EE405A
Local Map Generation & Motion Planning

School of Electrical Engineering
KAIST

Contents

Notice

- **Final Project Scenario**
  - Autonomous Racing → **Autonomous Exploration**
  - Find **numbers** or **QR codes** as much as possible while exploring corridor environments (Segmentation → **Object detection**-based perception module)
  - (Optional) Utilizing GPT for final project mission

- **Assignment of this week**
  - The assignment of this week will be provided next week (~ Tuesday).
  - Implementing map generation & path planning + Deploying in our gazebo-based simulation.

    (Week 3, https://github.com/Guri-cccc/EE405A-2023-F1-simulation)

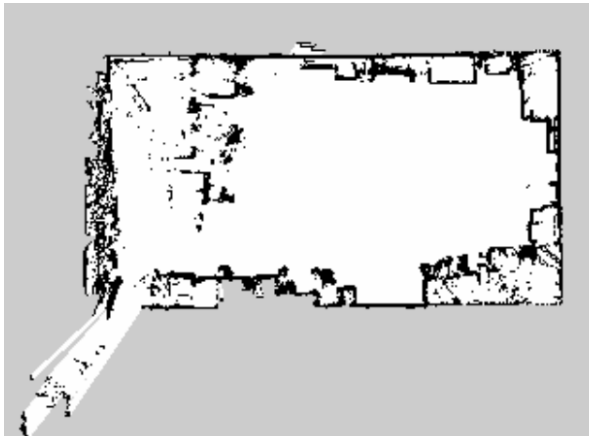
Local Map Generation

Map Generation for Motion Planning
Occupancy Grid Map

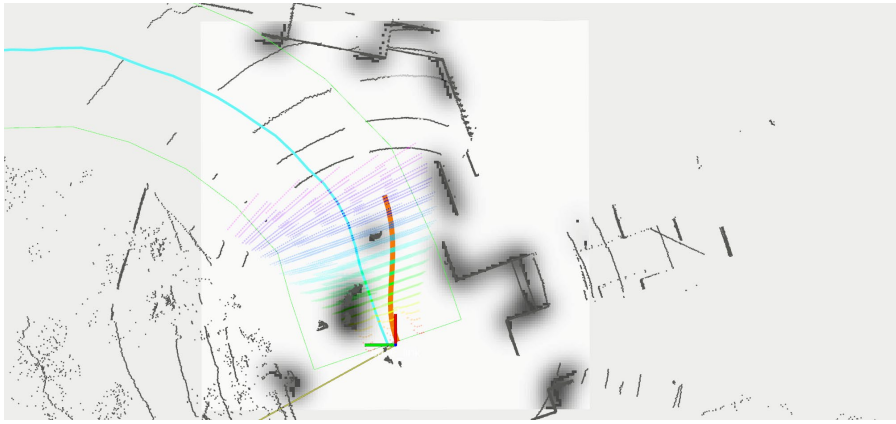
Map Generation For Motion Planning

➢ Map Generation for Motion Planning

- ❑ To navigate in a complex area, a robot should take into account the surrounding environment.
- ❑ As shown in the right side figure, collision-free path can be computed based on the grid map.
- ❑ In order to come up with dynamic obstacles, we need to generate a dynamic occupancy grid map in real-time.
- ❑ An advantage of the occupancy grid map is that it can represent a possibility for the collision.



Generated static map using Hector SLAM



Example of dynamic occupancy grid map – We implemented motion planning algorithm for the obstacle avoidance and we generated Gaussian distributed occupancy grid map in real-time.

Occupancy Grid Map

➢ Occupied grid map (Basic)

- ❑ We can generate a probabilistic occupancy grid map considering the sensor model(Sonar, Laser, Camera ..)
- ❑ Grid map can be generated by accumulating a scan to the corresponding grid cell.

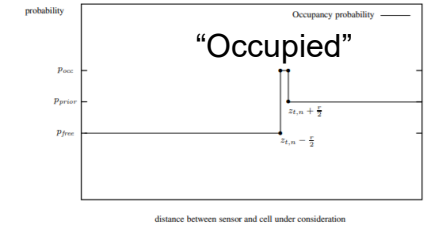


Fig. 2.3. Sensor model for a laser range finder. It depicts the probability that a cell is occupied depending on the distance of that cell from the laser sensor.

Laser sensor model

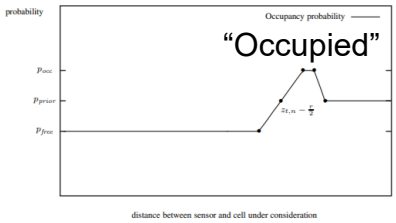
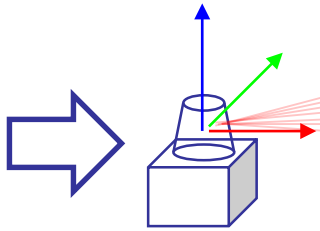


Fig. 2.4. Probability that a cell on the optical axis of the sensor is occupied depending on the distance of that cell from the sensor.

Sonar sensor model



0	5	0	0
0	2	0	0
0	2	0	0
0	0	0	0

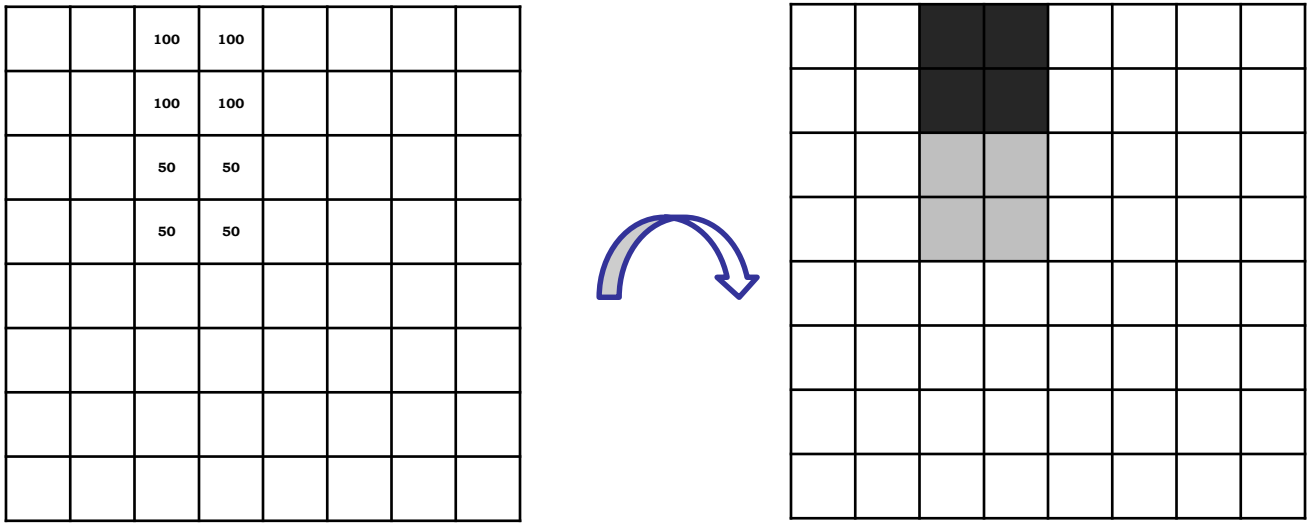
Accumulated scan to the grid cell(origin)

Stachniss, Cyrill. *Robotic mapping and exploration*. Vol. 55. Springer, 2009.

Occupancy Grid Map

➢ Occupied grid map (Basic)

- ❑ You can make a basic occupancy grid map using obstacle height information using 3D depth pointcloud from RealSense D435 camera.
- ❑ Firstly, to detect the position of obstacles, you can use depth point cloud to generate a height map.
- ❑ Using the generated height map, compute the cost map based on the height value.

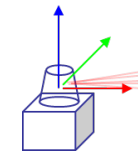


Heightmap Costmap

Occupancy Grid Map

➢ Occupied grid map (Advanced)

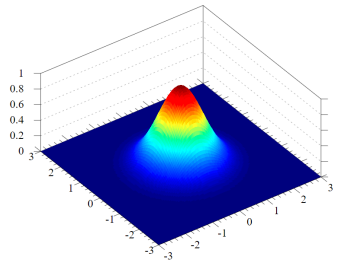
- ❑ You can give padding for the collision buffer. Normally we set this padding value considering the size of the vehicle(robot).
- ❑ Personally, we gave a Gaussian distribution for the padding to regard it as a collision possibility. You can give a covariance value considering the size of the vehicle.



2-D laser

1	2	1	0
1	5	1	0
1	2	1	0
1	1	1	0

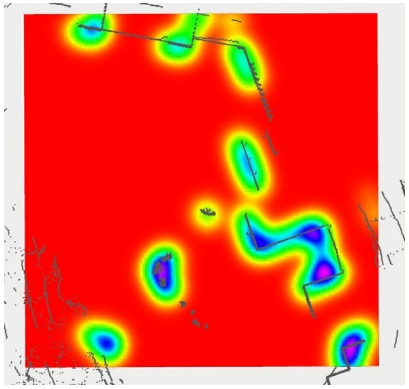
Constant padding
(padding size:1)



2-D Gaussian Distribution

0	5	0	0
0	2	0	0
0	2	0	0
0	0	0	0

Accumulated scan to the grid cell(origin)



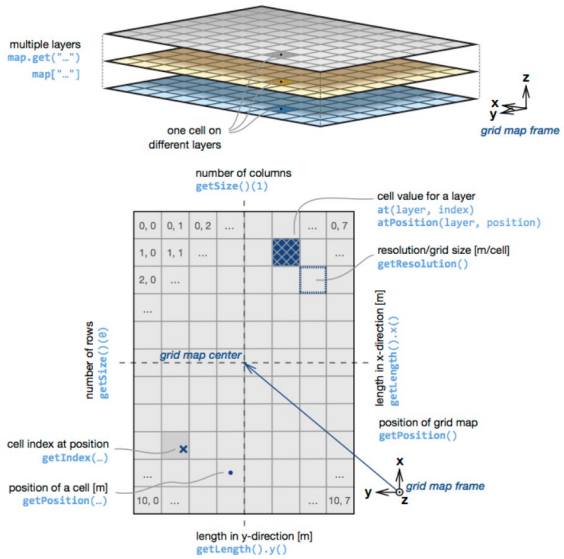
Gaussian dist. padded grid map

Stachniss, Cyrill. *Robotic mapping and exploration*. Vol. 55. Springer, 2009.

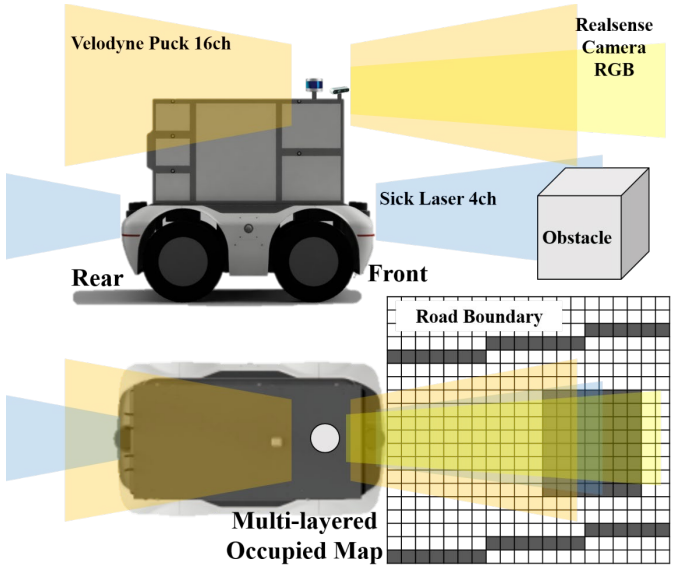
Occupancy Grid Map

➢ Occupied grid map (Advanced)

- ❑ If you use multiple sensors such as 3-D Laser, camera, 2-D Laser sensors, you can integrate the data from each sensor using a universal grid map library: https://github.com/anybotics/grid_map



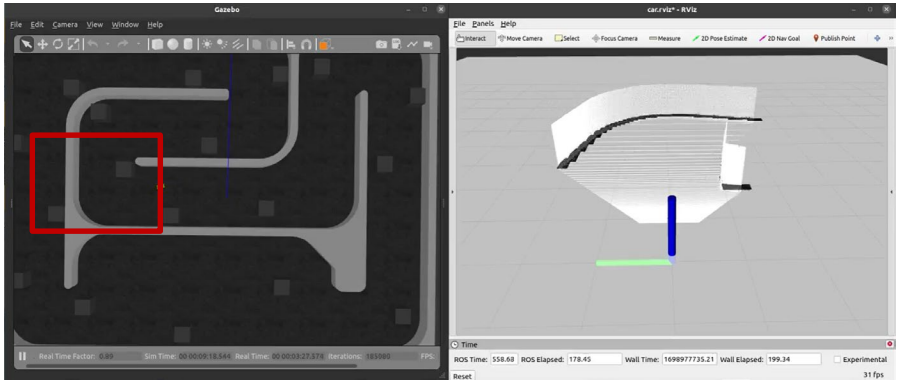
Fankhauser, Péter, and Marco Hutter. "A universal grid map library: Implementation and use case for rough terrain navigation." *Robot Operating System (ROS)*. Springer, Cham, 2016. 99-120.



Example of multi-layered occupied grid map –
From our lab project : cooperative delivery robot for postman

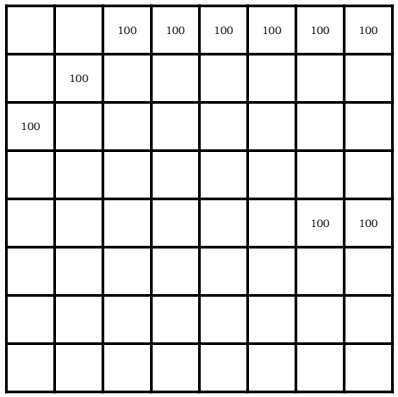
Occupancy Grid Map

- **Occupied grid map using Point Cloud data from D435i**
  - Our camera sensor (Realsense D435i) provides point cloud data based on depth image.
  - We can utilize the 3D point cloud to generate real-time local map.
  - Height-based scheme can be used to represent traversable / non-traversable region around the ego vehicle.

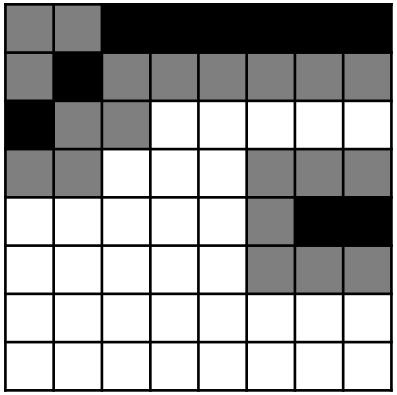


Gazebo simulation

Point cloud



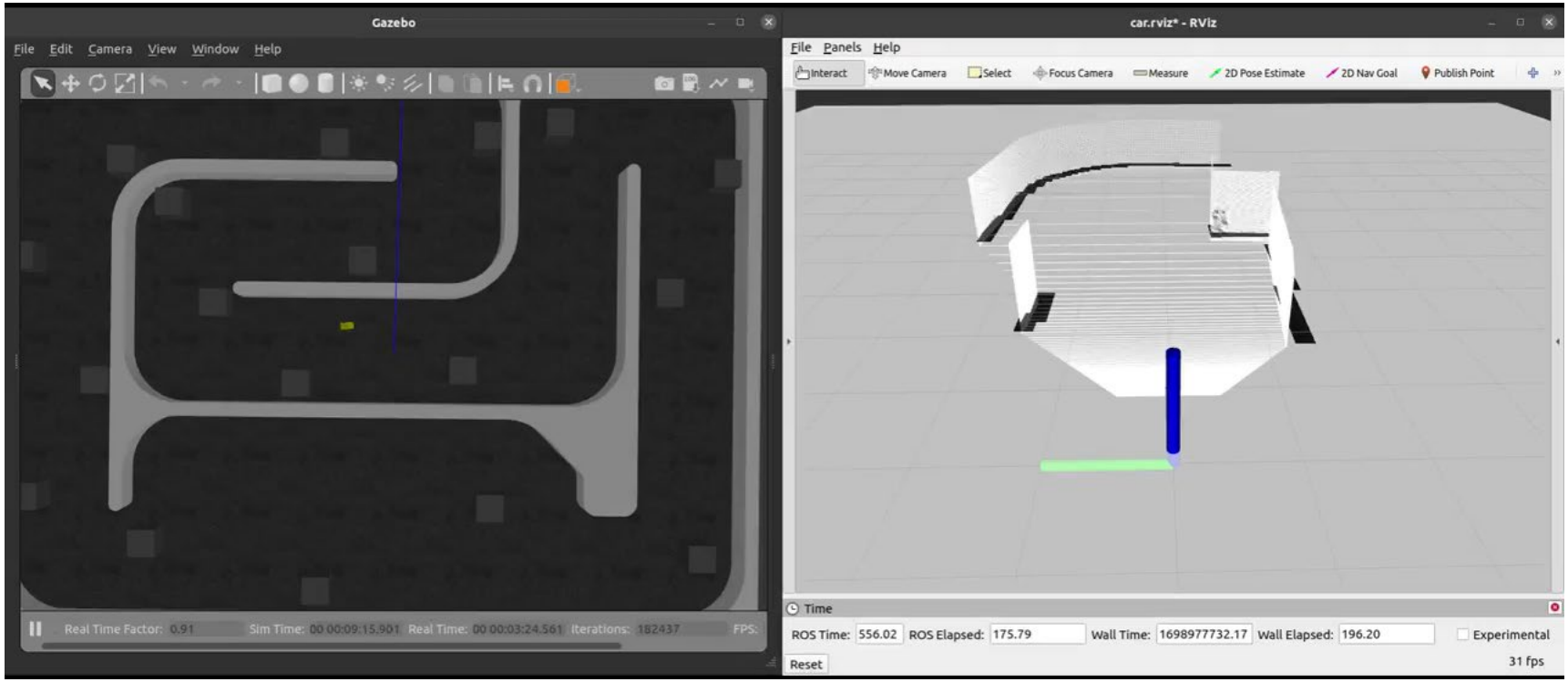
Height map



Cost map

Occupancy Grid Map

- **Occupied grid map using Point Cloud data from D435i**
  - Gazebo simulation: Week 3, https://github.com/Guri-cccc/EE405A-2023-F1-simulation.
  - Local cost map generation will be provided as an assignment in next week.



Gazebo simulation

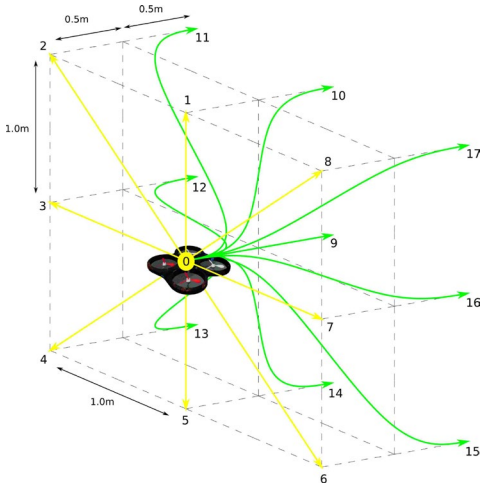
Real-time local cost map generation

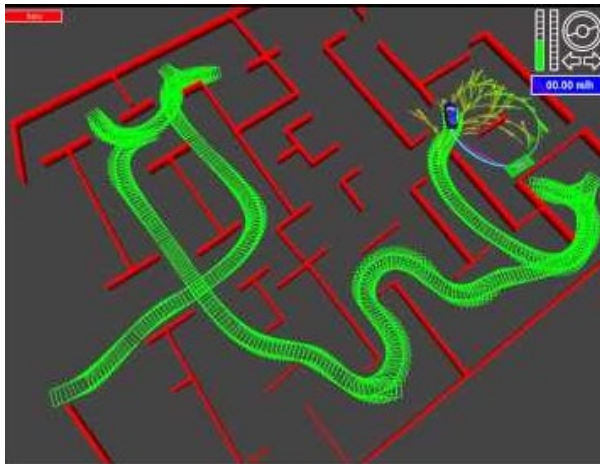
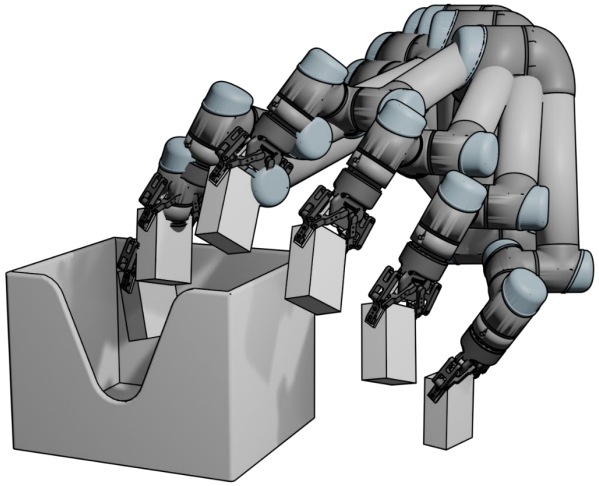
Motion Planning

Motion Planning Methods
Motion Primitive-based Algorithm

Motion Planning Methods

- **Motion planning**
  - Motion planning, also path planning is a computational problem to find a sequence of valid configurations that moves the object from the source to destination.
  - Several major categories for motion planning will be introduced in this lecture:
    - ✓ Graph Search-based motion planning
    - ✓ Random Sampling-based motion planning
    - ✓ Motion Primitives-based planning

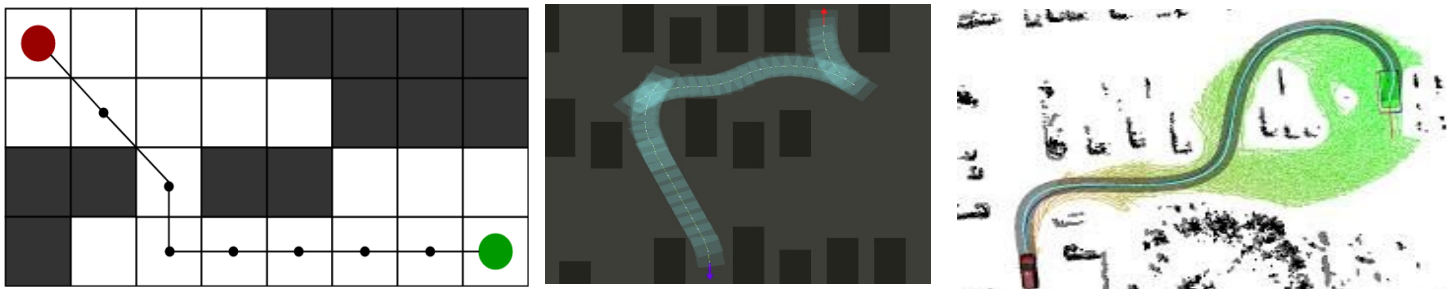


Motion planning in various robotics applications

Motion Planning Methods

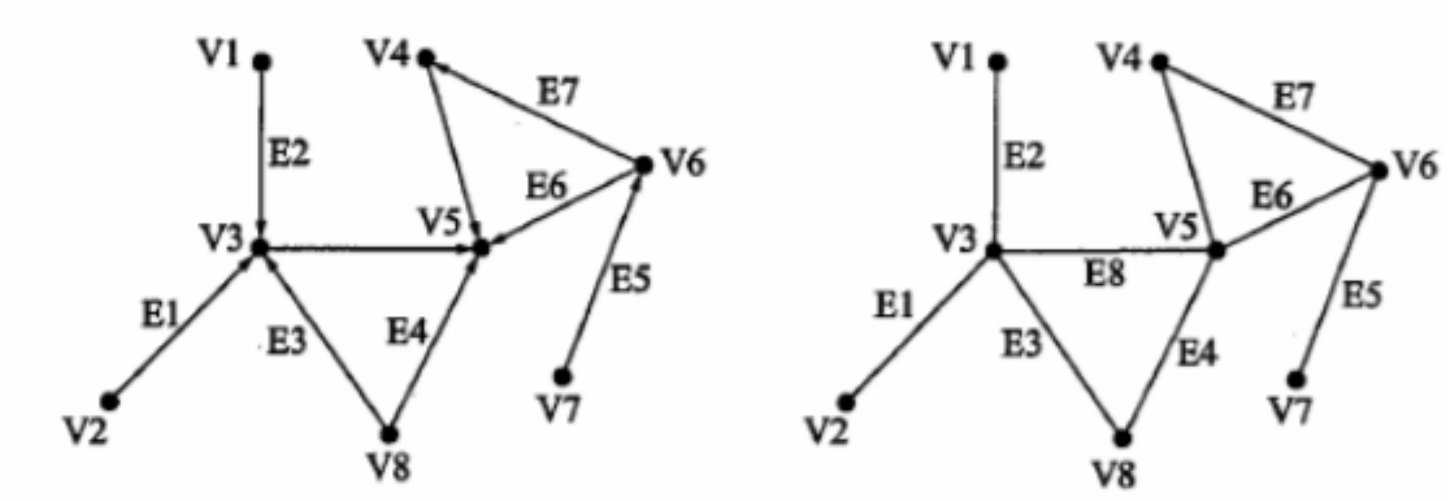
- **Graph search-based motion planning**
  - It represents the environment in a discrete grid (discrete grid can be represented as a graph)
  - Grid-based approaches overlay a grid on configuration space, and assume each configuration is identified with a grid point.
  - At each grid point, the robot is allowed to move to adjacent grid points as long as the line between them is collision-free.
  - This discretizes the set of actions, and search algorithms (like A*) are used to find a path from the start to the goal.



Examples of the grid-based motion planning:
A* and Hybrid A* algorithm

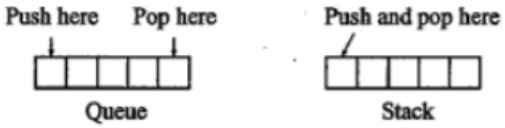
Motion Planning Methods

- **Graph search--based motion planning**
  - Collection of edges and nodes (vertices)

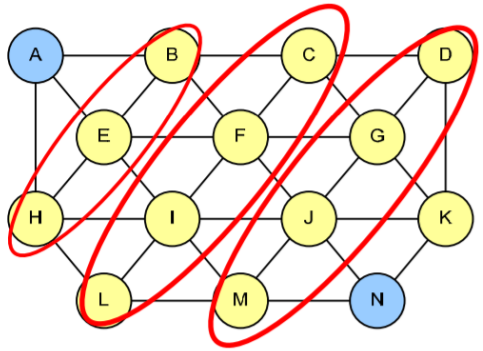


Motion Planning Methods

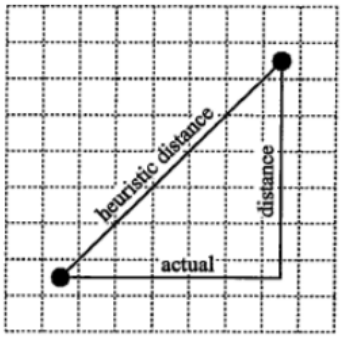
- **Graph search-based motion planning**
  - Uniform Search
    - Use no information obtained from the environment
    - Blind search: BFS(*Breadth-First Search*, width), DFS(*Depth-First Search*)

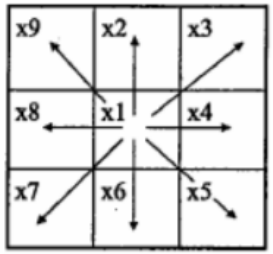


  - Informed Search: A*
    - Use evaluation function
    - Usually more efficient
    - Heuristic search: A*, D*, etc.



An example of BFS graph search from A to N





$c(x1, x2) = 1$
$c(x1, x9) = 1.4$

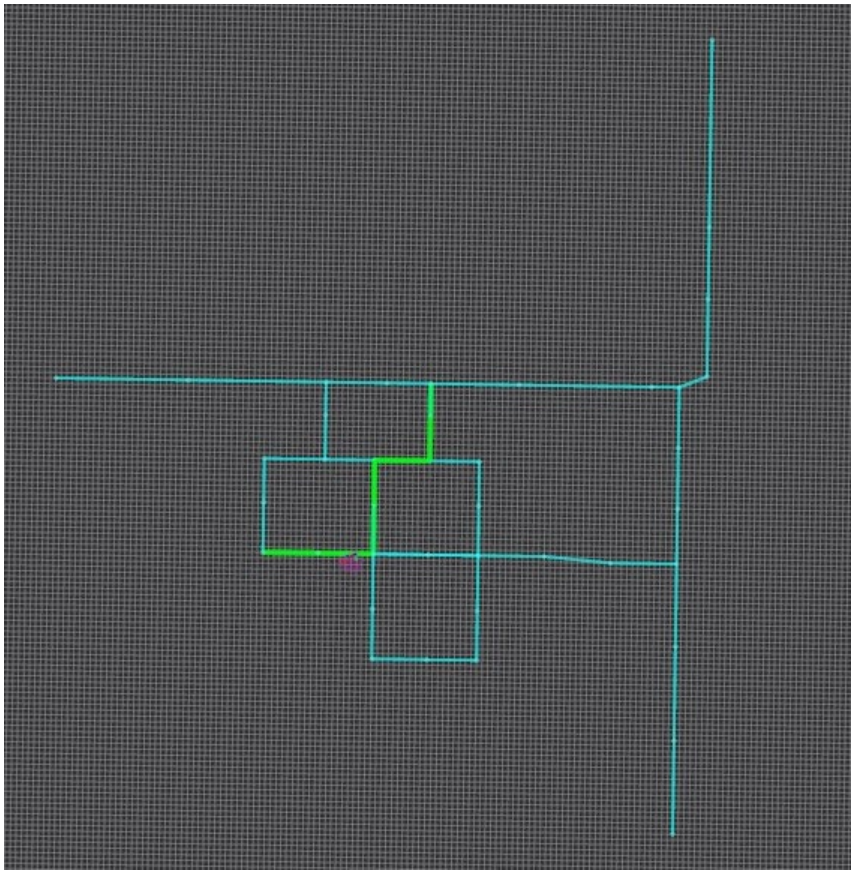
$c(x1, x8) = 10000$, if x8 is in obstacle, x1 is a free cell

$c(x1,x9) = 10000.4$, if x9 is in obstacle, x1 is a free cell

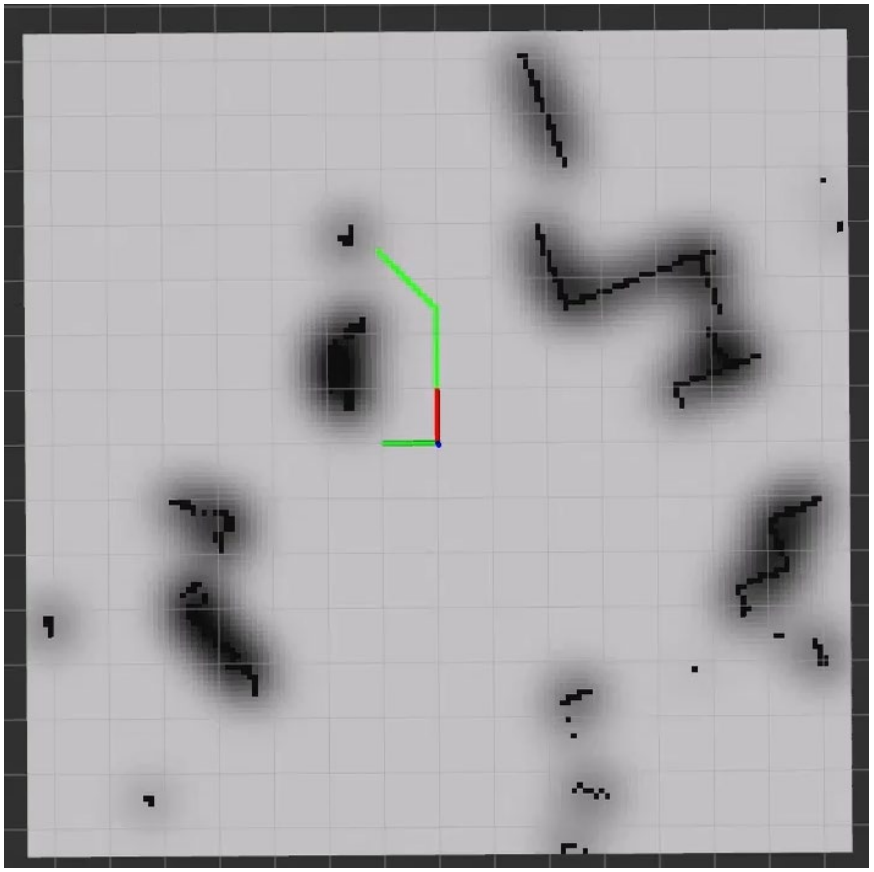
Heuristic-based A* algrorithm

A* Algorithm

- **Example of implementing A*-based planner**



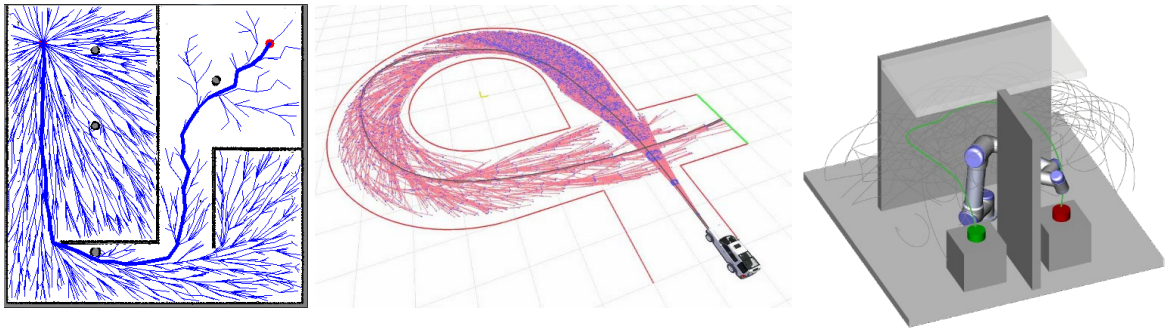
Roadmap-based global path planning



Grid map-based local path planning

Motion Planning Methods

- **Random sampling-based motion planning**
  - Random sampling can overcome the disadvantage of the fixed graph discretization that other planner search only over the set of paths.
  - Instead of grid search, nodes are sampled in the configuration space *C* for finding collision-free path in probabilistic manner (e.g., Rapidly exploring Random Tree (RRT)).
  - Since their running time is not (explicitly) exponentially dependent on the dimension of the *C*, they works well for high-dimensional *C* (i.e., 3 dimentional environment).



Examples of the sampling-based motion planning: RRT and RRT*

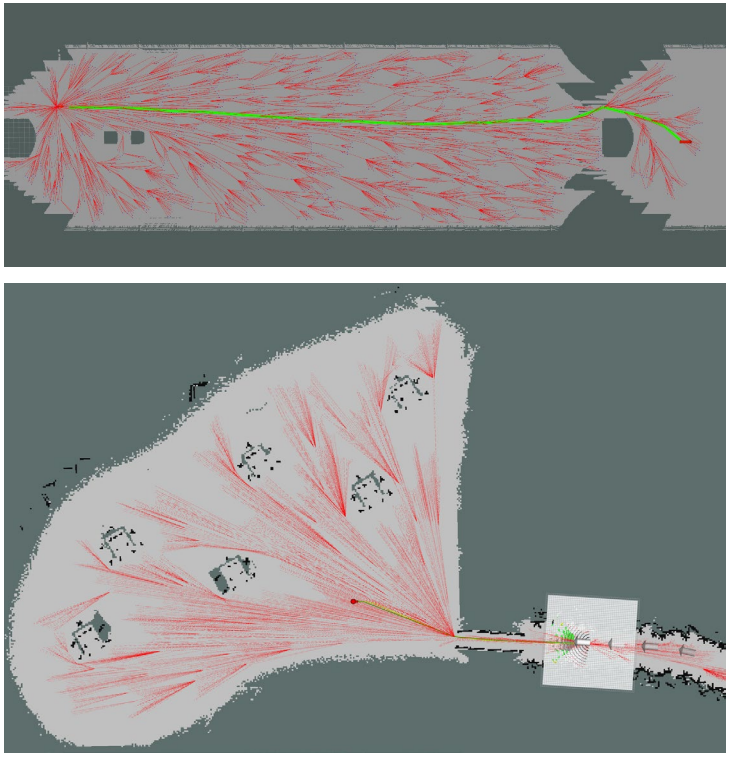
Algorithm 1 RRT

Input: initial node x_{init}, max nodes $K \leftarrow \mathbb{N}$
$V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$;
for $i = 0$ **to** K **do**
  $x_{rand} \leftarrow \text{Sample}(i)$
  $x_{nearest} \leftarrow \text{Nearest}(V, x_{rand})$
  $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand})$
  if $\text{ObstacleFree}(x_{nearest}, x_{new})$ **then**
    $V \leftarrow V \cup \{x_{new}\}$
    $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
  end if
end for
return (V, E)

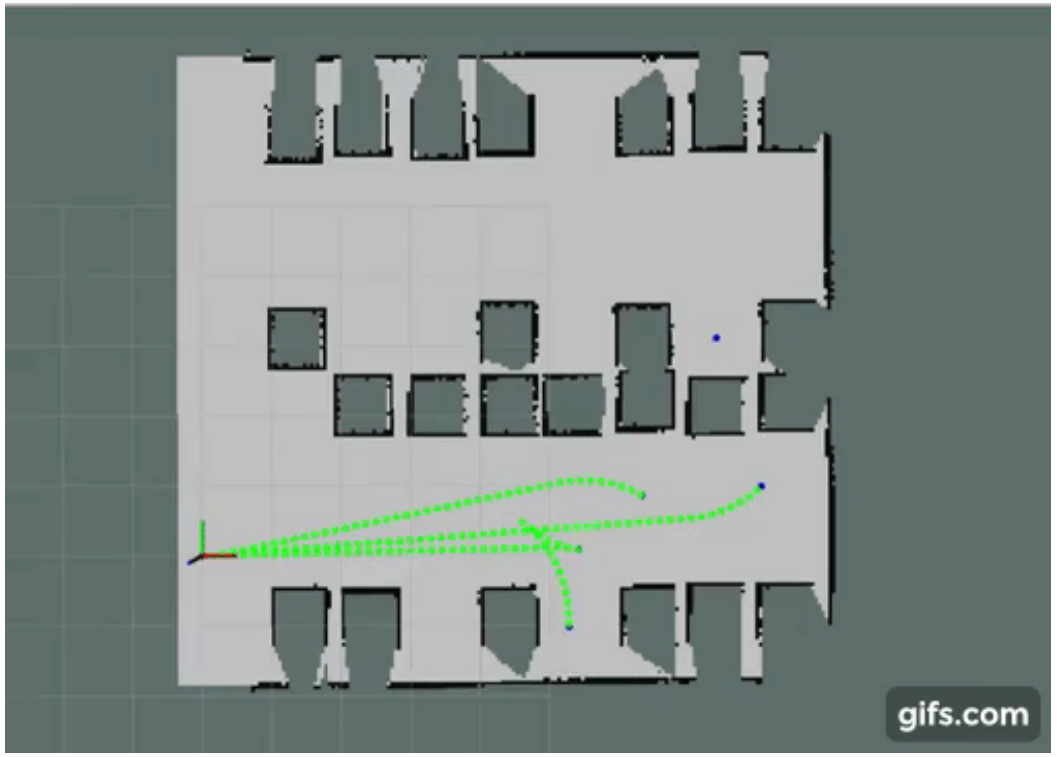
Algorithm of RRT

Motion Planning Methods

- **Example of implementing RRT*-based planner**



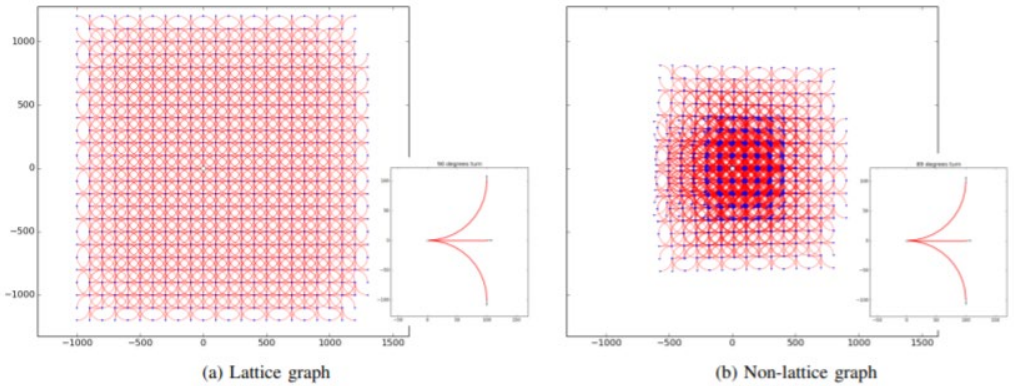
RRT*-based global path planning



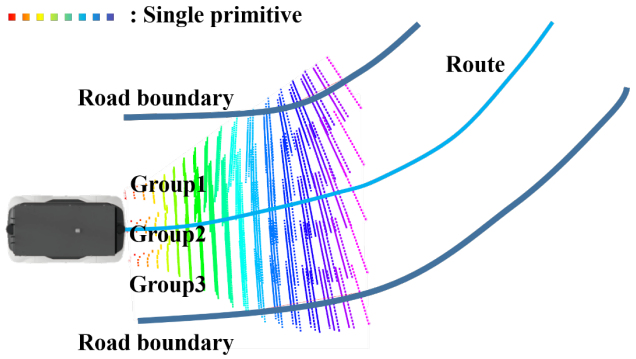
Kino-dynamic RRT*-based path planning

Motion Planning Methods

- **Motion primitives-based planning**
  - It generates a sample considering vehicle's constraints to consider a reachability.
  - It builds a cost function to consider the cost minimized(optimal) path in the generated paths.
  - It also discretized the set of actions as like A* algorithm, and find a drivable paths in the candidates.



Examples of lattice graph planner



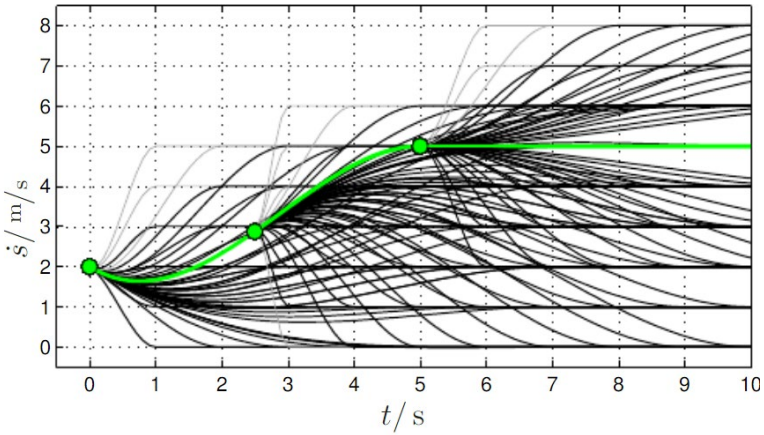
Examples of sampling based planner - our lab poseman robot project

Paden, Brian, et al. "A survey of motion planning and control techniques for self-driving urban vehicles." *IEEE Transactions on intelligent vehicles* 1.1 (2016): 33-55.

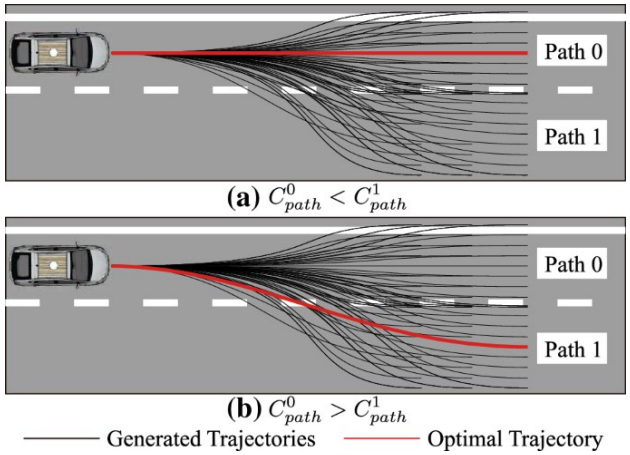
Motion Primitive-based Algorithm

- **Motion primitives-based planning algorithm**
  - Motion primitives-based path planning relies on a set of precomputed, feasible motions or maneuvers known as “motion primitives.”
  - The core idea behind using motion primitives is to discretize the continuous control space into a finite set of actions that can be efficiently searched.
  - The planner performs 1) Generating motion primitives, 2) Computing cost of primitives, and 3) selecting minimum cost motion primitive as planned path.

[1]



[2]



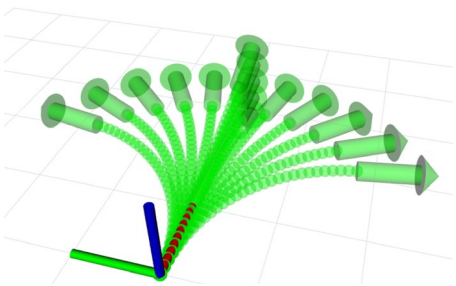
[1]: Werling, M., Ziegler, J., Kammel, S., & Thrun, S. (2010, May). Optimal trajectory generation for dynamic street scenarios in a frenet frame. In 2010 IEEE international conference on robotics and automation (pp. 987-993). IEEE.

[2]: Yoneda, K., Iida, T., Kim, T., Yanase, R., Aldibaja, M., & Suganuma, N. (2018). Trajectory optimization and state selection for urban automated driving. Artificial Life and Robotics, 23(4), 474-480.

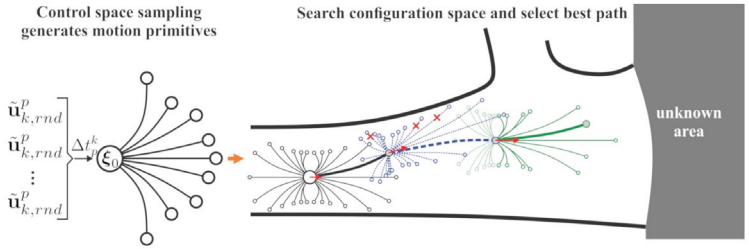
Motion Primitive-based Algorithm

- **Motion primitives-based planning algorithm**

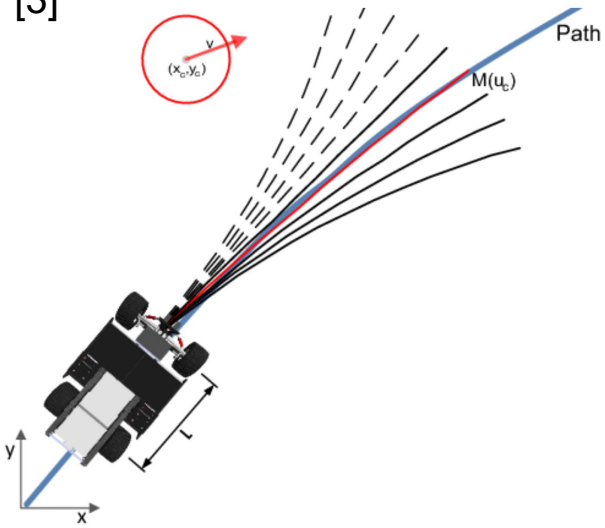
[1]



[2]



[3]



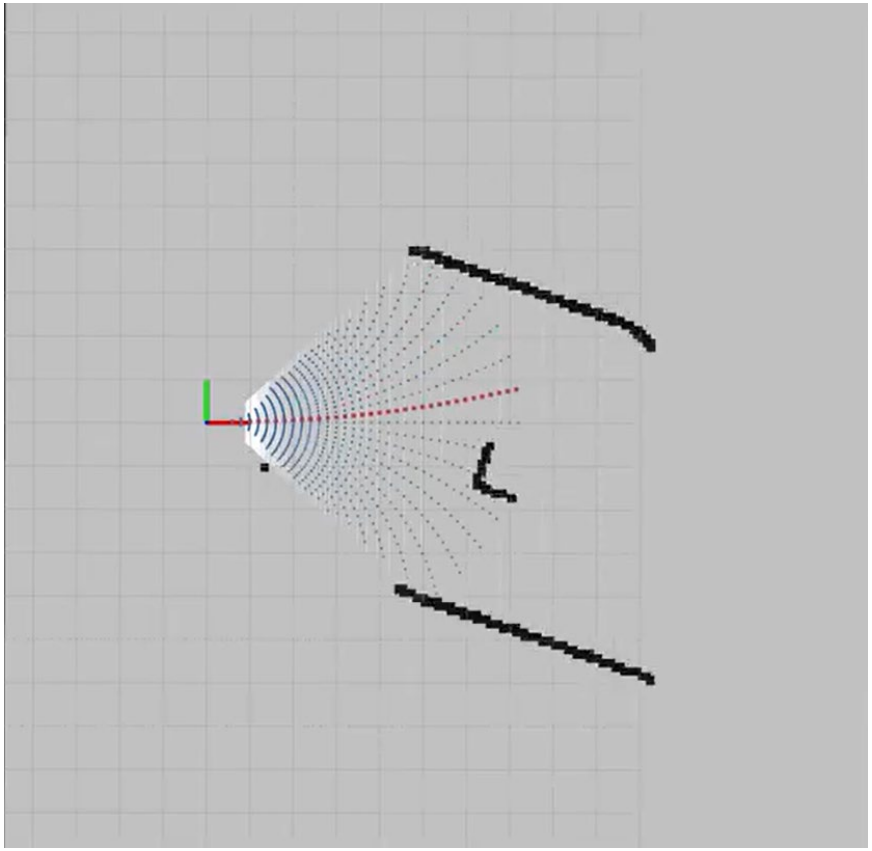
[1]: Scharff Willners, J., Gonzalez-Adell, D., Hernández, J. D., Pairet, È., & Petillot, Y. (2021). Online 3-dimensional path planning with kinematic constraints in unknown environments using hybrid A* with tree pruning. Sensors, 21(4), 1152.

[2]: Dharmadhikari, M., Dang, T., Solanka, L., Loje, J., Nguyen, H., Khedekar, N., & Alexis, K. (2020, May). Motion primitives-based path planning for fast and agile exploration using aerial robots. In 2020 IEEE International Conference on Robotics and Automation (ICRA) (pp. 179-185). IEEE.

[3]: Garrote, L., Premebida, C., Silva, M., & Nunes, U. (2014, July). An RRT-based navigation approach for mobile robots and automated vehicles. In 2014 12th IEEE International Conference on Industrial Informatics (INDIN) (pp. 326-331). IEEE.

Motion Primitives-based Algorithm

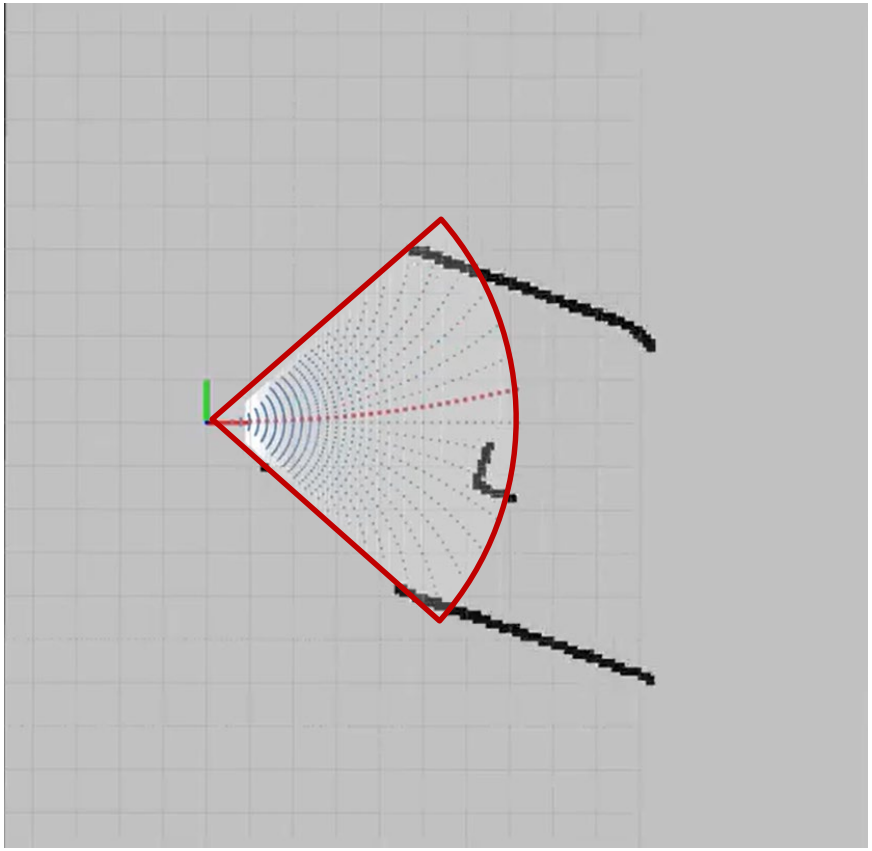
- **Motion primitives-based planning algorithm**



1. **Generating motion primitives**
2. **Computing cost of primitives**
3. **Selecting minimum cost motion**

Motion Primitives-based Algorithm

- **Motion primitives-based planning algorithm**

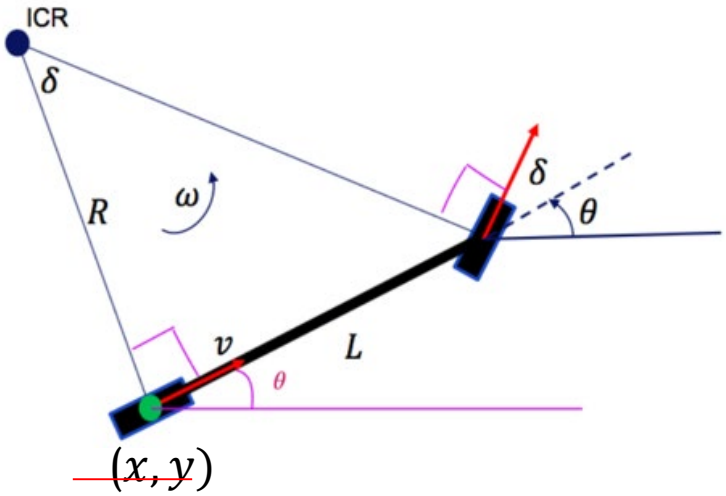


1. Generating motion primitives

- Radial motion primitives
- Kinematic model-based
- Dynamic model-based

Motion Primitives-based Algorithm

➢ Motion primitives-based planning algorithm



✓ Velocity

$$\dot{x} = v\cos\psi$$
$$\dot{y} = v\sin\psi$$

✓ Acceleration

$$\dot{v} = a$$

✓ Instantaneous Center of Rotation (ICR)

$$\tan\delta = \frac{L}{R} \qquad v = R\omega = R\dot{\psi}$$

$$\Rightarrow \quad \dot{\psi} = \frac{v}{L}\tan\delta$$

$$\Rightarrow \quad \begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\psi}\\ \dot{v}\end{bmatrix} = \begin{bmatrix}v\cos\psi\\ v\sin\psi\\ \frac{v}{L}\tan\delta\\ a\end{bmatrix}$$

State : $\{x, y, \psi, v\}$
Control input : $\{\delta, a\}$

x : position x
y : position y
ψ : yaw angle
v : velocity

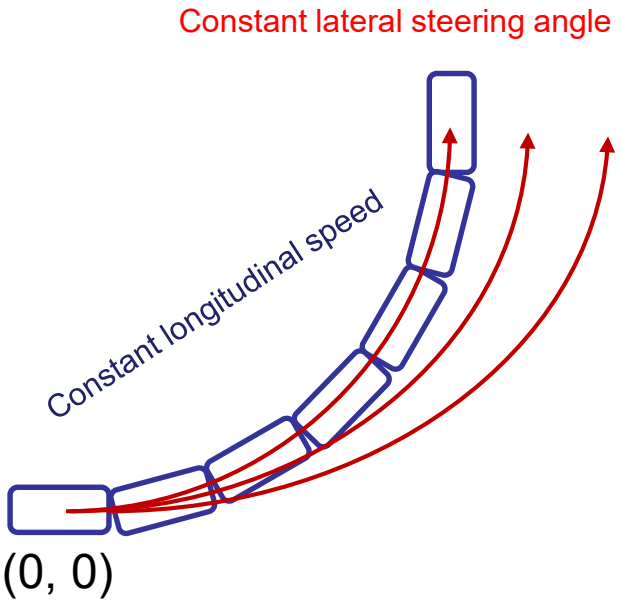
δ : steering angle
a : acceleration

Discrete model in code

x_(t+1) = x_t + x_dot * dt
y_(t+1) = y_t + y_dot * dt
ψ_(t+1) = ψ_t + ψ_dot * dt
v_(t+1) = v_t + v_dot * dt

Motion Primitives-based Algorithm

- **Motion primitives-based planning algorithm**



For loop (lateral domain, steering angle)

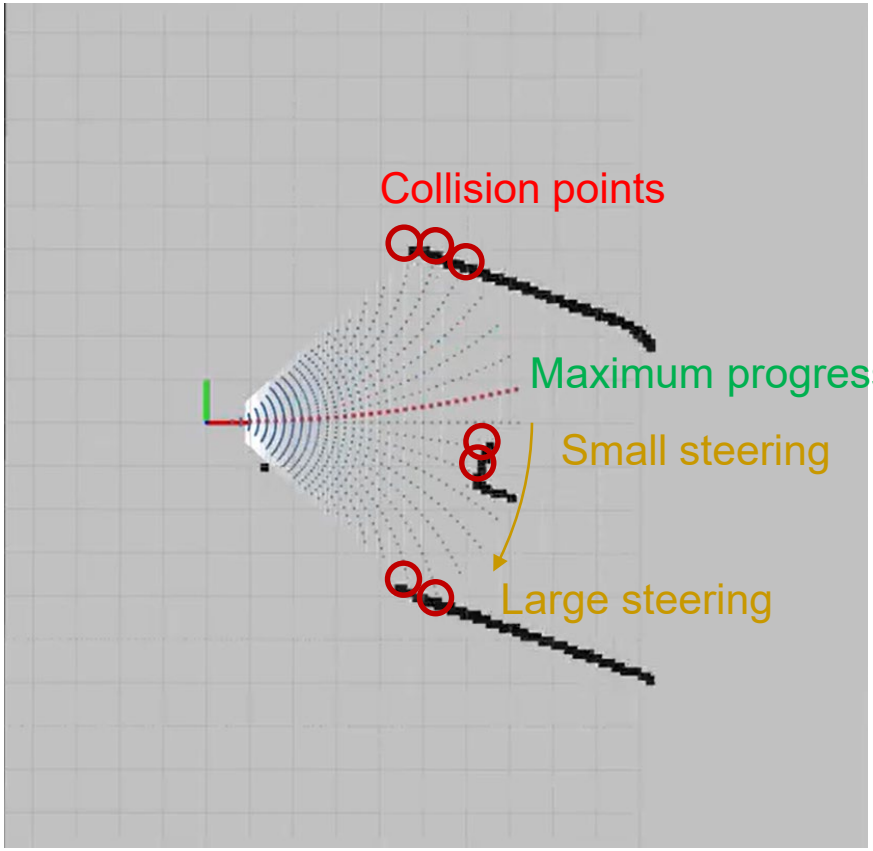
For loop (longitudinal domain, constant speed)

Update & Append motions using

the vehicle model equations

Motion Primitives-based Algorithm

- **Motion primitives-based planning algorithm**

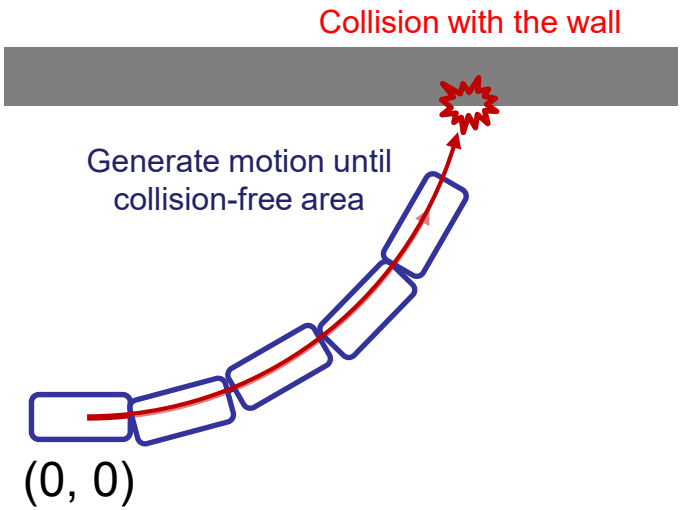


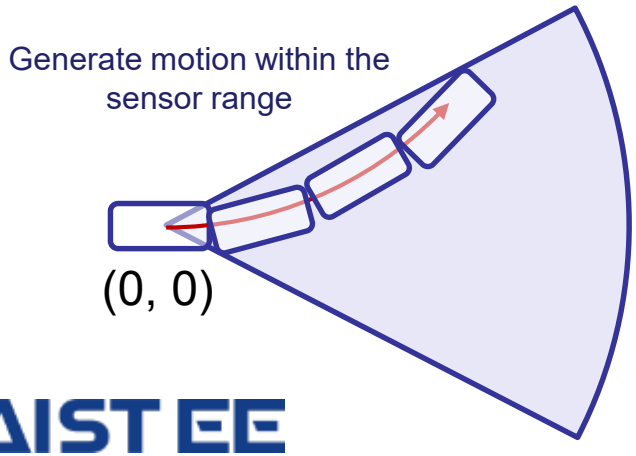
2. Computing cost of primitives

- Collision cost term C_{colli}
- Steering cost term C_{steer}
- Progress term C_{prog}
- …

Motion Primitives-based Algorithm

- **Motion primitives-based planning algorithm**

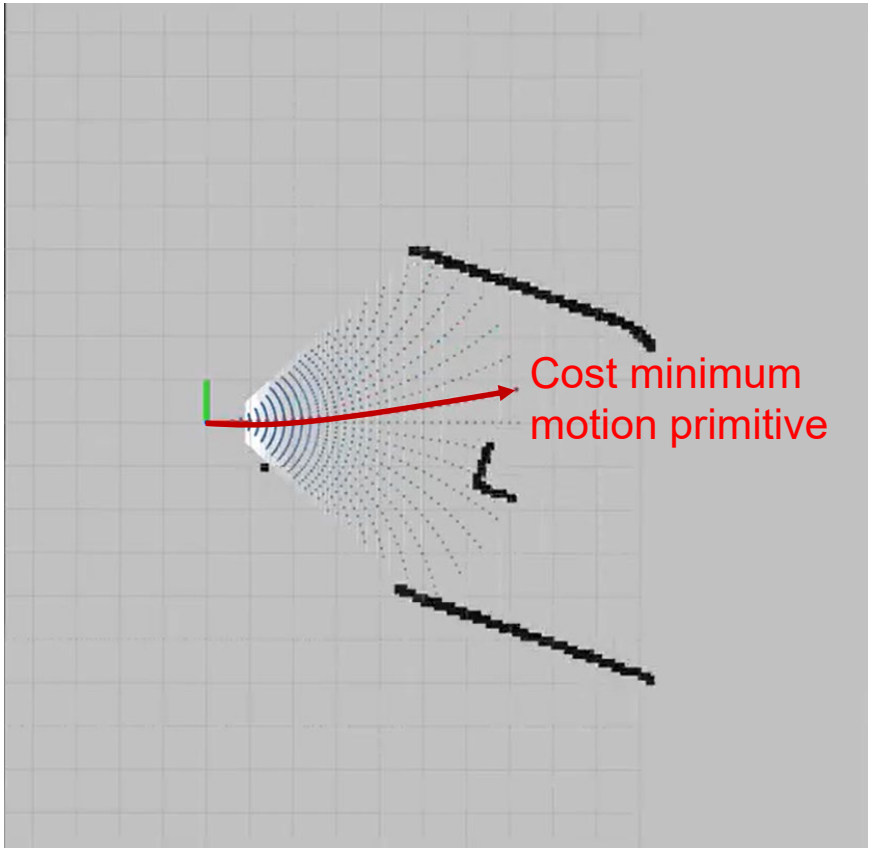





```
For loop (lateral domain, steering angle)
    For loop (longitudinal domain, constant speed)
        while (not is_collision AND in_range)
            Update & Append motions using the
            vehicle model equations
```

Motion Primitives-based Algorithm

- **Motion primitives-based planning algorithm**



3. Selecting minimum cost motion

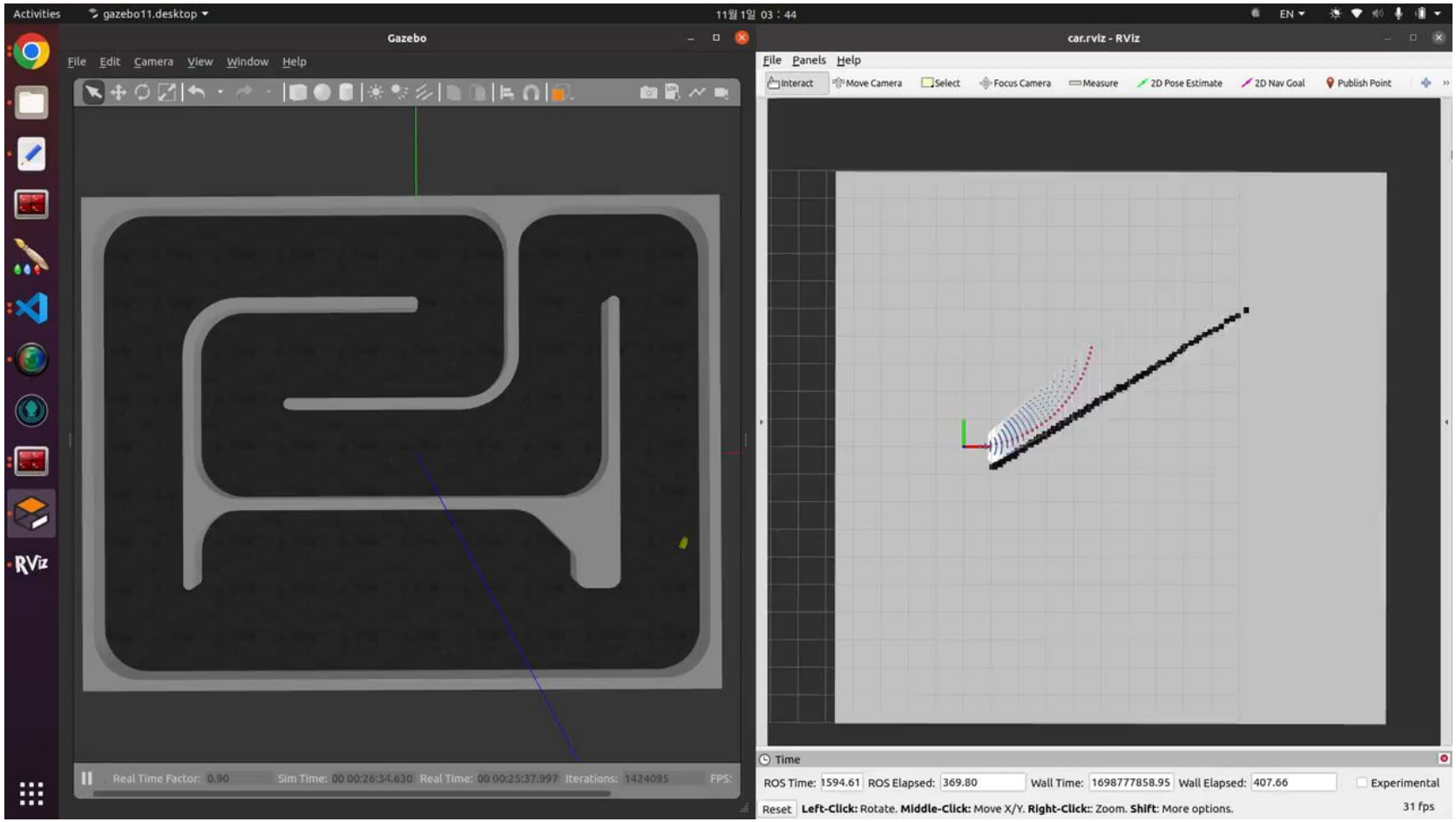
- Compute the weighted summation of the cost terms:

$$C_{total} = w_{colli}C_{colli} + w_{steer}C_{steer} + w_{prog}C_{prog} + w_X C_X$$

- Find the minimum cost motion primitive.

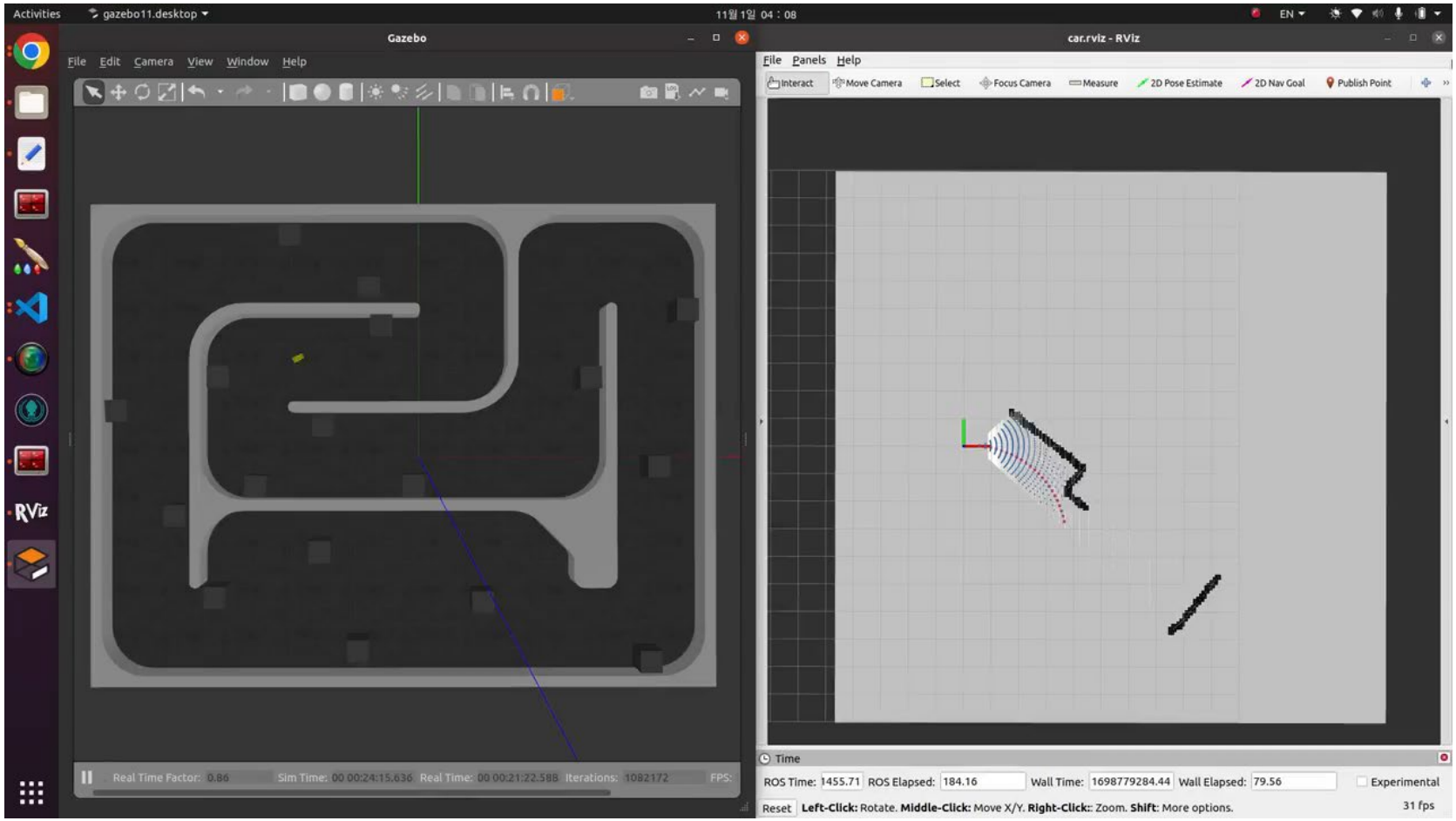
Motion Primitives-based Algorithm

- **Motion primitives-based planning algorithm**
  - Motion primitives-based planner will be provided as an assignment in next week.



Motion Primitives-based Algorithm

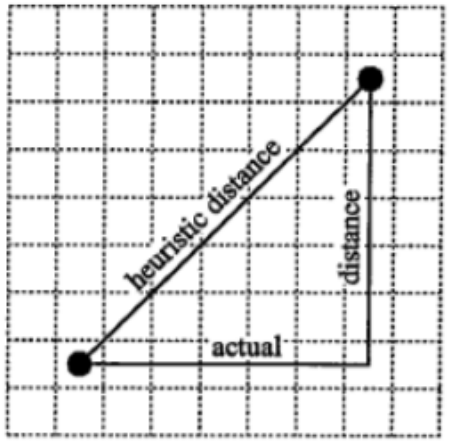
- **Motion primitives-based planning algorithm**
  - Motion primitives-based planner will be provided as an assignment in next week.



Appendix

A* Algorithm

- **Evaluation Cost Function** $f(n) = g(n) + h(n)$
  - The value of $f(n)$ determines how "good" a path is.
- **Operating Cost Function** $g(n)$
  - Actual operating cost having been already traversed.
    - ✓ e.g.) Actual 'distance' to current location.
- **Heuristic Function** $h(n)$
  - Information used to find the promising node before traversing



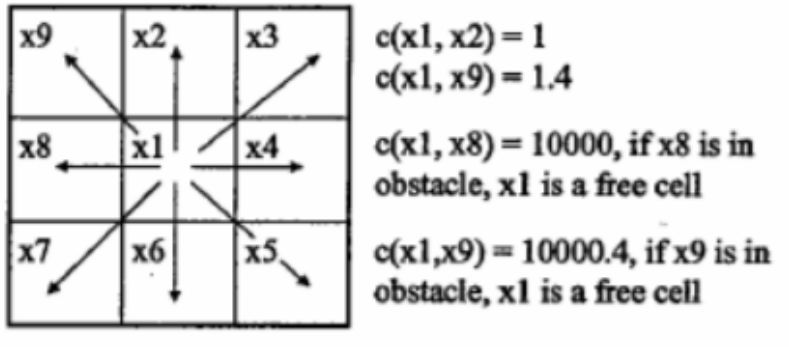


Fig. Cost on a grid

A* Algorithm

- **2 lists to store information:**
  - Open list(O): stores nodes for expansions.
  - Closed list(C): stores nodes which we have explored.

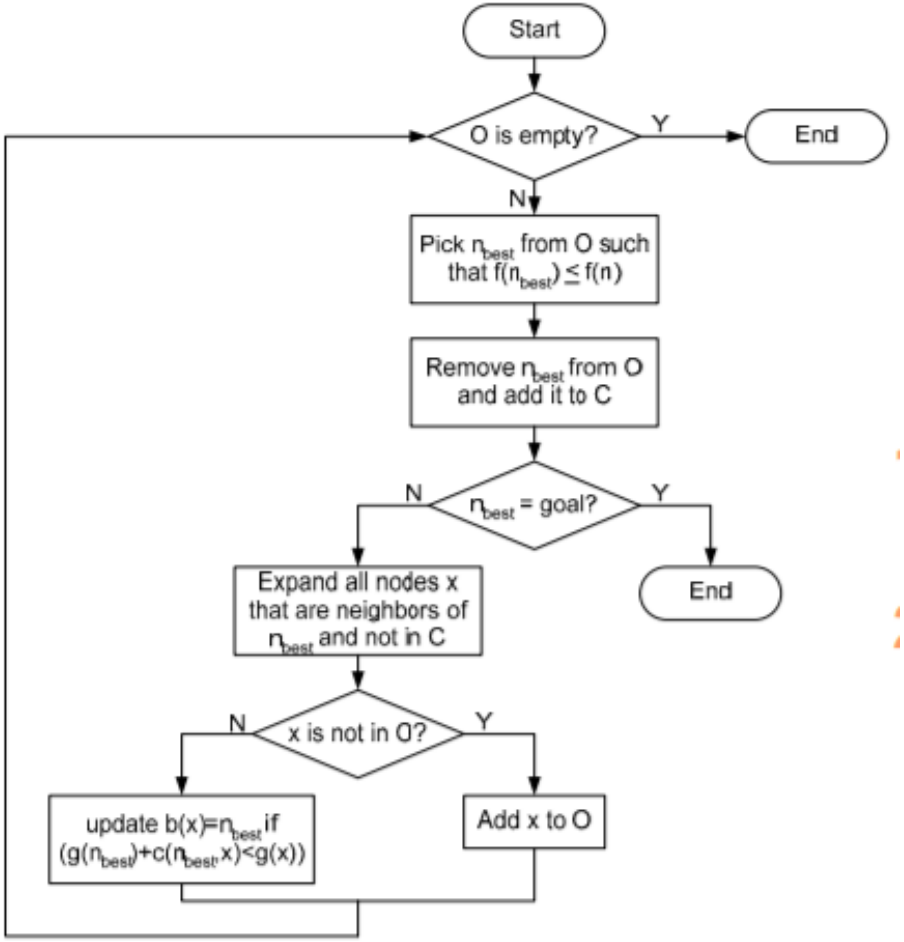
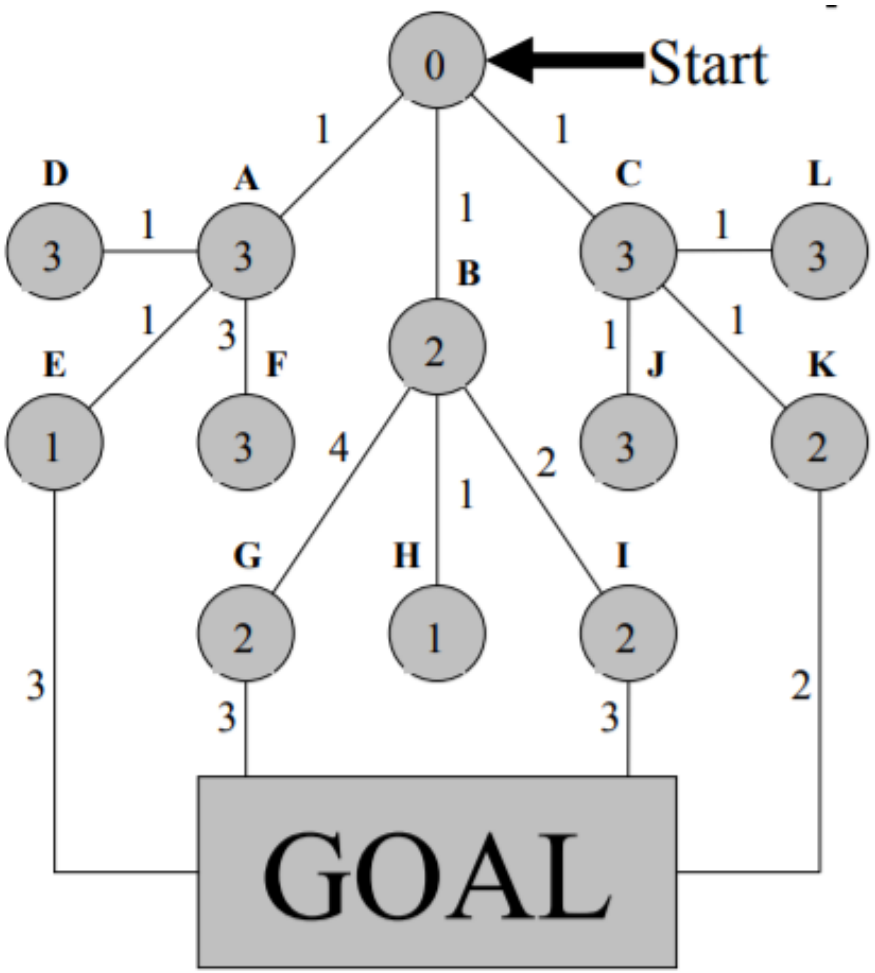
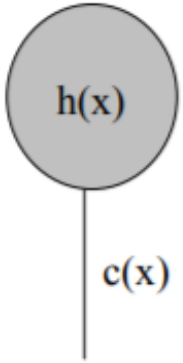


Fig. Flow chart of A* algorithm

A* Algorithm



- **Legend**
  - $h(n)$: heuristic function that estimates the *cost-to-go.*
  - $c(n)$: 'distance' from a node to the next node.
  - $g(n)$: sum of all previous arc costs, $c(n)$, from start to n.



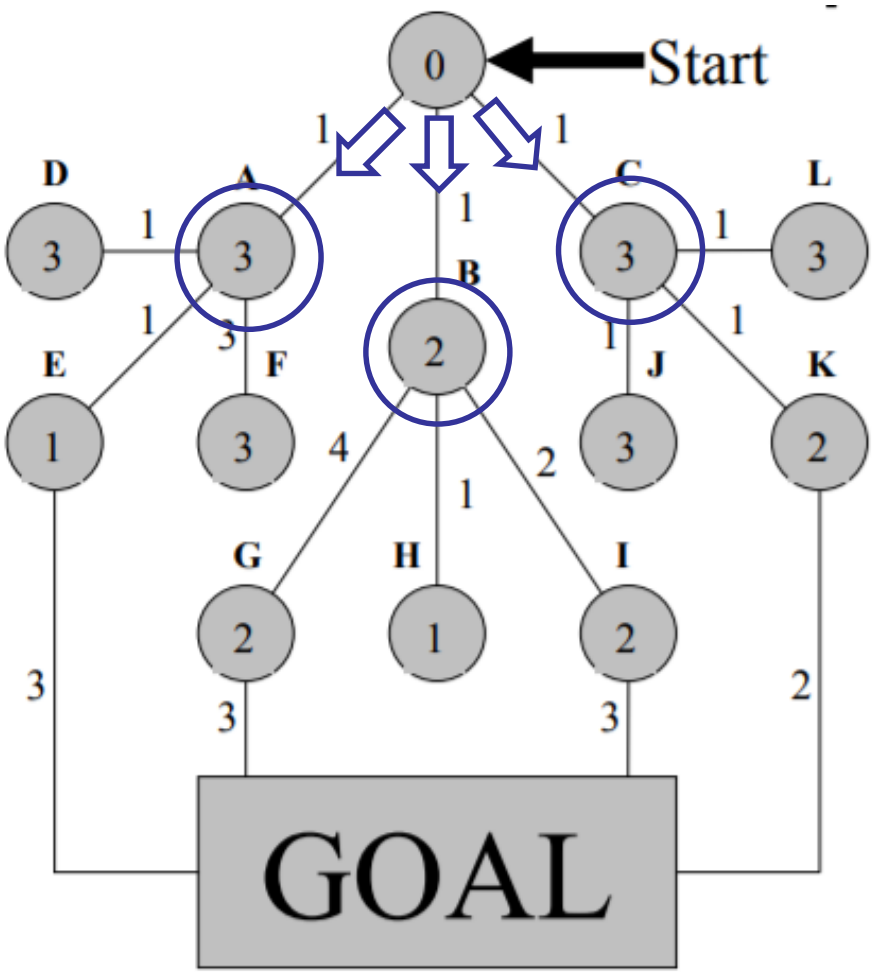
- **Evaluation Cost Function**

$$f(n) = g(n) + h(n)$$

  - e.g.) $g(H) = 1 + 1, h(H) = 1$

$$\therefore f(H) = 3$$

A* Algorithm



- **Expand the start node**
  - If the goal is not found, expand the first node in the priority queue(in this case, B).

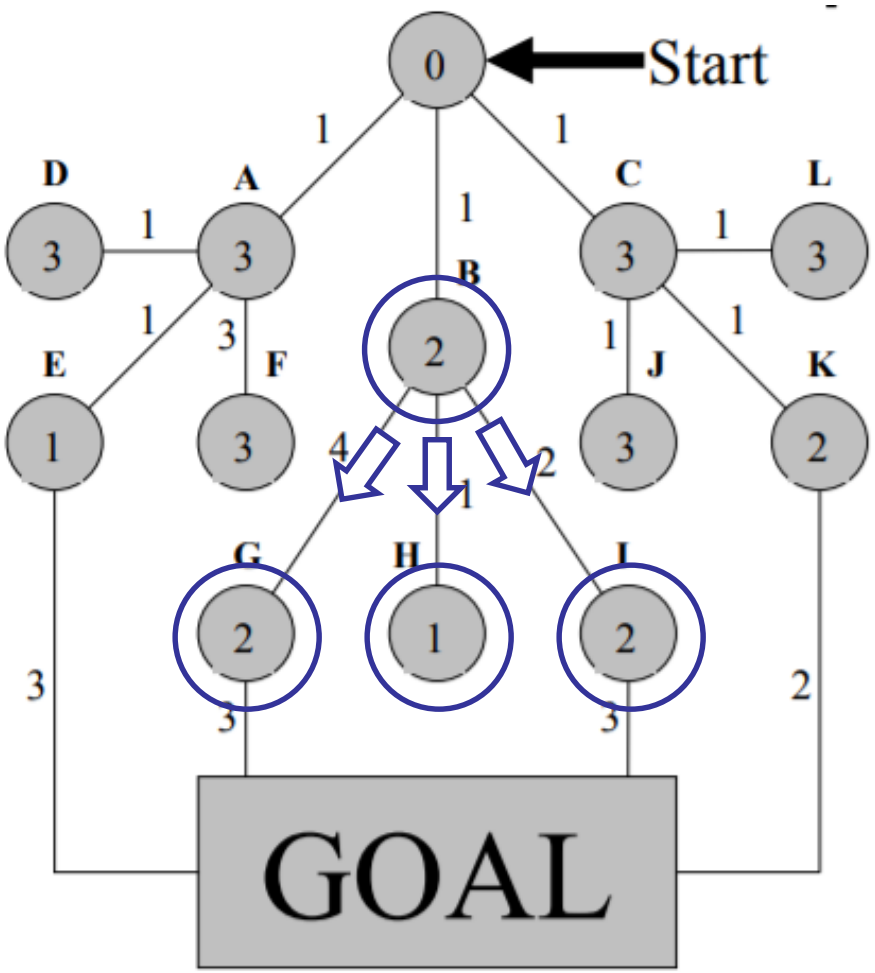
B(3)
A(4)
C(4)

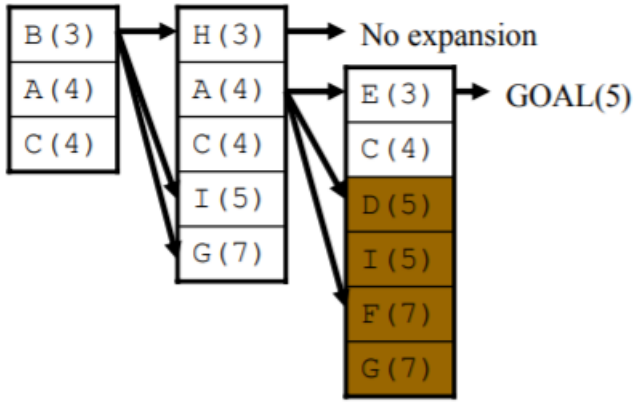
- **Insert nodes to the priority queue**
  - Continue until the goal is found.
  - Or, until the priority queue is empty.

H(3)
A(4)
C(4)
I(5)
G(7)

- **Keep a pointer to nodes' respective parents.**
  - e.g.) Nodes A, B and C point to *start*.

A* Algorithm



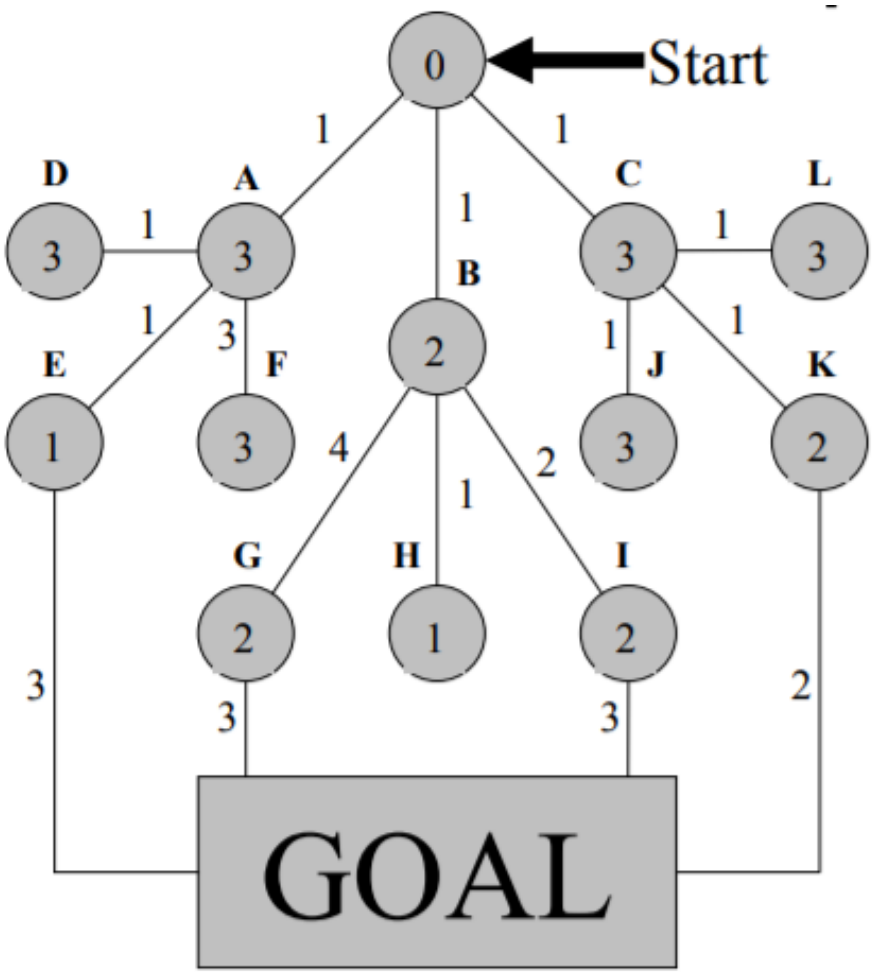


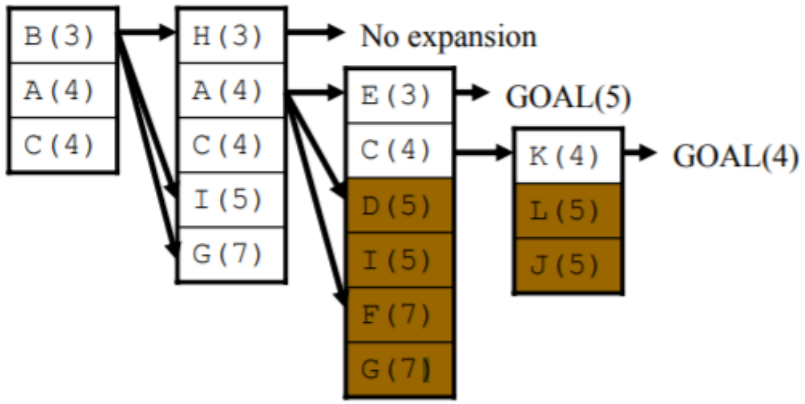
- **Found a path to the goal**

  Start→A→E→Goal (*cost **5***)

  - We can throw away nodes with cost(priority) ≥ 5, because arc costs are assumed non-negative.

A* Algorithm





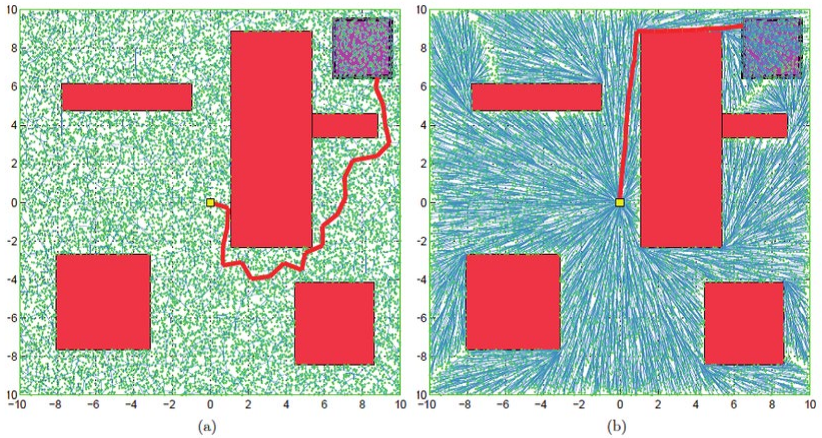
- **Found a new path to the goal**

  Start→**C**→**K**→Goal (*cost **4***)

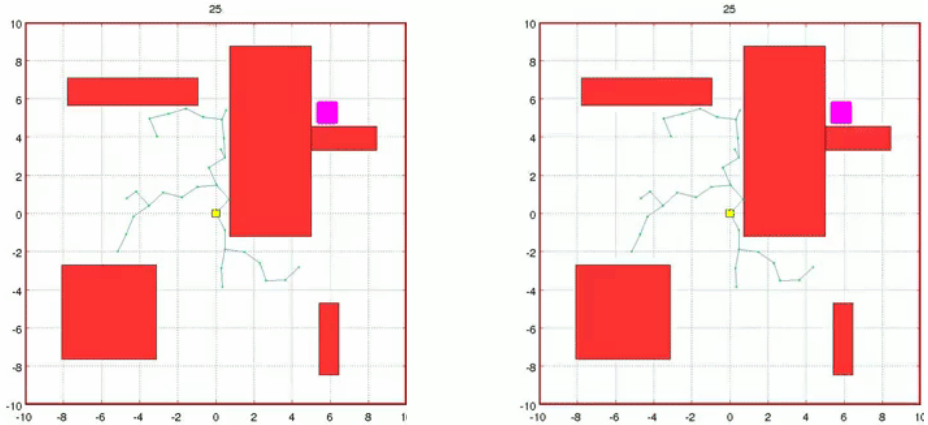
  - Repeating the process, we could find a shorter path through node K.
  - If the priority queue still wasn't empty, we would continue expanding while throwing away nodes with cost higher than 4.

Rapidly exploring Random Tree (RRT)

- **Rapidly-exploring Random Tree (RRT) [1]**
  - Rapidly build a space-filling tree toward unexplored collision-free region.
  - Incremental tree construction from randomly sampled nodes.
  - Inherently biased to grow toward large unsearched area.
  - Efficiently search nonconvex, high-dimensional spaces.



The results of the RRT (Left) and RRT* (Right) algorithms

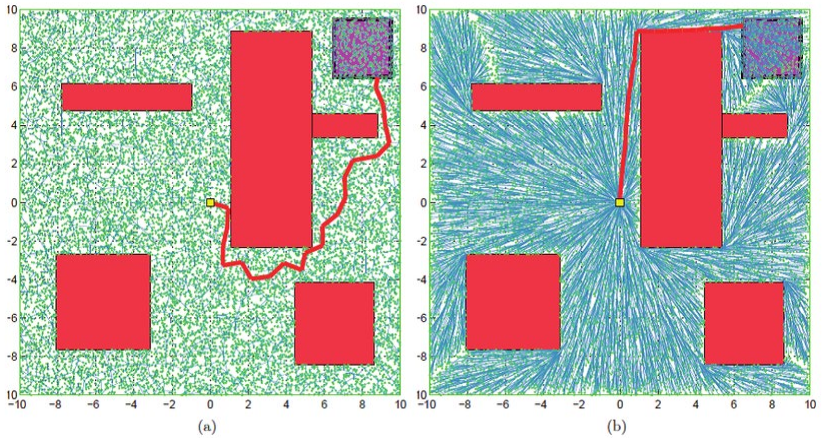


Exploring trees of the RRT (Left) and RRT* (Right) algorithms

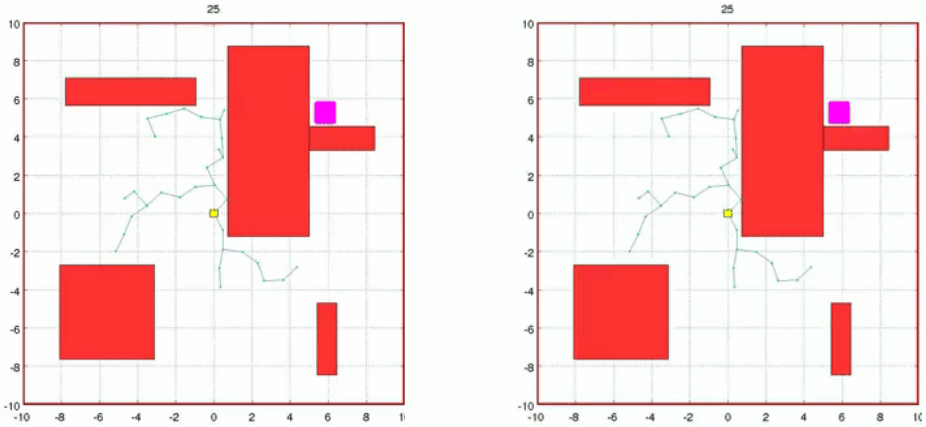
[1]: Rapidly-exploring Random Tree
[2]: Sampling-based Algorithms for Optimal Motion Planning

Rapidly exploring Random Tree (RRT)

- **Sampling-based Algorithms for Optimal Motion Planning (RRT*) [2]**
  - Solution path from the RRT is not optimal (path is usually curvy, zig-zag, jerky).
  - RRT* has the favorable property of an **almost sure convergence** to an optimal solution [2].
  - Additional *Rewire*, *Parent* procedures for finding the optimal path are applied.
  - RRT*'s optimality is theoretically proven as the number of samples approaches infinity.



The results of the RRT (Left) and RRT* (Right) algorithms



Exploring trees of the RRT (Left) and RRT* (Right) algorithms

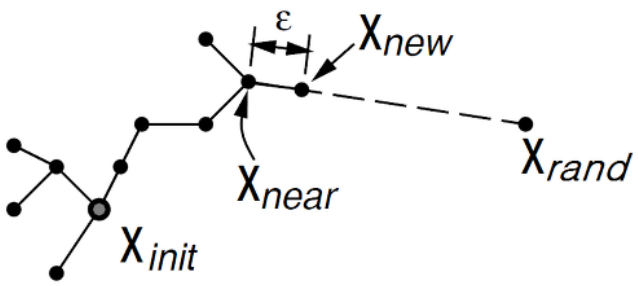
[1]: Rapidly-exploring Random Tree
[2]: Sampling-based Algorithms for Optimal Motion Planning

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT [1]**

Algorithm 1 RRT

1: **Input:** initial node x_{init}, max nodes $K \leftarrow \mathbb{N}$
2: $V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$;
3: **for** $i = 0$ **to** K **do**
4: $x_{rand} \leftarrow \text{Sample}(i)$
5: $x_{nearest} \leftarrow \text{Nearest}(V, x_{rand})$
6: $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand})$
7: **if** $\text{ObstacleFree}(x_{nearest}, x_{new})$ **then**
8: $V \leftarrow V \cup \{x_{new}\}$
9: $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
10: **end if**
11: **end for**
12: **return** (V, E)



Set an initial node x_{init} and maximum number of the loop K.

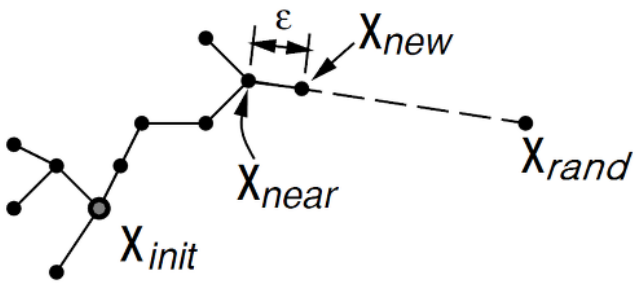
[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT [1]**

Algorithm 1 RRT

1: **Input:** initial node x_{init}, max nodes $K \leftarrow \mathbb{N}$
2: $V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$;
3: **for** $i = 0$ **to** K **do**
4: $x_{rand} \leftarrow \text{Sample}(i)$
5: $x_{nearest} \leftarrow \text{Nearest}(V, x_{rand})$
6: $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand})$
7: **if** ObstacleFree$(x_{nearest}, x_{new})$ **then**
8: $V \leftarrow V \cup \{x_{new}\}$
9: $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
10: **end if**
11: **end for**
12: **return** (V, E)



Sample a random node x_{rand} in the collision-free space.

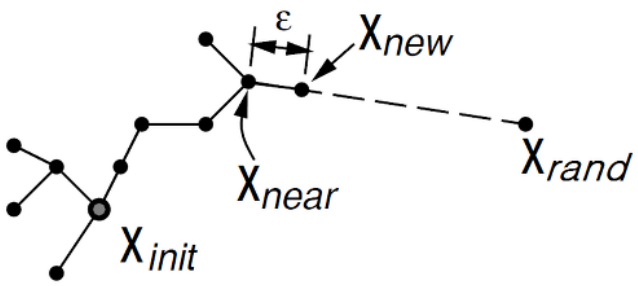
[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT [1]**

Algorithm 1 RRT

1: **Input:** initial node x_{init}, max nodes $K \leftarrow \mathbb{N}$
2: $V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$;
3: **for** $i = 0$ **to** K **do**
4: $x_{rand} \leftarrow \text{Sample}(i)$
5: $x_{nearest} \leftarrow \text{Nearest}(V, x_{rand})$
6: $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand})$
7: **if** $\text{ObstacleFree}(x_{nearest}, x_{new})$ **then**
8: $V \leftarrow V \cup \{x_{new}\}$
9: $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
10: **end if**
11: **end for**
12: **return** (V, E)



Find a nearest node $x_{nearest}$ in the constructed tree with the randomly sampled node x_{rand}.

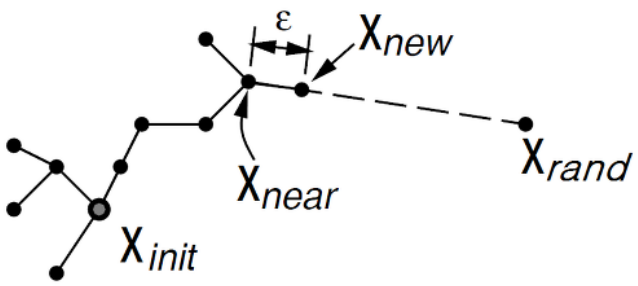
[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT [1]**

Algorithm 1 RRT

1: **Input:** initial node x_{init}, max nodes $K \leftarrow \mathbb{N}$
2: $V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$;
3: **for** $i = 0$ **to** K **do**
4: $x_{rand} \leftarrow \text{Sample}(i)$
5: $x_{nearest} \leftarrow \text{Nearest}(V, x_{rand})$
6: $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand})$
7: **if** $\text{ObstacleFree}(x_{nearest}, x_{new})$ **then**
8: $V \leftarrow V \cup \{x_{new}\}$
9: $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
10: **end if**
11: **end for**
12: **return** (V, E)



Stretch the tree with an ε length from $x_{nearest}$ to x_{rand} and make a new node x_{new}.

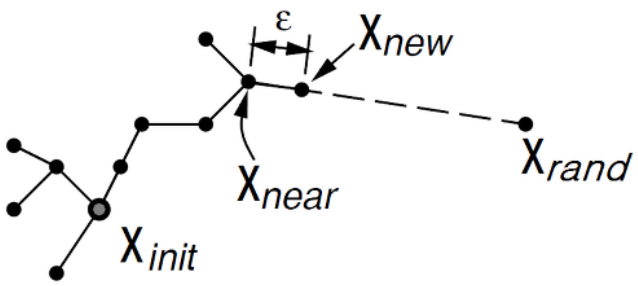
[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT [1]**

Algorithm 1 RRT

1: **Input:** initial node x_{init}, max nodes $K \leftarrow \mathbb{N}$
2: $V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$;
3: **for** $i = 0$ **to** K **do**
4: $x_{rand} \leftarrow \text{Sample}(i)$
5: $x_{nearest} \leftarrow \text{Nearest}(V, x_{rand})$
6: $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand})$
7: **if** $\text{ObstacleFree}(x_{nearest}, x_{new})$ **then**
8: $V \leftarrow V \cup \{x_{new}\}$
9: $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
10: **end if**
11: **end for**
12: **return** (V, E)



If the edge $(x_{nearest}, x_{new})$ is in the obstacle-free region, add the x_{new} in the node set V and add the branch (edge) $(x_{nearest}, x_{new})$ in the edge set E.

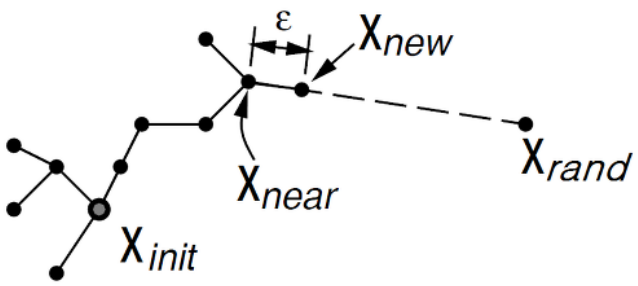
[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT [1]**

Algorithm 1 RRT

1: **Input:** initial node x_{init}, max nodes $K \leftarrow \mathbb{N}$
2: $V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$;
3: **for** $i = 0$ **to** K **do**
4: $x_{rand} \leftarrow \text{Sample}(i)$
5: $x_{nearest} \leftarrow \text{Nearest}(V, x_{rand})$
6: $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand})$
7: **if** $\text{ObstacleFree}(x_{nearest}, x_{new})$ **then**
8: $V \leftarrow V \cup \{x_{new}\}$
9: $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
10: **end if**
11: **end for**
12: **return** (V, E)



Repeat this process until finding the solution path.

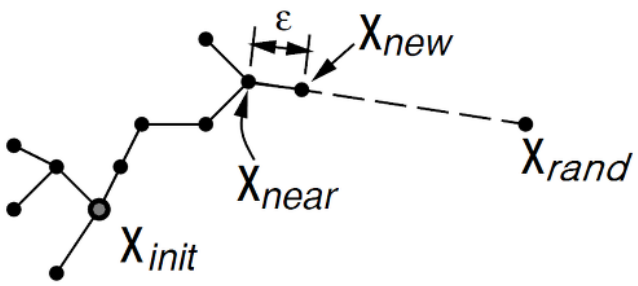
[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT* [1]**

Algorithm 2 RRT*

1: **Input:** initial node x_{init}, max nodes $K \leftarrow \mathbb{N}$
2: $V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$;
3: **for** $i = 0$ **to** K **do**
4: $x_{rand} \leftarrow \text{Sample}(i)$
5: $x_{nearest} \leftarrow \text{Nearest}(V, x_{rand})$
6: $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand})$
7: $V \leftarrow V \cup \{x_{new}\}$
8: **if** ObstacleFree($x_{nearest}$,x_{new}) **then**
9: $X_{near} \leftarrow \text{Near}(V, x_{new})$
10: $x_{nearest} \leftarrow \text{Parent}(X_{near}, x_{nearest}, x_{new})$
11: $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
12: $E \leftarrow \text{Rewire}(X_{near}, E, x_{new})$
13: **end if**
14: **end for**
15: **return** (V, E)



← Same processes with RRT.

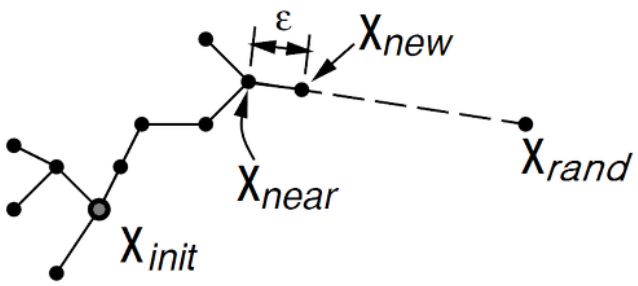
[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT* [1]**

Algorithm 2 RRT*

1: **Input:** initial node x_{init}, max nodes $K \leftarrow \mathbb{N}$
2: $V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$;
3: **for** $i = 0$ **to** K **do**
4: $x_{rand} \leftarrow \text{Sample}(i)$
5: $x_{nearest} \leftarrow \text{Nearest}(V, x_{rand})$
6: $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand})$
7: $V \leftarrow V \cup \{x_{new}\}$
8: **if** $\text{ObstacleFree}(x_{nearest}, x_{new})$ **then**
9: $X_{near} \leftarrow \text{Near}(V, x_{new})$
10: $x_{nearest} \leftarrow \text{Parent}(X_{near}, x_{nearest}, x_{new})$
11: $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$
12: $E \leftarrow \text{Rewire}(X_{near}, E, x_{new})$
13: **end if**
14: **end for**
15: **return** (V, E)



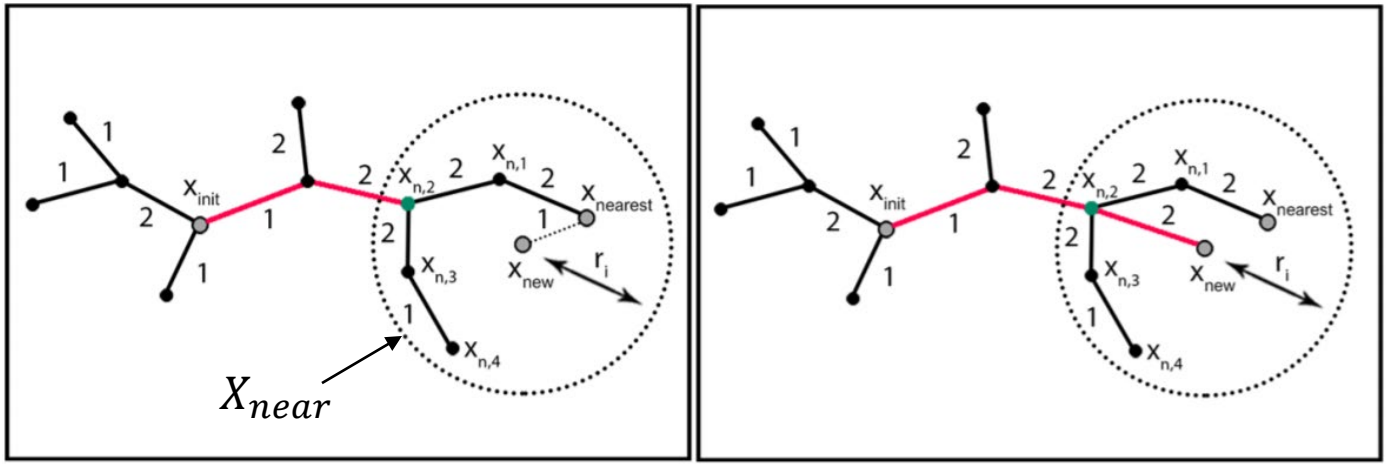
← *Parent* process for finding new parent node of the x_{new}.

← *Rewire* process for rewiring the branches of the tree to find a more optimal path.

[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT* [1]**



Algorithm 3 Parent

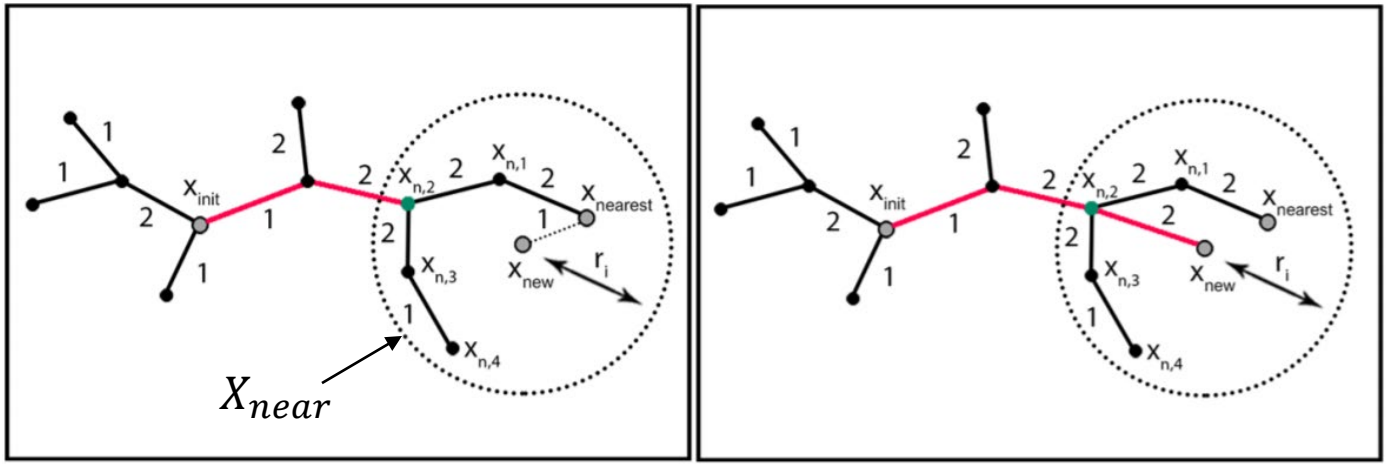
```
1: Input: X_near, x_nearest, x_new
2: for x_near ∈ X_near do
3:     if ObstacleFree(x_near, x_new) then
4:         c' ← x_near.cost + c(Line(x_near, x_new))
5:         if c' < x_new.cost then
               x_nearest ← x_near
6:         end if
7:     end if
8: end for
9: return x_nearest
```

Parent process gets arguments $X_{near}, x_{nearest}, x_{new}$.
X_{near} is a set of nodes in an area within the range of length r_i.

[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT* [1]**



Algorithm 3 Parent

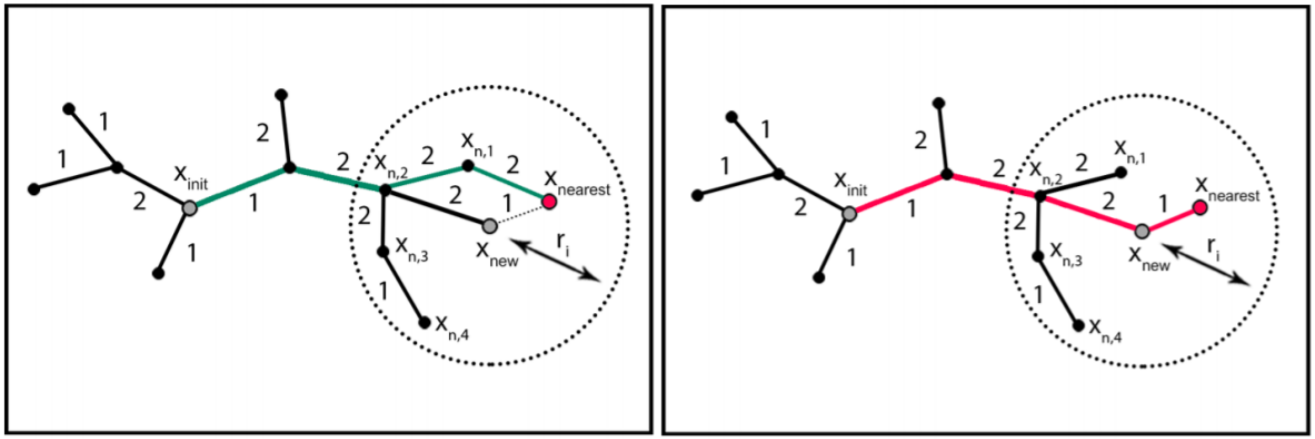
```
1: Input: X_near, x_nearest, x_new
2: for x_near ∈ X_near do
3:    if ObstacleFree(x_near, x_new) then
4:       c' ← x_near.cost + c(Line(x_near, x_new))
5:       if c' < x_new.cost then
            x_nearest ← x_near
6:       end if
7:    end if
8: end for
9: return x_nearest
```

For each node x_{near} in X_{near},
1) check collision-free of the edge (x_{near}, x_{new}).
2) If no collision, calculate the cost of the path with the edge.
3) If the cost is lower (more optimal) than the previous one, update the parent node of the x_{new} as x_{near}.

[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT* [1]**



Algorithm 4 Rewire

```
1: Input: X_near, E, x_new
2: for x_near ∈ X_near do
3:    if ObstacleFree(x_new,x_near) and x_near.cost > x_new.cost + c(Line(x_near, x_new)) then
4:       x_parent ← x_near.parent
5:       E ← E \{(x_parent, x_near)}
6:       E ← E ∪ {(x_new, x_near)}
7:    end if
8: end for
9: return E
```

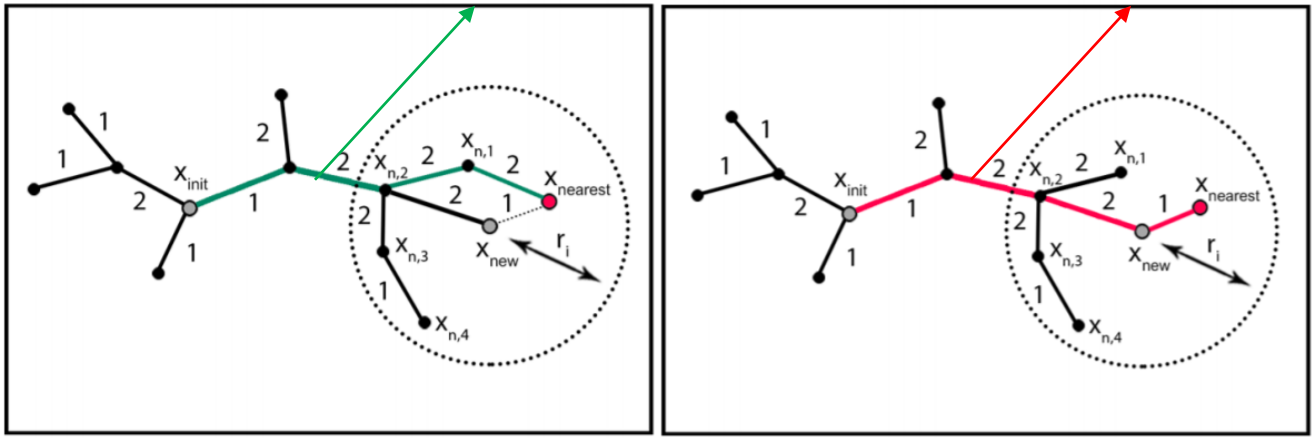
Rewire process get arguments X_{near}, E, x_{new}.

[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT* [1]**

Path cost: $x_{near}.\text{cost} > x_{new}.\text{cost} + \text{c}(\text{Line}(x_{near}, x_{new}))$



Algorithm 4 Rewire

1: **Input:** X_{near}, E, x_{new}
2: **for** $x_{near} \in X_{near}$ **do**
3: **if** ObstacleFree(x_{new},x_{near}) **and** $x_{near}.\text{cost} > x_{new}.\text{cost} + \text{c}(\text{Line}(x_{near}, x_{new}))$ **then**
4: $x_{parent} \leftarrow x_{near}.\text{parent}$
5: $E \leftarrow E \setminus \{(x_{parent}, x_{near})\}$
6: $E \leftarrow E \cup \{(x_{new}, x_{near})\}$
7: **end if**
8: **end for**
9: **return** E

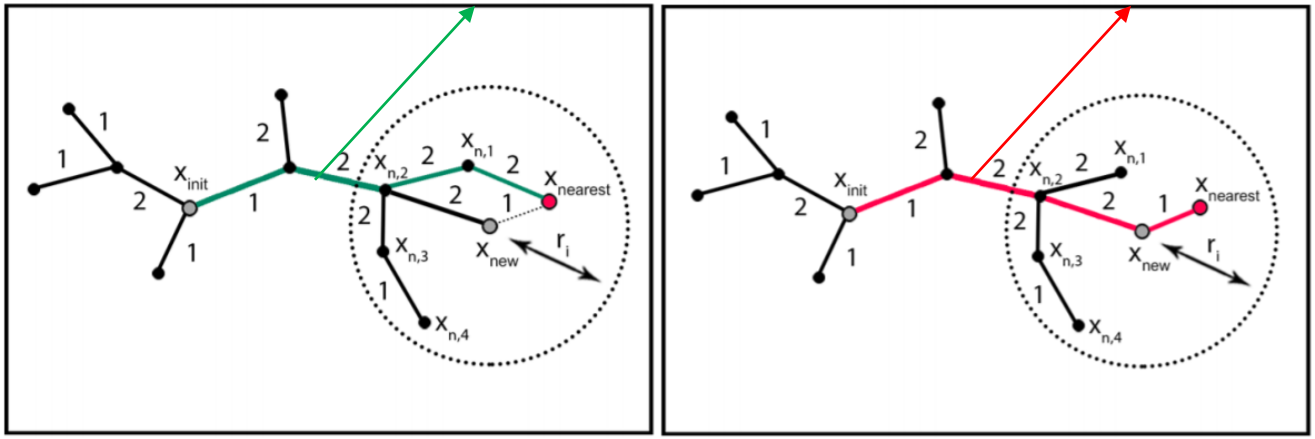
For each node x_{near} in X_{near},
1) check collision-free of the edge (x_{near}, x_{new}).
2) calculate the cost of the path with x_{near} and x_{new}.
3) if there is no collision and the cost is lower than the previous one, update the parent node of the x_{near} as x_{new} (rewire to the more optimal path).

[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Rapidly exploring Random Tree (RRT)

- **Pseudo-code for RRT* [1]**

Path cost: $x_{near}.\text{cost} > x_{new}.\text{cost} + \text{c}(\text{Line}(x_{near}, x_{new}))$



Algorithm 4 Rewire

1: **Input:** X_{near}, E, x_{new}
2: **for** $x_{near} \in X_{near}$ **do**
3: **if** ObstacleFree(x_{new},x_{near}) **and** x_{near}.cost $>$ x_{new}.cost + c(Line(x_{near}, x_{new})) **then**
4: $x_{parent} \leftarrow x_{near}$.parent
5: $E \leftarrow E \setminus \{(x_{parent}, x_{near})\}$
6: $E \leftarrow E \cup \{(x_{new}, x_{near})\}$
7: **end if**
8: **end for**
9: **return** E

For each node x_{near} in X_{near},
1) check collision-free of the edge (x_{near}, x_{new}).
2) calculate the cost of the path with x_{near} and x_{new}.
3) if there is no collision and the cost is lower than the previous one, update the parent node of the x_{near} as x_{new} (rewire to the more optimal path).

[1]: An Overview of the Class of Rapidly-Exploring Random Trees

Q & A