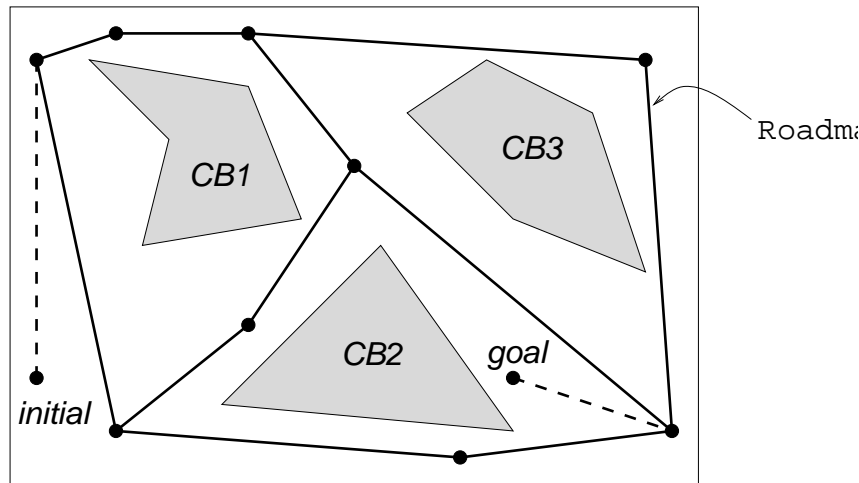


Probabilistic Roadmap Methods

Acknowledgement: Parts of these course notes are based on notes from courses given by Jean-Claude Latombe at Stanford University (and Chapter 4 in his text *Robot Motion Planning*, Kluwer, 1991), and O. Burçhan Bayazit at Washington University in St. Louis.

Roadmap Methods

Idea: capture the connectivity of \mathcal{C}_{free} with a **roadmap** (graph or network) of one-dimensional curves



PATH PLANNING WITH A ROADMAP

input: configurations \mathbf{q}_{init} and \mathbf{q}_{goal} , and \mathcal{B}

output: a path in \mathcal{C}_{free} connecting \mathbf{q}_{init} and \mathbf{q}_{goal}

1. build a roadmap in \mathcal{C}_{free} (preprocessing)
 - roadmap nodes are free configurations (or semi-free)
 - two nodes connected by edge if can (easily) move between them
2. connect \mathbf{q}_{init} and \mathbf{q}_{goal} to roadmap nodes v_{init} and v_{goal} (in same connected component)
3. find a path in the roadmap between v_{init} and v_{goal}
 - directly gives a path in \mathcal{C}_{free}

Hard part is building the roadmap

Completeness of a Planner

Definitions:

- a **complete planner** is one that correctly returns a path when one exists and declares that no path exists otherwise
- a planner is **resolution complete** if it correctly returns a path when one exists *at the discretization level chosen* and returns failure otherwise
Sometimes, one also requires that the discretization can be set fine enough so that if a solution will be found if one exists
- a planner is **probabilistically complete** if, whenever a path exists, the probability that it does not find one converges to zero as computation time increases

and some other oft used terms...

- a planner is often said to be **heuristic** to imply that it is incomplete (but there are heuristics that preserve completeness)
- a planner is often said to be **exact** if it is complete and uses algebraic representations of objects (exact is often opposed to “heuristic”)
Note that algebraic representations are no more “exact” than, say, bitmap representations (no representation of the real world is totally accurate)
- **local** and **global planners**: “local” is often used for planners with minimal preprocessing whereas “global” planners fully precompute a representation of \mathcal{C}_{free} 's connectivity (many planners between these extremes)

Moral

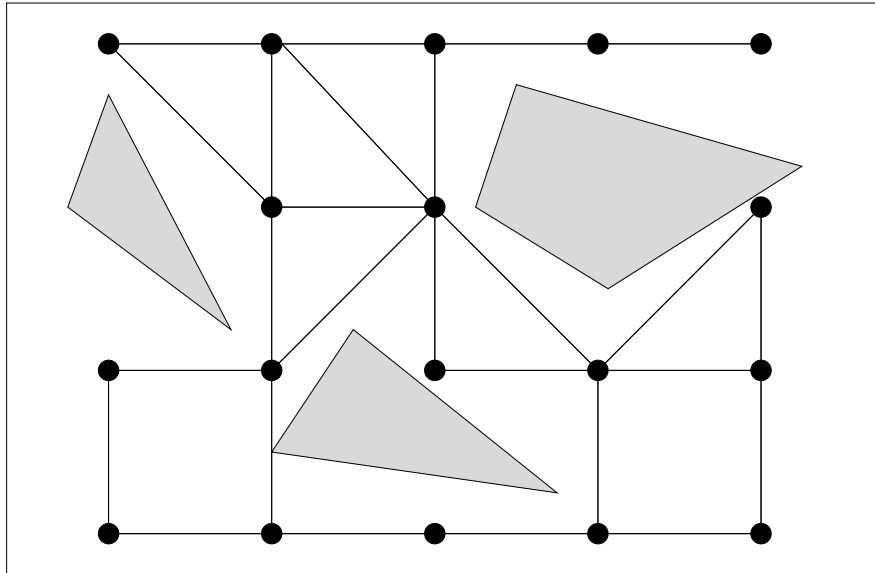
So far everything we've seen suggests:

- we should use complete planners only for special cases and for low-dimensional C-space (2 or 3)
- we should consider trading a limited amount of completeness for a major gain in computational efficiency
 - probabilistic completeness
 - resolution completeness

Moral: *What we probably want are heuristic, possibly incomplete, but practically efficient planners*

PRMs: Probabilistic Roadmaps

Basic Idea: Avoid explicitly constructing C-obstacles by sampling C-space



PREPROCESSING: BUILDING THE ROADMAP

1. pick n configurations at random (a uniform grid above)
2. keep those that are in \mathcal{C}_{free} (collision checking)
 - V is the resulting set of free configurations (vertices of the roadmap)
3. for each pair of configurations in V
 - try to connect them using a simple, fast, deterministic motion planner
 - each successful connection becomes a roadmap edge $e \in E$
 - the planner used in this step is the **local planner** – example is 'straight line in C-space' (check at discrete intervals for collision)

QUERY PROCESSING

1. connect start and goal configurations to the roadmap
2. search roadmap for a path between connection points

PRMs: Probabilistic Roadmaps

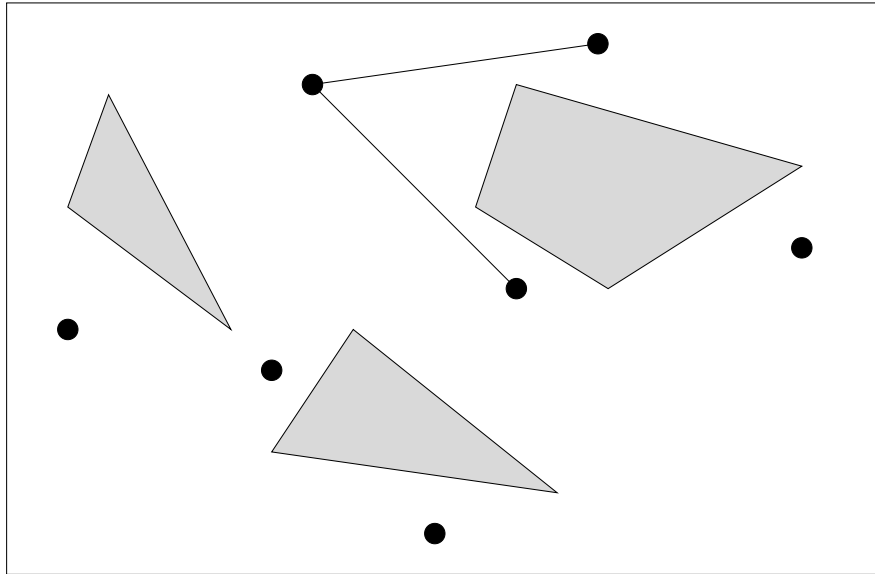
We just saw a simplified view of PRMs . . . Some of their attributes:

- *can* be applied to high-dimensional C-space (robots with many links)
- a *practical* method (at last!)
- *probabilistically complete* (complete if run long enough)
- roadmap construction time (preprocessing) depends on scene
 - ranges from a few minutes to sometimes many hours
 - can be adjusted by 'tuning' the parameters
- after roadmap is built, motion planning queries are very fast
 - typically less than a second
- independently discovered [Overmars 1992] and [Kavraki and Latombe 1994] (reported jointly in “Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces,” *IEEE Trans. Robotics and Automation*, 12(4), 1996, pp. 566-580)

Basic Philosophy:

- building an explicit representation of C-space takes too long
 - instead, sample C-space and 'learn' only as much about its connectivity as we need for planning
-

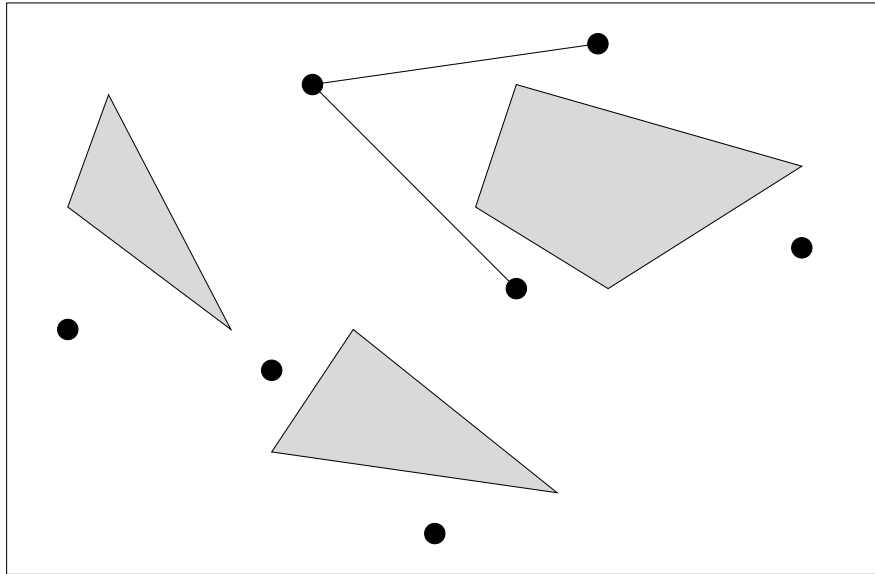
Building the Roadmap – The Learning Phase



PREPROCESSING (LEARNING PHASE): BUILDING THE ROADMAP

1. initialize R 's nodes V and edges E , i.e., $V := E := \emptyset$
2. randomly generate **free configuration** c and add it to V (roadmap nodes)
3. find a set $N_c \subset V$ of **candidate neighbors** for c among the current roadmap nodes V and sort them by increasing order of distance from c
4. for each node $c' \in N_c$ (in order of distance from c)
 - (a) if c' and c are not in the same connected component of R try to connect them using a **local planner** (otherwise go on to next neighbor in N_c)
 - (b) if c and c' can be connected by local planner, then add edge (c, c') to E and update R 's connected components
5. goto step 2 (stop when time is up or Roadmap is 'good enough')

Building the Roadmap – Subroutines and Parameters



Subroutines and Parameters:

- Major Operation: Validity Check of Sample – e.g., collision detection (in workspace with favorite method)
- How to generate random configurations?
 - uniformly select value for each parameter (dof)
- Local planner?
 - straight line in C-space
 - deterministic, so don't have to store path in R (only edge)
- Parameters (depends on scene)
 - time to spend building roadmap
 - how to pick candidate neighbors (N_c)
 - how to measure 'distance' between configurations

Building the Roadmap – Subroutines and Parameters

Generating Random Configurations

Approach: pick value for each dof of robot from its valid range

- rigid object in \mathbb{R}^3 :
 - $(x, y, z) \in \mathbb{R}^3$ for reference point
 - $(\theta_1, \theta_2, \theta_3) \in [0, 2\pi)$ orientation
- articulated object in \mathbb{R}^2 :
 - $(x, y) \in \mathbb{R}^2$ for reference point
 - $\theta_i \in [0, 2\pi)$ for revolute joint i

Picking Candidates for Connection

Goal: pick points that are 'easy' to connect ...

Intuitively, we'd like to try 'close' points ..., but what is close?

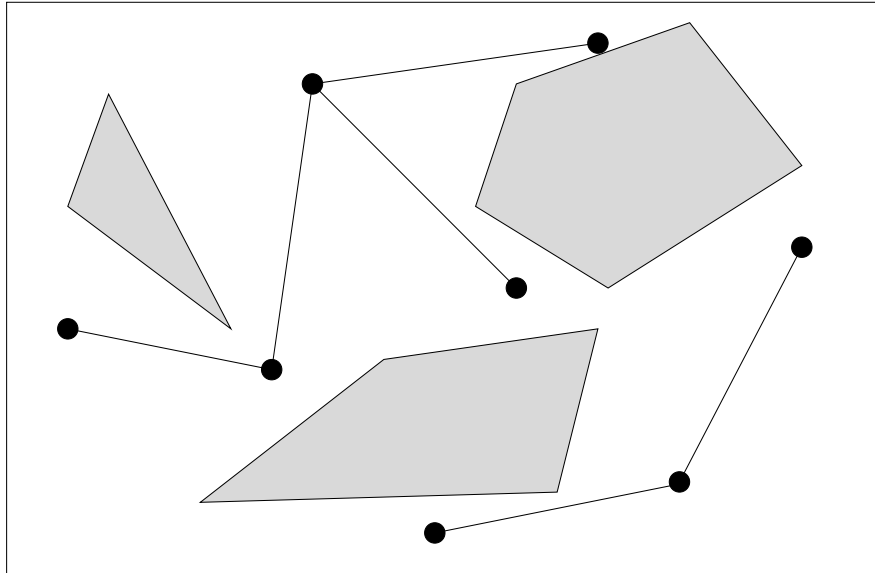
- Euclidean-type distance in C-space
e.g., for a planar articulated robot with m links and a fixed-base (reference point)

$$d(x, y) = \left(\sum_{i=1}^m (j_i(x) - j_i(y))^2 \right)^{1/2}$$

where $j_i(x)$ is the position of the i th joint when the robot is in configuration x and the difference is the 'smallest' angle between the two joints

Building the Roadmap – Expansion Step

Problem: sometimes the resulting roadmap might not be 'good enough'



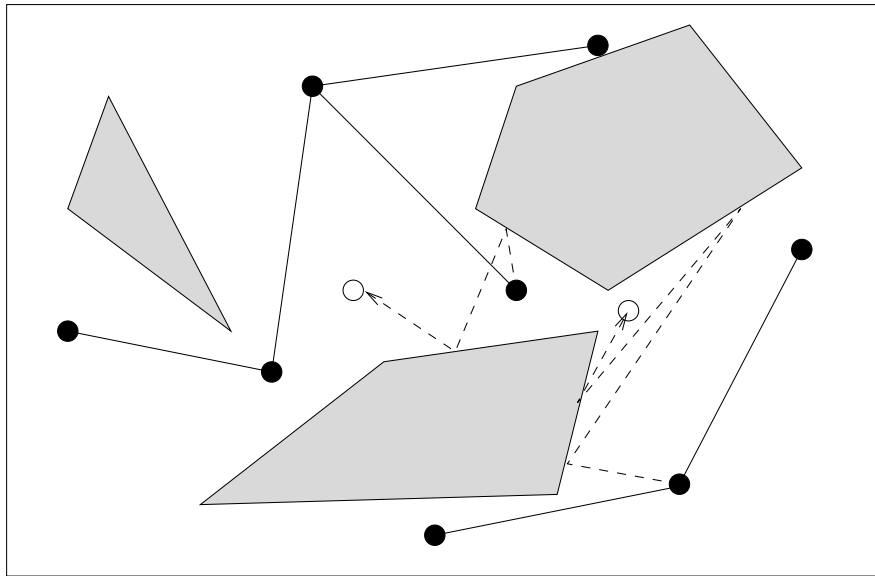
Idea: Try to 'expand' Roadmap in difficult regions

- identify regions where need more roadmap nodes
- generate nodes in these regions
- connect them to existing roadmap and hopefully improve connectivity

Key: Defining 'hard regions' (heuristics)

- assign 'weight' to nodes representing 'complexity' of C-space in its vicinity
– e.g., number of nodes in V within certain distance
- generate new roadmap nodes 'near' current nodes with large weights

Building the Roadmap – Expansion Step (cont'd)



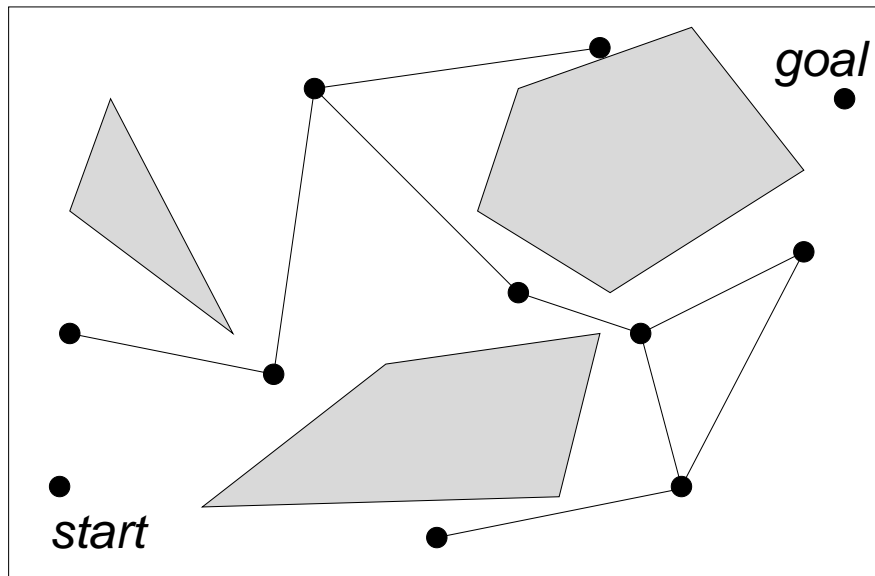
weights:

- for each $v \in V$ compute **failure ratio** of local planner connection attempts
 - $r_f(v) = \text{fail_lp}(v) / (\text{attempt_lp}(v) + 1)$
- the **weight** of node v is normalized failure ratio
 - $w(v) = r_f(v) / \sum_{w \in V} r_f(w)$

'expanding' a node:

- select nodes to 'expand' according to weights
 - $Pr(v \text{ is expanded}) = w(v)$
- expand v with short 'random walk' from v ending at a new node v'
 - pick direction at random and walk until bounce off C-obstacle
 - repeat a few times
- connect v to v' (add (v, v') to E , and **save** path)
- try to connect v' to nodes in different connected components

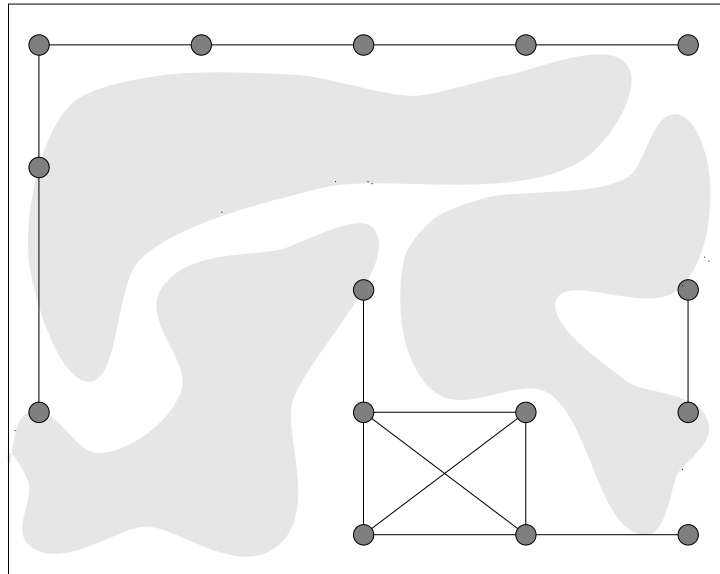
Answering Queries



QUERIES

1. connect start and goal configurations to the Roadmap
 - use local planner
 - can also use 'smarter' planner since can save paths at this point
2. search Roadmap for path between two connection points
 - regenerate motion plans for each edge in the path
3. typically very fast (less than a second)

Hard Cases for PRM



PRM: Overmars 1992, Kavraki/Latombe 1994

- generate candidate roadmap vertices according to uniform distribution in C-space
 - grid or randomly generate values for each dof
 - discard non-free configurations (collision check)
- heuristics for ‘enhancing’ roadmap in cluttered areas
 - help to some degree, but not in all cases
- works well if C-space is not cluttered (C-space mostly free)
- long, narrow passages in C-space absent from roadmap
- cannot be used for contact tasks (need points on C-surfaces)

PRM Variants

Many PRM variants have been proposed in the past 10 years, e.g.:

- Biased-Sampling Methods (OBPRM, MAPRM, Gaussian PRM, Bridge Test, etc)
- Methods for Highly Constrained Problems (closed chain systems)
- Lazy Evaluation Methods (Lazy PRM, Fuzzy PRM, C-PRM), Iterative Solution Methods (IRC)
- Cooperative User/Planner Systems

Also, many Tree-Based methods have been proposed

- Ariadnes Clew Algorithm
- RRT

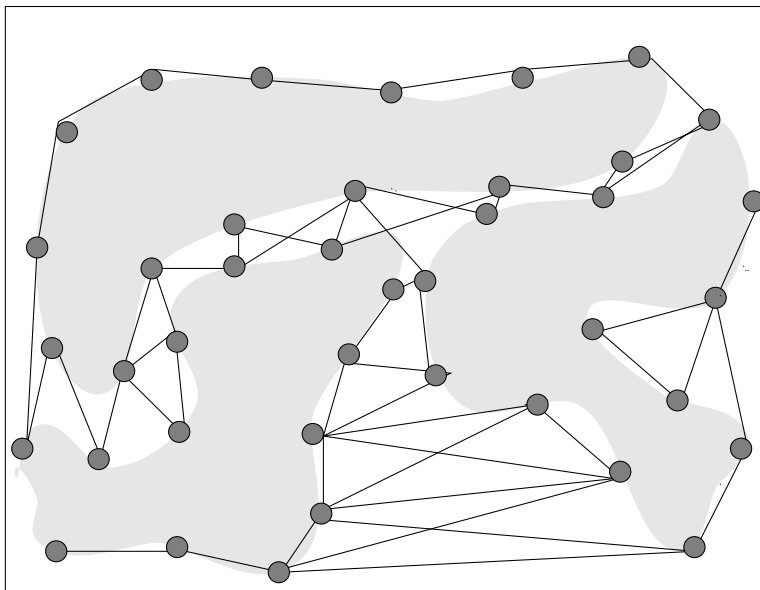
Obstacle-Based PRMs (OBPRMs)

Many applications have crowded C-space

- manufacturing – assembly, disassembly studies

Planning for contact tasks must be done on C-surfaces

- roadmap vertices must lie on C-obstacle surfaces



Obstacle-Based Probabilistic Roadmaps (OBPRM)

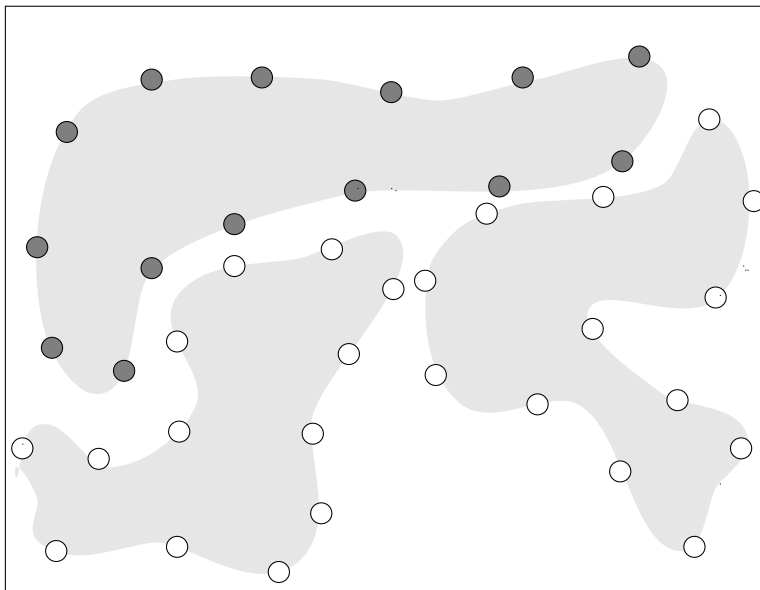
- roadmap nodes generated on C-obstacle surfaces
 - heuristic for “uniform” distribution
- finds paths in long, narrow C-space passages
- feasible for contact tasks

Generating Roadmap Candidate Nodes

Notation:

Workspace obstacles $S = \{s_1, s_2, \dots, s_n\}$

C-space obstacles $S' = \{s'_1, s'_2, \dots, s'_n\}$



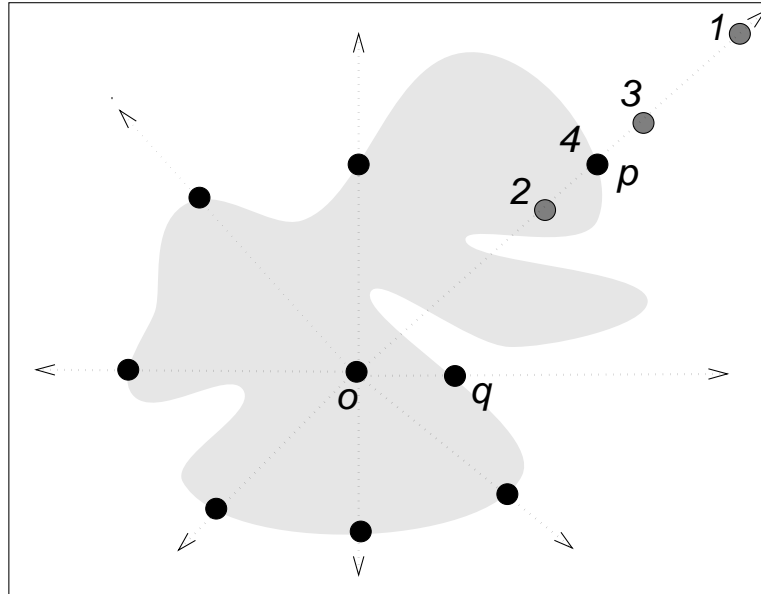
Basic Idea: generate candidate set V_i for each obstacle s_i

To generate V_i :

1. Try to generate m_i points p_1, p_2, \dots, p_{m_i} randomly distributed on surface of s'_i , and place in V_i
2. Remove all points from V_i that lie in the interior of some $s' \in S'$ (collision detection)

Finding Points on a C-obstacle

Problem: We don't want to compute the C-obstacle



A Heuristic Method:

1. Determine a point o (the origin) inside s'
e.g., any robot placement colliding with s
2. Select m rays with origin o and directions randomly distributed in C-space.
3. For each ray, use binary search to determine a point on the boundary of s' that lies on that ray.

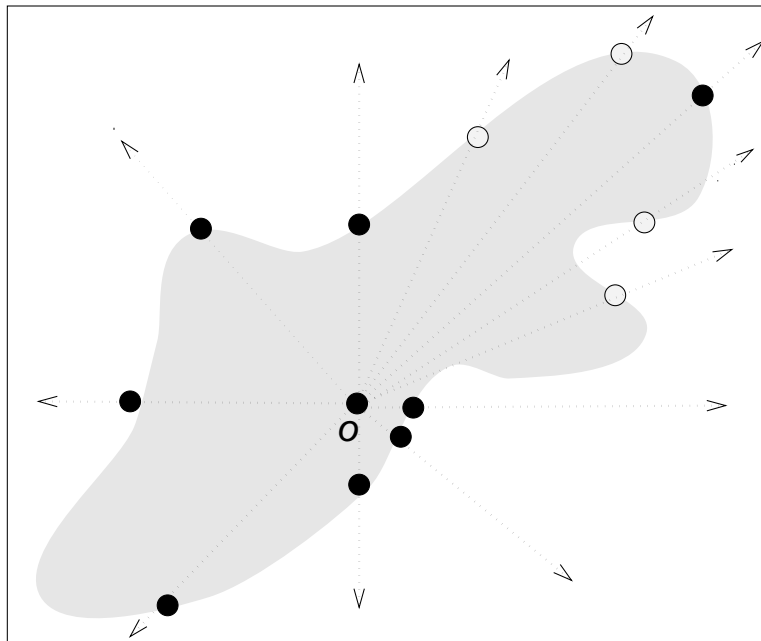
Some issues:

- selecting m , number of points to attempt to generate
- uniformity of the distribution

Sensitivity to Shape and Origin

Note: (near) uniform distribution only if

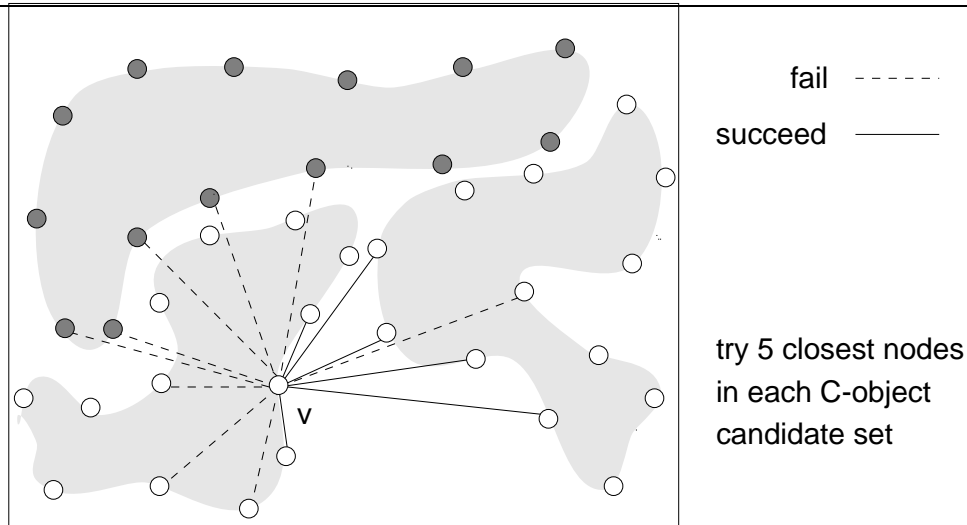
- C-object is roughly spherically shaped and
- the origin lies near its center



Optimizations for irregular shapes/bad origins:

1. compute nearest neighbor distance for points
 - generate points between points that are far apart
 - surface distance must also be far
2. compare surface normals of points with 'neighbors'
 - generate points between points with different normals
 - constraint surface curves between them

Connecting Roadmap Candidates



Goal: roadmap captures free space connectivity

– connection phase is *by far* most expensive of preprocessing

1. deterministic local planner:

- more powerful yields better connectivity but slower
- less powerful may sometimes fail, but can try more

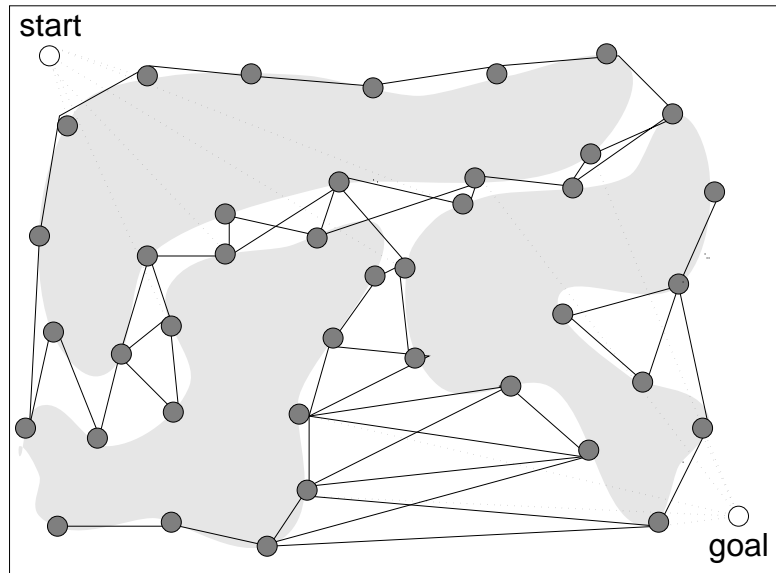
2. number of connection attempts per node:

- best connectivity if try to connect all node pairs
- must limit attempts to achieve reasonable run times

3. choice of connection candidates:

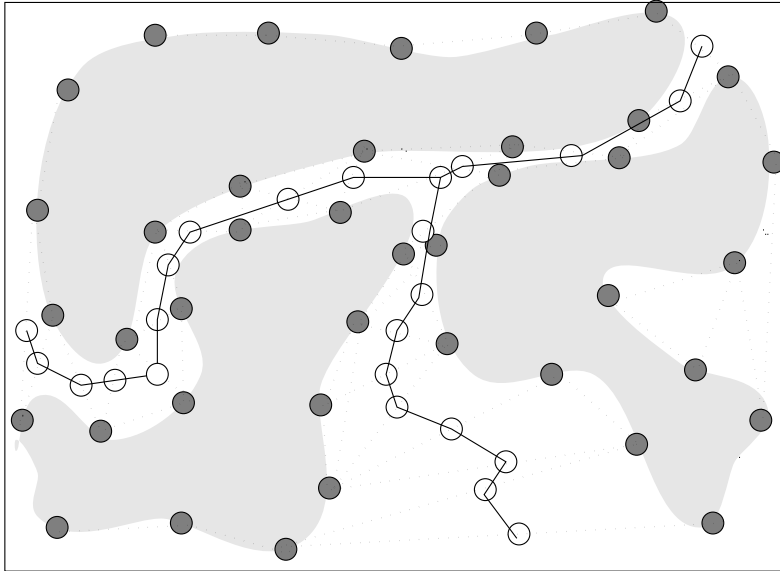
- need to identify likely candidates
- find k closest nodes (in each C-object set)
- reduce distance calculations – heuristic ‘partition’

Planning



1. connect start and goal to roadmap
 - need to connect to same connected component of roadmap
 - select candidates from each C-object set (as before)
 - attempt connection with simple local planner
 - if fail, can try random walk (can save motion plan)
2. find path in roadmap between two connection points
3. generate motion plan for each edge in roadmap path
 - redo the deterministic local planning

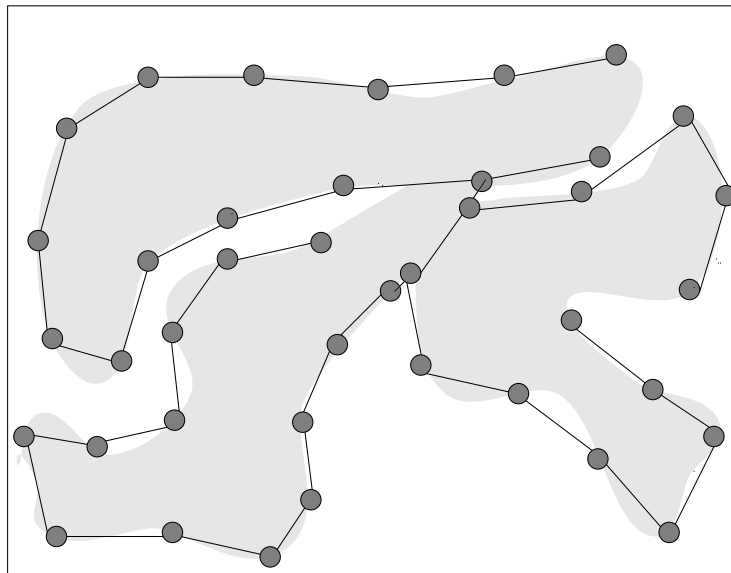
Improving Roadmap for Path Planning



Paths in roadmap ‘skip’ along C-objects (contact points)

- undesirable for path planning – hope to avoid contact
- can use smoothing techniques on final path
- can improve paths in roadmap (during preprocessing)
 1. add new node at midpoint of path found during initial connection phase
 2. attempt connections between new ‘corridor’ nodes

Contact Tasks



Planning for contact tasks done *on* constraint surfaces

- main difference – local planner
- selection of candidate nodes for connection to v
 - pick more from v 's C-object set (better chance)

Implementation Details

Simple Path Planner: get working prototype quickly

- 6-link (dof), planar articulated robot
- SGI Indy Workstation, 32 MB RAM
- local planner – straight-line planner in C-space

C-space distance metric

- Euclidean distance in C-space
- $j_i(x)$ position i th joint in configuration x

$$d(x, y) = \left(\sum_{i=1}^6 (j_i(x) - j_i(y))^2 \right)^{1/2}$$

Number of candidate attempts for C-obstacles

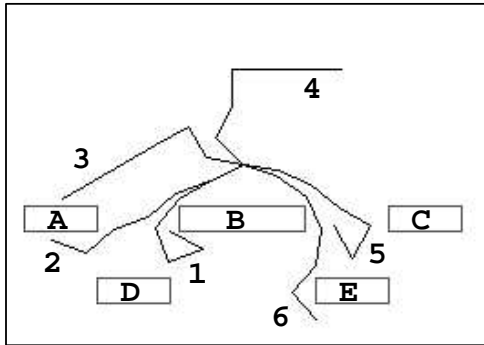
- for simplicity, we did same number for every C-obstacle
- about 4000 candidates per C-obstacle

Collision Detection

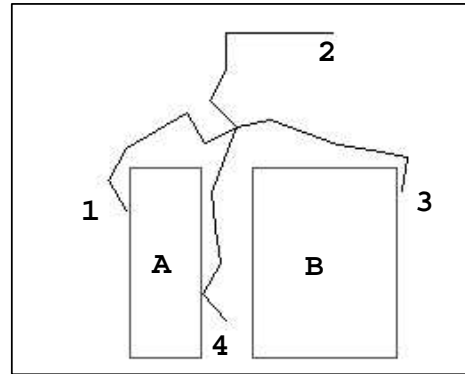
- done in workspace (easy, but not most efficient)

Simple Path Planner Implementation

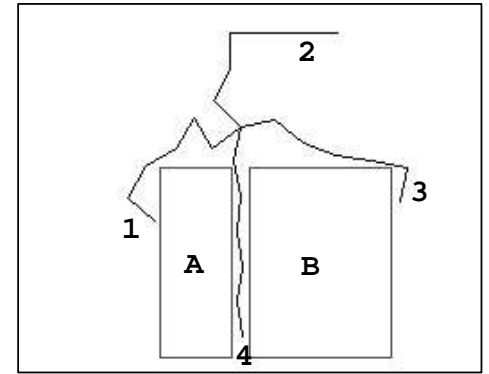
Scenario: 6 dof, planar articulated robot



E1



E2



E3

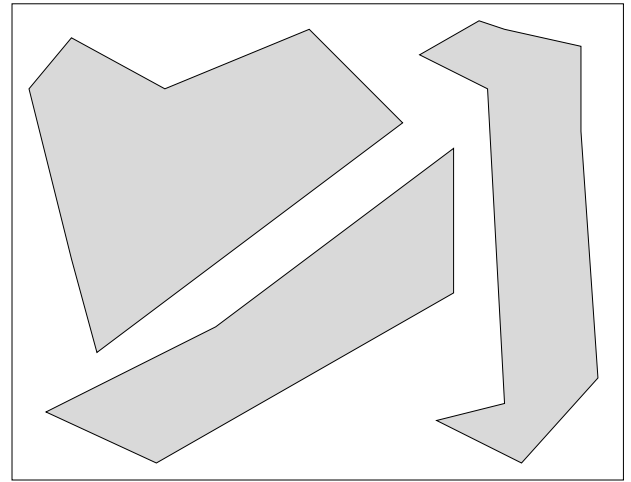
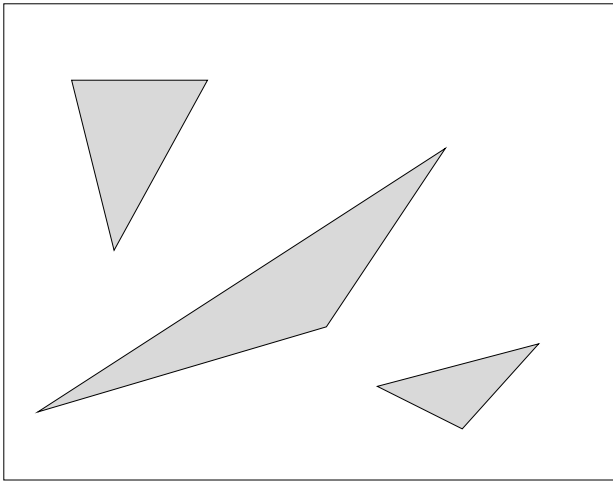
Preprocessing Times (seconds) and Statistics

Env	Generation		Connection	Roadmap Structure		
	#nodes	time	time	#cc	sizes	time
E1	21342	33	3880	1	21342	109
E2	14017	10	786	5	14011, 2(2), 1(2)	39
E3	9046	9	405	6	9041, 1(5)	14

Connection (Planning) Times (seconds)

Env	Configuration Number					
	1	2	3	4	5	6
E1	Fail	.0028	.0027	.0027	.0027	.0163
E2	.0016	.0012	.0104	.0012	–	–
E3	.0015	.0012	.0056	.0011	–	–

Mini HW #4: PRM and OBPRM Exercise



Consider the two-dimensional C-spaces shown above. For each case, draw roadmaps that might be produced using both PRM and OBPRM.

1. For both methods, use a straight line in C-space local planner and add nodes at the four corners of the boundary of C-space to the roadmap nodes.
2. For PRM, assume that 20 roadmap nodes are generated by throwing a 4×5 regular grid over C-space and that the roadmap that the roadmap is *not* enhanced. For the connection phase, try to connect each node to the 6 closest nodes.
3. For OBPRM, assume that 7 roadmap nodes are generated on the surface of each C-obstacle, and that the directions used to generate these nodes are found by breaking the available 360 degrees into 7 equal pieces. For the connection phase, try to connect each node to the 2 closest nodes on each C-obstacle, and to the four 'boundary' nodes.

What conclusions can you draw from this exercise? In particular, when does PRM outperform OBPRM and vice versa?

Summary of Basic Motion Planning Methods

Many of the methods we saw are not practical except in special circumstances, or maybe for 'local' planning.

However, we have seen three methods that can be applied to 'real problems', i.e., in high-dimensional C-spaces and for arbitrary \mathcal{W} and \mathcal{A}

- PRM and OBPRM only need collision-detection routines specific to problem instance
- RPP needs collision-detection and we also have to define potential function (but don't need to explicitly compute its gradient – use \mathbf{U} directly)
- in terms of precomputation, as stated, both PRM and OBPRM build roadmaps in advance
 - not strictly necessary, can be done on-line
- RPP precomputes workspace potentials for control points on robot
- PRM and OBPRM are probably more suitable as 'global' planners for more problem types, RPP is very good method for certain types of problems