PROJECT 2 - 100 POINTS

DUE DATE: OCTOBER 23, 2013, 11:59:59 PM PST
CSCI 561, FALL 2013

1. Introduction

In Project 1, we saw that moving an agent from a starting state or location to a goal state or location using local and global sensors is one of the most fundamental tasks in Artificial Intelligence. With this building block in place, we can now extend the problem to include many different nodes on a map, one of which is designated as the starting location and one of which is designated as the goal location. In this problem, the goal may not be reachable in a single straight-line shot because the robot is only allowed to travel from one node (city) to another if an edge (road) exists between those nodes. You can think of this as an abstract version of the cities problem (traveling to Bucharest) in chapter 3 of AIMA. This is how we will formalize the problem. In Figure 1, you will see Wheelbot planning and executing a path to the goal city. The starting city is yellow, and the goal city is green, all other visitable cities are blue. The red objects represent road blocks. Roads with road blocks are non-traversable, even though the robot's map says they are usable. The green path is the robot's planned path to the goal. The light blue links represent the roads between cities the robot knows about (from its map). Your job will be to plan a shortest cost path to the goal city from the starting city, re-planning as necessary when road blocks are discovered.

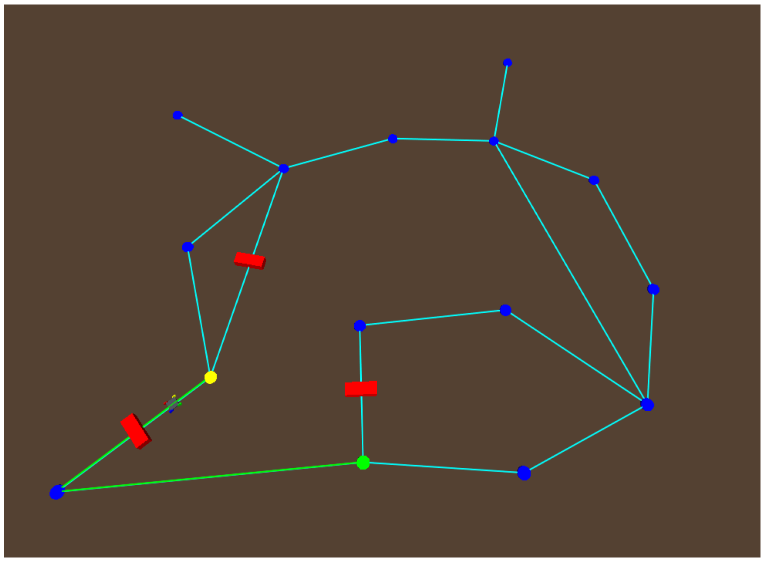

Figure 1. An Example Map for Project 2

2. Wheelbot

In this project, you will again be using your old friend Wheelbot, the 4-wheeled mobile robot. For the purposes of this assignment, Wheelbot has the following built-in knowledge base, actions, and sensors.

2.1. Wheelbot Knowledge Base.

(1) The Map: In this project, most of Wheelbot's knowledge about the world is represented as an object of class *Graph* that represents the map of cities. This knowledge is accessed and update using the *this->g* member of the Project2_Control class. The *Graph* class has two members, *graphPoints* and *graphEdges*.

  (a) *graphPoints* is defined as type *vector<PxVec3 >.* It contains all the 3D points representing the locations of the cities on the map. The index of the city in this vector is a suitable (unique) way of identifying each city.

  (b) *graphEdges* is defined as type *vector<pair <int, int>>.* It contains all the edges between cities in the graph. Each edge is a *pair<int, int* > representing the undirected graph edge between *node1* and *node2*. Here is how to interpret the edges:

```
pair<int, int> p = graphEdges[0]; //This is the first edge in the map
int node1 = p.first; //node1 is index of node1 in graphPoints
int node2 = p.second; //node2 is index of node2 in graphPoints
```

(2) Start node: the index of the node (city) at which you start on the map. It can be accessed and updated using the *static* member *start* of the Project2_Control class.

(3) Goal node: the index of the node (city) you are trying to reach on the map. It can be accessed as the *static* member *goal* of the Project2_Control class.

2.2. Wheelbot Actions.

(1) ACTION_FORWARD: Wheelbot moves forward until the next time step (to use, call *takeAction(ACTION_FORWARD)*)

(2) ACTION_BACKWARD: Wheelbot moves backward until the next time step (to use, call *takeAction(ACTION_BACKWARD)*)

(3) ACTION_LEFT: Wheelbot turns in place (counter-clockwise) until the next time step (to use, call *takeAction(ACTION_LEFT)*)

(4) ACTION_RIGHT: Wheelbot turns in place (clockwise) until the next time step (to use, call *takeAction(ACTION_RIGHT)*)

(5) ACTION_STOP: Wheelbot stops moving altogether until the next time step (to use, call *takeAction(ACTION_STOP)*)

These are the same actions as in Project1.

2.3. Wheelbot Sensors.

(1) Wheelbot Position/Orientation Sensor: Returns the current 3D pose of Wheelbot (position and orientation). To use this sensor, call *getCurrentWheelbotPose()*.

(2) Range Sensor: Returns an object representing the intersection of a ray (laser) along Wheelbot's positive x-axis and an obstacle. To use this sensor, call *getRangeResult()*. The range sensor can only detect the distance between Wheelbot and an obstacle – city hemispheres will not be registered as obstacles – when the object is within 0.3

meters of the obstacle. This sensor returns a *PxRaycastHit* object. Here is how to use this sensor:

```
PxRaycastHit hit = getRangeResult(); //Call range sensor
PxF32 dist = hit.distance(); //Distance between Wheelbot and obstacle.
```

The *PxRaycastHit* object will report a distance of 0.3 if nothing is detected and a distance less than 0.3 if something is detected. A good threshold value to detect an obstacle is around 2.8 meters. This will ensure no false positives.

3. Project Details/Requirements

This project will require you to modify the file Project2-Control.h in the project source code. You will find this file in your IDE under Behaviors/RM3D. In this project, you have (almost) complete control over Project2-Control.h You can add and modify code as you wish, so long as you meet the project requirements and the code is not between DO NOT MODIFY and END DO NOT MODIFY blocks. You should not need to delete any code, so please do not. You can add code without restriction, though. In this project, a map of 100 cities is randomly generated. This map is guaranteed to have no intersecting edges and some cycles. Some edges (roads) will be blocked by red road blocks (approximately 10% of them), indicating that the edge cannot be traversed. In addition, the goal city and starting city will be randomly selected from among all of the nodes. Wheelbot will initially be placed in a random orientation at the starting city and must make his way to the goal city, if possible. More specifically, the code you write must do the following:

(1) Use the A* search algorithm to plan a path from the starting city to the goal city. It must use the *plan* vector (as illustrated in the provided Project2-Control.h code) to visualize the current plan as a green path from the start city to the goal city. The cost of traveling from one city to another is simply the distance between the two city points. For a heuristic, you may use the straight-line distance between a city and the goal city. As discussed in class, this is always an admissible and consistent heuristic.
(2) Use the code you wrote in Project 1 (or the provided line following code) to execute the planned path by following only the edges (available roads) of the map.
(3) If you encounter an obstacle on the way to the goal (which your range sensor will indicate), then you must return to the most recent node (city) you visited and plan an alternate least cost path to the goal. The city from which you plan becomes your new start city. You must update the graph object (*this->g*) to indicate that the path just tried is not viable. Thus, your A* implementation must be able to plan a least-cost path from any city to any other city on any randomly generated map and must handle updates to the map between A* calls.
(4) If you have tried every possible path to the goal, you should be able to detect the fact that there are no possible paths to the goal (due to obstacles). When you encounter this situation, you should stop and print out an error message to the screen, such as "Failure: No possible path to goal."
(5) If you successfully reach the goal, you should stop and print out a success message to the screen, such as "Success!"

4. Guidelines

In this project, we are giving you great freedom with your Project2_Control.h file, primarily because everyone thinks about problems differently, and we don't want to constrain your solutions more than is necessary. However, there are some of things that are not allowed. First of all, anything between DO NOT MODIFY and END DO NOT MODIFY blocks cannot be changed. These blocks are placed around system level code that is necessary for the simulator and Wheelbot to function appropriately. As a general rule, you obviously cannot change the definition of the problem itself. You cannot change the goal city to make the problem easier, for instance. You also cannot update your knowledge of the world without appropriate cause. You must only update affected map edges (roads) when you encounter road blocks. In other words, when you encounter an obstacle, the only edge you can update is the one on which you found the obstacle. You may change your *start* node index, but not your *goal* node index. We (the instructors) will make the final decision on what code is and isn't appropriate. It is impossible to list all possibilities here. In general, any change the trivializes the problem or changes the definition of the problem without cause is incorrect and will receive little (if any) credit.

We will post a video on http://isi.edu/robots/CS561/Projects illustrating the desired behavior for this project on a few different runs. Another thing to note is the following: the map that is generated is guaranteed to have at least one path to the goal (ignoring obstacles). It is also guaranteed to have some cycles and some obstacles. Finally, it is guaranteed to be a proper planar graph (no two edges will intersect one another except at nodes) and all pairs of nodes are guaranteed two be at least 4 meters apart from one another. However, due to the complexity of generating random graphs, you may run into some degenerate cases. For example, Two edges may be too close to one another, leading Wheelbot to run into the side of an obstacle. You can safely ignore all such degenerate cases where Wheelbot cannot traverse the graph correctly. Simply restart the program in order to generate a new graph. Similarly, if you encounter a runtime error due to an invalid Wheelbot pose, simply restart the program and ignore it. We will not test these situations. With high probability, you will get a non-degenerate graph and a valid Wheelbot pose. However, in all other cases, your code must perform correctly. This is true even for cases such as when the goal and the starting city are one link apart (they will never be exactly the same city) or when an obstacle blocks the only link between the starting city and the rest of the map.

5. Setup

You will find the necessary files and PDF for Project 2 on http://isi.edu/robots/CS561/Projects in a zip file called Project_2_Files.zip. This zip archive contains 7 files:

(1) Project2.pdf: the PDF file for this project.
(2) BSensors.h: overwrite the file with the same name in CS561-Simulator/rm3d/PhysX/System/Modules/BBot with this file.
(3) Project2-Control.h: overwrite the file with the same name in CS561-Simulator/rm3d/PhysX/Behaviors/RM3D with this file.
(4) Project2.cpp: overwrite the file with the same name in CS561-Simulator/rm3d/PhysX/Environments/RM3D with this file.
(5) Simulator.h: overwrite the file with the same name in CS561-Simulator/rm3d/PhysX/System/Headers with this file.

(6) WheelbotModule.cpp: overwrite the file with the same name in
CS561-Simulator/rm3d/PhysX/System/Modules/Wheelbot with this file.
(7) WheelbotSensors.h: overwrite the file with the same name in
CS561-Simulator/rm3d/PhysX/System/Modules/Wheelbot with this file.

Once you have performed the file overwrites as specified above, you should be able to run the Project2 configuration without any errors. When the program starts up, you will see a randomly generated map, and Wheelbot will start moving along a randomly selected edge that leads out of the start city (node).

6. Submission

For this project, all you need to submit is your Project2-Control.h file. This time, we will create a Blackboard assignment for this project, and you will be able to submit it there. More details will be provided in class as the deadline approaches.