CS B551: Homework 6

This assignment is due on 11/29, by 5:00pm. Submit your policy.py, agent.py, and hw6_answers.txt files on Oncourse.

Multi-agent Planning With Limited Sensing

In HW5 and HW6 you will be programming agents that interact with other agents on a grid. They also have limited sensors that will require intelligent perception techniques. Over the course of two assignments you will program these agents to perceive their environments and reach a goal position while avoiding other agents. At the end of the course, your agent will compete with the agents of all other students in a tournament.

In HW6 your job is to implement the agents' **policy**. A complete perception system is provided for you, but you may modify it if you wish.

Running the Program

To run the GUI, the Tkinter library be installed in your Python distribution. Most major Python distributions do include Tkinter. Run the GUI with the command 'python driver.py'.

The right hand side depicts the agents moving around the environment as triangles. You can choose between several example scenarios using the drop down menu on the left. The 'Step' button performs one step of the agent simulation. 'Start' animates multiple steps of the simulation. 'Reset' resets the simulation to the initial state. The 'belief' drop-down and check boxes allow grid cells are shaded according to an agent's beliefs about goals and/or objects (Green=goal distribution, Red=object distribution). [*Note: this belief is updated before each action is taken, so the belief that is shown is the agent's belief just before arriving at the current state.*]

Code Structure

The program consists of the following files:

- **agent.py**: The bulk of the code is contained here. Contains structures defining actions, states, transition models, sensor models, beliefs, and basic agent policies. (You may modify this file if you wish.)
- **distribution.py**: Subroutines for creating, querying, and manipulating probability distributions. (You will not need to modify this file.)
- **gridmap.py**: Basic code for defining a map. (You will not need to understand or modify this file.)
- **multiagent.py**: A multi-agent simulator. (You will not need to understand or modify this file.)
- **policies.py**: Prototypes for agent policies.
- **reward.py**: Defines the reward function for the agents.
- **scenarios.py**: A set of scenarios for testing your agents. You may wish to add new scenarios to debug your policies.
- **driver.py**: The GUI driver program. You probably will not need to edit this file.

Agent Environment

State/Actions. N agents move about on a grid. The grid contains static obstacles (blocked off squares), and no two agents can occupy the same square. Each agent has one of 8 directions (any of the 4 primary directions plus 4 diagonals), and has **5 available actions**: stay still, move forward, turn left, turn right, and move left +forward, and move right+forward. Each agent has a **goal square** G that it tries to reach. However, the **location of the goal is not known**, and instead the agent must look for it using its sensors..

Sensors. For each agent, the N-1 other agents are considered as moving **objects** $O_1,\ldots,O_{N-1}$. The agent's own pose $Q_A=(x,y,d)$ is observable where x,y are the coordinates of the grid cell and d is the agent's direction. Each agent does not precisely sense its goal G or the position/orientation of $O_1,\ldots,O_{N-1}$. Instead, it only contains the following sensors:

- A **visual sensor** that points in the forward direction with a 90 field of view. This sensor reports whether the goal is seen (a percept V_G), and whether each object is seen (percepts $V_{O1},\ldots,V_{ON-1}$). The visual sensor can report '0' if the object is not seen, 'L' if the object is in the left half of the field of view, and 'R' if the object is in the right half of the field of view. Occlusions are ignored – the sensor can "see" through walls and other objects.
- A **proximity detector** that detects nearby objects. This sensor reports whether an object is within 2 units or less (binary percepts $P_{O1},\ldots,P_{ON-1}$).

These sensor models assumes that all objects are exactly associated to each percept (e.g., each agent has a unique color or markings). The goal sensor is very accurate, so V_G reports the visibility of the goal with 0% error. The visual object sensor reports the incorrect value of V_{Oi} 5% of the time, and the proximity object sensor reports the incorrect value of P_{Oi} 10% of the time.

Utilities. Every step that the agent is not at its goal incurs a cost of 1. If an agent hits a wall, it incurs a moderate cost (10). If an agent hits another agent, it incurs a large cost (20). No reward or cost is accumulated if the agent is on its goal.

Multiple agents. Each of the N agents is moved in round-robin fashion. When an agent takes a turn, it senses the current positions of other agents, computes an action, and executes the action all at once. (This simplifies the problem because an agent doesn't need to worry about outdated percepts or multiple agents moving into the same square)

Agent Architecture

Sensing. Each agent in this problem maintains a probability distribution over goals and object states that gets updated after every step (this is performed in the `Agent.sense` method). This is known as the agent's **belief state**. In HW5 you will be implementing the subroutines that are needed to update the belief state properly following an observation (the `Agent.update_belief` method).

Acting. After sensing, the agent's `Agent.act` method is called. The act function simply evaluates the **agent's policy** on the current belief state. You will be implementing and testing different policies in this assignment.

Representing and updating belief states. An agent's belief state B(X) is a distribution over possible states $X=(Q_A,G,Q_{O1},\ldots,Q_{ON-1})$, where the (x,y,d) pose of the agent is denoted Q_A and the poses of other objects are denoted $Q_{O1},\ldots,Q_{ON-1}$. Each belief state B is represented in **factored form** – that is, the probability distribution B is the product of individual distributions: $B(X) = I[Q_A] \times B_G(G) \times B_{O1}(Q_{O1}) \times \ldots \times B_{ON-1}(Q_{ON-1})$. Here, I is the indicator function, B_G is a distribution over goal positions G, and each B_{Oi} is a distribution over object pose Q_{Oi}. Letting the superscript t denote time, the belief update computes the distribution $B^{t+1} = P(X^{t+1}|S^t,B^t)$

The agent has sensor and transition model for goals and each object. The agent's sensor and transition model for goals is correct: $P(S_G^t|G^t,Q_A^t)$ is deterministic, and goals don't move so the transition model $P(G^{t+1}|G^t)$ is just the identity function. The agent's object sensor model $P(S_{Oi}^t|Q_{Oi}^t,Q_A^t)$ is correct – in other words, it has the correct probabilities that describe the behavior of the real sensor. On the other hand, *its transition model* $P(Q_{Oi}^{t+1}|Q_{Oi}^t)$*is just an approximation*. This approximation is needed because it is difficult to model how intelligent agents make decisions (in fact, it is non-Markovian, particularly with interactions that involve history). Instead, the transition model simply assumes that each object selects from one available action uniformly at random at every step.

Questions

1. Implement the `AgentGoalPursuingPolicy` so that the agent always moves toward the closest goal position that is consistent with its belief. Part of this method is provided for you. To start, use the `search_path` function found in gridmap.py. But `search_path` returns a path of 2D grid cells, not poses, and this path does not necessarily respect the agent's steering constraints. Your policy should choose the agent's actions to steer it along the chosen path.

   Test your policy on 'Goal Seek Scenario' 1-3 and 'Maze Scenario 1'.

   Written questions:
   a) Assuming your agent had perfect knowledge about the goal position, how would you implement an *optimal* agent?
   b) In 'Goal Seek Scenario 1' the policy is given a small probability (5%) of choosing a random action. In 'Goal Seek Scenario 2', the policy performs goal seeking 100% of the time. What differences in behavior do you observe? Describe at least one possible method for solving 'Goal Seek Scenario 2' in a more efficient way than occasionally choosing a random action. (You may wish to test such a method on these scenarios in preparation for Question 3).

2. The `AgentCollisionAvoidingPolicy` is supposed to implement a *courteous* agent that moves away from nearby objects. The provided template code selects the action that minimizes the value of a potential field function (a local search). This potential field is higher near other objects, so the agent will try to steer around them when possible. But the current implementation is not very effective. If an object is in front of the agent, the agent cannot immediately move backward, and turning left or right will not reduce the potential field value.

(In other words, the potential function has many plateaus).

a) Evaluate the quality of the existing policy on the collision avoidance scenarios 'Avoidance Scenario' 1-4. In scenarios 1 and 2, your agent is getting out of the way of open-loop agents. In scenario 3, your agent must avoid a "suicidal" agent. In scenario 4, your agent avoids several suicidal agents and open loop agents

b) Implement an improved policy. We suggest two possible strategies: perform a deeper search in the state space, or search on 2D grid positions and then steer toward the best grid cell.

Written questions:

a) Specify an appropriate performance metric for a courteous agent.
b) Describe your improved policy. What design decisions did you make, and why?
c) Perform experiments to how your improved policy performs in simulation, and compare your results to the original policy. State your results in terms of the performance metric from part a).

3. In `AgentStudentPolicy`, implement an agent that performs both goal seeking and collision avoidance. This will be the policy that will be tested in the tournament.

Written questions:

a) Describe your strategy for integrating goal seeking and collision avoidance. How do you trade off between these competing demands?
b) Describe your strategy for handling uncertainty in goal and object positions.
c) Use the simulator to evaluate how your custom strategy performs compared to your answers in Questions 3 and 4 on the 'Goal Seeking…', 'Maze …', 'Avoidance…', and 'Hallway…' scenarios. You may need to edit the constructors in scenarios.py in order to instantiate your `AgentStudentPolicy` class in place of `AgentGoalPursuingPolicy` and `AgentObstacleAvoidingPolicy`.