

POLITECNICO DI MILANO

AUTONOMOUS VEHICLES

S. Arrigoni



POLITECNICO
MILANO 1863



Motion Planning

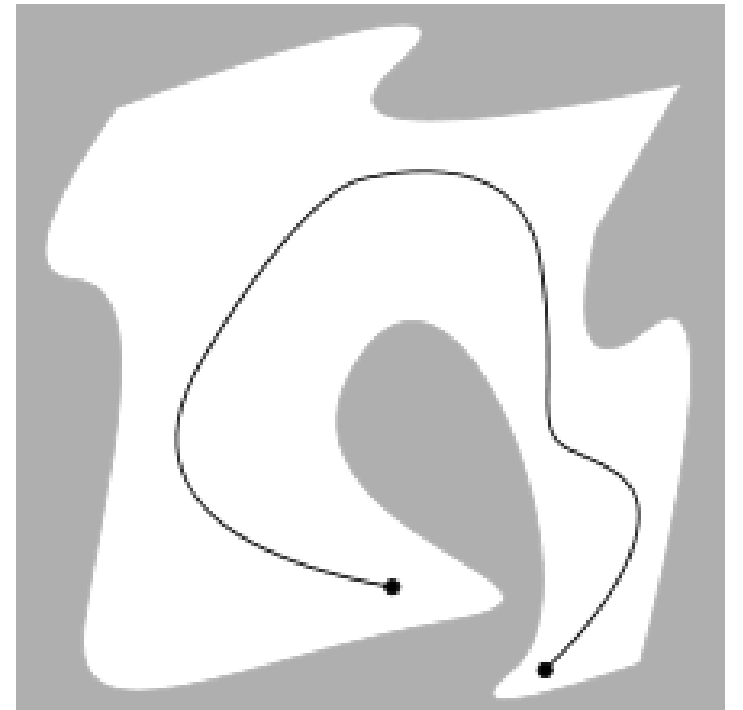
in-depth study

Introduction

What is motion planning?

Define a sequence of actions to take a robot from one known state (**initial**) to another known state (**goal**), while:

- avoiding obstacles;
- respecting motion constraints;
- optimizing a cost function (*hopefully*)



Introduction

Assumptions

- Ideal knowledge of the robot and the world
- A priori knowledge of actual and goal positions
- motion rules of our robot are fixed

Introduction

Popular approaches:

- **Potential fields** [*Rimon, Koditschek, '92*]: create forces on the robot that pull it toward the goal and push it away from obstacles
- **Grid-based planning** [*Stentz, '94*]: discretizes problem into grid and runs a graph-search algorithm (Dijkstra, A*, ...)
- **Combinatorial planning** [*LaValle, '06*]: constructs structures in the configuration (C-) space that completely capture all information needed for planning
- **Sampling-based planning** [*Kavraki et al, '96; LaValle, Kuffner, '06, etc.*]: uses collision detection algorithms to probe and incrementally search the C-space for a solution, rather than completely characterizing all of the C_{free} structure

Introduction

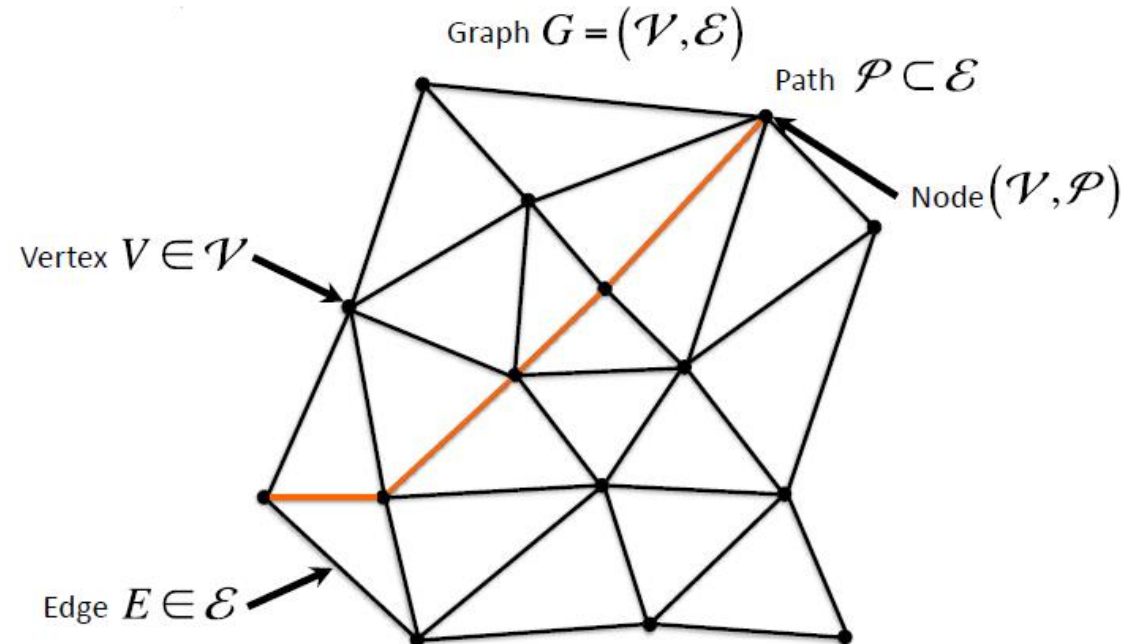
Discrete graph representation

It is common to convert the planning problem into a (discrete) **graph** representation

➡ use one of the existing search algorithms on the graph

Where **edges** can:

- Have a direction (*directed graph*)
- Have a cost (*weighted graph*)



Introduction

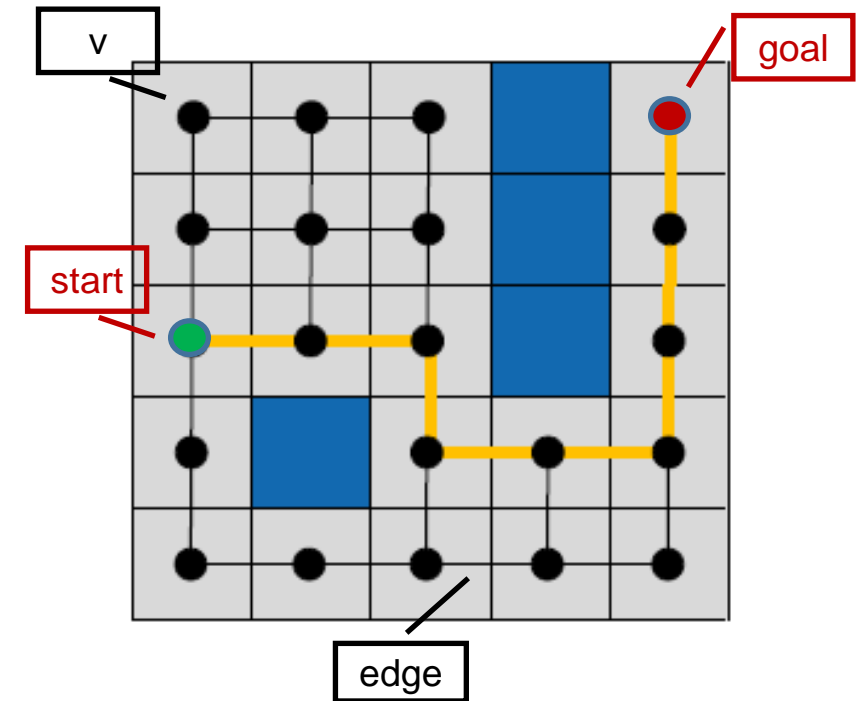
Grid based approach

Discretize the continuous world **into a grid**:

- Each cell is either free or forbidden
- Robot moves between adjacent free cells
- Goal: find sequence of free cells from start

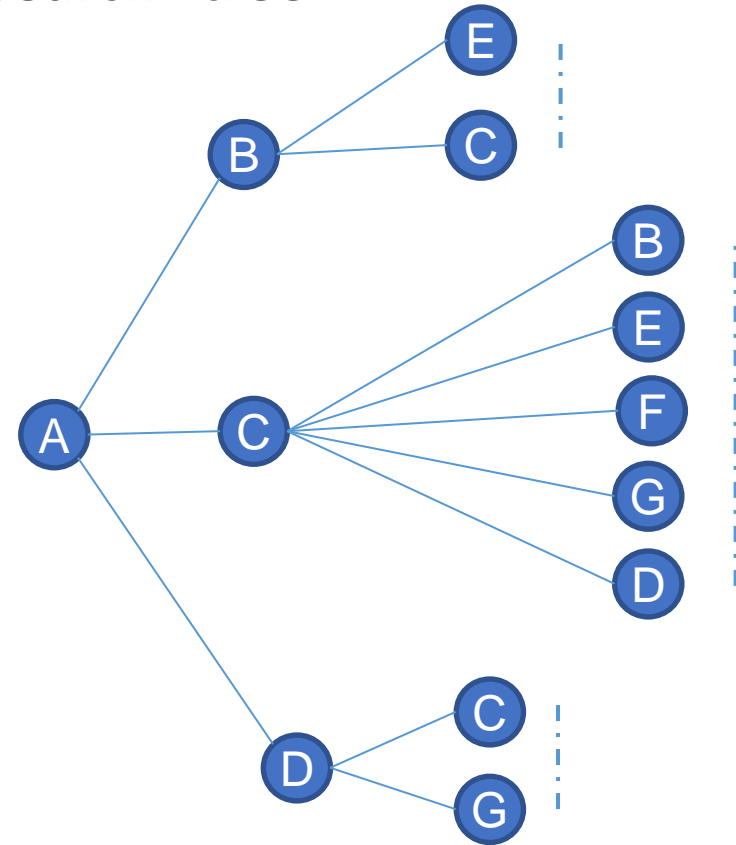
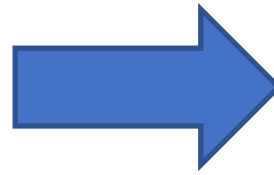
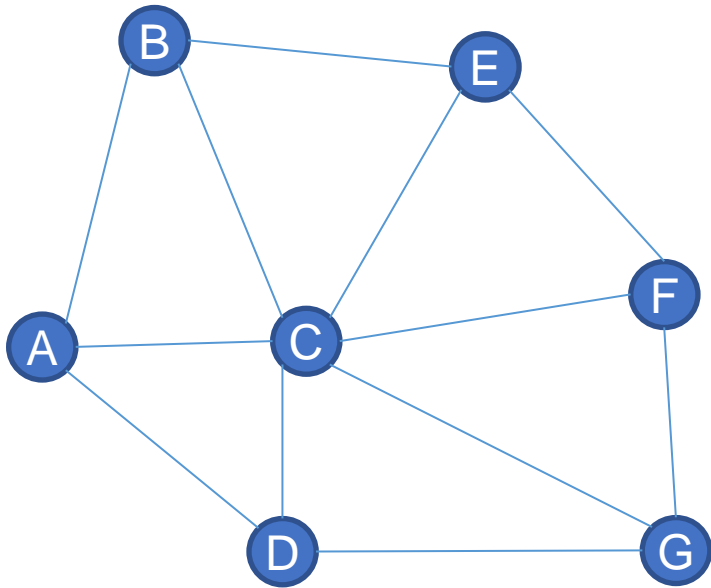
Mathematically, this corresponds to pathfinding in a discrete graph $G = (V, E)$

- Each vertex $v \in V$ represents a free cell
- Edges $(v, u) \in E$ connect adjacent grid cells



Introduction

Define **Graph** and a possible **research “tree”**

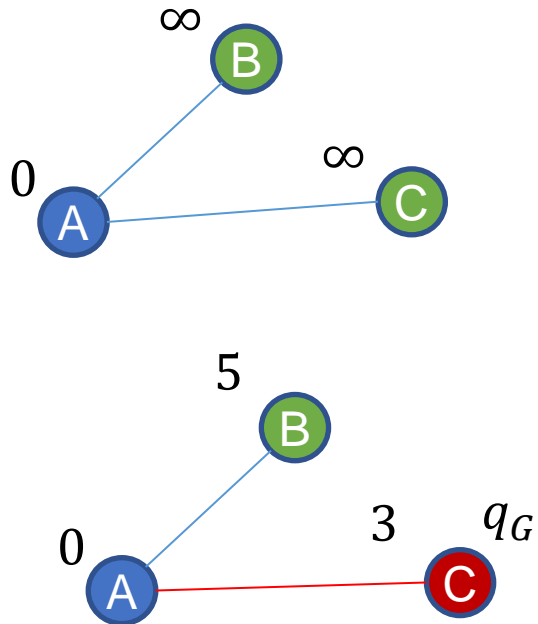


Introduction

Graph search algorithms

Label correcting algorithms:

best path optimizing *cost-of-arrival* $C(q)$
 q_I to q

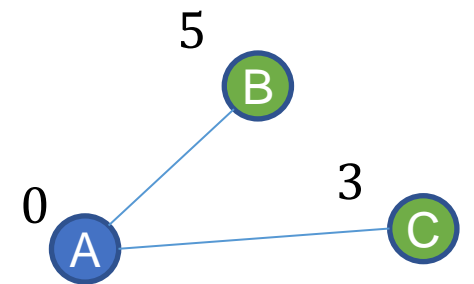


Define neighbours of q :

Evaluate and update $C(q)$:

Define new q

Repeat until $q = q_G$



Introduction

Label correcting algorithms:

best path optimizing *cost-of-arrival* $C(q)$
 q_I to q

Initialize $C(\text{nodes}) = \infty$:

Remove q from queue and **find** q' :

Evaluate and **update** $C(q')$:

if lower set q as q' parent

if $q' \neq q_G$

Add q' in queue

Get next (q)



terminate



Introduction

How to get next node?

Breadth-First-Search (BFS, Bellman-Ford): Maintain Q as a **list** – First in/first out

Q: frontier/open/alive set of nodes
= next nodes to be visited

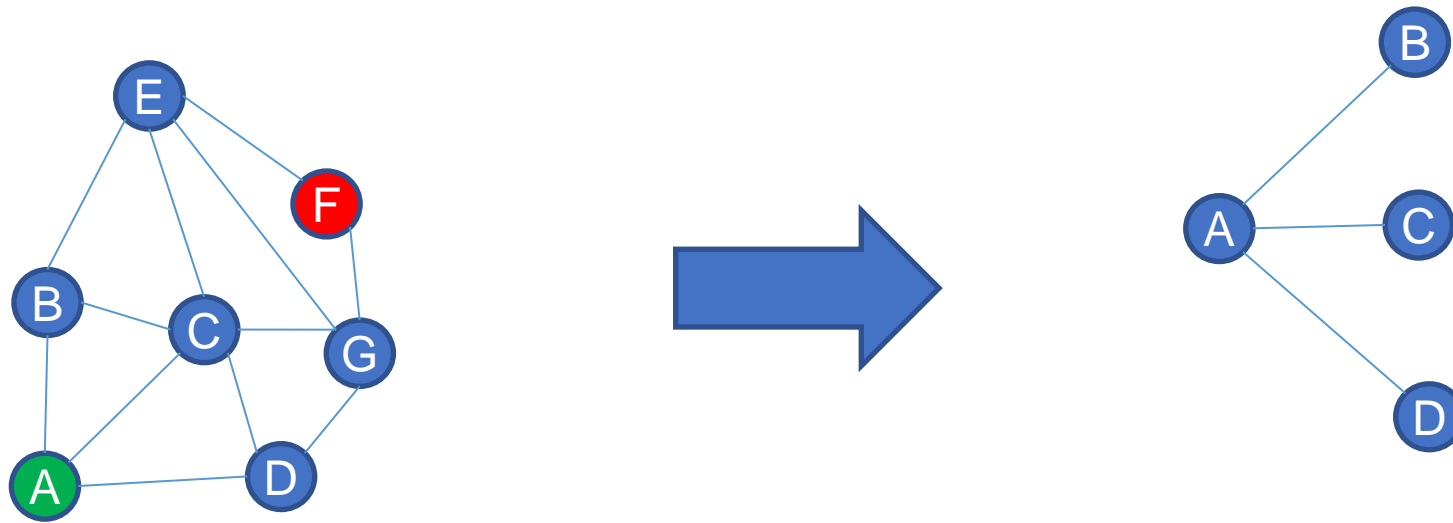
Update cost for **all edges up to current depth** before proceeding to greater depth

- Complete (will find the solution if it exists)
- Guaranteed to find the shortest (number of edges) path
- First solution found is the optimal path
- Can deal with negative edge (transition) costs

Introduction

How to get next node?

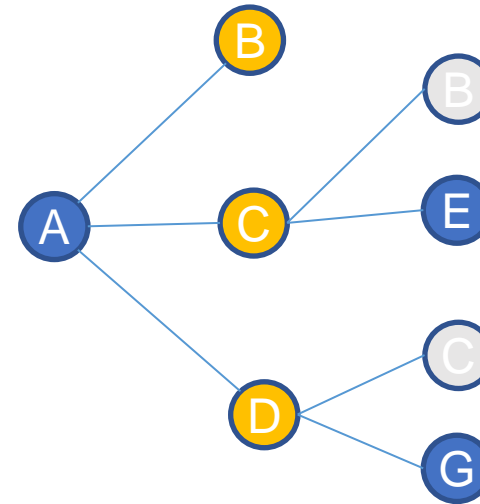
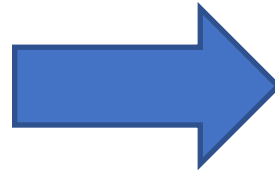
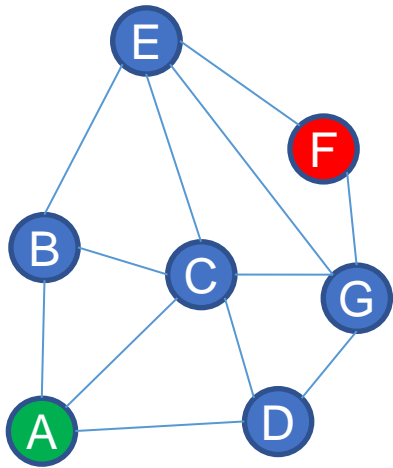
Breadth-First-Search (BFS, Bellman-Ford): Maintain Q as a **list** – First in/first out



Introduction

How to get next node?

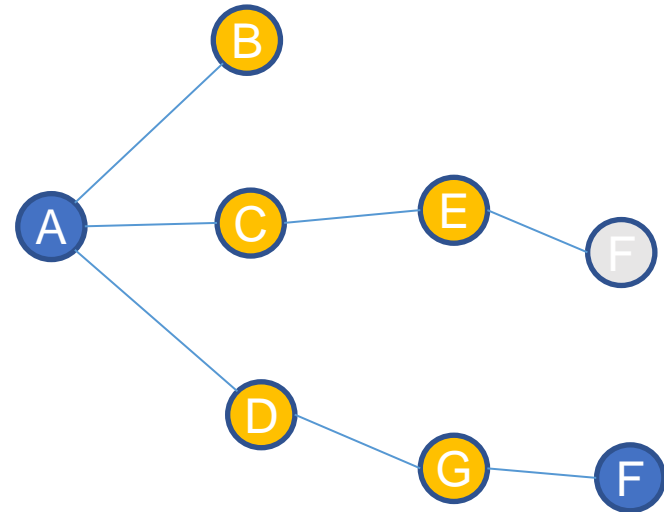
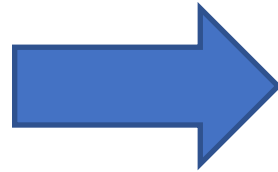
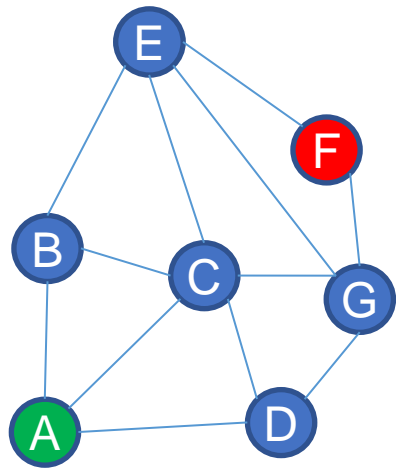
Breadth-First-Search (BFS, Bellman-Ford): Maintain Q as a **list** – First in/first out



Introduction

How to get next node?

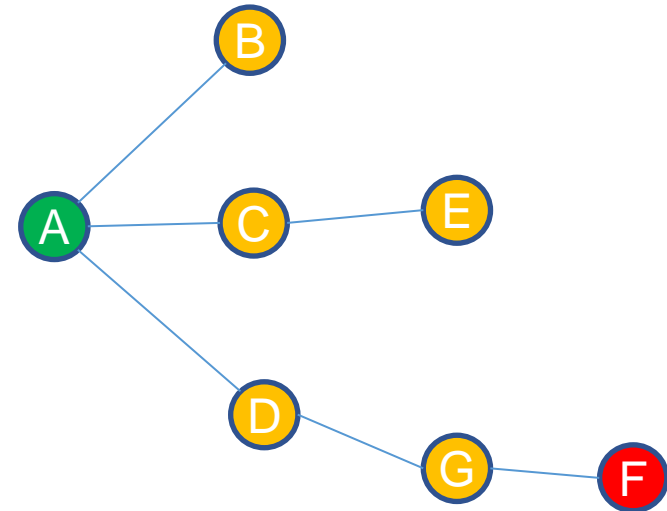
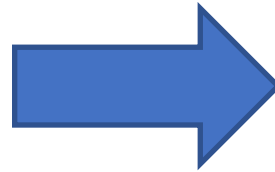
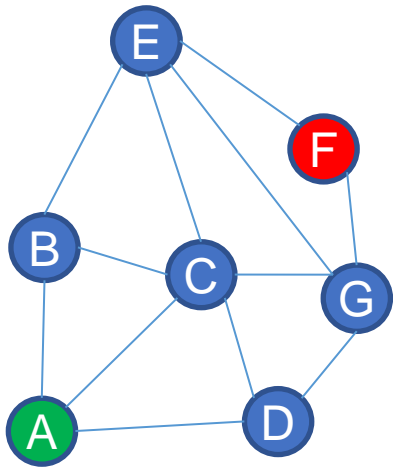
Breadth-First-Search (BFS, Bellman-Ford): Maintain Q as a **list** – First in/first out



Introduction

How to get next node?

Breadth-First-Search (BFS, Bellman-Ford): Maintain Q as a **list** – First in/first out



Introduction

How to get next node?

Depth-First-Search (DFS): Maintain Q as a **stack** – Last in/first out

- Lower memory requirement (only need to store part of graph)

Q: frontier/open/alive set of nodes
= next nodes to be visited

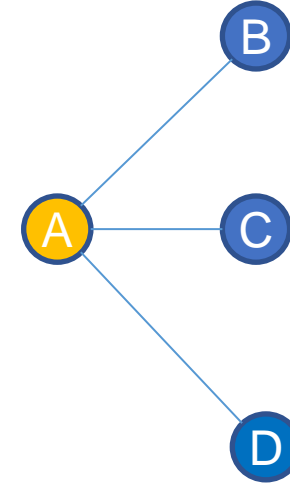
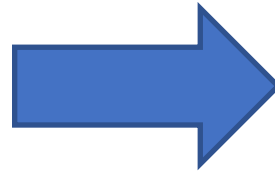
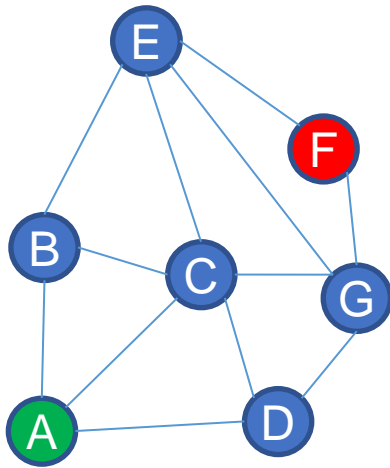
Starts at the root node and explores as far as possible
along each branch before backtracking

- Lower memory footprint than BFS with high-branching
- Both BFS and DFS are simple to implement (but generally inefficient).
- not complete for infinite trees

Introduction

How to get next node?

- Depth-First-Search (DFS):** Maintain Q as a **stack** – Last in/first out
- Lower memory requirement (only need to store part of graph)

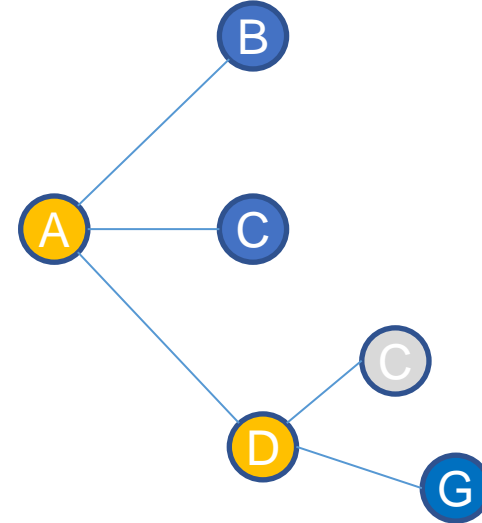
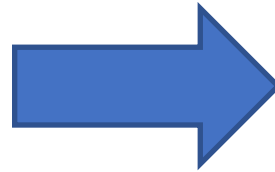
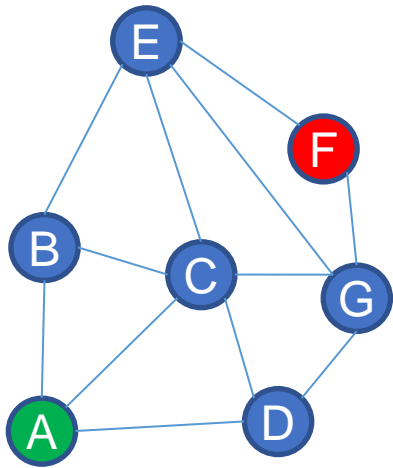


Introduction

How to get next node?

Depth-First-Search (DFS): Maintain Q as a **stack** – Last in/first out

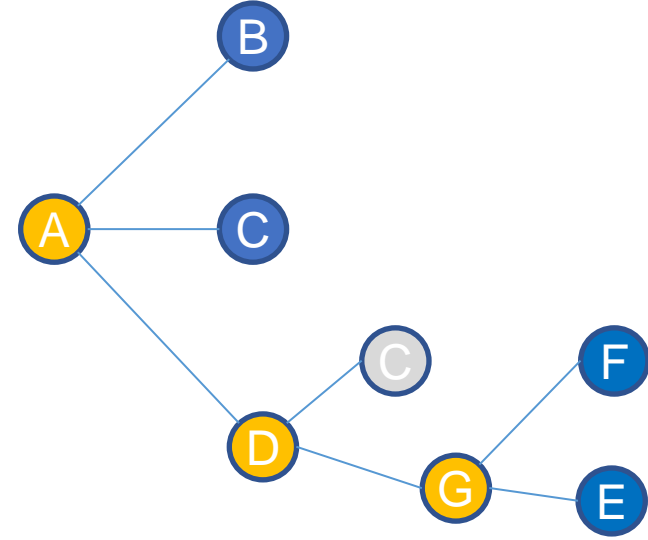
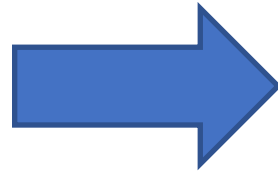
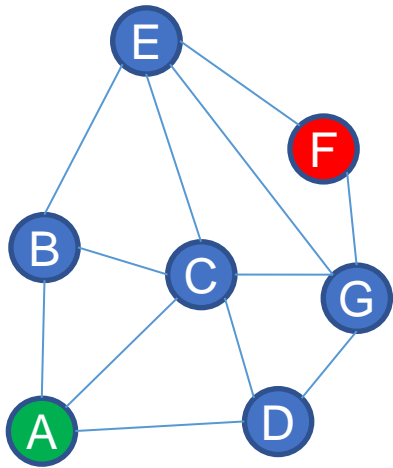
- Lower memory requirement (only need to store part of graph)



Introduction

How to get next node?

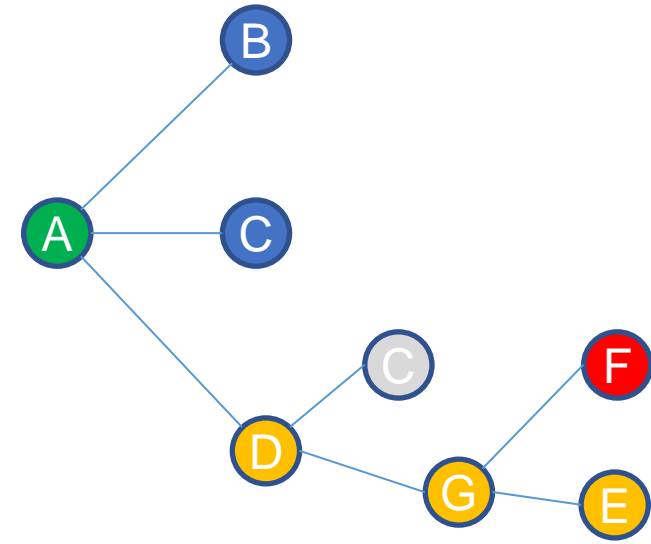
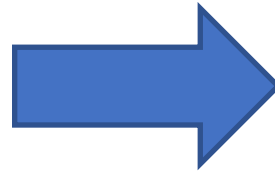
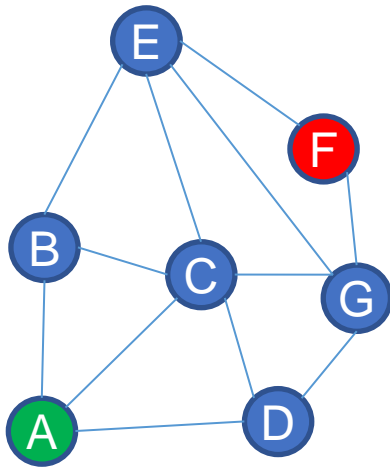
- Depth-First-Search (DFS):** Maintain Q as a **stack** – Last in/first out
- Lower memory requirement (only need to store part of graph)



Introduction

How to get next node?

- Depth-First-Search (DFS):** Maintain Q as a **stack** – Last in/first out
- Lower memory requirement (only need to store part of graph)



Dijkstra

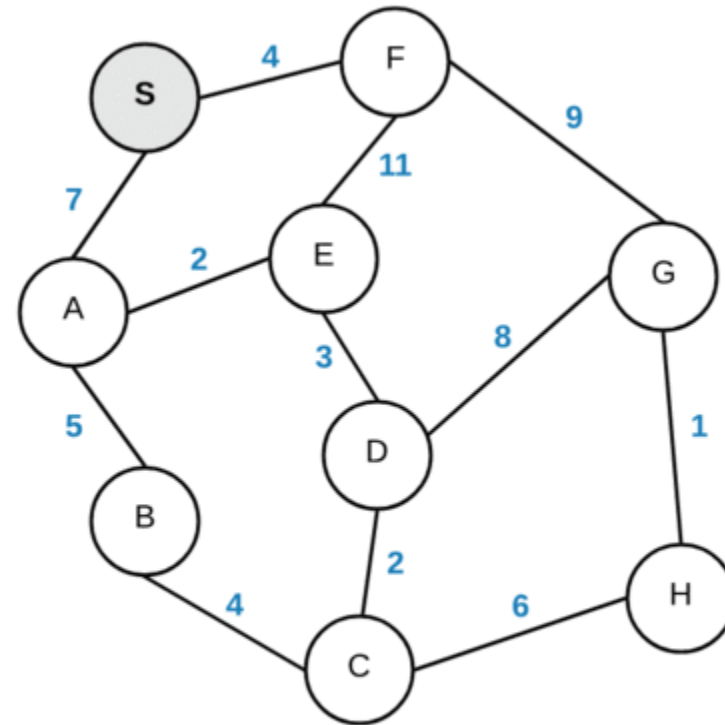
Dijkstra

Best-First (BF, Dijkstra): next q : $\rightarrow q = \operatorname{argmin}_{q \in Q} C(q)$

- Node will enter Q at most **once**
- Requires costs to be non-negative

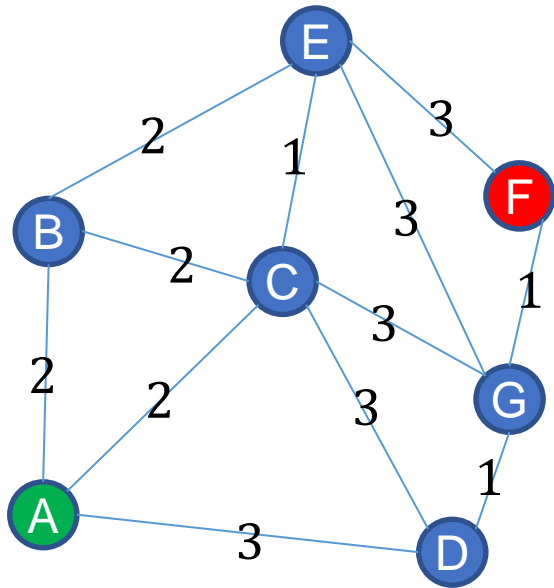
Expanding **from closest** to start (BFS with edge costs)

Q is ordered according to currently known best cost to arrive



Dijkstra

How it works



Q

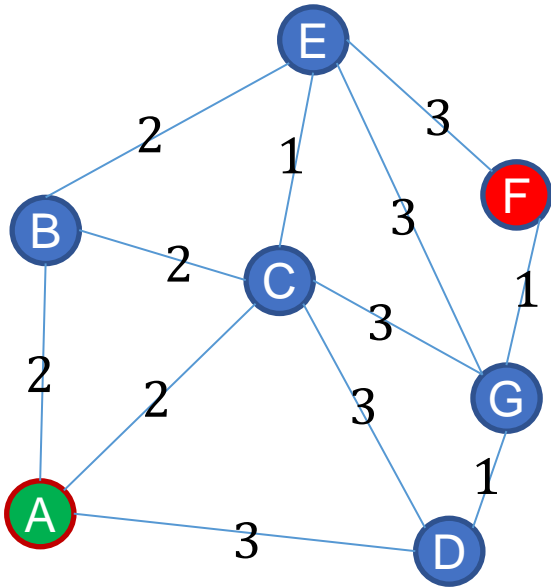
Checked

A(0)

Add in Q ordered by cost

Dijkstra

How it works



Q

B(2), C(2),
D(3)

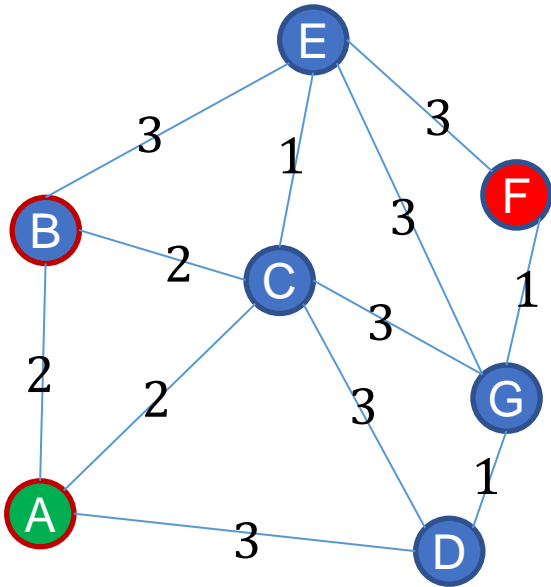
Checked

A(0)

*Move in checked;
place connected in Q ranked by cost*

Dijkstra

How it works



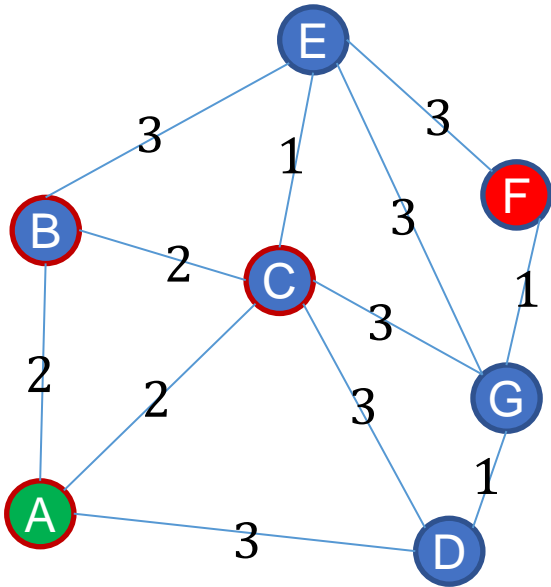
Q
C(2),
D(3),E(5)

Checked
A(0), B(2)

*Move in checked;
place connected in Q ranked by cost
(update if the same is lower)*

Dijkstra

How it works



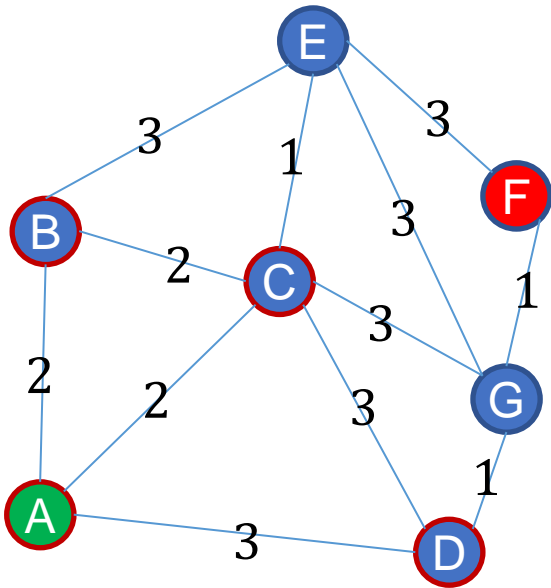
Q
D(3), E(3),
G(5)

Checked
A(0),
B(2), C(2)

*Move in checked;
place connected in Q ranked by cost
(update if the same is lower)*

Dijkstra

How it works



Q
E(3),G(4)

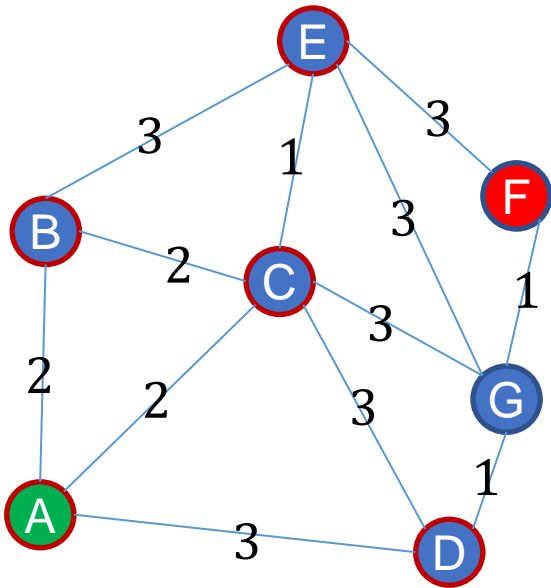
Checked

A(0),
B(2),C(2),
D(3)

*Move in checked;
place connected in Q ranked by cost
(update if the same is lower)*

Dijkstra

How it works



Q
G(4), F(6)

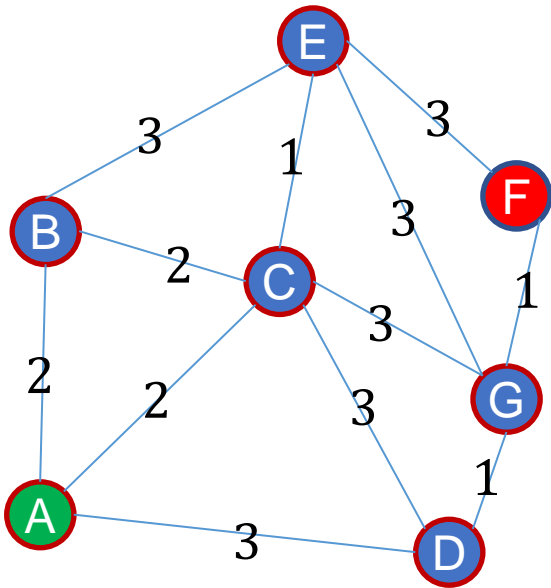
Checked

A(0),
B(2), C(2),
D(3), E(3)

*Move in checked;
place connected in Q ranked by cost
(update if the same is lower)*

Dijkstra

How it works



Q

F(5)

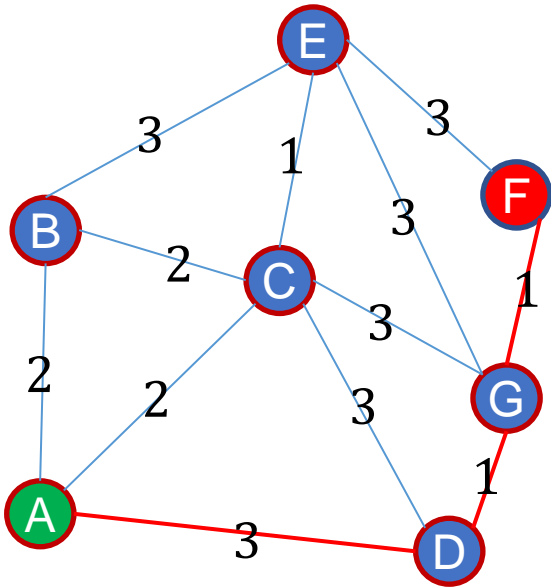
Checked

A(0),
B(2), C(2),
D(3),
E(3), G(4)

*Move in checked;
place connected in Q ranked by cost
(update if the same is lower)*

Dijkstra

How it works



Q

[]

Checked

A(0),
B(2), C(2),
D(3), E(3),
G(4), F(5)

Goal reached

Consider parenting: $F \leftarrow G \leftarrow A$

Software requirements

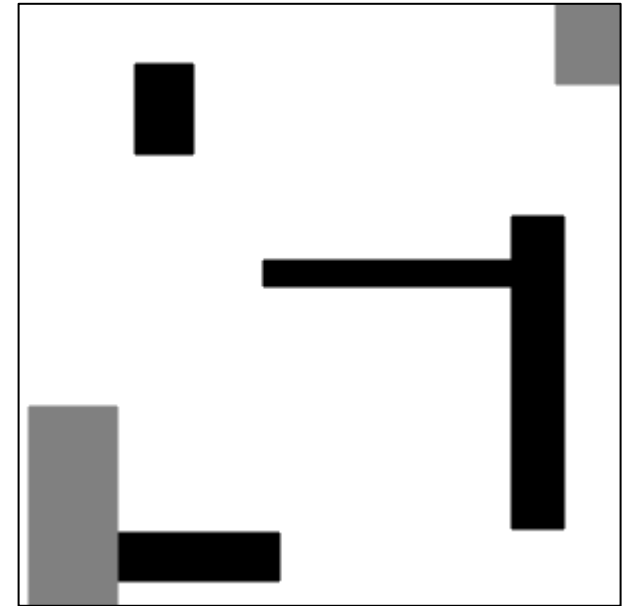


Implementation: an example

Create Image

- Dimension: \sim 30x30 pixels;
- black walls; grey unknown;
- Save as *.png (24 bit);

- ✓ Win: Paint
- ✓ Ubuntu: KolourPaint



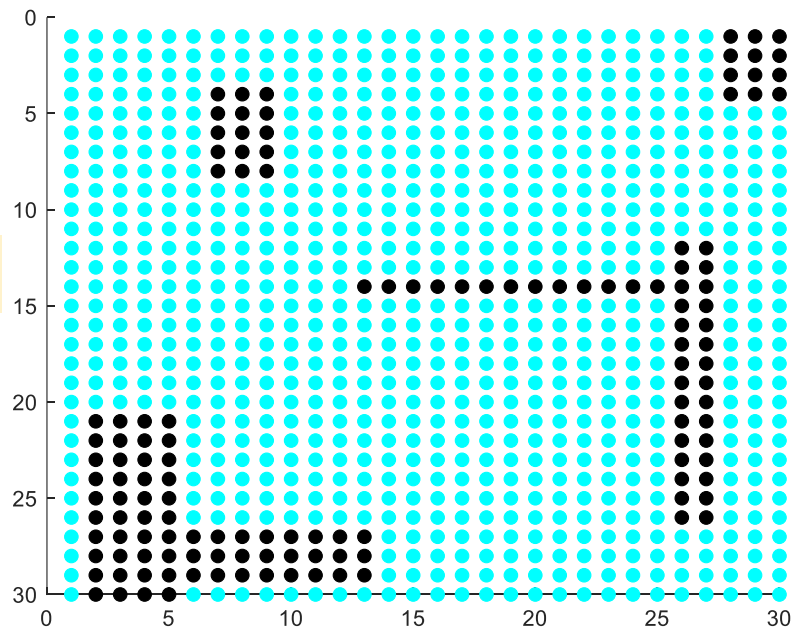
Implementation: an example

Image-Map

```
map_rgb = imread('mappa_test_red.png');  
BW = im2bw(map_rgb,0.7);
```

RGB to bool Matrix
(try different threshold values..)

```
plot_map(BW)
```



Implementation: an example

Initialize variables & edge - matrix

% initialize variables

```
G=-1*ones(size(BW,1)*size(BW,2));
```

```
dist=inf*ones(size(BW,1)*size(BW,2),1);
```

```
prec=inf*ones(size(BW,1)*size(BW,2),1);
```

```
odelist=-1*ones(size(BW,1)*size(BW,2),1);
```

Edge- matrix (tree dependencies)

C(q)

parenting

Q (open queue)

Implementation: an example

Edge - matrix

Example: 4 nodes in a row

1	2	3	4



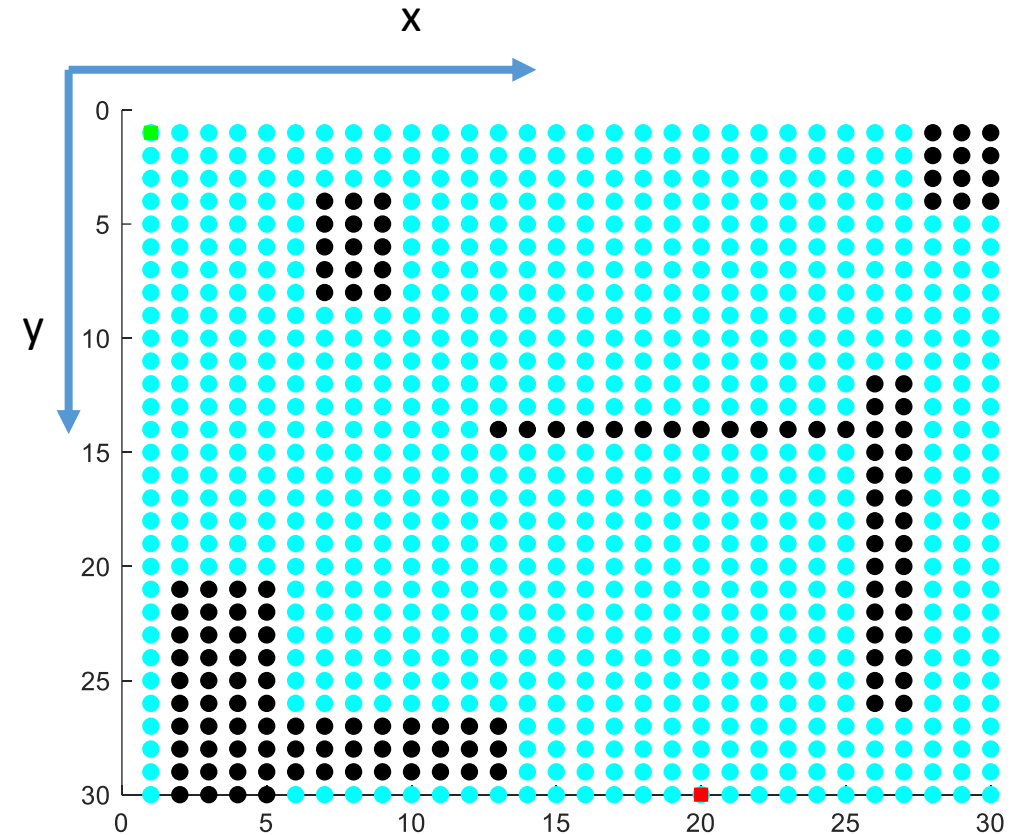
Is it connected to that node?

	1	2	3	4
1	0	1	-1	-1
2	1	0	-1	-1
3	-1	1	-1	1
4	-1	-1	-1	0

Implementation: an example

Start- goal

```
start_pos = [1,1];  
goal_pos = [20,30];  
  
start= (start_pos(2)-1)*size(BW,1)+start_pos(1);  
goal= (goal_pos(2)-1)*size(BW,1)+goal_pos(1);  
  
plot(start_pos(1),start_pos(2),'sg','MarkerFaceColor','g')  
plot(goal_pos(1),goal_pos(2),'sr','MarkerFaceColor','r')
```



Implementation: an example

Dijkstra: initialization

```
dist(start)=0;  
act_node=start;
```

```
[~,con_nodes]=find(G(act_node,:)>0);  
nodelist(con_nodes,1) = 1;  
nodelist(act_node,1)=0;
```

$C(q) = 0$

actual q

Find connected &
add in Q

remove q from Q

Implementation: an example

Dijkstra: steps 1/2

```
while any(nodelist(:,1)==1) && act_node~=goal
    i_con=length(con_nodes);
    while i_con>0
        if dist(con_nodes(i_con))>dist(act_node)+1
            dist(con_nodes(i_con))= dist(act_node)+1;
            prec(con_nodes(i_con))=act_node;
        end
        i_con=i_con-1;
    end
end
```

Q ~= 0 & q ~= goal

evaluate C(q')

update parenting

Implementation: an example

Dijkstra: steps 2/2

```
[min_val,~]=min(dist(nodelist(:,1)==1));  
new_nodes=find(dist==min_val);  
tmp_i=1;  
while nodelist(new_nodes(tmp_i),1)~=1  
    tmp_i=tmp_i+1;  
end  
act_node=new_nodes(tmp_i);  
nodelist(act_node,1)=0;  
[~,con_nodes]=find(G(act_node,:)>0);  
i_con=length(con_nodes);  
while i_con>0  
    if nodelist(con_nodes(i_con),1) ~= 0  
        nodelist(con_nodes(i_con),1) = 1;  
    end  
    i_con=i_con-1;  
end  
end
```

select q' in Q
according to $C(q')$

remove q from Q

Find connected &
add in Q

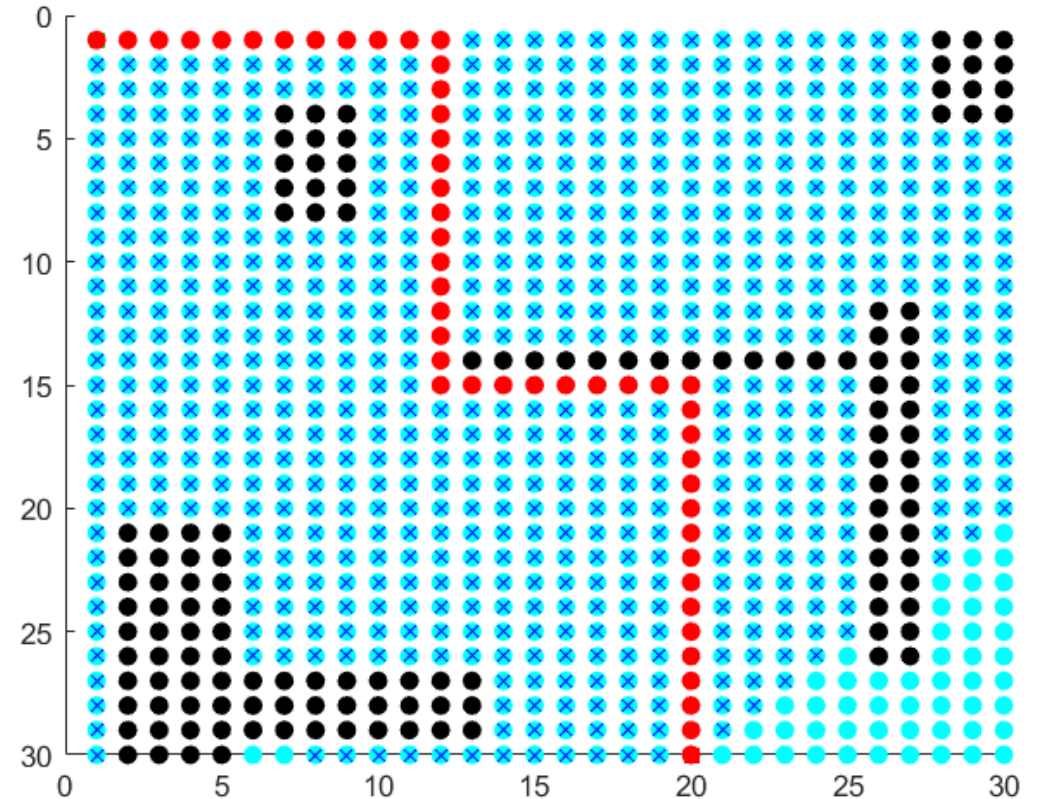
Implementation: an example

Dijkstra: shortest path

```
if dist(goal)<inf
    sol_id=goal;
    path=[];
    while(sol_id~=start)
        path=[sol_id path];
        sol_id=prec(sol_id);
    end
    path=[sol_id path];
```

If a solution is available

recorsively define the
sequence





Assignment IV

What we expect from you

Starting from initial Dijkstra code (v-I) provided:

1. Modify the code to consider diagonals movements as allowed (v-II);
2. Modify the code to implement A* algorithm (v-III);
3. Implement both (v-IV).

Apply the algorithms (all) to the maps created from images (estimate also start&goal) provided and compare the results.

Hints: diagonals & A*

Diagonals:

- Adjacent nodes dist = 1;
- Diagonal nodes dist = $\sqrt{2}$;

A*:

Dijkstra orders by optimal “cost-to-arrival”

A* orders by “cost-to-arrival”+ (approximate) “cost-to-go”

$$C(q') = C(q) + C(q, q') + h(q')$$

where $h(q')$ is an underestimate of “cost-to-go”

Results

- What is mandatory for the report?
 - Plot of shortest path;
 - Evaluate total number of nodes analyzed;
 - Plot the sequence of «actual nodes» computed
 - (compare & comment results)

That's it for today...

See you next time!

S. Arrigoni



POLITECNICO
MILANO 1863