COMP3702 Artificial Intelligence (Semester 2, 2024)
Assignment 2: BEEBOT MDP – **Report Template**

Name: Harsha Joshi

Student ID: 45561623

Student email: s4556162@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.

Question 1 (Complete your full answer to Question 1 on pages 1 and 2, and keep page 2 blank if you do not need it)

a) The following table defines the four components in relation to BeeBot (i) and identifies where they are represented in the code (ii). Note the code indentation may be inaccurate to save spacing in the document.

State Space	**(i)**	The state space represents all configurations that BeeBot can be in. For BeeBot, the state space includes its position on the grid and any relevant environmental factors like thorns or goals.

	(ii)	`def initialise_states(self):`
		`frontier = deque([self.environment.get_init_state()])`
		`visited = set()`
		`while len(frontier) > 0:`
		`node = frontier.popleft()`
		`for action in BEE_ACTIONS:`
		`next_state = self.environment.apply_dynamics(node, action)[1]`
		`if next_state not in visited:`
		`frontier.append(next_state)`
		`visited.add(next_state)`
		`return {state: 0 for state in visited}`

Action Space	**(i)**	The action space defines the set of all possible actions BeeBot can take from any given state. These actions include moving in the following directions: forward, reverse, spin left, spin right. The last two are regarding the bee slipping.
	(ii)	`BEE_ACTIONS = [FORWARD, REVERSE, SPIN_LEFT, SPIN_RIGHT]`
Transition Function	**(i)**	The transition function determines the probability of moving from one state to another after taking an action. In BeeBot's case, this includes non-deterministic outcomes where BeeBot may not always move to the intended position due it's environmental uncertainties (e.g., slippage in movement).

	(ii)	`probability_nodriftcwccw_nodouble_move = (1-self.environment.drift_cw_probs[action]-self.environment.drift_ccw_probs[action])*(1- self.environment.double_move_probs[action])`
		`probability_nodriftcwccw_double_move = (1-self.environment.drift_cw_probs[action] - self.environment.drift_ccw_probs[action]) * (self.environment.double_move_probs[action])`
		`probability_driftcw_nodouble_move = (self.environment.drift_cw_probs[action]) * (1 - self.environment.double_move_probs[action])`
		`probability_driftcw_double_move = (self.environment.drift_cw_probs[action]) * (self.environment.double_move_probs[action])`
		`probability_driftccw_nodouble_move = (self.environment.drift_ccw_probs[action]) * (1 - self.environment.double_move_probs[action])`
		`probability_driftccw_double_move = (self.environment.drift_ccw_probs[action]) * (self.environment.double_move_probs[action])`
		`all_possible_probabilities_per_state_list = [(probability_nodriftcwccw_nodouble_move, [action]), (probability_nodriftcwccw_double_move, [action, action]),`
		`(probability_driftcw_nodouble_move, [SPIN_RIGHT, action]), (probability_driftcw_double_move, [SPIN_RIGHT, action, action]),`
		`(probability_driftccw_nodouble_move, [SPIN_LEFT, action]), (probability_driftccw_double_move, [SPIN_LEFT, action, action])]`

Reward Function	**(i)**	The reward function assigns a value to each state-action pair, based on the outcome. BeeBot gains positive rewards for reaching the goal, negative rewards for encountering thorns, and potentially some neutral or minor negative reward for each step taken (to encourage efficiency).

	(ii)	`min_reward = 0`
		`for action in action_list:`
		`reward, new_state = self.environment.apply_dynamics(new_state, action)`
		`if reward < min_reward:`
		`min_reward = reward`

b) The discount factor (γ) in MDPs is "(a number between 0-1) is a clever way to scale down the rewards more and more after each step so that, the total sum remains bounded" (Intuitive Tutorials, 2020). This helps manage infinite horizon problems where rewards could theoretically accumulate indefinitely. A discount factor close to 1 (e.g., 0.99) implies that future rewards are valued nearly as much as immediate rewards, making the agent more forward-looking. A lower discount factor (e.g., 0.5) implies that the agent is more short-sighted and prefers immediate rewards.

Question 1 (Complete your full answer to Question 1 on pages 1 and 2, and keep page 2 blank if you do not need it)

(c) From Module 0, pg. 73 Lecture Notes:

Dimension	**Value**	**Justification**
Planning horizon	Infinite Stage	BeeBot continuously plans until it reaches the goal or fails, making the problem unbounded to time.
Sensing uncertainty	Fully Observable	BeeBot can observe all relevant aspects of its environment without ambiguity. It has full access to the current state, meaning no hidden information about the world.
Effect uncertainty	Stochastic	The outcomes of the BeeBot's actions are probabilistic, meaning there is some randomness involved in the environment's response. Even though the agent chooses actions, it may not always result in a predictable outcome (from environment.py)
Computational limits	Perfect Rationality	BeeBot is considered computationally capable of calculating optimal actions and can perfectly reason about its environment and future decisions without limitations.
Learning	Knowledge is Given	BeeBot does not need to learn from experience. It starts with full knowledge of the environment's structure, transitions, and rewards, meaning its focus is purely on planning rather than learning (environment.py).

Question 2 (Complete your full answer to Question 2 on page 3)

a) VI implementation uses asynchronous updates to update state values based on the Bellman equation, where for each state, the action is selected such that the expected utility is maximised by considering all possible next states, stopping when the maximum difference between the old and new state values falls below a convergence threshold. For PI, the function alternates between using policy evaluation (solving a system of linear equations to compute state values based on the current policy) and policy improvement (performing a one-step lookahead for each state to choose the optimal action), repeating until the policy converges to an optimal value.

b) The table below demonstrates three representative testcases for VI and PI each to compare the performance measures: time to converge to the solution (in seconds, rounded to 2 decimal places) and number of iterations to converge to the solution.

	Time (sec to 2 decimal places)	**Number of Iterations**
Testcase 1 – VI	24.14	133
Testcase 2 – VI	44.57	171
Testcase 3 – PI	6.03	14
Testcase 4 – VI	58.17	301
Testcase 5 – PI	5.67	21
Testcase 6 – PI	264.05	24

c) Performance wise, PI takes fewer iterations as compared to VI because it directly solves for state values under policy evaluation, rather than gradually refining estimates like VI. Hence, this results in PI converging faster in terms of the number of iterations. However, VI can take more iterations as it iterates over every state, updating values iteratively based on the Bellman equation (HuggingFace, 2024).

Though PI takes fewer iterations, it does not always guarantee faster runtime due to the cost of solving linear equations during policy evaluation. For example, Testcase 6 (PI) took much longer (264.05 seconds) compared to other PI test cases despite the low number of iterations (24). This suggests that in some cases, solving the linear equations for policy evaluation becomes computationally expensive, especially when the state space or complexity of transitions increases. VI took significantly more iterations but less time for most test cases, except for larger problem sizes like Testcase 4 (VI), where the time (58.17 seconds) was higher.

Going into the specifics, Testcase 1 (VI) and Testcase 2 (VI) have an average number of iterations and runtime, but as the complexity increases, Testcase 4 (VI) requires 301 iterations and significantly more time (58.17). This demonstrates that VI struggles with larger or more complex state spaces because it updates each state iteratively until convergence (Gupta, 2020). Testcase 3 (PI) and Testcase 5 (PI) demonstrate PI's efficiency in small to medium-sized problems with faster convergence (6.03 and 5.67 seconds respectively). However, Testcase 6 (PI) shows that for more complex problems, PI's policy evaluation step can become a bottleneck (Feldt et. al, 2024), leading to much higher runtimes, even though the number of iterations remains relatively low.

There are certain reasons for these differences, namely the nature of convergence and state space complexity. In terms of convergence, VI slowly refines value estimates over many iterations, while PI alternates between solving for values and improving the policy. This makes PI converge faster in terms of iterations with a trade-off in computational cost. Further, larger state spaces or more complex transitions (Testcase 6) increase the computation upon PI's policy evaluation function, which involves solving linear systems. In VI, the time increases linearly with more iterations though is still manageable depending on the size of the state space.

Overall, the differences with the testcases align with the nature of VI and PI. PI is more efficient in terms of iterations, but its policy evaluation step becomes a bottleneck for large problems, leading to long runtimes. VI spreads out computation over many iterations, which makes it slower to converge in terms of iterations but sometimes faster in runtime for certain test cases due to simpler updates for its subsequent iterations.

Question 3 (Complete your full answer to Question 3 on page 4)

a) The Markov Decision Process (MDP) optimal policy is determined by balancing expected rewards and transition probabilities. There are two factors that influence the policy:

- Thorn Penalty: represents the cost of passing through states with thorns. As the thorn penalty increases, the agent is more likely to avoid these states, even if they lead to a shorter or quicker path. A high thorn penalty makes the agent favour safer routes, prioritising states that are less costly, even if they take longer to reach the goal.
- Transition Probabilities: represent the uncertainty of outcomes after taking an action. Lower transition probabilities indicate higher uncertainty, making the agent less likely to choose risky paths where the outcomes are less predictable. When transition probabilities favour the risky path (i.e., higher probability of success), the agent is more likely to opt for the riskier but faster route.

Using the Bellman equation (HuggingFace, 2024), which accounts for expected rewards and transitions over time, it can be predicted that:

- Higher thorn penalty: The agent will likely avoid the risky path, preferring the safer, costlier bottom route.
- Higher transition probability for risky path: The agent will choose the risky, potentially lower-cost top path if the reward is high enough to justify the risk.
- Lower transition probability for risky path: The agent will likely opt for the safer bottom path since the risk of failure or a bad outcome becomes too great.

Thus, the balance of these factors will determine whether the agent prioritises riskier or safer paths.

b) 3 values for the thorn penalty and 3 values for the transition probabilities were chosen to demonstrate how optimal policy changes, from ex6.txt.

Thorn Penalty	Transition Probability		
	0.1 (Low Probability)	0.5 (Medium Probability)	0.9 (High Probability)
1 (Low Thorn Penalty)	Top (Risky)	Top (Risky)	Top (Risky)
5 (Medium Thorn Penalty)	Bottom (Safe)	Top (Risky)	Top (Risky)
10 (High Thorn Penalty)	Bottom (Safe)	Bottom (Safe)	Top (Risky)

- Low Thorn Penalty (1): Even with low transition probabilities (0.1), the low thorn cost makes the agent prefer the risky top path across all transition probabilities.
- Medium Thorn Penalty (5): The agent switches to the bottom (safe) path at lower transition probabilities (0.1) but sticks with the risky top path when there is more certainty (0.5 or 0.9).
- High Thorn Penalty (10): When the thorn penalty is high, the agent avoids the risky path at low and medium transition probabilities, but at high transition probability (0.9), the agent chooses the risky path despite the high penalty due to the greater likelihood of success.

These results generally align with expectations. The agent's optimal behaviour switches from preferring the risky top path when the thorn penalty is low and transition probabilities are high, to favouring the safe bottom path when the thorn penalty is high or when the transition probabilities are low. This reflects the influence of expected rewards and probabilities predicted by the Bellman equation.

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

Feldt, S., & Ristivojevic, Z. (2024). *Universal anomalous heat conduction and transport in one-dimensional systems*. arXiv. https://arxiv.org/abs/2405.04118

Gupta, A. (2020, April 25). Reinforcement learning: An easy introduction to value iteration. Towards Data Science. https://towardsdatascience.com/reinforcement-learning-an-easy-introduction-to-value-iteration-e4cfe0731fd5

Hugging Face. (2024). *The Bellman equation and optimality*. Hugging Face. https://huggingface.co/learn/deep-rl-course/en/unit2/bellman-equation

Intuitive Tutorials. (2020, November 15). *Discount factor in reinforcement learning*. Intuitive Tutorials. https://intuitivetutorial.com/2020/11/15/discount-factor/

Poole, D., & Mackworth, A. (2023). *Artificial intelligence foundations of computational agents: State space search*. 3rd Edition. Cambridge University Press. https://artint.info/3e/html/ArtInt3e.Ch1.S5.html

Code Sourced From: Tutorial 7 2024 Solutions (COMP3702)

Question 1: Module 0, pg. 73 Lecture Notes