



# Robótica Móvel, Assignment 2

Diogo Monteiro, 97606



# Main Loop

The agent's main loop is as follows:

While map is not fully explored:

    Update map using sensors

    If there is a movement plan:

        Move according to plan

        Additional map updates during movement

    Else:

        Select next destination to explore

        Create movement plan to destination

Calculate final plan through all beacons

Create movement plan to starting position

Move to starting position

(Logic related to simulator was omitted, in truth there is another, external, while loop, however it is not relevant to explain the mapping and planning logic)



# Movement

Movement is made on a grid basis. Given a list of positions, the agent moves to each consecutive position.

To move, the agent calculates the angle to the current destination and based on certain thresholds, either rotates in place to face the destination, moves forward while turning slightly, or moves directly forward. In a way, it uses a sort of bang-bang controller.

When moving, updates its pose estimate.

Ideally, the robot should always move in the cardinal directions to navigate the maze.



# Pose Correction

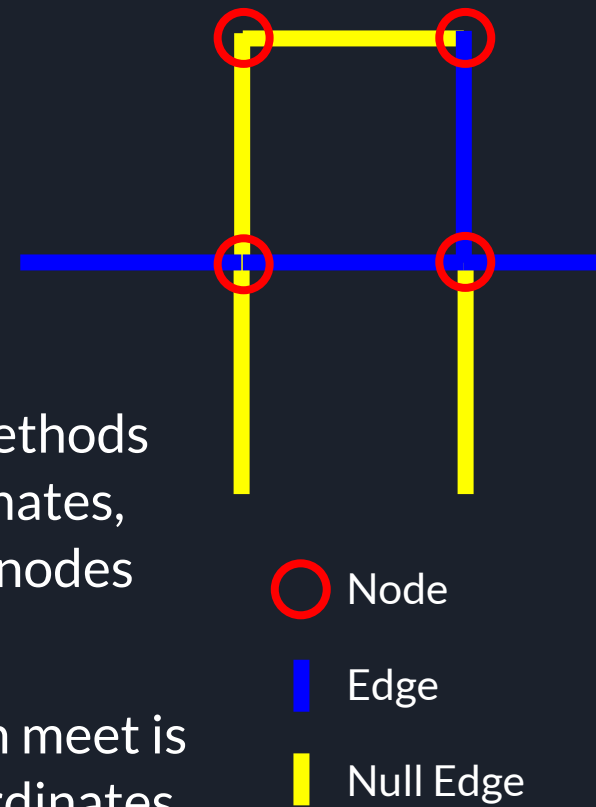
The predicted orientation is completely replaced by the compass sensor measure, which despite having noise, does not accumulate error and is accurate enough.

There was an attempt to correct the position using the line sensor measures and knowledge of the world, but it did not perform well and as such is not applied...

# Map Representation

To represent the internal map, a graph-like class was developed, called a *nodemap*. This class implements methods for setting and getting nodes given their world coordinates, creating paths between nodes, a path finder between nodes using  $A^*$ , among others.

In the robot's environment, each place where lines can meet is considered a node. These places have even world coordinates. The lines between these place are represented by edges between nodes.





# Map Representation - Nodes

Each node in the nodemap stores the following:

- x and y world coordinates
- Edges: a list of references to other nodes it's connected to
- Null Edges: a set of the directions where it has no path ("up", "down", "left", "right")
- If it's been visited before

Additionally they have the "reachable" property, which is true when there's at least one Edge, and the "explored" property, which is True when it has been visited before and the summed lengths of the Edges list and the Null Edges set are 4, otherwise false.



# Map Rep. - Adding an Edge

The `nodemap` class implements a method which given two sets of coordinates, adds an edge to the corresponding nodes. If a node doesn't exist, one is created on the spot.

This same method can also add null edges when specified.

For example, adding an edge between  $(2, 0)$  and  $(4, 0)$  would add a reference to `node(4,0)` in `node(2,0)`'s Edge list and vice-versa.

Adding a null edge instead would put "right" in `node(2,0)`'s null edge set and "left" in `node(4,0)`'s.



# Map Representation - Reasoning

Edges represent the presence of a line connecting two nodes, meaning we can move from one to the other.

Null edges mean we cannot move in that direction from that node.

Null edges are required to define when a node has been fully explored. For a node to be fully explored, each cardinal direction from it must be either an edge or a null edge. Additionally, it must have been visited as to not miss any beacons.



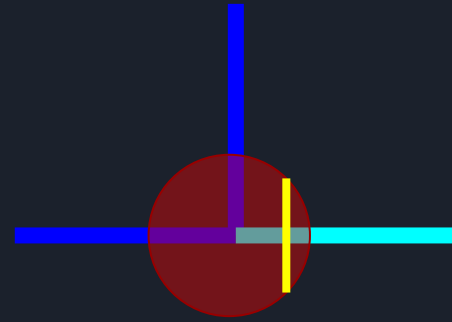


# Mapping - Updating

When the robot is right on top of a node, it checks its ground sensor for beacons to register.

If it's also facing one of the cardinal directions, it checks its line sensor's central measure to update the nodemap. If a line is detected, an edge is added between the current node and the one right after in the robot's current facing, otherwise a null edge is added between those same nodes.

If the robot's orientation is not within a threshold for facing "up", "down", "left" or "right", or if the robot's position is too far from the closest predicted node, then the update is skipped.



Robot represented in red, with line sensor in yellow. Detect presence or absence of highlighted line.



# Mapping - Registering Side-paths

The update step described before is responsible for setting edges in front of the robot. To find edges to the sides, another strategy is employed.

The agent stores something called a “node neighborhood”, a dictionary structure (or a None value).



# Mapping - Registering Side-paths

A “node neighborhood” refers only to one node at a time, and stores the following:

- Position of the node it's referring to
- Facing: either “up”, “down”, “left” or “right”
- Boolean value for relative left (default False)
- Boolean value for relative right (default False)
- Number of measures
- Position of robot on neighborhood creation



# Mapping - Registering Side-paths

When the agent is performing a movement plan, it updates the node neighborhood as it moves, when the central line sensor is within a certain range of a node.

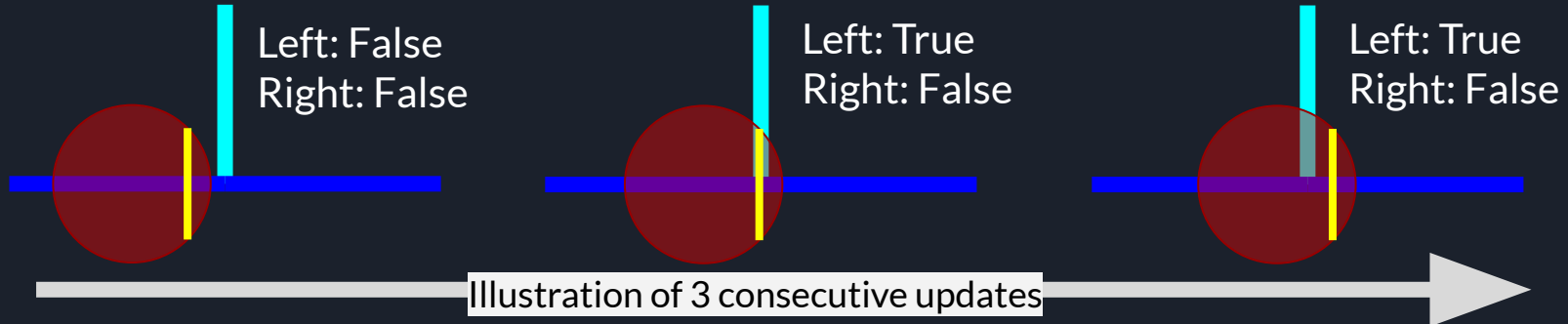
If the neighborhood was set to None, a new one is created for the current closest node, setting the “facing” to match the robots’ current orientation.

# Mapping - Registering Side-paths

## Neighborhood update:

Skip update (ignore next steps) if robot not currently facing cardinal direction or node is fully explored

- Increment number of measures
- If line sensor's left end detects line: set "left" boolean to True
- If line sensor's right end detects line: set "right" boolean to True





# Mapping - Registering Side-paths

Committing neighborhood to edges:

When the robot finishes moving towards a node, it attempts to create edges on that node according to the neighborhood.

If the “left” boolean is True, then an edge is created to the relative left of the neighborhood, according to the “facing” value. For example, given the neighborhood for node(6,-2), if the “facing” is “right”, the edge is created between nodes (6,-2) and (6, 0).

If the “left” boolean is False, the same logic from above applies but creating a null edge between the nodes instead.

The same logic applies to the “right” boolean.



# Mapping - Registering Side-paths

## Committing neighborhood to edges:

The process described only occurs if there are enough readings, and if the robot moved a minimum distance. This is an attempt to minimize the interference of noise and ensure that the line sensor properly passes through the area where the line could be located.

Regardless of whether the neighborhood is successfully committed or not, it is then set to None.



# Mapping - Registering Side-paths

## Reasoning for this approach:

The intent behind this process is to allow the agent to detect perpendicular paths as it's moving to prevent the need to look around every time it reaches an unexplored node, therefore saving time.

The reason for multiple readings instead of a “one-and-done” approach is, as stated before, to minimize noise interference and ensure that the line sensor does not miss the line when it's there.





# Mapping

To sum up the mapping process:

- Every main loop iteration, the agent scans for beacons and for paths in front of it, whenever possible
- During movement, the agent defines and updates a neighborhood for the closest node to the line sensor
- When arriving at a node, sideways edges are created according to the neighborhood, and the neighborhood is reset.

Additionally, the agent writes to the map file after every update.




# Exploring

While there are still unexplored nodes, the agent will pick one to move to, plan a path to it, and move there. While the agent is moving, it updates the nodemap when possible using the methods described before.

After reaching an unexplored destination, the robot will spin in place to register all the paths to that node, if it failed to do so during its movement.

To select the next destination, the nodemap class provides a “closest\_unexplored\_node” method, which returns the corresponding set of coordinates. This selection is made using a heuristic to select the closest node, consisting of the manhattan distance, plus 1, minus 1 if the node has a direct edge to the current one. This gives priority to nodes which can be reached with a single movement.



# Finalizing - Final Path Planning and Moving to Start

After fully exploring the map, the agent then plans the final path that starts at position 0, goes through all the beacons, and ends at position 0, in the shortest path possible.

To do so, first we calculate the shortest path between every pair of beacons. Then, we generate all permutations of the beacons, excluding 0, and add 0 to the start and end of each one. For each permutation, we concatenate the path between each successive pair of beacons in the sequence to form a plan (taking care to not have duplicate positions).

The shortest path found after iterating through all the permutations is then written to a file.

Finally, the robot moves to the starting position and sends the finished signal to the simulator.



# Results and Shortcomings

As stated near the beginning, there was no position estimation correction applied, and as such the robot is hardly resistant to motor noise. As such, it is entirely up to luck if the position estimate stays correct when running the simulation with motor noise.

This of course affects the mapping capabilities, as they depend on the position estimate as well. Running the simulator without motor noise, the agent is usually capable of correctly mapping the maze, though there are still times where an extra edge is set where there shouldn't be one. If this extra edge leads the robot to collide with the wall, then its position estimate will be entirely offset, leading the rest of the mapping process to suffer.



# Results and Shortcomings

It seems clear then that the edge detection methods used are not fool-proof, and could use more work.

One limitation of the implementation is that once an edge is set, it cannot be changed. On one hand this keeps the agent from constantly changing the map and from introducing mistakes, but on the other hand it also prevents it from correcting mistakes. Perhaps a probabilistic approach would be suited for this problem.