# PROJECT 3

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

## 1. Introduction

In Project 2, we gave Wheelbot the ability to plan a route from one location to another and to execute that route. We also kept track of Wheelbot's local sensor readings in order to plan detours when routes were blocked by unknown obstacles. In this project, we will take this idea further, endowing Wheelbot with the ability to learn unknown maps as well as to draw inferences that help it avoid dangerous places.

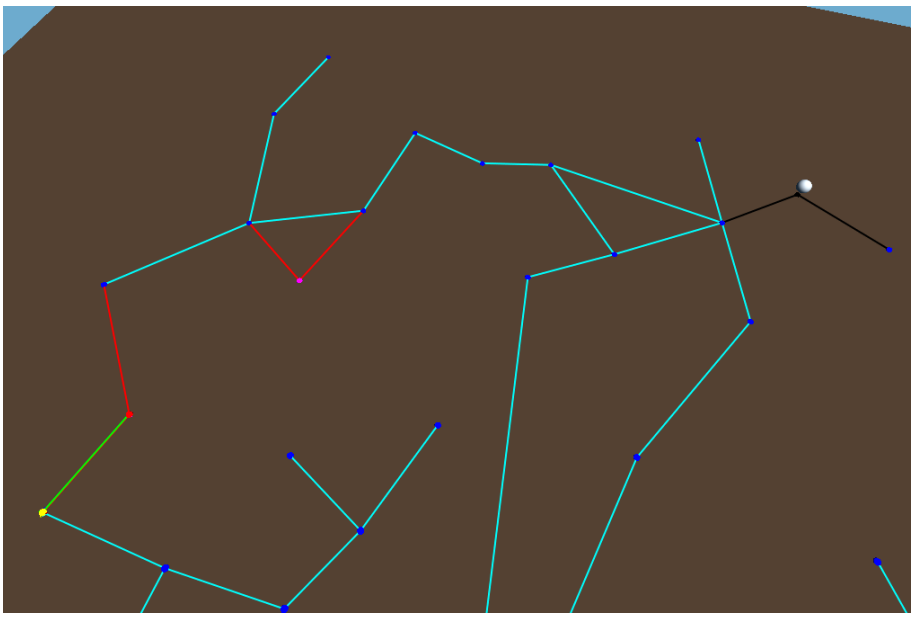

Figure 1. An Example Map for Project 3

## 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 *learnedG* member of the Project3_Control class. The

*Graph* class has two members, *graphPoints* and *graphEdges*. In contrast to Project 2, in Project 3, Wheelbot's map is initially empty. It knows nothing except for the fact that the world can be represented as a planar graph with nodes and edges.

(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
```

2.2. **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)*)

(6) ACTION_SHOOT: Wheelbot shoots an arrow in a given direction (global frame) a given distance (to use call *takeAction(ACTION_SHOOT)*). Here is an example: *takeAction(ACTION_SHOOT, PxVec2(1,0), 1.0)* would shoot an arrow 1.0 unit along the global X-axis.

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) Number of Map Nodes Sensor: Returns the number of nodes in graph *g*. To use this sensor, call *getNumGraphNodes()*.

(3) Number of Map Edges Sensor: Returns the number of edges in graph *g*. To use this sensor, call *getNumGraphEdges()*.

(4) At Node Sensor: Returns true if Wheelbot is currently at a map node (in graph *g*), and false if Wheelbot is between nodes in *g*. To use this sensor, call *atANode()*.

(5) Current Node Sensor: Returns the *PxVec3* object corresponding to the global, 3D position of the current node in graph *g*. *PxVec3(-100000)* is returned if Wheelbot is not currently at a node. To use this sensor, call *getCurrentNode()*.

(6) Actions For Current Node Sensor: Returns the actions available at the current node in graph *g* as a *vector <PxVec2 >*. These vectors are specified in the global frame relative to Wheelbot's current position and are normalized. To use this sensor, call *getActionVectorsForCurrentNode()*.

(7) Action Vector for Action at Node in Learned Graph Sensor: Returns the action vector corresponding to the given action at the given node in graph *learnedG*, the graph to be learned. This sensor tells you how to execute an action in the parts of the graph you have already learned. To use this sensor, call *getVectorForActionAtNode()*.
(8) Wind Sensor: Returns a vector of floats corresponding to the winds felt by the robot at the current node in graph *g*. These indices of these winds correspond exactly to the indices of the action vectors returned by *getActionVectorsForCurrentNode()*. The magnitude of the wind is the same as the distance from the current node in graph *g* to the pit node in graph *g*. To use this sensor, call *getCurrentWinds()*.
(9) Smell Sensor: Returns a vector of floats corresponding to the smells smelled by the robot at the current node in graph *g*. These indices of these smells correspond exactly to the indices of the action vectors returned by *getActionVectorsForCurrentNode()*. The magnitude of the smell is the same as the distance from the current node in graph *g* to the wumpus node in graph *g*. To use this sensor, call *getCurrentSmells()*.

## 3. Project Details/Requirements

This project will require you to modify the file Projecte-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 Project3-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. In this project, a graph of some number of nodes and edges is automatically generated for you. Based on the settings you select in Project3.cpp, this graph may or may not contain Wumpuses and pits as well. The goal of this project is to use the sensors and actions available to you in order to learn the map *g* without falling into a pit or running into a Wumpus. You must do this without directly accessing or modifying the graph *g* in any way. More specifically, the code you write must do the following:

(1) Use the given sensors to explore the map *g*. Whenever you come across a new node, you should add it to the graph *learnedG* (which you can directly access). You should also add new edges from the old node to the new node and from the new one to the old one (as each edge is traversable both ways). If you have done this correctly, the rendering of the added nodes and edges will be done for you.
(2) Use the code you wrote in Project 2 (or the provided A* code) to execute planned paths along the nodes in *learnedG* when you need to explore nodes that you haven't explored yet. In other words, when you go to explore a new node (or continue exploring an old one), you should always take the shortest path to that node before you start or continue exploring it.
(3) Before you go in a new direction from an existing node, you need to check whether or not you can feel wind or smell a stench. If you feel wind, there is probability 1 that a pit exists in that direction at a distance equal to the strength of the wind. In this case, rather than exploring this path, you can infer the approximate location of the new pit node based on this distance. Take care in making sure that you are not seeing the same pit from two different angles!
(4) If you can smell a stench, there is probability 1 that a Wumpus exists in that direction at a distance equal to the strength of the stench. In this case, you must shoot at the

Wumpus. You will either kill it, or it will flee to a new random neighboring node in $g$ (if available). You MUST shoot at the Wumpus every time you head in its direction to either move it (or kill it) so you can continue exploring the map. If you trap the Wumpus in a corner, you should continue shooting at it until you kill it, as it can't move anywhere.

(5) Once you have explored every node that you can fully, you can check whether or not the graph you made has the same number of nodes and the same number of edges as the graph $g$ to be learned. If so, you can declare success and end if your graph corresponds almost exactly to graph $g$. You don't need to program any sort of comparison between $g$ and *learnedG*, just ensure visually that they look almost the same.

(6) If you have run out of nodes to explore and your graph is not the same size as $g$, then you can declare failure. If your code is working, then it must be the case that pit renders one or more nodes reachable. You must indicate that this is the situation.

## 4. Guidelines

In this project, we are giving you great freedom with your Project3_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 map to make the problem easier, for instance, nor can you update *learnedG* without appropriate cause. The only time you should update *learnedG* is when you discover a new node. At this point, you can add new nodes and edges. To reiterate, there is absolutely no reason in this project to access or update $g$ (the graph to be learned) directly. All functionality you will need and all data you will need to complete the problem are exposed via *learnedG*, Wheelbot's sensors, and Wheelbot's actions. 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.
An additional difference between Project 2 and Project 3 is that this project will be graded in levels. The levels of this project, the points awarded for each level (approximately), and the settings necessary to put Project3 into the appropriate mode, are given below. Each level contains all levels below it. In other words, you cannot skip lower levels to get to higher ones. They must be done in order. To configure the project for a certain level, set the settings as specified below in Project3.cpp.

(1) Level I: In this level, Wheelbot must successfully explore maps with no pits and no Wumpuses in either a Depth-First Search or Breadth-First search fashion. More points will be awarded for Depth-First search strategies, as they are more difficult. This level awards 3 - 5 points (approximately). To activate this level, set *numPits = 0*, *numWumpuses = 0*, and *shootingNoise = false* in Project3.cpp (the *init()* function).
(2) Level II: In this level, Wheelbot must successfully explore maps with an indeterminate number of pits and no Wumpuses. This level awards 5 - 6 points (approximately). To activate this level, set *numPits* $\geq$ *1*, *numWumpuses = 0*, and *shootingNoise = false* in Project3.cpp (the *init()* function).
(3) Level III: In this level, Wheelbot must successfully explore maps with an indeterminate number of pits and at least one Wumpus. Wheelbot must explore as much of the map as possible and kill all the Wumpuses it comes across.This level awards 6 - 8 points (approximately). To activate this level, set *numPits* $\geq$ *1*, *numWumpuses* $\geq$ *1*, and *shootingNoise = false* in Project3.cpp (the *init()* function).
(4) Level IV: In this level, Wheelbot must deal with a noisy shooting mechanism, meaning that it may not kill the Wumpus when it tries to shoot it. Wheelbot must explore as much of the map as possible, while trying to kill the Wumpus each time it is about to traverse an edge that is smelly. If Wheelbot traps the Wumpus, it must continue firing arrows until the Wumpus is dead. Due to the noise in the shooting mechanism, it may be that Wheelbot does not kill all Wumpuses. This is fine, so long as Wheelbot tries each time it comes across a Wumpus during exploration. Since the Wumpuses move, Wheelbot may encounter the same one multiple times. This level awards 8 - 10 points (approximately). To activate this level, set *numPits* $\geq$ *1*, *numWumpuses* $\geq$ *1*, and *shootingNoise = true* in Project3.cpp (the *init()* function).

There may be a Level V added in the next couple of weeks. If this happens, the point values would be shifted downward to accommodate the new level, which would award approximately 9 -10 points. This is yet to be determined, but it will not be extra credit. It will be part of the project itself.

## 5. Setup

You will find the necessary files and PDF for Project 3 on http://isi.edu/robots/CS561/Projects in a zip file called Project_3_Files.zip. This zip archive contains 3 files:

(1) Projec3.pdf: the PDF file for this project.
(2) Project3-Control.h: overwrite the file with the same name in CS561-Simulator/rm3d/PhysX/Behaviors/RM3D with this file.

(3) Project3.cpp: overwrite the file with the same name in CS561-Simulator/rm3d/PhysX/Environments/RM3D with this file.

Once you have performed the file overwrites as specified above, you should be able to run the Project3 configuration without any errors. When the program starts up, you will see a randomly generated map and Wheelbot shooting is arrows in random directions. There will also be a node that Wheelbot "learned" by placing it arbitrarily out in front of itself. This dummy code in Project3-Control.h should be removed and replaced with your code.

## 6. Submission

For this project, all you need to submit is your Project3-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.