Motion Planning

Jana Kosecka
Department of Computer Science

- Discrete planning, graph search, shortest path,
- A* methods
- Road map methods
- Configuration space

Slides thanks to http://cs.cmu.edu/~motionplanning, Jyh-Ming Lien

Discrete Planning

- Review of some discrete planning methods
- Given state space and transition function and initial state, find a set of states which lead to goal state
  (state space is discrete)
- Use well developed search and graph traversal algorithms to find the path
- Path: set of vertices in the graph

- For more details CS 583, CS 580

Previously

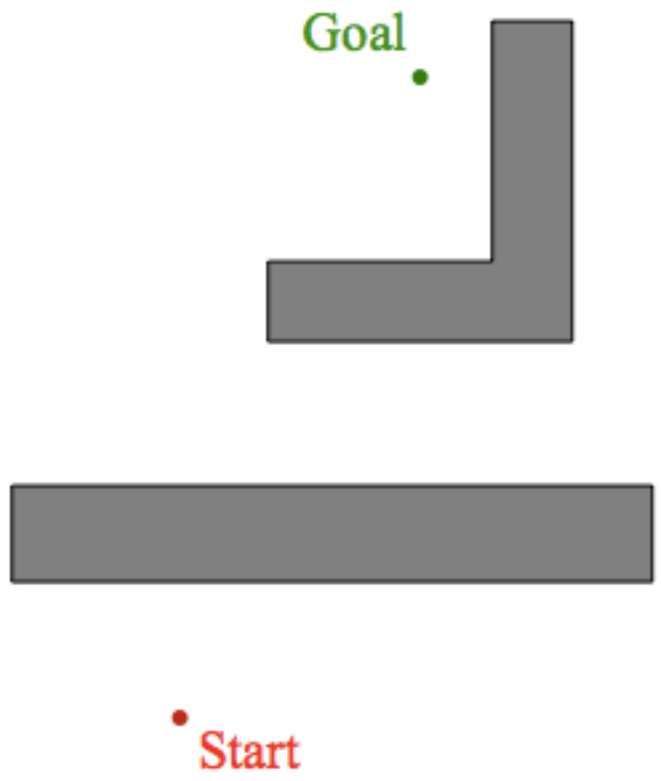
- Reactive Navigation Strategies
- Braitenberg vehicles no map of the world, no memory
- Extensions to more complex behaviors and behavior assemblages
- Steering behaviors

Today:

- Navigation with some memory
- Map based approaches
- More general motion planning methods

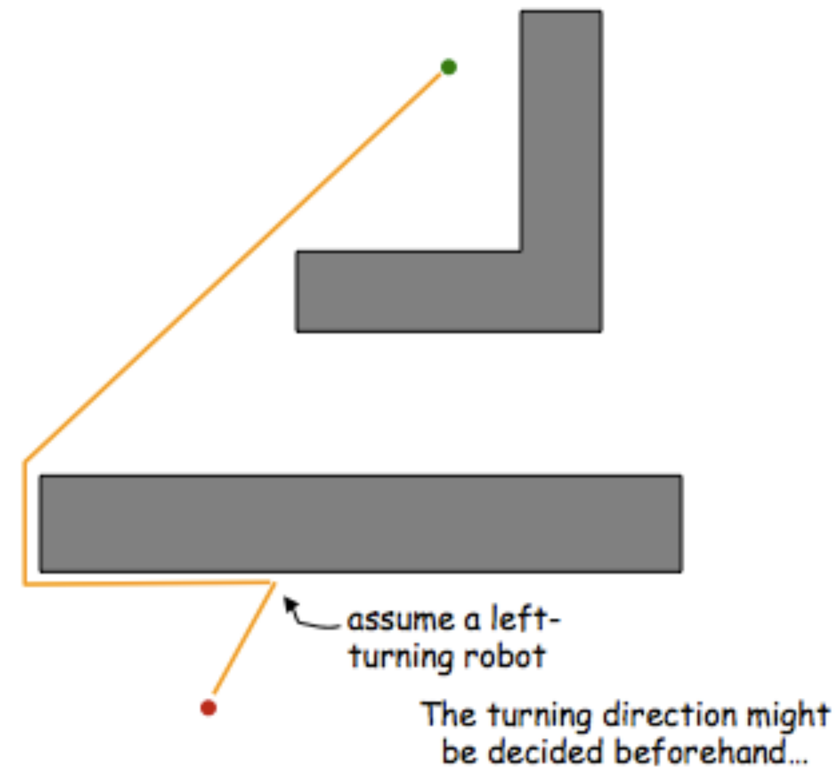
Bug Algorithms

- Extensions of reactive strategies
- Assumes local knowledge of the environment
- And global knowledge of the goal



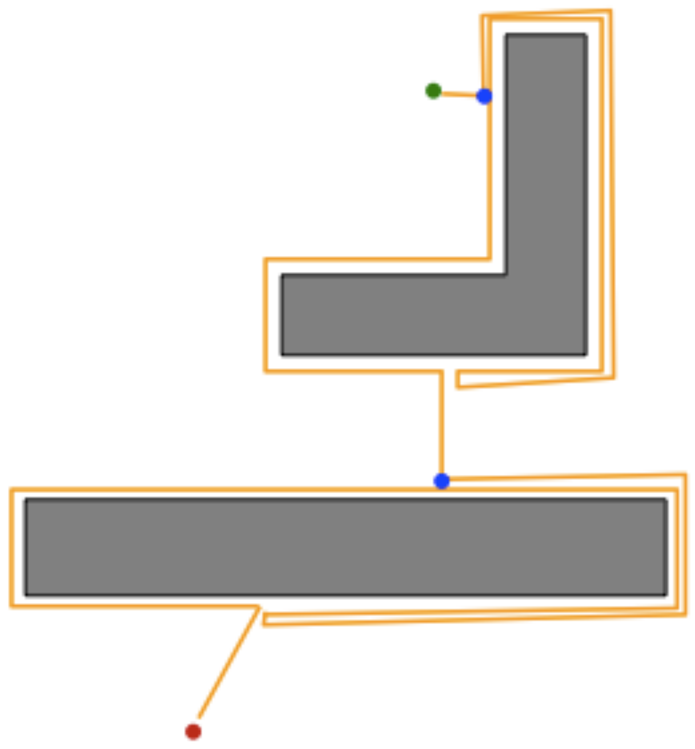
Bug Algorithms

- Algorithm
- Head towards the goal until you encounter obstacle
- Follow obstacle until you can turn to goal again
- Continue


assume a left-turning robot
The turning direction might be decided beforehand...

Bug 1

- Head towards the goal
- Circumnavigate obstacle and
- Remember how close you got to
- The goal
- Return to the closes point by
- Wall following
- Continue

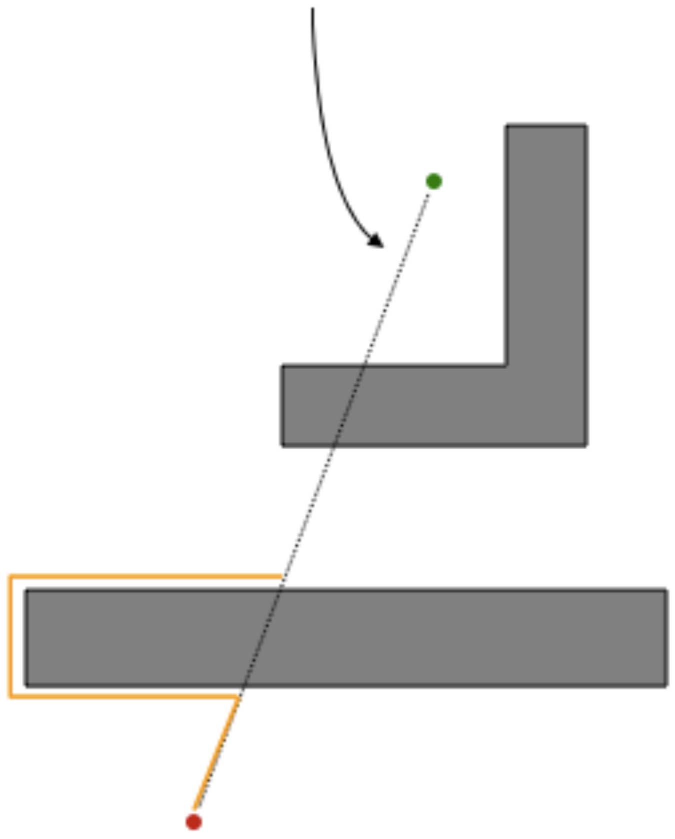


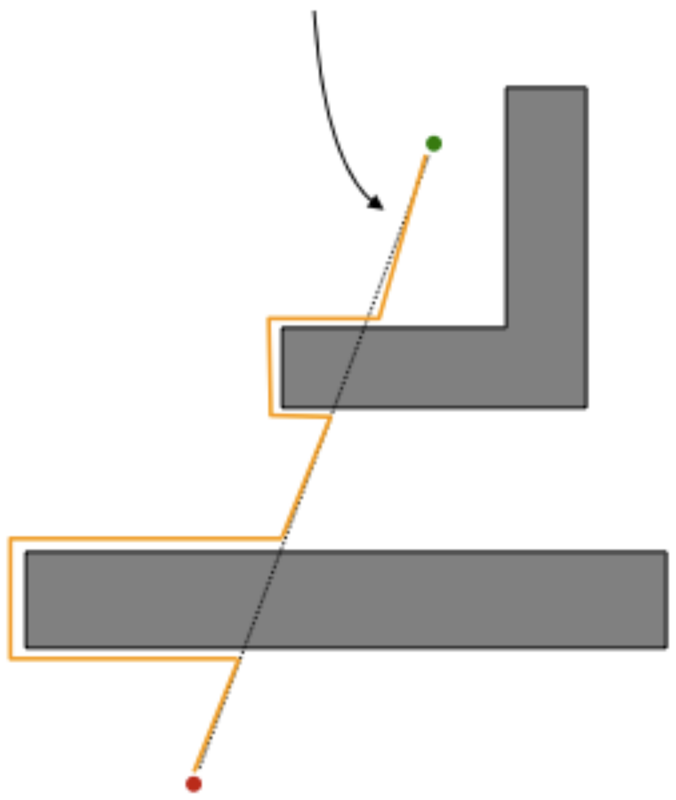
Completeness

- Algorithm is complete if in finite time will find the path to the goal or declare failure if such path does not exist
- Incomplete algorithm – never terminates, or does not find the goal

Bug 2

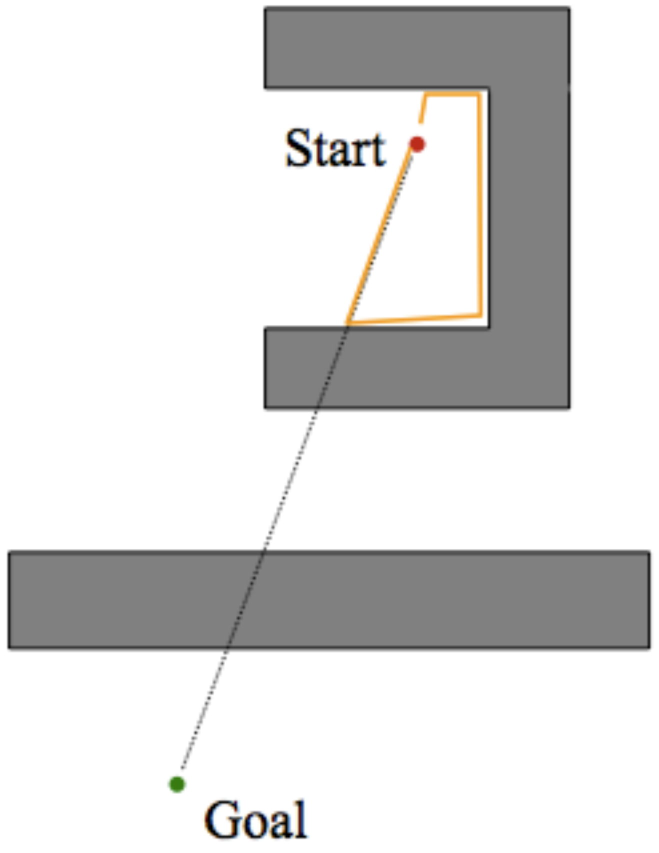
- Draw a line to the goal. Follow the line until obstacle, then circumnavigate to obstacle until you reach the goal
- heads towards the goal again

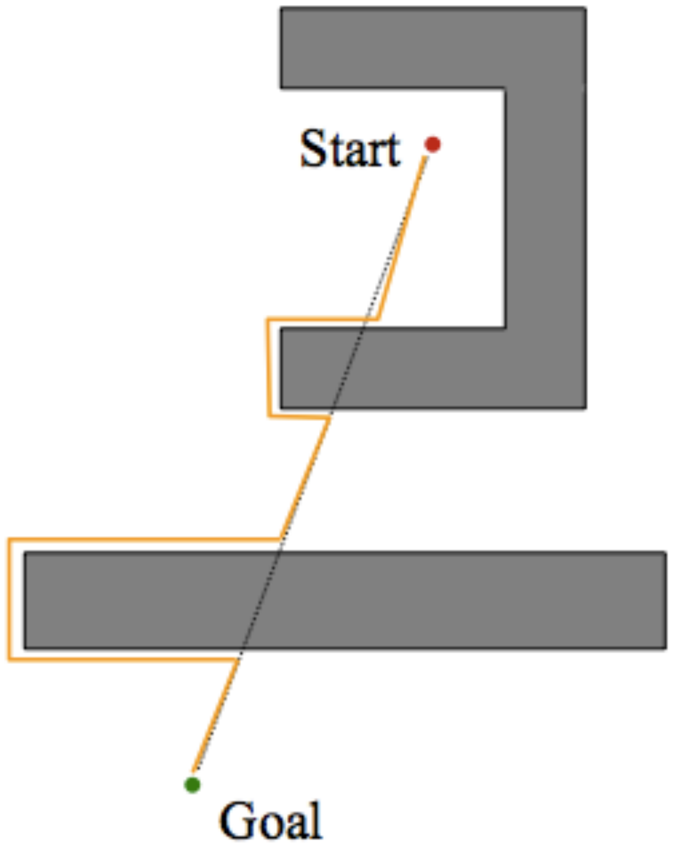




Potential problem

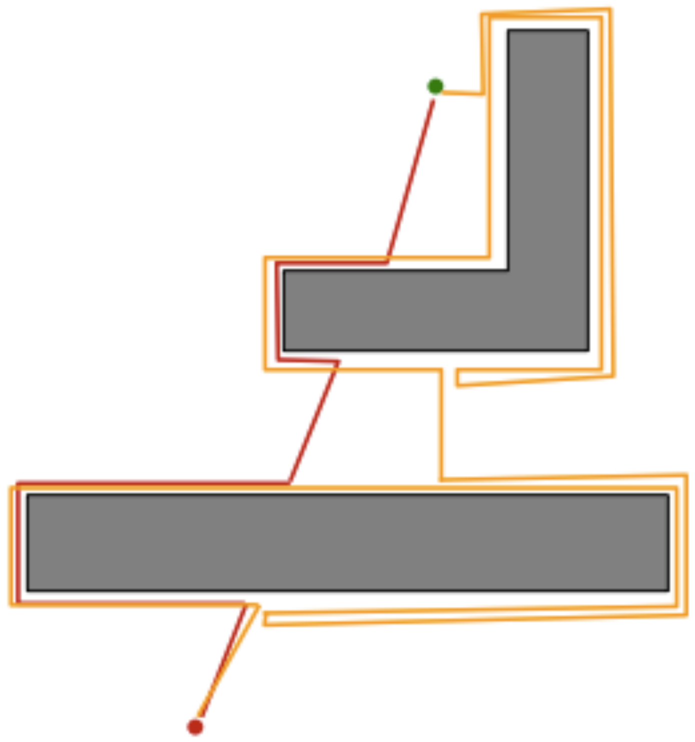
- Follow the obstacle until you encounter line closer to the goal
- Leave the obstacle and continue



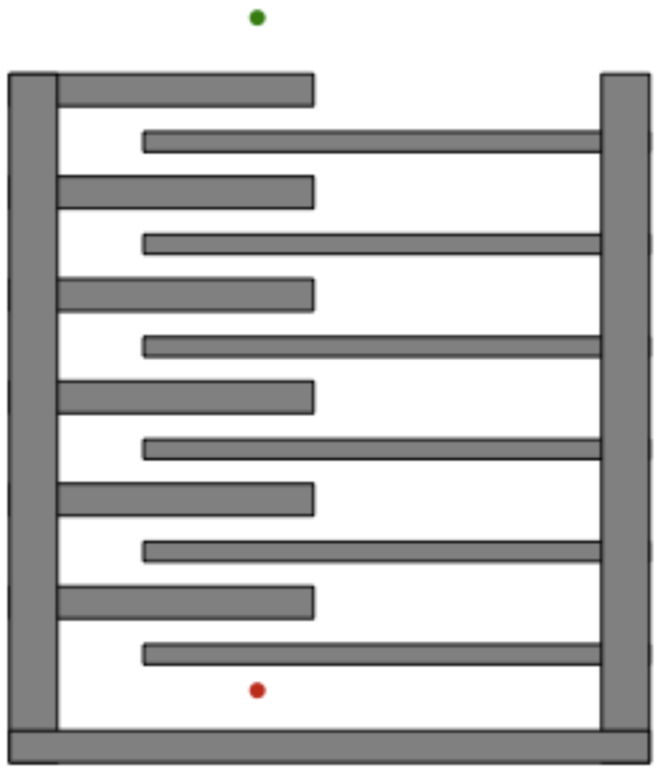


Bug 1 vs Bug 2

- Bug 2 beats Bug 1



Bug 1 beats Bug2



Bug 1 vs Bug 2

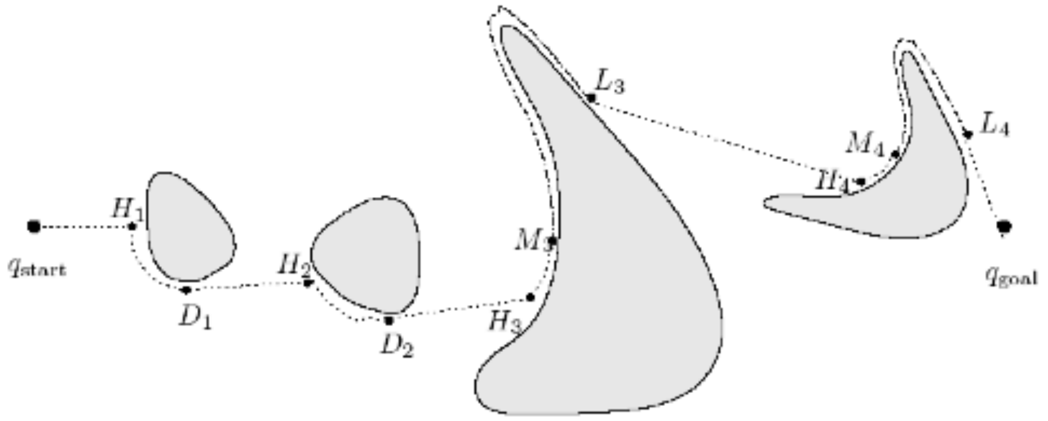
- Bug 1 is an exhaustive search algorithm
  - looks at all choices before committing

- Bug 2 is a greedy algorithm, takes the first thing that looks better

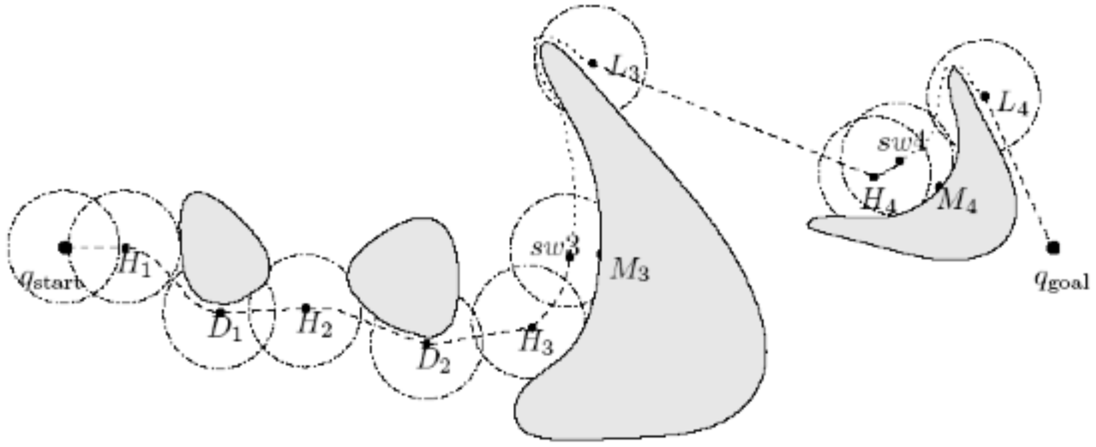
- Bug 2 in many cases out performs Bug 1, but Bug 1 is overall more reliable

Examples in practice

- Example using zero range sensors



- Example with finite range sensors



Map based planning

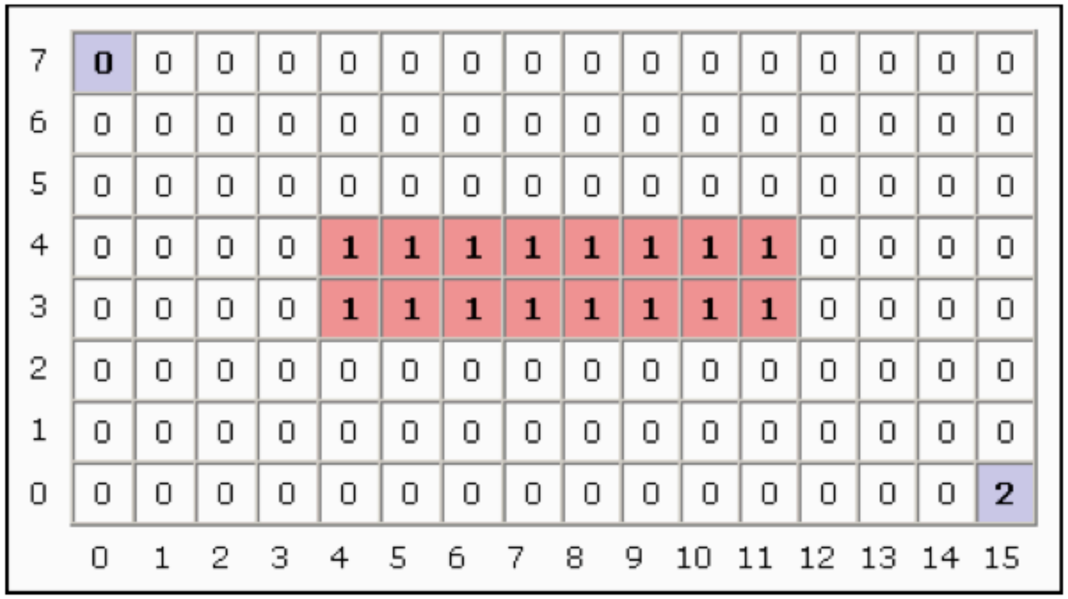
- Consider a simple case of map
- Grid with cells marked as occupied or empty
- Formulate path finding problem as a search
- What is the sequence of states you need to visit in order to reach the goal

Navigation functions - Discrete version

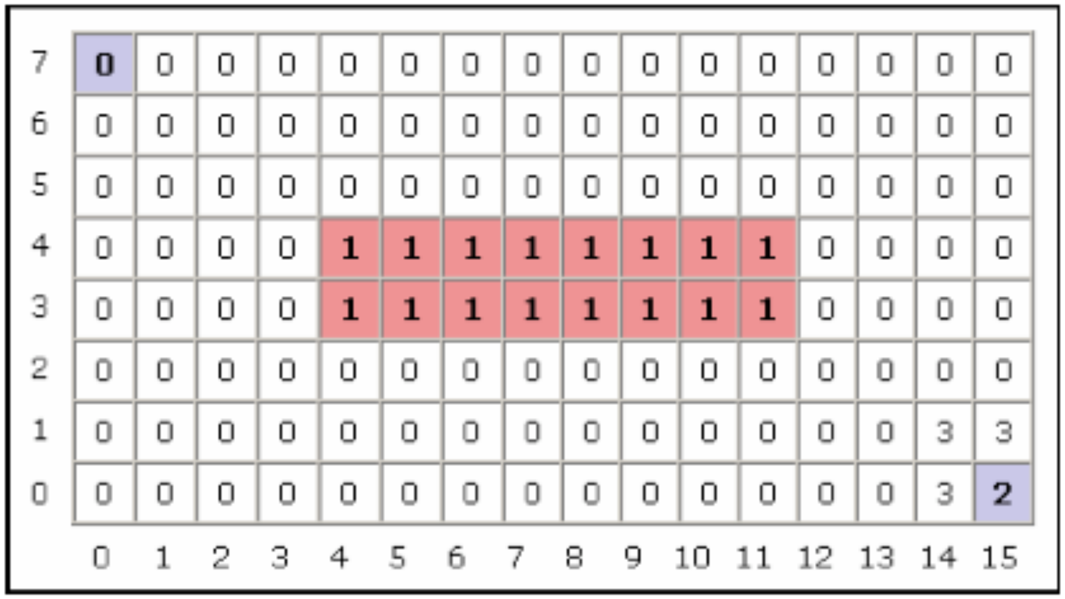
- Useful potential functions – for any starting point you will reach a goal: Navigation function (see previously how to define these in continuous spaces)
- In discrete state space – navigation function has to have some value for each state, consider grid
- Goal: will have zero potential
- Obstacles have infinite potential
- Example of useful navigation function: optimal cost to go to goal G* (example) , assuming that for each state

$$l(x,u) = 1$$

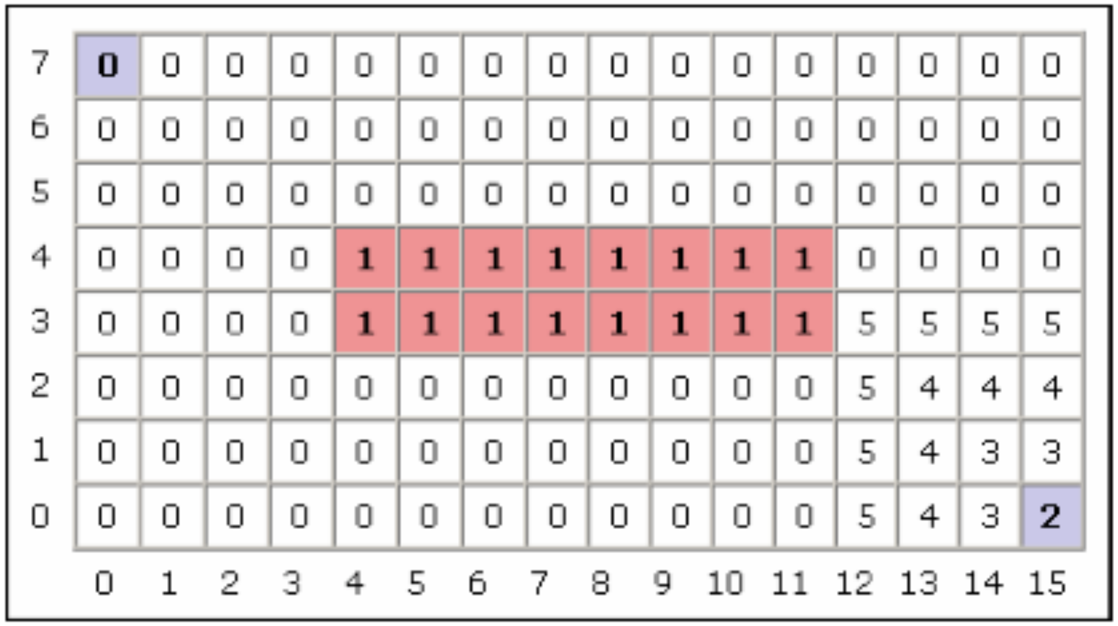
Wavefront planner



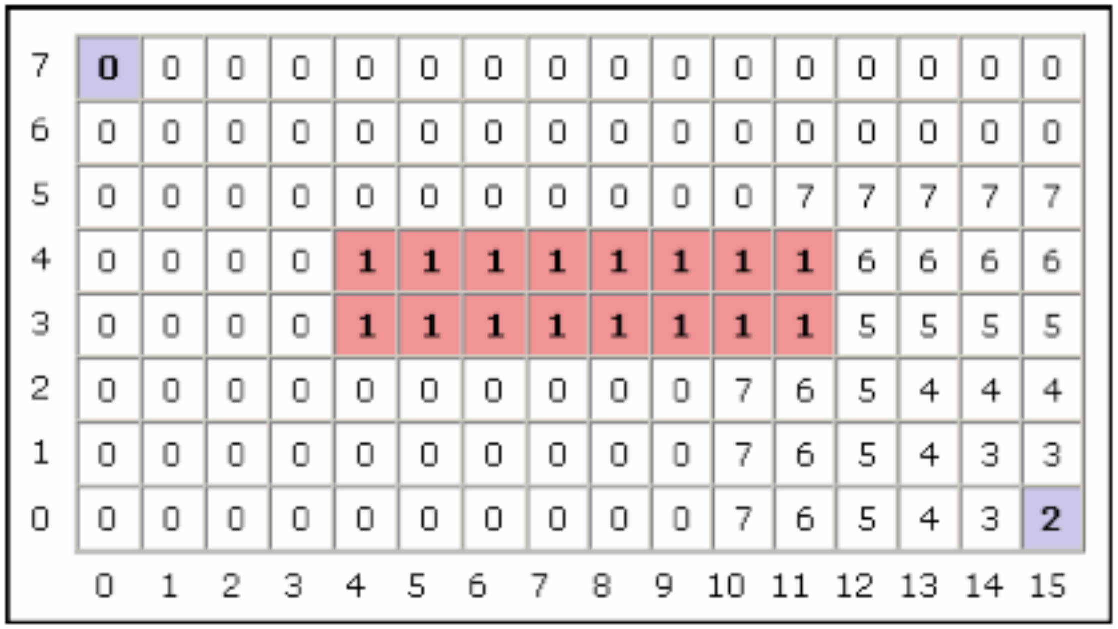
7	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0
3	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



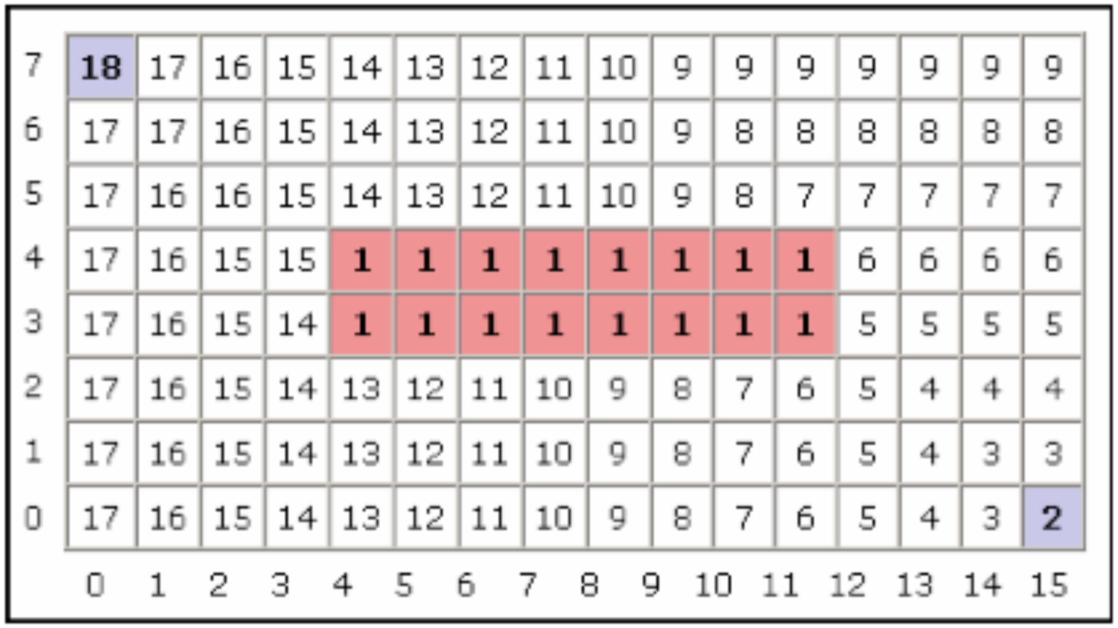
7	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0
3	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	**2**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



7	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0
3	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	5	5	5	5
2	0	0	0	0	0	0	0	0	0	0	0	0	5	4	4	4
1	0	0	0	0	0	0	0	0	0	0	0	0	5	4	3	3
0	0	0	0	0	0	0	0	0	0	0	0	0	5	4	3	**2**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



7	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	7	7	7	7	7
4	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	6	6	6	6
3	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	5	5	5	5
2	0	0	0	0	0	0	0	0	0	0	7	6	5	4	4	4
1	0	0	0	0	0	0	0	0	0	0	7	6	5	4	3	3
0	0	0	0	0	0	0	0	0	0	0	7	6	5	4	3	**2**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

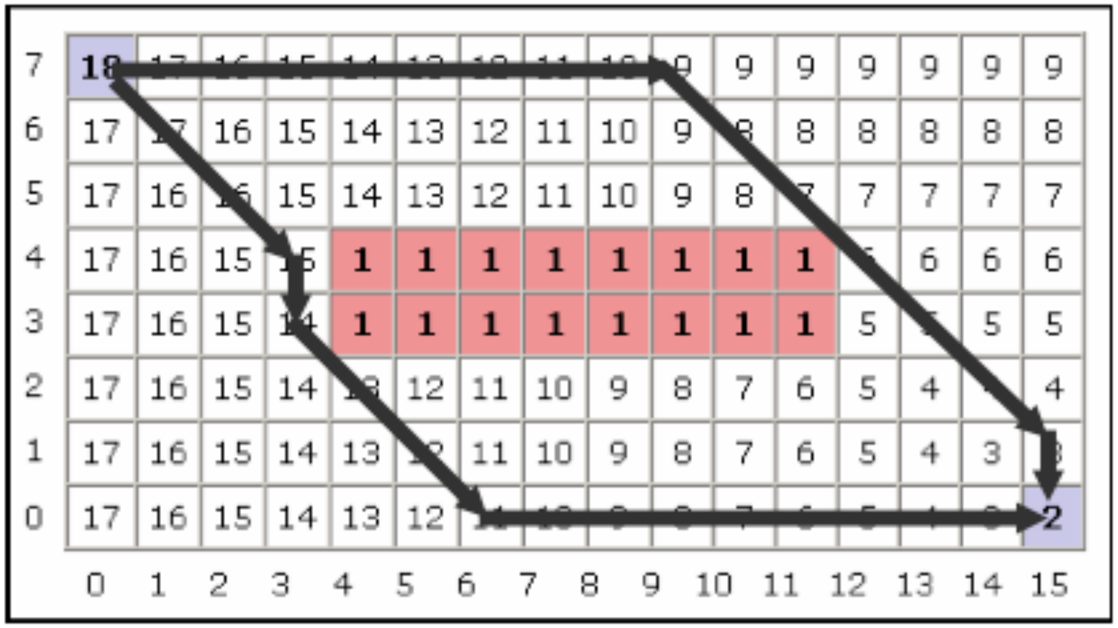


7	**18**	17	16	15	14	13	12	11	10	9	9	9	9	9	9	9
6	17	17	16	15	14	13	12	11	10	9	8	8	8	8	8	8
5	17	16	16	15	14	13	12	11	10	9	8	7	7	7	7	7
4	17	16	15	15	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	6	6	6	6
3	17	16	15	14	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	5	5	5	5
2	17	16	15	14	13	12	11	10	9	8	7	6	5	4	4	4
1	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	3
0	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	**2**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- Zeros should exist only in unreachable regions exist
- Ones for regions which cannot be reached
- Other book keeping methods can be used

Finding the path

- From any initial grid cell, move toward the cell
- with the lowest number – follow the steepest
- descent

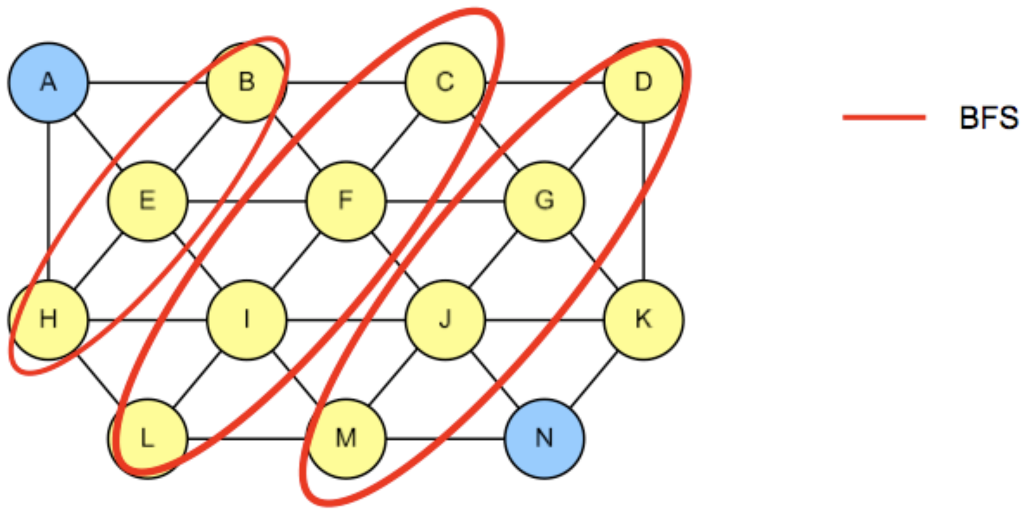


Wavefront planner

- Can be pre-computed using distance transform

Path Planning as search

- Various performance measures of search:
- Optimality, completeness, time and space complexity
- Uninformed search – blind no information is gathered from the environment (BFS, DFS)
- Informed search - some evaluation function is used
- Dijkstra algorithm – shortest path
- A* algorithm – heuristics for pruning the search
- D* dynamic version of A*



Generalization of occupancy grid

- Occupancy grid – cells are occupied or empty
- We can define more general notion of a cost
- C(x,y) is infinite for obstacles
- C(x,y) can be large for hard to traverse regions
- i.e. cost is inversely proportional to traversability

- You can also incorporate passenger comfort, related to roughness of terrain.

A*

- Complete provided finite boundary condition
- Optimal in terms of path cost
- Memory inefficient
- Exponential growth of search space with respect to the length of the solution
- How can we use it in partially known, or dynamically changing environment D*

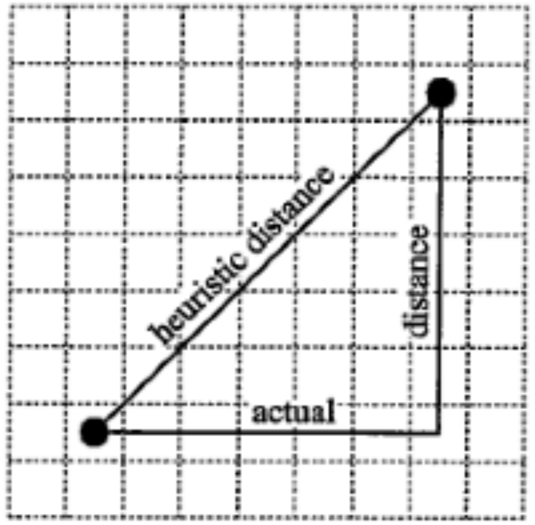
- Same as Dijkstra – different way of updating the cost of each node
- As long as heuristics under-estimates the cost to go – A* is optimal
- In may practical problems it is hard to find good heuristics

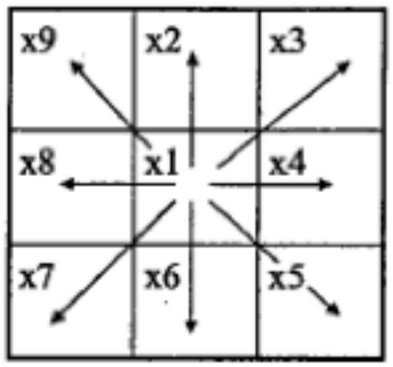
A*

- Extension to Dijkstra, tries to reduce the number of states
- Using heuristic estimate of the cost to go
- Evaluation function f(n) = g(n) + h(n)
- Operating cost function g(n) – cost so far
- Heuristic function h(n)

  information used to find promising node to take next

  heuristics is admissible if it never overestimates the actual cost





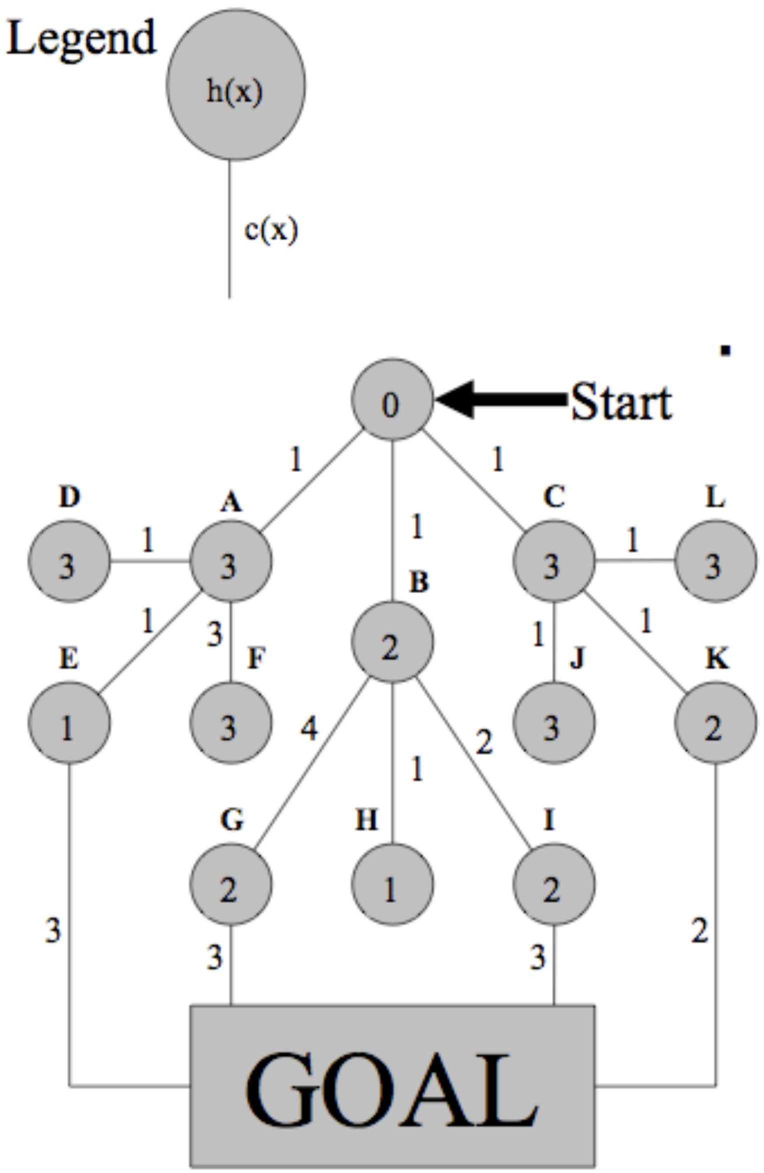
c(x1, x2) = 1
c(x1, x9) = 1.4

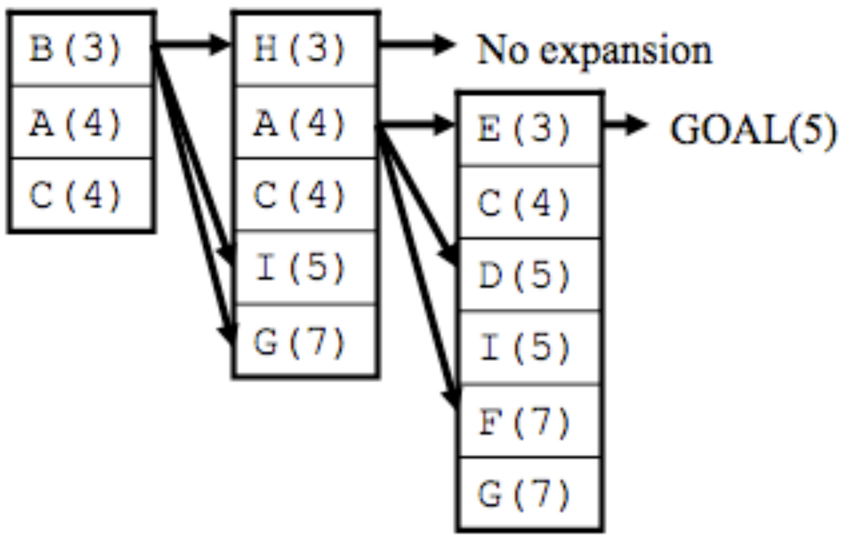
c(x1, x8) = 10000, if x8 is in obstacle, x1 is a free cell

c(x1,x9) = 10000.4, if x9 is in obstacle, x1 is a free cell

Cost on a grid

A*

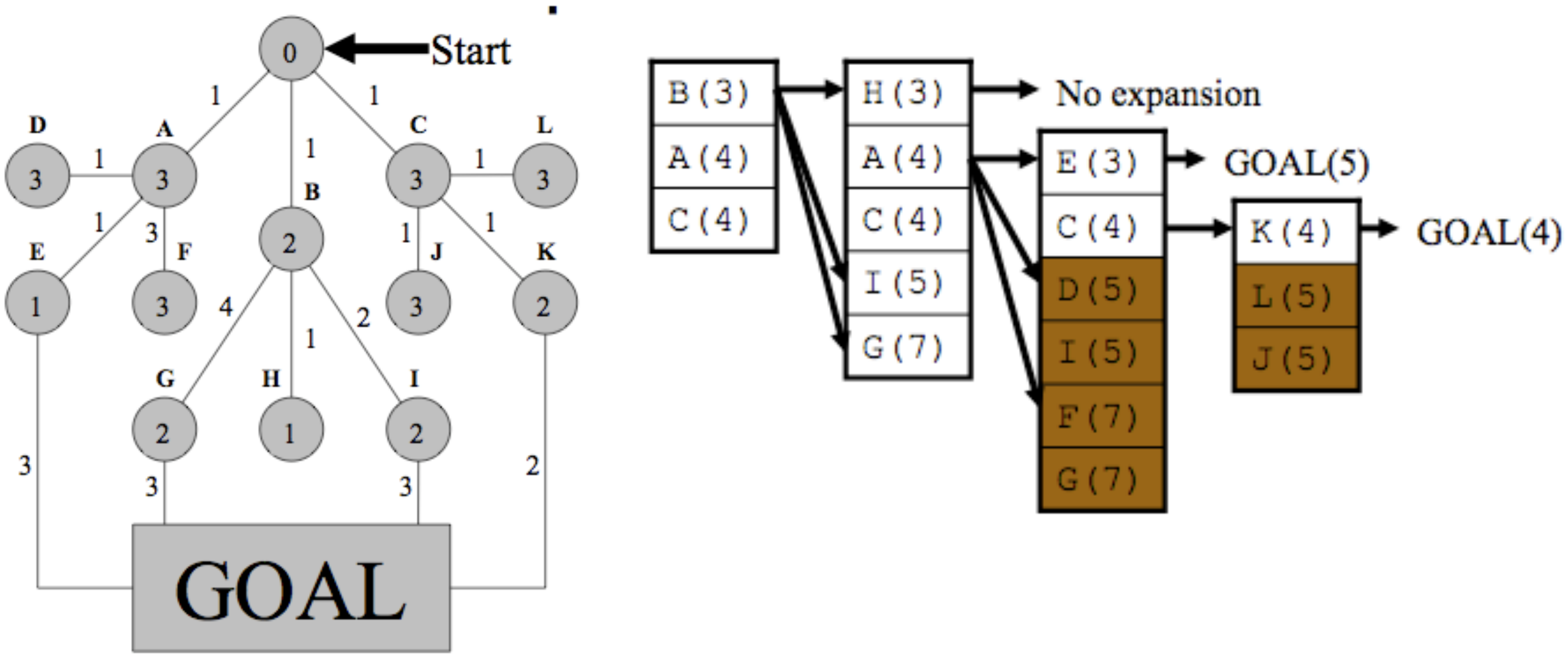




Start => A => E => goal

Nodes with higher or equal priority then goal can be pruned away
There can still be shorter path through the remaining nodes

A*

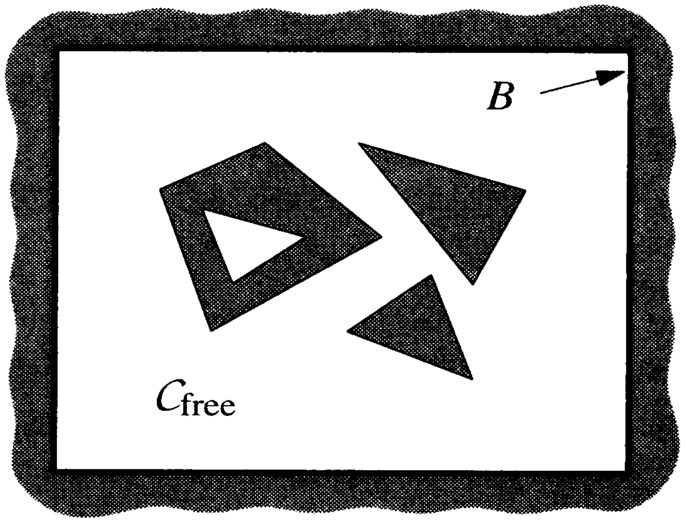


Keep on expanding nodes with the priority level lower then goal
Path: Start => C = > K => Goal

Classical Motion Planning

- Given a continuous space with obstacles plan a path between configuration A and configuration B
- Given a point robot and a workspace described by polygons
- Large class of methods – transform continuous space to discrete

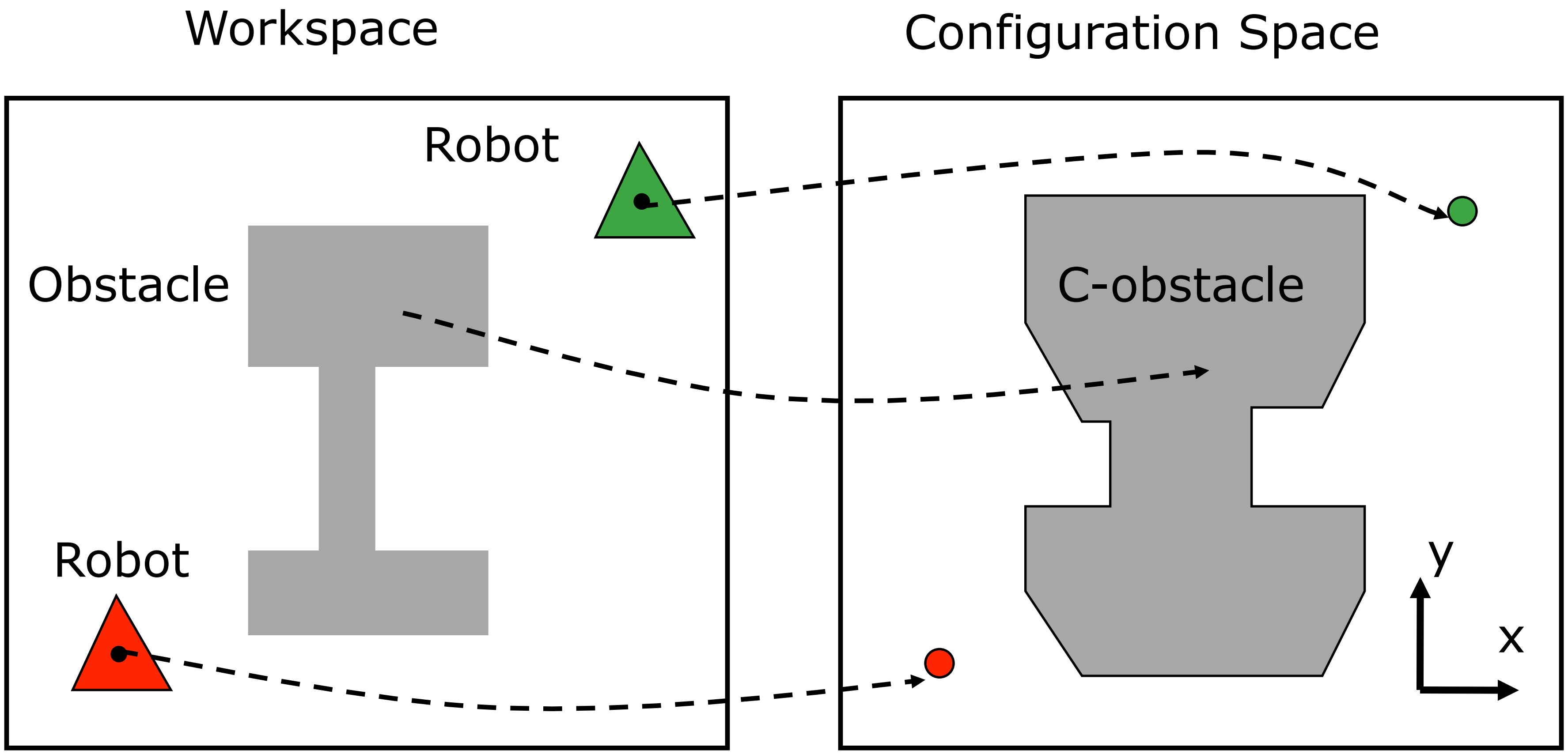
- **Roadmap methods**
  - Visibility graph
  - Cell decomposition
  - Retraction



Configuration Space

- Well, most robot is not a point and can have arbitrary shape
- What should we do if our robot is not a point?
- Convert rigid robots, articulated robots, *etc.* into points
- Apply algorithms for moving points

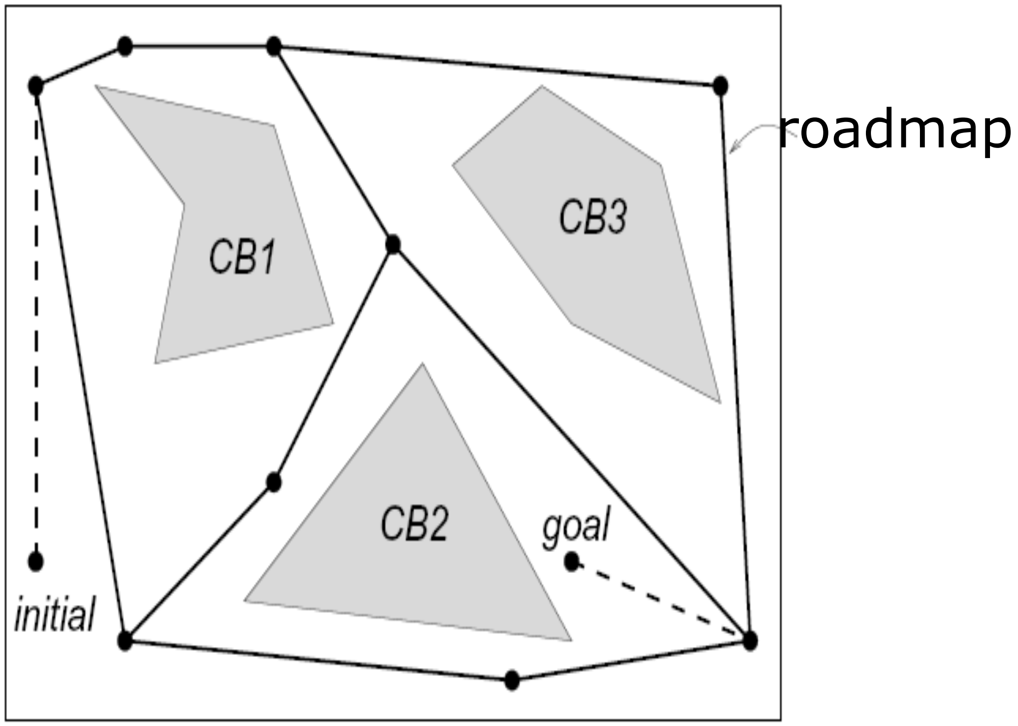
Configuration Space

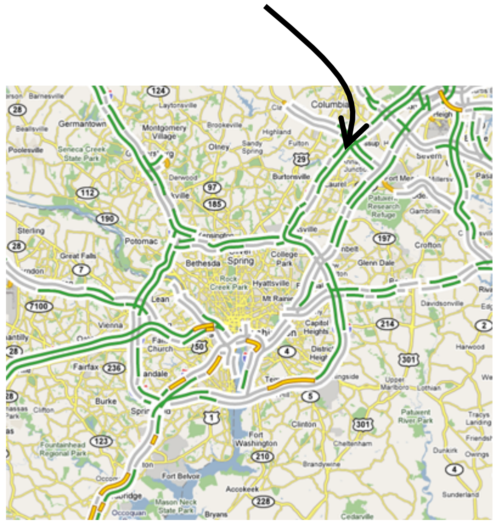


- for now C-obstacle is a polygon.
- how to compute C-space bit later

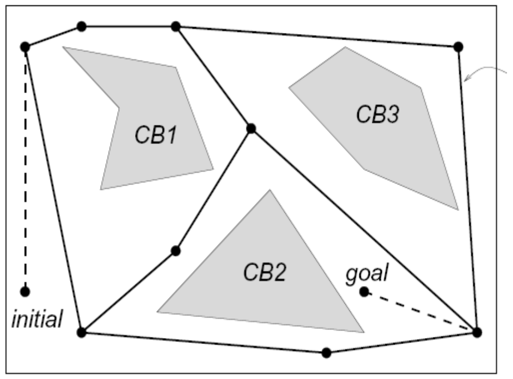
Roadmap Methods

Capture the connectivity of C_{free} with a roadmap (graph or network) of one-dimensional curves





Roadmap Methods



Path Planning with a Roadmap
Input: configurations q_{init} and q_{goal} , and B
Output: a path in C_{free} connecting q_{init} and q_{goal}

1. Build a roadmap in C_{free} (preprocessing)
• roadmap nodes are free configurations
• two nodes connected by edge if can (easily) move between them

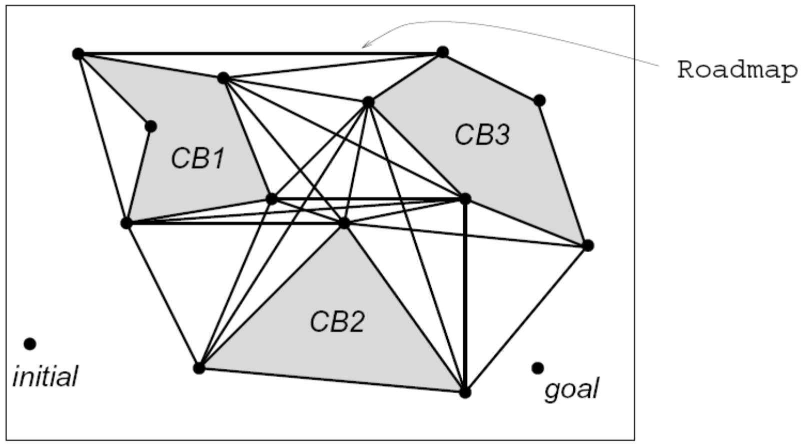
2. Connect q_{init} and q_{goal} to roadmap nodes v_{init} and v_{goal}

3. Find a path in the roadmap between v_{init} and v_{goal}
- directly gives a path in C_{free}

difficult part

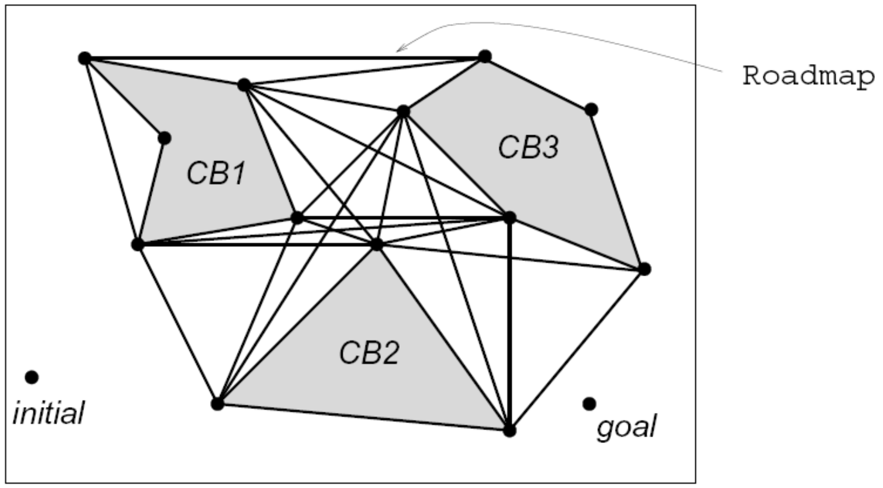
Visibility Graph

- A visibility graph of C-space for a given C-obstacle is an undirected graph G where
  - nodes in G correspond to vertices of C-obstacle
  - nodes connected by edge in G if
    - they are connected by an edge in C-obstacle, or
    - the straight line segment connecting them lies entirely in *Cfree*
  - (could add q_{init} and q_{goal} as roadmap nodes)



Visibility Graph

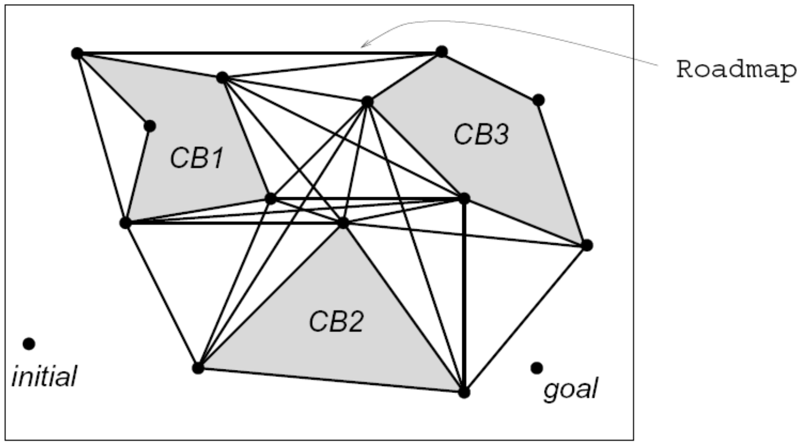
- **Brute Force Algorithm**
  - add all edges in C-obstacle to G
  - for each pair of vertices (x, y) of C-obstacle, add the edge (x, y) to G if the straight line segment connecting them lies entirely in cl(C-free)
    - test (x; y) for intersection with all O(*n*) edges of C-obstacle
    - O(n^2) pairs to test, each test takes O(*n*) time



Complexity: O(n^3), *n* is number of vertices in C-obstacle

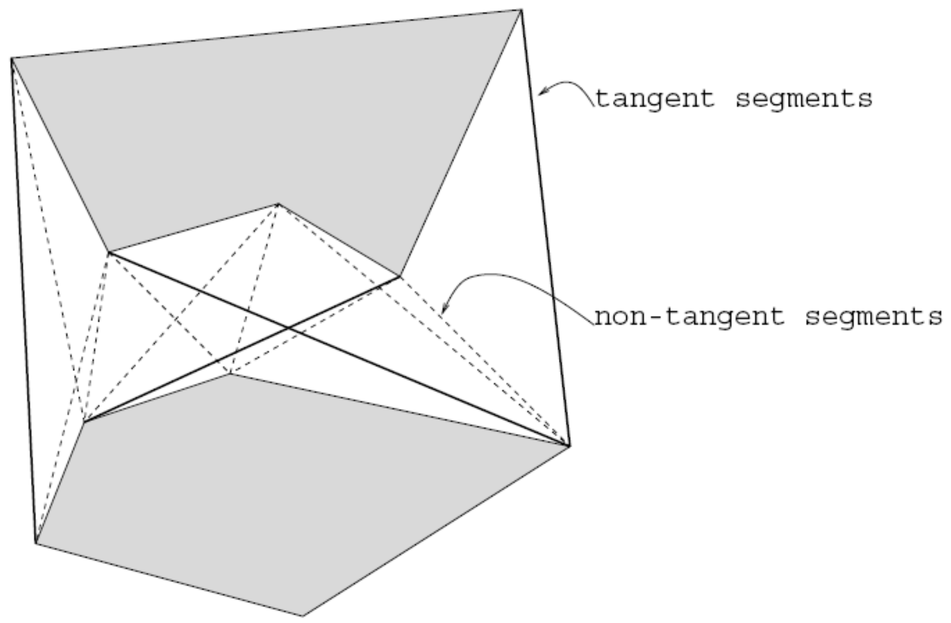
Visibility Graph

- **Visibility graphs – good news**
  - are conceptually simple
  - shortest paths (if the query cannot see each other)
  - we have efficient algorithms if Workspace is polygonal
    - *$O(n^2)$,* where n is number of vertices of C-obstacle
    - *$O(k + n \log n)$*, where k is number of edges in G
  - we can make a 'reduced' visibility graph (don't need all edges)



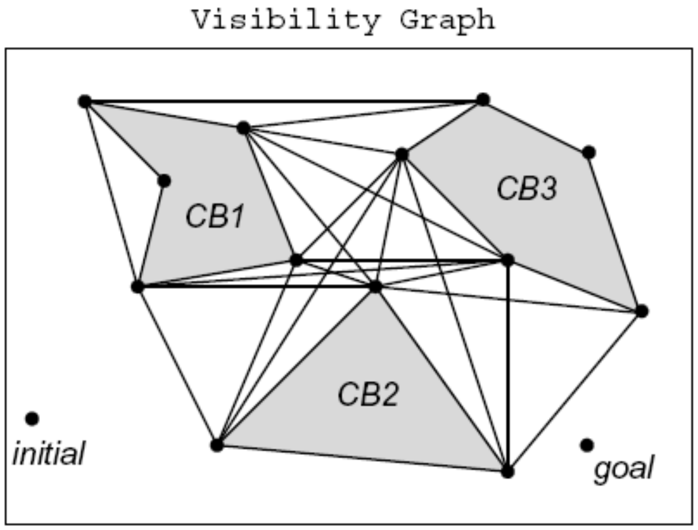
Reduced Visibility Graph

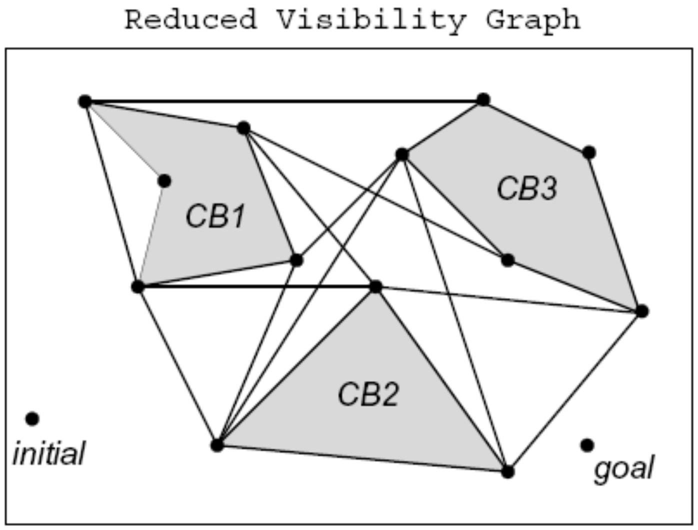
- we don't really need all the edges in the visibility graph (even if we want shortest paths)
- Definition: Let *L* be the line passing through an edge (x; y) in the visibility graph G. The segment (x; y) is a tangent segment *iff* L is tangent to C-obstacle at both x and y.
- Line segment is tangent if extending the line beyond each of the end points would not intersect the obstacles



Reduced Visibility Graph

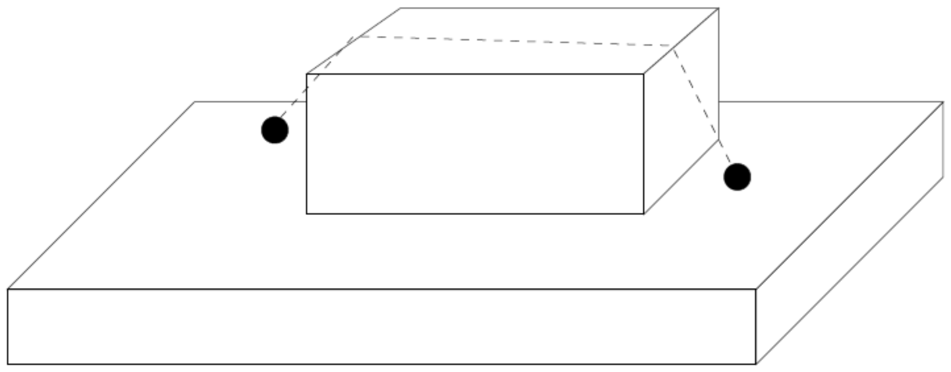
- It turns out we need only keep
  - convex vertices of C-obstacle
  - non-CB edges that are tangent segments





Visibility Graph in 3-D

- Visibility graphs don't necessarily contain shortest paths in R^3
  - in fact finding shortest paths in R^3 is NP-hard [Canny 1988]
  - $(1 + \varepsilon^2)$ approximation algorithm [Papadimitriou 1985]

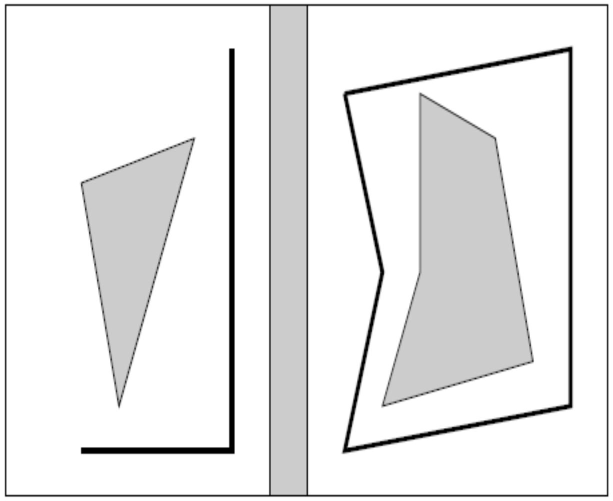


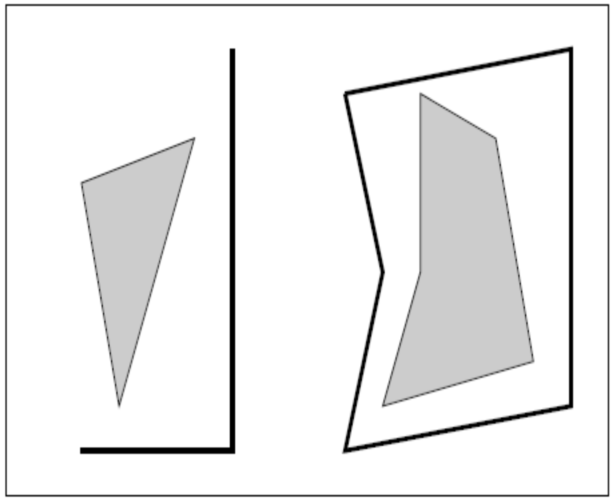
Bad news: Visibility graphs really only suitable for 2D C

Retraction Approach

- Basic Idea: 'retract' C_{free} onto a 1-dimensional subset of itself (the roadmap).
- a map $\rho : C_{free} \rightarrow R$, $R \subset C_{free}$, is a retraction *iff* it is continuous and its restriction to R is the identity map (i.e., $\rho(R) = R$)
  - thus, $\rho(x) \in R$ for all $x \in C_{free}$ and $\rho(y) = y$ for all $y \in R$
  - a retraction $\rho : C_{free} \subset R$ is connectivity preserving iff for all $x \subset C_{free}$, x and $\rho(x)$ belong to the same connected component of C_{free}

Retraction Approach


a retraction


not a retraction
(not continuous)

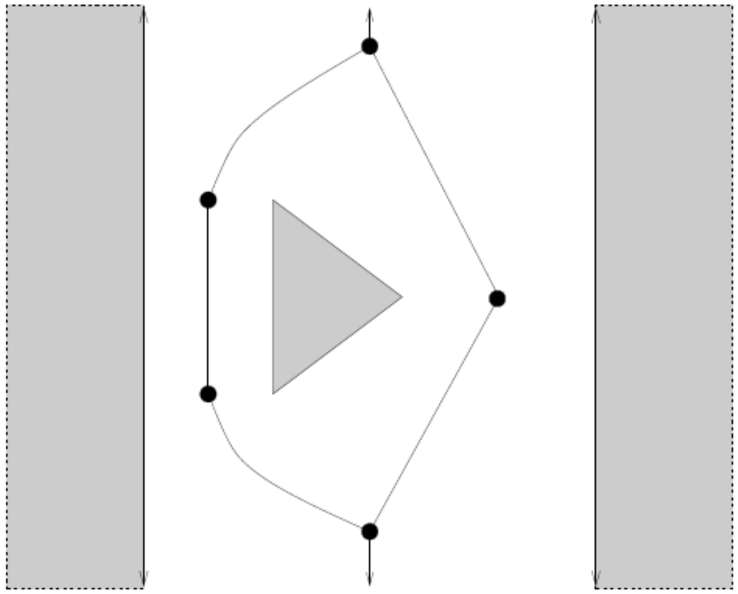
Fact: There exists a free path from q_{init} to q_{goal} iff there exists a path in
R between ρ (qinit) and ρ (qgoal).

This is why retractions make good roadmaps.

Retraction Approach

- Retraction Example: Generalized Voronoi Diagrams (or Medial axis)

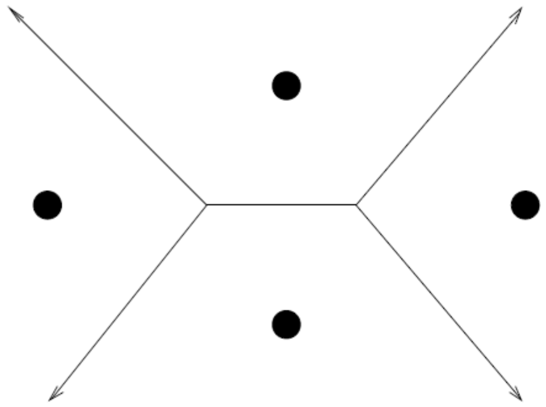
Generalized Voronoi Diagram
$\mathrm{Vor}(C_{free}) = \{ q \in C_{free} \mid \mathrm{card}(\mathrm{near}(q)) > 1 \}$
i.e., points in C_{free} with at least two nearest neighbors in C_{free}'s boundary



Voronoi Diagram for Point Sets

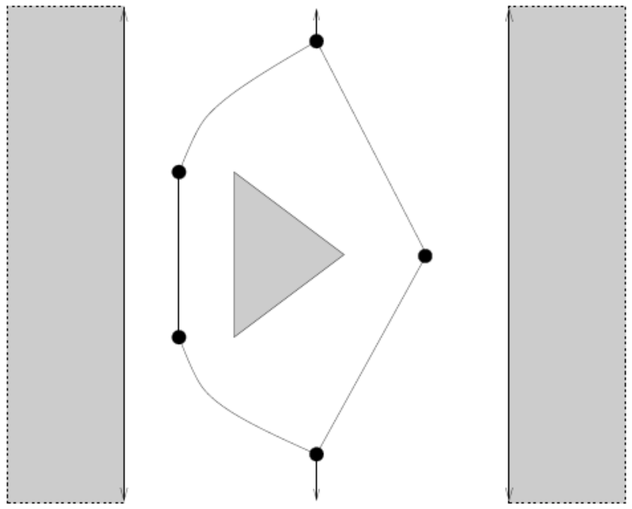
- Voronoi diagram of point set *X* consists of straight line segments, constructed by
  - computing lines bisecting each pair of points and their intersections
  - computing intersections of these lines
  - keeping segments with more than one nearest neighbor

- segments of Vor(*X*) have largest clearance from *X* and regions identify closest point of *X*



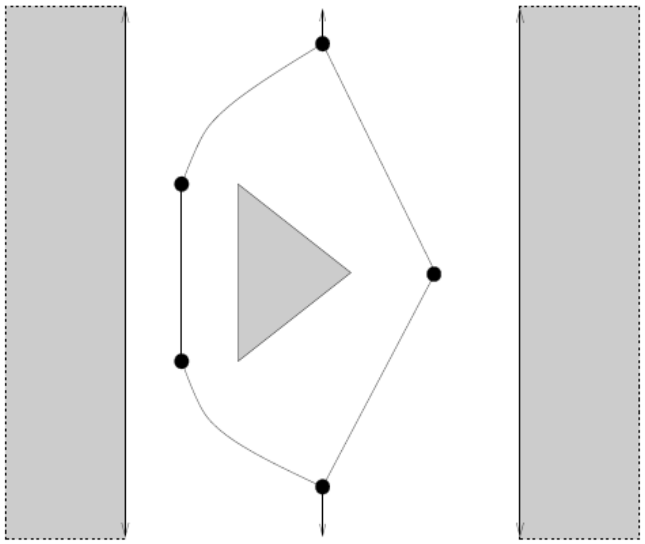
Voronoi Diagram for Point Sets

- When C = R^2 and polygonal C-obstacle, Vor(Cfree) consists of a finite collection of straight line segments and parabolic curve segments (called arcs)
  - straight arcs are defined by two vertices or two edges of C-obstacle, i.e., the set of points equally close to two points (or two line segments) is a line
  - parabolic arcs are defined by one vertex and one edge of C-obstacle, i.e., the set of points equally close to a point and a line is a parabola



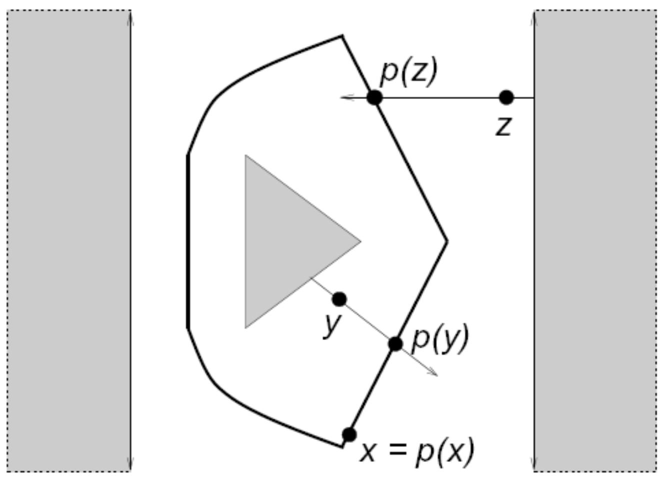
Voronoi Diagram for Point Sets

- Naive Method of Constucting V or (Cfree)
  - compute all arcs (for each vertex-vertex, edge-edge, and vertex-edge pair)
  - compute all intersection points (dividing arcs into segments)
  - keep segments which are closest only to the vertices/ edges that defined them



Retraction

- Retraction $\rho : C_{free} \rightarrow \text{Vor}(C_{free})$



To find a path:

1. compute $\text{Vor}(C_{free})$
2. find paths from q_{init} and q_{goal} to $\rho(q_{init})$ and $\rho(q_{goal})$, respectively
3. search $\text{Vor}(C_{free})$ for a set of arcs connecting $\rho(q_{init})$ and $\rho(q_{goal})$

Cell Decomposition

- Idea: decompose C_{free} into a collection K of non-overlapping cells such that the union of all the cells exactly equals the free C-space
- Cell Characteristics:
  - geometry of cells should be simple so that it is easy to compute a path between any two configurations in a cell
  - it should be pretty easy to test the adjacency of two cells, i.e., whether they share a boundary
  - it should be pretty easy to find a path crossing the portion of the boundary shared by two adjacent cells
- Thus, cell boundaries correspond to 'criticalities' in *C*, i.e., something changes when a cell boundary is crossed. No such criticalities in *C* occur within a cell.

Cell Decomposition

- **Preprocessing:**
  - represent C_{free} as a collection of cells (connected regions of C_{free})
    - planning between configurations in the same cell should be 'easy'
  - build connectivity graph representing adjacency relations between cells
    - cells adjacent if can move directly between them
- **Query:**
  - locate cells k_{init} and k_{goal} containing start and goal configurations
  - search the connectivity graph for a 'channel' or sequence of adjacent cells connecting k_{init} and k_{goal}
  - find a path that is contained in the channel of cells

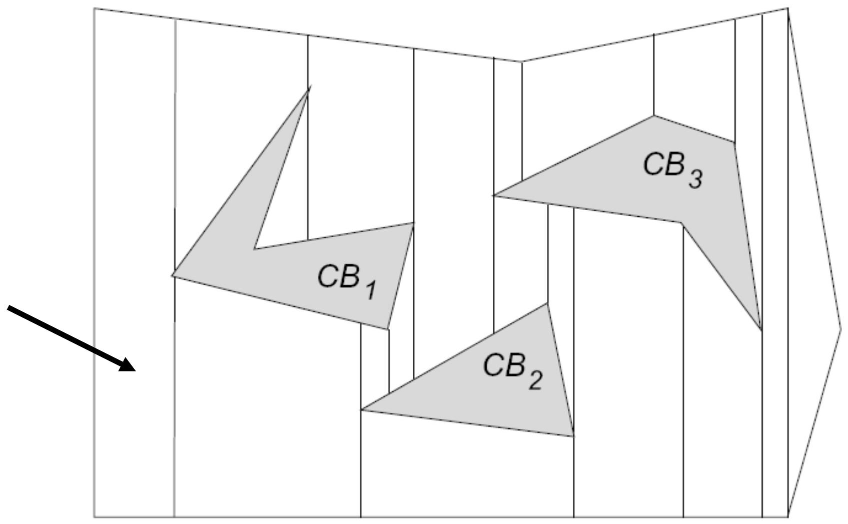
Difficult

- Two major variants of methods:
  - exact cell decomposition:
    - set of cells exactly covers C_{free}
    - complicated cells with irregular boundaries (contact constraints)
    - harder to compute
  - approximate cell decomposition:
    - set of cells approximately covers C_{free}
    - simpler cells with more regular boundaries
    - easier to compute

Trapezoidal Decomposition

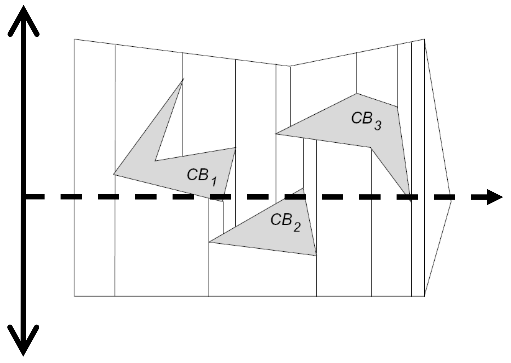
- Basic Idea: at every vertex of C-obstacle, extend a vertical line up and down in Cfree until it touches a C-obstacle or the boundary of Cfree

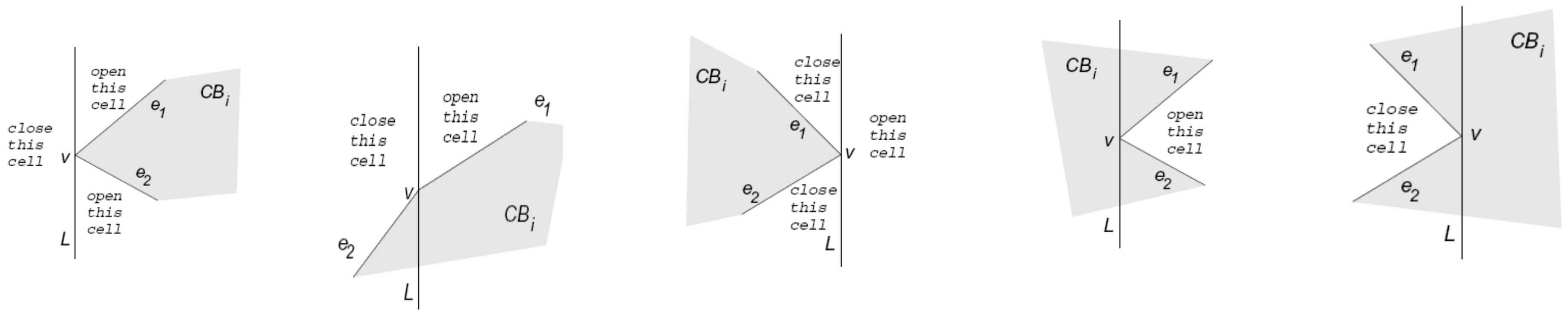
trapezoid



Trapezoidal Decomposition

- Sweep line algorithm
  - Add vertical lines as we sweep from left to right
  - Events need to be handled accordingly





trapezoidal decomposition can be built in O(n log n) time

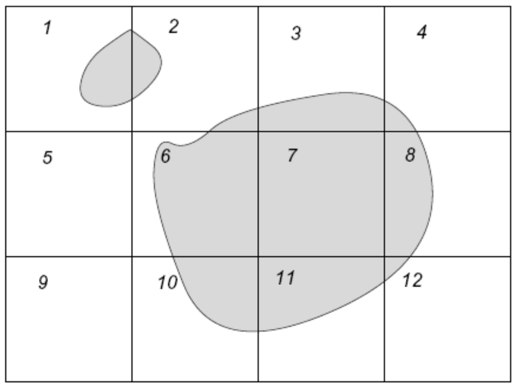
Approx. Cell Decomposition

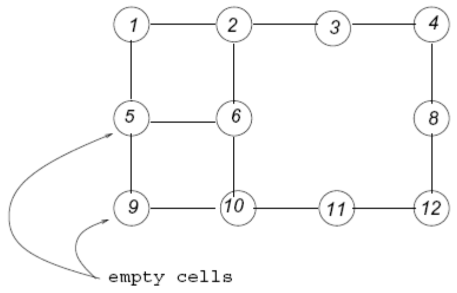
- Construct a collection of non-overlapping cells such that the union of all the cells approximately covers the free C-space!
- Cell characteristics
  - Cell should have simple shape
  - Easy to test adjacency of two cells
  - Easy to find path across two adjacent cells

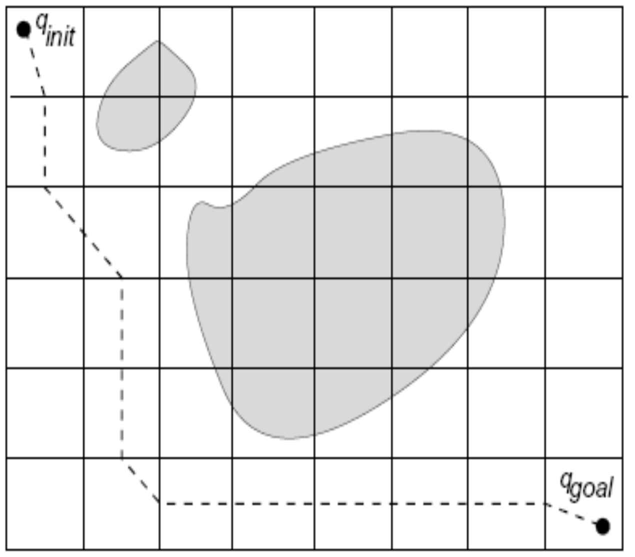
Approx. Cell Decomposition

- Each cell is
- – Empty
- – Full
- – Mixed

- Different resolution
- – Different roadmap

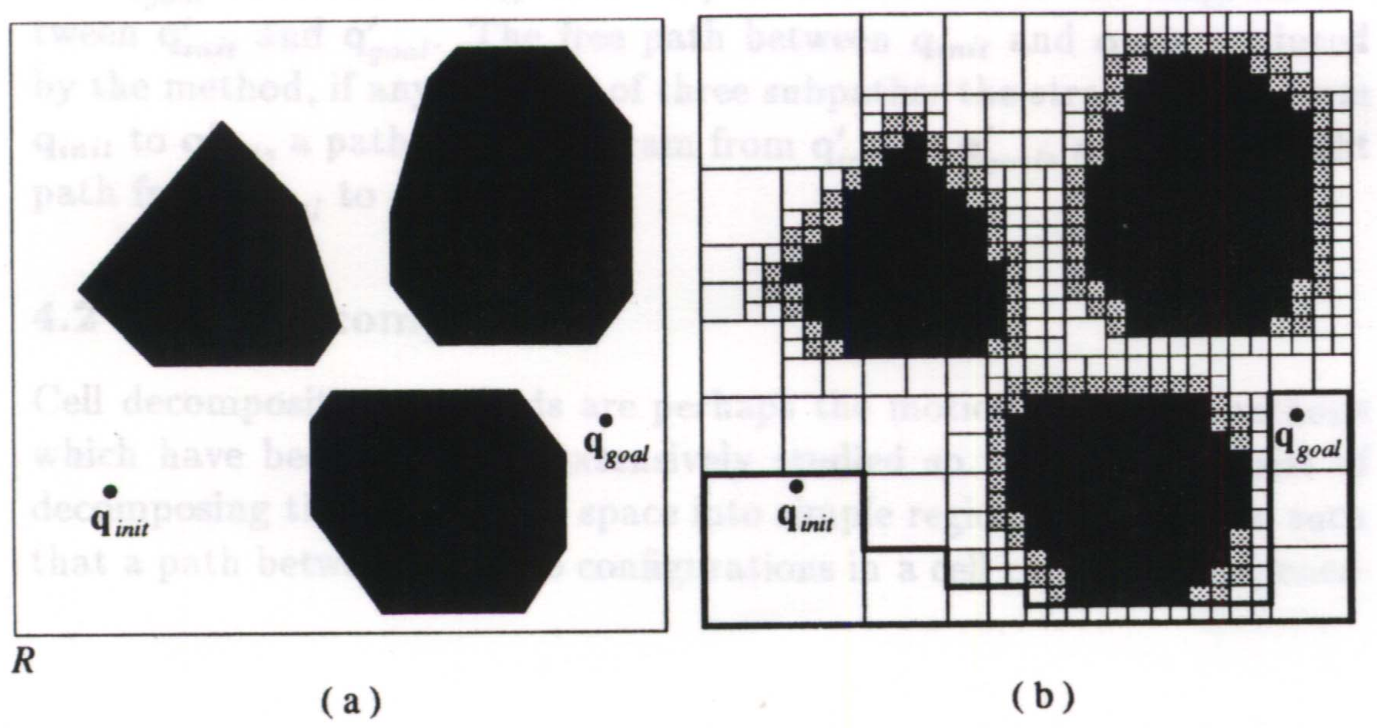






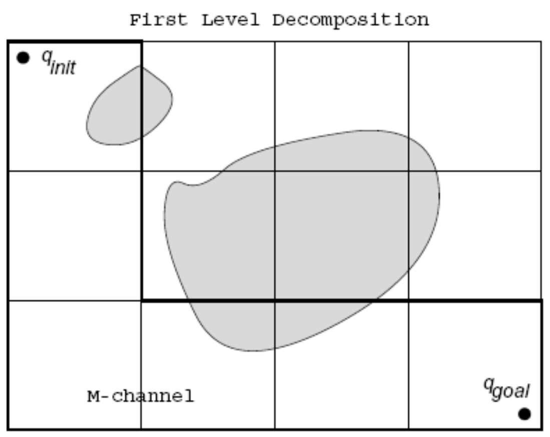
Approx. Cell Decomposition

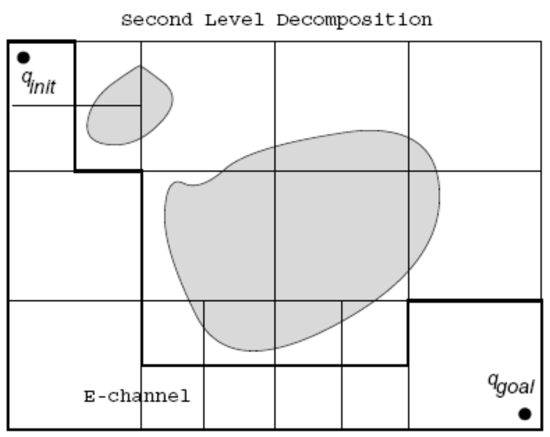
- Higher resolution around CBs



Approx. Cell Decomposition

- Hierarchical approach
  - Find path using empty and mixed cells
  - Further decompose mixed cells into smaller cells





Approx. Cell Decomposition

- Advantages:
  - simple, uniform decomposition
  - easy implementation
  - adaptive
- Disadvantages:
  - large storage requirement
  - Lose completeness

- Bottom line 1: We sacrifice exactness for simplicity and efficiency

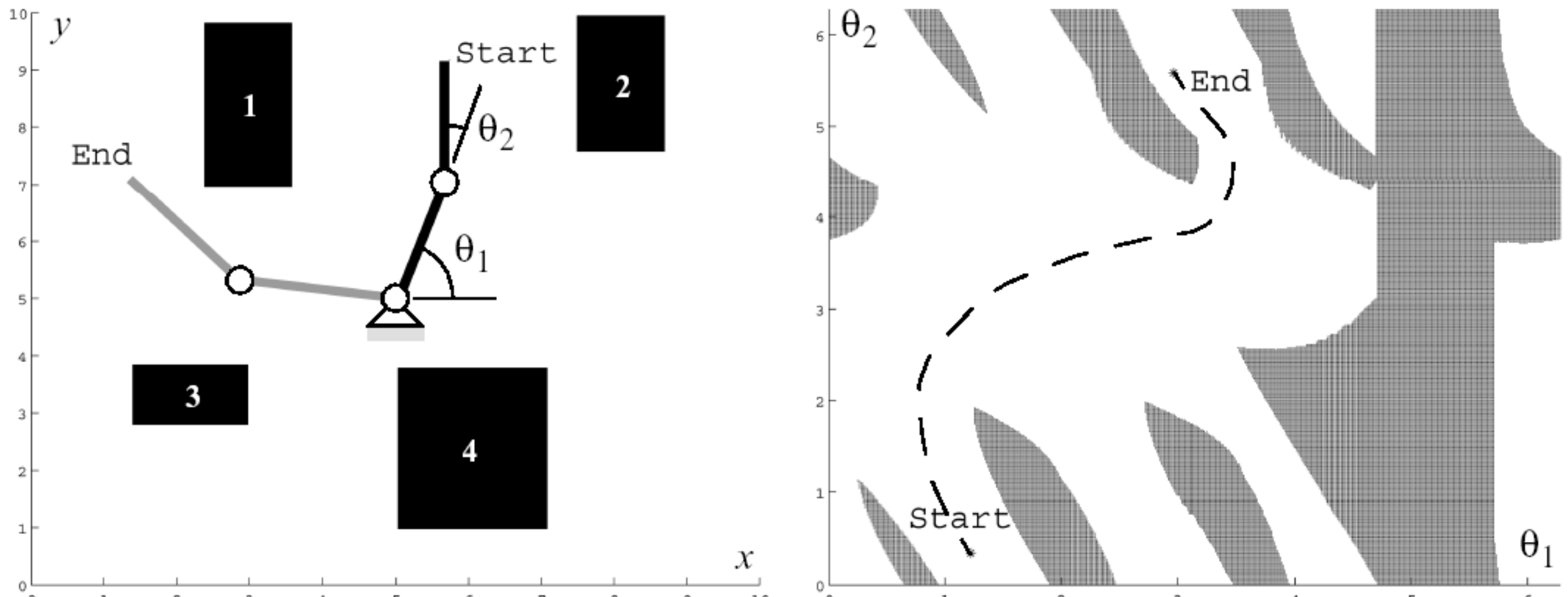
- Bottom line 2: Approx. cell decomposition methods are practically for lower dimension C, i.e., dof <5, b/c they generate too many cells, i.e. (N^d) cells in d dimension

Configuration Space

- Well, most robot is not a point and can have arbitrary shape
- What should we do if our robot is not a point?
- Convert rigid robots, articulated robots, *etc.* into points
- Apply algorithms for moving points

Configuration Space

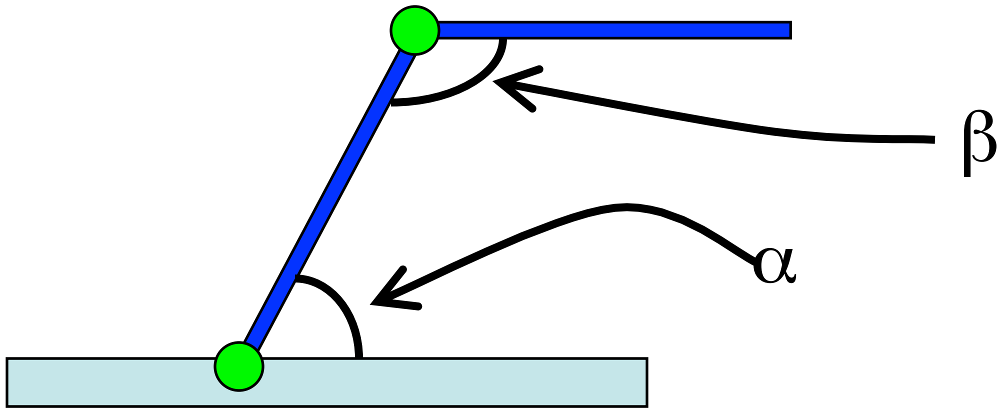
- Mapping between the workspace and configuration space



a) b)

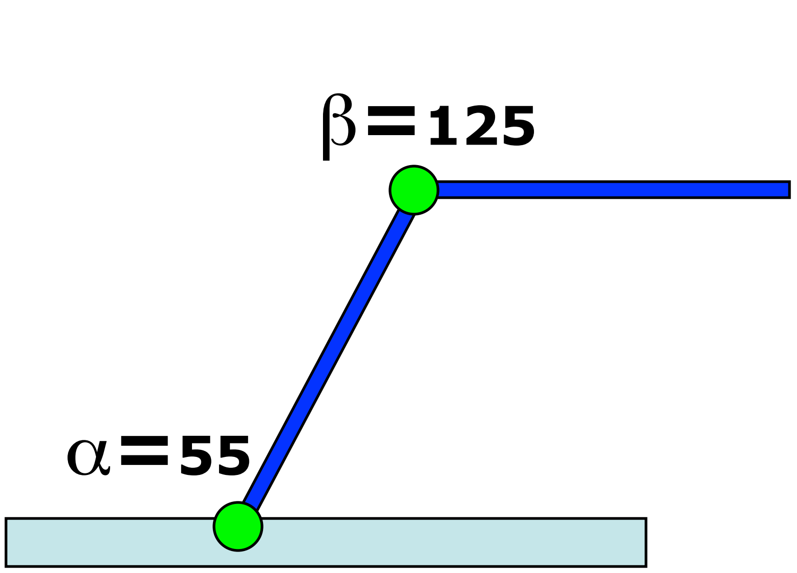
Workspace

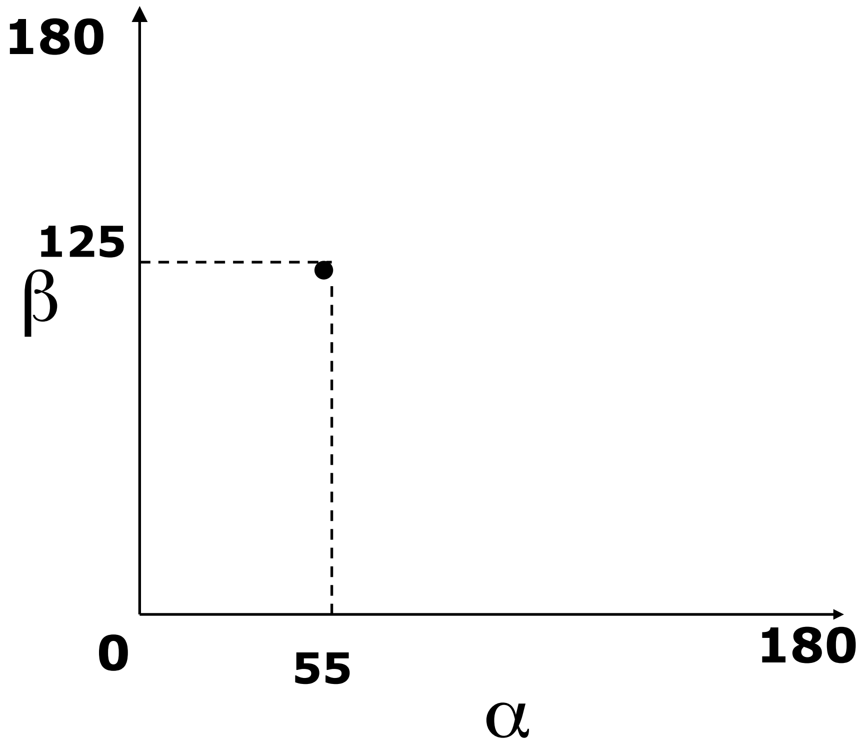
Degree of freedom (DOF)



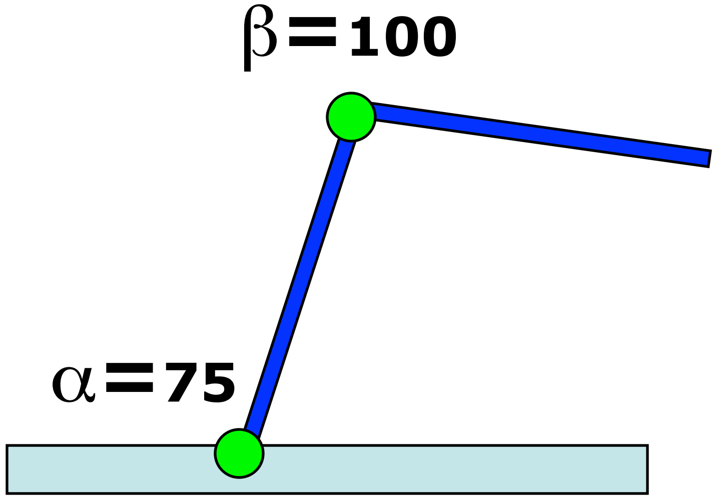
Configuration Space

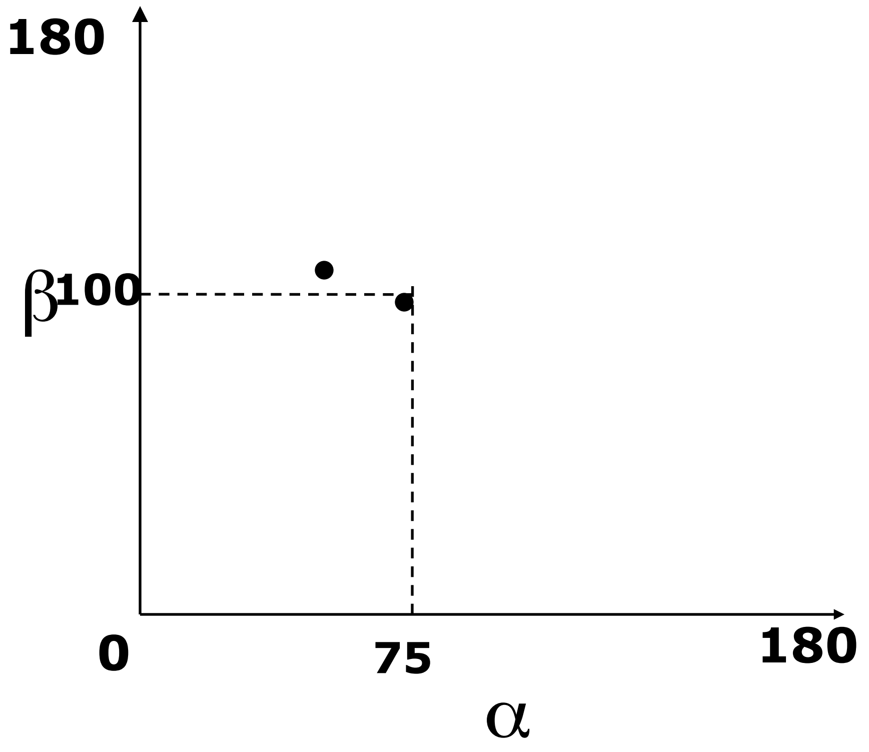
C-Space





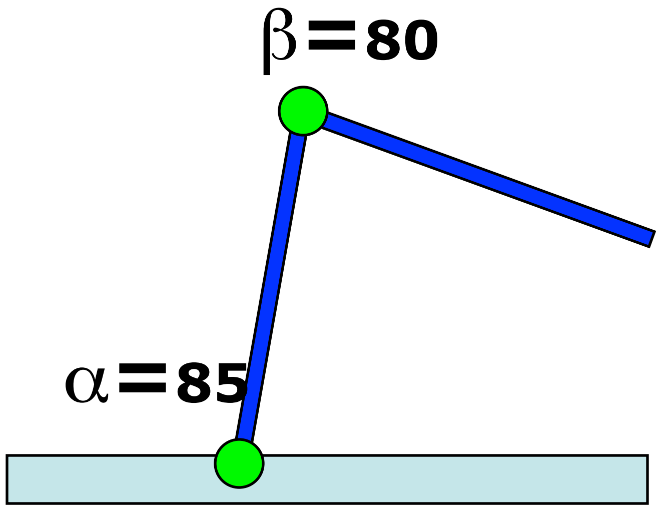
C-Space

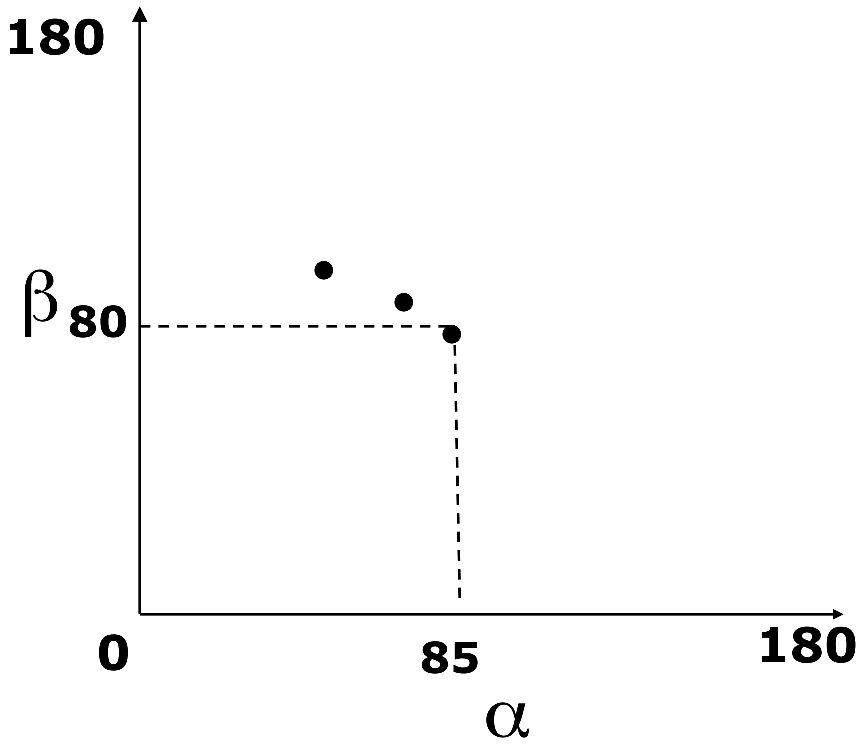




C-Space

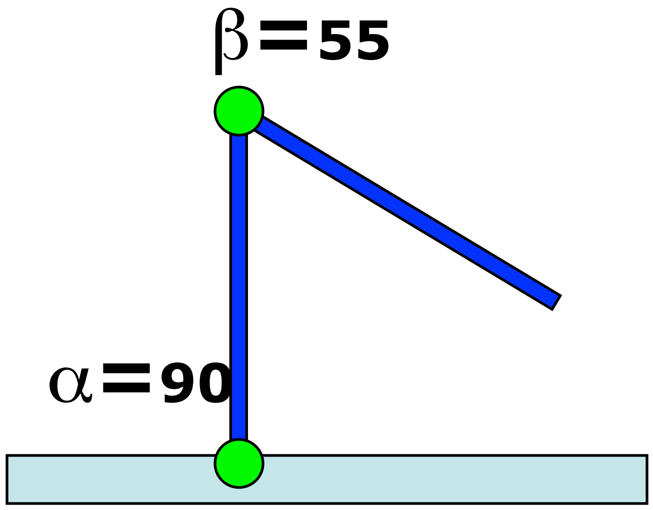
C-Space

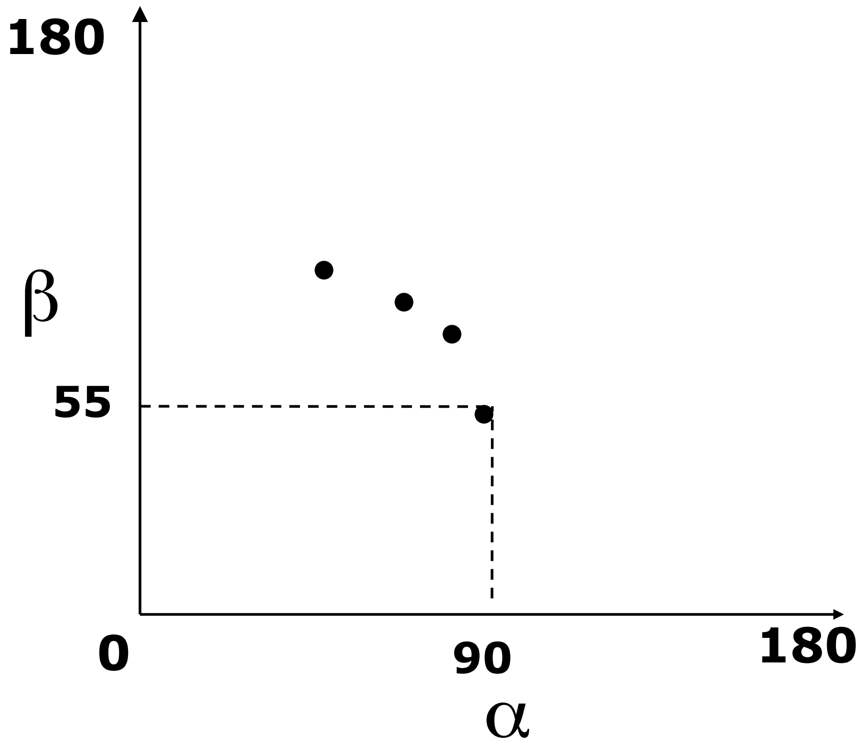




C-Space

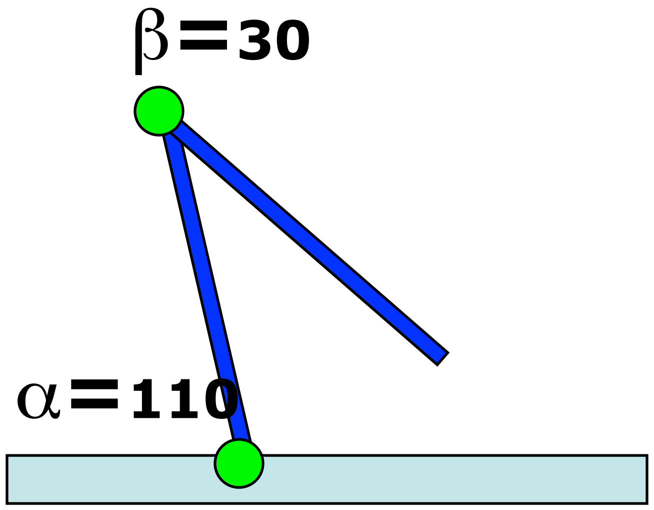
C-Space

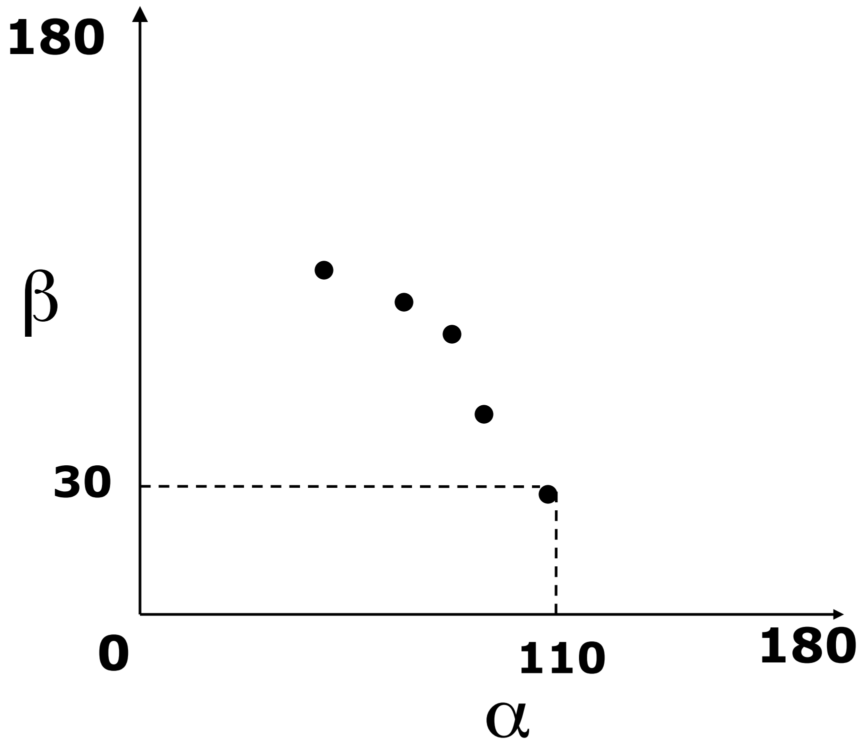




C-Space

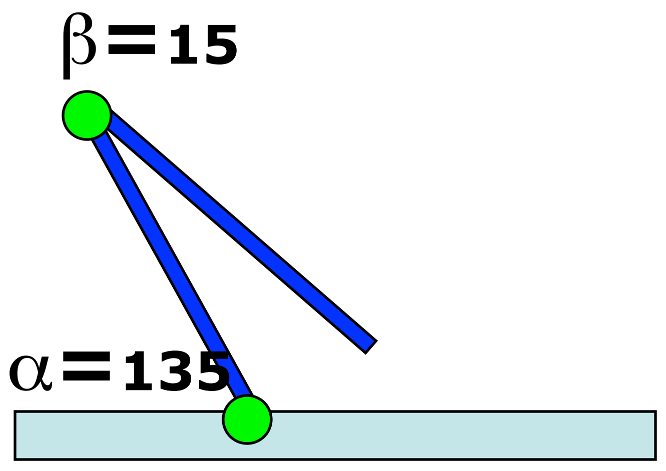
C-Space

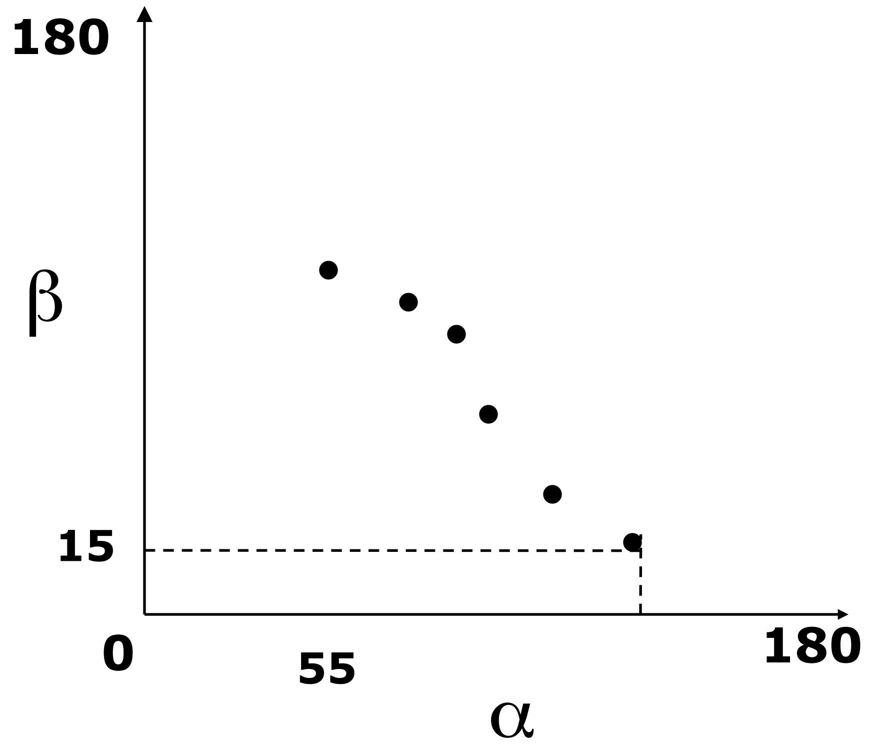




C-Space

C-Space

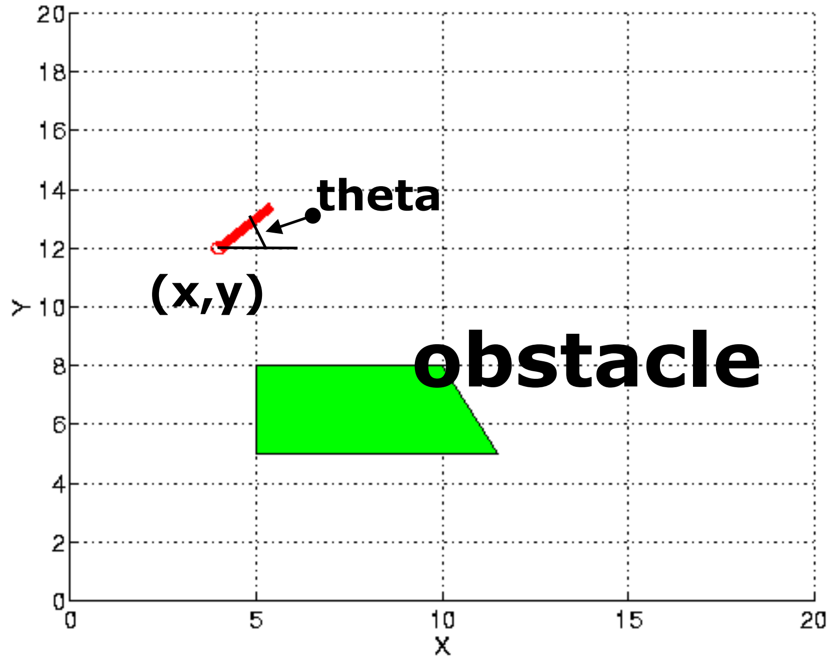




C-Space

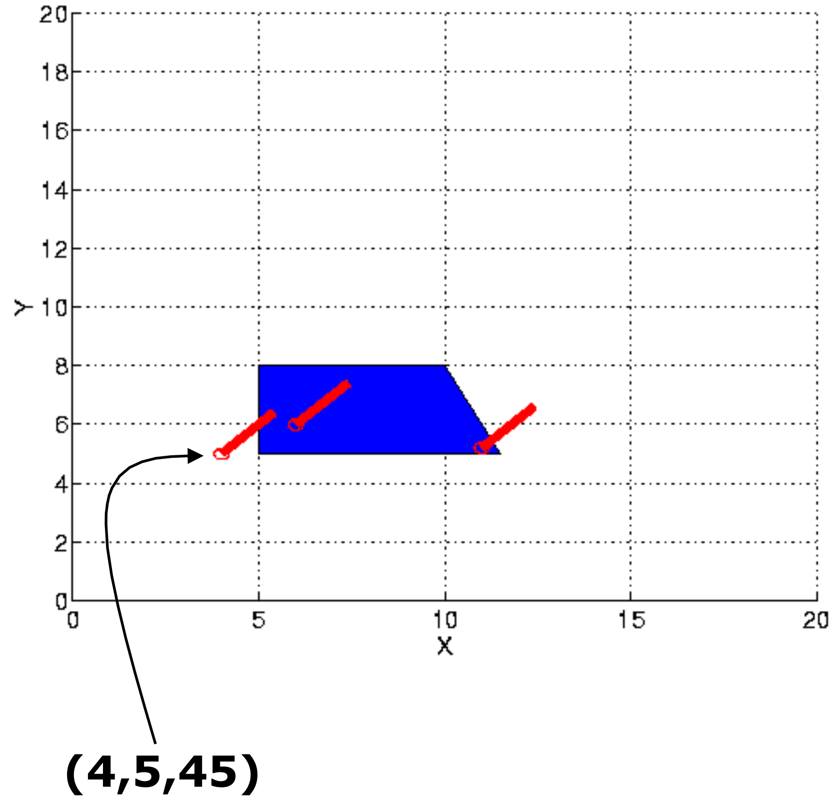
Workspace Obstacle

Workspace



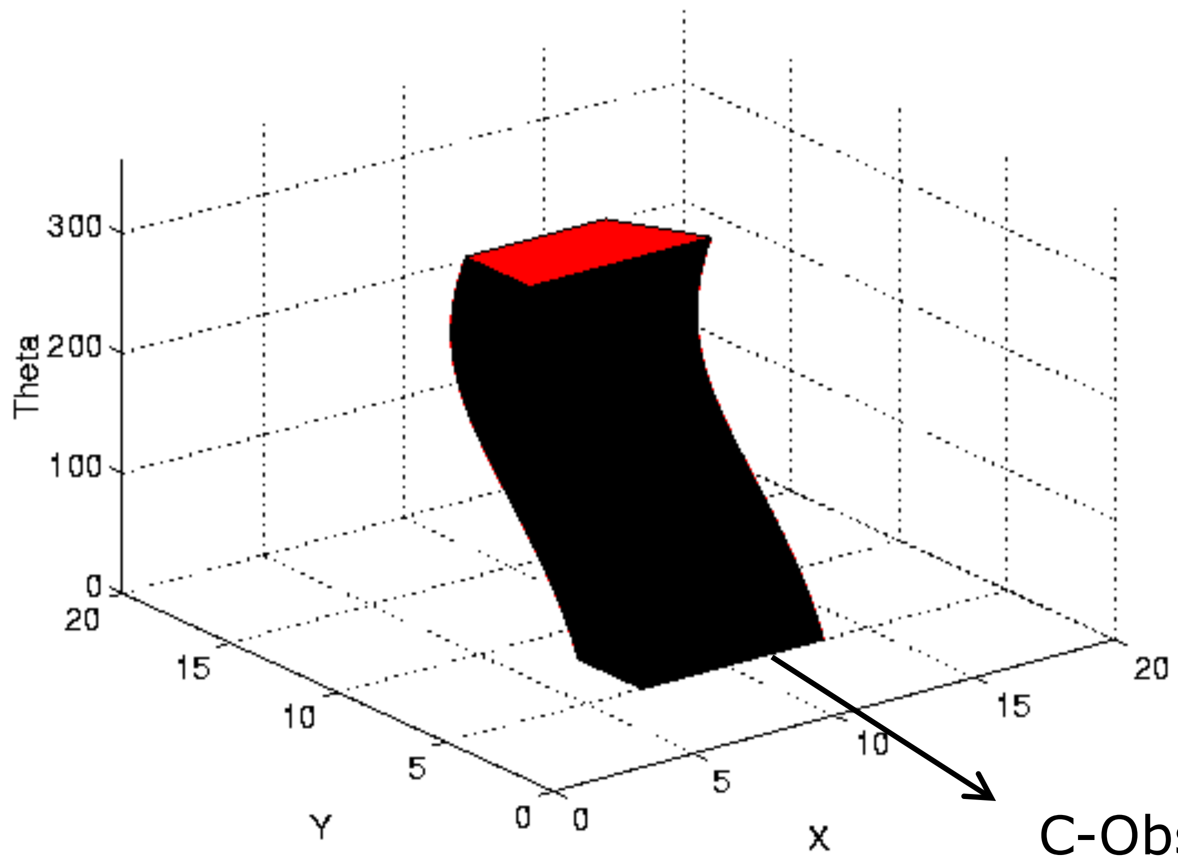
Configuration (x,y,theta)

Workspace



C-Space Obstacle

•



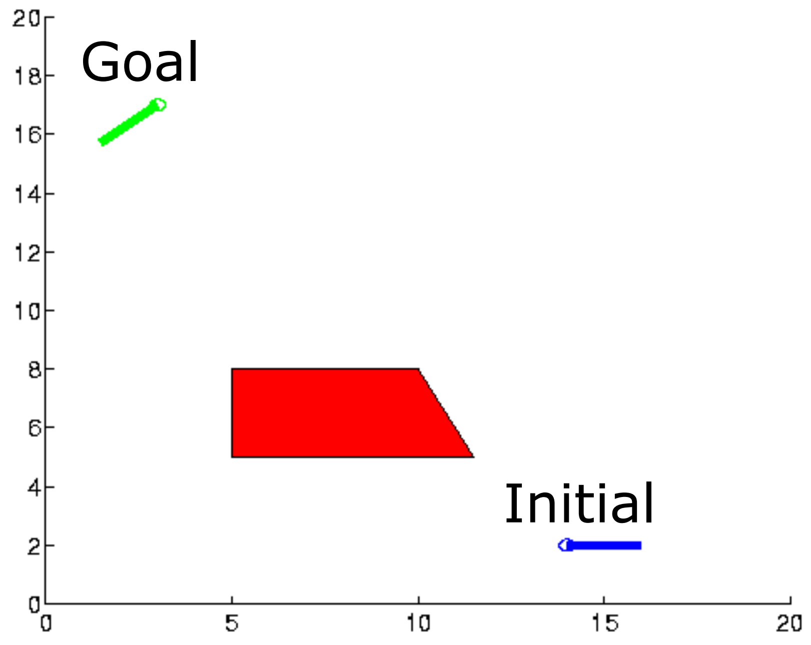
C-Obstacle

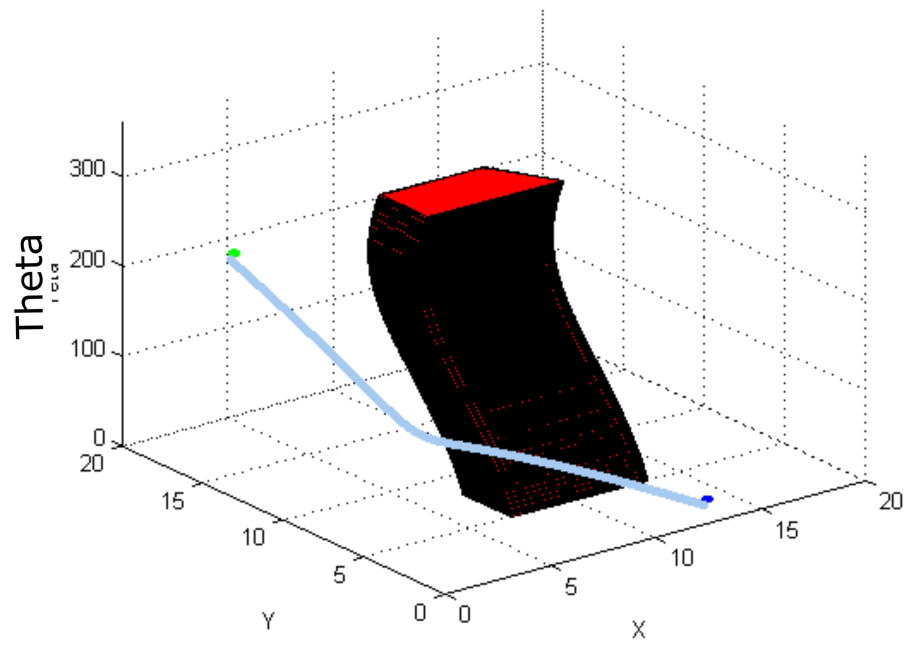
Really look like this ? Every point in the C-obst corresponds to the configuration where the robot would collide with the obstacle

Finding a Path

Find a path in workspace for a robot

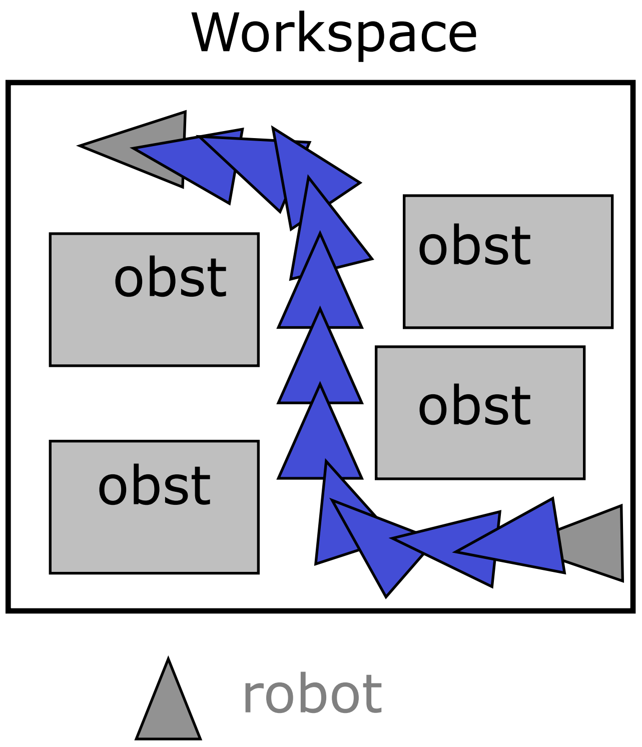
Find a path in C-space for a point

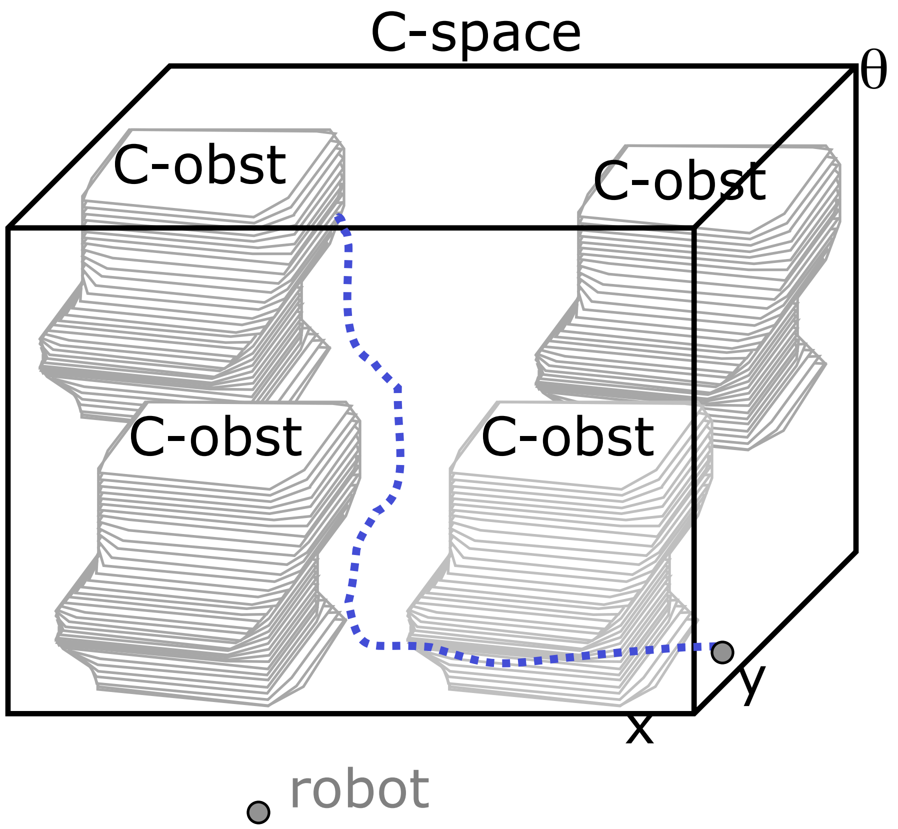




Motion Planning in C-space

Simple workspace obstacle transformed
Into complicated C-obstacle!!



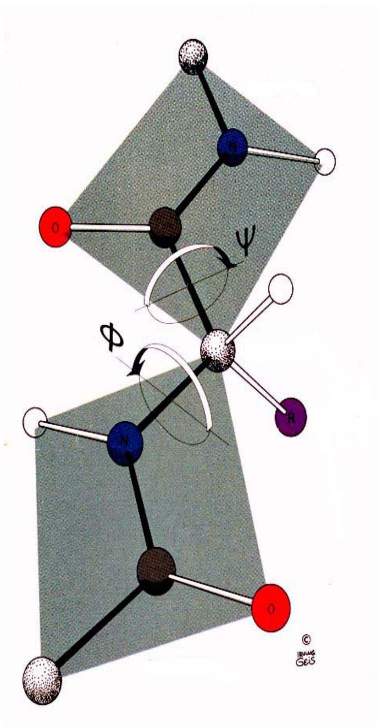


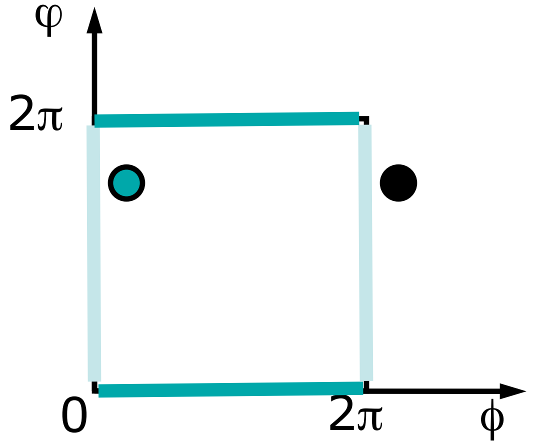
Path is swept volume

Path is 1D curve

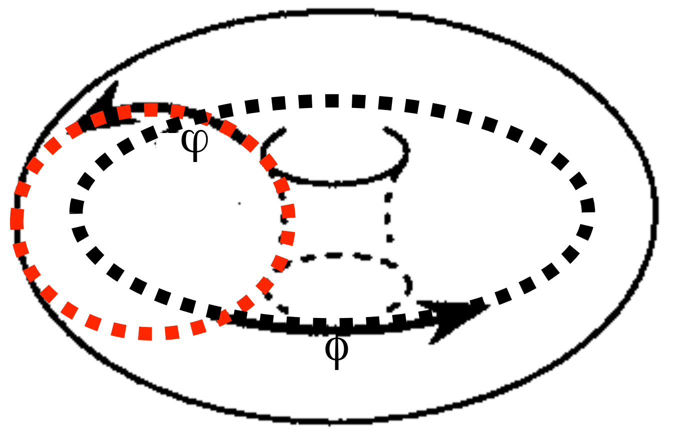
Topology of the configuration pace

- The topology of C is usually **not** that of a Cartesian space R^n.



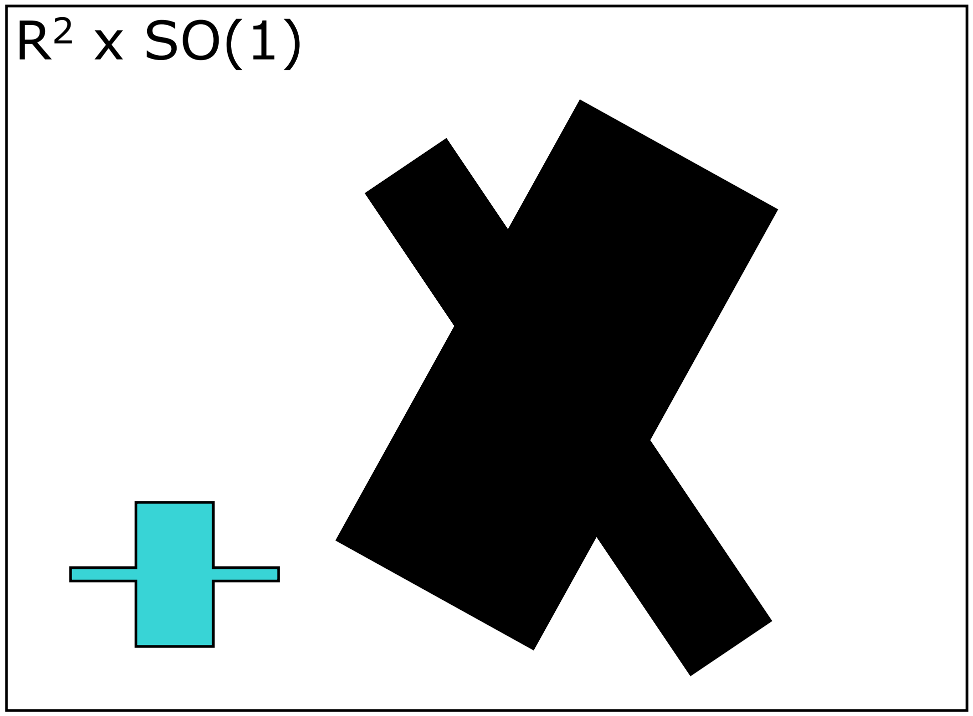


$C = S^1 \times S^1$

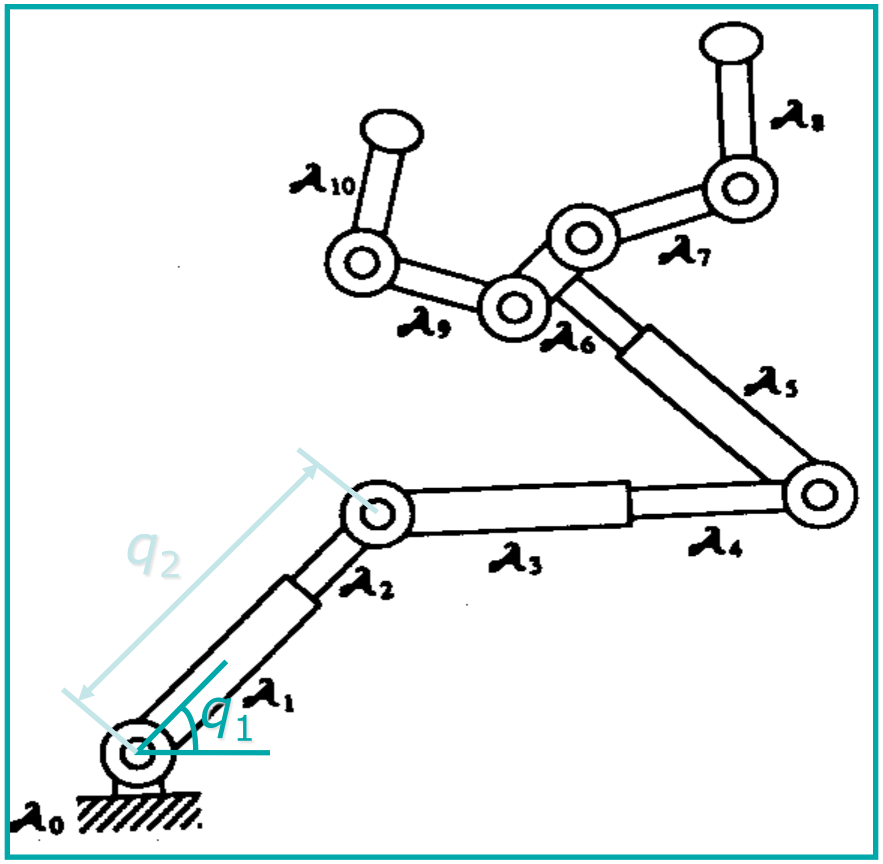


Example: rigid robot in 2-D workspace

- dim of configuration space = **???**
- Topology **???**



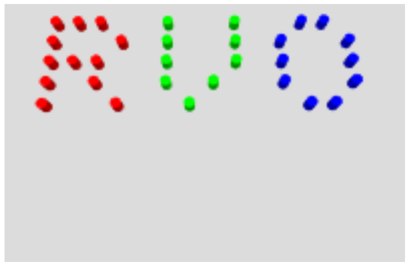
Example: articulated robot



An articulated object is a set of rigid bodies connected at the joints.

- Number of DOFs?
- What is the topology?

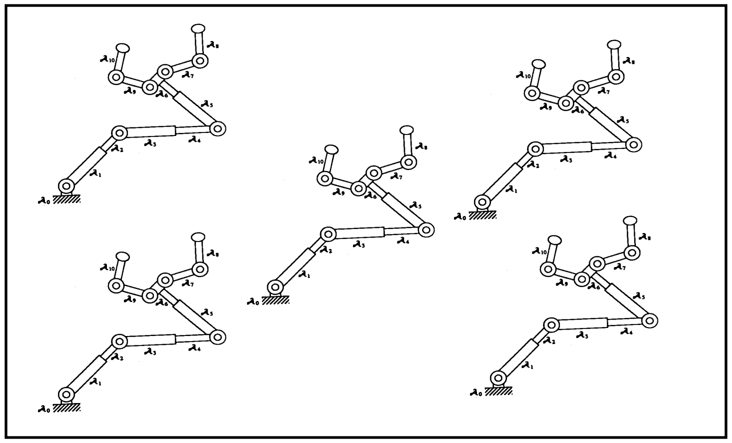
Example: Multiple robots


ROV, GAMMA group, UNC

- Given *n* robots in 2-D
- What are the possible representations?

- What is the number of dofs?

J.J. Kuffner et al.

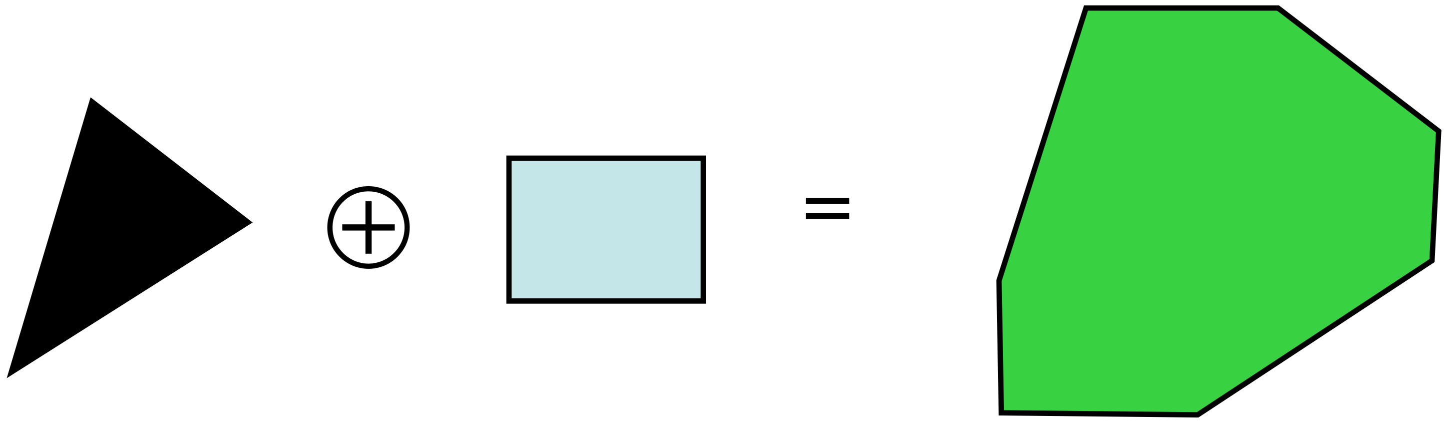

5 articulated robots

Computing C-obstacles

- For polygonal obstacles and polygonal translating robots – how to compute C-obstacles
- Minkowski sum allows us to solve problems with translational robots

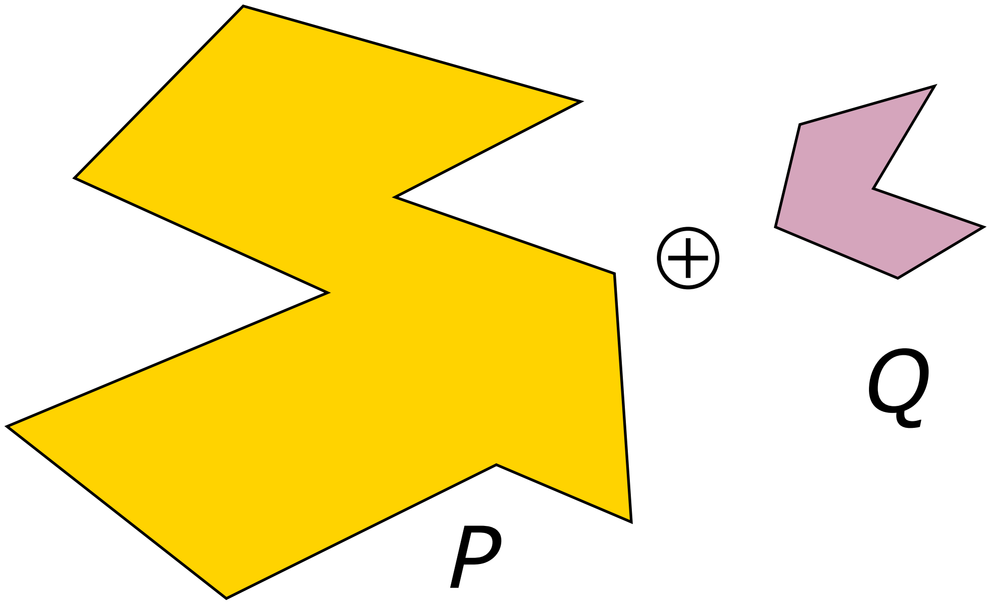
Minkowski sum of convex polygons

- The Minkowski sum of two convex polygons *P* and *Q* of *m* and *n* vertices respectively is a convex polygon *P* + *Q* of *m* + *n* vertices.
  - The vertices of *P* + *Q* are the "sums" of vertices of *P* and *Q*.



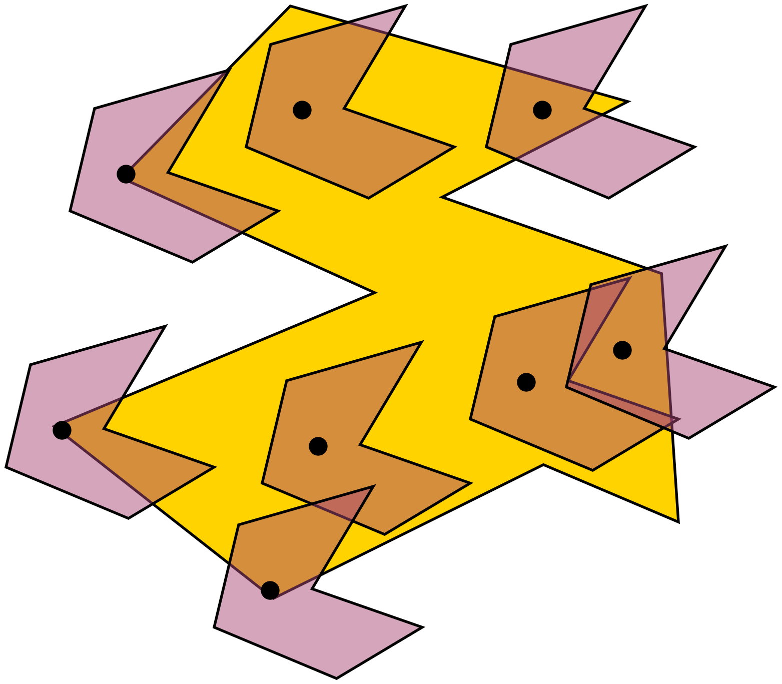
Minkowski Sum

- Minkowski sum
  - $P \oplus Q = \{p+q \mid p \in P,\ q \in Q\}$



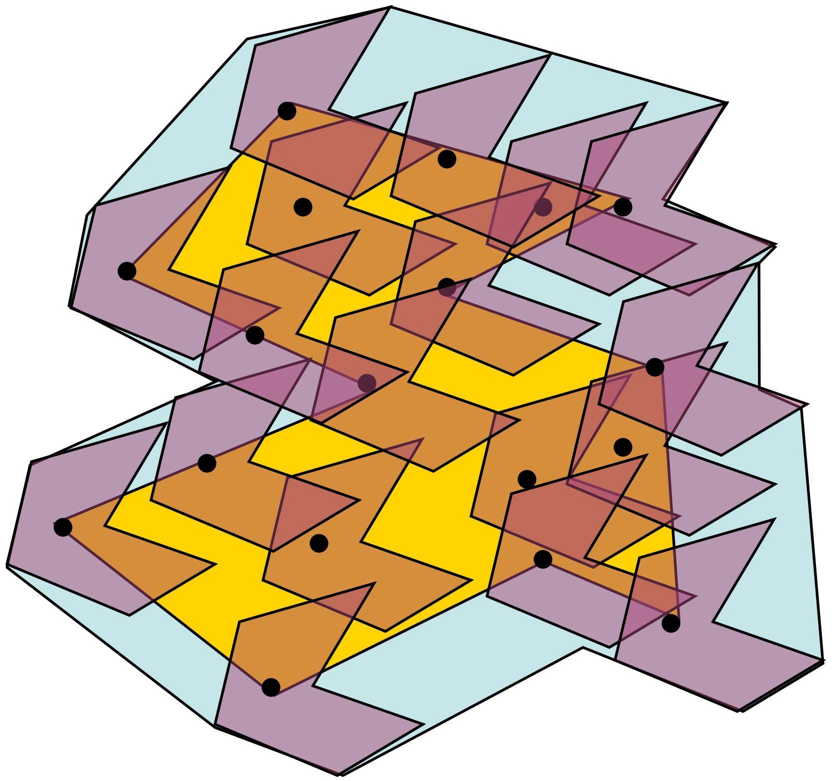
Minkowski Sum

- Minkowski sum
  - $P \oplus Q = \{p+q \mid p \in P,\ q \in Q\}$



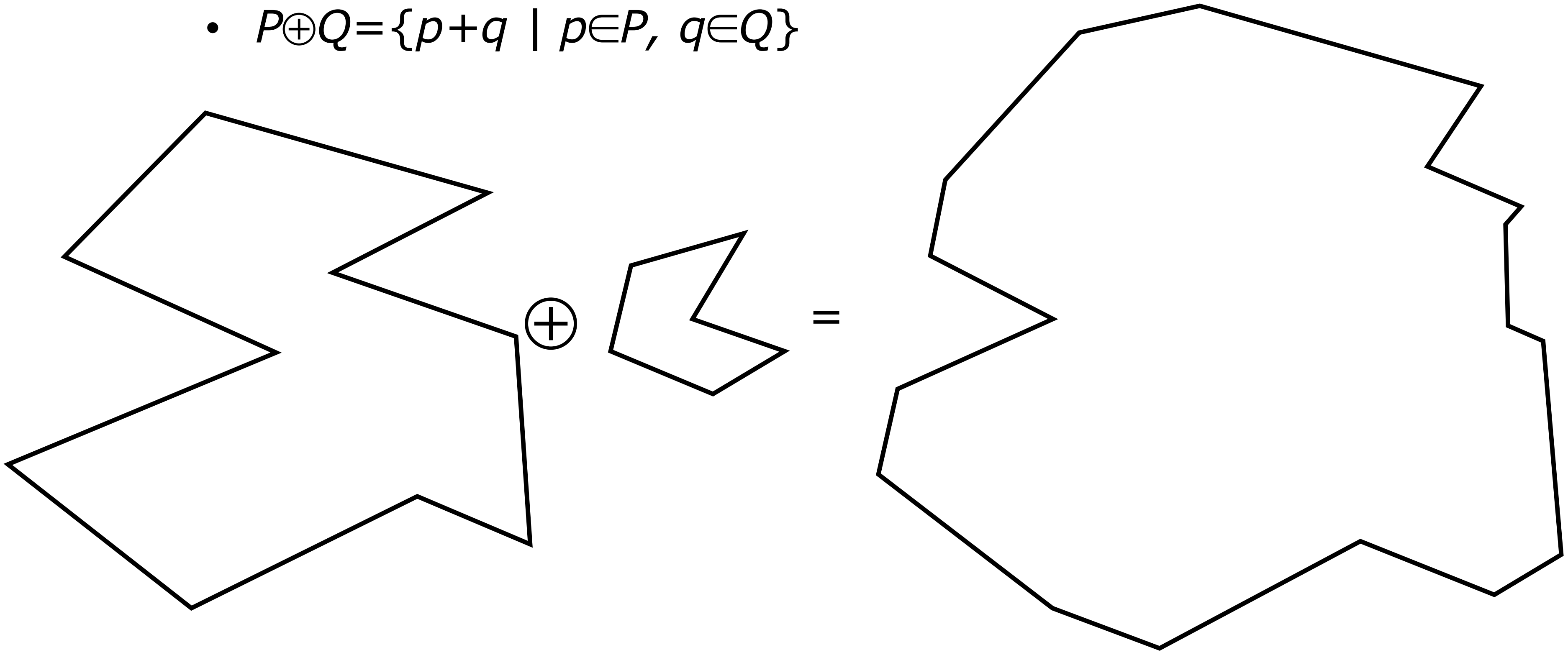
Minkowski Sum

- Minkowski sum
  - $P \oplus Q = \{p+q \mid p \in P, q \in Q\}$



Minkowski Sum

- Minkowski sum
  - $P \oplus Q = \{p+q \mid p \in P,\ q \in Q\}$



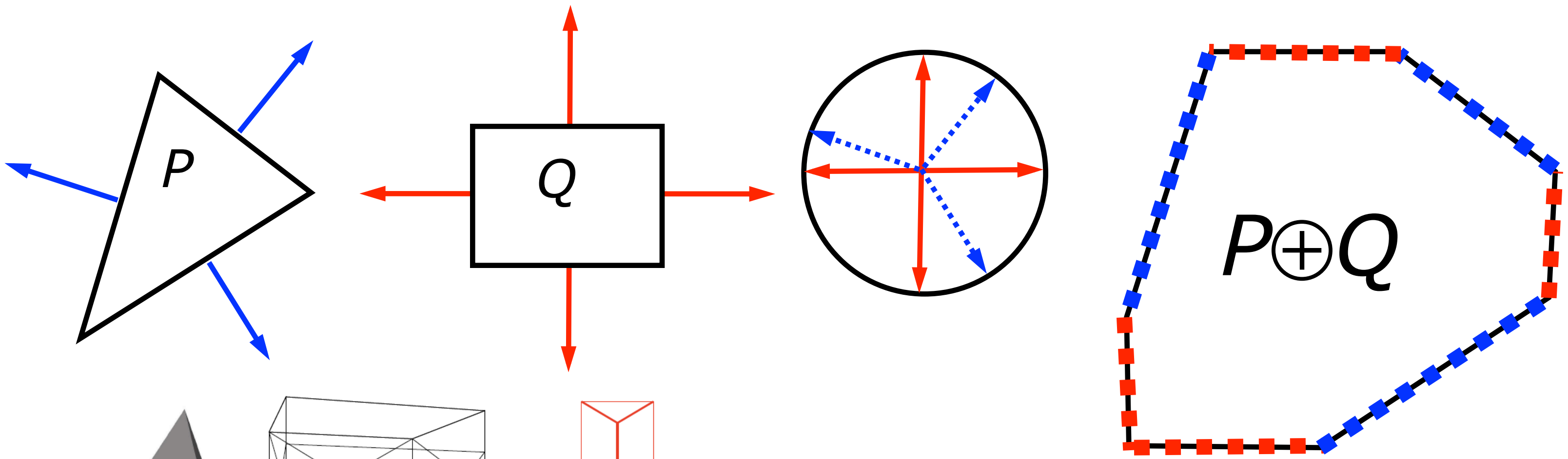
Algorithm

- Sort normals to the edges of the polygon
- Every edge of C-obst is either edge of the polygon or edge of the robot. Every edge is used exactly once, we need to determine the ordering of the edges
- Sort inward angles on the robot counterclockwise
- Sort outward angles of the obstacle normals
- Use incrementally the edges which correspond to the sorted normals in the order they are encountered

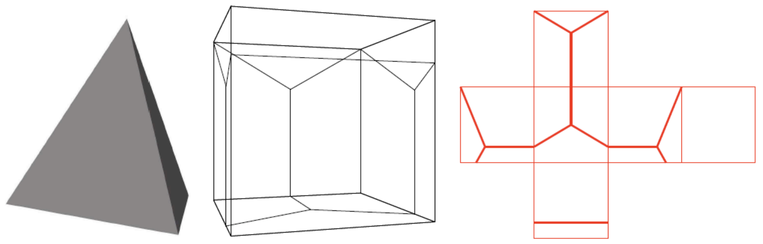
Compute Minkowski Sum

- Convex object
  - Use Gaussian map
  - Compute convex hull of Point-based Minkowski sum (slower)

2D

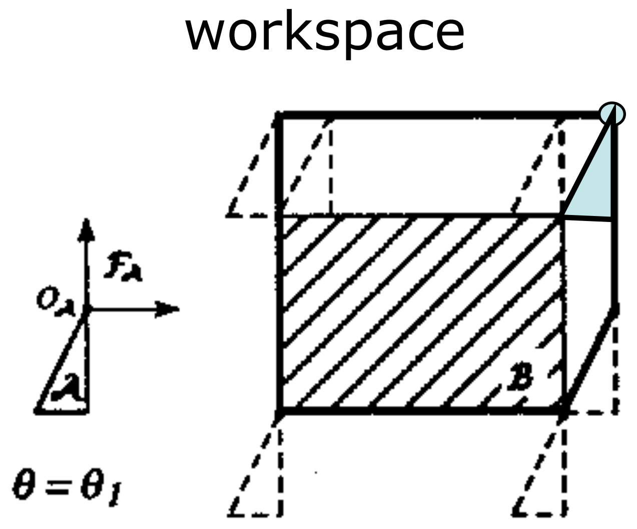


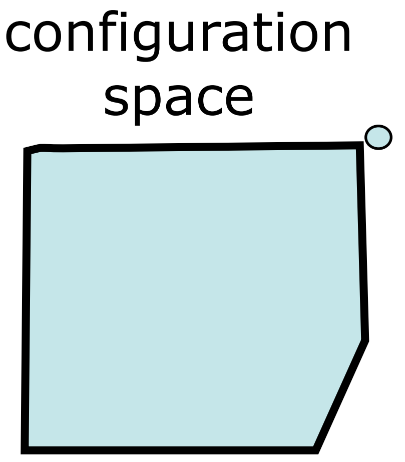
3D



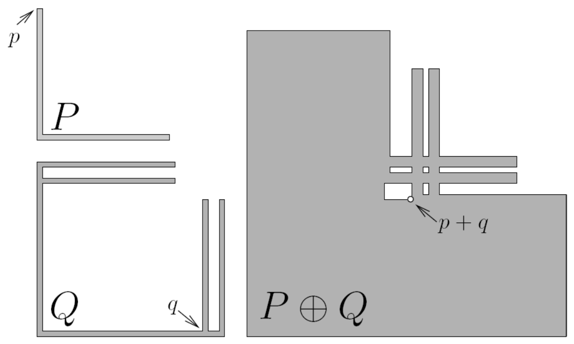
[Fogel and Halperin 06]

Polygonal robot translating in 2-D workspace



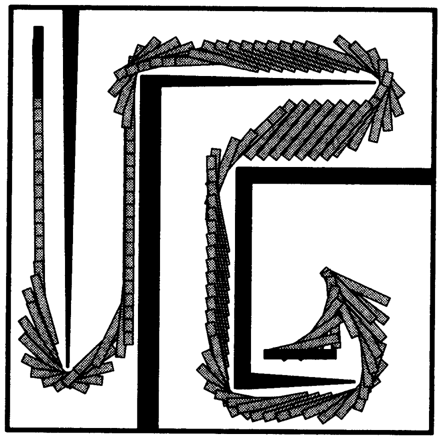


The complexity of the Minkowski sum is $O(n^2)$ in 2D

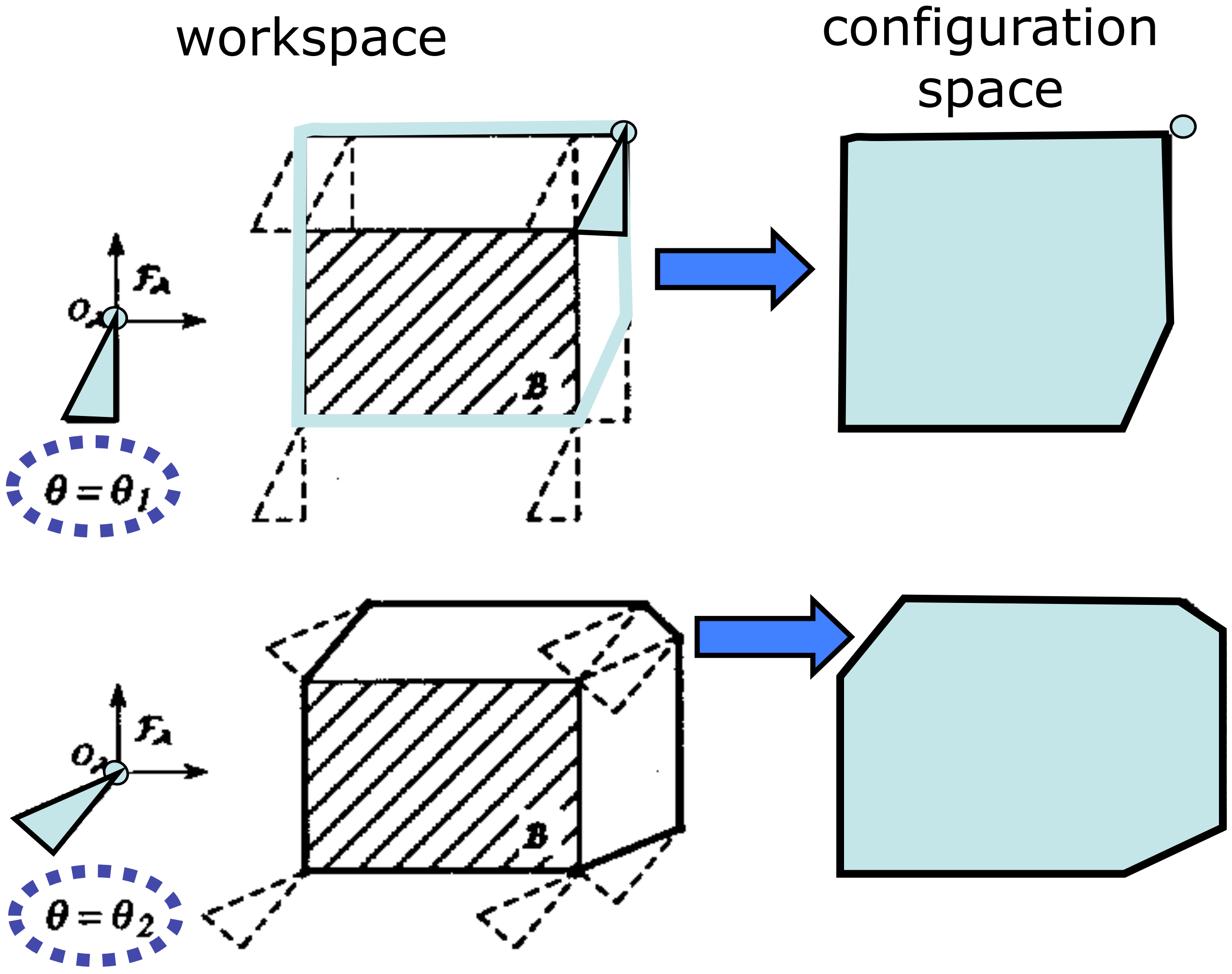


Robot with Rotations

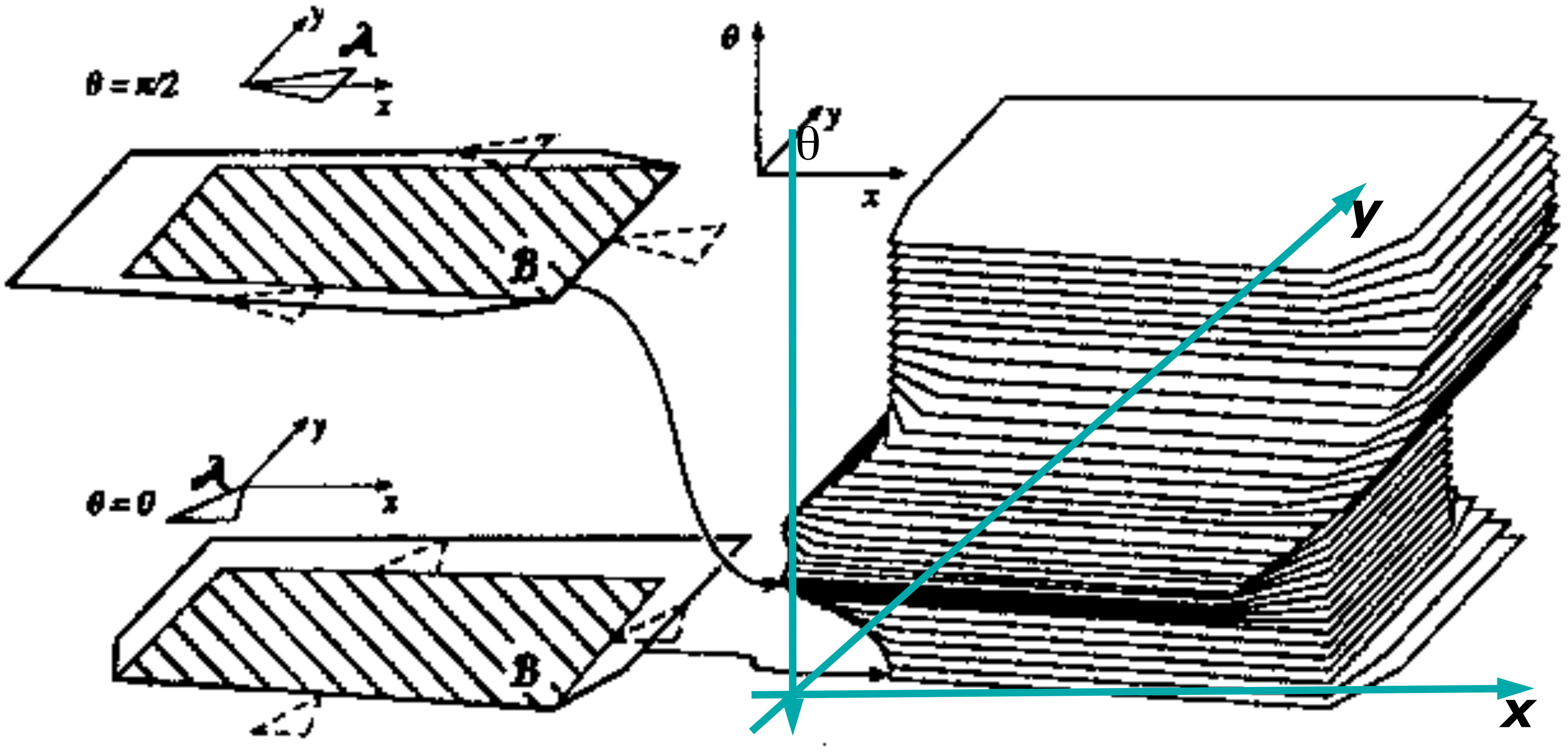
- If a robot is allowed rotation in addition to translation in 2D then it has 3 DOF
- The configuration space is 3D: (x,y,φ) where φ is in the range [0:360)



Polygonal robot translating & rotating in 2-D workspace

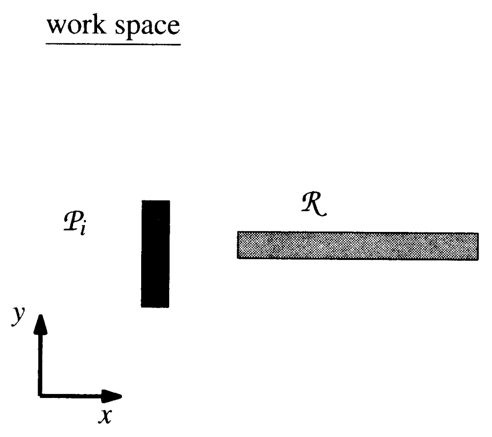


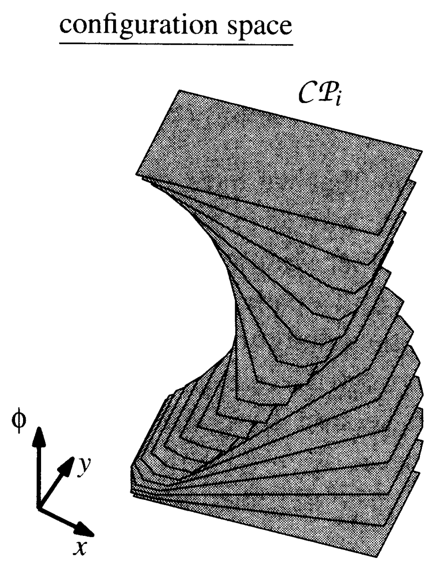
Polygonal robot translating & rotating in 2-D workspace



Mapping to C-Space

- The obstacles map to “twisted pillars” in C-Space
- They are no longer polygonal but are composed of curved faces and edges





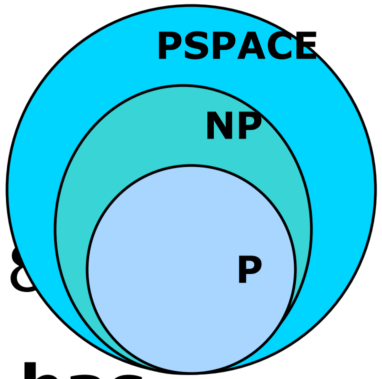
Computing Free Space

- Exact cell decomposition in 3D is really hard
- Compute z: a finite number of slices for discrete angular values
- Each slice will be the representation of the free space for a purely translational problem
- Robot will either move within a slice (translating) or between slices (rotating)

Hard Motion Planning

- Configuration Space methods – complex even for low dimensional configuration spaces
- Plus – always guarantee finding a plan if it exists in finite time (or answer no)
- Idea behind sampling cased motion planning – sacrifice completeness for efficiency – weaker guarantee – notion on probabilistic completeness

The Complexity of Motion Planning



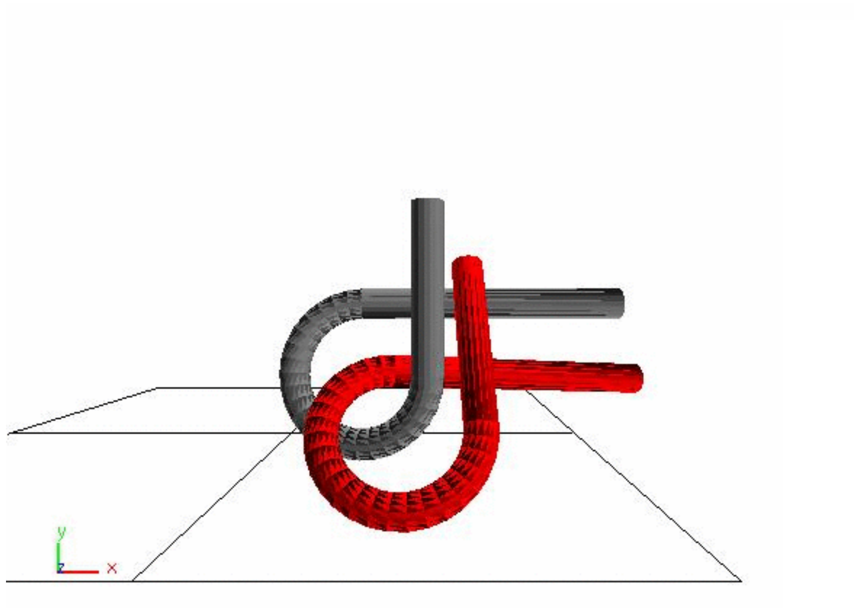
- General motion planning problem is
- PSPACE-hard [Reif 79, Hopcroft et al. 84 & 8
- PSPACE-complete [Canny 87]

The best deterministic algorithm known has running time that is exponential in the dimension of the robot's C-space [Canny 86]

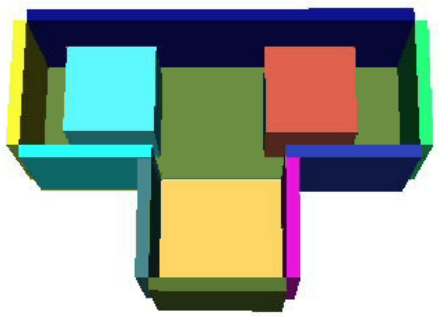
- C-space has high dimension - 6D for rigid body in 3D space
- simple obstacles have complex C-obstacles impractical to
  compute explicit representation of free space for more
  than 4 or 5 dof

Hard Motion Planning Problems

The Alpha Puzzle



Swapping Cubes Puzzle



- Separate two shapes (one considered robot) – another obstacle
- Exchange the positions of two cubes (one needs move to empty space)
- All these planning problems are considered in continuous spaces

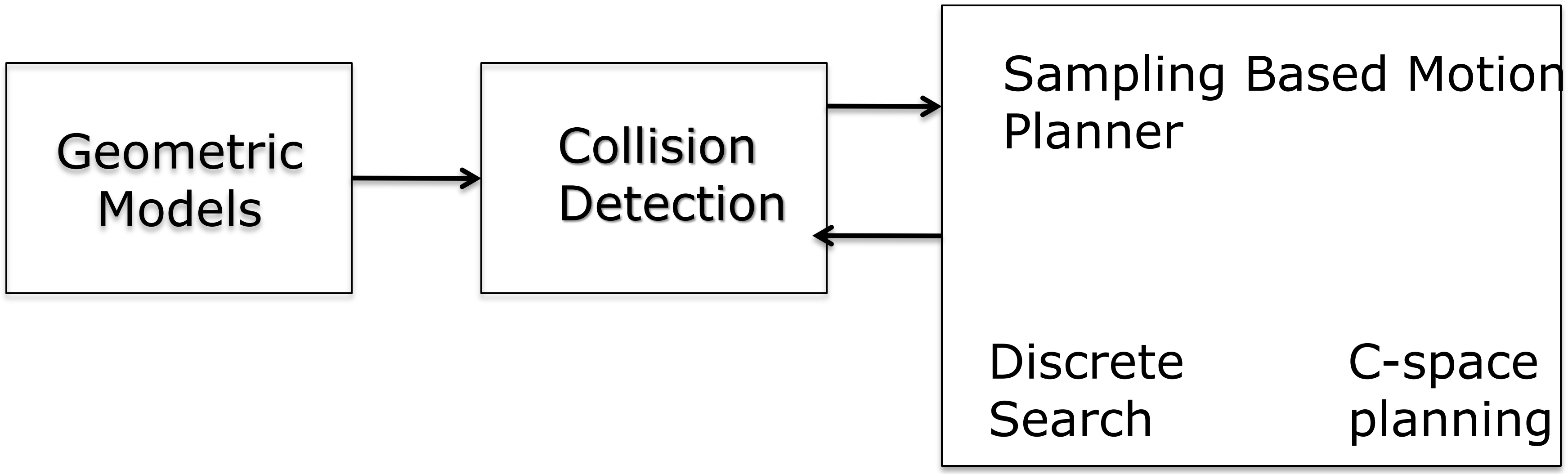
Probabilistic Methods

- Resort to sampling based methods
- Avoid computing C-obstacles
  - Too difficult to compute efficiently

- Idea: Sacrifice completeness to gain simplicity and efficiency

- Probabilistic Methods
  - Graph based
  - Tree based

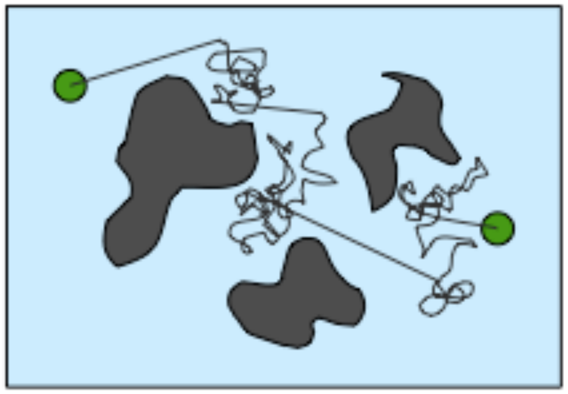
Sampling Based Motion Planning



Idea : Generate random configurations
Check whether they are collision free
Connect them using Local planners

Probabilistic Motion Planning

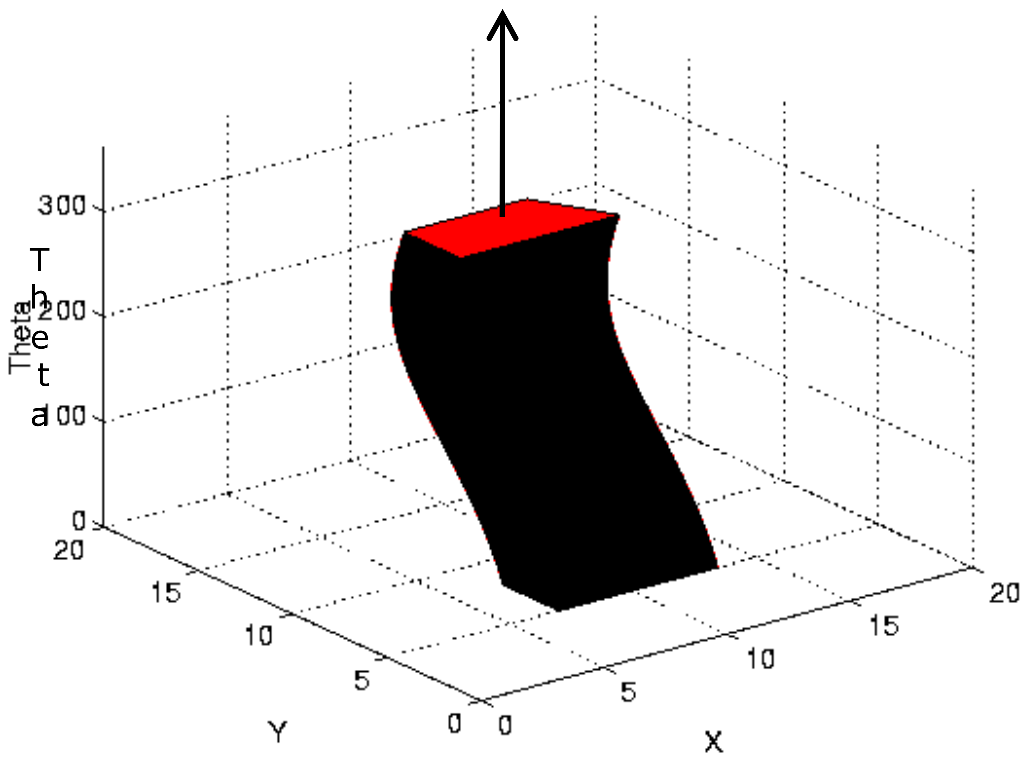
- First encounter with randomized techniques – in the context of potential field based methods
- Use random walk to escape local minima (May take long time)
- Idea – potential function gives as a cost to go g(q)
- If local planner is not successful in reducing the cost to go
- Switch to random walk mode from current node, terminate if node with lower g(q) is found or number of iterations have been reached
- If better node has not been found back-track – pick one of the nodes enc walk and restart best first search



Probabilistic Roadmap Method [Kavraki, Svestka, Latombe,Overmars 1996]

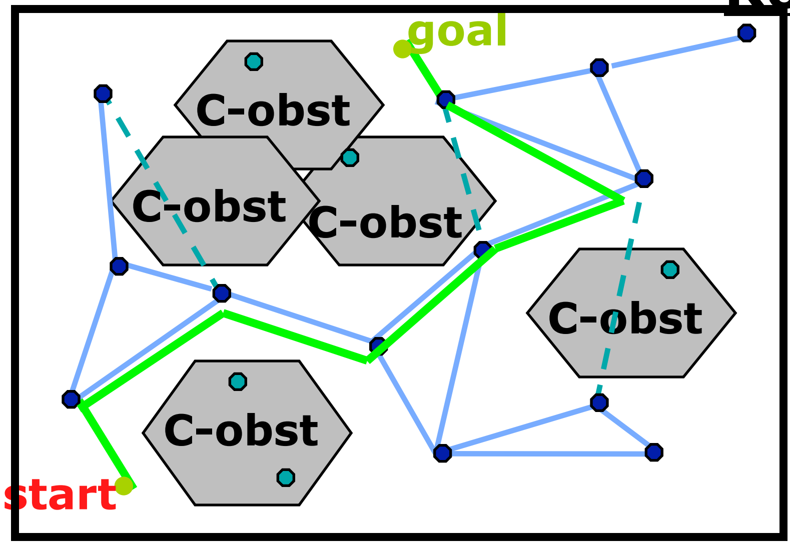
Explicit representation of the configuration space is unknown

- Theta



Probabilistic Roadmap Method

C-space



Roadmap Construction (Pre-processing)

1. Randomly generate robot configurations (nodes)
   - discard nodes that are invalid
2. Connect pairs of nodes to form **roadmap**
   - simple, deterministic *local planner* (e.g., straightline)
   - discard paths that are invalid

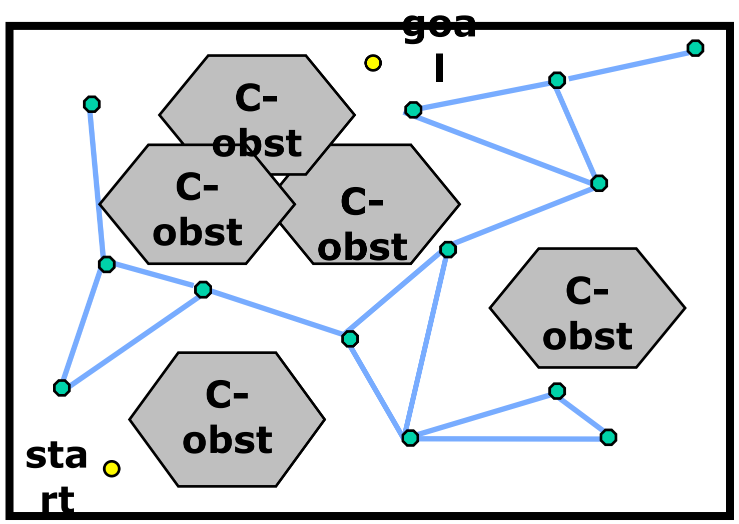
Query processing

1. Connect *start* and *goal* to roadmap
2. Find path in roadmap between *start* and *goal*
   - regenerate plans for edges in roadmap

Probabilistic Roadmap Method

- Important sub-routines
  - Generate random configurations
  - Local planners
  - Distance metrics
  - Selecting k-nearest neighbors (becoming dominant in high dimensional space)
  - Collision detection (>80% computation)

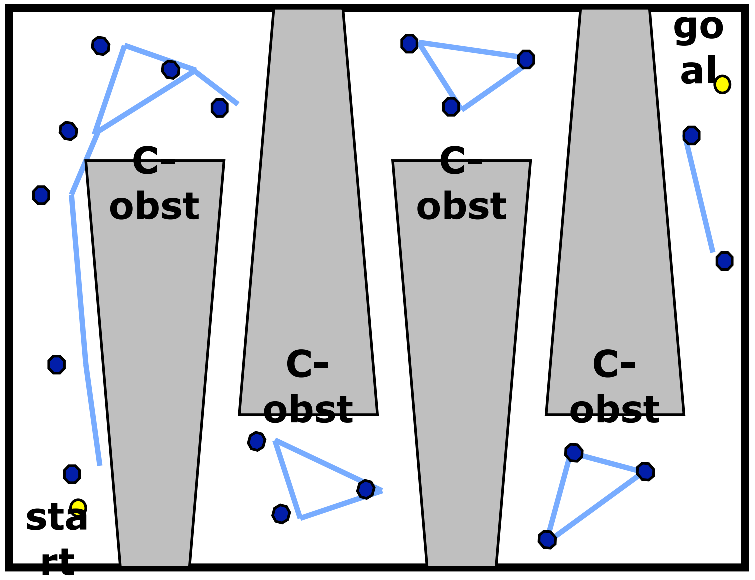
PRMs: Pros & Cons



PRMs: The Good News

1. PRMs are *probabilistically complete*
2. PRMs apply easily to high-dimensional C-space
3. PRMs support fast queries w/ enough preprocessing

Many success stories where PRMs solve previously unsolved problems



PRMs: The Bad News

1. PRMs don’t work as well for some problems:
– unlikely to sample nodes in *narrow passages*
– hard to sample/connect nodes on constraint surfaces

Related Work (selected)

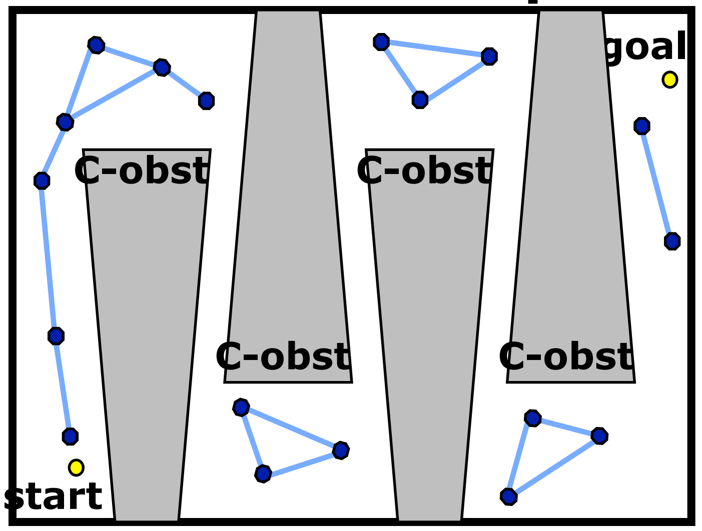
- **Probabilistic Roadmap Methods**
- Uniform Sampling (original) [Kavraki, Latombe, Overmars, Svestka,
  92, 94, 96]
- Obstacle-based PRM (OBPRM) [Amato et al, 98]
- PRM Roadmaps in Dilated Free space [Hsu et al, 98]
- Gaussian Sampling PRMs [Boor/Overmars/van der Steppen 99]
- Bridge test [Hsu et al 03]
- Visibility Roadmaps [Laumond et al 99]
- Using Medial Axis [Kavraki et al 99, Lien/Thomas/Wilmarth/Amato/
  Stiller 99, 03, Lin et al 00]
- Generating Contact Configurations [Xiao et al 99]
- Using workspace clues

An Obstacle-Based PRM

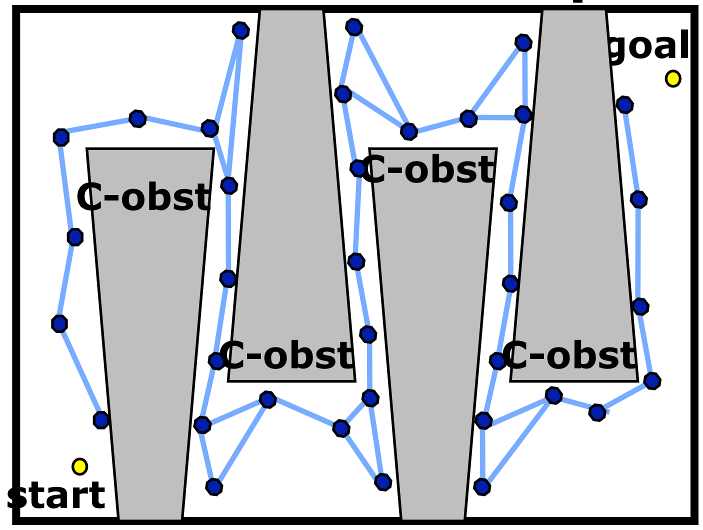
To Navigate Narrow Passages we must sample in them

• most PRM nodes are where planning is easy (not needed)

PRM Roadmap



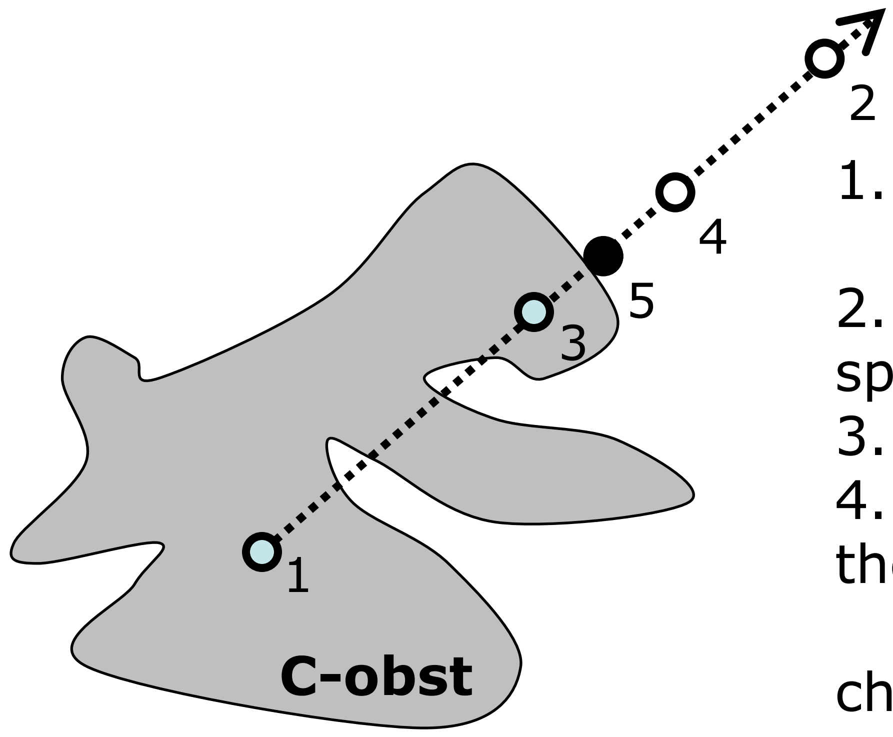
OBPRM Roadmap



Idea: Can we sample nodes near C-obstacle surfaces?

- we cannot explicitly construct the C-obstacles...
- we do have models of the (workspace) obstacles...

Finding Points on C-obstacles

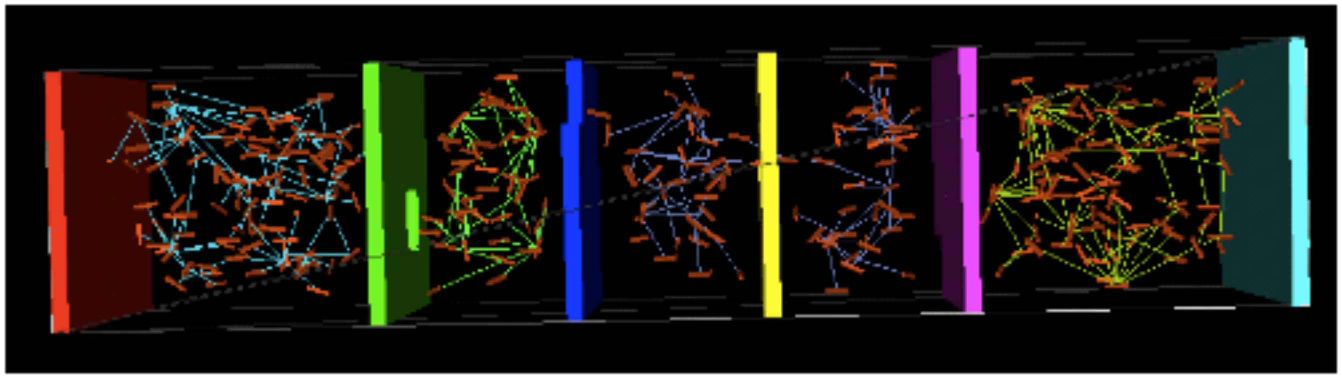


Basic Idea (for workspace obstacle S)

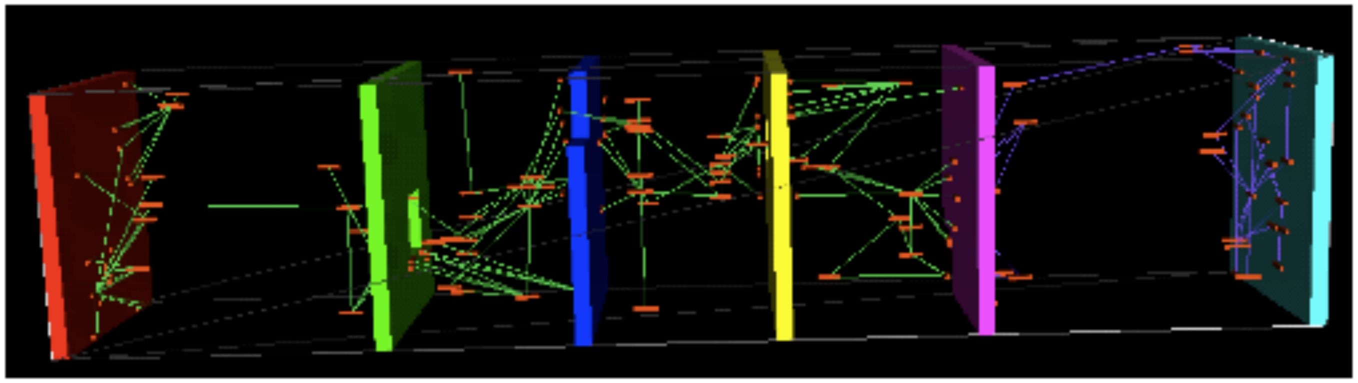
1. Find a point in S's C-obstacle
   (robot placement colliding with S)
2. Select a random direction in C-space
3. Find a free point in that direction
4. Find boundary point between them
   using binary search (collision checks)

Note: we can use more sophisticated heuristics to try to cover C-obstacle

OBPRM

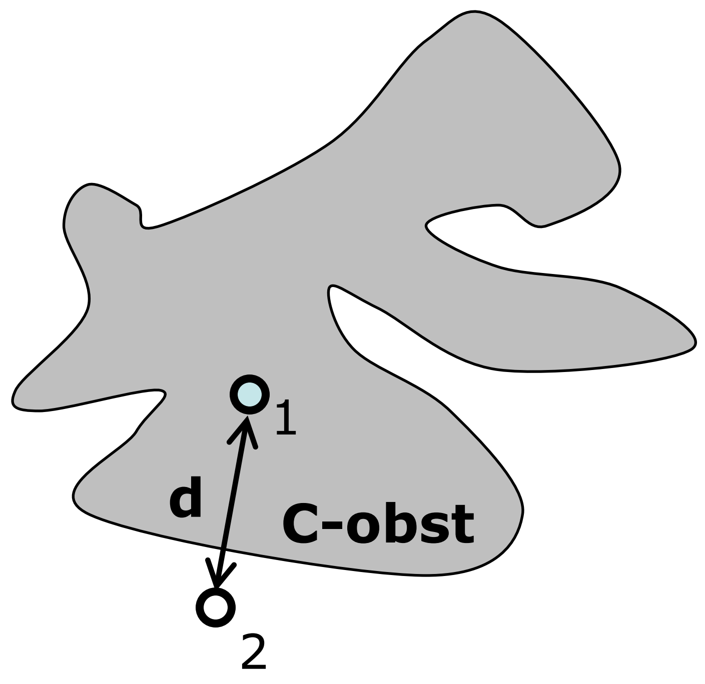


PRM
- 328 nodes
- 4 major CCs



OBPRM
- 161 nodes
- 2 major CCs

Gaussian Sampling PRM



1. Find a point in S's C-obstacle
   (robot placement colliding with S)

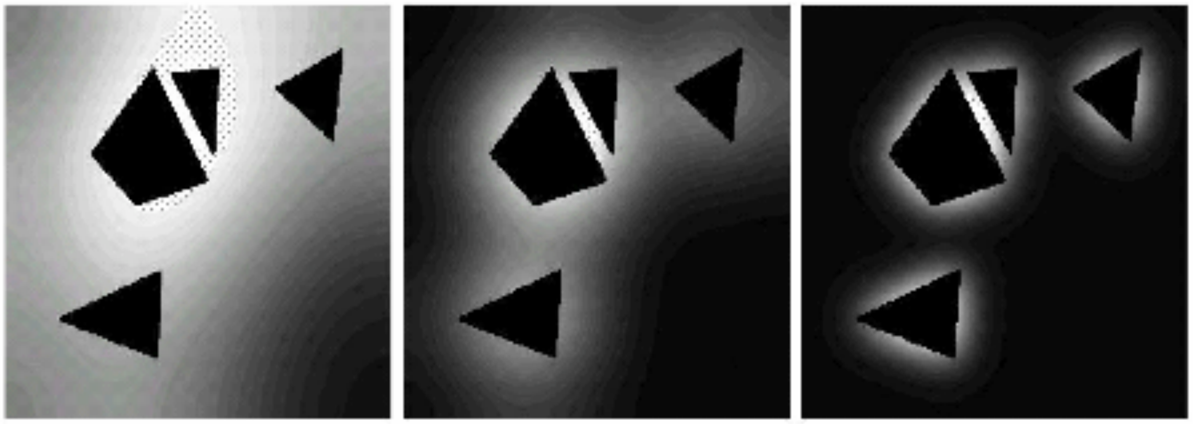
2. Find another point that is within distance *d* to the first point, where d is a random variable in a *Gaussian distribution*

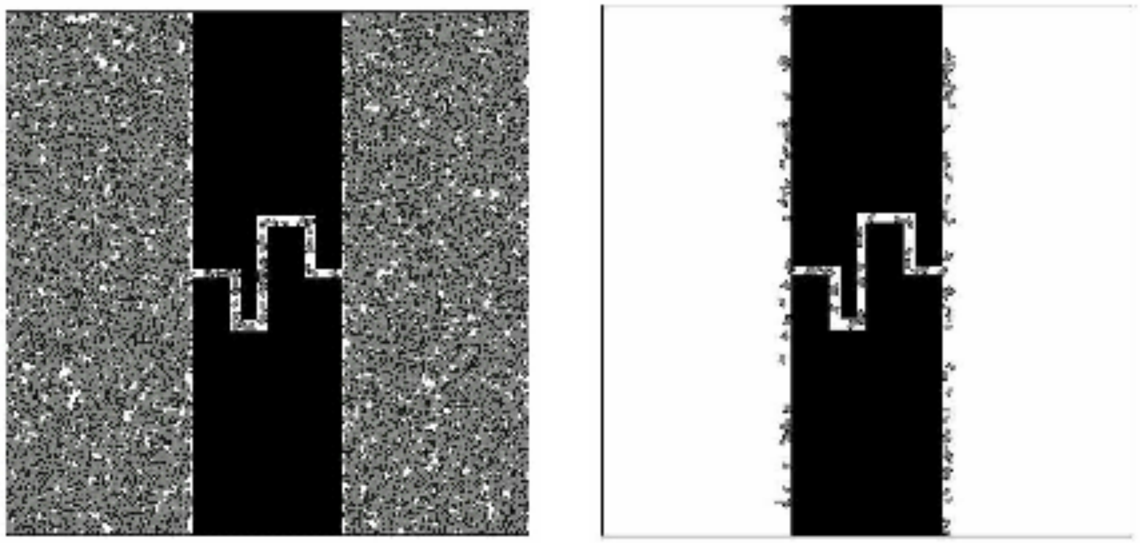
3. Keep the second point if it is collision free

Note

- Two paradigms: (1) OBPRM: Fix the samples (2) Gaussian PRM: Filter the samples
- None of these methods can (be proved to) provide guarantee that the samples in the narrow passage will increase!

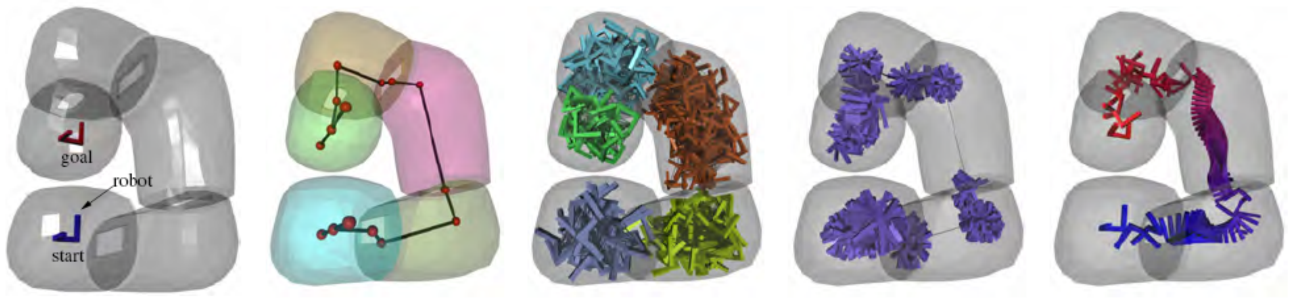
Gaussians





Related Work (selected)

- **Probabilistic Roadmap Methods**
- Uniform Sampling (original) [Kavraki, Latombe, Overmars, Svestka, 92, 94, 96]
- Obstacle-based PRM (OBPRM) [Amato et al, 98]
- PRM Roadmaps in Dilated Free space [Hsu et al, 98]
- Gaussian Sampling PRMs [Boor/Overmars/van der Steppen 99]
- Bridge test [Hsu et al 03]
- Visibility Roadmaps [Laumond et al 99]
- Using Medial Axis [Kavraki et al 99, Lien/Thomas/Wilmarth/Amato/Stiller 99, 03, Lin et al 00]
- Generating Contact Configurations [Xiao et al 99]
- Using workspace clues



Issues

- How do we determine a random free configuration
- We would like to sample nodes uniformly from C_free
- Draw each of the coordinates from the interval of corresponding DOF – use uniform probability per interval
- For each sample check for collision between the robot and obstacles and robot itself
- If collision free add to V otherwise discard

- Collision detection and sampling – large topics

Collision Detection

- Treated as black box - takes most of the computation
- In 2D convex robot and obstacle, there exist linear time collision detection algorithms
- For more complex non-convex bodies – hierarchical methods (create bounded regions – to avoid checking bodies which are far apart

  - have a quick way of computing whether two regions intersect (Bound. Regions: spheres, axis aligned boxes), the composite bounding regions are represented by trees
- Check for free collision free configuration
- Check for free collision free path segment
- Consider that the path is a straight line, parametrized by [0,1], sample the interval and check each sample whether its collision free
- There exist algorithms with guarantees trickier to implement

Planning in high dimensional spaces

- Single query planning (greedy technique can take a long time)

- Multiple query planning – spreads out uniformly, requires lot of samples to cover the space

- Next incremental sampling and search methods that yields good performance without parameters tunning. Idea gradually construct search tree, such that it densely covers the space

Incremental Sampling and Searching

- Single query model – given start and goal q find a path
- Analogy with the discrete search algorithms
- Samples are states, edges are paths connected them (as opposed to actions previously)
- Graphs are undirected; Ingredients

1. Initialize the graph
2. Select vertex for expansion
3. Generate set of new vertices
4. For some new vertices run a local planner and check whether its collision free
5. If yes insert an edge to the graph
6. Keep on going until termination condition is satisfied

Incremental Search and Sample

- Why not just discretizing configuration space ?
- For high dimensions large number of states can be wasted exploring various cavities of the C-space
- For low dim spaces grid points them selves can serve as roadmap points (need to be checked for collisions etc)

- How to choose a resolution of the discretization (start coarse , iteratively refine)
- Another option – abandon discretization and work with continuous problem (like randomized potential fields) or RRT’s

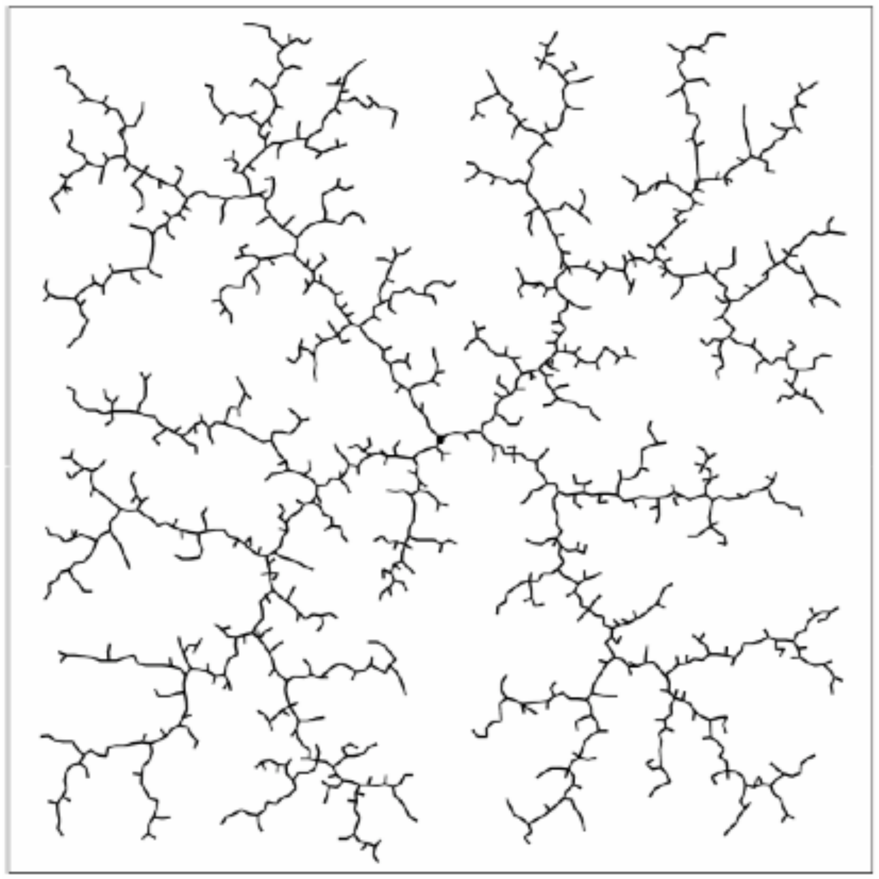
Rapidly-Exploring Random Tree (RRT)

- Tree Based single shot planners – compute the respresentation of C-free for single start and goal

- RRTs: Rapidly-exploring Random Trees
- **Rapidly-exploring random trees: Progress and prospects**. S. M. LaValle and J. J. Kuffner. In *Proceedings Workshop on the Algorithmic Foundations of Robotics*, 2000.)
  - Incrementally builds the roadmap tree

- Extends to more advanced planning techniques
  - Integrates the control inputs to ensure that the kinodynamic constraints are satisfied

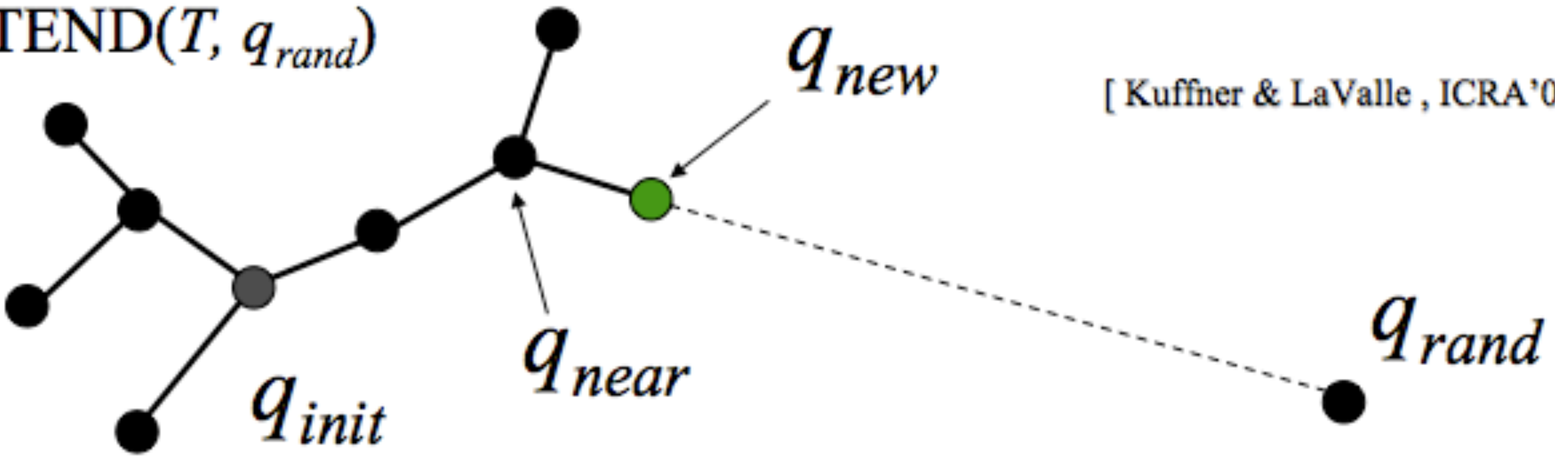
Rapidly-Exploring Random Trees



RRT's

```
BUILD_RRT ($q_{init}$) {
    $T.init(q_{init})$;
    for $k$ = 1 to K do
        $q_{rand}$ = RANDOM_CONFIG();
        EXTEND($T$, $q_{rand}$)
}
```

EXTEND(T, q_{rand})



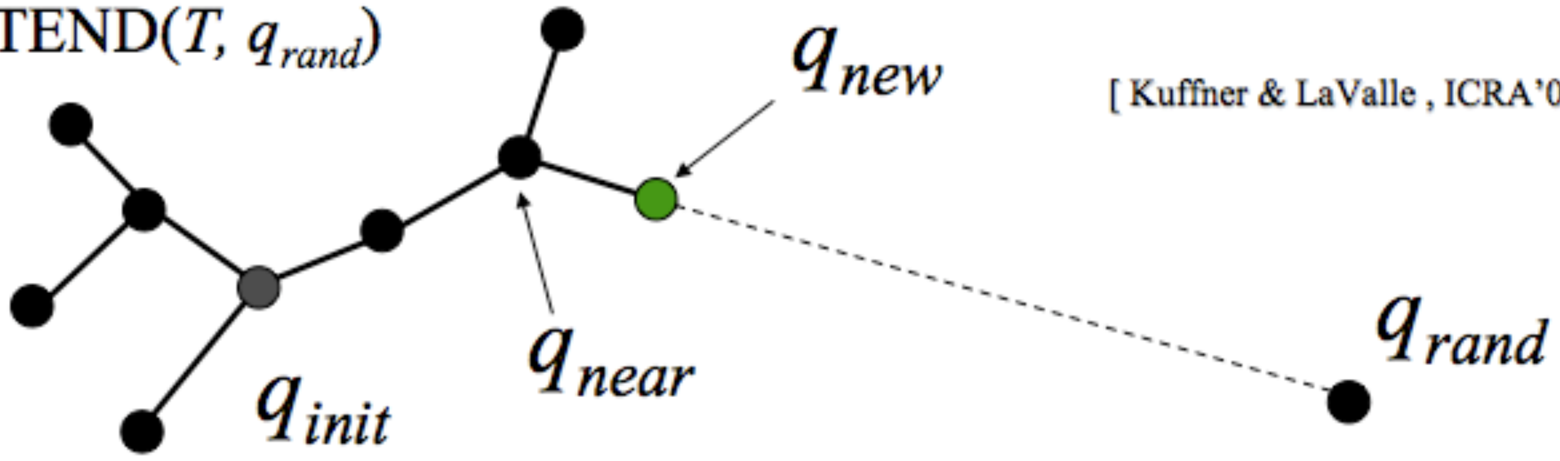
[Kuffner & LaValle , ICRA'00]

RRT’s

```
BUILD_RRT ($q_{init}$) {
    $T.init(q_{init})$;
    for $k$ = 1 to K do
        $q_{rand}$ = RANDOM_CONFIG();
        EXTEND($T$, $q_{rand}$)
}
```

Details:
Step length: how far to sample
Sample just at the end point
Sample all along, small steps

EXTEND(T, q_{rand})



[Kuffner & LaValle , ICRA’00]

Naïve Random Tree

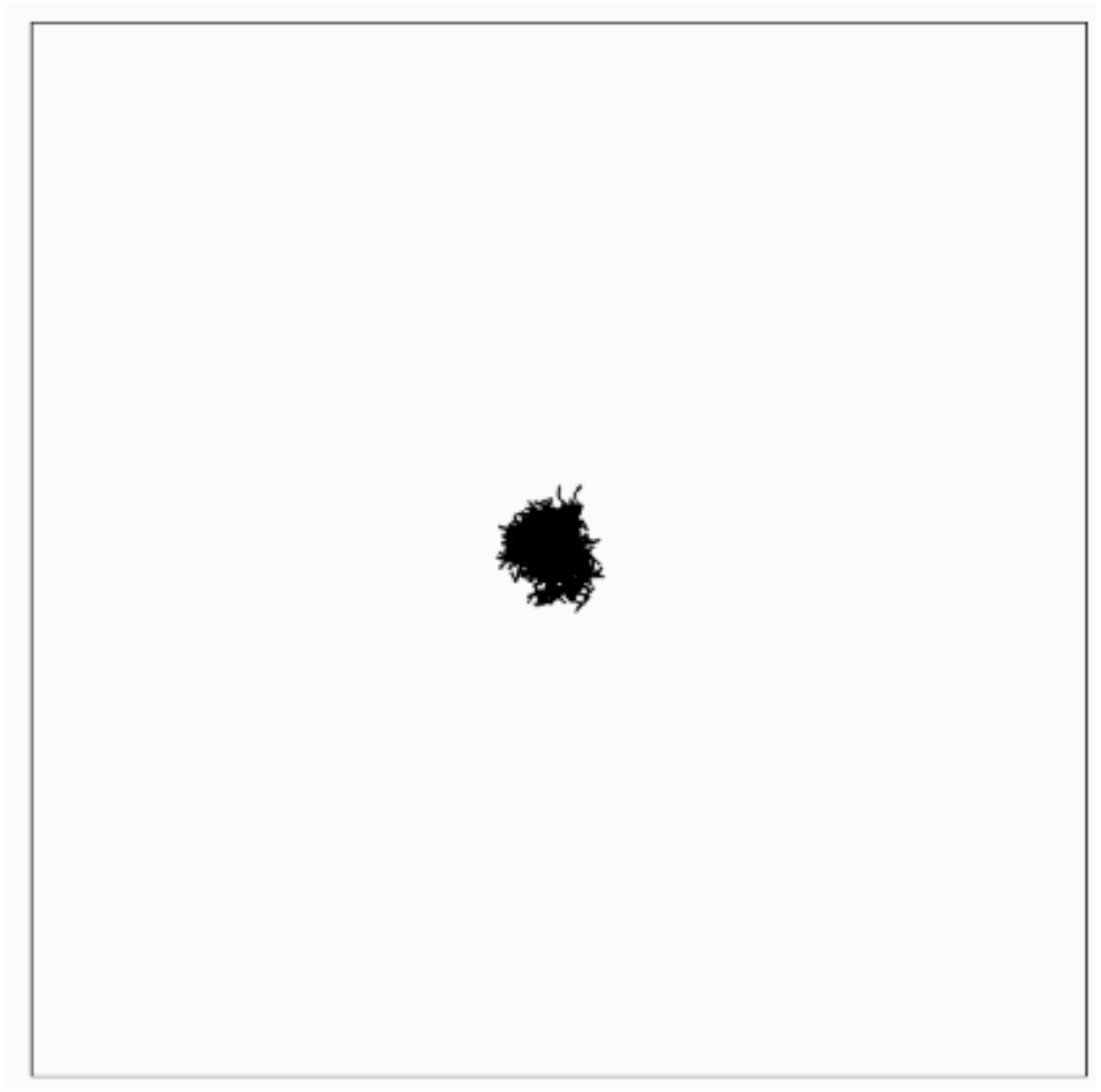
Start with middle

Sample near this node

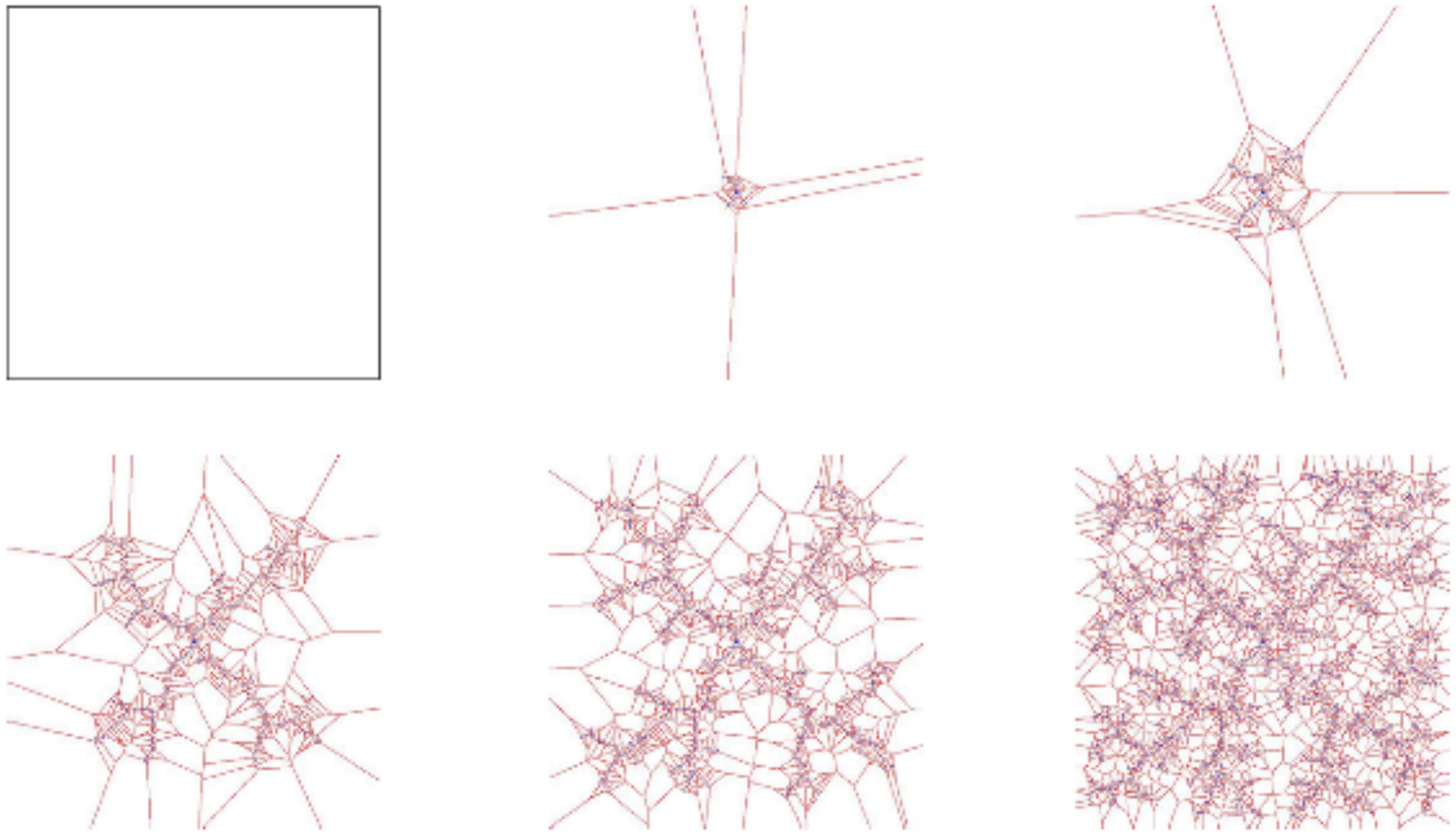
Then pick a node at random in tree

Sample near it

End up Staying in middle

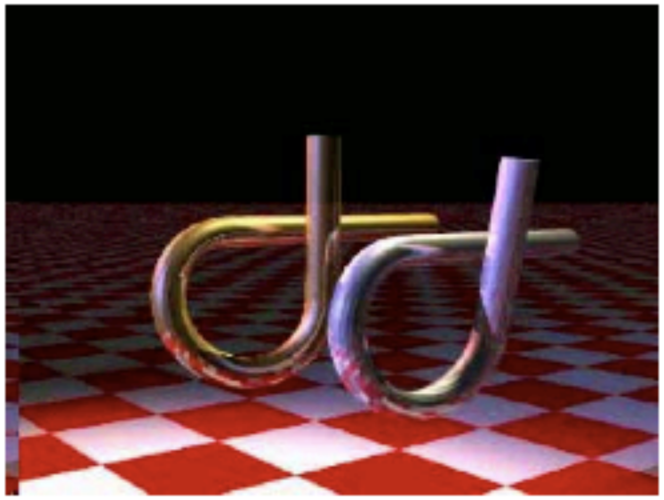


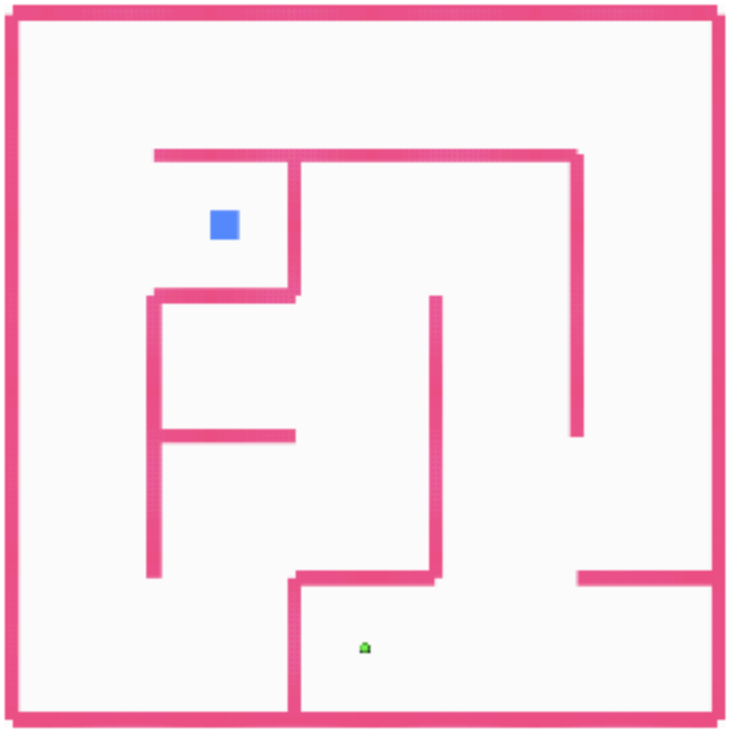
RRT’s are biased towards large Voronoi cells



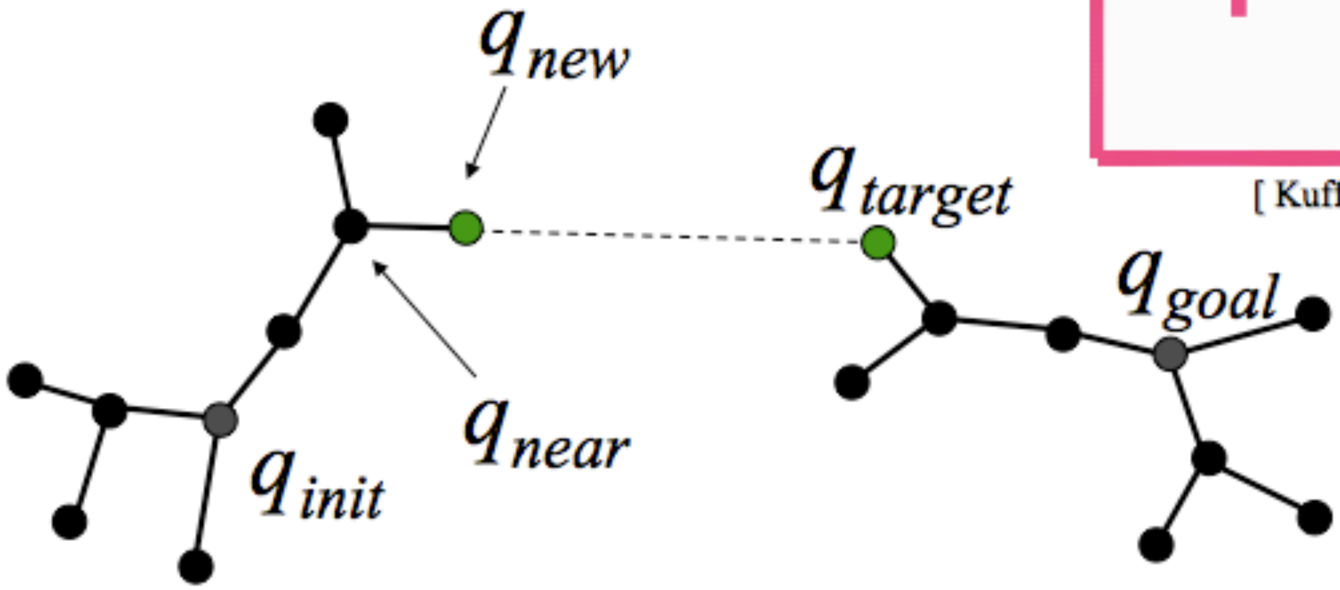
The nodes most likely to be closest to a randomly chosen point in state space are those with the largest Voronoi regions. The largest Voronoi regions belong to nodes along the frontier of the tree, so these frontier nodes are automatically favored when choosing which node to expand.

Grow two RRT's together



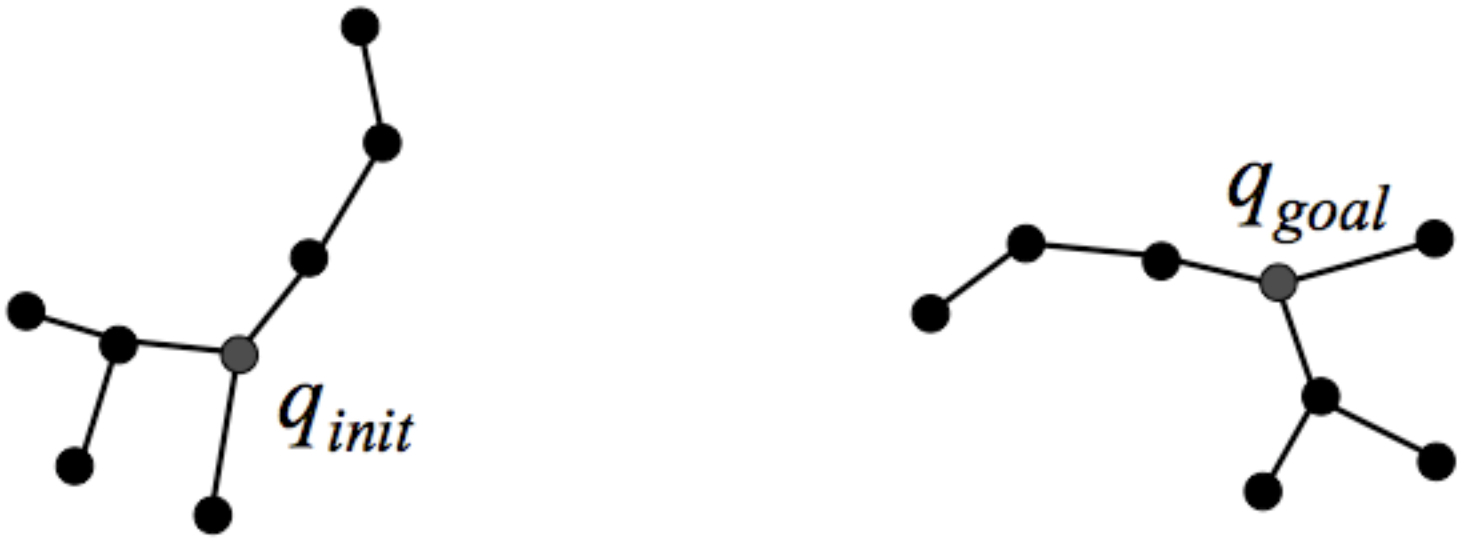


[Kuffner, LaValle ICRA '00]



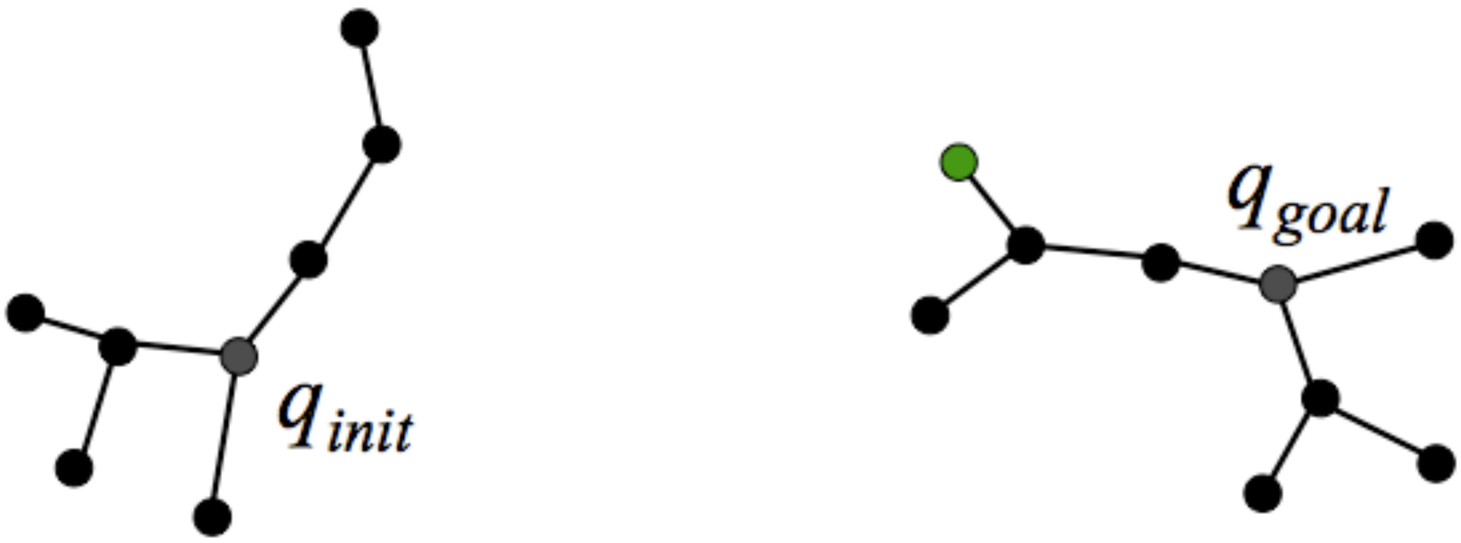
Two RRT’s

A single RRT-Connect iteration...



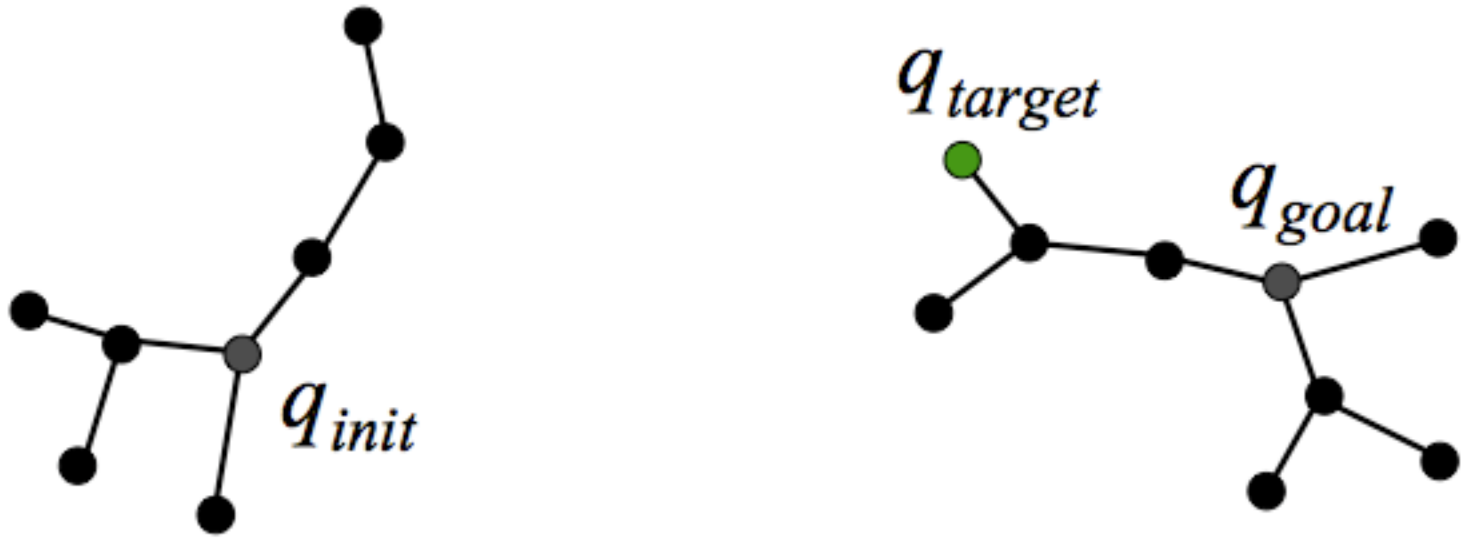
Two RRT's

1) One tree grown using random target



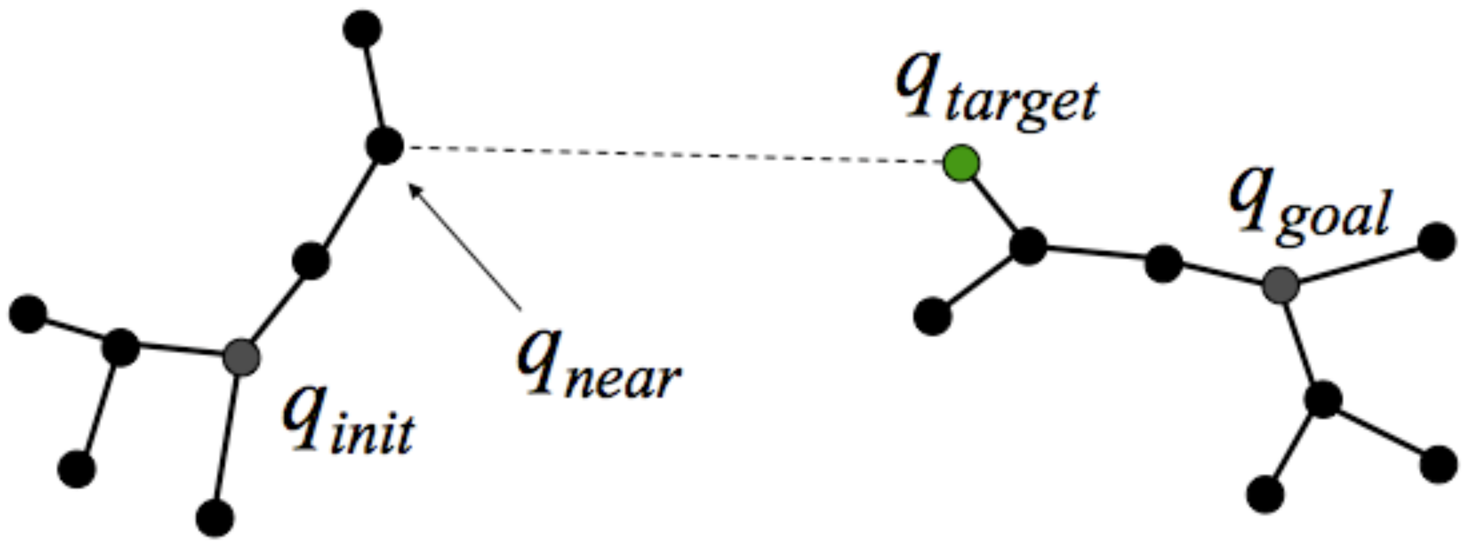
Two RRT’s

2) New node becomes target for other tree

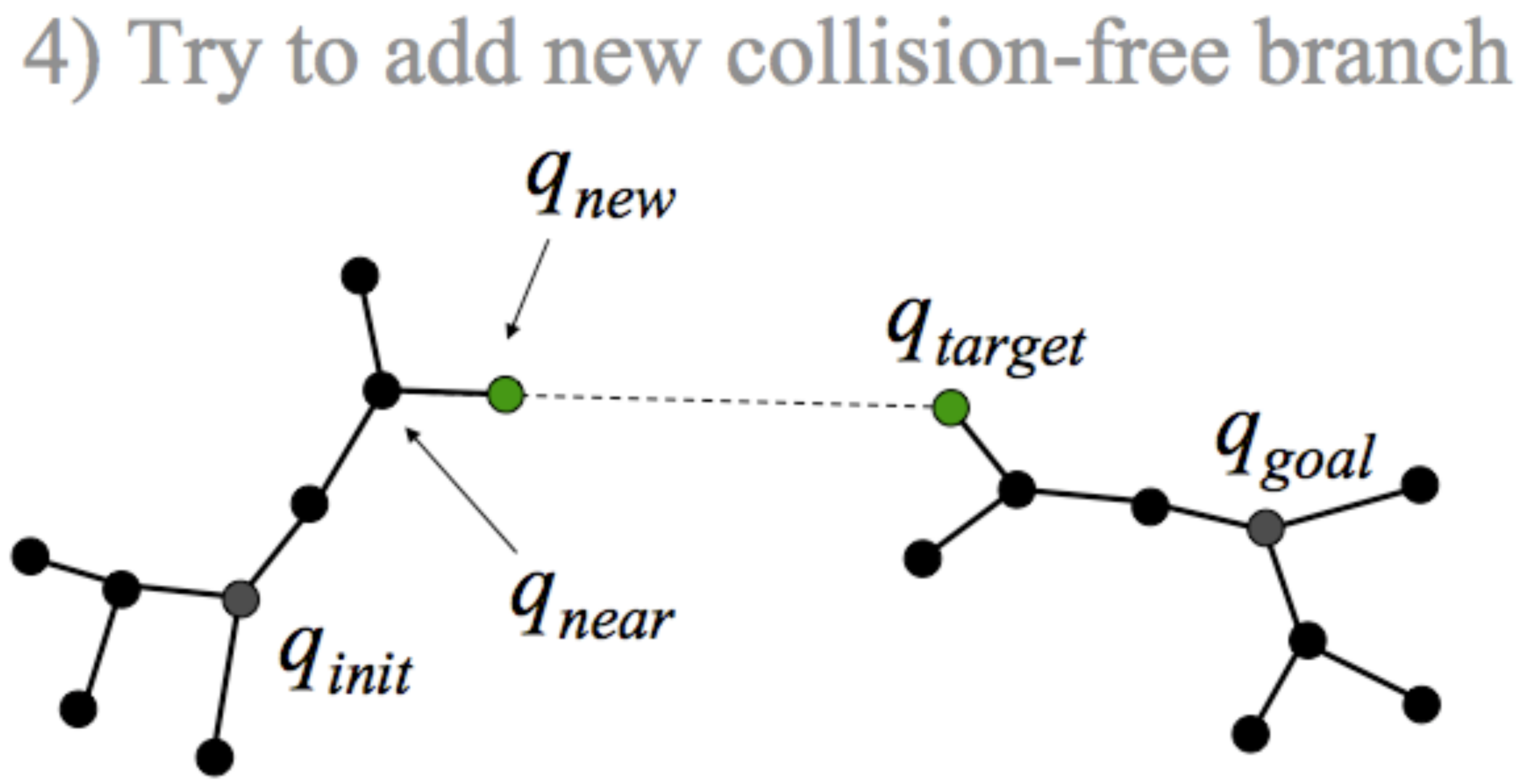


Two RRT's

3) Calculate node "nearest" to target

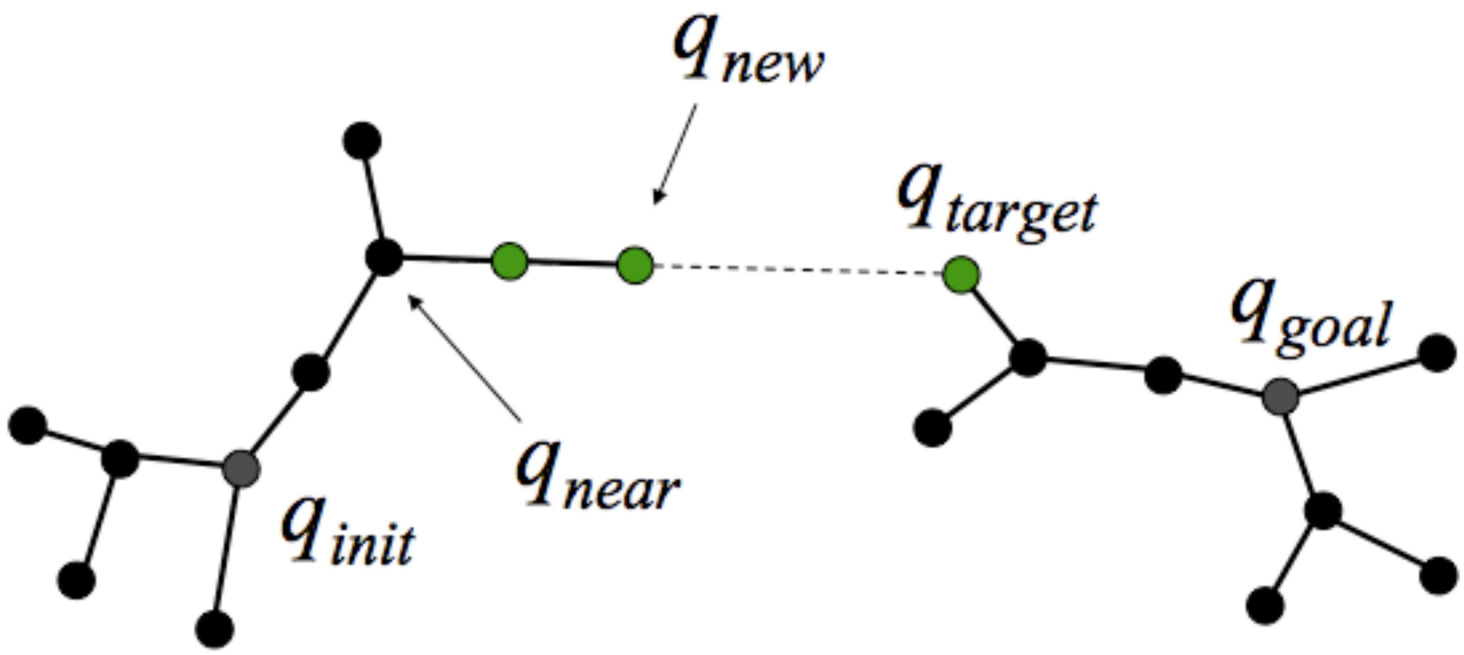


Two RRT's


4) Try to add new collision-free branch
q_{new}
q_{target}
q_{goal}
q_{near}
q_{init}

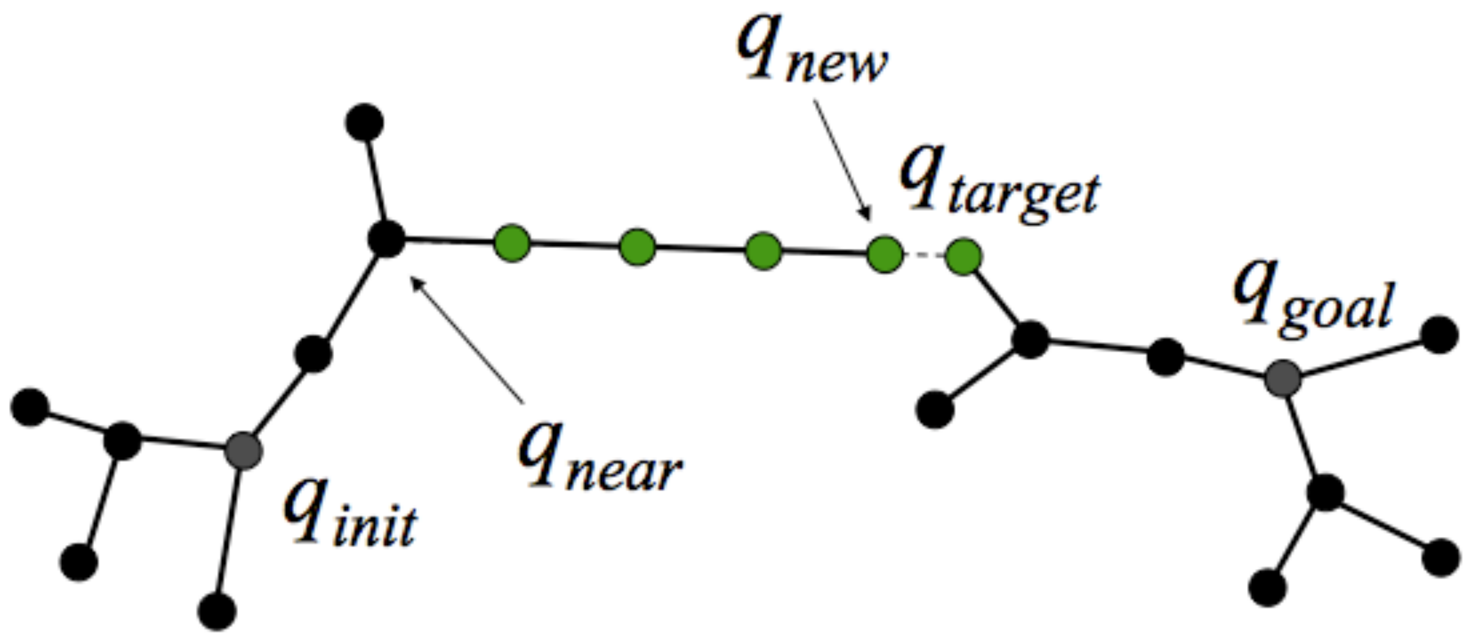
Two RRT’s

5) If successful, keep extending branch



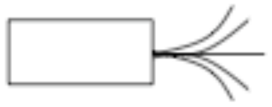
Two RRT's

5) If successful, keep extending branch



Taking actions into account

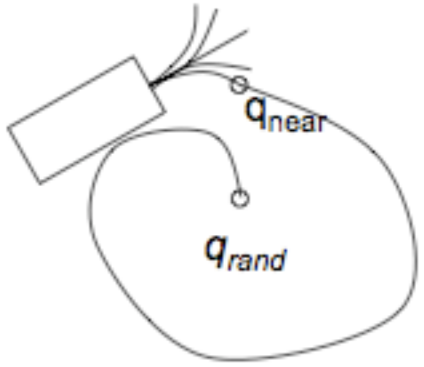
q_{near}

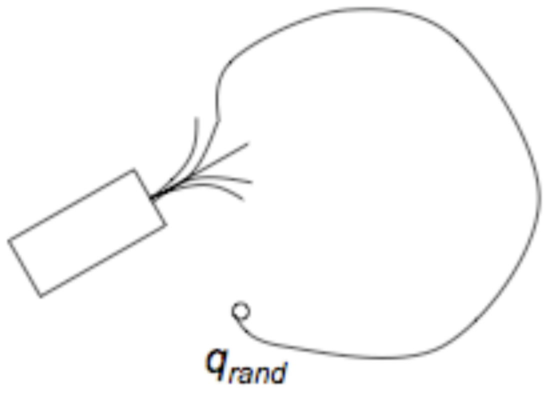


$q' = f(q,u)$ --- use action u from q to arrive at q'

chose $u_* = \arg\min(d(q_{rand}, q'))$

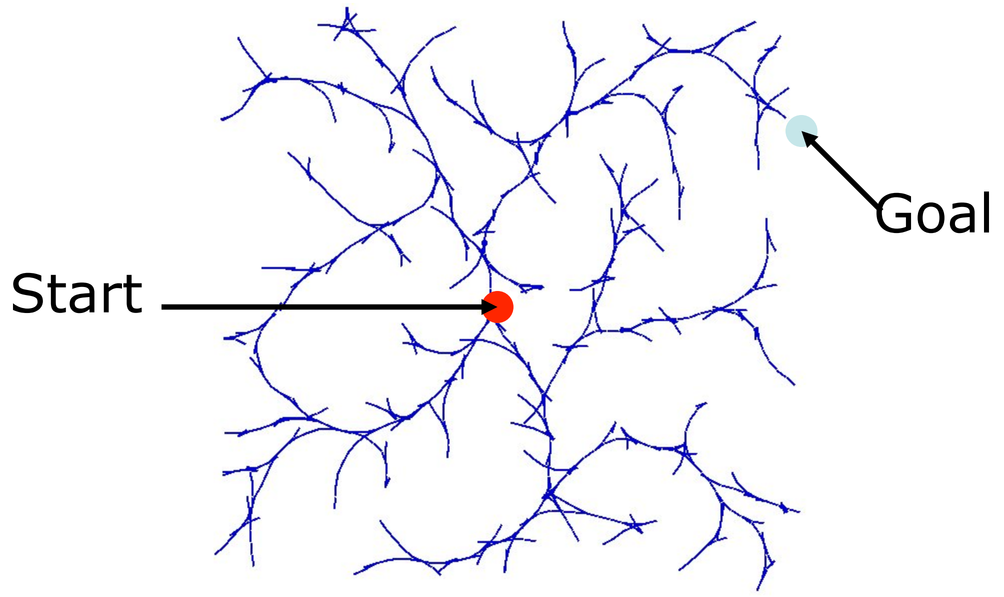
Is this the best?





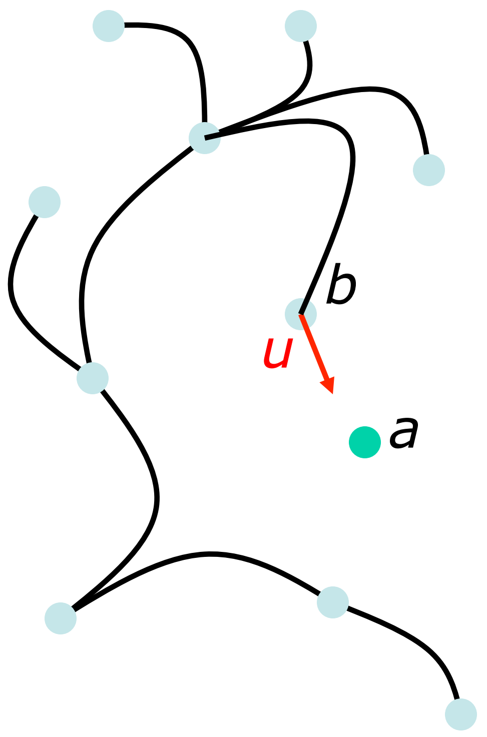
How it Works

- Build a rapidly-exploring random tree in state space (*X*), starting at s_{start}
- Stop when tree gets sufficiently close to s_{goal}



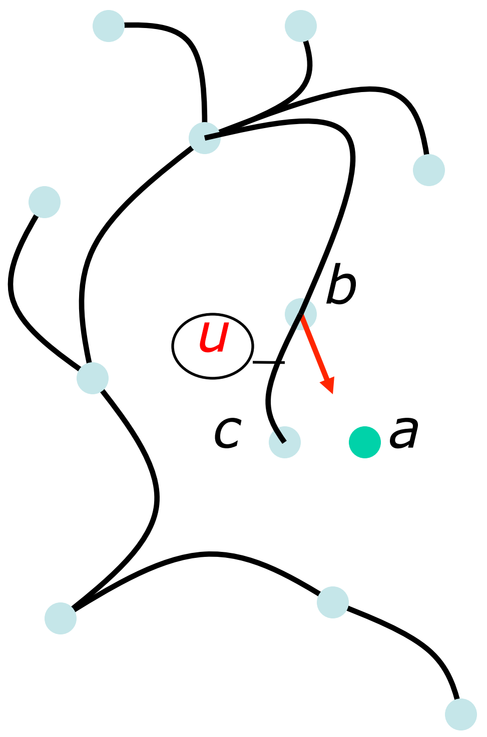
Building an RRT

- To extend an RRT:
  - Pick a random point *a* in *X*
  - Find *b*, the node of the tree closest to *a*
  - Find control inputs *u* to steer the robot from *b* to *a*



Building an RRT

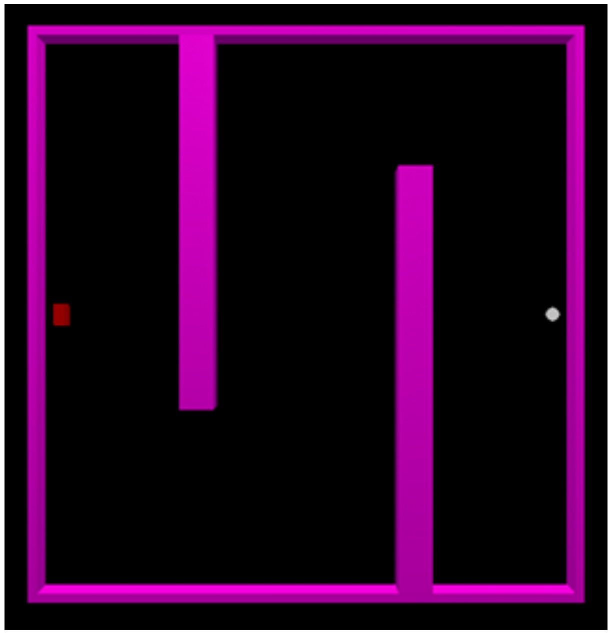
- To extend an RRT (cont.)
  - Apply control inputs *u* for time δ, so robot reaches *c*
  - If no collisions occur in getting from *a* to *c*, add *c* to RRT and record *u* with new edge

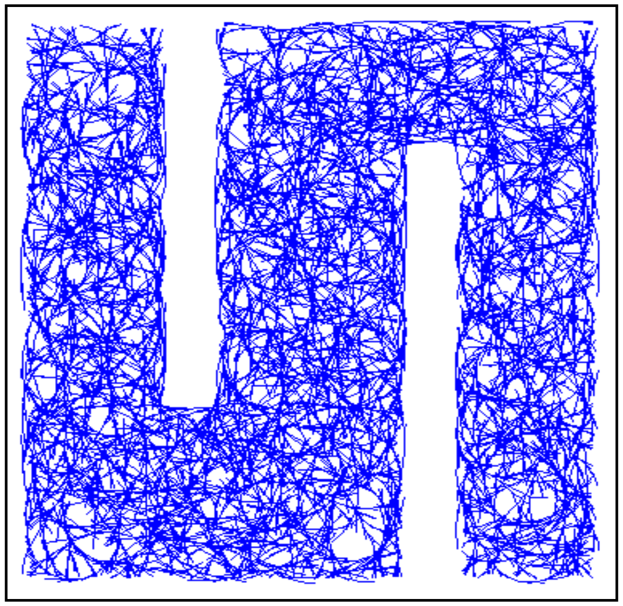


Executing the Path

- Once the RRT reaches s_{goal}
  - Backtrack along tree to identify edges that lead from s_{start} to s_{goal}
  - Drive robot using control inputs stored along edges in the tree

Problem of Simple RRT Planner

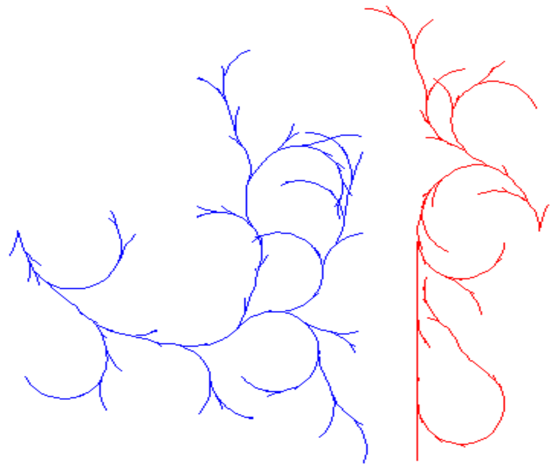




- Problem: ordinary RRT explores *X* uniformly
  $\rightarrow$ slow convergence
- Solution: bias distribution towards the goal – once in a while choose goal as new random configuration (5-10%)
- If goal is choose 100% time then it is randomized potential planner

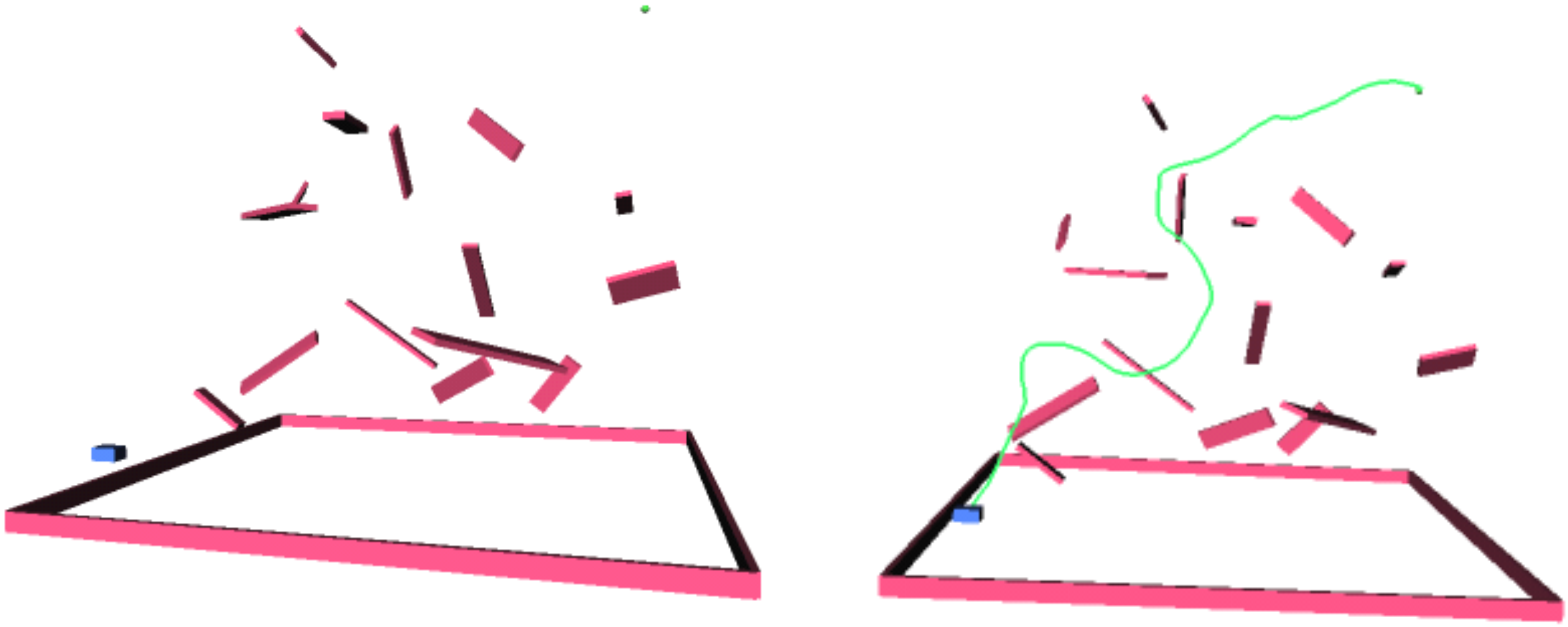
Bidirectional Planners

- Build two RRTs, from start and goal state

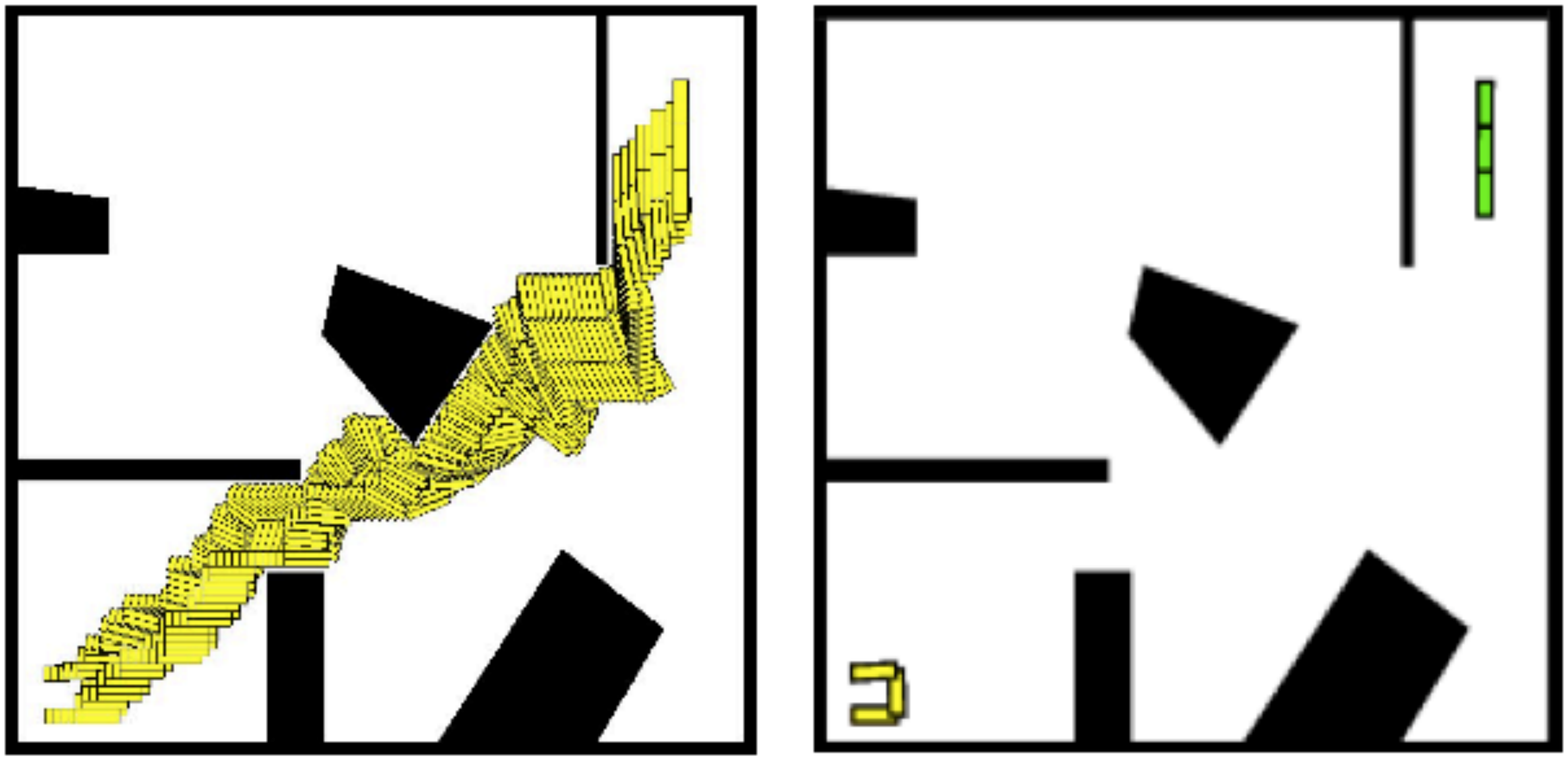


- Complication: need to connect two RRTs
  - local planner will not work (dynamic constraints)
  - **bias** the distribution, so that the trees meet

Bidirectional RRT Example



Articulated Robot example



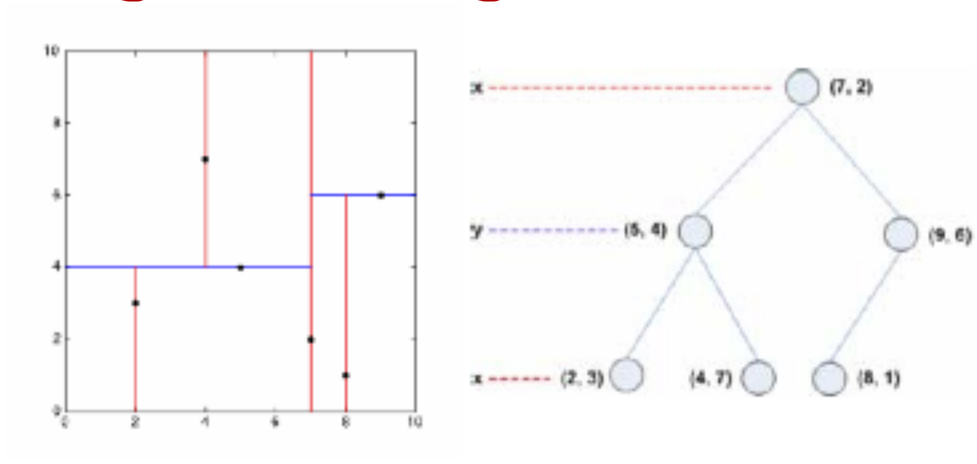
RRT’s

- Link
- http://msl.cs.uiuc.edu/rrt/gallery.html

- Issues/problems
- Metric sensitivity
- Nearest neighbour efficiency
- Optimal sampling strategy
- Balance between greedy search and exploration

- Applications in mobile robotics, manipulation, humanoids, biology, drug design, areo-space, animation
- Extensions – real-time RRT’s, anytime RRT’s dynamic domains RRT’sm deterministic RRTs, hybrid RRT’s

Efficient nearest neighbour algorithms



- How to find NN in high dimensional spaces

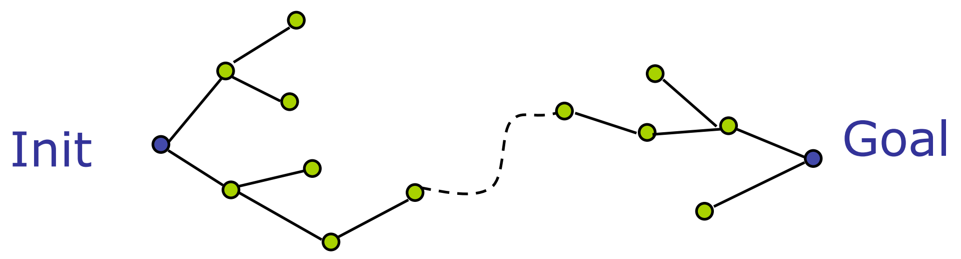
- KD trees – recursively choose a plane P that splits the set
  evenly in a coordinate direction
- Store P at the node
- Apply to children sets Sl and Sr
- Requires O(dn) storage

- Various hashing strategies

Expansion Space Tree (EST)

Path Planning in Expansive Configuration Spaces, D. Hsu, J.C. Latombe, & R. Motwani, 1999.

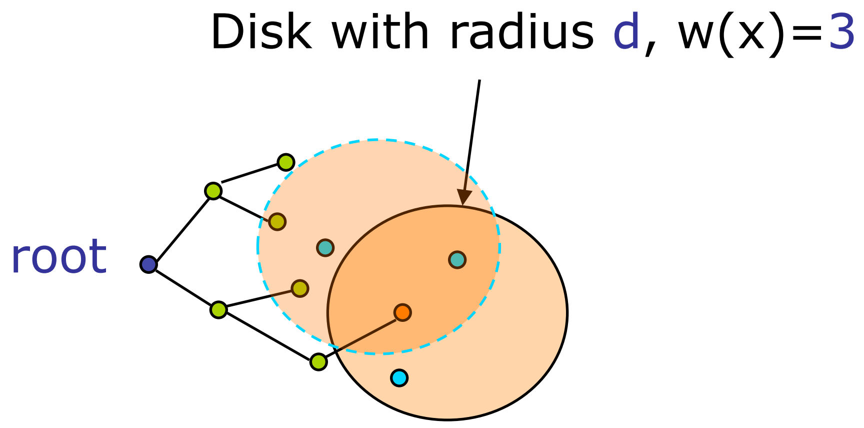
1. Grow two trees from Init position and Goal configurations.
2. Randomly sample nodes around existing nodes.
3. Connect a node in the tree rooted at Init to a node in the tree rooted at the Goal.



Expansion + Connection

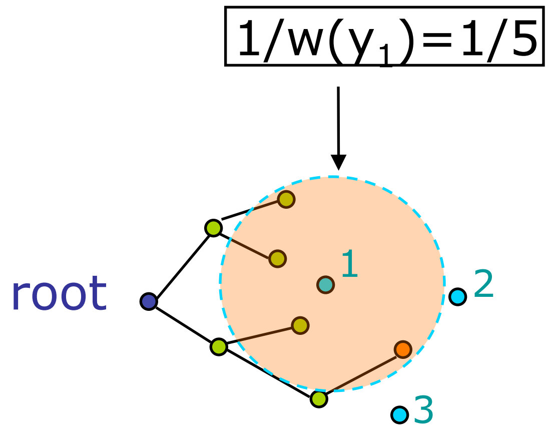
Expansion

1. Pick a node x with probability 1/w(x).
2. Randomly sample k points around x.
3. For each sample y, calculate w(y) – number of samples in neighbourhood d of y
4. which gives probability 1/w(y) with which the vertex will be taken



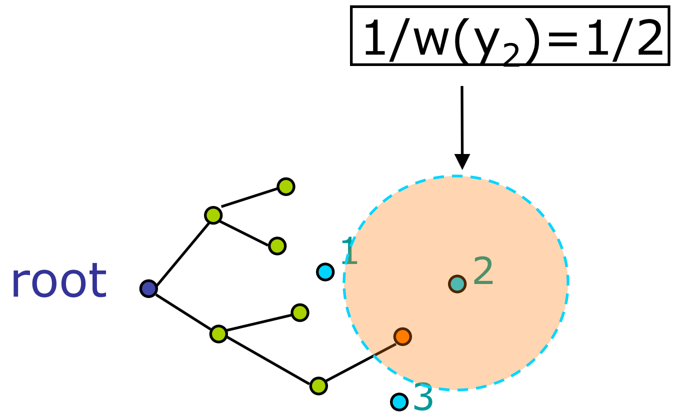
Expansion

1. Pick a node x with probability 1/w(x).
2. Randomly sample k points around x.
3. For each sample y, calculate w(y), which gives probability 1/w(y).



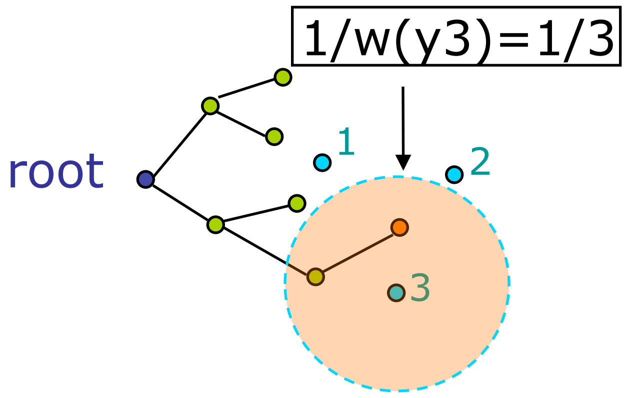
Expansion

1. Pick a node x with probability 1/w(x).
2. Randomly sample k points around x.
3. For each sample y, calculate w(y), which gives probability 1/w(y).



Expansion

1. Pick a node x with probability 1/w(x).
2. Randomly sample k points around x.
3. For each sample y, calculate w(y), which gives probability 1/w(y).

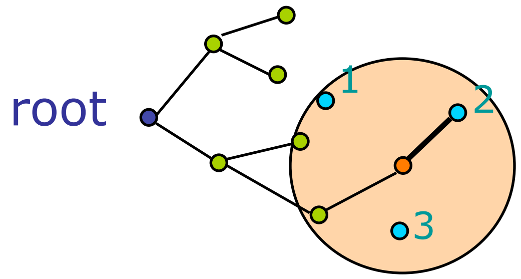


Expansion

1. Pick a node x with probability 1/w(x).
2. Randomly sample k points around x.
3. For each sample y, calculate w(y), which gives probability 1/w(y). If y

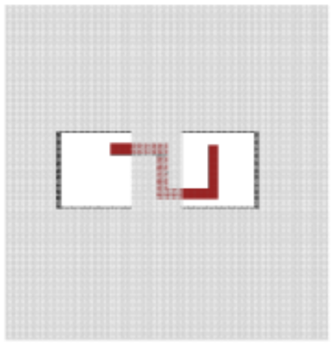
   (a) has higher probability; (b) collision free; (c) can see x

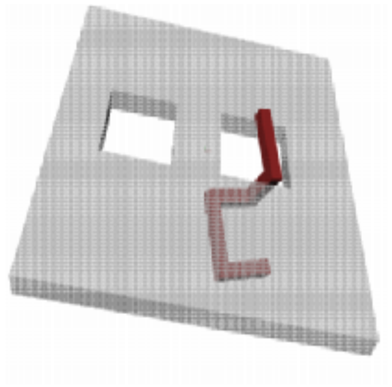
   then add y into the tree.

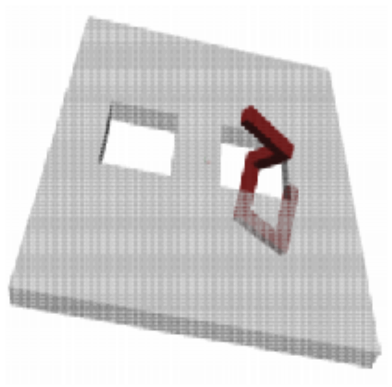


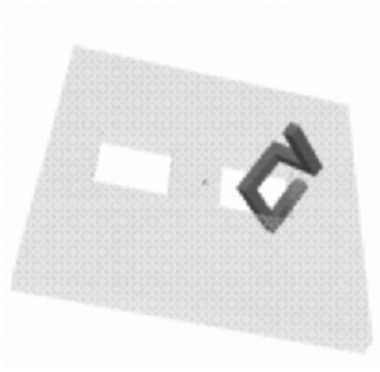
Requires tunning of various parameters k, d, number of iter

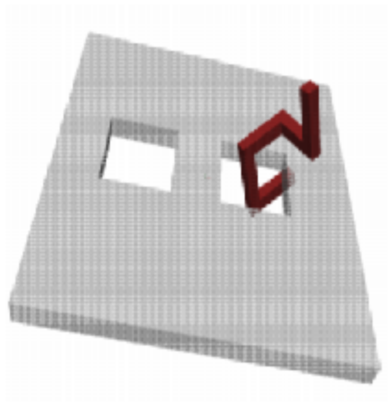
Computed example

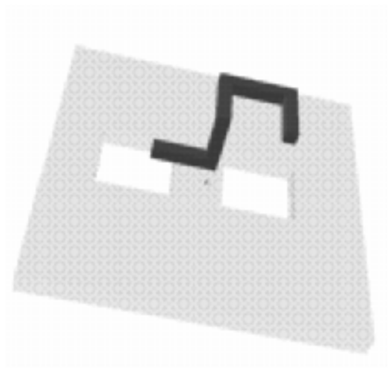

(a)


(b)


(c)


(d)


(f)


(g)

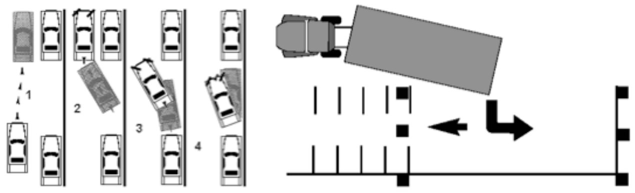
Conclusion

- Motion planning is difficult (intractable)
- Roadmap methods
  - Probabilistic Motion Planners

We will return to planning when considering partial information, dynamically changing worlds, uncertainty

What is not covered?

- Other types of motion planning
  - With constraints
    - Close-chain constraint
    - Nonholonomic constraint
    - Differential constraints
  - Manipulate planning
  - Assembly planning
  - Planning with uncertainty
  - Planning for multiple robots, dynamic env
  - Planning for highly articulated objects
  - Planning for deformable objects
  - …



Little Seiko

Additional Readings

- **Gross motion planning—a survey**, Y. K. Hwang and N. Ahuja, ACM Computing Surveys, 1992 (survey paper)
- **Robot Motion Planning**. J.C. Latombe. Kluwer Academic Publishers, Boston, MA, 1991.
- **Motion Planning: A Journey of Robots, Molecules, Digital Actors, and Other Artifacts**. Jean-Claude Latombe, IJRR, 1999 (survey paper)
- **Planning Algorithms**, Steven LaValle, 2006, Cambridge University Pres, (Free download at http://planning.cs.uiuc.edu/)