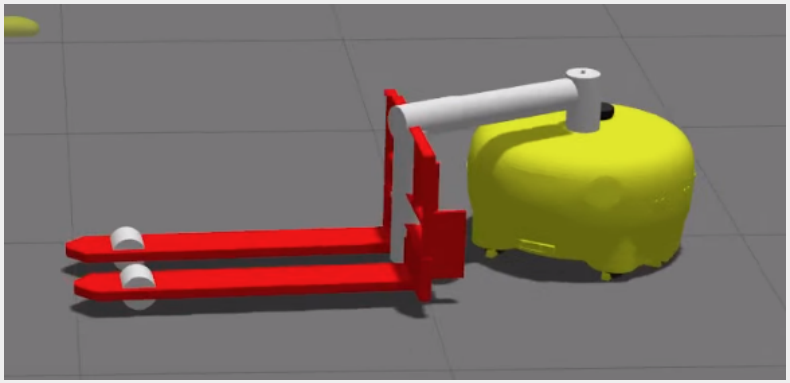
Motion Planning for Autonomous Vehicles

Path planning

Geesara Kulathunga



March 10, 2023

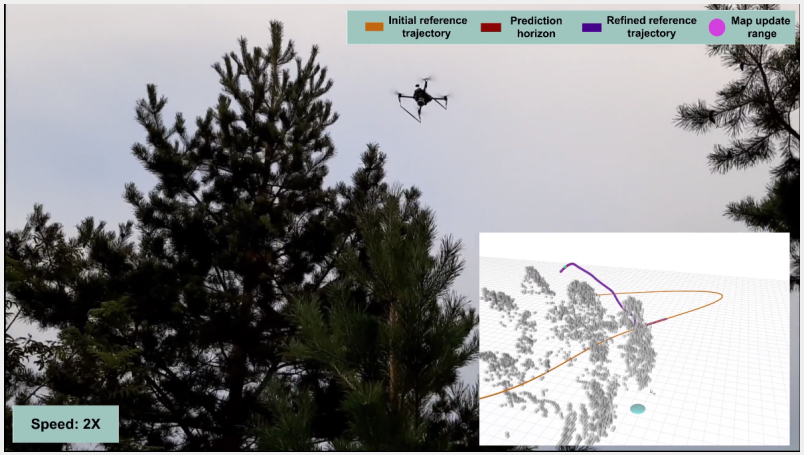
Path planning

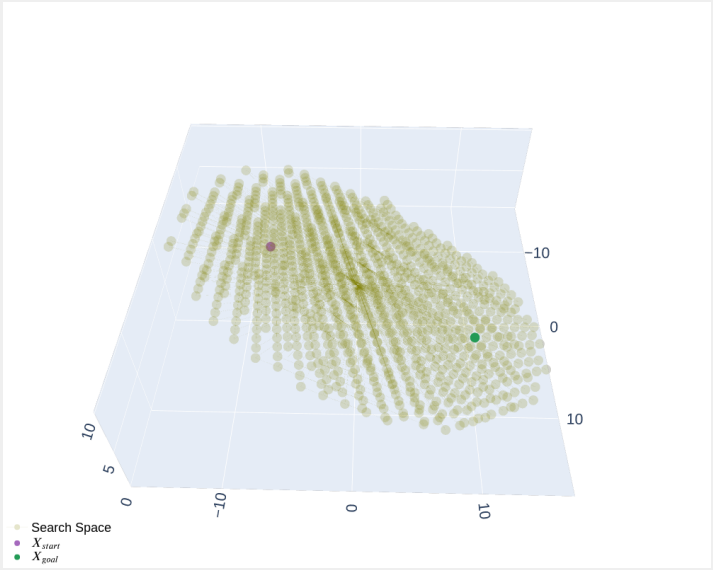
Contents

- Configuration Space vs Search Space for Robot
- Path Planning Problem Formulation
- Search-based Planning: Mapping
- Search-based Planning: Graph
- Graph Searching
  - Depth First Search
  - Breath First Search
  - Cost Consideration
  - Dijkstra's Algorithm
  - Greedy Best First Search
  - A*: Combination of Greedy Best First Search and Dijkstra's Algorithm
  - A*: Design Consideration
  - Graph-based search problem classification
  - KinoDynamic A*:Heuristics, Generating motion primitives, finding neighbours
  - Hybrid A*: Motion model, finding neighbours, the cost to go h, and cost so far g

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)**

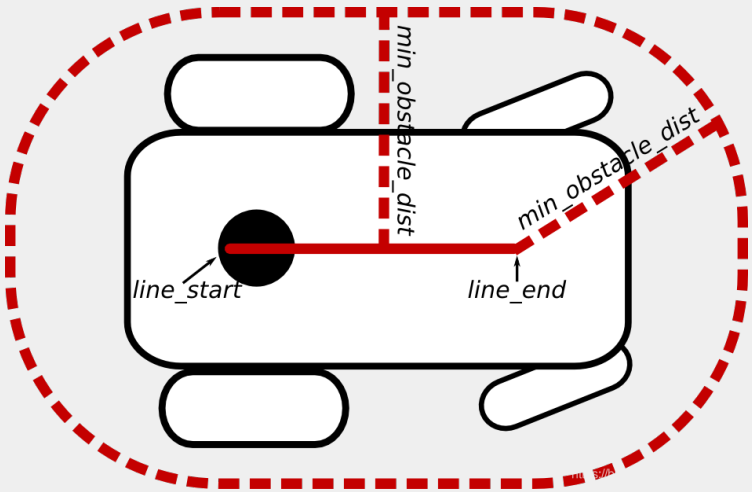


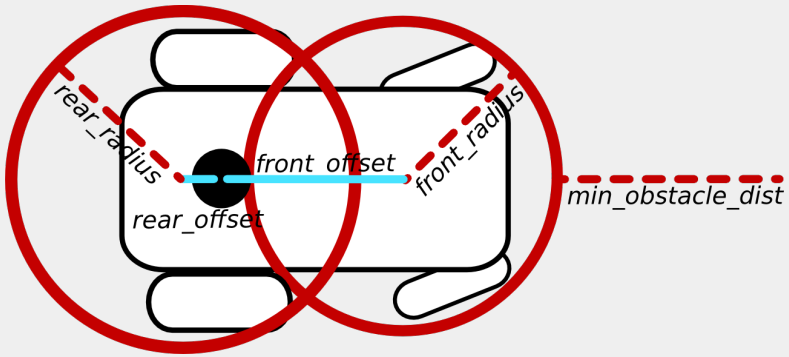


Robot configuration (robot footprint): specification of robot representation. **Robot configuration space**: possible all robot configuration with respect to robot degree of freedom (DOF)

Configuration Space vs Search Space for Robot

Robot configuration (robot footprint): specification of robot representation

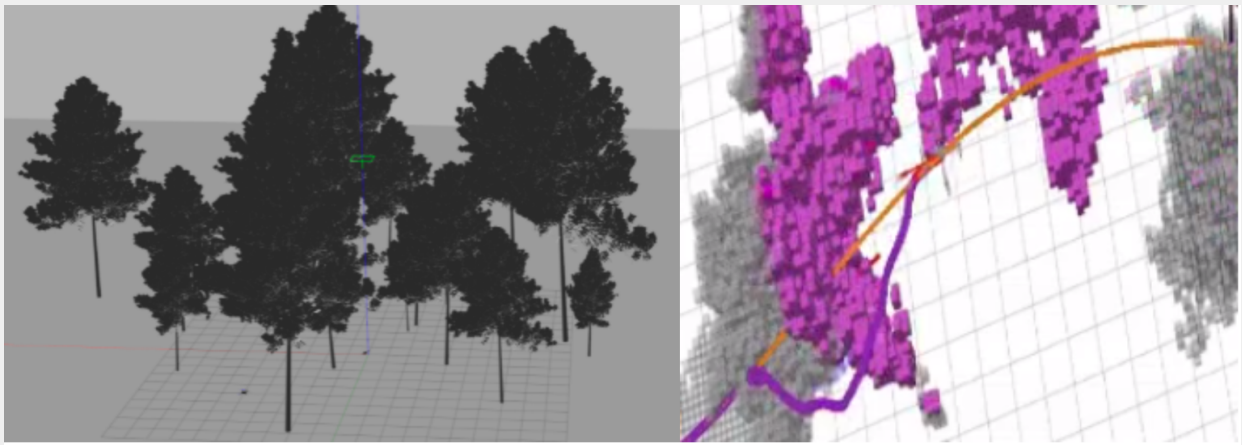




https://www.ncnynl.com/archives/201809/2602.html

Configuration Space for Obstacles

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



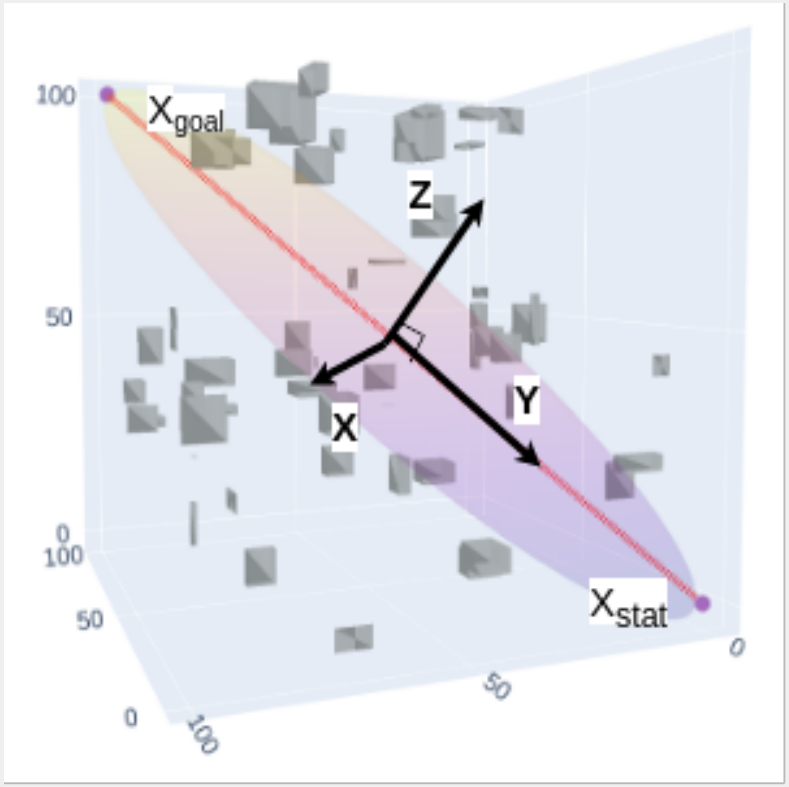
(a) (b)

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

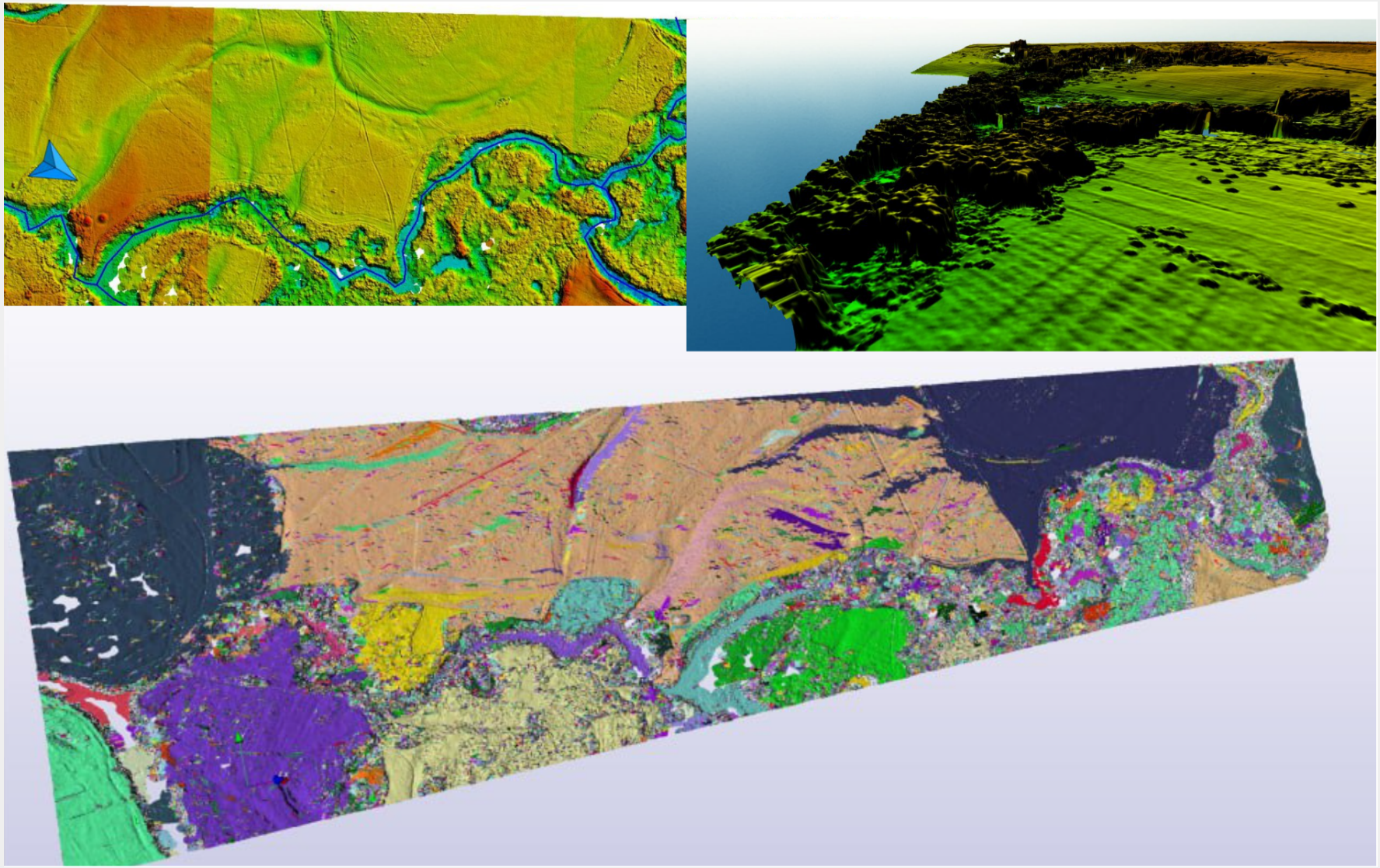
- $X = (0, 1)^d$ as the configuration space where $X \in \mathbb{R}^d$
- X_{obs} is the regions occupied by the obstacles
- $X \setminus X_{obs}$ becomes an open set where $cl(X \setminus X_{obs})$ can be defined as the traversable space (X_{free}) where $cl(.)$ is denotes the closure of set $X \setminus X_{obs}$
- 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



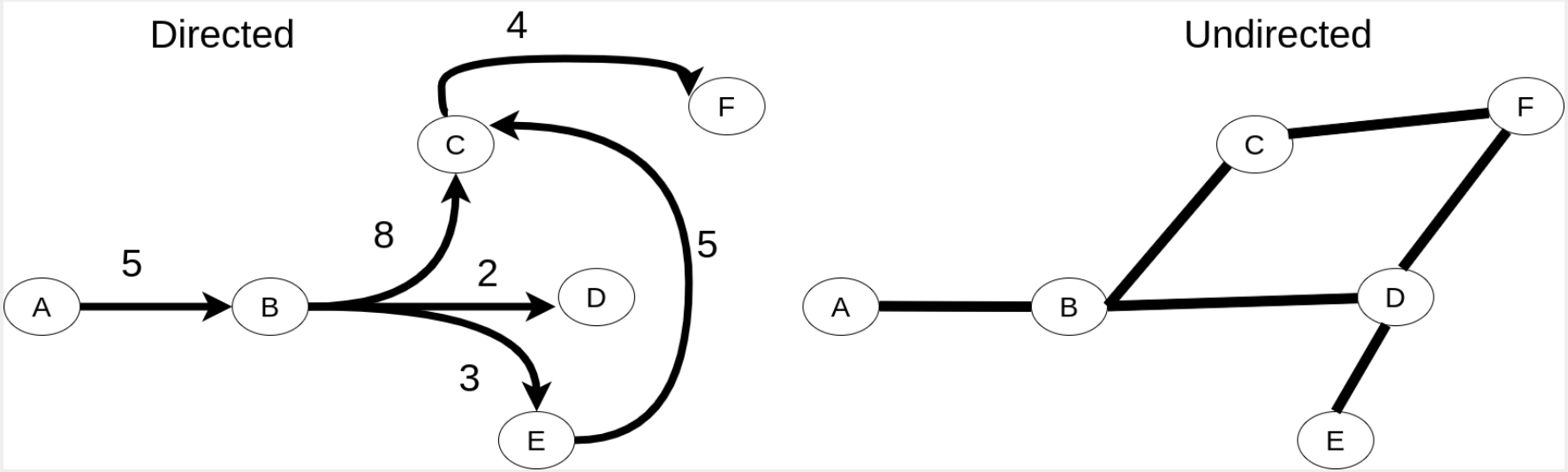
What kind of things do you have to consider for defining $cl(X \setminus X_{obs})$ in this workspace?

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

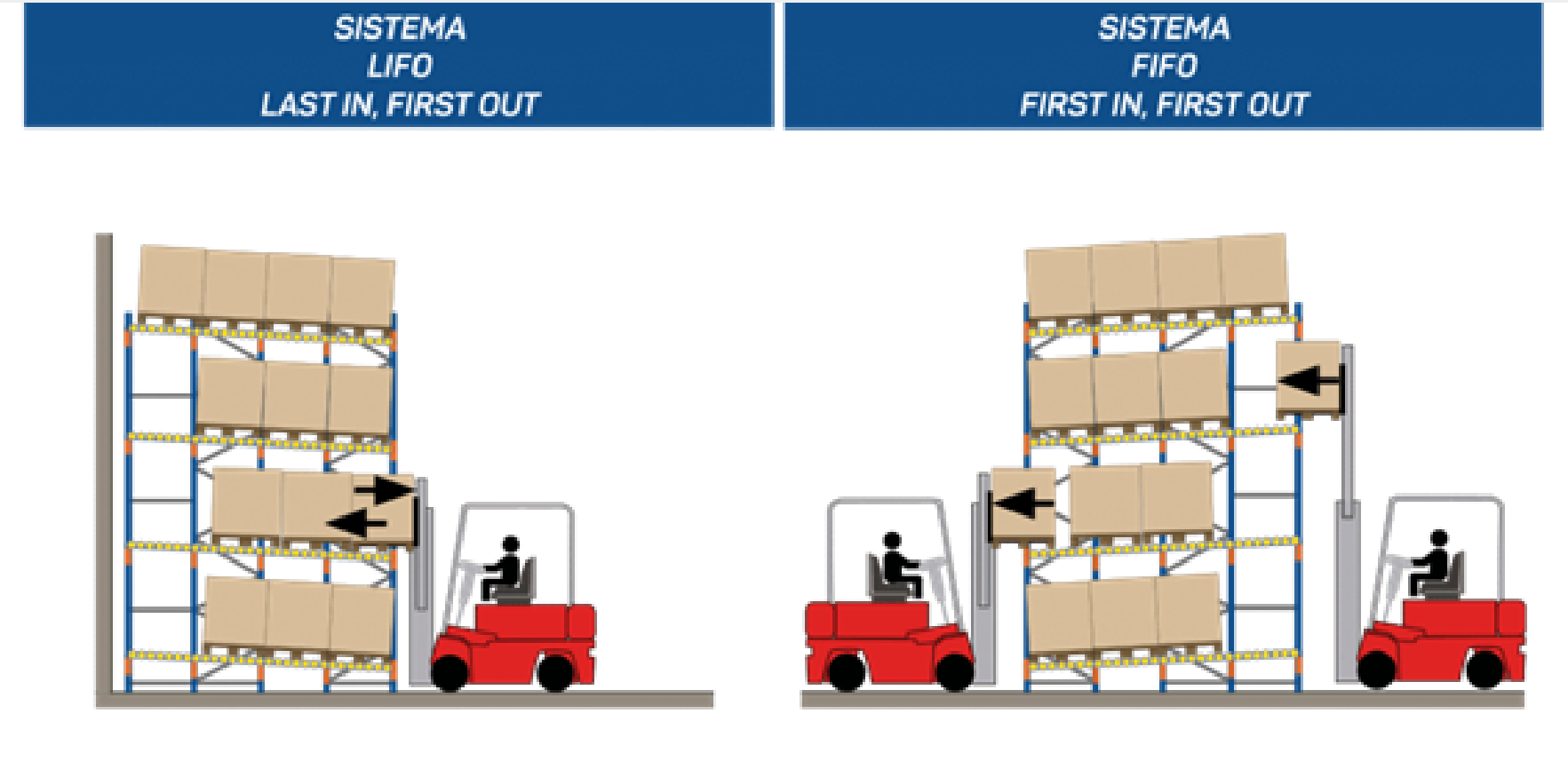
Graph Searching

- 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

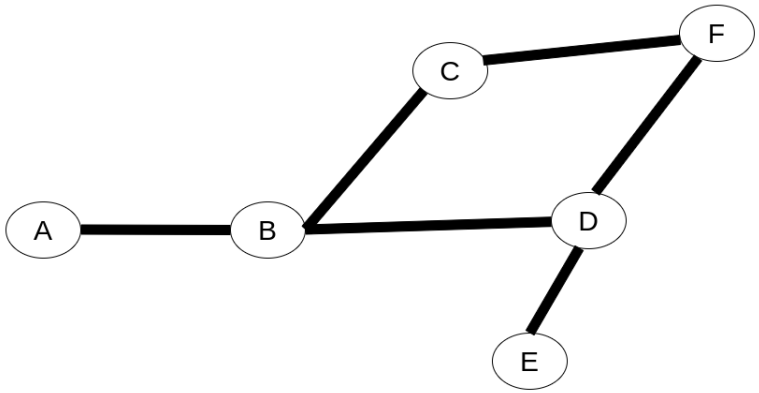
Graph Searching



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

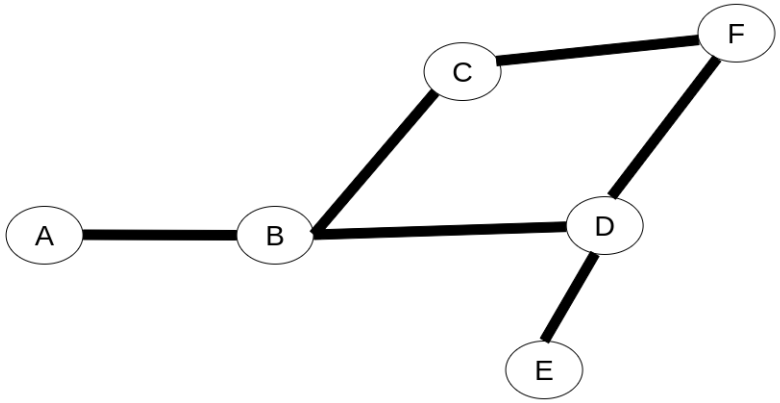
```
1: procedure DFS(G, n)                    ▷ The graph G, starting node n
2:     n.visited ← true
3:     for eache ∈ G.neighbours(n) do
4:         if e.visited == false then
5:             dfs(G, v)
6:         end if
7:     end for
8: end procedure
```

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
```