University of Nottingham
UK | CHINA | MALAYSIA

Algorithms, Data Structures and Efficiency

Coursework:

Designing a Small Game with Algorithms and Data Structures

Yuyang ZHANG

20514470

scyyz26@nottingham.edu.cn

April 23, 2025

School of Computer Science
University of Nottingham Ningbo China

1 Introduction

Treasure Hunt is a 2D grid-based game where the goal for the user is to find 3 hidden treasures located on a 20x20 grid map while avoiding obstacles. The game tests the player's navigation and decision-making abilities and is implemented in Java (JavaSE-23). Maps including obstacles and treasure locations are randomly generated during each game to ensure that every game is a unique experience.

There are two goals for this project: 1) design an engaging game that leverages algorithms and data structures, and 2) analyze the efficiency of the game in terms of runtime 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

2.1 Game Rules and Features

Players can move their characters in the map in four directions: up, down, left, and right. In addition, the game has a score system. The player's initial score is 100 points. Each move reduces the player's score by 1 point. To aid navigation, the game includes a "hint" function that can show the next step leading to the nearest treasure, but each hint costs 3 points to reduce the related score. If the player hits a wall, they lose 10 points. Players get 20 points for finding a treasure. When the player reaches the boundary of the map, a pop-up will be displayed to remind the player that this is the boundary. The game ends when the player successfully finds all three treasures or when the player score drops below 0.

I have also designed additional features to improve the playability of the game. When the score is below 60 points, an alert will show up at the top of the map to recommend that the player get a hint. When the score is below 10 points, the player can click the magic bottle to add 20 points, but it is only allowed once per game. In addition, players can view the top 5 scores on the ranking page. Finally, for game developers, clicking the

space bar can show the shortest path for players to reach the treasure, which helps to detect whether the algorithm is effective. If the player has reached the edge of the map and wants to continue walking, a prompt will pop up to indicate the Boundary Tip. I also beautified the UI. I set the entire game interface to a grassy area, where obstacles and treasures are invisible. When the player walks through the grass, the grass will turn into flowers. This design not only makes the game visually pleasant, but also clearly marks the route the player has walked. In addition, I also made the corresponding beautification designs for the player, treasure, obstacles, the magic bottle, the start interface, and the end interface. Regarding the design of the start and end interfaces, I included four buttons on the start interface: Start, Info, Top 5 Ranking, and Exit. The Info page introduces the game rules. In the end interface, I designed three buttons: Restart, Exit, and Home. The Home button returns the user to the start interface.

2.2 Technical Implementation

In the game, a 20x20 2D int array is used to represent the game map, where each grid represents a position in the game, such as a path, an obstacle, a treasure, or a player's position. Representing the map in a two-dimensional array is easy to implement and allows us to manage the state of each position effectively.

- **Path**

  When the map is initialized, an empty path is represented by a constant PATH_VALUE, which is 0. This means that the player can move freely on a grid that has a value of 0. The representation 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 a constant OBSTACLE_VALUE, which has a value of 1. In the game, the player cannot move past an obstacle. The number of 15 obstacles are generated per game, and 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 player can reach the treasure. The

random distribution of obstacles increases the challenge of the game, forcing players to choose their paths carefully.

- **Treasure**

  Treasures are represented by a constant TREASURE_VALUE, which has a value of 2. When the map is generated, three treasures are randomly placed on empty path grids. The program ensures that the treasure does not overlap with obstacles.

- **Player**

  To ensure the simplicity and intuitiveness of the game, the player always starts at the upper left corner of the map, as shown in the game interface. The starting grid will not have an obstacle or a treasure.

According to the above description, different elements can be clearly distinguished in the map: path (0), obstacle (1), treasure (2), and the player's current position(represented by HEIGHT and WIDTH). 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.

2.3 Map generation

In my design, the game map is represented by a 2D int array, where each element of the array represents a location on the map, such as a path, obstacle, or treasure. When the map is created, the program first generates the obstacles and verifies that the obstacle does not make other map location unreachable from the starting position. After that, the location of the treasure is randomly placed on the empty path grid and is ensured not to overlap with obstacles.

2.4 Path Finding

- **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 and is used both in validating obstacle positions and determining the shortest path from the player's starting position to the treasure.

- **A* algorithm**

  Combining BFS and heuristic estimation, A* optimizes the search process by using Manhattan distance as a heuristic function. A* algorithm considers both the path cost from the starting position to the current position and the distance to the target treasure, which can find the path more efficiently, especially in complex maps.

2.5 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 implements the correct game logic and enhances the challenge and fun of the game, ensuring that players can find treasures through appropriate paths and successfully complete the task.

2.6 Data Structure

- **Array**

  Used to represent the layout of the game map, making it easier to manage the status of each grid. It are suitable for representing grid-type map structures and is very efficient in operation and access.

- **Boolean Array**

  In the *isValidObstaclePosition* method, use the *isVisit* Boolean array to mark whether each position has been visited.

- **Queue**

  In the *isValidObstaclePosition* method, use the queue list to store the map positions to be visited and visit these positions in the order of the BFS algorithm.

- **List**

  Used to store treasure locations on the map. *treasureList* is an ArrayList that stores all items of type Treasure.

- **Priority Queue**

  Used in the A* algorithm. The *openSet* is a priority queue that stores the nodes to be processed, including the current position and the corresponding heuristic score. The priority queue is sorted according to the f-score (g + h) of the nodes to ensure that the nodes with the lowest f-score are processed first.

- **Hash Map**

  Used in A* algorithm.

  1. *cameFrom* is used to store the predecessor node of each node to help rebuild the path later.
  2. *gScore* is used to store the g-score of each node (the actual distance from the starting point to the node).

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 and verify the position is valid. The method creates a queue and starts at one of the corner tile and expands in four directions. For each valid next step(not an obstacle or outside of map), each step is added to the queue. Because queue is first in first out, the nearest tiles will be explored first hence it's BFS. As a result, all tiles on the map will be traversed and the time complexity is $O(M^2)$, where M is

the width/height of the map(in our case 20). Assuming that N obstacles need to be placed(in our case 15), 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 $O(T)$.

3.1.2 Pathfinding

- **A***

  When a player hits the hint button or in developer mode, getShortPathByAStar() is called to return the shortest path to each treasure. A* algorithm uses heuristic estimation. The method starts with a nested loop to locate the three treasures on the map, which costs $O(M^2)$ where M is the height/width of the map(20). Then we enter the core loop for A*, which in the worst case(every tile is visited) will run $O(M^2)$. A* maintains a priority queue and each tile is represented as a node in the queue that has a calculated Manhattan heuristic distance(hScore) and cost to reach the node(tentativeG). In each while loop iteration, we may compute and enqueue $O(M^2 * T)$ nodes (insert T nodes for every valid neighbor, T is number of treasures). Therefore, the total time complexity of A* is $O(M^2 * T * log(M^2 * T)$. However, since we know $T = 3$ we can simplify the runtime to $O(M^2 * log(M^2)$.

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.

- **Obstacle**

  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 it may hold $O(M^2 * T)$ nodes in the worst case. In addition, we have hash maps with space complexity $O(M^2)$. Therefore, since we know $T = 3$, the overall space complexity is $O(M^2)$.

3.3 Performance Impact

3.3.1 Algorithm

- **BFS**

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

- **A***

  For the runtime and space complexity I calculated the worst case. Even though A* has a slower runtime in the worst case compared to Dijkstra, I still chose it because, in reality, the heuristic estimation would help locate the treasure more efficiently, unlike Dijkstra, which has a relatively fixed runtime. In complex maps, A* can significantly reduce the search space and improve search efficiency, especially when M^2 grows.

  Although there is still room for optimization in the time complexity of the current implementation of the A* algorithm, since we have three treasures and need to find the shortest path and simulate obstacles at the same time, the time complexity of this version is the best method we can think of through different data structures.

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.

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.