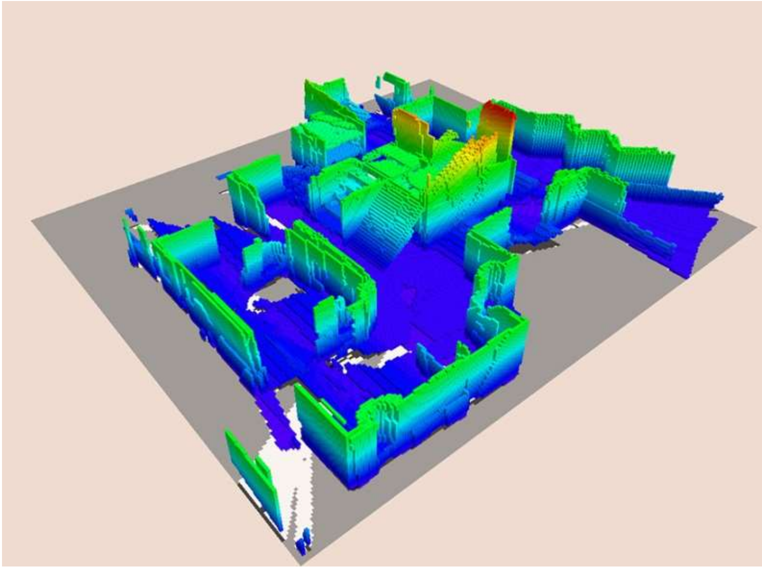
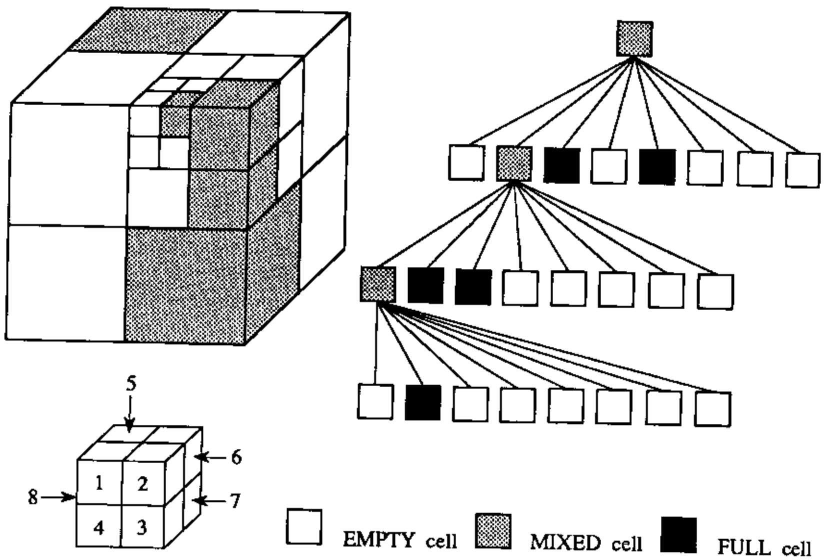
Path Planning in 3D Maps

3D Occupancy Grid

- 3D regular grid
  - occupancy, free space, unknown
  - x,y,z ~ metric Cartesian coordinates
  - simple but memory intensive (predominant for 2D, less for 3D)
- octree
  - recursive decomposition in 8 cells
  - each occupied cell further divided



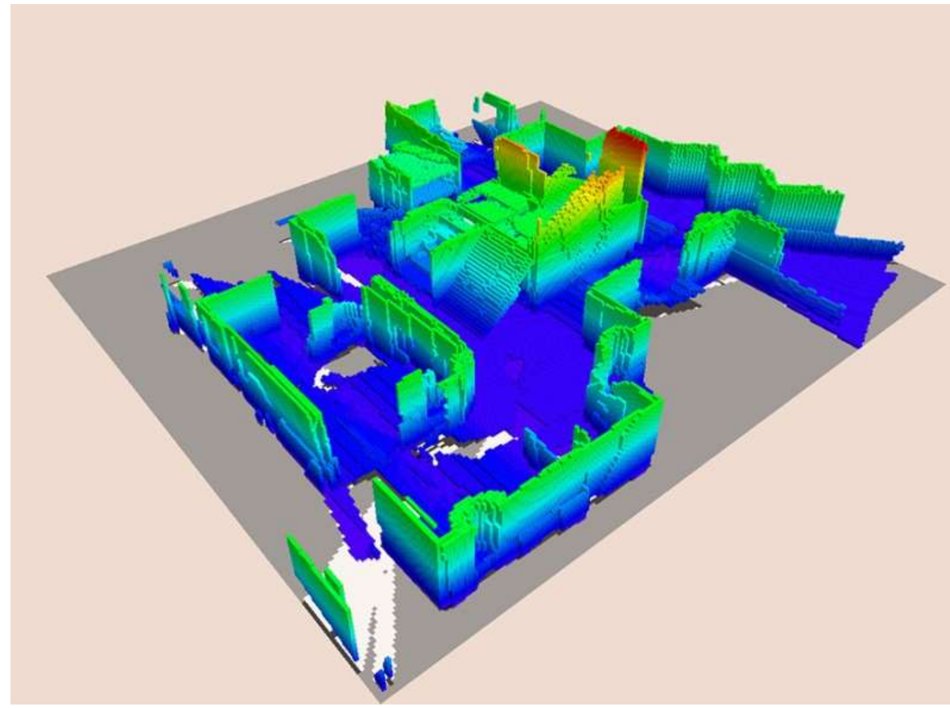


3D Occupancy Grid

3D path-planning

aerial & underwater systems

- just like in 2D
- roadmaps by adjacency
- A* or Dijkstra for planning



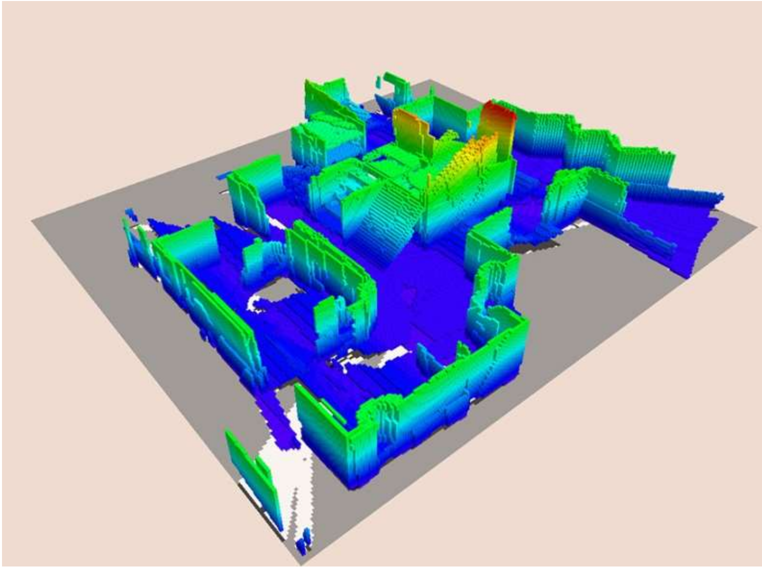
ground system

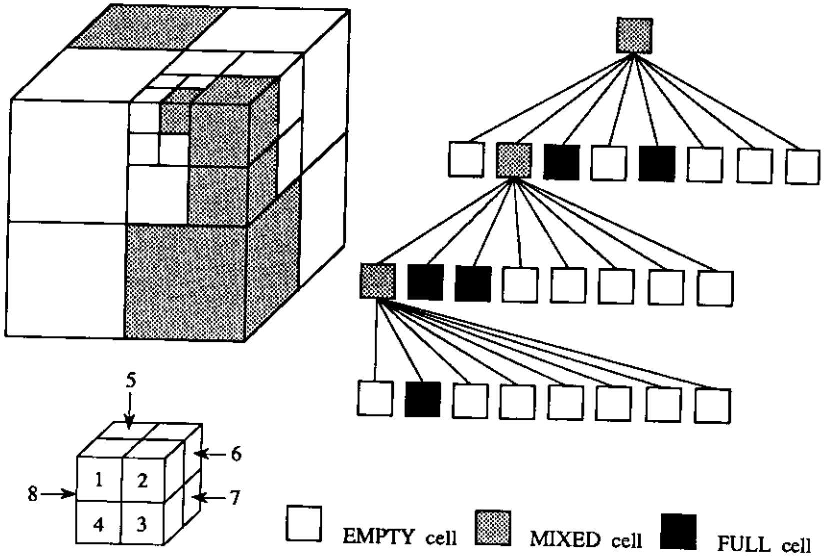
- drives on 2D surface
- down-project 3D to 2D or 2.5D

3D Occupancy Grid & Octree

Volumetric Representations

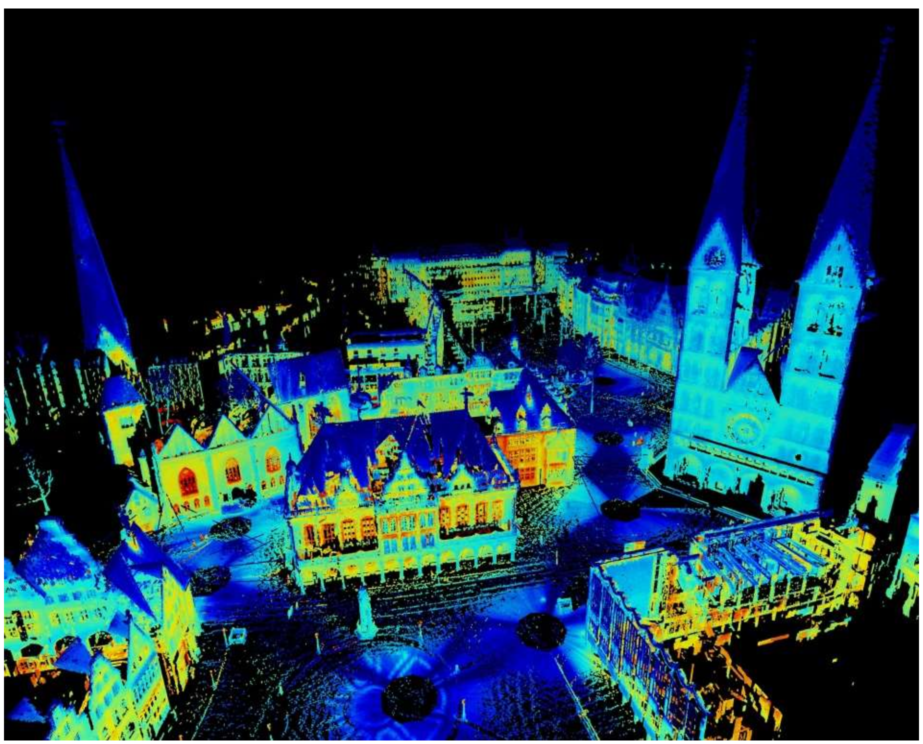
- 3D regular grid
  - simple but memory intensive
  - predominant for 2D, less for 3D
- octree
  - recursive decomposition in 8 cells
  - each occupied cell further divided
  - compact representation
  - but not straightforward to use for path-planning
    - can use adjacency and cell centers
    - for roadmap generation and A*
    - but does not lead to shortest paths (consider a large empty volume)

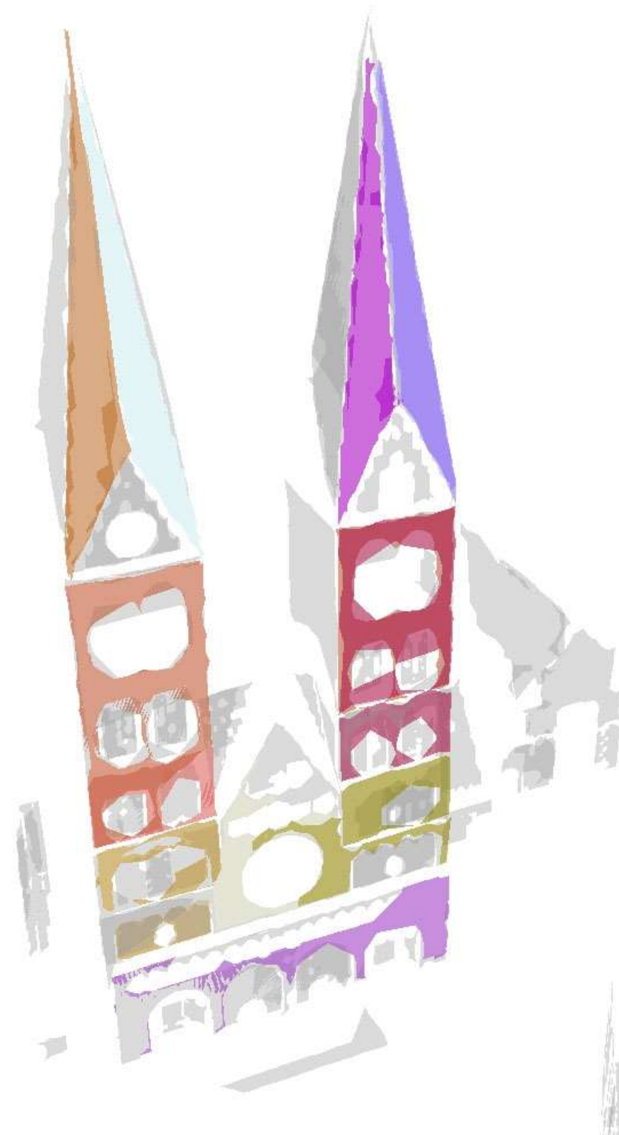




3D Surface Models

- in contrast to volumetric (grid or octree)
- surface representations, e.g.,
  - generate meshes from points
  - or fit (larger) surfaces, e.g., planes into the raw data of range sensors (point cloud)





Sampling-based Path Planning

aka Randomized Graph Search

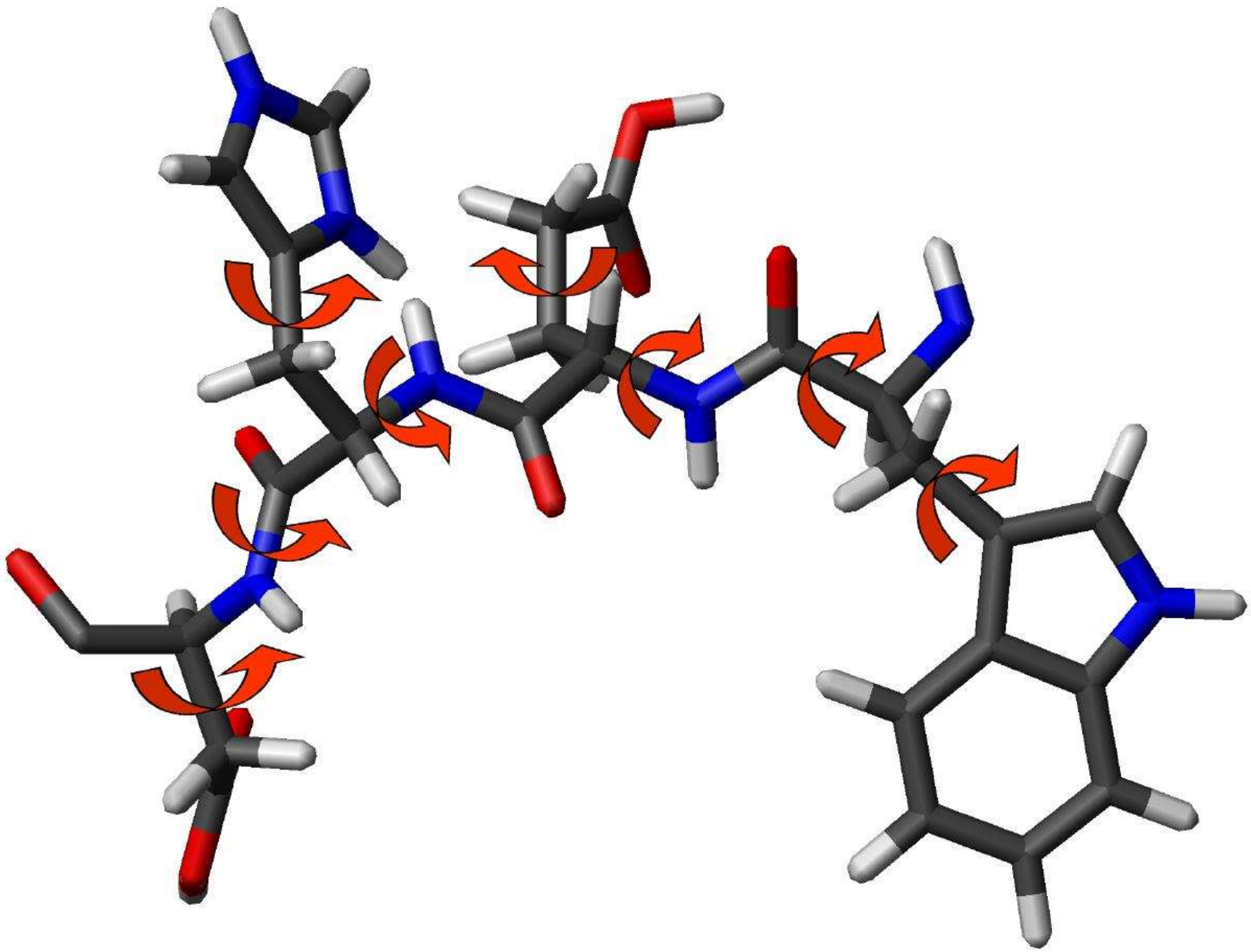
- maps can get large in 3D
  - and even already in 2D
  - especially for grids, and even quad/octrees
- plus even more so for multiple DoFs
  - robot-arms, bio-informatics, etc.
  - i.e., when planning in high-dimensional C-space

=> very large search space

also, how to handle surface representations?

Example: Bioinformatics

- motion planning with many DoF
- search whether there is a configuration of a molecule
- that “fits” to an other molecule (e.g., for drug design)



Probabilistic Roadmap (PRM)

Start with empty $R=(V,E)$

Generate a random free location c and add to V

Choose a subset V_c of candidate neighbors around c

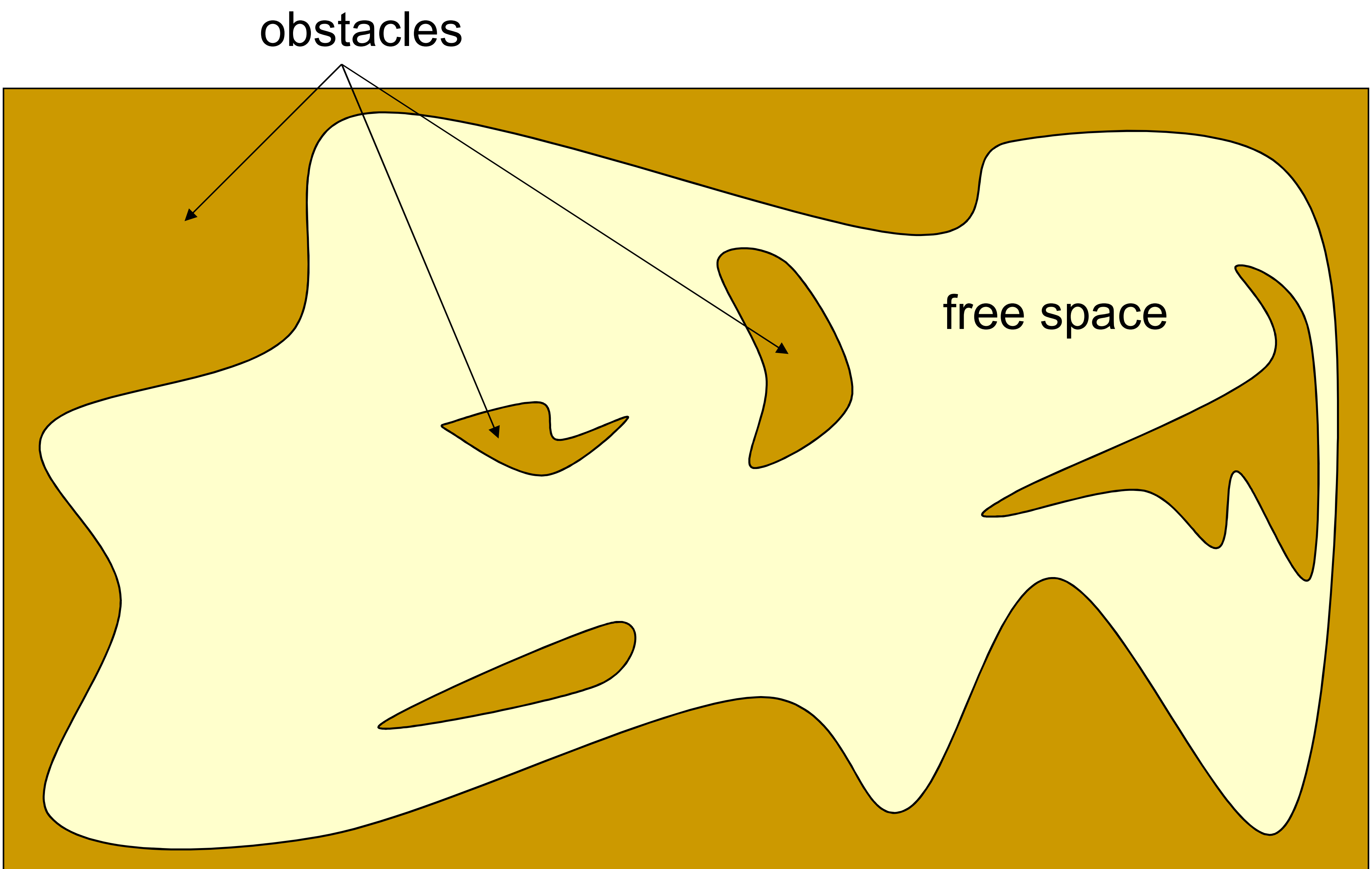
Try to connect c to each of selected nodes in V_c

- select only the nodes not graph-connected to c
- use maybe a local planner

Add the edge found to E

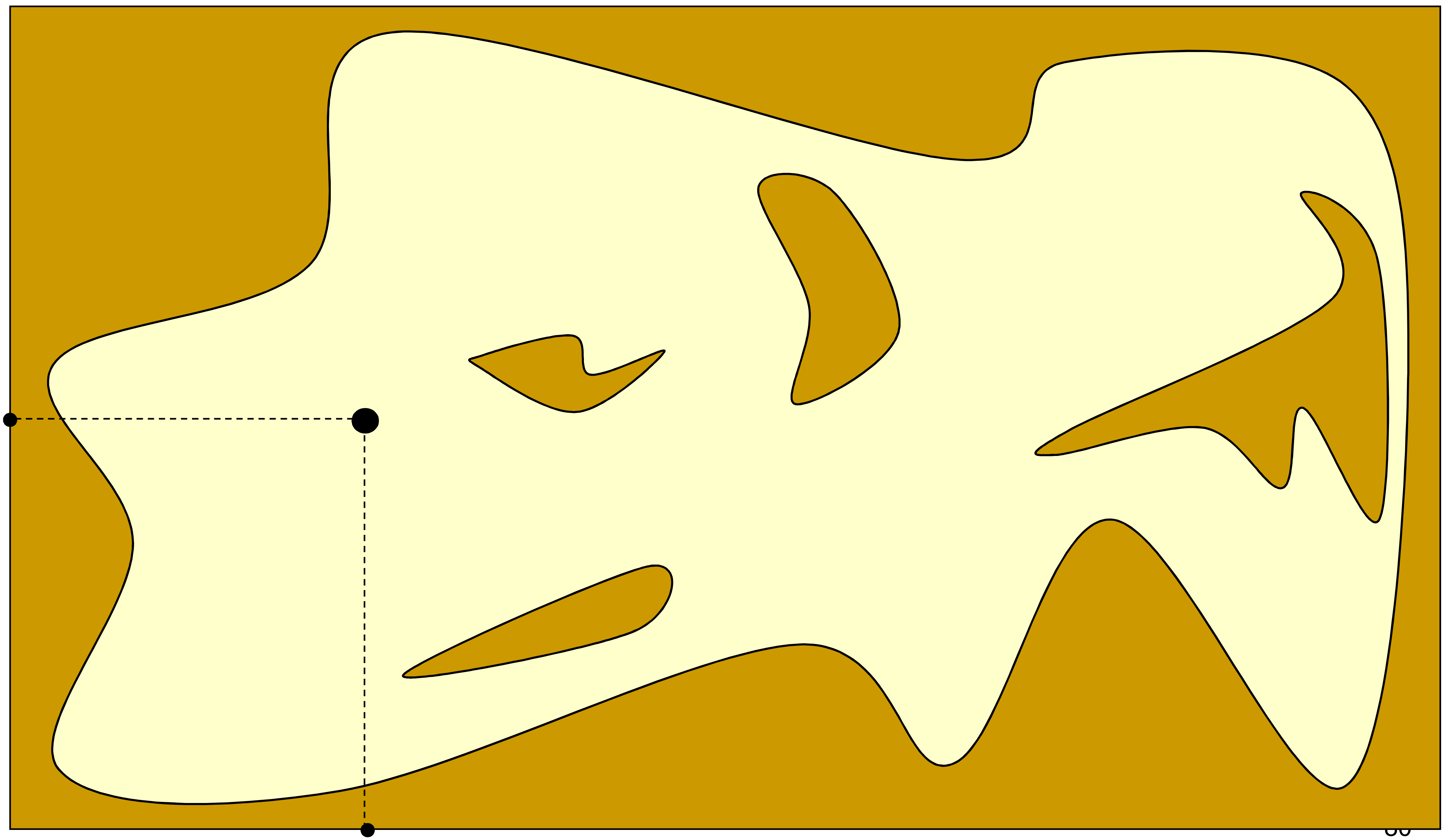
Repeat the above until satisfied

Probabilistic Roadmap (PRM)



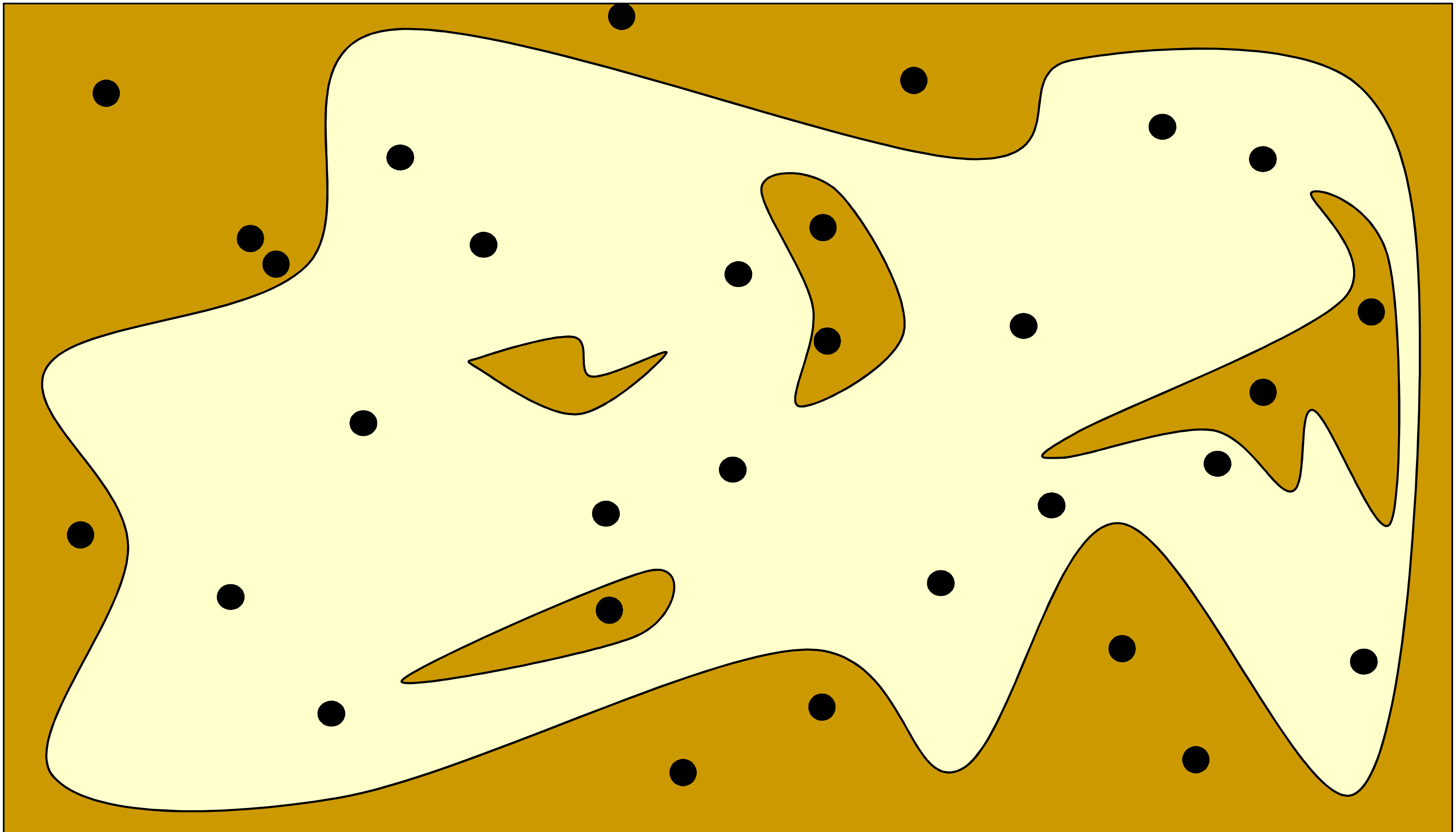
Probabilistic Roadmap (PRM)

Configurations are sampled by picking coordinates at random



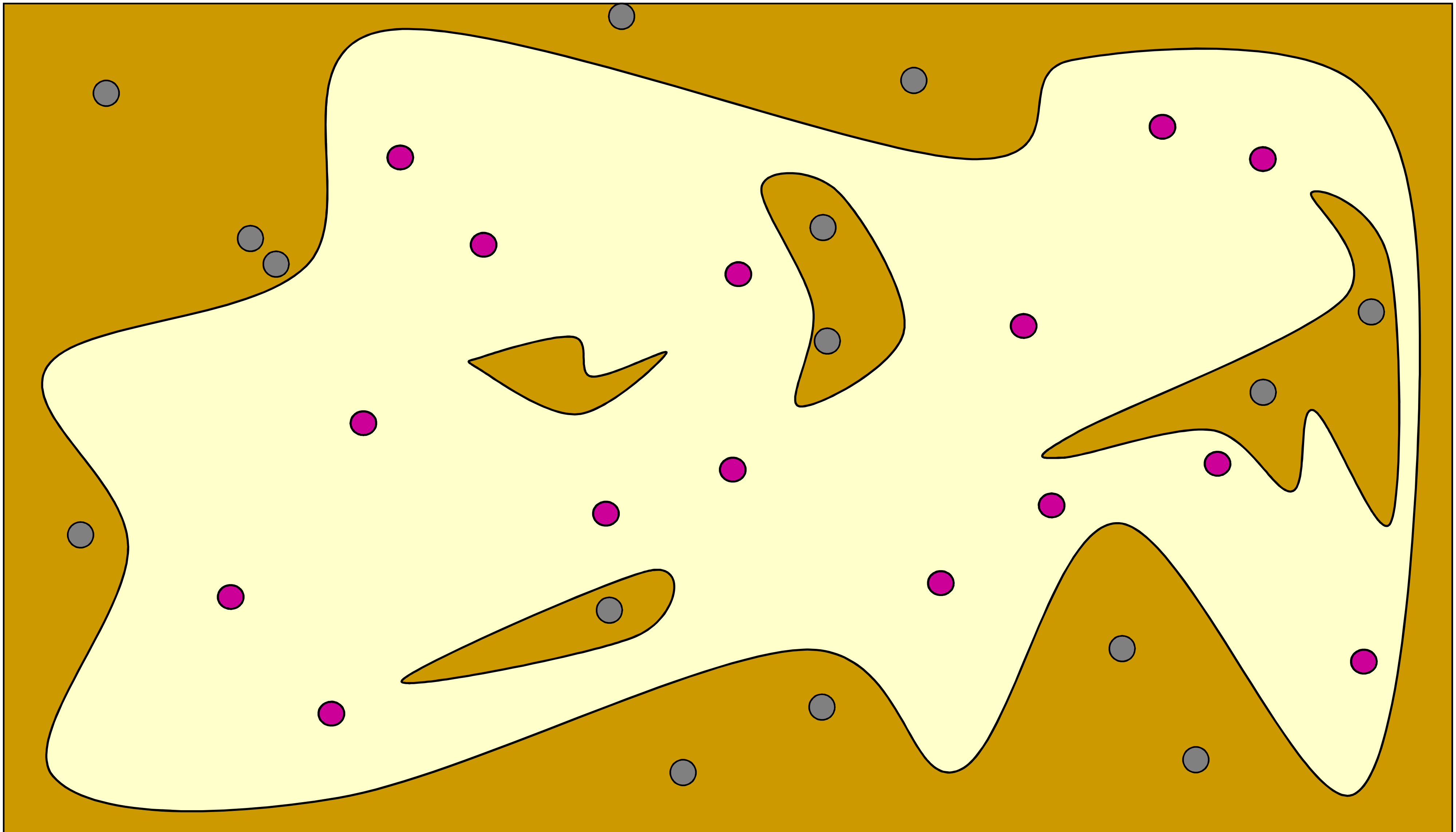
Probabilistic Roadmap (PRM)

Configurations are sampled by picking coordinates at random



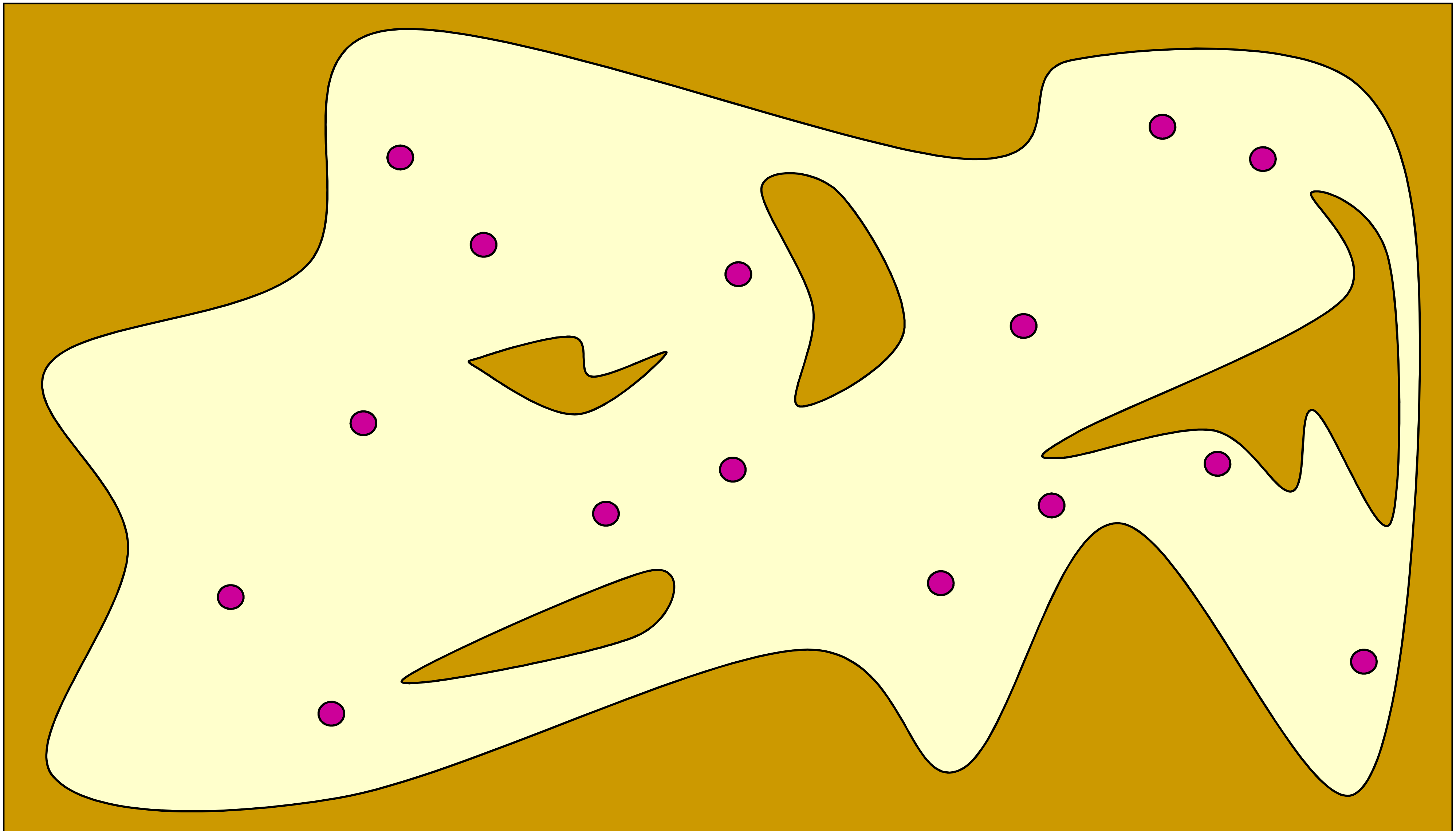
Probabilistic Roadmap (PRM)

Sampled configurations are tested for collision



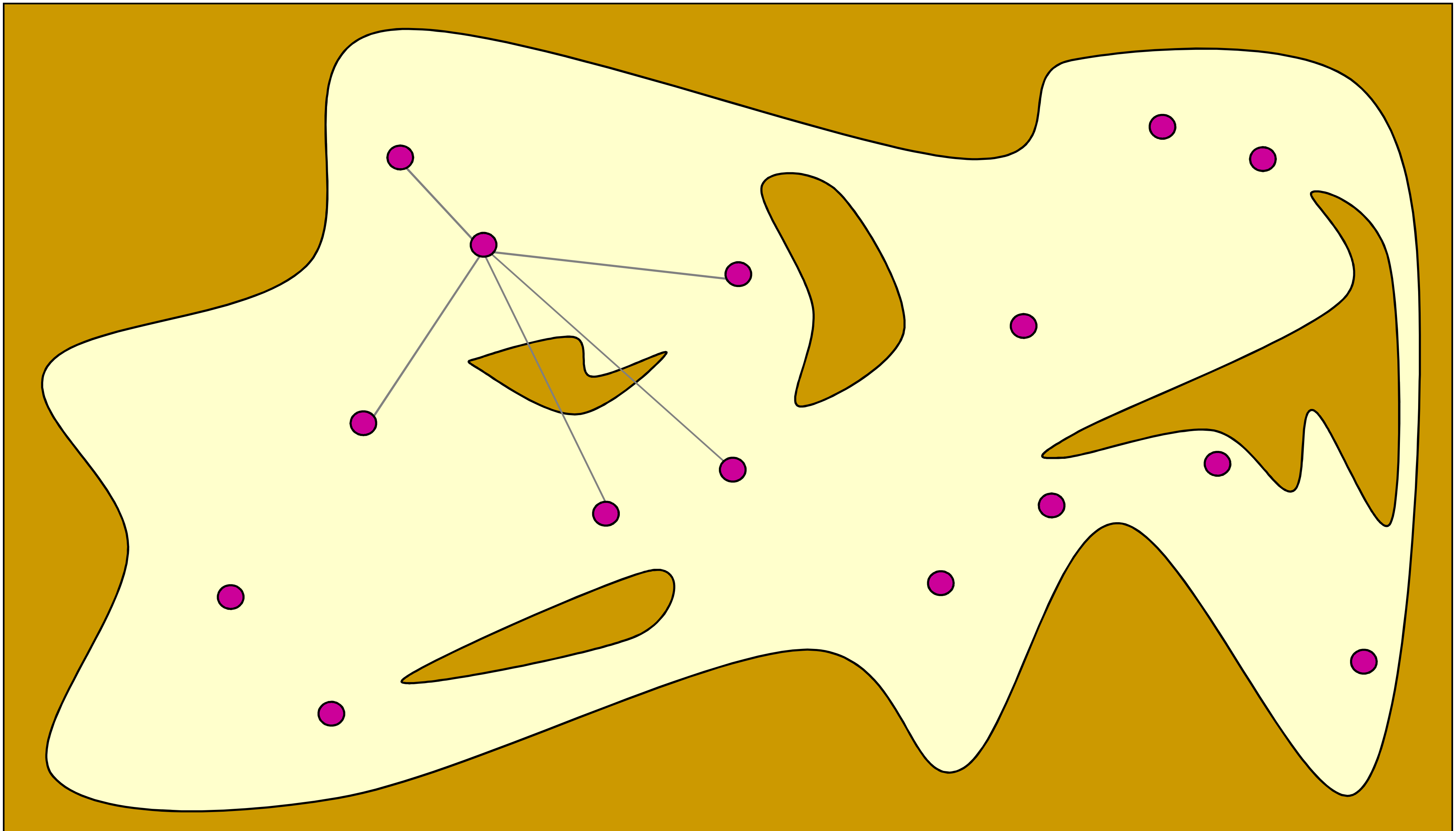
Probabilistic Roadmap (PRM)

The collision-free configurations are retained as milestones



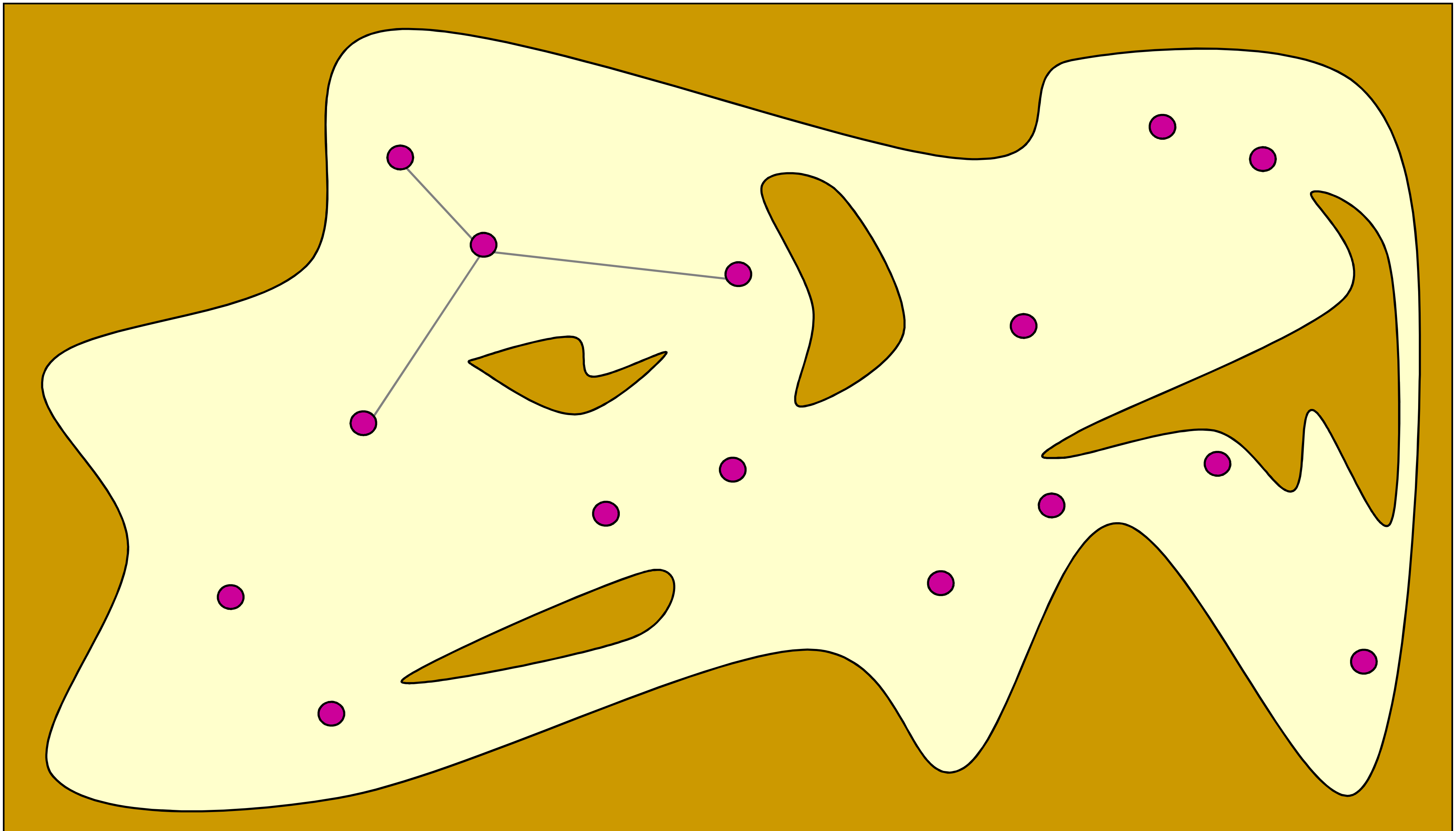
Probabilistic Roadmap (PRM)

Each milestone is linked by straight paths to its nearest neighbors



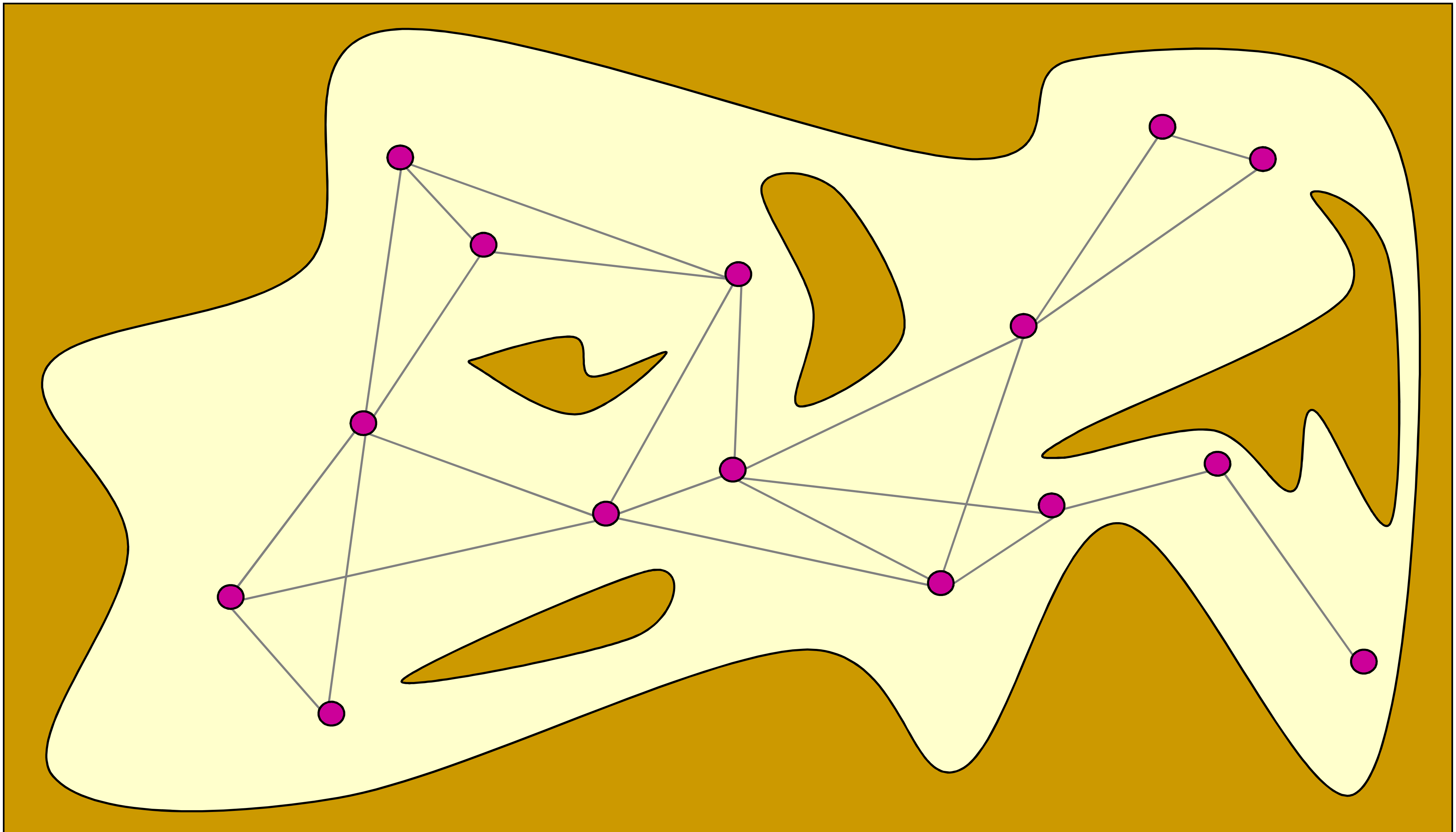
Probabilistic Roadmap (PRM)

Remove paths with collisions



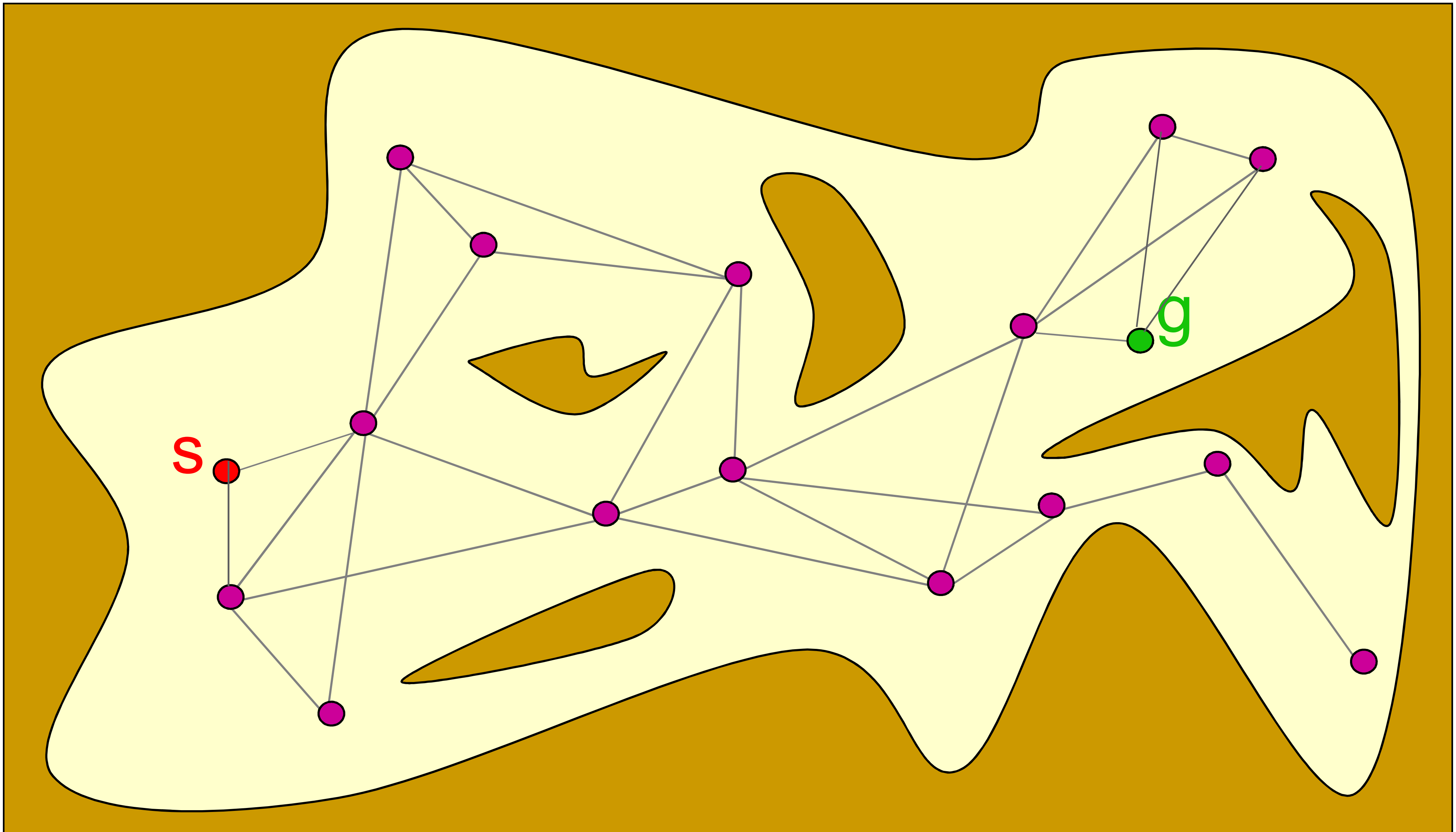
Probabilistic Roadmap (PRM)

The collision-free links are retained as local paths to form the PRM



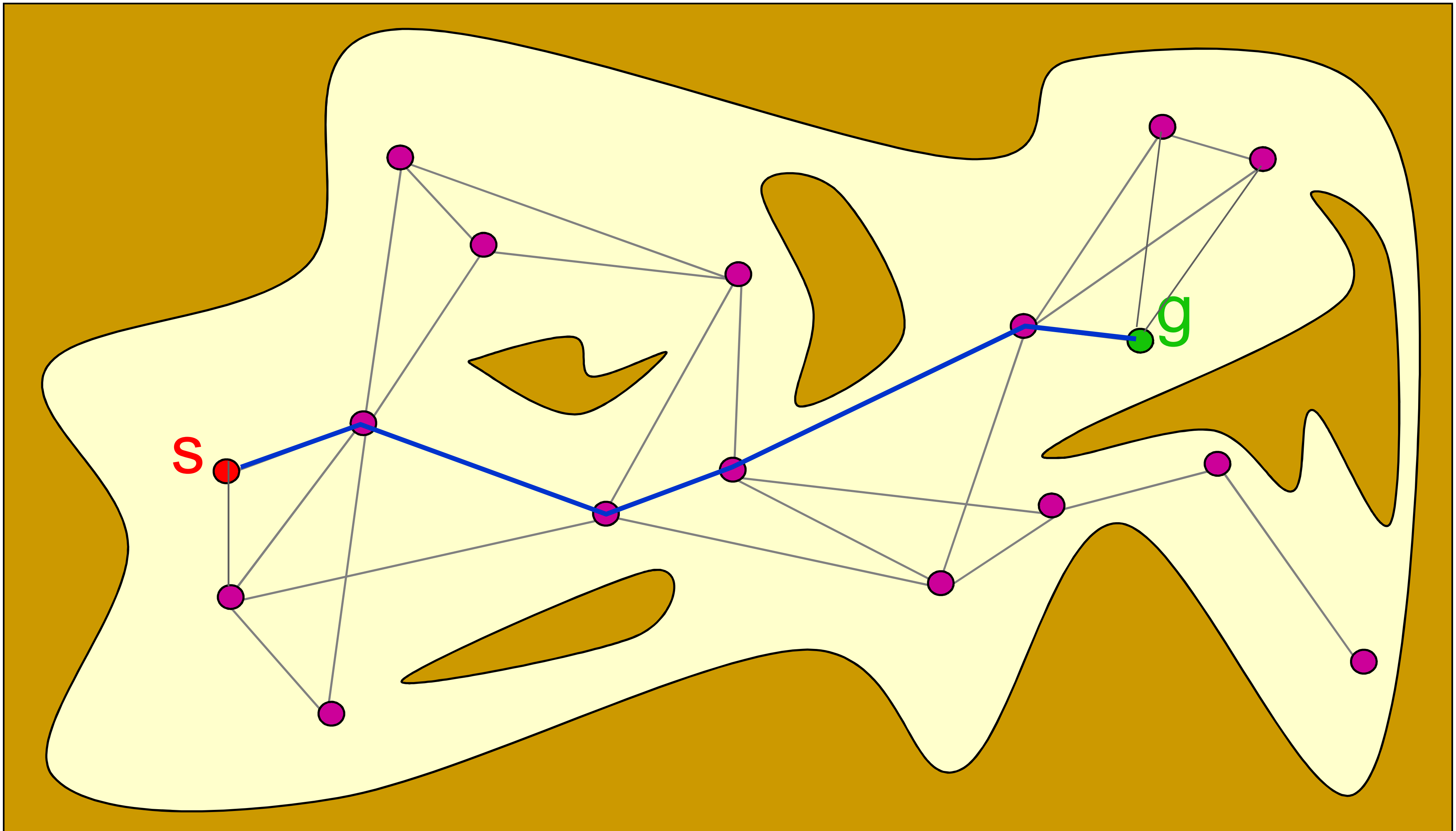
Probabilistic Roadmap (PRM)

The start and goal configurations are included as milestones



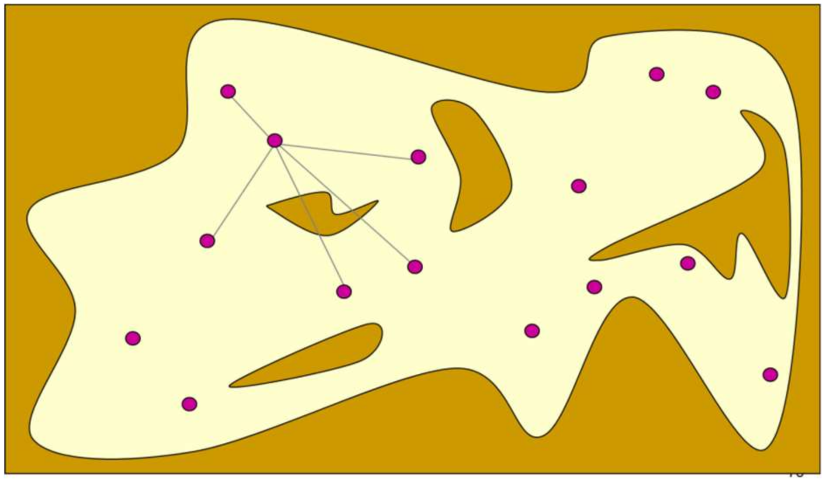
Probabilistic Roadmap (PRM)

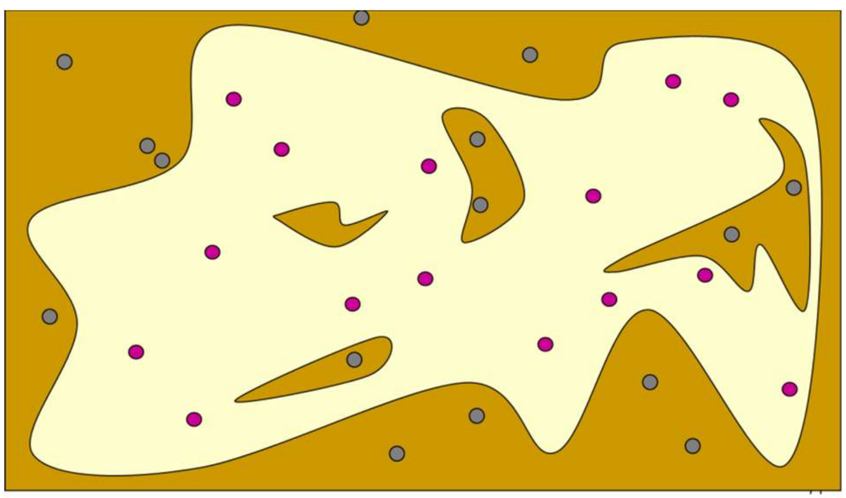
The PRM is searched for a path from s to g



PRM & Gridmaps

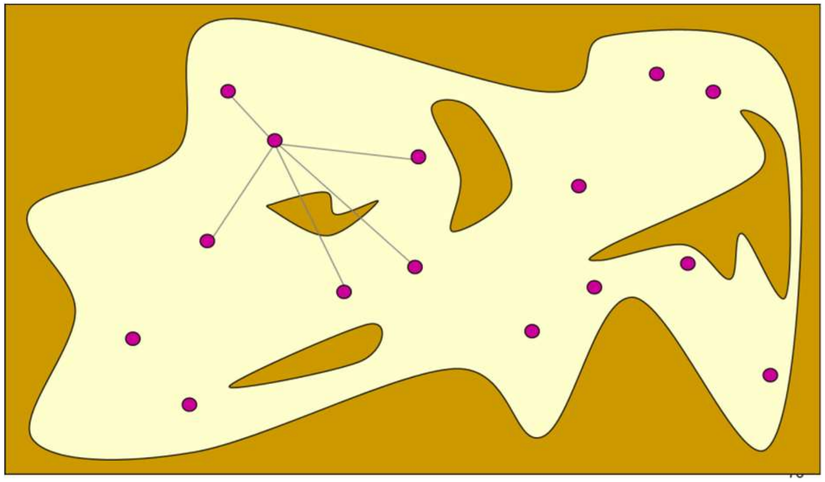
- "in free space?" check
  - trivial: corresponding cell marked *free* or not
  - note: most non-free cells of a volume marked as *unknown*
- straight line collisions
  - Bresenham line algorithm
- or use of local path-plan
  - e.g., A^* in bounded volume

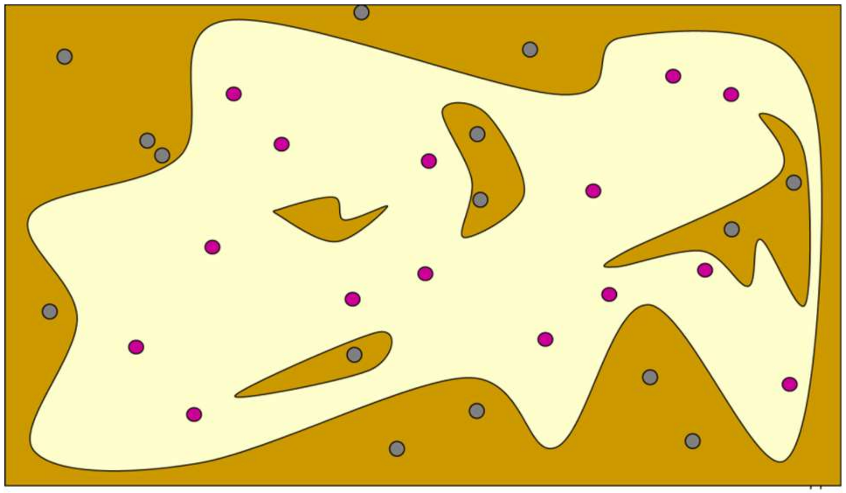




PRM & Surface Representations

- straight line collisions
  - with surfaces (polygons or meshes) are “easy” to check
  - i.e., part of “standard” computational geometry
- “in free space?” check
  - slightly more tricky
  - requires in- and outside of the surfaces (via surface normal)





(but both aspects out of scope of this lecture)

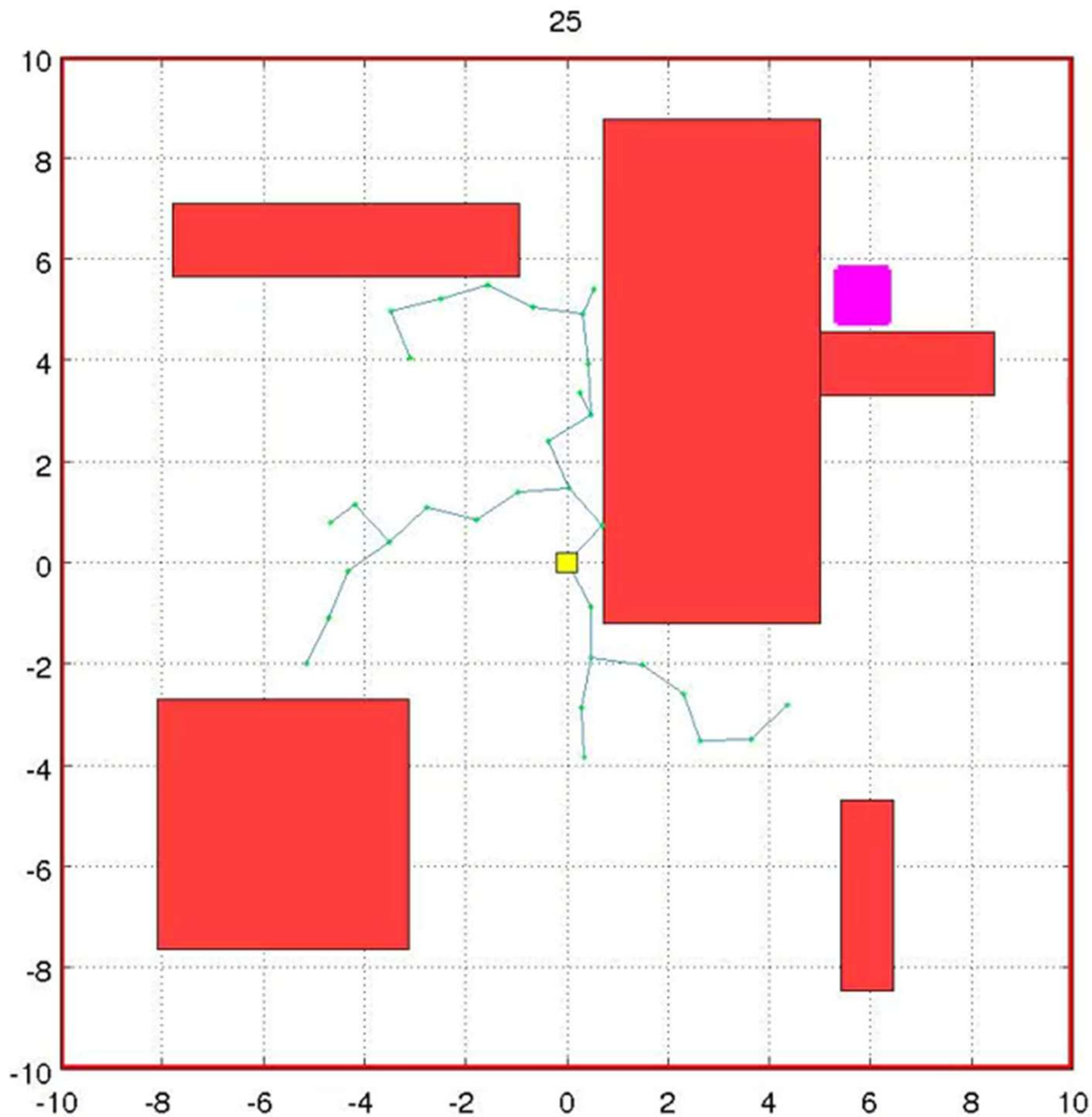
Rapidly Exploring Random Tree (RRT)

basic algorithm:

1. start with the initial configuration as root of tree
2. pick a random state in the configuration space
3. find the closest node in the tree
4. extend that node toward the state if possible
5. goto 2

(note: many variations exist)

Rapidly Exploring Random Tree (RRT)



Obstacle Space Revisited

Obstacle-Space

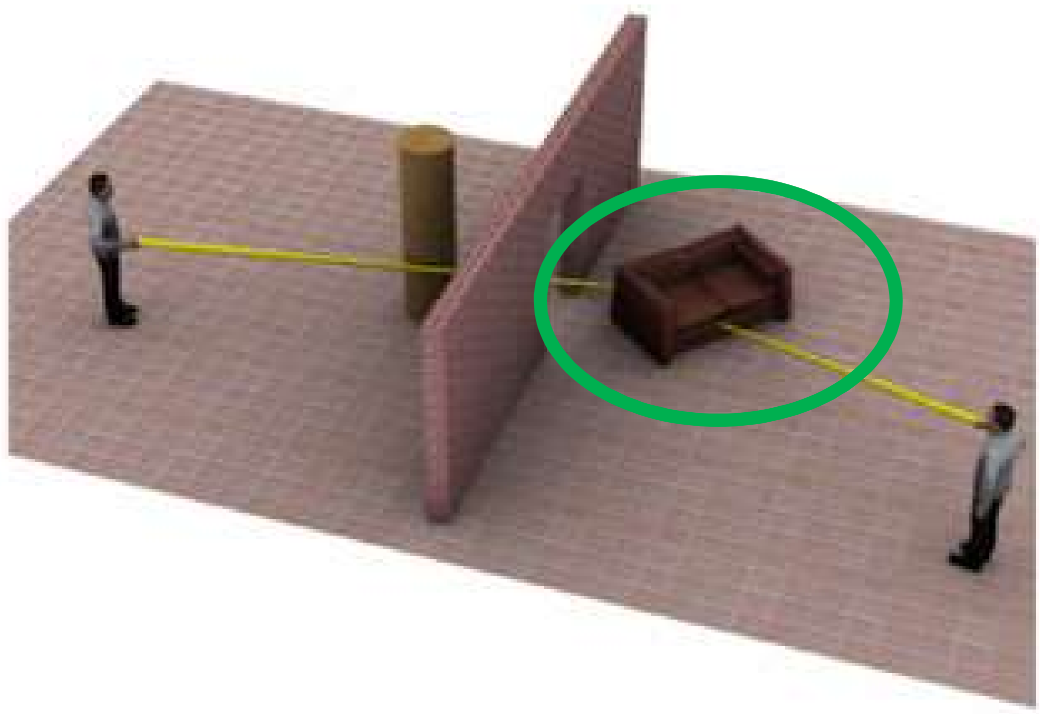
path-planning so far

- mobile systems
- with simple geometries
- i.e., bounding sphere for obstacle growing
- moving in 2D or 3D

what about
complex geometries, respectively complex motions?

Piano Mover's Problem

object shape matters in path-planning



simple obstacle growing
will not work in this case

Configuration Space (C-Space)

[re-cap]

partitioned into

- free configurations (aka free space): robot and obstacles do not overlap
- contact configurations (aka contact space): robot and obstacles touch
- blocked configurations (aka obstacle space): robot and obstacles overlap

Obstacle Space

C-obstacles can get quite tricky

- option1: general (simple) approach
  - discretize DoF & enumerate them
  - check for collision for each configuration

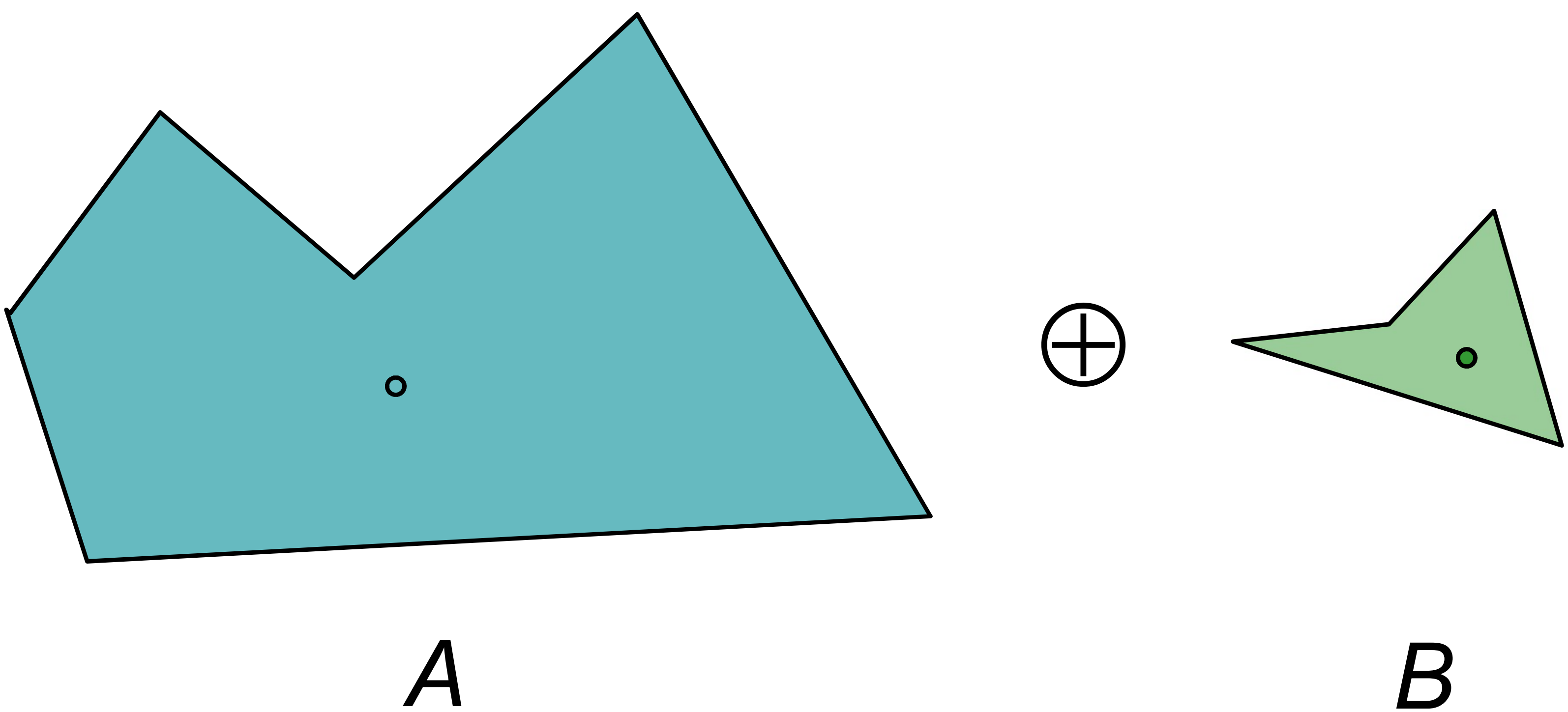
- option2: Minkowski sum
  - works for rigid bodies in 2D and 3D
  - under translations

Minkowski Sum

$$A \oplus B = \{a + b \mid a \in A, b \in B\}$$

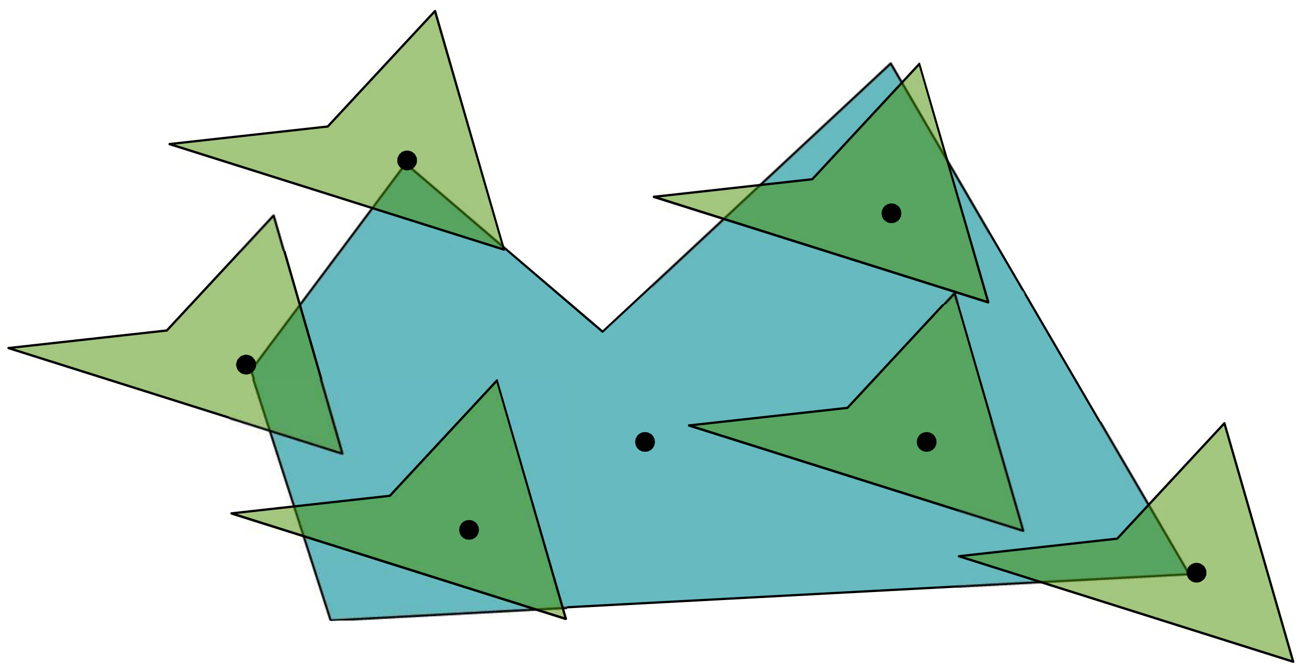
A, B sets

a, b vectors

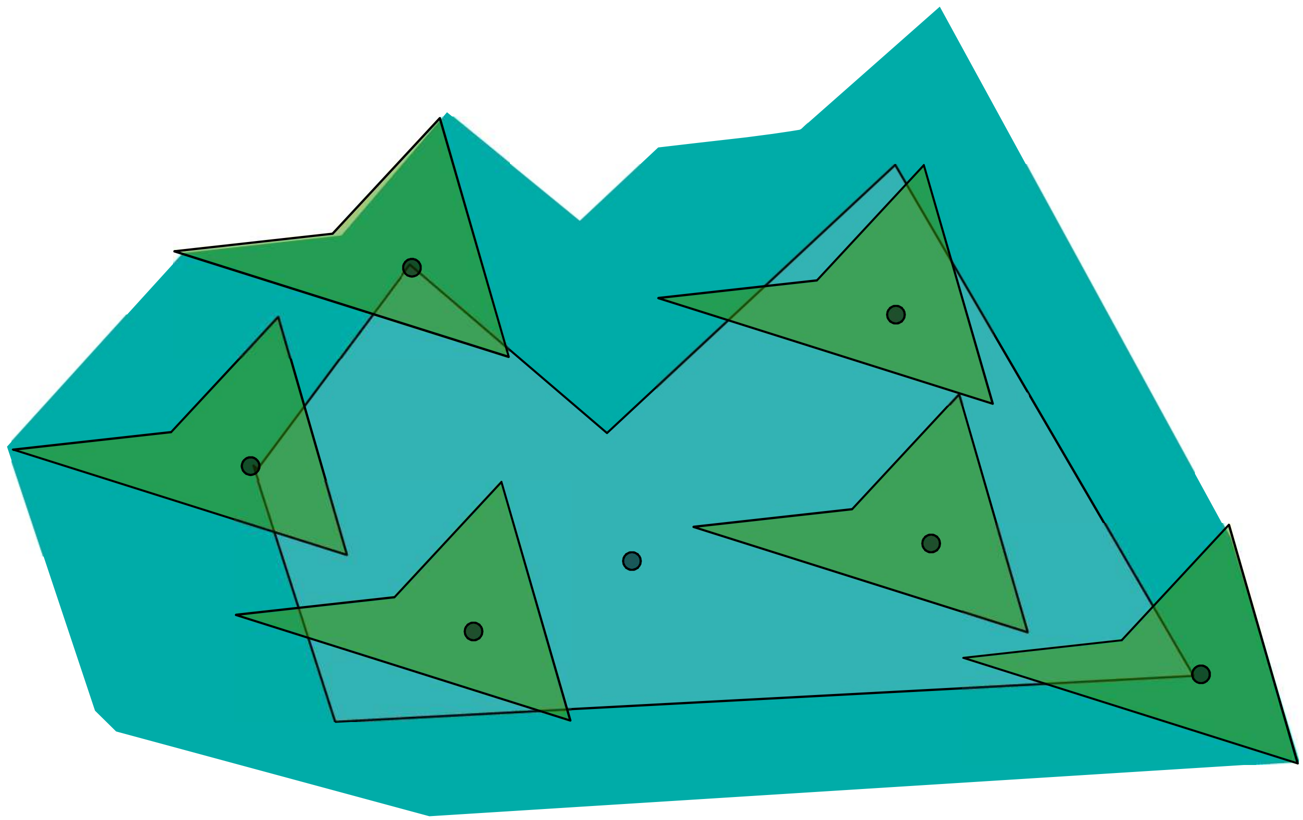


Minkowski Sum

$$A \oplus B = \{a + b \mid a \in A, b \in B\}$$



Minkowski Sum



Minkowski Sum

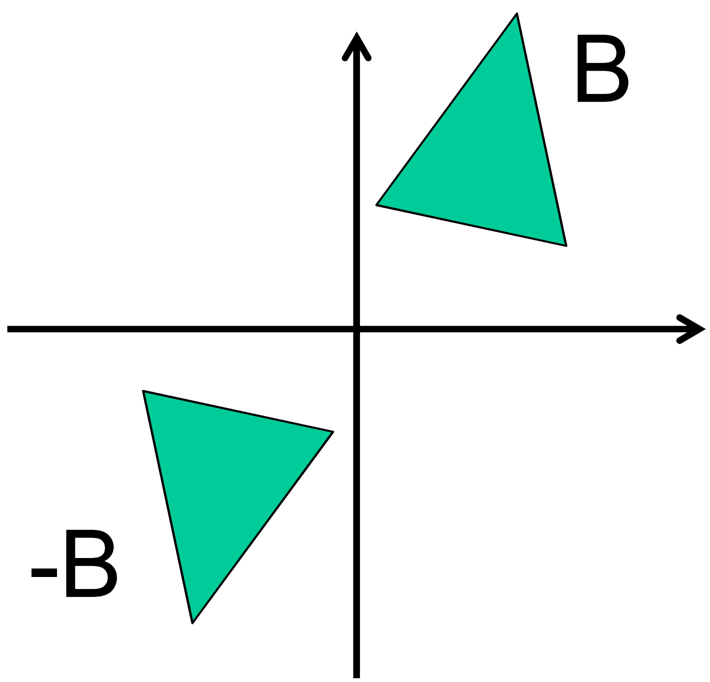
$$A \oplus B = \{a + b \mid a \in A, b \in B\}$$

Minkowski Sum

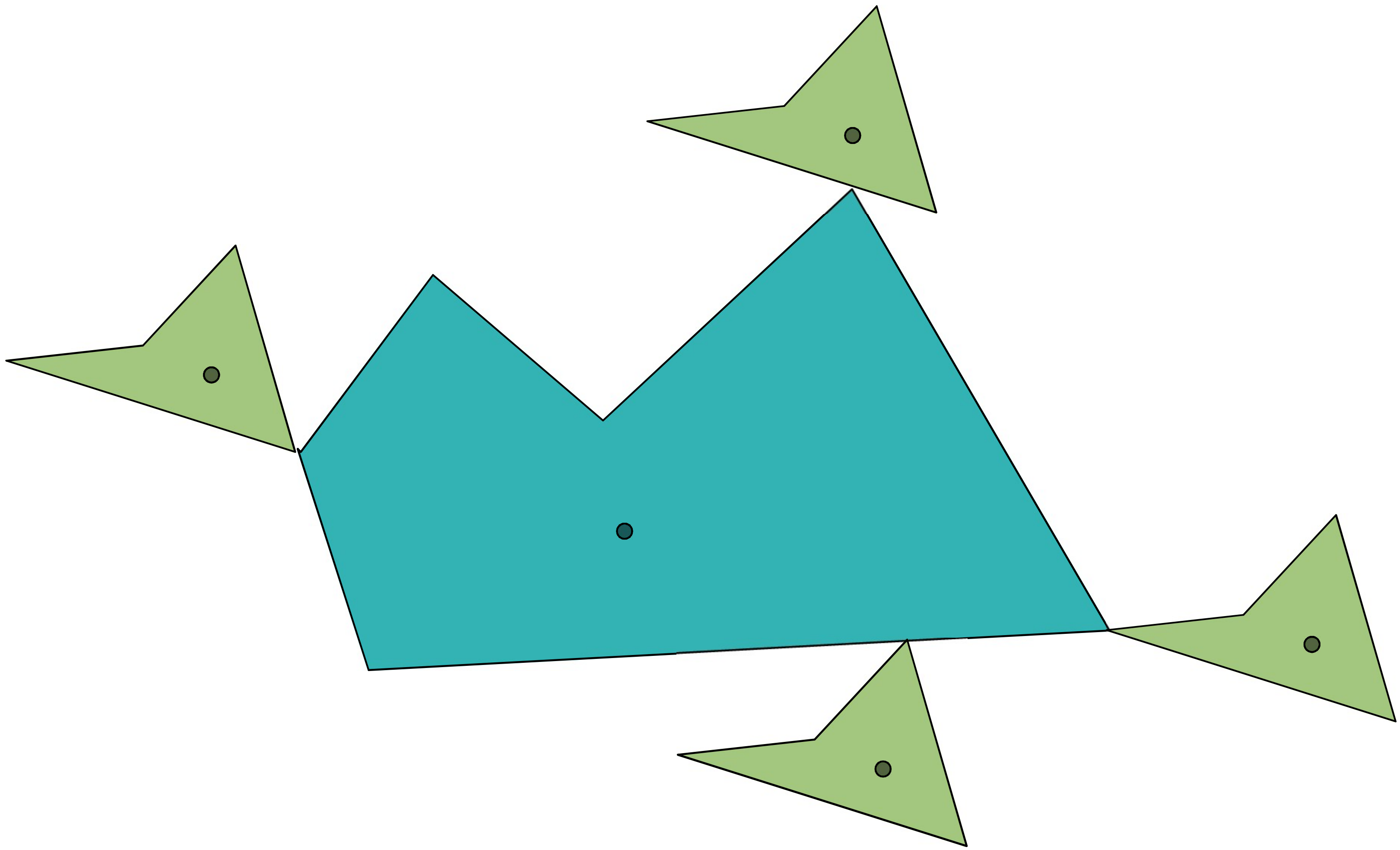
Minkowski difference

$$A \ominus B = \{a - b \mid a \in A, b \in B\}$$
$$= A \oplus (-B)$$

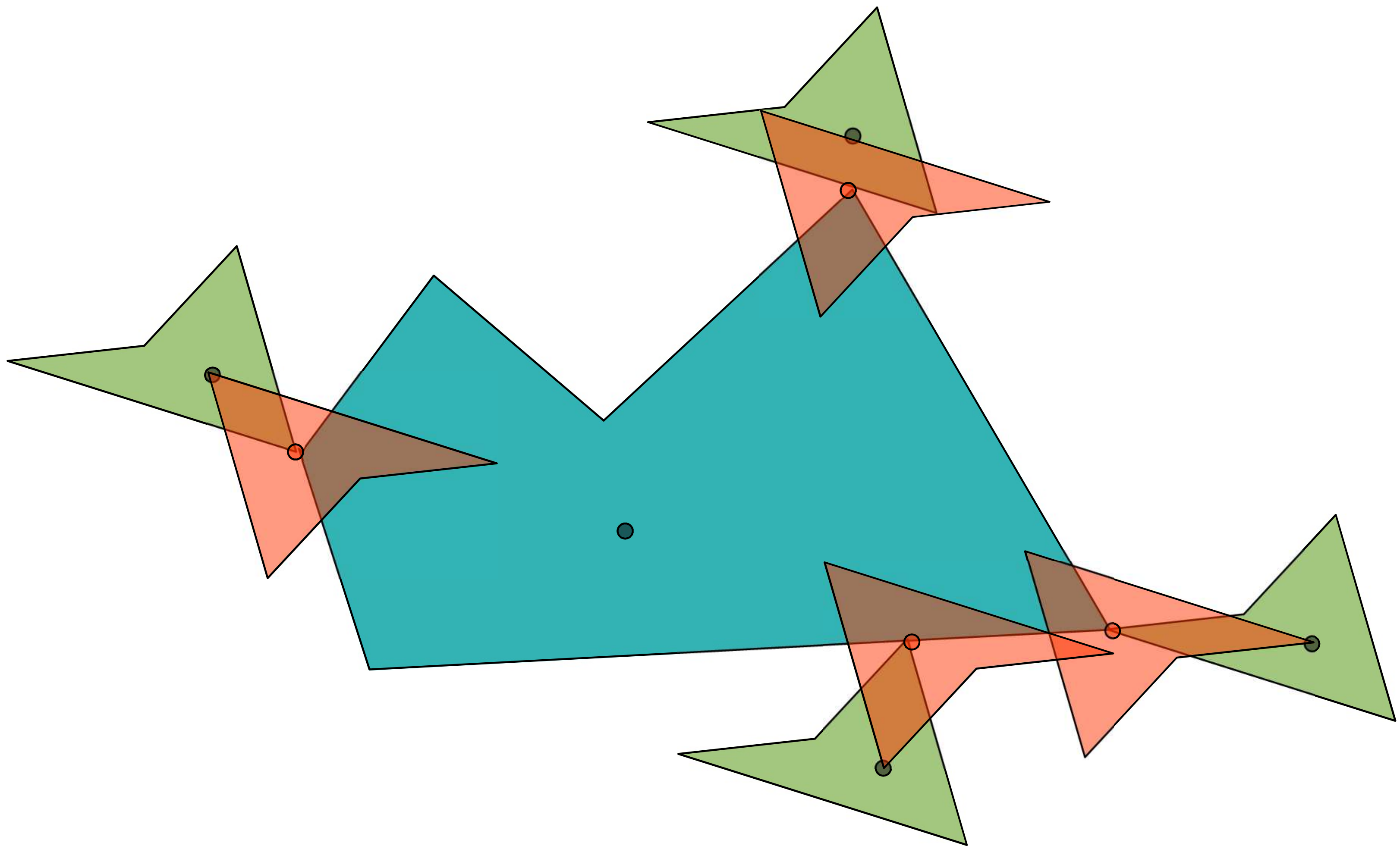
$$-B = \{-b \mid b \in B\}$$



Tracing Out Collision Possibilities



Minkowski Difference

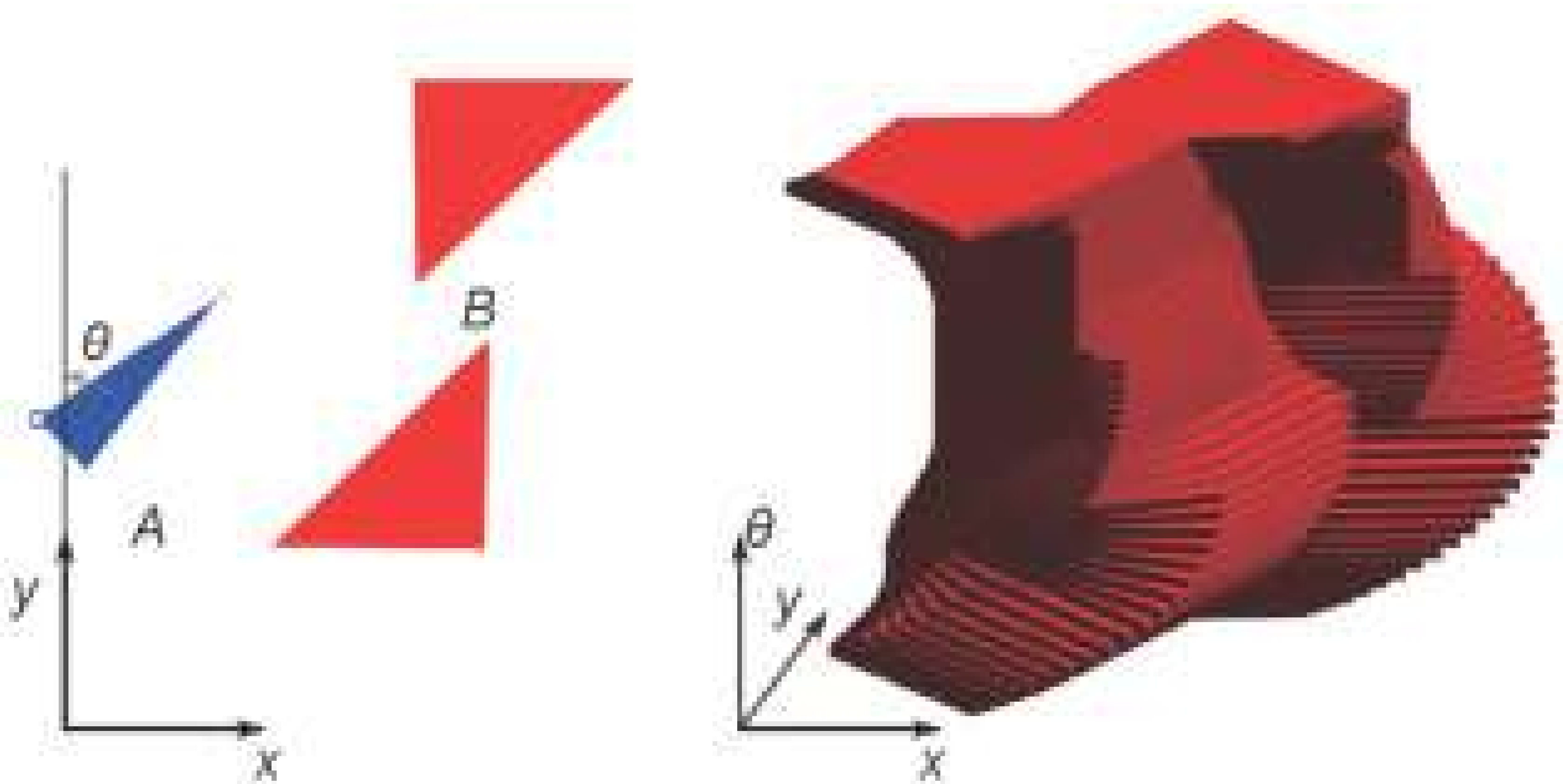


C-obstacle = obstacle minus rigid body

(can be efficiently computed for polygons)

What about rotations?

- discretize rotations
- compute Minkowski diff for each angle



Generating C-Obstacle can be tough...

- methods exist including rotations and kinematic chains (multiple links & joints like robot arms)
- but computational complexity is an issue
- hence often Randomized Graph Search
  - e.g., PRM or RRT
  - with geometric collision tests of configurations in Cartesian space (e.g., via computational geometry)