Durham University

COMP52815 Robotics - Planning and Motion

Durham University

Robotics – Planning and Motion

COMP52815

Dr Junyan Hu & Prof Farshad Arvin

Email: junyan.hu@durham.ac.uk
Room: MCS 2060

Lecture: Learning Objectives

The aim of this lecture is to design path planning for robot navigation.

- Objectives:
  1. Obstacle avoidance
  2. Search algorithms
     - Breadth-first
     - Depth-first
     - Wavefront
  3. Dijkstra's Algorithm
  4. A* Algorithm

See also:

- Introduction to Mobile Robot Control, Spyros G. Tzafestas, 2014

Navigation

Navigation

There are two primary scenarios for robot navigation:

- Known environment
- Unknown environment

For a known environment, a full spatial model (map) exists and the task becomes a search for a path (trajectory) to an end goal.
This can becomes more complicated if the environment is dynamic (obstacles move).

In an unknown environment, no map exists so the challenge is more focused on complete exploration of the area.

Path planning

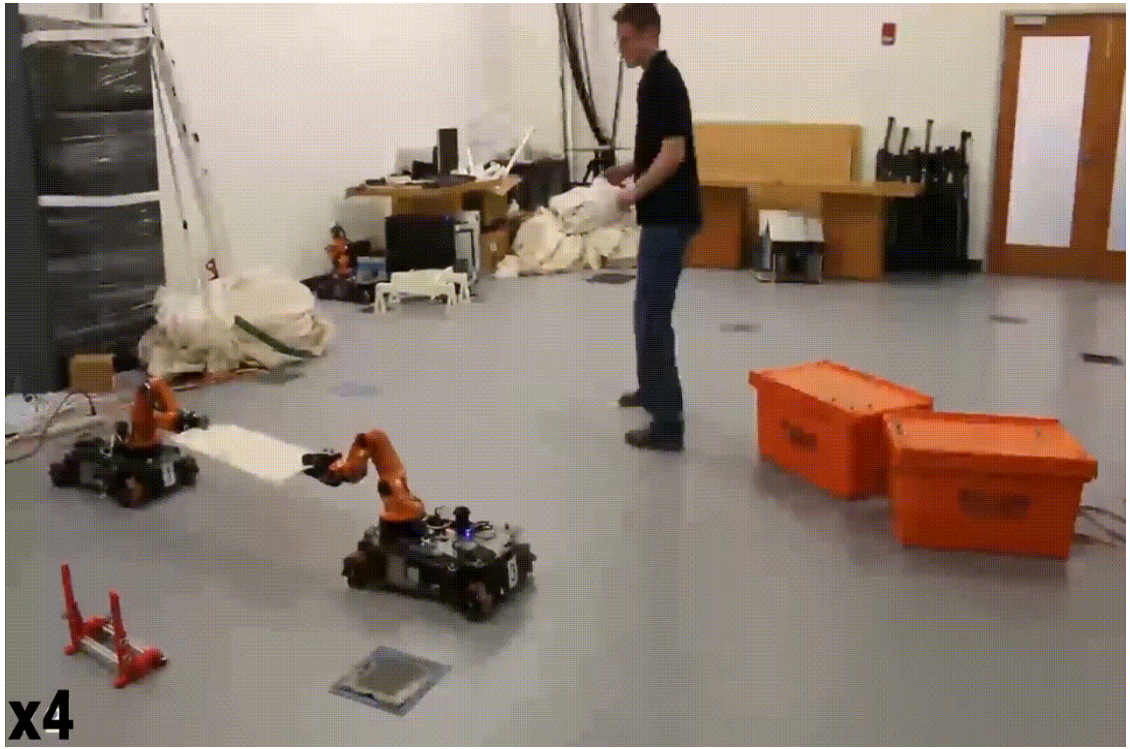
Given a complete map of an environment and a target location, path planning is the process of identifying (searching for) a geometric path which will get the robot to the goal (kinematics).

Trajectory planning is the process of applying a time constraint to the path.

Search algorithm performance is measured in a number of ways:

- Completeness
- Optimality
- Time Complexity
- Space Complexity

Path planning



Obstacle Avoidance

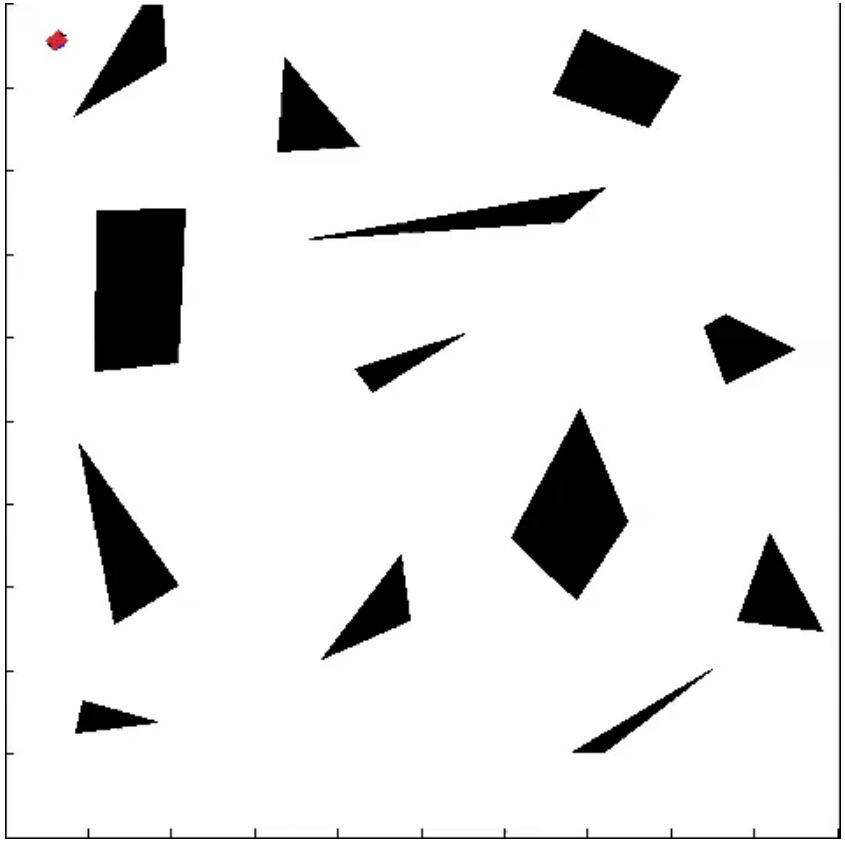
- Robot needs to navigate through the environment without running into obstacles.
- Robot needs to utilize exteroceptive sensors to identify obstacles.
- Example of exteroceptive sensors are: camera, LiDAR, sonar, etc...

Kinect Sensor © Microsoft.

Sonar © SparkFun Electronics.

LiDAR © Hokuyo Automatic Inc.

Obstacle Avoidance



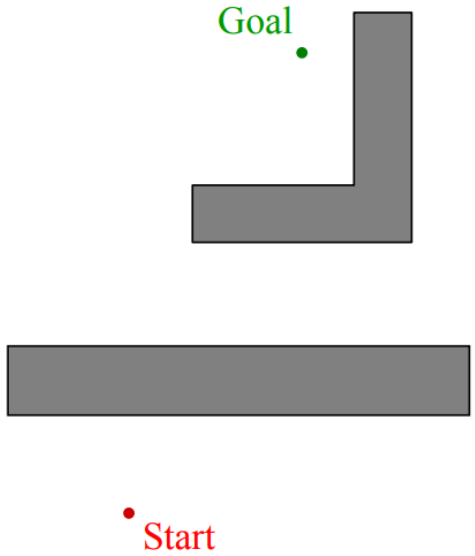
Robot needs to know when it reaches the goal.

Obstacle Avoidance

The bug algorithm:

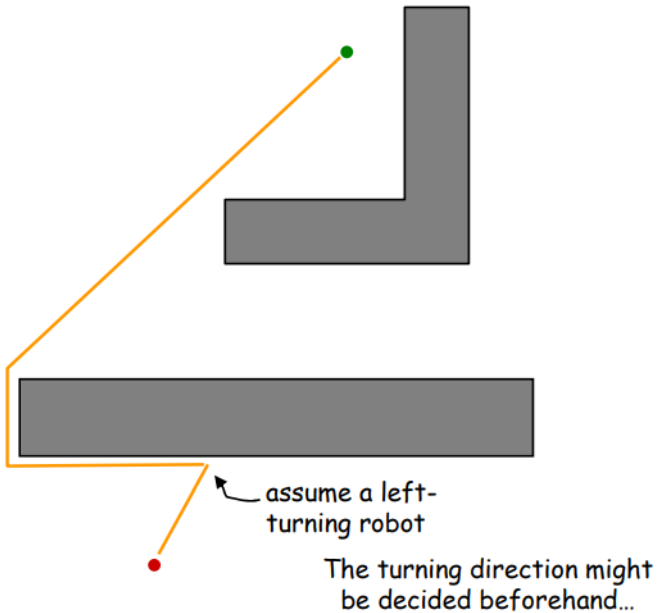
Obstacle Avoidance

The bug algorithm:



- known direction to goal
  - robot can measure distance d(x,y) between pts x and y
- otherwise local sensing walls/obstacles

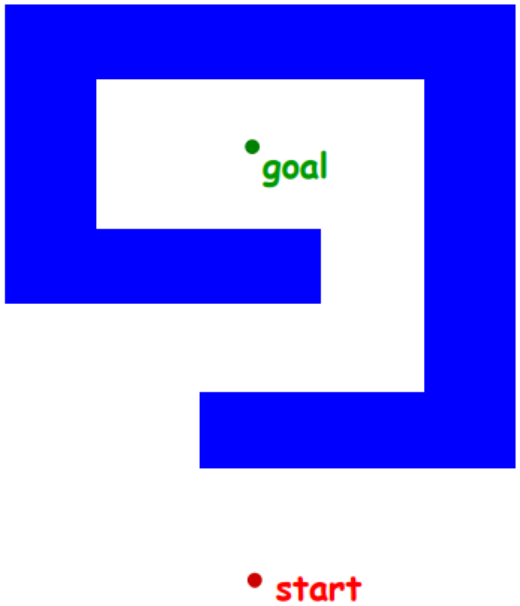
Bug 0 Strategy



1) head toward goal
2) follow obstacles until you can head toward the goal again
3) continue

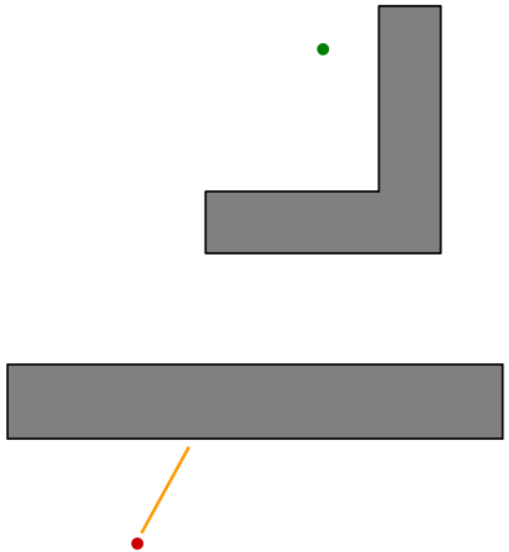
Bug 0 Strategy

Bug 0 won’t work well in this map!



Bug 1 Strategy

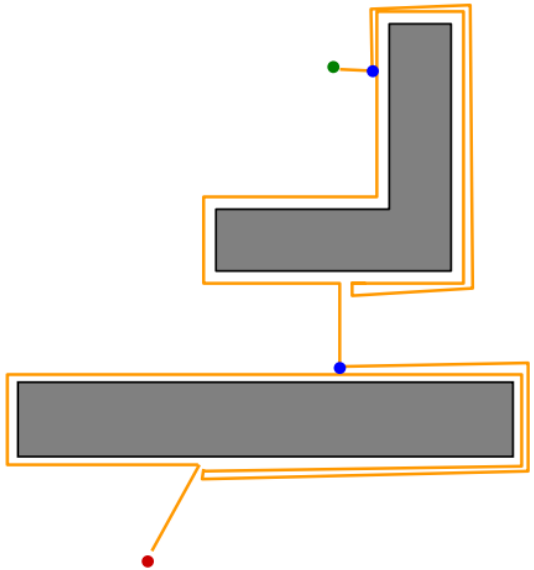
Improve algorithm by adding memory!



1) head toward goal
2) if an obstacle is encountered, circumnavigate it and remember how close you get to the goal
3) return to that closest point (by wall-following) and continue

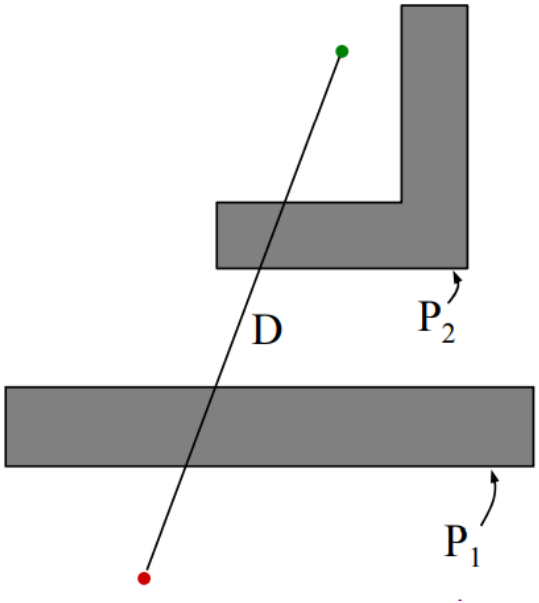
Bug 1 Strategy

Improve algorithm by adding memory!



1) head toward goal
2) if an obstacle is encountered, circumnavigate it and remember how close you get to the goal
3) return to that closest point (by wall-following) and continue

Bug 1 Path Bound



D = straight-line distance from start to goal
Pi = perimeter of the *i* th obstacle

Lower bound: D

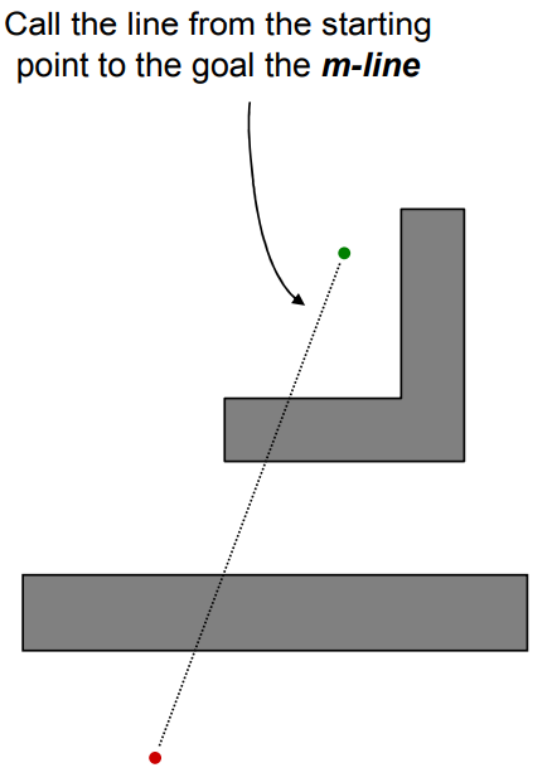
What's the shortest distance it might travel?

Upper bound: $D+1.5\sum_i P_i$

What's the longest distance it might travel?

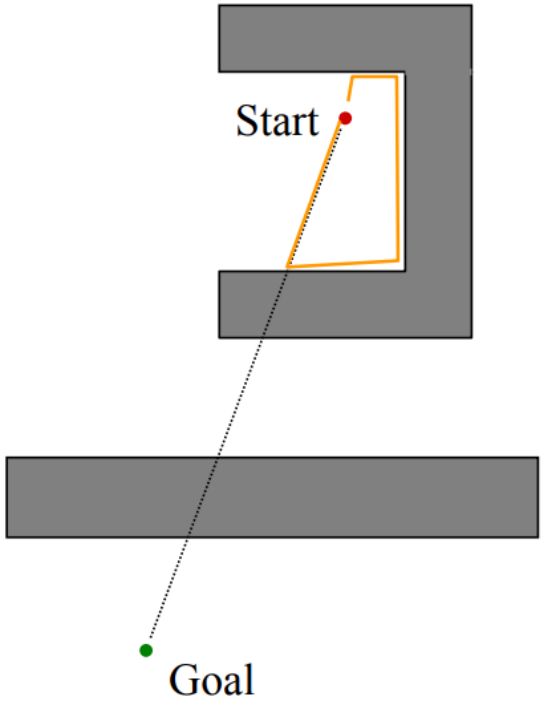
Durham University

Bug 2 Strategy



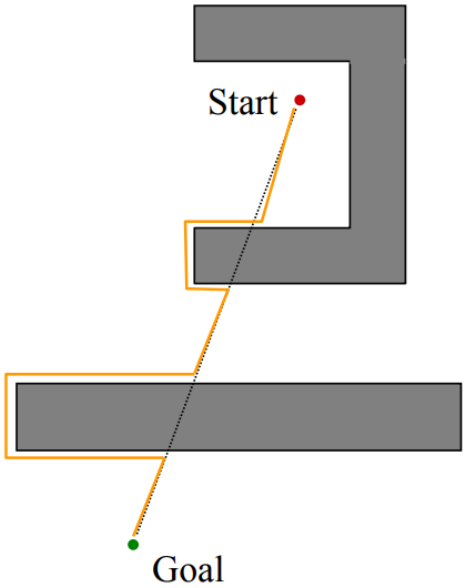
1) head toward goal on the m-line
2) if an obstacle is in the way, follow it until you encounter the m-line again.
3) Leave the obstacle and continue toward the goal

Bug 2 Strategy



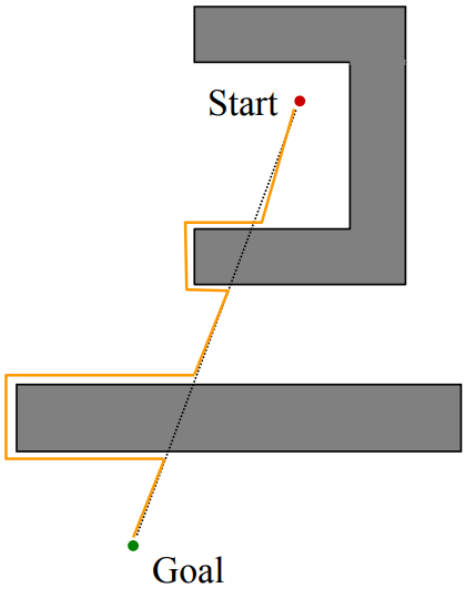
- In this case, re-encountering the m-line brings you back to the start
- Implicitly assuming a static strategy for encountering the obstacle (“always turn left”)

Bug 2 Strategy



1) head toward goal on the m-line
2) if an obstacle is in the way, follow it until you encounter the m-line again *closer to the goal*.
3) Leave the obstacle and continue toward the goal

Bug 2 Path Bound



D = straight-line distance from start to goal
Pi = perimeter of the *i* th obstacle

Lower bound: D

What's the shortest distance it might travel?

Upper bound: $D+\sum_i \frac{n_i}{2} P_i$

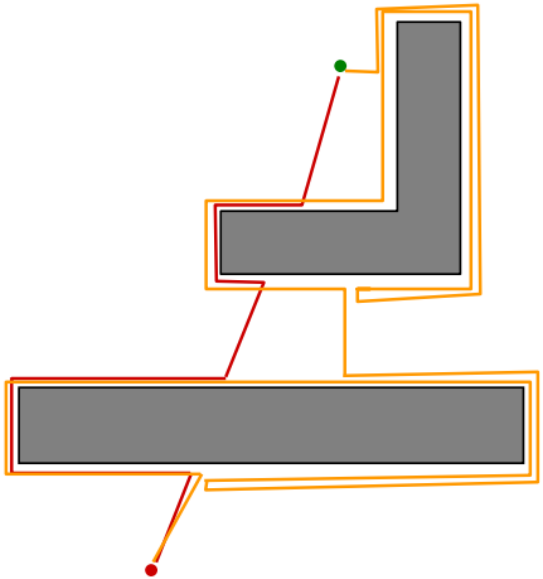
What's the longest distance it might travel?

n_i = # of m-line intersections of the *i* th obstacle

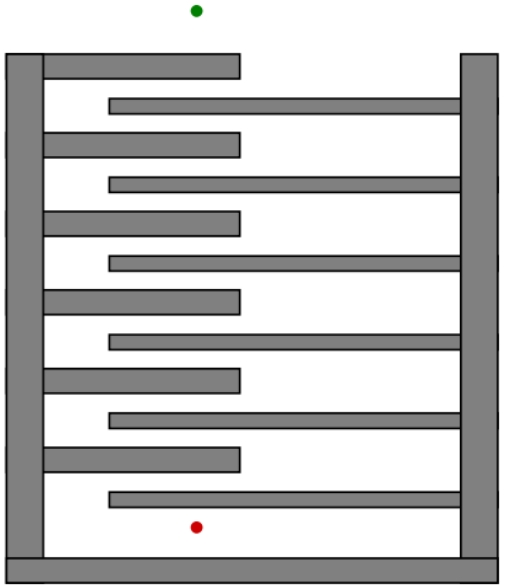
Durham University

Bug 1 VS Bug 2

Bug 2 beats Bug 1



Bug 1 beats Bug 2



Durham University

Bug 1 VS Bug 2

- BUG 1 is an exhaustive search algorithm
  – it looks at all choices before commiting
- BUG 2 is a greedy algorithm
  – it takes the first thing that looks better
- In many cases, BUG 2 will outperform BUG 1, but
- BUG 1 has a more predictable performance overall

Configuration Space

Mobile robots operate in either a 2D or 3D Cartesian workspace with between 1 and 6 degrees of freedom.

The configuration of a robot completely specifies the robot's location.

The configuration of a robot, C, with k DOF can be described with k values: C = $\{q_1, q_2, \ldots, q_k\}$.

These values can be considered a point, p in a k-dimension space (C-space).

C-Space

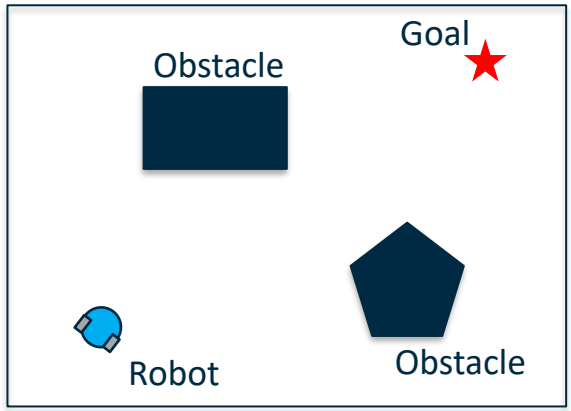
Wheeled mobile robots can be modelled in such a way that the C-Space maps almost directly to the workspace.

$$C = \{x, y, \varphi\}$$

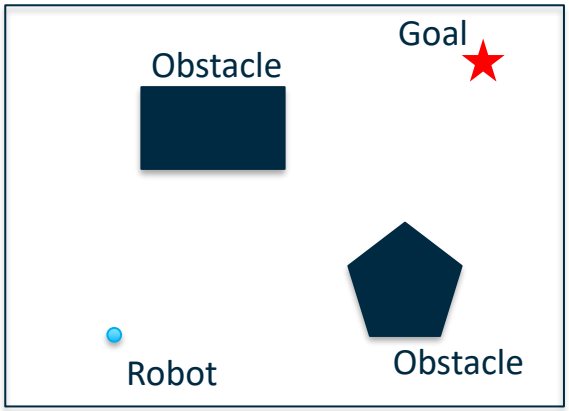
The assumption is often made that the robot is holonomic, however this is not the case for differential drive robots. If the orientation of the robot is not important, this assumption is valid.

C-Space for Mobile Robots

If we assume a circular, holonomic robot, the C-space of a mobile robot is almost identical to the physical space.



Physical Space

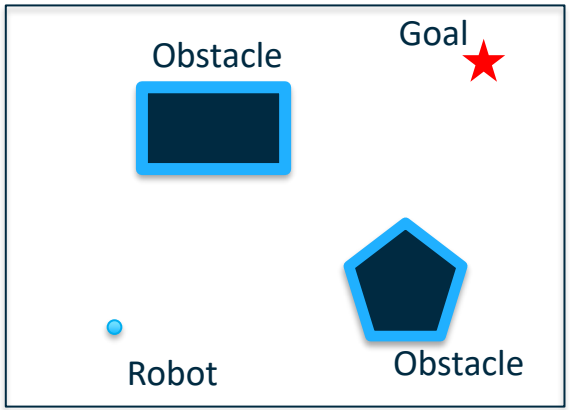


C-Space

C-Space Modification

The robot in C-space is represented as a point, however the robot in the physical space has a finite size.

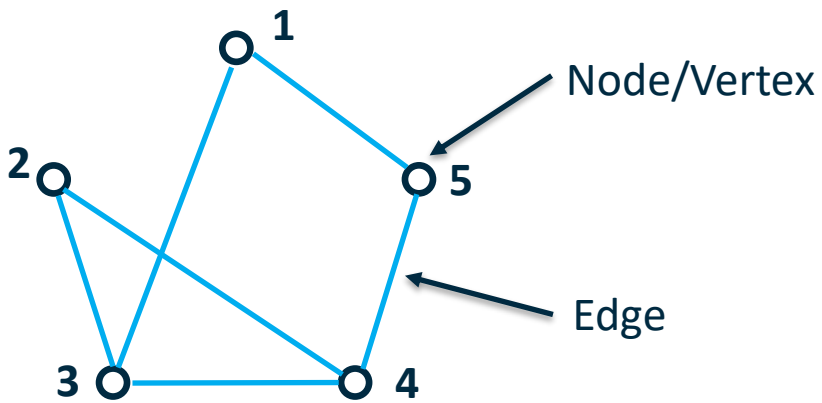
To map the obstacles in C-space, they have to be increased in size by the radius of the robot.



Graphs

The standard search methods used for planning a route are based on graphs.

A graph, G, is an abstract representation which is made up of nodes (Vertices), $V(G)$, and connections (Edges), $E(G)$.



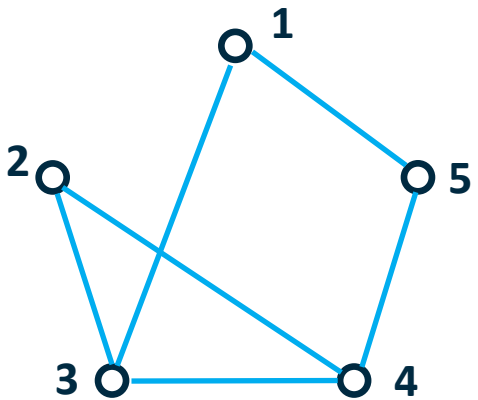
Graph Definitions

The graph below has a vertex set: $V(G) = \{1,2,3,4,5\}$

The degree of a vertex is the number edges with that vertex as an end point (so the degree of node 1 would be 2).

An edge connects two vertices and is defined as (i, j) i.e. connecting vertex i to vertex j.

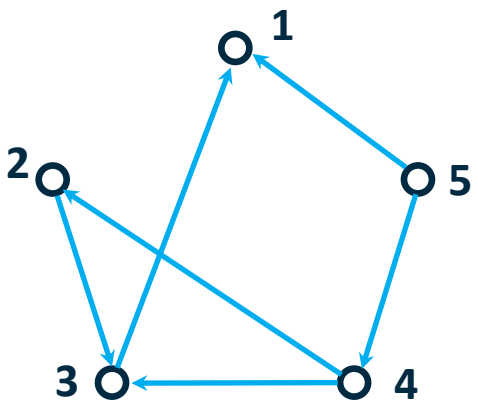
The formal definition of the graph is $G = (V, E)$.



Graph Direction

The previous graph is known as an undirected graph, i.e. you can move from node to node in both directions.

A directed graph means that you can only travel between nodes in a single direction.



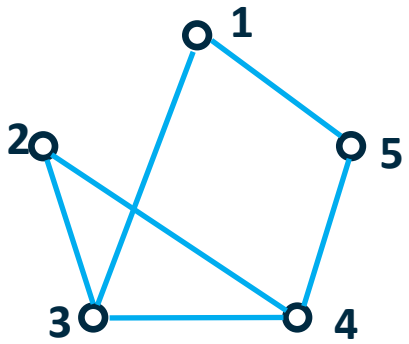
Adjacency Matrix

Graphs can be mathematically represented as an adjacency matrix, A, which is a $V \times V$ matrix with entries indicating if an edge exists between them.

$$a_{ij} = \begin{cases} 1 & if\,(i,j) \in E \\ 0 & otherwise \end{cases}$$

Adjacency Matrix

- **Example:**



$a_{12} = a_{21} = 0$ $a_{13} = a_{31} = 1$ $a_{14} = a_{41} = 0$ $a_{15} = a_{51} = 1$

$a_{23} = a_{32} = 1$ $a_{24} = a_{42} = 1$ $a_{25} = a_{52} = 1$

$a_{34} = a_{43} = 1$ $a_{35} = a_{53} = 0$

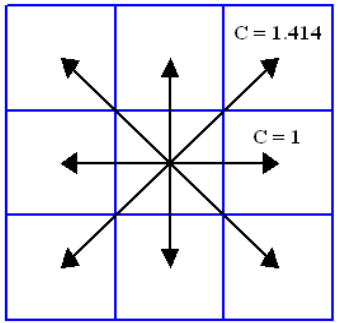
$a_{45} = a_{54} = 1$

Grid movement

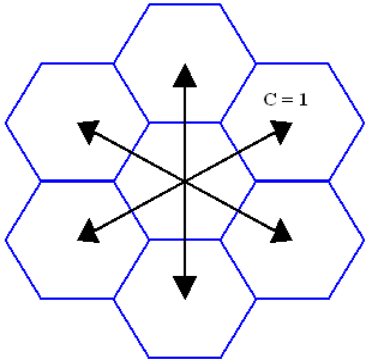
- **The cell decomposition method** approaches the path-planning problem by discretising the environment into cells; each cell can either be an obstacle or free space. Then, a search algorithm is employed to determine the shortest path though these cells to go from the start position to the goal position. This strategy tends to work well in dense environments and some of its algorithms can handle changes in the environment efficiently.

Grid movement

The environment of the robot is a continuous structure that is perceived by the robot sensors. Storing and processing this complex environment in a simple format can be quite challenging. One way to simplify this problem is by discretising the map using a grid. The grid discretises the world of the robot into fixed-cells that are adjacent to each other.



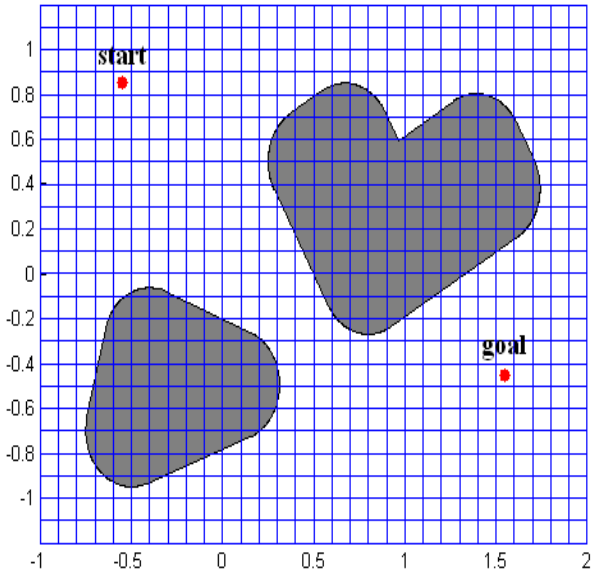
8-connected grid

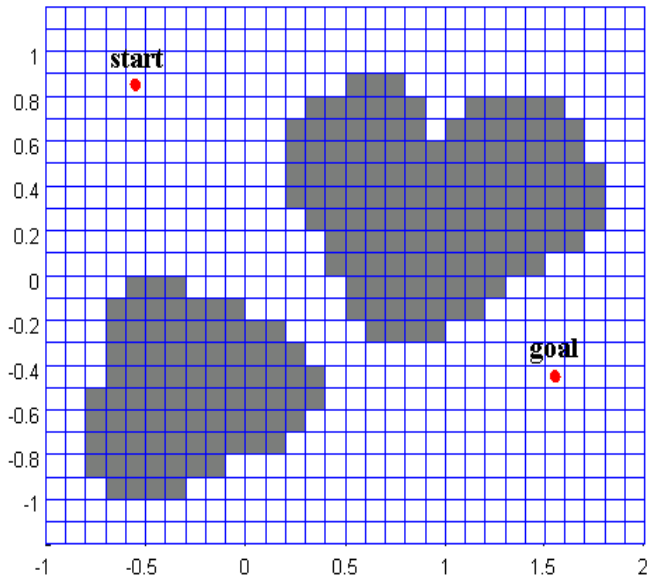


6-connected grid

Grid movement

- Grid-based discretisation results in an approximate map of the environment. If any part of the obstacle is inside a cell, then that cell is occupied; otherwise, the cell is considered as free space.





Types of Search Algorithms

Search algorithms can be broadly placed into two categories:

- Uninformed
  - E.g. Breadth-First, Depth-First, Wavefront
- Informed
  - E.g. Dijkstra, A*, D*, variants of both

Uninformed searches have no additional information about the environment.

Informed searches have additional information through the use of evaluation functions or heuristics.

Breadth-First Pseudo-Code

A basic search algorithm is called Breadth-First which begins at a start node and searches all adjacent nodes first. It then progresses onto the next ‘level’ node and searches all nodes on that level before it progresses. It terminates when it reaches its goal.

This search method provides an optimal path on the assumption that the cost of traversing each edge is the same.

Breadth-First Example

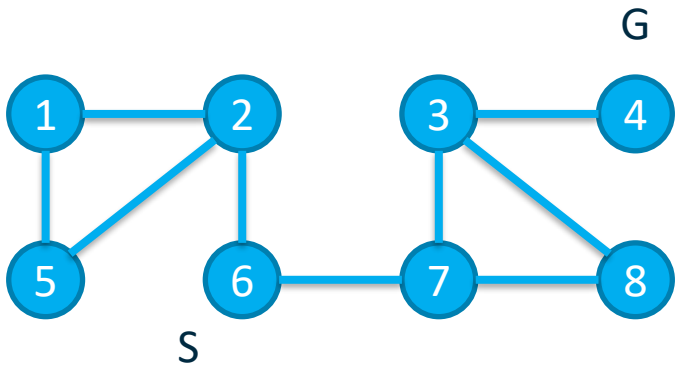
Start Node: 6

End Node: 4

Distance to Node = 3

Path to Node:

6 -> 7 -> 3 -> 4

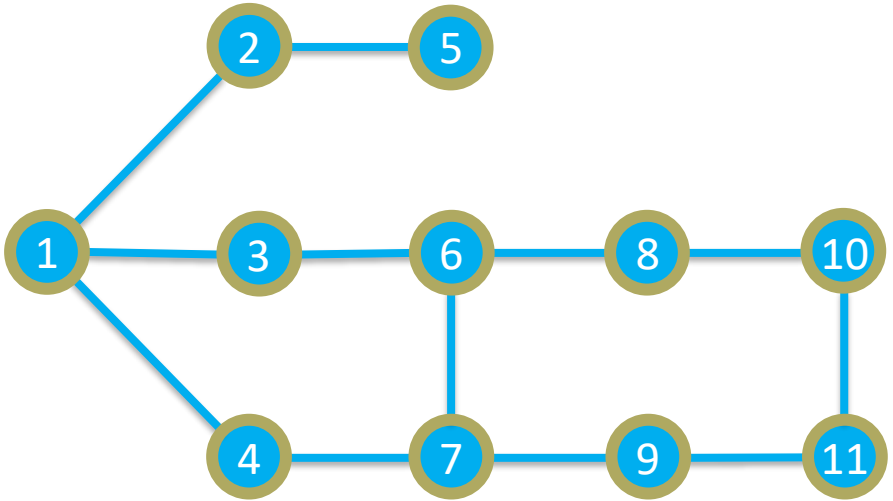


Breadth-First Exercise

What is the shortest distance to the goal and what is the route?

Start Node: 1

End Node: 11



Depth-First Search

DFS is similar to BSF, however the algorithm expands the nodes to the deepest level first.

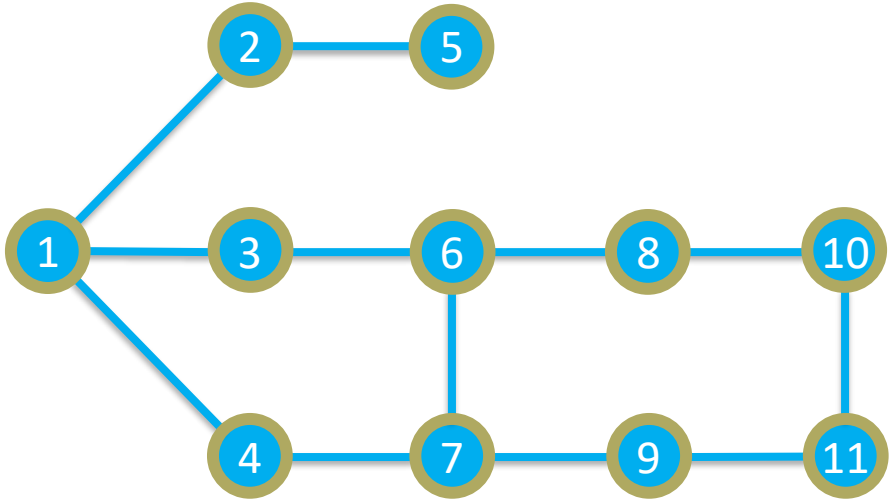
There is some redundancy in that the algorithm may have to backtrack to previous nodes.

Depth-First Exercise

What is the shortest distance to the goal and what is the route?

Start Node: 1

End Node: 11



Wavefront Expansion Algorithm

A specific implementation of the BFS algorithm for mobile robots is the Wavefront algorithm (also known as NF1 or grassfire).

This algorithm is used to find routes in fixed cell arrays and works backwards from the goal to the start point.

Wavefront Propagation Pseudo Code

Wavefront Propagation

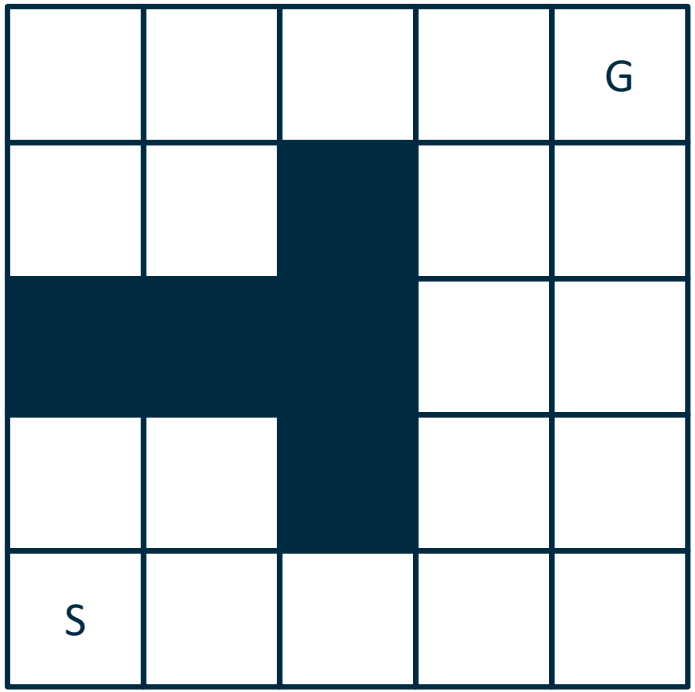
1. Start with a binary grid; ‘0’’s represent free space, ‘1’’s represent obstacles
2. Set the value of the goal cell to ‘2’
3. Set all ‘0’-valued grid cells adjacent to the ‘2’ cell with ‘3’
4. Set all ‘0’-valued grid cells adjacent to the ‘3’ cell with ‘4’
5. Continue until the wave front reaches the start cell (or has populated the whole grid)

Wavefront Path Planning Pseudo Code

Extract the path using gradient descent

1. Given the start cell with a value of ‘x’, find the neighbouring grid cell with a value ‘x-1’. Add this cell as a waypoint.
2. From this waypoint, find the neighbouring grid cell with a value ‘x-2’. Mark this cell as a waypoint.
3. Continue until you reach the cell with a value of ‘2’

Wavefront Example



0	0	0	0	2
0	0	1	0	0
1	1	1	0	0
0	0	1	0	0
0 (S)	0	0	0	0

Wavefront Example

6	5	4	3	2
7	6	1	4	3
1	1	1	5	4
11	10	1	6	5
10 (S)	9	8	7	6

6	5	4	3	2
7	6	1	4	3
1	1	1	5	4
11	10	1	6	5
10 (S)	9	8	7	6

Durham University

Dijkstra's Algorithm

Up till now, the edges of the graphs we have considered have all had the same weight. As observed in the previous example, this doesn't necessarily provide the optimal route.

Dijkstra's algorithm is similar to the BFS, however edges can take any positive cost.

This algorithm finds the costs to all vertices in the graph rather than just the goal.

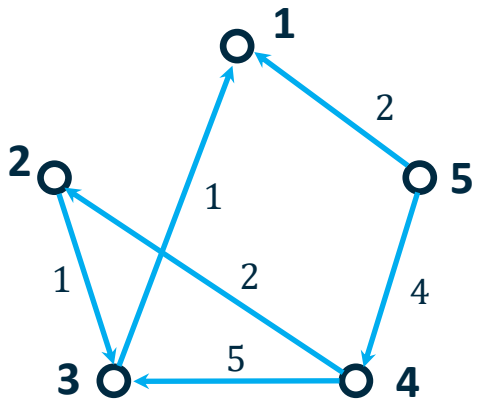
This is an informed search where the node with the lowest $f(n)$ is explored first. $g(n)$ is the distance from the start.

$$f(n) = g(n)$$

Dijkstra Pseudo Code

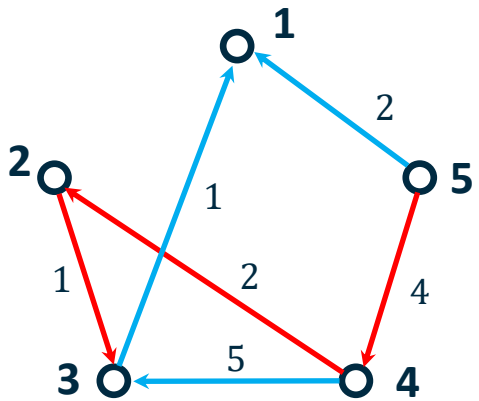
1. Initialise vectors
   a. Distance to all other nodes, $f[n]$ = Inf
   b. Predecessor vector (pred) = Nil (0)
   c. Priority vector, Q
2. For all nodes in the graph
   a. Find the one with the minimum distance
   b. For each of the neighbour nodes
      i. If the distance to the node from the start is shorter, update the path distance (if $f[u] + w(u,n) < f[n]$)
3. Find the path to the goal based on the shortest distances

Dijkstra Example



Node	1	2	3	4	5
1	0	0	0	0	0
2	0	0	1	0	0
3	1	0	0	0	0
4	0	2	5	0	0
5	2	0	0	4	0

Dijkstra Example



Start: 5, Goal: 3

Shortest Path = 7

Path = 5 -> 4 ->2 -> 3

A* Search Algorithm

One of the most popular search algorithms is called A* (A-star)

A* uses heuristics, additional information about the graph, to help find the best route.

$$f(n) = g(n) + h(n)$$

n is the node

$g(n)$ is the distance from the start node to n

$h(n)$ is the estimated distance to the goal from n

Nodes with the lowest cost are explored first.

Heuristics

For grid maps, the heuristic function can be calculation a number of ways depending on the type of movement allowed.

- von Neumann – Manhattan Distance
- Moore – Octile or Euclidean Distance

The heuristic function should be ‘admissible’, i.e. always underestimate the distance to the goal:

$$h(n) \leq h^*(n)$$

A* Pseudo Code

1. Initialise vectors
   a. Distance to all other nodes, $f[n]$ = Inf
   b. Predecessor vector (pred) = Nil (0)
   c. Priority vector, Q
2. From the starting node to the end node
   a. Find the node with the minimum $f(n)$. If there is more than one with the same value, select the node with the smallest $h(n)$
   b. For each of the neighbour nodes
      i. If $f(n)$ is smaller, update the path distance
      $$g[u] + w(n,u) + h(n) < f[n]$$
3. Find the path to the goal based on the shortest distances

A* Example

Using Octile Distances and Moore Movement.

	Goal		g = 1.4 h = 2.0 f = 3.4	g = 1.0 h = 3.0 f = 4.0
	g = 3.8 h = 1.0 f = 4.8			Start
	g = 3.4 h = 2.0 f = 5.4	g = 2.4 h = 2.4 f = 4.8	g = 1.4 h = 2.8 f = 4.2	g = 1.0 h = 3.8 f = 4.8
	g = 3.8 h = 3.0 f = 6.8	g = 2.8 h = 3.4 f = 6.2	g = 2.4 h = 3.8 f = 6.2	g = 2.8 h = 4.2 f = 7.0

Advanced Planning Algorithms

There are more advanced planning algorithms which increase performance and can operate in dynamic conditions:

- Anytime Replanning A*
- D*
- Anytime D*
- Potential Fields

Bench Test (40%)

- **Time:** Friday 6 Dec. 2024 14:00 (90 mins)
- **Format:** 10 questions
- **Place:** MCS 2094 (in Ultra)

- You are allowed:
  - to open book (notes, lecture slides, etc.)

- You can not:
  - Use phone (even for search on google etc)
  - Talk to other students
  - Use online tools, such as ChatGPT, etc.

Bench Test (40%)

- **Content:** Material from Week 1 to 9

- **Important:**
  - Do not submit the test before you finish your test
  - You will get ONE and only ONE attempt at this test

Lecture Summary

1. Obstacle avoidance
2. Search algorithms
   - Breadth-first
   - Depth-first
   - Wavefront
3. Dijkstra's Algorithm
4. A* Algorithm