ECE276B: Planning & Learning in Robotics
Lecture 8: Sampling-based Planning

Lecturer:

Nikolay Atanasov: natanasov@ucsd.edu

Teaching Assistants:

Tianyu Wang: tiw161@eng.ucsd.edu

Yongxi Lu: yol070@eng.ucsd.edu

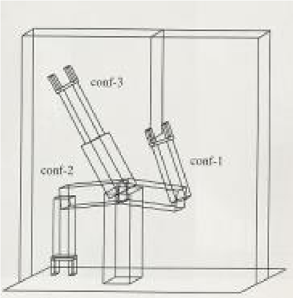
UC San Diego

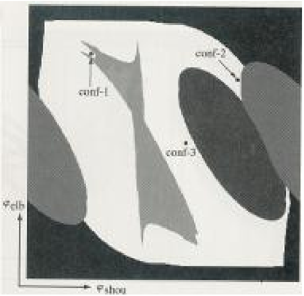
JACOBS SCHOOL OF ENGINEERING

Electrical and Computer Engineering

Search-based vs Sampling-based Planning

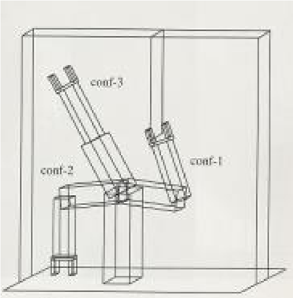
- Search-based planning:
  - Generates a systematic discrete representation (graph) of C_{free}
  - Searches the representation for a path guaranteeing to find one if it exists (resolution complete)
  - Can interleave the representation construction with the search, i.e., adds nodes only when necessary
  - Provides suboptimality bounds on the solution
  - Can get computationally expensive in high dimensions

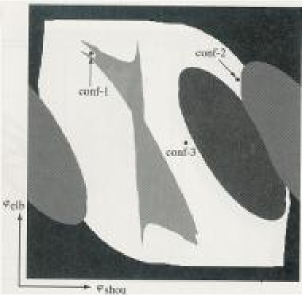


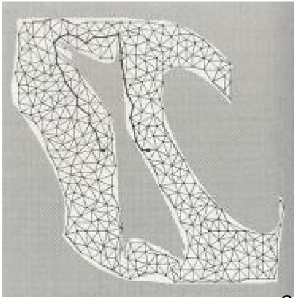


Search-based vs. Sampling-based Planning

- Sampling-based planning:
  - Generates a sparse sample-based representation (graph) of C_{free}
  - Searches the representation for a path guaranteeing that the probability of finding one if it exists approaches 1 as the number of iterations $\to \infty$ (probabilistically complete)
  - Can interleave the representation construction with the search, i.e., adds samples only when necessary
  - Provides asymptotic suboptimality bounds on the solution
  - Well-suited for high-dimensional planning as it is faster and requires less memory than search-based planning in many domains



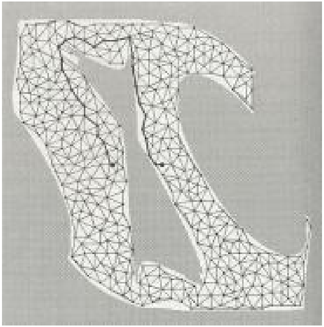




Probabilistic Roadmap (PRM)

Step 1. **Preprocessing Phase**: Build a roadmap (graph) $\mathcal{G}$ which, hopefully, should be accessible from any point in C_{free}

- **Nodes**: randomly sampled valid configurations $x_i \in C_{free}$
- **Edges**: added between samples that are easy to connect with a simple local controller (e.g., follow straight line)



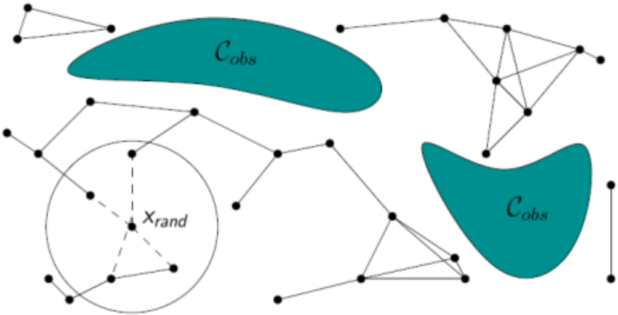
Step 2. **Query Phase**: Given a start configuration x_s and goal configuration x_τ, connect them to the roadmap $\mathcal{G}$ using a local planner, then search the augmented roadmap for a shortest path from x_s to x_τ

- **Pros and Cons**:
  - Simple and highly effective in high dimensions
  - Can result in suboptimal paths, no guarantees on suboptimality
  - Difficulty with narrow passages
  - Useful for multiple queries with different start and goal in the same environment

Step 1: Preprocessing Phase

Algorithm 1 Build Roadmap

$\mathcal{G}$.init()
for $i = 1, \ldots, N$ **do**
    Sample x_{rand}
    if $x_{rand} \in \mathcal{C}_{free}$ **then** ▷ New sample
        $\mathcal{G}$.add_vertex(x_{rand})
        for $x \in$ NEIGHBORHOOD($x_{rand}, \mathcal{G}$) **do** ▷ Region around sample
            if (**not** $\mathcal{G}$.same_component(x_{rand}, x)) **and** CONNECT(x_{rand}, x) **then**
                $\mathcal{G}$.add_edge(x_{rand}, x) ▷ Can be connected by local planner



Step 1: Preprocessing Phase

- Efficient implementation of $x \in \text{NEIGHBORHOOD}(x_{rand}, \mathcal{G})$:
  - select all nodes within a fixed radius from x_{rand}
  - select K nodes closest to x_{rand}
  - select K (often just 1) closest points from each of the components in $\mathcal{G}$
- $\mathcal{G}$.same_component(x_{rand}, x) may be replaced by "$|Children(x)| < K$"
- Sampling strategies:
  - Sample x_{rand} uniformly from C_{free}
  - Select an existing nodes with probability inversely proportional to how well connected it is and generate a random motion from it to get x_{rand}
  - Bias sampling towards obstacle boundaries
  - Bias sampling away from obstacles

PRM vs RRT

- **Rapidly Exploring Random Tree** (RRT):
  - One of the most popular planning techniques
  - Introduced by Steven LaValle in 1998
  - Many, many, many extensions and variants (articulated robots, kinematics, dynamics, differential constraints)

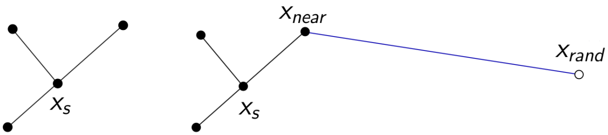
- **PRM**: a graph constructed from random samples. It can be search for a path whenever a start node x_s and goal node x_T are specified. PRMs are well-suited for repeated planning between different pairs of x_s and x_T (*multiple queries*)

- **RRT**: a tree is constructed from random samples with root x_s. The tree is grown until it contains a path to x_T. RRTs are well-suited for single-shot planning between a single pair of x_s and x_T (*single query*)

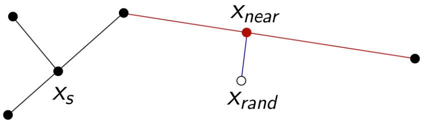
- There exist extensions of RRTs that try to reuse a previously constructed tree when replanning in response to map updates

Rapidly Exploring Random Tree (RRT)

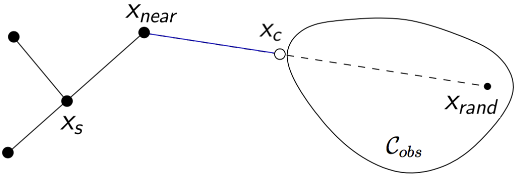
- Sample a new configuration x_{rand}, find the nearest neighbor x_{near} in $\mathcal{G}$ and connect them:



- If the nearest point x_{near} lies on an existing edge, then split the edge:



- If there is an obstacle, the edge travels up to the obstacle boundary, as far as allowed by a collision detection algorithm



Rapidly Exploring Random Tree (RRT)

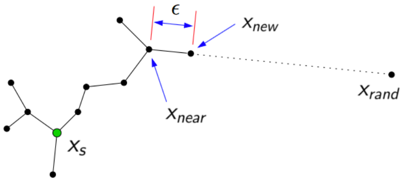
- What about the goal? Occasionally (e.g., every 100 iterations) add the goal configuration x_T and see if it gets connected to the tree
- RRT can be implemented in the original workspace (need to do collision checking) or in configuration space
- Challenges with a C-Space implementation:
  - What distance function do we use to find the nearest configuration?
    - e.g., distance along the surface of a torus for a 2 link manipulator
  - An edge represents a path in C-Space. How do we construct a collision-free path between two configurations?
    - We do not have to connect the configurations all the way. Instead, use a small step size ϵ and a local steering function to get closer to the second configuration.

Rapidly Exploring Random Tree (RRT)

- **No preprocessing**: starting with an initial configuration x_s build a graph (actually, tree) until the goal configuration x_τ is part of it

Algorithm 2 Build_RRT(x_s)

```
T.init(x_s)
for i = 1 ... N do
    Sample x_rand
    EXTEND(T, x_rand)
return T
```

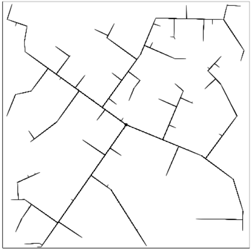


Algorithm 3 Extend($\mathcal{T}, x_{rand}$)

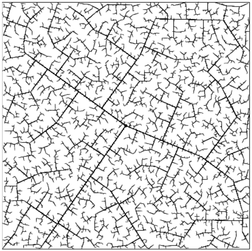
```
x_near ← NEARESTNEIGHBOR(T, x_rand)       ▷ closest node in the tree
x_new ← STEER(x_near, x_rand)             ▷ moves by at most ϵ from x_near towards x
if OBSTACLEFREE(x_near, x_new) then
    T.add_vertex(x_new)
    T.add_edge(x_near, x_new)
    if x_new = x_rand then
        return Reached
    else
        return Advanced
return Trapped
```

Rapidly Exploring Random Tree (RRT)

- RRT without ϵ (called Rapidly Exploring Dense Tree (RDT)):

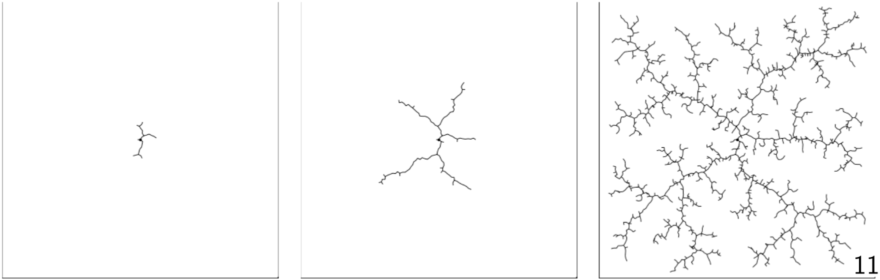


45 iterations



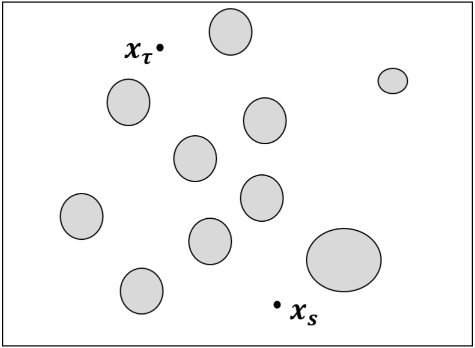
2345 iterations

- RRT with ϵ



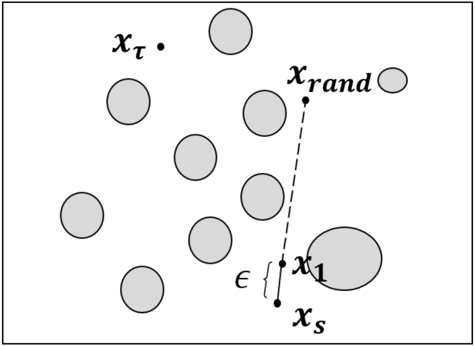
Example: RRT Algorithm

- Start node x_s
- Goal node x_τ
- Gray obstacles



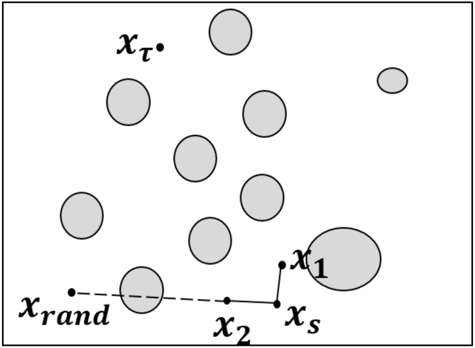
Example: RRT Algorithm

- Sample x_{rand} in the workspace
- Steer from x_s towards x_{rand} by a fixed distance ϵ to get x_1
- If the segment from x_s to x_1 is collision-free, insert x_1 into the tree



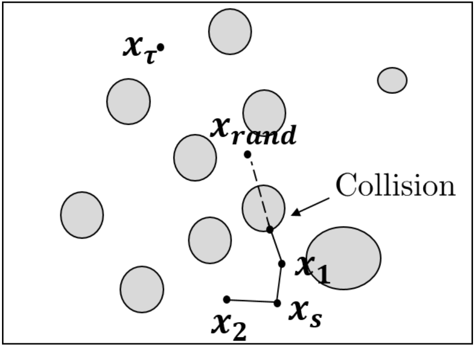
Example: RRT Algorithm

- Sample x_{rand} in the workspace
- Find the closest node x_{near} to x_{rand}
- Steer from x_{near} towards x_{rand} by a fixed distance ϵ to get x_2
- If the segment from x_{near} to x_2 is collision-free, insert x_2 into the tree



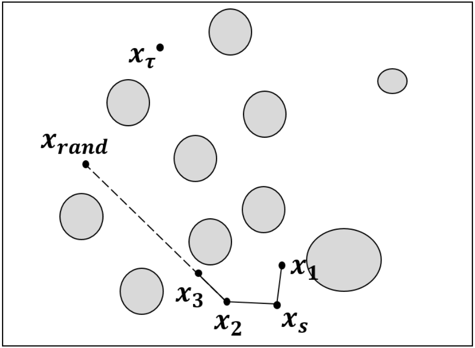
Example: RRT Algorithm

- Sample x_{rand} in the workspace
- Find the closest node x_{near} to x_{rand}
- Steer from x_{near} towards x_{rand} by a fixed distance ϵ to get x_3
- If the segment from x_{near} to x_3 is collision-free, insert x_3 into the tree



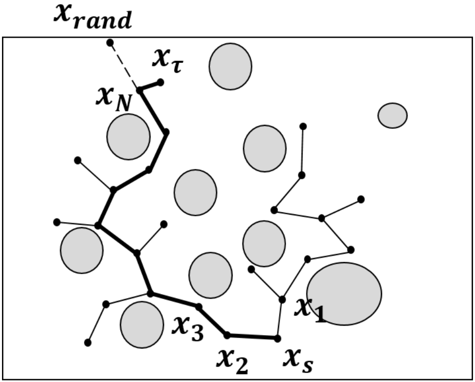
Example: RRT Algorithm

- Sample x_{rand} in the workspace
- Find the closest node x_{near} to x_{rand}
- Steer from x_{near} towards x_{rand} by a fixed distance ϵ to get x_3
- If the segment from x_{near} to x_3 is collision-free, insert x_3 into the tree



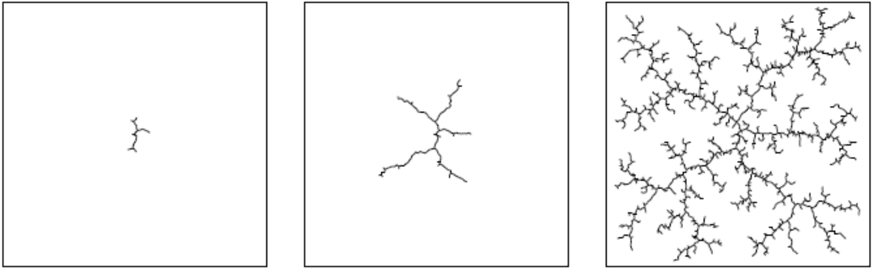
Example: RRT Algorithm

- Continue until a node that is a distance ϵ from the goal is generated
- Either terminate the algorithm or search for additional feasible paths

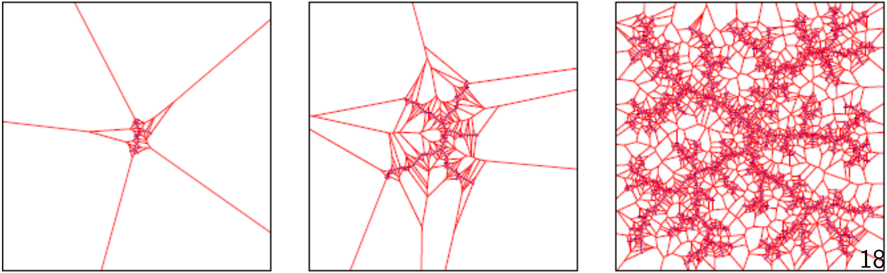


Sampling in RRTs

- The vanilla RRT algorithm provides uniform coverage of space

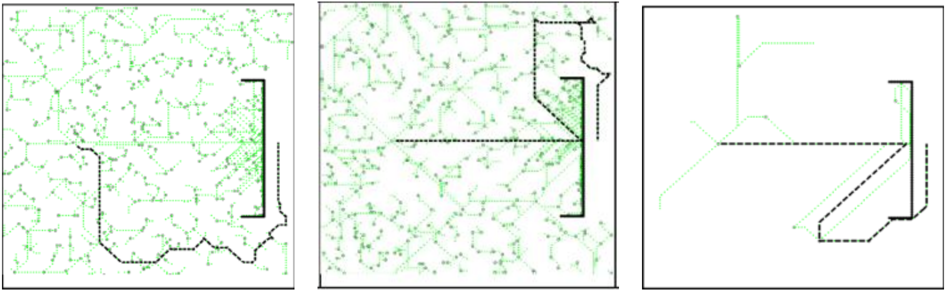


- Alternatively, the growth may be biased by the largest Voronoi region



Sampling in RRTs

- Goal-biased sampling: with probability $(1 - p_g)$, x_{rand} is chosen as a uniform sample in C_{free} and with probability p_g, $x_{rand} = x_T$

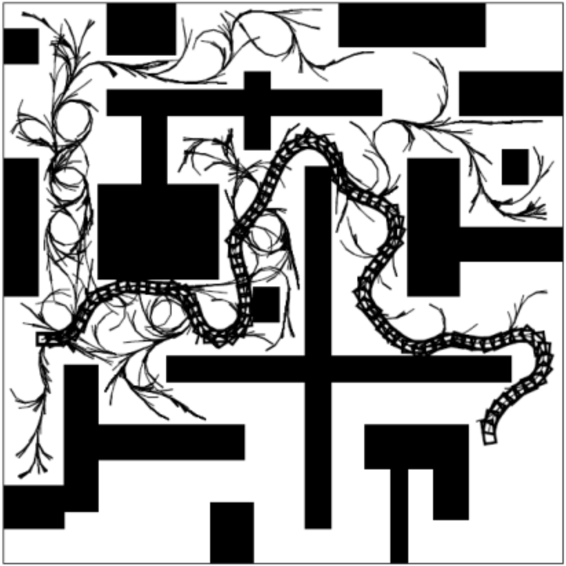


(a) $p_g = 0$ (b) $p_g = 0.1$ (c) $p_g = 0.5$

Handling Robot Dynamics with Steer()

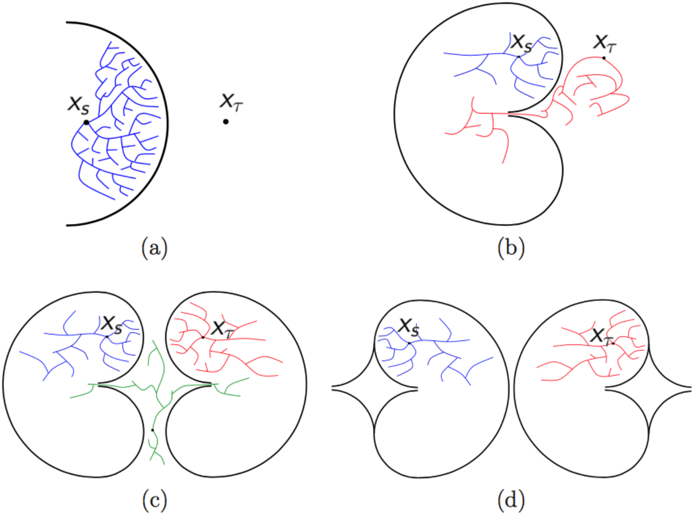
- Steer() extends the tree towards a given random sample x_{rand}
- Consider a car-like robot with non-holonomic constraints (can't slide sideways) in $SE(2)$. Obtaining a feasible path from $x_{rand} = (0, 0, 90^\circ)$ to $x_{near} = (1, 0, 90^\circ)$ is as hard as the original problem
- Steer() resolves this by not requiring the motion to get all the way to x_{rand}. We just apply the best control input for a fixed duration to obtain x_{new} and a dynamically feasible trajectory to it

Example: 5 DOF Kinodynamic Planning for a Car



Bug Traps

- Growing two trees, one from start and one for goal, often has better performance in practice.



Bi-directional RRT

Algorithm 4 BALANCED_BIDIRECTIONAL_RRT(x_s, x_τ)

$\mathcal{T}_a$.init(x_s); $\mathcal{T}_b$.init(x_τ);
for $i = 1 \ldots N$ **do**
    Sample x_{rand}
    $x_{near} \leftarrow \textsc{NearestNeighbor}(\mathcal{T}_a, x_{rand})$
    $x_c \leftarrow \textsc{Steer}(x_{near}, x_{rand})$
    if $x_c \neq x_{near}$ **then**
        $\mathcal{T}_a$.add_vertex(x_c)
        $\mathcal{T}_a$.add_edge(x_{near}, x_c)
        $x'_{near} \leftarrow \textsc{NearestNeighbor}(\mathcal{T}_b, x_c)$
        $x'_c \leftarrow \textsc{Steer}(x'_{near}, x_c)$
        if $x'_c \neq x'_{near}$ **then**
            $\mathcal{T}_b$.add_vertex(x'_c)
            $\mathcal{T}_b$.add_edge(x'_{near}, x'_c)
        if $x'_c = x_c$ **then return** SOLUTION
    if $|\mathcal{T}_b| < |\mathcal{T}_a|$ **then** $\textsc{Swap}(\mathcal{T}_a, \mathcal{T}_b)$
FAILURE

RRT-Connect

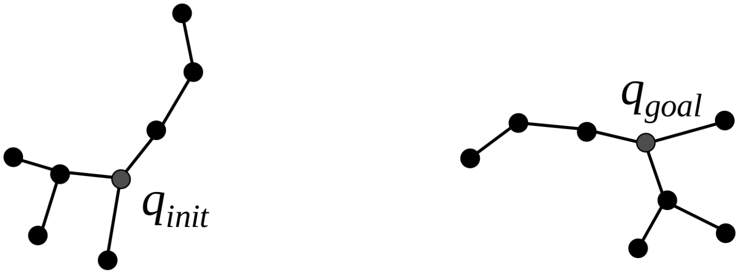
- J. Kuffner and S. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning," ICRA'00
- Bi-directional tree + relax the ϵ constraint on tree growth

Algorithm 5 RRT_CONNECT(x_s, x_τ)

$\mathcal{T}_a$.init(x_s); $\mathcal{T}_b$.init(x_τ);
for $k = 1 \ldots K$ **do**
    Sample x_{rand}
    if not $\textsc{Extend}(\mathcal{T}_a, x_{rand})$ = Trapped **then**
        if $\textsc{Connect}(\mathcal{T}_b, x_{new})$ = Reached **then**   ▷ x_{new} was just added to $\mathcal{T}_a$
            return $\textsc{Path}(\mathcal{T}_a, \mathcal{T}_b)$
    $\textsc{Swap}(\mathcal{T}_a, \mathcal{T}_b)$
return Failure

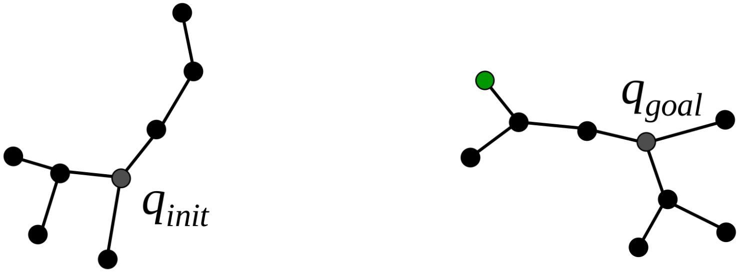
function $\textsc{Connect}(\mathcal{T}, x)$
    repeat
        $S \leftarrow \textsc{Extend}(\mathcal{T}, x)$
    until not (S = Advanced)
    return S

Example: Single RRT-Connect Iteration



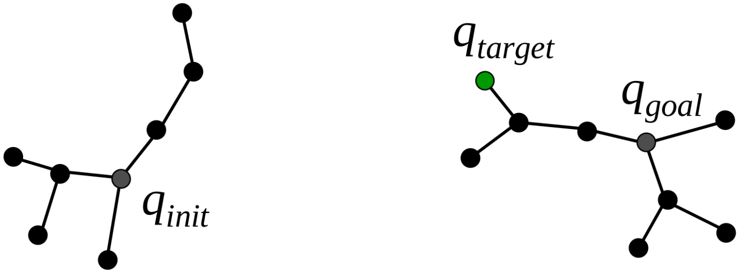
Example: Single RRT-Connect Iteration

- One tree is grown to a random target



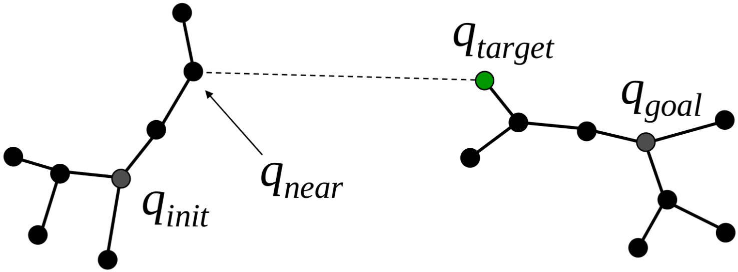
Example: Single RRT-Connect Iteration

- The new node becomes a target for the other tree



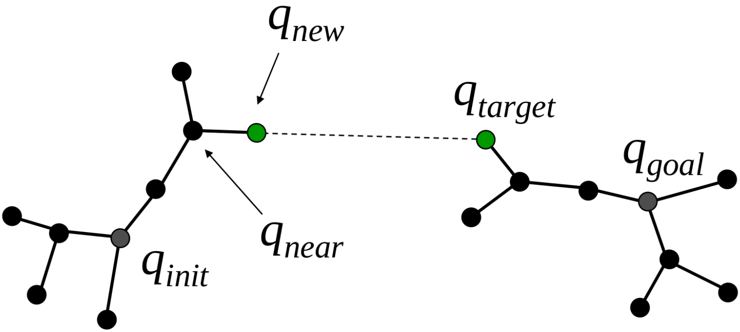
Example: Single RRT-Connect Iteration

- Determine the nearest node to the target



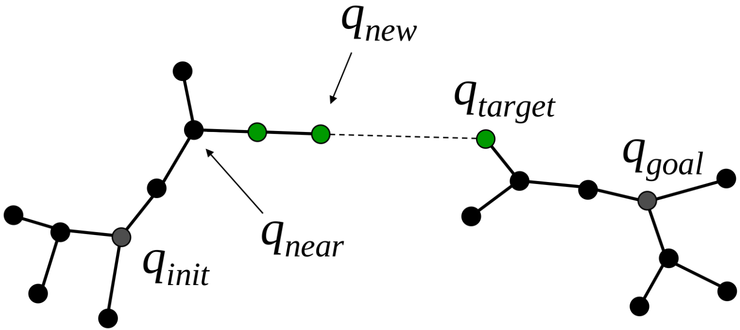
Example: Single RRT-Connect Iteration

- Try to add a new collision-free branch



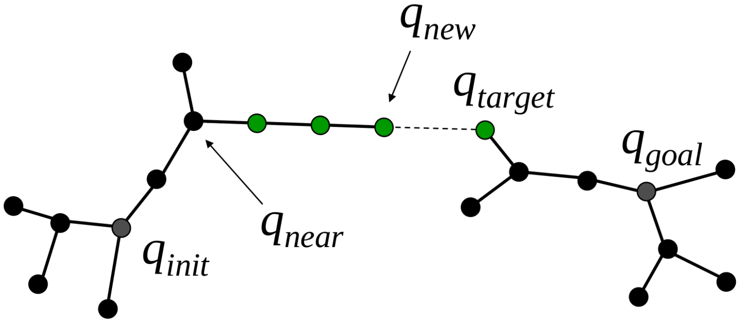
Example: Single RRT-Connect Iteration

- If successful, keep extending the branch



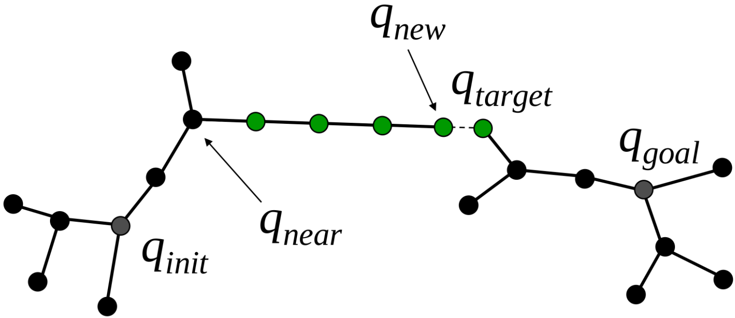
Example: Single RRT-Connect Iteration

- If successful, keep extending the branch



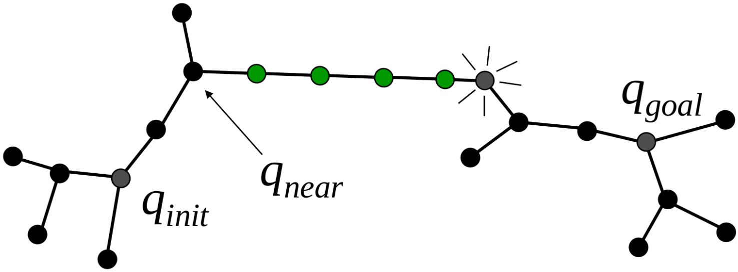
Example: Single RRT-Connect Iteration

- If successful, keep extending the branch



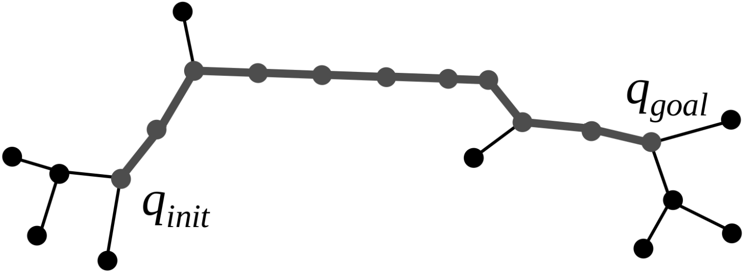
Example: Single RRT-Connect Iteration

- If the branch reaches all the way to the target, a feasible path is found!

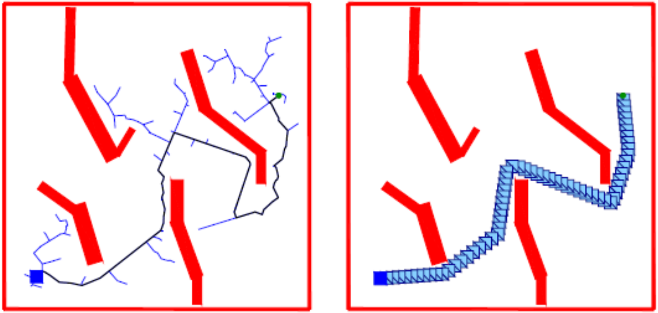


Example: Single RRT-Connect Iteration

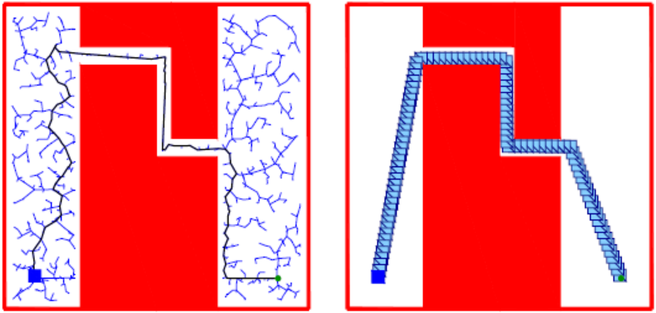
- If the branch reaches all the way to the target, a feasible path is found!



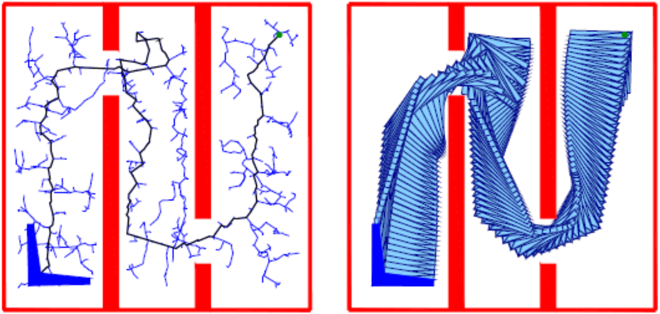
Example: RRT-Connect



Example: RRT-Connect



Example: RRT-Connect

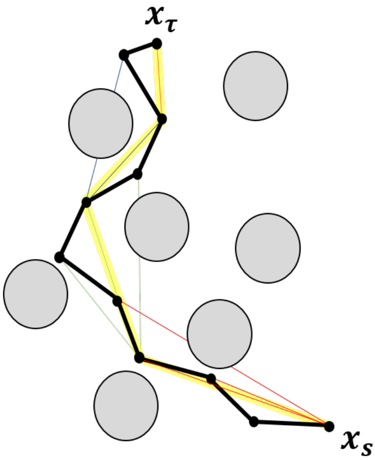


Why are RRTs so popular?

- The algorithm is very simple once the following subroutines are implemented:
  - Random sample generator
  - Nearest neighbor
  - Collision checker
  - Steer
- Pros:
  - Sparse exploration requires little memory and computation
  - RRTs find feasible paths quickly in practice
  - Can add heuristics on top, e.g., bias the sampling towards the goal
- Cons:
  - Solutions can be highly sub-optimal and require path smoothing as a post-processing step
  - The smoothed path is still restricted to the same homotopy class

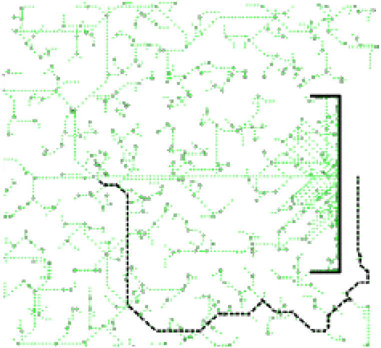
Path Smoothing

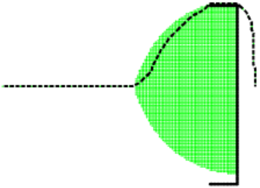
- Start with the initial point (1)
- Make connections to subsequent points in the path (2), (3), (4), ...
- When a connection collides with obstacles, add the previous waypoint to the smoothed path
- Continue smoothing from this point on



Search-based vs Sampling-based Planning

- RRT:
  - Sparse exploration requires little memory and computation
  - Solutions can be highly sub-optimal and require post-processing (path smoothing) which may be difficult
- Weighted A*:
  - Systematic exploration may require a lot of memory and computation
  - Returns a path with (sub-)optimality guarantees





RRT Guarantees

- RRT and RRT-Connect are **probabilistically complete**: the probability that a feasible path will be found if one exists, approaches 1 exponentially as the number of samples approaches infinity
- Assuming C_{free} is connected, bounded, and open, for any $x \in \mathcal{C}_{free}$, $\lim_{N\to\infty} \mathbb{P}(\|x - x_{near}\| < \epsilon) = 1$, where x_{near} is the closest node to x in $\mathcal{T}$
- RRT is **not optimal**: the probability that RRT converges to an optimal solution, as the number of samples approaches infinity, is zero under reasonable technical assumptions (S. Karaman, E. Frazzoli, RSS'10)
- **Problem**: once we build an RRT we never modify it

RRT*

- S. Karaman, E. Frazzoli, "Incremental Sampling-based Algorithms for Optimal Motion Planning," RSS'10
- RRT*: RRT + rewiring of the tree to ensure asymptotic optimality

Algorithm 1: Body of RRT and RRG Algorithms

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$; $i \leftarrow 0$;
while $i < N$ **do**
    $G \leftarrow (V, E)$;
    $x_{\text{rand}} \leftarrow \texttt{Sample}(i)$; $i \leftarrow i + 1$;
    $(V, E) \leftarrow \texttt{Extend}(G, x_{\text{rand}})$;

Algorithm 2: $\texttt{Extend}_{RRT}$

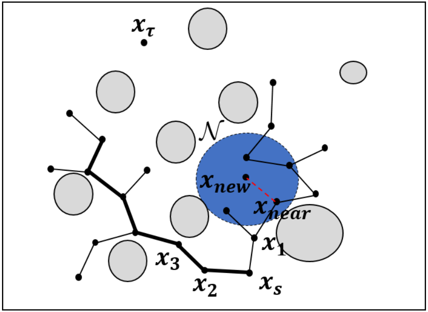
$V' \leftarrow V$; $E' \leftarrow E$;
$x_{\text{nearest}} \leftarrow \texttt{Nearest}(G, x)$;
$x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x)$;
if $\texttt{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V' \leftarrow V' \cup \{x_{\text{new}}\}$;
    $E' \leftarrow E' \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
return $G' = (V', E')$

Algorithm 4: $\texttt{Extend}_{RRT^*}$

$V' \leftarrow V$; $E' \leftarrow E$;
$x_{\text{nearest}} \leftarrow \texttt{Nearest}(G, x)$;
$x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x)$;
if $\texttt{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V' \leftarrow V' \cup \{x_{\text{new}}\}$;
    $x_{\text{min}} \leftarrow x_{\text{nearest}}$;
    $X_{\text{near}} \leftarrow \texttt{Near}(G, x_{\text{new}}, |V|)$;
    for *all* $x_{\text{near}} \in X_{\text{near}}$ **do**
        if $\texttt{ObstacleFree}(x_{\text{near}}, x_{\text{new}})$ **then**
            $c' \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$;
            if $c' < \texttt{Cost}(x_{\text{new}})$ **then**
                $x_{\text{min}} \leftarrow x_{\text{near}}$;
    $E' \leftarrow E' \cup \{(x_{\text{min}}, x_{\text{new}})\}$;
    for *all* $x_{near} \in X_{\text{near}} \setminus \{x_{\text{min}}\}$ **do**
        if $\texttt{ObstacleFree}(x_{\text{new}}, x_{\text{near}})$ *and* $\texttt{Cost}(x_{\text{near}}) >$ $\texttt{Cost}(x_{\text{new}}) + c(\texttt{Line}(x_{\text{new}}, x_{\text{near}}))$ **then**
            $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
            $E' \leftarrow E' \setminus \{(x_{\text{parent}}, x_{\text{near}})\}$;
            $E' \leftarrow E' \cup \{(x_{\text{new}}, x_{\text{near}})\}$;
return $G' = (V', E')$

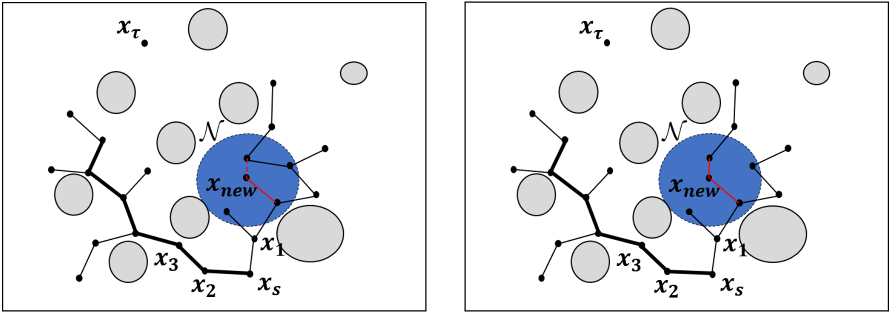
RRT*: Extend Step

- Generate a new potential node x_{new} identically to RRT
- Instead of finding the closest node in the tree, find all nodes within a neighborhood $\mathcal{N}$
- Let $x_{nearest} = \arg\min_{x_{near} \in \mathcal{N}} g_{x_{near}} + c_{x_{near}, x_{new}}$, i.e., the node in $\mathcal{N}$ that lies on the currently known shortest path from x_s to x_{new}
- Add node: $\mathcal{V} \leftarrow \mathcal{V} \cup \{x_{new}\}$
- Add edge: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(x_{nearest}, x_{new})\}$
- Set the label of x_{new} to $g_{x_{new}} = g_{x_{nearest}} + c_{x_{nearest}, x_{new}}$

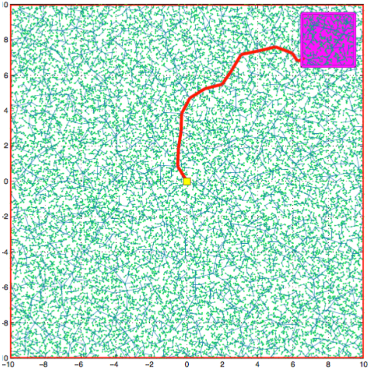


RRT*: Rewire Step

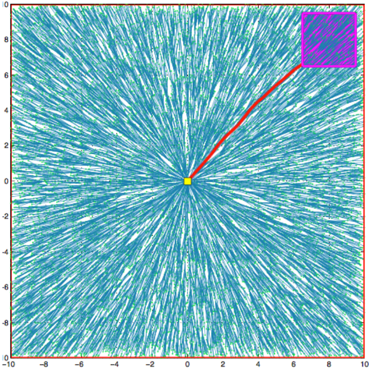
- Check all nodes $x_{near} \in \mathcal{N}$ to see if re-routing through x_{new} reduces the path length (**label correcting!**):
- If $g_{x_{new}} + c_{x_{new},x_{near}} < g_{x_{near}}$, then remove the edge between x_{near} and its parent and add a new edge between x_{near} and x_{new}



RRT vs RRT*



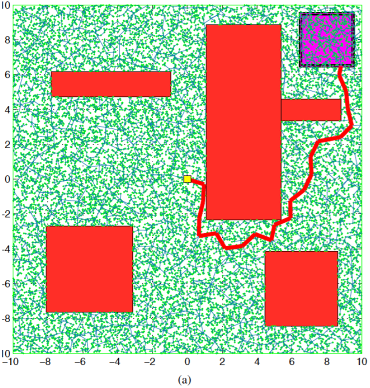
(a) RRT



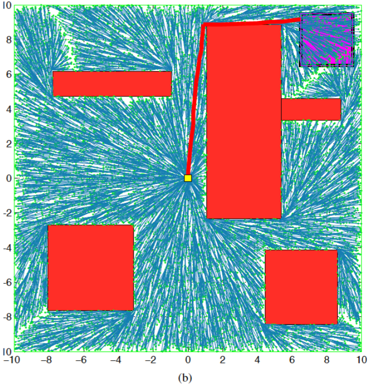
(b) RRT*

- Same nodes in the tree, only the edge connections are different. Notice how the RRT* edges are almost straight lines (optimal paths).
- S. Karaman and E. Frazzoli, "Incremental Sampling-based Algorithms for Optimal Motion Planning," International Journal of Robotics Research, 2010.

RRT vs RRT*



(a) RRT



(b) RRT*

- Same nodes in the tree, only the edge connections are different. Notice how the RRT* edges are almost straight lines (optimal paths).
- S. Karaman and E. Frazzoli, "Incremental Sampling-based Algorithms for Optimal Motion Planning," International Journal of Robotics Research, 2010.