COMP3702 Artificial Intelligence (Semester 2, 2024)

Assignment 2: BeeBot MDP – **Report Template**

Name: Peter Huang

Student ID: 45367115

Student email: s4536711@uq.edu.au

Note: Please edit the name, student ID number and student email to reflect your identity and **do not modify the design or the layout in the assignment template**, including changing the paging. a2-COMP3702-[SID].pdf

**Question 1 MDP problem definition (15 marks)**

**a) Define the State space, Action space, Transition function, and Reward function components of the BeeBot MDP planning agent as well as where these are represented in your code.** **Ensure you complete both parts of the question: (10 marks)**

i) Define the four components in relation to BeeBot

ii) Identify where they are represented in your code

**State Space**:

The state space of an MDP represents all possible configurations of the environment. In the BeeBot problem, the state space is represented using the State class in state.py. The State class defines the configuration of the environment by initializing the various components (BeeBot's position and orientation, widget positions, etc.). Additionally, it ensures that states are initialized in a valid way (force\_valid=True). The BFS method, bfs\_find\_all\_valid\_states, is used to generate all possible states reachable from the initial state. States are updated dynamically as actions are taken and transitions occur.

**Action Space**:

The action space represents all the possible actions that BeeBot can take from a given state. In the hexagonal grid, BeeBot can perform the following actions: (FORWARD, REVERSE, SPIN\_LEFT, SPIN\_RIGHT). Actions are represented in the constants.py file as BEE\_ACTIONS, and each action is referenced in the Solver class:

**Transition Function**:

The transition function defines the probability of reaching a new state s' given the current state s and action a. In the BeeBot environment, In the BeeBot environment, the transition function incorporates action dynamics to account for both deterministic and probabilistic outcomes:

Deterministic outcomes: BeeBot may perform a forward, reverse, or spin action as intended.

Probabilistic outcomes: BeeBot’s movement may involve drifting or double moves, which add randomness to the transition. For example, when moving forward, BeeBot may end up drifting clockwise or counterclockwise, or performing a double move.

The transition function, implemented in the get\_transition\_outcomes method, calculates the probability of various outcomes for a given action, using both the action dynamics and the environment’s conditions (e.g., thorns, widgets).

**Reward Function**:

The reward function assigns a reward value for each transition from a state s to a new state s' after performing an action a. In the BeeBot environment, the reward includes as thorn penalties or collision penalties. The reward is computed in the compute\_expected\_reward function by considering the outcomes of each action and incorporating the associated costs and penalties. Action dynamics, like pushing a widget or drifting into thorns will directly impact the reward by modifying the expected value of the reward for a state-action pair. **Policy Iteration** explicitly uses the reward function through self.r\_model, as the reward model needs to be updated during the policy evaluation phase. In contrast, **Value Iteration** implicitly uses the rewards within the value update step, as it directly computes the Q-value for each state-action pair in every iteration.

**b) Describe the purpose of a discount factor in MDPs. (2.5 marks)**

The discount factor (𝛾) in a Markov Decision Process (MDP) is a value between 0 and 1 that determines the relative importance of future rewards compared to immediate rewards. If 𝛾 is close to 1, future rewards are given as much importance as immediate rewards, encouraging the agent to plan for long-term gains. Conversely, if 𝛾 is closer to 0, the agent focuses more on immediate rewards and is less concerned with future outcomes. The discount factor ensures that the cumulative value is finite and well-defined. Without a discount factor, the sum of future values could grow indefinitely in environments with infinite horizons, making it harder to compute optimal policies.

In the BeeBot MDP problem, the discount factor is referenced as self.environment.gamma in the provided environment.py file, which is set when the environment is initialized. The purpose of the high discount factor (0.999) for all 6 tests in the BeeBot problem is to: encourages BeeBot to consider the long-term effects of its actions, leading to more strategic planning that balances immediate movement costs with long-term goals. By considering the long-term rewards or penalties, BeeBot can avoid unnecessary penalties, such as thorn penalties, collision penalties, and movement costs, by planning for future states effectively.

**(c) State and briefly justify what the following dimensions of complexity are for the BeeBot MDP agent: (2.5 marks)**

|  |  |  |
| --- | --- | --- |
| **Dimension** | **Value** | **Justification** |
| Planning horizon | Indefinite | BeeBot plans continuously without a predefined number of stages, but the planning ends when a terminal state is reached. |
| Sensing uncertainty | Fully observable | BeeBot has complete knowledge of the environment at all times, including its position, orientation, the locations of widgets, thorns, and obstacles. There is no ambiguity in the agent's perception of the environment. |
| Effect uncertainty | Stochastic | The BeeBot environment incorporates probabilistic dynamics. When BeeBot takes an action, such as moving forward, it may not always result in the intended movement due to the presence of drift, or double move. This stochastic nature is modeled in the transition function, where BeeBot's actions have probabilistic outcomes based on the environment's dynamics. |
| Computational limits | Perfect rationality | There are no computational constraints incorporated into the algorithms, such as Value Iteration (VI) or Policy Iteration (PI). The BeeBot evaluates all possible states and actions without regard for time or memory limitations. |
| Learning | Knowledge is given | The BeeBot agent operates with a fully defined model of its environment and does not need to learn from data or past experiences. The environment's states, actions, and transition dynamics are explicitly modeled and provided in the code. Thus, BeeBot does not engage in any learning process; all knowledge is provided by the environment's design. |

Learning Refer to the P&M textbook <https://artint.info/3e/html/ArtInt3e.Ch1.S5.html> for the possible values and definitions of each dimension, and format your answer in a neatly formatted table with the following column headings: [Dimension, Value, Justification].

**Question 2 Comparison of algorithms and optimisations (15 marks)**

**a) Describe your implementations of Value Iteration and Policy Iteration in one sentence each. Include details such as whether you used asynchronous updates, and how you handled policy evaluation in PI. (2 marks)**

The implementation of Value Iteration utilizes synchronous updates, where in-place updates are employed instead of batch updates, Q-values are updated in-place using the latest available values, allowing future states within the same iteration to benefit from improved approximations and leading to faster convergence.

The implementation of Policy Iteration has two phases: policy evaluation, where the value function for the current policy is calculated by solving a system of linear equations, and policy improvement, where the policy is updated using a synchronous approach.

**b) Pick three representative testcases to compare the performance of VI and PI, reporting the numerical values for the following performance measures:**

* Time to converge to the solution. (3 marks)
* Number of iterations to converge to the solution. (3 marks)

In order to do this, you’ll need to modify the # solver type in your local copy of the test cases (the text files in the testcases directory). You should report the numerical results in a neatly formatted table.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Algorithm | Time to Converge (s) | Number of Iterations |
| Test 1 | Value Iteration | 3.99 | 120 |
| Policy Iteration | 0.83 | 11 |
| Test 2 | Value Iteration | 6.88 | 141 |
| Policy Iteration | 2.40 | 12 |
| Test 6 | Value Iteration | 95.24 | 332 |
| Policy Iteration | 87.59 | 22 |

**c) Discuss the difference between the numbers you found for VI and PI, including any differences observed between testcases. Explain and provide reasons for why the differences either make sense, or do not make sense. (7 marks)**

In all three test cases, Policy Iteration consistently converges faster than Value Iteration (VI). This is because PI directly evaluates the policy in each iteration, using a more structured policy evaluation approach, while VI iteratively updates values for all states. PI requires fewer iterations to converge because it improves the policy after evaluating it in each step.

PI converges in significantly fewer iterations compared to VI in all test cases. The difference is even more prominent in Test 6, where VI takes 332 iterations compared to only 22 for PI. This is because PI’s policy evaluation step refines the policy to optimality, reducing the total number of iterations required. VI, on the other hand, requires more iterations as it refines the value estimates for all states simultaneously without focusing directly on policy improvement in every iteration. PI does not use the max operator over actions during the value calculations, further reducing computational complexity. Solving Vπ​​(s) during policy evaluation involves solving a set of ∣S∣ linear equations with ∣S∣ unknowns, which is efficiently handled with linear algebra, speeding up the process substantially.

The time to converge for both algorithms increase significantly in Test 6 compared to Test 1 and Test 2. This is because Test 6 is a more complex environment with 9024 states, which leads to longer computation times for both VI and PI. However, the relative performance gap between VI and PI remains consistent, with PI continuing to outperform VI in terms of both time and iterations.

**Question 3 Investigating optimal policy variation (10 marks)**

One consideration in the solution of a Markov Decision Process (i.e. the optimal policy) is the trade off between a risky higher reward vs a lower risk lower reward, which depends on the probabilities of non-deterministic dynamics of the environment and the rewards associated with certain states and actions.

Consider testcase ex6.txt, which includes a risky (but lower cost) path through the top half of the grid, and a less risky (but higher cost) path through the bottom half of the grid. Explore how the policy of the agent changes with thorn penalty and transition probabilities.

**a) Describe how you expect the optimal path to change as the thorn penalty and transition probabilities change. Use facts about the algorithms and Bellman optimality equation to justify why you expect these changes to have such effects. (5 marks)**

The Bellman equation describes the condition for an optimal policy in a Markov Decision Process (MDP), where the value V(s) of a state s is the maximum expected reward achievable by taking the best possible action from that state. The equation is recursive, considering the transition probabilities P(s′∣s,a) of reaching a new state s′ after taking action a, the immediate reward R(s,a,s′), and the discounted future rewards through the discount factor γ. This balance between immediate and future rewards allows the agent to determine the best action to take in each state, forming the foundation of algorithms like Value Iteration and Policy Iteration for optimal decision-making in environments like BeeBot's. When the thorn penalty is high, the agent is expected to avoid paths through the risky areas (the top half of the grid), even if the immediate cost of moving through the bottom half is higher. A higher thorn penalty will reduce the value of states along the risky path, discouraging the agent from taking that route. The algorithm will calculate lower expected returns for actions leading to thorns, resulting in the agent favoring safer paths with higher cumulative rewards. As the transition uncertainty P(s′∣s,a) increases, the agent is more likely to choose the less risky path. This is because it is harder to predict the outcome of actions in the risky upper path, where the chance of hitting a thorn is amplified by the stochasticity of movement. The value of actions along the upper path will decrease, encouraging the agent to favor the lower, more predictable path, even though it incurs higher immediate costs.

**b) Picking three suitable values for thorn penalty and three suitable values for the transition probabilities, explore how the optimal policy changes over the 9 combinations of these factors. Present the results in a table, clearly denoting the optimal behaviour (i.e. top/risky, bottom/safe, something else) for each combination. Do the experimental results align with what you thought should happen? If not, why? (5 marks)**

|  |  |  |  |
| --- | --- | --- | --- |
| Thorn Penalty \ Transition Probabilities | Low Uncertainty  0.0, 0.0, 0.0, 0.0  0.0, 0.0, 0.0, 0.0  0.0, 0.0, 0.0, 0.0 | Medium Uncertainty  0.25, 0.10, 0.0, 0.0  0.05, 0.025, 0.0, 0.0  0.05, 0.025, 0.0, 0.0 | High Uncertainty  0.4, 0.10, 0.0, 0.0  0.1, 0.025, 0.0, 0.0  0.1, 0.025, 0.0, 0.0 |
| Low (1.0) | Top (Risky) | Top (Risky) | Bottom (Safe) |
| Medium (10.0) | Top (Risky) | Top (Risky) | Bottom (Safe) |
| High (1000.0) | Top (Risky) | Top (Risky) | Bottom (Safe) |

The experimental results do not align with expectations based on the Bellman optimality equation and general MDP principles. The BeeBot prefers the top (risky) path even when the thorn penalty is high, it suggests that the model’s handling of the reward function or transition dynamics may not be sufficiently preventing the agent from taking risks. The experimental results do align with expectations when uncertainty increases (higher probabilities of drifting or double moves), BeeBot tends to choose the safer path to avoid unintentional collisions or thorn penalties.

**Appendix/References** (Include references and usage of Generative AI on page 5)

COMP3702 teaching team (2024) grid\_world\_solution\_PI.py