CS B551: Homework 8

This assignment is due on 12/8, by 5:15pm. Submit your agent.py program on Oncourse, and bring your written answers to class or slip them under the door of Informatics East, Room 257.

Multi-agent Planning With Limited Sensing

In this assignment you will be programming agents that have limited sensors and that interact with other agents. The job of each agent is to reach a goal position while avoiding other agents. A partially-observable Markov decision process framework, including a sensor, transition, and reward model, are provided to you. Your assignment is to implement subroutines for belief updates and heuristic strategies for behaving in this environment.

At the end of class, we will hold a tournament in which your agent, running this policy, will compete with the agents of all other students. The tournament will test your agent alone and with one or more students in navigation, obstacle avoidance, and coordination tasks, and the agent that accumulates the highest average utility will be declared the winner. As long as all of your code stays in agent.py, you are also free to modify the agent transition models, belief update functions, and anything else that you need in order to build a good agent. Extra credit will be awarded to the top 3 agents.

Running the Program

To run the GUI, the Tkinter library must 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, until the simulation ends or you press the 'Pause' button. 'Reset' resets the simulation to the initial state. The 'Agent View' drop-down allows you to change which agent's beliefs and goals you are viewing. The check boxes allow you to change what information is displayed on the grid. 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 agent policies. Your code should be placed into this file.
- **distribution.py**: Subroutines for creating, querying, and manipulating probability distributions. The `normalize` subroutine should be useful in your project.
- **gridmap.py**: Basic code for defining a map. A search subroutine (`search_path`) is provided for use in your project.

- **multiagent.py**: A multi-agent simulator. You will not need to edit this file.
- **scenarios.py**: A set of scenarios for testing your agents. In question 4 you may need to edit this file in order to test new scenarios.
- **driver.py**: The GUI driver program. You will not probably 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 a **visual sensor** that points in the forward direction and a **proximity detector** that detects nearby objects. The visual sensor has a 90 field of view and reports only whether the goal is seen (a binary percept V_G), and whether each object is seen (binary percepts $V_{O1},\ldots,V_{ON-1}$). Occlusions are ignored – the sensor can "see" through walls and other objects. The proximity detector reports whether each object. The proximity detector 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 costs 1 unit of utility. If an agent hits a wall, it incurs a moderate cost (10). If an agent hits another agent, it incurs a large cost (100). 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 Question 1 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 Questions 2-4.

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: $B(X) = I[Q_A]xB_G(G)xB_{O1}(Q_{O1})x\ldots xB_{ON-1}(Q_{ON-1})$. 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 $B^{t+1} = P(X^{t+1}|S^t,B^t)$, with S the sensor reading.

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 – but 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 agents interact (in fact, it is non-Markovian). Instead, the transition model simply assumes that each object selects from one available action uniformly at random at every step.

Questions

1. The `Agent.update_belief` method calls two subroutines for updating goal and object beliefs. But the implementations of these subroutines are missing.

    a) Implement `Agent.update_goal_belief`. This requires updating the set of goal positions consistent with the observation at the current state. [*Hint: consult the algorithm of slide 14 of class 22.*]

    b) Implement `Agent.update_object_belief`. This requires two steps: predict and update. In the predict step, compute the belief on the object's position before receiving the observation. In the update step, compute the posterior distribution of object positions given the observation. [*Hint: consult slides 9 and 18 of class 12.*]

    To test whether you are updating the belief properly, you may use the 'Goal Update Test' and 'Object Update Test' scenarios in the GUI.

    Written questions:
    a) Why does the uncertainty on object positions often grow, while the uncertainty on goal positions decreases?
    b) Does the factored belief state representation perfectly represent the true Bayesian posterior of the belief state, given the observation? Why or why not? Does your answer depend on the number of obstacles N?

2. Implement the `AgentGoalPursuingPolicy` so that the agent always moves toward a 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 implementation should then return actions to steer the agent along the chosen path.

Written questions:
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 in preparation for Question 4).

3. The `AgentObstacleAvoidingPolicy` is supposed to move the agent away from nearby obstacles. 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 obstacles, so the agent will try to steer around them when possible. But the current implementation is not very effective. If an obstacle 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 two obstacle avoidance scenarios 'Avoidance Scenario 1' and 'Avoidance Scenario 2'.

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

    Written questions:
    a) Describe your improved policy. What design decisions did you make, and why?
    b) Describe how your improved policy performs in simulation, compared to the original policy. Report how many times the "dumb" agents (B and C, respectively) collide with your agents over each run (each run is 100 steps).

4. In `AgentStudentPolicy`, implement an agent that performs both goal seeking and obstacle avoidance. (This will be the policy that will be tested in the tournament.)

    Written questions:
    a) Describe your strategy for integrating goal seeking and obstacle 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 'Avoidance…', 'Goal Seeking…', 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`.