# Introduction

Assignment 1 sets out to achieve two main objectives. Firstly, to simulate the localisation and navigation of an NXT Mindstorms Robot placed in a random position within a given virtual arena. It then has to be able to successfully traverse this virtual arena to a specified endpoint whilst avoiding obstacles. Secondly, the simulated localisation and navigation algorithms have to be implemented on the actual NXT Mindstorms Robot, so that it is able to estimate its random starting position within the provided arena. It then has to be able to traverse to a predefined endpoint within a given level of accuracy. For the actual implementation, success criteria of different accuracy and time ranges have to be achieved as means of measuring the success of the implemented algorithm(s).

Hence, this report presents a solution which attempts to achieve both objectives set out by the coursework in 5 sections. Section 1 describes the Navigation and Path Planning algorithm implemented in 2 parts, namely the theory used to implement the algorithm, followed by a description of how the proposed algorithm is implemented. Section 2 discusses the implementation of a Particle Filter algorithm to perform the localisation of the robot.

Section 3 discusses the design and structure of the NXT Mindstorms Robot used to achieve both objectives, as well as the details of how calibration was conducted and incorporated into the solution. Section 4 details the optimisation techniques employed to improve the overall efficiency and performance of both the localisation and navigation algorithms in the simulation.

Finally, Section 5 provides an analysis and performance review of the actual robot, as well as highlighting any obstacles that were faced and overcome when implementing both algorithms on the actual system.

# Path Planning and Navigation

The Algorithm

The aim of the path planning and navigation system is to implement a ‘shortest path’ algorithm with obstacle avoidance capabilities. A viable solution to this problem is to identify a shortest path by first computing the visibility graph so as to assign a Euclidean distance to each edge within the map, and then to determine the shortest path using Dijkstra’s algorithm [1]. In such a solution, the robot is represented as a single point within an inflated, bounded area (boundaries of a specified size surrounding the walls of the map). This bounded area ensures the robot does not collide with the walls of the map due to its dimensions.

The theoretical elements of the proposed solution are discussed below, followed by a description of the actual algorithm implemented, in the subsequent Implementation section, to achieve the aim.

Visibility Graphs

The idea behind visibility graphs, is that they form the ‘road map’ from which the shortest path can be determined. Since the position of the goal relative to the starting point is already known, the visibility graphs can be formed by connecting all the visible vertices of the inflated boundaries (since there are no obstacles within the map) to identify all paths along these. These vertices are used to draw all possible edges, a connection between two vertices which do not pass through a boundary. This set of all possible connective edges makes up a graph. Finally, Dijkstra’s algorithm is applied to this constructed graph to compute a shortest path between the start position and the target position [1].

Dijkstra’s Algorithm

It computes the length of the optimal edge from a set of all possible edges between two vertices in the visibility graph [2]. The algorithm executes 6 important, but simple steps:

1. The initial node is set to the starting node and all remaining nodes are set to ‘unvisited’ which then form an ‘unvisited set’.
2. The initial node is assigned a tentative distance of zero, whilst all other nodes are assigned an infinite tentative value.
3. All unvisited nodes neighbouring the initial node are considered and their tentative distances are calculated. These distances are then compared to its previously assigned Euclidean distance value. If the current tentative distance is larger than the previous distance value, then this smaller value of the two is assigned to the node.
4. Once all neighbouring nodes have been considered, the initial node is set to visited and removed from the ‘unvisited set’, so as not to be reconsidered.
5. If the target node has been removed from the set or marked as visited, or else if the smallest tentative distance of the nodes is infinity, then the shortest path has been identified.
6. If these conditions are not met, then the node with the smallest tentative distance is set as the next initial node whereby the algorithm returns to step 2.

To summarise, the proposed solution uses visibility graphs to convert the navigation problem into a graph search from which Dijkstra’s algorithm can determine the shortest path. This path is found by attempting to improve on an initial tentative distance assigned to nodes, through the above 6 step process.

Implementation

The path planning and navigation solution comprises of a main script which calls various function scripts to execute the algorithm consisting of visibility graphs and Dijkstra’s algorithm. It should be noted that the solution was built from a BSD licensed skeleton algorithm [3] which provides the basic functions required to perform Dijkstra’s algorithm. The files that were provided as skeleton code are: ‘pathfinder.m’, ‘line\_of\_sight.m’ and ‘inflated\_boundaries.m’.

To separate the simulation and actual robot implementation libraries, a second library, ‘Bot.m’ is created. The particle filter returns a BotSim / Bot object that holds both the estimated location and angle of the real robot. After that, the angle and location can be retrieved using ‘bot.getBotAng()’ and ‘bot.getBotPos()’ respectively. These values get updated during navigation.

The main executable script, ‘robot\_navigation.n’ calls the ‘localise.m’ script to perform the localisation algorithm (particle filtering). This in turn calls 2 of the navigation function scripts, ‘boundary\_inflation.m’ and ‘get\_angle\_dist.m’ respectively. The original map boundaries and a specified boundary size, denoted by *robot\_size* is passed to ‘boundary\_inflation’. It should also be noted that ‘robot\_size’ (boundary size) is set differently for the simulation and the actual robot. In the simulation, the size of the boundaries is set according to the size of the robot, however this had to be reduced for maps 2 and 3 where large boundaries meant that most given starting positions were outside the boundaries. For the actual robot, the boundary size was set to reproduce the actual floor space of the arena, which is much smaller when compared to the virtual map. This is due to the robot being represented as a single point and not a large structure as a whole. Finally, ‘boundary\_inflation’ then, using only shifted midpoints of boundary lines which are directly visible to the original map wall midpoints, computes and returns the coordinates of the inflated boundaries.

Within ‘boundary\_inflation.m’, ‘line\_of\_sight.m’ is called to determine whether or not a given target node is visible from a given observer position. It carefully ensures that the path between the current observer node (which is checked to ensure that its coordinates coincide with a wall) and the target node does not pass beyond the boundaries, hence inferring that the robot has a direct line of sight to the target node.

Subsequently, the ‘get\_angle\_dist.m’ function script is called. It is passed both the start and endpoints, as well as the boundary size and inflated boundary coordinates. Using these values, it is able to calculate both the angle and length of a path between 2 waypoints. However, before this is done, the start points and endpoints, as well as the boundary coordinates are passed to ‘pathfinder.m’. This performs Dijkstra’s algorithm using the visibility graph representation of all the nodes on the boundary map produced by the ‘line\_of\_sight’ function. The result is an array of waypoint coordinates that make up the shortest path from the given start point to the specified endpoint.

These waypoints are then returned to ‘get\_angle\_dist.m’ which then uses each consecutive pair of waypoint coordinates to compute the angle and the Euclidean distance of the path between them. The returned ‘angle, distance’ parameters are then used to turn and translate the robot from its estimated position to the target. Nonetheless, due to the constant changing of the axes of rotation of the robot, the correct angle of rotation is computed by deducting the estimated angle of the robot – returned by the particle filter – from the angle returned by ‘get\_angle\_dist.m’.

Rotation is done at full speed to reduce the friction between the castor wheel and arena floor, whilst translations occur at half the speed set for the ‘ultraScan()’ and ‘move()’ functions. These movement instructions are located within ‘localise.m’ and are thresholded to ensure that the simulated robot terminates within at least 0.02 cm of the specified end target.

# References

|  |  |
| --- | --- |
| [1] | M. v. K. M. O. O. S. Mark de Berg, “Chapter 15: Visibility Graphs,” in *Computational Geometry*, 2 ed., Springer, pp. 307 - 317. |
| [2] | T. J. M. Philip L. Frana, “An interview with Edsger W. Dijkstra,” *Communications of the ACM,* vol. 53, no. 8, pp. 41 - 47, August 2010. |
| [3] | M. F. A. Razak, “Pathfinder v2,” MathWorks. |