CS B551: Homework 5

This assignment is due on 11/8, by 5:00pm. Submit your agent.py program and hw5_answers.txt 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 HW5 your job is to implement the agents' **perception system**. (You will not necessarily use your results for HW5 in HW6. Even if you do not complete this assignment, a complete perception system will be provided for you in HW6.)

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. Your HW5 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 assignment.
- **gridmap.py**: Basic code for defining a map. (You will not need to understand or modify this file for HW5.)
- **multiagent.py**: A multi-agent simulator. (You will not need to understand or modify this file.)
- **policies.py**: Prototypes for agent policies. (You will not need to understand or modify this file for HW5.)
- **reward.py**: Defines the reward function for the agents. (You will not need to understand or modify this file for HW5.)

- **scenarios.py**: A set of scenarios for testing your agents. You may wish to add new scenarios to debug your sensing system.
- **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.

For HW5 the paths of the agents will be fixed.

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.

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).

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. The `Agent.predict_belief` and `Agent.update_belief` methods are used in filtering (i.e., state estimation). These methods call three subroutines for updating goal beliefs, predicting object beliefs, and updating 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 20.*]

    b) Implement `Agent.predict_object_belief`. This requires computing the belief on the object's next position before receiving the observation. [*Hint: consult slides 11,19-20 of class 20.*]

    c) Implement `Agent.update_object_belief`. This requires computing the posterior distribution of object positions given the observation. [*Hint: consult slides 19-20 of class 20.*]

    To test whether you are updating the belief properly, you may use the 'Goal Update Test' and 'Object Update Test' scenarios in the GUI. The "Give omniscient" button immediately provides the agent with the current exact state of the objects and goal, which should help you implement question b.

    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. Now consider the problem of estimating an object's most likely trajectory after the agent has observed a sequence of observations. When the "Most likely explanations" GUI checkbox is checked, the `agent.most_likely_trajectory` method is called to estimate a path for a moving object $Q_k^1,\ldots, Q_k^T$ from robot poses and sensor input $(Q_A^1,S_{Ok}^1),\ldots, (Q_A^T,S_{Ok}^T)$. The method currently only computes the sequence of states that maximizes the observation probabilities, without considering the transition probabilities. This is a problem because it can generate nonsensical estimates.

   Implement the Viterbi algorithm to correctly consider the transition probabilities when computing the most likely sequence [*Hint: consult slides 29-31 of class 20*].

   Written questions.

   a) Consider the sequence of states S_F estimated by performing filtering, and then producing the most probable state estimates at each time step. Let S_V be the sequence estimated by the Viterbi algorithm for the same observation sequence. Why might S_F and S_V differ? (You might want to add some temporary print statements to output S_F as you step through some examples)

   b) Suppose an agent's transition model takes into account how an object moves in accordance with its own pose. In other words, the transition model $P(Q_{Oi}^{t+1}|Q_{Oi}^t,Q_A)$ depends explicitly on Q_A. Does the implementation of `most_likely_trajectory` need to be changed? How about if the transition model depends on how the object reacts to other objects around it?