Algorithms, Data Structures and Efficiency

Coursework: Designing a Small Game with Algorithms and Data Structures

Yuyang ZHANG

20514470

scyyz26@nottingham.edu.cn

April 13, 2025

School of Computer Science

University of Nottingham Ningbo China

1 Introduction

Treasure Hunt is a 2D grid-based game that tests the player's navigation and decision-making abilities in a randomly generated environment. The game is implemented in Java (JavaSE-23) and places the player on a 20x20 2D grid map, with the main goal of finding three hidden treasures while avoiding obstacles. Maps are randomly generated during each game to ensure that every game is a unique experience.

Players move their characters in the map in four directions: up, down, left, and right. In addition, the game uses a scoring system. The player's initial score is 100 points. Each move consumes 1 coin and the related score is reduced by 1 point. To aid navigation, the game includes a "hint" function that can show the next step leading to the treasure, but each use costs 3 coins and 3 points to reduce the related score. If the player hits a wall, they lose 10 coins and the related score is reduced by 10. The game ends when the player successfully finds all three treasures. The goal of the game is to find all three treasures with the least amount of gold while effectively managing resources and avoiding obstacles.

The goals of this project were twofold: to design an engaging game, including algorithms and data structures, and to analyze its efficiency in terms of time and space complexity. This report provides an overview of the design and implementation of the game, as well as the algorithms and data structures used for the game implementation, and evaluates the challenges and trade-offs.

2 Design and Implementation

In the game, a 20x20 2D array is used to represent the game map, where each grid represents a position in the game, such as an path, an obstacle, a treasure, and a player's position. Representing the map in a two-dimensional array is not only clear but also easy to implement, and can effectively manage the state of each position.

- **Path**
  When the map is initialized, the path is represented by the default value 0, which means that the player can move freely on the grid with a value of 0. The represen-

tation of the empty path is simple and intuitive, which helps to easily distinguish different types of grids when generating obstacles and treasures later.

- **Obstacle**

  Obstacles are represented by OBSTACLE_VALUE, which has a value of 1. In the game, obstacles are not traversable. The program uses the isValidObstaclePosition() method to check whether the position is valid each time an obstacle is placed. This method uses the BFS algorithm to ensure that obstacles do not make certain areas of the map inaccessible, thereby ensuring that the location of the treasure can always be found by the player. Only positions that meet the conditions will be set to 1, indicating that the grid is occupied by an obstacle and the player cannot pass through these positions. The random distribution of obstacles increases the challenge of the game, forcing players to carefully choose their paths.

- **Treasure**

  Treasures are represented by TREASURE_VALUE, which has a value of 2. When the map is generated, treasures are randomly placed on empty path grids. Whenever a treasure is successfully placed, the corresponding position is assigned a value of 2, indicating that the grid is the location of the treasure, and the program ensures that the treasure does not overlap with obstacles. There are 3 treasures, and the player's goal is to find them.

- **Player**

  To ensure the simplicity and intuitiveness of the game, the player is fixed in the grid in the upper left corner of the game interface.

According to the above description, different elements can be clearly distinguished in the map: path (0), obstacles (1), treasure (2) and the player's position. This structured representation method ensures the readability and ease of operation of the map, and also provides a clear foundation for subsequent path finding, map generation and game logic.

2.1 Map generation

In my design, the game map is represented by a 2D array, where each cell represents a location on the map, such as a path, obstacle, or treasure. When the map is generated, the program randomly generates the location of the obstacle and verifies whether the obstacle is valid through the isValidObstaclePosition method. This method uses the BFS algorithm to ensure that the placement of obstacles does not make other areas of the map unreachable, avoid blocking the player's path, and ensure that the player can find the path to the treasure from the starting position. At the same time, the location of the treasure is randomly placed on the empty path grid and ensured not to overlap with obstacles, so that the treasure can be found by the player.

2.2 Pathfinding

- **BFS algorithm**

  Used to find the shortest path, it traverses layer by layer through a queue and uses visit markers to avoid repeated visits to grids. The BFS algorithm is suitable for path finding in games because it can guarantee the shortest path from the player's starting position to the treasure.

- **A* algorithm**

  Combines BFS and heuristic estimation, and optimizes the search process by using Manhattan distance as a heuristic function. The A* algorithm not only considers the path cost from the starting position to the current position, but also combines the distance to the target treasure, which can find the path more efficiently, especially in complex maps.

2.3 Treasure Placement

The location of the treasure is determined by traversing the array, ensuring that the treasure can be randomly placed in a valid location and does not overlap with obstacles. This process effectively avoids the treasure being placed in an unreachable area and ensures the player's gaming experience.

In conclusion, the combination of algorithms and data structures effectively supports map generation, path finding, and treasure placement. It not only ensures the smooth operation of the game logic, but also enhances the challenge and fun of the game, ensuring that players can find treasures through appropriate paths and successfully complete the task.

3 Efficiency Analysis

3.1 Time Complexity

The time complexity analysis mainly involves the efficiency of map generation and path finding (BFS and A*).

3.1.1 Map Generation

- **Obstacle**

  Obstacle generation is done by randomly selecting and verifying the position is valid using the isValidObstaclePosition() method. Each time an obstacle is placed, the isValidObstaclePosition method uses the BFS algorithm to traverse the entire map to verify the validity of the position. Since the time complexity of BFS is $O(M^2)$, the time complexity of placing each obstacle is $O(M^2)$, where M is the size of the map. Assuming that N obstacles need to be placed, the total time complexity is $O(N * M^2)$.

- **Treasure**

  Treasure generation only requires randomly selecting a path location for placement, and the time complexity of placing a treasure each time is $O(1)$. Assuming there are T treasures to be placed, the total time complexity of treasure generation is $O(T)$.

Pathfinding

- **BFS**

  BFS is used to find the shortest path to the treasure from the starting point. In the M*M game map, in the worst case, BFS needs to visit all grids on the entire map, so the time complexity is $O(M^2)$, where M is the edge length of the map.

- **A***

  A* algorithm combines heuristic estimation (Manhattan distance). A* needs to maintain a priority queue, where the time complexity for each node is $O(logM)$.

Since there are at most M^2 nodes, the total time complexity of A* is $O(M^2 * logM)$, where M is the edge length of the map.

Therefore, based on all the above, the overall time complexity is O(N * M² + T + M² * log M), where N is the number of obstacles, M is the size of the map, and T is the number of treasures.

3.2 Space Complexity

The space complexity analysis mainly involves the space requirements of map generation, path finding.

3.2.1 Map Generation

- **Map**

  The map uses a two-dimensional array to store the contents of each grid. Therefore, the space complexity of the map is $O(M^2)$, where M is the size of the map.

- **Onstacle**

  Each obstacle requires a certain amount of space, so its space complexity is $O(N)$, where N is the number of obstacles.

- **Treasure**

  Each treasure requires a certain amount of space, so its space complexity is O(T), where T is the number of treasures.

3.2.2 Pathfinding

- **BFS**

  BFS uses a Boolean array isVisit[][] to mark visited cells, and its space complexity is $O(M^2)$. In addition, BFS uses a queue to store nodes to be visited. The size of the queue is at most the total number of cells in the map, so the space complexity of the queue is also $O(M^2)$.

- **A***

  The A* algorithm uses a priority queue to manage the nodes to be explored, and its space complexity is $O(M^2)$. In addition, hash maps need to be stored, and their space complexity is also $O(M^2)$.

Therefore, the overall space complexity is O(M^2). Although the storage of treasures and obstacles requires additional space, their space overhead is small.

3.3 Performance Impact

3.3.1 Algorithm

- **BFS**

  BFS is suitable for smaller maps, with a time complexity of O(M^2), and performs well in this case. For larger maps, BFS may be less efficient.

- **A***

  A* combines heuristic functions with BFS to find the shortest path more efficiently. Especially in complex maps, A* can significantly reduce the search space and improve search efficiency, especially when M^2 grows.

3.3.2 Obstacle

The number of obstacles directly affects the efficiency of path finding. The more obstacles there are, the more complex the path finding becomes, because more cells will be marked as impassable, resulting in a reduction in the search space. When using BFS, obstacles affect path exploration, while Algorithm A* reduces the impact of obstacles through heuristic estimation and improves the efficiency of path search.

3.3.3 Map

As the map size increases, the performance of path finding algorithms in particular will be affected. Although a 20x20 map can guarantee good performance, Algorithm A* will have a more obvious advantage if the map size increases to 50x50 or larger. Algorithm A* finds paths more efficiently through priority queues and heuristic estimation, avoiding unnecessary searches. BFS, on the other hand, may need to traverse more cells when the map size increases, resulting in reduced performance.

In my algorithm, I used the strategy of ”trading space for time” to improve efficiency, especially in the design of A algorithm* and BFS algorithm. By using data structures such as priority queues, hash tables, and access mark arrays, the efficiency of path finding is significantly improved. Although these optimizations increase space overhead, they effectively reduce calculation time, especially in the case of complex maps, significantly improving the performance of the game. This balance between space and time allows the game to maintain good performance on maps of different sizes, ensuring a smooth user experience.

4 Challenges and Trade-offs

4.1 Challenges

During the implementation of the game, several challenges were encountered, especially in the design of obstacle generation and path finding algorithms. Not only do we need to ensure the integrity of the map, but we also need to ensure the efficiency of the path finding algorithm and ensure the playability of the game by randomly generating maps.

- **Map Generation**

  During the map generation process, it is necessary to ensure that the random placement of obstacles does not block the player's path to the treasure. Because the obstacles are randomly placed, it is necessary to ensure that the placement of obstacles does not affect the player's reachable area, especially between the starting position and the treasure in the game. To solve this problem, I chose to use the BFS algorithm to check the validity of the obstacles. Although this method works, it also increases the complexity of obstacle generation because each obstacle needs to be verified by a BFS search.

- **Pathfinding**

  The game needs to ensure that players can find the shortest path from the starting position to the treasure. Choosing a suitable path finding algorithm is a challenge. Initially, the BFS algorithm was used for path finding. Although this method is simple and easy, it is inefficient in complex maps. To solve this problem, I chose to use the A* algorithm, which combines heuristic estimation (Manhattan distance) to optimize the search process.

4.2 Trade-offs

During the implementation process, multiple trade-offs were made between simplicity and efficiency, space and time complexity in order to balance efficiency and maintainability.

- **Simplicity and Efficiency**

  Initially, we chose the BFS algorithm for path finding because it is simple to implement and guarantees to find the shortest path. However, when the map becomes more complex, especially when there are multiple treasures, BFS becomes less efficient. To address this, we decided to use A* algorithm, which significantly improves path finding efficiency despite being more complex.

- **Space and Time Complexity**

  The strategy of ”trading space for time” was used to improve efficiency, especially in the design of A* algorithm and BFS algorithm. The efficiency of path finding was significantly improved by using data structures such as priority queues, hash tables, and access marker arrays. Although these optimizations increase space overhead, they effectively reduce calculation time, especially in the case of complex maps, significantly improving the performance of the game.

Throughout the implementation process, by selecting appropriate algorithms and data structures, I effectively balanced the relationship between simplicity and efficiency, time complexity and space complexity, ensuring that the game can run smoothly on maps of different sizes.

5 Conclusion

In this assignment, I designed and implemented a treasure hunt game, which included random map generation, placement of obstacles and treasures, and players finding the shortest path. In addition, I understood and applied algorithms and data structures to ensure efficient operation of the game and a good user experience.

Finally, through this assignment, I learned how to effectively combine algorithms and data structures to solve practical problems. When implementing path search, I deeply understood the different characteristics and application scenarios of BFS and A algorithm*, and how they affect game performance and user experience. At the same time, I also realized how to balance time complexity and space complexity when optimizing performance, and choose appropriate algorithms and data structures to ensure that the game can run smoothly in various scenarios.