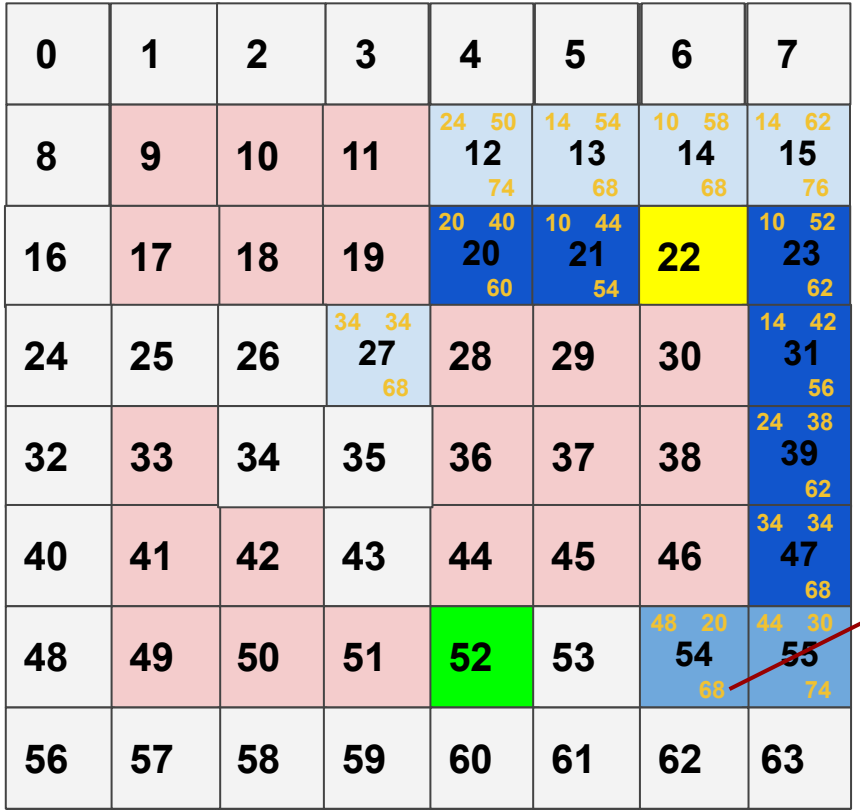
Skip if the node with minimum cost has no unvisited child nodes



If there are two cells with the minimum ***f(.)*** and with the same ***h(.)*** cost, then randomly select one

A* Search Algorithm

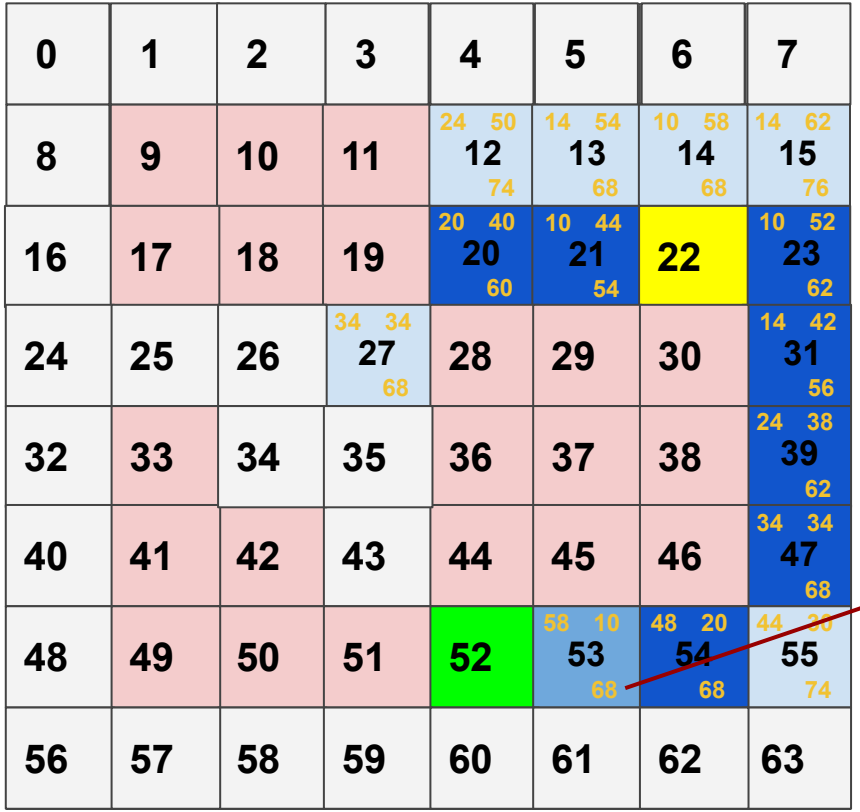
Expand all the child nodes
- Compute ***h(.)*** cost: top-right corner
- compute ***g(.)*** cost: top-left corner
- ***f(.) = g(.) + h(.)***: bottom-right corner



Select the cell with the minimum ***f(.)*** and minimum ***h(.)*** cost

A* Search Algorithm

Expand all the child nodes
- Compute ***h(.)*** cost: top-right corner
- compute ***g(.)*** cost: top-left corner
- ***f(.) = g(.) + h(.)***: bottom-right corner



Select the cell with the minimum ***f(.)*** and minimum ***h(.)*** cost

A* Search Algorithm

We reached the goal node, so stop subtree expansion

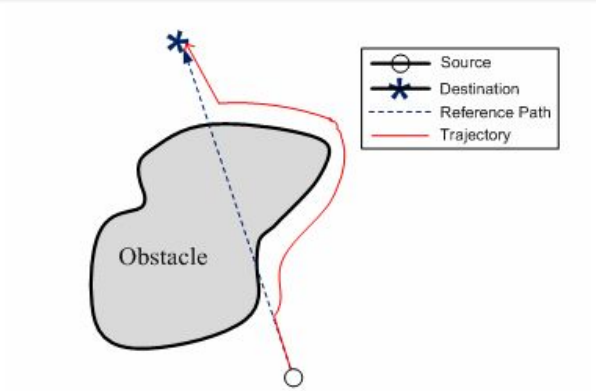
0	1	2	3	4	5	6	7
8	9	10	11	24 50 12 74	14 54 13 68	10 58 14 68	14 62 15 76
16	17	18	19	20 40 20 60	10 44 21 54	22	10 52 23 62
24	25	26	34 34 27 68	28	29	30	14 42 31 56
32	33	34	35	36	37	38	24 38 39 62
40	41	42	43	44	45	46	34 34 47 68
48	49	50	51	52	58 10 53 68	48 20 54 68	44 30 55 74
56	57	58	59	60	61	62	63

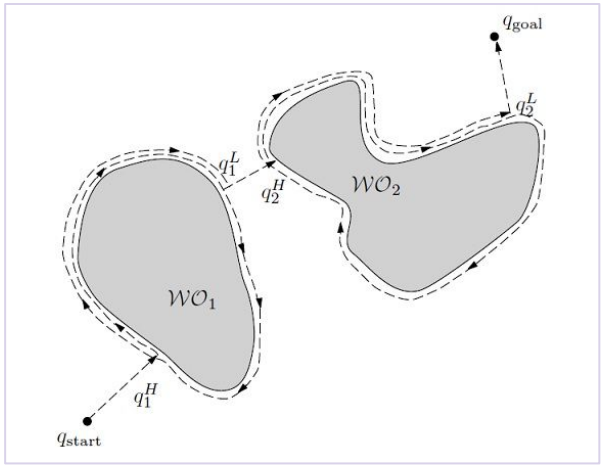
A* Search Algorithm

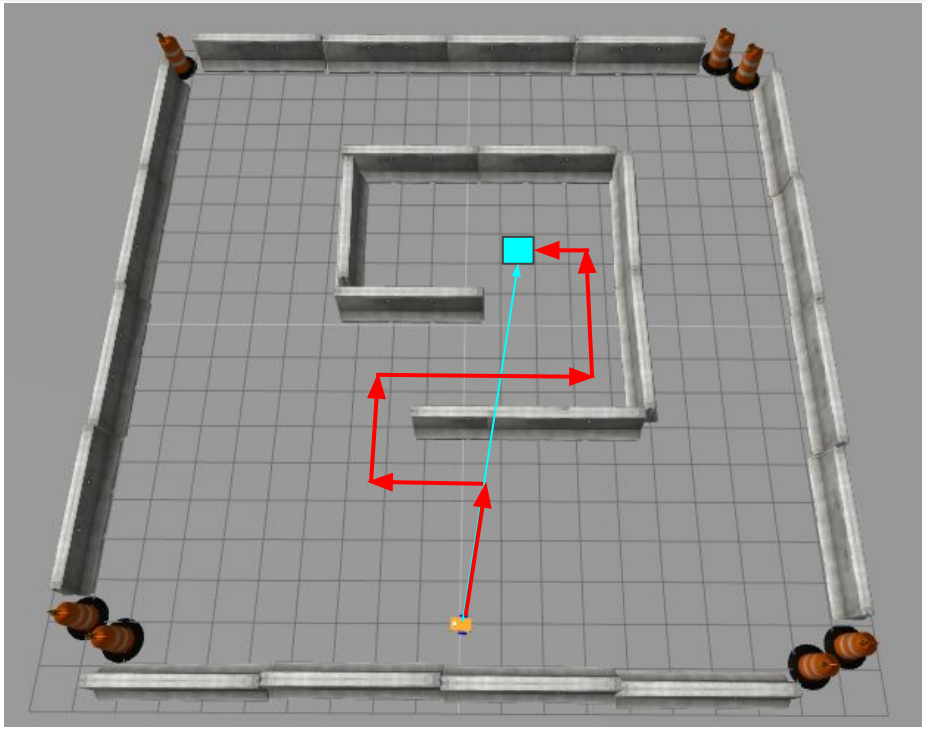
Going backward from the end node can lead us to the optimal path

0	1	2	3	4	5	6	7
8	9	10	11	24 50 12 74	14 54 13 68	10 58 14 68	14 62 15 76
16	17	18	19	20 40 20 60	10 44 21 54	22	10 52 23 62
24	25	26	34 34 27 68	28	29	30	14 42 31 56
32	33	34	35	36	37	38	24 38 39 62
40	41	42	43	44	45	46	34 34 47 68
48	49	50	51	52	58 10 53 68	48 20 54 68	44 30 55 74
56	57	58	59	60	61	62	63

Bug Algorithm: Obstacle Avoidance







See more at: https://aishack.in/tutorials/obstacle-avoidance-bug-algorithm/

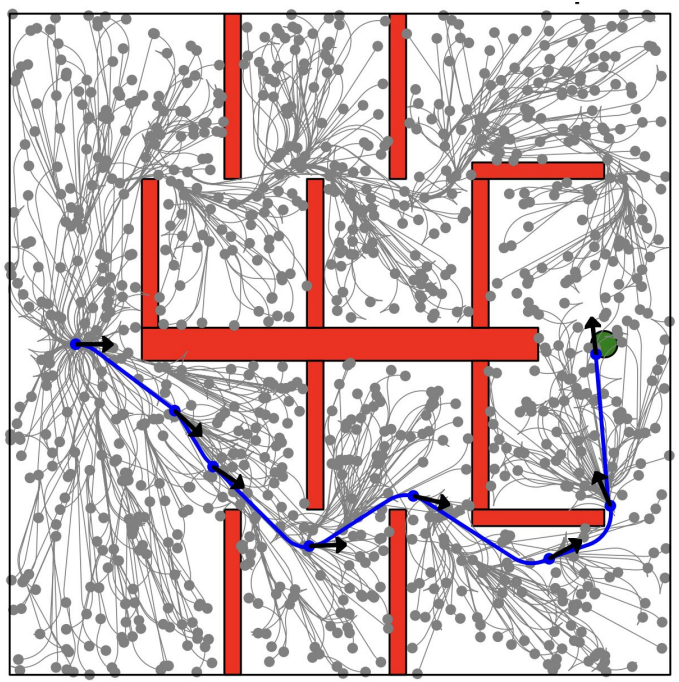
Sampling-based Algorithms

General recipe:

- Seed the graph
- Sample a new node
- Choose to which other node it might connect to
- Decide which edges to add
- Decide which edges to remove

Sampling-based methods advantages:

- Easy and flexible to implement: different collision checking and steering functions can be used for different scenarios and vehicle kinematics
- Robust: can use conservative collision checking and steering functions
- Scalable: scale well with dimensions

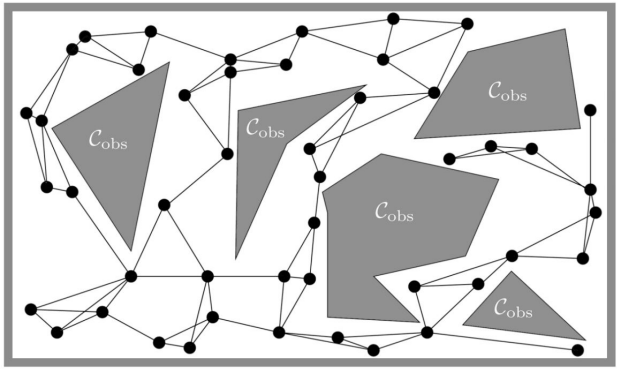


Schmerling et al (ICRA)

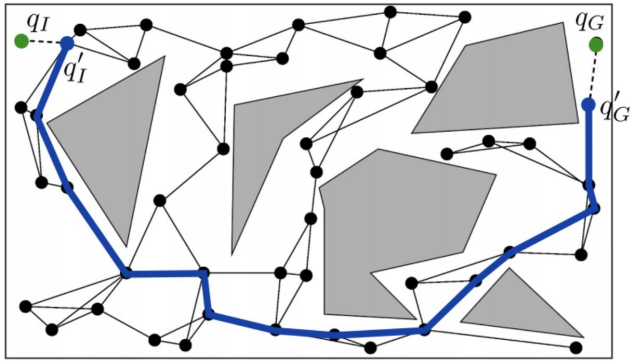
PRM: Probabilistic Road Map

PRM: Probabilistic road map

- PRM algorithm contains two phases
  - Learning phase
    - Sample *n* points in configuration space $\mathcal{C}_{free}$
    - Connect random configurations using a local planner
  - Query phase
    - Connect start and goal configurations with the PRM
    - Use the graph search to find the path
- Probabilistic completeness
- Efficient if we need multiple queries on the same graph



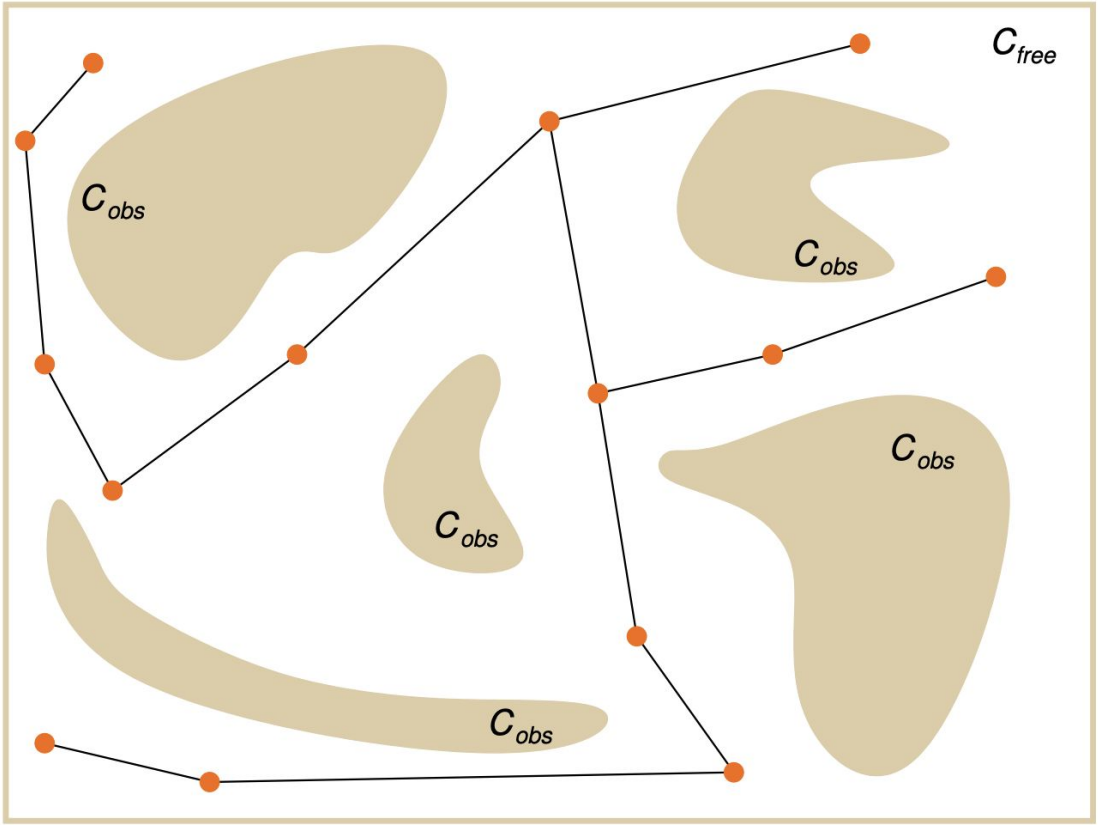
Learning phase: Build a roadmap by sampling free space



Query phase: Use search-based algorithm to find valid path

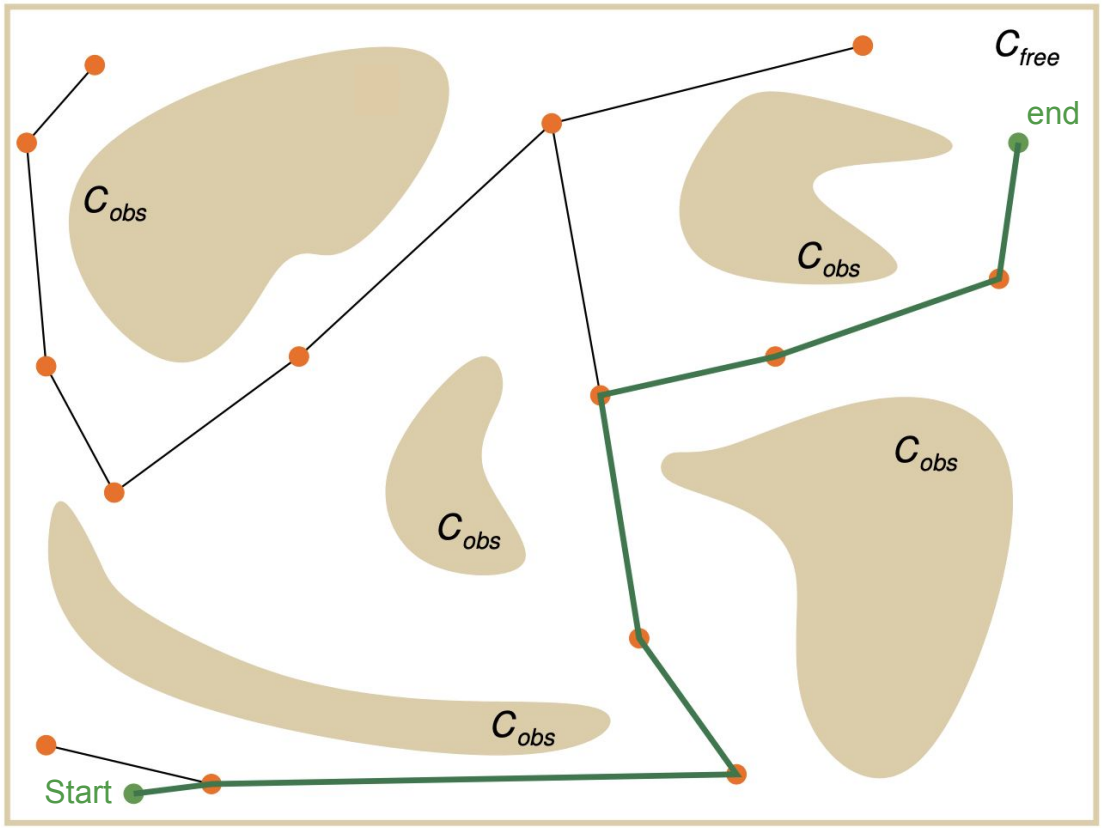
PRM: Probabilistic Road Map

- **Learning phase**: Given problem domain
  - Define free configuration space (C_{free}) and obstacles (C_{obs})
- **Learning phase**: Random configuration
  - Sample at random from configuration space
- **Learning phase**: Connecting samples
  - Points are connected to the nearest *k* neighbors if a local planner guarantees collision-free paths
- **Learning phase**: Connected roadmap



PRM: Probabilistic Road Map

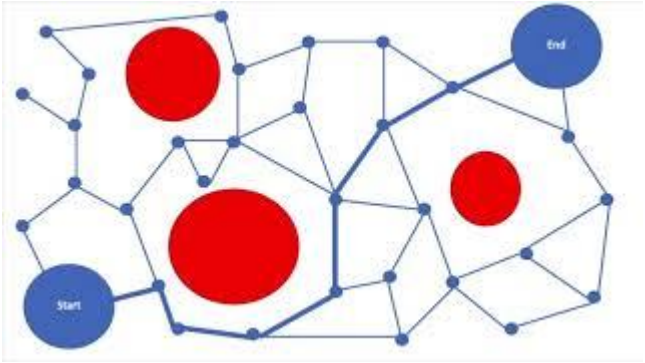
- **Query phase**: Query configurations
  - Query start and goal configurations
- **Query phase**: Final found path
  - Apply graph search to find final path



RRT: Rapidly-exploring Random Trees

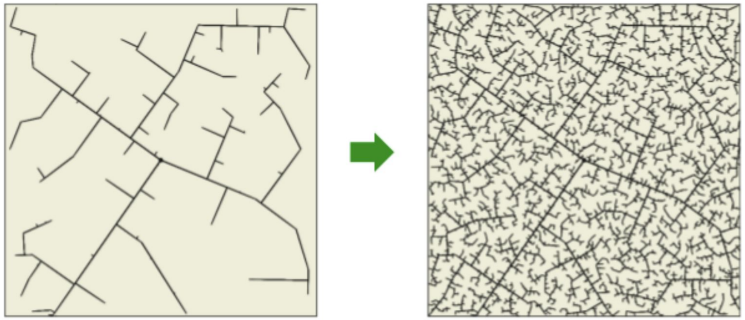
PRM: *Multi-query algorithm*

- Pros:
  - Generate a single roadmap that is then used for planning queries several times
- Cons:
  - Narrow passages
  - Lack of connectivity
  - Not suitable for dynamic environments



RRT: *Single-query algorithm*

- For each planning problem constructs a new roadmap to characterize the subspace that is relevant to the problem
- Idea: aggressively probe and explore the configuration space by expanding incrementally
- Create a tree instead of graph: no graph search needed!
- Much more efficient than PRM if only a single query needed



RRT: Rapidly-exploring Random Trees

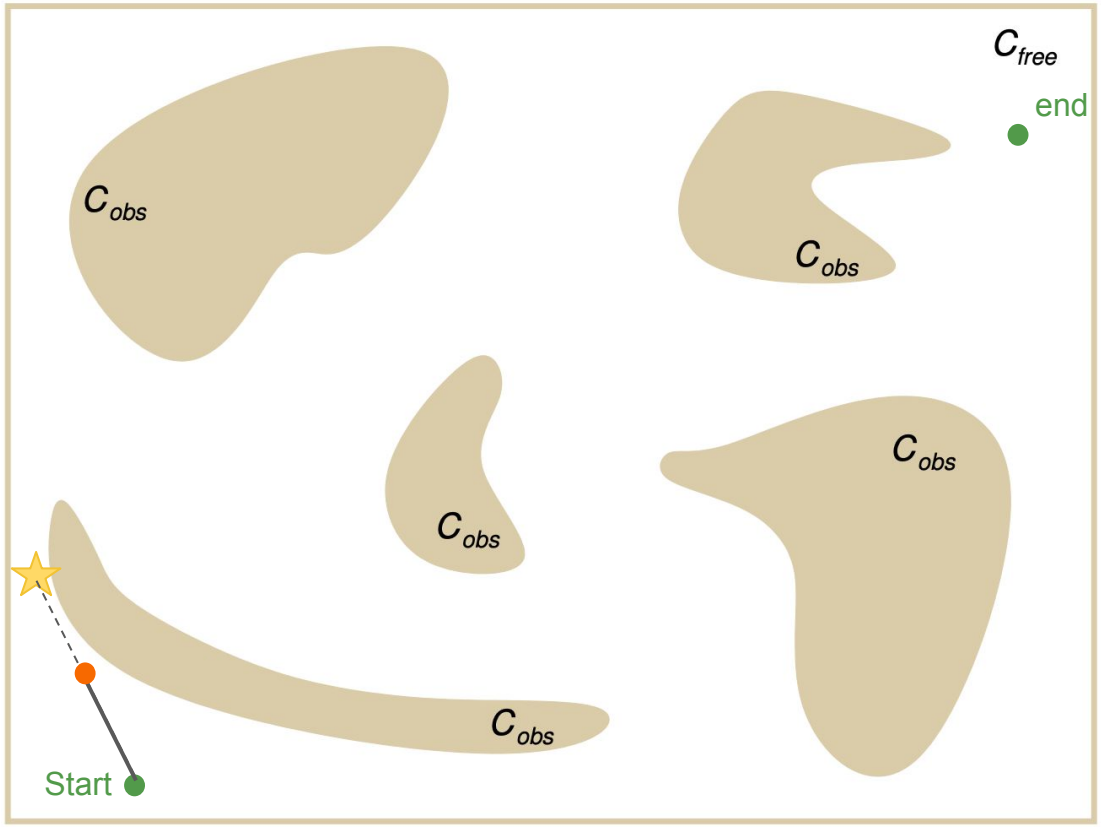
Define free configuration space and obstacles

Define start point and goal point

Starting by choosing a random point in free space

Then create a line between the new point and the start node

Grow the tree from the start node by placing a new point at a distance of delta from the start node along that line as long as it guarantees collision-free paths

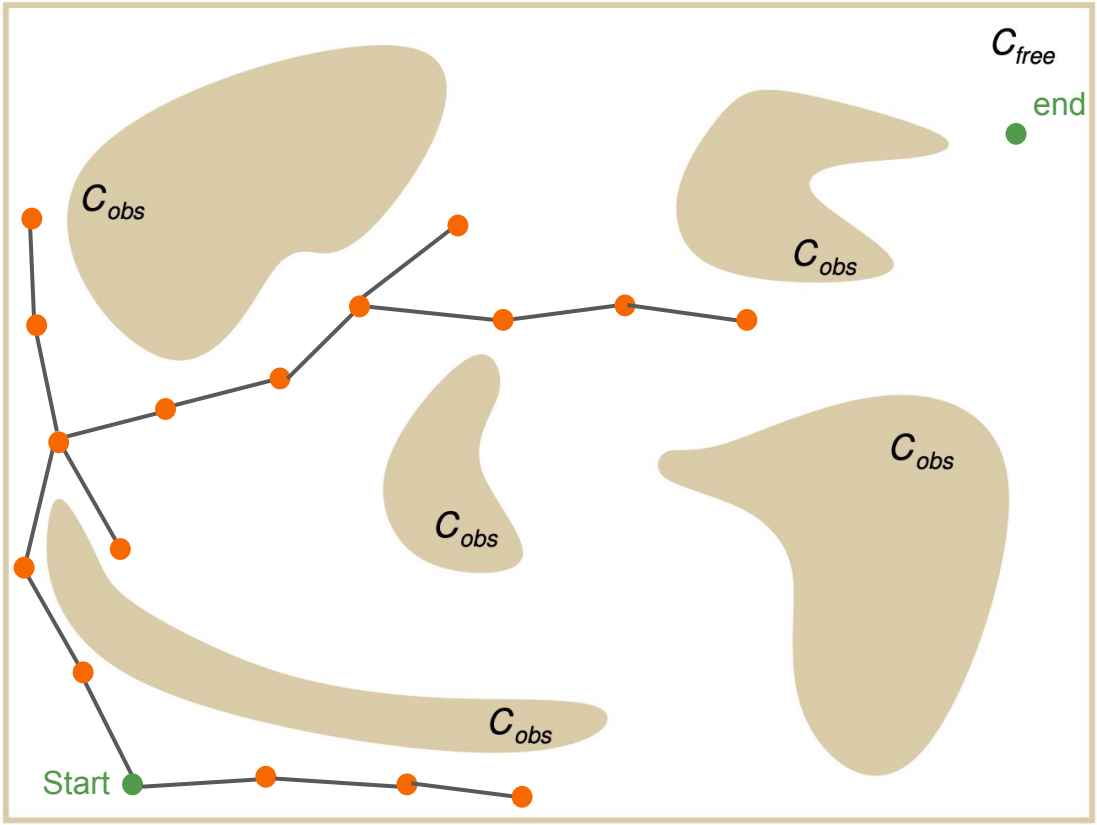


RRT: Rapidly-exploring Random Trees

On each iteration of RRT, follow the same process to add new points and expand the tree

Each time a new points is added, an edge is created to the nearest node with the tree

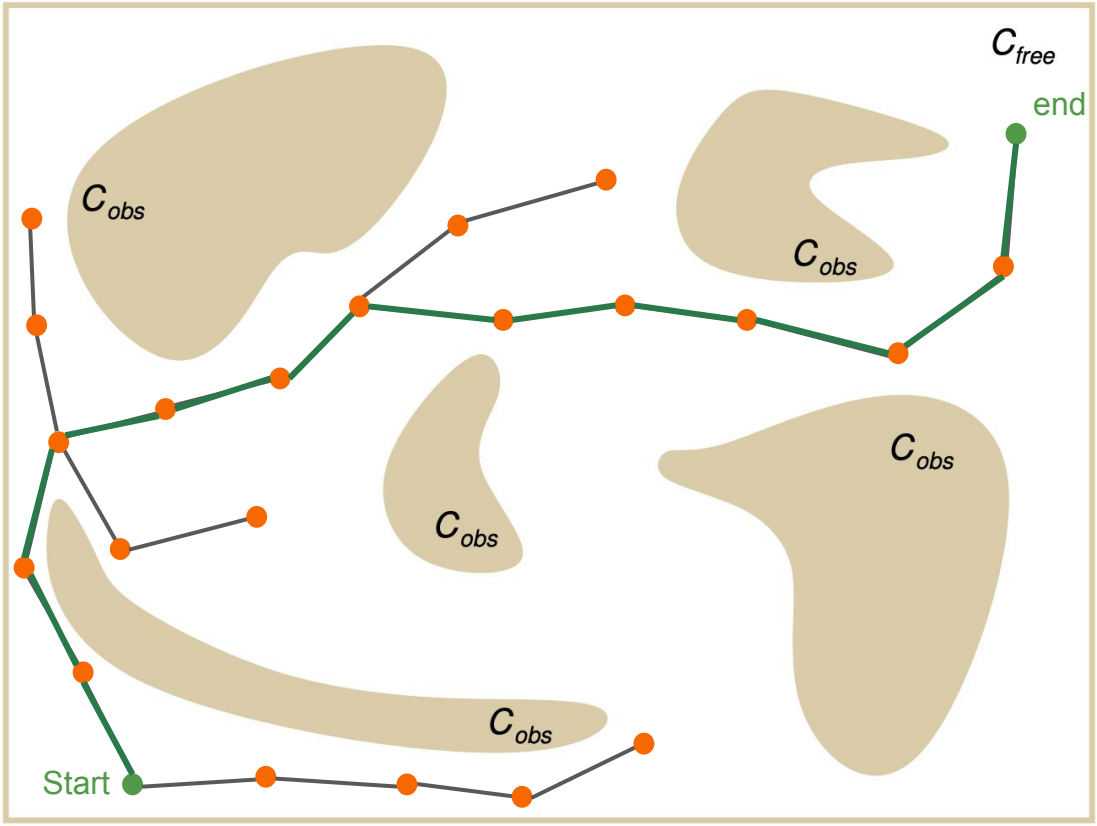
The tree continues to expand in to the free space following this pattern



RRT: Rapidly-exploring Random Trees

Once in a while, use the goal position instead of a random sample to increase the chance of finding the path

Once the tree reaches the goal node, we can return the path from the start node to the goal node



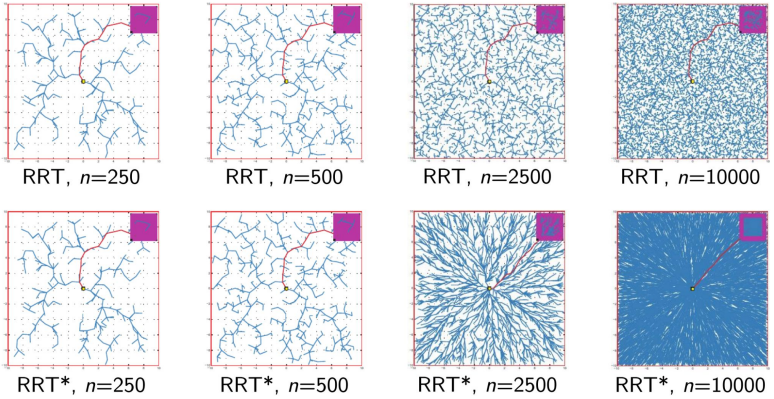
RRT*

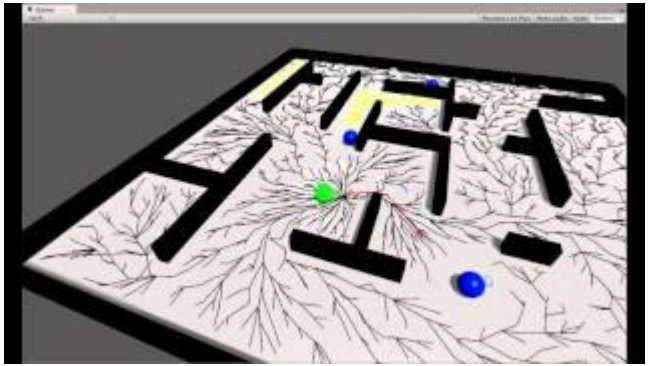
RRT:

- Rapidly explores the space
- Probabilistic completeness
- Not find optimal paths

RRT*: *Optimal version of RRT*

- Differs from RRT in two ways:
  - Parent selection, using cost-to-come
  - Rewiring step
- Asymptotically optimal and complete
- Sacrifices speed, considers optimal path to everything





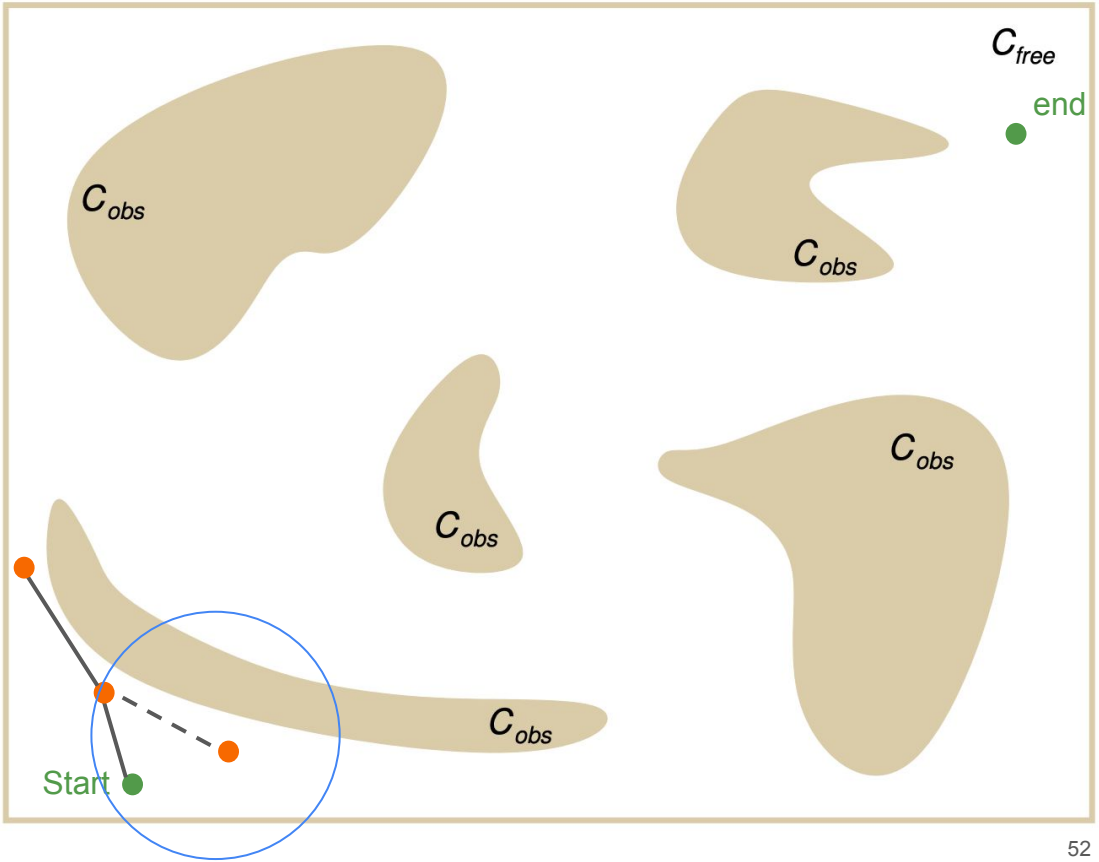
RRT*

Define free configuration space and obstacles

Define start point and goal point

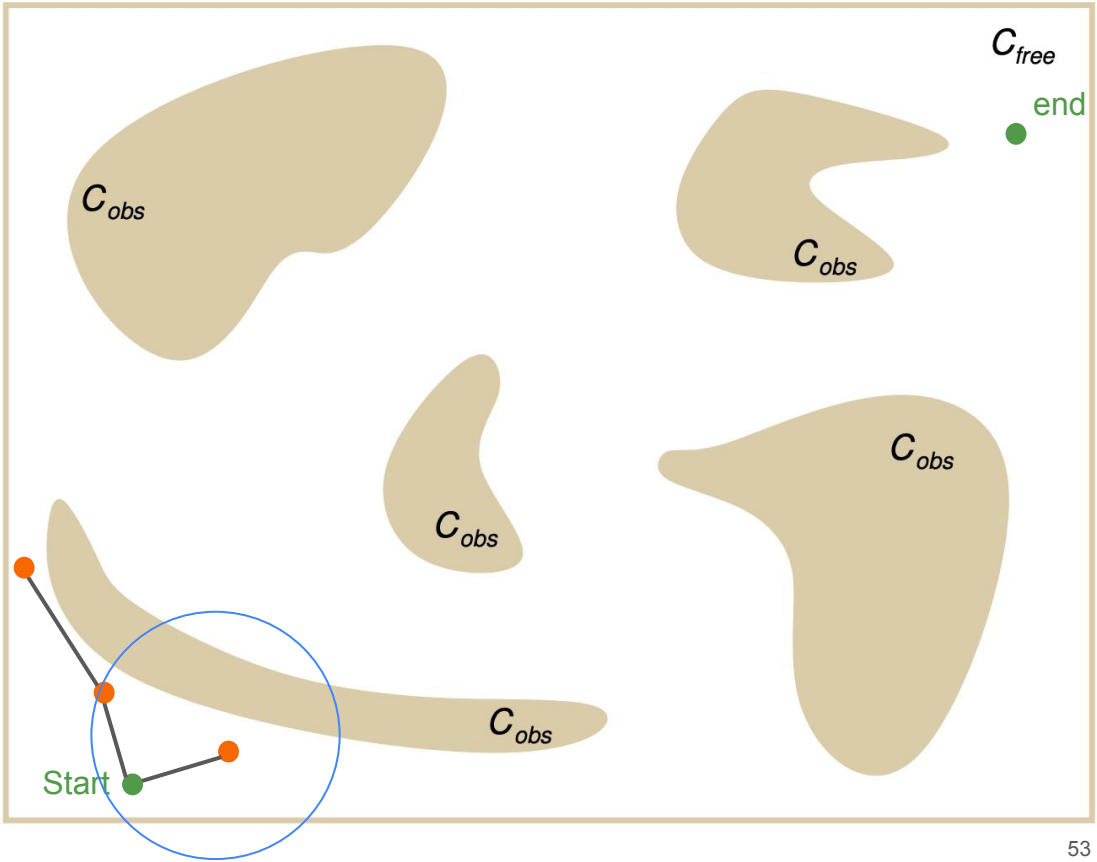
RRT* works similarly to RRT

One modification from RRT:
When a point is added, neighboring points are checked within a certain radius



RRT*

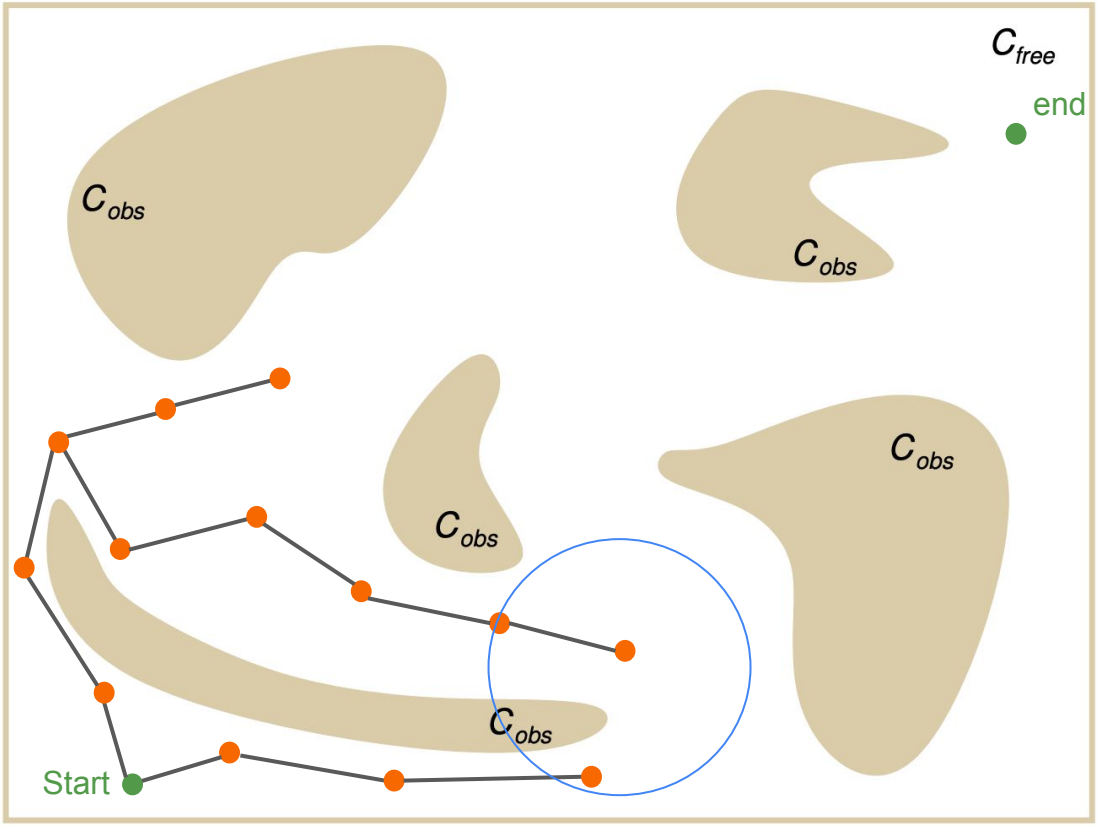
If a neighboring point results in a shorter path to the start node, that node is used as this node's parent



RRT*

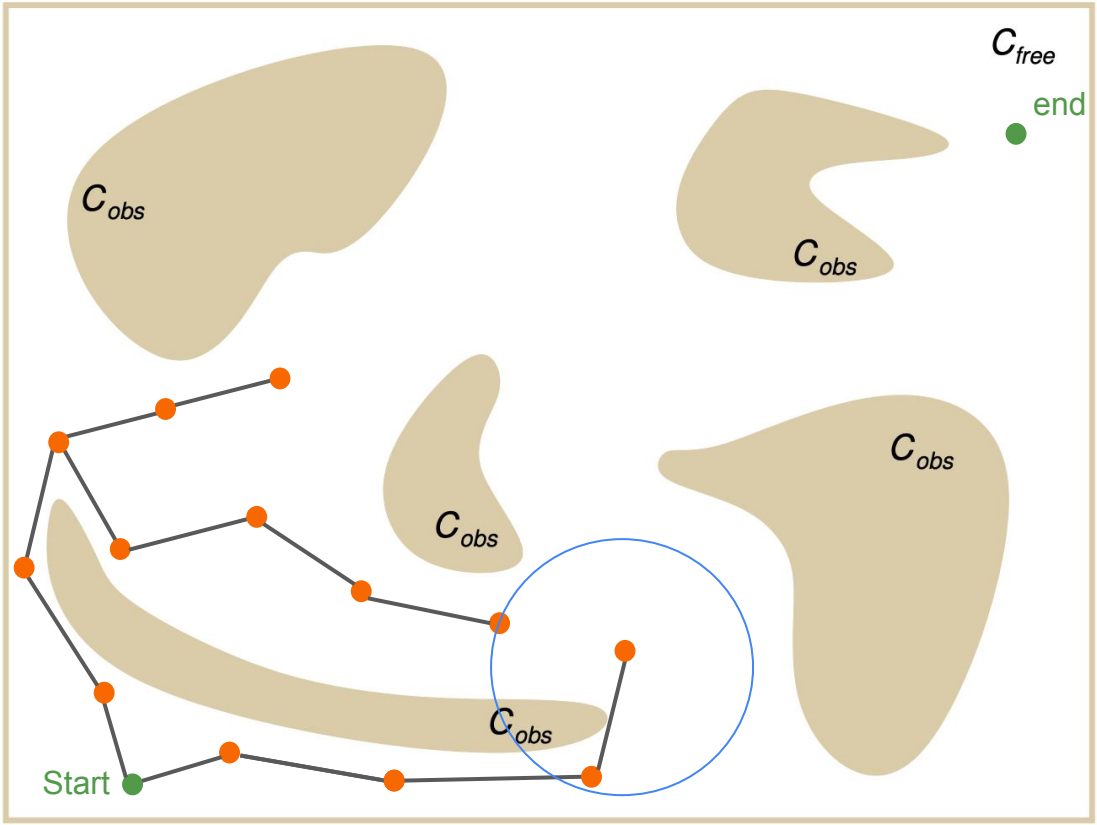
The tree continues to expand in to the free space following this pattern

Another modification from RRT:
If a neighbor's cost to the start node can be decreased by rerouting it through a new node



RRT*

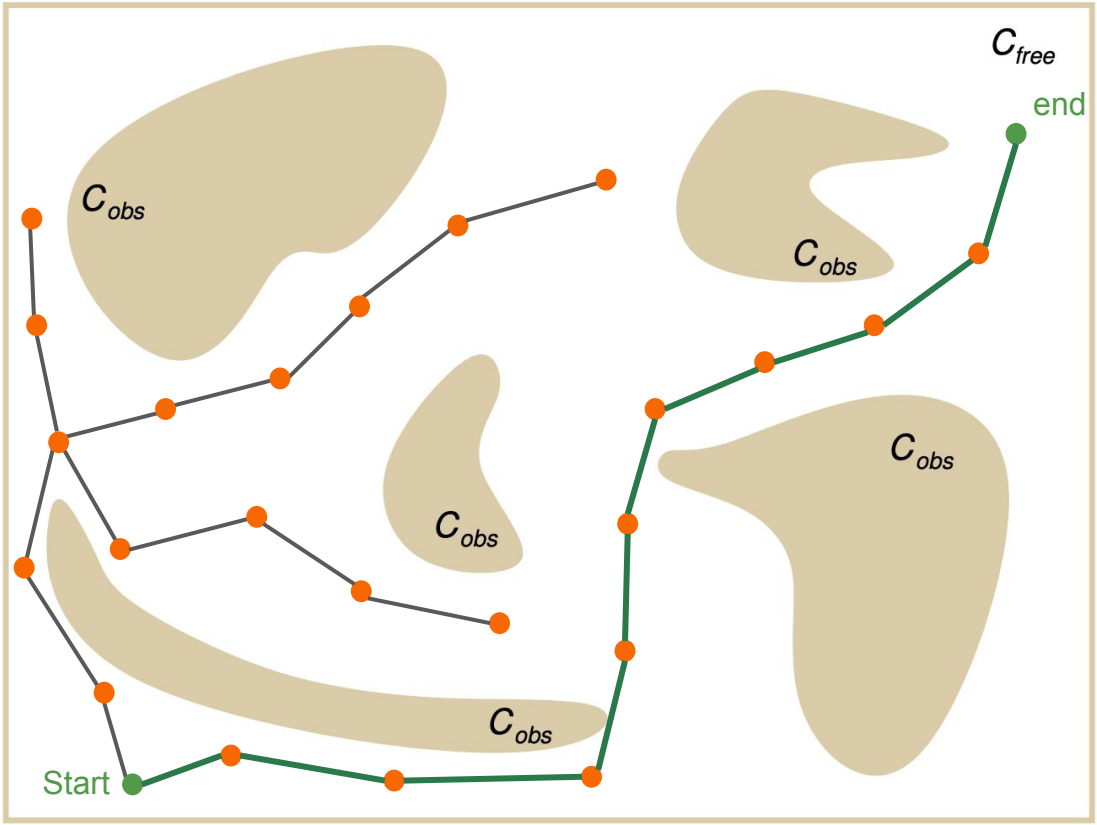
Then the node is rewired with this new node



RRT*

Once in a while, use the goal position instead of a random sample to increase the chance of finding the path

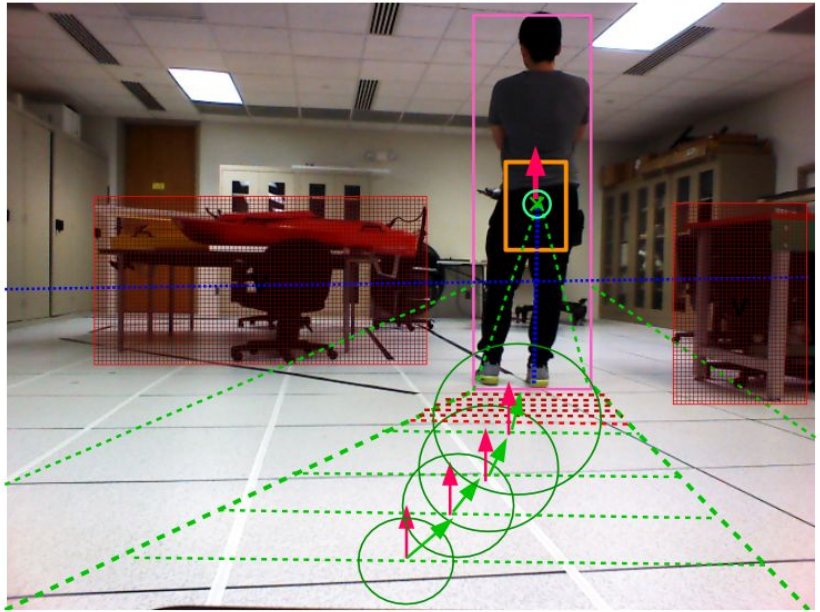
The optimal path is finally found



Target-Centric Planning

With no prior global map

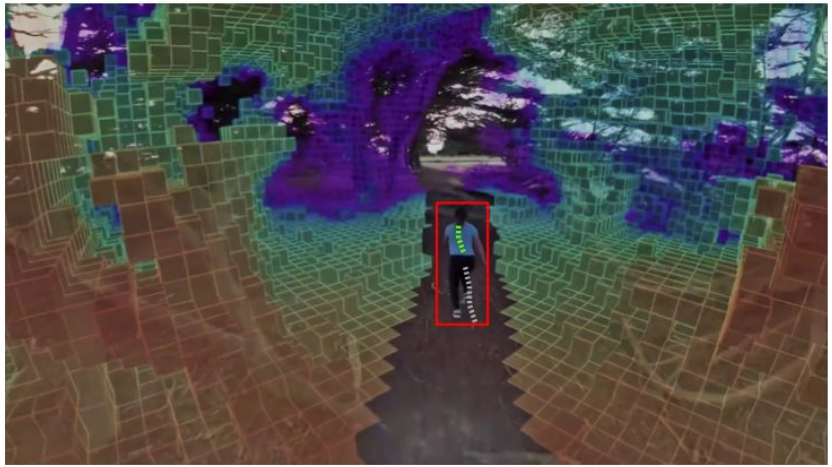
- Perform SLAM: create + maintain a dynamic map
- Execute task specific planning
  - Target tracking / following
  - Source-destination navigation
- **Pros:**
  - Faster planning! do not worry about the whole environment - focus on the target
  - Avoid obstacles as they come, no need to plan ahead of time
  - Adaptive and
- **Cons:**
  - No optimality guarantees
  - Dynamic agents make things challenging
  - Robots task execution performance largely depends on the accuracy of SLAM



Person-following ground robot:

- Detect person and estimate distance
- Estimate ground plane (homography)
- Plan straight line distance to remain immediately behind the person

Target-Centric Planning



Person-following aerial robot:
- Multi-camera visual inertial SLAM
- Detect and track person in the 3D map
- Any target tracker, obstacle avoidance, and efficient trajectory controller

https://www.youtube.com/@Skydio

How to lead instead of follow?
- Leading a human companion
  - Use cases: guide robots in museums/airports
- Following while staying ahead (drones)
  - Use cases: entertainment and sports

Islam et al

Advanced Topics

Advanced topics for the ‘Robotics II’ course

- Planning under uncertainty
- Visual servoing
  - Partially observable image-defined paths
  - Multi-robot collaborative planners
- Active planners, online learning-based algorithms
- Planning for Human-Robot Interaction (HRI)
  - Explicit and implicit interaction
  - Socially-complaint motion planning
- Reinforcement Learning (RL)-based algorithms
  - Optimal value RL
  - Policy gradient RL
- Learning from Demonstration (LfD) and imitation learning-based models
- Next-best-view planners