CSCI/ECEN 3302
Introduction to Robotics

Alessandro Roncone

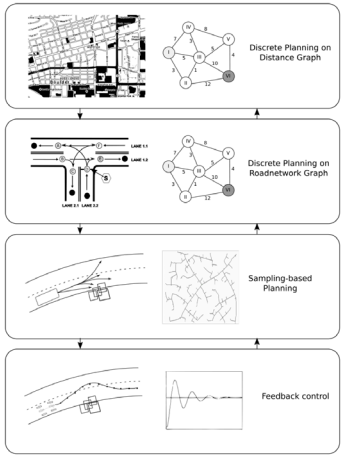
aroncone@colorado.edu https://hiro-group.ronc.one

Administrivia

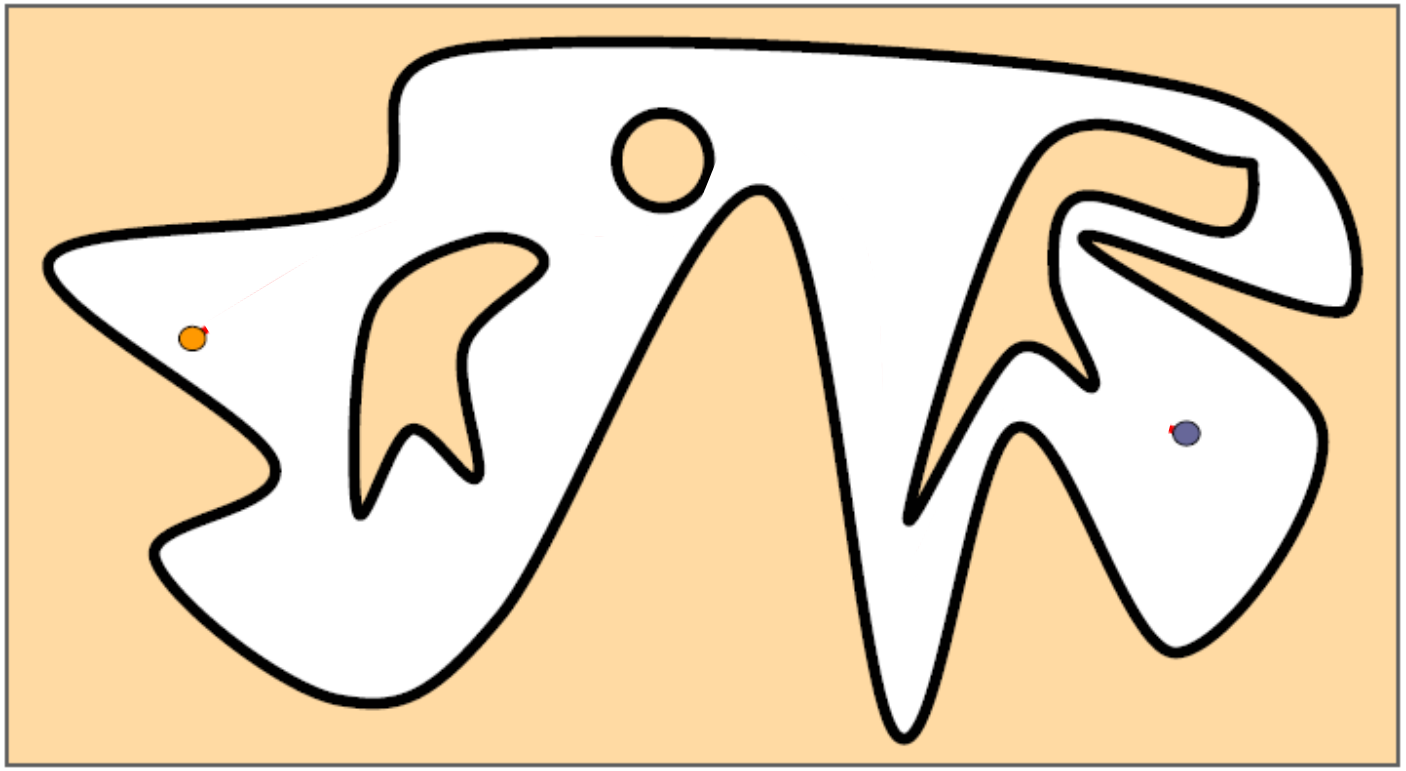
- Lab 2
  - Grades are out
  - Next in line: HW1, then Lab 3, then Lab 4/5
- Lab 4
  - Deadline tonight!
- Lab 5
  - Three weeks long
  - To be concluded before the Spring Break
- HW2 to come up soon – depending on how much I can cover today vs next Tue

Chapter 13

RECAP
World Representations

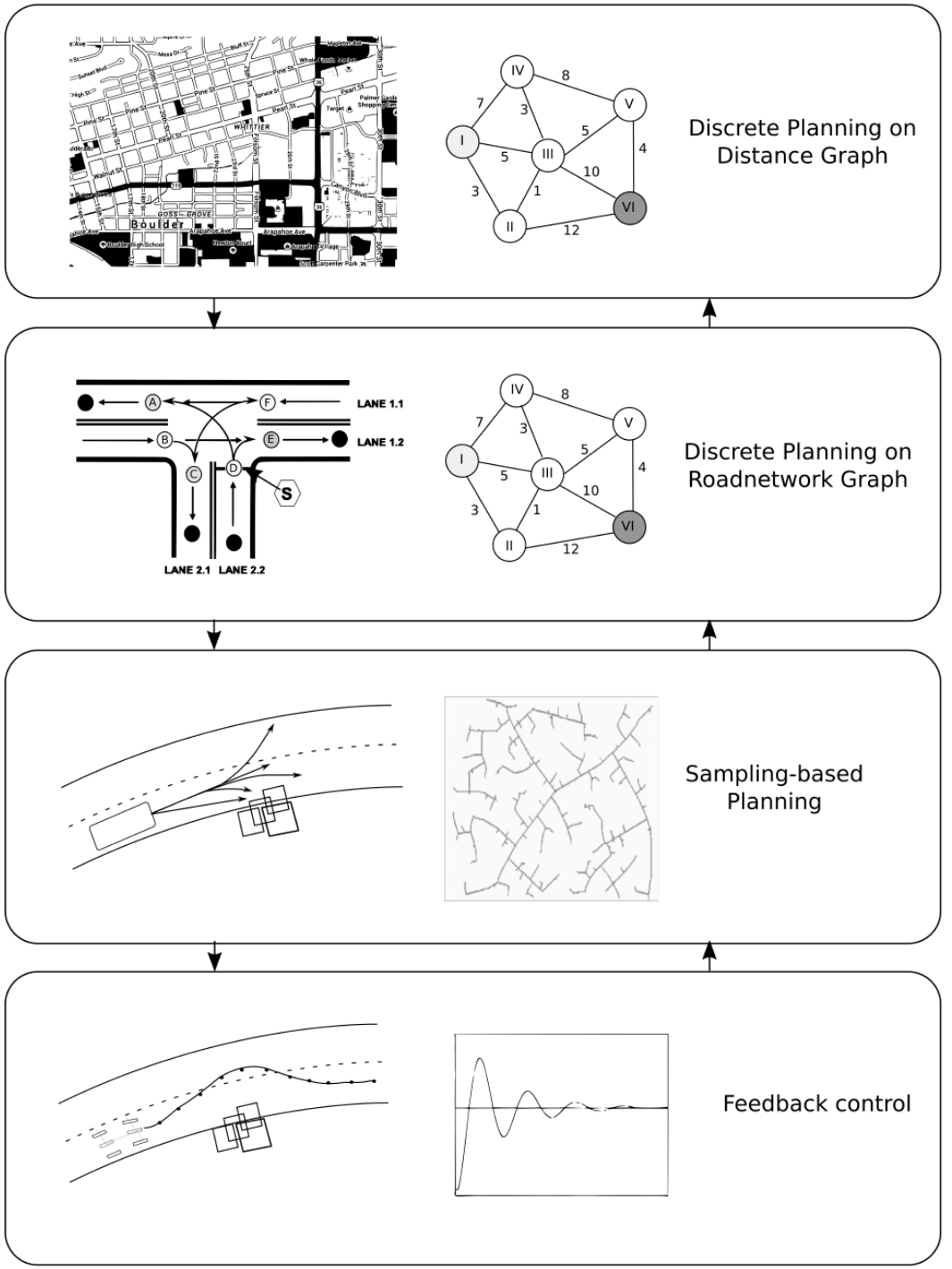


Moving from Start to Goal state



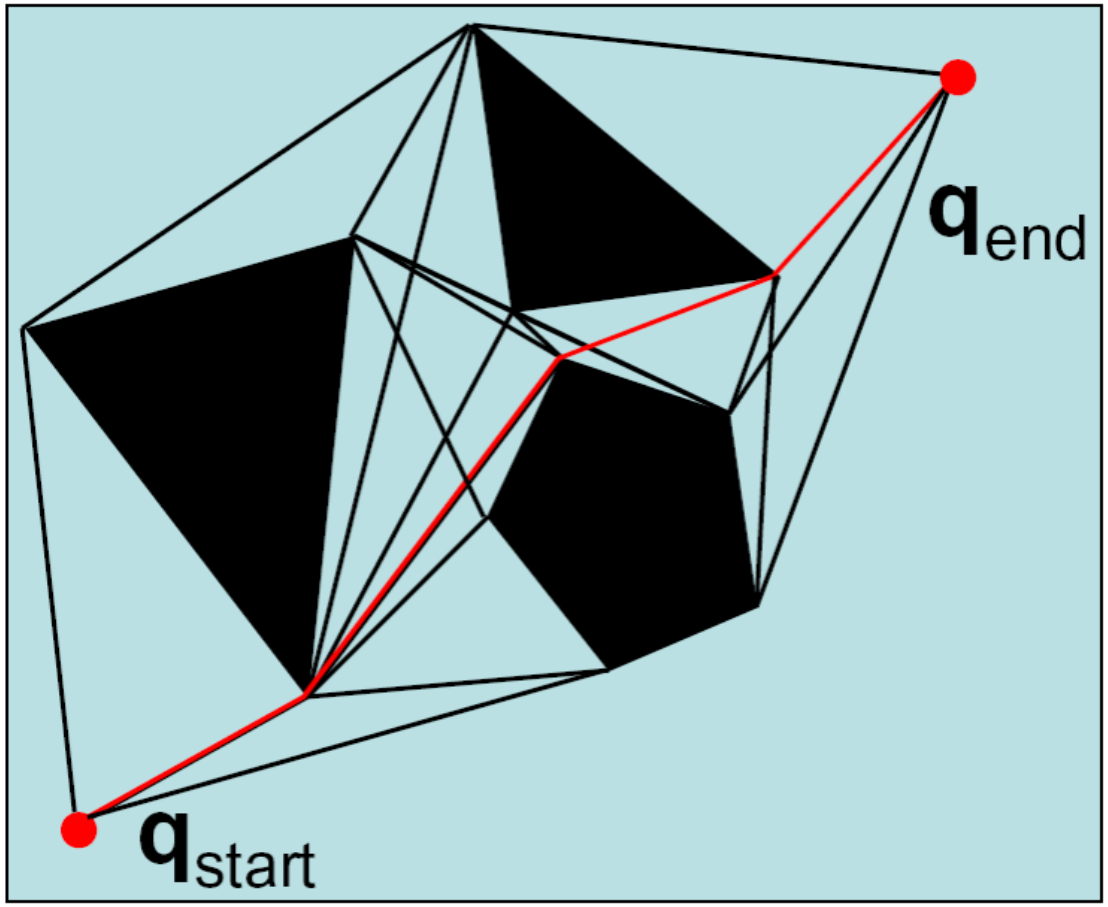
- General approach: Reduce what amounts to an intractable problem in continuous C-space to a tractable problem in a discrete space.
- Tools:
  - Environment Representations
  - Search Algorithms

Planning across length scales



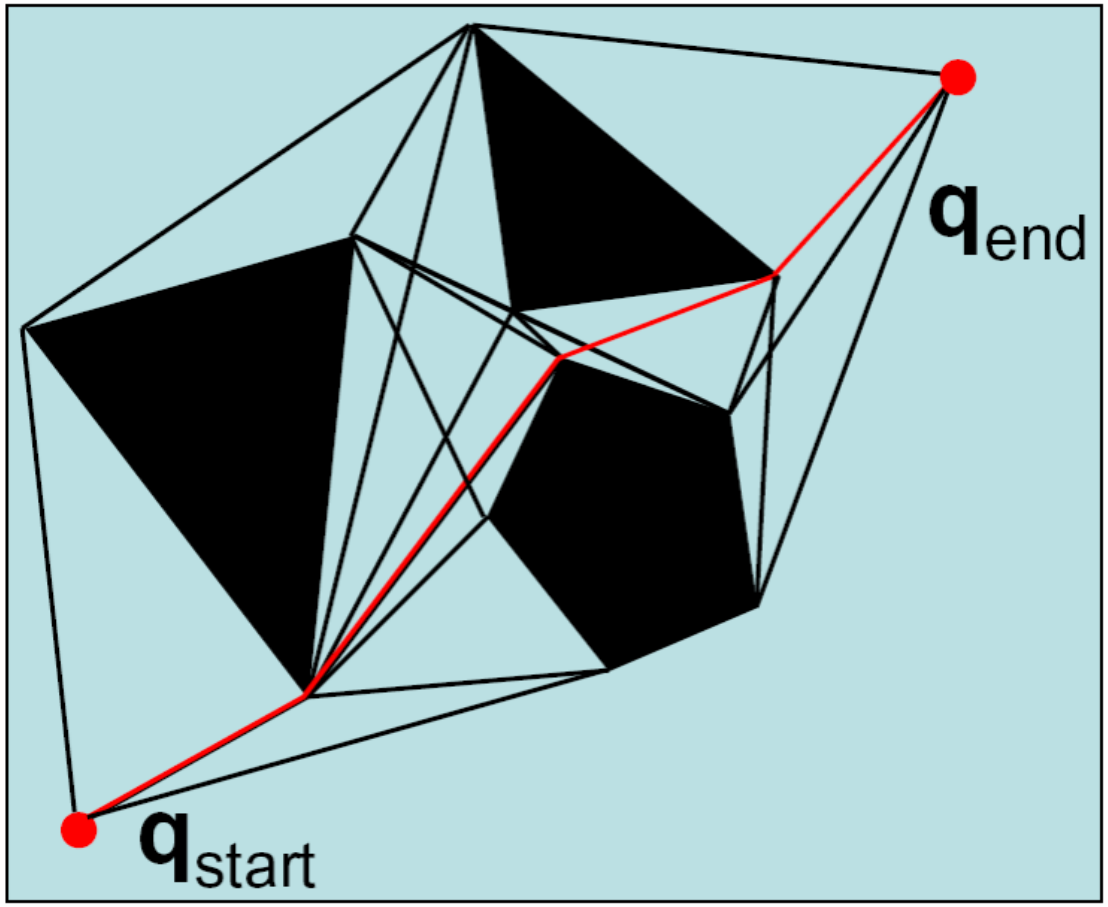
Some Well-Known Representations

- Visibility Graphs
- Roadmap
- Cell Decomposition
- Potential Field



Some Well-Known Representations

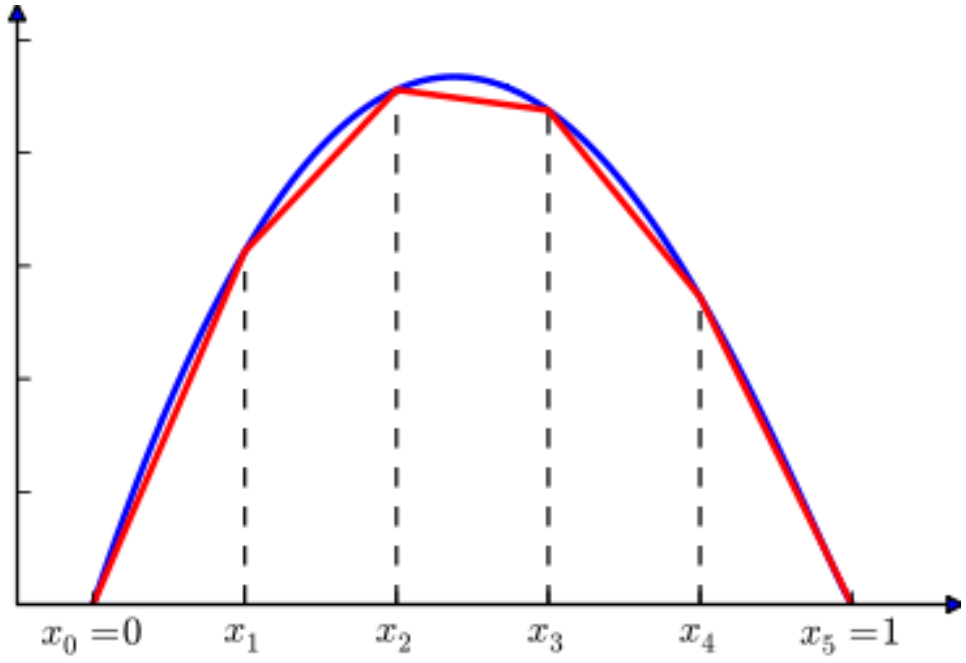
- Visibility Graphs
- Roadmap
- Cell Decomposition
- Potential Field



Visibility Graph Method

- If there is a collision-free path between two points, then there is a polygonal path that bends only at the obstacles vertices.

- A polygonal path is a piecewise linear curve:

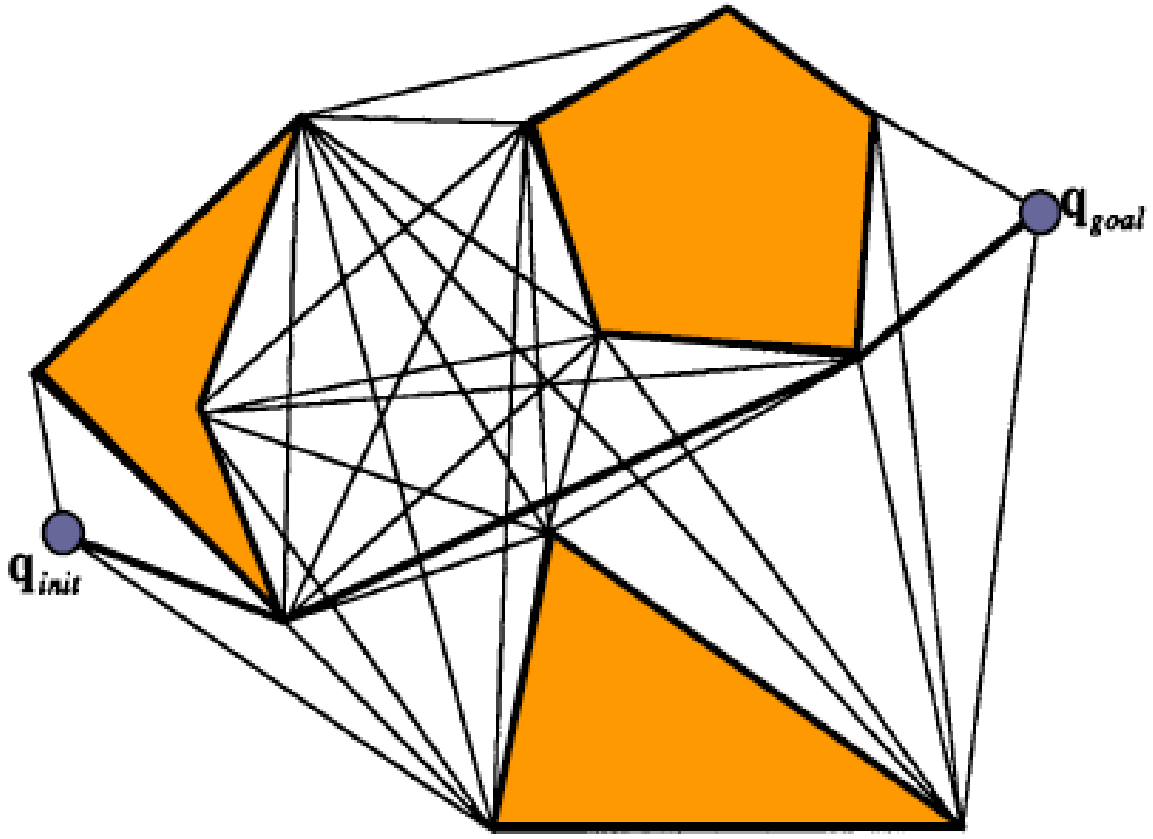


Visibility Graph Algorithm

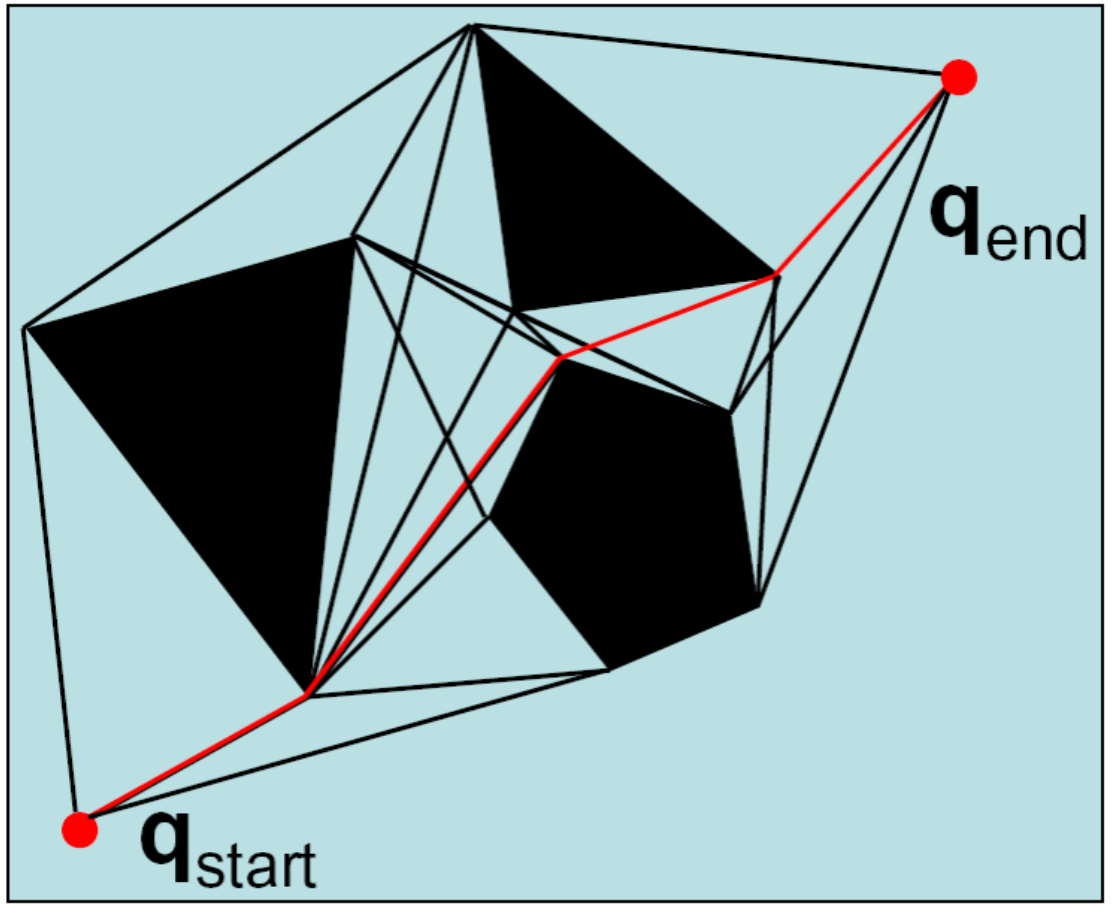
Input: q_{init}, q_{goal}, polygonal obstacles

Output: visibility graph G

```
1.    for every pair of nodes u,v
2.      if segment (u,v) is an obstacle edge then
3.        insert edge (u,v) into G;
4.      else
5.        for every obstacle edge e
6.          if segment (u,v) intersects e
7.            go to (1);
8.        insert edge (u,v) into G.
```

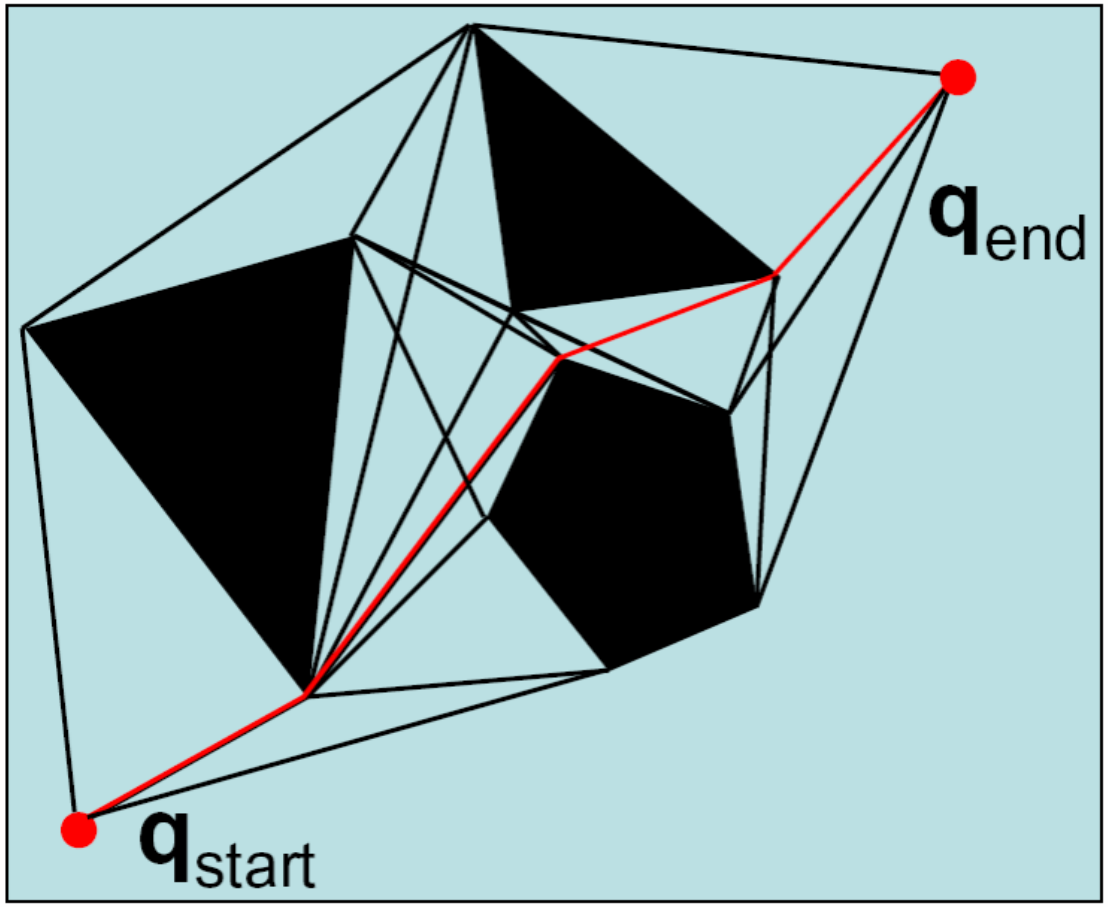


Visibility Graph: Strengths/Weaknesses?

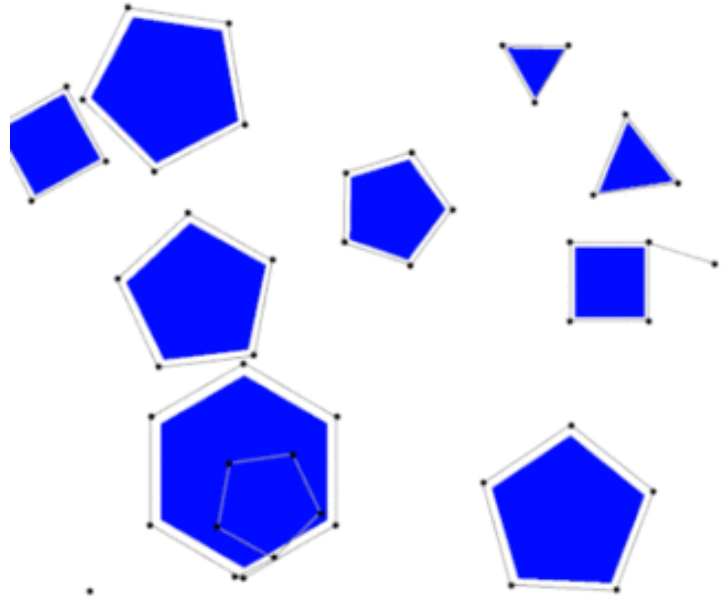


Some Well-Known Representations

- Visibility Graphs
- Roadmap
- Cell Decomposition
- Potential Field



Road mapping

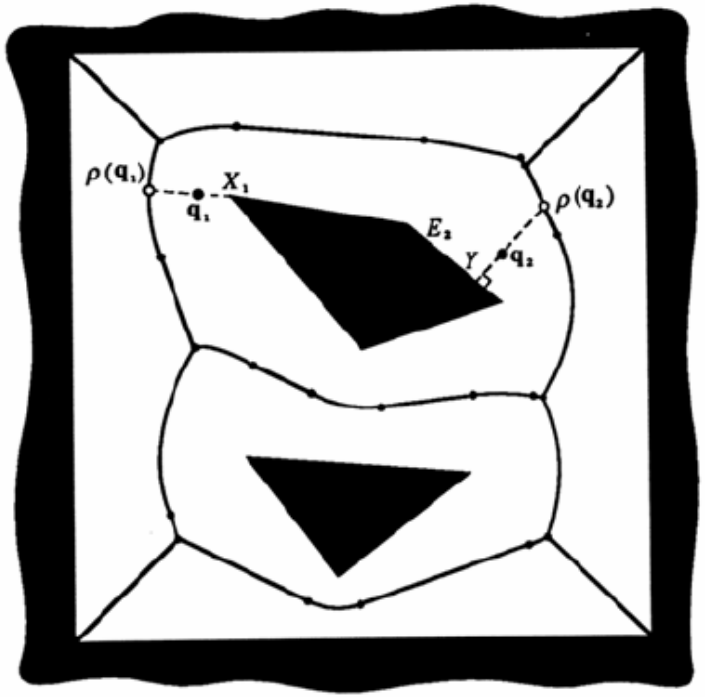


Probabilistic Road Mapping

- Take random samples, test them to see they are in free space, and connect them via nearest neighbor
- Use a graph search algorithm to find the optimal path

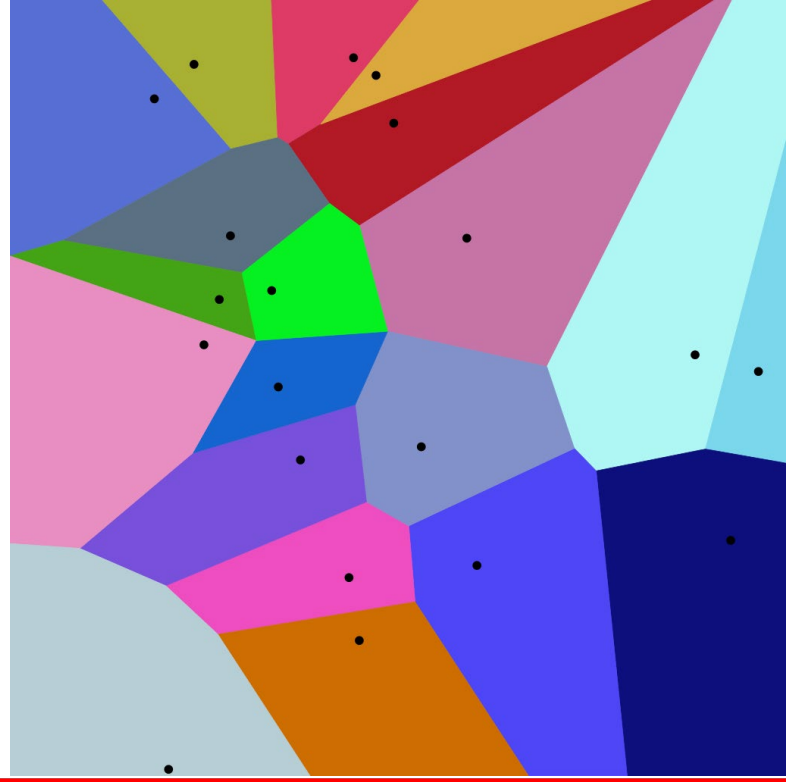
Voronoi Road Mapping

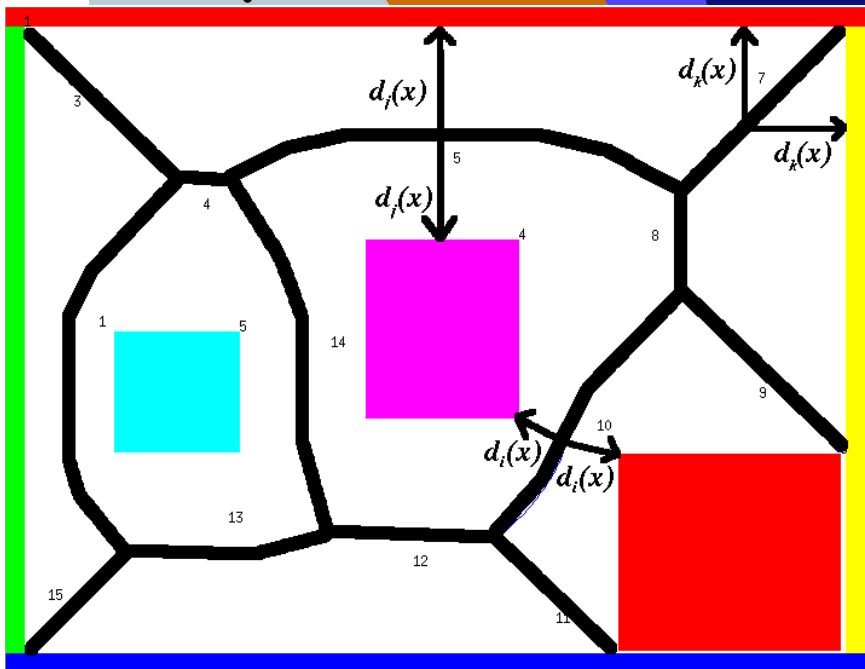
- Introduced by computational geometry researchers.
- Generate paths that maximize clearance
- Applicable mostly to 2-D configuration spaces



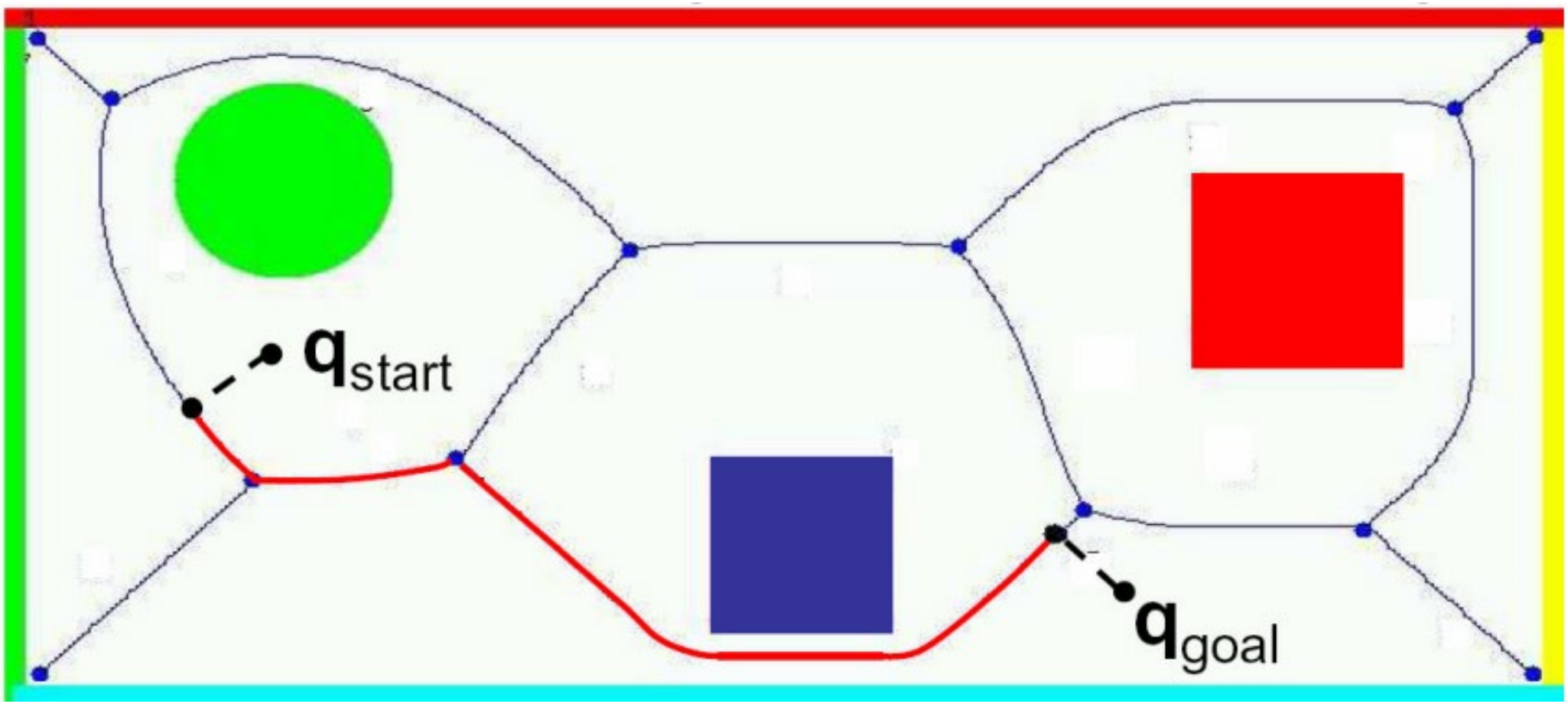
Voronoi Road Mapping

- Voronoi Diagram: partition of a plan with n points into n convex polygons whose points are closer to the generating point than any other
  - It is based on the topology of the environment
  - It creates a skeleton-like structure obtained via retraction, i.e. by progressively shrinking the free space.
  - Solutions have maximum clearance!



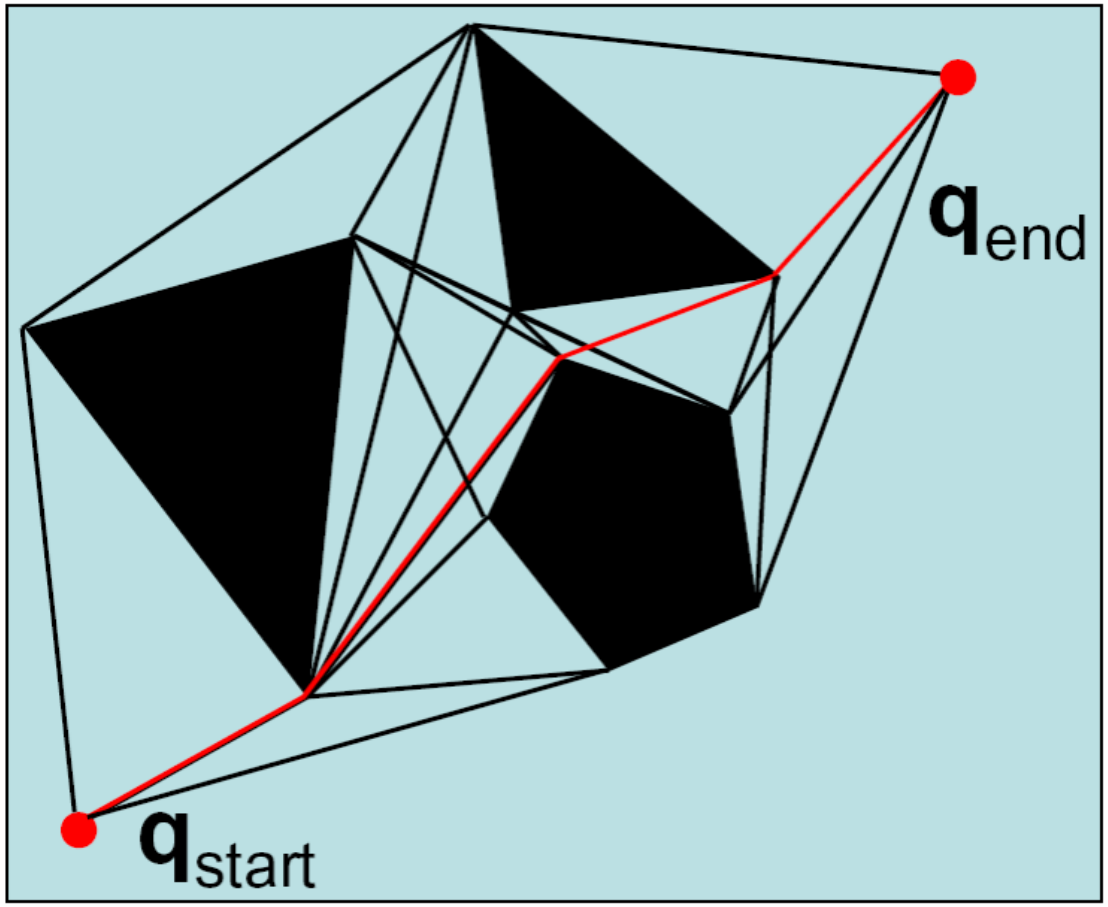


Voronoi Road Mapping: Strengths / Weaknesses

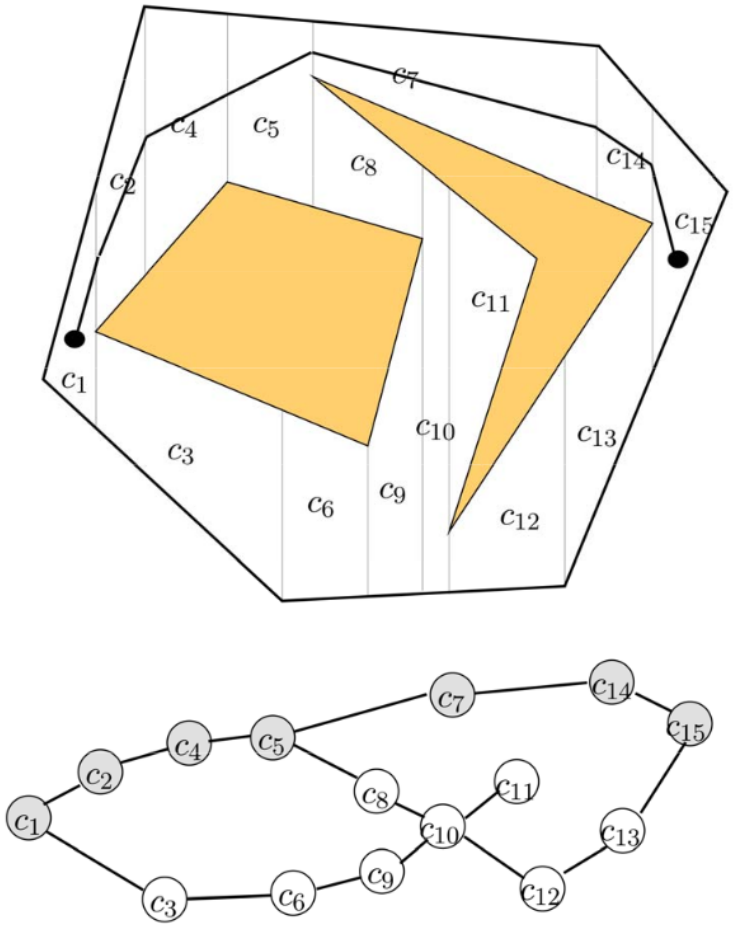


Some Well-Known Representations

- Visibility Graphs
- Roadmap
- Cell Decomposition
- Potential Field



Cell Decomposition



Exact cell decomposition

- Divides a space **F** precisely into sub-units

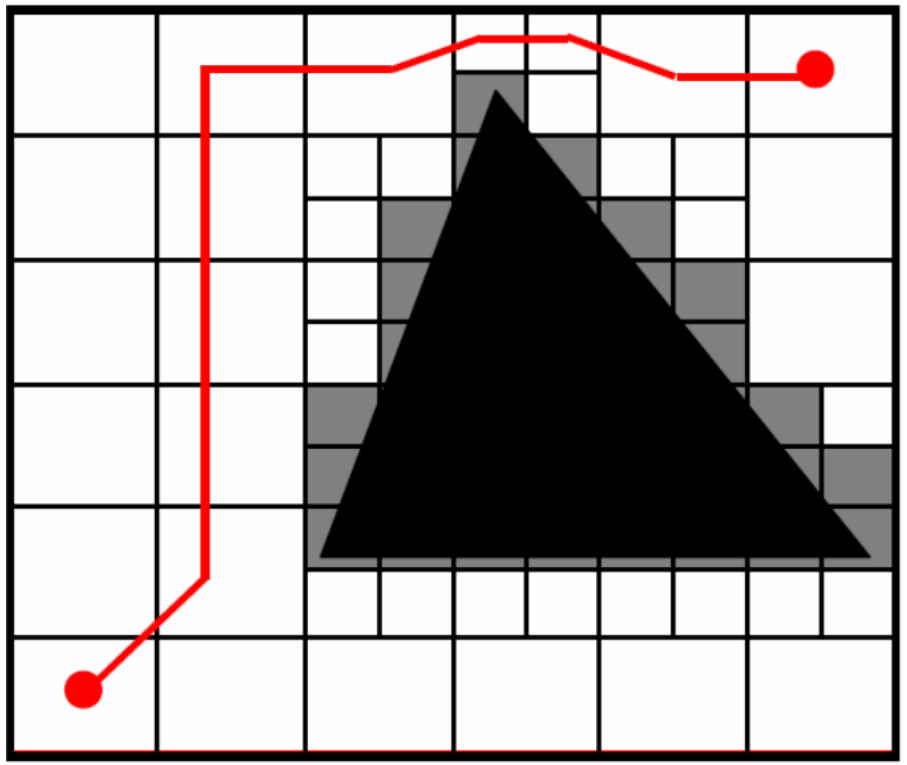
Approximate cell decomposition

- ***F*** is represented by a collection of non-overlapping cells whose union is contained in ***F*.**
- Cells usually have simple, regular shapes, e.g., rectangles, squares.
- Facilitates hierarchical space decomposition

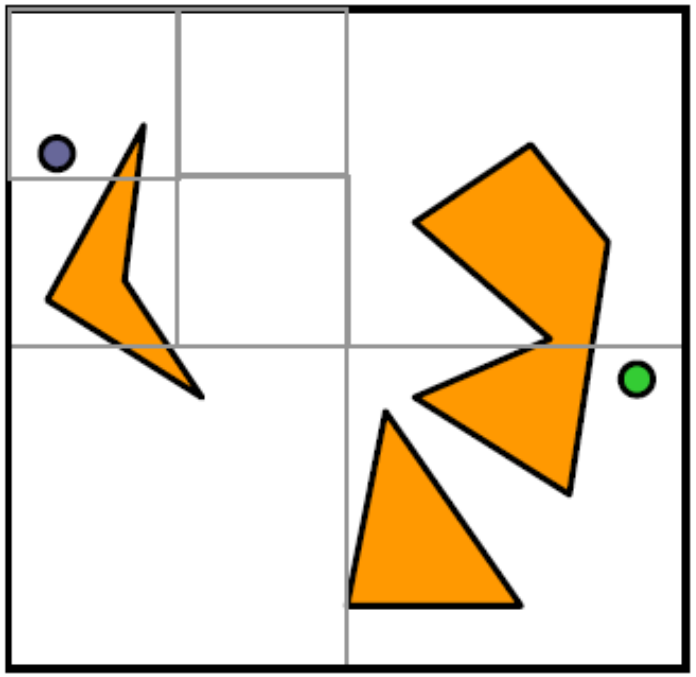
Cell Decomposition

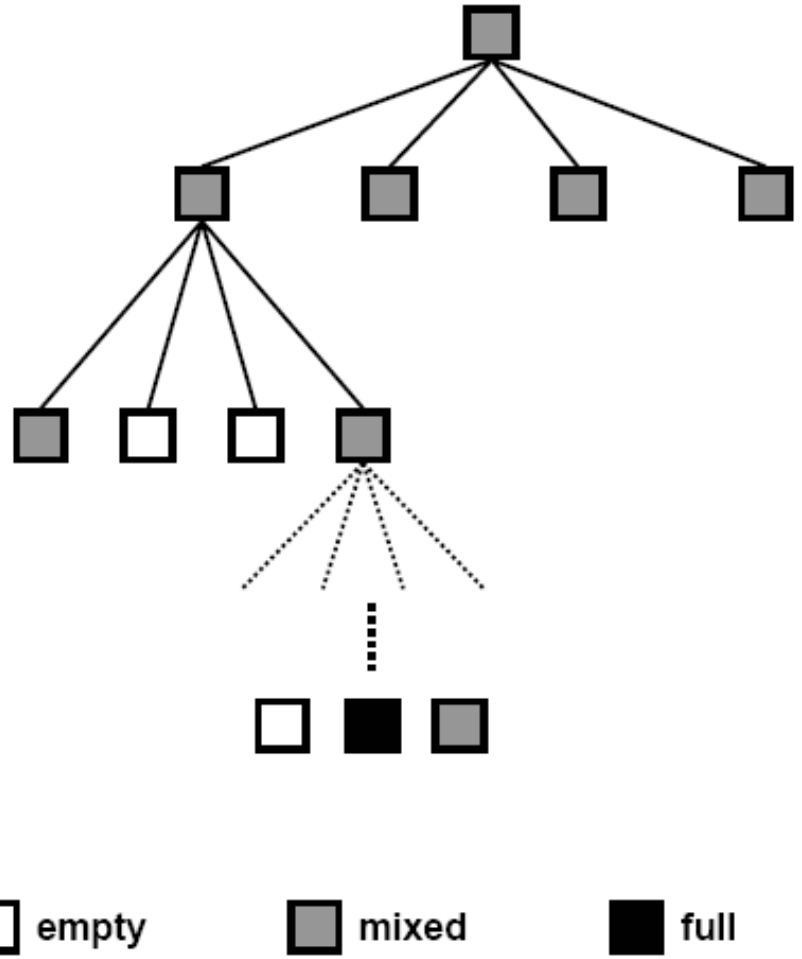
Not necessarily *complete*

(Complete: If a solution exists, it will eventually be found)

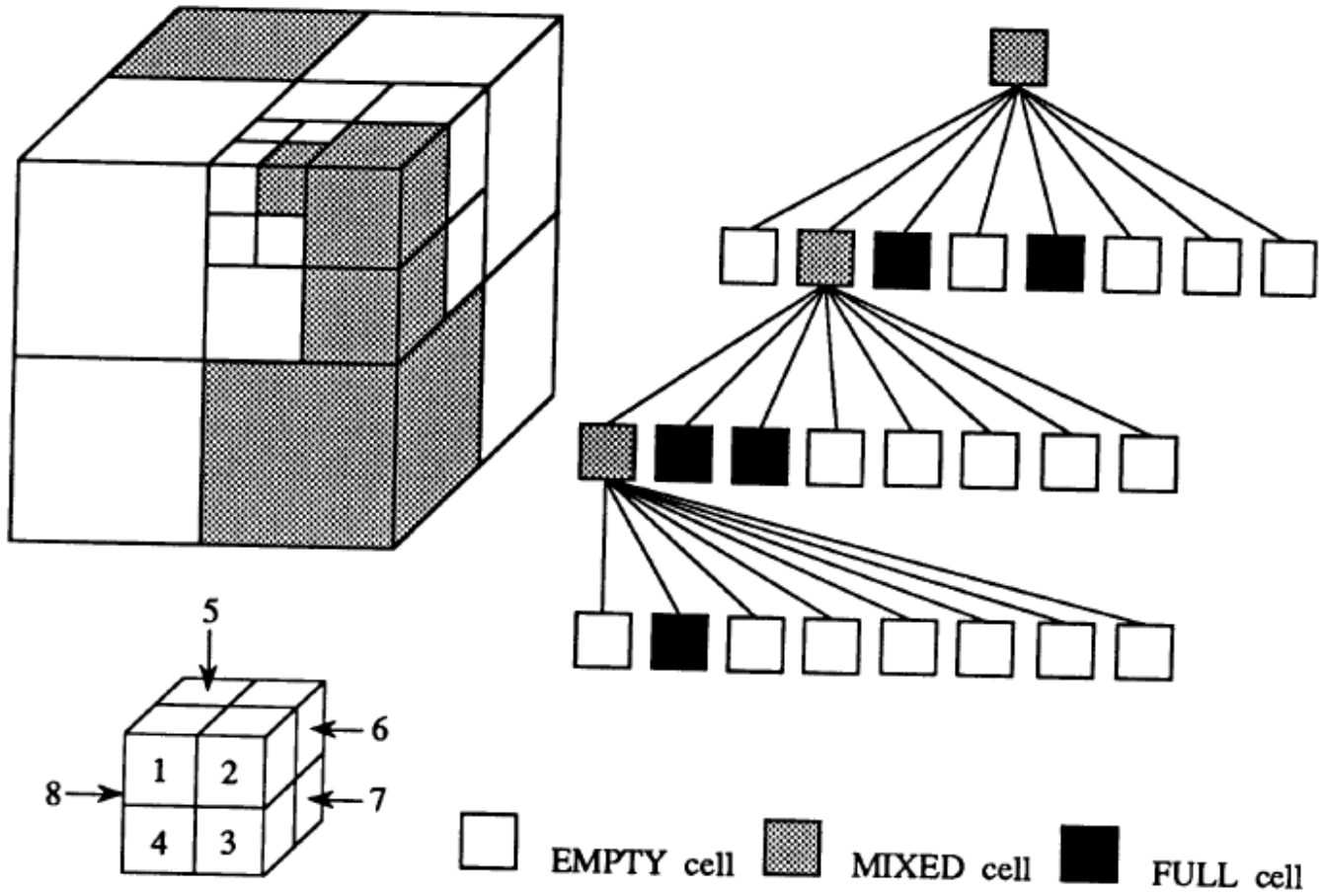


Quadtree Decomposition



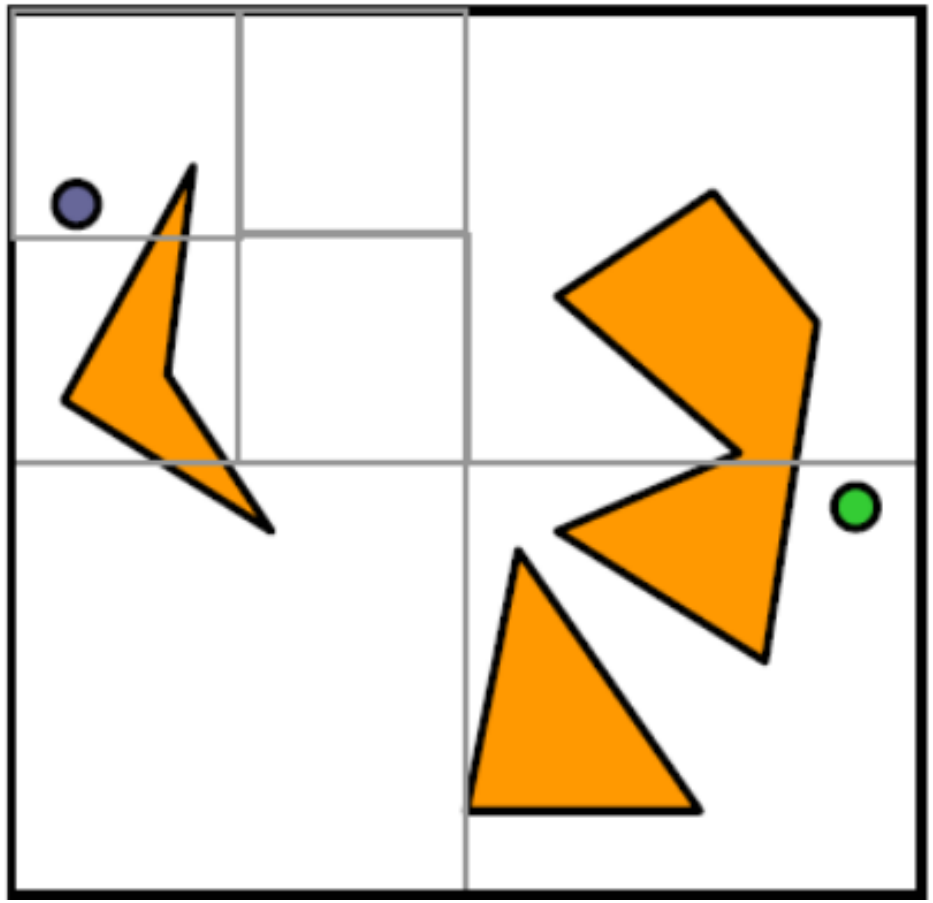


Octree Decomposition

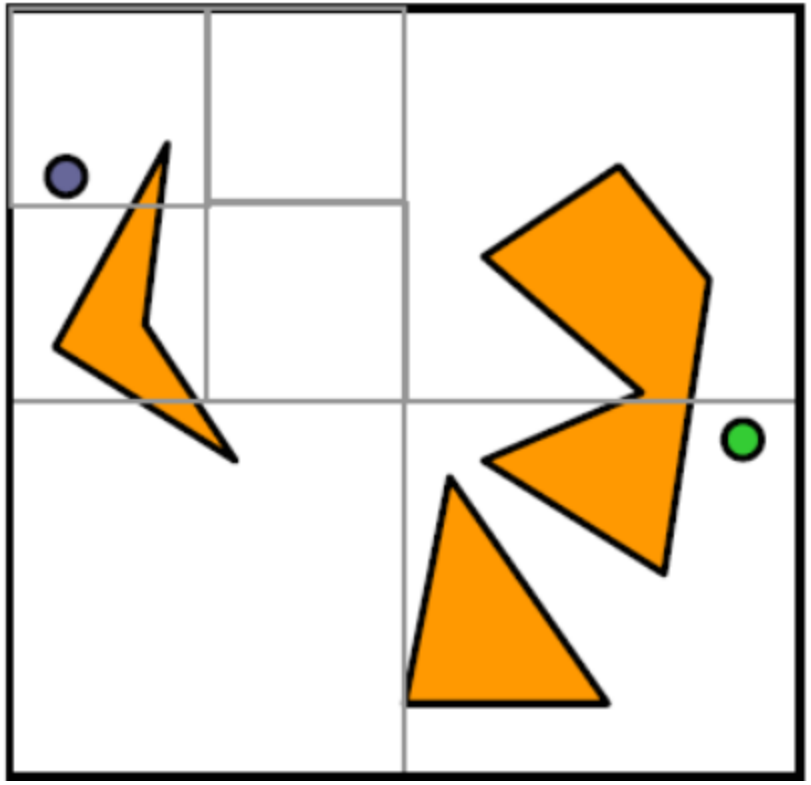


Cell Decomposition: Algorithm Outline

1. Decompose the free space F into cells
2. Search for a sequence of mixed or free cells that connect the initial and goal positions
3. Further decompose the mixed
4. Repeat (2) and (3) until a sequence of free cells is found

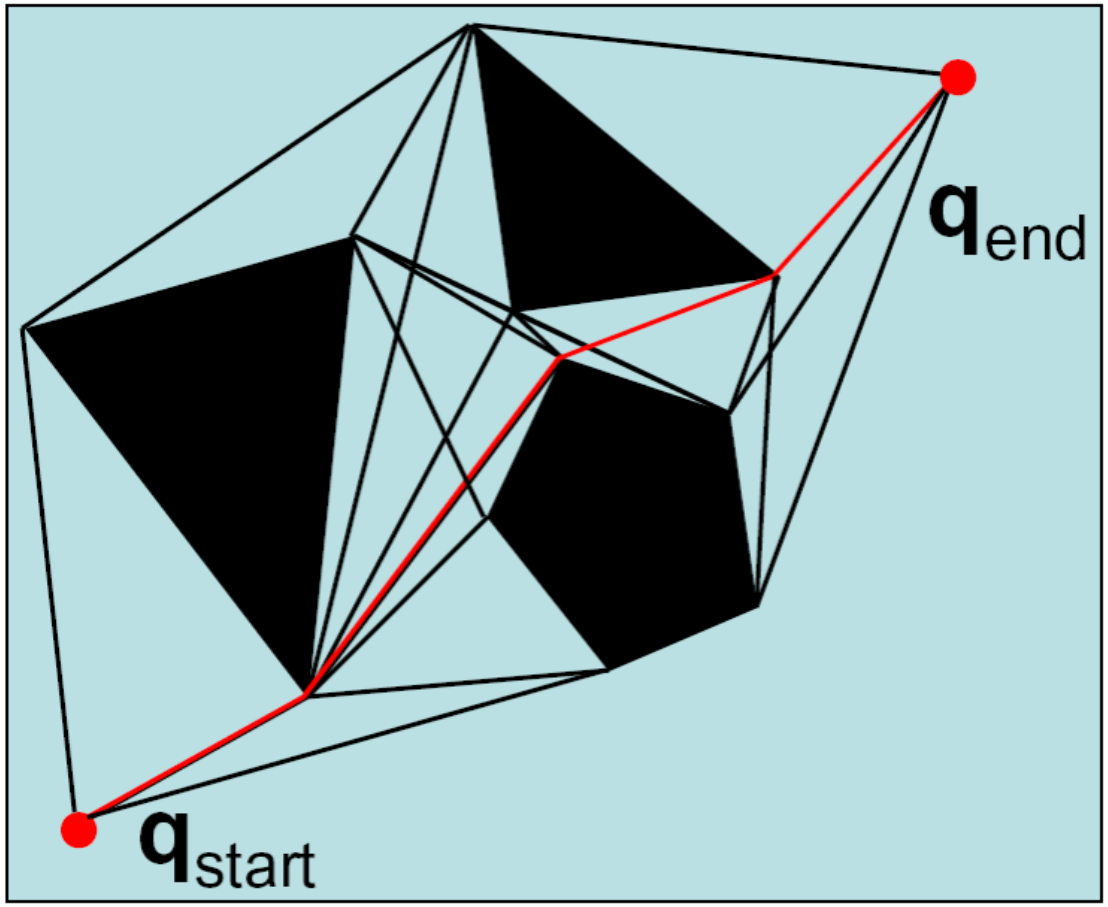


Cell Decomposition: Strengths/Weaknesses



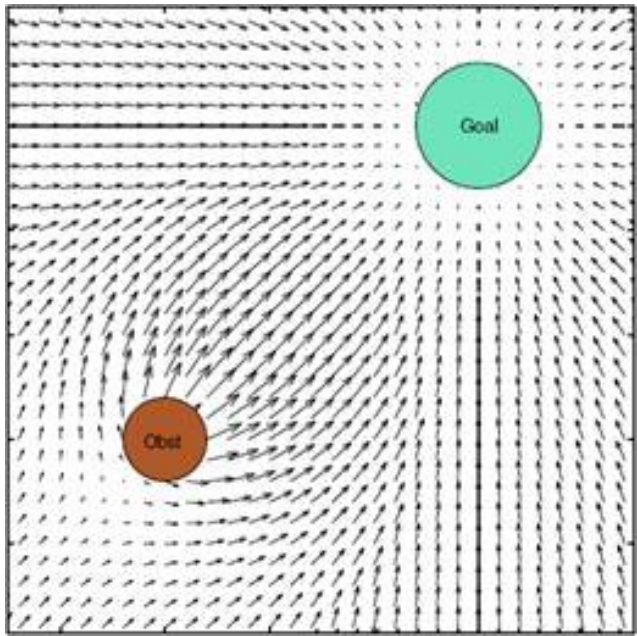
Some Well-Known Representations

- Visibility Graphs
- Roadmap
- Cell Decomposition
- Potential Field

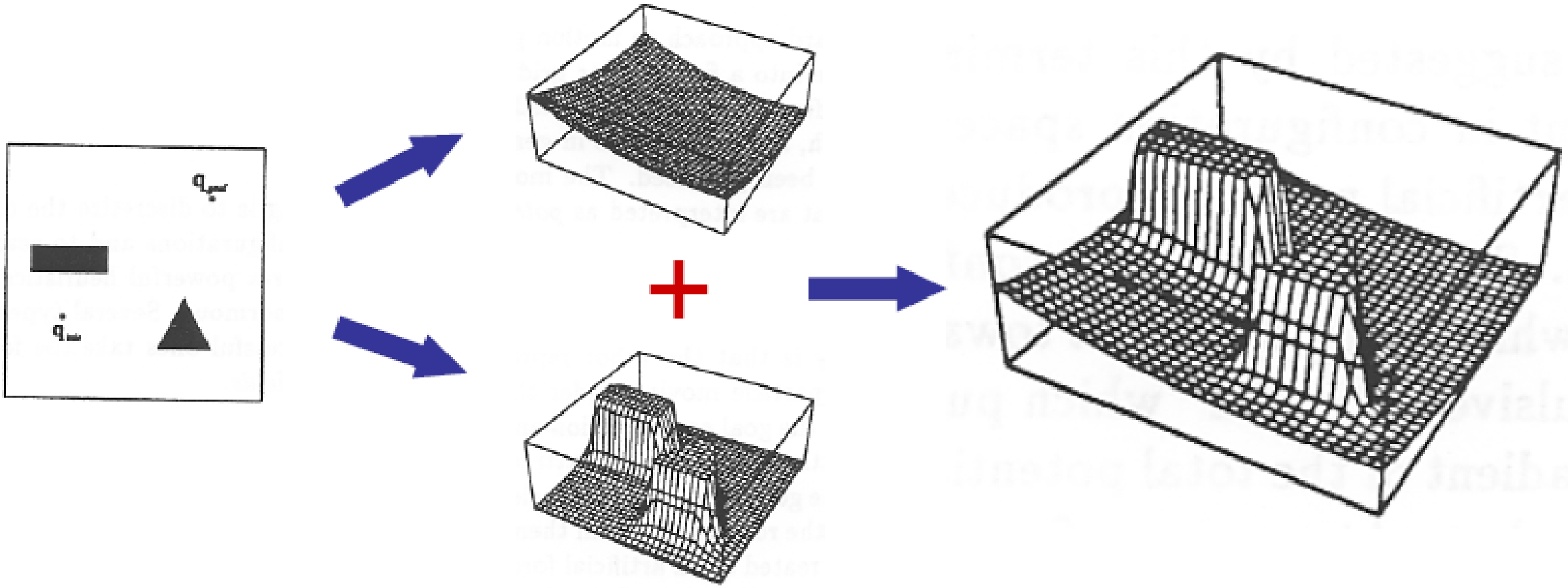


Potential Fields

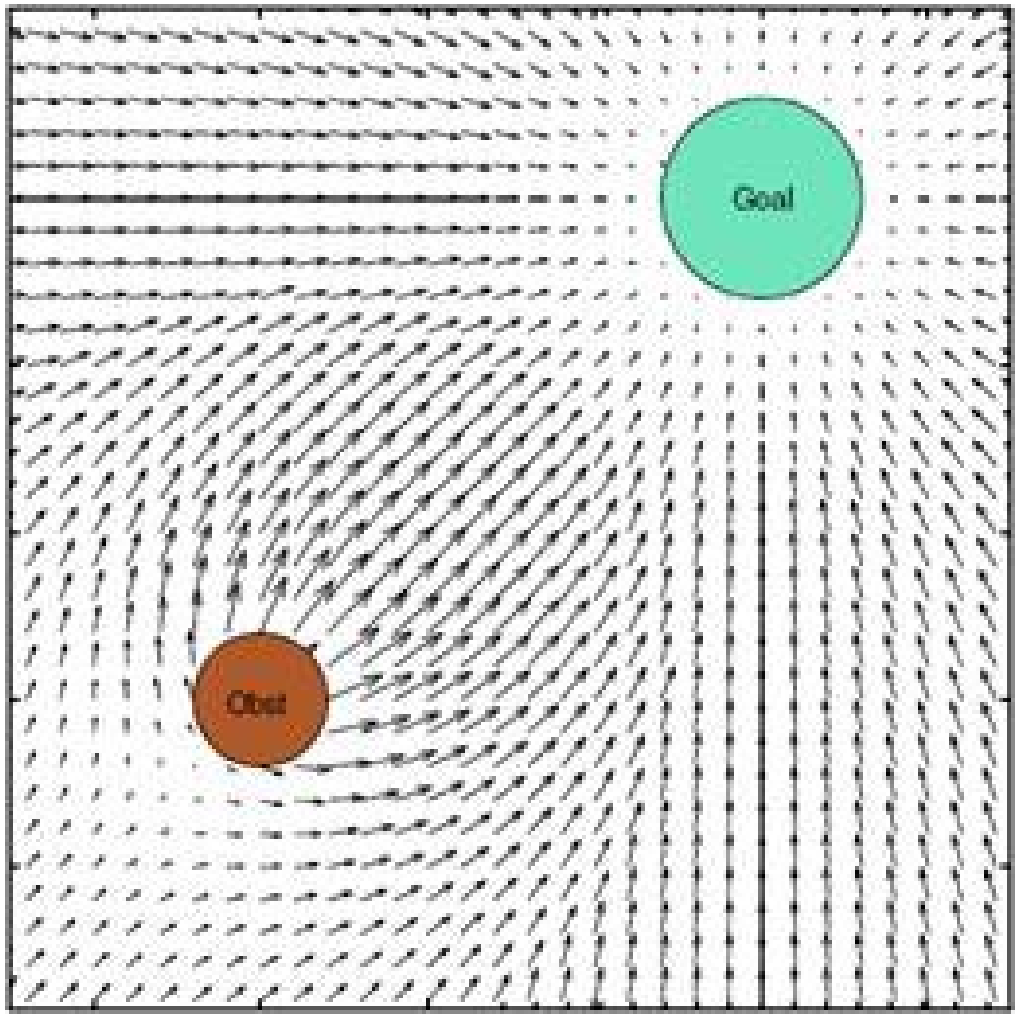
- Initially proposed for real-time collision avoidance [Khatib 1986].
- A potential field is a scalar function over the free space.
- To navigate, the robot applies a force proportional to the negated gradient of the potential field.
- A navigation function is an ideal potential field that
  - has global minimum at the goal
  - has no local minima
  - grows to infinity near obstacles
  - is smooth



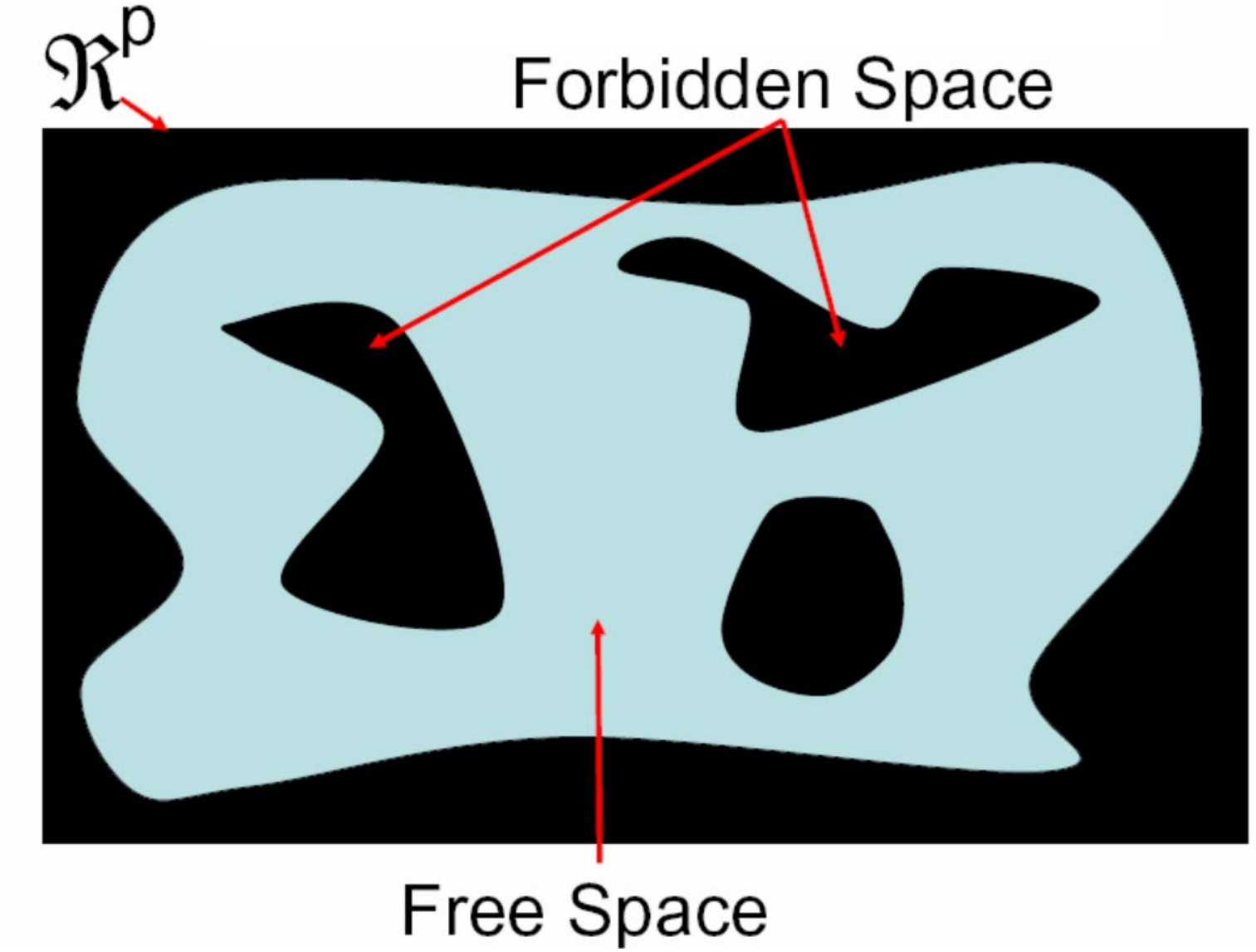
How Does It Work?



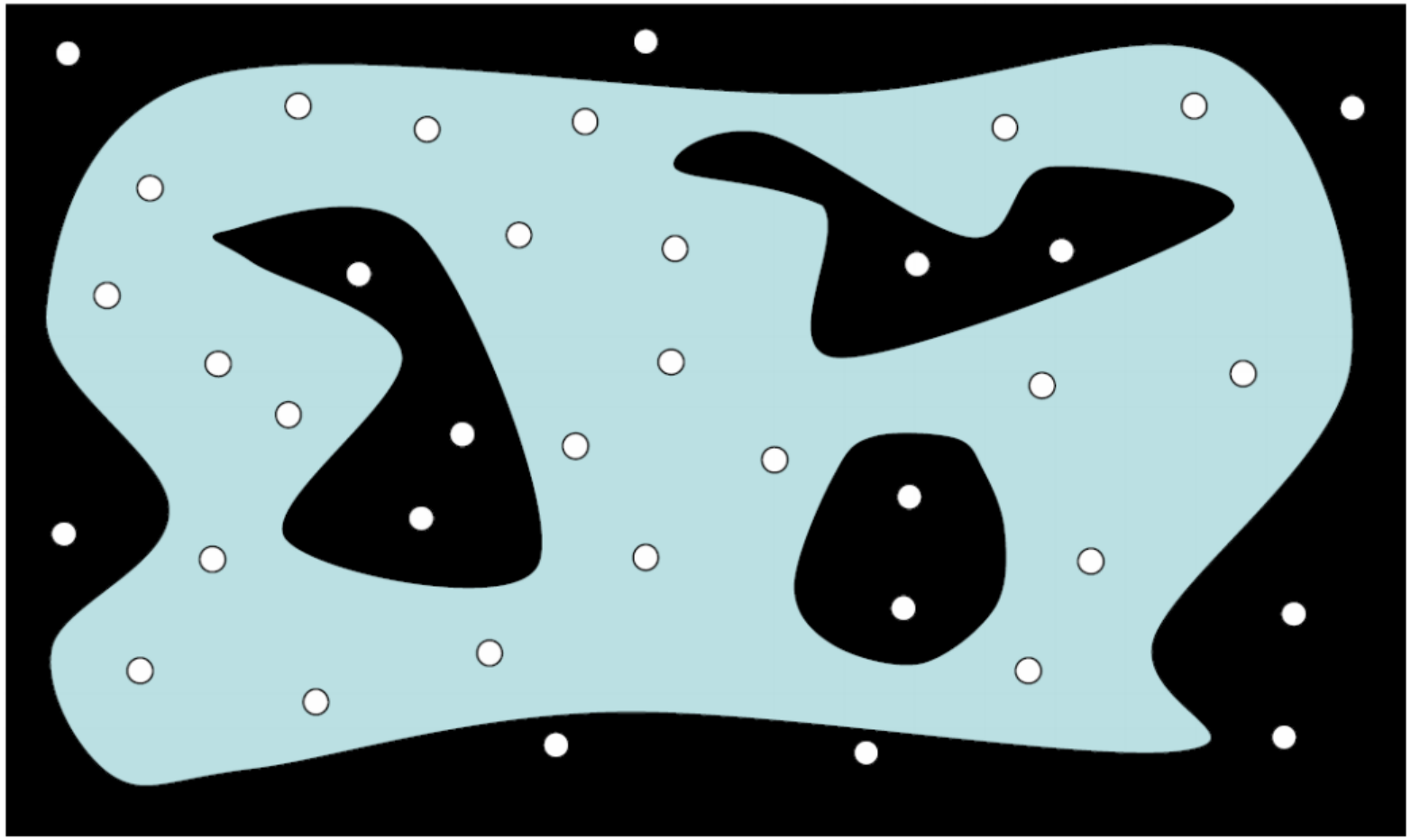
Potential Fields: Strengths/Weaknesses



Probabilistic Road Map (PRM)

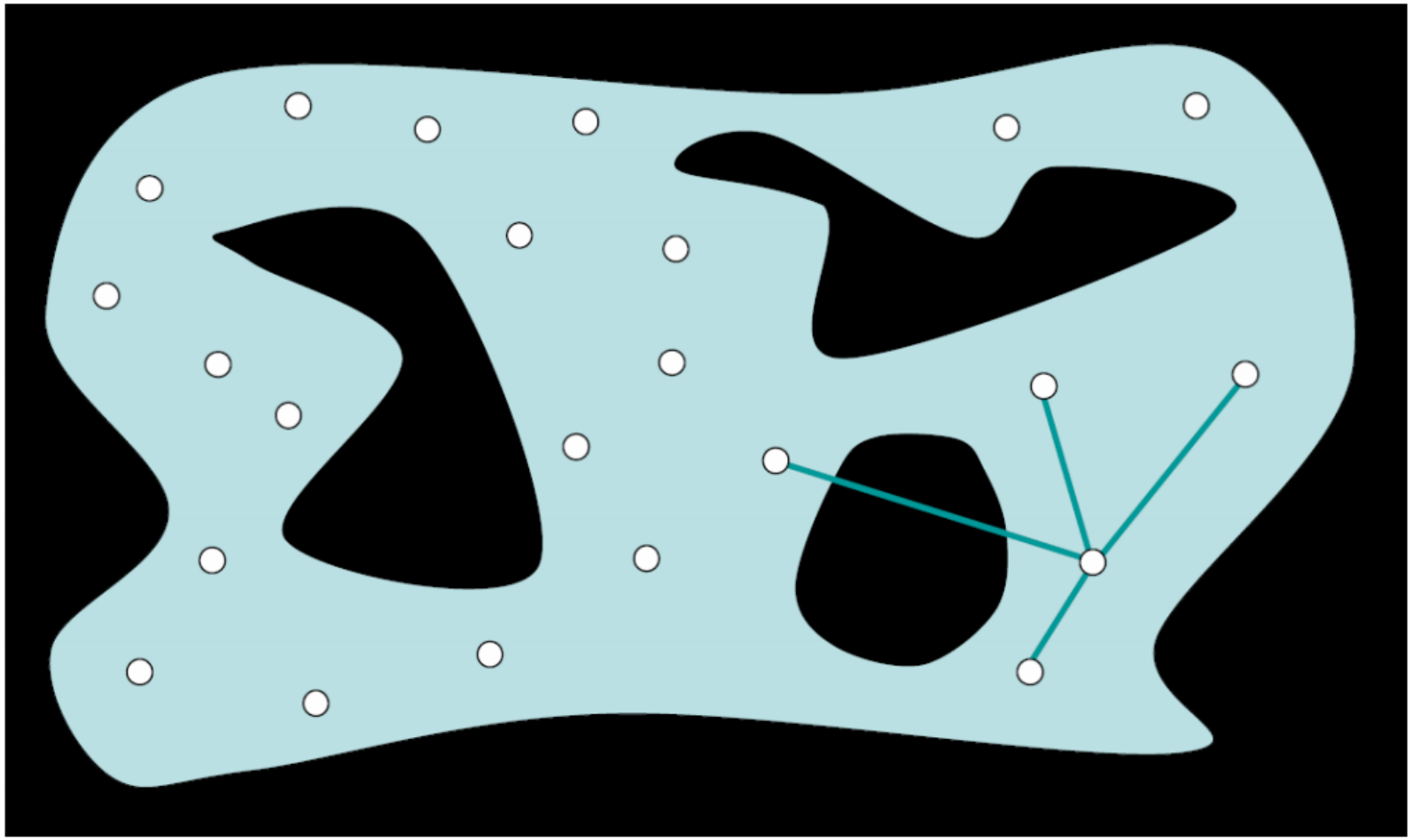


Probabilistic Road Map (PRM)



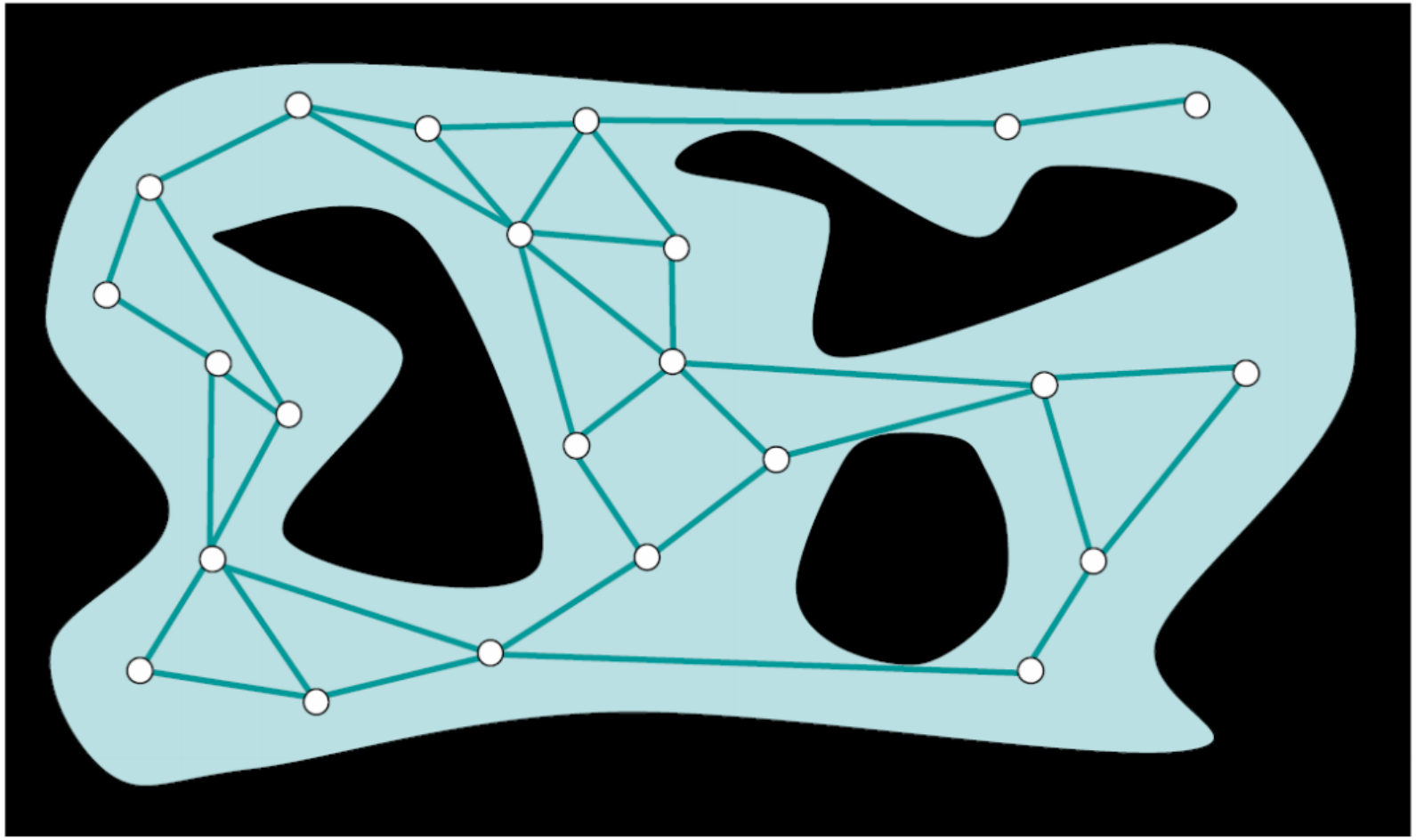
Sample random locations!

Probabilistic Road Map (PRM)



Remove points in forbidden areas
Link each point to its K nearest neighbors

Probabilistic Road Map (PRM)

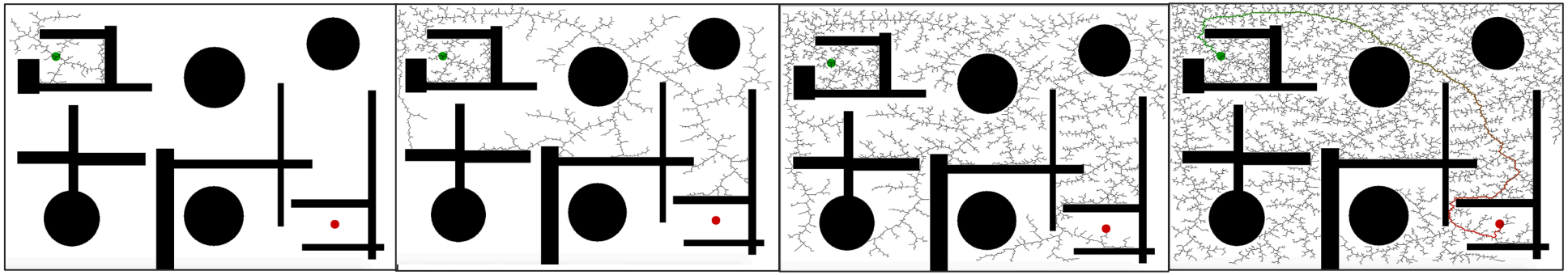


Remove edges crossing forbidden areas

How to sample points?

- Uniformly vs randomly
- Sample more near places with few neighbors
- Bias samples to exist near obstacles
- Use human-provided waypoints
- Something better?

Rapidly-exploring Random Trees (RRT)



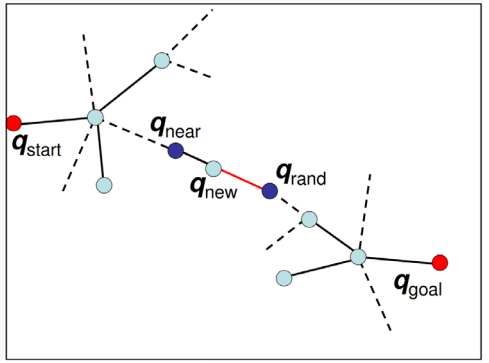
Rapidly Exploring Random Trees [RRT]

Basic idea:

- Build up a tree through generating “next states” in the tree by executing random controls
- However: to ensure good coverage, we need something better than the above!!

Rapidly-exploring Random Trees

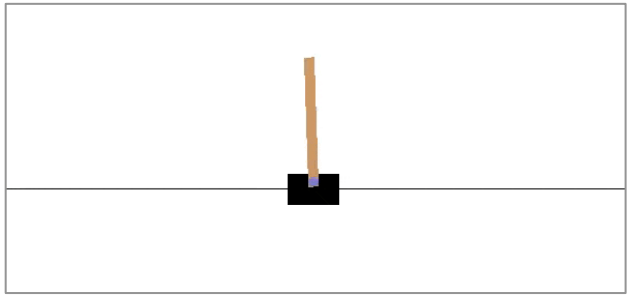
```
Algorithm BuildRRT
  Input: Initial configuration q_init, number of vertices in RRT K, incremental distance Δq)
  Output: RRT graph G

  G.init(q_init)
  for k = 1 to K
    q_rand ← RAND_CONF()
    q_near ← NEAREST_VERTEX(q_rand, G)
    q_new ← NEW_CONF(q_near, q_rand, Δq)
    G.add_vertex(q_new)
    G.add_edge(q_near, q_new)
  return G
```

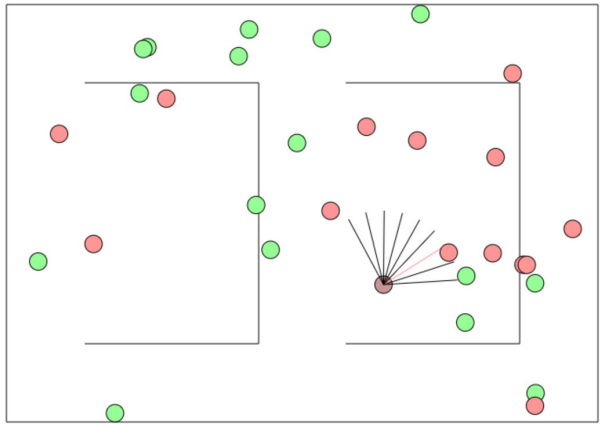


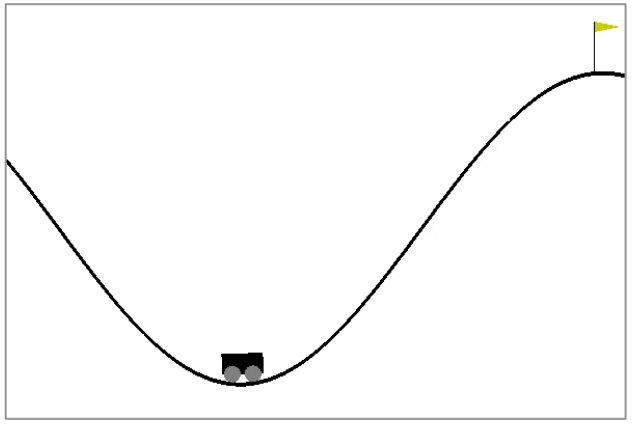
How can we make use of these representations?

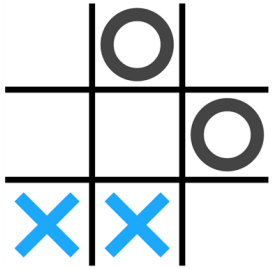
Search algorithms provide a way to find a path!

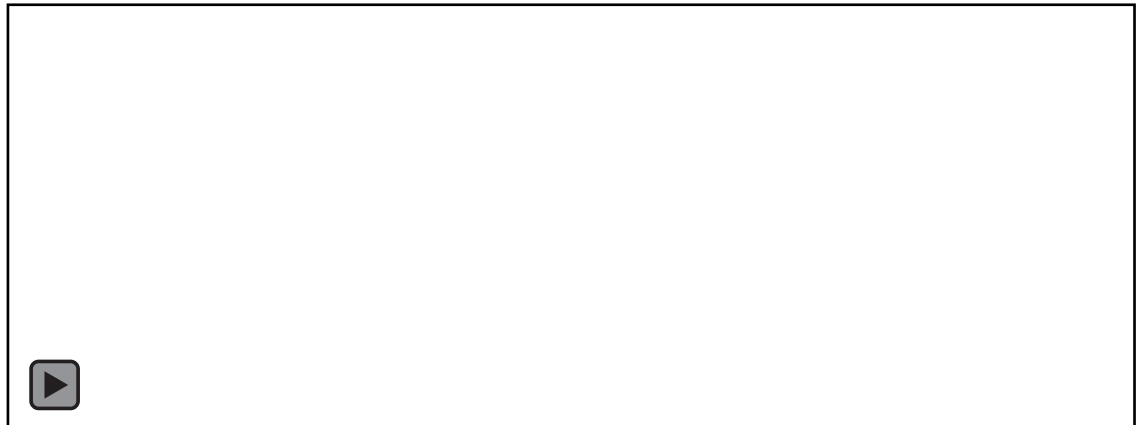
State Representation is Critical





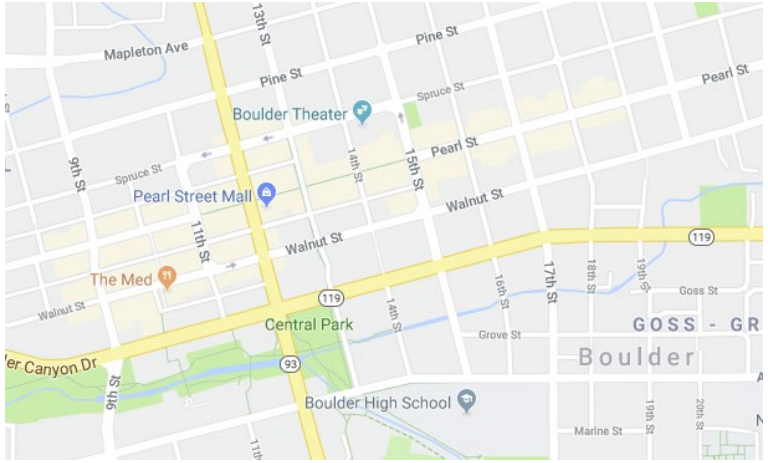


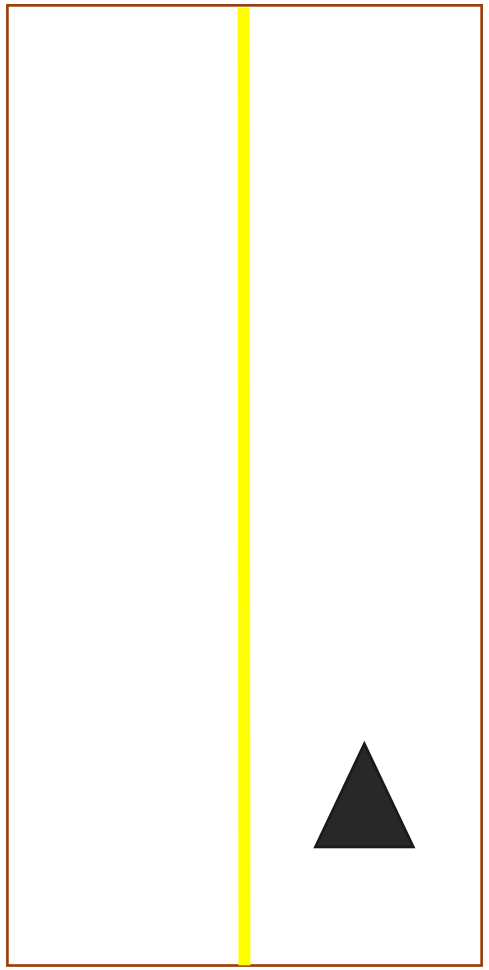




Exercise: Building an Autonomous Car

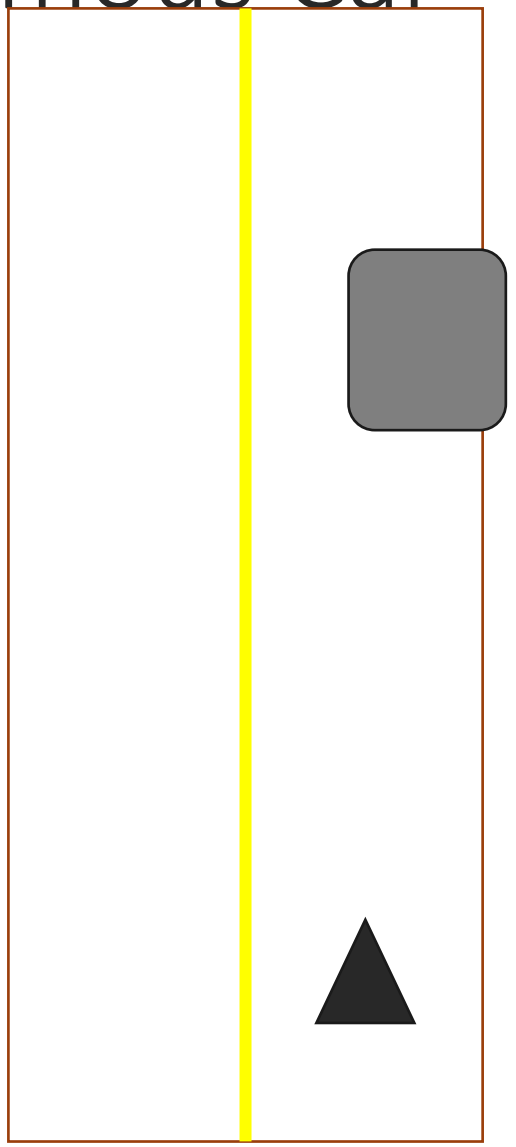
1. World Representation
2. Planning System



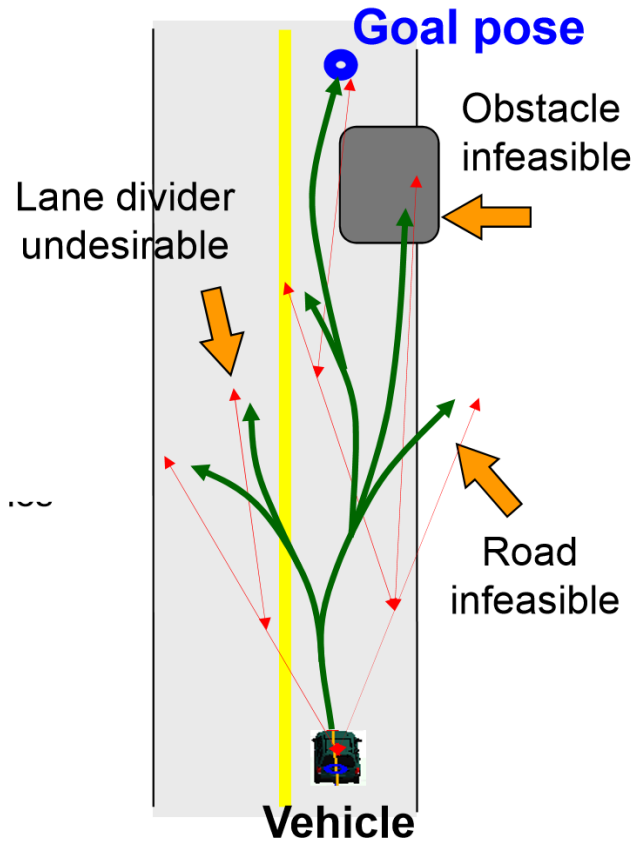


Exercise: Building an Autonomous Car

How can we get our agent to accommodate obstacles?



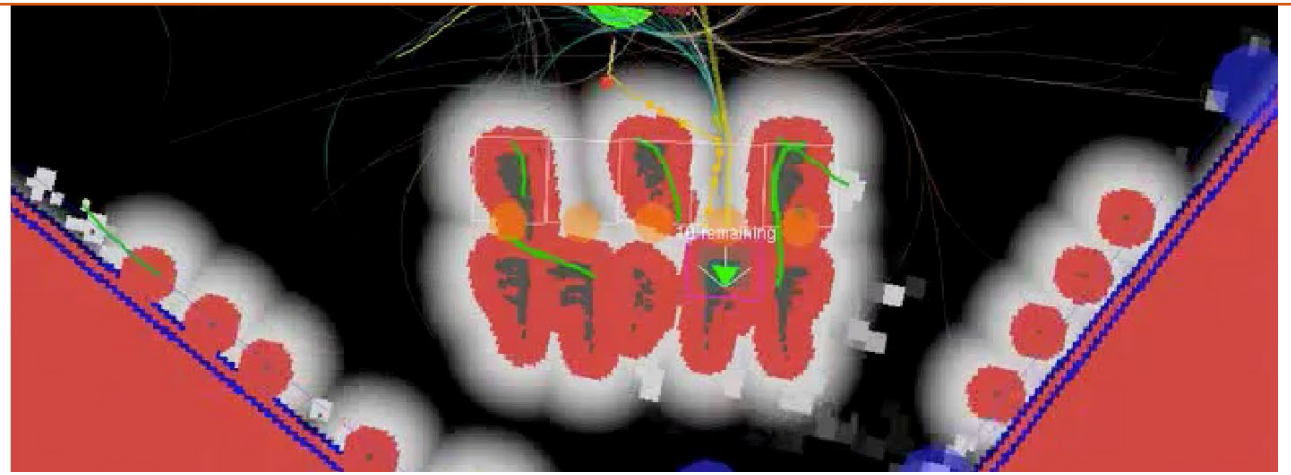
Exercise: Building an Autonomous Car



RRT for Autonomous Parking

How can we make use of these representations?

Search algorithms provide a way to find a path!



Chapter 13

Module 2 – COMPUTATION

Part III – Search Algorithms

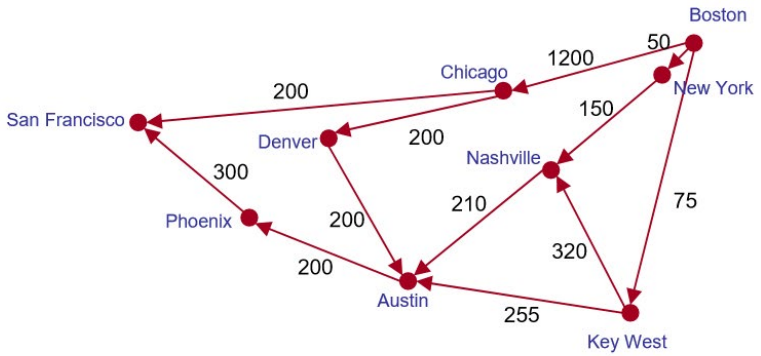
Search Algorithms

Allow us to use the world representations:

- Basics
- Uniform Cost
- Informed Search

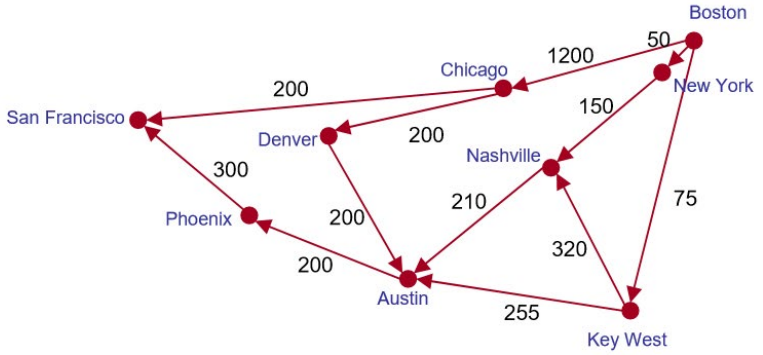
Goal Formulation

- Well defined function that identifies both the goal states and the conditions under which to achieve it
  - E.g. “Fly from Boston to San Francisco”
- Metrics for optimality may depend on:
  - Least amount of money
  - Fewest number of transfers
  - Shortest amount of time in the air
  - Shortest amount of time in airports

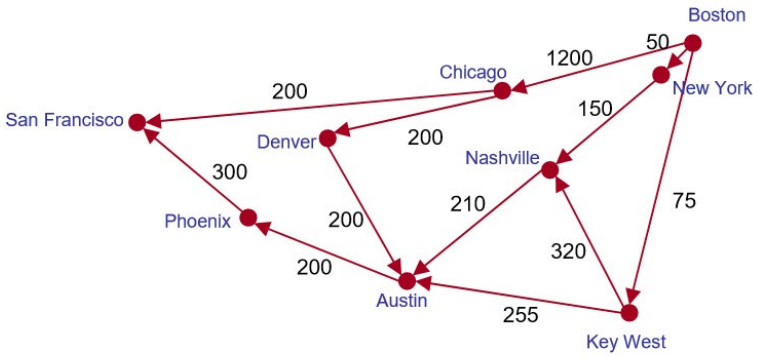


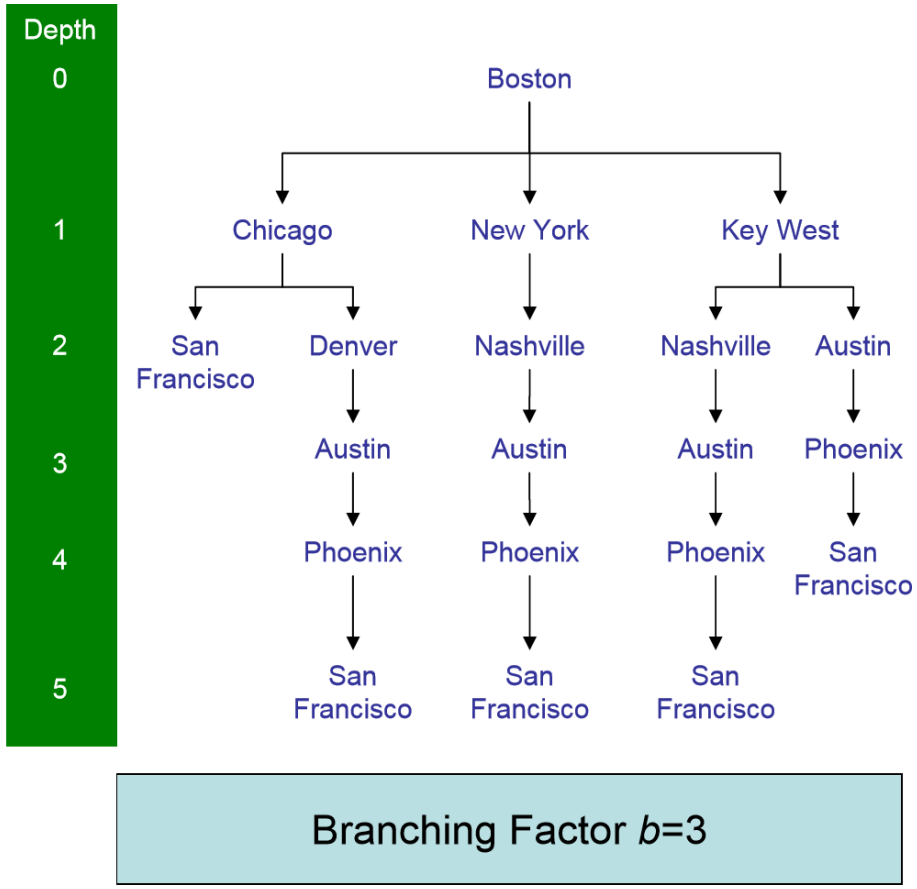
Problem formulation

- Well defined problems
  - Fully observable
  - Deterministic
  - Discrete Set of possible actions
- State Space: the set of all states that are reachable from an initial state by any sequence of actions
- Path: sequence of actions leading from one state to another



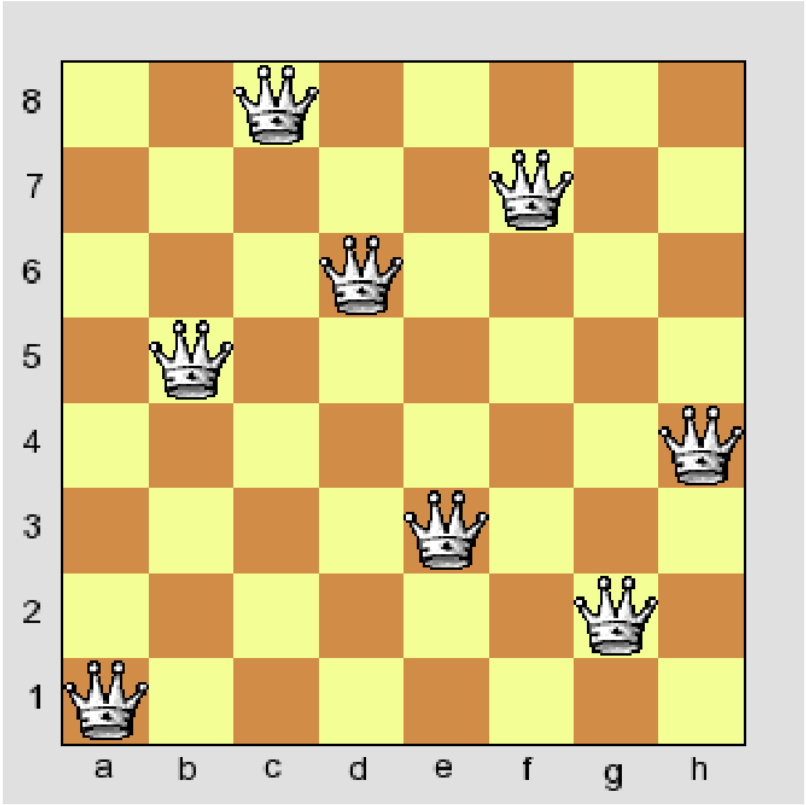
Search as a problem solving technique!!





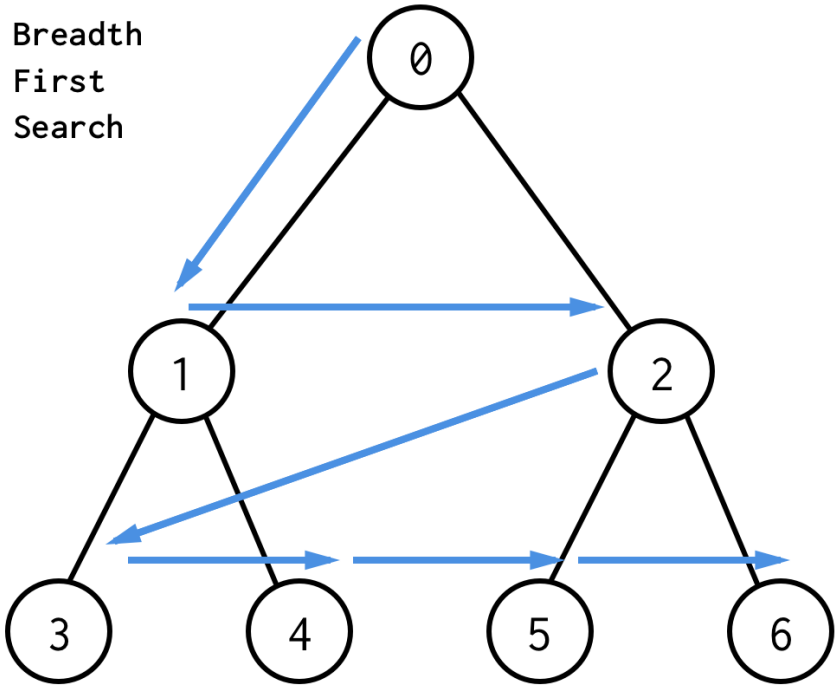
Problem Formulation Matters!
The 8 Queens Problem

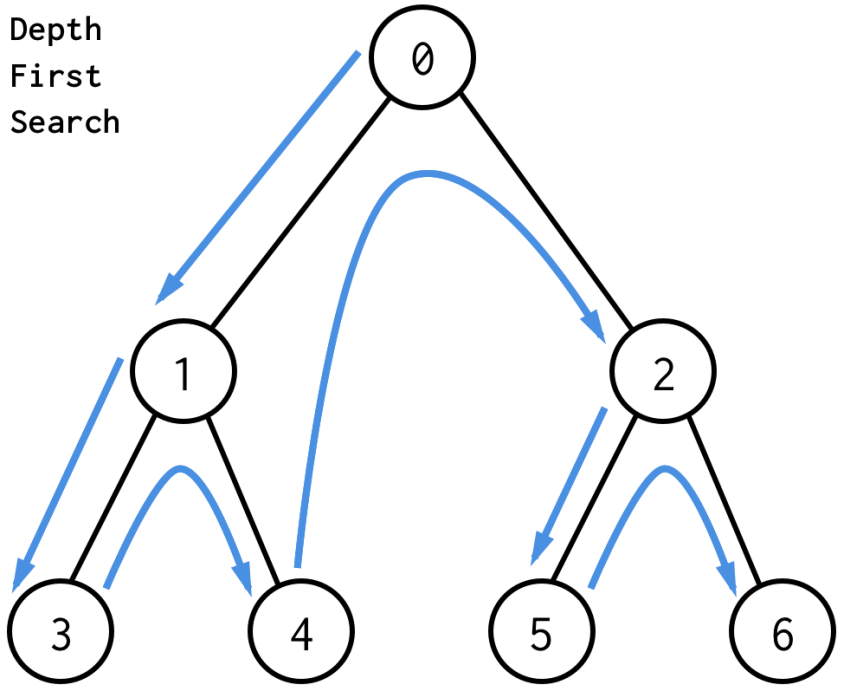
- Formulation #1:
  - Place a queen on any open square
  - Repeat until all queens are placed
  - State space of $\frac{64!}{56!} = 1.78 * 10^{14}$
- Formulation #2:
  - Place a queen on any row 1 square
  - Place a queen on any row 2 square
  - State space of $8^8 = 1.68 * 10^7$
- Formulation #3:
  - Place a queen on any row 1 square
  - Place a queen on any row 2 square not sharing a column…
  - State space of $8! = 40{,}320$



Brute force becomes intractable for problems of n ≥ 20, as 20! = 2.433×10^{18}

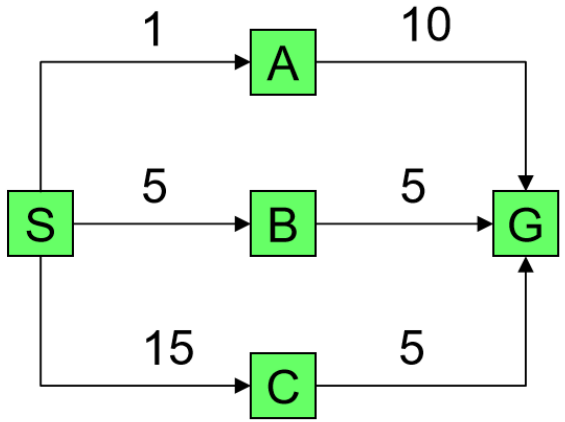
Breadth First vs Depth First search

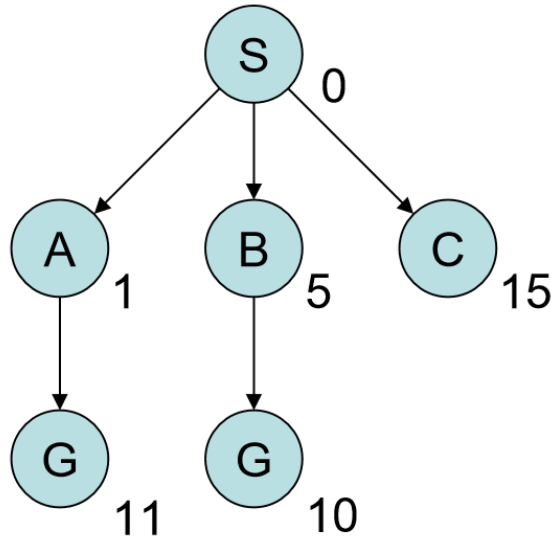




- Finds the most **shallow** solution
- Finds the most **deep** solution

Uniform Cost Search

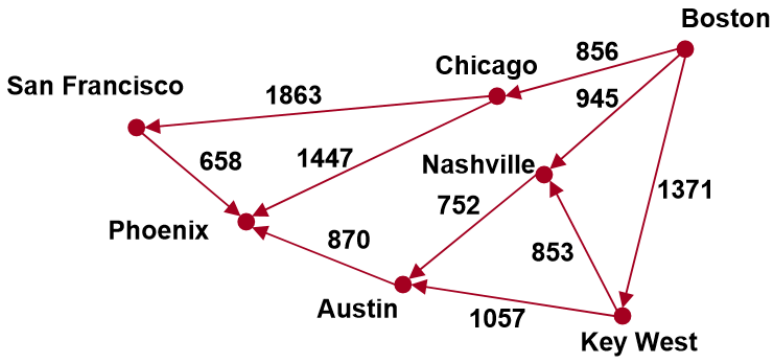




- Travel from the start (S) to the goal (G)
- Cost associated with each link

- Always expand the fringe node with the lowest cost
- Breadth-first search is uniform search with cost=depth

Greedy Best-First-Search



- Minimize estimated cost to reach a goal (in this case, the distance to Phoenix)

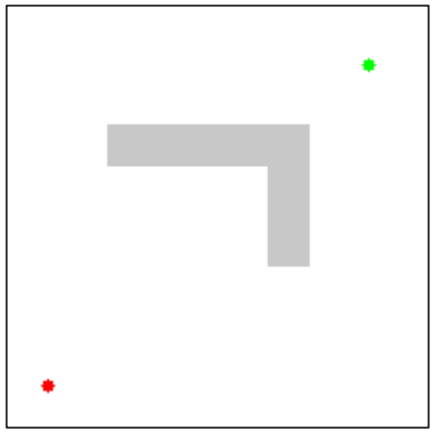
	Straight Line Distance to Phoenix
Boston	2299
Chicago	1447
Nashville	1444
Key West	1927
Austin	870
San Francisco	658

Search Algorithms: Uniform Cost

Dijkstra's Algorithm:

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:             // Initialization
6          dist[v] ← INFINITY                  // Unknown distance from source to v
7          prev[v] ← UNDEFINED                 // Previous node in optimal path from source
8          add v to Q                          // All nodes initially in Q (unvisited nodes)
9
10     dist[source] ← 0                        // Distance from source to source
11
12     while Q is not empty:
13         u ← vertex in Q with min dist[u]    // Node with the least distance
14                                             // will be selected first
15         remove u from Q
16
17         for each neighbor v of u:           // where v is still in Q.
18             alt ← dist[u] + length(u, v)
19             if alt < dist[v]:               // A shorter path to v has been found
20                 dist[v] ← alt
21                  prev[v] ← u
22
23     return dist[], prev[]
```

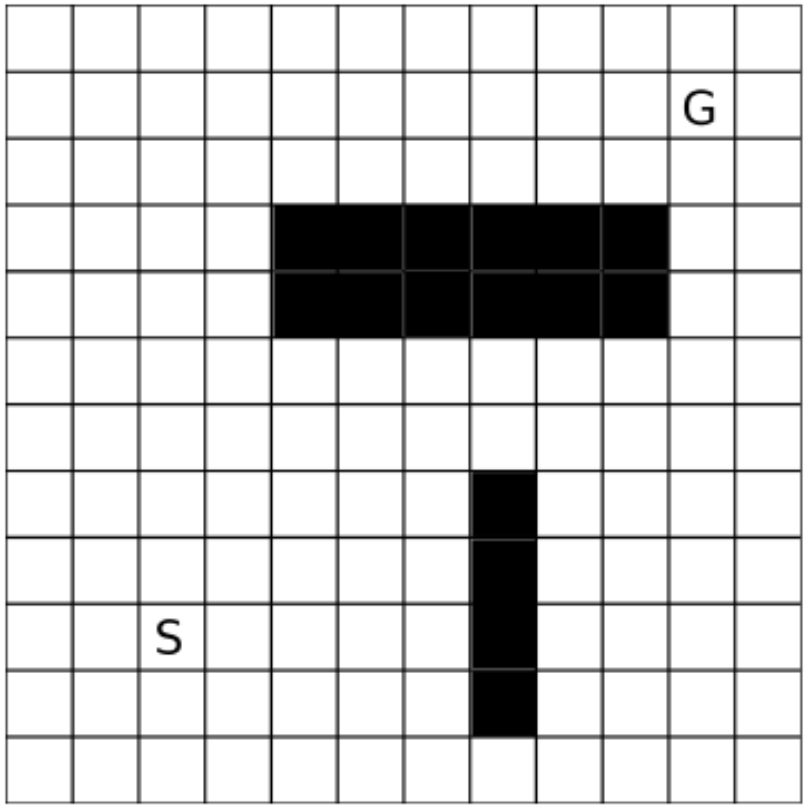
Finds shortest paths between nodes in a graph



```
1  S ← empty sequence
2  u ← target
3  if prev[u] is defined or u = source:          // Do something only if the vertex is reachable
4      while u is defined:                       // Construct the shortest path with a stack S
5          insert u at the beginning of S        // Push the vertex onto the stack
6          u ← prev[u]                           // Traverse from target to source
```

Dijkstra's Algorithm on a Grid

Problem: A lot of useless exploration

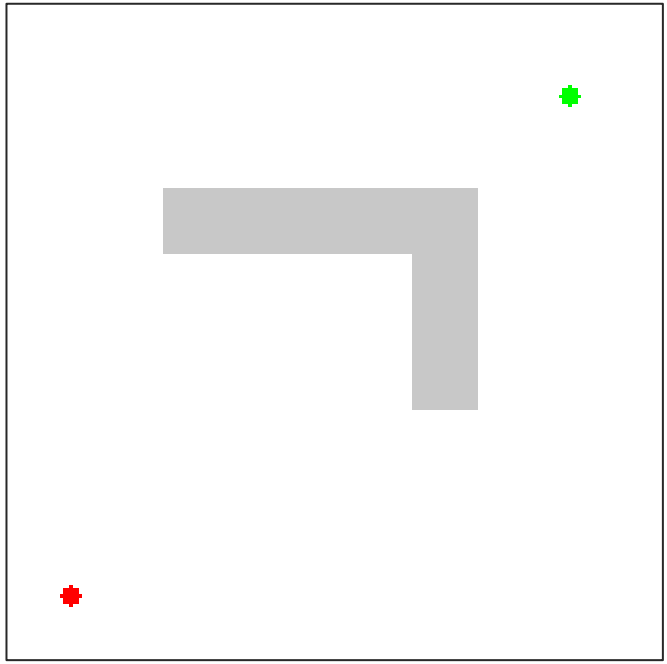


11	10	9	10	11	12						
10	9	8	9	10	11	12				G	
9	8	7	8	9	10	11	12				
8	7	6	7								
7	6	5	6								
6	5	4	5	6	7	8	9	10	11	12	
5	4	3	4	5	6	7	8	9	10	11	12
4	3	2	3	4	5	6		10	11	12	
3	2	1	2	3	4	5		11	12		
2	1	S	1	2	3	4		10	11	12	
3	2	1	2	3	4	5		9	10	11	12
4	3	2	3	4	5	6	7	8	9	10	11

11	10	9	10	11	12	13	14	15	16		
10	9	8	9	10	11	12	13	14	15	G	
9	8	7	8	9	10	11	12	13	14	15	
8	7	6	7							14	15
7	6	5	6							13	14
6	5	4	5	6	7	8	9	10	11	12	13
5	4	3	4	5	6	7	8	9	10	11	12
4	3	2	3	4	5	6		10	11	12	13
3	2	1	2	3	4	5		11	12	13	14
2	1	S	1	2	3	4		10	11	12	13
3	2	1	2	3	4	5		9	10	11	12
4	3	2	3	4	5	6	7	8	9	10	11

Informed Search

Finds shortest path between two nodes in a graph

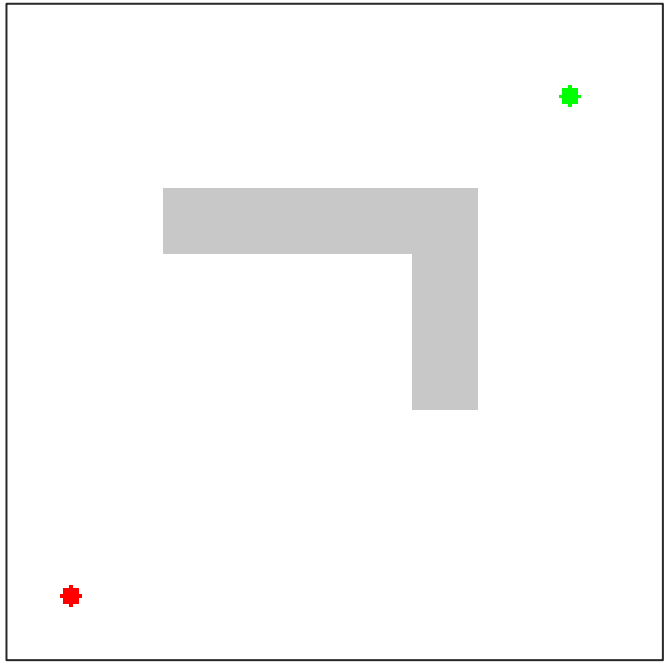


- Dijkstra's Algorithm:
- Lots of wasted exploration!
- We usually know which two nodes we want to find a path between:
  - Why compute all distances from source?

- Heuristic: A function that ranks choices in a search algorithm to help choose how to branch/explore.

A* Search Algorithm

Finds shortest path between two nodes in a graph



- Adds heuristic to Dijkstra's Algorithm

  to bias exploration!

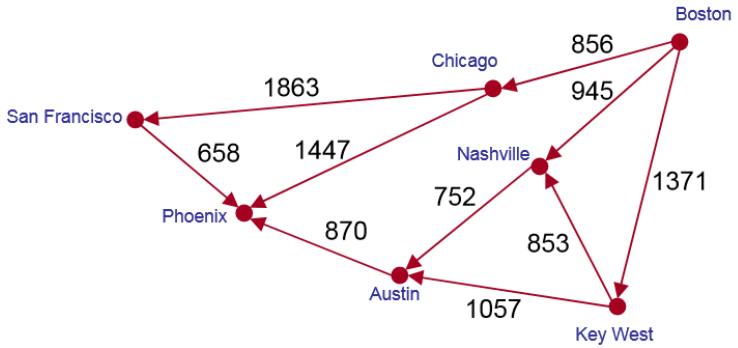
- Insight:

$$f(n) = g(n) + h(n)$$

- $f(n)$ = Cost so far + Est. cost to go

Admissible Heuristic: An **underestimate** of the shortest possible path from node n to the goal.

A* Search Algorithm



- Combine Greedy search with Uniform Cost Search
- Minimize the total path cost (f) =
  actual path so far (g) +
  estimate of future path to goal (h)

	Distance to Phoenix
Boston	2299
Chicago	1447
Nashville	1444
Key West	1927
Austin	870
San Francisco	658

A* Search Algorithm

- Complete: Yes

  (If an answer exists, this will find it)

- Optimal: Yes

  (Returned answer will be the best possible)

- Optimally Efficient: Yes

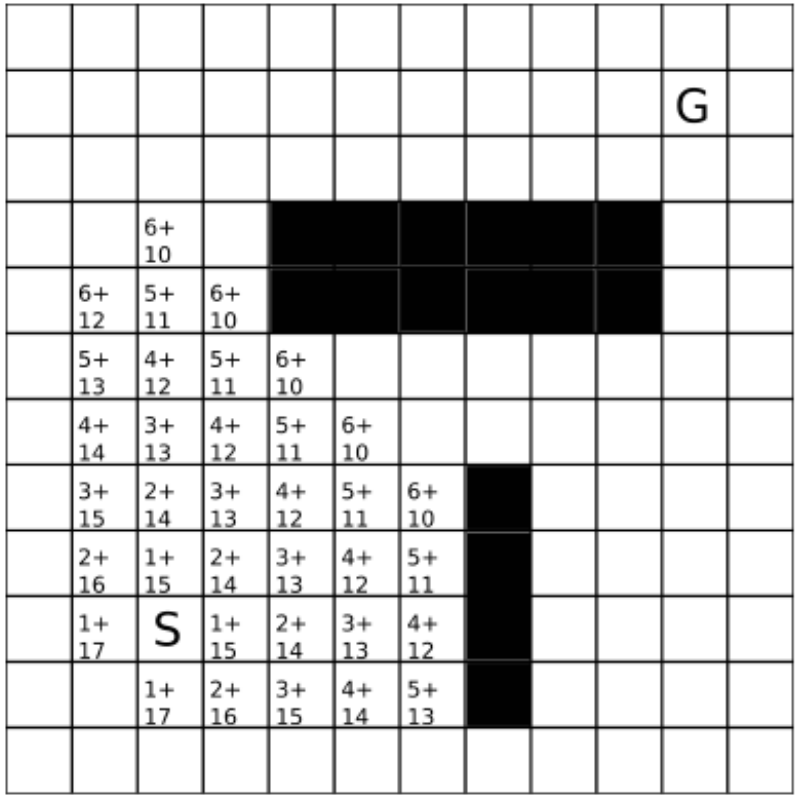
  (No algorithm with same heuristic will expand fewer nodes)

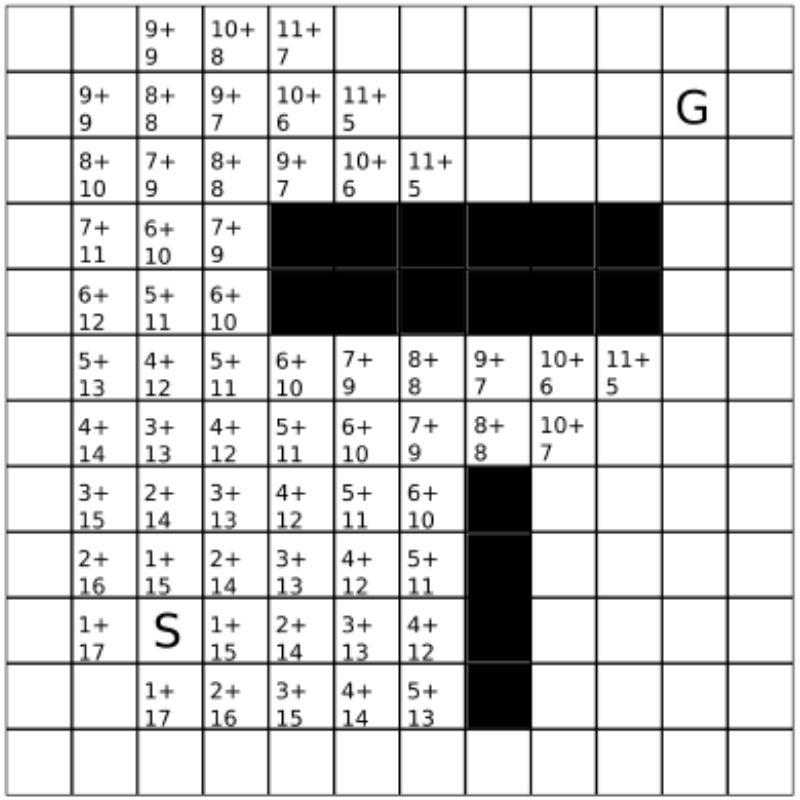
- Finding good admissible heuristics is challenging!

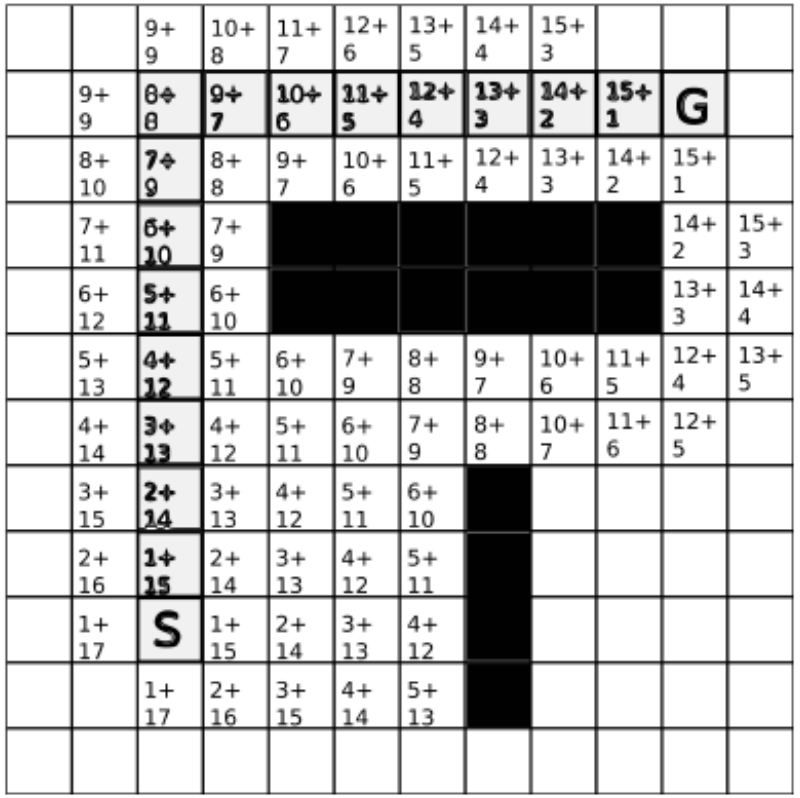
```
function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```

```
function A*(start, goal)
    // The set of nodes already evaluated
    closedSet := {}

    // The set of currently discovered nodes that are not evaluated yet.
    // Initially, only the start node is known.
    openSet := {start}

    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := the empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity

    // The cost of going from start to start is zero.
    gScore[start] := 0

    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity

    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)

        for each neighbor of current
            if neighbor in closedSet
                continue        // Ignore the neighbor which is already evaluated.

            if neighbor not in openSet  // Discover a new node
                openSet.Add(neighbor)

            // The distance from start to a neighbor
            tentative_gScore := gScore[current] + dist_between(current, neighbor)
            if tentative_gScore >= gScore[neighbor]
                continue        // This is not a better path.

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor, goal)

    return failure
```

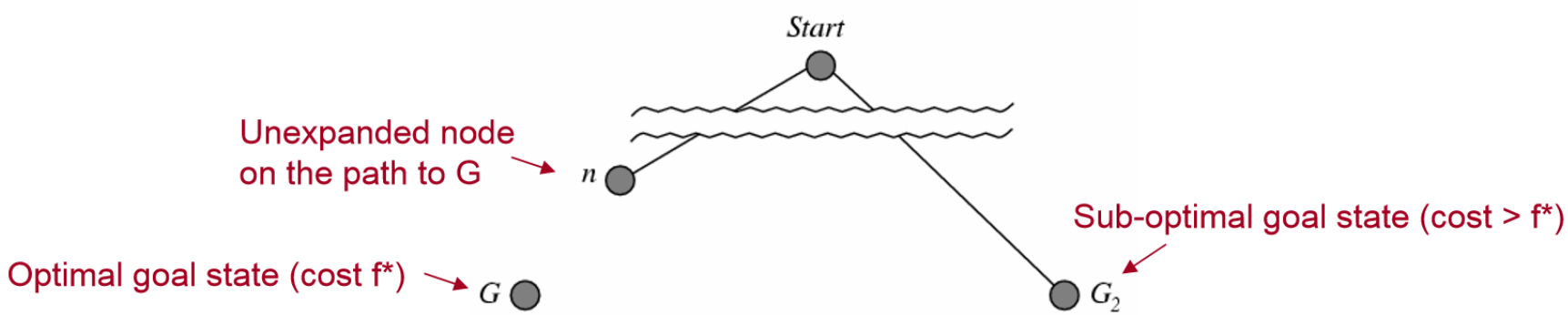
Dijkstra plus directional heuristic: A*







Optimality of A*



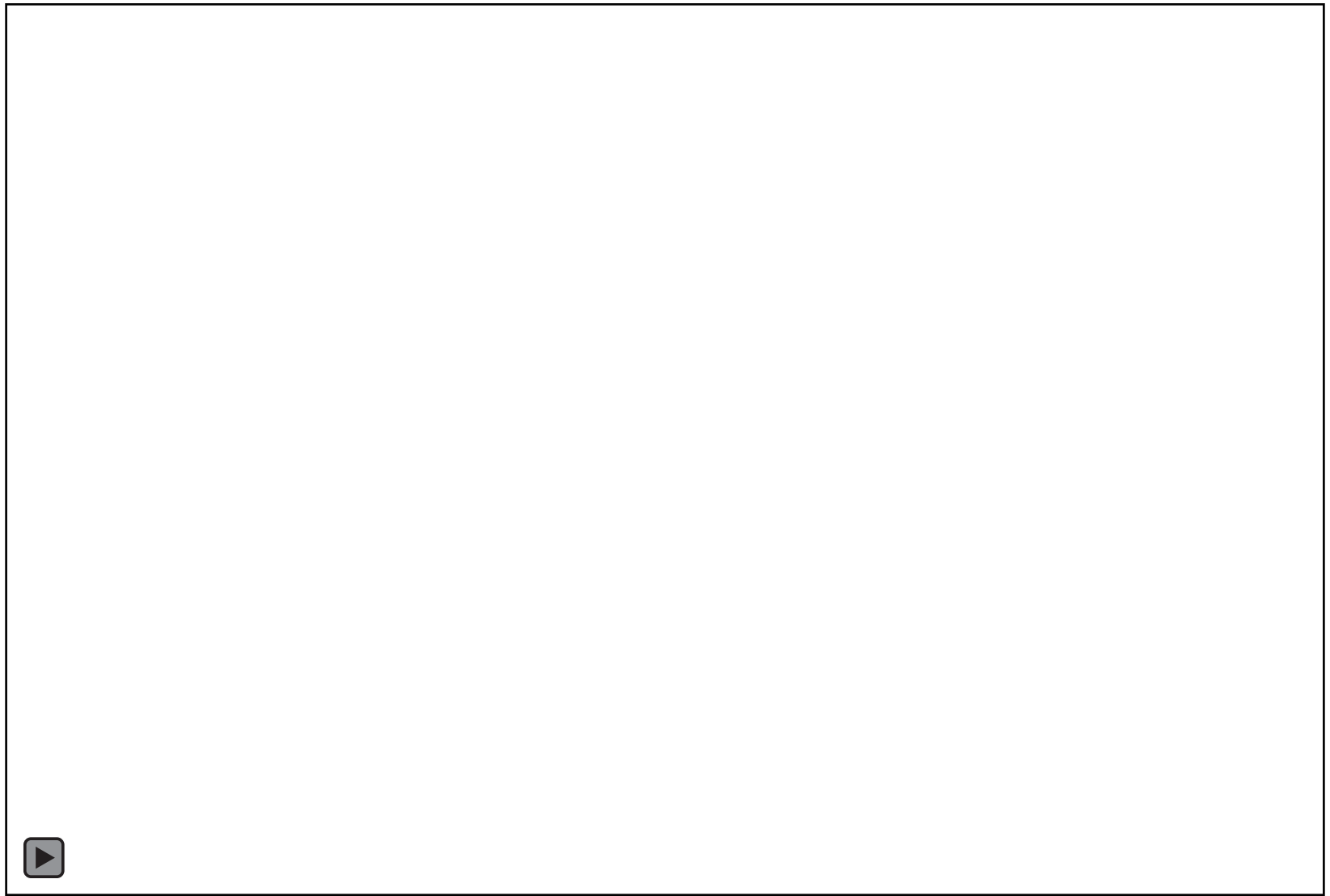
- Assume that G_2 has been chosen for expansion over n
- Because h is admissible

  $f^* \geq f(n)$
- If n is not chosen for expansion over G_2, we must have

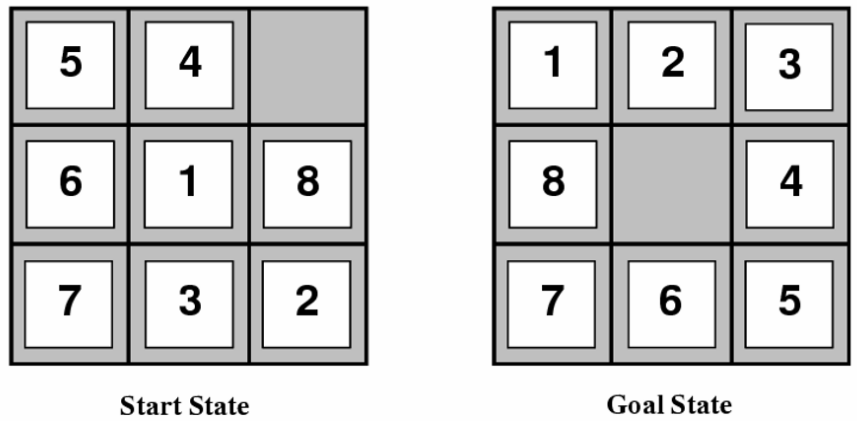
  $f(n) \geq f(G_2)$
- Combining these, we get

  $f^* \geq f(G_2)$
- However, this violates our assertion that G_2 is sub-optimal
- Therefore, A* never selects a sub-optimal goal for expansion

Many variants of A* exist… (D* [lite])



Heuristic Selection



- Must be admissible (never over-estimate)
- Heuristics for the 8-Puzzle

Heuristic Selection Effects

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- Compare iterative-deepening with A* using
  h1 (# misplaced tiles) and *h2* (city block distance)
- Effective branching factor b*
  - Number of expanded nodes = $1 + b^* + (b^*)^2 + \ldots + (b^*)^{depth}$
  - b* remains relatively constant across many measurements

How do humans solve the path/motion planning problem?

Take-home lessons

- First step in addressing a planning problem is choosing a suitable map representation
- Reduce robot to a point-mass by inflating obstacles
- Grid-based algorithms are complete, sampling-based ones probabilistically complete, but usually faster
- Most real planning problems require combination of multiple algorithms