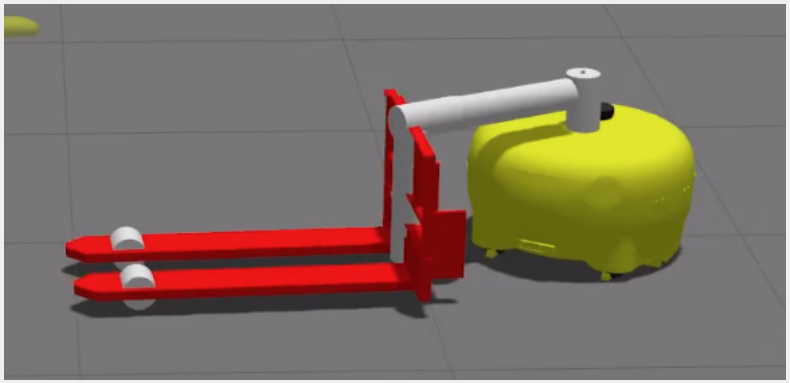
Motion Planning for Autonomous Vehicles

Path planning

Geesara Kulathunga



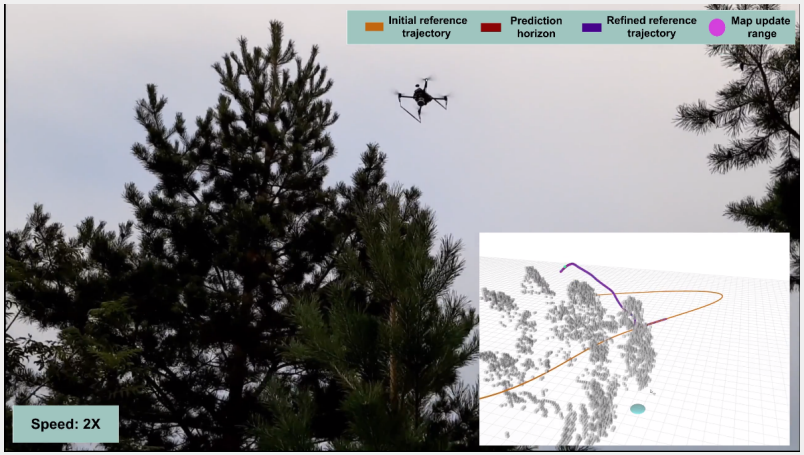
March 11, 2023

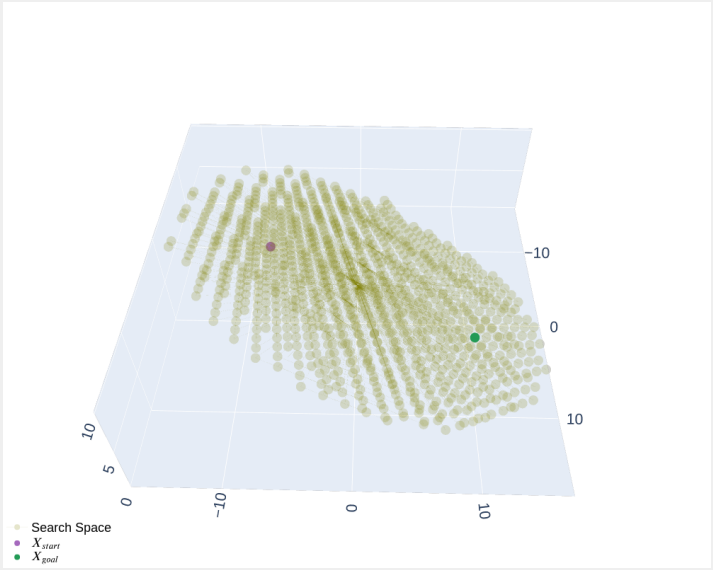
Path planning

Contents

Configuration Space vs Search Space for Robot

The **configuration space** is the space of all possible configurations robot can move which is a subset of the **search space (workspace)**

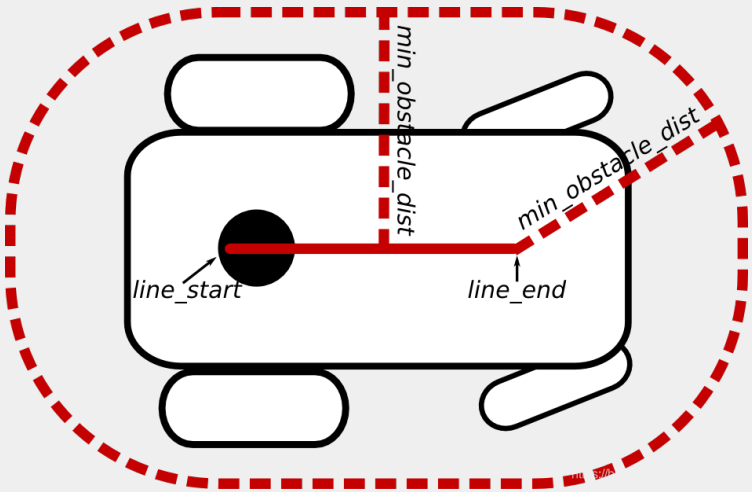


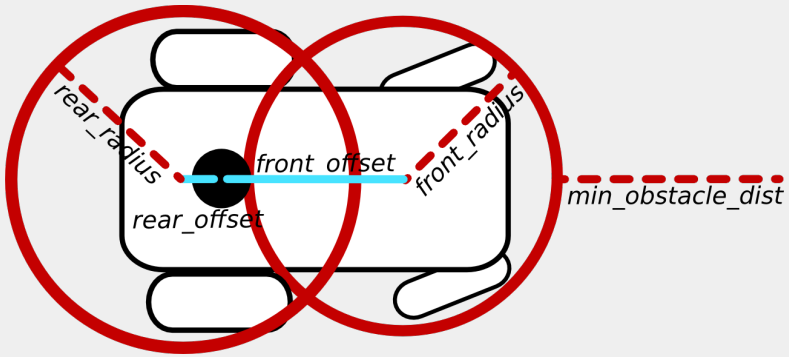


Robot configuration (robot footprint): specification of robot representation. **Robot configuration space**: possible all robot configuration with respect to robot degree of freedom (DOF)

Configuration Space vs Search Space for Robot

Robot configuration (robot footprint): specification of robot representation

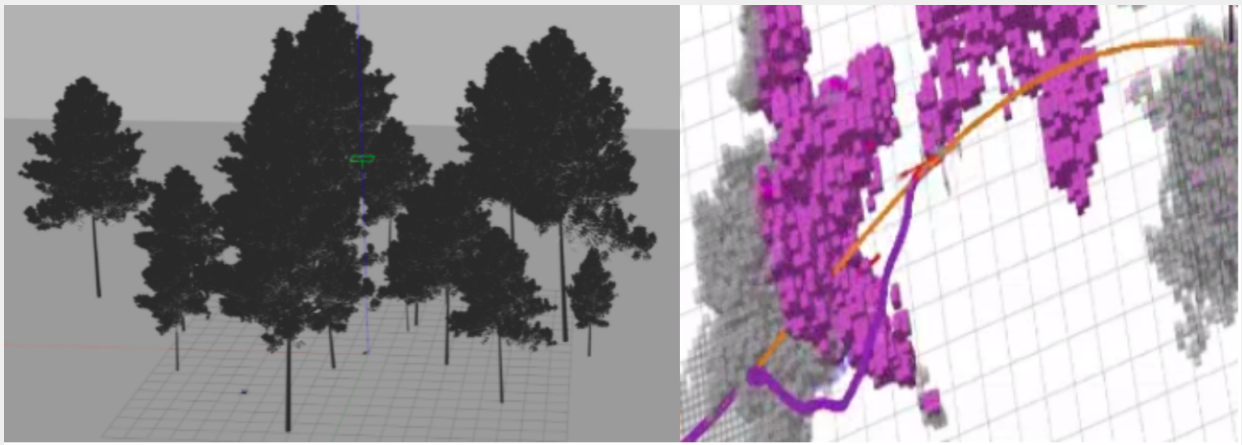




https://www.ncnynl.com/archives/201809/2602.html

Configuration Space for Obstacles

Planning in the workspace is computationally hard and time-consuming: both the robot and obstacles have different shapes



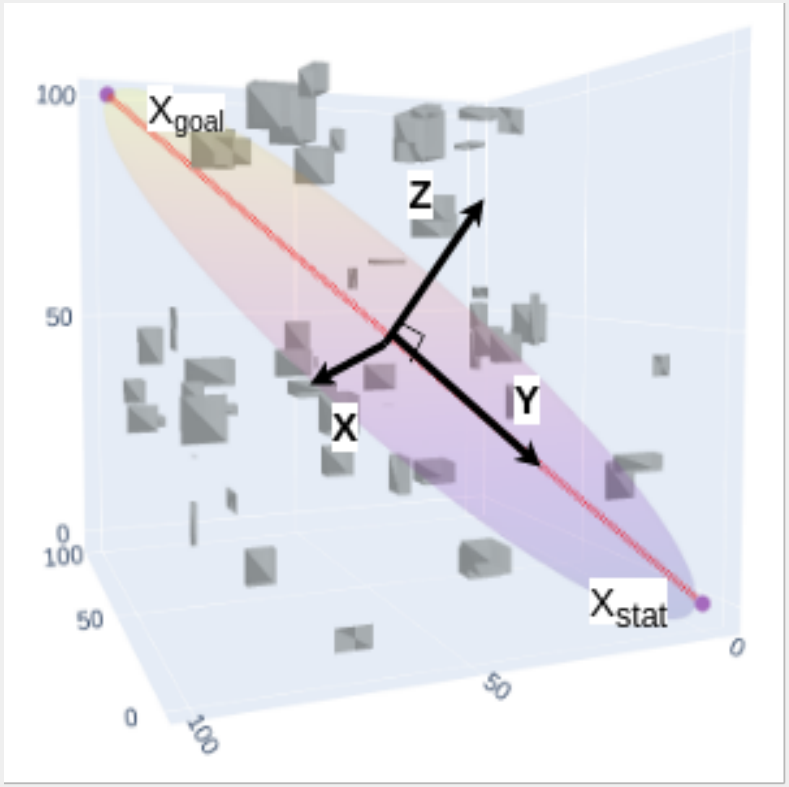
(a) (b)

Obstacle representation in the **configuration space** is extremely complicated. Only **approximations** are applied with known pieces of information and computational capabilities

Path Planning Problem Formulation

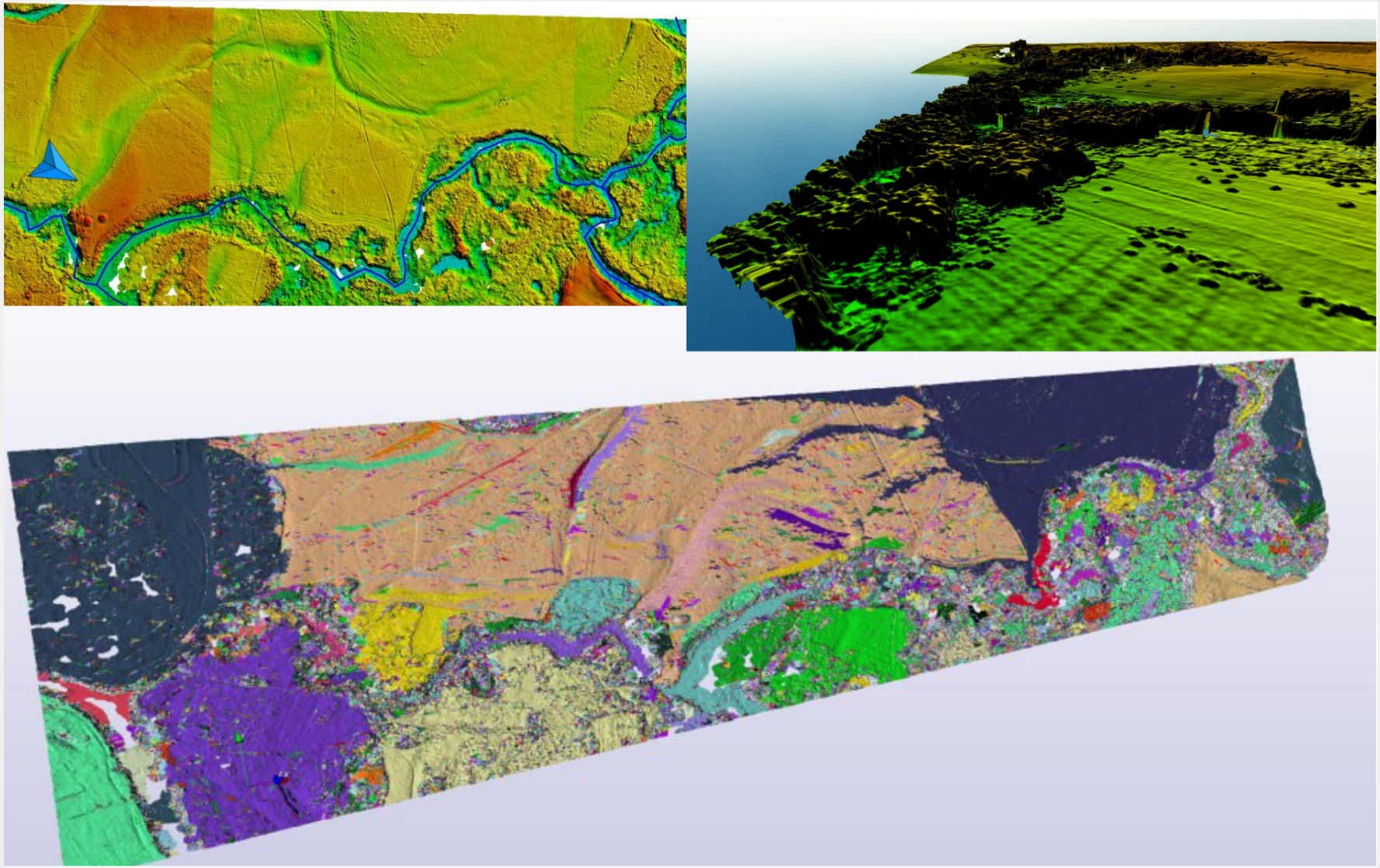
- $X = (0, 1)^d$ as the configuration space where $X \in \mathbb{R}^d$
- X_{obs} is the regions occupied by the obstacles
- $X \setminus X_{obs}$ becomes an open set where $cl(X \setminus X_{obs})$ can be defined as the traversable space (X_{free}) where $cl(.)$ is denotes the closure of set $X \setminus X_{obs}$
- X_{start} and X_{goal} are the start and target position for planning, where $X_{start} \in X_{free}$ and $X_{goal} \in X_{free}$
- $\sigma : [0, 1]^d \rightarrow \mathbb{R}^d$ denotes the waypoints of the planned path, provided that $\sigma(\tau) \in X_{free}$ for all $\tau \in [0, 1]$; This path is called as a obstacle free path

Path Planning Problem Formulation



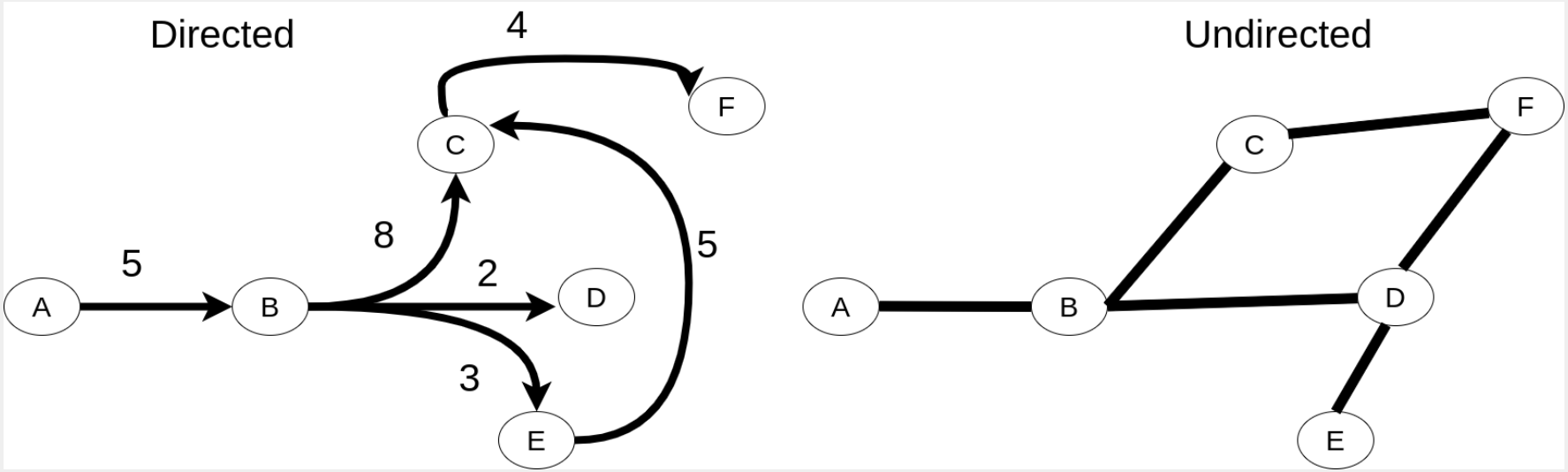
What kind of things do you have to consider for defining $cl(X \setminus X_{obs})$ in this workspace?

Search-based Planning : Mapping



Search-based Planning : Graph

Graphs have edges and **nodes** whose **connectivity** is defined by **directed or underacted edges**. In motion planning, the mathematical representation of the search-based path planning algorithm is called **state space graph**



Search tree: searching the graph in a defined way produces a tree. Back-tracing a node in the search tree gives a path/paths from start to end node

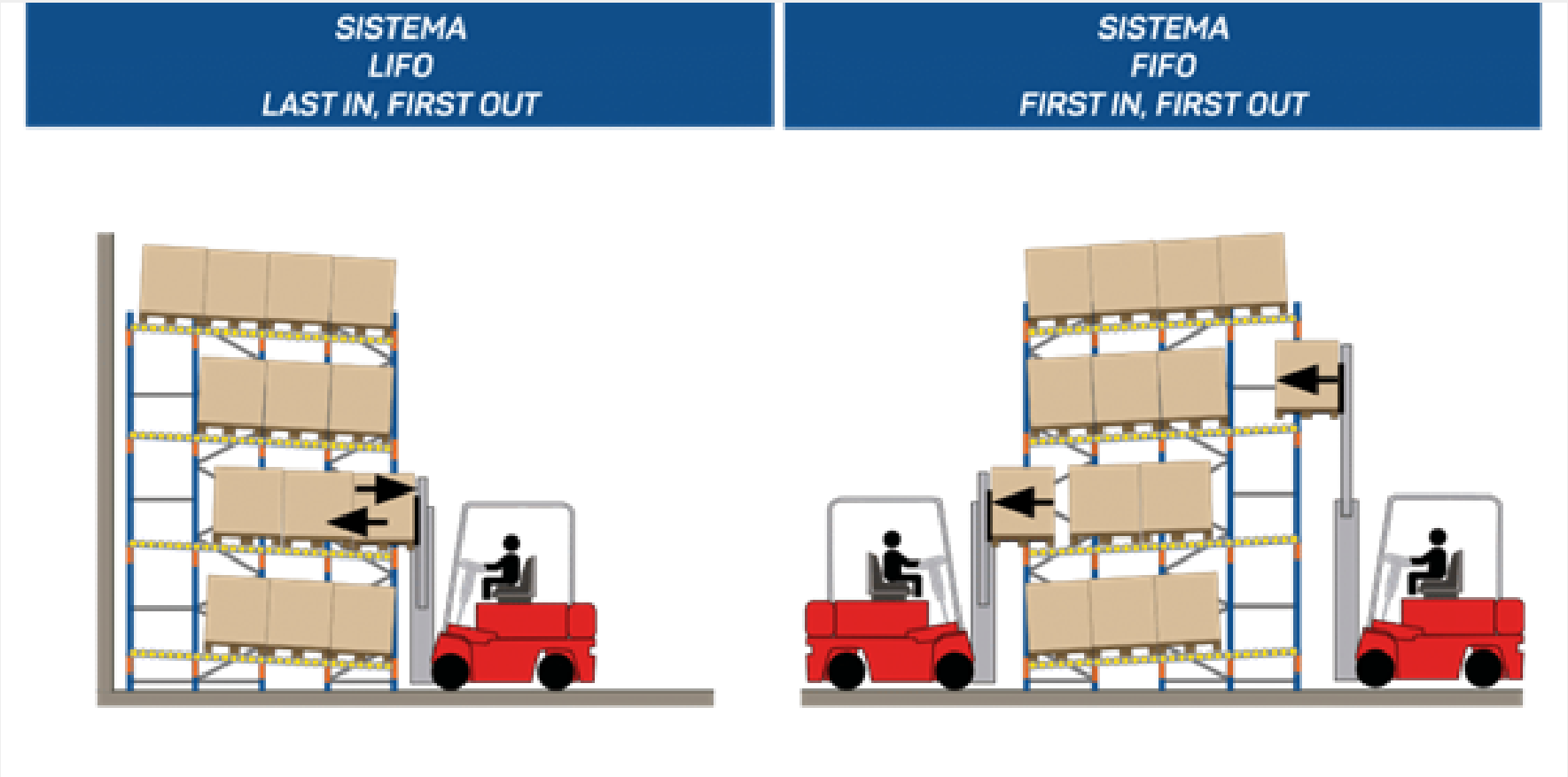
Graph Searching

- Need a **container** to keep all the nodes that are **needed to explore**
- Starts the container with starting node X_{start}
- Repeat
  - Visit **each node from the container** and **remove** it if it does not match with **defined constraints**, e.g., obstacle-free or obstacle-occupied
  - Check the desired constraint, if so terminate searching
  - Node whose constraints are satisfied, **obtain neighbours** of it
  - **Push** back **neighbours** back into the **container**

Graph Searching

1. When to **terminate the search**? in the case of the **empty container** or **met the desired constraint** to terminate the search.
2. What if the **graph** is **cyclic**? when a **node is removed** from the container **it should never be considered**
3. **What are the ways** to **remove the right node** such that the search meets the **desired constraint** as **soon** as it can

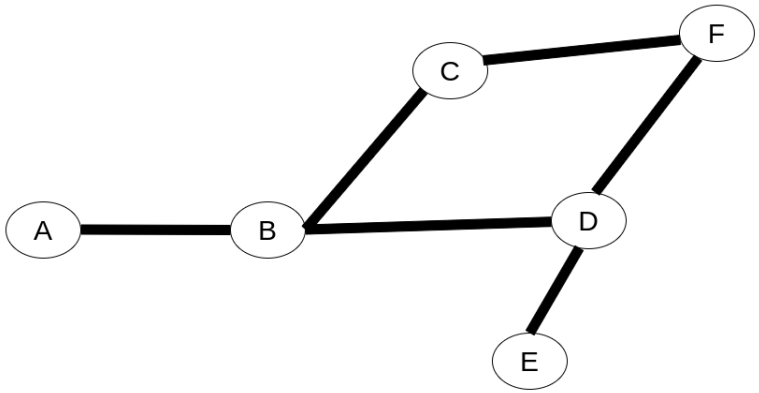
Graph Searching



https://controlinventarios.files.wordpress.com/2021/11/image-10.png

Graph Searching: Depth First Search

The strategy is to expand the **deepest node** in the container



Algorithm 1 Depth first search

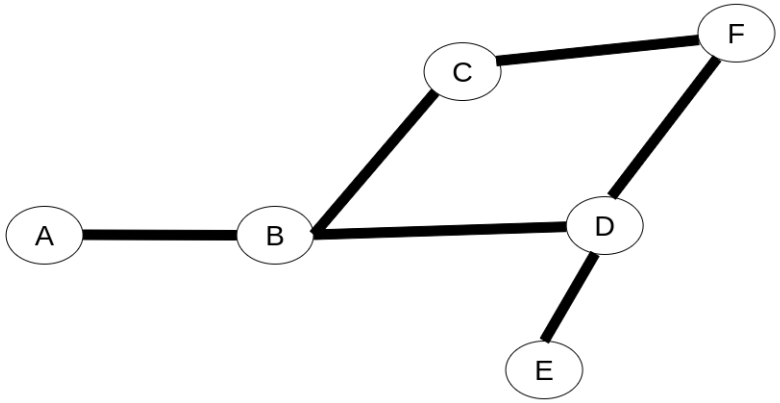
```
1: procedure DFS(G, n)                    ▷ The graph G, starting node n
2:     n.visited ← true
3:     for eache ∈ G.neighbours(n) do
4:         if e.visited == false then
5:             dfs(G, v)
6:         end if
7:     end for
8: end procedure
```

step 1	visited	A
	container	B
step 2	visited	A, B
	container	C, D
step 3	visited	A, B, C
	container	F, D
step 4	visited	A, B, C, F
	container	D
step 5	visited	A, B, C, F, D
	container	E
step 6	visited	A, B, C, F, D,E
	container	

Uses Stack data structure

Graph Searching: Breath First Search

The strategy is to expand the **shallowest node** in the container



Algorithm 2 Breath first search

```
1:  procedure BFS(G, n)
2:      container ← []                          ▷ The graph G, starting node n
3:      n.visited ← true
4:      container.push(n)
5:      while container.empty() do
6:          v ← container.pop()
7:          for each e ∈ G.neighbours(v) do
8:              if e.visited == false then
9:                  container.push(e)
10:             end if
11:         end for
12:     end while
13: end procedure
```

step 1	visited	A
	container	B
step 2	visited	A, B
	container	C, D
step 3	visited	A, B, C
	container	D, F
step 4	visited	A, B, C, D
	container	F, E
step 5	visited	A, B, C, D, F
	container	E
step 6	visited	A, B, C, D,F, E
	container	

Graph Searching: Breath First Search

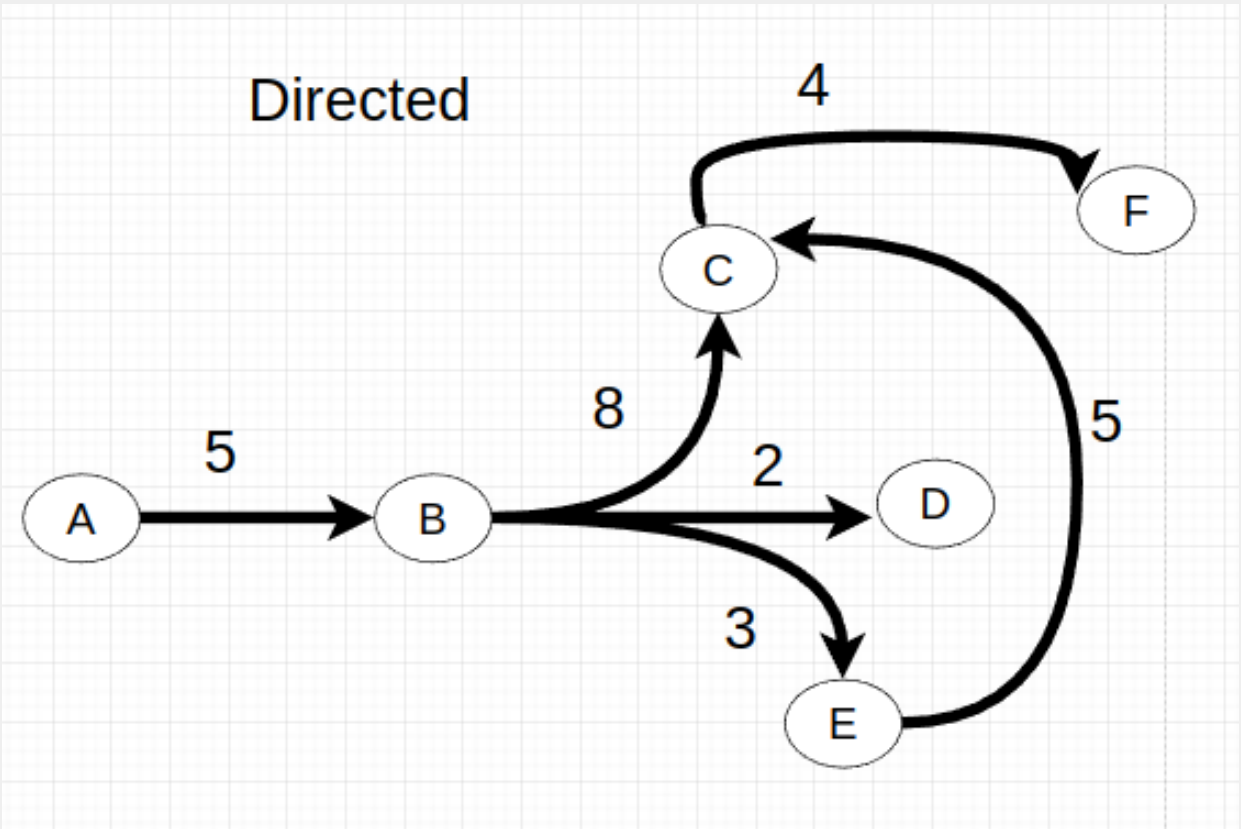
Breath first search can also be used for **flow field path-finding, distance map (or cost map)** generation, and much more.

Algorithm 3 Breath first search

```
1:  procedure BFS(G, n, g, dist)
2:      openset ← []      ▷ The graph G, starting node n, goal node g, minimum
    allowable residual dist
3:      n.id ← will_be
4:      openset.insert(n)
5:      while openset.empty() do
6:          v ← container.pop()
7:          v.id ← was_there
8:          if dist < |v − g| then
9:              terminate ← v
10:             return
11:         end if
12:         for each e ∈ G.neighbours(v) do
13:             if e.id! = was_there then
14:                 e.parent ← v
15:                 e.id ← will_be
16:                 openset.push(e)
17:             end if
18:         end for
19:     end while
20: end procedure
```

Graph Searching: Costs Consideration

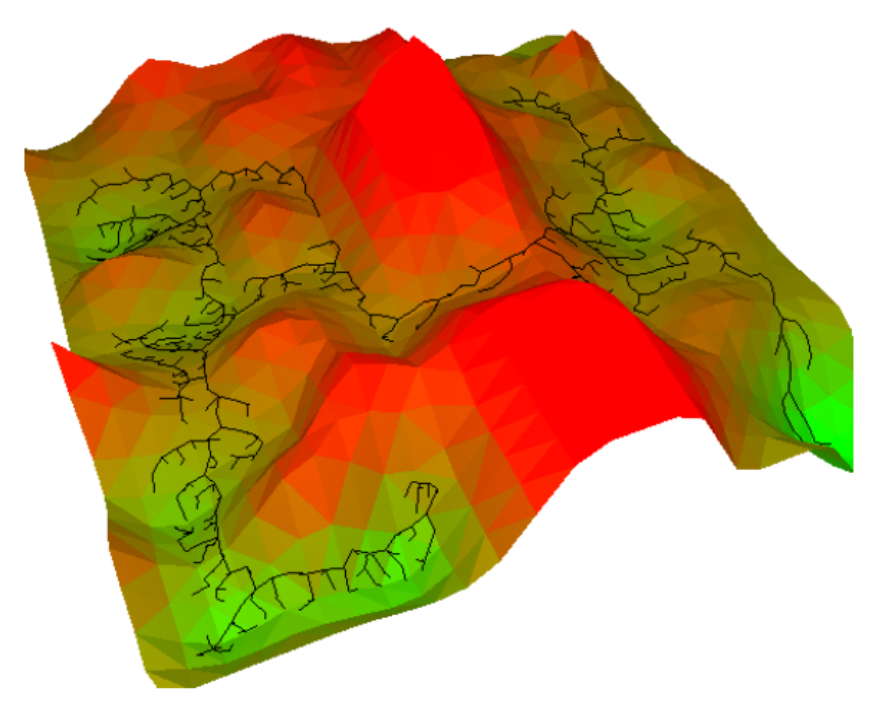
In real-world scenarios, different types of costs, e.g., length, time, and energy, are associated with nodes. If all **costs are equal**, the **breath first search** gives **the optimal solution**.



How can we find the **least-cost path solution**?

Graph Searching: Costs Consideration

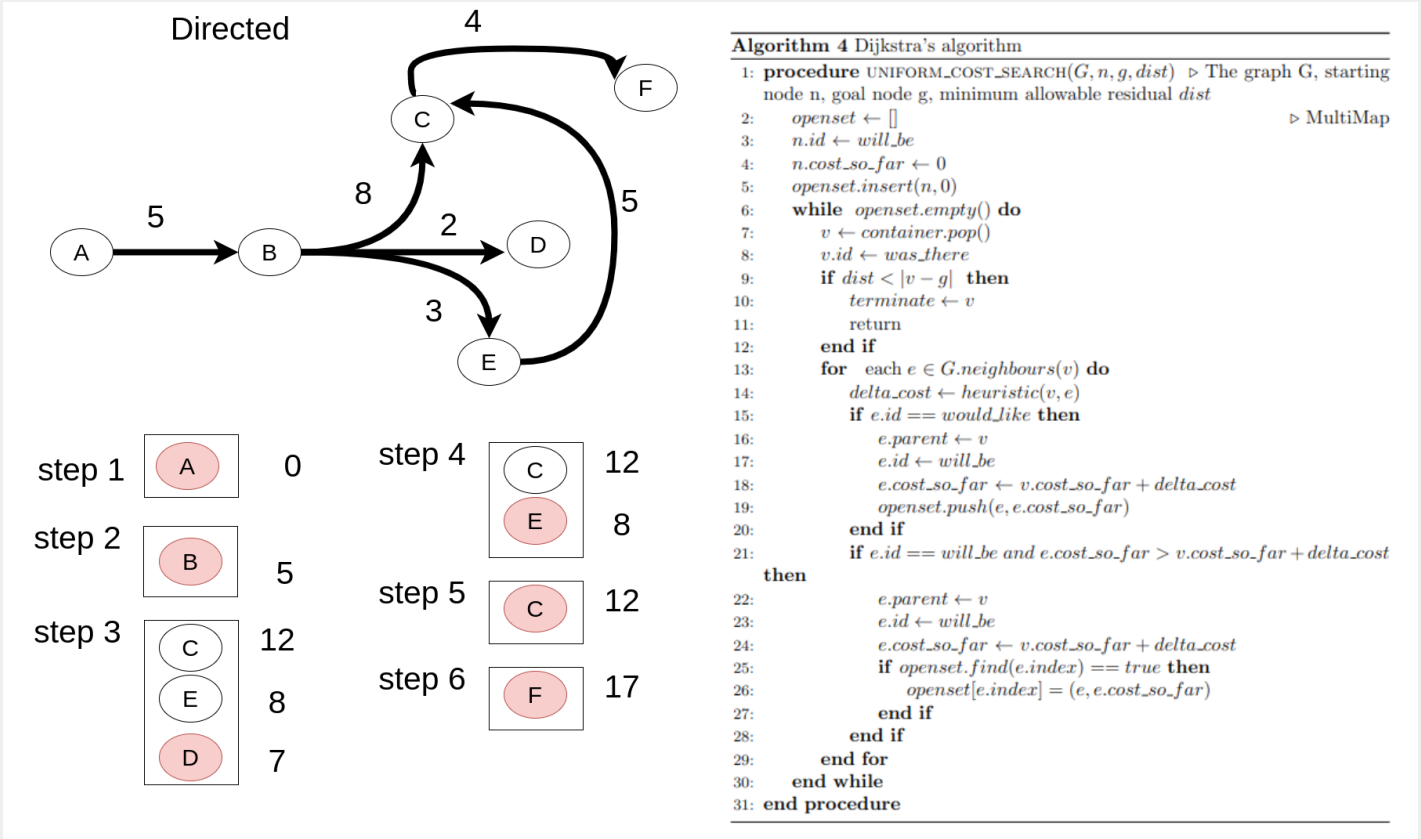
The strategy is to expand the node with **cheapest accumulated cost g(v)**, where g(v) is the best-accumulated cost from the start node to node v. **Node** that has been **expanded** ensures having the **minimum cost from the start node**



Jaillet, L., Cortes, J., Simeon, T. (2008, September). Transition-based RRT for path planning in continuous cost spaces. In 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems (pp. 2145-2150). IEEE.

Graph Searching: Dijkstra's Algorithm

The algorithm is **complete and optimal**



Algorithm 4 Dijkstra's algorithm

```
1: procedure UNIFORM_COST_SEARCH(G, n, g, dist)  ▷ The graph G, starting node n, goal node g, minimum allowable residual dist
2:     openset ← []                                                    ▷ MultiMap
3:     n.id ← will_be
4:     n.cost_so_far ← 0
5:     openset.insert(n, 0)
6:     while openset.empty() do
7:         v ← container.pop()
8:         v.id ← was_there
9:         if dist < |v − g| then
10:            terminate ← v
11:            return
12:        end if
13:        for each e ∈ G.neighbours(v) do
14:            delta_cost ← heuristic(v, e)
15:            if e.id == would_like then
16:                e.parent ← v
17:                e.id ← will_be
18:                e.cost_so_far ← v.cost_so_far + delta_cost
19:                openset.push(e, e.cost_so_far)
20:            end if
21:            if e.id == will_be and e.cost_so_far > v.cost_so_far + delta_cost
   then
22:                e.parent ← v
23:                e.id ← will_be
24:                e.cost_so_far ← v.cost_so_far + delta_cost
25:                if openset.find(e.index) == true then
26:                    openset[e.index] = (e, e.cost_so_far)
27:                end if
28:            end if
29:        end for
30:    end while
31: end procedure
```

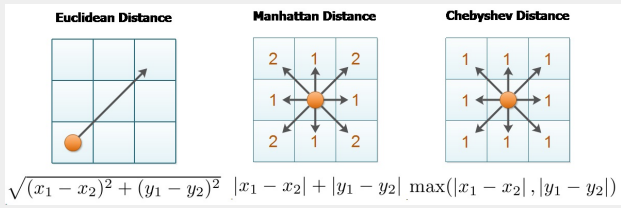
The algorithm incorporates **only the accumulated cost till the considered node**, and **every direction** has to incorporate to make the **next move**. **No information** about the **goal location**

Graph Searching: Greedy Best First Search

The algorithm incorporates the **cost to the goal** h(n) with heuristic cost estimation, e.g., Manhattan distance, Euclidean distance. Hence, start **exploring towards goal direction**

Algorithm 5 Greedy breath first search

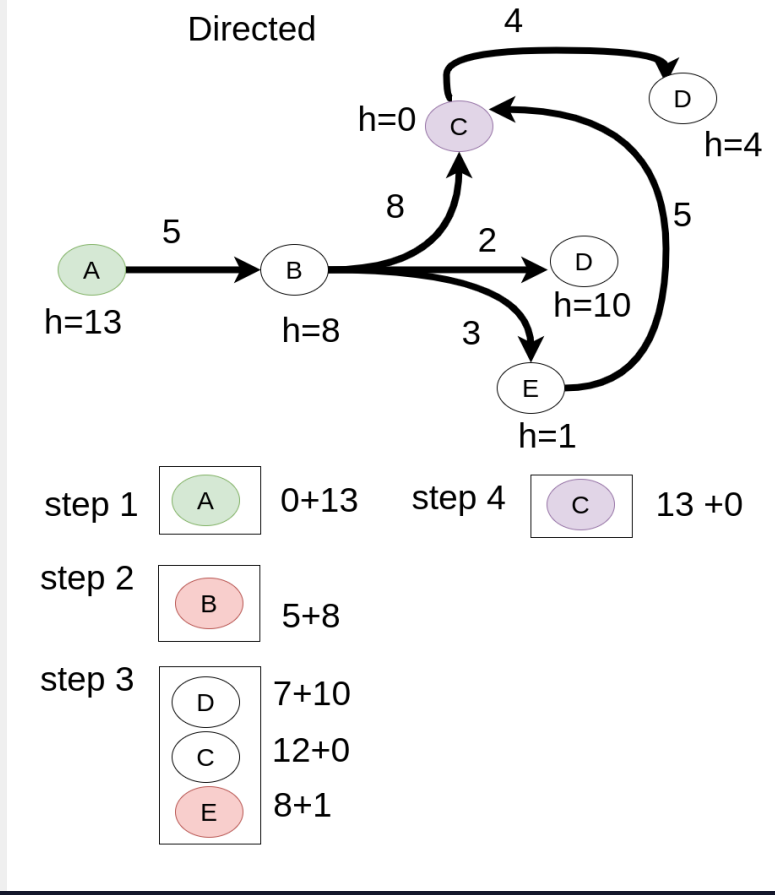
```
1:  procedure GBFS(G, n, g, dist)   ▷ The graph G, starting node n, goal node
    g, minimum allowable residual dist
2:      openset ← []                                          ▷ Priority queue
3:      n.id ← will_be
4:      n.cost_to_go ← 0
5:      openset.insert(n, 0)
6:      while openset.empty() do
7:          v ← container.pop()
8:          if dist < |v − g| then
9:              terminate ← v
10:             return
11:         end if
12:         v.id ← was_there
13:         for each e ∈ G.neighbours(v) do
14:             if e.id! = was_there then
15:                 e.parent ← v
16:                 e.id ← will_be
17:                 e.cost_to_go ← heuristic(g, e)
18:                 openset.push(e, e.cost_to_go)
19:             end if
20:         end for
21:     end while
22: end procedure
```



https://iq.opengenus.org/euclidean-vs-manhattan-vs-chebyshev-distance/

A*: Combination of Greedy Best First Search and Dijkstra's Algorithm

The strategy to expand the node with the **cheapest cost = g(n) + h(n)**, g(n) **accumulated cost** from the start node to the node n and h(n) estimated **heuristic cost** from node n **to the goal node**

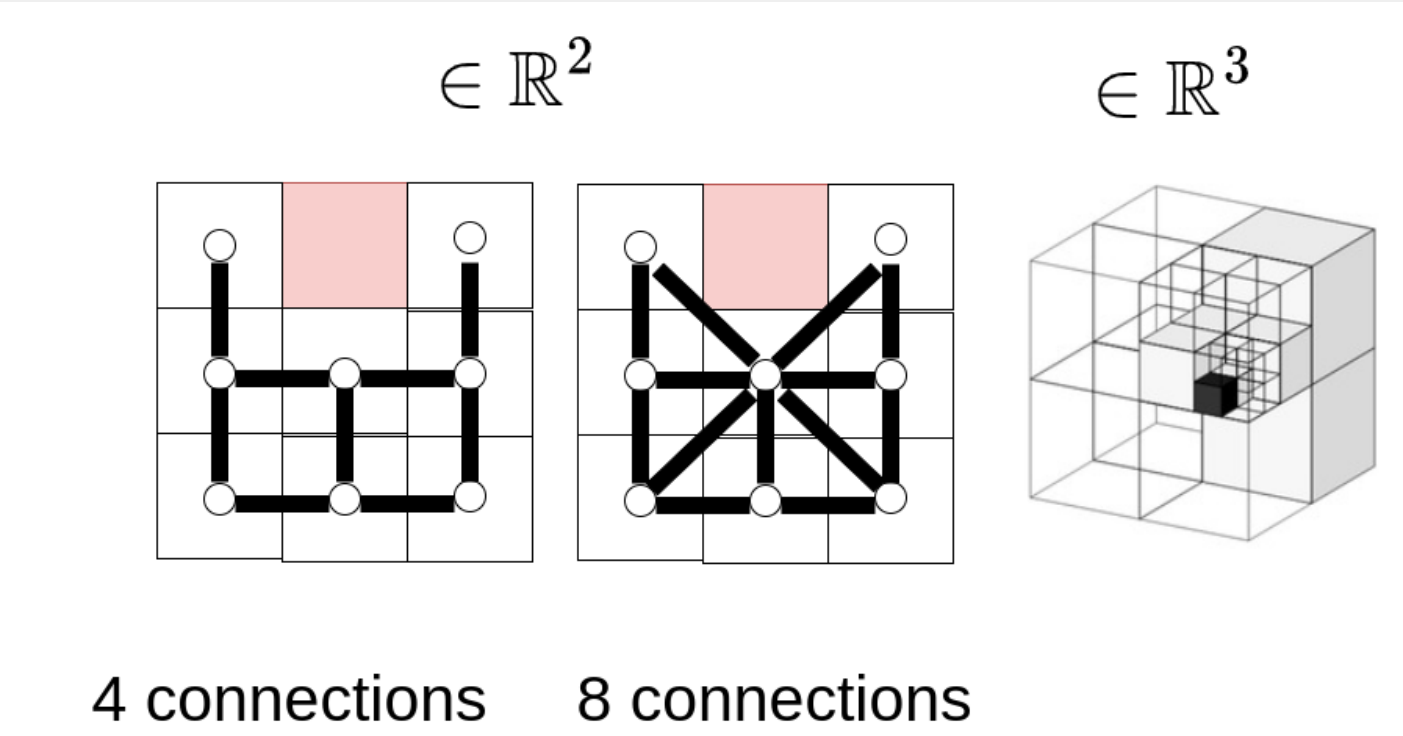


Algorithm 6 AStar algorithm

```
1: procedure ASTAR(G, n, g, dist) ▷ The graph G, starting node n, goal node g, minimum allowable residual dist
2:     openset ← []                                                    ▷ MultiMap
3:     n.id ← will_be
4:     n.cost_so_far ← 0
5:     n.cost_so_go ← heuristic(n, g)
6:     openset.insert(n, n.cost_so_go)
7:     while openset.empty() do
8:         v ← container.pop()
9:         v.id ← was_there
10:        if dist < |v − g| then
11:            terminate ← v
12:            return
13:        end if
14:        for each e ∈ G.neighbours(v) do
15:            delta_cost ← heuristic(v, e)
16:            if e.id == would_like then
17:                e.parent ← v
18:                e.id ← will_be
19:                e.cost_so_far ← v.cost_so_far + delta_cost
20:                e.cost_so_go ← e.cost_so_far + heuristic(e, g)
21:                openset.push(e, e.cost_so_go)
22:            end if
23:            if e.id == will_be and e.cost_so_far > v.cost_so_far + delta_cost
   then
24:                e.parent ← v
25:                e.id ← will_be
26:                e.cost_so_far ← v.cost_so_far + delta_cost
27:                e.cost_so_go ← e.cost_so_far + heuristic(e, g)
28:                if openset.find(e.index) == true then
29:                    openset[e.index] = (e, e.cost_so_go)
30:                end if
31:            end if
32:        end for
33:    end while
34: end procedure
```

A*: Design Consideration

- The heuristic h is admissible if $f = h(n) \leq h^*(n)$ for all "n", where $h^*(n)$ is the least cost from node n to the goal node.
- The cost to go and the cost so far can be weighted in a defined way, e.g., closer to the goal $g(n) + \alpha h,\ \alpha > 1$. This strategy is called weighted A*.



A*: Design Consideration

- Priority queue in c++ can be implemented in several ways
  - std::priority_queue
  - std::make_heap
  - std::multimap
- If more than one close-by node has the same f value, compare their h values and add a deterministic random value to the h, this idea is called **Tie Breaker**

$$\begin{aligned} \delta x_1 &= |n.x - g.x|, \quad \delta y_1 = |n.y - g.y| \\ \delta x_2 &= |s.x - g.x|, \quad \delta y_2 = |s.y - g.y| \end{aligned} \tag{1}$$

$$cross = |\delta x_1 \times \delta y_2 - \delta x_2 \times \delta y_1| h = h + cross \times 0.001$$

there are many ways to modify these heuristics. One of the recently proposed approach: **Jump Point Search (JPS)[1]**

[1]. Harabor, D., Grastien, A. (2014, May). Improving jump point search. In Proceedings of the International Conference on Automated Planning and Scheduling (Vol. 24, pp. 128-135).

Graph-based search problem classification

- **Search problem** can be seen in two different ways: **Uninformed** or **Informed search problem**. In an uninformed search besides, the problem definition no further definition is provided (e.g., **breadth-first, depth-first**, etc), On the contrary, an informed search for future information pursues deterministic or heuristic way (**A*, JPS, D***, etc)
- Performance of the search problem can be estimated in four different ways[1]
  - **Completeness**: does the algorithm find the solution when there is one?
  - **Optimality**: is the solution the best one of all possible solutions in terms of path cost?
  - **Time complexity**: how long does it take to find a solution?
  - **Space complexity**: how much memory is needed to perform the search?

[1]. http://ais.informatik.uni-freiburg.de/teaching/ss11/robotics/slides/18-robot-motion-planning.pdf

KinoDynamic A*:Heuristics

- In general, design a trajectory x(t) such that x_o = a, and x_T = b whose order of **degree five**, which is called a **quintic polynomial**:

$$x(t) = a_o + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \tag{2}$$

KinoDynamic A*:Heuristics

- In general, design a trajectory x(t) such that x_0 = a, and x_T = b whose order of **degree five**, which is called a **quintic polynomial**:

$$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \tag{2}$$

- Consider the initial condition $x_0 = a, \dot{x}_0 = 0, \ddot{x}_0 = 0$ at $t = 0$ and final condition $x_T = b, \dot{x}_T = 0, \ddot{x}_T = 0$ at $t = T$ are given.

KinoDynamic A*:Heuristics

- In general, design a trajectory x(t) such that x_0 = a, and x_T = b whose order of **degree five**, which is called a **quintic polynomial**:

$$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \tag{2}$$

- Consider the initial condition $x_0 = a, \dot{x}_0 = 0, \ddot{x}_0 = 0$ at $t = 0$ and final condition $x_T = b, \dot{x}_T = 0, \ddot{x}_T = 0$ at $t = T$ are given.
- Hence, the objective is to find the optimal value of T

$$a_0 = x_0, \quad a_1 = \dot{x}_0, \quad a_2 = \ddot{x}_0/2$$

$$A = \begin{bmatrix} T^3 & T^4 & T^5 \\ 3T^2 & 4T^3 & 5T^4 \\ 6T & 12T^2 & 20T^3 \end{bmatrix}, \quad b = \begin{bmatrix} x_f - a_0 - a_1 - a_2 T^2 \\ \dot{x}_f - a_1 - 2a_2 T \\ \ddot{x}_f - 2a_2 \end{bmatrix} \Rightarrow \begin{bmatrix} b - a \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} a_3 \\ a_4 \\ a_5 \end{bmatrix} = A^{-1} b \tag{3}$$