University of Nottingham
UK | CHINA | MALAYSIA

COMP2069 Algorithms, Data Structures and Efficiency

Designing a Small Game with Algorithms and Data Structures

Author
Mithilesh TEW
hfymt5
20509703

Submitted on 25th April 2025

School of Computer Science
University of Nottingham Ningbo China

Contents

1 Introduction

The "Treasure Hunt" game is an interactive grid-based adventure that challenges the player to navigate a character through a randomly generated map, locate hidden treasures, and avoid obstacles while managing their in-game score. Built using Java, the game is designed to challenge the player's strategic thinking and problem-solving skills by requiring efficient pathfinding and resource management. The game takes place on a 20×20 grid, where the player controls a character that can move in four directions, which is up, down, left, and right. The objective is to find all three hidden treasures while avoiding walls (obstacles) that penalize the player upon collision. The player starts with a score of 100, which decreases with each move or penalty incurred. The map is randomly generated at the start of each game, ensuring a unique layout every time. Key features of the game include:

- Map Generation: The map is procedurally generated with walls and treasures placed randomly, ensuring each playthrough offers a unique experience. Walls are strategically placed to create a challenging but solvable.
- Player Movement: The player can move in four directions (up, down, left, right), with each move costing 1 point from the initial score of 100. Colliding with a wall deducts 10 points and block movement.
- Treasure Collection: The game ends successfully when all three treasures are found.
- Hint System: Players can use two hint algorithms to reveal the shortest path to the nearest treasure. Each hint costs 3 points, encouraging strategic use.

2 Representation of Game Map and Other Entities

The "Treasure Hunt" game is based on a structured representation of the game world, which includes multiple things including the player, treasures , obstacles, and the map itself. This design assures efficient gameplay dynamics while relying on proper data structures and algorithms to maintain functionality and efficiency. The following is a full description of how each component is represented and controlled within the game.

2.1 Game Map Represention

The base of the game is based on a 20×20 grid, which serves as its playing field. This grid is implemented using a two-dimensional array that stores the state of each cell. To improve clarity and maintainability, the game uses enumerations (enums) to define different cell types and their visibility status.

The grid is represented by two primary 2D arrays, which is the `CellType[][] grid` and `Visibility[][] visibility`. `CellType[][] grid` is 2D array storing the type of each cell (empty, wall, treasure, player, or path). While `Visibility[][] visibility` is parallel 2D array tracking whether a cell is visible or hidden to the player. This dual-array way enables the game to effectively manage both the physical layout of the map and the player's exploration status.

When a new game starts, a random map is generated or loaded from JSON file that pre-generated by using `MapGenerator`. The initialization process involves reading the map data and populating the `grid array` by converting numerical values into their corresponding `CellType (e.g., 0 → EMPTY, 1 → WALL, 2 → TREASURE, 3 → PLAYER)`. Lastly, setting initial visibility because most cells start as `HIDDEN`. Only for the player's starting position is immediately `VISIBLE`. This method ensures that each game session has a different layout while maintaining consistency in gameplay rules.

2.2 Player Representation

The player is a central entity in the game. It is represented by Point `playerPos`, a Java `Point` object storing the player's current `(x, y)` coordinates on the grid. For movement handling, the `PlayerMovement` class processes keyboard inputs and updates player's position accordingly. For each move by the player deducts 1 point from the player's score, encouraging efficient navigation. If the player attempts to move into a wall, the movement is blocked, and the score is reduced by 10 points. If the player manages to collect a treasure, it will increase the treasure counts. As the player moves, adjacent cells are revealed, resulting in a dynamic exploration experience.

2.3 Treasure Representation

Treasures are represented by `CellType.TREASURE`. Three treasures are randomly placed on the map, but their placement is checked to ensure that they are accessible

from the player's starting point. During map generation, a Breadth-First Search (BFS) algorithm ensures that a path exists between the player's starting place to each treasure. If a treasure is discovered to be unreachable, the map generator modifies the layout by removing obstructing walls. When the player steps on a treasure, the treasure count increases and the cell changes to empty.

2.4 Obstacle Representation

Obstacles are represented by `CellType.WALL`. Walls are not placed at random, but they are arranged in clusters and maze-like patterns that resemble natural obstacles. The generator uses a combination of scattered walls, small clusters, and structured maze portions to keep the map interesting and navigable.

3 Algorithms and Data Structures Used

The "Treasure Hunt" game makes use of numerous fundamental algorithms and data structures to perform map generation, pathfinding, and treasure placement. These solutions enable efficient gameplay while also demonstrating fundamental computer science principles.

3.1 Map Generation

The map generation process uses a **procedural generation**. The map generation follows a three-phase process. A blank 20×20 grid is created with all cells set to walkable (`EMPTY`). The player's starting position is fixed at (0,0), providing a consistent reference point for pathfinding validation. Walls are distributed using a weighted randomization system to create diverse navigation challenges. Scattered walls (80%) are placed individually and randomly, forming basic obstacles. Structural walls (10%) are L-shaped, acting as natural barriers to guide movement. Clustered walls (10%) are dense 3×3 blocks, creating high-difficulty zones. For accessibility verification, an **Breath-First Search (BFS)** algorithm systematically checks that all treasure locations retain path connection from their starting point. A **BFS** algorithm systematically checks that all treasure locations retain path connection from their starting point. When unreachable treasures are found, the **A* Search** algorithm removes obstructive walls to provide the most efficient path.

The game uses key data structures for map management. A `2D array (int[][] grid)` holds the initial map layout before converting to cell types. `ArrayLists of Points` track potential treasure and obstacle locations. A `queue (LinkedList)` is used for BFS in reachability checks, exploring adjacent cells to ensure map connectivity.

3.2 Pathfinding

The game implements two pathfinding algorithms for hints.

Breadth-First Search (BFS) algorithm serves as the game's primary pathfinding method for hint generation. This algorithm expands uniformly in all directions from the player's current position, using a queue to manage the frontier of exploration. A hash map tracks the traversal path, allowing for quick backtracking once the target treasure has been identified. With a time and space complexity of O(V + E), BFS algorithm consistently finds the shortest path in an unweighted 20×20 grid, ensuring consistent performance. The **A* Search** implementation introduces heuristic-based optimization. A* enhances pathfinding with a Manhattan distance heuristic, guiding the search toward the goal efficiently. It uses a min-heap (priority queue) to explore the most promising paths first. While maintaining O(V + E) complexity, A* typically reduces the search space by 40–60% compared to BFS in practice.

The game uses various data structures to support efficient pathfinding. A `Queue (LinkedList)` is used in BFS to manage the order of cell exploration. For A*, a `Priority Queue` selects the most promising paths first based on heuristic cost. `HashMaps` help reconstruct the path after reaching the goal, while `Sets (HashSets)`

track visited nodes to avoid redundant exploration.

3.3 Treasure Placement

Treasures are placed using a **reachability-guaranteed** method. In the candidate selection phase, treasures are limited to the far half of the map. Empty cells are collected into a list of potential spots. This encourages players to traverse a significant portion of the environment rather than finding rewards too early. Once possible locations have been discovered, a reachability verification stage is carried out using **BFS**. This ensures that each treasure is accessible from the starting position. If a location turns out to be unreachable, the system performs a path correction method. This entails performing an **A*** search to find the smallest blocked path and selectively removing the fewest number of walls necessary to construct a valid route.

To ensure fairness and accessibility, the treasure placement system relies on several key data structures and algorithms. `ArrayLists` are used to hold possible treasure placements, allowing for dynamic management of candidate spots. A `queue ('LinkedList')` is used during BFS to validate reachability, ensuring that each treasure can be accessible from the beginning point. Additionally, **graph traversal methods** are utilised to ensure that no treasure is isolated, resulting in a playable and well-connected map.

4 Time and Space Complexity of Algorithms

Big-O Notation, known as **O()**, describes how an algorithm's runtime or memory usage grows as the input size increases. It gives a high-level overview of the algorithm's efficiency, particularly when dealing with huge data sets.

In graph-based algorithms, such as grid pathfinding, **V** stands for **Vertices**, which represent the nodes in our search space. In a 20×20 grid, this means there are 400 cells, and each cell is treated as a vertex.

E indicates **Edges**, which represent the possible connections between these vertices. Edges in a grid are moving channels between adjacent cells that allow you to go from one node to the next. Understanding V and E is critical for assessing the complexity of graph algorithms.

4.1 Algorithms Used

Breadth-First Search (BFS)
Time Complexity: O(V + E)
Space Complexity: O(V)

A Search*
Time Complexity: O(E + V log V)
Space Complexity: O(V)

4.2 Algorithms Not Used

Depth-First Search (DFS)
Time Complexity: O(E + V)
Space Complexity: O(V)

Dijkstra's Algorithm
Time Complexity: O(E + V log V)
Space Complexity: O(V)

Greedy Best-First Search (Greedy BFS)
Time Complexity: O(E + V log V)
Space Complexity: O(V)

Uniform Cost Search (UCS)
Time Complexity: O(E)
Space Complexity: O(V)

5 The Impact of Algorithmic Choices Game's Performance

"Treasure Hunt"'s pathfinding and map generation algorithms were chosen carefully, and they have a significant impact on the game's technical performance and player experience. The game achieves an optimal balance between computational speed, memory use, and gameplay quality by including Breadth-First Search (BFS) and A* algorithms while purposefully rejecting alternatives such as Depth-First Search (DFS) or Dijkstra's algorithm.

The use of BFS for core pathfinding operations results in several significant performance benefits. BFS, with its O(V + E) time complexity (equivalent to $O(N^2)$ for a N×N grid), ensures complete exploration of the 20×20 game grid within 1-3 milliseconds. This makes it excellent for map validation during procedural creation, where dependability is more crucial than absolute speed. The algorithm's level-by-level exploration naturally suits grid-based accessibility systems and assures players always receive accurate information about available treasures.

The hint system's use of A* search displays much more advanced optimisation. While preserving O(E + V log V) complexity in the average case, the use of Manhattan distance heuristics reduces the search space by 40-60% when compared to BFS. This corresponds to hint generating durations of only 0.5-2 milliseconds, representing a significant quality-of-life increase for players who frequently use this feature. The algorithm's ability to prioritise possible paths using heuristic advice makes it ideal for real-time hint generation during games.

Algorithmic decisions have a direct impact on player experience in several subtle but significant ways. BFS's promise of finding the shortest path assures that map validation always provides fair, solvable levels, which is essential for player satisfaction. When paired with the A* hint system, players receive quick and statistically optimal advice, resulting in a rewarding learning curve as their navigation skills develop.

In this circumstance, rejecting alternatives such as Greedy Best-First Search is very wise. While that approach has a similar time complexity as A*, the lack of optimality guarantees may annoy players with misleading clues. Similarly, while Dijkstra's approach may technically work, it would have the same time complexity as A without the advantage of heuristic optimisations, resulting in excessively slow hint generation.

6 Challenges

The development of Treasure Hunt's algorithmic systems presented several technical and design challenges that required careful problem-solving. These hurdles emerged primarily in pathfinding optimization, map generation reliability, and system integration.

6.1 Pathfinding Algorithms

Creating an appropriate heuristic for the A* algorithm posed various obstacles. Initially, the Euclidean distance was used, which accidentally favored diagonal paths and resulted in poor grid routes. Attempts to improve this by employing overstated heuristics put at risk the algorithm's ideal performance. The problem was rectified by moving to Manhattan distance, which was more consistent with grid-based movement rules, resulting in more accurate pathfinding.

6.2 Map Generation

One key problem was ensuring treasures reachability. Early versions frequently placed treasures in isolated regions enclosed by walls, making them inaccessible. The validation system at the time was unable to detect such unreachable zones. This issue was resolved by adding a multi-stage validation procedure that used Breadth-First Search (BFS) to check connectivity, as well as a fallback correction utilising A* pathfinding to ensure that all valuables were accessible.

Another challenge was to balance the distribution of obstacles. Initially, random wall placements produced maps that were either too easy, with little complexity, or too tough, with thick clusters that completely barred movement. To remedy this, a hybrid obstacle placement method was devised, which included scattered and clustered walls. This method resulted in maps with a more appealing and playable balance of challenge and navigability.

6.3 Performance Optimization

One of the most important factors in performance optimisation was real-time hint responsiveness. The hint generation was noticeably slow in early implementations, particularly when doing complex pathfinding tasks. Smooth gameplay was impacted by memory spikes that accompanied these delays. By restricting the search depth and implementing path caching, the latency and memory consumption were greatly decreased.

The grid representation was another source of performance issues. The object-heavy cell structures used in the early designs resulted in a large memory overhead per cell. Primitive arrays and bitmasking were used in the system's restructuring to increase efficiency. In addition to lowering memory usage, this increased access speed, making the game experience more responsive and fluid overall.

6.4 Gameplay Integration

There were difficulties while integrating gaming features, especially with visual path display. Early iterations suffered with performance decreases brought on by frequent full-grid redraws and flickering during path changes. Differential repainting, in which only the altered cells were redrew, was used to fix this. This method produced a smoother visual experience while lowering rendering overhead.

There were also synchronisation problems with the score system. When movement expenses and pathfinding deductions conflicted, particularly when several clues were asked quickly after one another, race circumstances developed. Atomic score updates and an operation queuing scheme were implemented to solve this. Even with quick user interactions, these adjustments guaranteed scoring stability and dependability.

7 Trade-offs

The implementation of Treasure Hunt required careful consideration of competing priorities, forcing deliberate trade-offs between simplicity and efficiency, time and space complexity, and optimality and performance.

7.1 Simplicity vs. Efficiency

I combined both Breadth-First Search (BFS) and A* in our pathfinding approach to strike a compromise between ease of use and effectiveness. Because of its simplicity and dependability, it employs a straightforward queue structure and consistently finds the shortest path, making it simple to implement and debug. BFS was selected for map validation. A*, on the other hand, was chosen to offer in-game tips. Despite being more complicated, A*'s heuristic-based optimisation allows it to exceed BFS in terms of speed. Alternatively, using only BFS would have simplified code maintenance but made hint generation noticeably slower during complex searches.

7.2 Time vs. Space Complexity

I intentionally balanced time and space complexity in our pathfinding system to maximise maintainability and performance. Despite using more memory, A* was selected for hint generation over BFS because it offered 40–60% faster search times. The performance increase outweighed the additional memory, which was mostly used for the priority queue and heuristic storage.

By copying the map instead of changing it right away, i managed to tolerate $O(N^2)$ memory use during validation for map construction. The duplicated map improved code maintainability by enabling cleaner validation logic and simpler debugging, even if an in-place method might have reduced memory.

7.3 Optimality vs. Performance

I carefully weighed performance assurances against optimality in my implementation, especially in path correction and heuristic design. I decided to use the Manhattan distance as the heuristic for A*. It is perfectly admissible, guaranteeing that all paths stay optimal, even though it is a little slower than inadmissible alternatives. This choice demonstrated my preference for path accuracy over slight performance improvements.

I also achieved a balance between accuracy and simplicity when dealing with inaccessible treasure. Even though BFS is slower, I chose it for initial validation because it is simple. A* provided focused and effective modifications, but it was only used when correction was required.

8 Conclusion

The development of the game offered priceless insights into the real-world implementation of data structures, algorithms, and efficiency optimisation. I discovered that algorithm selection needs to find a balance between theoretical efficiency and practical usability by putting BFS for map validation and A* for pathfinding hints into practice. While A*'s heuristic optimisations produced quicker, more player-friendly tips, BFS made sure that maps were generated correctly.

Decisions over data structures, especially the move from object-oriented cells to primitive arrays, brought attention to how crucial cache and memory speed are. To provide fluid gameplay, this trade-off between runtime performance and coding simplicity was essential.

My understanding of algorithmic trade-offs has also improved because of the research. In fact, simpler, well-optimized methods (like BFS and A*) might produce better results than theoretically optimal ones, which frequently add needless complexity.

In the end, "Treasure Hunt" proved that efficiency is about producing a smooth player experience rather than just speed or memory. My capacity to identify issues, put solid solutions in place, and make well-informed design choices was enhanced by the project. These abilities will serve as a roadmap for my future work in algorithm-driven apps.