COMP3702 Artificial Intelligence (Semester 2, 2024)

Assignment 1: Search in BeeBot – **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.

**Question 1** (5 marks) Define the following eight dimensions of complexity of BeeBot: planning horizon, representation, computational limits, learning, sensing uncertainty, effect uncertainty, number of agents, and interactivity. Justify your selection.

|  |  |  |
| --- | --- | --- |
| **Dimension** | **Value** | **Justification** |
| Planning horizon | Indefinite Stage | The exact number of steps required to reach the goal is not fixed and depends on the environment's configuration, which makes the planning horizon indefinite. The bot needs to plan for an indefinite number of stages until the goal is achieved. |
| Representation | States | In the BeeBot environment, the state representation includes the bot's position, orientation, and the positions of various obstacles and targets. The environment is discrete, and each unique configuration of these elements corresponds to a specific state. Therefore, the environment is best represented by states. |
| Computational limits | Bounded Rationality | While the BeeBot is designed to find an optimal path, it operates under computational constraints. It cannot explore every possible action in every state due to the exponential growth of possibilities. |
| Learning | Knowledge is Given | The Beebot operates with a predefined set of rules and heuristics. There’s no mechanism for the bot to learn or adapt its behavior from previous experiences within the same simulation. The bot uses the information given to it, such as the layout of the environment, to make decisions. |
| Sensing uncertainty | Fully Observable | The BeeBot operates in an environment where all the necessary information is available for decision-making. The positions of obstacles, targets, and the bot’s own state are fully observable at all times. There’s no uncertainty about the current state of the environment. |
| Effect uncertainty | Deterministic | The actions performed by BeeBot have deterministic outcomes. For example, moving forward by one unit will always result in the bot moving forward unless an obstacle blocks the path. There is no randomness in the effect of actions taken by the bot. |
| Number of agents | Single Agent | The BeeBot environment involves a single bot navigating the environment. Although there may be multiple entities (like obstacles and targets) within the environment, there is only one decision-making agent, which is BeeBot. |
| Interactivity | Offline | The planning and decision-making process in BeeBot occur before the bot starts moving. The bot follows a pre-determined path based on the information it had when the planning phase was completed. There is no real-time interaction during execution. |

**Question 2** (5 marks)Describe the components of your Agent Design for BeeBot. Specifically, define the Action Space, State Space, Transition Function and Utility/Cost Function generally, and what these components are for the BeeBot agent design problem. Refer to the methods and definitions in the support code to support your answer.

**Action Space** defines all possible actions an agent can take in a given state. Each action moves the agent from one state to another within the environment.

* In BeeBot, the **action space consists of four possible actions in the** **perform\_action method of the Environment class**, which defines how each action affects the state.
* The method first checks if the action is a spin (SPIN\_LEFT or SPIN\_RIGHT).
* For SPIN\_LEFT, the BeeBot’s orientation is updated by rotating 60 degrees counterclockwise. For example, if the BeeBot is facing BEE\_UP, spinning left changes its orientation to BEE\_UP\_LEFT.
* For SPIN\_RIGHT, the BeeBot’s orientation is updated by rotating 60 degrees clockwise.
* If the action is a movement (FORWARD or REVERSE), the method proceeds to calculate the BeeBot’s new position. The direction in which the BeeBot is facing is stored in forward\_direction, which is equivalent to state.BEE\_orient.
* For the FORWARD action, the BeeBot moves one cell in the direction it is currently facing.
* For the REVERSE action, the BeeBot moves one cell in the opposite direction to where it is facing. The reverse direction is determined by looking up the current orientation in a dictionary that maps each orientation to its opposite.

**State Space** is the set of all possible configurations of the environment that the agent might encounter. It represents the agent’s knowledge of its current situation.

* **The state space for BeeBot is encapsulated in the State class**, which provides methods to initialize, compare, and hash states, as well as store the current configuration of the BeeBot and widgets.
* BEE\_posit: The current position of the BeeBot on the grid, represented by a (row, col) tuple.
* BEE\_orient: The current orientation of the BeeBot, which can be one of the six possible orientations (up, down, up-left, up-right, down-left, down-right).
* widget\_centres: The positions of all widgets in the environment.
* widget\_orients: The orientations of all widgets.

**Transition Function** defines how the environment's state changes in response to an action, mapping a given state and action pair to a resulting state. This function determines the outcome of applying an action within the current context.

* In the BeeBot environment, **the transition function is implemented through the perform\_action method in the Environment class.** This function is deterministic, meaning that for any specific state and action combination, the resulting state will consistently be the same.
* The function takes the current state, which includes the BeeBot's position, orientation, and the positions and orientations of widgets, along with the intended action (such as moving forward, reversing, or spinning). Based on the action, the function computes the new state by adjusting the BeeBot's position or orientation and checks for any collisions or boundary violations. The function then returns a success flag, indicating whether the action was valid, the associated action cost, and the new state that reflects the updated configuration.

**Utility/Cost Function** assigns a numerical value to each possible outcome. In a cost-based approach, the agent seeks to minimize the total cost of reaching a goal.

* In BeeBot, the cost function is defined by the cumulative cost of all actions the BeeBot takes to reach the goal. Each action that the BeeBot performs has an associated base cost, as **defined in the ACTION\_BASE\_COST dictionary in the constants.py file**. Moving forward or backward incurs a cost of 1.0, while rotating left or right costs 0.1. Pushing a widget forward adds an extra cost of 0.8, making the total cost for that action 1.8.
* The objective for the BeeBot agent is to minimize the total cumulative cost from the initial state to the goal state, where all target cells are covered by widgets.
* The cumulative cost of actions, minimized by search algorithms implemented in **solve\_ucs and solve\_a\_star** **methods within the Solver class.**
* In Uniform Cost Search (UCS), the algorithm expands nodes in the state space based on the cumulative path cost, always selecting the node with the lowest cost for expansion. This ensures that the first solution found is the one with the minimum possible cost.
* In contrast, the A\* search algorithm uses both the cumulative cost (g(n)) and a heuristic function (h(n)) to estimate the total cost of reaching the goal.

**Question 3.** (15 marks) Compare the performance of Uniform Cost Search and A\* search in terms of the following statistics for each testcase:

**Comparison of Performance between UCS and A**\*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case** | **Search Algorithm** | **Nodes Visited** | **Nodes Expanded** | **Nodes on Frontier** | **Run Time (seconds)** |
| **1** | **UCS** | **5497** | **1759** | **34** | **0.038** |
| **1** | **A\*** | **3480** | **1102** | **164** | **0.026** |
| **2** | **UCS** | **82038** | **26706** | **471** | **0.638** |
| **2** | **A\*** | **65955** | **21390** | **1354** | **0.587** |
| **3** | **UCS** | **3219927** | **1050690** | **5171** | **43.80** |
| **3** | **A\*** | **2673885** | **872140** | **26833** | **42.70** |
| **4** | **UCS** | **6163510** | **1938828** | **77931** | **84.40** |
| **4** | **A\*** | **2815199** | **884703** | **84121** | **44.63** |
| **5** | **UCS** | **21426726** | **6827467** | **50110** | **319.43** |
| **5** | **A\*** | **20281870** | **6439770** | **85481** | **366.21** |

**Uniform Cost Search (UCS)** explores the least-cost path to each node by expanding nodes in the order of their cumulative cost from the start. Since UCS doesn't use a heuristic function, it guarantees optimality and completeness, but it can be slow, especially when the state space is large and complex. **A\*** combines the cost to reach a node (g(n)) with a heuristic estimate of the cost from that node to the goal (h(n)). The purpose of the heuristic function, h(n) directs the search toward the goal more efficiently than UCS, resulting in fewer node expansions and faster overall execution.

As expected, A\* visited and expanded fewer nodes than UCS. The heuristic allowed A\* to focus on the most promising nodes, thereby reducing unnecessary exploration. This efficiency was evident in test cases 1-4, where the scenarios had fewer complex environments, allowing the heuristic to guide the search effectively. In **Test Case 5**, **A**\* expanded nearly as many nodes as UCS, with only a marginal reduction. This outcome indicates that while the heuristic still provided guidance, its effectiveness was somewhat compromised by the increased complexity of the environment. The proximity of the heuristic estimate (h(n)) to the actual cost (g(n)) likely caused **A**\* to behave more like UCS, thereby diminishing the relative benefits.

**A**\* maintained a larger number of nodes on the frontier than UCS, mainly in Test Case 5. This is consistent with expectations because **A**\* keeps a broader set of nodes under consideration as it balances both the current path cost and the estimated remaining cost. In environments where the heuristic provides significant guidance, this approach results in a more extensive yet strategic exploration of the state space. Furthermore, **A**\* achieved faster run times than UCS in most test cases (1-4), largely due to the reduction in expanded nodes. However, in Test Case 5, **A**\* was slower. The additional computational overhead required to manage the heuristic, combined with the complexity of the environment, likely offset the gains from fewer node expansions.

Overall, A\* demonstrated its superiority over UCS, achieving the intended benefits by significantly reducing both the number of nodes expanded and the time required to reach the goal. The heuristic guided the search process with remarkable precision, allowing A\* to consistently outperform UCS across these scenarios. These results not only validate the effectiveness of the heuristic in less complex environments but also underscore its power in optimizing search efficiency. In Test Case 5, while A\* still managed to reduce the number of node expansions compared to UCS, it exhibited a longer run time. This suggests that the heuristic was closely aligned with the actual cost (g(n)), causing A\* to behave more like UCS. Alternatively, the heuristic may not have been sufficiently tuned for the increased complexity of this scenario, resulting in more computations that did not yield significant time savings. Despite this, the heuristic's capacity to reduce node expansions—even in such a challenging environment—highlights its fundamental strength and adaptability.

**Question 4**

The **preprocess\_heuristic function** is composed of multiple components, each addressing a specific aspect of the problem. The movement cost, widget push cost, and rotation cost components ensure that the heuristic accurately reflects the complexities of the environment. The precomputation of these costs further enhances the efficiency of the search process. Together, these components create a powerful and admissible heuristic that effectively guides the A\* search algorithm, leading to optimal solutions with reduced computational overhead.

**distance = abs(i - target[0]) + abs(j - target[1])**

**move\_cost = distance \* ACTION\_BASE\_COST[FORWARD]**

* The movement cost component of the heuristic estimates the cost of moving the BeeBot from its current position to a widget or a target location. Given the hexagonal grid structure, movement is not as straightforward as in a square grid. Therefore, the heuristic uses a Manhattan-like distance adapted for the hex grid, which is calculated using the absolute differences in the row and column indices. This calculation assumes that the BeeBot moves in the most direct path to minimize the number of steps. The cost is then multiplied by the base action cost for forward movement, ensuring that the heuristic reflects the actual cost of navigating the grid. This component is justified because it represents the effort required for the BeeBot to traverse the grid. Since the heuristic is based on the minimum possible distance, it guarantees that the cost estimate will not exceed the actual cost, preserving the admissibility of the heuristic.

**widget\_move\_cost = distance \* ACTION\_PUSH\_COST[FORWARD]**

* In the BeeBot task, pushing a widget is more expensive than moving without one. Therefore, the heuristic must account for the additional cost of moving widgets to target locations. This component estimates the cost of pushing a widget over the distance calculated by the movement cost. This cost is added to the movement cost, reflecting the higher effort required when the BeeBot is pushing a widget. The ACTION\_PUSH\_COST[FORWARD] is used to ensure that the heuristic accurately represents the increased cost of pushing. Without this component, the heuristic would underestimate the cost of pushing, leading to suboptimal search paths. By including this cost, the heuristic remains realistic and admissible.

**rotation\_cost = 2 \* ACTION\_BASE\_COST[SPIN\_LEFT]** # Adjust rotation cost if needed

* One of the challenges in the BeeBot task is the need to rotate widgets to fit them into target positions. Rotation is expensive, especially if the widget must be rotated multiple times to achieve the correct orientation. The heuristic must account for these potential costs to provide an accurate estimate. This component assumes a worst-case scenario where two rotations are needed to align the widget correctly. The cost is based on the action cost for spinning left, but this can be adjusted depending on the specific scenario. Including rotation costs in the heuristic is justified for scenarios where widgets must be rotated to achieve the goal. By assuming a worst-case scenario, the heuristic remains conservative and admissible, ensuring that it does not underestimate the true cost of reaching the goal.

**for i in range(self.environment.n\_rows):**

**for j in range(self.environment.n\_cols):**

**min\_cost = float('inf')**

**for target in self.environment.target\_list:**

**# Combine all components**

**total\_cost = move\_cost + widget\_move\_cost + rotation\_cost**

**min\_cost = min(min\_cost, total\_cost)**

**self.precomputed\_costs[(i, j)] = min\_cost**

* To ensure that the heuristic can be applied efficiently during the search, the costs associated with movement, pushing, and rotation are precomputed for every possible grid position. This allows the heuristic to quickly estimate the cost of any given state during the search process. By precomputing these costs, the heuristic can be applied in constant time during the search, significantly improving the efficiency of the algorithm. Precomputing the costs balances the trade-off between the upfront computational effort and the real-time efficiency of the heuristic. This approach ensures that the heuristic is both accurate and computationally feasible, leading to faster and more efficient search performance.

The **compute\_heuristic function** applies the precomputed costs and address sub-problems in the BeeBot task.

Sub-Problem 1: Moving to the Nearest Widget

**min\_move\_cost = float('inf')**

**for widget\_centre in state.widget\_centres:**

**dist = abs(state.BEE\_posit[0] - widget\_centre[0]) + abs(state.BEE\_posit[1] - widget\_centre[1])**

**move\_cost = dist \* ACTION\_BASE\_COST[FORWARD]**

**min\_move\_cost = min(min\_move\_cost, move\_cost)**

* The first sub-problem the heuristic addresses is the cost associated with moving the BeeBot to the nearest widget. Given that the BeeBot's position may vary widely within the grid, identifying the least costly move to a widget is fundamental for efficiency. This component iteratively calculates the Manhattan-like distance from the BeeBot's current position to each widget. The distance is then multiplied by the base cost of forward movement to determine the cost of reaching that widget. By selecting the minimum cost across all widgets, this heuristic ensures that the BeeBot follows the most efficient path toward the nearest widget, reducing unnecessary movements and contributing to the overall admissibility of the heuristic.

Sub-Problem 2: Moving Each Widget to Target Cells

**widget\_to\_target\_cost = 0**

**for widget\_centre in state.widget\_centres:**

**base\_cost = self.precomputed\_costs[widget\_centre]**

**widget\_to\_target\_cost += base\_cost**

* Once the BeeBot has reached a widget, the next challenge is to move that widget to its target position. This sub-problem involves calculating the cost associated with pushing the widget, rotating it, and positioning it correctly over the target cells. The heuristic controls precomputed costs that already account for the complexities of movement, pushing, and rotation. By summing these precomputed costs for each widget, the heuristic ensures that all relevant factors are considered in the estimated cost. This approach is efficient because it avoids recalculating costs during the search process, and it remains admissible because the precomputed costs represent the minimum possible cost to reach the target cells.

Combining the Sub-Problems

**total\_cost = min\_move\_cost + widget\_to\_target\_cost**

* The total heuristic cost is computed by combining the cost of moving to the nearest widget and the cost of moving each widget to its target cells. This combination ensures that all major aspects of the problem are considered in the heuristic. By summing these costs, the heuristic provides a comprehensive estimate of the total cost required to reach the goal from the current state. This method maintains the heuristic's admissibility, ensuring that it does not overestimate the true cost, while also offering a practical and computationally efficient means of guiding the search process.

The heuristic implemented in the **compute\_heuristic function** is precisely designed to maintain **admissibility**, ensuring that the estimated cost to reach the goal (h(n)) never exceeds the true minimal cost (g(n)). This is achieved by calculating the distance on a hexagonal grid, factoring in the higher costs associated with pushing widgets, and conservatively estimating rotation costs based on worst-case scenarios. Each component of the heuristic—distance calculation, widget movement costs, and rotation costs—contributes to a realistic and minimal cost estimate, guaranteeing that the A\* algorithm remains focused on finding an optimal solution without being misled by an inflated heuristic.

In addition to ensuring admissibility, the **preprocess\_heuristic function** precomputes the costs associated with moving to target cells, considering movement, pushing, and rotation. This precomputation allows the heuristic to be applied in constant time during the search, significantly reducing the computational burden during runtime. By front-loading the computation, the search process benefits from faster heuristic evaluations, for handling large state spaces efficiently. This accuracy ensures that the A\* search algorithm is guided effectively, reducing the likelihood of exploring unnecessary paths and thereby improving search efficiency. Overasll, the heuristic strikes a careful balance between being admissible (ensuring optimality) and being computationally efficient (ensuring the search process is feasible within a reasonable time frame). By incorporating both lower-bound cost estimates and efficient precomputation strategies, the heuristic ensures that the A\* algorithm remains both optimal and practical, even in complex problem scenarios.

**Appendix/References**

Most of my script is adapted from 2024 Tutorial 3 Solutions