Peter Huynh

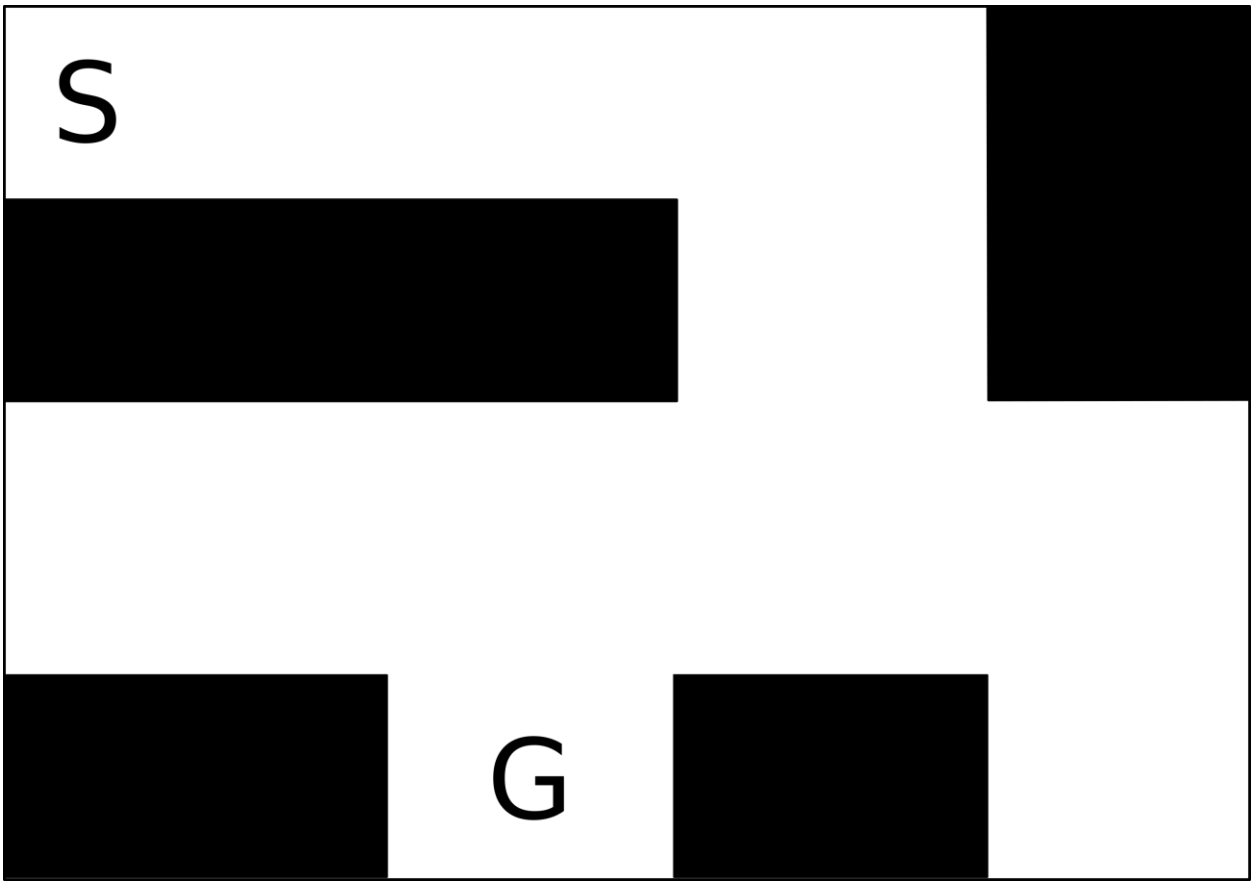
September 28, 2016

CSCI 3302

Lab 3.2

Mapping

Example of a 4x4 grid with obstacles:



Goal:

- Implement a basic discrete map representation and map an environment.
- Use coordinate transforms to map sensor readings into world coordinates
- Understand discrete and algorithmic aspects of mapping

Required:

- A Sparki robot

- A working implementation of odometry and line following
- Line following map
- Box and cylindrical obstacles

Overview:

Mapping and navigation are standard primitives in autonomous robots. Although the different map representations and planning algorithms vary drastically in complexity, the problems remain the same: localization and sensing is uncertain, maps are quickly corrupted, and both mapping and navigation have to deal with limited on-board resources. In this exercise you will use Sparki's ultrasound sensor to map objects in the environment. You will initially display the objects' coordinates on the screen and then implement data structures and helper functions that allow you (1) storing a map in Sparki's memory and (2) are suitable for path planning.

Instructions:

1. Place a couple of obstacles into the line following loop so that Sparki can still do the course
2. Record readings from the Ultrasound distance sensor and turn them into X-Y coordinates of the robot (using the known angle of the servo motor) and then into world coordinates. Display these points on Sparki's display. You have to choose a suitable scale factor to convert X-Y coordinates into pixels.
3. Create a data structure for maintaining a 4x4 map, for example "int map[4][4]". Use '0's for obstacles and '1's for navigable space. Write a function that maps a float X-Y coordinate into a discrete map coordinate. Use this function to determine whether a cell in the map is navigable or not.
4. An algorithm such as Dijkstra's is not necessarily made for grid maps, but calculates the distances from any node of a graph to a desired goal location. As an input it takes a matrix that contains the distance between every pair of nodes. You will need three functions: one that assigns a single number to each index of your 2D map, one that takes an index and returns the 2D coordinates, and one that returns the cost to move from one cell to another given by its index (not its coordinates). This last function should return "1" if two nodes of your graph are adjacent and obstacle-free, and a high integer, for example "99" otherwise.
5. Provide a write-up that answers the following questions.

(I) What is the drawback of a data structure that stores the distances between every possible pairs of nodes, and how does the implementation from (5) address this problem?

The main issue with storing distance in each node is that you'll have to access each individual node in order to calculate the optimal path with an algorithm. My implementation is a bit primitive, and defaults the distance between each node to 55. The world is set to be full traverse-able, and cuts off a node for obstacles and boundary limits. I also have a different array for check what is incomplete.

(II) Provide code snippets showing your data structure and functions from 4.

```
void findDistance()
{
    char incompletedMapNodes[16] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
  int nodes[16] = {0,50,50,50,50,50,50,50,50,50,50,50,50,50,50,50}, nodesLeft = 10,  minimumNode, minimumCost, newCost;

  // For Dijkstra's, calculates the proper node traversal cost
  while(nodesLeft > 0)
  {
    // Resets minimum values to dummy values
    minimumNode = 0;
    minimumCost = 300;

    // Searches for minimums every loop for any incompleted node
    for(i = 0; i < 16; i++)
    {
      if(incompletedMapNodes[i] > 0 && (nodes[i] < minimumCost || minimumNode < 0))
      {
        minimumNode = i;
        minimumCost = nodes[i];
      }
    }

    // Set row and column to the minimum values divided by 4 and remainder of 4, respectively
    currentRow = minimumNode / 4;
    currentColumn = minimumNode % 4;
    // New adjusted cost to node it min + 1, incompleted node set to completed
    newCost = minimumCost + 1;
    incompletedMapNodes[minimumNode] = 0;

    // Checks for column and row within boundaries and if the current node cost less than the new calc'd cost
    // Also checks if on an incompleted node and if position is not a blocked point.
    // If so, adjust node to new cost.
    if( currentColumn + 1 <= 3 &&
        nodes[(currentRow * 4 + currentColumn + 1)] > newCost &&
        incompletedMapNodes[(currentRow * 4 + currentColumn + 1)] > 0 &&
        worldMap[currentRow][currentColumn + 1] != 'X')
      nodes[(currentRow * 4 + currentColumn + 1)] = newCost;
    if( currentColumn - 1 >= 0 &&
        nodes[(currentRow * 4 + currentColumn - 1)] > newCost &&
        incompletedMapNodes[(currentRow * 4 + currentColumn - 1)] > 0 &&
        worldMap[currentRow][currentColumn - 1] != 'X')
        nodes[(currentRow * 4 + currentColumn - 1)] = newCost;
    if( currentRow + 1 <= 3 &&
        nodes[((currentRow + 1) * 4 + currentColumn)] > newCost &&
        incompletedMapNodes[((currentRow + 1) * 4 + currentColumn)] > 0 &&
        worldMap[currentRow + 1][currentColumn] != 'X')
      nodes[((currentRow + 1) * 4 + currentColumn)] = newCost;
    if( currentRow - 1 >= 0 &&
        nodes[((currentRow - 1) * 4 + currentColumn)] > newCost &&
        incompletedMapNodes[((currentRow - 1) * 4 + currentColumn)] > 0 &&
        worldMap[currentRow - 1][currentColumn] != 'X')
      nodes[((currentRow - 1) * 4 + currentColumn)] = newCost;

    nodesLeft--;
  }
}
```

```
int worldMap[4][4] = {{1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1}};

/*
 *  worldMap - Initiallized as an open map to traverse, will set 0s when spotting road blocks
 *  ------------
 *  | 1 1 1 1 |
 *  | 1 1 1 1 |
 *  | 1 1 1 1 |
 *  | 1 1 1 1 |
 *  ------------
 */
```

```
// Object Detection, changes world map if sees object within 20 cm
cmDetect = sparki.ping();
if(cmDetect != -1)
{
  if(cmDetect < 20)
  {
    // Northern Object
    if(currentColumn+1 <= 3 && (theta <= 5 && theta >= -5) || (theta <= 365 && theta >= 355) || (theta <= -355 && theta >= -365))
      worldMap[currentRow][currentColumn+1] = 0;
    // East
    else if(currentRow + 1 <= 3 && (theta >= 85 && theta <= 95) || (theta >= -275 && theta <= -265))
      worldMap[currentRow+1][currentColumn] = 0;
    // West
    else if(currentRow-1 >= 0 && (theta >= 265 && theta <= 275) || (theta >= -95 && theta <= -85))
      worldMap[currentRow-1][currentColumn] = 0;
    // South
    else if(currentColumn-1 >= 0 && (theta >= 175 && theta <= 185) || (theta >= -185 && theta <= -175))
      worldMap[currentRow][currentColumn-1] = 0;
  }
}
```