Sampling-based Path Finding

10. 05. 2021

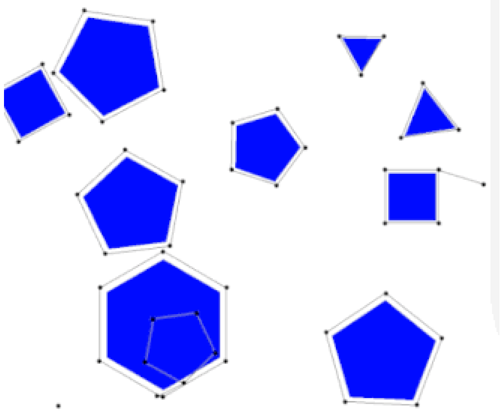
Hongkai Ye

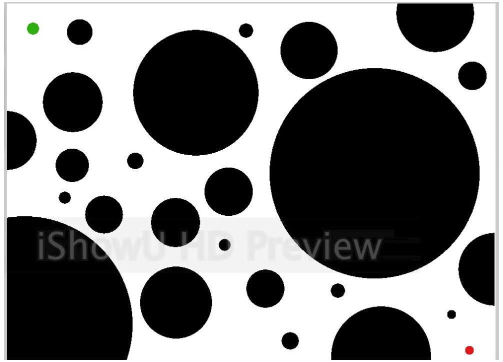
浙江大學

Preliminaries

Sampling-based Path Planners

- Explore the connectivity of the environment efficiently;
- Do not attempt to explicitly construct the C-Space and its boundaries;
- Probabilistic completeness;
- Suboptimal or asymptotically optimal;
- Different approaches for sampling incrementally and in batch.





Brief History

- PRM 1996 TRA
- **RRT 1998**
- RRT-connect 2000 ICRA
- Visibility PRM 2000
- PRM* 2011 IJRR
- RRG 2011 IJRR
- **RRT* 2011 IJRR**
- RRT*-Smart 2012 ICMA
- **RRT# 2013 ICRA**
- FMT* 2013 ISRR
- **Informed-RRT* 2014 IROS**
- RRTx 2014 WAFR 2015 IJRR
- BIT* 2015 ICRA
- SST* 2016 IJRR
- RABIT* 2016 ICRA
- DRRT 2017 RSS
- AIT* 2020 ICRA
- **GuILD 2021 Arxiv**
- ...

Two Fundamental Tasks

- **Exploration**

Acquires information about the topology of the search space, i.e., how subsets of the space are connected.

- **Exploitation**

Incrementally improves the solution by processing the available information computed by the exploration task.

Content

浙江大学

1、Probabilistic Road Map (PRM)

2、Rapidly-exploring Random Tree (RRT)

3、Optimal sampling-based path planning methods (RRT*)

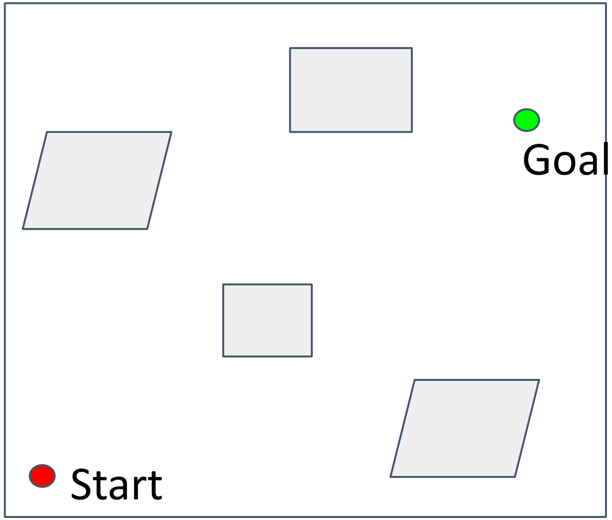
4、Accelerate convergence: RRT#, Informed-RRT*, and GuILD

5、Kinodynamic RRT*

Probabilistic Road Map

What is PRM?

- A graph structure
- Divide planning into two phases:
  - Learning phase:
  - Query phase:
- Checking sampled configurations and connections between samples for collision can be done efficiently.
- A relatively small number of milestones and local paths are sufficient to capture the connectivity of the free space

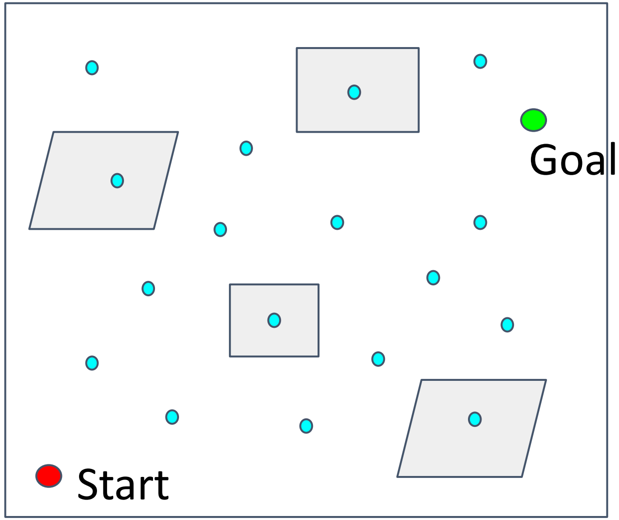


Probabilistic Road Map

Learning phase:

Detect the C-space with random points and construct a graph that represents the connectivity of the environment

- Sample *N* points in C-space
- Delete points that are not collision-free

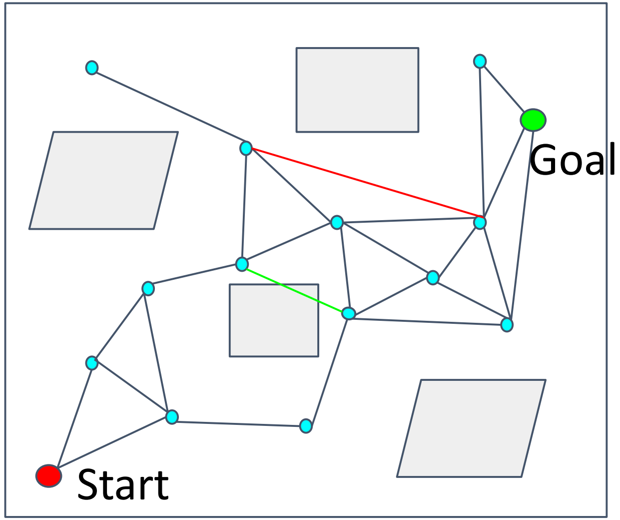


Probabilistic Road Map

Learning phase:

Detect the C-space with random points and construct a graph that represents the connectivity of the environment

- Connect to nearest points and get collision-free segments.
- Delete segments that are not collision free.

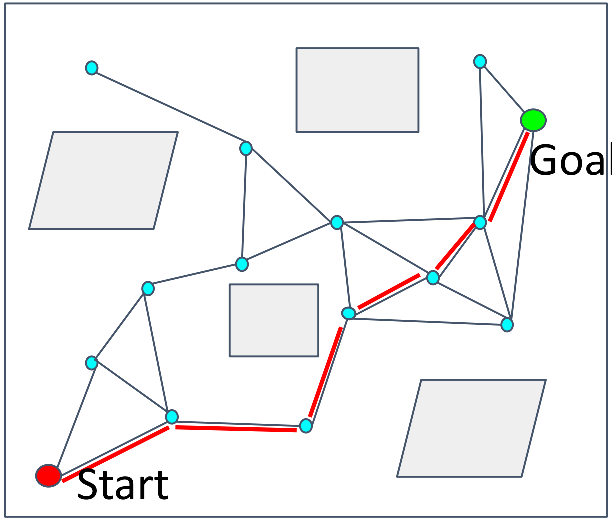


Probabilistic Road Map

Query phase:

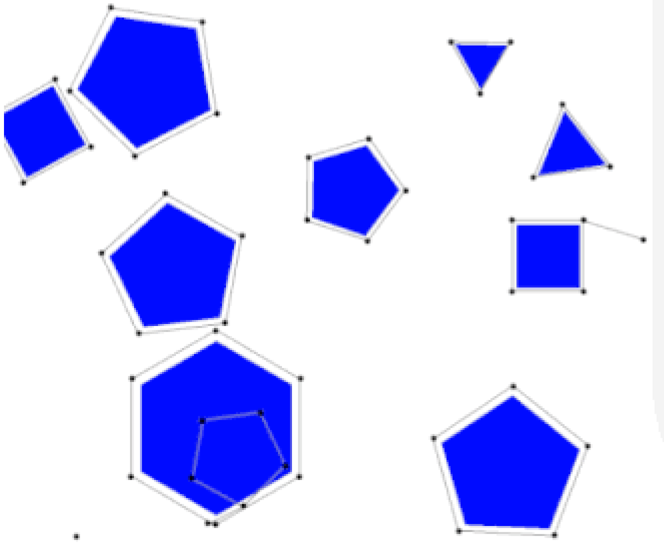
Search on the road map to find a path from the start to the goal (using Dijkstra's algorithm or the A* algorithm).

- Road map is now similar with the grid map (or a simplified grid map).
- Can conduct multiple queries.

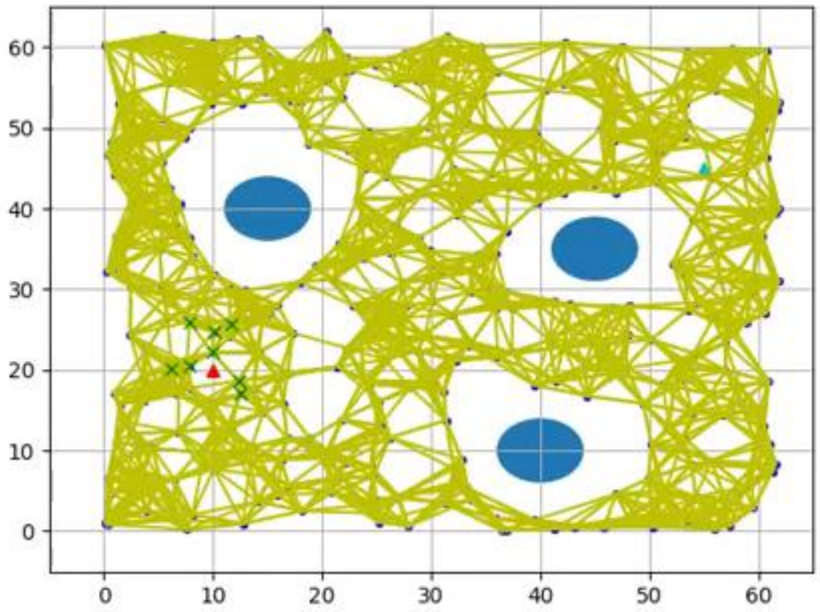


Probabilistic Road Map

Learning phase



Query phase



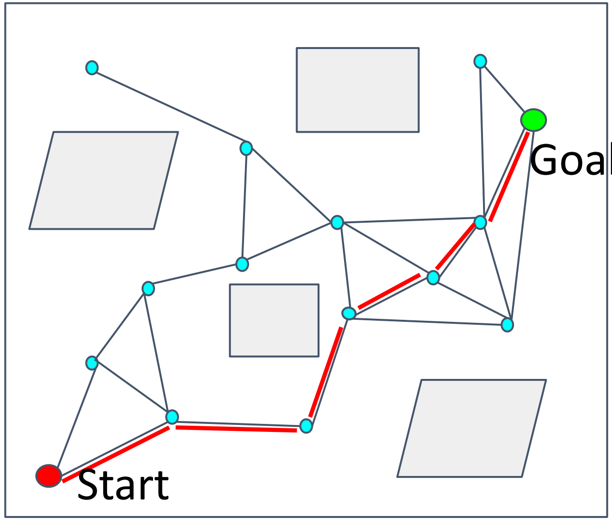
Probabilistic Road Map

Pros

- Probabilistically complete

Cons

- Build graph over state space but no particular focus on generating a path.
- Not efficient



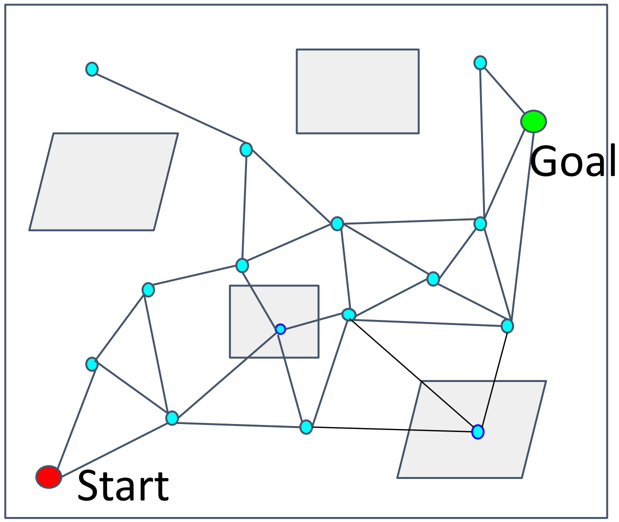
Probabilistic Road Map

Improving efficiency

Collision-checking process is time-consuming, especially in complex or high-dimensional environments.

Lazy collision-checking:

Check collisions only if necessary

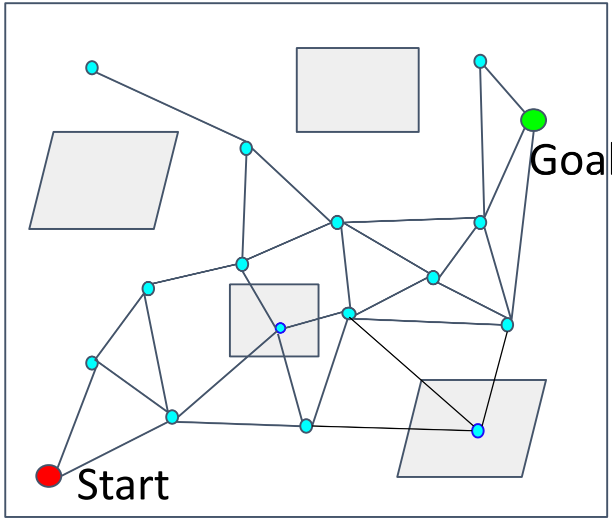


Probabilistic Road Map

Improving efficiency

Lazy collision-checking

- Sample points and generate segments without considering the collision (Lazy).

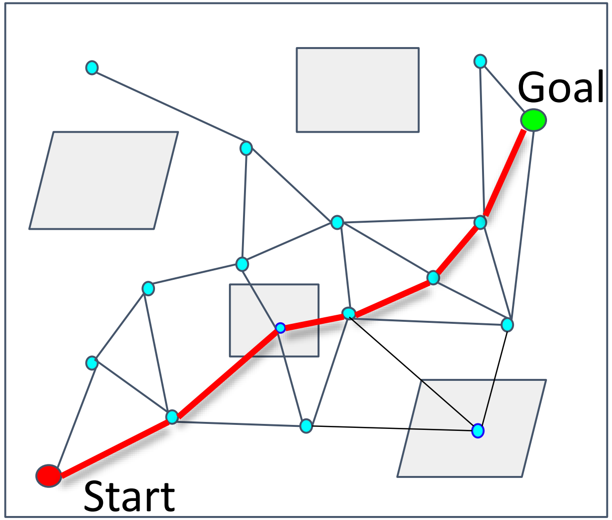


Probabilistic Road Map

Improving efficiency

Lazy collision-checking

- Sample points and generate segments without considering the collision (Lazy).
- Find a path on the road map generated without collision-checking.

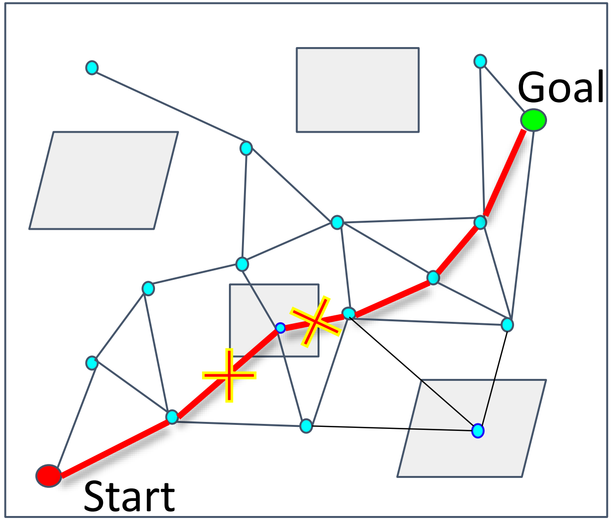


Probabilistic Road Map

Improving efficiency

Lazy collision-checking

- Sample points and generate segments without considering the collision (Lazy).
- Find a path on the road map generated without collision-checking.
- Delete the corresponding edges and nodes if the path is not collision free.

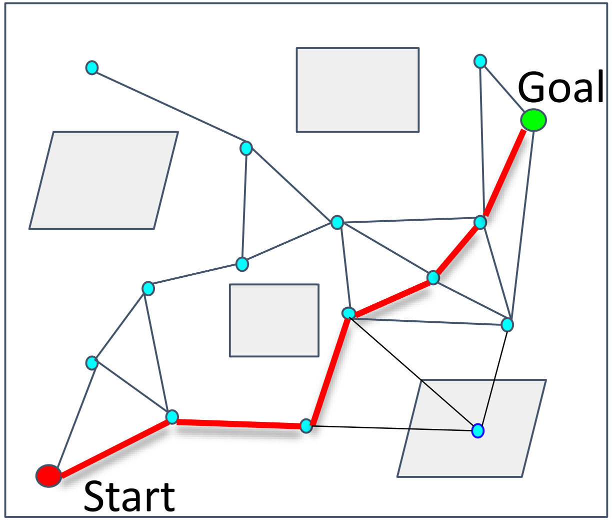


Probabilistic Road Map

Improving efficiency

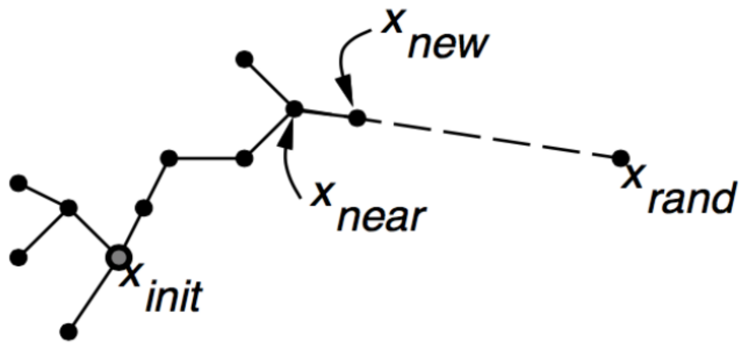
Lazy collision-checking

- Sample points and generate segments without considering the collision (Lazy).
- Find a path on the road map generated without collision-checking.
- Delete the corresponding edges and nodes if the path is not collision free.
- Restart path finding.



Rapidly-exploring Random Trees

Build up a tree from start to goal through generating "next states" in the tree by executing random controls

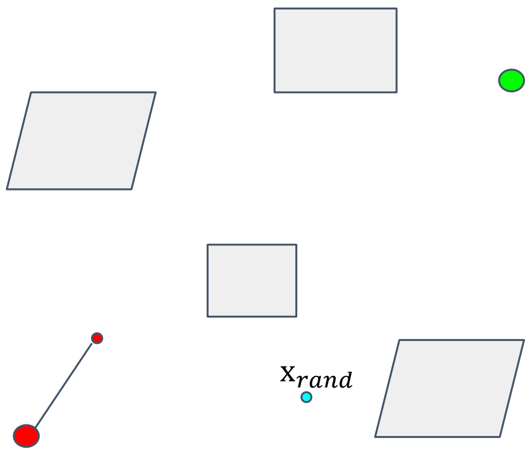


Rapidly-exploring Random Trees

浙江大学

Algorithm 1: RRT Algorithm

Input: $\mathcal{M}$,x_{init},x_{goal}
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
  $x_{rand} \leftarrow Sample(\mathcal{M})$;
  $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
  $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
  $E_i \leftarrow Edge(x_{new}, x_{near})$;
  if *CollisionFree*($\mathcal{M}$, E_i) **then**
    $\mathcal{T}.addNode(x_{new})$;
    $\mathcal{T}.addEdge(E_i)$;
  if $x_{new} = x_{goal}$ **then**
    Success();

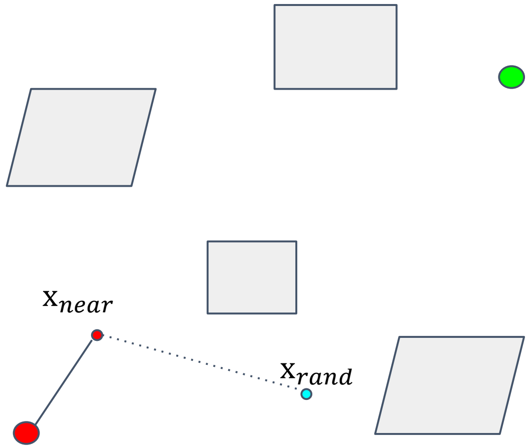


Sample a node Xrand in the free space

Rapidly-exploring Random Trees

Algorithm 1: RRT Algorithm

Input: $\mathcal{M}$,x_{init},x_{goal}
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
  $x_{rand} \leftarrow Sample(\mathcal{M})$;
  $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
  $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
  $E_i \leftarrow Edge(x_{new}, x_{near})$;
  if *CollisionFree*($\mathcal{M}$, E_i) **then**
    $\mathcal{T}.addNode(x_{new})$;
    $\mathcal{T}.addEdge(E_i)$;
  if $x_{new} = x_{goal}$ **then**
    Success();

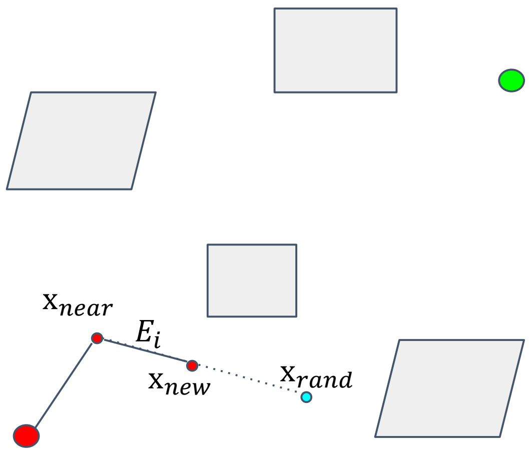


Find the nearest node Xnear in current tree

Rapidly-exploring Random Trees

Algorithm 1: RRT Algorithm

Input: $\mathcal{M}$,x_{init},x_{goal}
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
  $x_{rand} \leftarrow Sample(\mathcal{M})$;
  $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
  $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
  $E_i \leftarrow Edge(x_{new}, x_{near})$;
  if *CollisionFree*($\mathcal{M}$, E_i) **then**
    $\mathcal{T}.addNode(x_{new})$;
    $\mathcal{T}.addEdge(E_i)$;
  if $x_{new} = x_{goal}$ **then**
    Success();

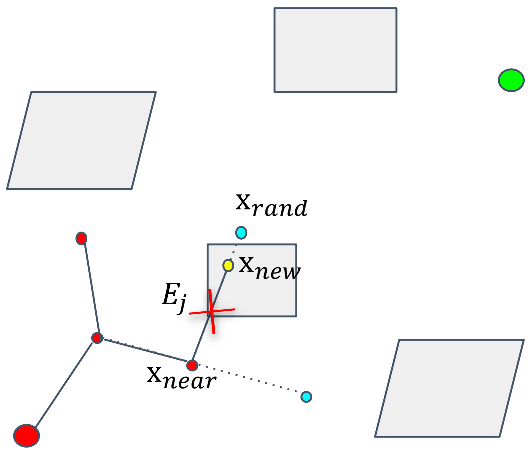


Grow a new node Xnew and path Ei from Xnear

Rapidly-exploring Random Trees

Algorithm 1: RRT Algorithm

Input: $\mathcal{M}$,x_{init},x_{goal}
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
  $x_{rand} \leftarrow Sample(\mathcal{M})$;
  $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
  $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
  $E_i \leftarrow Edge(x_{new}, x_{near})$;
  if *CollisionFree*($\mathcal{M}$, E_i) **then**
    $\mathcal{T}.addNode(x_{new})$;
    $\mathcal{T}.addEdge(E_i)$;
  if $x_{new} = x_{goal}$ **then**
    Success();

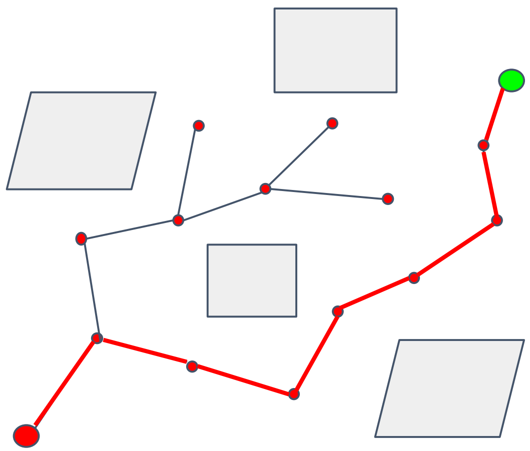


Do not grow if collision

Rapidly-exploring Random Trees

Algorithm 1: RRT Algorithm

Input: $\mathcal{M}$,x_{init},x_{goal}
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
  $x_{rand} \leftarrow Sample(\mathcal{M})$;
  $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
  $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
  $E_i \leftarrow Edge(x_{new}, x_{near})$;
  if *CollisionFree*($\mathcal{M}$, E_i) **then**
    $\mathcal{T}.addNode(x_{new})$;
    $\mathcal{T}.addEdge(E_i)$;
  if $x_{new} = x_{goal}$ **then**
    Success();



Repeat sampling for n times until the tree reaches the goal or goal region

Rapidly-exploring Random Trees

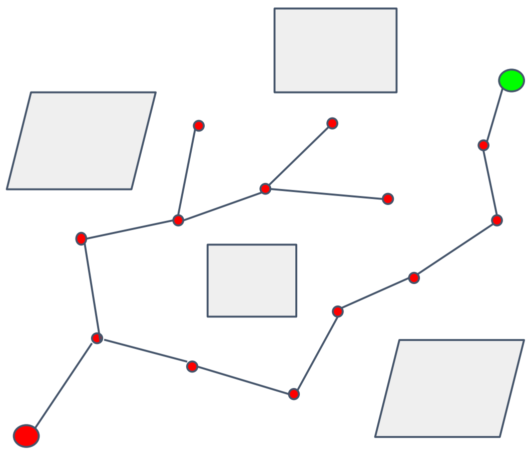
Rapidly-exploring Random Trees

Pros:

- Aims to find a path from the start to the goal
- More target-oriented than PRM

Cons:

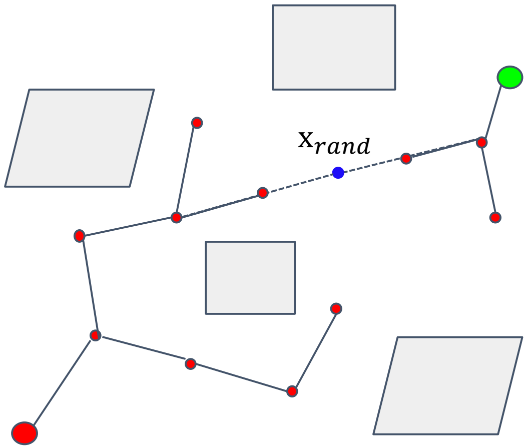
- Not optimal solution
- Not efficient (leave room for improvement)
- Sample in the whole space



Rapidly-exploring Random Trees

Improving efficiency

Bidirectional RRT / RRT Connect

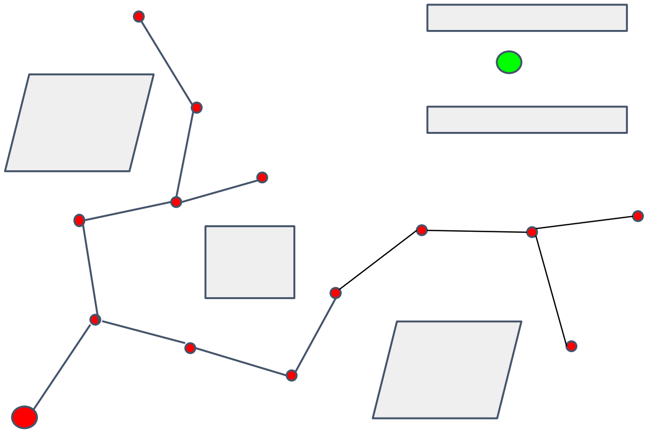


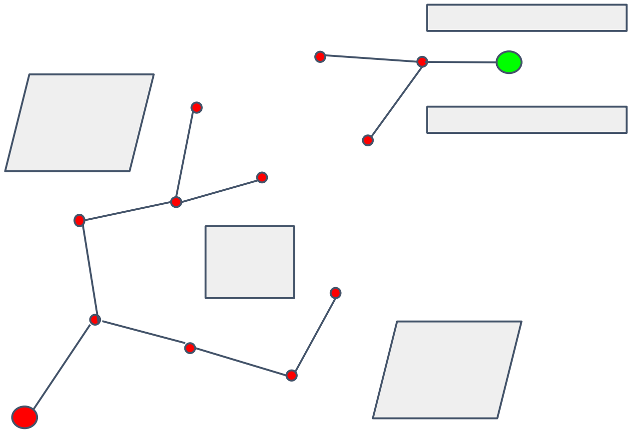
- Grow a tree from both the start point and the goal point.
- Path found when two trees are connected.

Rapidly-exploring Random Trees

Improving efficiency

Bidirectional RRT / RRT Connect

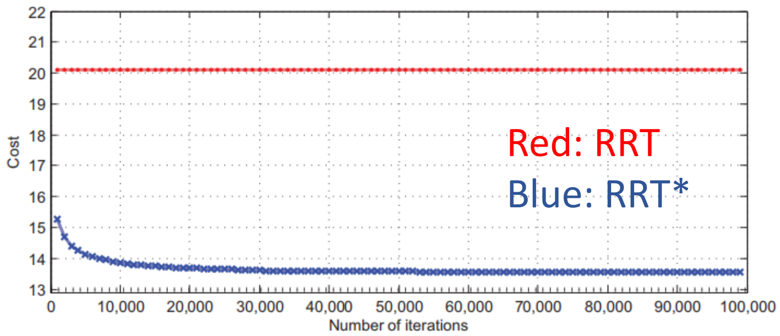


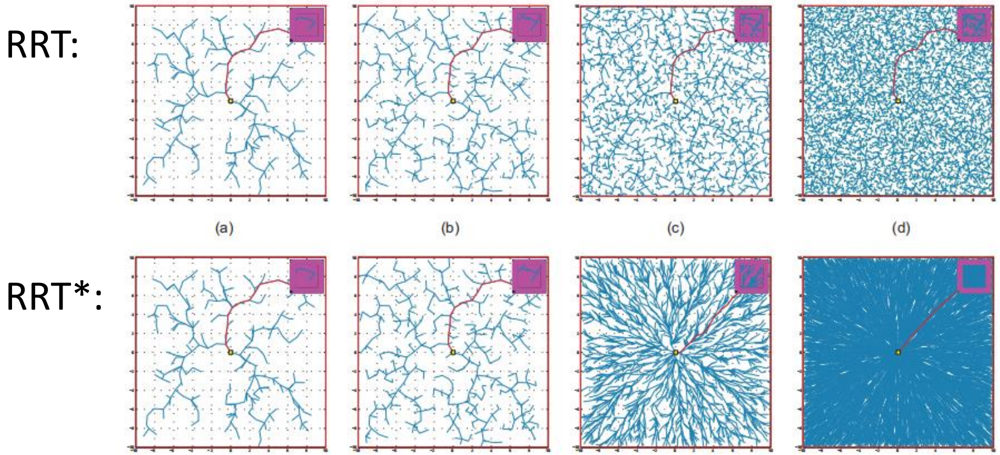


Optimal sampling-based path planning methods

Rapidly-exploring Random Tree* (RRT*)

- An improvement of RRT
- Probabilistic Complete
- Asymptotically optimal

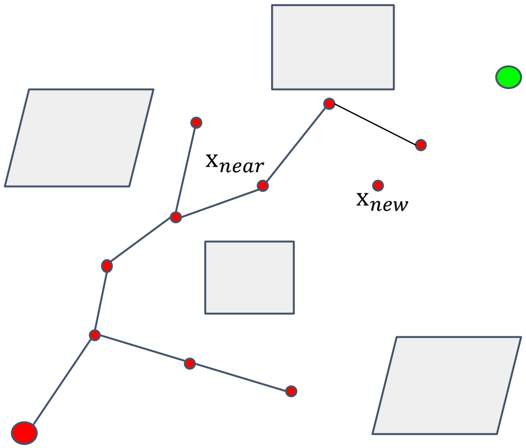




(a) (b) (c) (d)

(e) (f) (g) (h)

Rapidly-exploring Random Tree* (RRT*)



Algorithm 2: RRT Algorithm

Input: $\mathcal{M}$,x_{init},x_{goal}

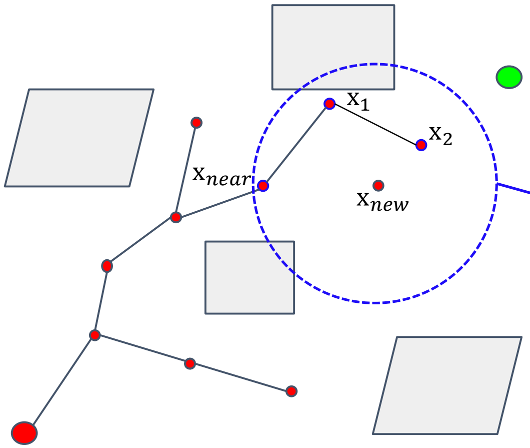
Result: A path Γ from x_{init} to x_{goal}

$\mathcal{T}$.init();

for $i = 1$ *to* n **do**

- $x_{rand} \leftarrow Sample(\mathcal{M})$;
- $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
- $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
- **if** *CollisionFree*(x_{new}) **then**
  - $X_{near} \leftarrow NearC(\mathcal{T}, x_{new})$;
  - $x_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{new})$;
  - $\mathcal{T}.addNodEdge(x_{min}, x_{new})$;
  - $\mathcal{T}.rewire()$;

Rapidly-exploring Random Tree* (RRT*)

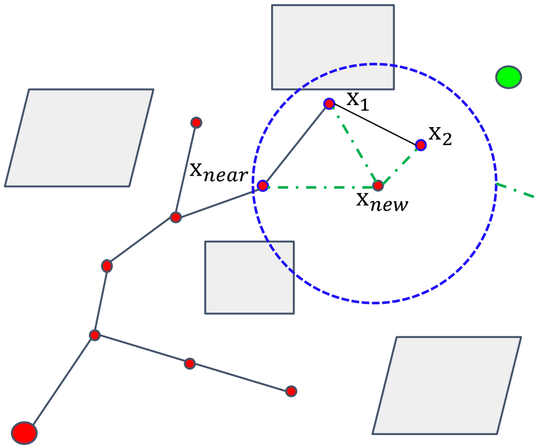


Algorithm 2: RRT Algorithm

Input: $\mathcal{M}$,x_{init},x_{goal}
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
    $x_{rand} \leftarrow Sample(\mathcal{M})$;
    $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
    $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
    if *CollisionFree*(x_{new}) **then**
        $X_{near} \leftarrow NearC(\mathcal{T}, x_{new})$;
        $x_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{new})$;
        $\mathcal{T}.addNodEdge(x_{min}, x_{new})$;
        $\mathcal{T}.rewire()$;

Consider N nearing nodes

Rapidly-exploring Random Tree* (RRT*)

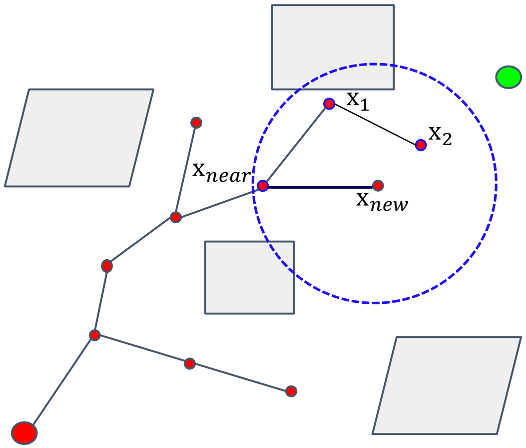


Algorithm 2: RRT Algorithm

Input: $\mathcal{M}$,x_{init},x_{goal}
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
    $x_{rand} \leftarrow Sample(\mathcal{M})$;
    $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
    $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
    if *CollisionFree*(x_{new}) **then**
        $X_{near} \leftarrow NearC(\mathcal{T}, x_{new})$;
        $x_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{new})$;
        $\mathcal{T}.addNodEdge(x_{min}, x_{new})$;
        $\mathcal{T}.rewire()$;

Consider history cost instead of only local information

Rapidly-exploring Random Tree* (RRT*)



Algorithm 2: RRT Algorithm

Input: $\mathcal{M}$,x_{init},x_{goal}

Result: A path Γ from x_{init} to x_{goal}

$\mathcal{T}$.init();

for $i = 1$ *to* n **do**

  $x_{rand} \leftarrow Sample(\mathcal{M})$;

  $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;

  $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;

  if *CollisionFree*(x_{new}) **then**

    $X_{near} \leftarrow NearC(\mathcal{T}, x_{new})$;

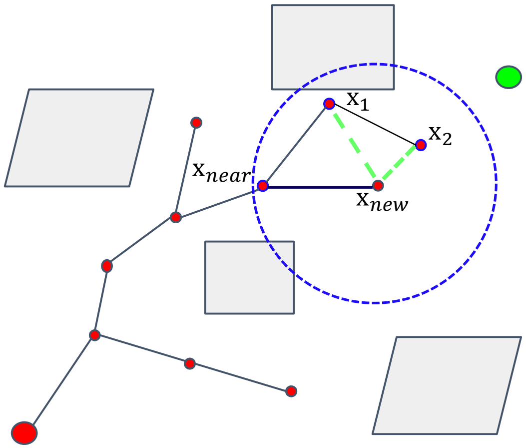
    $x_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{new})$;

    $\mathcal{T}.addNodEdge(x_{min}, x_{new})$;

    $\mathcal{T}.rewire()$;

Consider history cost instead of only local information

Rapidly-exploring Random Tree* (RRT*)

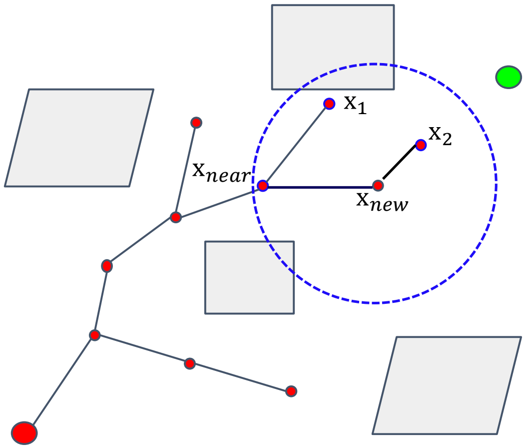


Algorithm 2: RRT Algorithm

Input: $\mathcal{M}$,x_{init},x_{goal}
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
  $x_{rand} \leftarrow Sample(\mathcal{M})$;
  $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
  $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
  if *CollisionFree*(x_{new}) **then**
    $X_{near} \leftarrow NearC(\mathcal{T}, x_{new})$;
    $x_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{new})$;
    $\mathcal{T}.addNodEdge(x_{min}, x_{new})$;
    $\mathcal{T}.rewire()$;

Rewire to improve local optimality

Rapidly-exploring Random Tree* (RRT*)



Algorithm 2: RRT Algorithm

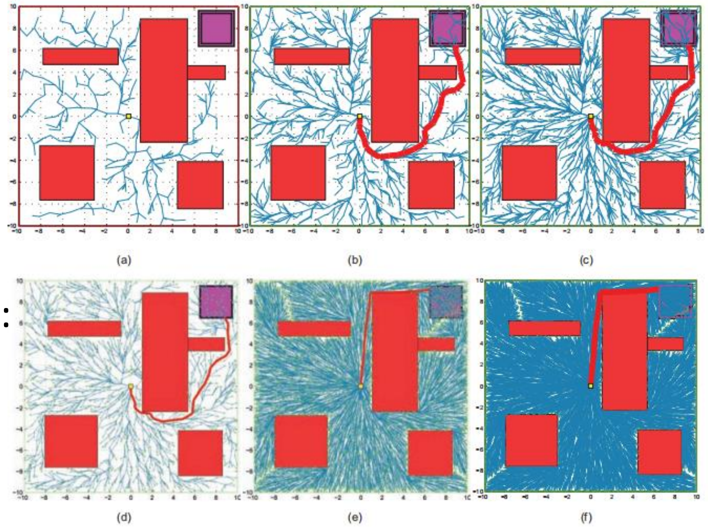
Input: $\mathcal{M}$,x_{init},x_{goal}
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
  $x_{rand} \leftarrow Sample(\mathcal{M})$;
  $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
  $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
  if *CollisionFree*(x_{new}) **then**
    $X_{near} \leftarrow NearC(\mathcal{T}, x_{new})$;
    $x_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{new})$;
    $\mathcal{T}.addNodEdge(x_{min}, x_{new})$;
    $\mathcal{T}.rewire()$;

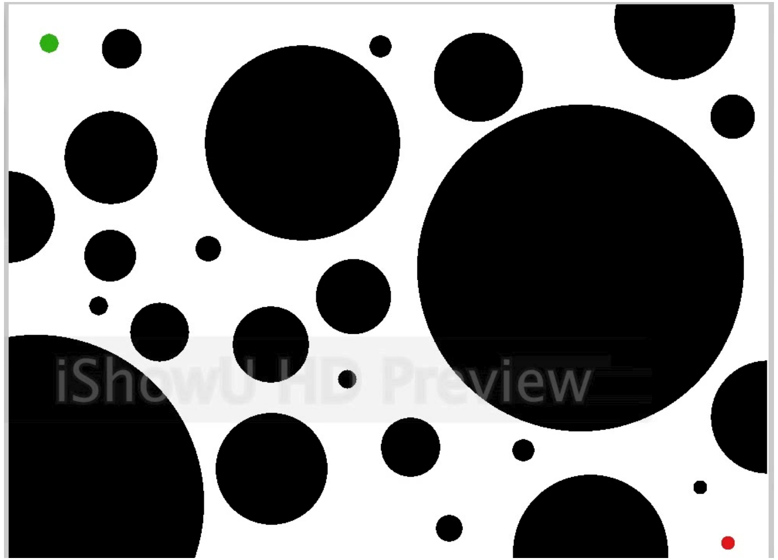
Rewire to improve local optimality

Rapidly-exploring Random Tree* (RRT*)

RRT:

RRT*:

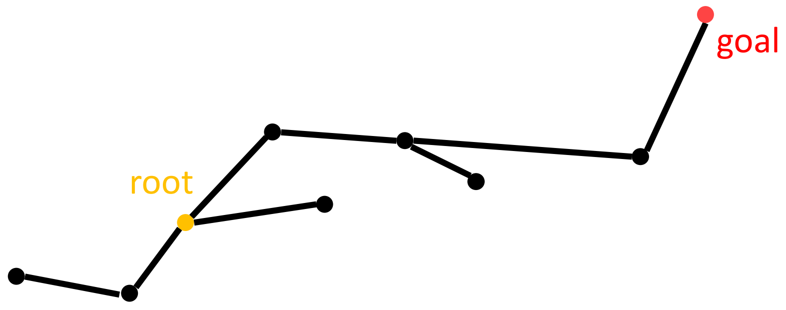




Accelerate convergence

RRT#

Flaws in the exploitation of RRT*

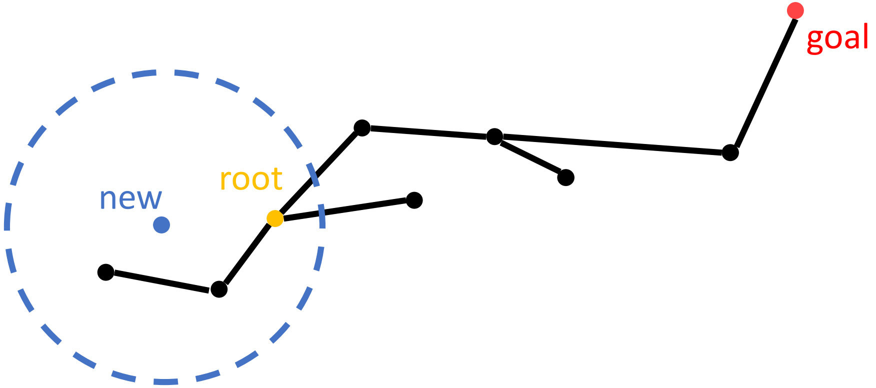


Over-exploitation

No need to “Rewire” non-promising vertexes.

RRT#

Flaws in the exploitation of RRT*

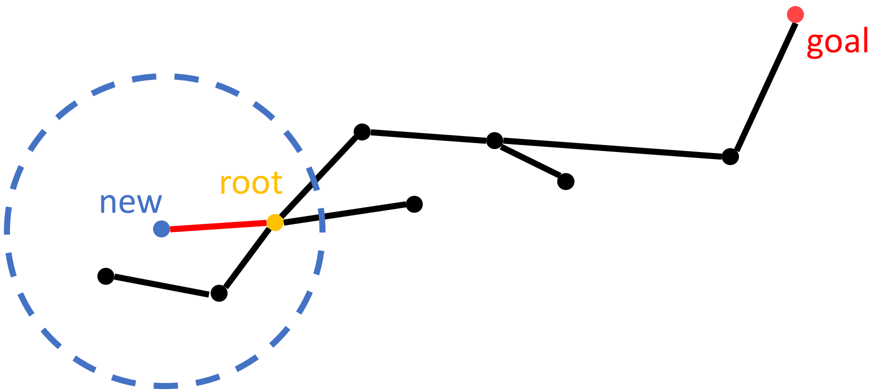


Over-exploitation

No need to “Rewire” non-promising vertexes.

RRT#

Flaws in the exploitation of RRT*

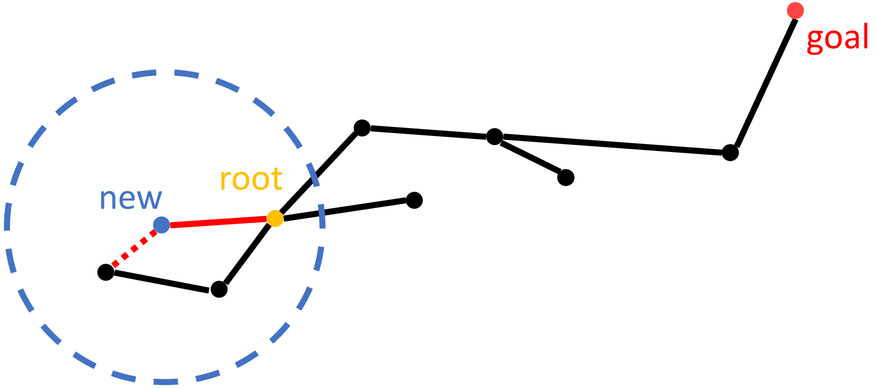


Over-exploitation

No need to “Rewire” non-promising vertexes.

RRT#

Flaws in the exploitation of RRT*

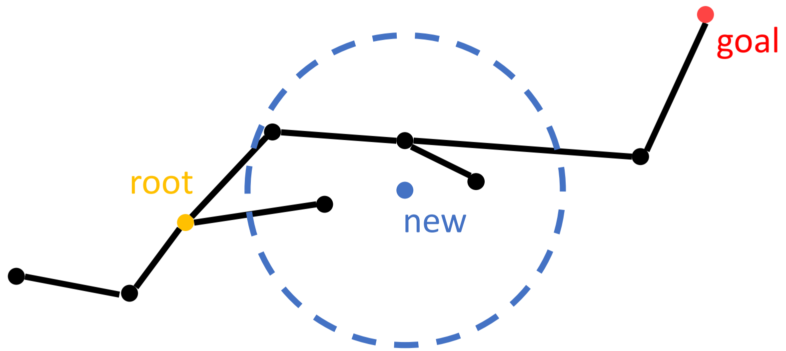


Over-exploitation

No need to “Rewire” non-promising vertexes.

RRT#

Flaws in the exploitation of RRT*



Over-exploitation

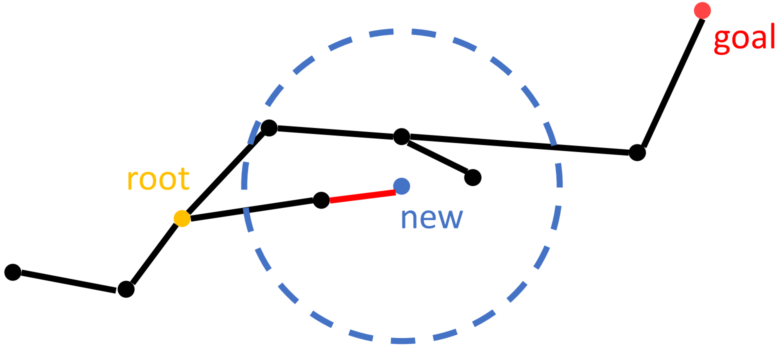
No need to “Rewire” non-promising vertexes.

Under-exploitation

- “Rewire” only happens after a new node is add to the spanning tree and only works on its “Near” nodes.
- It does not offer any guarantees that the interim path at any intermediate iteration is optimal.

RRT#

Flaws in the exploitation of RRT*



Over-exploitation

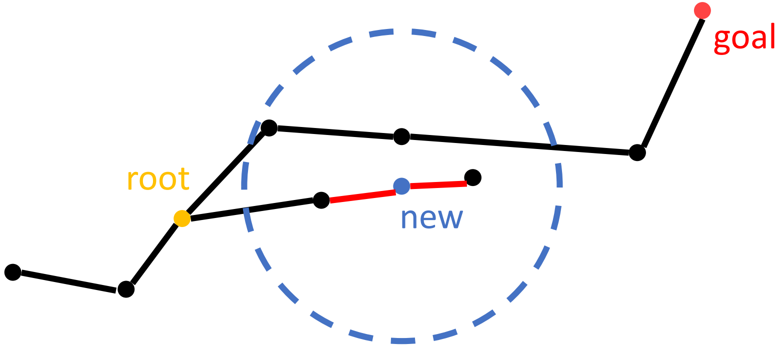
No need to “Rewire” non-promising vertexes.

Under-exploitation

- “Rewire” only happens after a new node is add to the spanning tree and only works on its “Near” nodes.
- It does not offer any guarantees that the interim path at any intermediate iteration is optimal.

RRT#

Flaws in the exploitation of RRT*



Over-exploitation

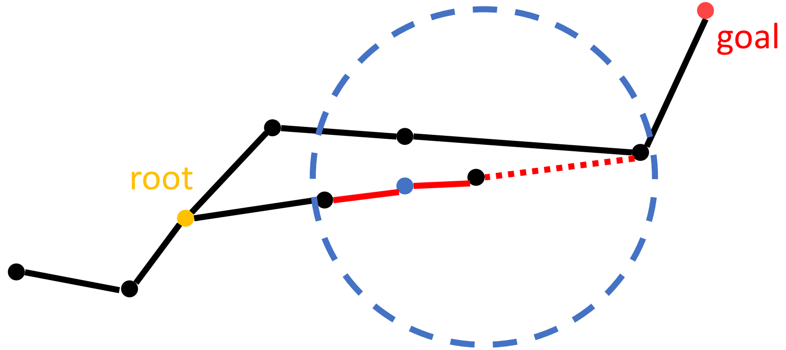
No need to “Rewire” non-promising vertexes.

Under-exploitation

- “Rewire” only happens after a new node is add to the spanning tree and only works on its “Near” nodes.
- It does not offer any guarantees that the interim path at any intermediate iteration is optimal.

RRT#

Flaws in the exploitation of RRT*



Over-exploitation

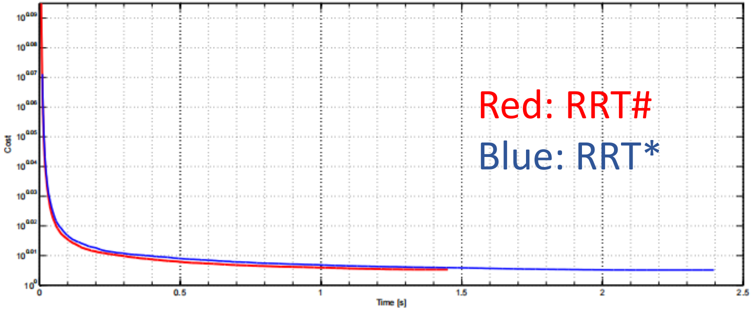
No need to “Rewire” non-promising vertexes.

Under-exploitation

- “Rewire” only happens after a new node is add to the spanning tree and only works on its “Near” nodes.
- It does not offer any guarantees that the interim path at any intermediate iteration is optimal.

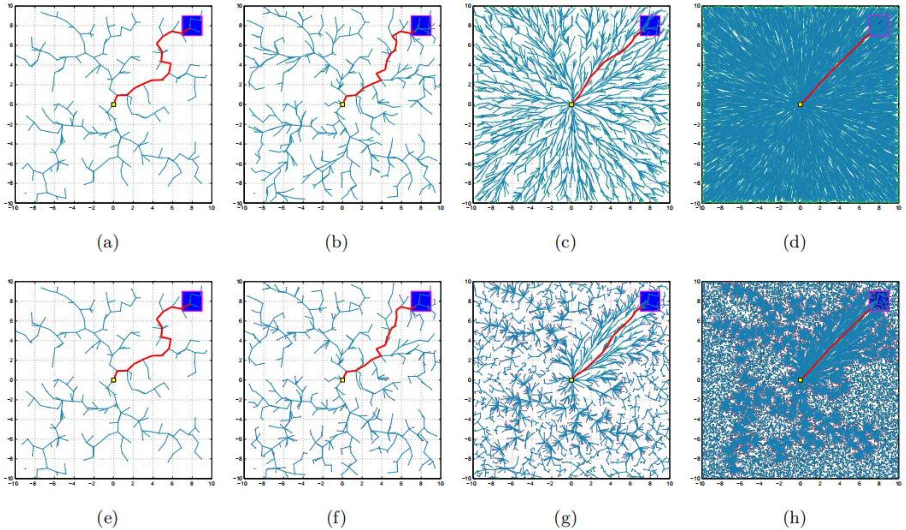
RRT#

- An improvement of RRT*
- Probabilistic Complete
- Asymptotically optimal

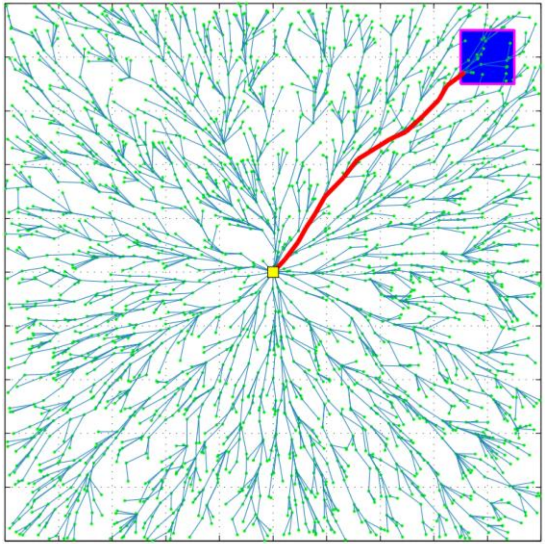


RRT*

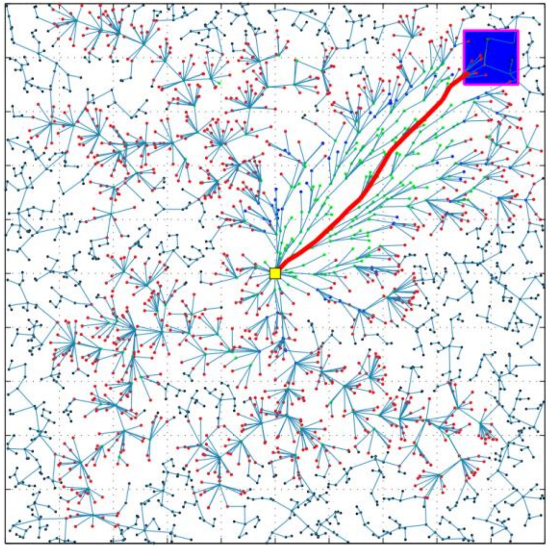
RRT#



RRT#



RRT*

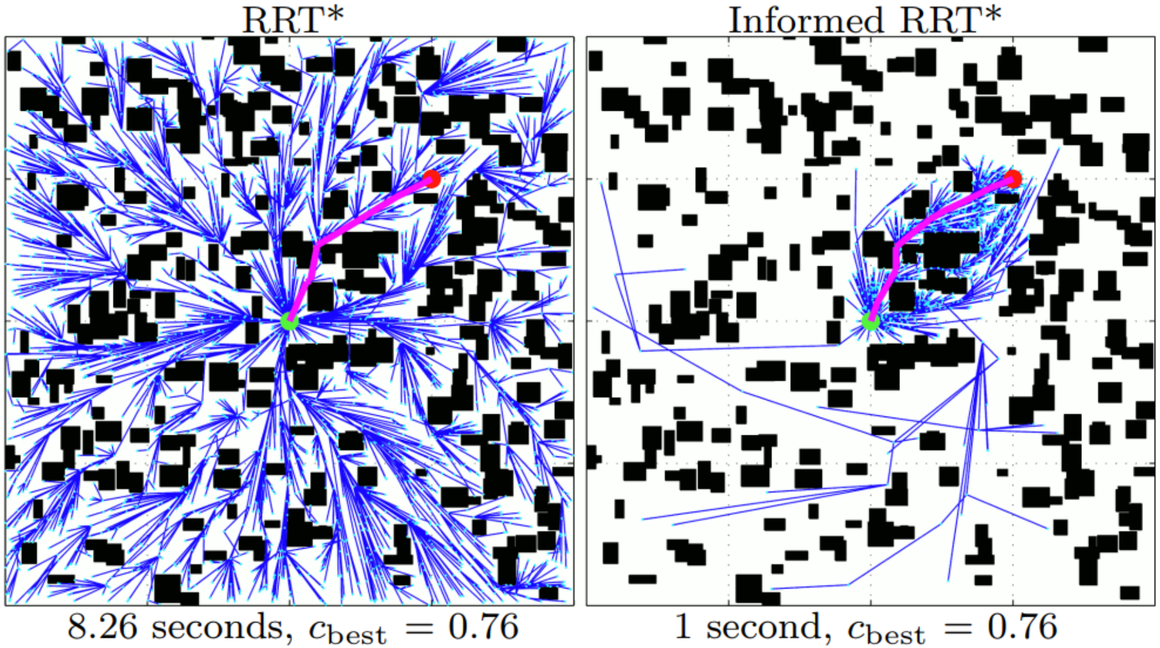


RRT#

Cons:

- Maintain a priority queue to make each consistent node optimal is not worthy in some degree.

Informed RRT*

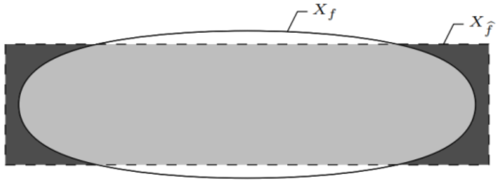


Informed RRT*

Informed sets:

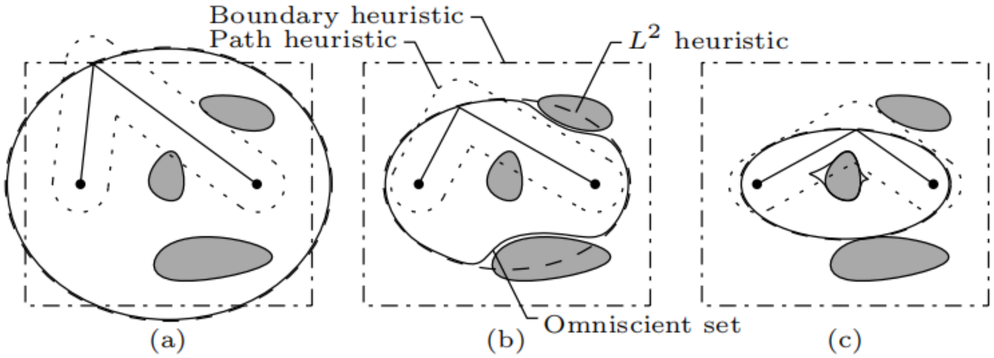
estimates of the omniscient set

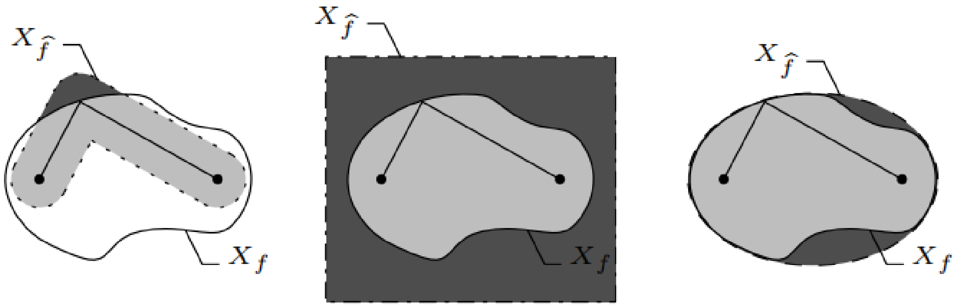
- Bounding boxes
- Path heuristics
- *L*2 heuristic



$$\mathrm{Precision}\left(X_{\hat{f}}\right) := \frac{\lambda\left(X_{\hat{f}} \cap X_f\right)}{\lambda\left(X_{\hat{f}}\right)} \equiv \frac{\square}{\square + \blacksquare}$$

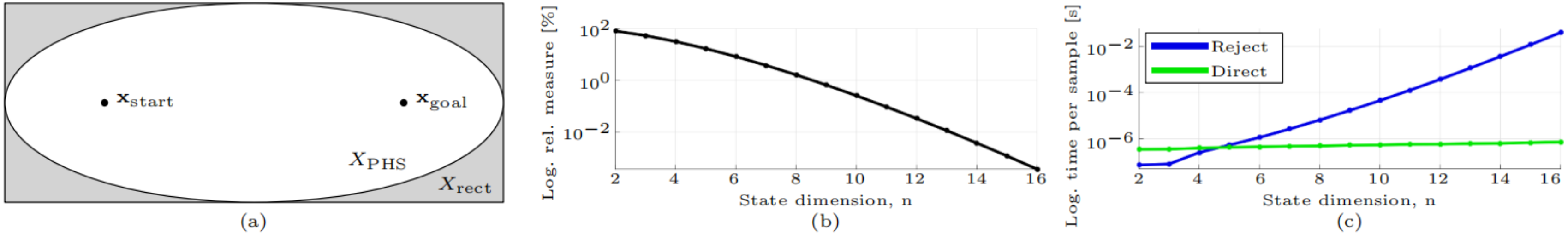
$$\mathrm{Recall}\left(X_{\hat{f}}\right) := \frac{\lambda\left(X_{\hat{f}} \cap X_f\right)}{\lambda\left(X_f\right)} \equiv \frac{\square}{\square + \square}$$





(a) Path biasing (b) Bounding box (c) L^2 informed set

Informed RRT*



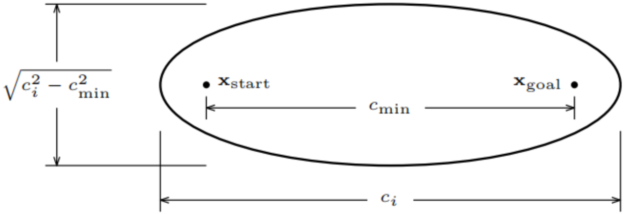
Direct Sampling in the L2 Informed set

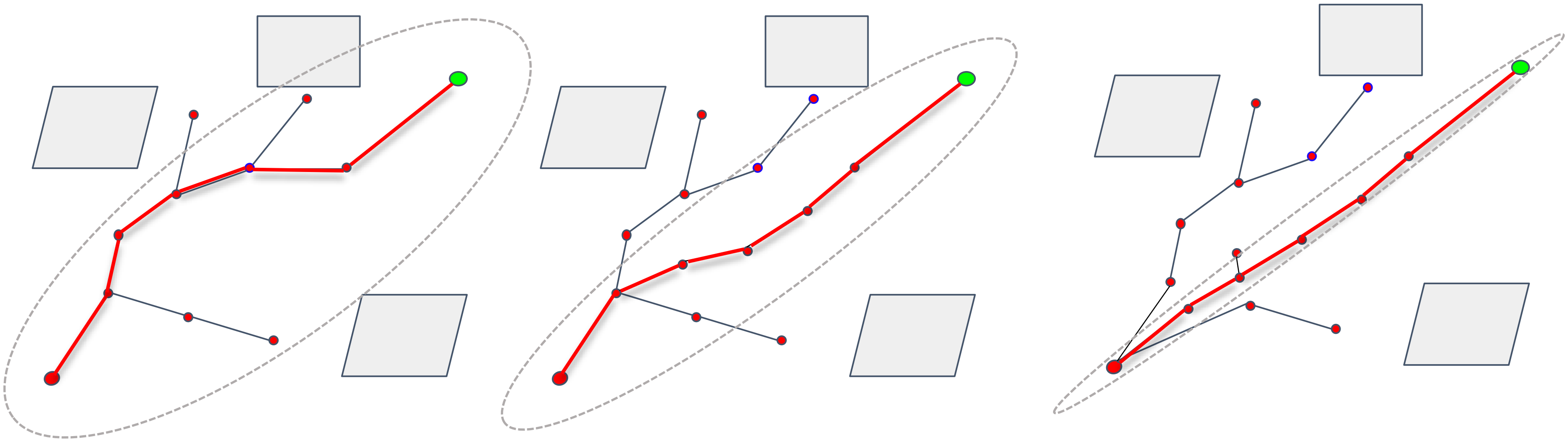
vs

Reject sampling in the bounding rectangular

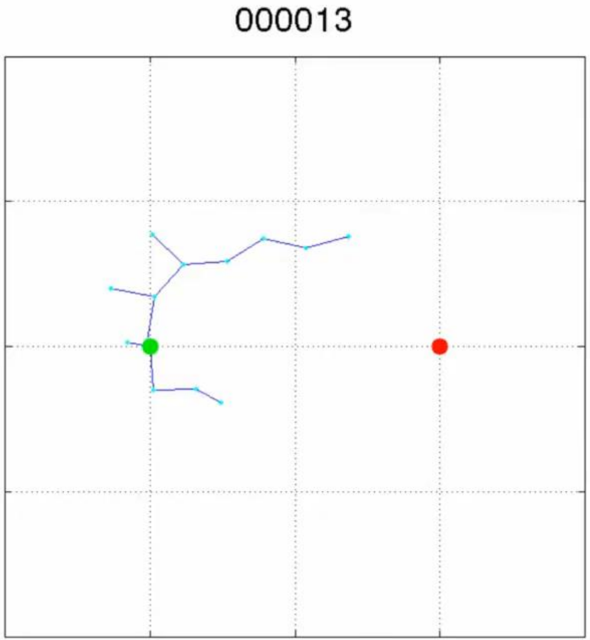
Informed RRT*

L2 Informed set





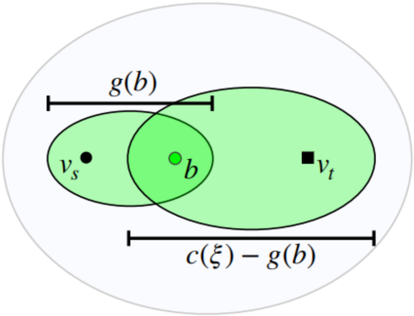
Informed RRT*

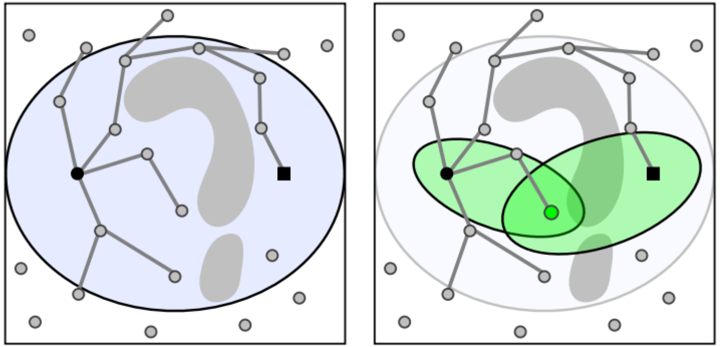


Cons:

- Can only apply to L2 norm cost (Euclidean distance).

GuILD *Guided Incremental Local Densification*





(a) (b)

Local Subsets (green):

Defined by

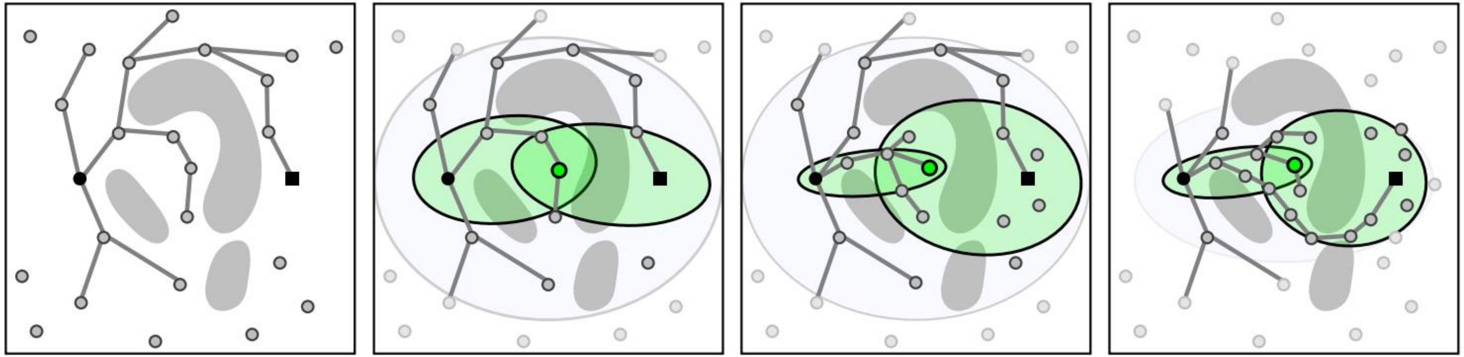
- a beacon b
- the cost-to-come on the search tree $g(b)$
- the current best solution cost $c(\xi)$.

Sampling in the L2 Informed set

vs

Sampling in the Local Subsets

GuILD *Guided Incremental Local Densification*

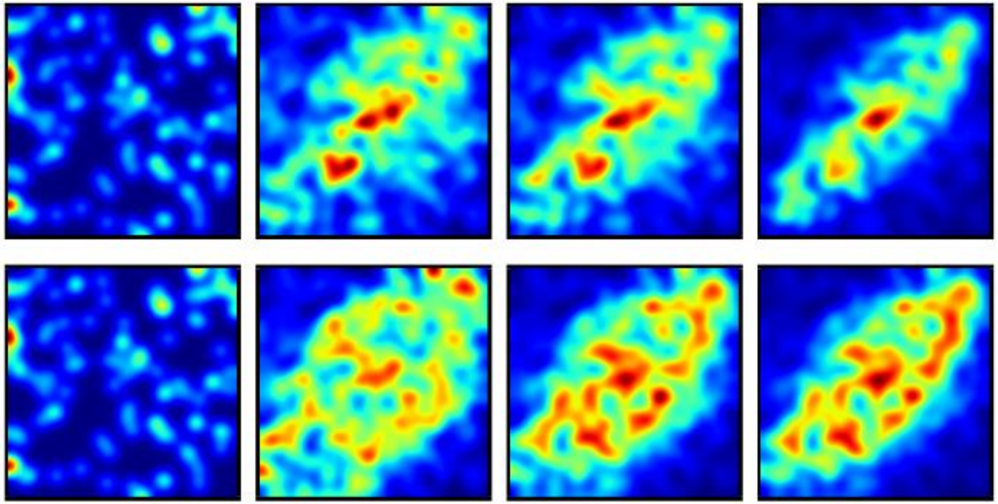


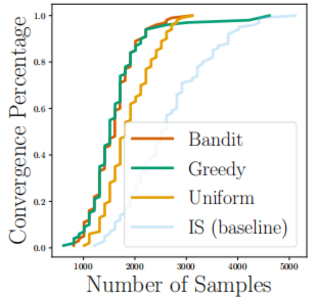
(a) (b) (c) (d)

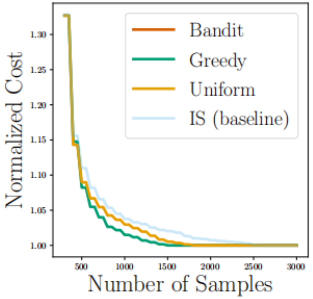
The Informed Set is unchanged. However, GuILD leverages the improved cost-to-come in the search tree to update the Local Subsets.
The start-beacon set shrinks to further focus sampling, and the remaining slack between the beacon's and goal's cost-to-comes is used to expand the beacon-target set.

GuILD *Guided Incremental Local Densification*

Forest



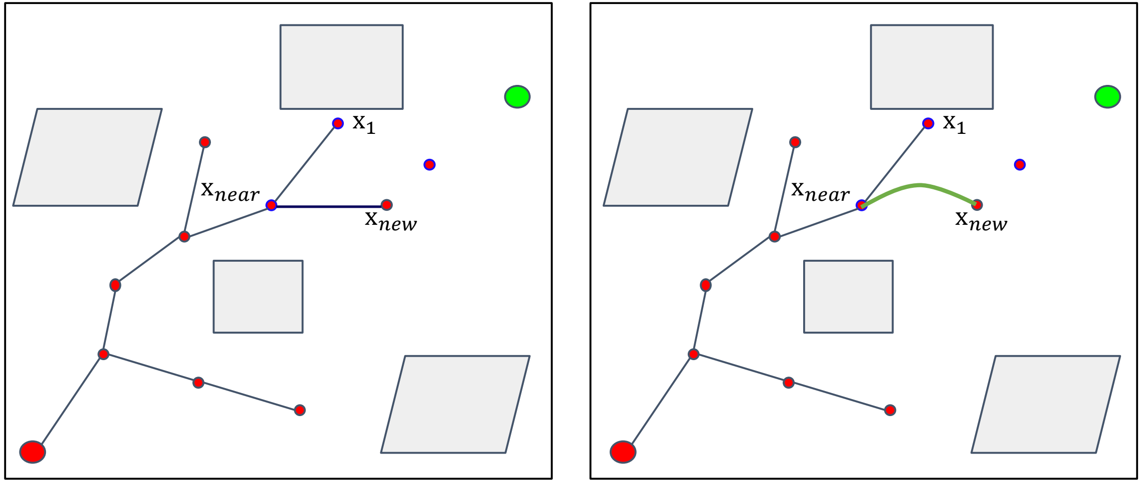




Sample heat-maps for Uniform (top) and IS (bottom) on the Forest environment

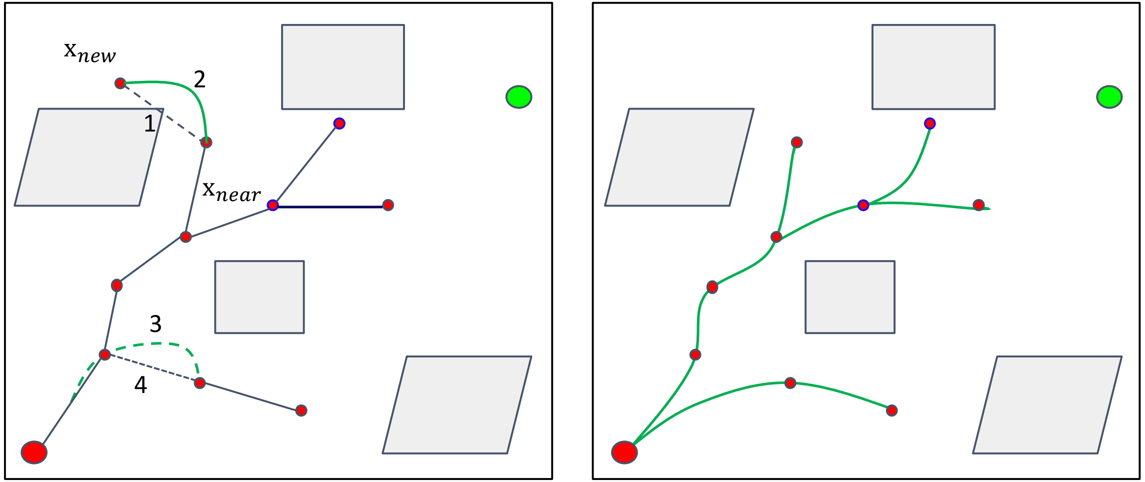
Kinodynamic Variants

Kinodynamic-RRT*



Change **Steer()** function to fit with motion or other constraints in robot navigation.

Kinodynamic-RRT*



Change **Steer()** function to fit with motion or other constraints in robot navigation.

Kinodynamic-RRT* *Workflow*

浙江大学

Similar to RRT* but different in details

Algorithm 2: RRT Algorithm

Input: $\mathcal{M}, x_{init}, x_{goal}$
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
  $x_{rand} \leftarrow Sample(\mathcal{M})$;
  $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
  $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
  if *CollisionFree*(x_{new}) **then**
    $X_{near} \leftarrow NearC(\mathcal{T}, x_{new})$;
    $x_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{new})$;
    $\mathcal{T}.addNodEdge(x_{min}, x_{new})$;
    $\mathcal{T}.rewire()$;

Kinodynamic RRT*

Input: E, x_init, x_goal

Output: A trajectory T from x_init to x_goal

T.init();
for i = 1 to n **do**
  x_rand ← Sample(E);
  X_near ← Near(T, x_rand);
  x_min ← ChooseParent(X_near, x_rand);
  T.addNode(x_rand);
  T.rewire();

1. How to "Sample"

Kinodynamic RRT*

Input: E, x_init, x_goal
Output: A trajectory T from x_init to x_goal
T.init();
for i = 1 to n **do**
    x_rand ← Sample(E);
    X_near ← Near(T, x_rand);
    x_min ← ChooseParent(X_near, x_rand);
    T.addNode(x_rand);
    T.rewire();

LTI system state-space equation:

$$\dot{x}(t) = Ax(t) + Bu(t) + c$$

For example for double integrator systems,

$$x = \begin{bmatrix} p \\ v \end{bmatrix}, A = \begin{bmatrix} 0 & I \\ 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 \\ I \end{bmatrix}$$

Instead of sampling in Euclidean space like RRT, it requires to **sample in full state space**.

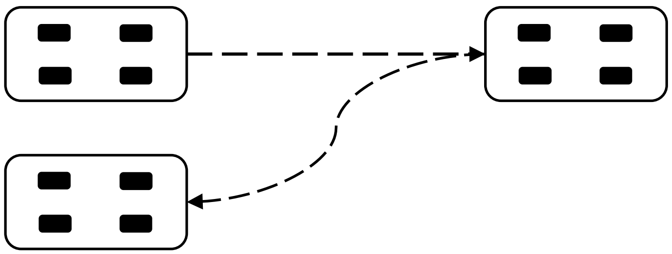
2. How to define "Near"

Kinodynamic RRT*

Input: E, x_init, x_goal

Output: A trajectory T from x_init to x_goal

```
T.init();
for i = 1 to n do
    x_rand ← Sample(E);
    X_near ← Near(T, x_rand);
    x_min ← ChooseParent(X_near, x_rand);
    T.addNode(x_rand);
    T.rewire();
```



A car can not move sideways

If without motion constraints, Euclidean distance or Manhattan distance can be used.

In state space with motion constraints, bringing in optimal control.

2. How to define “Near”

If bring optimal control, we can define **cost functions** of transferring from states to states.

$$c[\pi] = \int_0^{\tau} \left(1 + u(t)^T R u(t)\right) dt$$

For many applications, a quadratic form of time-energy optimal is adopted.

Two states are near if the cost of transferring from one state to the other is small.
(Note that the cost may be different if transfer reversely)

浙江大学

2. How to define "Near"

$$c[\pi] = \int_0^{\tau} \left(1 + u(t)^T R u(t)\right) dt$$

If we know the arriving time **τ** and the control policy **u(t)** of transferring, we can calculate the cost.

And thankfully, it's all in classic optimal control solutions. (OBVP)

浙江大学

3. How to "ChooseParent"

Kinodynamic RRT*
Input: E, x_init, x_goal
Output: A trajectory T from x_init to x_goal

T.init();
for i = 1 to n **do**
    x_rand ← Sample(E);
    X_near ← Near(T, x_rand);
    x_min ← ChooseParent(X_near, x_rand);
    T.addNode(x);
    T.rewire();

Now if we sample a random state, we can calculate control policy and cost from those state-nodes in the tree to the sampled state.

Choose one with the minimal cost and **check $x(t)$ and $u(t)$ are in bounds**.

If no qualified parent found, sample another state.

4. How to find near nodes efficiently

```
Kinodynamic RRT*
----------------------------------------------
Input: E, x_init, x_goal
Output: A trajectory T from x_init to x_goal

T.init();
for i = 1 to n do
    x_rand ← Sample(E);
    X_near ← Near(T, x_rand);
    x_min ← ChooseParent(X_near, x_rand);
    T.addNode(x);
    T.rewire();
```

Every time we sample a random state ***x_rand***, it requires to check every node in the tree to find its parent, that is solving a OBVP for each node, which is not efficient.

浙江大学

4. How to find near nodes efficiently

```
Kinodynamic RRT*
------------------------------------------------
Input: E, x_init, x_goal
Output: A trajectory T from x_init to x_goal

T.init();
for i = 1 to n do
    x_rand ← Sample(E);
    X_near ← Near(T, x_rand);
    x_min ← ChooseParent(X_near, x_rand);
    T.addNode(x);
    T.rewire();
```

If we set a **cost tolerance** r, we can actually calculate bounds of the states (forward-reachable set) that can be reached by x_rand and bounds of the states (backward-reachable set) that can reach x_rand with cost less than r.

And if we store nodes in form of a kd-tree, we can then do range query in the tree.

浙江大学

4. How to find near nodes efficiently

$$c[\tau] = \tau + [x_1 - \bar{x}(\tau)]^T G(t)^{-1} [x_1 - \bar{x}(\tau)].$$

This formula describes how cost of transferring from state x_0 to state x_1 changes with arrival time τ.

We can see that given initial state x_0, cost tolerance r and arrival time τ, the forward-reachable set of x_0 is:

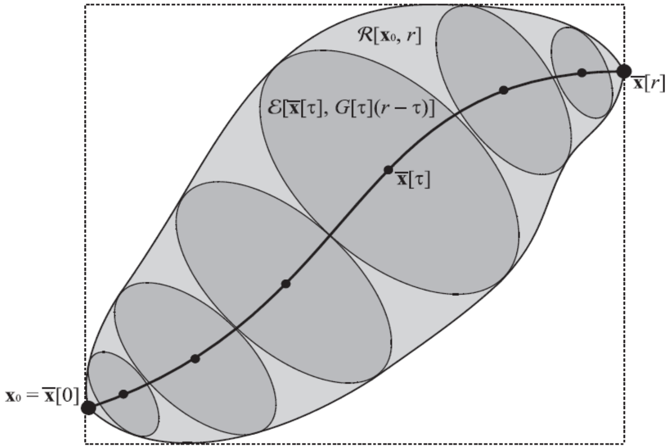
$$\begin{aligned} &\{x_1 \mid \tau + [x_1 - \bar{x}(\tau)]^T G(t)^{-1} [x_1 - \bar{x}(\tau)] < r\} \\ &= \left\{x_1 \mid [x_1 - \bar{x}(\tau)]^T \frac{G(t)^{-1}}{r-\tau} [x_1 - \bar{x}(\tau)] < 1\right\}. \\ &= \mathcal{E}[\bar{x}(\tau), G(t)(r-\tau)]. \end{aligned}$$

Kinodynamic-RRT* *Problems when it comes to motion constraints*

浙江大学

4. How to find near nodes efficiently

$$
\begin{aligned}
&\{x_1 \,|\tau + [x_1 - \bar{x}(\tau)]^T G(t)^{-1}[x_1 - \bar{x}(\tau)] < r\} \\
&= \left\{x_1 \mid [x_1 - \bar{x}(\tau)]^T \frac{G(t)^{-1}}{r-\tau}[x_1 - \bar{x}(\tau)] <1\right\}. \\
&= \mathcal{E}[\bar{x}(\tau), G(t)(r-\tau)].
\end{aligned}
$$



where $\mathcal{E}[x, M]$ is an **ellipsoid** with center x and positive definite weight matrix M, formally defined as:

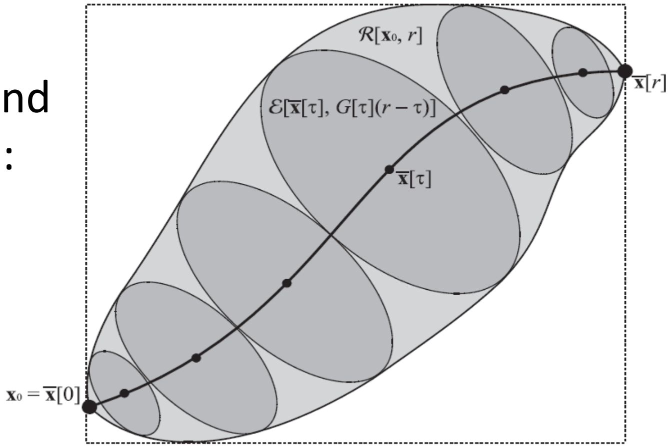
$$\mathcal{E}[x, M] = \{x' \mid (x' - x)^T M^{-1}(x' - x)<1\}.$$

Hence, the forward-reachable set is the union of high dimensional ellipsoids for all possible arrival times τ.

4. How to find near nodes efficiently

For simplification, we sample several τs and calculate axis-aligned bounding box of the ellipsoids for each τ and update the maximum and minimum in each dimension:

$$\prod_{k=1}^{n}\begin{bmatrix} min\{0<\tau<r\}\left(\bar{x}(\tau)_k-\sqrt{G[\tau]_{(k,k)}(r-\tau)}\right), \\ max\{0<\tau<r\}\left(\bar{x}(\tau)_k+\sqrt{G[\tau]_{(k,k)}(r-\tau)}\right)\end{bmatrix}.$$



Similar for the calculation of the backward-reachable set.

浙江大学

4. How to find near nodes efficiently

```
Kinodynamic RRT*
Input: E, x_init, x_goal
Output: A trajectory T from x_init to x_goal
T.init();
for i = 1 to n do
    x_rand ← Sample(E);
    X_near ← Near(T, x_rand);
    x_min ← ChooseParent(X_near, x_rand);
    T.addNode(x);
    T.rewire();
```

When do "Near" query and "ChooseParent" , X_near can be found from the **backward-reachable set** of x_rand.

浙江大学

5. How to "Rewire"

Kinodynamic RRT*
Input: E, x_init, x_goal
Output: A trajectory T from x_init to x_goal

```
T.init();
for i = 1 to n do
    x_rand ← Sample(E);
    X_near ← Near(T, x_rand);
    x_min ← ChooseParent(X_near, x_rand);
    T.addNode(x);
    T.rewire();
```

When "Rewire", we calculate the **forward-reachable set** of x_rand, and solve OBVPs.

感谢各位

Thanks for Listening