CSCI 360 – Project #3 - 15 Points

Part 1 - Navigating a Faulty Wheelbot Using MDPs - 10 Points

Our beloved Wheelbot has developed a fault and its movement has become erratic: It can no longer make diagonal moves (meaning that it is now operating on a 4-neighbor grid) and, when trying to move in a direction, there is a chance that it ends up moving in a different direction (meaning that its movement is now non-deterministic; the specifics of this non-determinism are discussed later). This fault makes it dangerous for Wheelbot to move close to obstacles because it can accidentally crash into them due to its non-deterministic movements. Lucky for Wheelbot, all the obstacles in the environment are known to it (there are no hidden obstacles this time), so it has all the necessary information to plan a short, yet safe path to a nearby repair station.

In this part of the project, you are required to model the problem described above as an MDP and use value iteration to determine the optimal policy for this MDP.

Modeling the Problem as an MDP

We now describe how you should model this problem as an MDP.

- Your MDP should have a state for each cell, whether it is a blocked cell, unblocked cell, or the target location.
- States corresponding to blocked cells or the target location are goal states. That is, there are no actions that can be executed in these states.
- There are four actions that can be executed in a non-goal state: Wheelbot can try to move up, down, left, or right. When Wheelbot tries to move in a direction, there is only a 50% probability that it succeeds. With 10% probability, it moves in the opposite direction, and, with 20% probability each, it moves in an orthogonal direction. For instance, if Wheelbot tries to move right, it moves right 50% of the time, left 10% of the time, up 20% of the time, and down 20% of the time.
- Each action has a base cost of 1. If an action results in a state that corresponds to a blocked cell, then there is an additional cost of 100, for a total cost of 101. If the action results in the state that corresponds to the target location, then there is no additional cost.

Determining the Optimal Policy

You will use undiscounted value iteration to calculate the optimal policies. Initialize each state's value to 0 and iterate until none of the states' values are changing by more than 0.001 during an iteration. Remember that you are **minimizing costs** when picking the best action to execute at a state.

Simulator Class

We have modified the code for the Simulator class, we no longer have the Robot class, and we have changed the Point2D class to store a pair of integers, rather than a pair of floats. Below is the list of functions in the Simulator class that you are allowed to call:

- GetHeight() and GetWidth() return the dimensions of the map as integers.
- GetObstacleLocations() returns a Point2D vector, enumerating the locations of all obstacles on the map.
- GetTarget() returns the target location as a Point2D.

Since Wheelbot's actions are non-deterministic, even if Wheelbot is following an optimal policy, it can still crash into an obstacle before reaching its target. We still have a simulation, but Wheelbot's performance in the simulation will not be used to grade this project (see the Project3 Class section for more details). We have also modified the simulation so that, once a simulation is complete, you can hit 'enter' to start a new simulation, without having to run your program again.

Project3 Class

In this project, you will modify the Project3.h and Project3.cpp files in the provided 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. We will test your project by copying your Project3.h and Project3.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 library **data structures** and **containers**. Additionally, if you have previously implemented data structures you wish to re-use, please do.

The provided Project3.h and Project3.cpp files include a skeleton implementation of the Project3 class, with the following functions:

- Project3(Simulator* sim1): This is the constructor for the class. Here, you should: (1) Query the simulator for the dimensions of the map, obstacle locations, and the target location, and store this information (preferably by constructing an appropriate 2D grid). (2) Construct the MDP. (3) Calculate the optimal policy, store the value of each state, and store the policy.

  main.cpp will call the constructor before the simulation begins.

- getOptimalAction(Point2D loc): This is the function that will be called by main.cpp at each step of the simulation to determine the action that Wheelbot should execute, given that it occupies the cell 'loc'. It should return NO_ACTION if 'loc' is the target location or a blocked cell. Otherwise, it should return MOVE_UP, MOVE_DOWN, MOVE_LEFT, or MOVE_RIGHT.
- getValue(Point2D loc): This function should return the value of the state that corresponds to the cell 'loc'. We will use this function and the function getOptimalAction(Point2D loc) to grade your projects.

Part 2 - Resolution with Propositional Logic - 5 Points

For Problem 2 (page 12) from Section 7 of `http://modelai.gettysburg.edu/2011/clue/clue.pdf`, express all relevant problem facts as a knowledge base in propositional logic. Clearly explain the meaning of your propositional symbols. Use resolution theorem proving to solve the problem by hand by deriving false (the empty clause). We suggest that you read through Sections 2-4, as they will help you to understand reasoning via resolution.

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, April 20th, 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.