COMP2069.Algorithms Data Structures and Efficiency

Coursework Report: Treasure Hunt Game

April 22, 2025

20513824
Yuanhao Dai
scyyd8@nottingham.edu.cn

Contents

1 Introduction **2**
1.1 Game Overview 2
1.2 Game Objective 2

2 Design and Implementation **3**
2.1 Game Map Representation 3
2.2 Game Entities and Interactions 3
2.3 Algorithms and Data Structures Used 4

3 Efficiency Analysis **9**
3.1 Time Complexity 9
3.2 Space Complexity 9
3.3 Impact on Performance 10

4 Challenges and Trade-offs **11**
4.1 Challenges 11
4.2 Trade-offs and Balancing 11

5 Conclusion **12**

1 Introduction

1.1 Game Overview

The game is a "treasure hunt" game in a two-dimensional grid format using pathfinding algorithms and various data structures. The player will control a character to navigate through a 20x20-sized map, searching for three hidden treasures. The map, treasures, obstacles, and the player's starting position are all randomly generated. The location of the treasures is unknown to the player and they will only appear when the player approaches them. The game has four directions that the player can move in (up, down, left, and right), and the player can finds the shortest path to the treasures through three different prompts, namely the directions guided by BFS, the greedy algorithm, and A* algorithm respectively. The player starts with 100 points, and if the points are depleted, the game will fail.

1.2 Game Objective

The main objective of this game is to enable players to collect all three treasures which are hidden on the randomly generated map. At the beginning of the game, all obstacles and treasures will be presented as white squares like the surrounding empty spaces. Only when they are approached or bumped into by the players then their original colors be revealed. Players must avoid the obstacles, and the map and the starting position will change every time a new round of the game begins. The initial score for players is 100 points, but points will be deducted to varying degrees when players move, use the hint function, or bump into obstacles. Points will only be added when the treasures are found. The game ends when all the treasures are found or when the player's score has been reduced to zero.

2 Design and Implementation

2.1 Game Map Representation

The game map is represented by a 2D grid of size 20x20. Each cell can contain one of the following elements:

- **Empty spaces**: Represented by a space ' '.
- **Obstacles**: Represented by 'O'.
- **Treasures**: Represented by 'T'.
- **Player**: Represented by 'P'.

This grid map is generated by the `MapManager` class. When this class is called, it will initialize the map. The `GameView` class will then render the map with colors. After the map is generated, the player will be randomly generated first, then treasures will be placed at random positions, and finally obstacles will be randomly placed while ensuring that they do not overlap with the player or the treasures. texttttPlayer can move according to user input.

2.2 Game Entities and Interactions

This game involves multiple key entities, each of which is managed by different classes:

- **Player**: The `Player` class is responsible for handling the player's movement and interactions with other entities in the game. The player can move in four directions (up, down, left, right), and starts with 100 points. Each move deducts 1 point from the player. Interactions with other entities, such as colliding with obstacles or discovering treasures, will cause changes in the player's score.
- **Treasure**: The treasures are managed by `TreasureManager`, and they will be randomly placed after the player is generated. Each time a treasure is placed, it will ensure that there is at least a 5-step distance between it and the player, and also between each other, to prevent the treasure's location from being too close to the player or having multiple treasures placed consecutively. This will greatly reduce the difficulty of the game. These restrictions ensure that the player must carefully explore the correct route instead of relying solely on luck. Collecting a treasure can increase 10 points of score, and the treasure will be removed from the corresponding list, and the original position will be replaced by a blank area. The treasures are initially hidden and appear white to the player, similar to the blank area. Only when the player is within a three-step range will it turn yellow. This design is to optimize the player's experience, allowing them to confirm the treasure's location earlier and keeping the game difficulty at an appropriate level.
- **Obstacle**: The `ObstacleManager` class randomly places obstacles on the map and manages them. Each obstacle hinders the movement of the player. If the player collides with an obstacle, their score will be deducted by 10 points. After placing the obstacles, a secondary check will be conducted to ensure that there are no situations where treasures and the player are trapped. For players, the obstacles

are initially hidden; their initial color is white, and players can easily attach them. However, once an obstacle is hit, it turns black and will no longer deduct points upon subsequent hits. This design is to prevent players from being deducted a large amount of points by accidently hitting an obstacle, and players can also make plans for the next move based on the distribution of obstacles.

- **Path Finder**: This game employs three path-finding algorithms (Breadth-First Search, Greedy Algorithm, and A* Algorithm) to assist players in finding the shortest path to the nearest treasure. These algorithms are implemented in different classes (`BFSPathFinder`, `GreedyPathFinder`, and `AStarPathFinder`), and corresponding buttons will be displayed at the bottom of the game window. Clicking on the corresponding button will receive a prompt (upward, downward, leftward, or rightward) given by the respective algorithm. Each time this function is used, three points of score will be deducted, and players also need to plan the timing of using the points reasonably.

2.3 Algorithms and Data Structures Used

- **Abstract base class PathFinder:** PathFinder is the abstract parent class for all path-finding classes, providing data structures and algorithmic foundations for its subclasses. The main data structures include:
  - **List<int[]>treasures and path:** Stores the positions of treasures and the paths to the nearest treasures. They are dynamic lists that support addition and deletion operations, and are used for path calculation and storing the positions of treasures.
  - **int[] nextStep and nearest:** Stores coordinate points. `nextStep` stores the coordinates of the next position, and `nearest` stores the coordinates of the nearest treasure.

  Algorithms:

  - **Path finding:** By using the methods `findPathToNearestTreasure` and `pathsAreClear`, the path for the player to reach the nearest treasure is found, and it is ensured that there are no obstacles on this path.
  - **Legal movement judgment:** Through the `isValidMove` method, verify whether the player's movement is legal, ensuring that it does not exceed the boundaries or collide with obstacles.
  - **Shortest Path:** The `findNearestTreasure` method selects the nearest treasure to the player by calculating the Manhattan distance and locates its position.

- **BFSPathFinder Class:** `BFSPathFinder` extends the abstract `PathFinder` class and implements the BFS (Breadth-First Search) algorithm to find the shortest path to the nearest treasure. The main data structures and algorithms used are:
  - **Queue<int[]>queue:** A queue used for the Breadth-First Search (BFS) algorithm, for storing the nodes (coordinates) to be explored.

- **boolean[][] visited:** A two-dimensional array is used to record which nodes have been visited. By using this array, repeated visits to nodes can be avoided, and breadth-first search (BFS) can ensure that unvisited nodes can be explored.
- **Map<String, int[]>parentMap:** A mapping that stores the parent node of each visited node is used to reconstruct the path after finding the target node.
- **int[] directions:** An array that defines the possible directions of movement on the grid (up, right, down and left). It is used to explore adjacent nodes during the breadth-first search process.

Algorithms:

- **findPathToNearestTreasure:** This method overrides the abstract method in the parent class `PathFinder`. It first finds the nearest treasure from the player by calling the `findNearestTreasure` method, and then uses BFS to calculate the shortest path to that treasure by calling the `findPathBFS` method.
- **Breadth-First Search (BFS):** The `findPathBFS` method implements the breadth-first search algorithm, which is used to find the shortest path from the player's current position to the target. It uses a queue to explore nodes layer by layer and employs an visited array to ensure that each node is processed only once.
- **Path Reconstruction:** The `reconstructPath` method utilizes the `parentMap` to reconstruct the path. It performs a backward traversal along the parent nodes and then reverses the path to reach the target node from the starting node.
- **Clear Paths:** The `pathsAreClear` method verifies the existence of valid paths to each treasure using the breadth-first search (BFS) algorithm. It traverses each treasure and attempts to find a path from the player's position to that treasure. If it discovers that a treasure cannot be reached, it returns false.

- **GreedyPathFinder Class:** `GreedyPathFinder` extends the abstract `PathFinder` class and implements a greedy pathfinding algorithm to find the shortest path to the nearest treasure. The main data structures and algorithms used are:

  - **PriorityQueue<GreedyNode>openSet:** A priority queue used to store the nodes to explore, ordered by their heuristic value (distance to the target). The node with the lowest heuristic is explored first.
  - **Set<String>visited:** A set that tracks visited nodes to avoid revisiting them during the pathfinding process.
  - **GreedyNode:** A custom class representing a node in the pathfinding process. It stores the node's coordinates, heuristic value, and reference to its parent node for path reconstruction.
  - **int[] directions:** An array defining the possible directions to move on the grid (up, right, down, left). This helps in exploring neighboring nodes during the greedy search.

Algorithms:

- **findPathToNearestTreasure:** This method overrides the abstract method in the parent class `PathFinder`. It locates the nearest treasure by calling the `findNearestTreasure` method, and then calculates the path using the greedy algorithm by calling the `findPathGreedy` method.
- **Greedy Path-Finding Algorithm:** The `findPathGreedy` method implements the greedy algorithm. It utilizes a priority queue to explore nodes based on the heuristic values of the nodes. This algorithm first explores the node with the lowest heuristic value to ensure reaching the nearest treasure. The `visited` set is used to prevent visiting nodes repeatedly.
- **Heuristic Function:** The `Heuristic` method calculates the Manhattan distance between a node and the target node. This distance is used as a heuristic criterion to determine the "proximity" of a node to the target node.
- **Path Reconstruction:** The `reconstructPath` method reconstructs the path from the target node to the starting node by backtracking through the parent nodes stored in each `GreedyNode`. Then, the path is reversed to provide the correct sequence from the starting node to the target node.

- **AStarPathFinder Class:** `AStarPathFinder` extends the abstract `PathFinder` class and implements the A* algorithm to find the shortest path to the nearest treasure. The main data structures and algorithms used are:
  - **PriorityQueue<AStarNode>openSet:** This is a priority queue used to store the nodes to be explored. Its elements are sorted according to the total cost (f = g + h). In the A* algorithm, the node with the lowest total cost will be explored first.
  - **Map<String, AStarNode>allNodes:** Store a mapping table for all nodes used for path reconstruction. Each node is stored through a unique key (represented as a string in the form of coordinates), and the corresponding `AStarNode` object contains detailed information of the node.
  - **AStarNode:** AStarNode is a custom class used to represent a node in the A* path planning algorithm. It stores the coordinates of the node, the g-value (cost), the f-value (total cost), and a reference to its parent node, facilitating the reconstruction of the path.
  - **int[] directions:** An array that defines the possible directions of movement on the grid (up, right, down, left). It is used to explore adjacent nodes during the A* search process.

  Algorithms:

  - **findPathToNearestTreasure:** This method overrides the abstract method in the parent class `PathFinder`. It first finds the nearest treasure to the player by calling the `findNearestTreasure` method, and then calculates the optimal path using the A* algorithm by calling the `findPathAStar` method.

  - **A* Path Optimization Algorithm:** The `findPathAStar` method implements the A* algorithm, which is used to find the optimal path from the player's position to the nearest treasure. It uses a priority queue (`openSet`) to explore nodes, where these nodes are sorted according to their total cost (that is, the cost to reach the node (g) plus the heuristic function (h) estimating the cost from the node to the goal). This algorithm manages the set of nodes to be explored through the priority queue. Each time, the node with the minimum total cost is extracted from the set for expansion, and the total cost of its adjacent nodes is updated. This iterative process continues until the target node is found.
  - **Heuristic Function:** `The heuristic` method calculates the Manhattan distance between the current node and the target node. This heuristic value is adjusted based on whether the current position is close to the map edge. For nodes close to the map edge, the distance value is increased as a penalty term to prevent the situation where the player is guided to go beyond the map.
  - **Path Reconstruction:** The `reconstructPath` method reconstructs the path from the target node to the starting node by traversing the parent nodes stored in the `AStarNode` objects. Then, it reverses the path to arrange it in the correct order from the starting node to the target node.

- **MapManager Class:** `MapManager` manages the game's map, including initializing and resetting the map, as well as tracking obstacles and their interactions with the player. The main data structures and algorithms used are:

  - **char[][] map:** A 2D array representing the game map. Each cell on the map can store a character indicating whether the space is empty, an obstacle, the player, or a treasure.
  - **boolean[][] obstacleTouched:** A 2D array used to track whether obstacles have been touched by the player. Each cell corresponds to a position on the map, with `true` indicating the obstacle at that position has been touched.

  Algorithms:

  - **initializeMap:** Initializes the map by setting all cells to empty and resetting the `obstacleTouched` array to `false`. The obstacles are then placed at their respective positions based on data from the `GameModel`.
  - **resetMap:** Resets the map by clearing all cells that do not contain a player or treasure, setting them back to empty. This method ensures that non-essential elements (like obstacles) are cleared during game resets.

- **ObstacleManager Class:** `ObstacleManager` manages the placement and validation of obstacles on the map. It interacts with the `GameModel` to modify the map and uses the `PathFinder` to ensure the player's path is clear after obstacles are placed. The main data structures and algorithms used are:

  - **Set <int[]>obstacles:** A set used to store the positions of obstacles. The use of a set ensures that there are no duplicate obstacle positions.

Algorithms:

- **placeObstacles:** This method randomly places obstacles on the map. It repeatedly generates random coordinates and places obstacles in empty spaces. After placing each obstacle, it checks if the path is clear using the `pathFinder`. If placing the obstacle blocks the path, it is removed; otherwise, the obstacle is successfully placed.

- **Player Class:** `Player` class represents the player in the game, handling the player's movement, placement, and interaction with the game map.

  Algorithms:

  - **placePlayer:** This method randomly selects a blank area on the map to place the player.
  - **movePlayer:** Players can move in the four directions (up, down, left, right) by inputting the characters ('w', 's', 'a', 'd'), and check whether the new position is valid.
    - If the new position exceeds the boundary, the player will be penalized and a message will be displayed.
    - If the new position is an obstacle, the player will be penalized and the obstacle will be marked as "touched" to prevent excessive penalties, and a message will be given.
    - If the new position is a treasure, the player will collect the treasure, the score will increase, and the player will also be moved to the new position.
    - If the new position is empty, the player will move to the empty space, and the score will decrease by 1.

- **TreasureManager Class:** `TreasureManager` manages the placement and removal of treasures in the game. It interacts with the `GameModel` to modify the map and track treasure positions. The main data structures and algorithms used are:

  - **List<int[]>treasures:** A list to store the positions of treasures on the map. Each treasure is represented by an array of two integers (x, y) indicating its position.

  Algorithms:

  - **placeTreasures:** This method will randomly place a specified number of treasures at random locations on the map. It ensures that these treasures will not be too close to the player (at least 5 spaces away), nor will they be close to other treasures (at least 5 spaces away). This method will continuously generate new locations until an effective one is found.
  - **removeTreasure:** This method will remove the treasure from the list when it is collected. It will traverse the treasure list, search for the treasure with matching coordinates (x, y), and remove it from the list once it is collected.

3 Efficiency Analysis

3.1 Time Complexity

- **BFS**: BFS uses queues to expand nodes, and the time complexity of each queue operation is O(1). The queue can store at most O(n) nodes, where n is the number of all possible nodes. When each node is expanded, its 4 adjacent nodes need to be checked, and the checking process takes O(1) time. Therefore, the overall time complexity is O(n).

- **Greedy Algorithm**:The greedy algorithm uses a priority queue to store the nodes to be expanded, and the queue is sorted according to the value of the heuristic function. The time complexity of each insertion and deletion operation is $O(\log n)$, where n is the number of nodes in the queue. For the nodes to be expanded, each node has at most four directions, so the time complexity of the expansion operation is $O(1)$, because the expansion of each node only involves looking at its adjacent nodes. The greedy algorithm does not guarantee the shortest path, it only relies on the heuristic function to make local optimal choices, so it may not expand all nodes in some cases. Moreover, since the greedy algorithm usually needs to explore the nodes in the entire graph or grid, in the worst case, all nodes may be expanded. Therefore, the total number of expansions in the worst case is $O(n)$, where n is the number of all possible nodes in the graph. Therefore, the time complexity of the entire algorithm is $O(n \log n)$, where n is the number of all possible nodes.

- **A* Algorithm**: The A* algorithm uses a priority queue to store the nodes to be expanded. The priority queue is arranged according to the heuristic total cost (f = g + h). Each insertion and deletion operation is adjusted according to this cost value. Therefore, for each node, the time complexity is $O(\log n)$, where n is the number of nodes in the queue. For the nodes to be expanded, since each node has at most four directions, it always takes $O(1)$ time. Since the A* algorithm guarantees the shortest path, all possible nodes need to be explored in the worst case, so the A* algorithm will expand at most $O(n)$ nodes. In summary, if we assume that each node will be inserted and deleted at most once, and the expansion operation of each node is $O(1)$, then the overall time complexity is $O(n \log n)$, where n is the number of all possible nodes.

3.2 Space Complexity

The space complexity of BFS, Greedy Algorithm, and A* Algorithm is $O(n)$. The reason is that in each algorithm, the space complexity is mainly composed of the following parts:

1. **Node storage:** Each algorithm needs to store the nodes to be expanded. For example, A* and Greedy Algorithm use priority queues, and BFS uses queues. All these data structures may need to store all nodes in the worst case, so $O(n)$ space is required in the worst case.

2. **Auxiliary data structure:** Each algorithm requires additional data structures to ensure that the pathfinding target will not be changed arbitrarily in the case of multiple target nodes. For example, BFS uses the access mark array method, A* and Greedy Algorithm use parent node mapping, and there is also information such

as path reconstruction. The space complexity of these structures is also proportional to the number of nodes.

Since each algorithm may need to store n nodes in the worst case, their space complexity is $O(n)$.

3.3 Impact on Performance

- **Breadth First Search (BFS):**
  BFS is guaranteed to find the shortest path. I used this algorithm to ensure that the map generated a path to the treasure for the player. The downside is that it can be slow, especially when there are many obstacles and the map is large. Because BFS needs to explore many nodes, this increases memory usage and processing time. Therefore, it may not be efficient on large-scale problems.

- **Greedy Algorithm:**
  Compared to BFS, the greedy algorithm usually executes faster because it only selects the node closest to the goal without traversing the entire graph. However, it is not guaranteed to find the shortest path. In addition, multiple goals may cause duplicate reverse prompts. Although it is good for providing quick prompts, it is not always the best path selection.

- **A* Algorithm:**
  The A* algorithm is very efficient on complex maps and can guarantee to find the shortest path. For grids with more obstacles, it performs well while ensuring the path is optimal, which is suitable for scenes with higher performance requirements. However, compared to the greedy algorithm, A* uses more memory because it needs to store more node information. At the same time, A* can cause the player to run into walls and sometimes be unable to find a viable path.

4 Challenges and Trade-offs

4.1 Challenges

- **Map generation and path planning**: Because of the randomness in map generation, players and treasures can be surrounded by obstacles, making it impossible to complete the game.
- **Balancing game difficulty**: The placement of obstacles and treasures is hidden, which is hard for players. Too many obstacles can make the game too difficult, but treasures placed close to each other can make the game too easy.
- **Performance**: While breadth-first search guarantees the shortest path, it can be slow on large grids. Greedy algorithms and A* are faster but do not always guarantee the optimal solution.
- **Incorrect local optima**: Both greedy algorithms and A* can fail to find the correct path. If there's a wall between the player and the treasure, the algorithm may not find the right direction. This can lead to the player getting stuck in loops, or A* sometimes pushing the player off the map for the optimal solution.

4.2 Trade-offs and Balancing

- **Obstacle Generation**: Originally, I intended to let the players decide the number of obstacles to generate, but in the end, I kept the number of obstacles at 50. At the same time, I restricted the players that neither the treasure nor the blank areas could be surrounded by four obstacles simultaneously. This way, the map could be maximally utilized.
- **Treasure Generation**: When generating treasures, there must be a distance of more than five steps between the treasure and the player. The same rule applies to the treasures among themselves. Players need to explore all areas of the map as much as possible.
- **Penalty Mechanism**: Both the greedy algorithm and the A* algorithm have penalty mechanisms. If the greedy algorithm wants to take the opposite prompt from the previous step, it will be punished. The A* algorithm will also receive a penalty if it guides the player towards the map boundary.
- **Map Selection**: To ensure that the game can be passed, after generating the game map, I chose to use BFS to check whether each treasure is reachable for the player. Even so, this would reduce efficiency.

5 Conclusion

In this coursework, I designed a simple treasure-hunting game with an MVC pattern as the framework, and implemented the hint function by using three common path-finding algorithms (BFS, greedy algorithm and A* algorithm). At the same time, I also utilized various data structures such as priority queues, queues, hash tables and arrays, and studied and analyzed how these data structures affect the performance and behavior of the algorithms.

BFS guarantees to find the shortest path, but it may be slow in complex grids. The greedy algorithm is faster to execute, but it cannot guarantee the shortest path. The A* algorithm strikes a balance between time and optimality, ensuring the shortest path and having high efficiency, but it uses more memory. The priority queue is the core data structure for both A* and the greedy algorithm, enabling efficient selection of the next expanded node based on the heuristic function. The queue is used in BFS to expand nodes layer by layer, ensuring the shortest path, but it is less efficient than priority-based queues in large-scale graphs. Hash tables and sets are used to store the states of nodes, facilitating quick search and update. Arrays and sets are used to store map data or track visited nodes, although they have high access efficiency, they consume a large amount of memory.

Through this assignment, I have a deeper understanding of the relationship between algorithms, data structures, and efficiency. In many cases, there is a trade-off between time and space complexity. For example, the A* algorithm uses more memory to improve search efficiency, while BFS, although it guarantees the shortest path, may consume more computing resources. Therefore, choosing the right algorithm and data structure is crucial to optimizing performance. Understanding these relationships is especially important in practical applications because the performance requirements and data scales in different scenarios vary greatly.