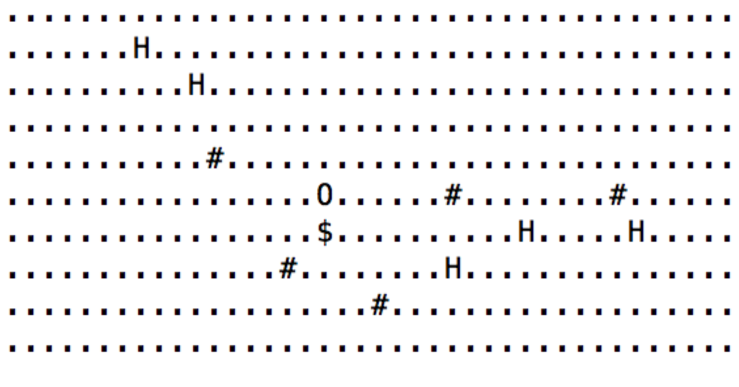
CSCI 360 – Project #2 - 10 Points

Part 1 - Navigating Wheelbot in Partially Known Environments - 7 Points

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. In this part of the project, you will implement an A* search to navigate Wheelbot to its goal location on a map with obstacles. Some of these obstacles are initially known and some of them are hidden.

Before executing a move action, Wheelbot should use its local obstacle sensors to discover any hidden obstacles around it. Wheelbot should always follow a shortest path to its goal, assuming that the grid only contains the initially known obstacles and the discovered hidden obstacles (any hidden obstacles that have not been discovered yet, or the knowledge that there might be hidden obstacles, should not affect the path that Wheelbot follows). **Contrary to Project 1, in this project, diagonal movements have a cost of 1.5** (the rest have cost 1), and Wheelbot should minimize the total cost of the edges along the path, rather than the total number of edges traversed. Wheelbot can move to any cell that does not contain an obstacle and it dies if it moves to a cell that contains an obstacle.

Below is a screen shot of the text-based simulator that you will use for this project. It has been extended to represent known obstacles (#) and hidden obstacles (*H*).



```
........................................
.......H................................
..........H.............................
........................................
...........#............................
.................0......#........#......
.................$..........H.....H.....
...............#........H...............
....................#...................
........................................
```

Figure 1: A screenshot of the text-based simulator. The environment is laid out in a discrete grid. Each period represents an unoccupied location in the environment. The robot's location is represented by the *0*, while the target location is represented by *$*. Known obstacles are represented by # and hidden obstacles are represented by *H*.

Simulator Details

In this course, you will be working primarily with a robot called Wheelbot, an abstract mobile robot encoded in the *Robot* class. **At each simulation step, Wheelbot can read its sensors and then take one action, moving it one space in the given direction**. For the purposes of this assignment, Wheelbot has the following actions and sensors.

Wheelbot Actions

1. MOVE_UP: Wheelbot moves up one space in the grid
   (to use, call *r1→setRobotAction(MOVE_UP)*)
2. MOVE_DOWN: Wheelbot moves down one space in the grid
   (to use, call *r1→setRobotAction(MOVE_DOWN)*)
3. MOVE_LEFT: Wheelbot moves left one space in the grid
   (to use, call *r1→setRobotAction(MOVE_LEFT)*)
4. MOVE_RIGHT: Wheelbot moves right one space in the grid
   (to use, call *r1→setRobotAction(MOVE_RIGHT)*)
5. MOVE_UP_RIGHT: Wheelbot moves up and right diagonally one space in the grid
   (to use, call *r1→setRobotAction(MOVE_UP_RIGHT)*)
6. MOVE_UP_LEFT: Wheelbot moves up and left diagonally one space in the grid
   (to use, call *r1→setRobotAction(MOVE_UP_LEFT)*)
7. MOVE_DOWN_RIGHT: Wheelbot moves down and right diagonally one space in the grid
   (to use, call *r1→setRobotAction(MOVE_DOWN_RIGHT)*)
8. MOVE_DOWN_LEFT: Wheelbot moves down and left diagonally one space in the grid
   (to use, call *r1→setRobotAction(MOVE_DOWN_LEFT)*)

These are the same actions as in Project 1. *r1* is an instance of Wheelbot, and it will be provided for you to use in the code you need to modify.

Wheelbot Sensors

1. Robot Position Sensor: Returns the 2D position of the robot. To use this sensor, call *r1→ getPosition()*, which returns a 2D point. This is a local sensor of the Wheelbot.
2. Target Position Sensor: Returns the 2D position of the target. To use this sensor, call *sim1→ getTarget()*, which returns a 2D point. This is a global sensor, which returns the absolute target location regardless of Wheelbot's position.

3. Local Obstacle Sensor: Returns the 2D positions of all the obstacles in the eight cells around Wheelbot. To use this sensor, call *r1*$\rightarrow$ *getLocalObstacleLocations()*, which returns a vector of 2D points.

sim1 is an instance of the simulation environment and will be provided for you in the code you need to modify. The location of the robot and the target are returned as instances of the *Point2D* class that has the variables x (row) and y (column). The top-left corner of the map has coordinates (0,0).

We have also added several new functions to the Simulator class, which you can use to obtain information about the environment:

1. You can get the dimensions of the grid by calling *sim1*$\rightarrow$ *getWidth()* and *sim1*$\rightarrow$ *getHeight()*. Both functions return integers.

2. You can get a list of all the initially known obstacles by calling *sim1*$\rightarrow$ *getKnownObstacleLocations()*, which returns a vector of 2D points. Note that this function will not return any hidden obstacles that have been discovered by the agent.

Running on Linux

If you choose to code in a Linux environment, you might encounter the following errors:

- `stoi is not a member of std`
  Solution: add `-std=c++11` to Makefile
  (for example, `g++ -std=c++11 -o proj2 main.cpp Robot.cpp Project2.cpp`).

- `usleep was not declared in this scope`
  Solution: replace line 127 (`usleep(1000*waitCounter);`) in main.cpp:
  ```
  struct timespec req, rem;
  req.tv_nsec = 1000000*waitCounter;
  req.tv_sec = 0;
  nanosleep(&req, &rem);
  ```

Project Details/Requirements

This project will require you to modify the Project2.h and Project2.cpp files in the project source code. **You are not to modify any other file that is part of the simulator.** You can, however, add new files to the project to implement new classes as you see fit. Feel free to look at Robot.h, which defines Wheelbot, Simulator.h, which defines the environment, and Vector2D.h, which provides 2D vectors and points. We will test your project by copying your Project2.h and Project2.cpp files, as well as any files you have added to the project, into our simulation environment and running it.

Feel free to use the C++ STL. Additionally, if you have previously implemented data structures you wish to re-use, please do. However, you must not use anything that trivializes the problem. For instance, do not use a downloaded A* algorithm package. You must implement A* and the extensions yourself.

The provided Project2.h and Project2.cpp files include a skeleton implementation of the Project2 class, with the following functions:

1. *Project2(Simulator* sim1)*: This is the constructor for the class. Here, you should query the simulator for the dimensions of the map and all the initially known obstacles, and store this information (preferably by constructing the appropriate 2D grid). main.cpp will call the constructor before the simulation begins.

2. *RobotAction getOptimalAction(Simulator* sim1, Robot* r1)*: This is the function that will be called by main.cpp at each step of the simulation to determine the best action that Wheelbot should execute. In this function, you should first check Wheelbot's local obstacle sensors to discover any hidden obstacles around it and update your representation of the environment with the discovered obstacles (if any). Then, you should run an A* search on this (updated) environment to find a shortest path for Wheelbot and return the first action (MOVE_UP, MOVE_UP_RIGHT, etc.) it should execute to follow this path.

You should modify these two functions and implement an A* search (either as a new class, or part of the Project2 class). Your A* implementation should use the *Octile Distance* heuristic, which is the cost of an optimal path between two cells on an 8-neighbor grid, assuming that the grid has no obstacles. In Project 1, you might have observed that an optimal path between two locations on an 8-neighbor grid without obstacles can have moves in at most two directions: If the agent is not in the same row or column as its goal, it moves diagonally towards its goal, otherwise, it follows the straight line to its goal. This is exactly how the Octile distance is computed: by comparing the locations of two cells, (x_1, y_1) and (x_2, y_2), and computing the exact number of diagonal and cardinal moves necessary to go from one cell to the other. Specifically:

$$dx = |x_1 - x_2|$$

$$dy = |y_1 - y_2|$$

$$\texttt{number of diagonal moves} : \min(dx, dy)$$

$$\texttt{number of cardinal moves} : \max(dx, dy) - \min(dx, dy)$$

$$\texttt{Octile Distance} : 1.5 \times \min(dx, dy) + 1 \times (\max(dx, dy) - \min(dx, dy))^1$$

$$= dx + dy - 0.5 \times min(dx, dy)$$

For full credit, your robot must always follow a shortest path to the goal.

[1]Typically, the Octile distance is computed by assuming that diagonal movements cost $\sqrt{2}$, but for this project, we assume that they cost 1.5.

Part 2 - Experiments with Inflated Heuristics - 3 Points

We have seen different best-first search algorithms in class. Here is a quick list, along with the criteria for ordering the nodes in the fringe:

- Uniform-Cost Search: $f(s) = 1 \times g(s) + 0 \times h(s)$.
- A* Search: $f(s) = 1 \times g(s) + 1 \times h(s)$.
- Greedy Best-First Search: $f(s) = 0 \times g(s) + 1 \times h(s)$. Observe that, if the h-values are greater than 0 except for the goal state(s), we can rewrite this as $f(s) = 1 \times g(s) + C \times h(s)$, for a sufficiently large C, without changing the ordering for the nodes (since h-values will be the dominant factor in both cases).

With the small modification to Greedy Best-First Search, observe that all these algorithms differ in only how much weight they give to the h-values. In other words, all these algorithms use $f(s) = g(s) + w \times h(s)$ to order nodes, with different values of w ($w = 0$ for Uniform-Cost Search, $w = 1$ for A*, $w \to \infty$ for Greedy Best-First Search). The version of A* that uses $f(s) = g(s) + w \times h(s)$ to order nodes is called Weighted A*. For this project, we assume that the heuristic function that Weighted A* uses is consistent (before inflation).

In this part of the project, you will run experiments and compare the number of expansions and the resulting path lengths for different values of w. To do that, you will need to modify the code you have developed for the first part of the project as follows:

1. Make a new copy of your project (you do not need to submit the modified code).
2. Your A* implementation should now use $f(s) = g(s) + w \times h(s)$ to order nodes in the fringe (w can be a parameter for the algorithm).
3. Your A* implementation should now keep track of the number of expansions as well as the resulting path length.
4. We are only concerned with the number of expansions and the resulting path length of the first search (before the robot makes its first move). You can modify your code so that it prints out the relevant statistics after the first search either to a file or to the terminal. If you do decide to print the statistics to the terminal, the simulator will overwrite them when the robot moves. You can avoid this by simply adding a 'cin' command before the best move is returned, so that the robot does not move until you enter a word to the terminal.

After the proper modifications, run your code for the following values of w: 0, 1, 2, 3, 10. For each value of w, report the number of expansions and the resulting path length. Discuss your results, specifically:

- What trends do you observe in the number of expansions and resulting path lengths as w varies?
- For which values of w is Weighted A* guaranteed to find a shortest path? For which values of w is Weighted A* not guaranteed to find a shortest path (and why)?
- What can you say about the length of the resulting path as w increases? Why do you think that this is the case?
- What can you say about number of expansions as w increases? Why do you think that this is the case?

Submission

You should submit a zip archive of your modified source code, named 'Part1.zip', and a PDF file for your solution to Part 2, named 'Part2.pdf' to blackboard by Wednesday, March 2nd, 11:59pm. If you have any questions about the project you can post them on piazza (you can find the link on the course wiki) or come to the TAs' office hours.