Robotics
Spring 2023

Abhishek Gupta

TAs: Yi Li, Srivatsa GS

Courtesy of Maxim Likhachev, CMU, Dieter Fox, UW, Pieter Abbeel, UC Berkeley

Recap: Course Overview

- Filtering/Smoothing
- Localization
- Mapping
- SLAM

- Search
- Motion Planning
- TrajOpt
- Stability/Certification

- MDPs and RL
- Imitation Learning
- Solving POMDPs

Lecture Outline

Incremental Search – LPA*

Sampling Based Motion Planning - PRMs

RRT and RRT*

Informed Search Attempt 2: A* Search

Choose the next node to expand as the one that has the lowest heuristic + cost so far

Greedy best first

Uniform cost search

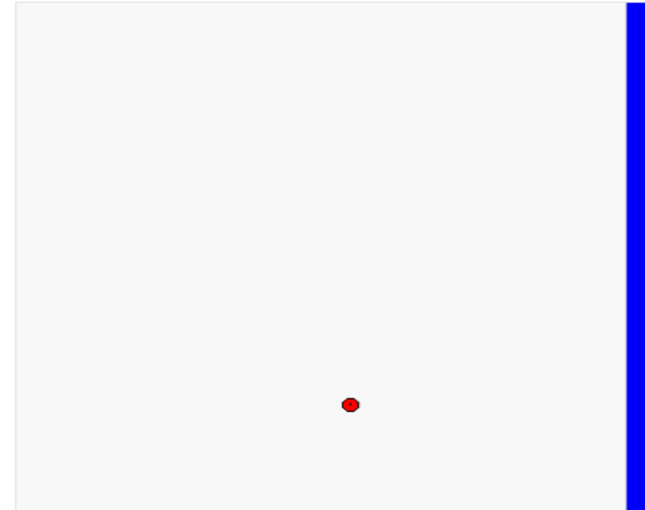
```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

Incremental version of A* (LPA*)

- Robot needs to re-plan whenever
  - new information arrives (partially-known environments or/and dynamic environments)
  - robot deviates off its path

ATRV navigating
initially-unknown environment

planning map and path

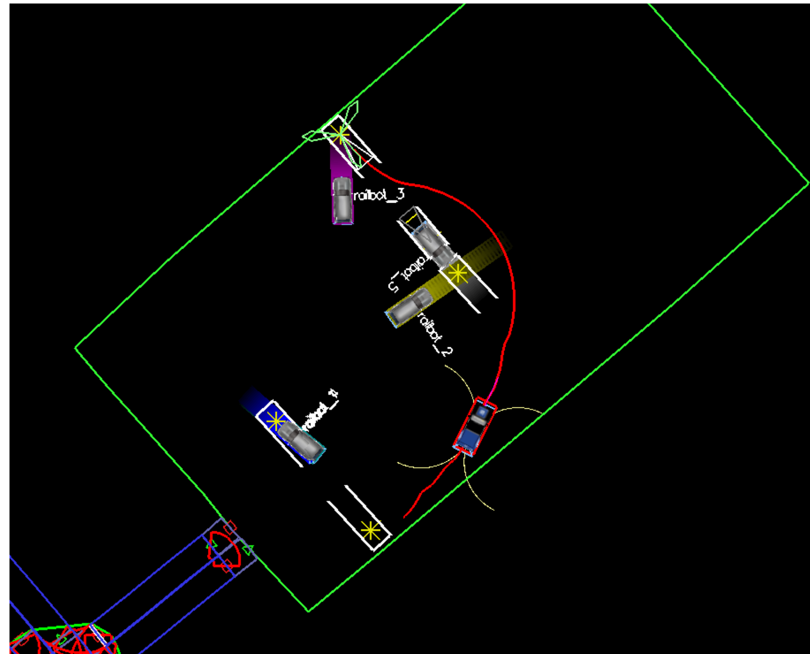


Incremental version of A* (LPA*/D* Lite)

- Robot needs to re-plan whenever
  - new information arrives (partially-known environments or/and dynamic environments)
  - robot deviates off its path

incremental planning (re-planning):
reuse of previous planning efforts

planning in dynamic environments



Tartanracing, CMU

Motivation for Incremental Version of A*

- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially

14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	6	6	6
14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	5	5	5
14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	3	3	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	1	2	3
14	13	12	11		9		7	6	5	4	3	2	1	s_{goal}	1	2	3
					9				5	4	3	2	1	1	1	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9				5	4	3	3	3	3	3	3	3
14	13	12	11	10	10		7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	11	11		7	6	5	5	5	5	5	5	5	5	5
14	13	12	12	12	12		7	6	6	6	6	6	6	6	6	6	6
					13		7	7	7	7	7	7	7	7	7	7	7
18	s_{start}	16	15	14	14		8	8	8	8	8	8	8	8	8	8	8

cost of least-cost paths to s_{goal} after the door turns out to be closed

14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	6	6	6
14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	5	5	5
14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	3	3	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	1	2	3
14	13	12	11		9		7	6	5	4	3	2	1	s_{goal}	1	2	3
					10				5	4	3	2	1	1	1	2	3
15	14	13	12	11	11		7	6	5	4	3	2	2	2	2	2	3
15	14	13	12	12	s_{start}				5	4	3	3	3	3	3	3	3
15	14	13	13	13	13		7	6	5	4	4	4	4	4	4	4	4
15	14	14	14	14	14		7	6	5	5	5	5	5	5	5	5	5
15	15	15	15	15	15		7	6	6	6	6	6	6	6	6	6	6
					16		7	7	7	7	7	7	7	7	7	7	7
21	20	19	18	17	17		8	8	8	8	8	8	8	8	8	8	8

Motivation for Incremental Version of A*

- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially

14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	6	6	6
14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	5	5	5
14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	3	3	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	1	2	3
14	13	12	11		9		7	6	5	4	3	2	1	s_{goal}	1	2	3
					9				5	4	3	2	1	1	1	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9				5	4	3	3	3	3	3	3	3
14	13	12	11	10	10		7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	11	11		7	6	5	5	5						
14	13	12	12	12	12		7	6									
					13		7										
18	s_{start}	16	15	14	14		8										

How to reuse these g-values from one search to another? – incremental A*

cost of least-cost paths to s_{goal} after the door turns out to be closed

14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	6	6	6
14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	5	5	5
14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	3	3	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	1	2	3
14	13	12	11		9		7	6	5	4	3	2	1	s_{goal}	1	2	3
					10				5	4	3	2	1	1	1	2	3
15	14	13	12	11	11		7	6	5	4	3	2	2	2	2	2	3
15	14	13	12	12	s_{start}				5	4	3	3	3	3	3	3	3
15	14	13	13	13	13		7	6	5	4	4	4	4	4	4	4	4
15	14	14	14	14	14		7	6	5	5	5	5	5	5	5	5	5
15	15	15	15	15	15		7	6	6	6	6	6	6	6	6	6	6
					16		7	7	7	7	7	7	7	7	7	7	7
21	20	19	18	17	17		8	8	8	8	8	8	8	8	8	8	8

Motivation for Incremental Version of A*

- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially

14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	6	6	6
14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	5	5	5
14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	3	3	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	1	2	3
14	13	12	11		9		7	6	5	4	3	2	1	s_{goal}	1	2	3
					9				5	4	3	2	1	1	1	2	3
14	13	12	11	10	9	8	7	6	5	4	3						
14	13	12	11	10	9												
14	13	12	11	10	10												
14	13	12	11	11	11												
14	13	12	12	12	12												
					13												
18	s_{start}	16	15	14	14												

Would # of changed g-values be very different for forward A*?

cost of least-cost paths to s_{goal} after the door turns out to be closed

14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	6	6	6
14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	5	5	5
14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	3	3	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	1	2	3
14	13	12	11		9		7	6	5	4	3	2	1	s_{goal}	1	2	3
					10				5	4	3	2	1	1	1	2	3
15	14	13	12	11	11		7	6	5	4	3	2	2	2	2	2	3
15	14	13	12	12	s_{start}				5	4	3	3	3	3	3	3	3
15	14	13	13	13	13		7	6	5	4	4	4	4	4	4	4	4
15	14	14	14	14	14		7	6	5	5	5	5	5	5	5	5	5
15	15	15	15	15	15		7	6	6	6	6	6	6	6	6	6	6
					16		7	7	7	7	7	7	7	7	7	7	7
21	20	19	18	17	17		8	8	8	8	8	8	8	8	8	8	8

Motivation for Incremental Version of A*

- Reuse state values from previous searches

cost of least-cost paths to s_{goal} initially

14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	6	6	6
14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	5	5	5
14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	3	3	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	1	2	3
14	13	12	11		9		7	6	5	4	3	2	1	s_{goal}	1	2	3
					9				5	4	3	2	1	1	1	2	3
14	13	12	11	10	9	8	7	6	5	4	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
14	13	12	11	10	9		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
14	13	12	11	10	10	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
14	13	12	11	11	11	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
14	13	12	12	12	12	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
					13	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
18	s_{start}	16	15	14	14		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Any work needs to be done if robot deviates off its path?

cost of least-cost paths to s_{goal} after the door [illegible] out to be closed

14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	6	6	6
14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	5	5	5
14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	4	4	4
14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	3	3	3
14	13	12	11	10	9	8	7	6	5	4	3	2	2	2	2	2	3
14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	1	2	3
14	13	12	11		9		7	6	5	4	3	2	1	s_{goal}	1	2	3
					10				5	4	3	2	1	1	1	2	3
15	14	13	12	11	11		7	6	5	4	3	2	2	2	2	2	3
15	14	13	12	12	s_{start}				5	4	3	3	3	3	3	3	3
15	14	13	13	13	13		7	6	5	4	4	4	4	4	4	4	4
15	14	14	14	14	14		7	6	5	5	5	5	5	5	5	5	5
15	15	15	15	15	15		7	6	6	6	6	6	6	6	6	6	6
					16		7	7	7	7	7	7	7	7	7	7	7
21	20	19	18	17	17		8	8	8	8	8	8	8	8	8	8	8

Incremental Version of A* - D* lite

- Reuse state values from previous searches

initial search by backwards A*

s_{goal}

s_{start}

initial search by D* Lite

s_{goal}

s_{start}

second search by backwards A*

s_{goal}

s_{start}

second search by D* Lite

s_{goal}

s_{start}

Key idea of LPA*

Reuse as many g-values as possible from previous search

Idea 1: Introduce an idea of g(s) and rhs(s), at convergence these are equal

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{start} \\ \min_{s' \in pred(s)}(g(s') + c(s', s)) & \text{otherwise.} \end{cases}$$

$$g^*(s) = \begin{cases} 0 & \text{if } s = s_{start} \\ \min_{s' \in pred(s)}(g^*(s') + c(s', s)) & \text{otherwise.} \end{cases}$$

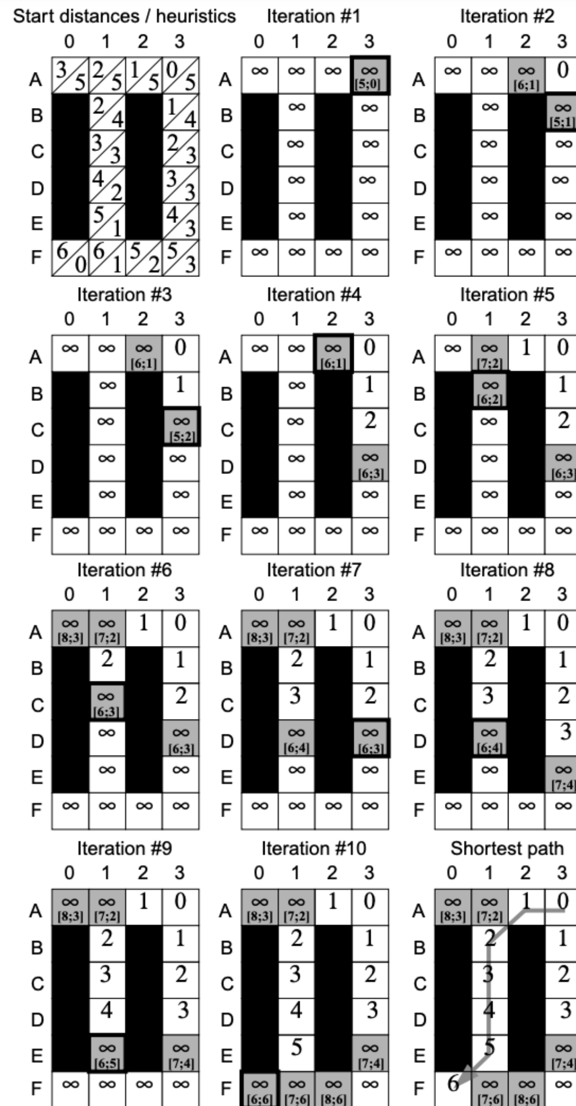
Corollary: To compute g* can do search or can ensure g(s) == rhs(s) everywhere

Idea 2: Only put locally inconsistent (g(s) == rhs(s)) cells in the queue, not all cells

Idea 3:

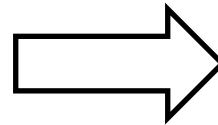
1. If g(s) > rhs(s) [Locally overconsistent] → bring g(s) down to rhs(s), update neighbors
2. If g(s) < rhs(s) [Locally underconsistent] → send g(s) to infinity, update neighbors

Example for LPA*



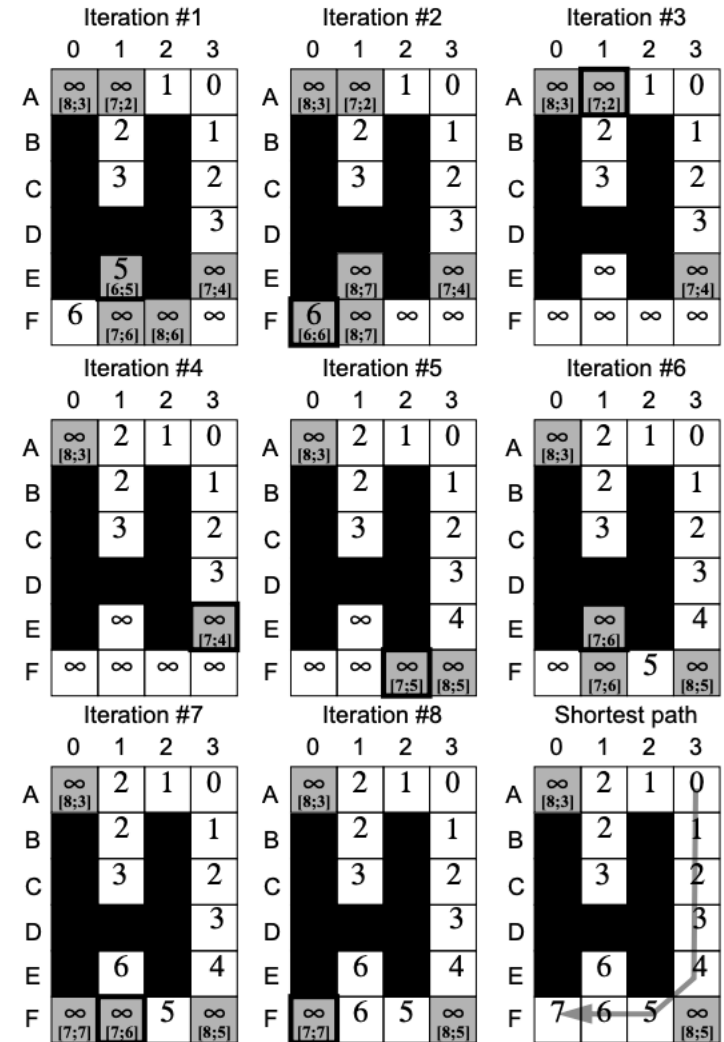
First search

Edge cost changed



Node being expanded next

Nodes in the queue



Second search

Broad Pseudocode of LPA*

Do A*, keep executing until you observe an error, then you need to replan

1. Carry over g
2. Update the edge costs, and see where LHS and RHS disagree, put those on the open queue.
3. if overconsistent, g is brought down to RHS → a shorter path exists
4. if underconsistent, g to infinity → no path can be trusted, set to infinity
   1. If locally consistent – removed
   2. If overconsistent then key is updated.
5. Idea 5: End when goal is consistent or priority queue has key greater than goal.
6. Idea 6: Finally get path greedily

Pseudocode of LPA*

procedure Initialize()
{02} $U = \emptyset$;
{03} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{04} $rhs(s_{start}) = 0$;
{05} U.Insert(s_{start}, $[h(s_{start}); 0]$);

procedure CalculateKey(s)
{01} return $[\min(g(s), rhs(s)) + h(s); \min(g(s), rhs(s))]$;

procedure UpdateVertex(u)
{06} if $(u \neq s_{start})$ $rhs(u) = \min_{s' \in pred(u)}(g(s') + c(s', u))$;
{07} if $(u \in U)$ U.Remove(u);
{08} if $(g(u) \neq rhs(u))$ U.Insert(u, CalculateKey(u));

procedure Main()
{17} Initialize();
{18} forever
{19} ComputeShortestPath();
{20} Wait for changes in edge costs;
{21} for all directed edges (u, v) with changed edge costs
{22} Update the edge cost $c(u, v)$;
{23} UpdateVertex(v);

procedure ComputeShortestPath()
{09} while (U.TopKey() $\dot{<}$ CalculateKey(s_{goal}) OR $rhs(s_{goal}) \neq g(s_{goal})$)
{10} $u =$ U.Pop();
{11} if $(g(u) > rhs(u))$
{12} $g(u) = rhs(u)$;
{13} for all $s \in succ(u)$ UpdateVertex(s);
{14} else
{15} $g(u) = \infty$;
{16} for all $s \in succ(u) \cup \{u\}$ UpdateVertex(s);

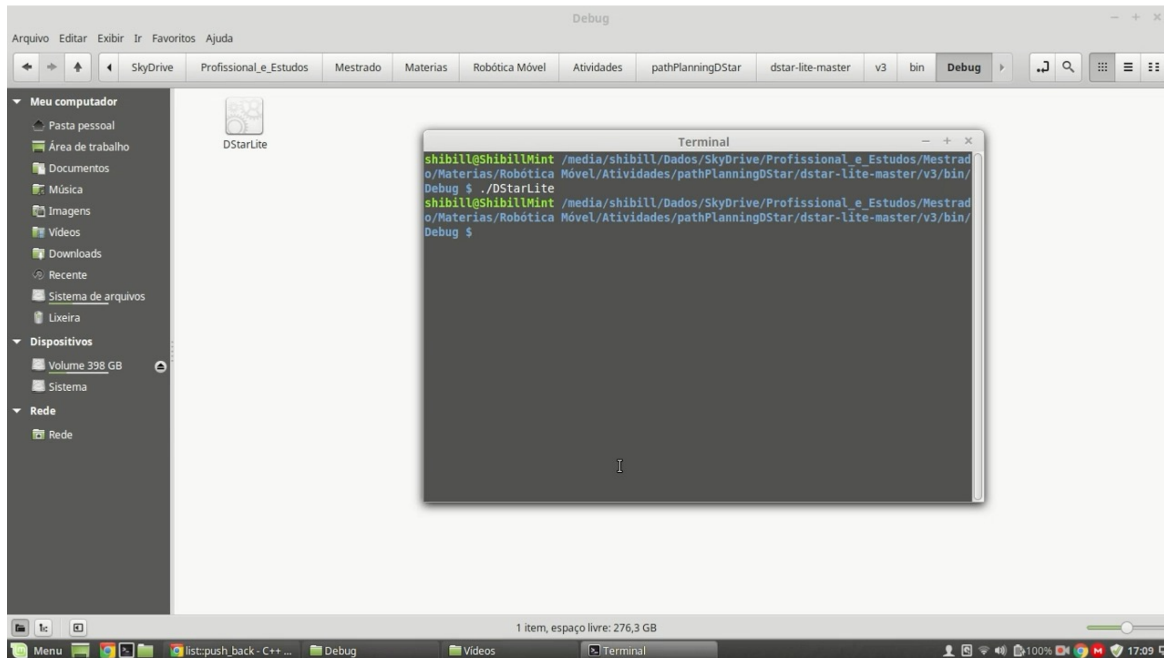
Going from LPA* to D* lite

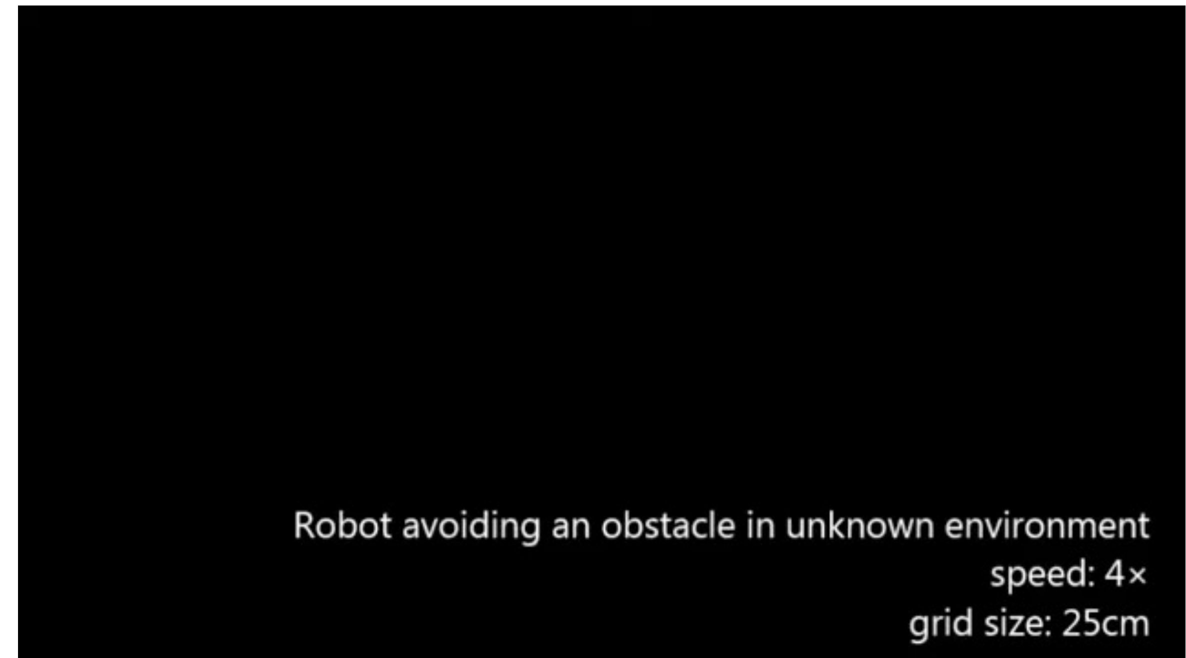
LPA* always computes paths from same start to goal, but robots often encounter change to the map/connectivity during execution → LPA* no longer works, g values are totally different

D* lite has 2 key ideas:

1. Perform search backwards → flip all the edges and compute distance from goal to every state (allows for changing start)
2. Account for changing goal using some extra bookkeeping in the heuristic (since the “goal” [current robot state] changes)

Incremental Search in Practice: D* lite





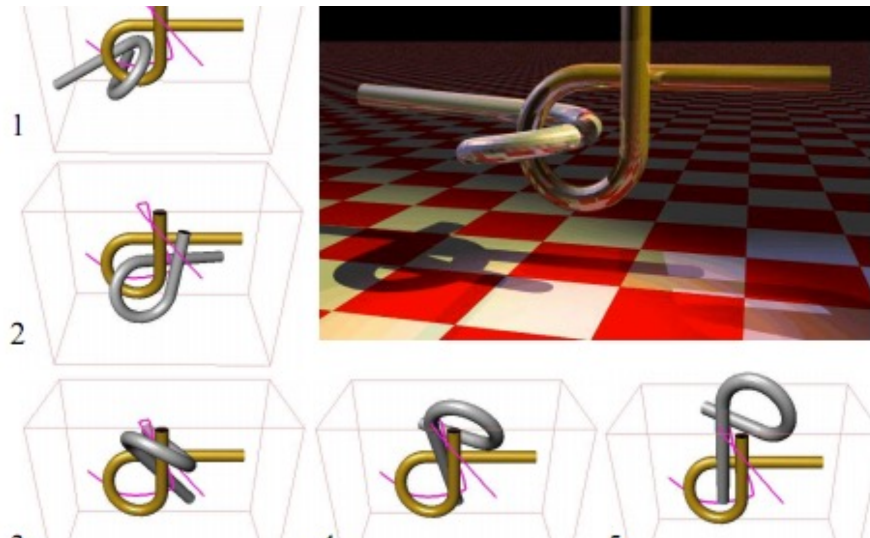
Lecture Outline

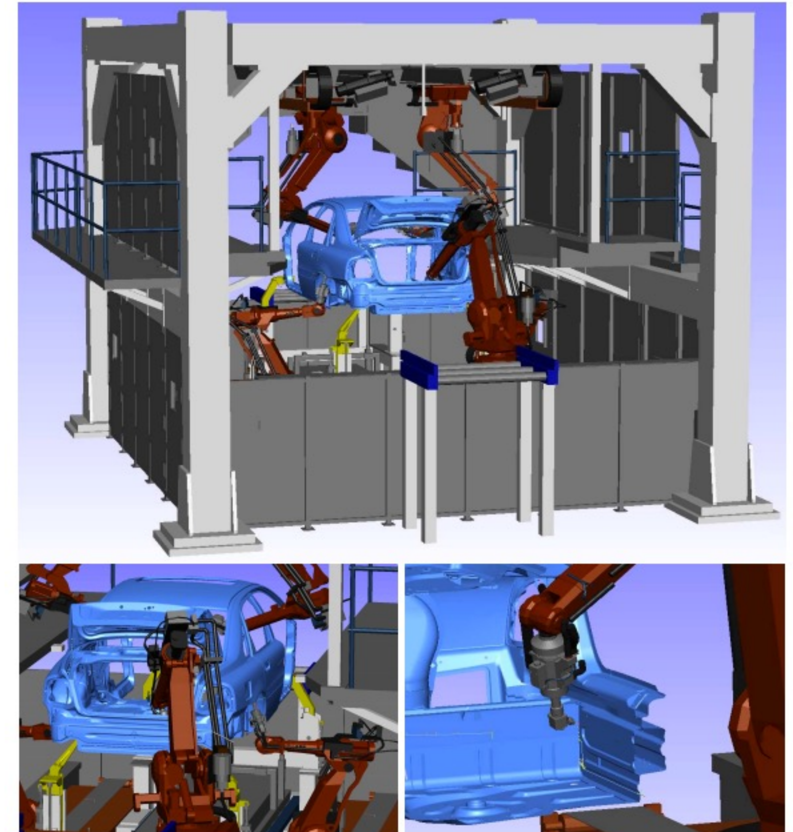
Incremental Search – LPA*

Sampling Based Motion Planning - PRMs

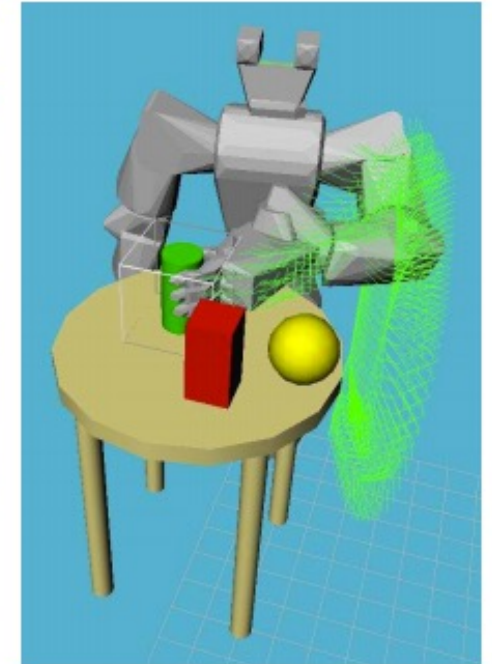
RRT and RRT*

Motion Planning in Practice



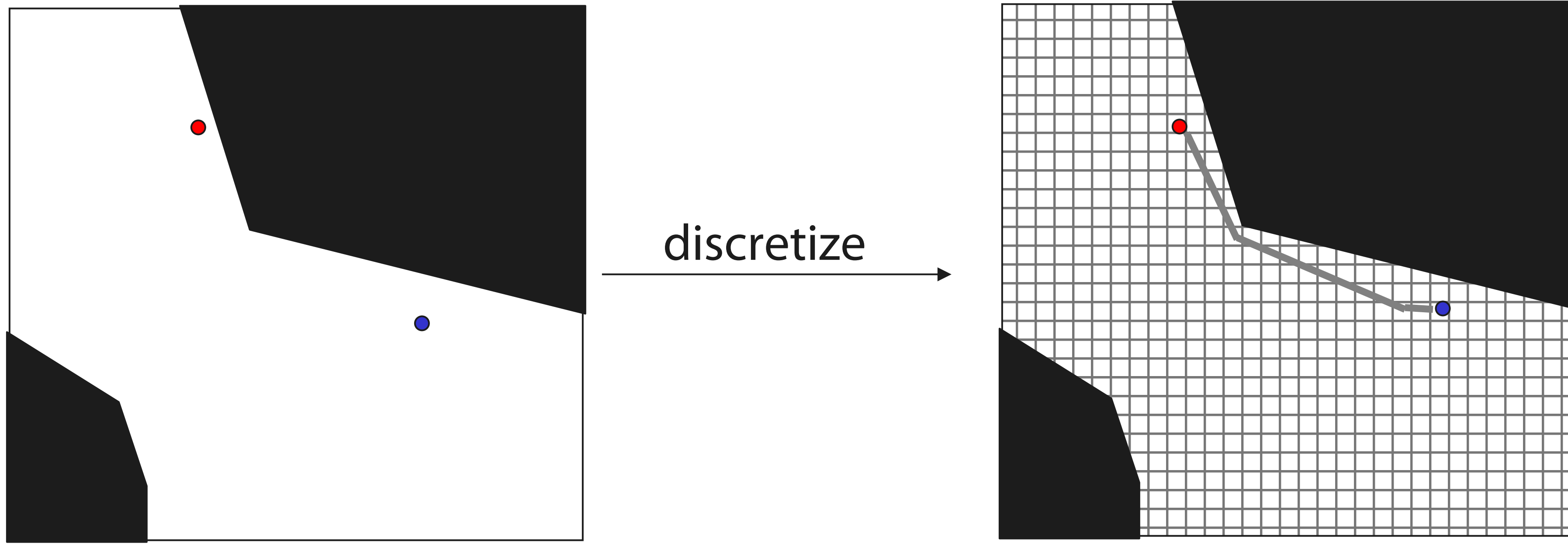


Motion Planning in Practice



Motion Planning in Practice

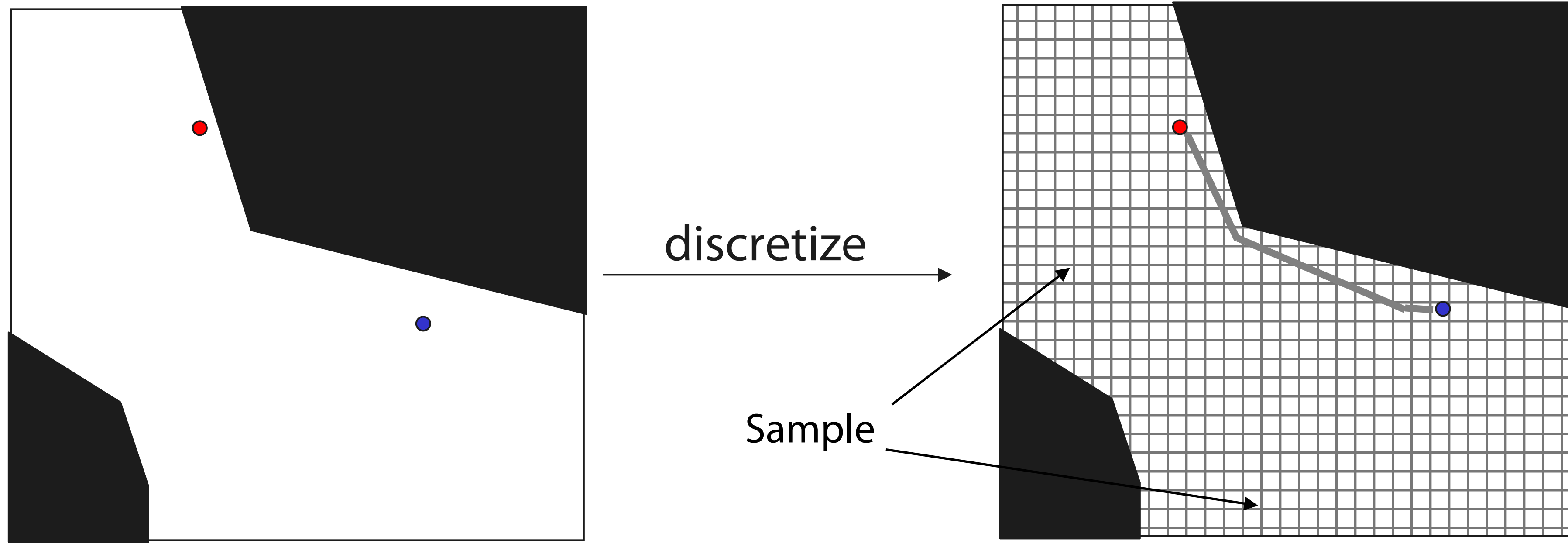
Why is A* not enough?



Scales poorly both computationally and memory wise for high dimensional, non-convex systems

Can be hard to know how to properly discretize the space, especially as dimensions of configuration space are very different

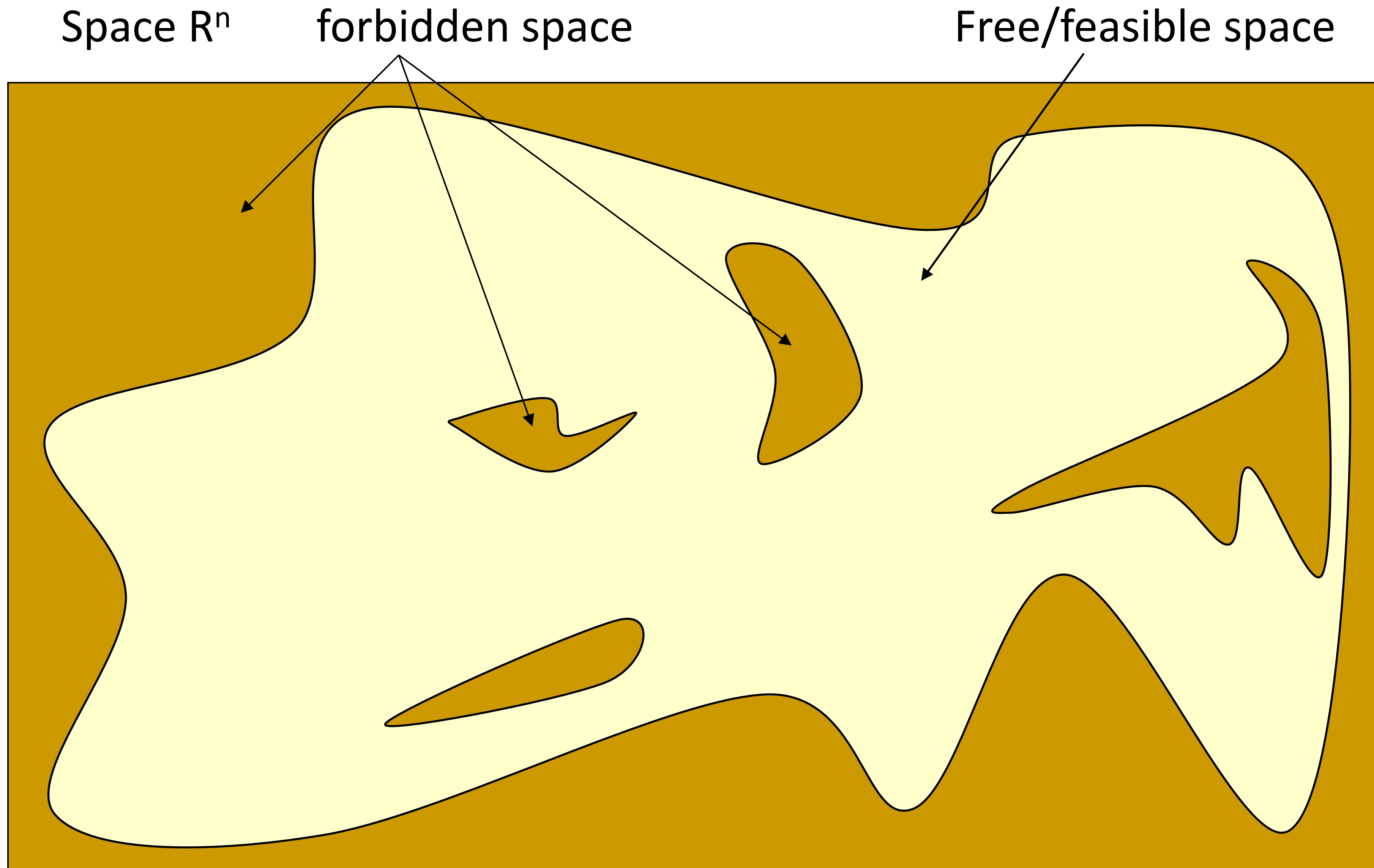
How can we do better?



What if we just "sampled" the discretization rather than did this deterministically?
→ Note this is still deterministic planning on the graph, just sample the graph

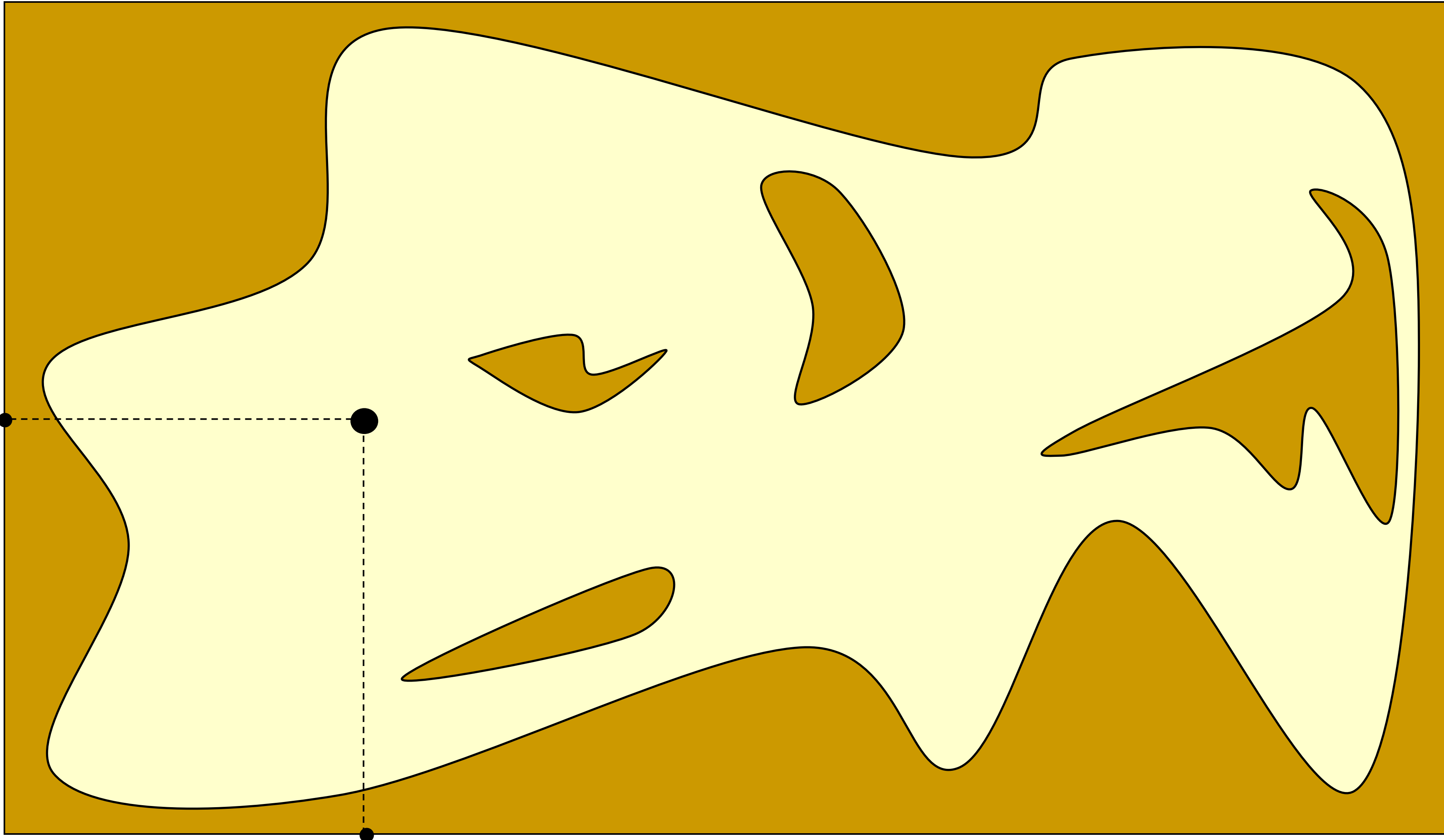
Not optimal, but maybe this is ok!

Probabilistic Roadmap (PRM)



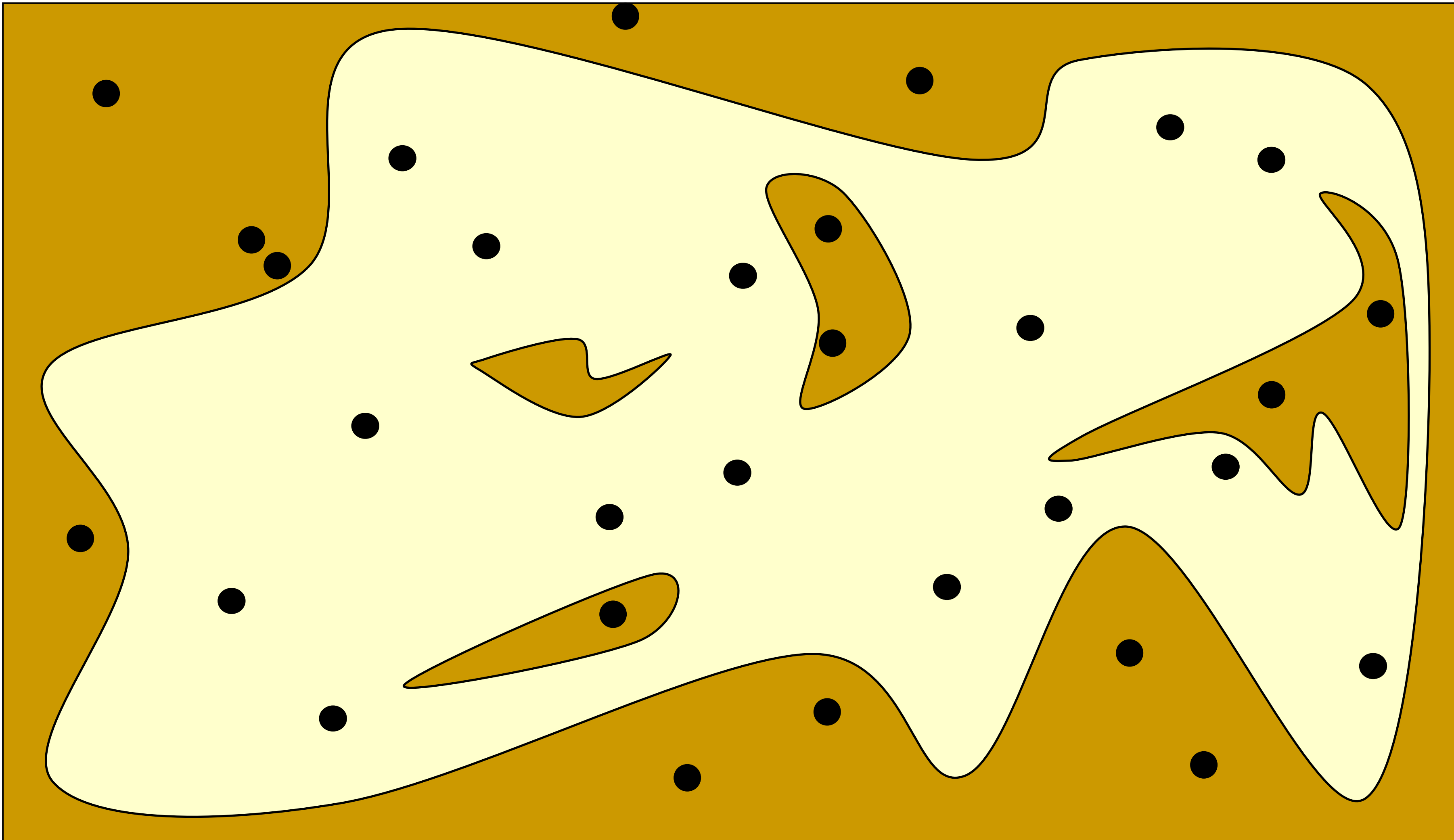
Probabilistic Roadmap (PRM)

Configurations are sampled by picking coordinates at random



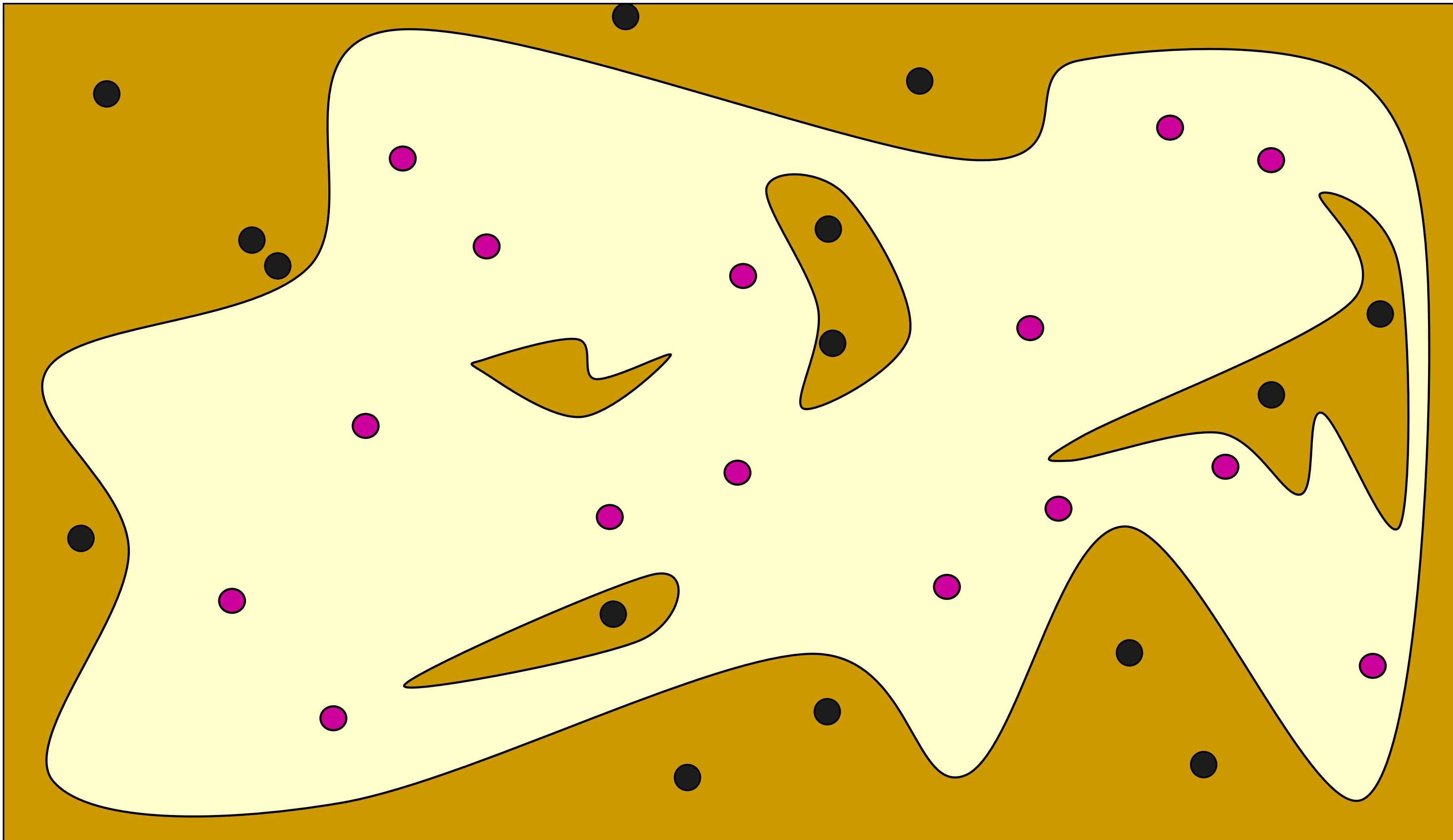
Probabilistic Roadmap (PRM)

Configurations are sampled by picking coordinates at random



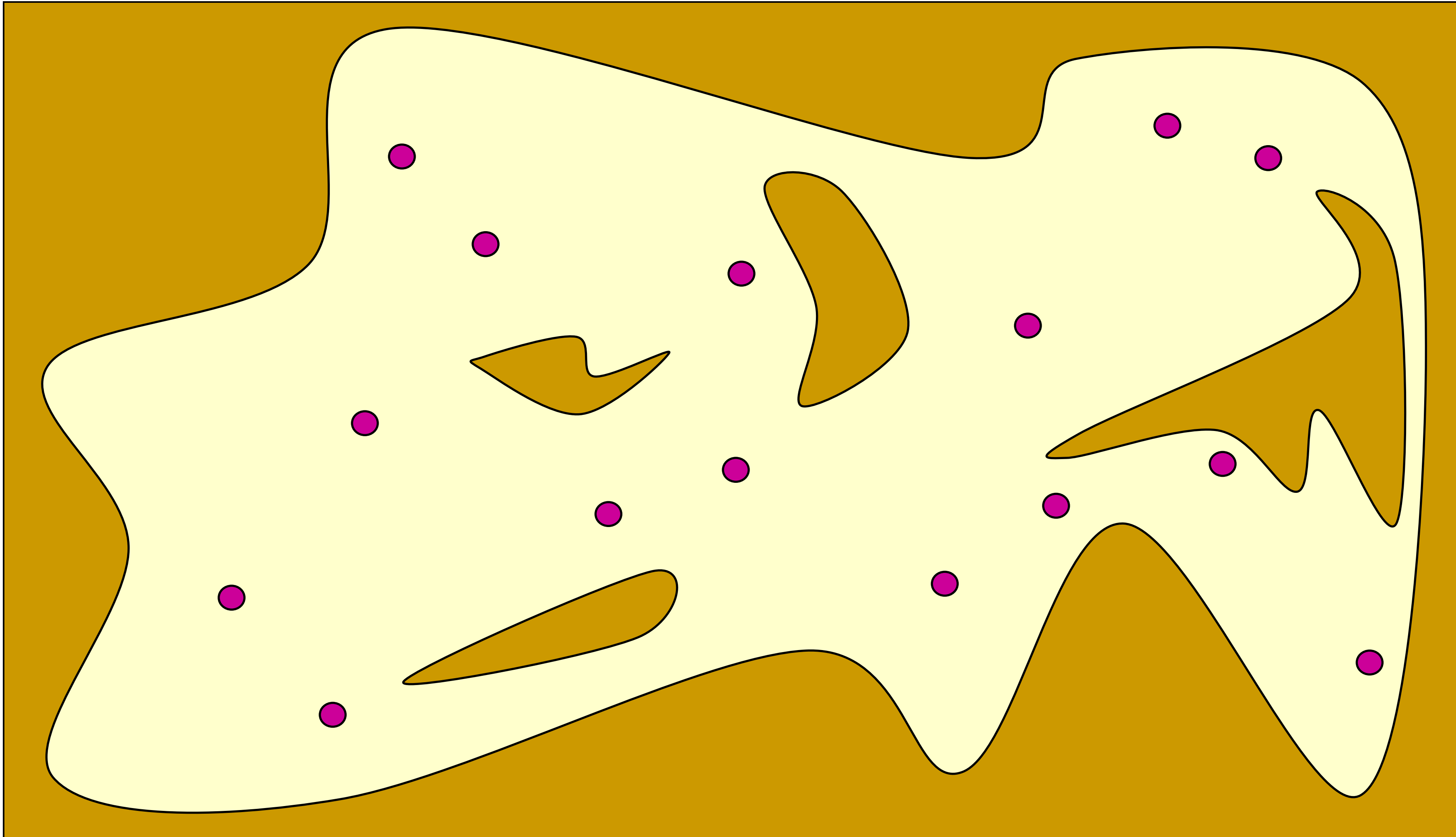
Probabilistic Roadmap (PRM)

Sampled configurations are tested for collision



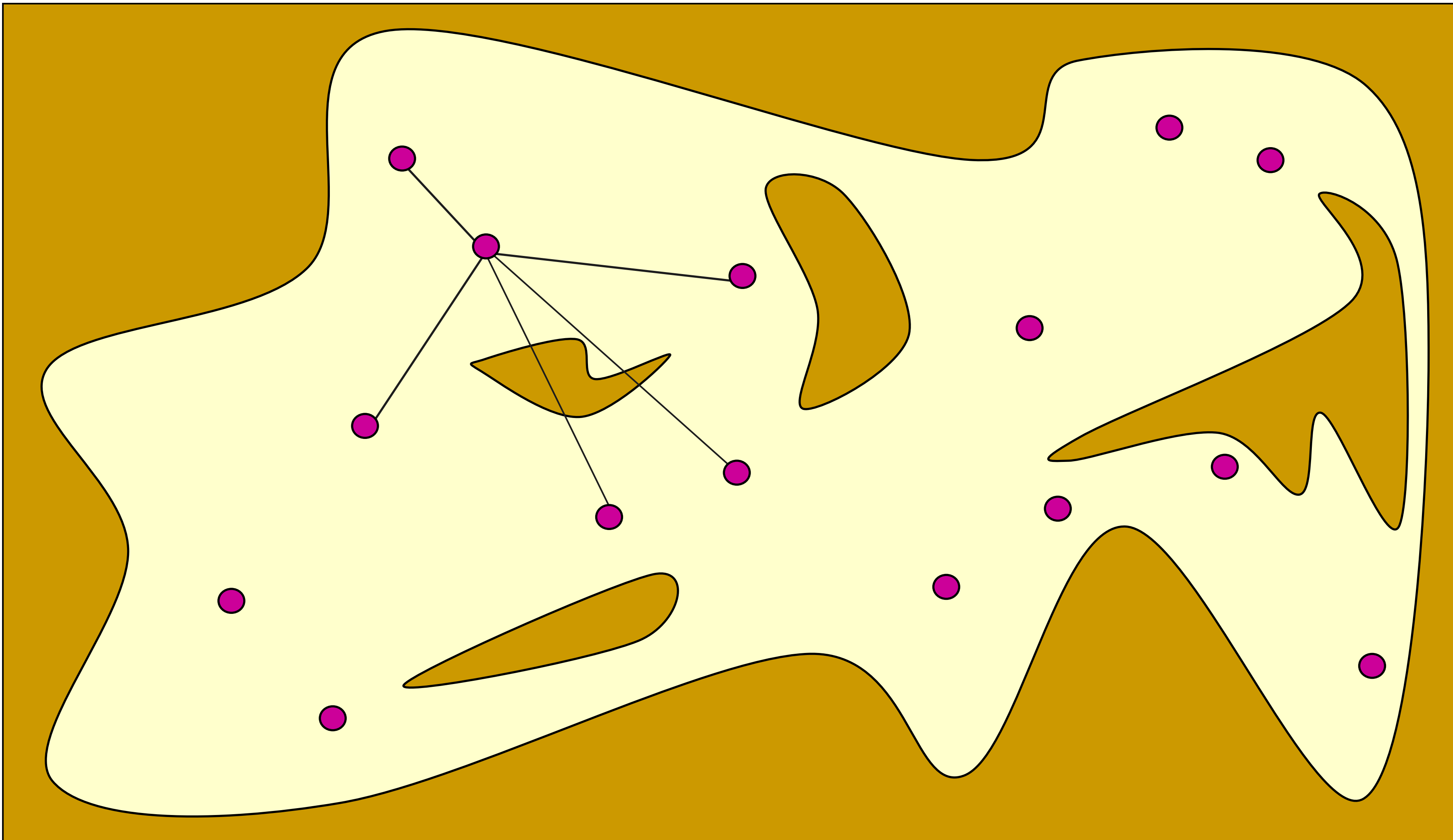
Probabilistic Roadmap (PRM)

The collision-free configurations are retained as milestones



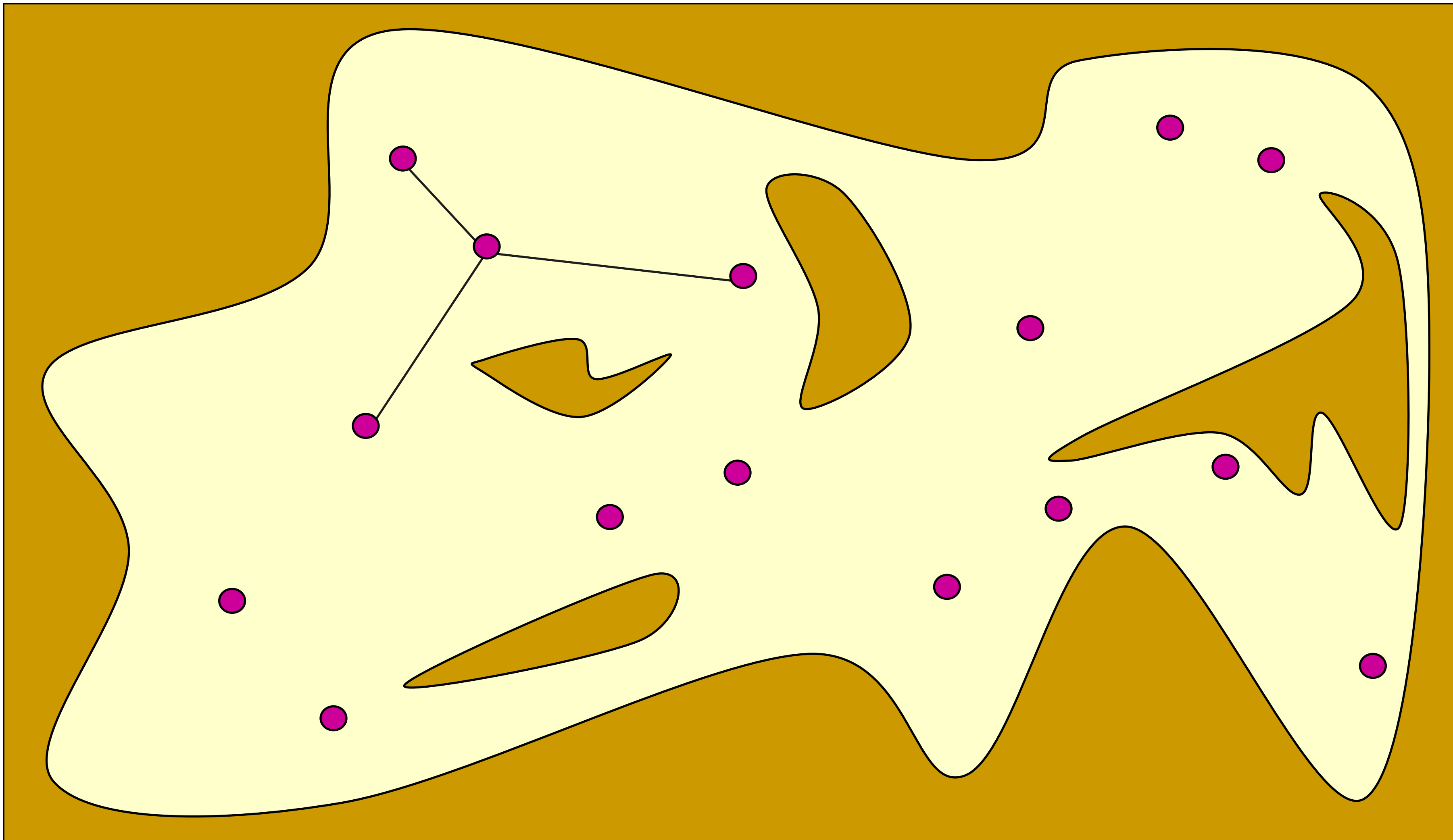
Probabilistic Roadmap (PRM)

Each milestone is linked by straight paths to its nearest neighbors



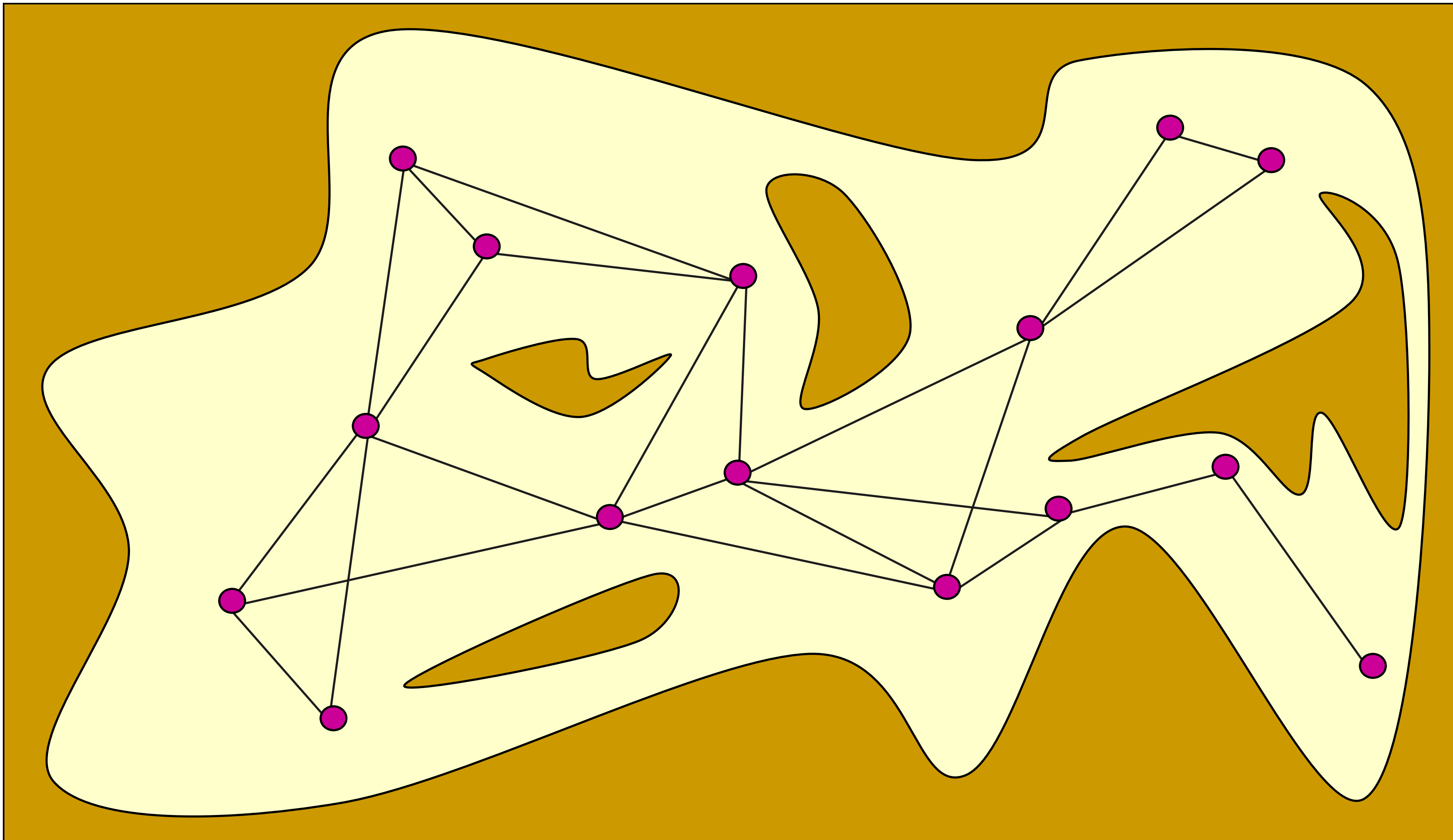
Probabilistic Roadmap (PRM)

Each milestone is linked by straight paths to its nearest neighbors



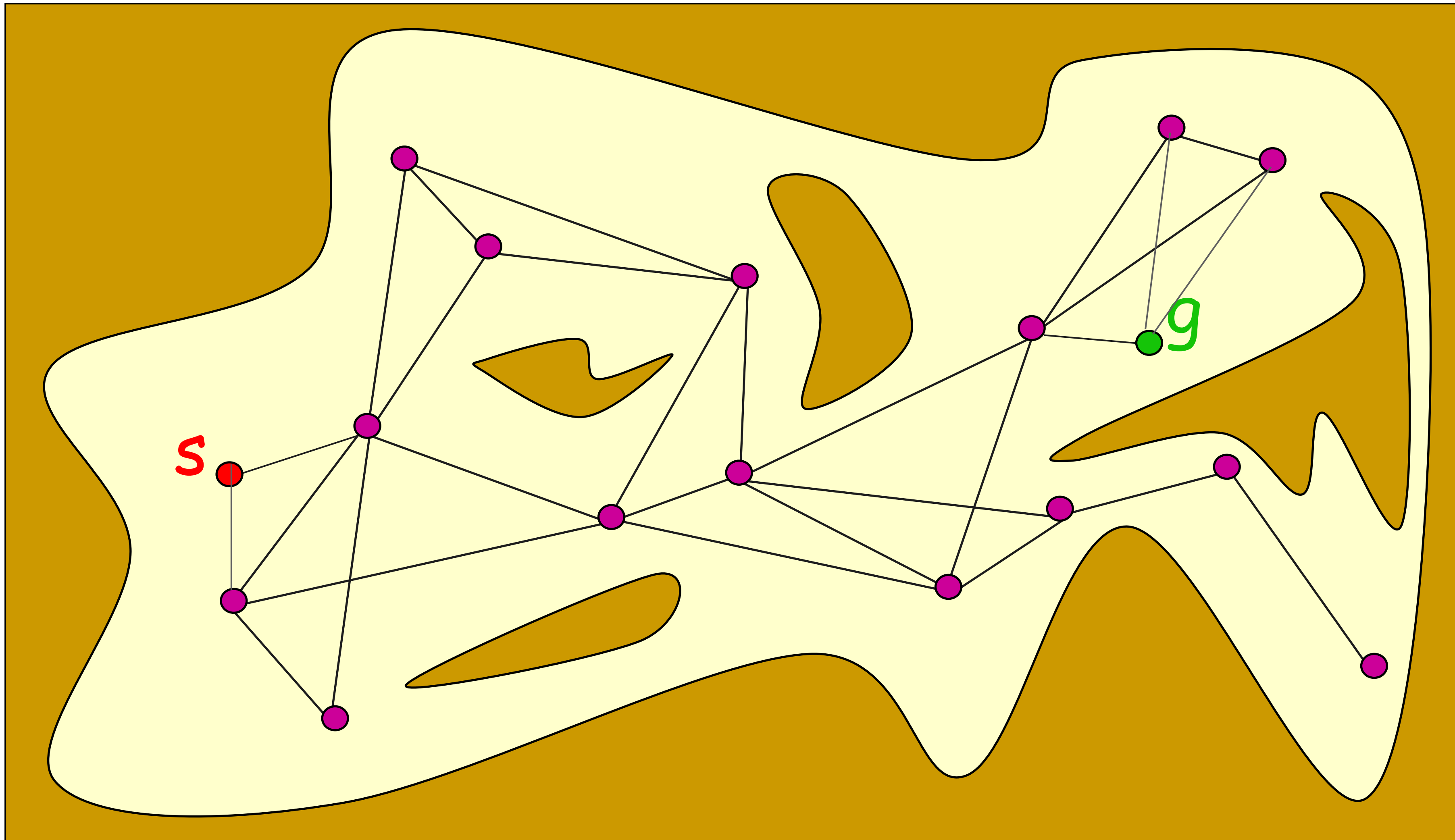
Probabilistic Roadmap (PRM)

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



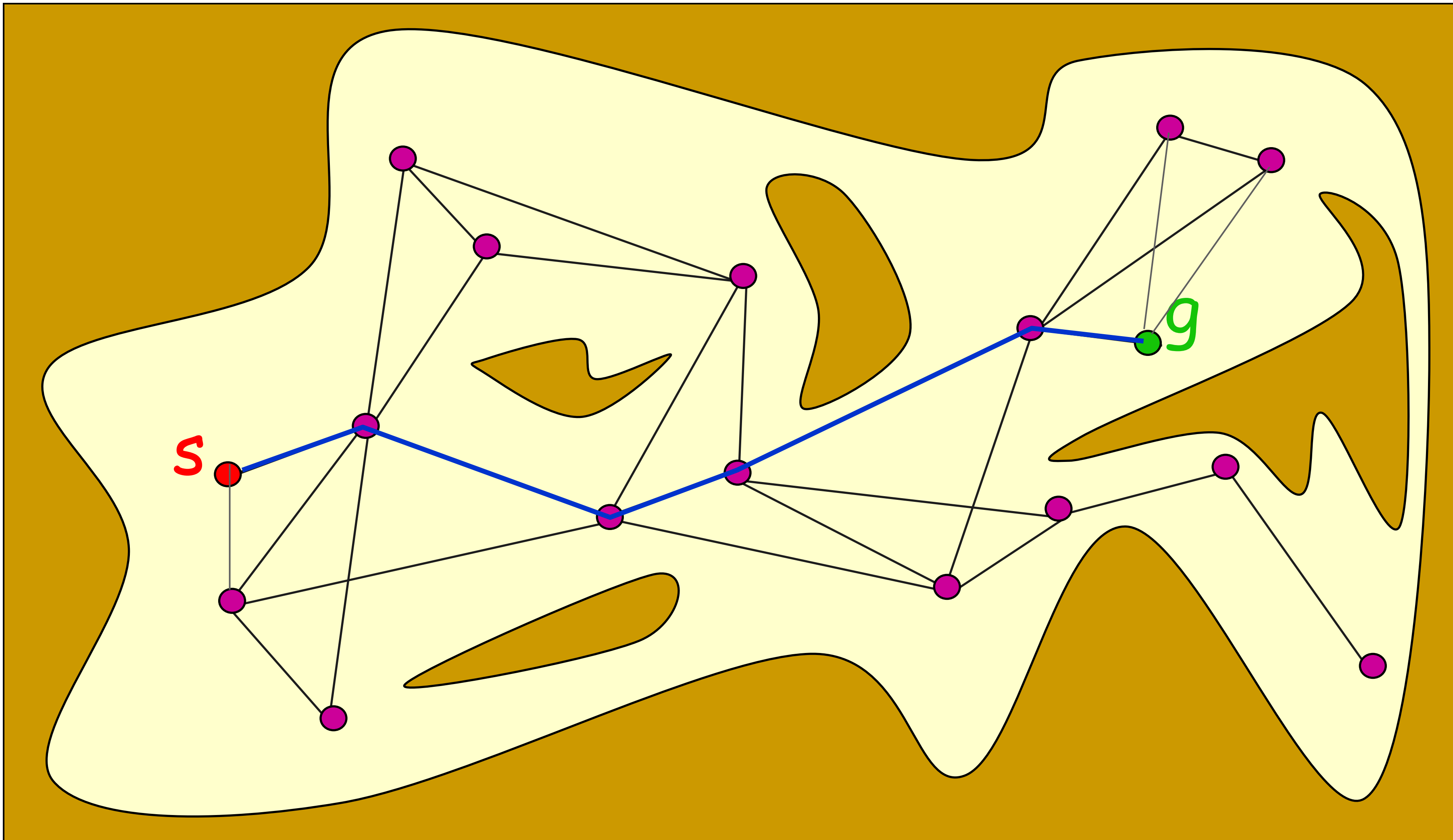
Probabilistic Roadmap (PRM)

The start and goal configurations are included as milestones



Probabilistic Roadmap (PRM)

The PRM is searched for a path from s to g



Probabilistic Roadmap

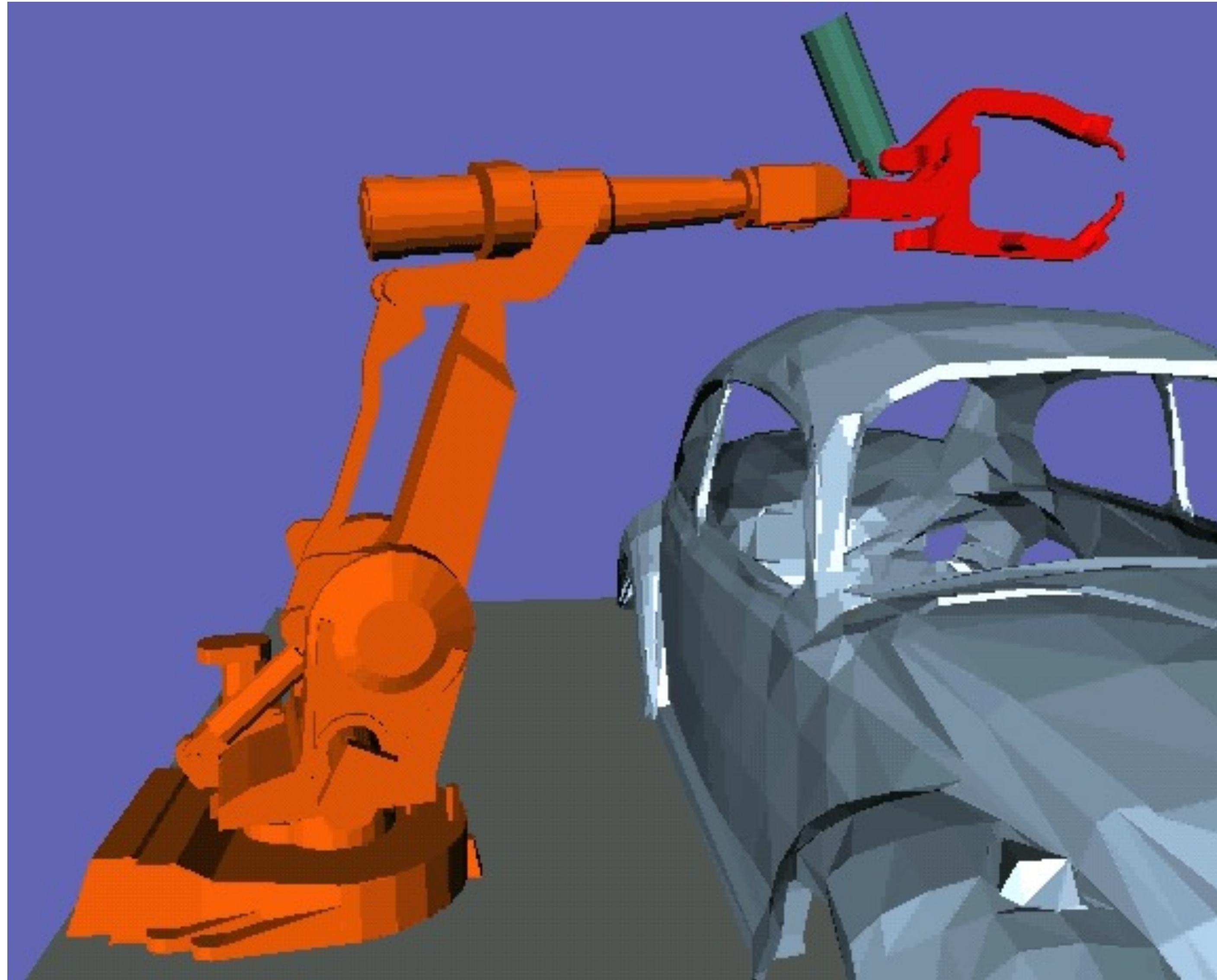
- Initialize set of points with x_S and x_G
- Randomly sample points in configuration space
- Connect nearby points if they can be reached from each other
- Find path from x_S to x_G in the graph

  - Alternatively: keep track of connected components incrementally, and declare success when x_S and x_G are in same connected component

Why are PRMs better than A*?

- A* is resolution complete, but wastes significant space and is hard to know how to discretize
- PRMs tradeoff optimality for speed, typically much faster due to sparse exploration of a space rather than complete discretization
- Trades off resolution completeness for probabilistic completeness
- Can control memory and computational usage

PRM Example 1

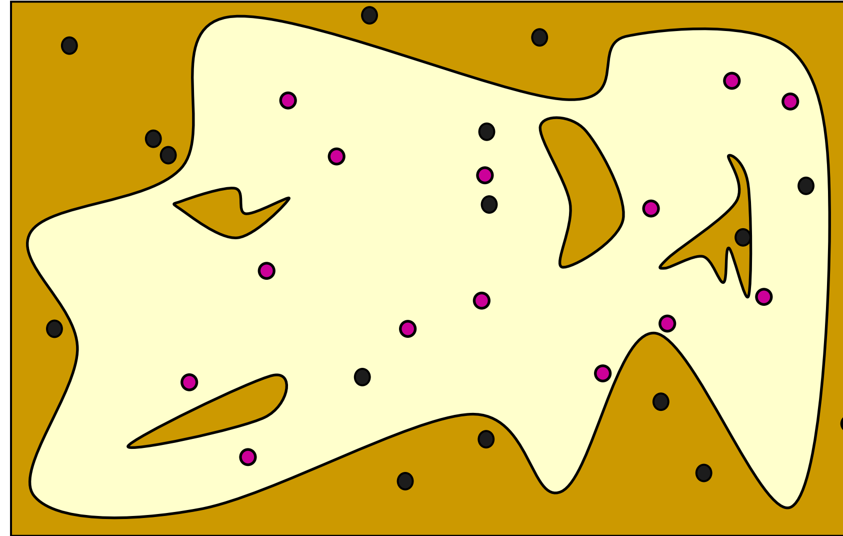


PRM Example 2

PRM's Pros and Cons

- Pro:
  - Probabilistically complete: i.e., with probability one, if run for long enough the graph will contain a solution path if one exists.
- Cons:
  - Required to solve 2-point boundary value problem
  - Build graph over state space but no focus on generating a path

Why are PRMs not enough?



- Sampling indiscriminately wastes a significant number of points in parts of the space not needed.
- Requires accurately solving the 2 point BVP for non-holonomic systems
- We don’t care about going from anywhere to anywhere, just start to goal
  - Q: can we build up the graph/tree incrementally with only a little more work and no 2-point BVP?

Lecture Outline

Incremental Search – LPA*

Sampling Based Motion Planning - PRMs

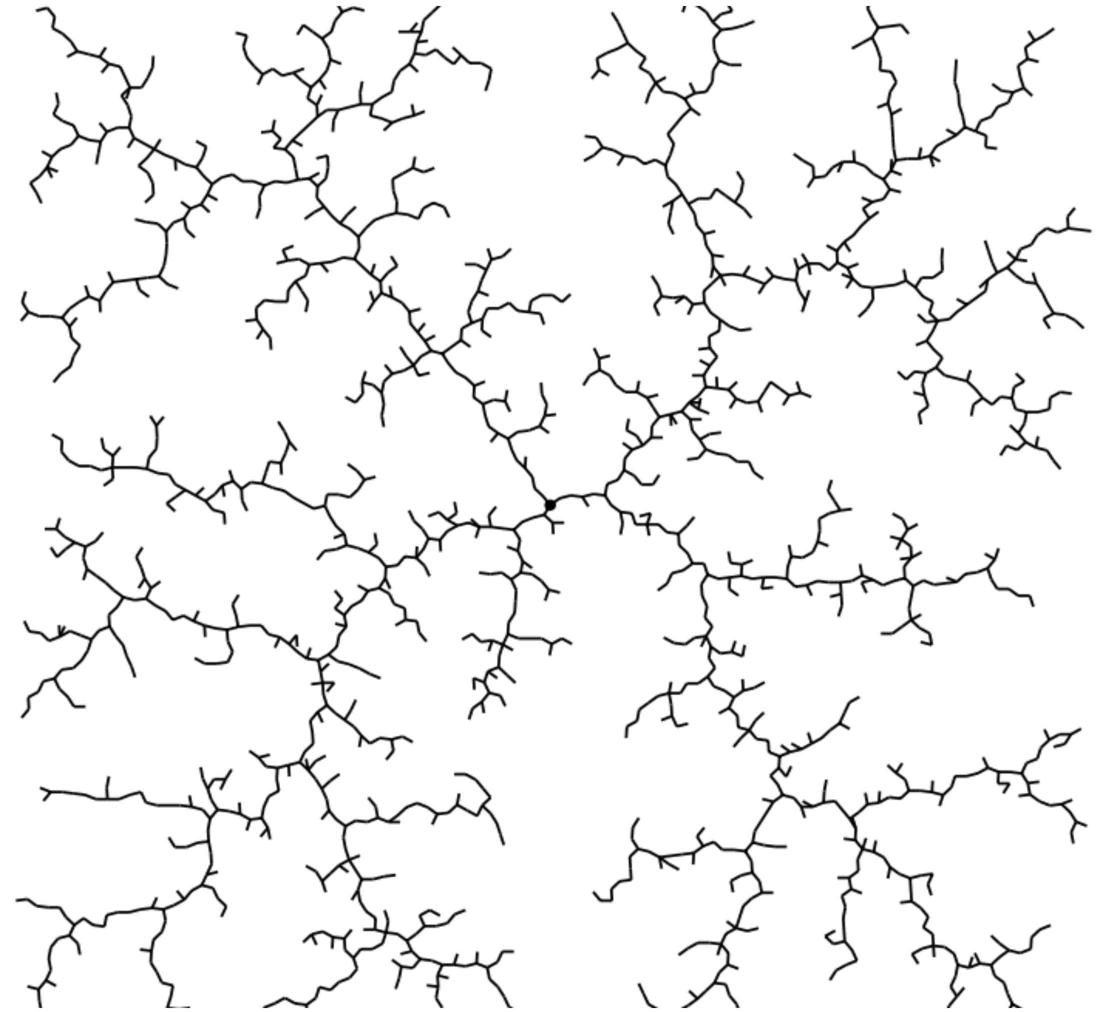
RRT and RRT*

Rapidly exploring Random Tree (RRT)

Steve LaValle (98)

- Basic idea:
  - Build up a tree through generating “next states” in the tree by executing random controls → no need to solve 2 point BVP exactly
  - Execute tree search in the RRT, same as before
  - Caveat: not exactly above to ensure good coverage

How to Sample

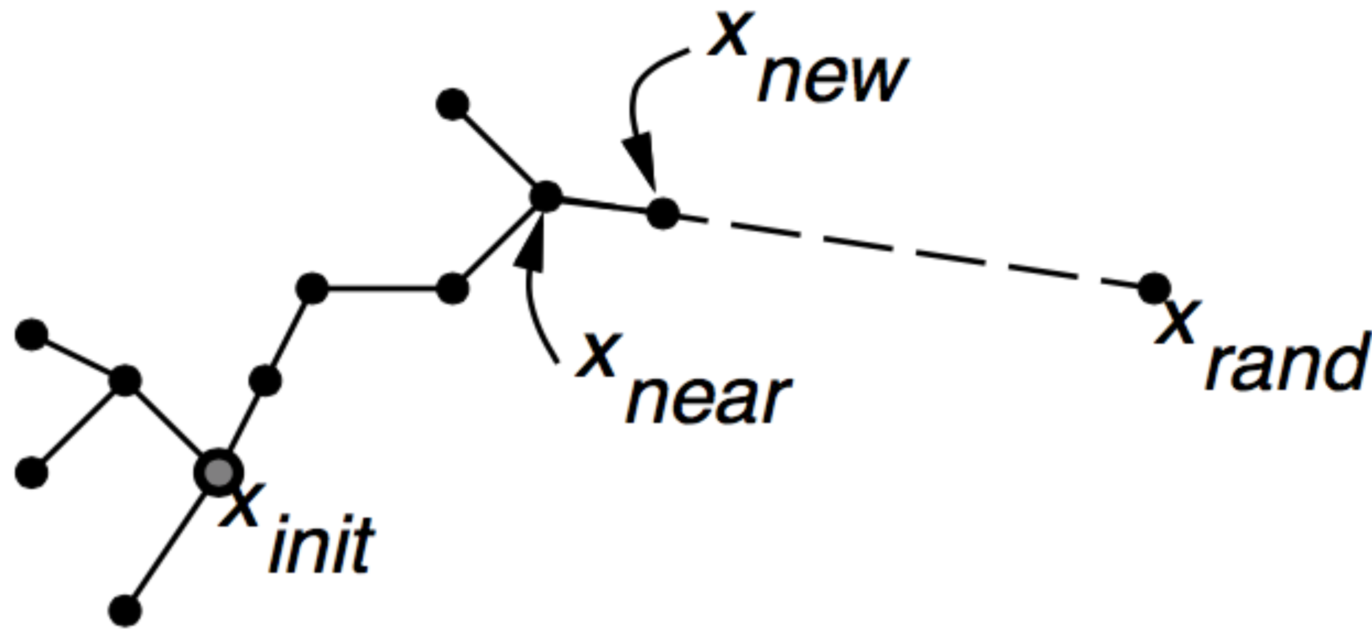


Rapidly exploring Random Tree (RRT)

- Select random point, and expand nearest vertex towards it
  - Biases samples towards largest Voronoi region

Rapidly exploring Random Tree (RRT)

- Select random point, and expand nearest vertex towards it
  - Biases samples towards largest Voronoi region

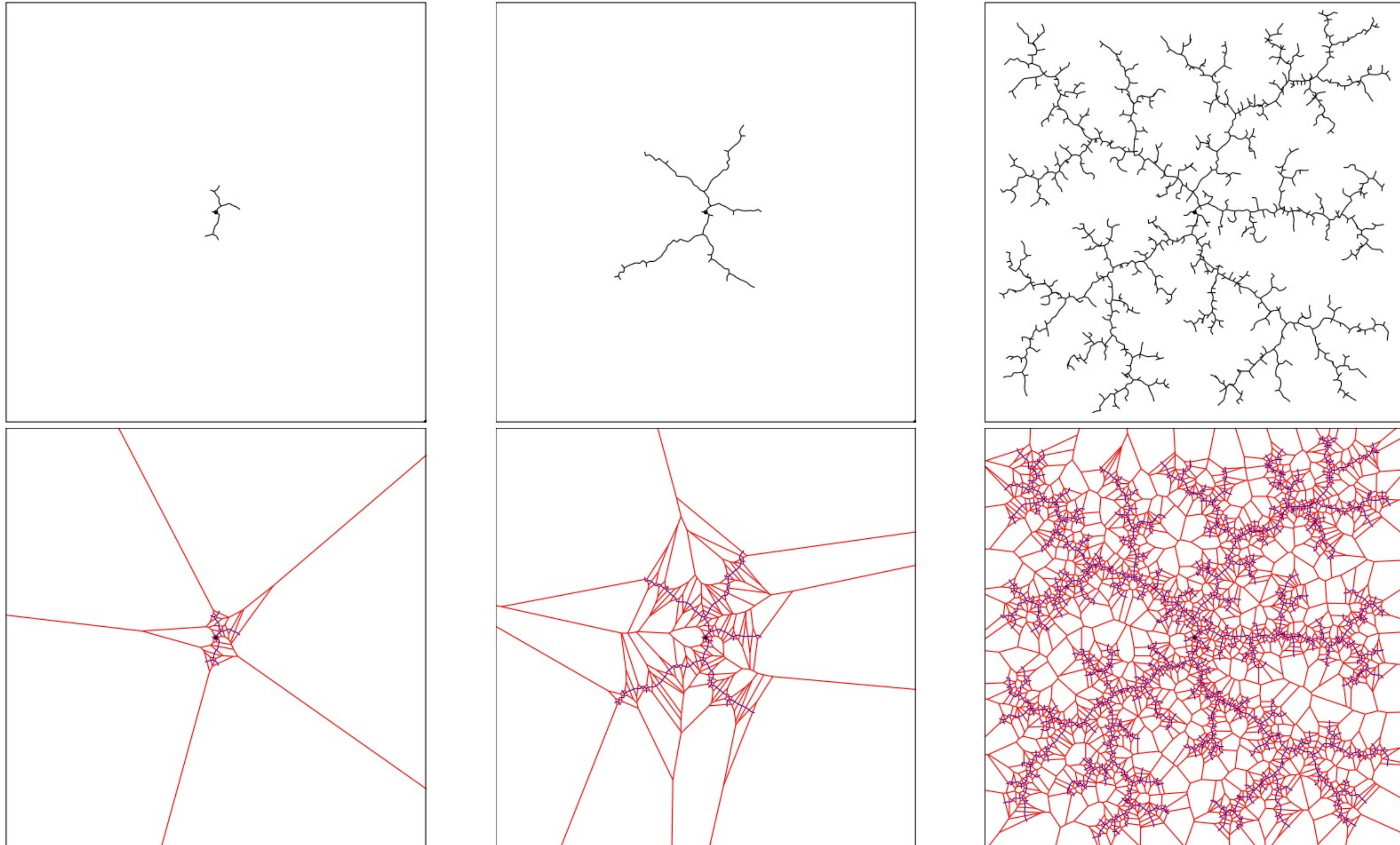


Rapidly exploring Random Tree (RRT)

GENERATE_RRT($x_{init}, K, \Delta t$)
1 $\mathcal{T}$.init(x_{init});
2 **for** $k = 1$ **to** K **do**
3 $\quad x_{rand} \leftarrow$ RANDOM_STATE();
4 $\quad x_{near} \leftarrow$ NEAREST_NEIGHBOR($x_{rand}, \mathcal{T}$);
5 $\quad u \leftarrow$ SELECT_INPUT(x_{rand}, x_{near});
6 $\quad x_{new} \leftarrow$ NEW_STATE($x_{near}, u, \Delta t$);
7 $\quad \mathcal{T}$.add_vertex(x_{new});
8 $\quad \mathcal{T}$.add_edge(x_{near}, x_{new}, u);
9 Return $\mathcal{T}$

RANDOM_STATE(): often uniformly at random over space with probability 99%, and the goal state with probability 1%, this ensures it attempts to connect to goal semi-regularly

Rapidly exploring Random Tree (RRT)



Source: LaValle and Kuffner 01

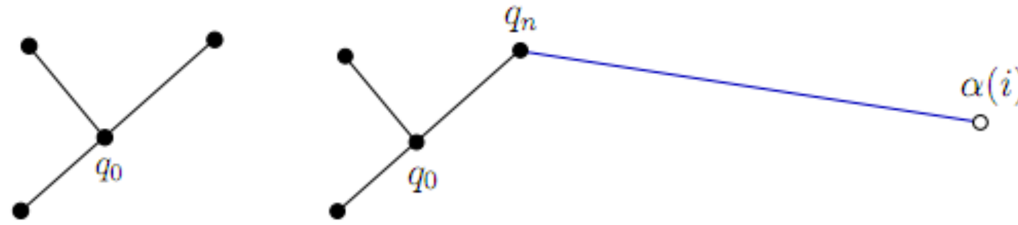
RRT Practicalities

- NEAREST_NEIGHBOR(x_{rand}, T): need to find (approximate) nearest neighbor efficiently
  - KD Trees data structure (upto 20-D) [e.g., FLANN]
  - Locality Sensitive Hashing

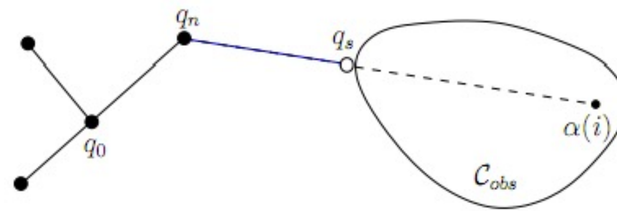
- SELECT_INPUT(x_{rand}, x_{near})
  - Two point boundary value problem
    - If too hard to solve, often just select best out of a set of control sequences. This set could be random, or some well chosen set of primitives.

RRT Extension

- No obstacles, holonomic:

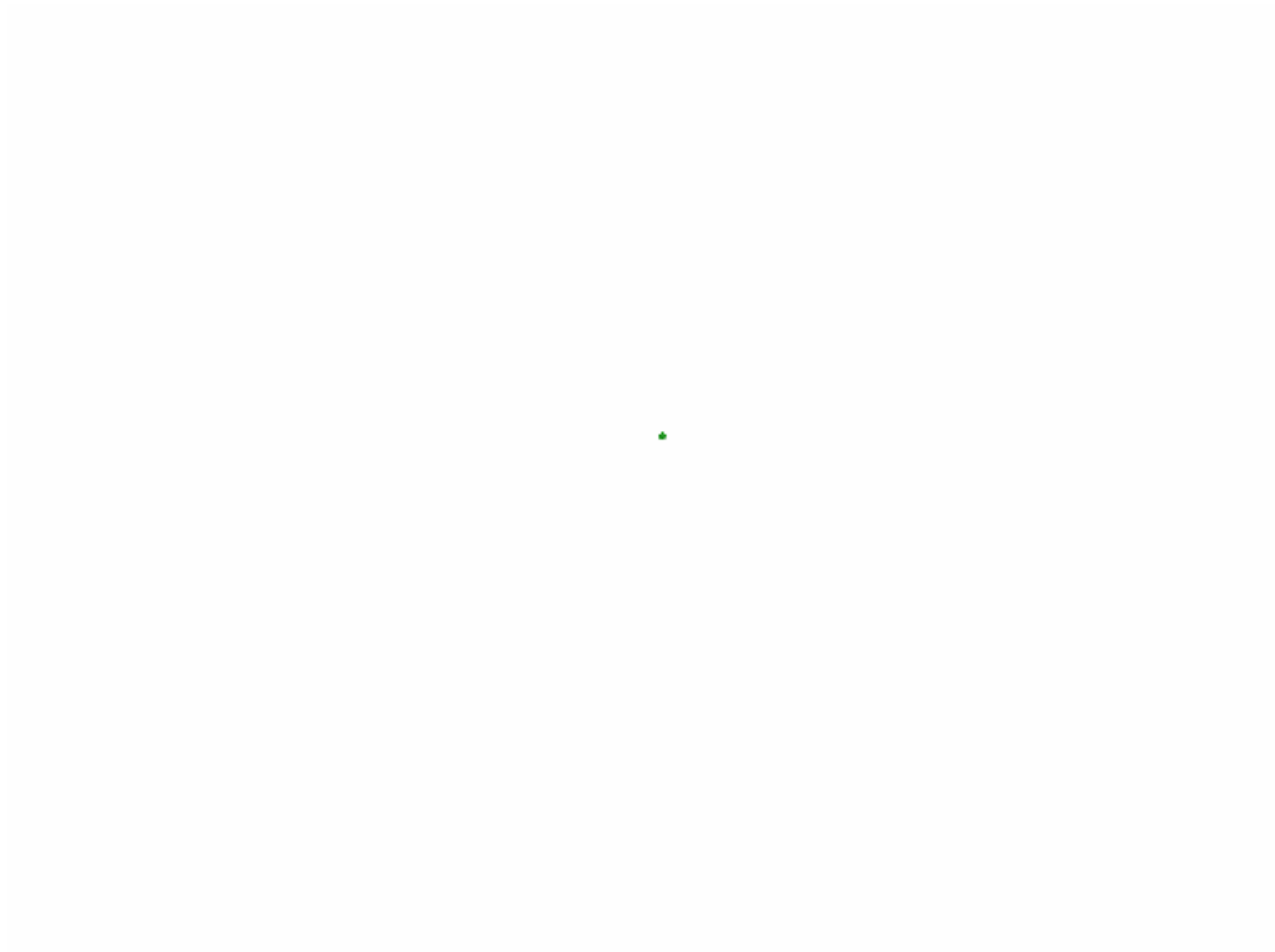


- With obstacles, holonomic:



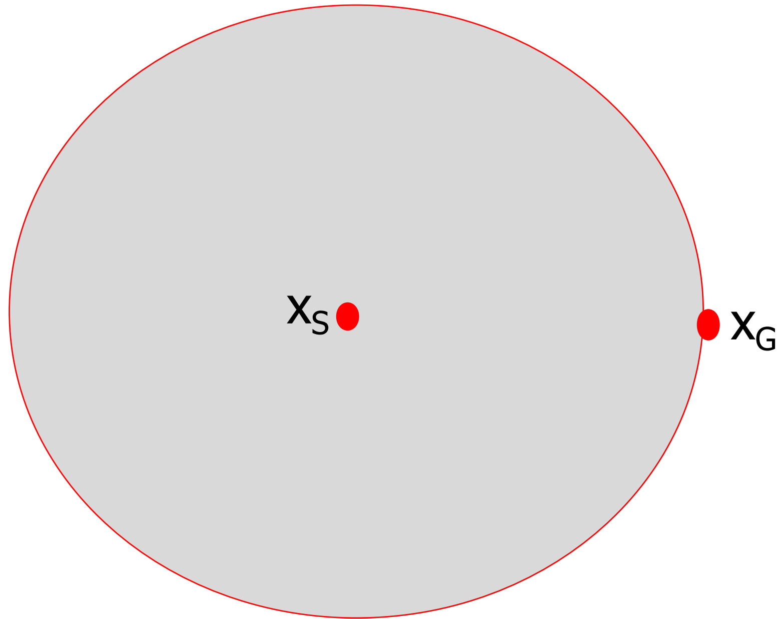
- Non-holonomic: approximately (sometimes as approximate as picking best of a few random control sequences) solve two-point boundary value problem

Growing RRT

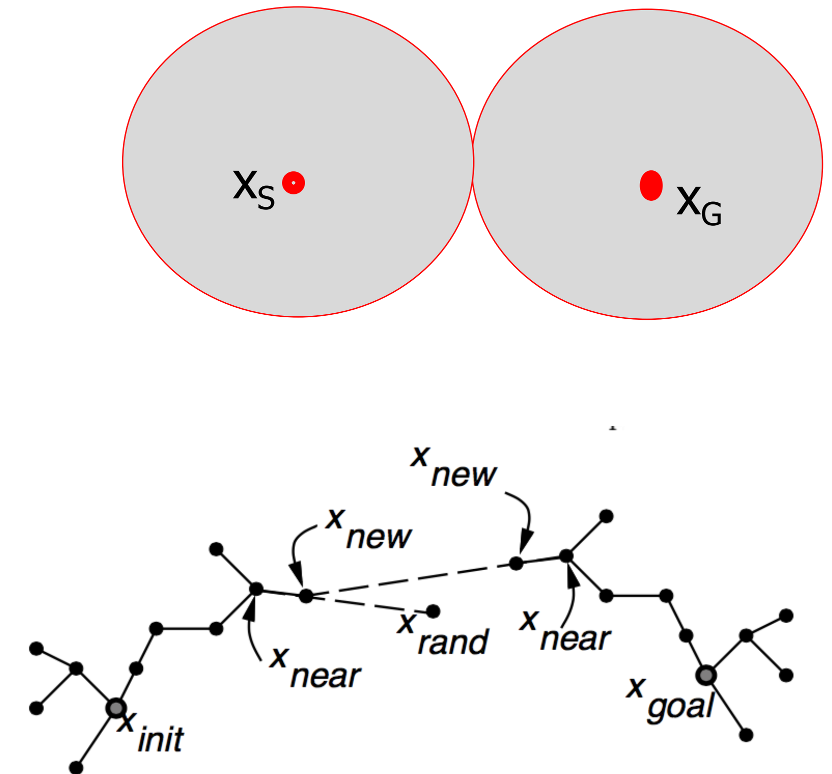


Bi-directional RRT

- Volume swept out by unidirectional RRT:



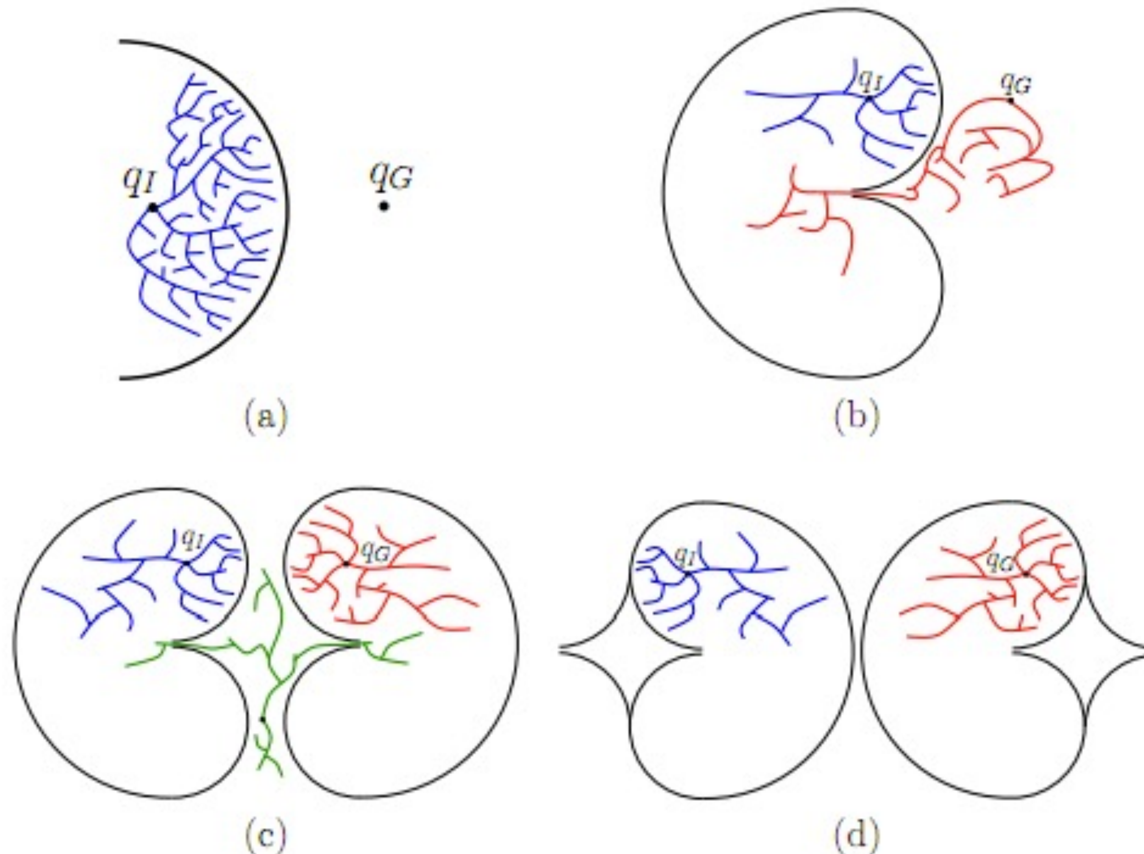
- Volume swept out by bi-directional RRT:



- Difference more and more pronounced as dimensionality increases

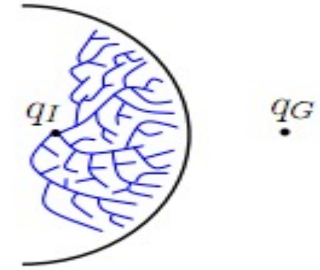
Multi-directional RRT

- Planning around obstacles or through narrow passages can often be easier in one direction than the other



Resolution-Complete RRT (RC-RRT)

- Issue: nearest points chosen for expansion are (too) often the ones stuck behind an obstacle

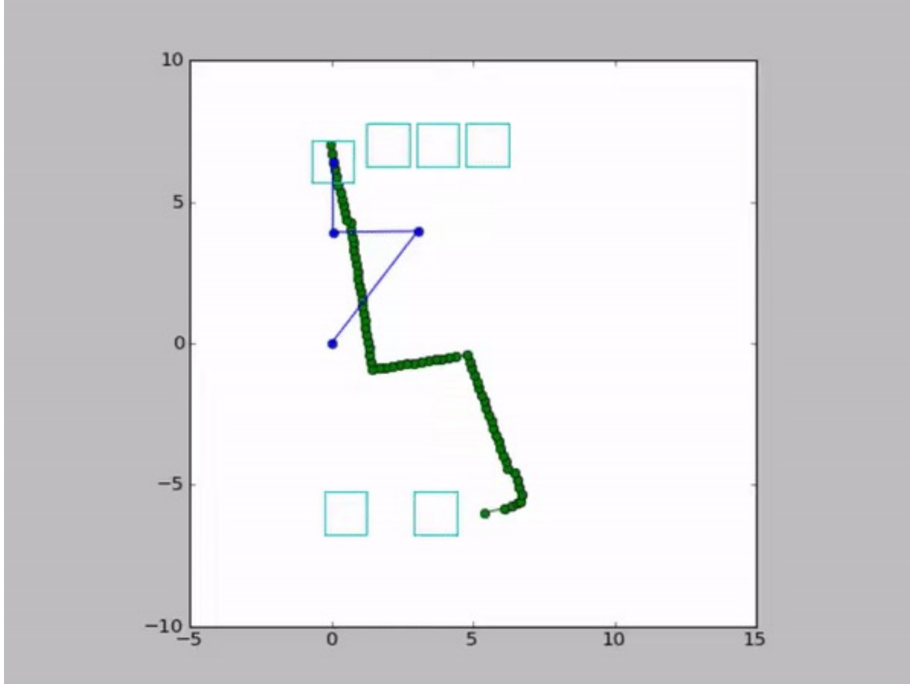


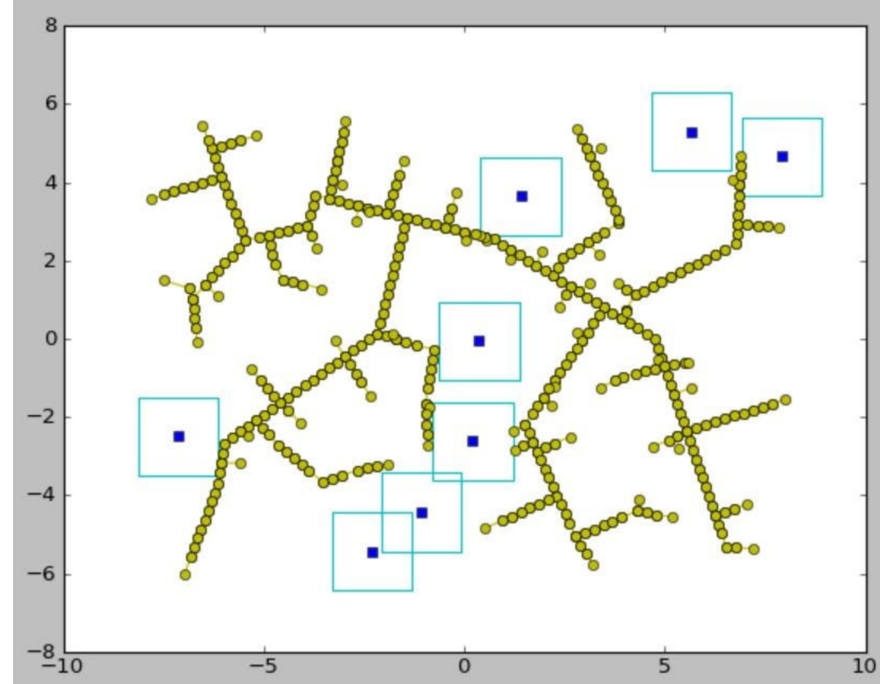
RC-RRT solution:

- Choose a maximum number of times, m, you are willing to try to expand each node
- For each node in the tree, keep track of its Constraint Violation Frequency (CVF)
- Initialize CVF to zero when node is added to tree
- Whenever an expansion from the node is unsuccessful (e.g., per hitting an obstacle):
  - Increase CVF of that node by 1
  - Increase CVF of its parent node by 1/m, its grandparent $1/m^2$, …
- When a node is selected for expansion, skip over it with probability CVF/m

Why is RRT not enough?

- RRT guarantees probabilistic completeness but not optimality (shortest path)
  - In practice leads to paths that are very roundabout and non-direct -> not shortest paths





Asymptotically optimal RRT → RRT*

- Asymptotically optimal version of RRT*
- Main idea:
  - Swap new point in as parent for nearby vertices who can be reached along shorter path through new point than through their original (current) parent
  - Consider path lengths and not just connectivity

RRT*

Algorithm 6: RRT*

1. $V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
2. **for** $i = 1, \ldots, n$ **do**
3. $\quad x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
4. $\quad x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
5. $\quad x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
6. $\quad$ **if** $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
7. $\quad\quad X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$;
8. $\quad\quad V \leftarrow V \cup \{x_{\text{new}}\}$;
9. $\quad\quad x_{\min} \leftarrow x_{\text{nearest}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
10. $\quad\quad$ **foreach** $x_{\text{near}} \in X_{\text{near}}$ **do** // Connect along a minimum-cost path
11. $\quad\quad\quad$ **if** $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
12. $\quad\quad\quad\quad x_{\min} \leftarrow x_{\text{near}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
13. $\quad\quad E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$;
14. $\quad\quad$ **foreach** $x_{\text{near}} \in X_{\text{near}}$ **do** // Rewire the tree
15. $\quad\quad\quad$ **if** $\texttt{CollisionFree}(x_{\text{new}}, x_{\text{near}}) \wedge \texttt{Cost}(x_{\text{new}}) + c(\texttt{Line}(x_{\text{new}}, x_{\text{near}})) < \texttt{Cost}(x_{\text{near}})$ **then** $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
16. $\quad\quad\quad E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
17. **return** $G = (V, E)$;

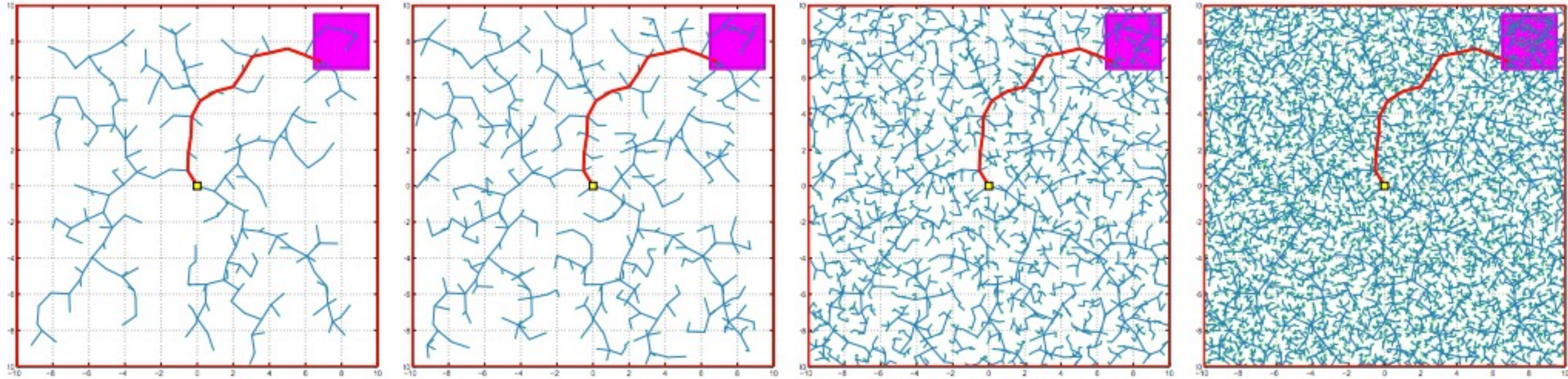
Connect new node to a better parent

Rewire nearby nodes through new node

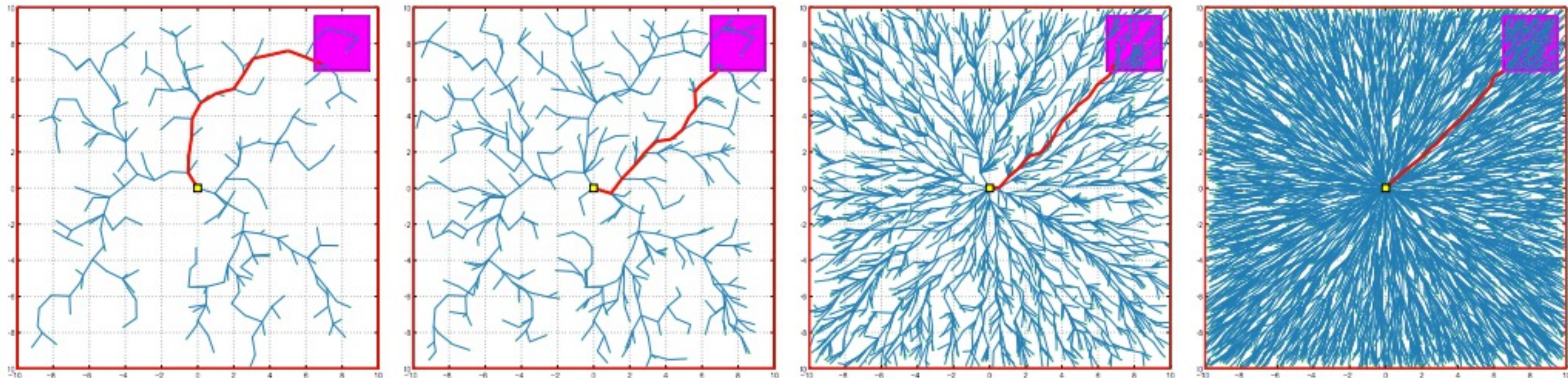
Source: Karaman and Frazzoli

RRT*

RRT

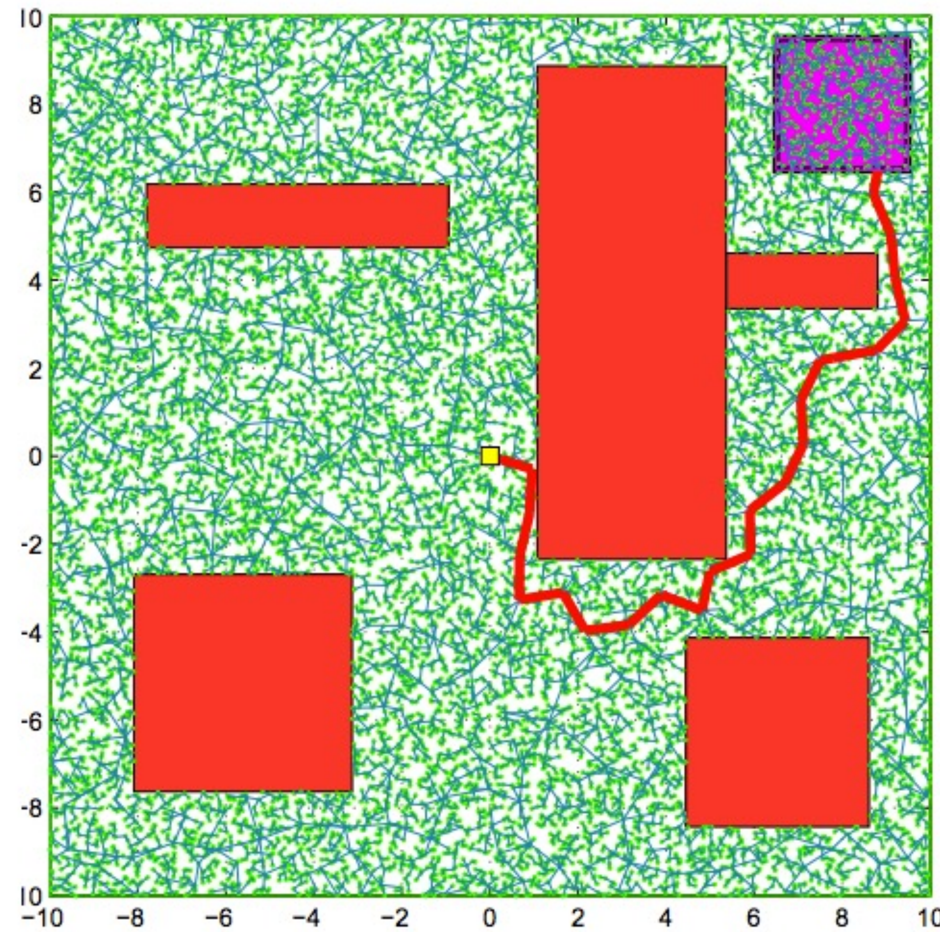


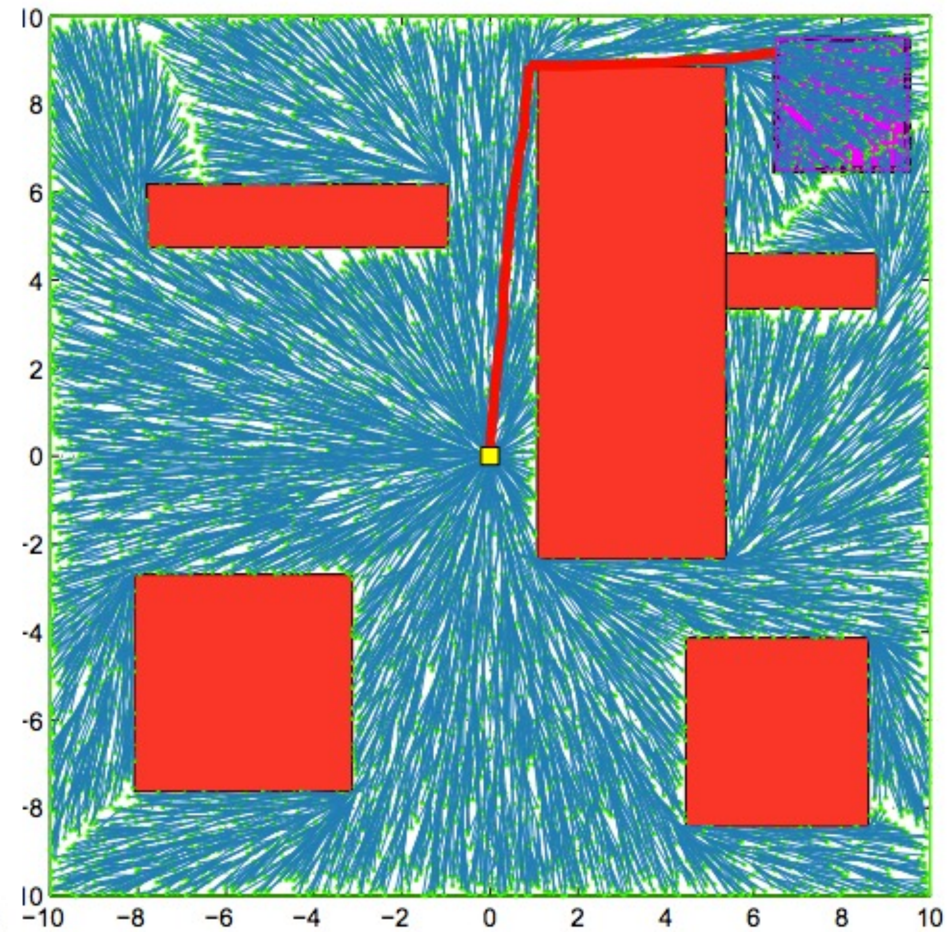
RRT*



Source: Karaman and Frazzoli

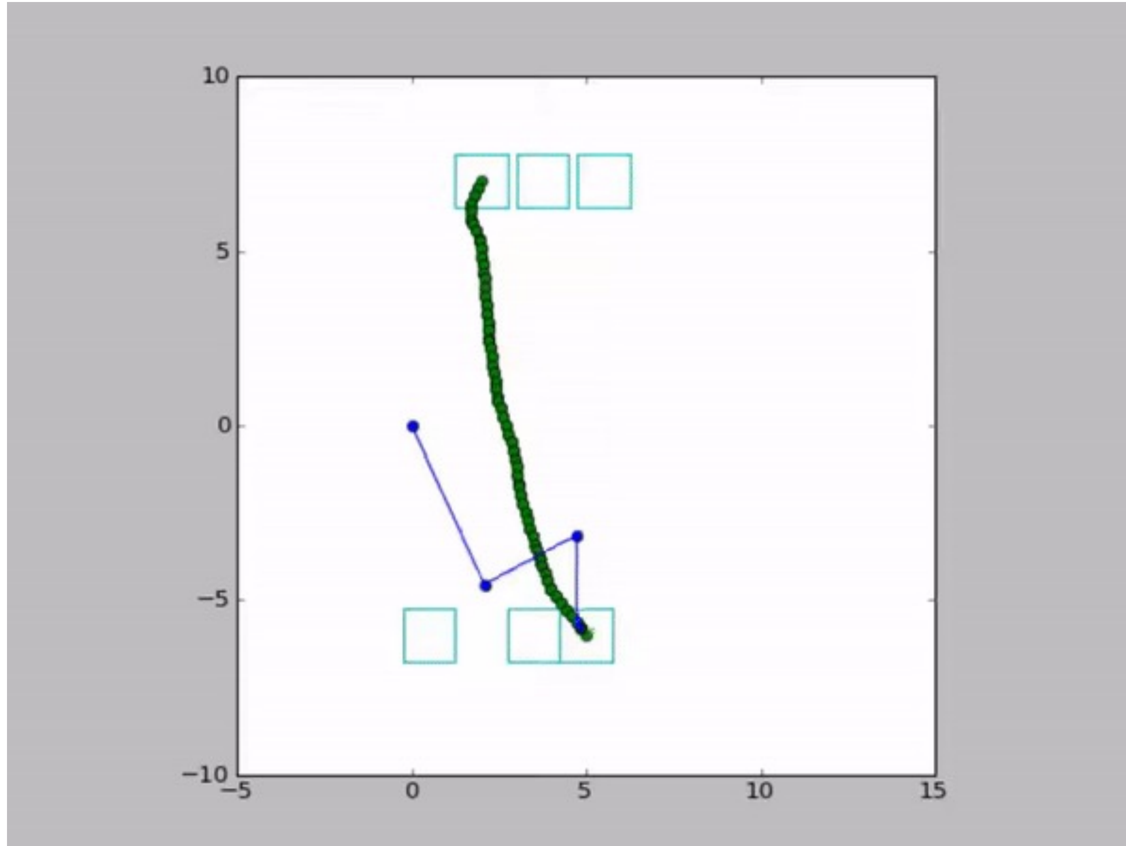
RRT*

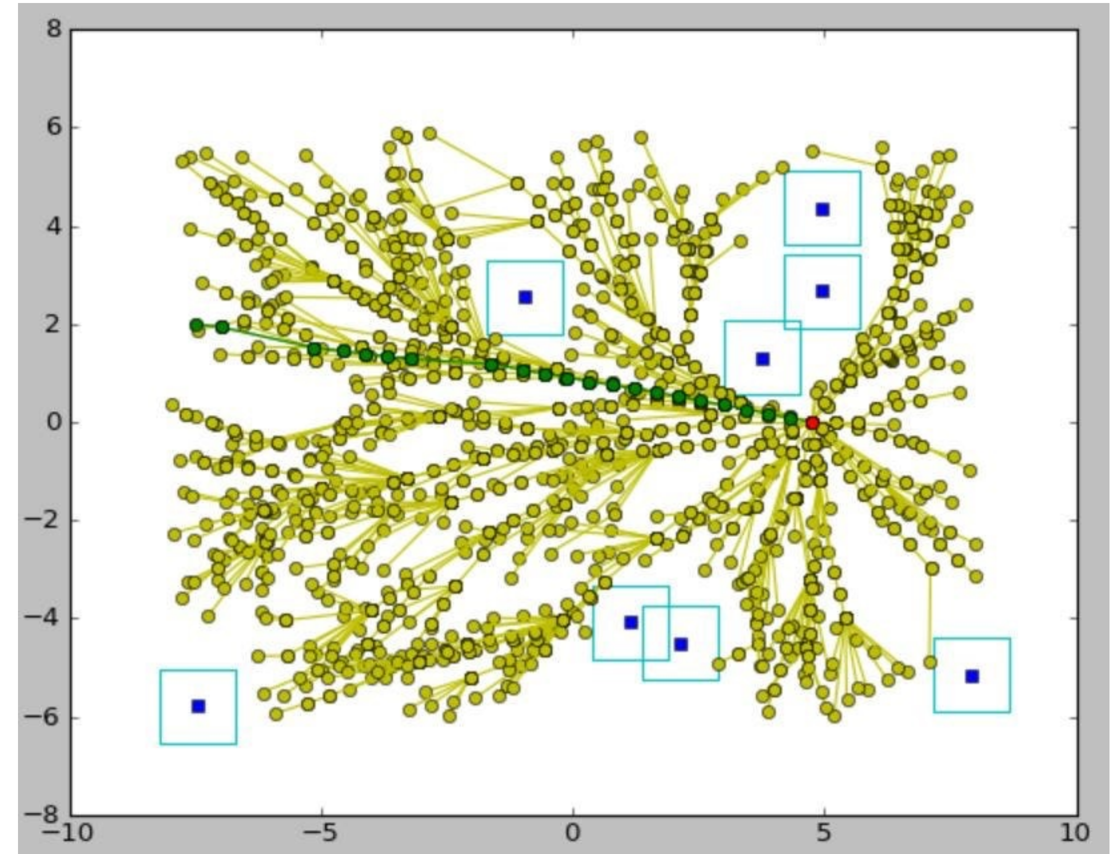




Source: Karaman and Frazzoli

RRT*





Source: Tim Chinenov

Theoretical properties of RRT*

- Probabilistically Complete

Theorem 23 (Probabilistic completeness of RRG and RRT*) *The RRG and RRT* algorithms are probabilistically complete. Furthermore, for any robustly feasible path planning problem* $(\mathcal{X}_{\text{free}}, x_{\text{init}}, \mathcal{X}_{\text{goal}})$, *there exist constants* $a > 0$ *and* $n_0 \in \mathbb{N}$, *both dependent only on* $\mathcal{X}_{\text{free}}$ *and* $\mathcal{X}_{\text{goal}}$, *such that*

$$\mathbb{P}\left(\{V_n^{\text{RRG}} \cap \mathcal{X}_{\text{goal}} \neq \emptyset\}\right) > 1 - e^{-a\,n}, \quad \forall n > n_0,$$

and

$$\mathbb{P}\left(\left\{V_n^{\text{RRT}^*} \cap \mathcal{X}_{\text{goal}} \neq \emptyset\right\}\right) > 1 - e^{-a\,n}, \quad \forall n > n_0.$$

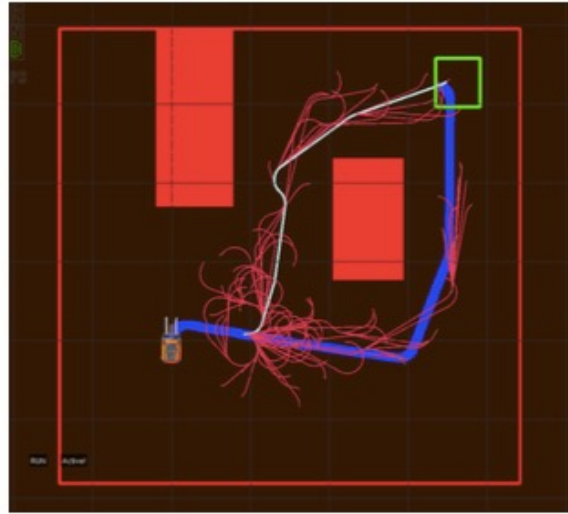
- Asymptotically optimal

Theorem 38 (Asymptotic optimality of RRT*) *If* $\gamma_{\text{RRT}^*} > (2\,(1+1/d))^{1/d} \left(\frac{\mu(X_{\text{free}})}{\zeta_d}\right)^{1/d}$, *then the RRT* algorithm is asymptotically optimal.*

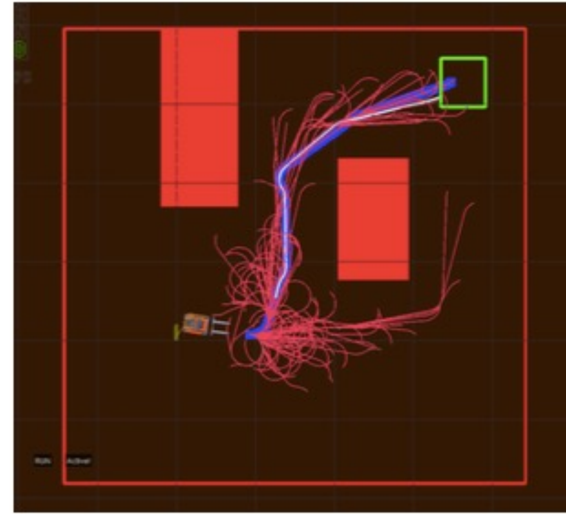
- Efficient

Lemma 42 (PRM*, RRG, and RRT*) $M_n^{\text{PRM}^*}, M_n^{\text{RRG}}, M_n^{\text{RRT}^*} \in O(\log n)$.

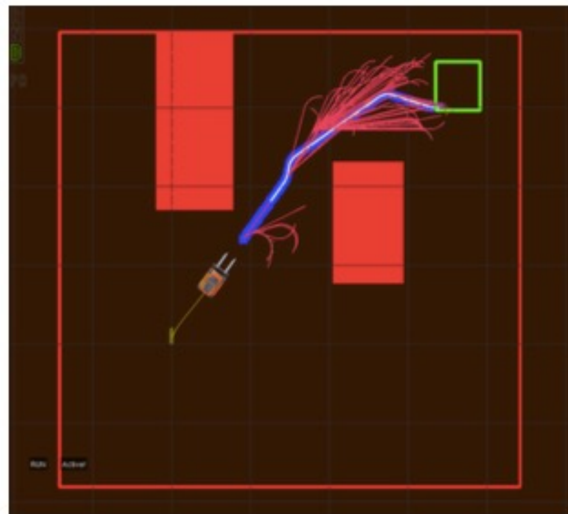
Real Time RRT*

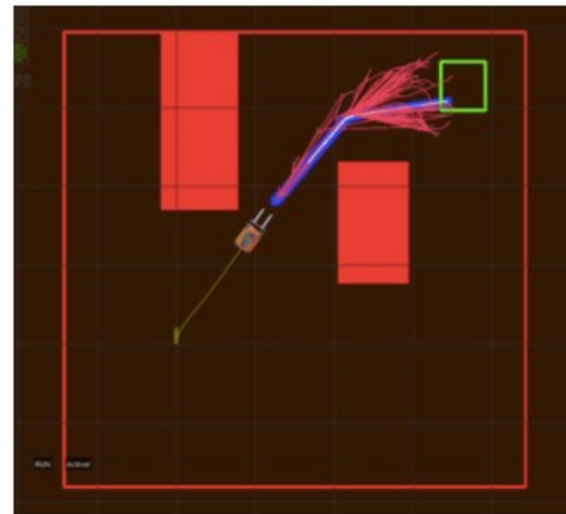


(a) RRT* run 1



(b) RRT* run 1





Combining the best of A* and RRT*

Batch Informed Trees (BIT*)

Sampling-based Optimal Planning via the Heuristically Guided Search of Implicit Random Geometric Graphs

Jonathan D. Gammell[1],
Siddhartha S. Srinivasa[2], and Timothy D. Barfoot[1]

[1]
Institute for Aerospace Studies
UNIVERSITY OF TORONTO

[2] Carnegie Mellon
THE ROBOTICS INSTITUTE

Post Processing for Motion Planning

Randomized motion planners tend to find not so great paths for execution: very jagged, often much longer than necessary.

→ In practice: do smoothing before using the path

- Shortcutting:
  - along the found path, pick two vertices x_{t1}, x_{t2} and try to connect them directly (skipping over all intermediate vertices)
- Nonlinear optimization for optimal control
  - Allows to specify an objective function that includes smoothness in state, control, small control inputs, etc.

Maxim Likhachev on A*/D* vs PRM/RRT

- Sampling-based methods are typically much easier to get working. One of the great thing about RRT is that it doesn't require careful discretization of the action space and instead takes advantage of an extend operator (i.e., local controller or an interpolation function) which naturally exists in most robotics systems

- For planning in a continuous space, when comparing a quick implementation of RRT and a quick implementation of Anytime version of A*, RRT is typically much faster due to sparse exploration of a space.

- A* and its variants are typically harder to implement because they require a) careful design of discretization of the state-space and action-space (to make sure edges land where they are supposed to land); b) careful design of the heuristic function to guide the search well.

Maxim Likhachev on A*/D* vs PRM/RRT

- A* and its variants (including anytime variants) typically generate better quality solutions and very consistent solutions (similar solutions for similar queries) which is beneficial in many domains.
- A* and its variants can often be made nearly as fast as RRT and sometimes even faster if one analyzes the robotic system well to derive a powerful heuristic function. Many robotic systems have natural low-dimensional manifolds (e.g., a 3D workspace for example) that can be used to derive such heuristic functions.
- A* and its variants can be applied to both discrete and continuous (as well as hybrid) systems, whereas sampling-based systems tend to be more suitable for continuous systems since they rely on the idea of sparse exploration. (Within the same point, it should be noted that A* and its variants apply to PRMs and its variants. PRM is just a particular graph representation of the environment.)

Maxim Likhachev on A*/D* vs PRM/RRT

- In summary, I think for continuous planning problems, A* and its variants require substantially more development efforts (careful analysis of the system to derive proper graph representation and a good heuristic function) but can result in a better performance (similar speed but better quality solutions and more consistent behavior).

Lecture Outline

Incremental Search – LPA*

Sampling Based Motion Planning - PRMs

RRT and RRT*

Recap: Course Overview

Filtering/Smoothing
Localization
Mapping
SLAM

Search
Motion Planning
TrajOpt
Stability/Certification

MDPs and RL
Imitation Learning
Solving POMDPs

Additional Resources

- Marco Pavone (http://asl.stanford.edu/):
  - Sampling-based motion planning on GPUs: https://arxiv.org/pdf/1705.02403.pdf
  - Learning sampling distributions: https://arxiv.org/pdf/1709.05448.pdf
- Siddhartha Srinivasa (https://personalrobotics.cs.washington.edu/)
  - Batch informed trees: https://robotic-esp.com/code/bitstar/
  - Expensive edge evals: https://arxiv.org/pdf/2002.11853.pdf
- Michael Yip (https://www.ucsdarclab.com/)
  - Neural Motion Planners: https://www.ucsdarclab.com/neuralplanning
- Lydia Kavraki (http://www.kavrakilab.org/)
  - Motion in human workspaces: http://www.kavrakilab.org/nsf-nri-1317849.html