e-Yantra Robotics Competition - 2018

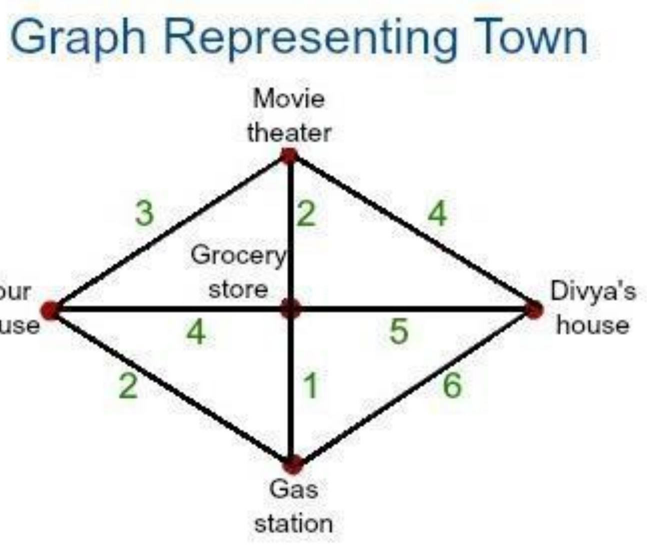
NS Task 1 Report <1776 >

Author Name(s)	**Sathyam Rajpal, Vaibhav Tiwari**
Team ID	**1776**
Date	**28/Nov/2018**

URL of the video uploaded: https://youtu.be/eILWbPiVfzQ

Q1. Describe the path planning algorithm you have chosen.

Ans 1. Suppose we're trying to find the shortest path from your house to Divya's house. We know the distances between various locations throughout town. If we let various locations be vertices and the routes between them be edges, we can create a weighted graph representing the situation. Quick definition: a weighted graph is a collection of vertices and edges with edges having a numerical value (or weight) associated with them.



This graph is a great example of a weighted graph using the terms that we just laid out. There are quite a few different routes we could take, but we want to know which one is the shortest. Thankfully, we've got a nice algorithm that can help us.

Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm we can use to find shortest distances or minimum costs depending on what is represented in a graph. You're basically working backwards from the end to the beginning, finding the shortest leg each time. The steps to this algorithm are as follows:

Step 1: Start at the ending vertex by marking it with a distance of 0, because it's 0 units from the end. Call this vertex your current vertex, and put a circle around it indicating as such.

Step 2: #Identify all of the vertices that are connected to the current vertex with an edge. Calculate their distance to the end by adding the weight of the edge to the mark on the current vertex. Mark each of the vertices with their corresponding distance, but only change a vertex's mark if it's less than a previous mark. Each time you mark the starting vertex with a mark, keep track of the path that resulted in that mark.

Step 3: Label the current vertex as visited by putting an X over it. Once a vertex is visited, we won't look at it again.

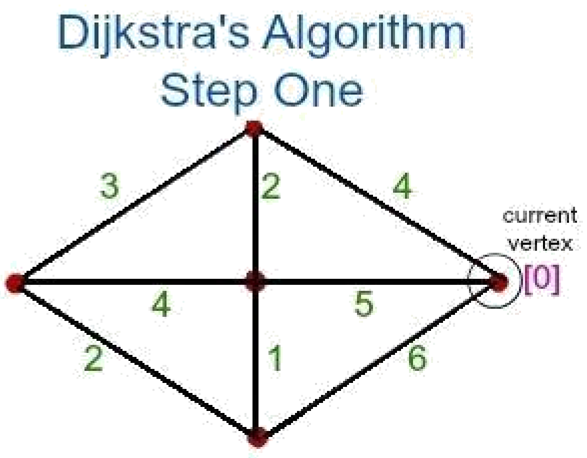
Step 4: Of the vertices you just marked, find the one with the smallest mark, and make it your current vertex. Now, you can start again from step 2.

Step 5: Once you've labeled the beginning vertex as visited - stop. The distance of the shortest path is the mark of the starting vertex, and the shortest path is the path that resulted in that mark.

Let's now consider finding the shortest path from your house to Divya's house to illustrate this algorithm.

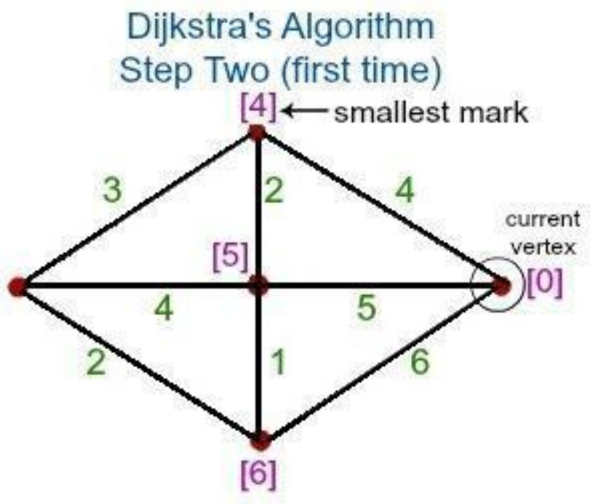
Application

First, we start at the ending vertex (Divya's house). Mark it with a zero and call this vertex the current vertex, putting a circle around it, as you can see on the graph:



The next step is to mark any vertices that are connected to Divya's house by an edge with the distance from the end. Let's quickly calculate these distances while we look at a visual representation of what's going on using our previous graph:

- Movie theater = 0 + 4 = 4
- Grocery store = 0 + 5 = 5
- Gas station = 0 + 6 = 6



We're now done with Divya's house, so we label it as visited with an X. We see that the smallest marked vertex is the movie theater with a mark of 4. This our new current vertex, and we start again at step 2.

So , basically!!

- **Set all vertices distances = infinity except for the source vertex, set the source distance = 0.**
- **Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.**
- **Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).**
- **Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.**
- **If the popped vertex is visited before, just continue without using it.**
- **Apply the same algorithm again until the priority queue is empty.**

Q2. Describe the algorithm's specific implementation i.e. how have you implemented it in your task?

Ans 2.

Why Dijkstra, why not A* or D*Lite or others?

We have used Dijkstra's Algorithm, to tackle this problem which is also called as a single source shortest path algorithm, based on greedy technique. Now the reason for choosing the above was the difficulty of deciding the heuristics, so basically, the distance was the only parameter that is the from the goal. Also, the microcontroller easily handles O(n2) complexity, so why not?

How has it been implemented?

We have created a list "visited[22]" of vertices, whose shortest distance from the source is already known**.** If visited[1], equals 1, then the shortest distance of vertex i is already known.

At each step, we mark visited[v] as 1. Vertex v is a vertex at the shortest distance from the source vertex. At each step of the algorithm, the shortest distance of each vertex is stored in an array distance[ ]. Now, delving in deeper into the details of the algorithm and how it has been applied.

· First, we have created an adjacency matrix named arena that contains the distance between the ith and the jth node at both, arena[i][j] and arena[j][i], both of them being equal. And thus the matrix is symmetric.

· We have initialized the visited array to zero.

· If the vertex "start" is the source vertex then visited[start] is marked as 1.

· Then we have created the distance matrix, by storing the distance of vertices from vertex no. "start" say 0, to n-1.

```
distance[0]=0;
for(i=1;i<22;i++)
distance[i]=cost[0][i];
```

The above piece of code initializes the value of the distance[i] of the ith node from the start, here '0'.Needless to say, the distance of the source vertex is taken as 0, that is.

The following code is then executed 22 times for all the 22 nodes.
{for(i=1;i<n;i++)

· ***Choose a vertex w, such that distance[w] is minimum and visited[w] is 0. Mark visited[w] as 1.***

· ***Recalculate the shortest distance of remaining vertices from the source.***

· ***Only, the vertices not marked as 1 in array visited[ ] should be considered for recalculation of distance. i.e. for each vertex v***

· ***if(visited[v]==0)***

· ***distance[v]=min(distance[v],***

· ***distance[w]+cost[w][v])}***

· The program contains two nested loops each of which has a complexity of O(n). n is the number of vertices. So the complexity of the algorithm is $O(n^2)$.

<u>How does the bot get a sense of direction, know where to go what turns to take!!?</u>

The whole arena has been laid as a Cartesian coordinate. Starting from the atop the nodes are numbered from 0 to 22. The tenth node becomes the centre as clearly been depicted. In the picture below. The coordinates have been stored in x[],and y[]. And an adjacency matrix has been stored in the arena.

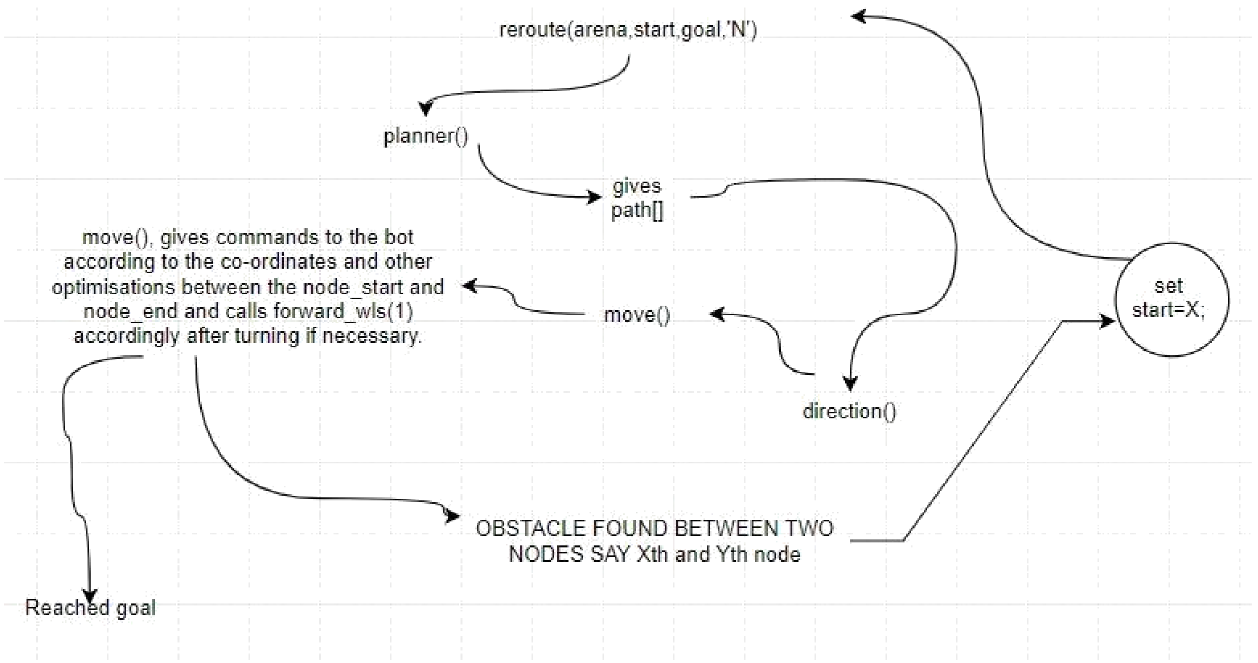

reroute(arena,start,goal,'N')
planner()
gives path[]
move(), gives commands to the bot according to the co-ordinates and other optimisations between the node_start and node_end and calls forward_wls(1) accordingly after turning if necessary.
move()
direction()
set start=X;
OBSTACLE FOUND BETWEEN TWO NODES SAY Xth and Yth node
Reached goal

FIGURE. 1

Now, to understand it much more clearly, let's take an example. say the bot has to go from 10 to 1. And there is an obstacle between 10 and 3.We set the goal to 3, and call reroute().

• ***g=1;reroute(arena,g,1);***

As it can be seen, in the code the reroute function calls planner, as

• ***planner (arena,10,1);***

The planner function execute Dijkstra's path planning algorithm calculates the shortest path between 10 and 1 which is stored in path[] as follows, the path comes out to be

10th node->8th node->1st node

The path[] being a global variable is then passed to,

• ***direction(path[],present orientation(N));***

direction calls, move(orientation, start, end), repetitively for **10th to 8th node**, and,**8th to 1st node,** move uses basic arithmetic, to find out the next move to make, for example, to go from 10th(start) to 8th(end) node, presently

facing **north(orientation),** so it should go forward thus we
call ***forward_wls(1);*** Similarly for right turns it calls right_turn_wls(), and for lefts it call left_turn_wls().
if an obstacle is encountered, the obstacle detecting condition in forward_wls(1), is triggered, and the reroute() is called again inside it, say the obstacle is encountered between 10 and 3 the re-route from 10 to 1 begins again as follows, after the arena[10][3]=0 and arena [3][10]=0 then consequentially, there gosts in the cost matrix defined in planner becomes 5000 each (bidirectionally). This will become much clearer from the chart below.

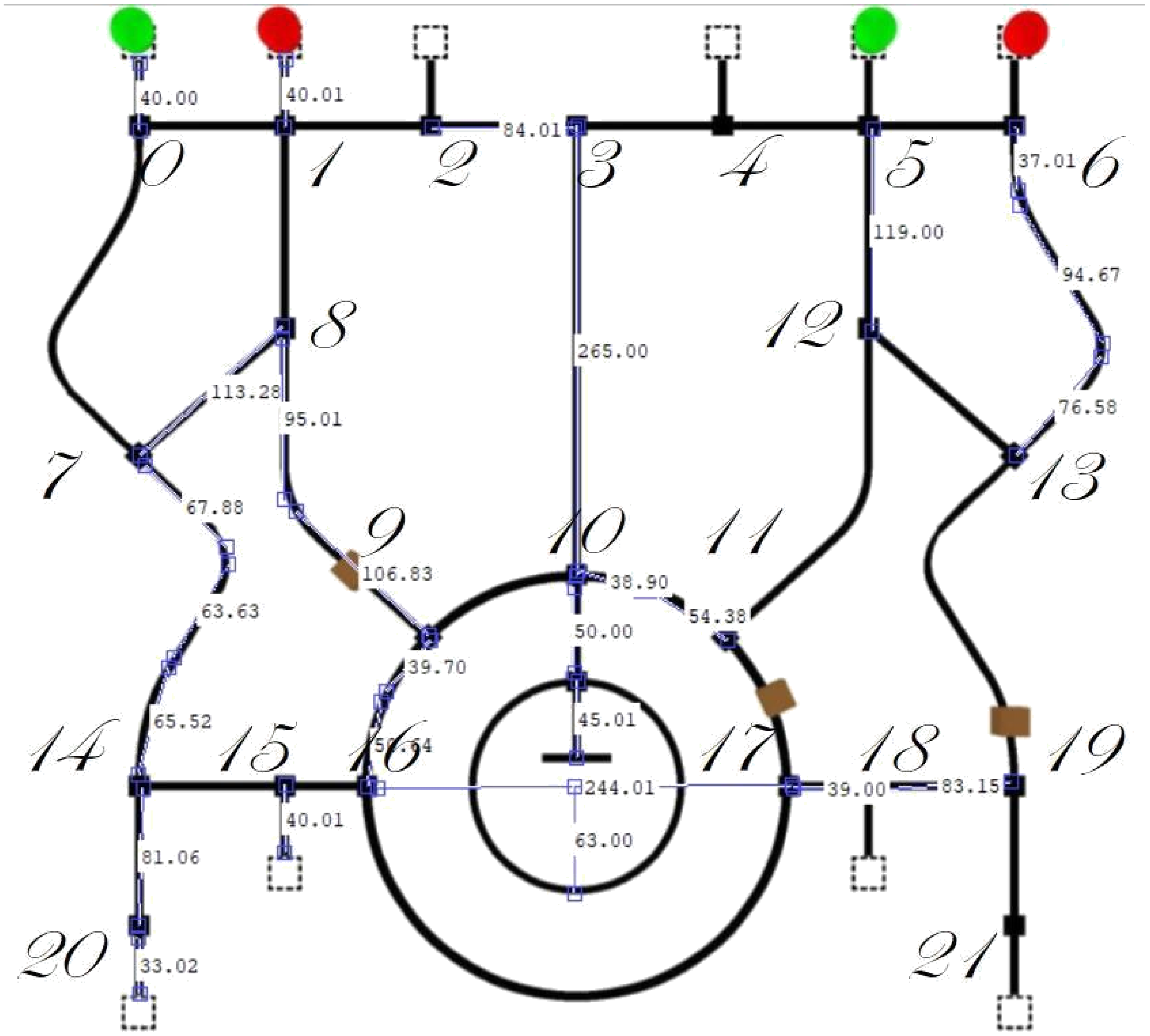


Figure. 2

This reroute function is then called in ***pick_it_up(start,pick_point, orientation)*** which directs the bot from origin to the pick point and checks the colour of the nut via check_color().

This reroute function is then called in ***go_place(start, goal, orientation, colour)*** which directs the bot from pick_point to the goal if the nut is picked, else calls itself again. recursively, ***go_place(pick_point,goal,orientation,color) ,*** until the nut is placed, also checks colors for the same.

The functions ***pick_it_up(start,pick_point, orientation) and go_place(start, goal, orientation, colour)*** are called in ***final(int start, int pick, int goal, char start_orientation, char pick_point_orientation)***, which adds the both of them with some optimizations to accomplish the task.

How does the bot follow the line?

We gave the normal line following algorithm a fair try but it was a struggle, so we turned to PID (Proportional integral and differential) methodology the reference of which we took fro here. But we had to use Matlab for the tuning of the bot by plotting the number of times alignments took place, or we can say the alignment function was called v/s Ki and Kp values and according to the graph the values converged at ,

Ki= .458 and Kd=5.15, where we made Kp firstly 120, the 110 and finally 105.

We will in future make use of encoders, just to let the robot "foresee" the turns and let it know what the speeds at various points should be. We'll also study some more about PID, and increase the speed but that would require a custom chassis.

THANK YOU FOR READING!