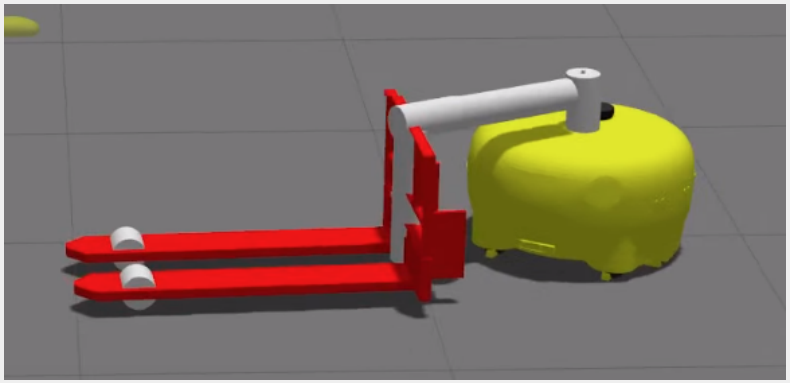
Motion Planning for Autonomous Vehicles

Path planning

Geesara Kulathunga



March 17, 2023

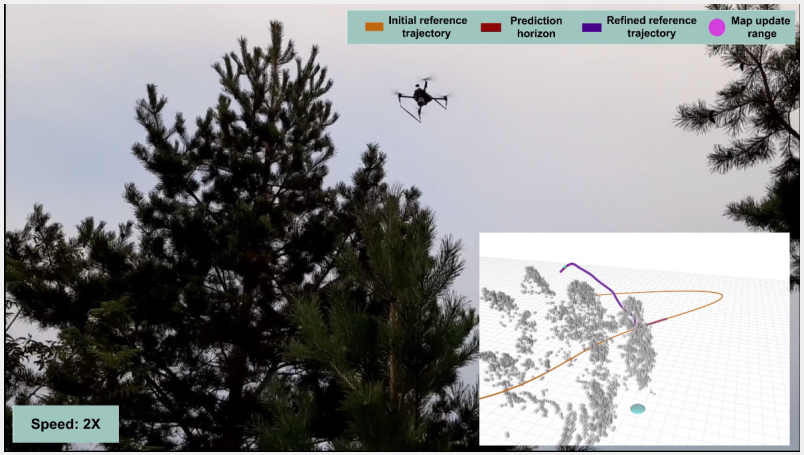
Path planning

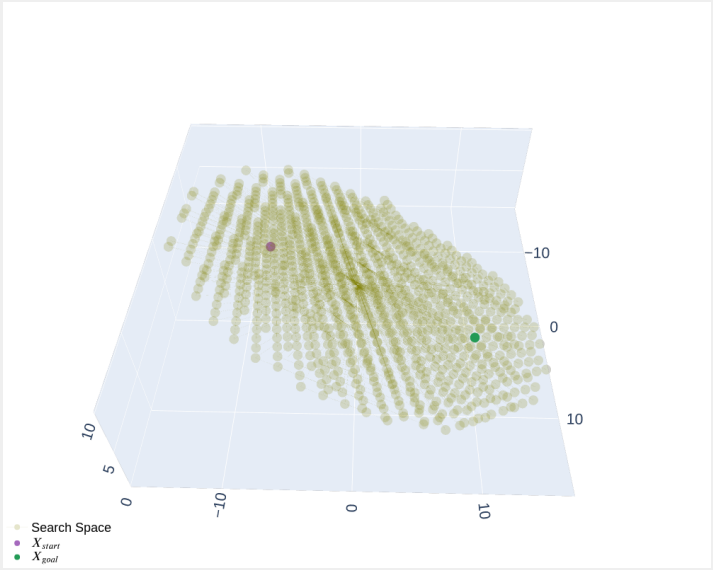
Contents

- 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)**

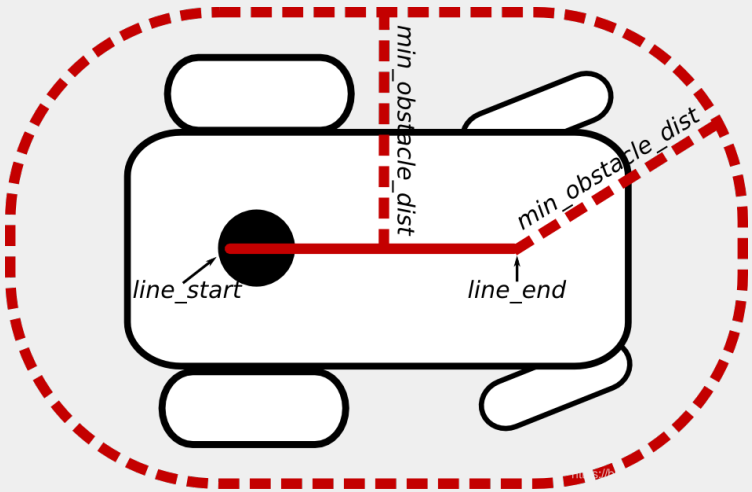


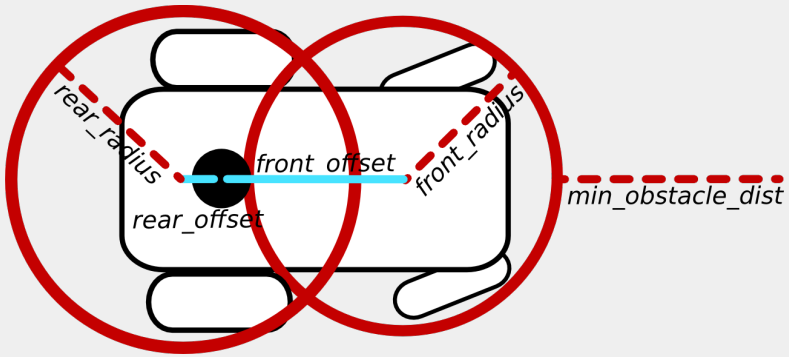


Robot configuration (robot footprint): specification of robot representation. **Robot configuration space**: possible all robot configuration with respect to robot degree of freedom (DOF)

Configuration Space vs Search Space for Robot

Robot configuration (robot footprint): specification of robot representation

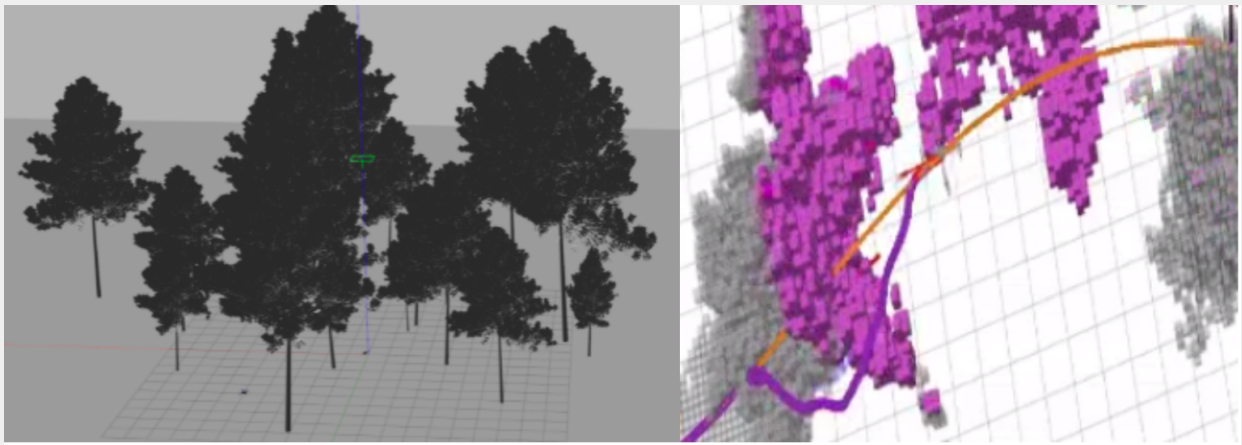




https://www.ncnynl.com/archives/201809/2602.html

Configuration Space for Obstacles

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



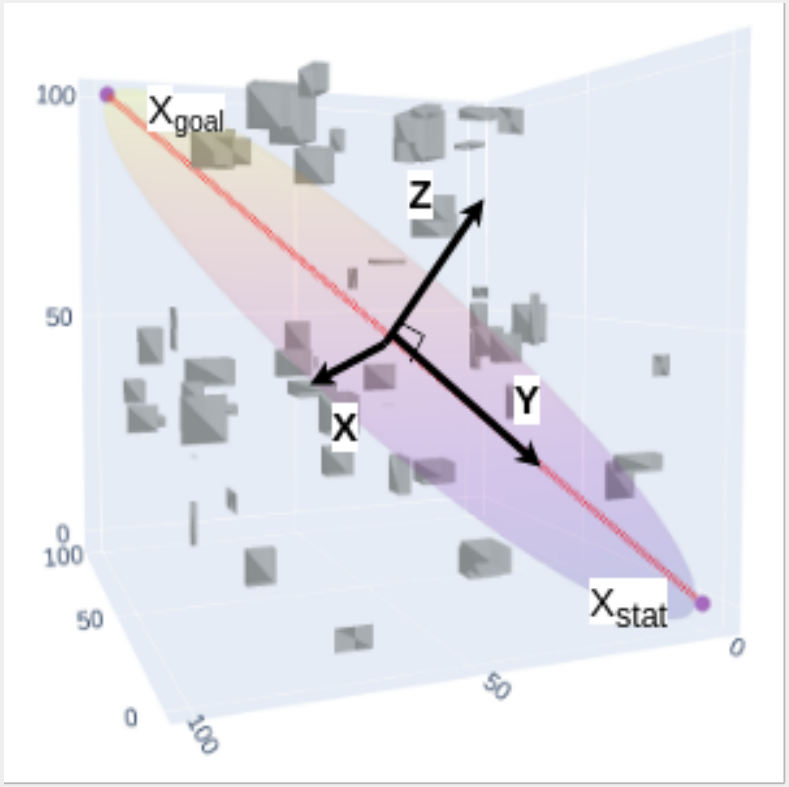
(a) (b)

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

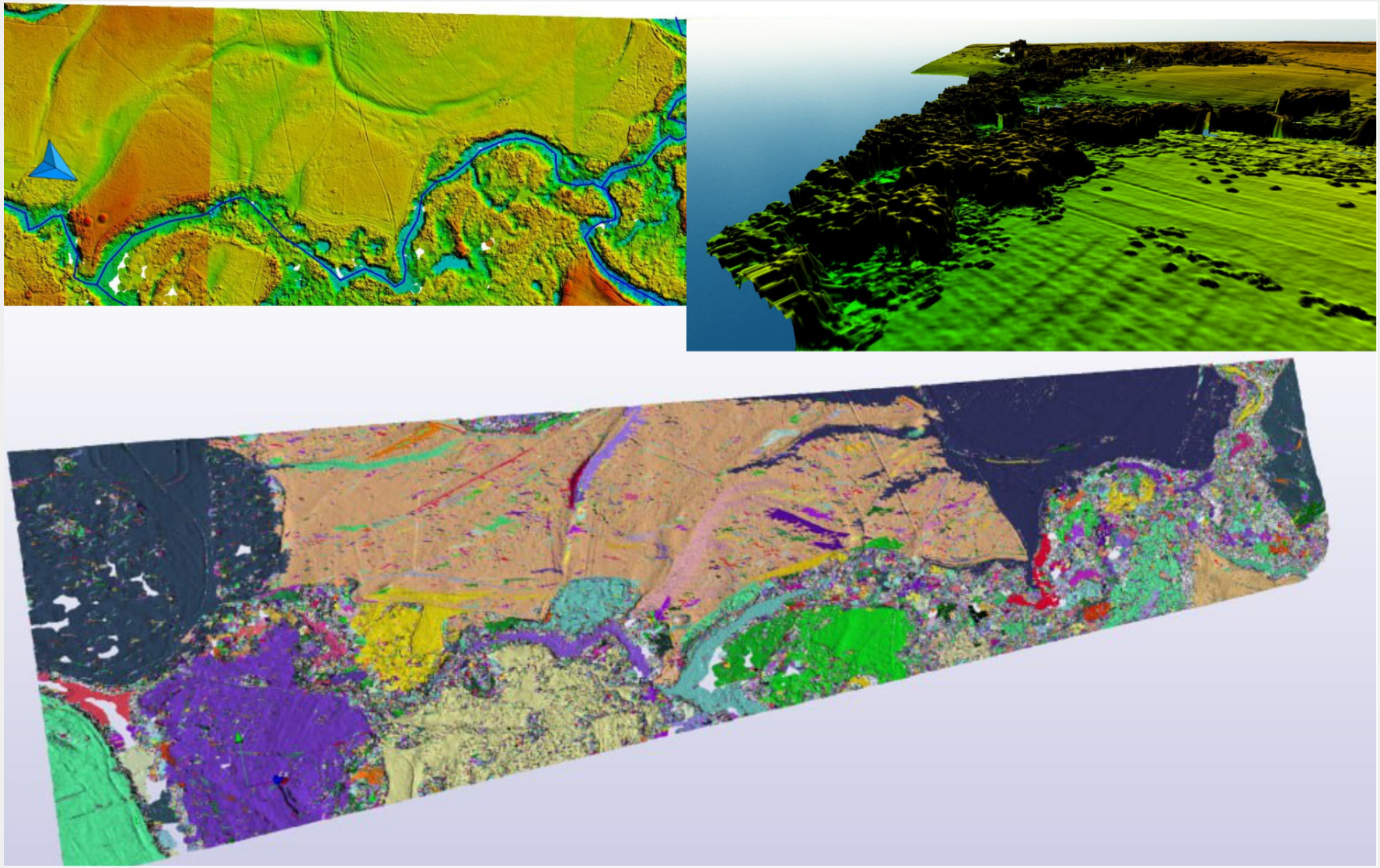
- $X = (0, 1)^d$ as the configuration space where $X \in \mathbb{R}^d$
- X_{obs} is the regions occupied by the obstacles
- $X \setminus X_{obs}$ becomes an open set where $cl(X \setminus X_{obs})$ can be defined as the traversable space (X_{free}) where $cl(.)$ is denotes the closure of set $X \setminus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



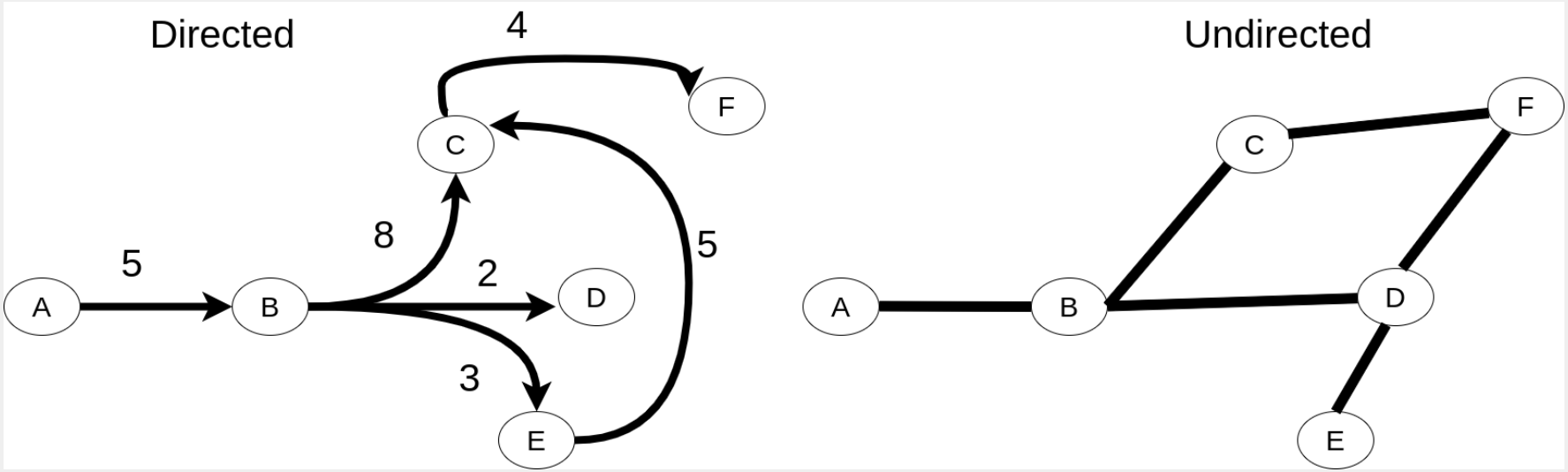
What kind of things do you have to consider for defining $cl(X \setminus X_{obs})$ in this workspace?

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

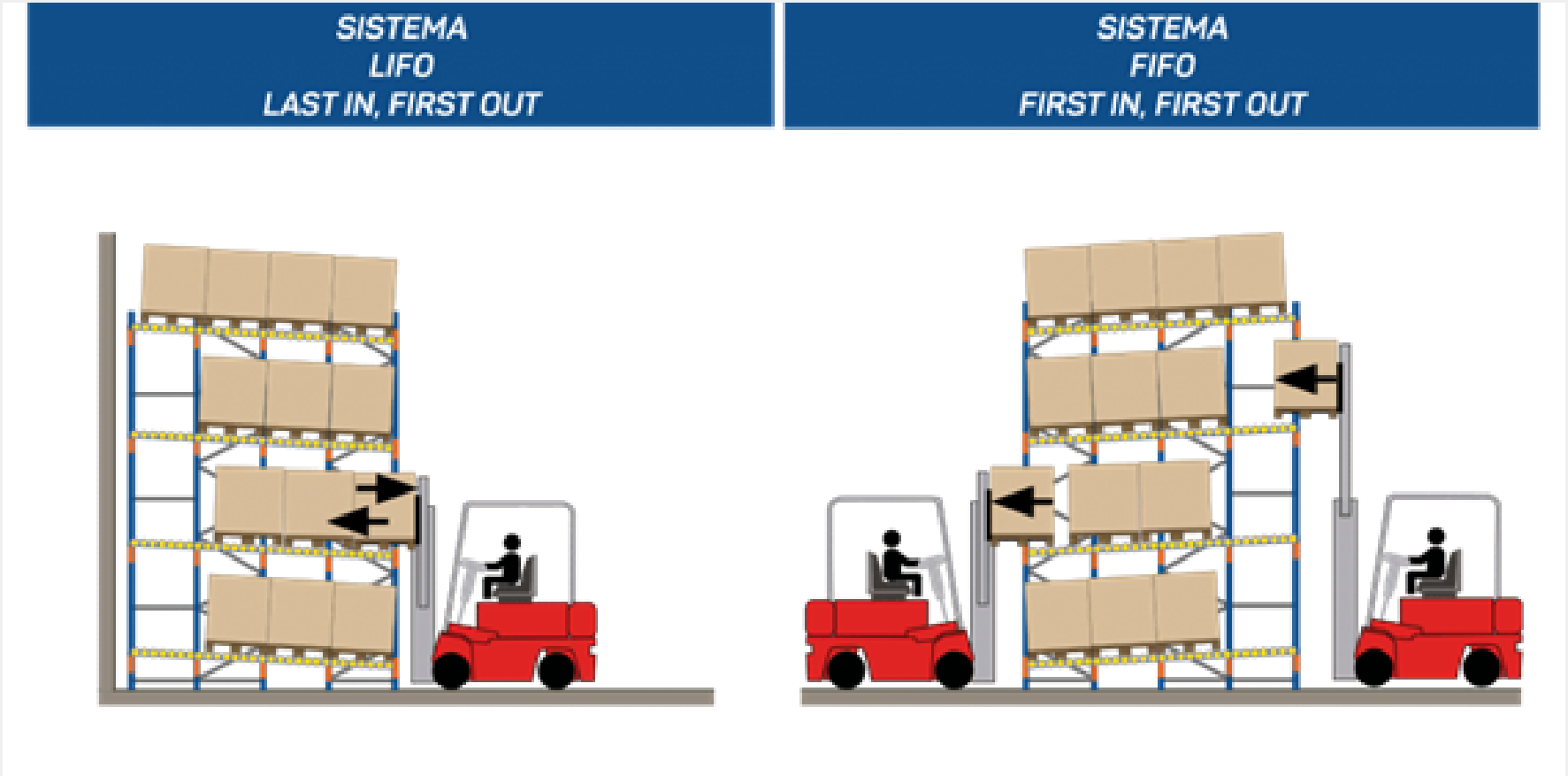
Graph Searching

- 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

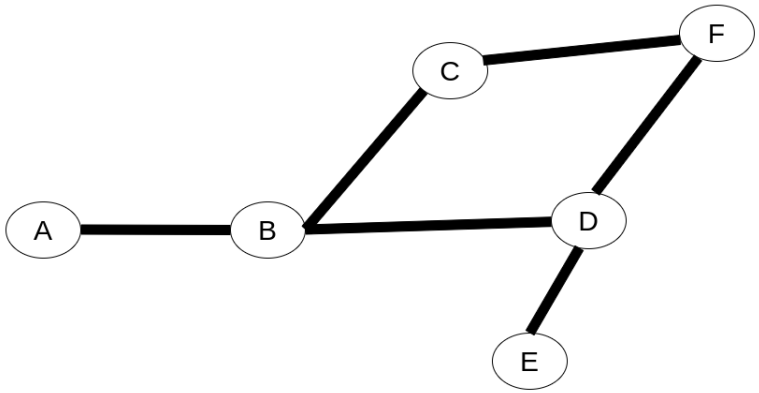
Graph Searching



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

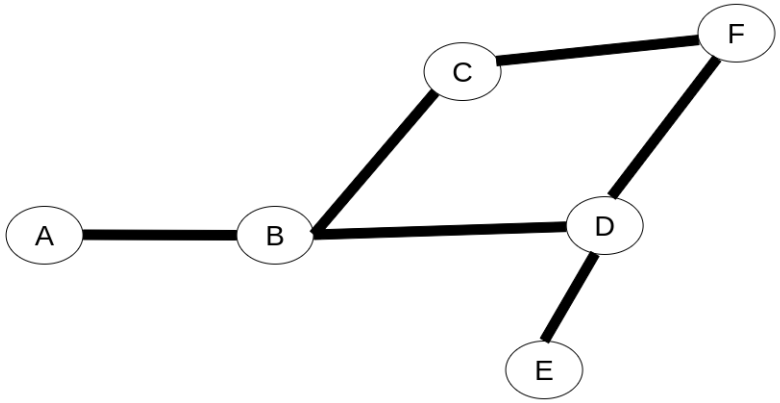
```
1: procedure DFS(G, n)                    ▷ The graph G, starting node n
2:     n.visited ← true
3:     for eache ∈ G.neighbours(n) do
4:         if e.visited == false then
5:             dfs(G, v)
6:         end if
7:     end for
8: end procedure
```

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

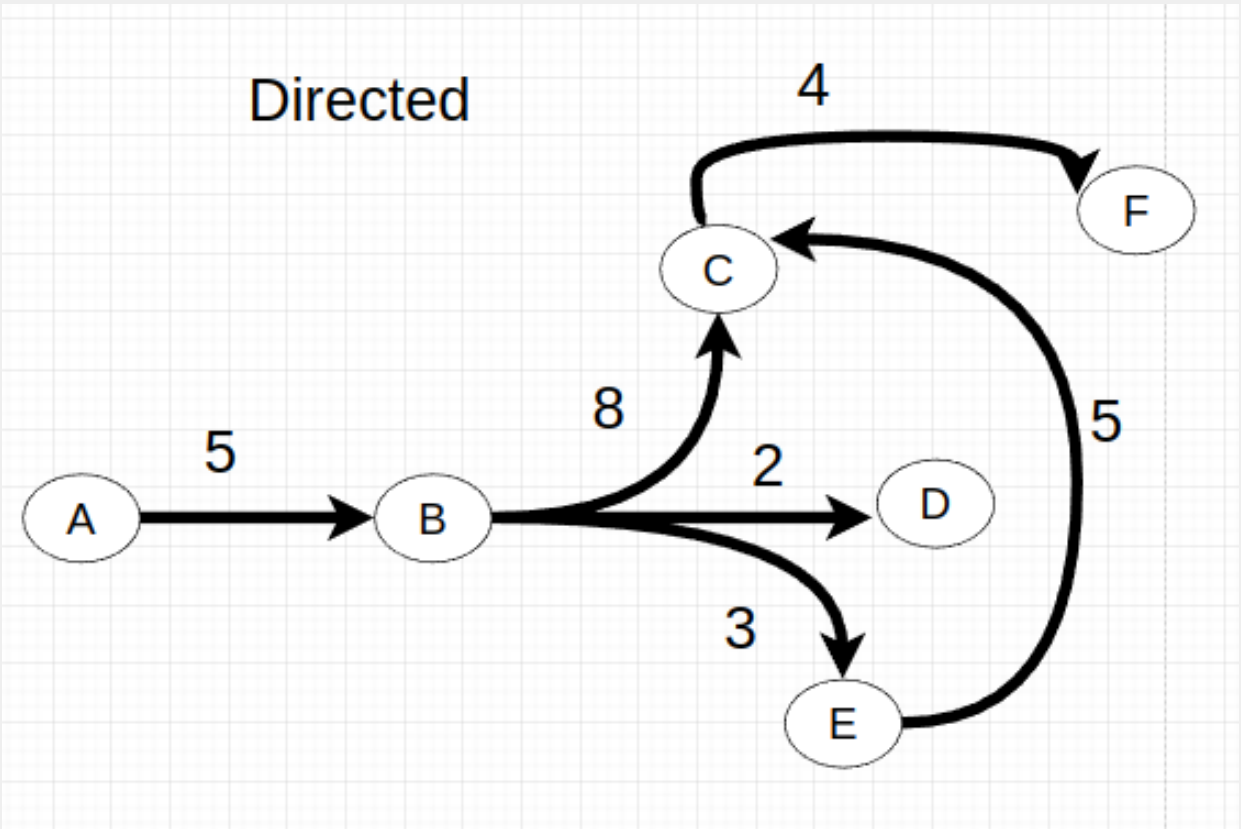
Breath first search can also be used for **flow field path-finding, distance map (or cost map)** generation, and much more.

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
```

Graph Searching: Costs Consideration

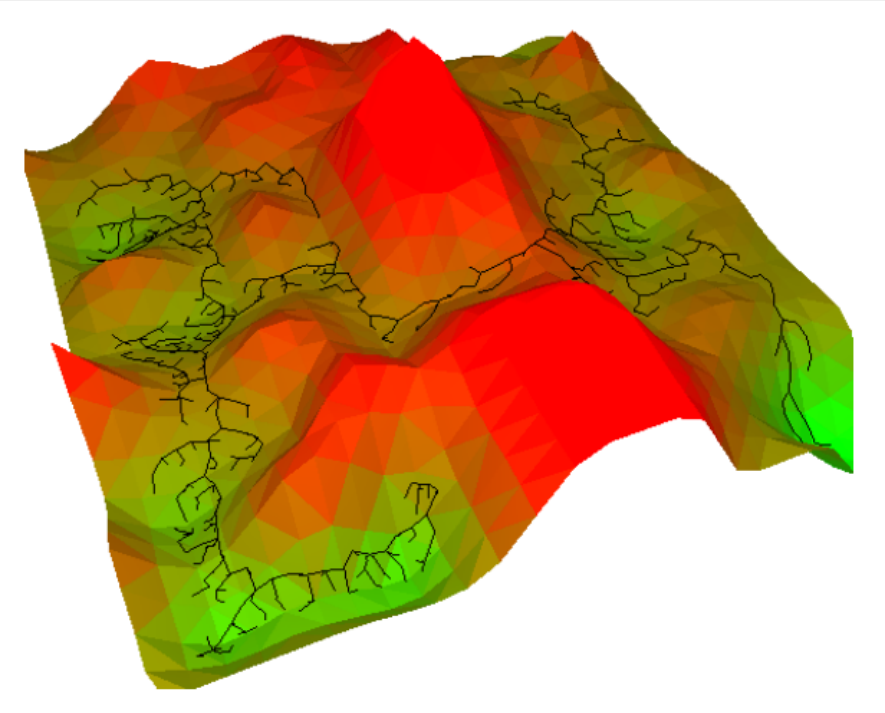
In real-world scenarios, different types of costs, e.g., length, time, and energy, are associated with nodes. If all **costs are equal**, the **breath first search** gives **the optimal solution**.



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

Graph Searching: Costs Consideration

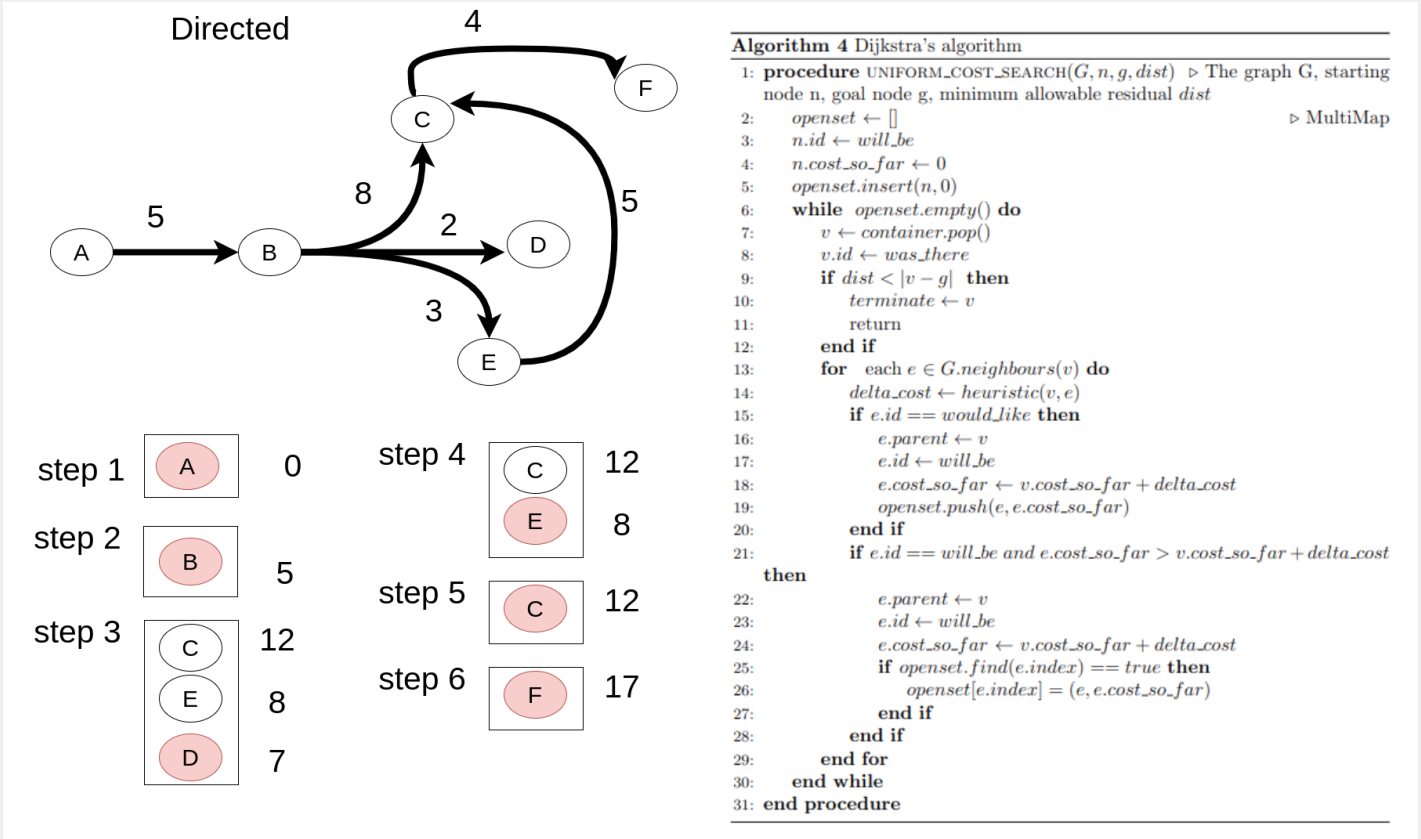
The strategy is to expand the node with **cheapest accumulated cost g(v)**, where g(v) is the best-accumulated cost from the start node to node v. **Node** that has been **expanded** ensures having the **minimum cost from the start node**



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
```

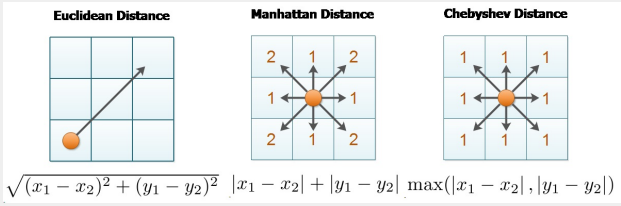
The algorithm incorporates **only the accumulated cost till the considered node**, and **every direction** has to incorporate to make the **next move**. **No information** about the **goal location**

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

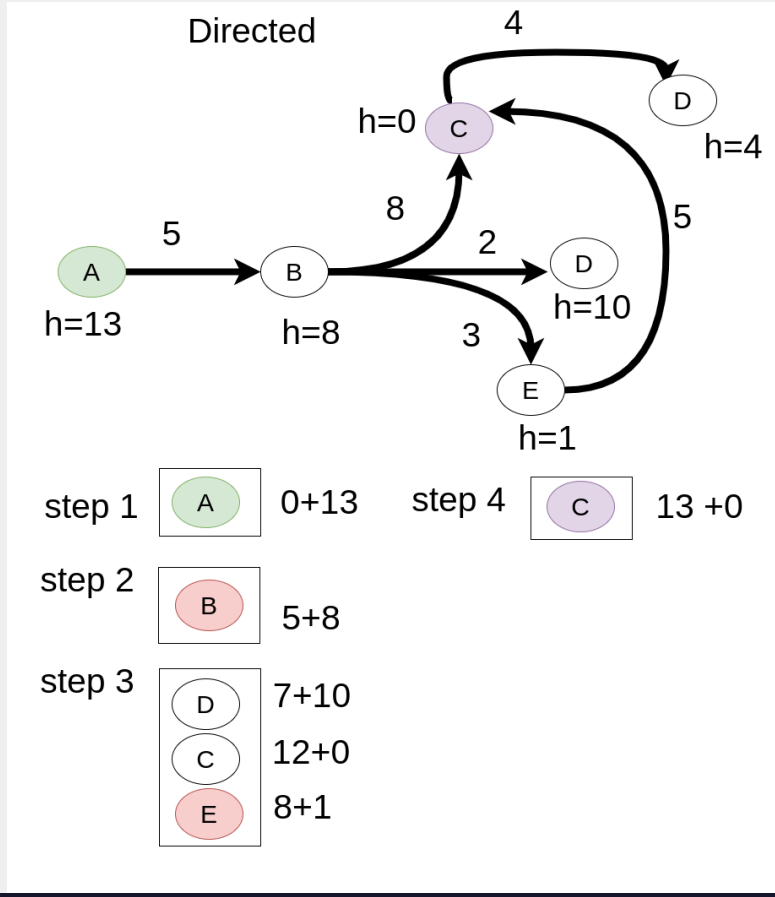
```
1:  procedure GBFS(G, n, g, dist)   ▷ The graph G, starting node n, goal node g, minimum allowable residual dist
2:      openset ← []                                   ▷ Priority queue
3:      n.id ← will_be
4:      n.cost_to_go ← 0
5:      openset.insert(n, 0)
6:      while openset.empty() do
7:          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**

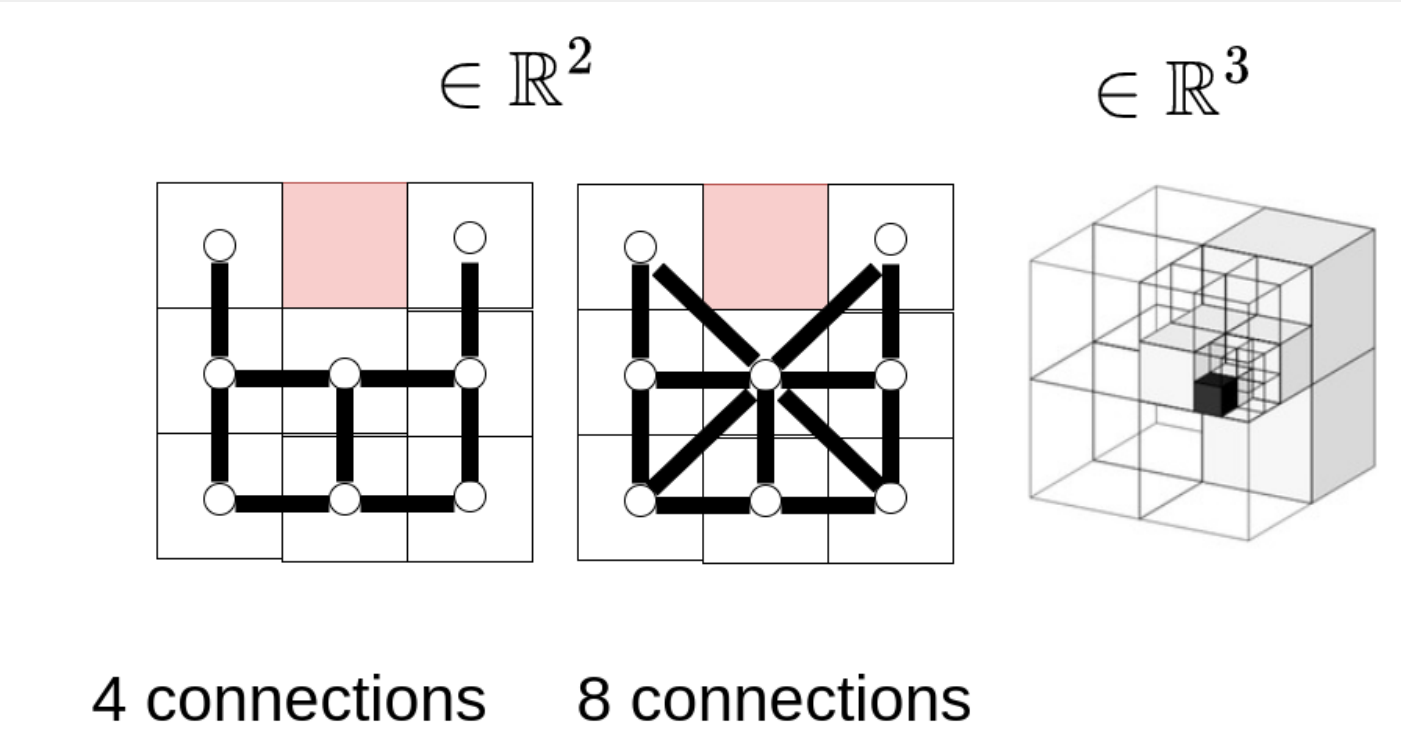


Algorithm 6 AStar algorithm

```
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

Consider KinoDynamic A*:Heuristics, where the objective function is expressed by:

$$J = \int_0^T g(x,u)dt = \int_0^T (1 + u^\top Ru)dt = \int_0^T (1 + a_x^2 + a_y^2 + a_z^2)dt \quad (2)$$

where system state $x = (p_x, p_y, p_z, v_x, v_y, v_z)$, system input $u = a_x, a_y, a_z$, and motion model of the system $\dot{x} = f(x,u) = (v_x, v_y, v_z, a_x, a_y, a_z)$. Pontryagin's minimum principle can be used to solve this problem:
$\dot{x}^*(t) = f(x^*(t), u^*(t)),\ x^*(0) = x(0)$

- Define the Hamiltonian

$$\begin{aligned} H(x,u,\lambda) &:= g(x,u) + \lambda^\top f(x,u), \quad \lambda = (\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5, \lambda_6) \\ &= (1 + a_x^2 + a_y^2 + a_z^2) + \lambda^\top f(s,u) \\ &= (1 + a_x^2 + a_y^2 + a_z^2) + \lambda_1 v_x + \lambda_2 v_y + \lambda_3 v_z + \lambda_4 a_x \\ &\quad + \lambda_5 a_y + \lambda_6 a_z \end{aligned} \quad (3)$$

- Since x is a function of p and v,

$$\dot{\lambda}^*(t) = -\frac{H(\cdot)}{\partial x} = -\left(\frac{\partial f(\cdot)}{\partial x}\right)^\top \lambda^*(t) - \frac{\partial g(\cdot)}{\partial x} \tag{4}$$
$$\dot{\lambda} = (0, 0, 0, -\lambda_1, -\lambda_2, -\lambda_3)^\top$$

The costate equation is determined easily, for the latter convenience the solution is written in constants α, β

$$\lambda(t) = \begin{bmatrix} 2\alpha_1 \\ 2\alpha_2 \\ 2\alpha_3 \\ -2\alpha_1 t - 2\beta_1 \\ -2\alpha_2 t - 2\beta_2 \\ -2\alpha_3 t - 2\beta_3 \end{bmatrix}$$

- The optimal input is solved as:

$$
\begin{aligned}
0 = \frac{H(\cdot)}{\partial u} &= \left(\frac{\partial g(\cdot)}{\partial u}\right)^{\top} \lambda^{*}(t) + \frac{\partial f(\cdot)}{\partial u} \\
u^{*}(t) &= \begin{bmatrix} \alpha_1 t + \beta_1 \\ \alpha_2 t + \beta_2 \\ \alpha_2 t + \beta_3 \end{bmatrix}
\end{aligned} \tag{5}
$$

- From which, i.e., $u^*(t)$, the optimal state trajectory is solved by integration, i.e., an integral on u to get v and get p in the integral v:

$$x^*(t) = \begin{bmatrix} \frac{\alpha_1}{6}t^3 + \frac{\beta_1}{2}t^2 + v_{xo}t + p_{xo} \\ \frac{\alpha_2}{6}t^3 + \frac{\beta_2}{2}t^2 + v_{yo}t + p_{yo} \\ \frac{\alpha_3}{6}t^3 + \frac{\beta_3}{2}t^2 + v_{zo}t + p_{zo} \\ \frac{\alpha_1}{2}t^2 + \frac{\beta_1}{2}t + v_{xo} \\ \frac{\alpha_2}{2}t^2 + \frac{\beta_2}{2}t + v_{yo} \\ \frac{\alpha_3}{2}t^2 + \frac{\beta_3}{2}t + v_{zo} \end{bmatrix}, \tag{6}$$

where initial state $x(0) = (p_{xo}, p_{yo}, p_{zo}, v_{xo}, v_{yo}, v_{zo})$.

The remaining unknowns α, β are solved for as a function of the desired end transnational variable components as defined in eq. **??**

$$\begin{bmatrix} \frac{T^3}{6} & 0 & 0 & \frac{T^2}{2} & 0 & 0 \\ 0 & \frac{T^3}{6} & 0 & 0 & \frac{T^2}{2} & 0 \\ 0 & 0 & \frac{T^3}{6} & 0 & 0 & \frac{T^2}{2} \\ \frac{T^2}{2} & 0 & 0 & T & 0 & 0 \\ 0 & \frac{T^2}{2} & 0 & 0 & T & 0 \\ 0 & 0 & \frac{T^2}{2} & 0 & 0 & T \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = \underbrace{\begin{bmatrix} \Delta p_x \\ \Delta p_y \\ \Delta p_z \\ \Delta v_x \\ \Delta v_y \\ \Delta v_z \end{bmatrix}}_{x(T)-x(0)} = \begin{bmatrix} p_{xf} - v_{x0}T - p_{x0} \\ p_{yf} - v_{y0}T - p_{y0} \\ p_{zf} - v_{z0}T - p_{z0} \\ v_{xf} - v_{x0} \\ v_{yf} - v_{y0} \\ v_{zf} - v_{z0} \end{bmatrix}, \tag{7}$$

where final state $x(T) = (p_{xf}, p_{yf}, p_{zf}, v_{xf}, v_{yf}, v_{zf})$.

Hence, solving for the unknown coefficients yields

$$\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = \begin{bmatrix} -\frac{12}{T^3} & 0 & 0 & \frac{6}{T^2} & 0 & 0 \\ 0 & -\frac{12}{T^3} & 0 & 0 & \frac{6}{T^2} & 0 \\ 0 & 0 & -\frac{12}{T^3} & 0 & 0 & \frac{6}{T^2} \\ \frac{6}{T^2} & 0 & 0 & -\frac{2}{T} & 0 & 0 \\ 0 & \frac{6}{T^2} & 0 & 0 & -\frac{2}{T} & 0 \\ 0 & 0 & \frac{6}{T^2} & 0 & 0 & -\frac{2}{T} \end{bmatrix} \begin{bmatrix} \Delta p_x \\ \Delta p_y \\ \Delta p_z \\ \Delta v_x \\ \Delta v_y \\ \Delta v_z \end{bmatrix} \tag{8}$$

After finding these, $x^*(t)$ and cost function J can be determined.

$$J = \frac{1}{T}\int_o^T (1 + a_x^2 + a_y^2 + a_z^2)dt$$
$$= T + \left(\frac{\alpha_1^2 T^3}{3} + \alpha_1\beta_1 T^2 + \beta_1 T\right) + \left(\frac{\alpha_2^2 T^3}{3} + \alpha_2\beta_2 T^2 + \beta_2 T\right) + \left(\frac{\alpha_3^2 T^3}{3} + \alpha_3\beta_1 T^2 + \beta_3 T\right) \quad (9)$$

The cost function is only a function of time, hence this kind of problem is called a minimum time problem.

Polynomial root finding Assume that

$$T + \left(\frac{\alpha_1^2 T^3}{3} + \alpha_1\beta_1 T^2 + \beta_1 T\right) + \left(\frac{\alpha_2^2 T^3}{3} + \alpha_2\beta_2 T^2 + \beta_2 T\right) + \left(\frac{\alpha_3^2 T^3}{3} + \alpha_3\beta_1 T^2 + \beta_3 T\right) := c_0 + c_1 x + c_2 x^2 + c_3 x^3$$

To find roots of a such polynomial, **matrix eigenvalue method** can be employed. The matrix eigenvalues are defined as

$$A\mathbf{y} = \lambda\mathbf{y}$$

If the roots of the polynomial are the eigenvalues of the matrix

$$A\mathbf{y} = x\mathbf{y}$$

That is to say if $\mathbf{y} = [x^2\ x\ 1]^\top$, matrix A is constructed as

$$\begin{bmatrix} -\frac{c_2}{c_3} & -\frac{c_1}{c_3} & -\frac{c_0}{c_3} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x^2 \\ x \\ 1 \end{bmatrix} = x \begin{bmatrix} x^2 \\ x \\ 1 \end{bmatrix}$$

Hybrid A*: Motion model

- What steering angles are possible? $[-\pi/2, \pi/2]$. It was assumed that the car moves in the direction that the rear wheels are pointing. When $\phi = \pi/2$, the front wheel perpendicular to the rear wheels, then car has to rotate in place. In other words, $\dot{x} = \dot{y} = 0$ because because the center of the rear axle does not translate. This behaviour is usually not possible because the front wheels would collide with the front axle when turned to $\phi = \pi/2$. Thus,

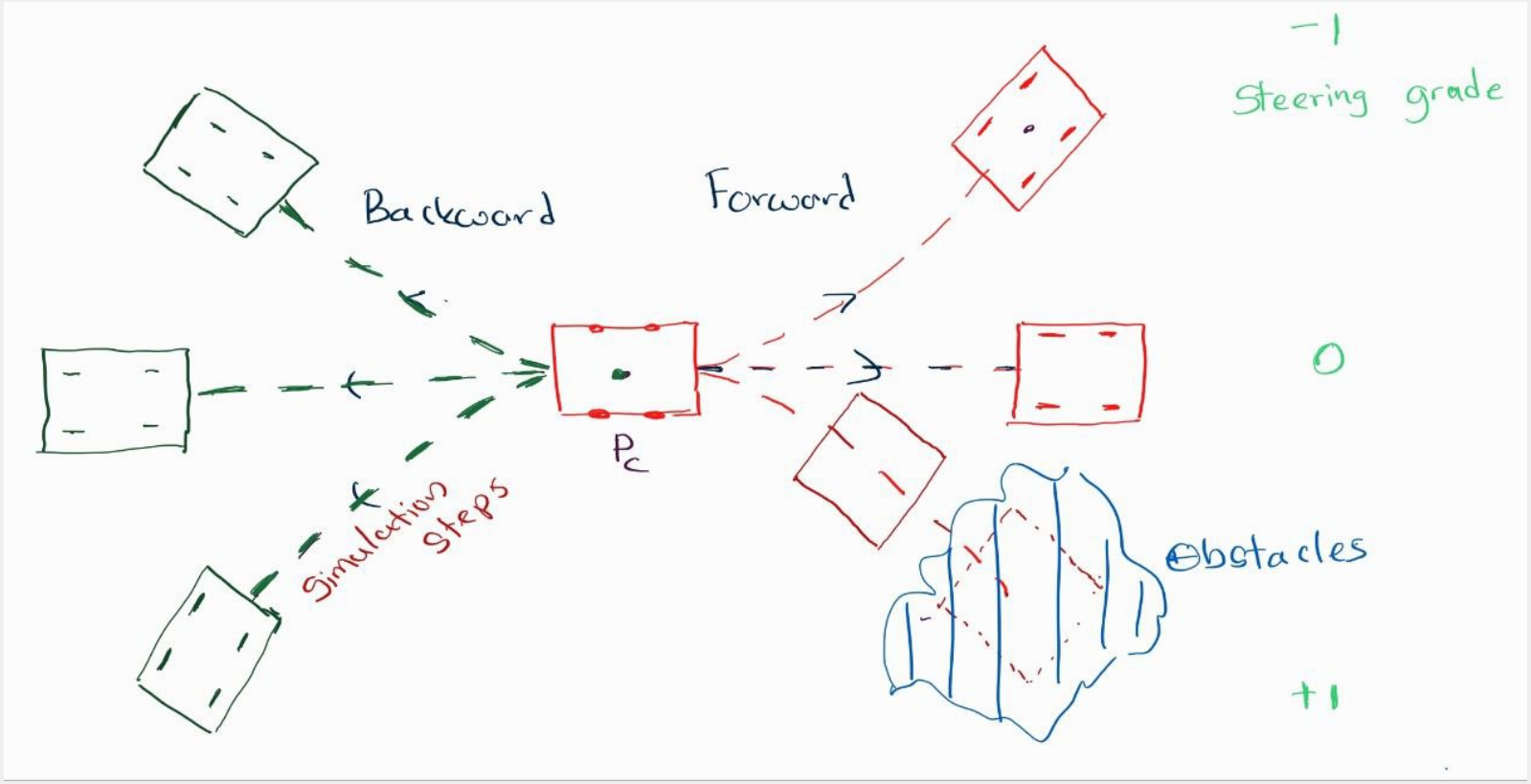
$$|\phi| \leq \phi_{max}$$

- A simple car is moving slowly to safely neglect dynamics, let's assume $|u_s| \leq 1$, where $u_s \in \{-1, 0, 1\}$. However, $0 < u_s < 1$, in this case, car can not drive in reverse

Hybrid A*: Motion model

- **Tricycle:** $U = [-1, -1] \times [-\pi/2, \pi/2]$, Assuming front-wheel drive, the vehicle can rotate in place if $u_\theta = \pi/2$. This kind of motion can be obtained using unicycle model
- **Simple car:** $U = [-1, -1] \times [-\phi_{max}, \phi_{max}]$, by requiring that $|u_\phi| < \phi_{max} < \pi/2$, a car with minimum turning radius $\rho_{min} = \frac{L}{tan\phi_{max}}$ is obtained
- **Reeds-Sheepp Car:** Further restrict the speed of the car, i.e., $u_s = \{-1, 0, 1\}$ or a car with three gears: reverse, park, and forward.
- **Dubins Car:** After removing reverse speed $u_s = -1$ from a Reeds-Sheepp car, $u_s = \{0, 1\}$ as the only possible speeds

Hybrid A*: finding neighbors



Hybrid A*: cost to go h

- When **close to the goal** location: use **appropriate kinematically feasible path**, e.g., Dubins or Reeds-Sheepp.
- When **far from the goal** location: **euclidean** distance or **manhattan** distance

Hybrid A*: cost so far g

Algorithm 13 Calculate cost so far g

```
1:  procedure CALCULATEG(p_c, p_n)                ▷ Current pos and neighbour pos
2:      λ_s, λ_sc, λ_r      ▷ steering penalty, steering change penalty, and reversing penalty
3:      seg_length                                  ▷ number of simulation steps
4:      if p_n.direction == FORWARD then
5:          if p_n.steering_grade != p_c.steering_grade then
6:              if p_n.steering_grade == 0 then
7:                  g ← seg_length · λ_sc
8:              end if
9:              if p_n.steering_grade != 0 then
10:                 g ← seg_length · λ_sc · λ_s
11:             end if
12:         end if
13:         if p_n.steering_grade == p_c.steering_grade then
14:             if p_n.steering_grade == 0 then
15:                 g ← seg_length
16:             end if
17:             if p_n.steering_grade != 0 then
18:                 g ← seg_length · λ_s
19:             end if
20:         end if
21:     end if
22:     if p_n.direction != FORWARD then
23:         if p_n.steering_grade != p_c.steering_grade then
24:             if p_n.steering_grade == 0 then
25:                 g ← seg_length · λ_r
26:             end if
27:             if p_n.steering_grade != 0 then
28:                 g ← seg_length · λ_sc · λ_s · λ_r
29:             end if
30:         end if
31:         if p_n.steering_grade == p_c.steering_grade then
32:             if p_n.steering_grade == 0 then
33:                 g ← seg_length · λ_r
34:             end if
35:             if p_n.steering_grade != 0 then
36:                 g ← seg_length · λ_s · λ_r
37:             end if
38:         end if
39:     end if
```

Hybrid A*

Algorithm 14 Hybrid A*

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