**ICT203 Assignment 1**

Table of Contents

[**Requirements/Specification** 2](#_Toc169037363)

[**Design/Algorithm** 4](#_Toc169037364)

## **Requirements/Specification**

**Problem Statement**

Develop an intelligent agent to navigate a taxi in a given environment to pick up and drop off passengers using the shortest path. The environment is a 5x5 grid with specific locations for picking up and dropping off passengers. The agent will use search algorithms to determine the optimal route for each task while maximising the reward.

**Assumptions and Conditions**

1. **Environment Grid:** The environment is a 5x5 grid where each cell represents a possible location for the taxi.
2. **Passenger Locations:** The taxi can pick up passengers from four possible locations: Red (0), Yellow (1), Green (2), Blue (3). If the passenger is inside the taxi, the state is represented by 4.
3. **Drop-off Locations:** The taxi can drop off passengers at four possible locations: Red (0), Yellow (1), Green (2), Blue (3).
4. **Actions:**

* 0 = move south
* 1 = move north
* 2 = move east
* 3 = move west
* 4 = pick up passenger
* 5 = drop off passenger

1. **Rewards:**

* Each step incurs a penalty of -1 point
* Proper pick-up and drop-off actions earn a reward of 10 points.

1. **Objective:** Maximise the final reward by picking up and dropping off passengers using the shortest possible path.

**Formalisation of the Problem**

1. **State Space:** Represented by the tuple (taxi\_position, passenger\_location, destination). The taxi\_position is represented by coordinates (x,y), and the passenger\_location and destination are integers from 0 and 3.
2. **Action Space:** Action is defined as (move south, move north, move east, move west, pick up, drop off).
3. **Transition Model:** Defines how the state responds to actions. For instance, moving north decreases the y-coordinate by 1, while picking up a passenger changes the passenger\_location to 4 (inside the taxi).
4. **Reward Function:** Each move action returns a reward of -1. Successful pick-up and drop-off actions return a reward of 10.
5. **Goal State:** Achieved when the passenger has been picked up and dropped off at the correct location.

**Search Algorithms**

The following search algorithms will be implemented:

1. **Breadth-First Search (BFS):** Explores the state spaces level by level
2. **Uniform Cost Search (UCS):** Explores the state space by expanding the least costly node.
3. **A\* Search (AFS):** Uses a heuristic to guide the search, aiming to find the shortest path more efficiently.

**Solution Design**

* Reading the environment
* Move to pick-up location
* Move to drop off location
* Calculate reward

**Assumptions**

* Taxi moves freely within grid boundaries
* Environment and initial state are set up correctly
* Actions are sufficient for navigation

**Expected Outputs**

* Path for each algorithm
* Total reward
* Total steps

## **Design/Algorithm**

**Solution Design Overview**

The solution revolves around navigating a taxi in a grid environment to efficiently pick up and drop off passengers using Breadth-First Search (BFS), A\* Search, and Uniform Cost Search algorithms. Each algorithm’s effectiveness will be compared to demonstrate their optimisation levels in pathfinding.

**BFS (Breadth-First Search):**

The bds method that was implemented employs Breadth-First Search (BFS) to find a path from the initial state to a goal in the “Taxi-v3” environment. It begins by initialising and deep copying the environment, then resetting it. The method starts with a queue containing the initial state and an empty path while maintaining a set of visited states. It then explores all actions from the current state, enqueueing new, unvisited states and updating steps and rewards. Once the goal is reached, it displays the path, total reward, steps, and time. If no solution is found, it prints a message, ensuring the discovery of the shortest path in an unweighted environment.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pseudocode Explanation: bfs(self) | | | | | |
| 1. | **Function** bfs(self): | | | | |
| 2. |  | **Initialise** startTime **to current time** | | | |
| 3. |  | **Initialise** totalSteps **to** 0 | | | |
| 4. |  | Copy the environment **to create** copyEnv | | | |
| 5. |  | Reset copyEnv **to** its initial state and store it **as** initialState | | | |
| 6. |  | Initialise a queue **with** the tuple (initialState, []) | | | |
| 7. |  | Initialise a **set** named visited and **add** initialState **to** it | | | |
| 8. |  | Initialise totalReward **to 0** | | | |
| 9. |  | **While** the queue is not empty: | | | |
| 10. |  |  | Dequeue a tuple (state, path) **from** the queue | | |
| 11. |  |  | **For** each possible **action** in the **action** space: | | |
| 12. |  |  |  | **Set** the environment state **to** state | |
| 13. |  |  |  | **Execute** the **action** in copyEnv and obtain newState, reward, and done | |
| 14. |  |  |  | **If** newState has not been visited: | |
| 15. |  |  |  |  | Increment totalSteps **by 1** |
| 16. |  |  |  |  | **Add** newState **to** visited |
| 17. |  |  |  |  | Enqueue a tuple (newState, path+[**action**]) **to** queue |
| 18. |  |  |  |  | **Add** reward **to** totalReward |
| 19. |  |  |  | **If** done is **true**: | |
| 20. |  |  |  |  | Display the results including path, totalReward, totalSteps, **time taken**, and **algorithm type**. |
| 21. |  |  |  |  | **Return** |
| 22. |  | **Print** “No solution found” | | | |
| 23. | **End Function** | | | | |

**UCS (Uniform Cost Search):**

The ucs method in the algo\_test class employs the Uniform Cost Search (UCS) algorithm. It begins by initialising the environment and priority queue with the initial state ad cost. The algorithm then explores the state with the lowest cumulative cost, updating steps and rewards as it progresses. If a new state has a lower price than previously recorded, it updates the cost and revisits the state. Once the goal is reached, it displays the path, total reward, steps, and execution time. If no solution is found, it prints a message.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pseudocode Explanation: ucs(self) | | | | | |
| 1. | **Function** ucs(self): | | | | |
| 2. |  | Initialise startTime **to** current **time** | | | |
| 3. |  | Initialise totalSteps **to 0** | | | |
| 4. |  | Make a deep copy **of** environment **as** copyEnv | | | |
| 5. |  | Reset copyEnv **to** its initial state **and** store it **as** initialState | | | |
| 6. |  | Initialise a priority queue **with** tuple (cost, initialState, []) | | | |
| 7. |  | Initialise a **set** named visited **and** add initialState **to** it | | | |
| 8. |  | Initialise a dictionary named cost **with key** initialState **and** value **0** | | | |
| 9. |  | Initialise totalReward **to 0** | | | |
| 10. |  | **While** the queue **is not** empty: | | | |
| 11. |  |  | Pop the tuple (currentCost, state, path) from the queue | | |
| 12. |  |  | **Set** done **to False** | | |
| 13. |  |  | **For each** possible action: | | |
| 14. |  |  |  | **Set** the environment state **to** state | |
| 15. |  |  |  | Execute the action **in** copyEnv **and** obtain newState, reward, **and** done | |
| 16. |  |  |  | Calculate newCost **as** currentCost+**1** | |
| 17. |  |  |  | **If** newState **is not** visited **or** cost **of** newState **is** greater than newCost: | |
| 18. |  |  |  |  | Increment totalSteps by **1** |
| 19. |  |  |  |  | Update cost **of** newState **to** newCost |
| 20. |  |  |  |  | Add newState **to** visited |
| 21. |  |  |  |  | Push a tuple (newCost, newState, path+[action]) **to** the queue |
| 22. |  |  |  |  | Update totalReward by adding reward |
| 23. |  |  |  | **If** done **is True**: | |
| 24. |  |  |  |  | **Display** the results including path, totalReward, totalSteps, **time taken**, and **algorithm type** |
| 25. |  |  |  |  | **Return** |
| 26. |  | **Print** “No solution found” | | | |
| 27. | **End Function** | | | | |

**A\* Search:**

The afs method in the algo\_test class implements the A\* Search (AFS) algorithm. It begins by initialising the environment and a priority queue with the initial state, cost, and an empty path. The algorithm then explores states, prioritising those with the lowest combined cost and heuristic value. The heuristic function is modified to return 0 if the goal state is reached and 1 otherwise, as the goal state is unknown. The method performs actions for each state to generate new states, updating costs and paths as necessary. If a new state is not in the visited set or has a lower cost than previously recorded, it updates the cost and revisits the state. Upon reaching the goal, the method displays the path, total reward, steps, and execution time. If no solution is found, it prints a message.

|  |  |  |  |
| --- | --- | --- | --- |
| Pseudocode Explanation: heuristic(self, done) | | | |
| 1. | **Function** Heuristic(self, done): | | |
| 2. |  | **If** done **is True**: | |
| 3. |  |  | **Return 0** |
| 4. |  | **Else**: | |
| 5. |  |  | **Return 1** |
| 6. | **End Function** | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pseudocode Explanation: afs(self) | | | | | |
| 1. | **Function** afs(self): | | | | |
| 2. |  | **Initialise** startTime **to** current **time** | | | |
| 3. |  | **Initialise** totalSteps **to 0** | | | |
| 4. |  | Make a deep copy **of** the environment **as** copyEnv | | | |
| 5. |  | Reset copyEnv **to** its initial state **and** store it **as** initialState | | | |
| 6. |  | Initialise a priority queue **with** tuple (cost, initialState, []) | | | |
| 7. |  | Initialise a **set** named visited **and** add initialState **to** it | | | |
| 8. |  | Initialise a dictionary named cost **with key** initialState **and** value **0** | | | |
| 9. |  | Initialise totalReward **to 0** | | | |
| 10. |  | **While** the queue **is not** empty: | | | |
| 11. |  |  | Pop the tuple (\_, state, path) from the queue | | |
| 12. |  |  | **Set** done **to False** | | |
| 13. |  |  | **For each** possible action: | | |
| 14. |  |  |  | **Set** the environment state **to** state | |
| 15. |  |  |  | Execute the action **in** copyEnv **and** obtain newState, reward, **and** done | |
| 16. |  |  |  | Calculate the newCost **as** cost **of** state+**1** | |
| 17. |  |  |  | **If** newState **is not** visited **or** cost **of** newState **is** greater than newCost: | |
| 18. |  |  |  |  | Increment totalSteps by **1** |
| 19. |  |  |  |  | Update cost **of** newState **to** newCost |
| 20. |  |  |  |  | Add newState **to** visited |
| 21. |  |  |  |  | Calculate priority **as** newCost+heuristic **of** done |
| 22. |  |  |  |  | Push a tuple (priority, newState, path+[action]) **to** the queue |
| 23. |  |  |  |  | Update totalReward by adding reward |
| 24. |  |  |  | **If** done **is True**: | |
| 25. |  |  |  |  | **Display** the results including path, totalReward, totalSteps, **time taken**, **and algorithm type** |
| 26. |  |  |  |  | **Return** |
| 27. |  | **Print** “No solution found” | | | |
| 28. | **End Function** | | | | |

## **Limitations**

A Limitation of the provided solution stems from the inherent uncertainty surrounding the goal state in the program. Since the goal state is unknown, the A\* search (AFS) and Uniform Cost Search (UCS) algorithms must operate without complete information, relying instead on a modified heuristic function. This simplified heuristic, which returns 0 when the goal state is reached and 1 otherwise, may not accurately estimate the distance to the goal. Consequently, while AFS and UCS may yield comparable rewards and paths due to their optimal nature, their performance in reaching the goal may vary considerably. This discrepancy highlights the challenge of navigating the state space effectively without precise knowledge of the goal state.

A screenshot of a computer program

Description automatically generated

## **Simulation and discussion**

The simulation results offer valuable insights into the performance of Breadth-First Search (BFS), Uniform Cost Search (UCS), and the A\* Search algorithms in the Taxi-v3 environment. Despite all three algorithms successfully reaching the goal state with identical paths and rewards, their execution times vary noticeably.

BFS, known for its simplicity, completed the task in approximately 0.016 milliseconds. However, its exhaustive exploration strategy led to a higher step count of 56, reflecting its thoroughness in traversing the state space.

In contrast, UCS and A\* Search, levering cost information, exhibited slightly longer execution times of around 0.018 and 0.028 milliseconds, respectively. Yet, they achieved this with fewer steps, indicating their efficiency in navigating the state space by prioritising lower-cost paths.

A screenshot of a computer program

Description automatically generated

These findings emphasise the critical role of algorithm selection, balancing completeness, optimality, and computational efficiency. While BFS ensures completeness, UCS and A\* search prioritise solution quality and reduce exploration, resulting in quicker execution times without compromising the solution's optimality.