Peter Huynh

October 3, 2016

CSCI 3302

Lab 3.2

Path Planning

Goal:

- Implement a shortest-path finding algorithm (Dijkstra's algorithm)
- Use shortest path information from Dijkstra to generate an actual route

Required:

- Sparki robot
- 2D grid data structure and helper functions to turn the 2D grid into a graph and calculate cost between nodes

Dijkstra's algorithm takes a graph with N nodes as an input and returns the cost of every single node to a user specified source node. For example code, have a look at:

http://www.c-program-example.com/2011/10/c-program-to-solve-dijkstras-algorithm.html

(Note that there is a typo in the code, it should be a "source node" not a "source matrix"). For path planning purposes, you treat every cell of your 2D map as a node of a graph with edge cost that are inferred from the map geometry and presence of obstacles. Provided with a "source" node, Dijkstra's algorithm will provide you with the distance of every single node to that source.

Instructions:

1. Implement a version of Dijkstra's algorithm on Sparki. Replace UI commands from the example with your data structures. In particular, replace the cost matrix with your distance function. Use a simple 4x4 map as in the last exercise to validate that Dijkstra's returns the correct distances.
2. Modify the code so that Dijkstra's stores the id of the parent node. That is the node which comes next on the shortest path in a separate data structure.
3. Write code that returns a sequence of vertices from a given location on the map to the source.

Deliverables:

Explain your algorithm in a write-up and show sample outputs of steps 1, 2 and 3.

Step 1:

Using the 4x4 3D map array from last lab, and a newly hard coded world map for obstacles, I was able to calculate the cost of each distance by combining a cost matrix, node, with a simple set of condition checks to determine the best traversal path. As seen in my function below, the first half was part of my program last lab, which was used to determine node traversal costs. The second half of my function is set to grab the distance values adjacent to the current node and pick the lowest one.

Step 2:

I use the bestPathMap array to store the parent node ID This array will be filled with nodes to determine the best path. These are set in the near end of my function, in the second half, right after the conditional checks for which adjacent node from the current position is the lowest costing.

Step 3:

This part is also done in the second half of my Dijkstra function. I set the current row and column variables to be the goal (Row 4, Column 2 – which is currentRow = 3, currentColumn = 1) Starting from the goal, the conditionals check for adjacent node distances, then set the lowest distance to a new matrix for the best path, and readjusts the row and column values to be repeated within a loop until both current row and column variables are 0, the starting point.

Below is the function and global variables I used for Dijkstra's:

```
// Globals
int i, j, cmDetect, north = 300, west = 300, east = 300, south = 300, currentColumn = 0,
currentRow = 0, theta = 0, startTime, endTime, totalTime;
float x = 0.0, y = 0.0, rightWheel = 0.0, leftWheel = 0.0, halfLeftRight = 0.0;
int worldMap[4][4] = {{1,1,1,0}, {0,0,1,0}, {1,1,1,1}, {0,1,0,1}};
int bestPathMap[4][4] = {{1,0,0,0},{0,0,0,0},{0,0,0,0},{0,1,0,0}};

/*
 *   worldMap - Hard coded map 3D array of the world
 *   1 = traverse-able, 0 = blocks
 *  -----------
 *  | 1 1 1 0 |
 *  | 0 0 1 0 |
 *  | 1 1 1 1 |
 *  | 0 1 0 1 |
 *  -----------
 *
 *   bestPathMap - Path map, initially only showing start and goal, will be filled with
optimal path
 *  -----------
 *  | 1 0 0 0 |
 *  | 0 0 0 0 |
 *  | 0 0 0 0 |
 *  | 0 1 0 0 |
 *  -----------
 */

void dijkstra()
{
  // Local Variables
  int incompletedMapNodes[16] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}, mapNodes[16] =
{0,50,50,50,50,50,50,50,50,50,50,50,50,50,50,50}, remainingNodes = 10,  minimumNode,
minimumCost, newCost;

  /* ********************
   *
   * Dijkstra Part 1
   * Calculates the proper traversal cost between each node
   *
   ******************** */
  while(remainingNodes > 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 && (mapNodes[i] < minimumCost || minimumNode < 0))
      {
        minimumNode = i;
        minimumCost = mapNodes[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.
    // Big if statements :(
    if( currentColumn + 1 <= 3 &&
        mapNodes[(currentRow * 4 + currentColumn + 1)] > newCost &&
        incompletedMapNodes[(currentRow * 4 + currentColumn + 1)] > 0 &&
        worldMap[currentRow][currentColumn + 1] != 0)
      mapNodes[(currentRow * 4 + currentColumn + 1)] = newCost;
    if( currentColumn - 1 >= 0 &&
        mapNodes[(currentRow * 4 + currentColumn - 1)] > newCost &&
        incompletedMapNodes[(currentRow * 4 + currentColumn - 1)] > 0 &&
        worldMap[currentRow][currentColumn - 1] != 0)
        mapNodes[(currentRow * 4 + currentColumn - 1)] = newCost;
    if( currentRow + 1 <= 3 &&
        mapNodes[((currentRow + 1) * 4 + currentColumn)] > newCost &&
        incompletedMapNodes[((currentRow + 1) * 4 + currentColumn)] > 0 &&
        worldMap[currentRow + 1][currentColumn] != 0)
      mapNodes[((currentRow + 1) * 4 + currentColumn)] = newCost;
    if( currentRow - 1 >= 0 &&
        mapNodes[((currentRow - 1) * 4 + currentColumn)] > newCost &&
        incompletedMapNodes[((currentRow - 1) * 4 + currentColumn)] > 0 &&
        worldMap[currentRow - 1][currentColumn] != 0)
      mapNodes[((currentRow - 1) * 4 + currentColumn)] = newCost;

    // One less node left to do
    remainingNodes--;
  }

  /* ********************
   *
   * Dijkstras Part 2
   * Finds ideal graph this loop, after finding all node distances. Starts from goal to
start.
   *
   ******************** */
  currentRow = 3, currentColumn = 1;
  while(currentRow > 0 || currentColumn > 0)
  {
    // Checks for map boundaries, then does compass value settings
    // In order of north, east, south and then west (CLOCKWISE) in priority
    if(currentColumn + 1 <= 3) north = mapNodes[(currentRow * 4 + currentColumn + 1)];
    if(currentColumn - 1 >= 0) south = mapNodes[(currentRow * 4 + currentColumn - 1)];
    if(currentRow + 1 <= 3) east = mapNodes[((currentRow + 1) * 4 + currentColumn)];
    if(currentRow - 1 >= 0) west = mapNodes[((currentRow - 1) * 4 + currentColumn)];

    // Finds shortest path out of compass directions, sets path & updates position
    if(north < east && north < south && north < west)
    {
      bestPathMap[currentRow][currentColumn+1] = 1;
      currentColumn += 1;
    }
    else if(east < north && east < south && east < west)
    {
      bestPathMap[currentRow+1][currentColumn] = 1;
      currentRow += 1;
    }
    else if(south < north && south < east && south < west)
    {
      bestPathMap[currentRow][currentColumn-1] = 1;
      currentColumn -= 1;
    }
    else if(west < north && west < east && west < south)
    {
      bestPathMap[currentRow - 1][currentColumn] = 1;
      currentRow -= 1;
    }
  }
}
```