KR Assignment 1

Noura Qassrawi

October 13, 2024

1 Task Environment

The Environment of Treasure Hunt Game is a 2D partially observable grid game where the agent navigates through the 2D game tiles to collect the coins across the grid, the agent also has to avoid the dynamically changing obstacles (walls) which keep changing locations across the game span. The environment grid is defined by it's size (width and height) and the positions of walls, coins and the agent. The grid cell can be of several types: empty, wall, coin or agent. The environment is of Stochastic nature as it has a level of uncertainty as a result of the dynamically changing walls as well as the randomly placed coins. The environment always has a valid path from agent to position to all goal coin positions despite random dynamic wall placement

1.1 Dynamic Walls Reach-ability Problem

The game grid through dynamic environment setup ensures walls are placed randomly across the grid. A path is always ensured from agent to all coins. every 20 steps the environment dynamically changes wall placements. During this process reach-ability problem is checked each time the walls change positions to ensure a valid path always exists for the agent

- **Random Wall Placement**: The first step for environment setup is randomly placed walls in the environment while ensuring that at least one tile in each row and column remains free, so there are no completely blocked sections.
- **Validation of Reach-ability**: After walls are employed, the coins are placed using BFS search across the grid. Looping through the grid, a check is applied on the (x,y) position to ensure agent can reach named location and collect all coins. This is achieved by deploying a problem-solving technique that employs a reach-ability check
- **Adjusting Wall Placement if Necessary**: if while placing the coins no locations in the grid can be found as reachable, the number of walls is reduced. This happens as a result of large number of walls passed to the a small grid environment. As a result of this process, a path is always ensured to each coin.

1.2 deeper dive into Reachability Problem

Technical explanation of how the environment is setup, In order to ensure a path always exists as described previously, Reachability Problem. This problem is modeled in a way that:

- The initial state is the agent's starting position.
- The goal is to reach all the positions where coins are located.
- The actions available to the agent are moving in four possible directions: left, right, up, and down.
- The environment contains walls that block the agent's movement. **Downside**: BFS takes quite some time as it iterates through every node, due to the nature of implemented game and since the walls randomly change every n steps, this takes time and delays the agents.

  Why use BFS?

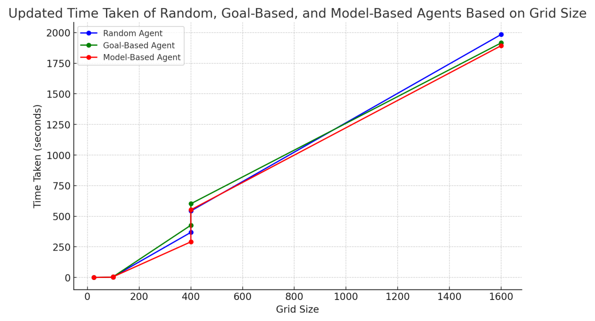

Figure 1: time complexity vs Grid size

- its a complete uninformed search algorithm, meaning it guarantees to find a solution to the problem without requiring any knowledge of the environment beyond what it explores

its suited for a reach-ability problem as it provides optimal shortest paths

2 Q1: Agents

To represent Agents, a basic Agent class is created ”BaseTreasureHuntAgent” which extends Agent class from aima-python, this class has the main functionalities used by other agents which extend this class.

2.1 Performance Measurements for all agents

- score: gains a 100 points for each coin, loses 1 point for each move, loses 2 points for running through a wall
- Collected Coins: number of coins collected
- Time Taken: This measures how long it takes to complete the task or collect all coins.
- Path Cost: Calculated by the number of moves made. Higher path cost indicates inefficiency in collecting coins.
- Space Complexity: how many nodes are stored in memory during the process

Agents Used:
Analyzing the results of each Agent performance, we can notice the following:

- Space complexity: for Random reflex agent since it doesn’t store anything, its always o(1) as it relies on current percept only however, model based agent always takes the mpst space since it stores visited nodes as well as it remembers the walls locations. a note to keep in mind remembering the walls location wasn’t very beneficial in performance after dynamically moving the walls
- Space complexity: for Random reflex agent since it doesn’t store anyhthing, its always o(1) as it relies on current percept only
- Performance: checking the ”number of collected coins” graph it indicates the Goal based Agent has the most collected coins since it doesn’t stop until goal is achieved, unlike Random agent which tends to go randomly as it has no vision of the environment

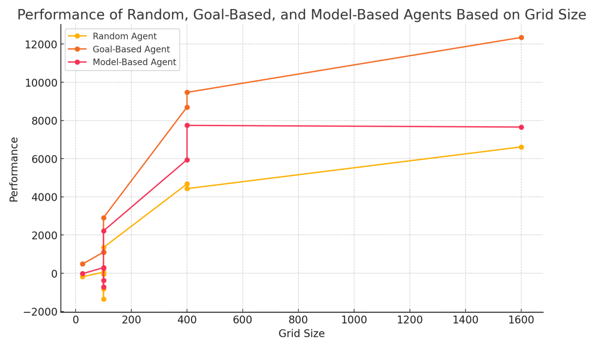

Figure 2: Agent Performance vs Grid size

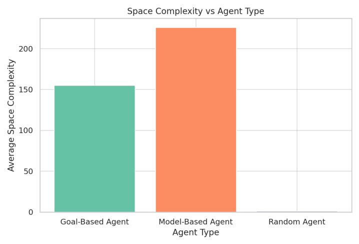

Figure 3: space complexity

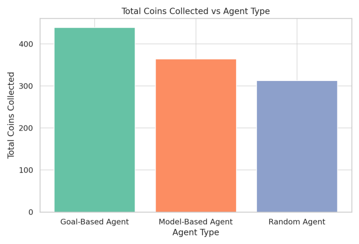

Figure 4: number of collected coins

3 Q2: Problem Statement and Search Problem

In Treasure Hunt Game Problem statement represents a path-finding problem where the goal is to collect all coins in the grid, while navigating around walls. Note that for this section, the walls are made static, they don't dynamically move in the middle of the game.

- The initial state is the agent's starting position which is (0,0).
- The goal is to collect all coins across the grid and game is considered complete.
- The actions are movement directions: left, right, up, and down.
- Path Cost Function: Each step (movement) is considered to have a cost of 1 (such as 1 per move), and there may is a negative cost (reward) for collecting a coin.

3.1 Uninformed Search Methods and Their Appropriateness

- **Breadth-First Search**:
  - Description BFS explores the nodes level by level, expanding the shallowest nodes first. It uses a FIFO queue and ensures that the first solution found is the shortest path in terms of the number of steps.
  - **Appropriateness**
    - **Advantages** BFS search is complete, always guarantees to find path to given search solution
    - **DisAdvantages** it uses alot of memory and while it functions well in small/medium environment, it is very slow with larger grid sizes, this can be seen in both Q1 and Q2 since it was utilized in "wall placement problem" as well as "Search Problem"
- **Depth-First search**:
  - Description DFS explores as deep as possible in the node graph. this code make it useful for finding coins that are deep in the grid. However this leads to a higher cost and a non-optimal path unlike BFS.
  - **Appropriateness**
    - **Advantages** DFS search is better in large grid systems where coins are put in deep nodes
    - **DisAdvantages** has a high path cost
  - **Uniform Cost search**:
    - Description
      Uniform Cost Search (UCS) expands the lowest-cost path first, ensuring it finds the most cost-efficient route. However, After imploying it in the treasure hunt game, where every path has the same cost (1), UCS didn't provide any advantage over simpler algorithms like Breadth-First Search because all paths are equally cheap, making UCS redundant in this case.
    - **Appropriateness**
      - **Advantages** Gurantees optinmality (still same as BFS in treasure hunt case)
      - **DisAdvantages** Slower on Uniform Costs (doesn't apply as well in treasure hunt case)

  Analyzing treasure hunt game results:
    - DFS brought the fastest solution however it came with a high cost
    - bith UCS and BFS behaved the same as a result of treasure hunt game path cost being 1 for all nodes, both solution worked well in small environments however, they perform poorly on larger grids as a result to high memory consumption since both have to store entire set of "level nodes" in memory

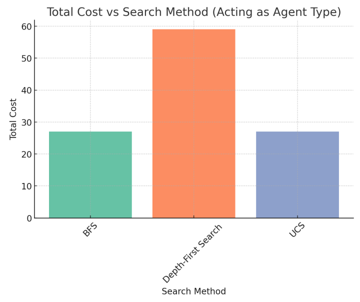

Figure 5: space complexity

3.2 Informed Search Methods and Their Appropriateness

Informed search algorithms use domain-specific information (heuristics) to guide the search toward the goal more efficiently compared to uninformed searches. In the treasure hunt problem, informed search methods help the agent by incorporating heuristics that estimate the distance or cost to the goal (i.e., the coins). By doing so, the search process can often find solutions faster and with fewer explored nodes.

For the purpose of treasure hunt environment, we implemented and compared Greedy Best-First Search and A Search*. Both of these algorithms rely on a heuristic function that estimates the remaining cost to the goal, but they use this information differently.

– Greedy Search: a search algorithm that explores a graph by attempting to expand the most promising node. This is done by a heuristic function by favoring nodes with lower heuristic values.

  Best first search is applied using manhattan distance to nearest coin heuristic on all the goal coins. It basically calculates the sum of the absolute differences in the x and y coordinates between the agent and the closest coin. The idea is that this heuristic will lead the agent towards the nearest coin, making the search more focused and seemingly faster by reducing unnecessary exploration.

– A* search: In our treasure hunt game, A search* is used with the Manhattan distance to all goals heuristic, which guides the agent by considering the total distance to all remaining coins. The algorithm balances the actual cost of reaching the current position (moves made) with an estimate of how far the agent is from collecting all the coins.

  The A* algorithm evaluates both how far the agent has traveled and how far it still has to go to collect all the coins, ensuring a more strategic and globally informed search. It guarantees finding the optimal path in terms of total cost.

– Recursive Best search: is a search algorithm that attempts to solve the problem of limited memory in informed search algorithms. It expands nodes similarly to A* but keeps track of only the current path in memory, and if a node's heuristic value exceeds the limit, it backtracks. RBFS uses $f(n) = g(n) + h(n)$ like A*, but instead of storing all nodes, it only keeps the current path. If a node's f-value exceeds the threshold (determined by the best alternative), the algorithm backtracks, updating the limit as it goes. This allows it to operate in a memory-efficient way but may involve more backtracking.

  Applying the above to treasure hunt game:

  most optimal path: A* has proven to get the most optimal path while RBFS gets stuck at larger grids where coins were placed near each other as it seems to get stuck in local optima .This happens because the search algorithm constantly tries to expand the node with the smallest Manhattan distance, even if that node doesn't actually lead to an efficient collection of coins. The search might waste time expanding nodes around one coin while neglecting a more efficient path to collect multiple coins in sequence.

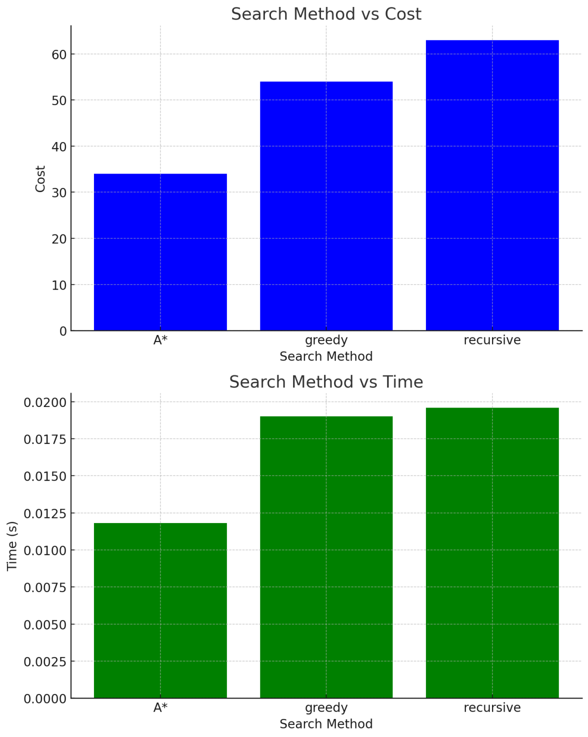

Figure 6: informed searchers