Robotics
Robot Navigation (2)

Tullio Facchinetti
<tullio.facchinetti@unipv.it>

Wednesday 30th October, 2024

http://robot.unipv.it/toolleeo

Maps

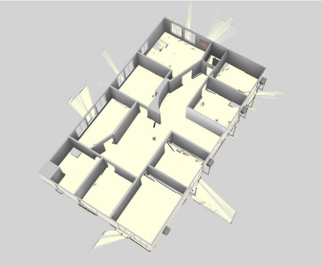
a map is a data structure that represents the environment where the robot (or a generic point) can move

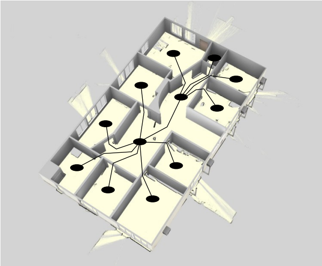
- it represents an important asset for path planning and localization
- it is useful for planning more than one trajectory in the same environment
- the mapping is the incremental process that builds a map using the information gathered from sensors

Different types of mapping

- topological mapping
- geometrical mapping
- occupancy grids

Topological mapping



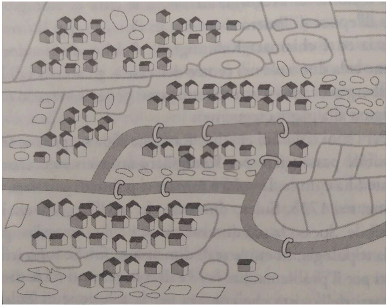


- the representation is based on graphs
- nodes represent relevant points in the environment (e.g., crossroads)
- edges determine the adjacency between nodes

Topological mapping

- scale may be ignored
- paths are rectified

Origin of topology



- The **Seven Bridges of Königsberg** in Prussia (now Kaliningrad, Russia) is a historically notable problem in mathematics
- The problem was to devise a walk through the city that would cross each of those bridges *once and only once*
- **Leonhard Euler** proved it impossible in 1736

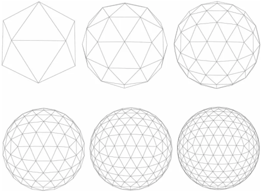
Origin of topology

- *Only the connection information is relevant*; the shape of pictorial representations of a graph may be distorted in any way, without changing the graph itself
- The existence/absence of edges between each pair of nodes is the only significant feature
- The research laid the foundations of **graph theory** and prefigured the idea of **topology**

Geometrical mapping

The representation of the obstacles uses geometrical primitives.

The environment is modeled as a set of lines.
In 3-dimensional spaces, triangles are usually used.



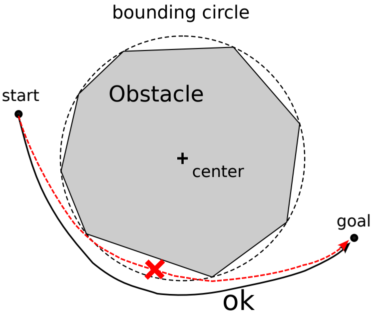
Geometrical mapping

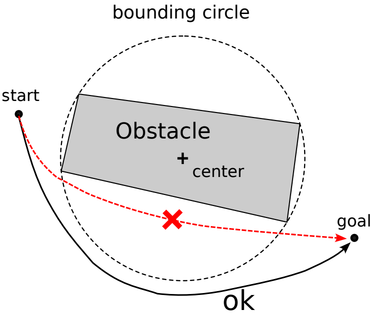
Which geometrical primitive should we use?

It is a trade-off between simplicity of description and accuracy of the representation of the obstacles

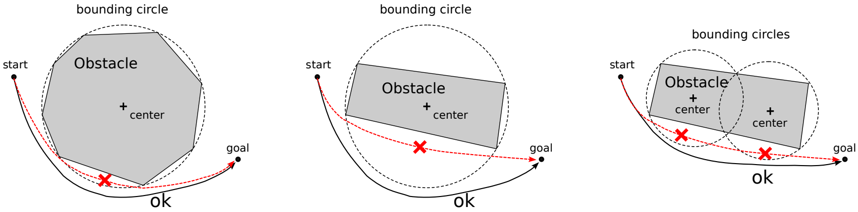
Geometrical mapping: examples

Use of the bounding sphere (bounding circle in 2D)





Geometrical mapping: examples

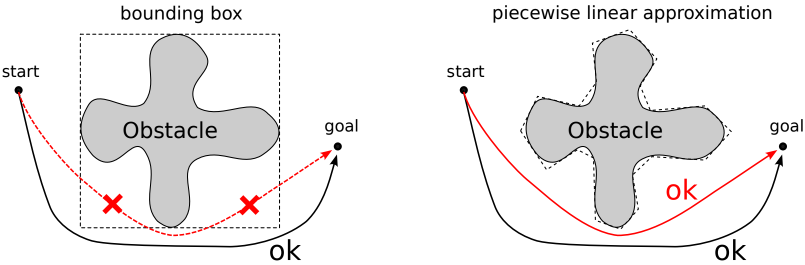


- Good accuracy for obstacles with circle-like shape
- The quality of the generated path depends on the accuracy of the representation

Trade-offs between accuracy and complexity (of the model):

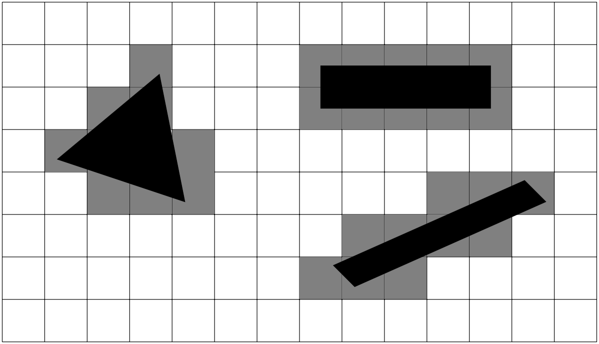
- Which is the best approximation of the obstacles?
- How many parameters are required to model an obstacle?

Geometrical mapping: examples



- Which model of the obstacle is more accurate?
- Which model allows more options for better (shorter) paths?
- How many parameters are required to model the obstacle?

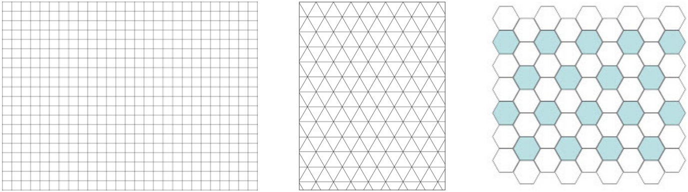
Occupancy grids



- grids are made by adjacent cells having adequate shapes
- for each cell, a flag (boolean value 0/1) indicates whether an obstacle occupies the cell

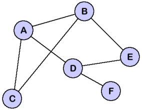
Occupancy grids

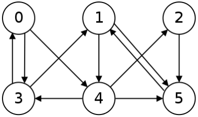
custom shapes of cells

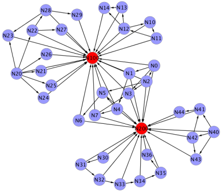


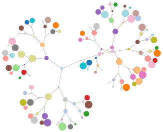
- cells can have any shape to suitably map the shapes in the environment
- the indication of the co-ordinates may require non-standard representation and/or extra information

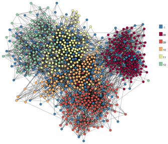
Graphs

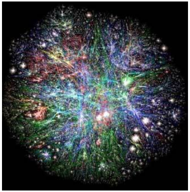

A
B
C
D
E
F


0
1
2
3
4
5

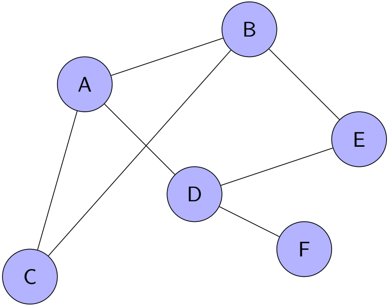

N28
N29
N14
N13
N10
N12
N11
N23
N22
N27
N26
N0
N20
N21
N1
N2
N25
N5
N3
N24
N4
N7
N44
N41
N6
N40
N42
N36
N30
N43
N31
N35
N32
N33
N34







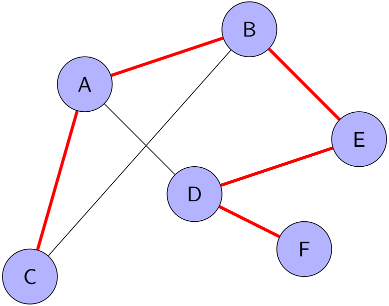
Basics about graphs



- made by n **nodes** $V_1, \ldots, V_n$ ((V)ertex)
- the set of nodes is $\{A, B, C, D, E, F\}$
- nodes are connected by m **edges** $E_1, \ldots, E_m$
- the edge between B and E can also be indicated as $\langle B, E \rangle$

Basics about graphs: some terms

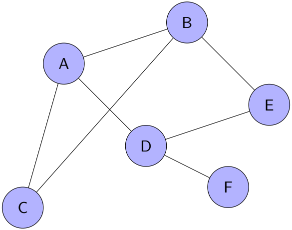
path: succession of nodes connected by edges



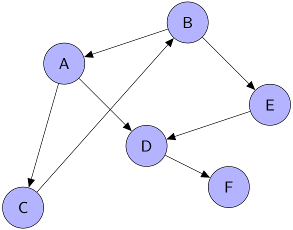
Example of path connecting C and F:
$C \rightarrow A \rightarrow B \rightarrow E \rightarrow D \rightarrow F$

Basics about graphs: some terms

(non)oriented graph: edges (do not) have an orientation (arrows)



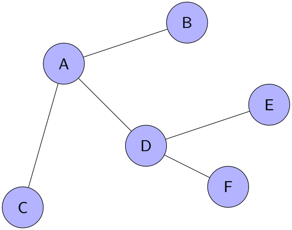
non-oriented graph



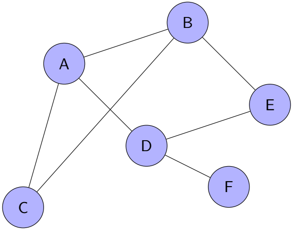
oriented graph

Basics about graphs: some terms

(a)cyclic graph: there are (no) closed circuits in the graph, i.e., there are (no) paths starting from and getting back to the same node going through distinct nodes



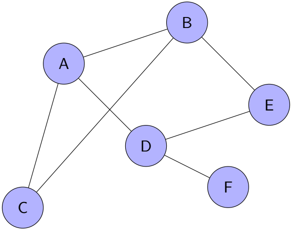
acyclic graph



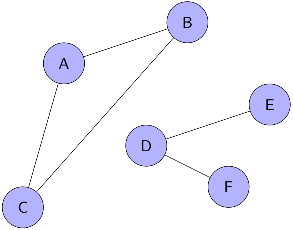
cyclic graph

Basics about graphs: some terms

(dis)connected graph: for each pair of nodes, there is (not) a path connecting it



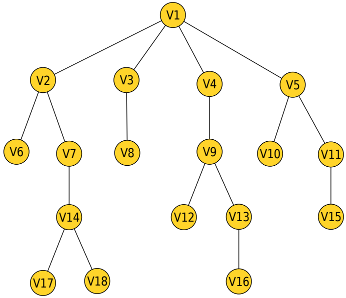
connected graph



disconnected graph

The tree

a tree is a non-oriented connected acyclic graph



Some terms (by examples):

- node $V1$ is said the **root** of the tree
- node $V2$ is said **parent** of $V6$ and $V7$
- $V6$ and $V7$ are the **children** of $V2$
- a **sub-tree** starts in a node and includes the set of nodes below

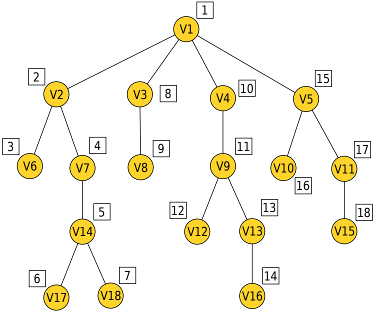
The visit of a tree

> the visit consists in examining (visiting) the nodes of a graph to search a node associated with the desired information

- the application of graphs to the robot navigation, thus to the motion from a starting point to the goal, uses a visit to generate the path to follow
- the searched node is the goal

Depth-first search (pre-order)

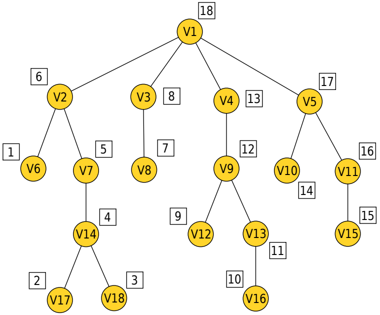
the parent node is visited first, then children are visited in depth-first post-order order



$$V_1 \to V_2 \to V_6 \to V_7 \to V_{14} \to V_{17} \to V_{18} \to V_3 \to V_8 \to V_4 \to V_9 \to V_{12} \to V_{13} \to V_{16} \to V_5 \to V_{10} \to V_{11} \to V_{15}$$

Depth-first search (post-order)

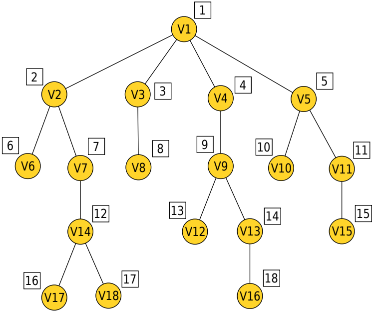
all children are visited with a depth-first pre-order search, before visiting the parent node



$V_6 \rightarrow V_{17} \rightarrow V_{18} \rightarrow V_{14} \rightarrow V_7 \rightarrow V_2 \rightarrow V_8 \rightarrow V_3 \rightarrow V_{12} \rightarrow V_{16} \rightarrow V_{13} \rightarrow V_9 \rightarrow V_4 \rightarrow V_{10} \rightarrow V_{15} \rightarrow V_{11} \rightarrow V_5 \rightarrow V_1$

Breadth-first search

it visits all nodes at the present depth prior to moving on to the nodes at the next depth level



$V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_5 \rightarrow V_6 \rightarrow V_7 \rightarrow V_8 \rightarrow V_9 \rightarrow V_{10} \rightarrow$
$V_{11} \rightarrow V_{14} \rightarrow V_{12} \rightarrow V_{13} \rightarrow V_{15} \rightarrow V_{17} \rightarrow V_{18} \rightarrow V_{16}$

Visibility maps

the map is based on a visibility graph

nodes

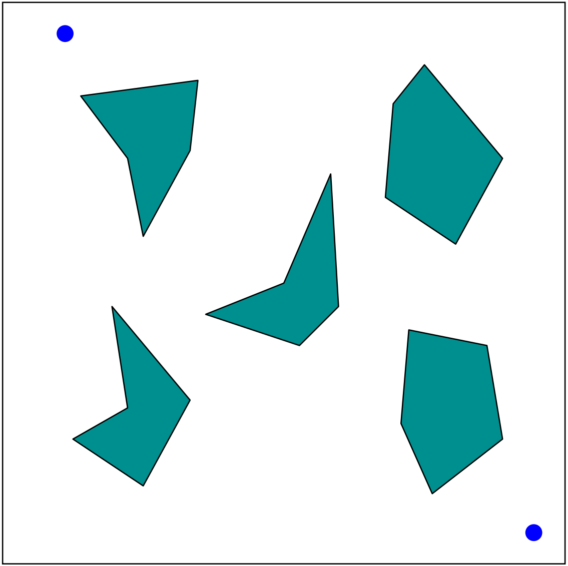
- the start location and the goal
- all the vertices of obstacles

edges

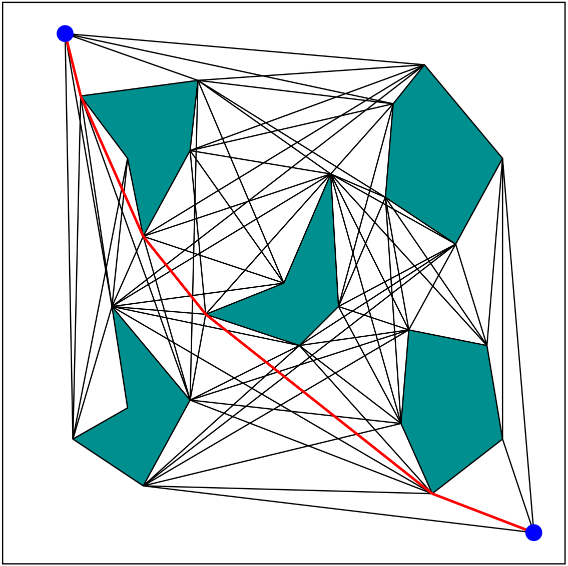
- there is an edge from node v to node w i.i.f.

$$\forall \lambda \in [0,1] : \lambda v + (1-\lambda) w \in \mathcal{Q}_{\text{free}}$$

Visibility graph: example



Visibility graph: example



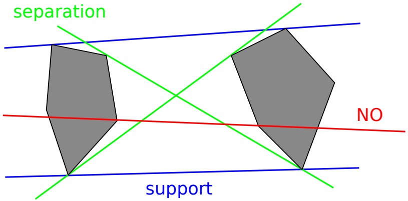
Reduced visibility graph

a visibility graph may include many redundant, non necessary edges

non necessary edges can be eliminated considering some peculiar features:

1. segments of support
2. segments of separation

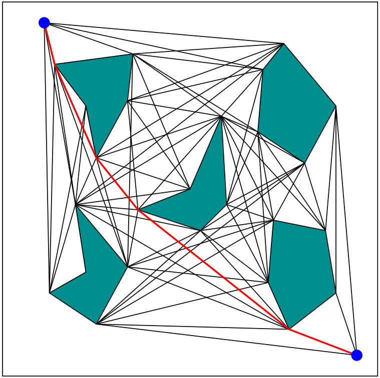
Reduced visibility graph

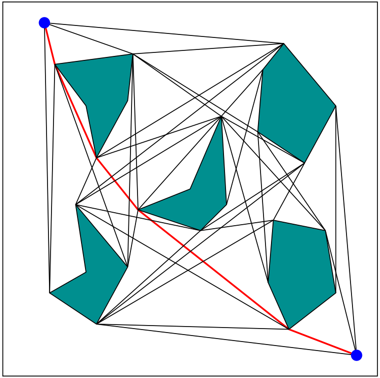


all the edges that are not support nor separation segments are eliminated

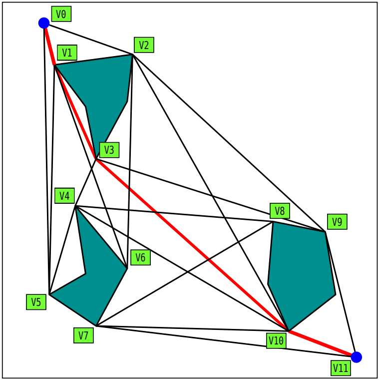
actually, all segments that would intersect an obstacle are eliminated

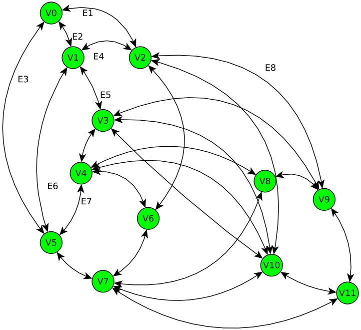
Example of reduced visibility graph



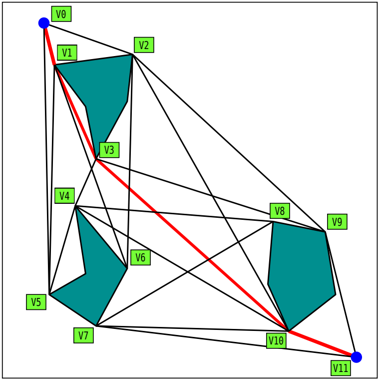


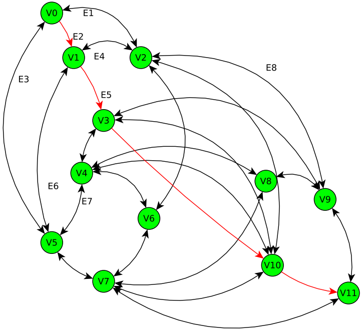
Representation of a visibility graph





Representation of a visibility graph





Visibility graph: construction

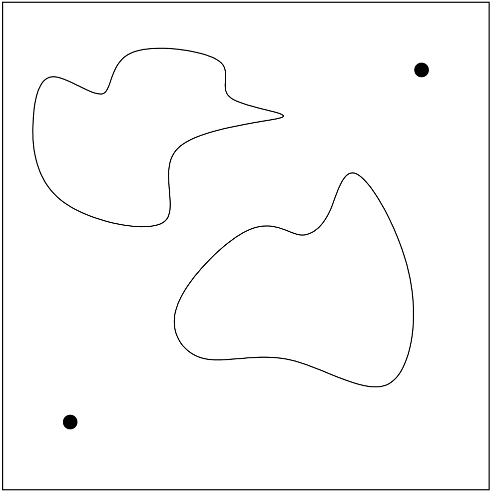
- $V = \{v_1, \ldots, v_n\}$ is the set of vertices
- for each $v_i \in V$, the segment $\overline{v_i v_j}$ must be checked for intersections with obstacles $\forall v_j \neq v_i$
- the number of segments $\overline{v_i v_j}$ to check for intersections is $O(n^2)$
  - there are n vertices
  - each vertex can be connected to the remaining $n - 1$ vertices
- for each segment $\overline{v_i v_j}$ the intersection must be checked against the edges of all obstacles, that are $O(n)$

the overall complexity is $O(n^3)$

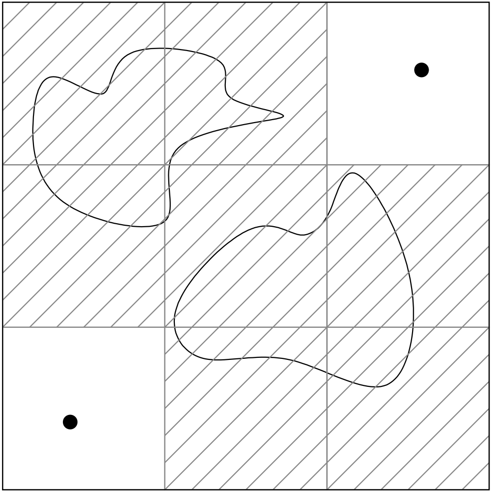
Grid-based maps

- the space is divided in adjacent cells
  - shape and size can change depending on the problem to solve
- the space is mapped such that a cell containing a piece of obstacle is marked as occupied; it is free otherwise
- the resolution of the map is determined by the size of cells

Effect of the resolution on path planning

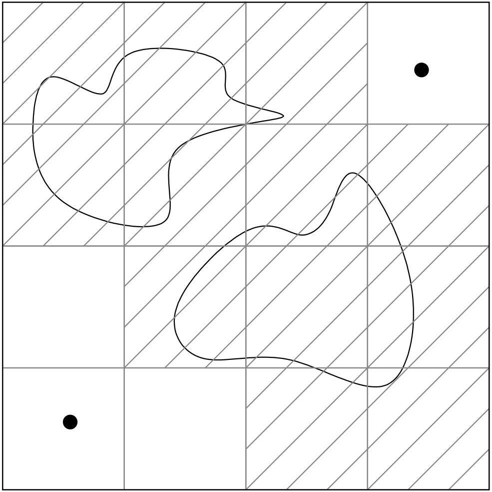


Effect of the resolution on path planning



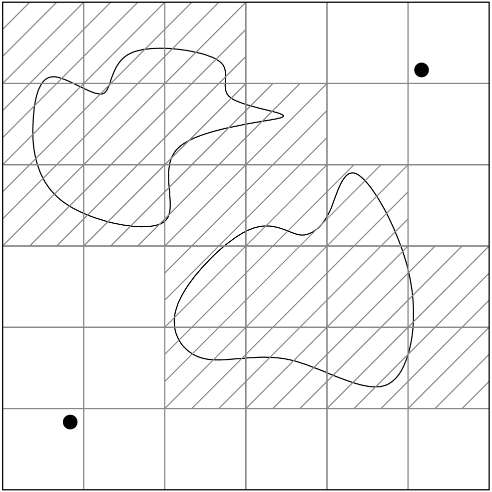
Resolution: 3×3 cells

Effect of the resolution on path planning



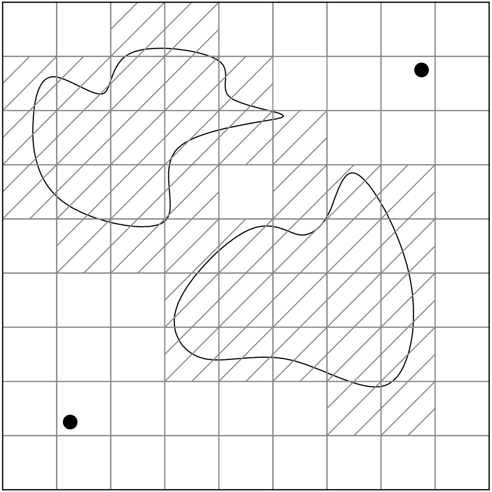
Resolution: 4×4 cells

Effect of the resolution on path planning



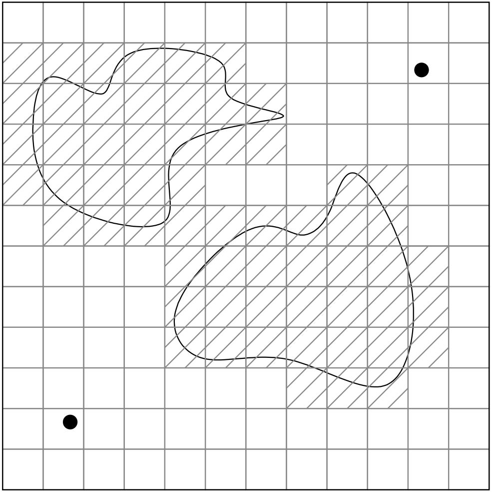
Resolution: 6×6 cells

Effect of the resolution on path planning



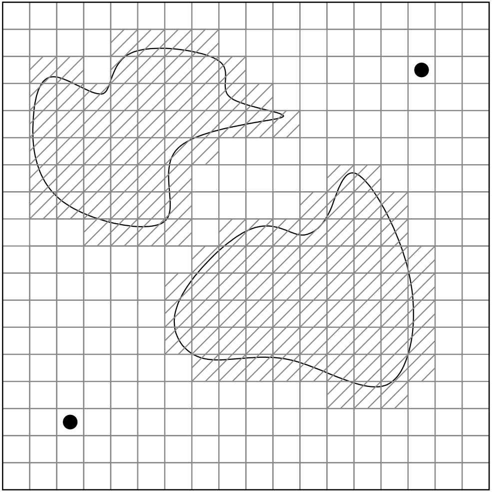
Resolution: 9×9 cells

Effect of the resolution on path planning



Resolution: 12×12 cells

Effect of the resolution on path planning



Resolution: 18×18 cells

Resolution completeness

- the success of the trajectory planning depends on the resolution
- an higher resolution increases the chance to find a path
- however, it requires more memory space to store the map: each cell requires at least 1 bit to mark it as free or occupied
- moreover, it requires more computing time to process the data, since there are more data

it is a trade-off between completeness and time/space requirements

The concept of “adjacent cell”

in case of **square cells**, the adjacency of two cells can be of two types:

4 points connectivity

d = 2	d = 1	d = 2
d = 1	d = 0	d = 1
d = 2	d = 1	d = 2

8 points connectivity

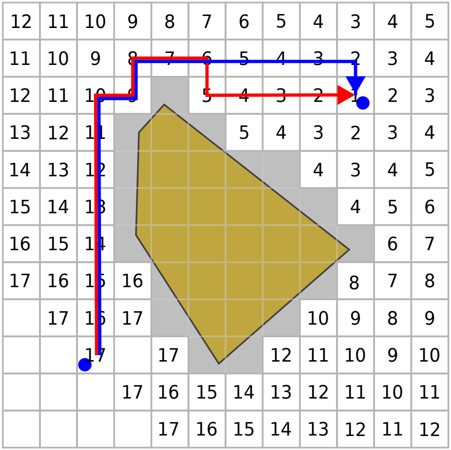
d = 1	d = 1	d = 1
d = 1	d = 0	d = 1
d = 1	d = 1	d = 1

d is the distance from the cell in the center, measured in number of cells (hops)

Wave-front algorithm

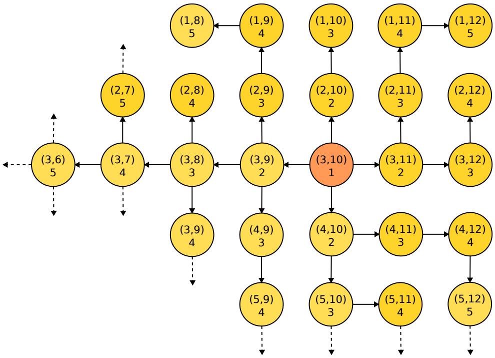
1. mark the cell containing the goal with $i = 1$
2. mark with $i + 1$ every adjacent cell to the one marked with i
3. repeat step 2 until the cell containing the starting point is marked or all cells have been marked
4. use the gradient descent to go from the starting cell to the goal cell

Wave-front algorithm: example

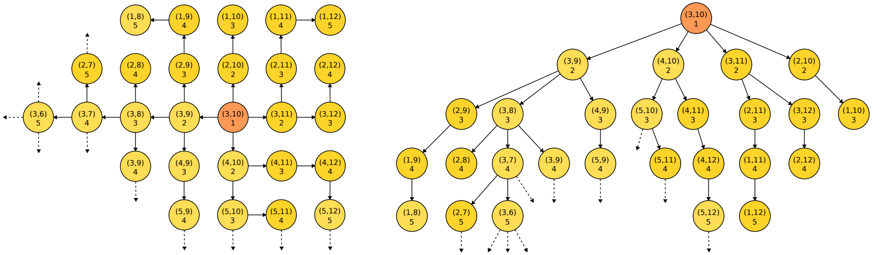


Wave-front algorithm and tree visit

	1	2	3	4	5	6	7	8	9	10	11	12
1	12	11	10	9	8	7	6	5	4	3	4	5
2	11	10	9	8	7	6	5	4	3	2	3	4
3	12	11	10	9		5	4	3	2	1	2	3
4	13	12	11				5	4	3	2	3	4
5	14	13	12						4	3	4	5
6	15	14	13							4	5	6
7	16	15	14								6	7
8	17	16	15	16						8	7	8
9		17	16	17					10	9	8	9
10			17		17			12	11	10	9	10
11				17	16	15	14	13	12	11	10	11
12					17	16	15	14	13	12	11	12

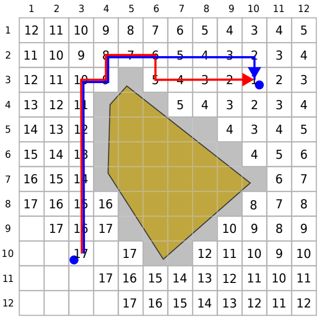


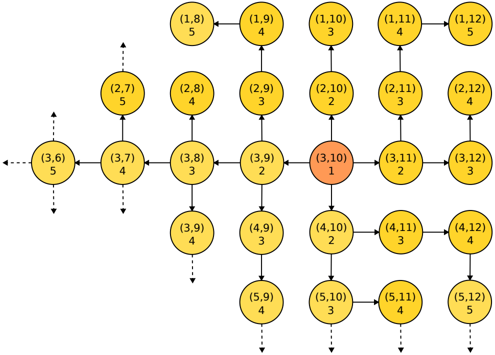
Wave-front algorithm and tree visit



the assignment of labels to the cells can follow the logic of the **breadth-first visit** of a tree

Efficiency of the wave-front algorithm





the breadth-first search **is inefficient**: it may visit (i.e., assign labels) to a large, uninteresting part of the area

Wave-front algorithm: characteristics

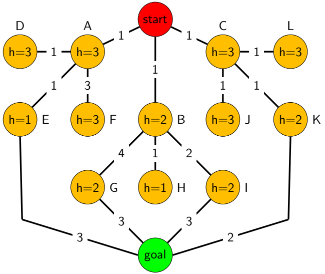
- **complete**: if a path exists (at a given resolution), it is found
- **low efficiency**: a large amount of “non necessary” cells can be visited to assign the numbers to the cells
- **optimal**: it finds the shortest path (measured in number of cells)

these features arise from the breadth-first search performed on the grid

The A^* algorithm

- developed to **find a path in a graph**
- based on the knowledge of the goal location
- uses an heuristic search
- the heuristic is used to select the direction of movement
- it takes into account the distance between the current location, the starting point and the goal

The A^* algorithm: some notation

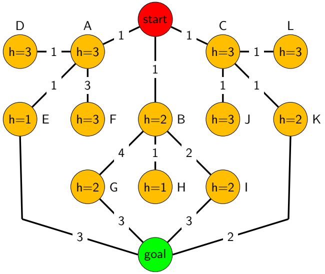


- $c(V_1, V_2)$: cost (e.g., length) of the edge connecting V_1 to V_2
- $Neigh(V)$: set of nodes adjacent to V
- O: set of nodes "under examination" (open set - priority queue)
- C: set of visited nodes (closed set)

Examples

- $c(A, D) = 1$
- $Neigh(C) = \{start, L, J, K\}$

The A^* algorithm: some notation



- $g(V)$: cost of the backward path from V to p_{start}
- $h(V)$: heuristic function; estimates the cost from V to p_{goal}
- $f(V) = g(V) + h(V)$: estimation of the total cost of the path from p_{start} to p_{goal} passing through V

Examples

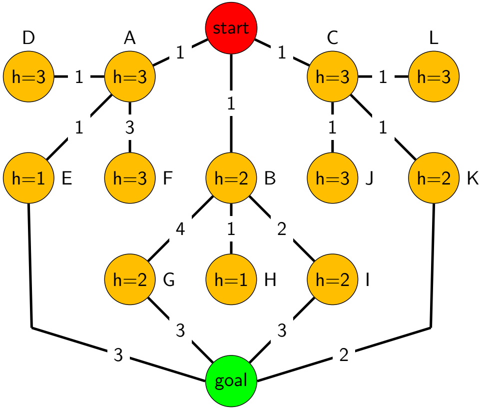
- $g(E) = 2$
- $h(E) = 1$
- $f(E) = g(E) + h(E) = 2 + 1 = 3$

The A^* algorithm: pseudo-code

input: the graph to analyze
output: the backward path from p_{goal} to p_{start}

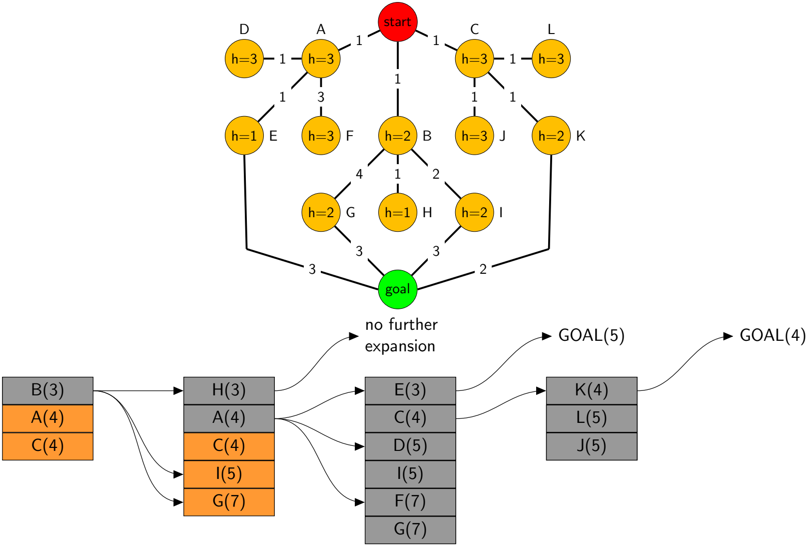
```
Add V_start to O
while O is not empty do
    Select V_best ∈ O : f(V_best) ≤ f(V) ∀V ∈ O
    Move V_best from O to C
    if V_best = p_goal then
        Path found (cost is g(p_goal))
        Move from O to C all nodes with cost c ≥ g(p_goal)
    end if
    for all V ∈ Neigh(V_best) : V ∉ C do
        if V ∉ O then
            add V to O
        else
            if g(V_best) + c(V_best, V) < g(V) then
                Connect V to V_best
            end if
        end if
    end for
end while
if No path found then
    There are no existing paths
end if
```

Example of application of A^*

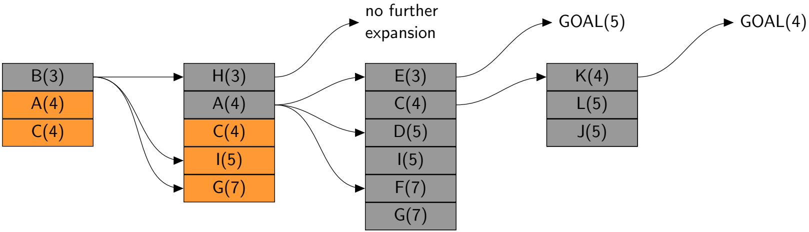


- nodes contain the value of the heuristics
- edges are labelled with edge's costs

Example of application of A^*



Example of application of A^*



- at each step the node in the priority queue having the lower cost is expanded
- once the goal is found, all the nodes in the priority queue having cost higher than the cost of the path are removed
- all the remaining nodes may bring to a lower cost path, thus they are examined

Features of A^*

completeness

- A^* generates a tree, which has no cycles by definition
- in a finite tree there is a finite number of distinct paths
- at most, every path is examined
- eventually, A^* terminates by finding a path if it exists

however...
completeness does not necessarily mean
that A^* is efficient

Features of A^*

efficiency

- A^* does not necessarily examine all the possible paths
- it explores in decreasing order all the paths that have the best chances (heuristic function) to lead to the goal
- it terminates when no nodes provide better chances than the current path
- this is the actual definition of "efficiency"
- if all paths are explored without finding a solution, then no valid path exists (completeness!)

however...
efficiency does not necessarily mean that A^* is optimal

Features of A^*

optimality

- once a path to the goal is found (assuming it has cost c):
    - every node in the priority queue having cost less than c are explored
    - such paths are explored until their cost remains less than c
- A^* explores new paths until the priority queue becomes empty
- it concludes the search by finding the path having the lowest cost path

a condition must hold to find an optimal path:

> the heuristic function must be *optimistic* to guarantee that the optimal path is found

Optimistic heuristic function

> an heuristic function is optimistic if it returns an estimate of the distance from the goal that is less or equal to the real distance

let's consider:

- a grid of square cells
- 4 points connectivity
- the distance between two cells is computed using the Manhattan distance

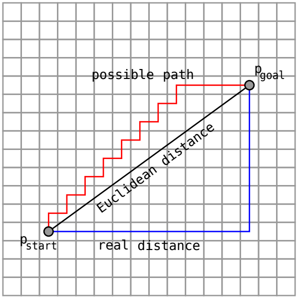
Optimistic heuristic function

Manhattan distance:

$$dist(V, p_{goal}) = \|V.x - p_{goal}.x\| + \|V.y - p_{goal}.y\|$$

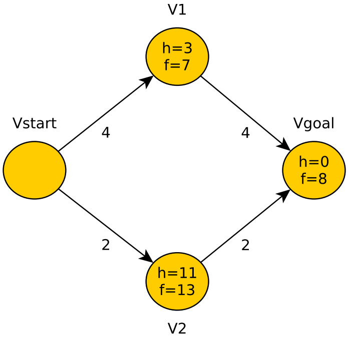
Euclidean distance:

$$dist_2(V, p_{goal}) = \sqrt{(V.x - p_{goal}.x)^2 + (V.y - p_{goal}.y)^2}$$



- the Euclidean distance (heuristic) is always less or equal to the real distance
- is optimistic

Example of non optimistic heuristic



- the heuristic is not optimistic: node $V2$ estimates its distance from the goal equal to 11, while it is 2
- the resulting path passes through $V1$ (cost 8) instead of passing through $V2$ (cost 4)

Example of application of A^*

assumptions:

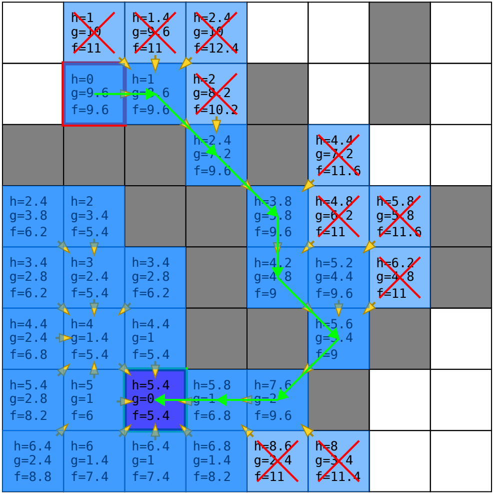
- grid composed by square cells
- 8 points connectivity

heuristic:

- horizontal and vertical distance between cells $= 1$
- diagonal distance $= 1.4$ (approximating $\sqrt{2}$)

> ATTENTION: we are not using an approximated Euclidean distance: the distance from a cell to the one located 2 cells on the right and 1 above is 2.4, not $\sqrt{5}$

Example of application of A^*



Simple variants of A^*

Greedy search

- assumes $f(V) = h(V)$: only considers the estimated best path from the current node

Dijkstra algorithm

- assumes $f(V) = g(V)$: does not use any heuristic
- grows the current shortest path from the starting node