CS2012: Artificial Intelligence
Report-2
Autonomous Robot Waiter Simulator

Thribhuvan, EC21B1093
Lokesh, EC21B1104

April 24, 2024

1 Introduction

Previously, we have seen how an Autonomous Robot Waiter Simulator can be implemented to simulate a robot that takes a food item and places it at a designated location by pushing it in any of the four directions, while ensuring optimal and shortest path, along with avoiding obstacles. Now, we shall take a look at the approach taken to solve the problem using AI and to simulate the system.

2 Algorithms Used

2.1 A* search

1. Distance Functions:
   We define two distance functions: Euclid distance and Manhattan distance. These functions calculate the Euclidean distance and Manhattan distance between two points, respectively.

2. Heuristic Function:
   The heuristic function calculates the heuristic value for a given state. It evaluates the estimated distance from the current state to the goal state. In our implementation, it calculates the sum of the Manhattan distances from each butter location to the nearest goal point on the map.

3. Setting Heuristic for Nodes:
   The heuristic function is assigned to the heuristic attribute of the Node class. This allows each node to access the heuristic function during the A* search.

4. Initialization:
   We initialize a priority queue (heap) and a set (visited) to keep track of visited states. It creates a root node with the initial state and adds it to the priority queue.

5. A* Search Loop:
   The A* search algorithm is executed within a loop that continues until the priority queue is empty. The algorithm retrieves the node with the lowest cost (priority) from the priority queue. If the retrieved node represents a goal state, the algorithm terminates, and the node is returned. Otherwise, the algorithm expands the current node by generating successor states (actions) and adding them to the priority queue if they have not been visited before.

6. Successor Generation:
   We generate successor states (child nodes) based on the current state and available actions (possible moves). These successor states are then added to the priority queue for further exploration.

2.2 Breadth First Search (BFS)

1. Initialization:
   The function initializes a frontier with a single node representing the initial state and an empty dictionary (visited) to keep track of visited states.

2. BFS Loop:
   The BFS loop continues until the frontier is empty. In each iteration, A node is dequeued from the frontier. The dequeued node is marked as visited by adding it to the visited dictionary. If the dequeued node represents the goal state, the function terminates, and the node is returned. Successor states (children) of the dequeued node are generated, and if they have not been visited before, they are added to the frontier.

3. Successor Generation:
   Similar to A* search, a successor method generates successor states (children) based on the current state and available actions (possible moves). These successor states are then added to the frontier for further exploration.

4. Returning Results:
   If a goal node is found during the BFS traversal, it is returned. Otherwise, the function terminates when the frontier is empty, indicating that no solution exists.

2.3 Bi-directional BFS

1. bd bfs Function:
   This function performs bidirectional BFS from an initial state to a specific goal state. It simultaneously explores states from both the initial and goal states until they meet. It returns two nodes where the two searches converge.

2. Initialization:
   Two frontiers (frontier1 and frontier2) are maintained to keep track of nodes to be explored from the initial state and the goal state, respectively. Two dictionaries (visited1 and visited2) are used to store visited states and corresponding nodes.

3. BFS Loop:
   The bidirectional BFS loop continues until one of the frontiers becomes empty or until the searches meet. In each iteration, nodes are dequeued from both frontiers (frontier1 and frontier2). If a node from one search is found in the visited set of the other search, it means that the searches have met. In this case, the function returns the corresponding nodes. Successor states (children) are generated for the current node of the initial state and added to frontier1. Predecessor states (parents) are generated for the current node of the goal state and added to frontier2.

4. Finding All Goal States:
   This section of the code generates all possible goal states by placing the robot in all positions around the food item. It creates new states for each possible position.

5. Performing Bidirectional BFS:
   The code then iterates over all possible goal states and performs bidirectional BFS for each pair of initial and goal states. It then checks if a valid path is found and updates the shortest path if needed.

6. Returning Results:
   The function returns the shortest path found, its length, and the total cost of the path.

2.4 Reverse BFS

1. Reverse BFS Function:
   The reverse bfs function performs a reverse BFS starting from a goal state. It explores states backward from the goal state towards the initial state.

2. Initialization:
   The function initializes a frontier with the goal state and a visited set to keep track of visited states.

3. BFS Loop:
   The reverse BFS loop continues until the frontier is empty. In each iteration, a node is dequeued from the frontier. If the dequeued node represents the initial state, the function terminates, and the node is returned. Predecessor states (parents) of the current node are generated, and if they have not been visited before, they are added to the frontier and marked as visited.

4. Generating Goal States:
   The code generates all possible goal states by placing the robot in all positions around the food item.

5. Performing Reverse BFS:
   The code iterates over all possible goal states and performs reverse BFS for each goal state. It then checks if a valid path is found and updates the shortest path if needed.

6. Returning Results:
   The function returns the shortest path found from a goal state to the initial state.

2.5 Iterative Deepening Search (IDS)

1. Depth-Limited Search (DLS) Function:
   The dls search function implements Depth-Limited Search, which is used as a subroutine in IDS. It explores nodes up to a specified depth limit and returns the goal node if found within that limit.

2. Iterative Deepening Loop:
   The main loop of IDS iterates over increasing depth limits, starting from first depth and ending at last depth. It repeatedly applies DLS with increasing depth limits until a solution is found or the maximum depth limit is reached.

3. DLS Implementation:
   Within the dls search function, if the time limit is exceeded, an exception is raised.The function explores nodes recursively up to the specified depth limit. It expands nodes and checks if the goal state is reached. If the goal state is found, the function returns the goal node. The function maintains a dictionary (visited states) to keep track of visited states to avoid revisiting already explored states.

4. Returning Results:
   If a goal node is found within the depth limit for any iteration of IDS, it is returned.

3 Simulation

For visualizing the path of the robot, we will be using Pygame library for graphical display. We intend to use it as follows:

1. Setup Pygame:
   Import the Pygame library and initialize it. Set up the display window and any other necessary variables.

2. Define the Grid:
   Create a grid representing the environment where the search is performed. We can represent obstacles, initial state, goal state, and explored nodes using different colors or symbols.

3. Implement Algorithm:
   Implement the search algorithms (A*, IDS, BFS and its variants) as a function that takes the grid, initial state, and goal state as input and returns the path or explored nodes.

4. Visualize Search:
   Within the main loop of the program, visualize the search process by updating the display based on the current state of the search. We can animate the exploration of nodes, display the path found, update colors to indicate visited nodes, etc.

5. Handle Events:
   Handle user events such as quitting the program or adjusting parameters like the speed of visualization.

6. Display Results:
   Once the search is complete, display the final path or any other relevant information.