Localisation and Navigation in Mobile Robots

DASE7503 Robotic Systems Integration

Contents

- Introduction
- Localisation
- Navigation
- Dijkstra’s Algorithm
- A* Algorithm
- SLAM

Introduction

- Localisation and navigation are the two most important tasks for mobile robots.
- If a robot does not know its exact position at the start of a planned trajectory, it will encounter problems in reaching its destination.
- In the past, a variety of separate algorithmic approaches was developed for localisation, navigation and mapping.
- However, today **probabilistic methods** that minimise uncertainty are applied to the whole problem complex at once, especially **simultaneous localisation and mapping (SLAM)**.
- There are navigation algorithms that operate with or without maps.
- Need to study navigation algorithms with maps, which require a nodal distance graph that has to be either **provided** or **needs to be extracted from the environment with an algorithm.**

Localisation

- For many applications, it is necessary to know a robot's position and orientation (pose) at all times.

- For example, a cleaning robot needs to make sure it covers the whole floor area without repeating lanes or getting lost, or an office delivery robot needs to be able to navigate a building floor and needs to know its pose relative to its starting point.

- This is a **nontrivial problem** in the absence of **global sensors**.

Localisation

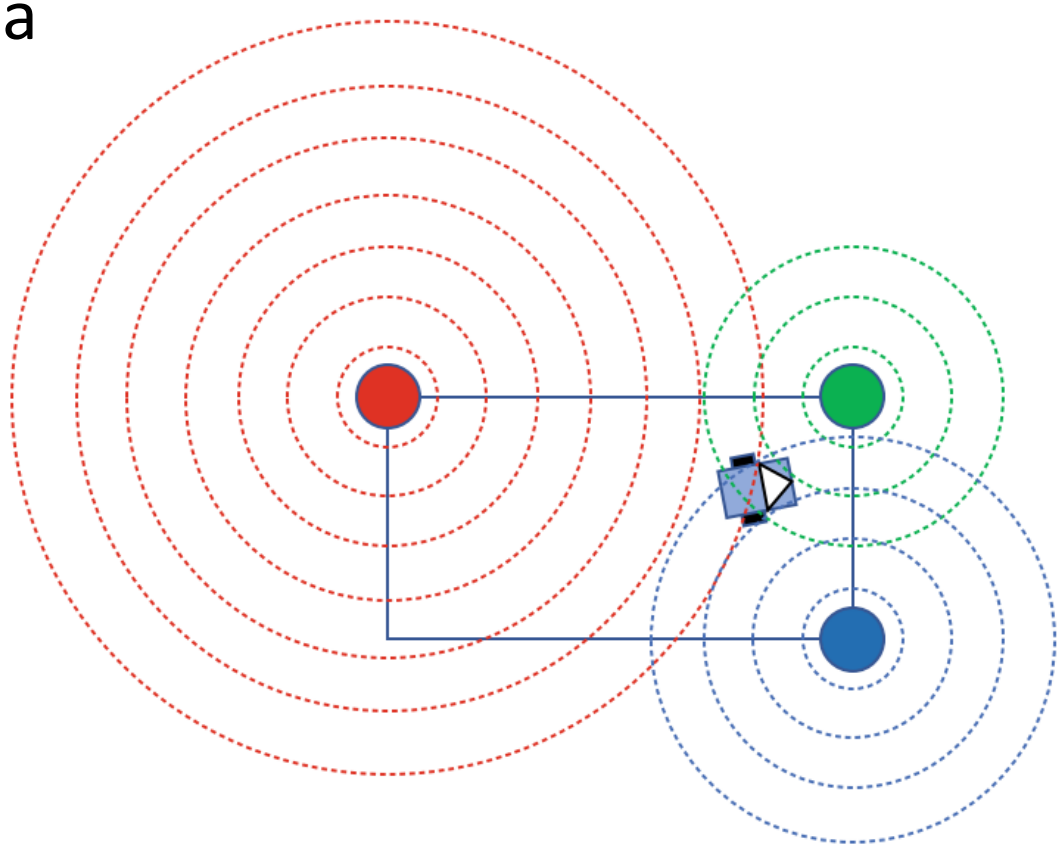
- The localisation problem can be solved by using a Global Positioning System.

- In an **outdoor** setting, this could be a Global Navigation Satellite System (GNSS) such as GPS.

- In an **indoor** setting, a global sensor network with **infrared**, **sonar**, **laser** or **radio beacons** can be employed.

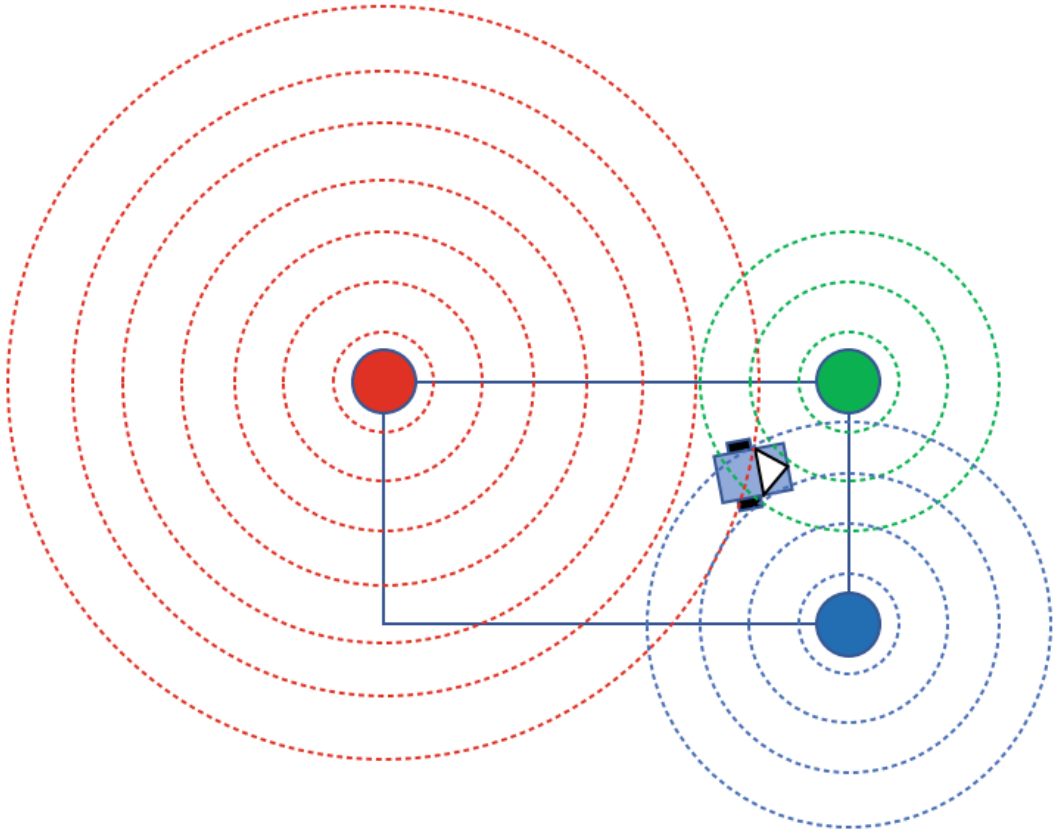
Localisation – Radio Beacons

- Assume a driving environment that has a number of synchronised radio beacons that are sending out radio or sonar signals at the **same regular time intervals**, but at different (distinguishable) frequencies (see Figure).
- By receiving signals from two or three different beacons, the robot can determine its **local position** from the time difference of the signals' arrival times.



Localisation – Radio Beacons

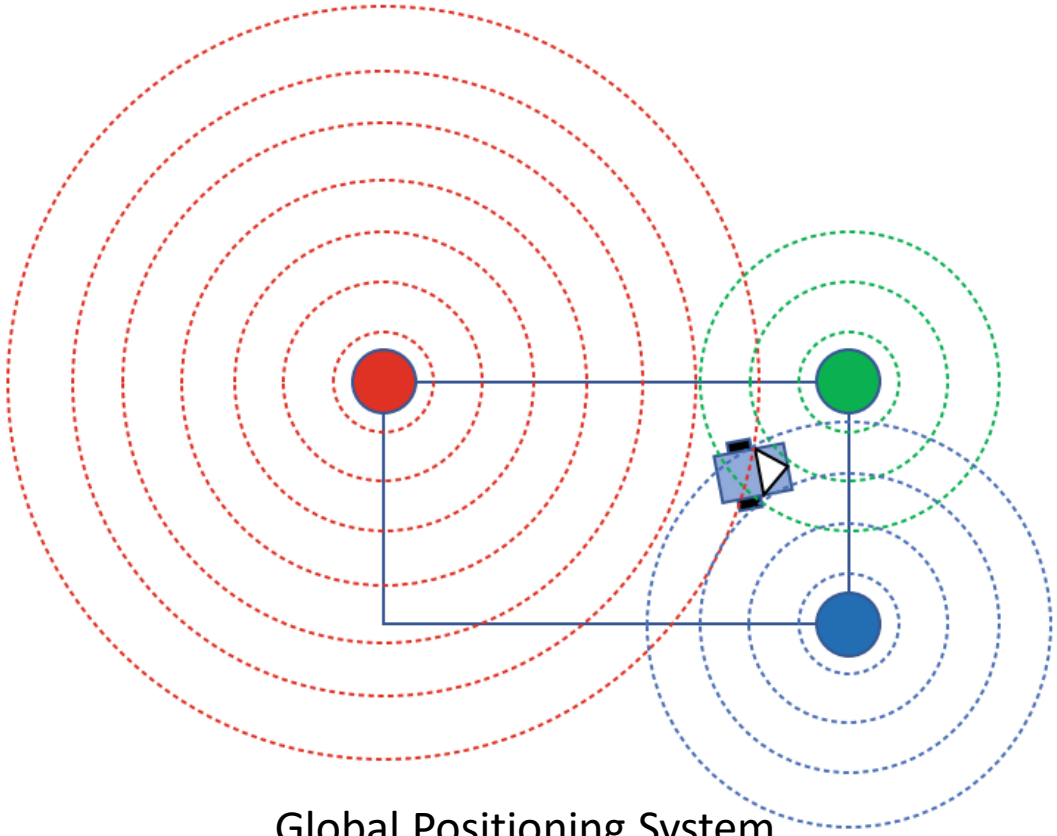
- Using two beacons can narrow down the **robot position to two possibilities**, since two circles have two intersection points.
- For example, if the two signals arrive at exactly the same time, the robot is located **in the middle** between the two transmitters.
- If, say, the left beacon's signal arrives before the right one, then the robot is closer to the left beacon by a distance proportional to the time difference.
- Using local position coherence, this may already be sufficient for global positioning. However, to be able to determine a 2D position without local sensors, three beacons are required.



Global Positioning System

Localisation – Radio Beacons

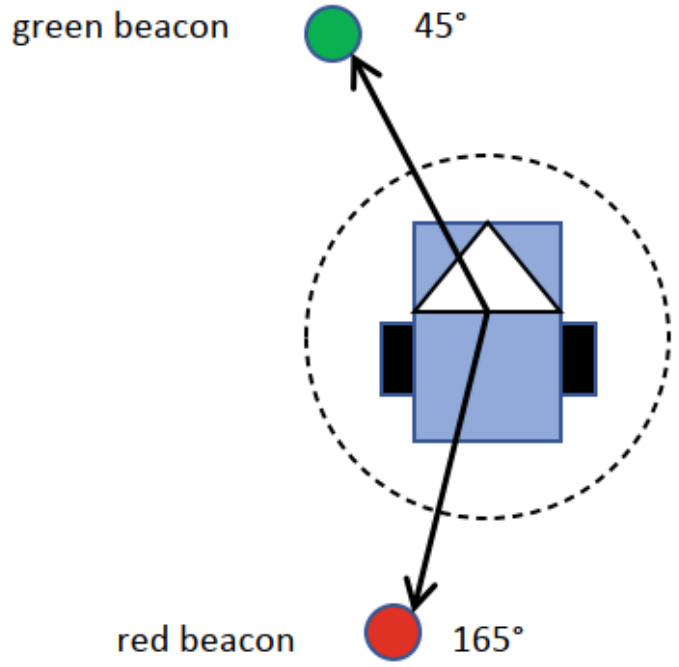
- Only the robot's position can be determined by this method, not its orientation. The orientation has to be deducted from the change in position (difference between two subsequent positions), which is exactly the method employed for satellite-based GNSS, or from an additional compass sensor.
- Using **global sensors** is in many cases not possible because of restrictions in the robot environment, or not desired because it limits the autonomy of a mobile robot (such as overhead or global vision systems for robot soccer in some competitions).
- On the other hand, in some cases, it is possible to convert a system with **global sensors** as in Figure to one with local sensors.
- For example, if the **sonar sensors** can be mounted on the robot and the beacons are converted to **reflective markers**, then there is an autonomous robot with local sensors.



Global Positioning System

Localisation – Light Beacons

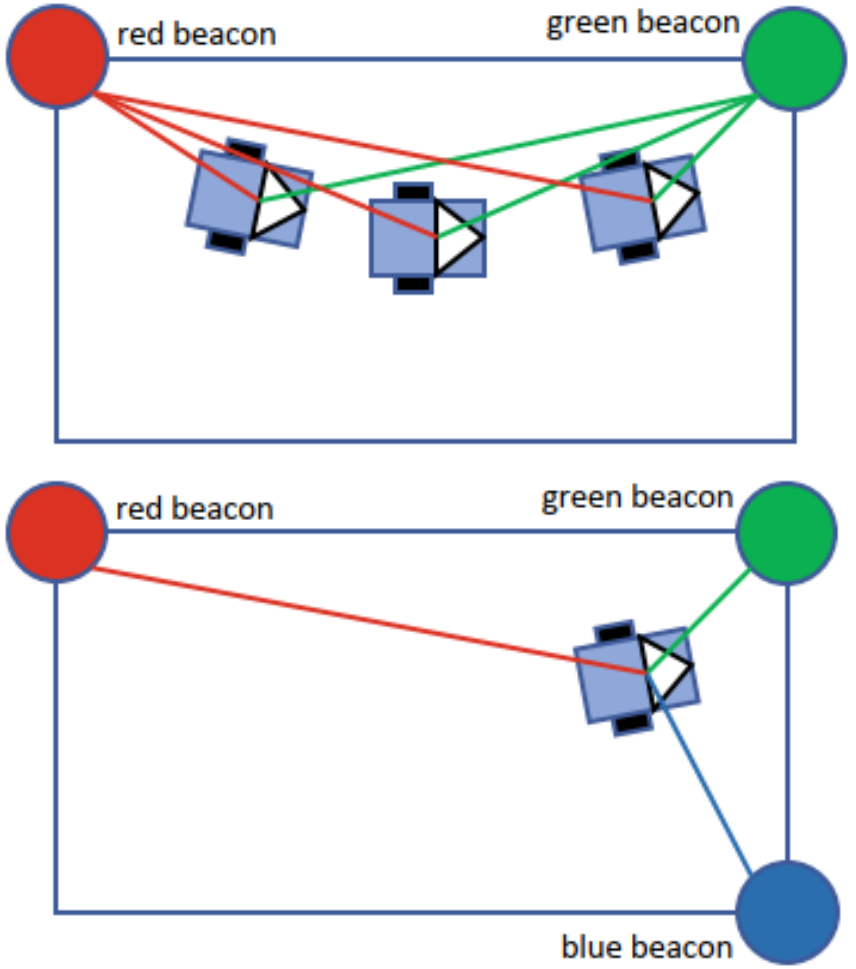
- Another idea is to use **light-emitting homing beacons** instead of radio or sonar beacons—the equivalent of a lighthouse.
- With two light beacons with **different, distinguishable colors**, the robot can determine its position at the intersection of the lines from the beacons at the measured angle.
- The advantage of this method is that the robot can determine its position and orientation.
- However, in order to do so, the robot has either to perform a 360° rotation, or it needs to possess an **omni-directional vision system** that allows it to determine the angle of each recognized light beacon.
- For example, after doing a 360° scan, as shown in Figure, the robot knows it sees a green beacon at an angle of 45° and a red beacon at an angle of 165° in its local coordinate system.



Beacon measurements on robot

Localisation – Light Beacons

- It is then needed to fit these two **vectors** into the robot's environment using the known fixed beacon positions.
- But since the robot's distance from either of the beacons is unknown, which of the many possible positions to be correct cannot be determined (see Figure, top).
- Knowing only two beacon angles is not sufficient for **localisation**.
- If the robot in addition knows its **global orientation**, for example, by using an on-board compass, or if three beacons are being used, then **correct localisation** is possible (see Figure, bottom).



Positioning using light beacons

Navigation in Robotics

- Mobile robot navigation is the process of a robot moving from one location to another using **sensors**, **path planning**, and **motion control**.
- Key components include **localisation** (determining the robot's position), **mapping** (creating a representation of the environment), **path planning** (calculating the best route), and **obstacle avoidance** (adjusting the path to prevent collisions).
- Autonomous navigation uses sensor data, such as from LiDAR, cameras, or IMUs, to perform these tasks without human intervention.

Key Components and Concepts of Mobile Robot Navigation

- **Perception and Localisation**: The robot uses sensors (e.g., LiDAR, cameras, IMUs) to perceive its surroundings and determine its position within the environment.
- **Mapping**: This involves creating a digital map of the environment. Some robots use Simultaneous Localisation and Mapping (SLAM) to build the map and locate themselves at the same time.
- **Path Planning**: Based on the map and its current location, the robot calculates the most efficient route to its destination. This can be a **global plan** or a **local adjustment** to avoid unexpected obstacles.
- **Motion Control**: This is the process of executing the planned path by controlling the robot's motors and actuators.
- **Obstacle Avoidance**: The robot uses its sensors to detect objects in real-time and adjusts its planned path to prevent collisions.
- **Types of Navigation**:
  - **Global Navigation**: Uses a prior known map to navigate.
  - **Local Navigation**: Navigates using only immediate sensor data to make decisions about motion and orientation, often for obstacle avoidance.

Dijkstra's Algorithm

- An Algorithm for computing all shortest paths from a **given starting node** in a **fully connected graph**. Time complexity for e edges and v nodes is O(e + v^2) for a naive implementation, which can be reduced to O(e + v ·log v).
- Distances between neighboring nodes are given as edge(n, m).
- Relative distances (non-negative) between all nodes.
- While some algorithms worked directly on the environment data, Dijkstra requires **a distance graph** to be constructed first.
  - Start ready set with start node.
  - In loop select node with the shortest distance in every step, then compute distances to all of its neighbors and store the shortest paths with corresponding path predecessors.
  - Add current node to ready set; loop finishes when all nodes are included.

Dijkstra's Algorithm

1. Initialisation step

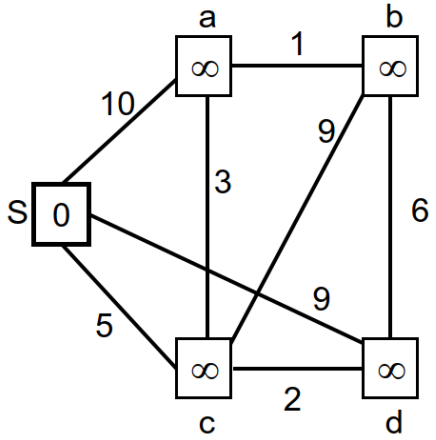
```
Set start distance to 0:                dist[s] = 0
Set all other distances to infinite:    dist[i] = ∞ (for i≠s)
Clear ready set:                        Ready = { }
```

2. Loop until all nodes are in Ready set

```
Select node n with shortest known distance that is not in Ready set
Ready = Ready + {n}
for "each neighbor node m of n":
  if (dist[n]+edge(n,m) < dist[m]) // then shorter path found
    {dist[m] = dist[n] + edge(n,m); //record new shortest distance
     pre[m] = n;                    // record predecessor node
    }
```

Dijkstra's Algorithm

- In the beginning (initialization step), **start node S** is reachable with distance 0 (by definition).
- The distances to all other nodes are **infinite** at this stage and we do not have a path predecessor recorded yet.
- Proceeding from step 0 to step 1, the node with the shortest distance from all nodes that are not yet included in the Ready set should be selected.
- Since Ready is still empty, all nodes should be looked at.
- Clearly S has the shortest distance (0), compared to infinite distances for all other nodes.



A distance graph

From s to:	S	a	b	c	d
Distance	0	∞	∞	∞	∞
Predecessor	-	-	-	-	-

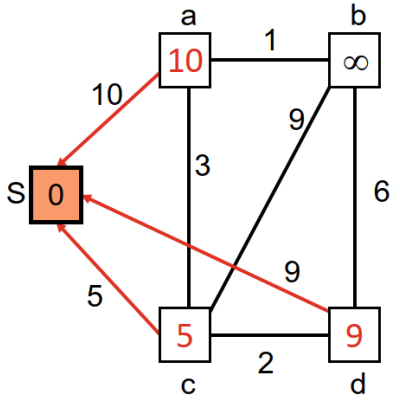
Step 0: Init list, no predecessors

Ready = {}

A table with the shortest distances found so far and the immediate path predecessors that lead to this distance

Dijkstra's Algorithm

- For step 1, S is now included into the Ready set and the distances and path predecessors (equal to S) for all its neighbors are being updated.
- Since S is neighbor to nodes a, c and d, the distances for these three nodes are being updated and their path predecessor is being set to S.
- When moving to step 2, as S is already in the Ready set, the node with the shortest path among a, b, c, and d should be selected.
- Among these, node c has the shortest path (5). The table is updated for all neighbors of c, which are S, a, b and d.
- New shorter distances are found for a, b and d, each entering c as their immediate path predecessor.

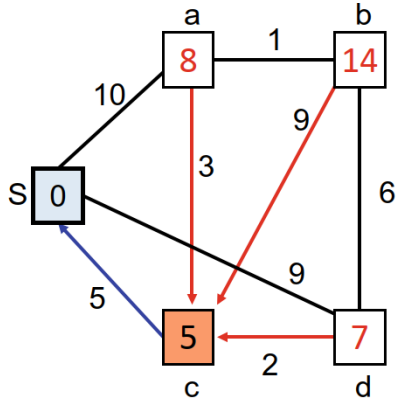


From s to:	S	a	b	c	d
Distance	0	10	∞	5	9
Predecessor	-	S	-	S	S

Step 1: Closest node is *S*, add to Ready

Update distances and pred. to all neighbors of *S*

Ready = {*S*}



From s to:	S	a	b	c	d
Distance	0	~~10~~ 8	14	5	~~9~~ 7
Predecessor	-	~~S~~ c	c	S	~~S~~ c

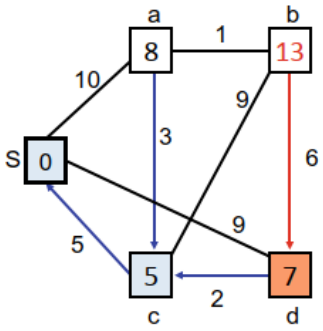
Step 2 : Next closest node is *c*, add to Ready

Update distances and pred for *a* and *d*

Ready = {*S*, *c*}

Dijkstra's Algorithm

- In the following steps 3 through 5, the algorithm's loop is repeated, until finally, all nodes are included in the Ready set and the algorithm terminates.
- The table now contains the shortest path from the start node S to each of the other nodes, as well as the path predecessor for each node, allowing us to reconstruct the shortest path.

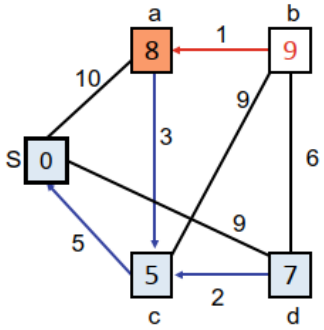


From s to:	S	a	b	c	d
Distance	0	8	14 13	5	7
Predecessor	-	c	c d	S	c

Step 3 : Next closest node is *d*, add to Ready

Update distances and pred. for *b*

Ready = {*S, c, d*}

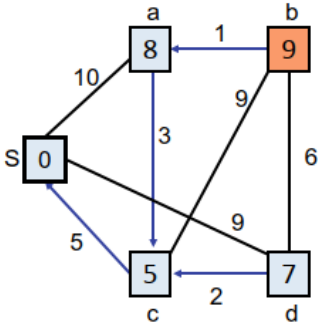


From s to:	S	a	b	c	d
Distance	0	8	13 9	5	7
Predecessor	-	c	d a	S	c

Step 4 : Next closest node is *a*, add to Ready

Update distances and pred. for *b*

Ready = {*S, a, c, d*}



From s to:	S	a	b	c	d
Distance	0	8	9	5	7
Predecessor	-	c	a	S	c

Step 5 : Closest node is *b* add to Ready

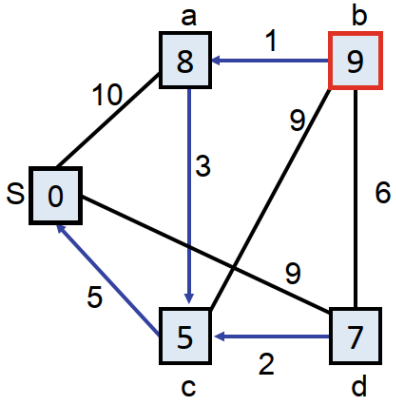
Check all neighbors of *s*

Ready = {*S, a, b, c, d*} *_complete!*

Dijkstra's Algorithm

- The Figure shows how to construct the shortest path from each node's predecessor.
- For finding the shortest path between S and b, we already know the shortest distance (9), but we have to reconstruct the shortest path backwards from b, by following the predecessors:

pre[b] = a
→pre[a] = c
→pre[c] = S
Therefore, the shortest path is: S → c → a → b



From s to:	S	a	b	c	d
Distance	0	8	9	5	7
Predecessor	-	c	a	S	c

Example: Find shortest path S → b

dist[*b*] = 9

pre[*b*] = *a*

↳ pre[*a*] = *c*

↳ pre[*c*] = *S*

Shortest path: S → c → a → b (total length is 9)

A* Algorithm

- A* or A-Star is a heuristic algorithm for computing the shortest path from one given start node to one given goal node. The average time complexity is $O(k \log_k v)$ for v nodes with branching factor k, but it can also be quadratic in the worst case.
- Distance graph with relative distance information between all nodes plus lower bound of distance to goal from each node (e.g., air-line or linear distance).
- Maintain sorted list of paths to goal, in every step expand only the currently shortest path by adding adjacent node with shortest distance, including estimate of remaining distance to goal.

A* Algorithm

- For the first step, there are three choices:
  - 1. {S, a} with min. length 10 + 1 = 11
  - 2. **{S, c} with min. length 5 + 3 = 8 (shortest path so far)**
  - 3. {S, d} with min. length 9 + 5 = 14
- Using a best-first algorithm, we explore the shortest estimated path first: We remove {S, c} and replace it with all possible extensions of this path with one additional step. All other paths remain unchanged. So the path list for our next step will be:

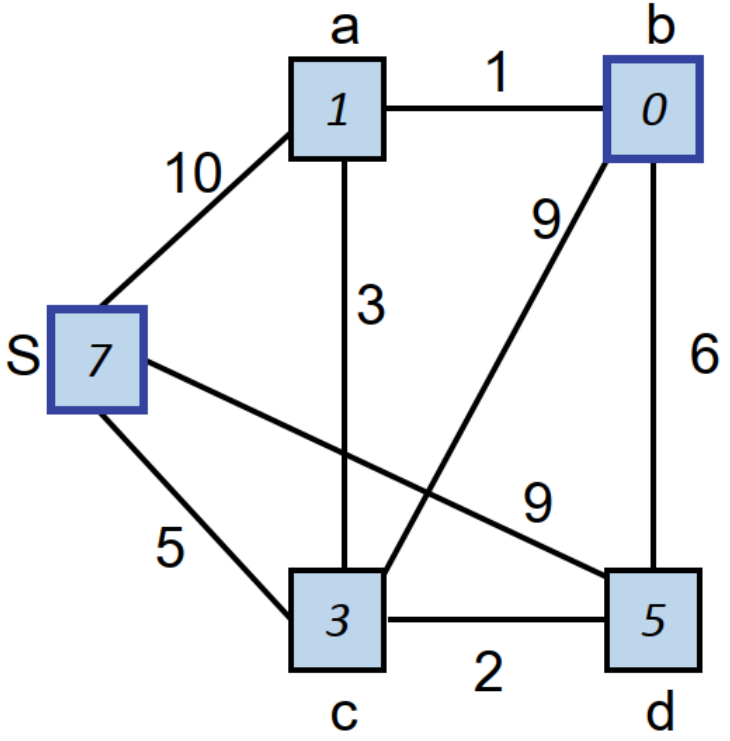
  1. {S, a} with min. length 10 + 1 = 11

  2:1. {S, c, a} with min. length 5 + 3 + 1 = 9 (shortest path so far)

  2:2. {S, c, b} with min. length 5 + 9 + 0 = 14

  2:3. {S, c, d} with min. length 5 + 2 + 5 = 12

  3. {S, d} with min. length 9 + 5 = 14

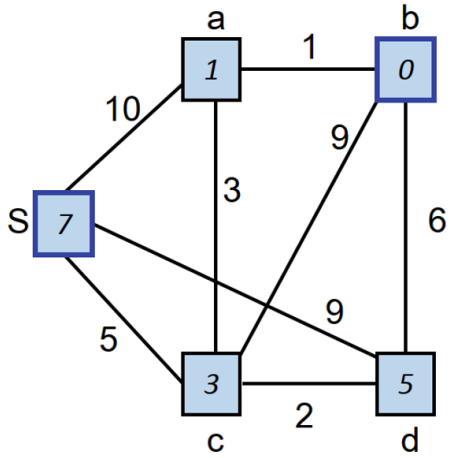


Node values are now lower bound distances to goal *b* (e.g. linear distances)

Edge values are distances between neighboring nodes

A* Algorithm

- As it turns out, the currently shortest partial path is {S, c, a}, which will now be replaced by its expansions with one step further.
- As it turns out, only one such extension exists: {S, c, a, b}. As before, all other paths remain unchanged:
  1. {S, a} with min. length 10 + 1 = 11
  2:1:1. {S, c, a, b} with min. length 5 + 3 + 1 + 0 = 9 (shortest path so far)
  2:2. {S, c, b} with min. length 5 + 9 + 0 = 14
  2:3. {S, c, d} with min. length 5 + 2 + 5 = 12
  3. {S, d} with min. length 9 + 5 = 14



A* Algorithm

- The shortest path in this current iteration is now {S, c, a, b}, and this path already reaches the goal node b.
- Since we now have a shortest path that reaches the goal, any other possible path to goal that we have not yet explored must be longer, as all calculated partial distances are minimal distances, so if anything, the second shortest partial path {S, c, d} can only be 12 or higher.
- We therefore have found the shortest path {S, c, a, b} and our algorithm terminates.

SLAM

- **SLAM (Simultaneous Localisation and Mapping)** is a fundamental problem in robotics and autonomous systems where a robot must simultaneously build a map of an unknown environment while keeping track of its own location within that map.
- It is crucial for robots operating in GPS-denied or dynamic environments, such as indoor navigation, drones, self-driving cars, and augmented reality.

Key Components of SLAM

- **Localisation** – Determining the robot's position and orientation (pose) relative to its surroundings.
- **Mapping** – Constructing or updating a representation (map) of the environment using sensor data.
- **Sensor Fusion** – Combining data from multiple sensors (e.g., LiDAR, cameras, IMUs, sonar) to improve accuracy.

How SLAM Works

- The robot moves through an unknown space, collecting sensor data.
- It detects landmarks (distinct features like walls, corners, or objects) to estimate its position.
- The algorithm refines both the robot's estimated pose and the map over time using **probabilistic techniques** (e.g., **Kalman filters**, **particle filters**, or **graph-based optimization**).

Types of SLAM

- **Visual SLAM (V-SLAM)** – Uses cameras (monocular, stereo, or RGB-D) to track features.
- **LiDAR SLAM** – Relies on laser scans for high-resolution mapping.
- **Legged or Wheeled SLAM** – Adapts to different robot locomotion methods.
- **Semantic SLAM** – Incorporates object recognition to improve mapping with labeled objects.

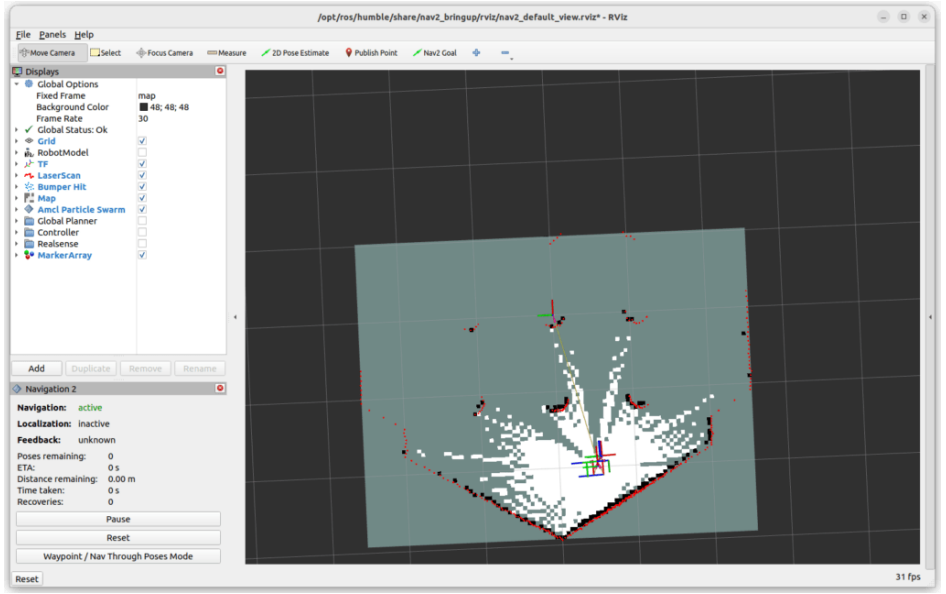
SLAM in ROS2

- SLAM in ROS2 is a crucial capability for autonomous robots to build a map of an unknown environment while simultaneously tracking their position within it.

- ROS2 provides several tools and packages to implement SLAM, leveraging modern algorithms

Key SLAM Package in ROS2

- **slam_toolbox**
  - The most popular SLAM package for ROS 2, offering real-time and offline SLAM capabilities.
  - Supports both 2D LiDAR and simulated data.
  - Features:
    - Incremental mapping with loop closure.
    - Lifelong mapping (updating existing maps).
    - Supports ROS 2 Jazzy, Foxy, Humble, and Rolling.
  - GitHub**: https://github.com/SteveMacenski/slam_toolbox
- **nav2 (Navigation2)**
  - The official ROS 2 navigation stack, which includes SLAM capabilities via slam_toolbox integration.
  - Works with rviz2 for visualization.
  - Supports multiple SLAM algorithms and sensor inputs (LiDAR, depth cameras).



Key SLAM Package in ROS2

- **cartographer_ros**
  - Google's Cartographer adapted for ROS 2.
  - Works well for both 2D and 3D SLAM.
  - Requires more computational resources but provides high-quality maps.
  - GitHub**: https://github.com/cartographer-project/cartographer_ros
- **rtabmap_ros**
  - Supports RGB-D, stereo, and LiDAR-based SLAM.
  - Good for 3D mapping and localisation.
  - GitHub**: https://github.com/introlab/rtabmap_ros

Basic SLAM Workflow in ROS2

- **Sensor Setup**
  - Ensure LiDAR (e.g., rplidar_ros2) or depth camera (e.g., realsense-ros) is publishing sensor data (/scan or /depth topics).
- **Launch SLAM Node**
  - Example with slam_toolbox:

    ros2 launch slam_toolbox online_async_launch.py
  - For Cartographer

    ros2 launch cartographer_ros cartographer.launch.py
- **Teleop or Autonomous Control**
  - Move the robot manually (using teleop_twist_keyboard) or autonomously to explore the environment.
- **Visualization (rviz2)**
  - View the map and robot pose in rviz2

    rviz2 -d $(ros2 pkg prefix slam_toolbox)/share/slam_toolbox/online_slam.rviz
- **Save the Map**
  - After mapping, save the generated map

    ros2 run nav2_map_server map_saver_cli -f my_map

Key Topics in SLAM

- /map (Occupancy grid map)
- /scan (LiDAR data)
- /tf (Transform tree for robot pose)
- /odom (Odometry data)

Challenges in SLAM

- **Computational Complexity** – Real-time processing is demanding.
- **Sensor Noise & Drift** – Errors accumulate over time.
- **Dynamic Environments** – Moving objects can corrupt the SLAM solution.
- **Loop Closure** – Recognising previously visited locations to correct drift.