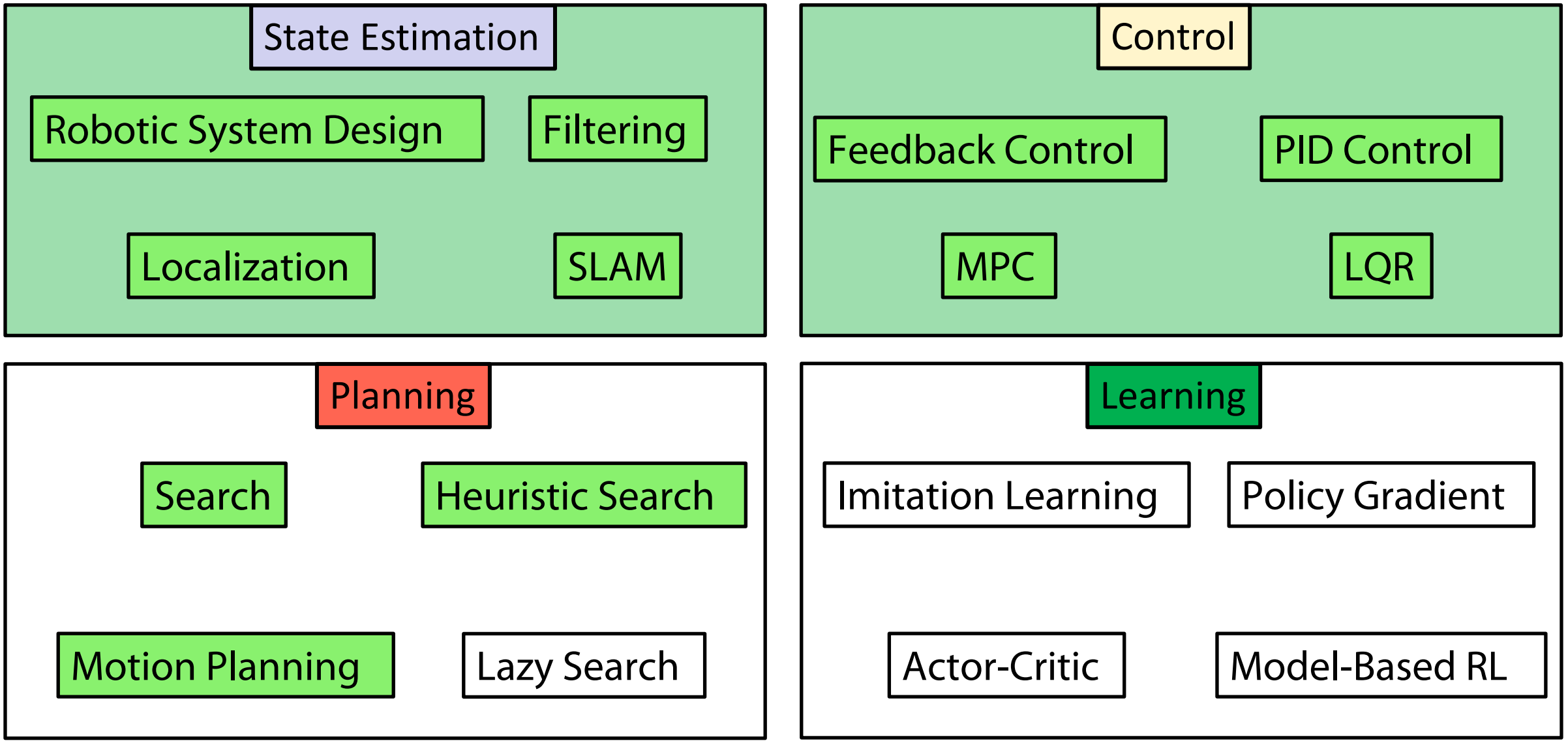
Autonomous Robotics
Winter 2025

Abhishek Gupta

TAs: Carolina Higuera, Entong Su, Bernie Zhu

Slides borrowed from many sources – Sidd Srinivasa, Sanjiban Choudhury, M Likhachev

Class Outline



A admissible heuristic is

0

Lecture Outline

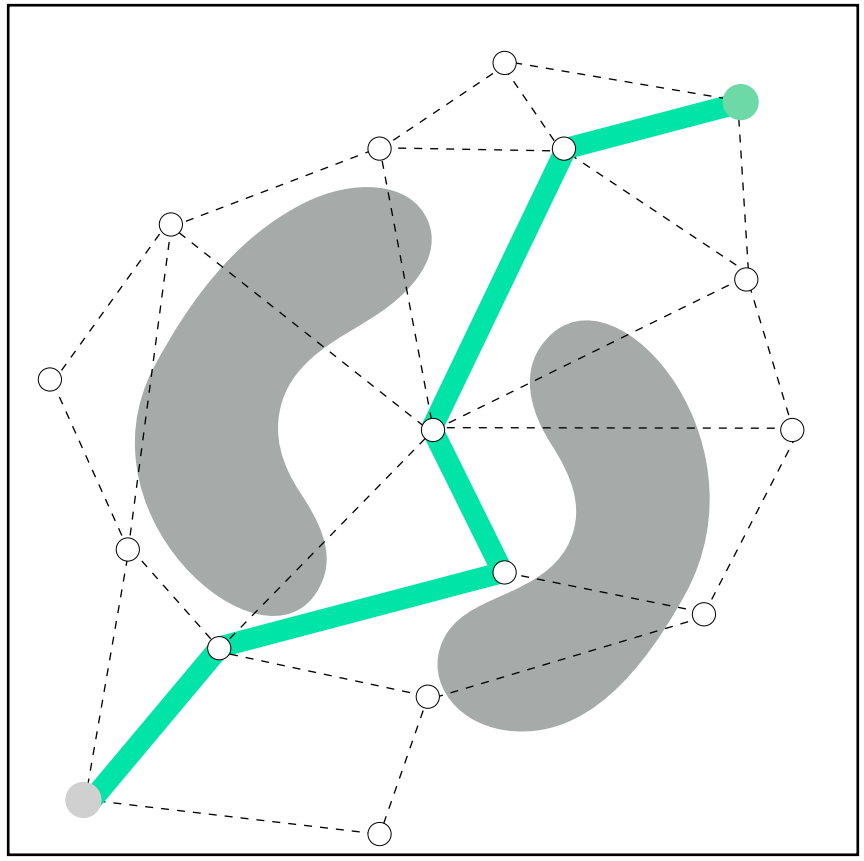
Recap

Lazy A*

From PRMs to RRT

RRT to RRT*

Minimal Cost Path on a Graph



START, GOAL

COST (E.G. LENGTH)

GRAPH (VERTICES, EDGES)

Pseudocode

Push *start* into OPEN

While *goal* not expanded

Pop *best* from OPEN

Add *best* to CLOSED

For every successor s'

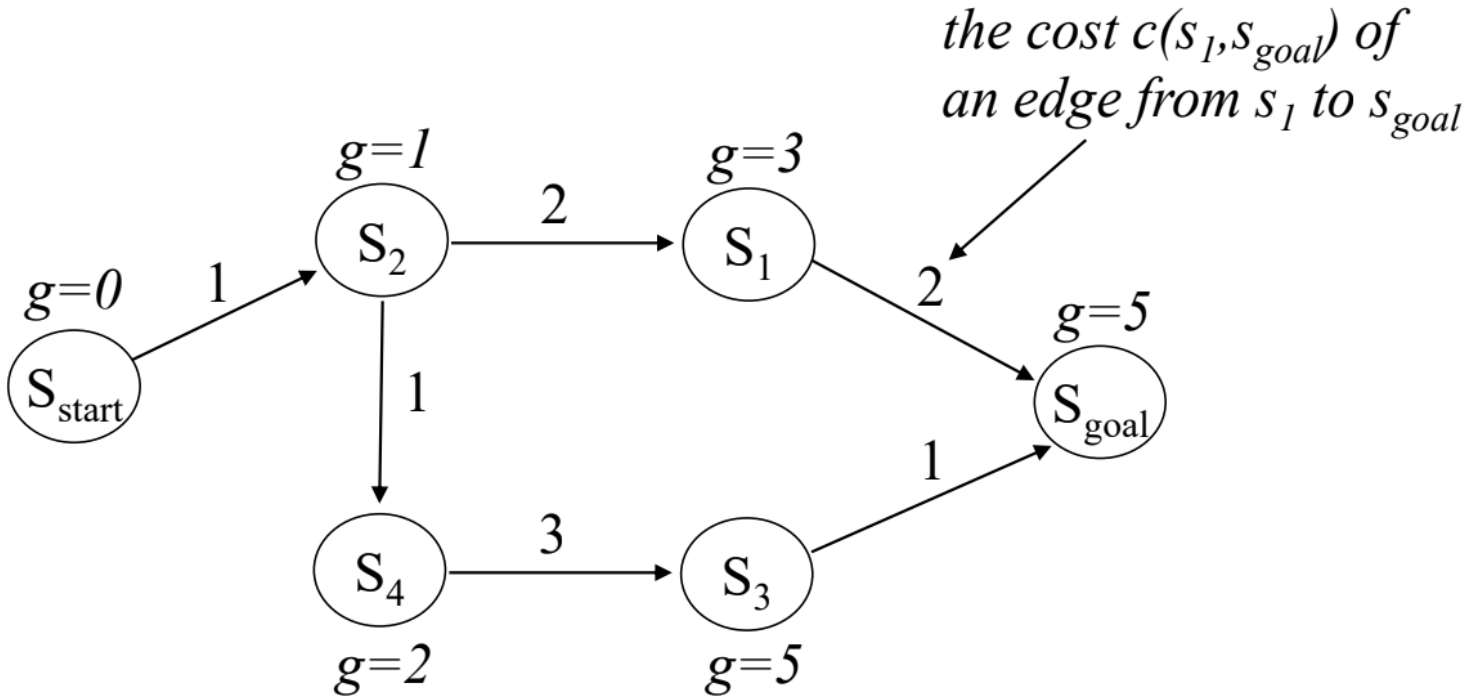
If $g(s') > g(s) + c(s,s')$

$g(s') = g(s) + c(s,s')$

Add (or update) s' to OPEN

Dijkstra's Algorithm

– optimal values satisfy: $g(s) = \min_{s'' \in pred(s)} g(s'') + c(s'',s)$



Nice property:

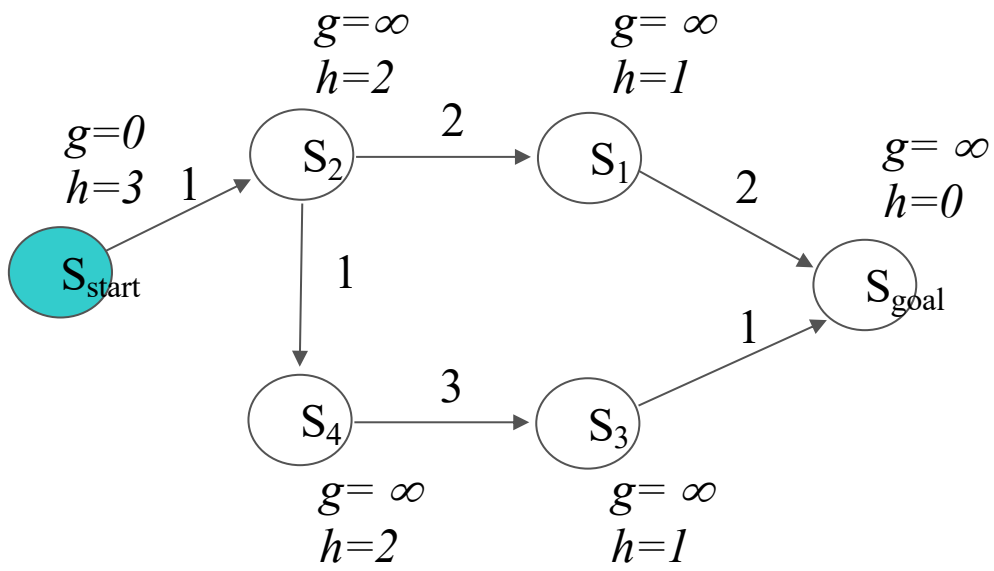
Only process nodes ONCE. Only process cheaper nodes than goal.

A* Search

- Computes optimal g-values for relevant states

while(s_{goal} is not expanded)
  remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
  insert s into *CLOSED*;
  for every successor s' of s such that s' not in *CLOSED*
    if $g(s') > g(s) + c(s,s')$
      $g(s') = g(s) + c(s,s')$;
      insert s' into *OPEN*;

CLOSED = {}
OPEN = $\{s_{start}\}$
next state to expand: s_{start}



Properties of heuristics

What properties should *h(s)* satisfy? How does it affect search?

Admissible: h(s) <= h*(s) h(goal) = 0

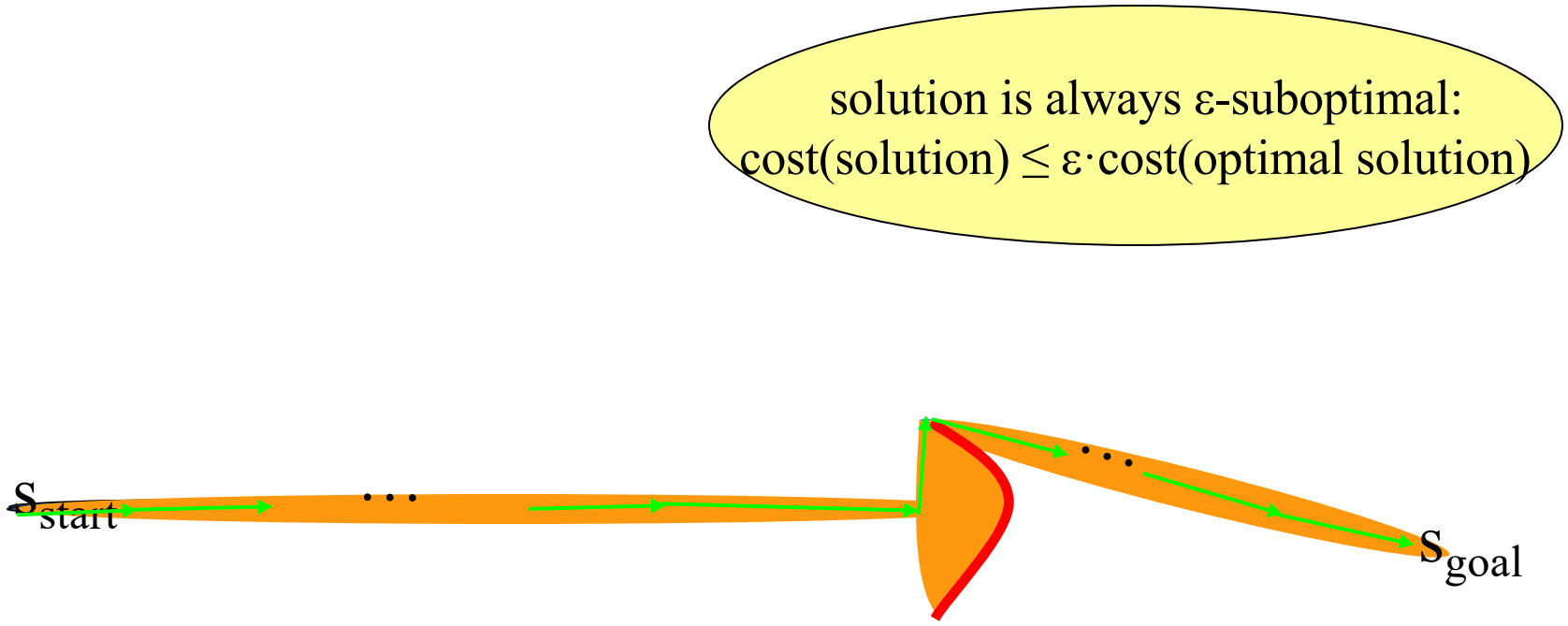
If this true, the path returned by A* is optimal

Consistency: h(s) <= c(s,s') + h(s') h(goal) = 0

If this true, A* is optimal AND efficient (will not re-expand a node)

Effect of the Heuristic Function

- Weighted A* Search: expands states in the order of $f = g+\varepsilon h$ values, $\varepsilon > 1$ = bias towards states that are closer to goal



Courtesy Max Likhachev

Lecture Outline

Recap

Lazy A*

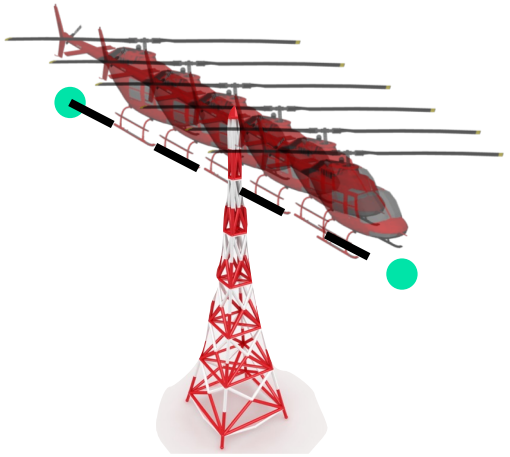
From PRMs to RRT

RRT to RRT*

But is the number of expansions really what we want to minimize in motion planning?

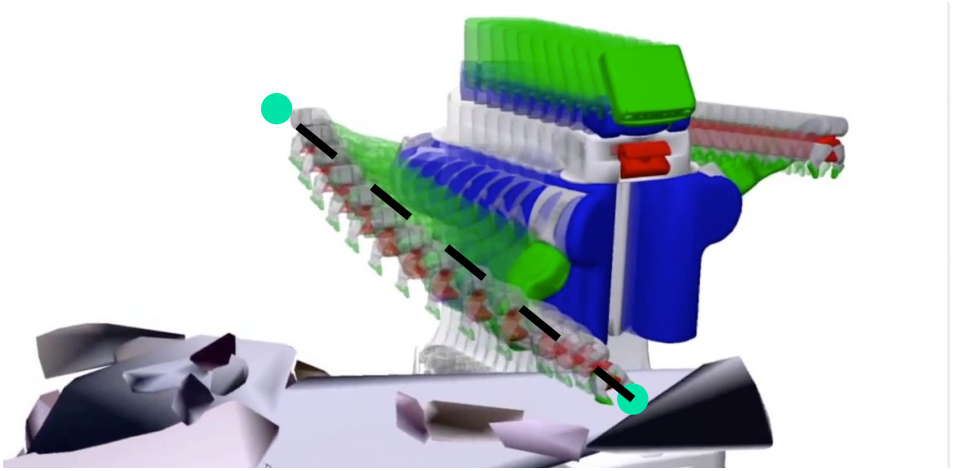
What is the most expensive step?

Edge evaluation is expensive



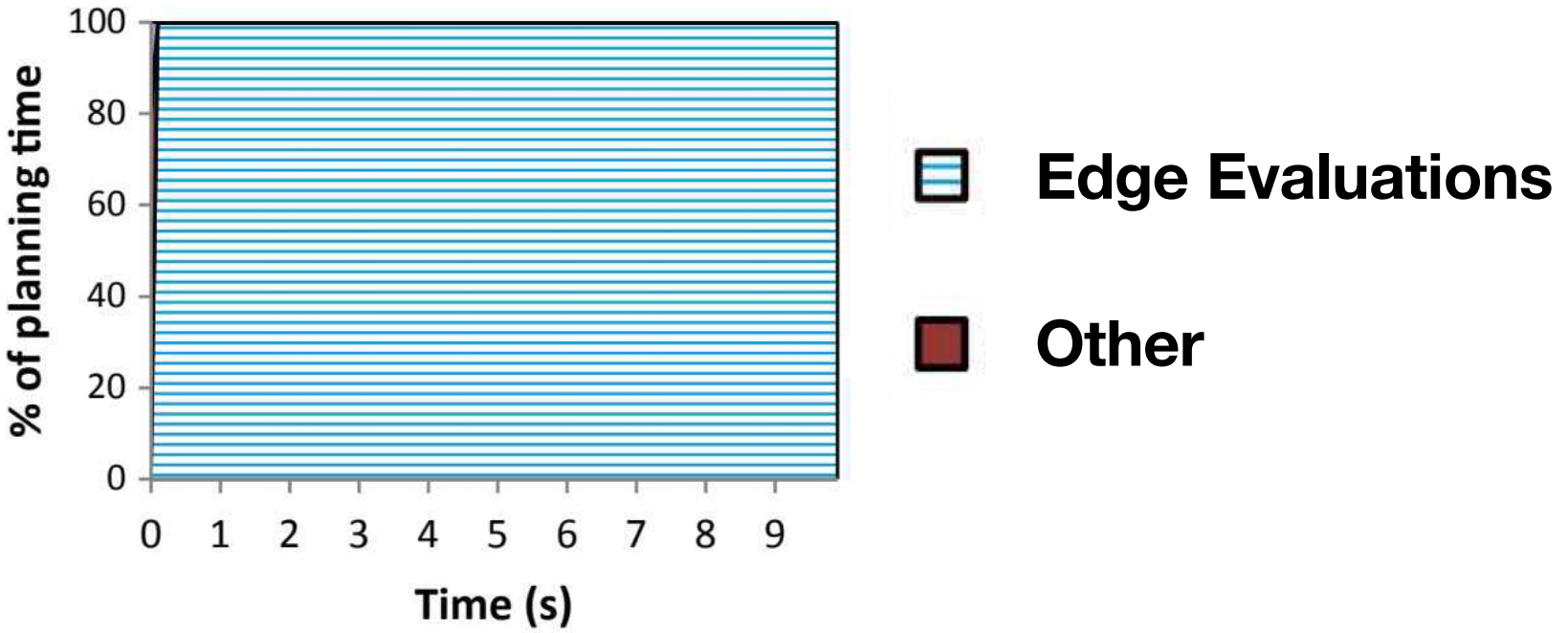
Check if helicopter intersects with tower

(Schulman et al. ’14)



Check if manipulator intersects with table

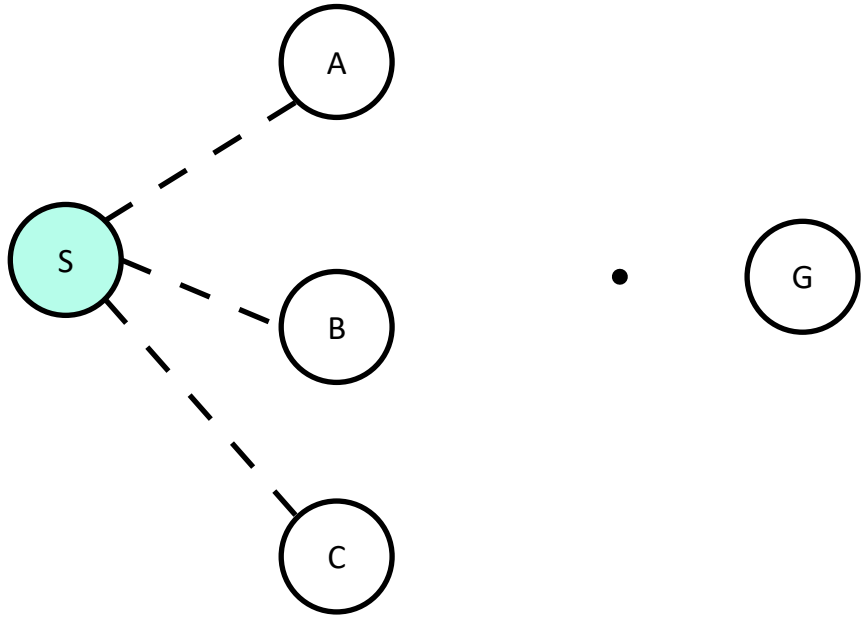
Edge evaluation dominates planning time



Hauser, Kris., Lazy collision checking in asymptotically-optimal motion planning. *ICRA* 2015

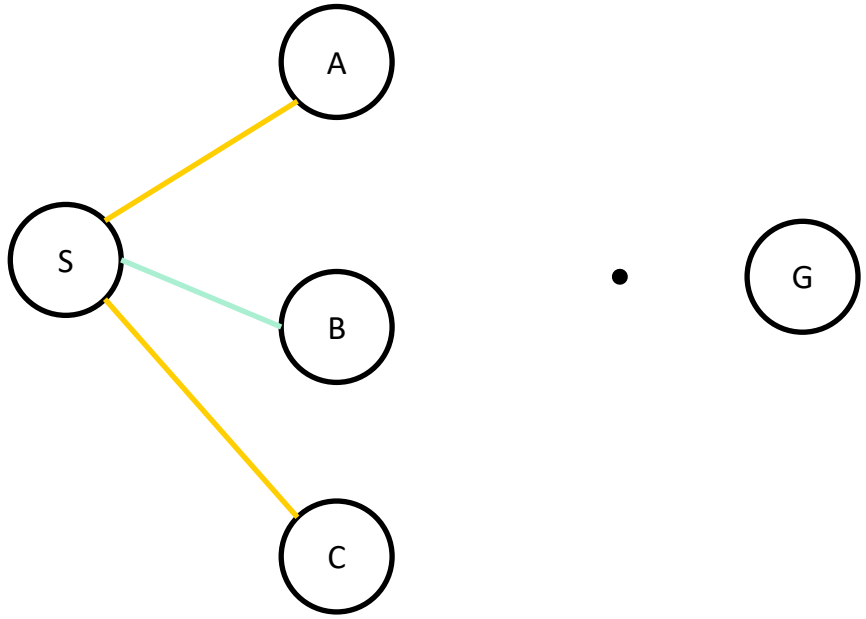
Let’s revisit Best First Search

Element (Node)	**Priority Value (f-value)**
Node S	f(S)



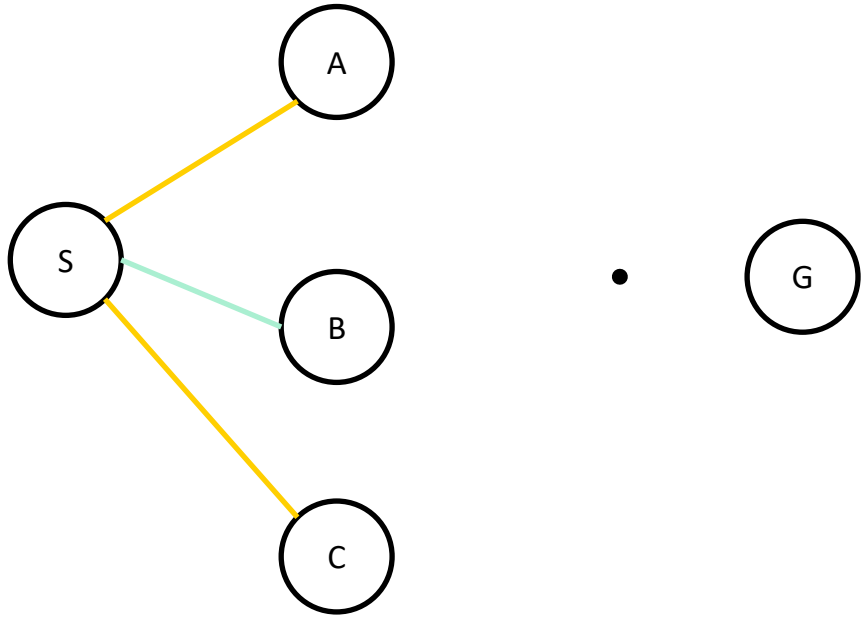
Let’s revisit Best First Search

Element (Node)	**Priority Value (f-value)**
~~Node S~~	~~f(S)~~
Node A	f(A)
Node C	f(C)



What if we never use C? Wasted collision check!

Element (Node)	**Priority Value (f-value)**
~~Node S~~	~~f(S)~~
Node A	f(A)
Node C	f(C)



The provable virtue of laziness:

Take the thing that's expensive (collision checking)

and

procrastinate as long as possible
till you have to evaluate it!

Lazy (weighted) A*

Cohen, Phillips, and Likhachev 2014

Key Idea:

1. When expanding a node, don't collision check edge to successors

   (be optimistic and assume the edge will be valid)

2. When expanding a node, collision check the edge to parent

   (expansion means this node is good and worth the effort)

3. Important: OPEN list will have multiple copies of a node

   (multiple candidate parents since we haven't collision check)

Lazy A*

Cohen, Phillips, and Likhachev 2014

Non lazy A*

while(s_{goal} is not expanded)
  remove *s* with the smallest *[f(s) = g(s)+h(s)]* from *OPEN*;

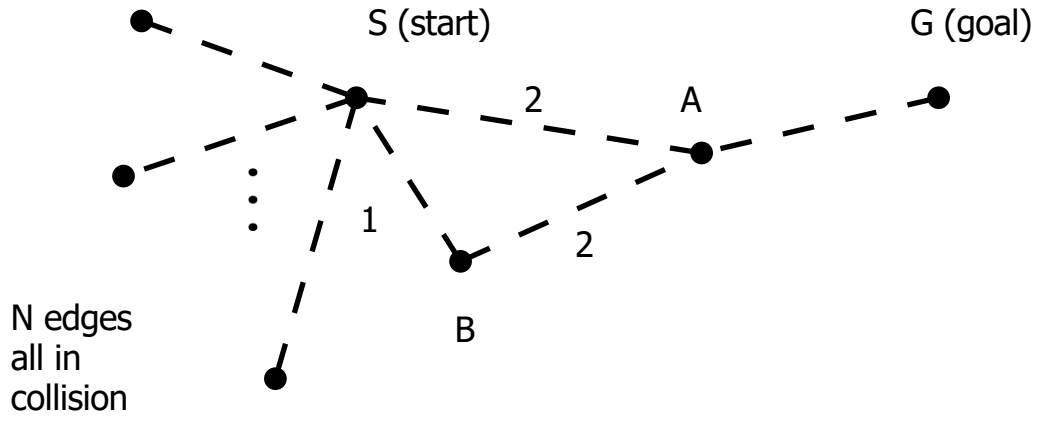
  insert *s* into *CLOSED*;
  for every successor *s'* of *s* such that *s'* not in *CLOSED*
    if *edge (s,s')* in collision
      c(s,s') = ∞
    if *g(s') > g(s) + c(s,s')*
      g(s') = g(s) + c(s,s');
      insert *s'* into *OPEN*;

Lazy A*

while(s_{goal} is not expanded)
  remove *s* with the smallest *[f(s) = g(s)+h(s)]* from *OPEN*;
  if s is in CLOSED
    continue;
  if *edge(parent(s), s)* in collision
    continue;
  insert *s* into *CLOSED*;
  for every successor *s'* of *s* such that *s'* not in *CLOSED*
    no collision checking of edge
    if *g(s') > g(s) + c(s,s')*
      g(s') = g(s) + c(s,s');
      insert *s'* into *OPEN*; // multiple copies

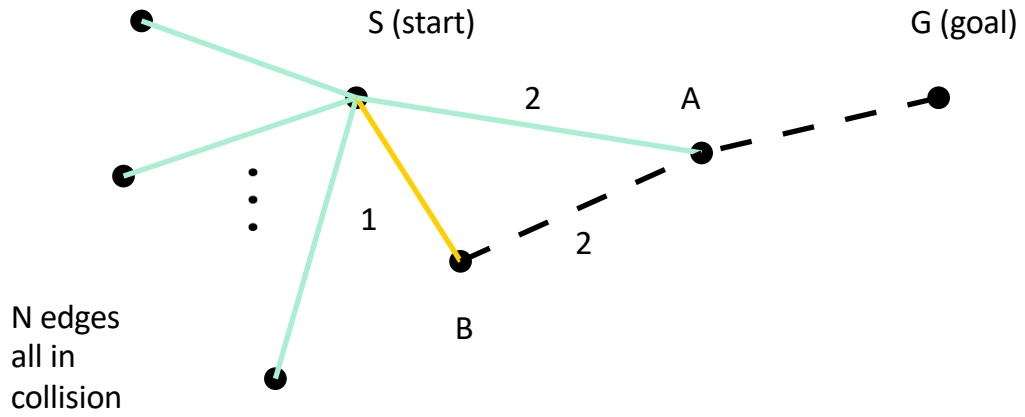
A*

Let’s say S-A is in collision and true shortest path is S-B-A-G



A*

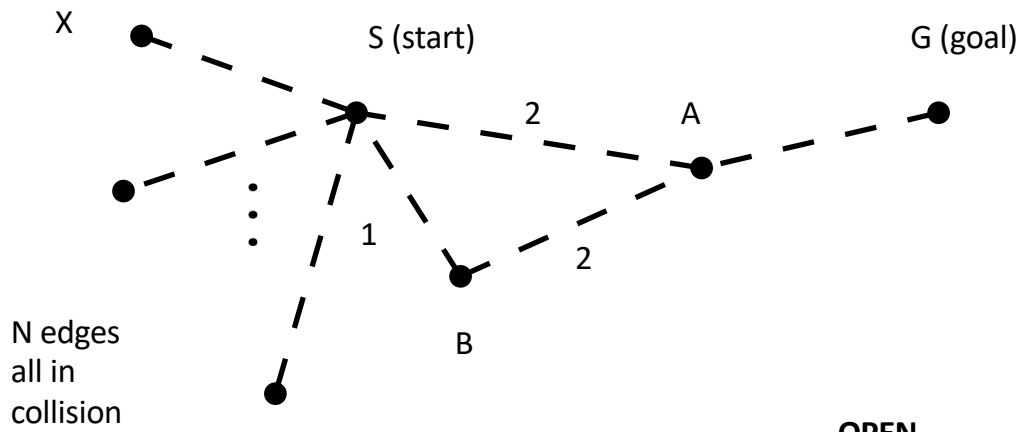
Let’s say S-A is in collision and true shortest path is S-B-A-G



A* will
collision check
all N+2 edges!

Lazy A*

Let’s say S-A is in collision and true shortest path is S-B-A-G

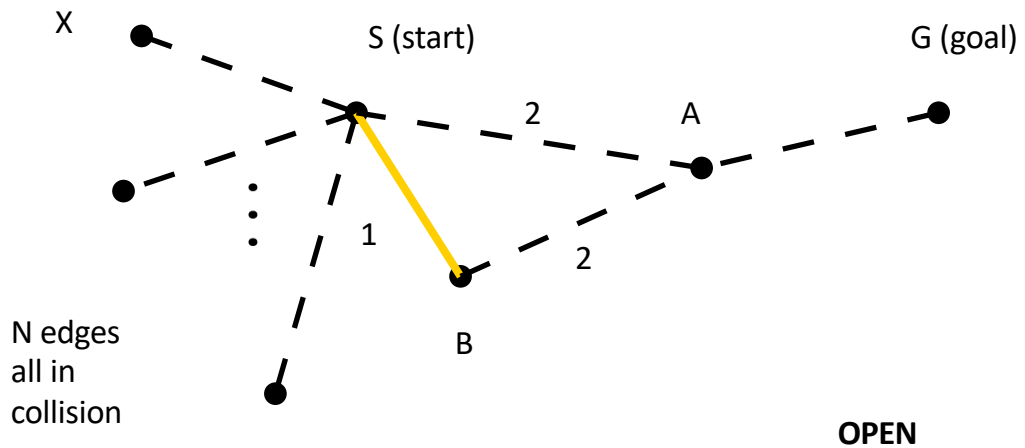


Lets set f(s) = g(s)

	OPEN	**CLOSED**	**CollChecked**
f = 1	B (from S)	S	
f = 2	A (from S)		
f = 1000	X (from S)		
	…..		

Lazy A*

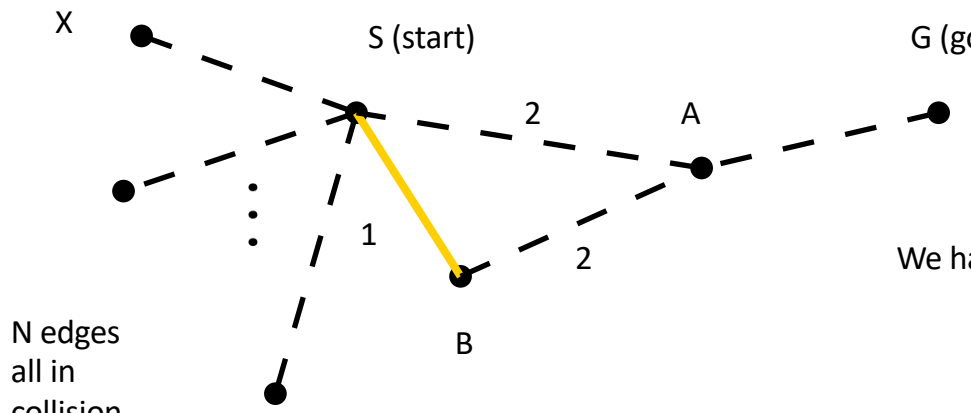
Let’s say S-A is in collision and true shortest path is S-B-A-G



	OPEN	CLOSED	CollChecked
f = 2	A (from S)	S	S-B
f = 3	A (from B)	B	
	…..		

Lazy A*

Let’s say S-A is in collision and true shortest path is S-B-A-G

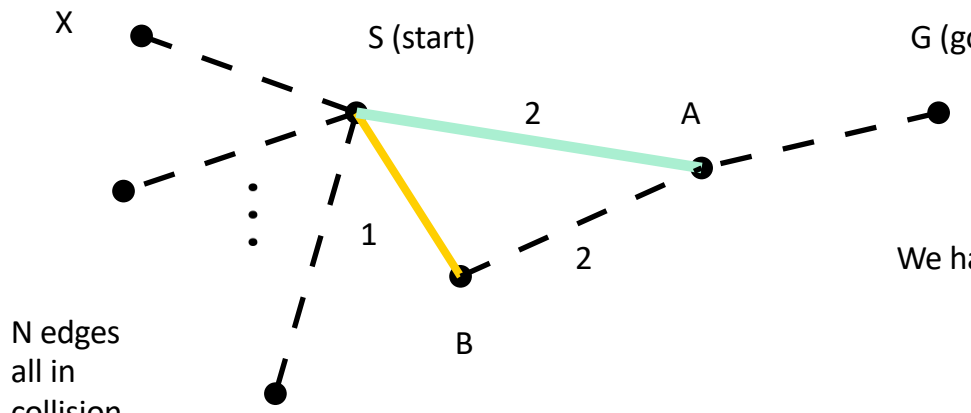


We have TWO copies of A in OPEN!

	OPEN	**CLOSED**	**CollChecked**
f = 2	A (from S)	S	S-B
f = 3	A (from B)	B	
	…..		

Lazy A*

Let’s say S-A is in collision and true shortest path is S-B-A-G

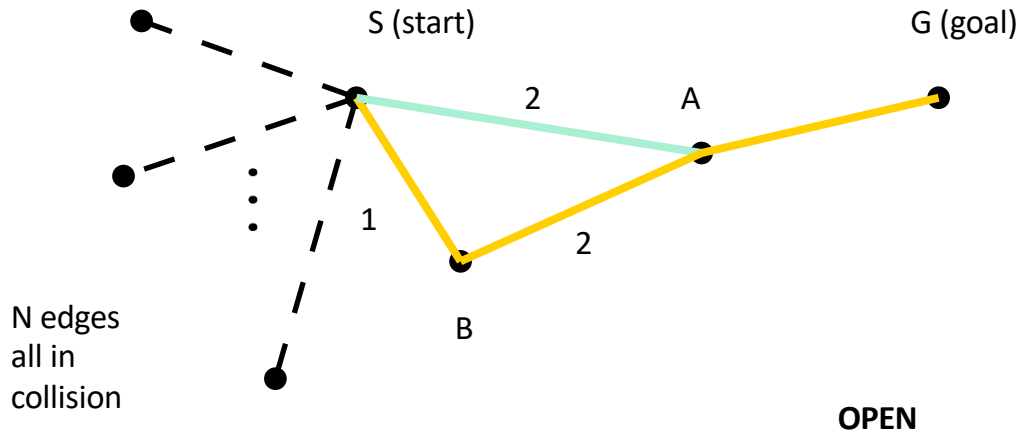


We have TWO copies of A in OPEN!

	OPEN	**CLOSED**	**CollChecked**
f = 3	A (from B)	S	S-B
		B	S-A
	…..	A	

Lazy A*

Let’s say S-A is in collision and true shortest path is S-B-A-G



OPEN	CLOSED	CollChecked
	S	S-B
	B	S-A
	A	B-A
	G	A-G

Lecture Outline

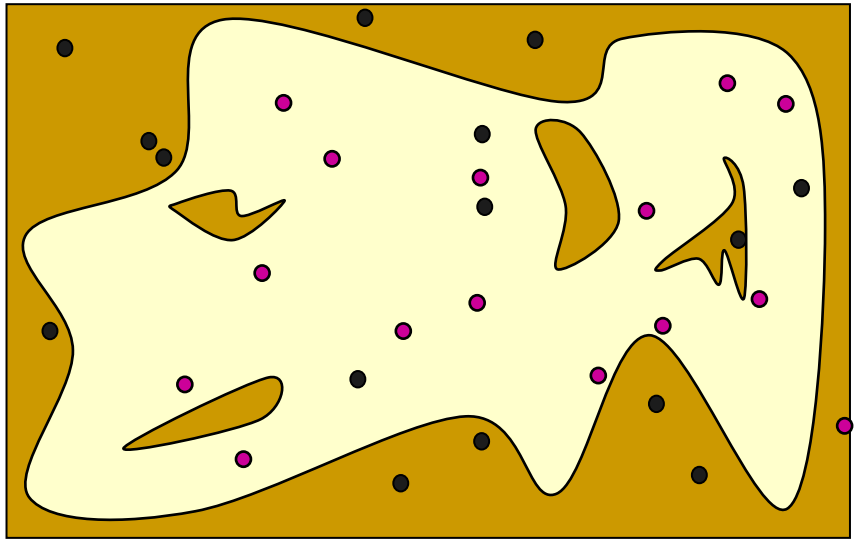
Recap

Lazy A*

From PRMs to RRT

RRT to RRT*

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?

Rapidly exploring Random Tree (RRT)

Steve LaValle (98)

- Basic idea:
  - A* → explicitly sample the whole graph to store in memory
  - RRT → sample the tree on the fly as you keep going, grow an "implicit graph"
    - Avoid memory costs
    - Avoid solving the 2 point BVP for non-holonomic control

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

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 $x_{rand} \leftarrow$ RANDOM_STATE();
4 $x_{near} \leftarrow$ NEAREST_NEIGHBOR($x_{rand}, \mathcal{T}$);
5 $u \leftarrow$ SELECT_INPUT(x_{rand}, x_{near});
6 $x_{new} \leftarrow$ NEW_STATE($x_{near}, u, \Delta t$);
7 $\mathcal{T}$.add_vertex(x_{new});
8 $\mathcal{T}$.add_edge(x_{near}, x_{new}, u);
9 Return $\mathcal{T}$

Sample a new state

Find closest point

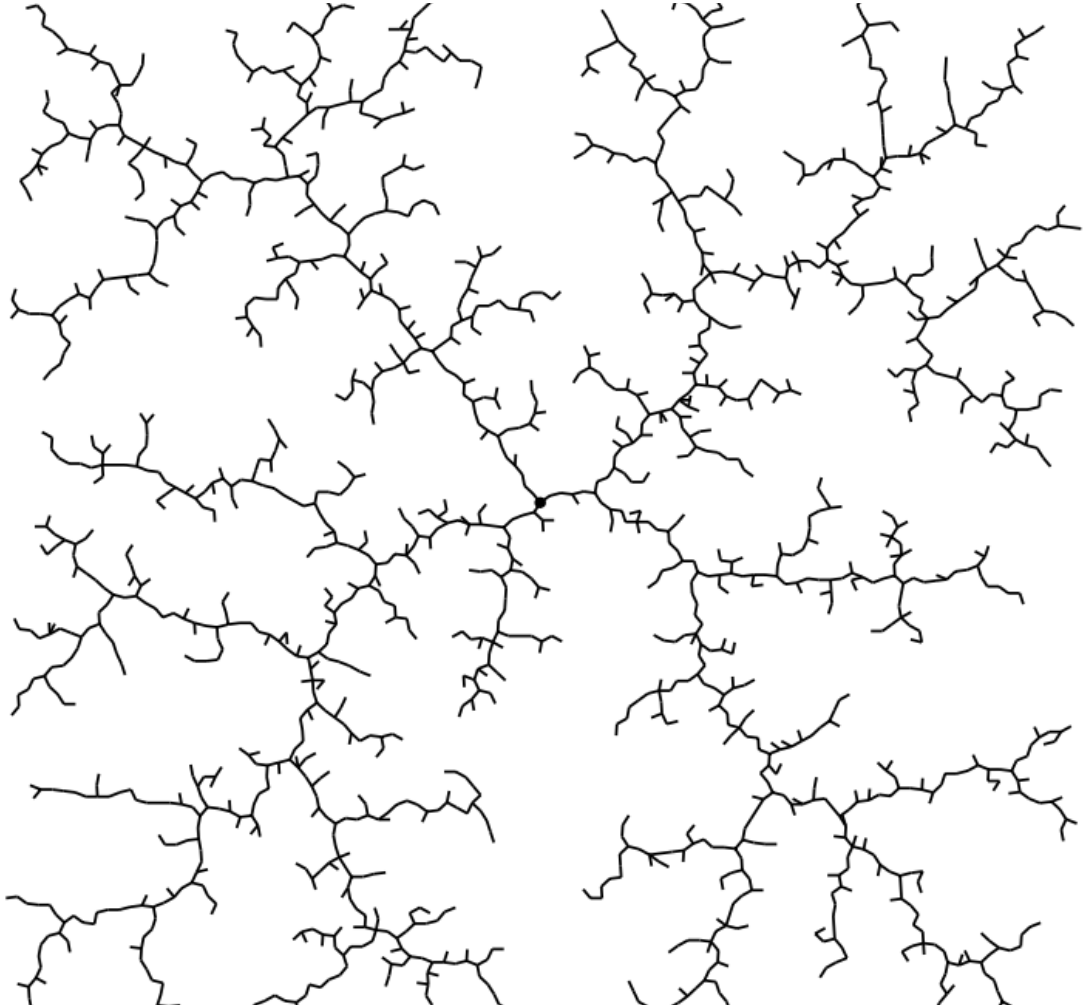
Approximately find controls

Steer forward

Add to tree

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

How to Sample

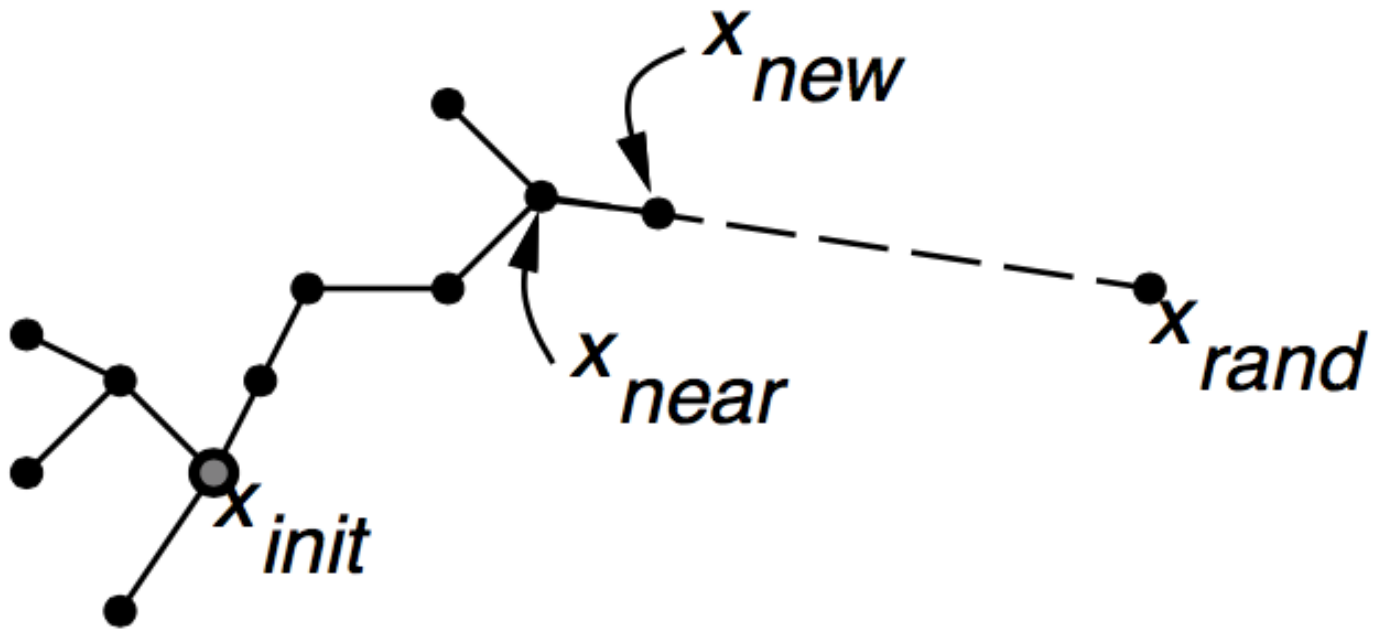


Rapidly exploring Random Tree (RRT)

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

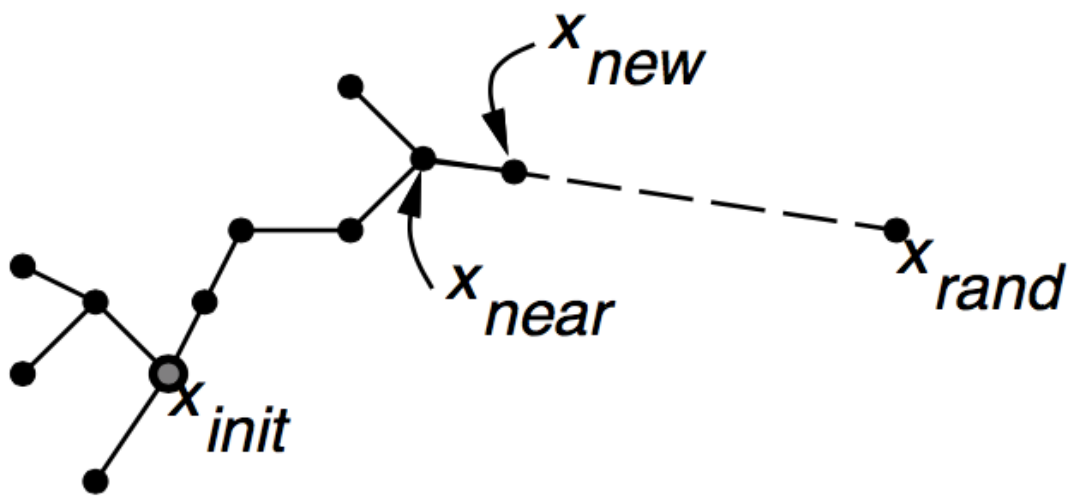
Rapidly exploring Random Tree (RRT)

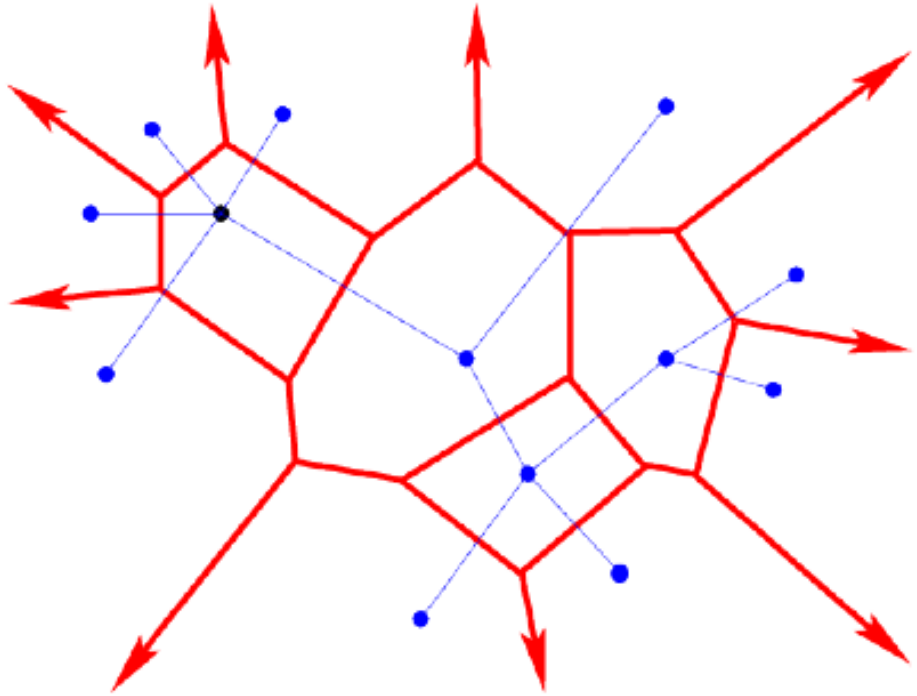
- Select random point, and expand nearest vertex towards it
  - Biases samples towards largest Voronoi region



Rapidly exploring Random Tree (RRT)

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





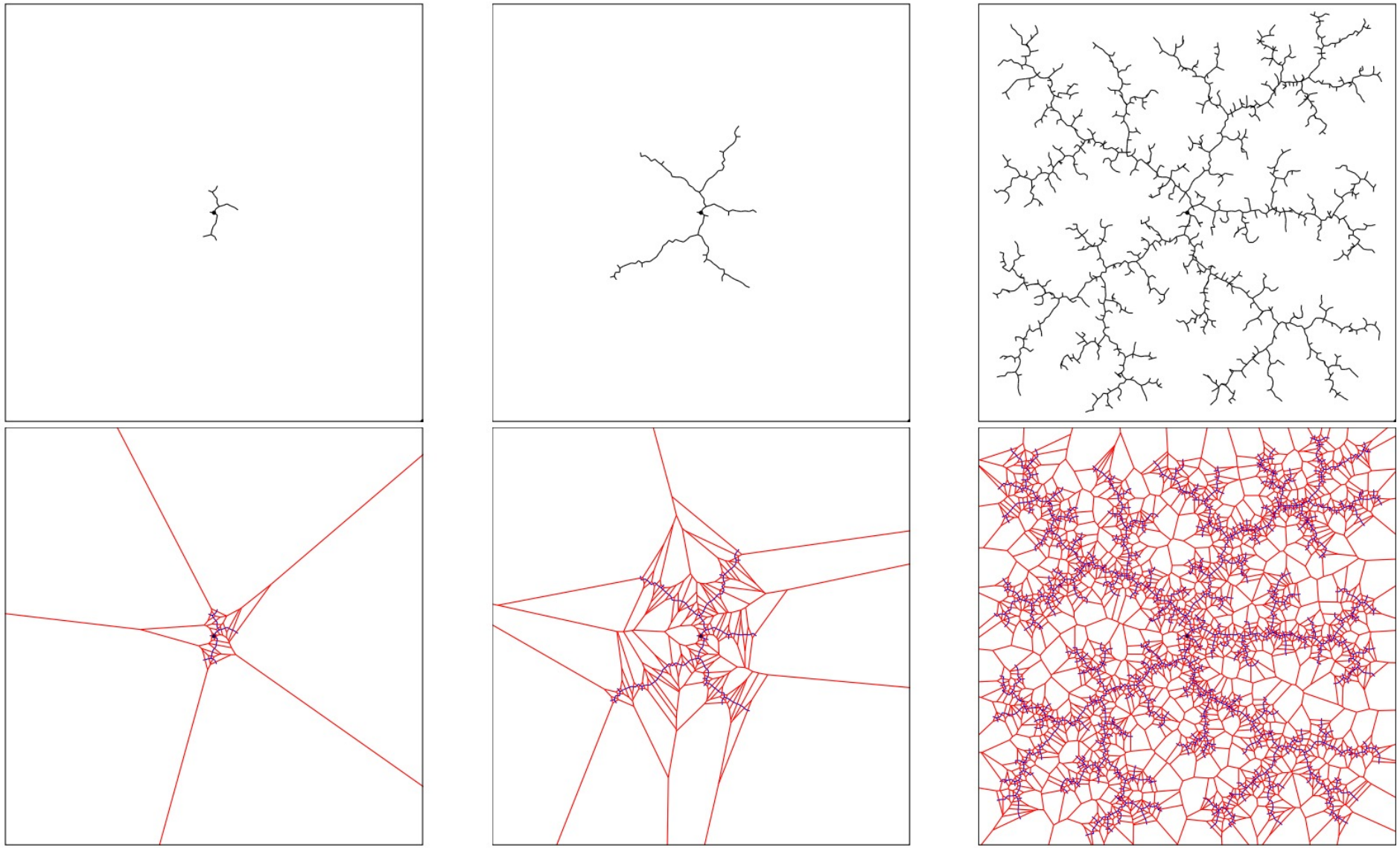
Rapidly exploring Random Tree (RRT)

GENERATE_RRT($x_{init}, K, \Delta t$)

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



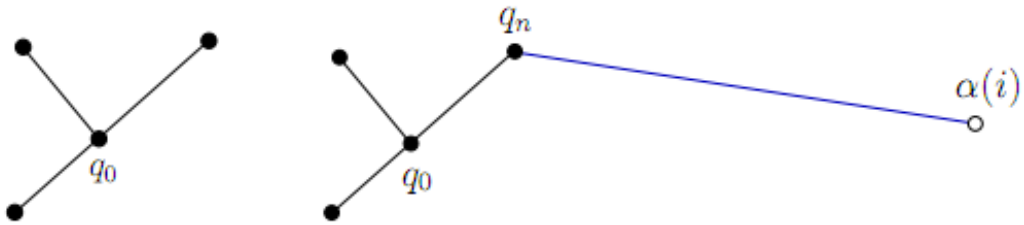
Source: LaValle and Kuffner 01

RRT Practicalities

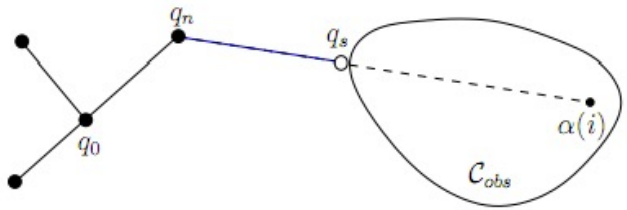
- NEAREST_NEIGHBOR(x_{rand}, T): need to find (approximate) nearest neighbor efficiently
  - KD Trees data structure (upto 20-D) [e.g., FLANN]
  - Locality Sensitive Hashing
- SELECT_INPUT(x_{rand}, x_{near})
  - **Approximate** two point boundary value problem
    - If too hard to solve, often just 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:

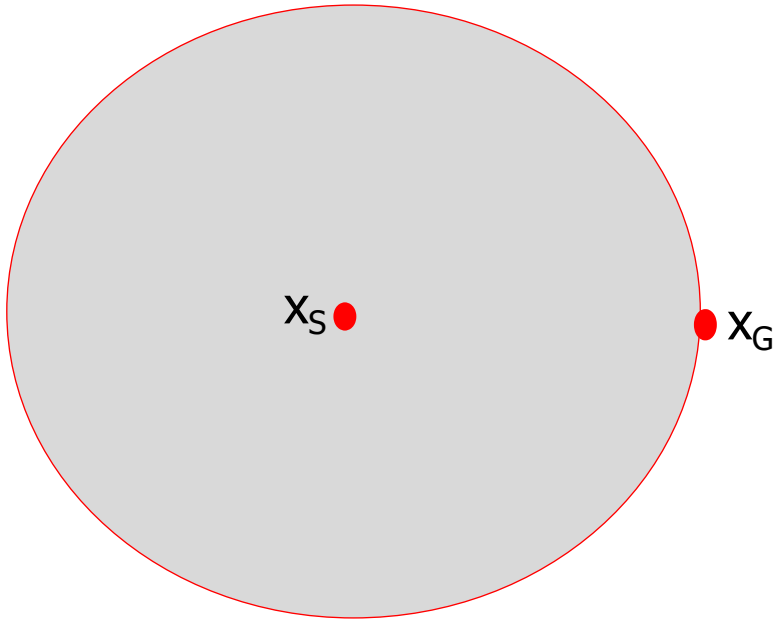


- Non-holonomic: approximately (sometimes as approximate as picking best of a few random control sequences) solve two-point boundary value problem

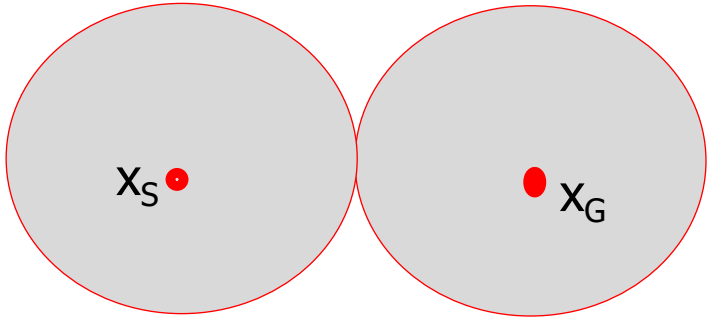
Growing RRT

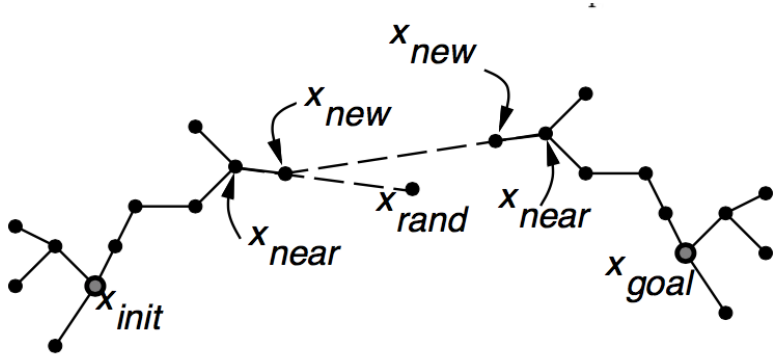
Bi-directional RRT

- Volume swept out by unidirectional RRT:



- Volume swept out by bi-directional RRT:

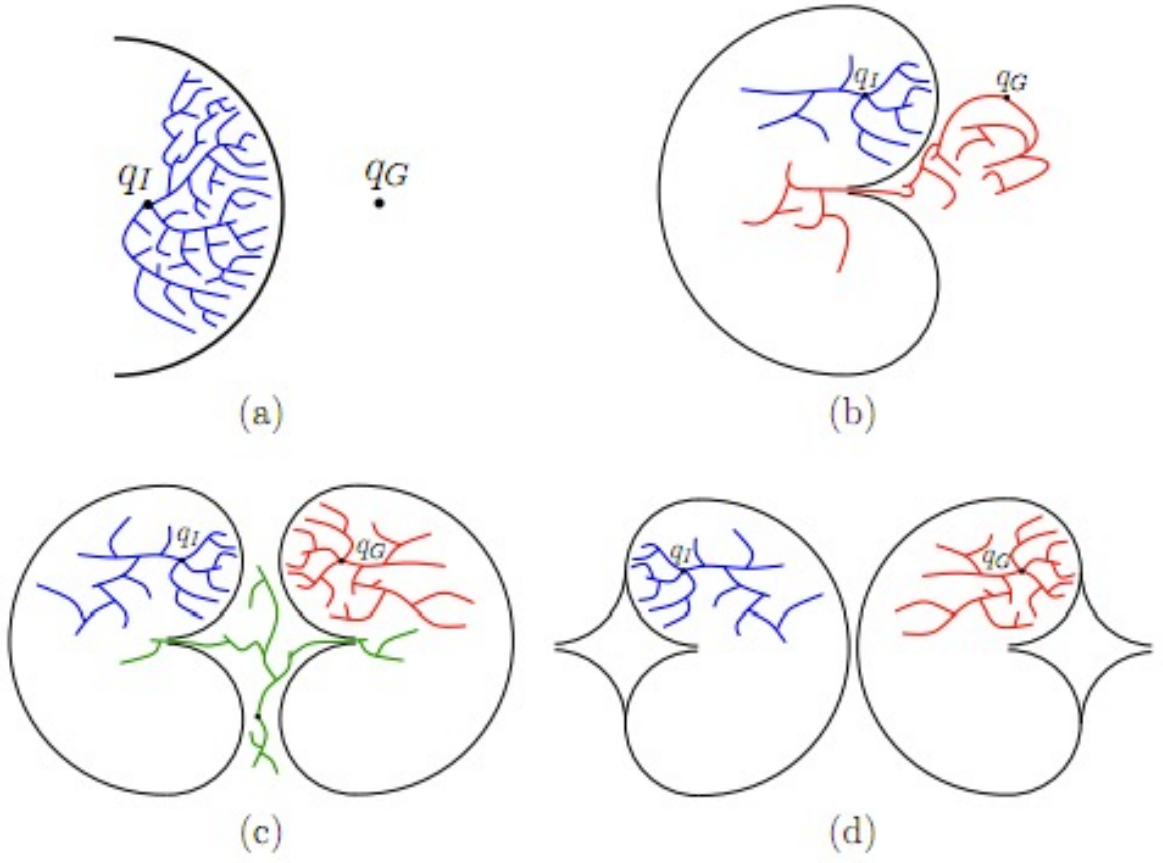




- Difference more and more pronounced as dimensionality increases

Multi-directional RRT

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



Lecture Outline

Recap

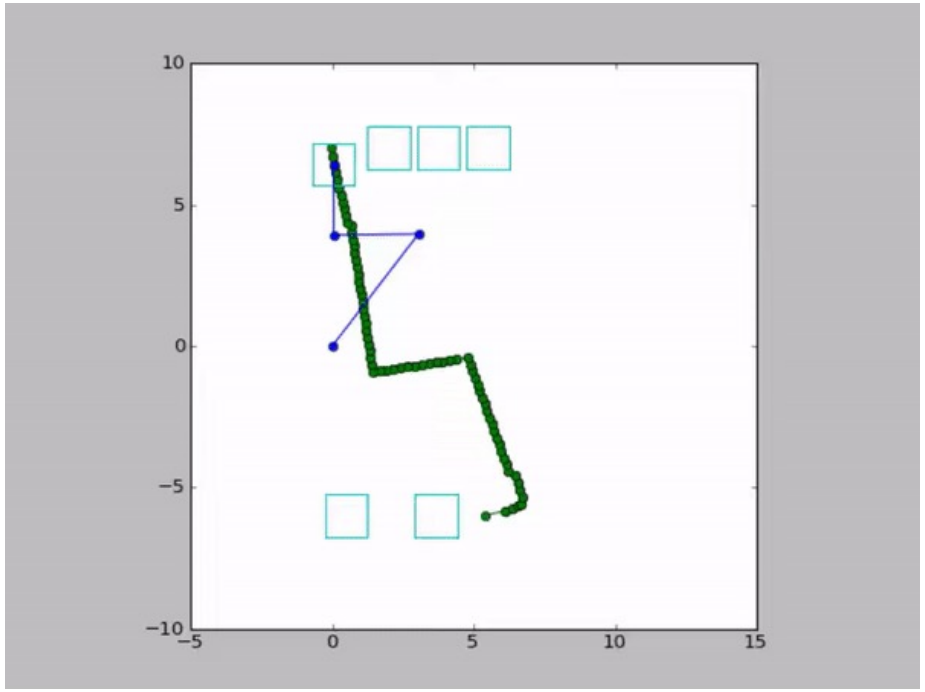
Lazy A*

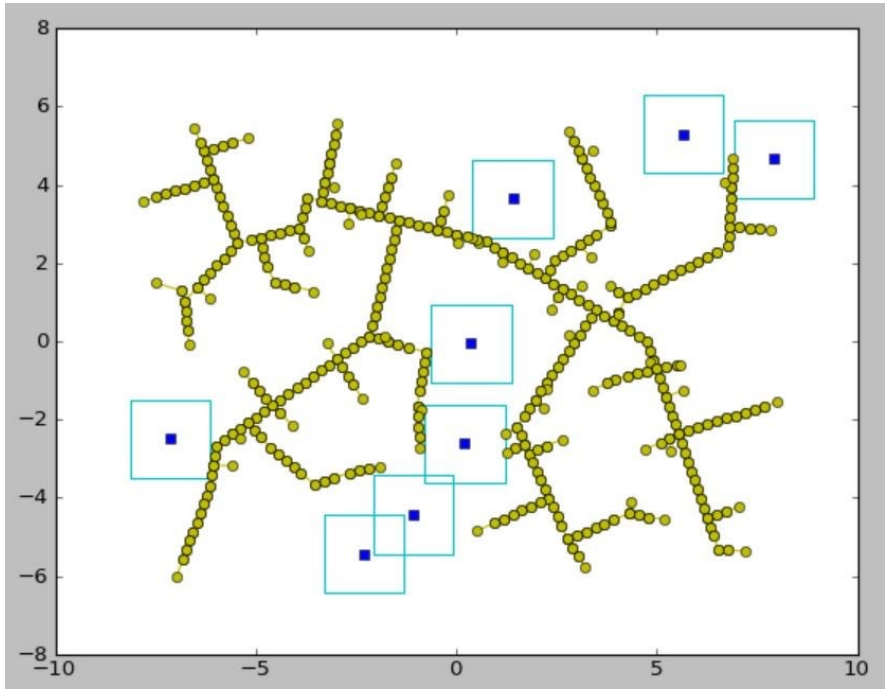
From PRMs to RRT

RRT to RRT*

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*

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$;
    $V \leftarrow V \cup \{x_{\text{new}}\}$;
    $x_{\min} \leftarrow x_{\text{nearest}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Connect along a minimum-cost path
      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**
        $x_{\min} \leftarrow x_{\text{near}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
    $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Rewire the tree
      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** $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
      $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
return $G = (V, E)$;

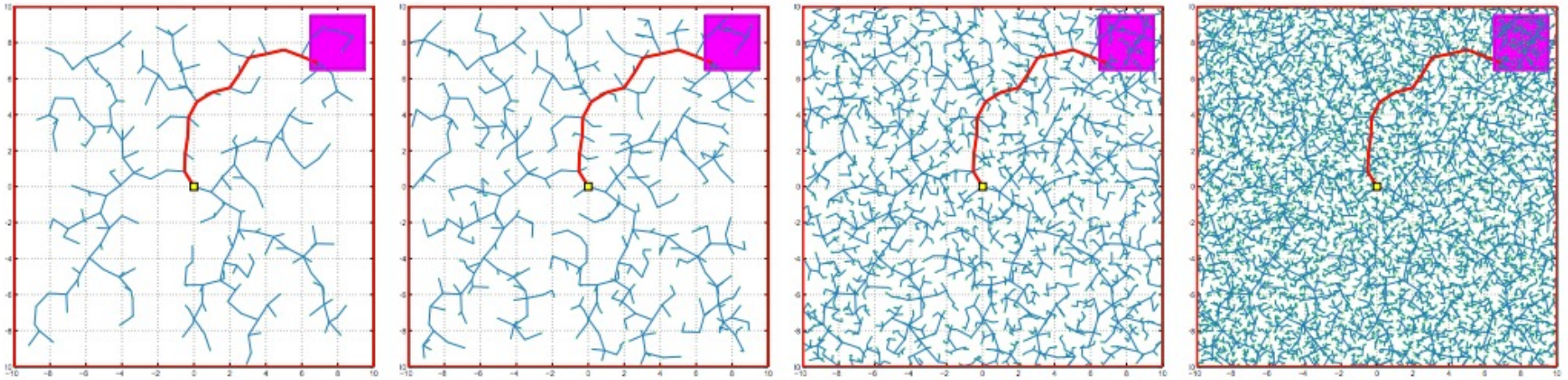
Connect new node to a better parent

Rewire nearby nodes through new node

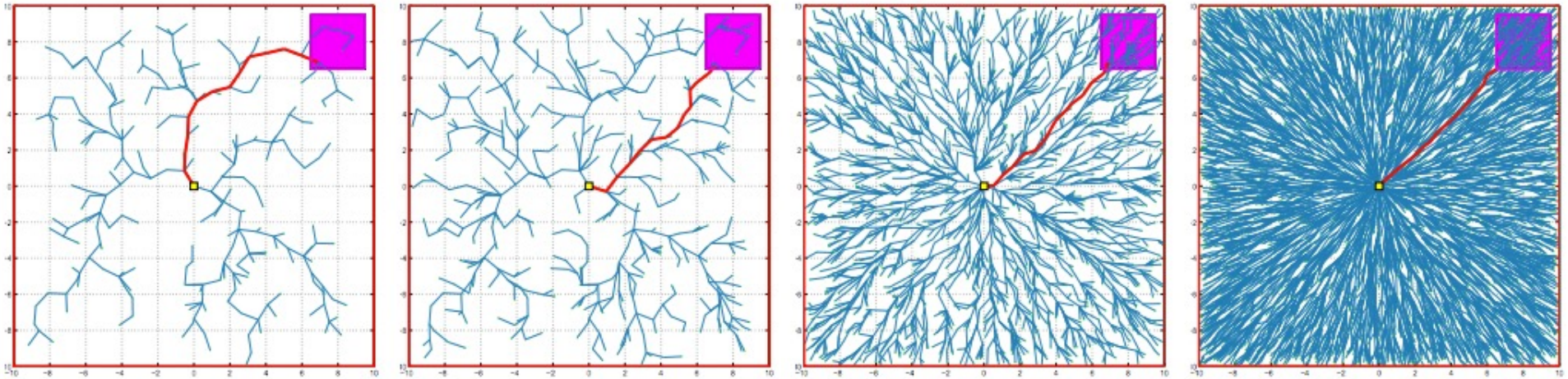
Source: Karaman and Frazzoli

RRT*

RRT



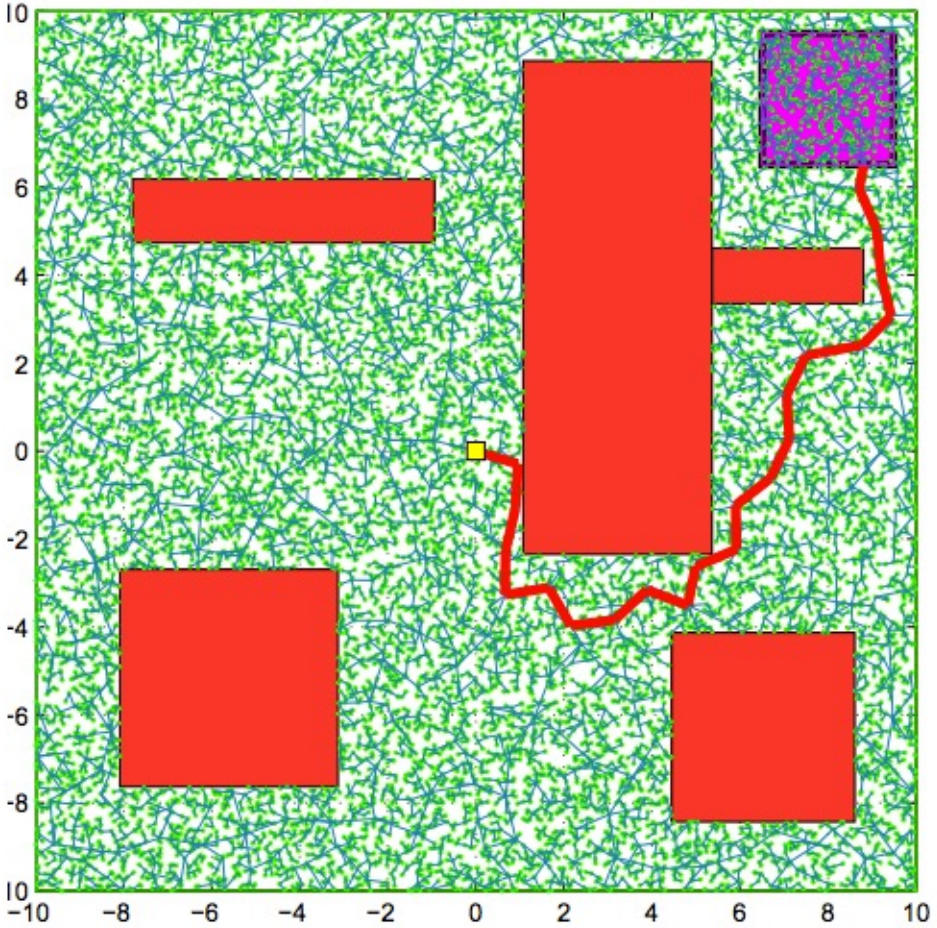
RRT*



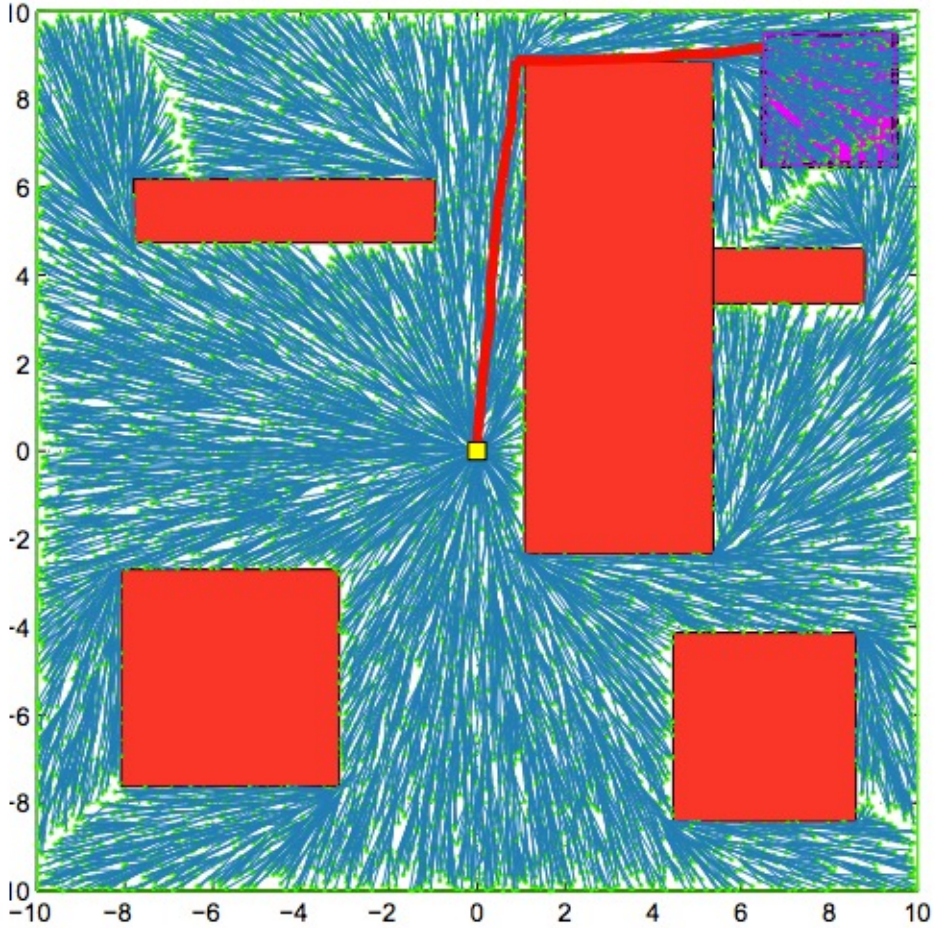
Source: Karaman and Frazzoli

RRT*

RRT

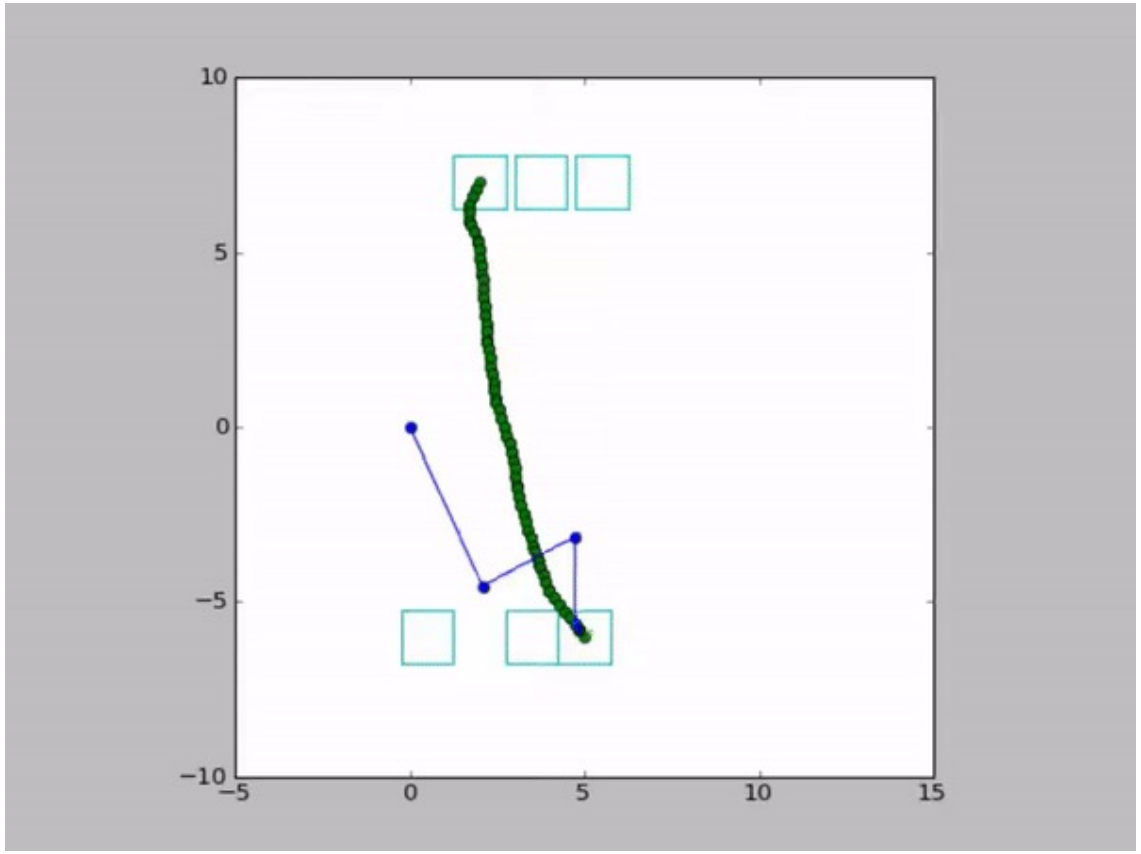


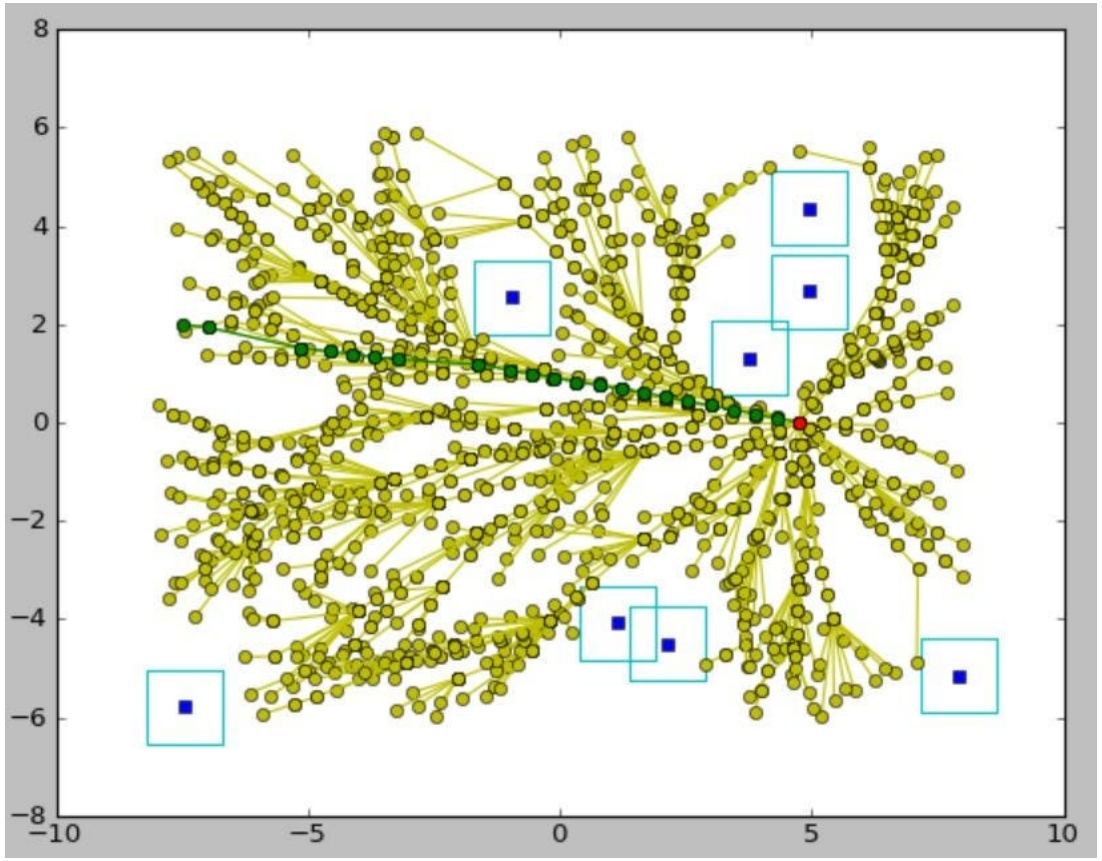
RRT*



Source: Karaman and Frazzoli

RRT*





Source: Tim Chinenov

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* 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

Recap

Lazy A*

From PRMs to RRT

RRT to RRT*

Class Outline