Using Greedy Best-First Search Algorithm for Finding the Optimal Path in Solving the Game “Helltaker”

Zaki Yudhistira Candra - 13522031
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (stei): 13522031@std.stei.itb.ac.id

***Abstract*—The way we traverse the world today**

The Pathfinding algorithm is one of the most prominent algorithms discovered by computer scientists. It helps us travel or solve problems efficiently without requiring much computing overhead. One of the well-known pathfinding algorithms is the Greedy Best First Search (GBFS) pathfinding algorithm. It uses heuristic to determine which node should be processed first. This paper will discuss the application of the GBFS algorithm in a simple tile-based 2020 game called “Helltaker”. By analyzing the game's levels and implementing the GBFS algorithm, this study aims to enhance the understanding of this algorithm and provide a concrete example of its real-world application.

Keywords—Helltaker, Video Games, Pathfinding, Algorithm, GBFS

I. Introduction

"Helltaker," developed by Polish game developer Łukasz Piskorz (vanripper), is a popular indie puzzle-adventure game released in May 2020. The game quickly gained a significant following due to its unique blend of challenging puzzles, engaging storyline, and charming character designs. Players navigate through a series of increasingly difficult levels to assemble a demon ‘harem’, each level requiring strategic moves to solve within a limited number of steps. If the player failed to fulfil the constraints given by each level, the player would have to restart the game.

The tile-based nature of the video game makes it an interesting and perfect subject for studying and implementing pathfinding algorithms. Since each level is a puzzle, solving them efficiently often requires not just intuition, but also an understanding of optimal pathfinding strategies. This paper aims to explore the application of the Greedy Best-First Search (GBFS) algorithm in finding the optimal path for solving the puzzles in "Helltaker." The GBFS algorithm, known for its simplicity and efficiency in certain scenarios, prioritizes paths that appear to be leading most directly to the goal.

The Greedy Best-First Search (GBFS) algorithm is chosen over other pathfinding algorithms such as A* and Uniform Cost Search due to the tile-based nature of "Helltaker." In this game, each tile or node has a fixed and uniform distance from its neighbors, making the heuristic used in algorithms like A* for calculating the shortest path to a node unnecessary and inefficient. GBFS, which focuses on exploring the most promising nodes based on a heuristic that estimates proximity to the goal, is well-suited for the consistent structure of the game's puzzle grids. More details will be discussed in the following chapter.

The primary motivation behind this study is to provide a deeper understanding of how a pathfinding algorithm can be applied to a game design, particularly in tile-based puzzle games. By analyzing the effectiveness of the GBFS algorithm in "Helltaker," this paper seeks to contribute to the broader field of game development. Additionally, it serves as a practical example of algorithm application, bridging the gap between textbooks concepts and real-world application.

Disclaimer: The game content discussed in this paper, including the puzzles and character designs, belongs to the original creator, Łukasz Piskorz (vanripper), and is not the work of the author. The author’s work only covers the analytical and applicational part of the paper.

II. Key Concepts

Before dicing deeper into this paper, we must first understand the relevant key concepts of this paper. There are 3 main concepts that requires a good understanding prior to reading this paper: Pathfinding algorithms particularly GBFS (Greedy Best-First Search), the core mechanics of the game “Helltaker”, and the problem identification and its step-by-step solution. Familiarity with these concepts will provide the necessary foundation for comprehending the discussion that follows.

A. Path Finding Algorithm

In the realm of computer programming, there are three main algorithms that are designed for pathfinding: The Uniform Cost Search (UCS) Algorithm, Greedy Best-First Search Algorithm (GBFS), and the AStar or A* algorithm which is the combination of the two. These algorithms are the extension of the Breadth First Search algorithm or the BFS algorithm. Used to traverse a tree-structured node-based data.

What differentiates these algorithms and the generic Breadth First Search (BFS) or the Depth First Search (DFS) algorithm is the usage of heuristics.

Heuristics are problem-solving methods or strategies that utilize practical and efficient approaches to finding solutions. In the context of pathfinding algorithms, a heuristic is a function that estimates the cost or distance from a given node to the goal node. Heuristics guide the search process by prioritizing nodes that are likely to lead to the goal more quickly. These estimates are not guaranteed to be accurate, but they are designed to be computationally inexpensive and to provide good enough approximations to make the search process more efficient. In the Greedy Best-First Search (GBFS) algorithm, the heuristic function helps to determine the most promising nodes to explore based on their estimated proximity to the goal.

The heuristic function of GBFS is denoted as:

$$g(x) = some\ function$$

While $g(x)$ denotes the estimated shortest path to the goal node. The concrete implementation of the heuristic function will be further discussed.

An example of the implementation of $g(x)$ could be inferred from the figure below.

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Figure 2.1. the table of straight-line distance estimates to Bucharest to the corresponding city (Source: Rinaldi Munir)

The Greedy Best-First Search (GBFS) algorithm processes nodes based on their estimated proximity to the goal, as determined by the heuristic function. The order in which nodes are processed and their prioritization in the queue depend entirely on this heuristic. For example, in Figure 2.1, the algorithm prioritizes processing the node with the shortest estimated distance to Bucharest, placing the closest city at the top of the queue.

B. Helltaker Game Mechanics

As mentioned in the previous sections, "Helltaker" is a tile-based puzzle game that requires the player to navigate through a maze-like level. A player starts at a designated position and must reach the end point or finish line within a fixed number of steps or movements. If the player exhausts the allocated steps before completing the level, they lose and would have to restart the level.

Figure 2.2. Helltaker first level game overview (Source: Writer's archive)

Figure 2.2 illustrates the first level of the game. The yellow circle at the top right represents the starting point, while the orange circle indicates the endpoint or finish line. Players must navigate their character from the yellow circled to the orange circled tile to complete the level.

Players navigate their character throughout the level using the W, A, S, D or The Arrow Up, Down, Left, and Right key on their keyboard. The W or Up moves the character north, S or Down moves the character to the south, the A or Left to the west, and the D or the Right to the east.

Players could only perform their movement on the set boundary of the level, indicated by the red floor and the void and debris. There are other elements of the game which will be elaborated in the next section.

Each movement decreases the moves counter by one, depending on the object encountered. The moves counter is situated at the bottom-left side of the screen. The roman number on the bottom-right side indicates the level of the game, figure 2.2 is situated in the first level of the game (I).

There are several game objects in the "Helltaker" game. Each object introduces a unique interaction with the player, enhancing the game's difficulty and puzzle elements.

Figure 2.3. the player object (Source: Writer's archive)

Figure 2.3 illustrates the player object; it gives the player information about the current whereabouts of the player. The player must move the player object to the finish line.

Figure 2.4. Demon object (Source: Writer's archive)

Figure 2.3 illustrates the demon object, which serves as one of the obstacles in the game. Players cannot move past the demon object directly; however, they can push it in the direction they are moving. If the demon object is pushed into a wall or any other object, it will be destroyed. Each push or destruction of the demon object costs the player one move, adding a layer of strategy to the gameplay as players must decide when and how to interact with these obstacles efficiently.

Figure 2.5. Stone object (Source: Writer's archive)

Figure 2.5 illustrates the Stone block object. This object behaves similarly to the demon object, with the key difference being that it cannot be destroyed. When a player attempts to push a Stone block and there is another object behind it in the direction of the push, the Stone block cannot be moved or destroyed. Despite this, the action still costs the player one move.

Figure 2.6. Debris (Source: Writer's archive)

Figure 2.6 illustrates the debris object. This object serves as a boundary for the player and cannot be interacted with or moved. In simpler terms, it effectively blocks the player's path. Blocked movement from this object does not cost the player a move.

Figure 2.7. Demon woman object (Source: Writer's archive)

Figure 2.5 illustrates the demon woman object. Each level has their own unique demon woman object. This object serves as the finish line or end point for each level. The player must move their character to the tile next to the demon woman object to complete the level.

Figure 2.8. Spike object (Source: Writer's archive)

Figure 2.8 illustrates the spike object, which is introduced in the second level of the game. The player can pass through this object as long as there is no stone object on top of it. When the player enters a spike tile, their move count decreases by two points, and exiting the tile decreases the move count by one point. Additionally, when demon objects are pushed onto a spike tile, they are destroyed.

Figure 2.9. Lock object (Source: Writer's archive)

Figure 2.9 illustrates the lock object, which is introduced in the third level of the game. Like the debris object, the lock object blocks the player's path to the finish line. However, it can only be removed if the player has retrieved the key object, allowing them to unlock and pass through it.

Figure 2.10. Key object (Source: Writer's archive)

Figure 2.10 illustrates the lock object, which is introduced in the third level of the game. The player must first reach this key object to unlock or remove the lock object that is preventing the player from finishing the level.

C. Manhattan Distance

The Manhattan distance, also known as the city block or cab distance, is a measure of the distance between two points in a grid-based system where movement can only occur horizontally or vertically, never diagonally.

The Manhattan distance between two points P_1 (x_1, y_1) and P_2(x_2, y_2) is calculated as the sum absolute distance between P_1 and P_2.

$$Mahattan\ distance = |x_1 - x_2| + |y_1 - y_2|$$

Visually, this corresponds to the distance a taxi would have to travel to reach one point starting from the other, moving only along the grid lines. This distance metric is used in this context since the player can only move horizontally or vertically, diagonal moves are not allowed in this game. Also, this metric is used to calculate the distance between game objects, particularly between the player and the finish line.

D. Heuristic Used

The heuristic employed in solving this problem is based on the Manhattan distance between the player's current position relative to the finish line minus the remaining moves. This ensures that the algorithm avoids getting stuck in loops and gives priority to positions with the fewest remaining moves. The heuristic is expressed in the following formula.

$$g(x) = |x_1 - x_2| + |y_1 - y_2| - moves\_left$$

The generated nodes will be sorted and processed based on their heuristic values, which represent the sum of the shortest distance from each node to the finish line and the remaining number of moves. Nodes with higher remaining moves but shorter distances to the finish line will be prioritized for processing.

III. Problem Identification

A. Game Levels

This paper primarily focuses on the implementation of the Greedy Best-First Search (GBFS) algorithm to solve the puzzle-based levels in the game. "Helltaker" features a total of 11 levels, with 10 of them being puzzle-based. For simplicity of the discussion, this paper will specifically analyze the first, second, third, and fourth levels, showcasing the application of the GBFS algorithm in solving these initial challenges.

Figure 3.1. First level (Source: Writer's archive)

The initial level of "Helltaker" features a straightforward layout designed to familiarize players with the game. It includes only block and demon objects, with a total of 23 available moves.

Figure 3.2. Second level (Source: Writer's archive)

The second level of "Helltaker" features an addition of spikes over the first level. Nevertheless, it is fundamentally the same with a slight increase in difficulty with a total of 24 available moves.

Figure 3.3. Third level (Source: Writer's archive)

The third level of "Helltaker" differs from the previous ones with the addition of the key mechanics. Players would have to retrieve the key first to remove the lock and reach the finish line, with a total of 32 available moves.

Figure 3.4. Fourth level (Source: Writer's archive)

The fourth level of "Helltaker" does not add new mechanics to the game. However, the position of stone blocks and some spikes add a level of complexity to the stage, hence increasing the difficulty from the third level. It has a total of 24 available moves.

B. Instruction Set

Players will concoct a sequence of instructions that will navigate their character to the finish line. However, the instructions must obey the constraints which is the limited number of moves available. A sequence is deemed valid when the player can reach the finish line following the given sequence without depleting the available moves midway through the sequence.

An example of an instruction sequence as follows:

[down, left, up, left, right, down, …., up]

The instructions will be executed sequentially from its left side all the way to its right side and will move the player accordingly.

This paper will focus on the generation of a valid sequence by implementing the Greedy Best First Search algorithm.

C. *Problem Modelling*

The algorithm will be using a node-based system. Each note will be represented as a state of the level. The state of the level contains the information below.

1. The layout of the level at that current moment, upon the movement of the player and the corresponding effects that might occur, such as the movement of a stone block, the destruction of a demon, etc.
2. The moves left counter
3. A flag for whether a key has been retrieved or not, only applies to levels that contain a key

Figure 3.5. State example (Source: Writer's archive)

For example, in figure 3.5, it resembles a state with 12 moves left, the key has been retrieved flag, and its layout.

Those three elements are the main discriminator between states.

When each node is processed, it will generate 4 other possible nodes. A node containing a state if the player moved Up, Down, Right, and Left. Each state will have its own heuristic value and will be sorted in a priority queue, the head of the queue has the highest priority to be processed.

The algorithm will stop if the queue is empty, or in other words, no path is available with the given layout and constraints.

IV. IMPLEMENTATION

The algorithm will be implemented in the Java language. A game model will be created to serve as an environment for the GBFS algorithm implementation.

A. *Game Modelling*

1. Game Layout

   The game will have its layout saved as an array of game objects. Its implementation is as follows.

Figure 4.1. Layout implementation (Source: Writer's archive)

2. Game objects

   The game objects will be represented in a new game object class as a symbol and name attribute.

Figure 4.2. Game objects implementation (Source: Writer's archive)

Each object will have a unique symbol and name representation. Here is the list of representations.

- The player game object will be denoted using the '$' symbol
- The demon game object will be denoted using the '*' symbol
- The stone block game object will be denoted using the '@' symbol
- The spike game object will be denoted using the '^' symbol
- The traversable floor will be denoted using the '.' Symbol
- The key game object will be denoted using the '?' symbol
- The lock game object will be denoted using the '+' symbol
- The finish point will be denoted using the '!' symbol
- The border of the game will be denoted using the '#' symbol
- The block that stacks to a spike will be denoted using the '&' symbol

Here is an example of the layout representation of the first level.

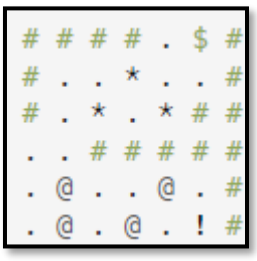


Figure 4.3. Sample layout implementation (Source: Writer's archive)

3. Player movement mechanics

The program could receive a set of strings that serves as an instruction for the player to modify its position. The set of strings contains “up”, “down”, “left”, and “right”. The program will move the player one tile according to the instructions given. Here is the detailed implementation. For the simplicity purposes, only a small portion of the code will be included in the paper, a link to the code repository will be provided.

Figure 4.4 Movement implementation (Source: Writer’s archive)

B. *State Implementation*

The state will save the layout as a string of symbols that symbolizes the layout of the level concatenated with the moves left counter for ease distinguishing. Here is the short implementation of the state. The state will also save the previous state or its parent node for backtracking purposes.

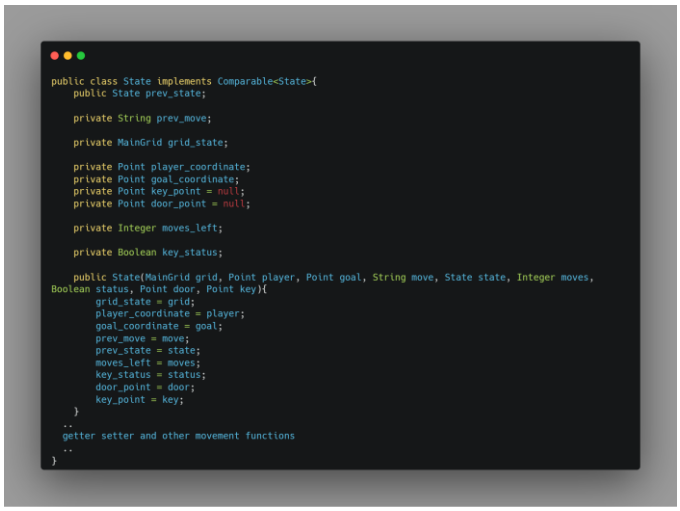


Figure 4.5. State implementation (Source: Writer’s archive)

The player’s movements will be carried out from the states generated by the algorithm node processing.

C. *Algorithm Implementation*

The algorithm will be implemented using a priority queue for node processing. Each node will be represented as a state. Here is the implementation of the algorithm.

The algorithm has a hash map to store the processed nodes to prevent the processed nodes from being reprocessed. It also has a sequence of strings which are later returned if an answer is found.

The state will be generated by applying four different movement instructions in each node. If a corresponding node generation is not possible, it will throw an exception and the node will not be generated.

Each generated node will be placed in the priority queue and be sorted based on its heuristic value. The calculation of the heuristic value could be inferred in the previous section.

Figure 4.6. Algorithm implementation (Source: Writer’s archive)

Figure 4.7. Node building implementation (Source: Writer’s archive)

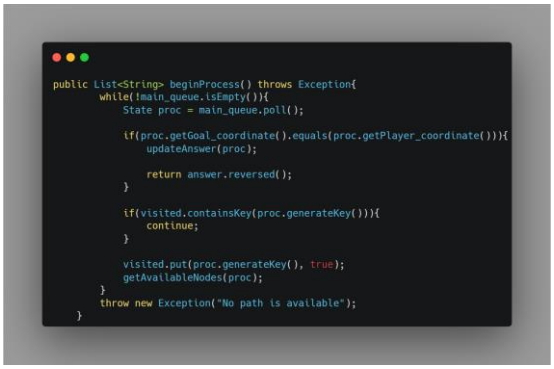
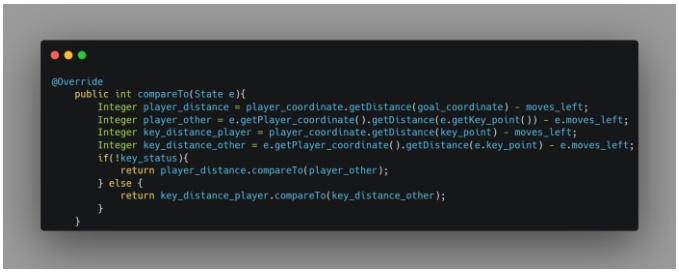


Figure 4.8. GBFS implementation (Source: Writer’s archive)

An intriguing aspect of the algorithm's development process involves the handling of visited states. Initially, states were saved as objects and stored in an array of visited states. However, this approach proved inefficient, prompting the author to convert each state into a string representation. By representing

states as strings, the need for storing state objects was eliminated, leading to a more efficient implementation of the algorithm.



```
@Override
    public int compareTo(State e){
        Integer player_distance = player_coordinate.getDistance(goal_coordinate) - moves_left;
        Integer player_other = e.getPlayer_coordinate().getDistance(e.getKey_point()) - e.moves_left;
        Integer key_distance_player = player_coordinate.getDistance(key_point) - moves_left;
        Integer key_distance_other = e.getPlayer_coordinate().getDistance(e.key_point) - e.moves_left;
        if(!key_status){
            return player_distance.compareTo(player_other);
        } else {
            return key_distance_player.compareTo(key_distance_other);
        }
    }
```

Figure 4.9. GBFS heuristic implementation (Source: Writer's archive)

V. TESTING

The testing will be carried out using a command line interface and level files inside the main program. Each level file contains the level size and moves given on its first line and its layout on the remaining lines. Here is the text file example, the 7 describe the level width, 6 is the level height, and 23 is the moves available for that level.

```
7 6 23
# # # # . $ #
# . . * . . #
# . * . * # #
. . # # # # #
. @ . . @ . #
. @ . @ . ! #
```

Figure 5.1. Level 1 txt file (Source: Writer's archive)

The expected output of the program is a sequence of instructions that can finish the level.

The main program will ask the user to prompt the level txt file and execute the algorithm, producing the appropriate result.

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Enter the level file:");
        Scanner scan = new Scanner(System.in);
        String file = scan.nextLine();

        Config con = new Config(file);
        con.getGrid().printContainer();
        System.out.println("Player : " + con.getPlayer());
        System.out.println("Goal : " + con.getGoal());

        State start = new State(con.getGrid(), con.getPlayer(), con.getGoal(), "start", null,
con.getMoves(), con.getKey_status(), con.getDoor_point(), con.getKey_point());

        try {

        } catch(Exception e) {
            System.out.println(e.getMessage());
        }

        if(con.getKey_status()){
            start.setDoor_point(con.getDoor_point());
            start.setKey_point(con.getKey_point());
            System.out.println("Key : " + con.getKey_point());
            System.out.println("Door : " + con.getDoor_point());

            start.getGrid_state().printContainer();
        }
        Algorithm gbfs = new Algorithm(start);
        try {
            List<String> res = gbfs.beginProcess();
            System.out.println(res);
            gbfs.printAnswer();
            System.out.println(res.size());
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }

        scan.close();
    }
}
```

Figure 5.2. Main program (Source: Writer's archive)

Level 1 Testing and result

The provided txt files:

Level_1.txt

```
7 6 23
# # # # . $ #
# . . * . . #
# . * . * # #
. . # # # # #
. @ . . @ . #
. @ . @ . ! #
```

Figure 5.3.a. Level 1 txt file (Source: Writer's archive)

The result:

```
Enter the level file:
level_1.txt
# # # # # # # # #
# # # # # . $ # #
# # . . * . . # #
# # . * . * # # #
# . . # # # # # #
# . @ . . @ . # #
# . @ . @ . ! # #
# # # # # # # # #
Player : (x=6,y=1)
Goal : (x=6,y=6)
[down, left, left, left, left, left, down, down, left, down, left, down, down, right, right, up
, up, right, right, right, right, down, right]
```

Figure 5.3.b. Level 1 result (Source: Writer's archive)

Sequence found:

Reached the finish line in **23** instructions

[down, left, left, left, left, left, down, down, left, down, left, down, down, right, right, up, up, right, right, right, right, down, right]

The sequence has been tested in the game and has proven to be valid.

Level 2 Testing and result

The provided txt files:

Level_2.txt

```
7 6 24
# . . . . # #
# * # ^ ^ . .
. ^ # # & & @
. . # # . ^ .
$ . # # ! * .
# # # # # . *
```

Figure 5.4.a. Level 2 txt file (Source: Writer's archive)

The result:

Reached the finish line in **20** instructions

Figure 5.4.b. Level 2 result (Source: Writer's archive)

Sequence found:

[up, up, right, up, up, up, up, right, right, right, down, right, right, down, down, down, down, left, left, down]

The sequence has been tested in the game and has proven to be valid

Level 3 Testing and result

The provided txt files:

Level_3.txt

Figure 5.5.a. Level 3 txt file (Source: Writer's archive)

The result:

Reached the finish line in **27** instructions

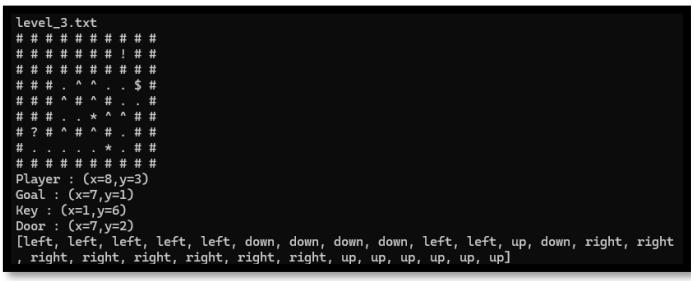


Figure 5.5.b. Level 3 result (Source: Writer's archive)

Sequence found:

[left, left, left, left, left, down, down, down, down, left, left, up, down, right, right, right, right, right, right, right, right, up, up, up, up, up, up]

The sequence has been tested in the game and has proven to be valid.

Level 4 Testing and result

The provided txt files:

Level_4.txt

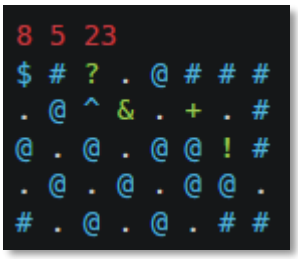


Figure 5.6.a. Level 4 txt file (Source: Writer's archive)

The result:

Reached the finish line in **23** instructions

Figure 5.6.b. Level 4 result (Source: Writer's archive)

Sequence found:

[down, down, down, right, down, down, right, up, up, right, down, down, right, up, up, right, right, down, down, right, right, up, up]

The sequence has been tested in the game and has proven to be valid.

Note: the algorithm has found a path that does not require the retrieval of the key, resulting in an achievement given from the game.

The sequence that requires the retrieval of the key would be:

[down, down, down, right, down, down, right, right, right, up, left, left, up, up, right, down, down, right, right, right, down]

With **20** instructions

VI. Conclusion

In this paper, we explored the application of the Greedy Best-First Search (GBFS) algorithm in solving the puzzle-based levels of the game "Helltaker." By focusing on the first four levels, we demonstrated how the GBFS algorithm, guided by a heuristic function, effectively navigates the tile-based game environment to find optimal paths.

The implementation process revealed several important considerations, such as the inefficiencies of storing state objects and the benefits of using string representations for visited states. These insights not only improved the algorithm's performance but also highlighted practical challenges and solutions in applying pathfinding algorithms to real-world scenarios.

In conclusion, this paper contributes to a deeper understanding of the GBFS algorithm and its practical applications, offering a concrete example of how theoretical concepts in computer science can be applied to enhance gameplay and solve puzzles in modern video games.

VIDEO LINK AT YOUTUBE

https://youtu.be/He5C3yrYVbA

REPOSITORY

https://github.com/ZakiYudhistira/Helltaker-Solver

ACKNOWLEDGMENT

The completion of this paper would not have been possible without the support of all IF2211 lecturers, especially Dr. Ir. Rinaldi Munir, M.T., who taught the K01 section of the algorithm strategy course. The author has gained a substantial amount of knowledge throughout the development of this paper. Special thanks are also extended for providing extensive learning resources for the students.

REFERENCES

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

[2] https://en.wikipedia.org/wiki/Helltaker

[3] Sierra, K., & Bates, B. (2003). Head First Java. O'Reilly.

STATEMENT

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 12 Juni 2024

Signed
Zaki Yudhistira Candra 13522031