深蓝学院
shenlanxueyuan.com

Sampling-based Path Finding

■ Lecture 3

Speaker: Hongkai Ye

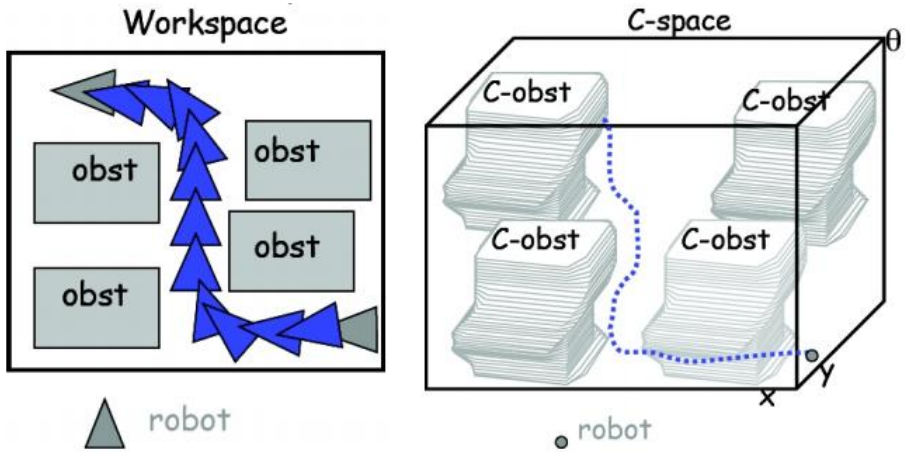
D. Eng. Candidate in Robotics
Zhejiang University

Acknowledgements: Slides based on a previous version, courtesy of Dr. Chaoqun Wang.

Preliminaries

From Workspace to Configuration Space

- Robot transformed into a point in configuration space.
- Paths transformed from swept volume into 1 dimensional curves.



Figs from Jyh-Ming Lien

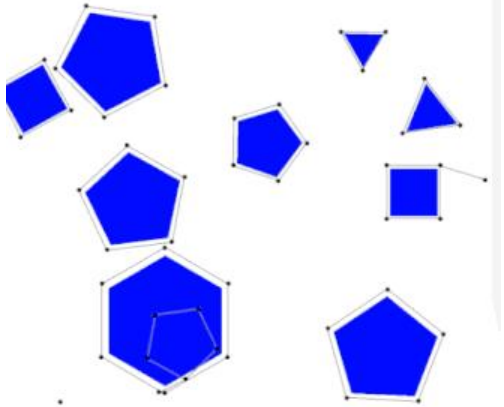
Preliminaries

Sampling-based Planners (*or Probabilistic Methods*)

capture the connectivity of the solution space to bring a feasible or optimal solution

by constructing Trees or Graphs connected by discrete C-space samples sampled incrementally or in batch in the continuous C-space without explicitly constructing it.

Their performance mostly depends on *sampling-strategy*, *collision-checking*, and *neighbor-search*.



Animation from Wikipedia

Preliminaries

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.

Terminology

- **Probabilistic Completeness**

A feasible solution will be found, if one exists, with probability one as the number of samples approaches infinity.

- **Asymptotical (Near) Optimality**

The cost of the returned solution converges almost surely to the optimum as the number of samples approaches infinity.

- **Anytime**

Quickly find a feasible but not necessarily optimal solution, then incrementally improve it over time.

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
- LBT-RRT 2016 TRO
- SST* 2016 IJRR
- RABIT* 2016 ICRA
- DRRT 2017 RSS
- AIT* 2020 ICRA
- **GuILD 2021 Arxiv**
- EIT* 2022 IJRR
- …

Content

1. Feasible path planning methods :

Probabilistic Road Map (PRM),

Rapidly-exploring Random Tree (RRT)

2. Optimal path planning methods:

RRT*

3. Accelerate convergence:

RRT#, Informed-RRT*, and GuILD

Feasible Path Planning Methods

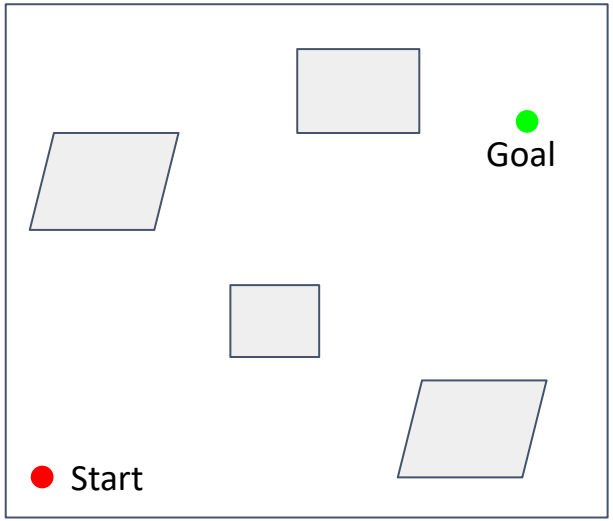
Probabilistic Road Map

What is PRM?

- Probe space connectivity in a graph structure;
- Fine coverage of free space;
- Divide planning into two phases:

  Learning phase

  Query phase

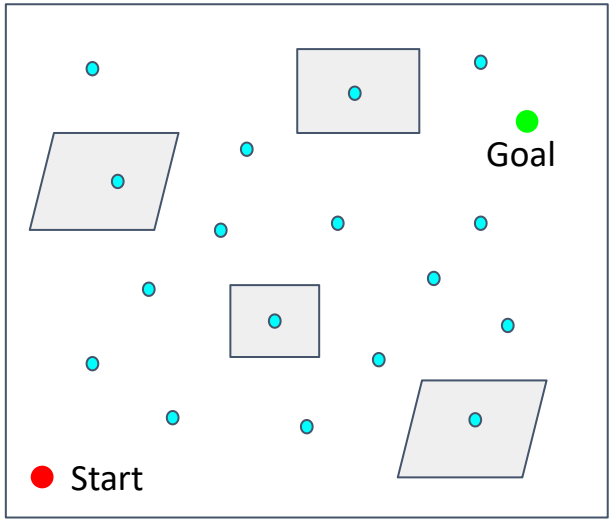


Probabilistic Road Map

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

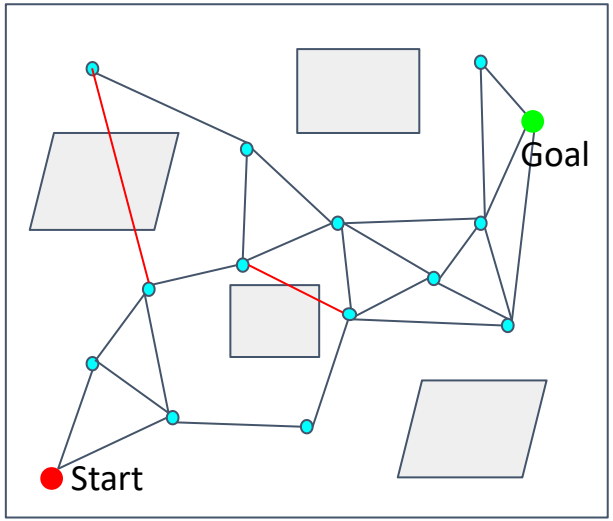


Probabilistic Road Map

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.

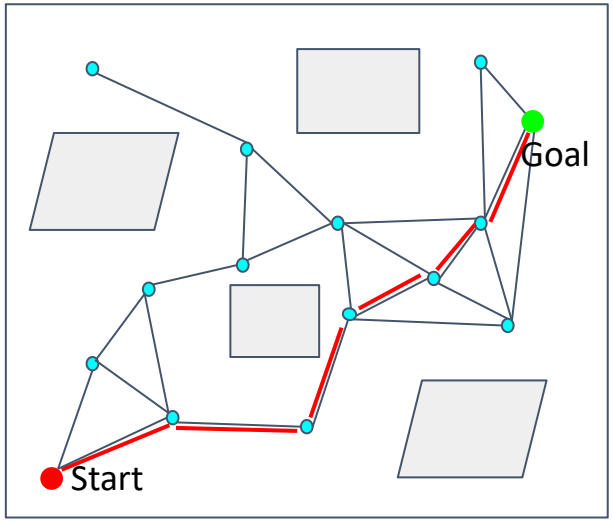


Probabilistic Road Map

Query phase:

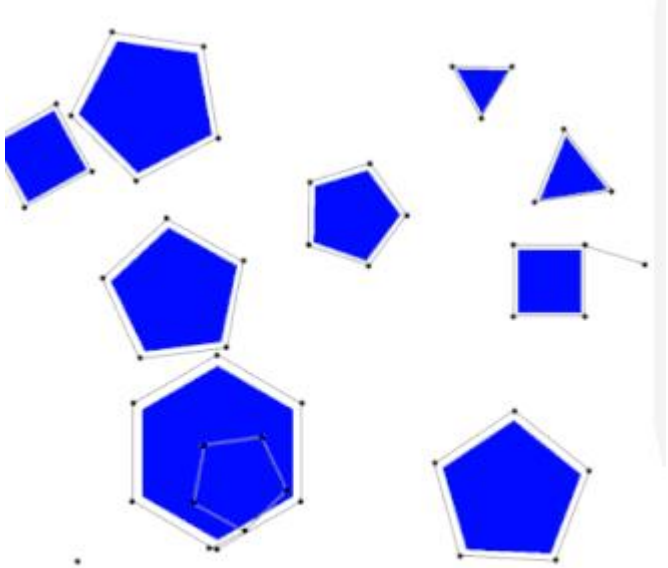
Search on the road map to find a path from the start to the goal (using Dijkstra's algorithm or the A* algorithm).

- Road map is now similar with the grid map (or a simplified grid map).

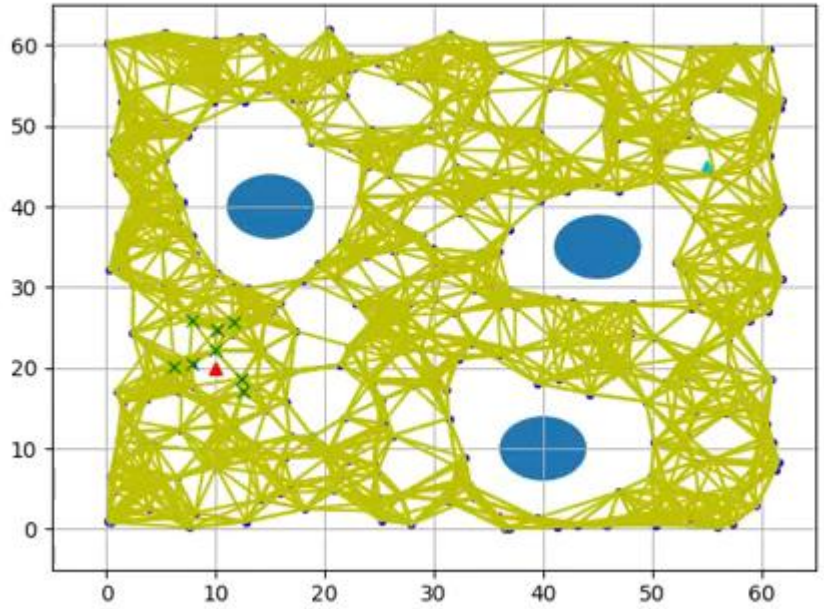


Probabilistic Road Map

Learning phase



Query phase



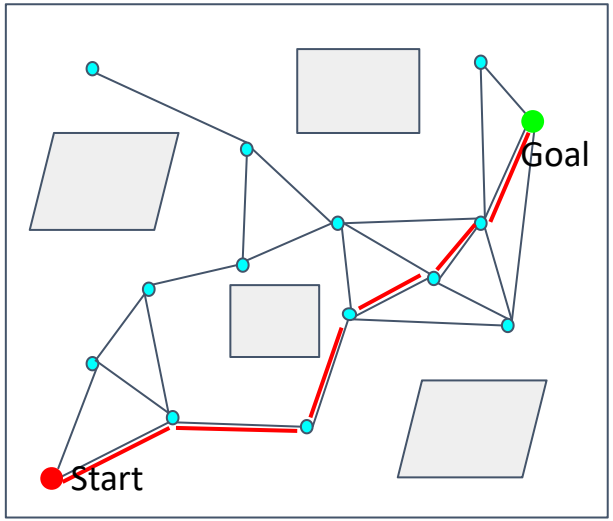
Probabilistic Road Map

Pros

- Can conduct multiple queries (static map).

Cons

- Build graph over state space but no particular focus on generating a path.



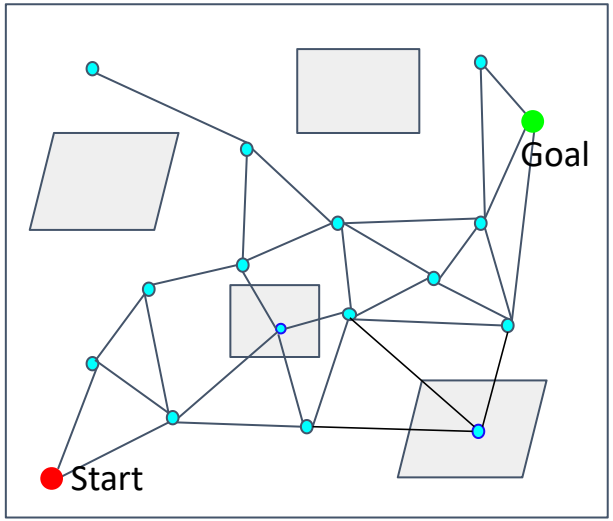
Probabilistic Road Map

Improving efficiency

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

Lazy collision-checking:

Check collisions only if necessary

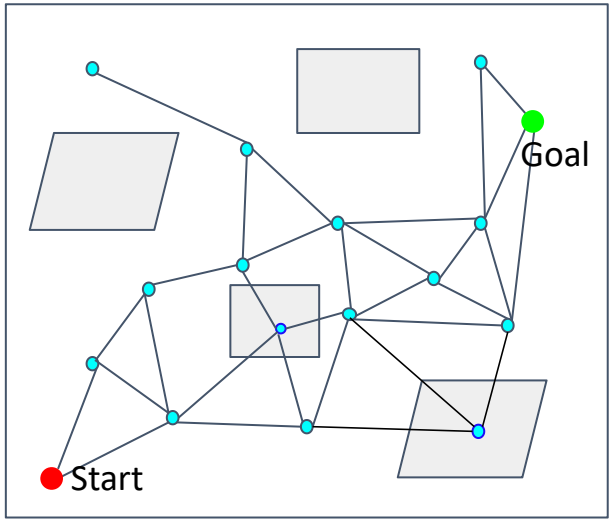


Probabilistic Road Map

Improving efficiency

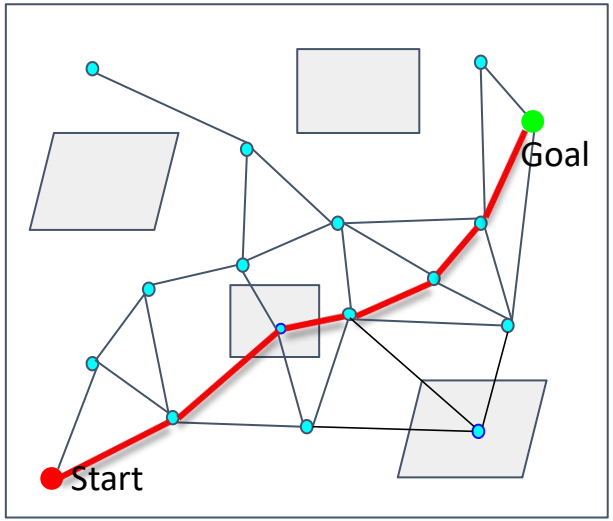
Lazy collision-checking:

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



Probabilistic Road Map

Improving efficiency

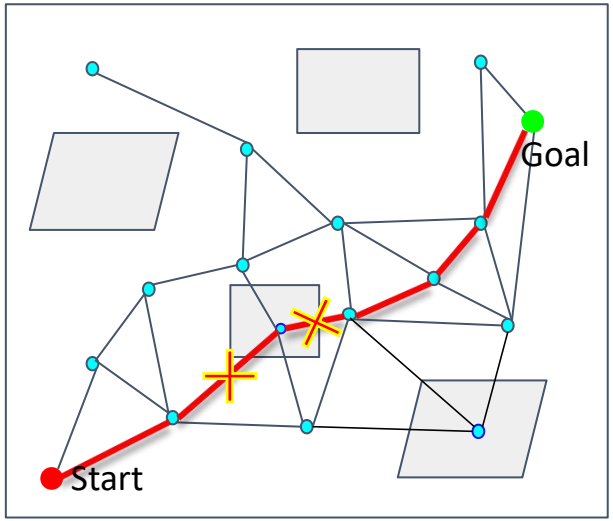


Lazy collision-checking:

- 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

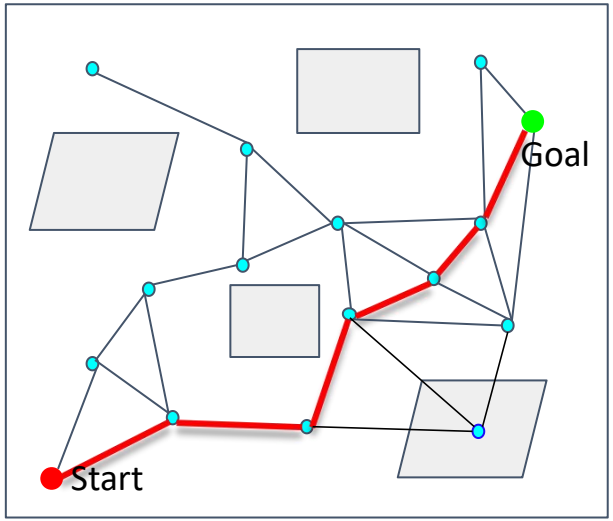


Lazy collision-checking:

- 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

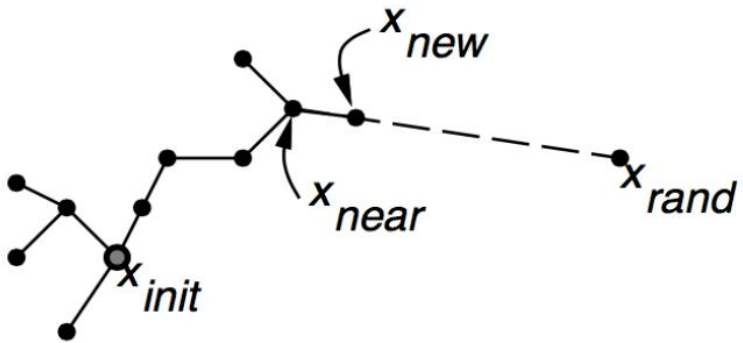


Lazy collision-checking:

- 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

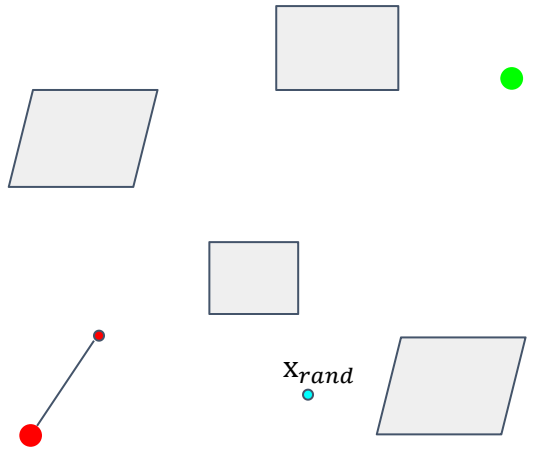
Build up a tree from start to goal through generating “next states” in the tree by executing random controls



Rapidly-exploring Random Trees

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();

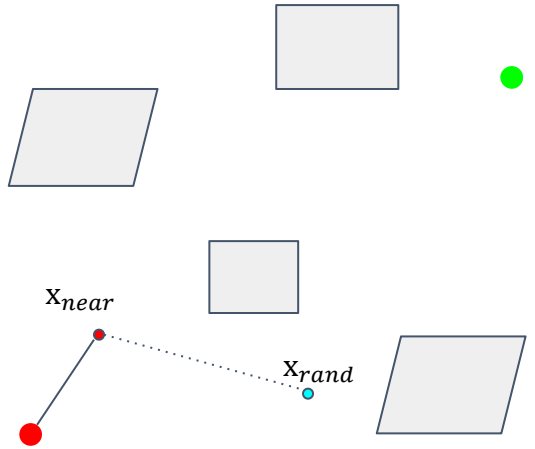


Sample a node Xrand in the free space

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();

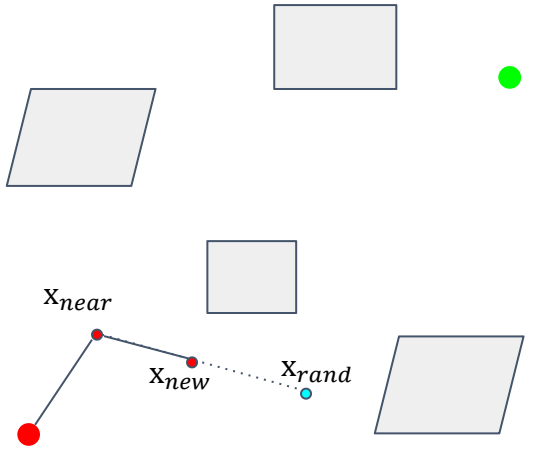


Find the nearest node Xnear in current tree

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();

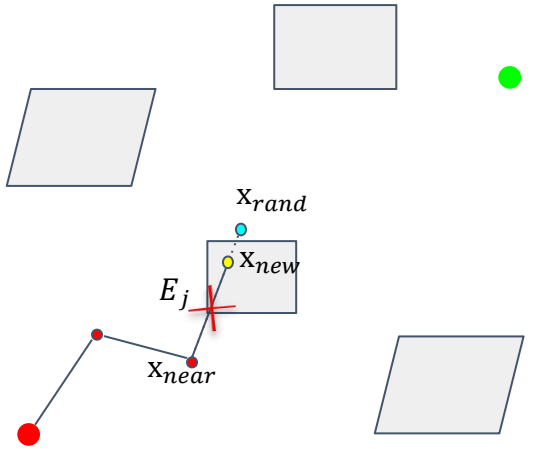


Grow a new node Xnew and path Ei from Xnear

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();

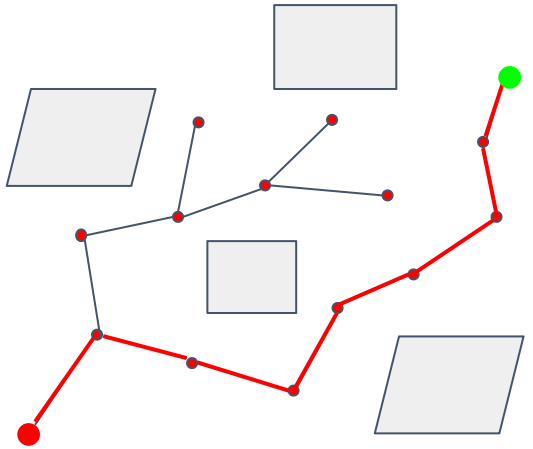


Do not grow if collision

Rapidly-exploring Random Trees

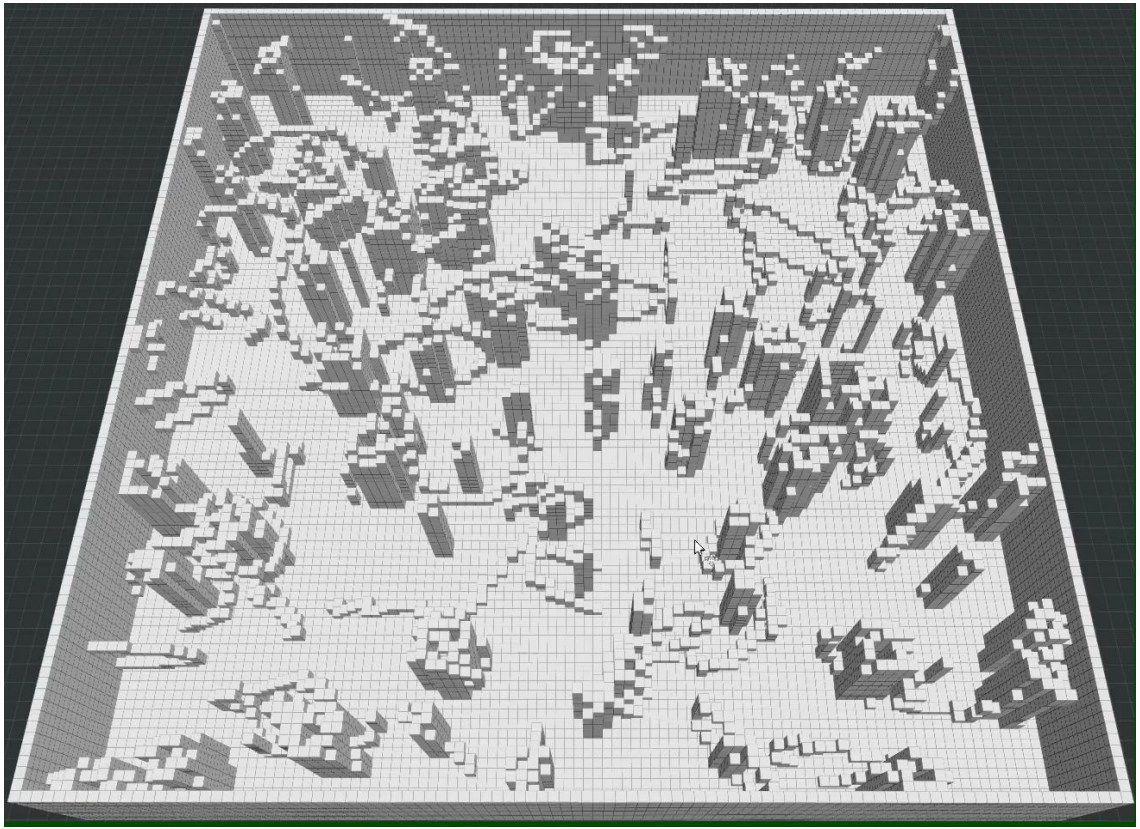
Algorithm 1: RRT Algorithm

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



https://github.com/ZJU-FAST-Lab/sampling-based-path-finding

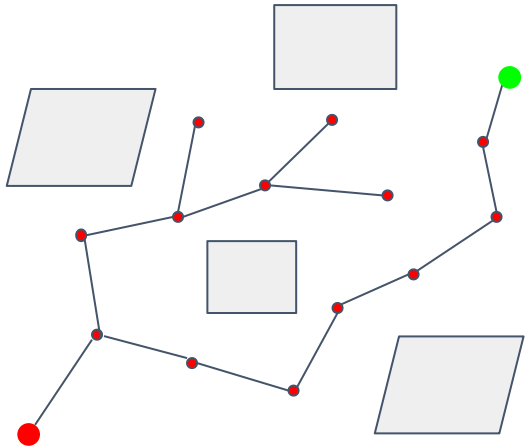
Rapidly-exploring Random Trees

Pros

- Easy to implement
- 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



Optimal Path Planning Methods

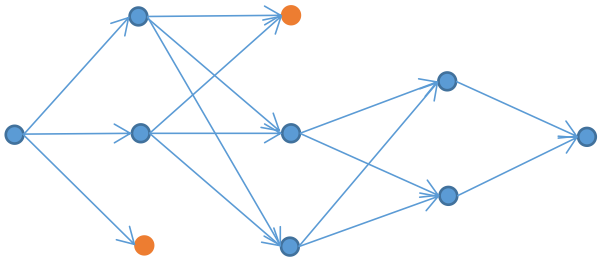
Optimal Criterion

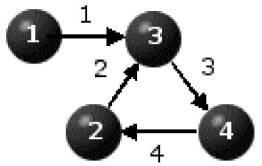
Dynamic Programming (DP) functional equation (discrete form)

$$f(j) = \min_{i \in B(j)} \{f(i) + D(i,j)\} \ , \ j \in C \backslash \{1\}$$

$B(j)$ represents for predecessors of j,

$D(i, j)$ the cost of transition from state i to state j.





$$f(1) = 0$$
$$f(2) = 4 + f(4)$$
$$f(3) = \min\{2 + f(2), 1 + f(1)\}$$
$$f(4) = 3 + f(3)$$

The DP functional equation
does not constitute an algorithm!

Optimal Criterion

Direct DP Methods

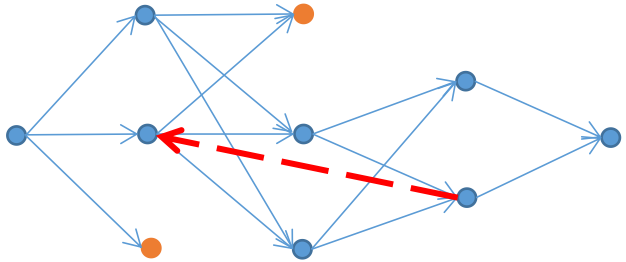
Generic Direct Method

$D(i,j) = \infty, \forall i,j \in C, i \geq j$

Initialization: $F(1) = 0$

Iteration: For $j = 2, \ldots, n$ Do:

$$F(j) = \min_{i \in B(j)} \{F(i) + D(i,j)\}$$



The cost-to-come value of each state can be determined by processing it **only once.**

- While computing the value of f(j), all the relevant values of f(i) must have **already been computed** somehow.
- The graph must be **acyclic**.

Optimal Criterion

Direct DP Methods

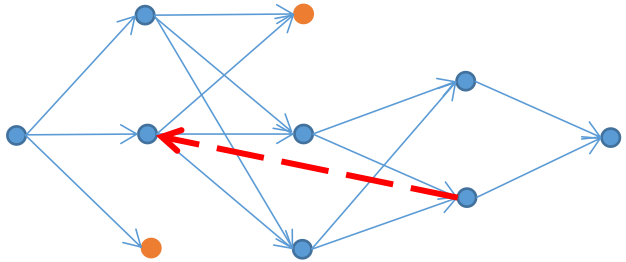
Generic Direct Method

$D(i,j) = \infty, \forall i,j \in C, i \geq j$

Initialization: $F(1) = 0$

Iteration: For $j = 2, \ldots, n$ Do:

$$F(j) = \min_{i \in B(j)} \{F(i) + D(i,j)\}$$



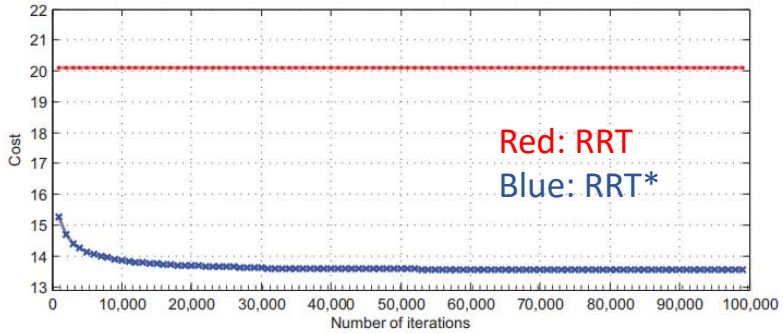
In motion planning, state graphs (or grids) are almost surely cyclic.

Each state can be in any level (step), and one can only access states by range queries or k-NN queries.

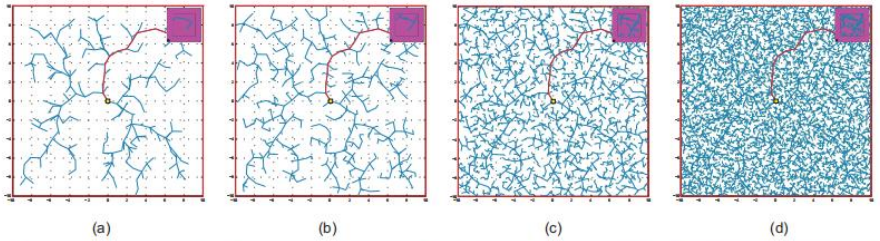
Sampling-based methods introduce some kinds of randomness in generating the implicit random geometric graph **(RGG) incrementally** or **in batch**. It can still be viewed as graph-searching.

Rapidly-exploring Random Tree* (RRT*)

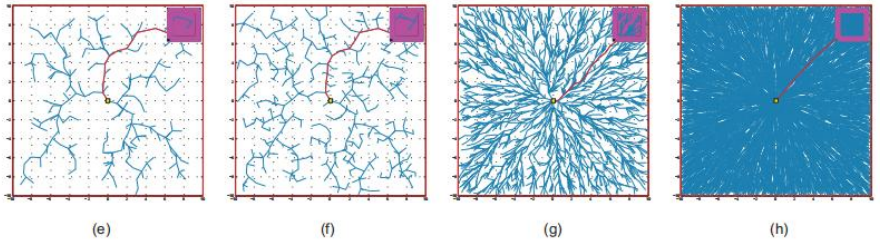
- An improvement of RRT
- Asymptotically optimal (under conditions)



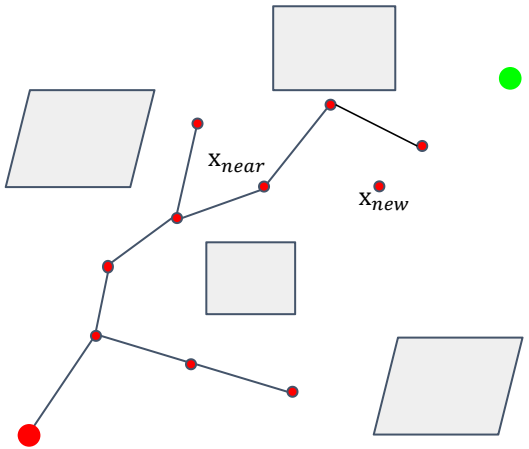
RRT:



RRT*:



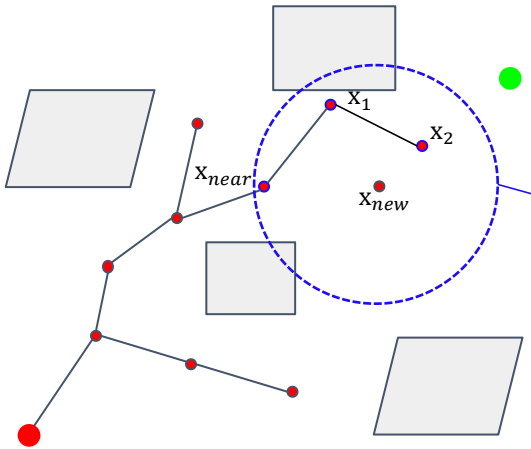
Rapidly-exploring Random Tree* (RRT*)



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*)

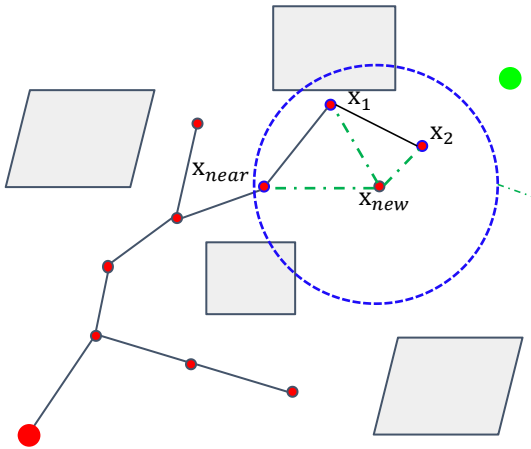


Algorithm 2: RRT*Algorithm

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*)

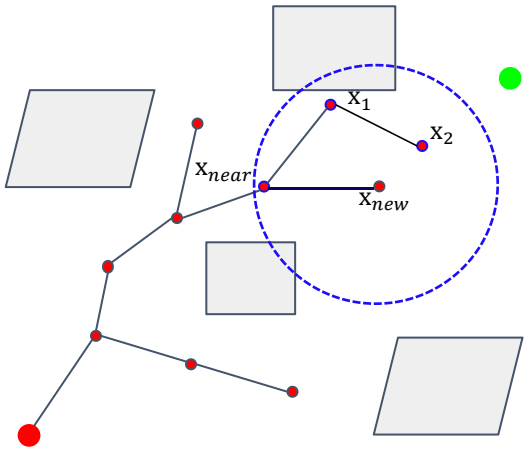


Algorithm 2: RRT*Algorithm

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*)

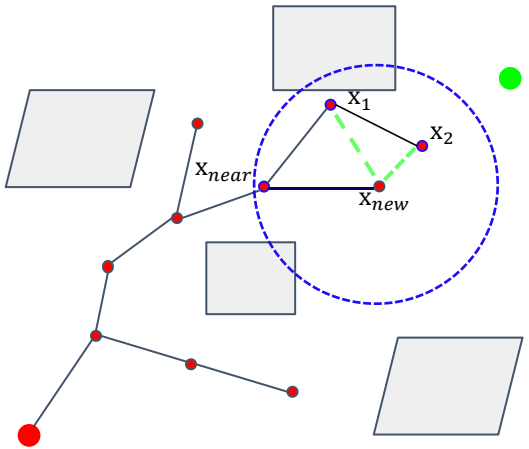


Algorithm 2: RRT*Algorithm

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*)

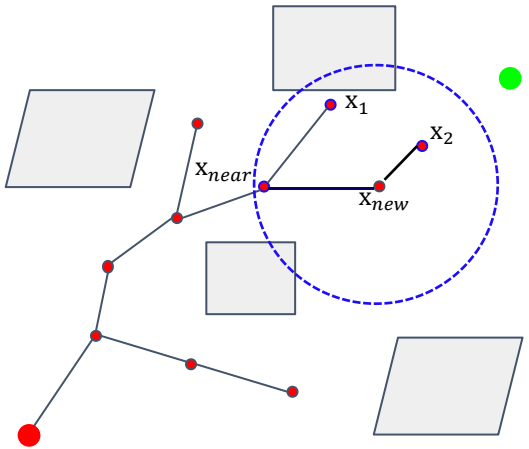


Algorithm 2: RRT*Algorithm

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*)



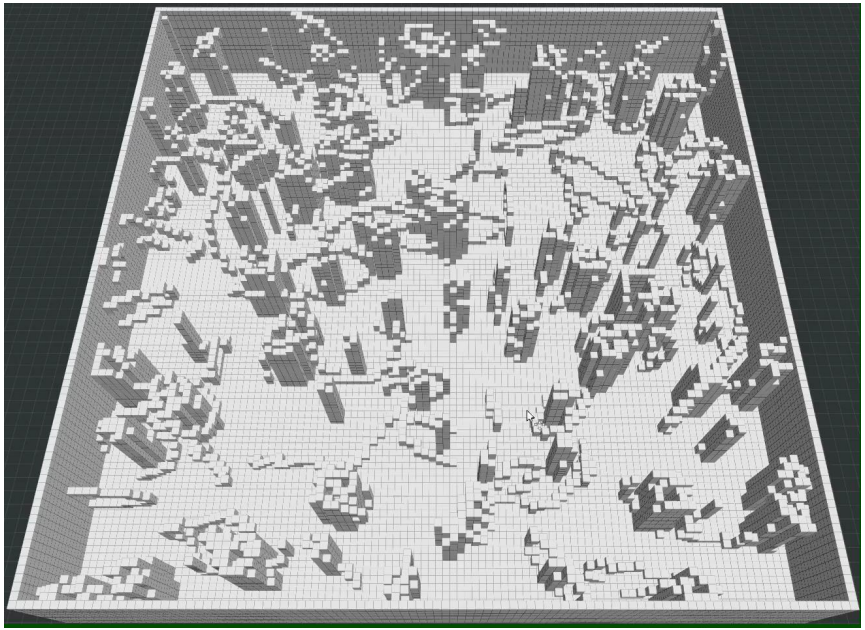
Rewire to improve local optimality

Algorithm 2: RRT*Algorithm

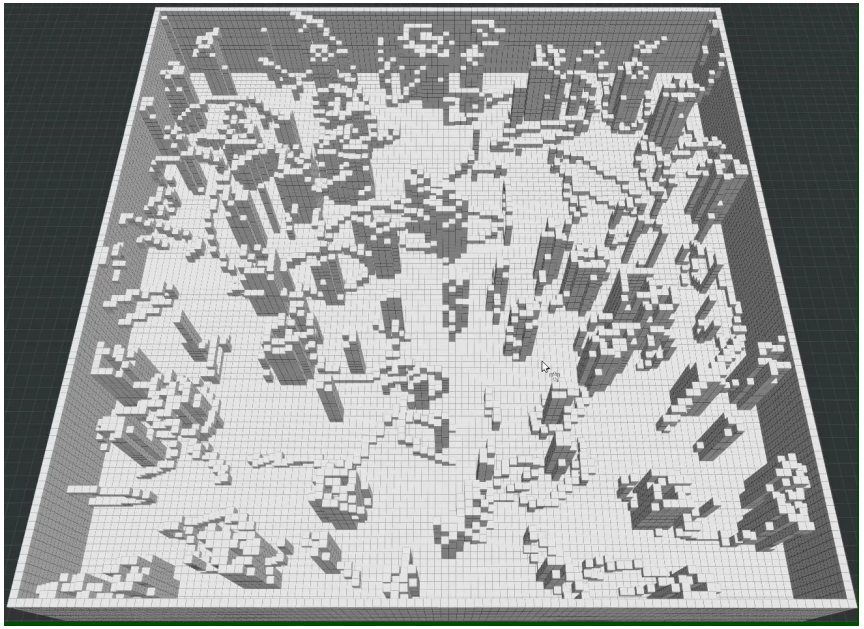
Input: $\mathcal{M}$,x_{init},x_{goal}
Result: A path Γ from x_{init} to x_{goal}
$\mathcal{T}$.init();
for $i = 1$ *to* n **do**
  $x_{rand} \leftarrow Sample(\mathcal{M})$;
  $x_{near} \leftarrow Near(x_{rand}, \mathcal{T})$;
  $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$;
  if *CollisionFree*(x_{new})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

RRT:



RRT*:



https://github.com/ZJU-FAST-Lab/sampling-based-path-finding

Rapidly-exploring Random Tree* (RRT*)

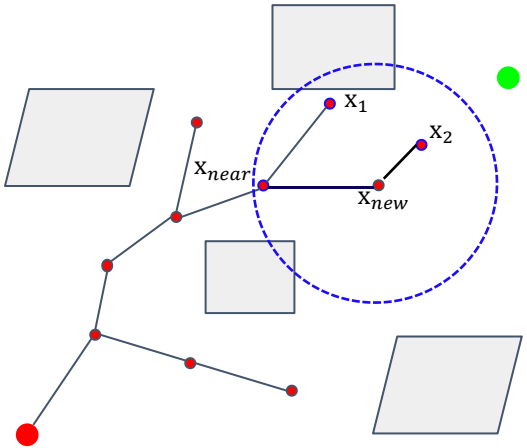
About the query range[1]

$$range = min\left\{\gamma \cdot \left(\frac{log(card(V))}{card(V)}\right)^{\frac{1}{d}}, \eta\right\}$$

To assure AO:

$$\gamma > \left(2\left(1+\frac{1}{d}\right)\right)^{\frac{1}{d}}\left(\frac{\mu(X_{free})}{\xi_d}\right)^{\frac{1}{d}}$$

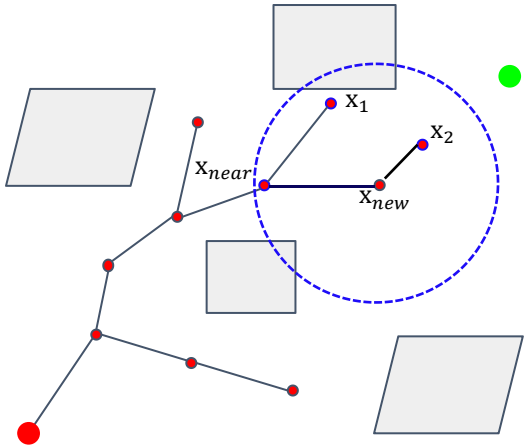
Empirically, for low dimensional planning, we can set a constant range a bit larger than the steer distance.



[1] Revised in Kiril Solovey et al., “Revisiting the Asymptotic Optimality of RRT*", ICRA, 2020

Practical implementation

- Bias Sampling
  Sample biasing toward the goal
- Sample Rejection
  Reject samples with g + h > c*
- Branch-and-bound (Tree Pruning)
  Prune the non-promising sub trees to reduce neighbor query cost.
- Graph Sparsify
  Reject samples by resolution. Introduce near optimality.



Practical implementation

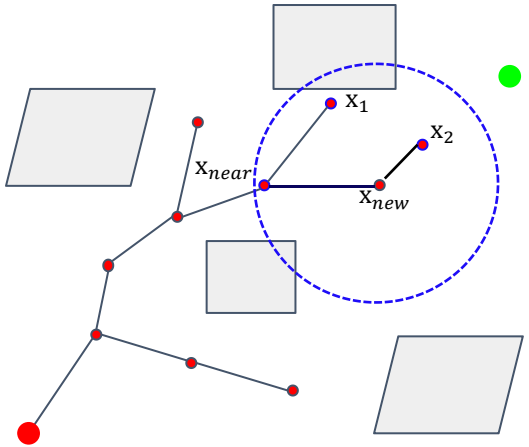
- Neighbor Query
  k-nearest or range near; Approximate nearest neighbor;
  Range tree for different dimension.

- Delay Collision Check
  Sort the neighbours by potential cost-to-come values.
  Check collisions in order and stop once a collision-free edge is found.

- Bi-directional search

- Conditional Rewire
  Rewire Until the first solution is found.

- Data Re-use
  Store the collision-checking results for ChooseParent and Rewire (only valid for symmetric cost).



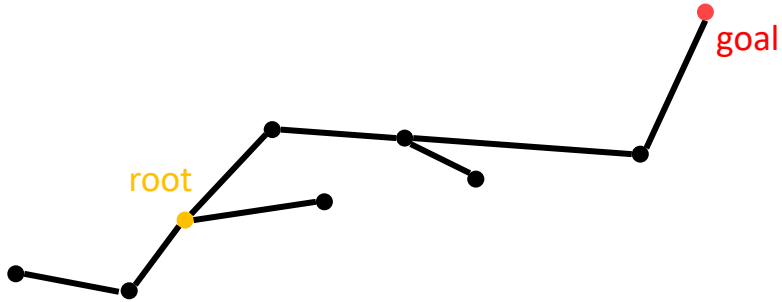
Accelerate Convergence

RRT#

Flaws in the exploitation of RRT*

Over-exploitation

No need to “Rewire” non-promising vertexes.

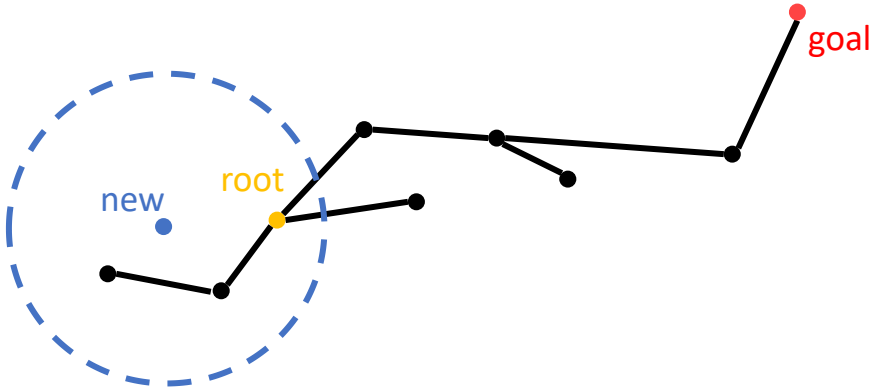


RRT#

Flaws in the exploitation of RRT*

Over-exploitation

No need to “Rewire” non-promising vertexes.

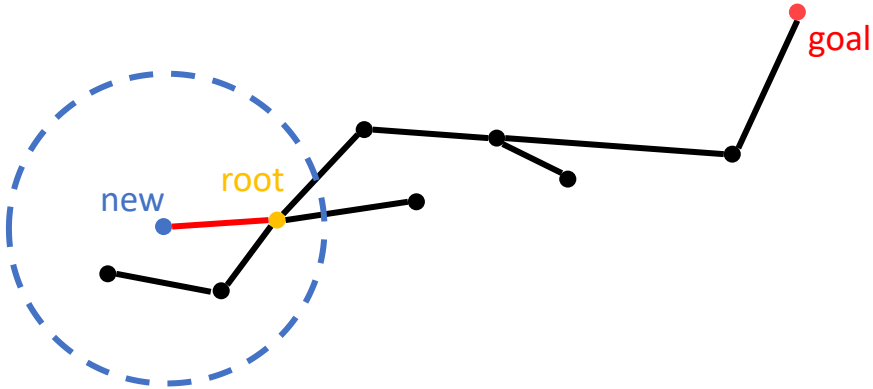


RRT#

Flaws in the exploitation of RRT*

Over-exploitation

No need to “Rewire” non-promising vertexes.

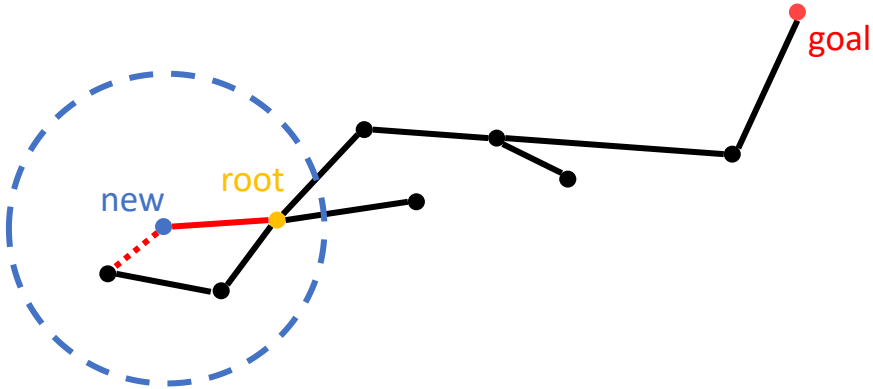


RRT#

Flaws in the exploitation of RRT*

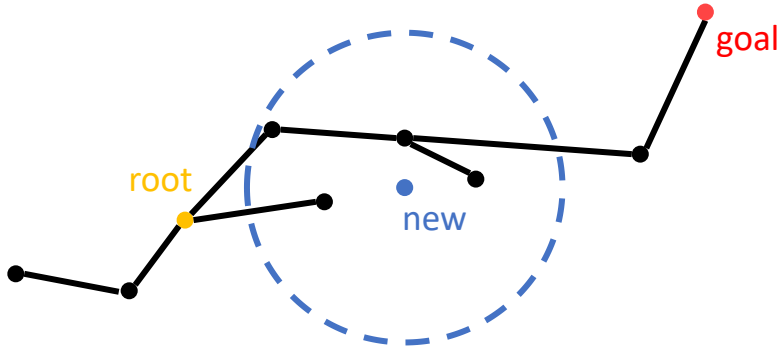
Over-exploitation

No need to “Rewire” non-promising vertexes.



RRT#

Flaws in the exploitation of RRT*



Over-exploitation

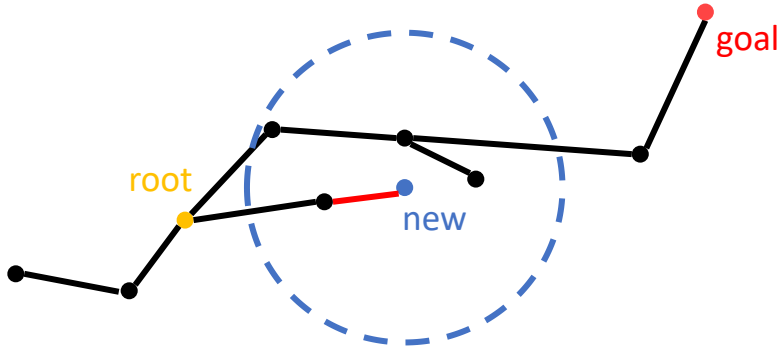
No need to “Rewire” non-promising vertexes.

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

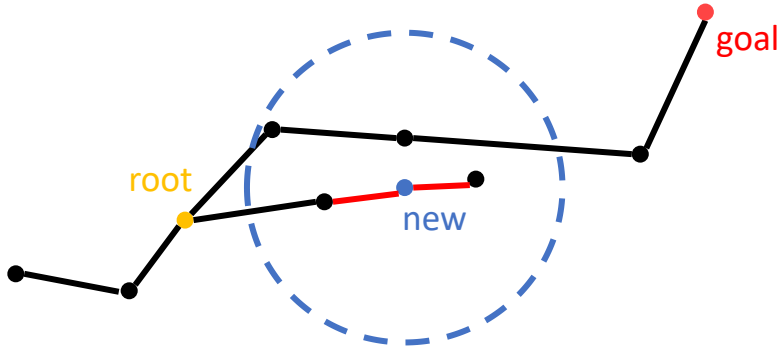
No need to “Rewire” non-promising vertexes.

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

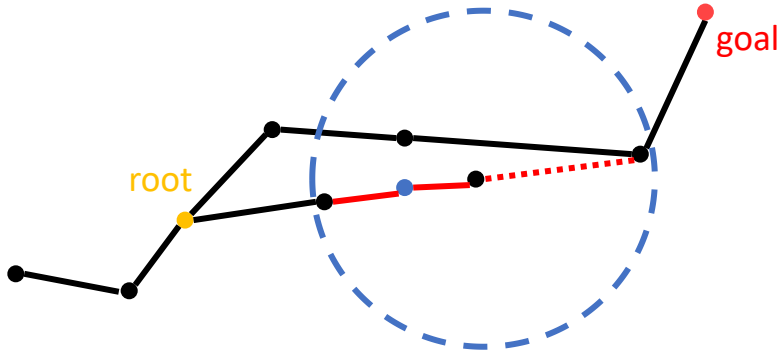
No need to “Rewire” non-promising vertexes.

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

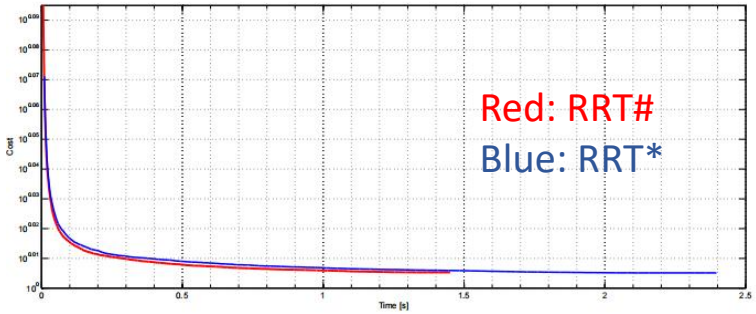
No need to “Rewire” non-promising vertexes.

Under-exploitation

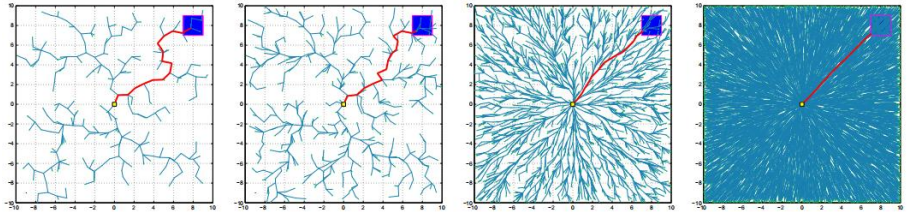
- “Rewire” only happens after a new node is add to the spanning tree and only works on its “Near” nodes.
- It does not offer any guarantees that the interim path at any intermediate iteration is optimal.

RRT#

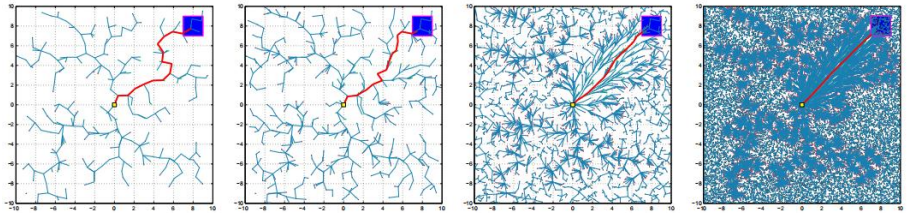
- An improvement of RRT*
- Asymptotically optimal (under conditions)



RRT*

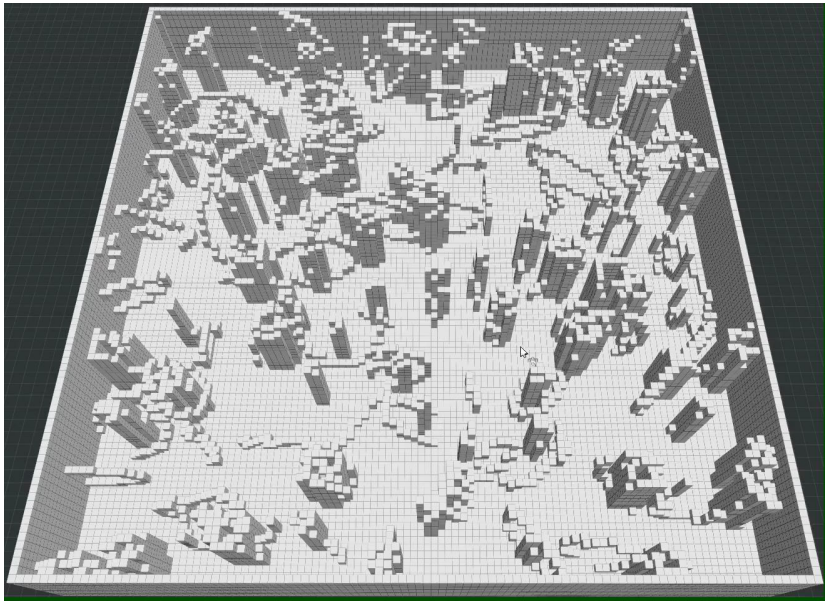


RRT#

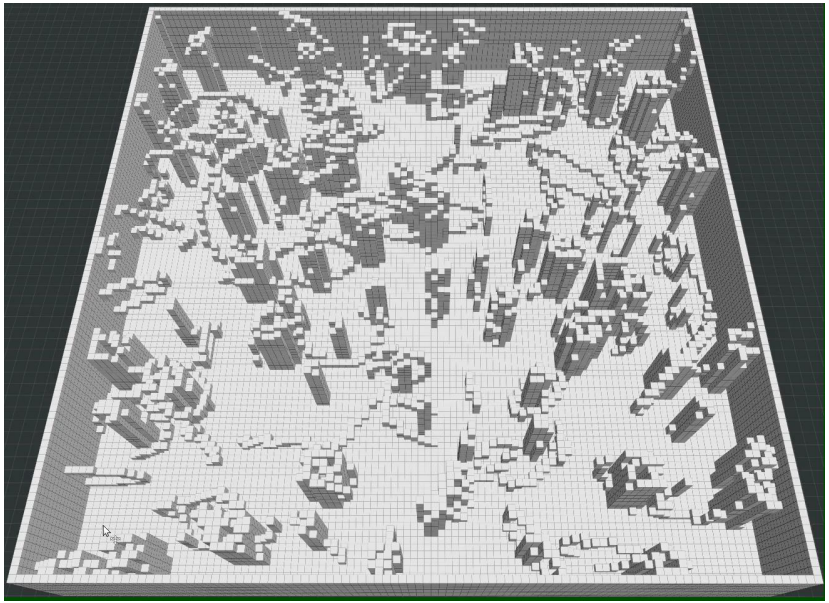


RRT#

RRT*



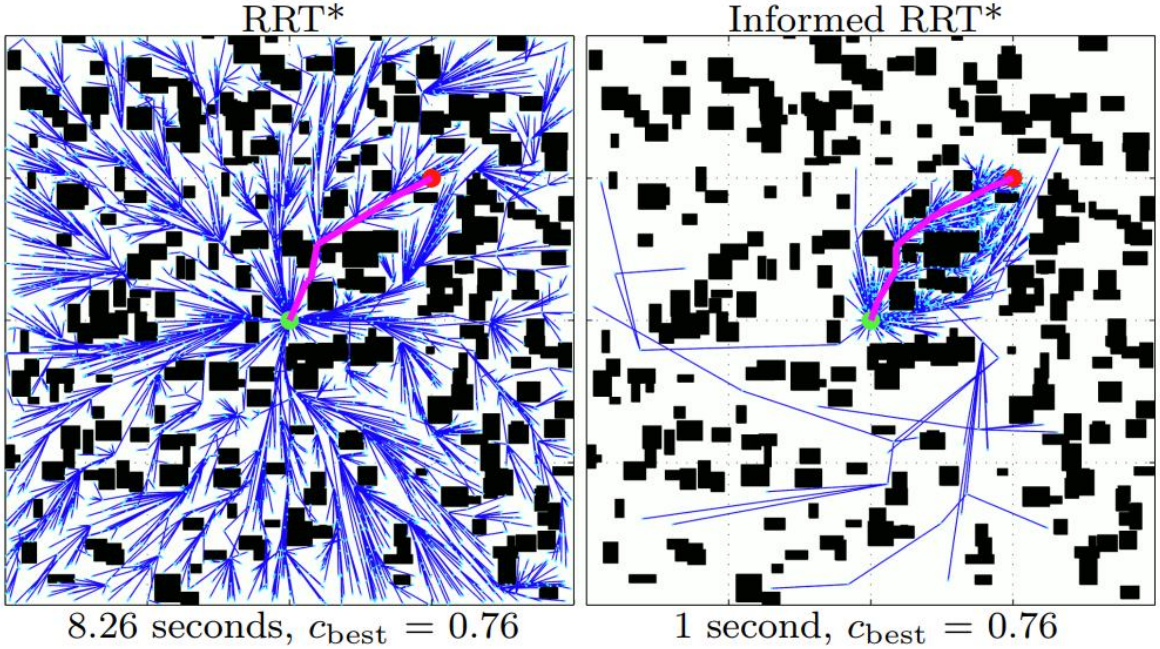
RRT#



https://github.com/ZJU-FAST-Lab/sampling-based-path-finding

Informed Sampling

Improve sampling efficiency

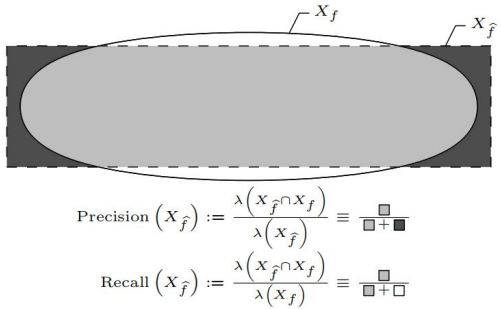


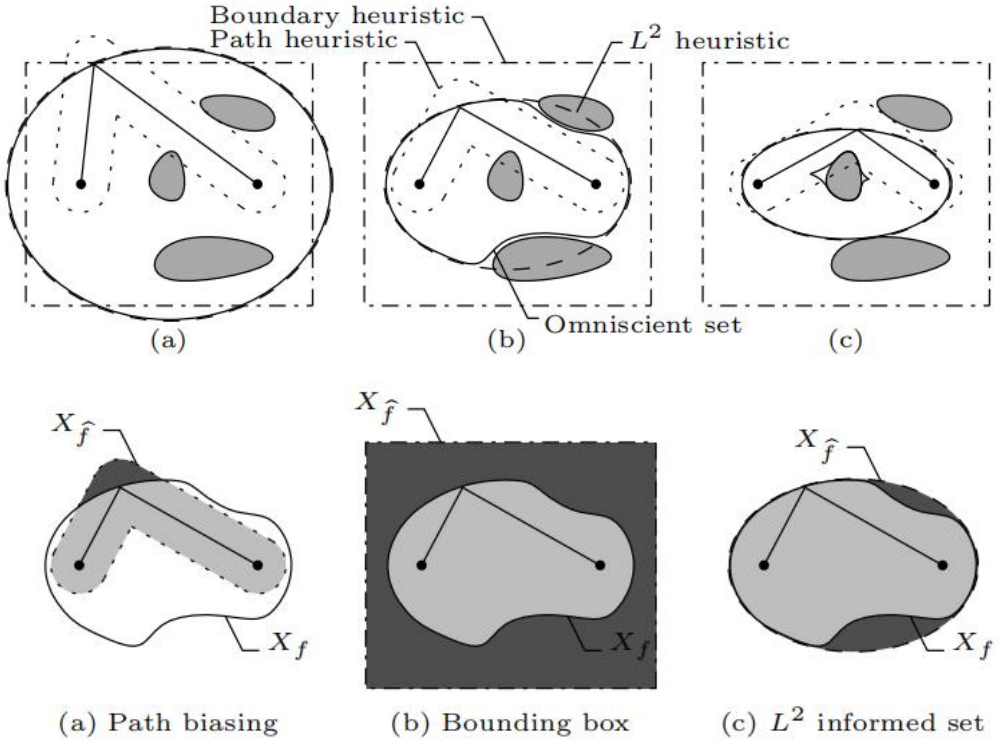
Informed Sampling

Informed sets:

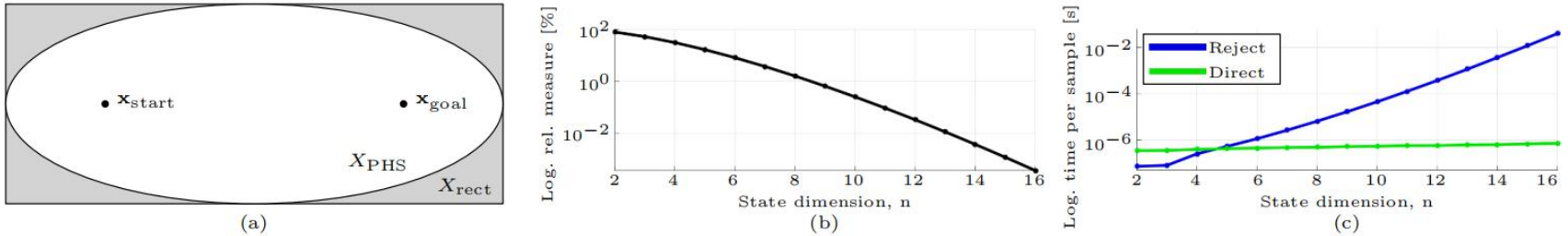
estimates of the omniscient set

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





Informed Sampling



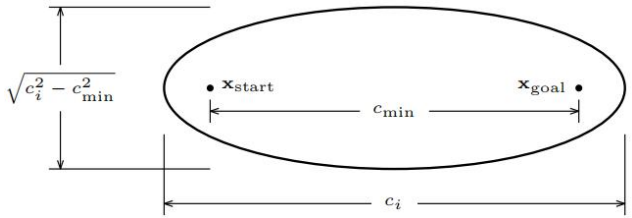
Direct Sampling in the L2 Informed set

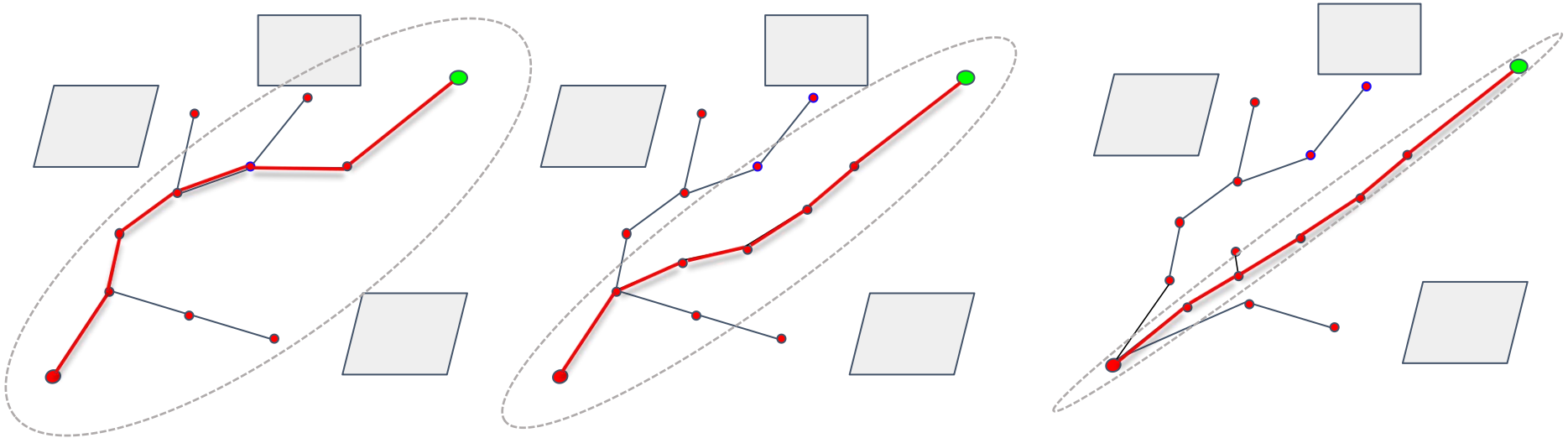
VS

Reject sampling in the bounding rectangular

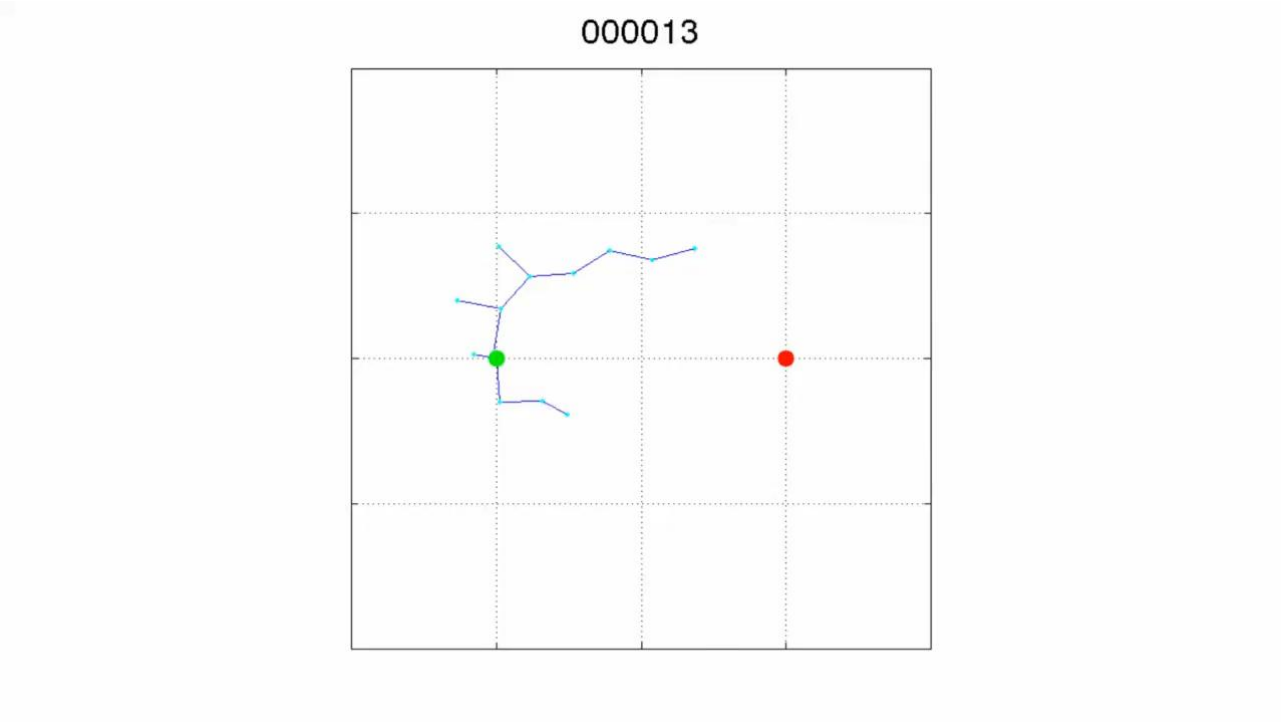
Informed Sampling

L2 Informed set



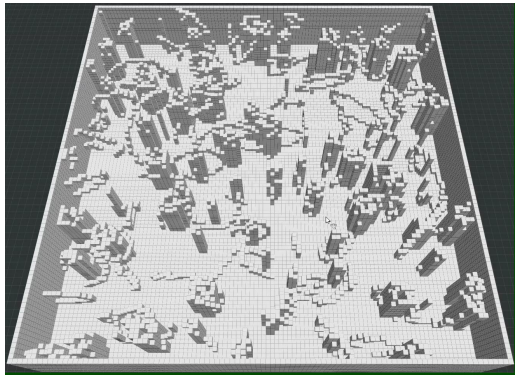
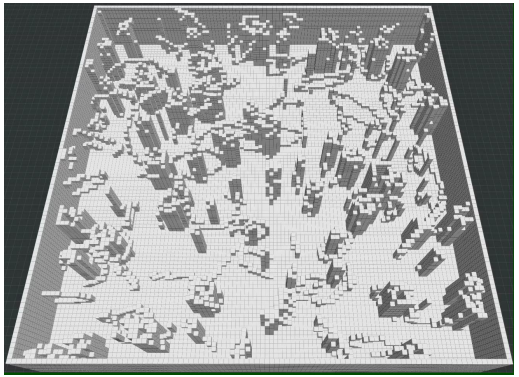
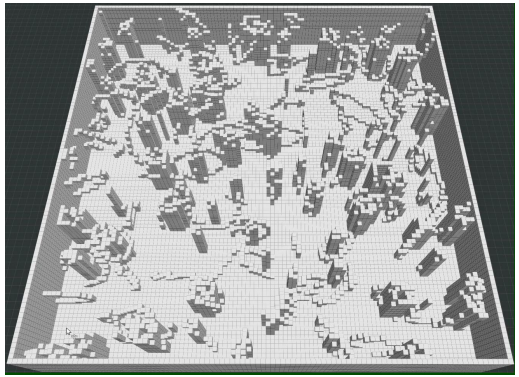
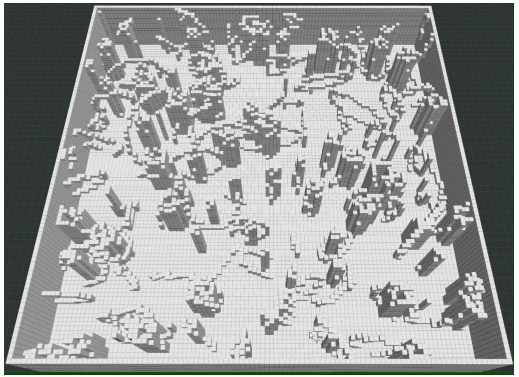


Informed Sampling



Cons: Can hardly apply to other cost except L2 norm cost (Euclidean distance).

Informed Sampling

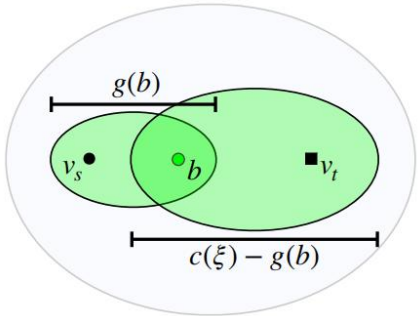
	random uniform sampling	L2-informed sampling
RRT*		
RRT#		

https://github.com/ZJU-FAST-Lab/sampling-based-path-finding

GuILD *Guided Incremental Local Densification*

Although the iteration may not have found a shorter path to the goal, new shorter paths to other vertices in the search tree can immediately improve the sampling strategy.

Ill informed set cases

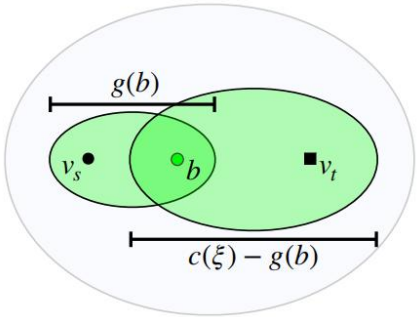


Local Subsets (green):

Defined by

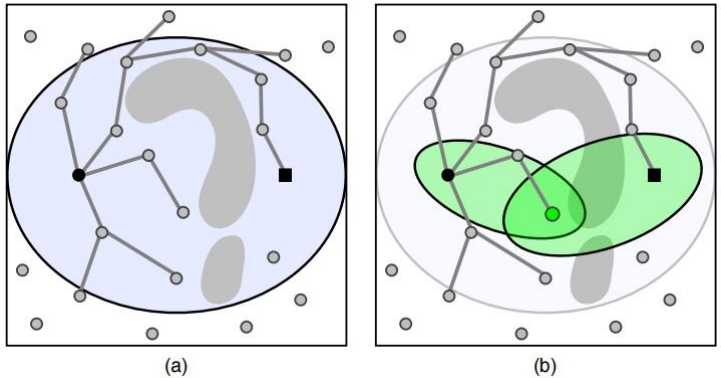
- A beacon node *b;*
- Its cost-to-come on the search tree $g(b)$;
- The current best solution cost $c(\xi)$.

GuILD *Guided Incremental Local Densification*



Properties of Local Subsets

- LSs are subsets of the IS;
- The measure of LSs is upper-bounded by that of the IS.

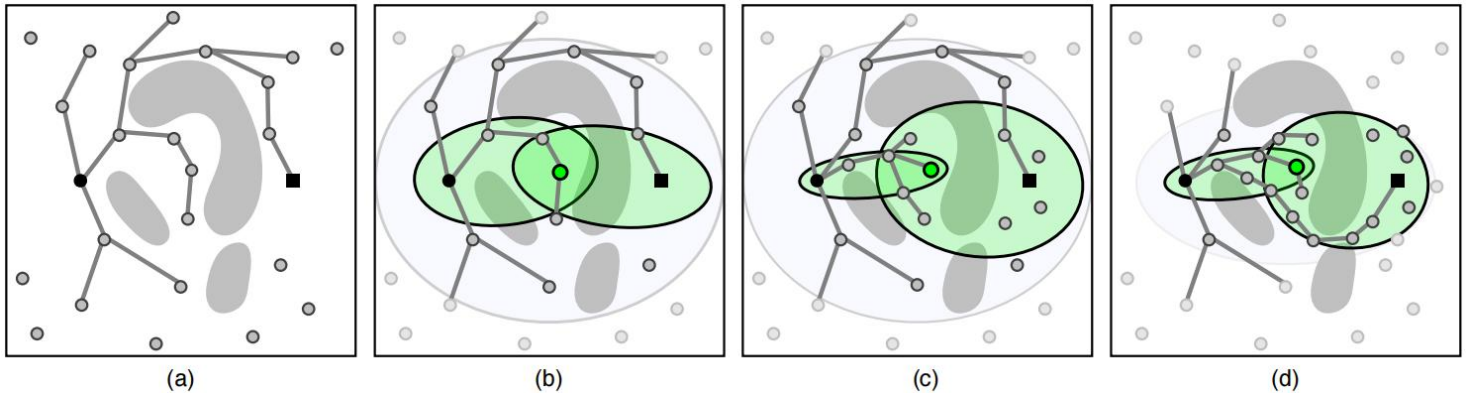


Sampling in the L2 Informed Set

vs

Sampling in the Local Subsets

GuILD *Guided Incremental Local Densification*

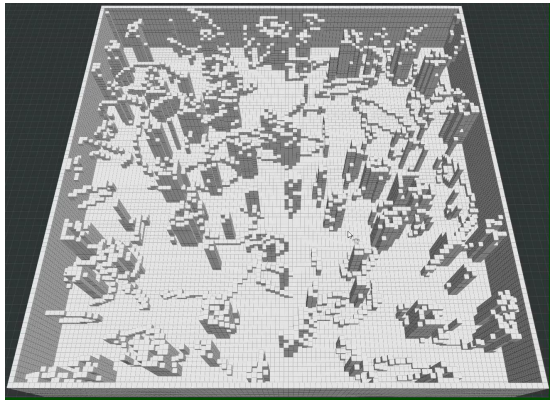
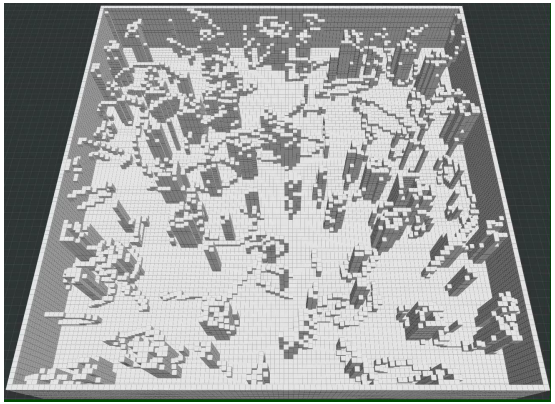
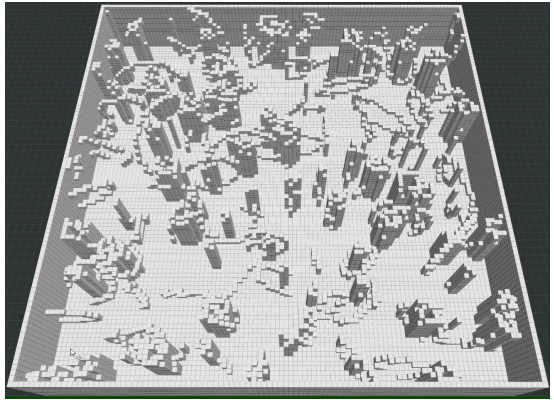
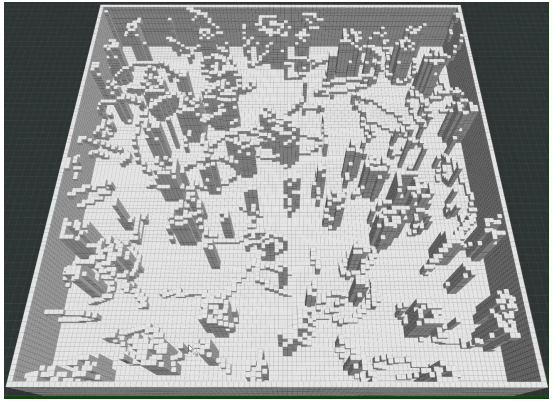


The Informed Set is unchanged. However, GuILD leverages the improved cost-to-come in the search tree to update the Local Subsets.

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.

There is a larger path length budget that can be expended between the beacon and target that could still yield a shorter path overall.

GuILD *Guided Incremental Local Densification*

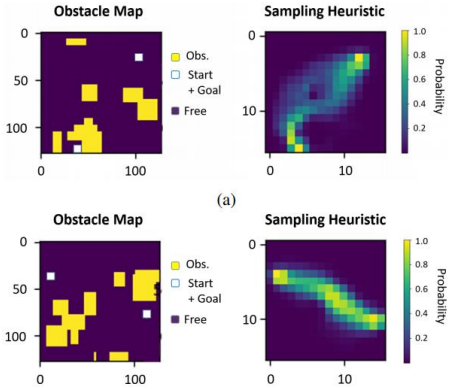
	random uniform sampling	GuILD sampling
RRT*		
RRT#		

https://github.com/ZJU-FAST-Lab/sampling-based-path-finding

Development Trend

Sampling Strategy

- Deterministic/Random;
- Batch/Incremental;
- Informed sampling;
- *Learned sampling;*

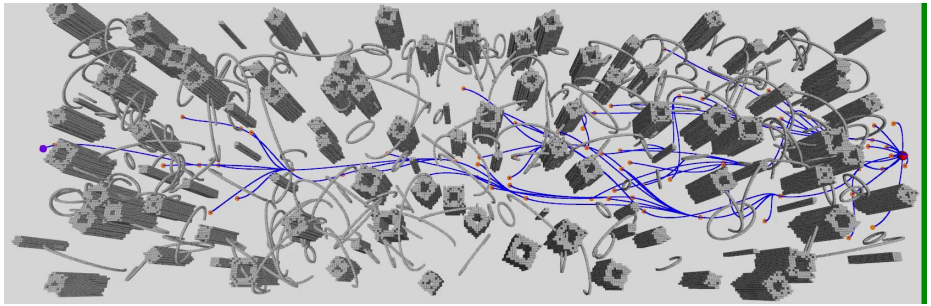


Hybirid Exploration-Exploitation Scheme

- Introduce local optimization.

Apply to Kinodynamic Systems

- The original purpose



Libraries off-the-shelf

- Open Motion Planning Library (OMPL) [1]
- Moveit with ROS [2]
- Fast-lab [3]
- …

[1] https://ompl.kavrakilab.org/
[2] https://moveit.ros.org/
[3] https://github.com/ZJU-FAST-Lab/sampling-based-path-finding

Assignments

1) Implement RRT in Matlab；

2) Implement RRT* in C++；

The framework is ready, only coding for ChooseParent and Rewire are required.

```
// TODO Choose a parent according to potential cost-from-start values
// ! Hints:
// !  1. Use map_ptr ->isSegmentValid(p1, p2) to check line edge validity;
// !  2. Default parent is [nearest node];
// !  3. Store your chosen parent-node-pointer, the according cost-from-parent and cost-from-start
// !       in [min_node], [cost_from_p], and [min_dist_from_start], respectively;
// !  4. [Optional] You can sort the potential parents first in increasing order by cost-from-start value;
// !  5. [Optional] You can store the collison-checking results for later usage in the Rewire procedure.
// ! Implement your own code inside the following loop
for (auto &curr_node : neighbour_nodes)
{

}
// ! Implement your own code inside the above loop
```

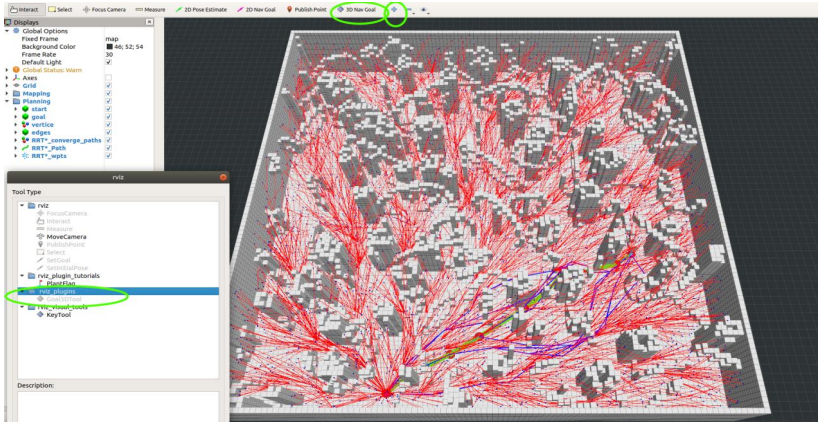
```
/* 3.rewire */
// TODO Rewire according to potential cost-from-start values
// ! Hints:
// !  1. Use map_ptr ->isSegmentValid(p1, p2) to check line edge validity;
// !  2. Use changeNodeParent(node, parent, cost_from_parent) to change a node's parent;
// !  3. the variable [new_node] is the pointer of X_new;
// !  4. [Optional] You can test whether the node is promising before checking edge collison.
// ! Implement your own code between the dash lines [----------] in the following loop
for (auto &curr_node : neighbour_nodes)
{
  double best_cost_before_rewire = goal_node_->cost_from_start;
  // ! ------------------------------------

  // ! ------------------------------------
  if (best_cost_before_rewire > goal_node_->cost_from_start)
  {
    vector<Eigen::Vector3d> curr_best_path;
    fillPath(goal_node_, curr_best_path);
    path_list_.emplace_back(curr_best_path);
    solution_cost_time_pair_list_.emplace_back(goal_node_->cost_from_start, (ros::Time::now() - rrt_start_time).toSec());
  }
}
```

Grading

Qualified: Finish 1)

Good: Finish 1) and 2)

Excellent: Implement Informed-RRT* based on 2)



Expected result

Reference

- L. E. Kavraki, P. Svestka, J. -C. Latombe and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," in *IEEE Transactions on Robotics and Automation*, 1996
- S. M. LaValle and J. J. Kuffner, Jr., "Randomized Kinodynamic Planning," *The International Journal of Robotics Research*, 2001
- S. Karaman and E. Frazzoli, "Sampling-Based Algorithms for Optimal Motion Planning," *The International Journal of Robotics Research*, 2011
- Sniedovich, Moshe. "Dijkstra's algorithm revisited: the dynamic programming connexion." *Control and Cybernetics*, 2006
- O. Arslan and P. Tsiotras, "Use of relaxation methods in sampling-based algorithms for optimal motion planning," *IEEE International Conference on Robotics and Automation*, 2013
- J. D. Gammell, T. D. Barfoot and S. S. Srinivasa, "Informed Sampling for Asymptotically Optimal Path Planning," in *IEEE Transactions on Robotics*, 2018
- Aditya Mandalika, Rosario Scalise, Brian Hou, Sanjiban Choudhury, Siddhartha S. Srinivasa, "Guided Incremental Local Densification for Accelerated Sampling-based Motion Planning", in *arXiv:2104.05037,* 2021

Thanks for Listening!