CPT204 CW3 Report

Module:	CPT204-2324-S2-Advanced OO Programming
Group ID:	Group 37
Student ID:	2145618; 2142074
Student Name:	Mingyuan Li; Zhenyang Zhao
Submission time:	2 June 2024

1. Object-oriented Principles

Object-oriented programming (OOP) is a paradigm centered around the concept of "objects" that encapsulate both data and the methods that manipulate this data. It provides significant advantages in modularity, code reuse, and ease of maintenance [1]. In developing our coursework 3, we have adhered to the fundamental principles of OOP—encapsulation, inheritance, polymorphism, and abstraction—which has resulted in a robust and maintainable software structure.

This report is structured into six sections as follows: The **Object-oriented Principles** explains our OOP design and the architecture of the essential modules in the project, excluding the Monster and Rogue. The **Monster Algorithm** describes the design and implementation of our Monster algorithm. The **Rogue Algorithm** illustrates the design and implementation of our Rogue algorithm. The **Monster Algorithm Analysis** involves the evaluation of its integration with graph algorithms, the verification of its correctness and superiority through 2 dungeon examples, and the analysis of its time complexity. The **Rogue Algorithm Analysis** encompasses the evaluation of its integration with graph algorithms, the validation of its correctness and superiority using 2 dungeon examples, and the examination of its time complexity. The **My Java Code** lists all the source code in our project in textual form.

1.1. Overview of the Project

Our project employed Agile methodology to complete two iterations of the Monster algorithm and three iterations of the Rogue algorithm, demonstrating significant superiority. We considered both Breadth-First Search (BFS) and Depth-First Search (DFS) for graph algorithms and ultimately selected BFS. Consistent code conventions were maintained, with camelCase naming for method variables; comprehensive method comments, necessary inline comments; and uniform code indentation. Besides the algorithms, we developed an interactive module to validate algorithm superiority, a user-friendly graphical interface replacing console displays, and an ingenious adjacency list design with associated two-dimensional coordinate encoding and decoding logic. After designing and implementing the algorithms, we selected six non-trivial dungeons as examples to illustrate the algorithms' logic and optimal. The final version outcomes were well-defined, and the code design exemplified a comprehensive application of OOP principles.

1.2. Encapsulation

Encapsulation involves integrating attributes and functions that operate on data into a single unit while restricting access to the inner workings of that class. The primary benefit of encapsulation is that it protects an object's state from unauthorized access and modification, thereby enhancing the system's security [2]. Generally, encapsulation is implemented using access modifiers such as private and public. Besides, private modifiers are used along with getter and setter methods to provide controlled access to and modification of private fields.

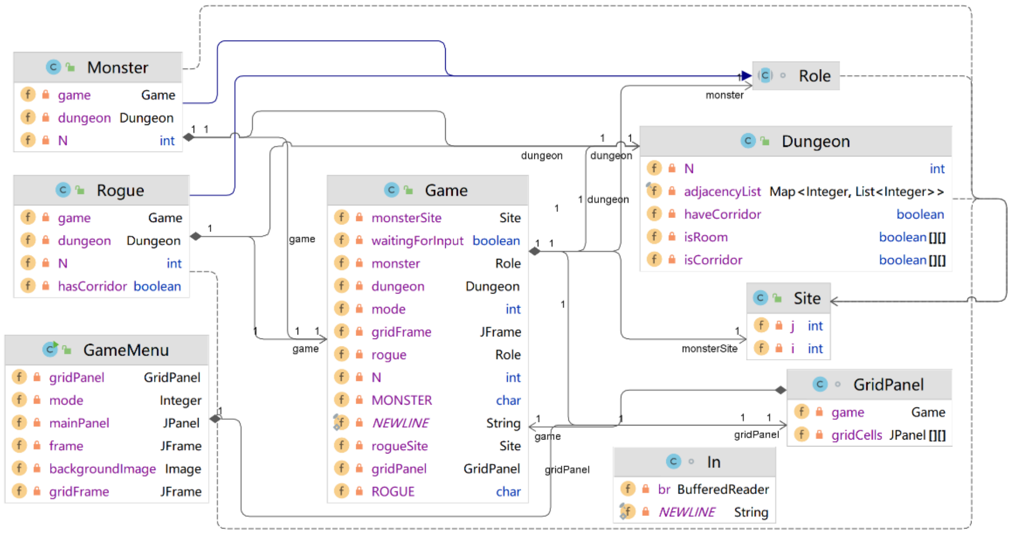
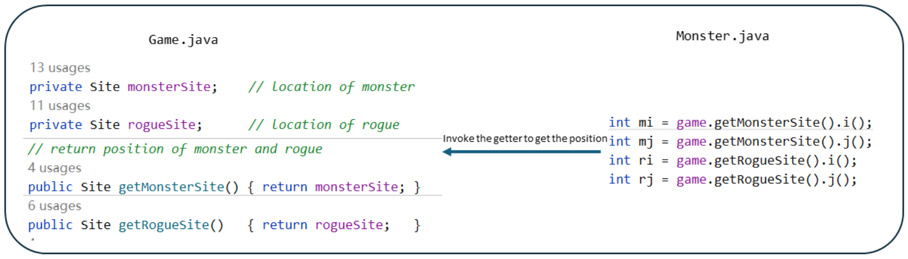


Figure 1: The Class Diagram of our Project

In our program, we ensured that all class fields were declared as private (Figure 1) and provided necessary getter and setter methods to access and update these fields. For example, consider the Game, Monster, and Rogue classes. Both monster and rogue have positions that are managed by the Game class. To ensure data integrity, the position fields are private, and appropriate getter methods are designed to allow `Monster` and `Rogue` to retrieve their positions and perform move operations. As shown in Figure 2, the Monster and `Rogue` classes access these positions via the `getMonsterSite` and `getRogueSite` methods, respectively. Additionally, we employed setter methods to update private variables, such as `setGridPanel`, which assigns a `GridPanel` object.



```
13 usages
private Site monsterSite;    // location of monster
11 usages
private Site rogueSite;      // location of rogue
// return position of monster and rogue
4 usages
public Site getMonsterSite() { return monsterSite; }
6 usages
public Site getRogueSite()   { return rogueSite;   }
```

```
int mi = game.getMonsterSite().i();
int mj = game.getMonsterSite().j();
int ri = game.getRogueSite().i();
int rj = game.getRogueSite().j();
```

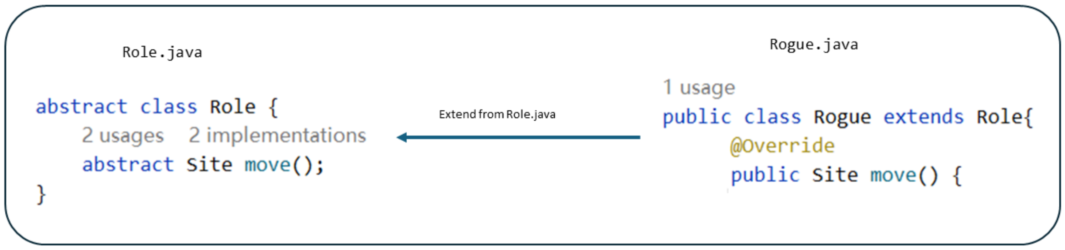
Figure 2: The Code Example of the Encapsulation

1.3. Inheritance

Inheritance allows a class, known as a subclass, to inherit fields and methods from another class, termed the superclass. This principle facilitates code reuse, enabling developers to create new classes based on existing ones, thereby promoting developing efficiency and reducing redundancy. Inheritance supports hierarchical class organization and the creation of more complex abstractions from simpler ones [3]. Typically, inheritance in Java is implemented using the `extends` keyword, which establishes a parent-child relationship between the subclass and the superclass. Moreover, subclasses use the `@Override` annotation to provide specific implementations of methods already defined in their superclasses.

In our structure design, we used inheritance to streamline our codebase and enhance reusability. We created a base class Role that encapsulates common methods shared by different game characters. Specifically, we included a move method in the `Role` class. Then, we derived specific classes like `Monster` and `Rogue` from this base class by using `extend`, inheriting its methods while overriding the move method to implement unique movement logic for each character. The implementation details are illustrated in Figure 3, and the inheritance relationships are depicted by the blue bold arrows pointing to the

Role class in Figure 1. In addition, to develop the graphical user interface (GUI), we created the GridPanel class that extends the JPanel class from the Swing library, thereby accelerating our development process.


Role.java

```
abstract class Role {
    2 usages   2 implementations
    abstract Site move();
}
```

Rogue.java

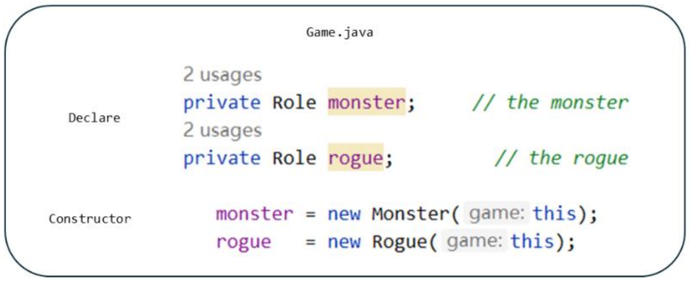
```
1 usage
public class Rogue extends Role{
    @Override
    public Site move() {
```

Figure 3: The Code Example of Inheritance

1.4. Polymorphism

Polymorphism enables objects to be treated as instances of their parent class rather than their actual class, allowing for dynamic method use. This capability enables programs to incorporate new functionality through subclass extensions without altering existing code [4]. The advantages of polymorphism include increased flexibility and the capacity for code generalization, enabling a single method to operate on objects of various types. Polymorphism typically achieves dynamic method invocation by referencing a subclass object through its parent class and invoking the subclass method with the @Override notation.

In our implementation of polymorphism, we utilized it to facilitate interactions between different types of game characters. We defined a Game class to manage the game, enabling interactions between instances of the Monster and Rogue classes. To achieve code reusability and extensibility, we declared the variables monster and rogue of type Role and initialized them in the Game constructor with the instance of the Monster class and Rogue class, respectively (Figure 4). This demonstrates polymorphism, as the monster and rogue variables can invoke methods defined in Role, with the appropriate overridden methods in subclasses being called at runtime.



Game.java

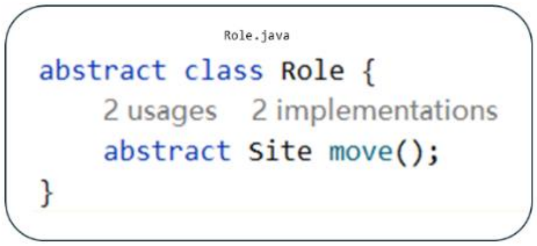
```
Declare        2 usages
               private Role monster;     // the monster
               2 usages
               private Role rogue;         // the rogue

Constructor        monster = new Monster( game: this);
                   rogue   = new Rogue( game: this);
```

Figure 4: The Code Example of Polymorphism

1.5. Abstraction

Abstraction involves reducing architecture complexity of system by focusing on the essential characteristics of an object while omitting non-essential details. Abstraction allows developers to manage larger systems by breaking them down into more manageable and understandable components, which promotes code clarity and reduces system complexity [5]. In Java, abstraction is achieved using abstract classes and interfaces. Abstract classes provide a partial implementation that other classes can extend, while interfaces define a contract that implementing classes must fulfill.



Role.java

```
abstract class Role {
    2 usages   2 implementations
    abstract Site move();
}
```

Figure 5: The Code Example of Abstraction

In our project, we employed abstraction by defining abstract classes and methods. Specifically, we created an abstract class, `Role`, which serves as a blueprint for `Monster` and `Rogue` (Figure 5). The Role class includes an abstract method `move()`, which must be implemented by any subclass, ensuring that all roles possess the capability to move. This approach not only promotes code reuse but also ensures a consistent class structure across different types of roles.

1.6. GUI

The four classes related to the GUI are `Game`, `GameMenu`, `GridPanel`, and `In`. Upon running the project, a `GameMenu` object is instantiated, initializing the basic GUI interface. A `JPanel` is then created within this interface to house the game. We designed four buttons here: three for different game modes and one for exit. If we start the game, the corresponding mode parameters are stored and create a `Game` object. Then, we designed a file list page to allow users to select a dungeon to play. Upon selecting a dungeon file, its file path is passed to the `In` class to read its contents, which are then combined with the mode and passed to the Game object. Subsequently, the map is drawn based on the dungeon file contents, and the play method is adjusted accordingly. When a victory condition is met, a popup page appears to end the game. Besides, we designed a user-friendly keyboard input method to control the character, utilizing the numeric keypad with 8 directional controls. For demonstration, the input interval for each control is set to about 1 second. Figure 6 provides further details on the GUI and color descriptions.

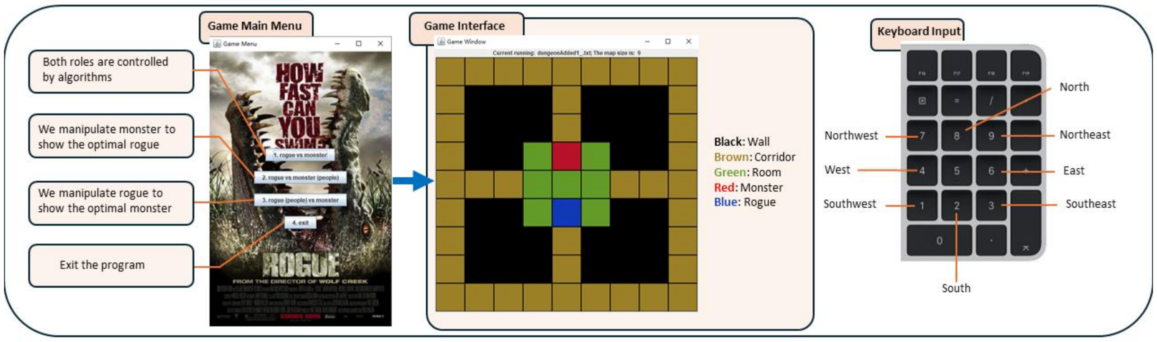
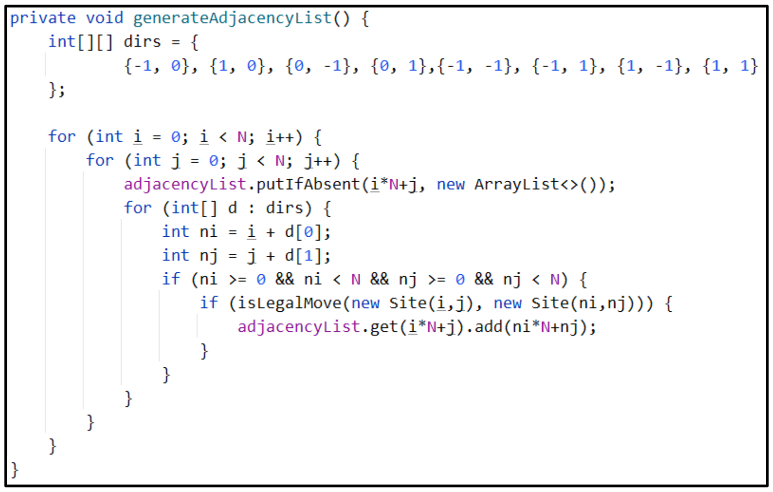


Figure 6: The GUI of Our Game and Keyboard Input Description

1.7. Adjacency List

The adjacency list plays a crucial role in our project, as it stores data for the undirected graph to be processed by the algorithm. In the adjacency list, each vertex represents a cell in the grid, and each edge denotes a legal move from one cell to an adjacent cell. We considered three possible types of cells: walls, corridors, and rooms, and developed methods to detect legitimate movements to establish the game's rules. Additionally, we designed an encoding logic to reduce the coordinate dimensions, thereby making the adjacency list usable.

The adjacency list and related judgment rules are contained within the Dungeon class, with the core implementation method being `generateAdjacencyList`. This method begins by defining eight possible movement directions (vertical, horizontal, and diagonal) using the `dirs` array. It then iterates over each cell in an N x N grid through nested loops. Next, the method evaluates neighboring cells in all eight directions, and if a neighboring cell lies within the grid boundaries and the movement to that cell is considered legal by the `isLegalMove` method, it is added to the current cell's adjacency list. The detailed method is shown in Figure 7.



```
private void generateAdjacencyList() {
    int[][] dirs = {
            {-1, 0}, {1, 0}, {0, -1}, {0, 1},{-1, -1}, {-1, 1}, {1, -1}, {1, 1}
    };

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            adjacencyList.putIfAbsent(i*N+j, new ArrayList<>());
            for (int[] d : dirs) {
                int ni = i + d[0];
                int nj = j + d[1];
                if (ni >= 0 && ni < N && nj >= 0 && nj < N) {
                    if (isLegalMove(new Site(i,j), new Site(ni,nj))) {
                        adjacencyList.get(i*N+j).add(ni*N+nj);
                    }
                }
            }
        }
    }
}
```

Figure 7： The Code of the generateAdjacencyList

Due to the two-dimensional nature of coordinates in the Site, we designed the encode logic to convert them into a one-dimensional integer for storage. Concurrently, we developed the decode logic to extract two-dimensional information from the adjacency list. The encode and decode processes can be expressed through the following mathematical formulas (1, 2, and 3):

$$index = i \times N + j \tag{1}$$

$$i = \frac{index}{N} \tag{2}$$

$$j = index \bmod N \tag{3}$$

where the index represents the data in the one-dimensional adjacencyList, i and j denote the row and column numbers respectively, and N is the size of the dungeon. Using the encode formula (1), if we have a 3x3 grid, the unique identifier for each node can be calculated as follows: the unique identifier for $(0,0)$ is $0 \times 3 + 0 = 0$; for $(0,1)$ it is $0 \times 3 + 1 = 1$; and so on, until $(2,2)$, where the unique identifier is $2 \times 3 + 2 = 8$. Therefore, based on the uniqueness of the index in the encoding process, the mathematical quantities obtained after decoding using Formulas (2) and (3) represent the i and j coordinates, respectively.

2. Monster Algorithm

The Monster algorithm embodies our fundamental pathfinding logic, focusing solely on the location of a single target, the Rogue. After two iterations, we finalized the current efficient Monster algorithm.

2.1. Design of Monster Algorithm

The basic logic of our Monster algorithm is to find the shortest path from rogue. We used BFS as pathfinding logic and recalculated paths based on new locations after each monster and rogue move.

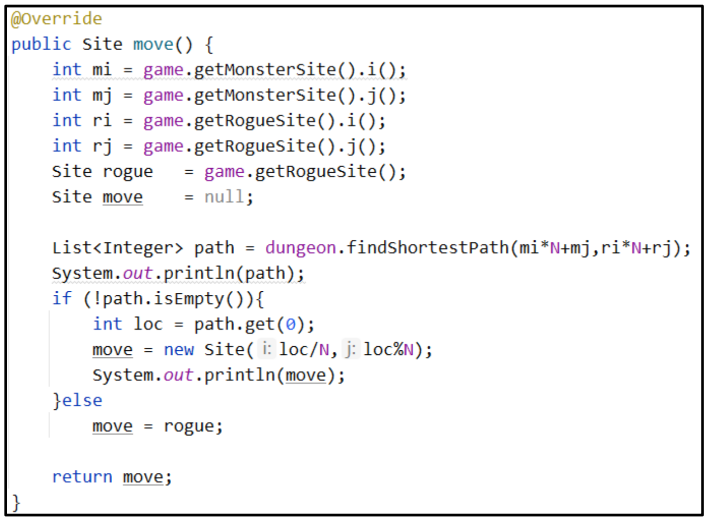
Iteration 1: Utilizing the basic logic, we designed a BFS-based pathfinding algorithm to determine the shortest path to the rogue. Specifically, BFS initiates exploring the whole graph from the location of the given monster and ends BFS when it reaches the rouge node. After the BFS completes, we trace back the parent nodes from the end node to construct the path and return it.

Iteration 2: During the testing of the initial version of the pathfinding algorithm, we observed that when paths included both straight-line and diagonal movements, the algorithm tended to prioritize straight-line movement first, followed by diagonal movement. This approach failed to yield the optimal path, which should prioritize diagonal movement. To address this issue, we optimized the diagonal routing logic for the monster. Specifically, we implemented a strategy where monster would move diagonally if they were not aligned in the same row or column and move straightly otherwise.

2.2. Implementation of Monster Algorithm

Building on our well-defined design, we implemented our ideas efficiently and iteratively. This section elaborates on the implementation of the final version. The core logic of our monster algorithm is encapsulated within the move method.

Initially, the coordinates of both the rogue and the monster are obtained from the game object. We use `findShortestPath` to determine the shortest path from the rogue. If a valid path is identified (the path list is not empty), the method extracts the first index's location from the path and converts it into a Site object using the coordinate decoding method. If no valid path is found, the monster uses a default location. The method is shown in Figure 8.



```java
@Override
public Site move() {
    int mi = game.getMonsterSite().i();
    int mj = game.getMonsterSite().j();
    int ri = game.getRogueSite().i();
    int rj = game.getRogueSite().j();
    Site rogue    = game.getRogueSite();
    Site move     = null;

    List<Integer> path = dungeon.findShortestPath(mi*N+mj,ri*N+rj);
    System.out.println(path);
    if (!path.isEmpty()){
        int loc = path.get(0);
        move = new Site( i: loc/N, j: loc%N);
        System.out.println(move);
    }else
        move = rogue;

    return move;
}
```

Figure 8: The Code of the Monster Algorithm's Move

`findShortestPath`: This method is the primary approach to find the optimal shortest path between monster and rogue. We employed three data structures to store important data: a queue for BFS exploration, a HashMap to record parent nodes for path reconstruction and a set to keep track of visited nodes. This method begins by checking if the start and end nodes are the same, returning immediately if they are. Then, we used the same coordinate decoding logic to calculate the coordinates of the start and end nodes and set a flag, `prioritizeStraight`, to store the straight-line state if the monster and rogue are aligned. We use the `while` loop and the queue to run the BFS, it starts from the given monster's position. During BFS, we used a `for` loop to categorize neighbors into straight-line and diagonal groups, processing based on priority to select the optimal path. Next, we used another `for` loop to mark unvisited neighbors, add them to the queue, and record the parent-children relationship map. Lastly, If the end node is reached, the path is reconstructed using the parent map; otherwise, an empty list is returned if no path is found.

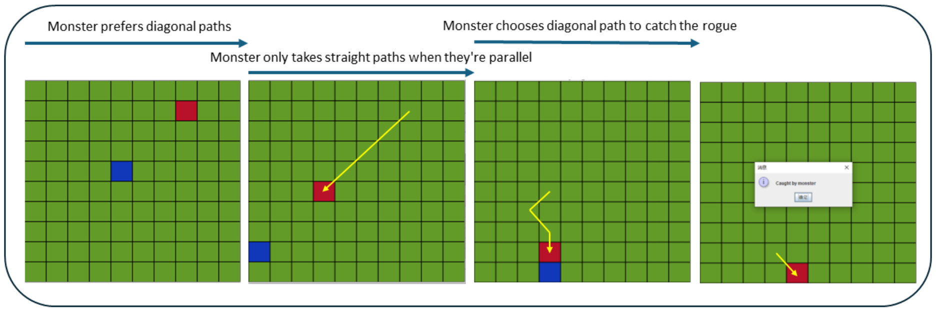

Monster prefers diagonal paths
Monster chooses diagonal path to catch the rogue
Monster only takes straight paths when they're parallel

Figure 9: The Demo of Rogue Algorithm Working Situation in dungeonA

To provide a more detailed description of our monster algorithm implementation, it is suitable to present a practical example (dungeonA), as shown in Figure 9. Initially, the rogue consistently chooses the southwest direction, indicating that monsters prefer diagonal paths. Then, the monster only follows straight paths when they are parallel. Finally, the rogue finds no escape as the monster prefers diagonal movement.

3. Rogue Algorithm

The Rogue algorithm presents a greater challenge compared to the Monster algorithm due to its requirement to account for wall and corridor situations, rather than merely focusing on single-target pathfinding. Consequently, we undertook three iterations to develop the current, highly effective Rogue algorithm.

3.1. Design of Rogue Algorithm

The basic logic of our algorithm is to find the most effective position (EP) from the monster that has the highest probability of escape. Then for EP, find the rogue's shortest path to it. We considered BFS and DFS as pathfinding logic, but due to the low efficiency of DFS time complexity, we finally adopted BFS as the traversal method.

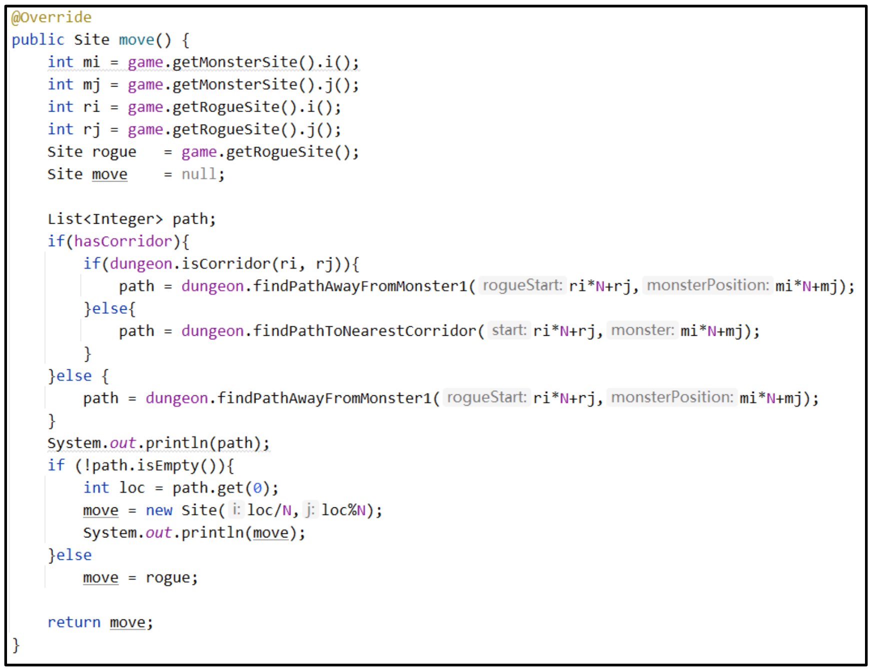
Iteration 1: Based on the basic logic, we designed a BFS-based pathfinding algorithm to maximize the distance from the monster. For the monster position, we found that the potential path might intersect or pass near the monster's position, we designated the 8 surrounding units as avoidance zones. Specifically, the BFS starts from the given rogue's position to explore the graph, skipping already visited nodes, the monster's position, and nodes adjacent to the monster. After BFS completes, the algorithm should calculate the farthest node from the monster based on the Manhattan distance. Finally, the path from the starting node to the farthest node is constructed and returned.

Iteration 2: During development, we discovered that corridors were important for rogues to escape because their unique directionality reduced the monster's movement space. However, in our initial design, the rogue did not prioritize accessing corridors. To address this, we optimized the rogue's behavior to prioritize corridor access. For maps with corridors, the rogue should assess whether its current location necessitates moving toward a corridor. Specifically, following the previous shortest distance search logic, we developed a new algorithm that prioritizes finding the shortest path to the nearest corridor. After entering the corridor, the rogue prioritizes circular paths related to the corridors to maximize the escape rate from monsters. If no corridors are available, the basic pathfinding logic is applied.

Iteration 3: After testing various maps, we observed that when the monster is near the closest corridor to the rogue, the rogue still heads to the corridor and gets caught. Consequently, we iterated the corridor pathfinding algorithm. If the monster is closer to the corridor than the rogue, the rogue selects the second nearest corridor. If the second corridor is still nearer to the monster, the algorithm will consider other corridors sequentially. If all corridors are closer to the monster, the rogue will disregard the corridors and instead choose the normal path that maximizes the distance from the monster.

3.2. Implementation of Rogue Algorithm

Based on our well-defined design, we implemented our ideas iteratively and efficiently. This section details the implementation of the final version. The top-level logic implementation of our rogue algorithm is encapsulated in the move method.



```java
@Override
public Site move() {
    int mi = game.getMonsterSite().i();
    int mj = game.getMonsterSite().j();
    int ri = game.getRogueSite().i();
    int rj = game.getRogueSite().j();
    Site rogue   = game.getRogueSite();
    Site move    = null;

    List<Integer> path;
    if(hasCorridor){
        if(dungeon.isCorridor(ri, rj)){
            path = dungeon.findPathAwayFromMonster1( rogueStart: ri*N+rj, monsterPosition: mi*N+mj);
        }else{
            path = dungeon.findPathToNearestCorridor( start: ri*N+rj, monster: mi*N+mj);
        }
    }else {
        path = dungeon.findPathAwayFromMonster1( rogueStart: ri*N+rj, monsterPosition: mi*N+mj);
    }
    System.out.println(path);
    if (!path.isEmpty()){
        int loc = path.get(0);
        move = new Site( i: loc/N, j: loc%N);
        System.out.println(move);
    }else
        move = rogue;

    return move;
}
```

Figure 10: The Code of the Rogue Algorithm's Move

Initially, it retrieves the coordinates of both the rogue and the monster from the game object. Depending on whether the dungeon has corridors (`hasCorridor`), the method determines the pathfinding method for the rogue. If the rogue is on a map that contains corridors and is currently located within one, `findPathAwayFromMonster1` is used to find a path with maximizes the distance from the monster; otherwise, we use `findPathToNearestCorridor` to find a path to the nearest optimal corridor. If there are no corridors, `findPathAwayFromMonster1` is used by default. If a valid path is found (the path list is not empty), the method extracts the first location from the path and converts it into a `Site` object using the coordinate decoding method. If no valid path is found, the rogue uses a default location. The method is shown in Figure 10.

1. `findPathToNearestCorridor`: We used three data structures to store important data: a queue for node exploration, a HashMap to record parent nodes for path reconstruction, and a set to track visited nodes. This method begins by enqueuing the starting node and marking it as visited. We leverage BFS with a `while` loop that continues until the queue is empty. At each iteration, the current node is dequeued and its index is converted using the same coordinate decoding method. Then, if the current node is a corridor, indicating a suitable corridor, the path from the start node is reconstructed using the parent HashMap and returned. Next, we used a `for` loop to traverse the neighbors of the current node. For each neighbor, we use two helper methods to check its proximity to the monster and whether it is a corridor not occupied by the monster. Last, the corridor neighbor with the lower distance to the monster is marked as visited, recorded in the parent HashMap, and enqueued. If no corridor is found by the time the queue is exhausted, an empty list is returned, indicating the absence of a viable path.
2. `findPathAwayFromMonster1`: We employed four data structures to store crucial data: a queue for BFS traversal, a set to track visited nodes, a map for recording parent-child relationships, and another set to store the monster's adjacent position. To detect the nodes adjacent to the monster's position, we use a `for` loop to traverse 8 units around the monster and add them to the `monsterAdjacent` set. Then, we use the `while` loop and the queue to run the BFS, it starts from the given rogue's position. It explores the whole graph, skipping nodes that have already been visited, represent the monster's position, or are adjacent to the monster. After BFS completion, we use a `for` loop to iterate through all visited nodes to identify the node farthest from the monster's position using the Manhattan distance. Finally, the path from the rogue to the farthest node is constructed and returned.

To provide a more detailed description of our rogue algorithm implementation, it is helpful to present a practical example (dungeonF), as illustrated in Figure 11. Initially, the rogue in the lower left attempts to move towards the nearest corridors on the right. However, the monster obstructs the path upward, prompting the rogue to decide to proceed to the top corridor. Ultimately, the rogue enters the corridors and stays there.

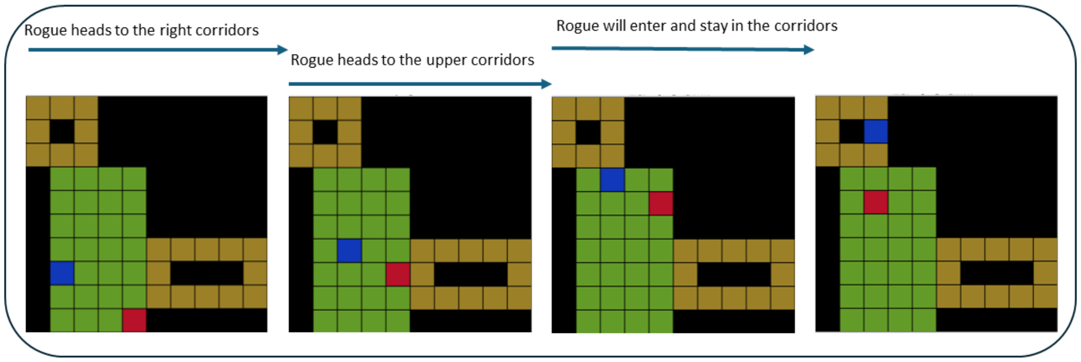

Rogue heads to the right corridors
Rogue heads to the upper corridors
Rogue will enter and stay in the corridors

Figure 11: The Demo of Rogue Algorithm Working Situation in dungeonF

4. Monster Algorithm Analysis

4.1. Integration with Graph Algorithms

In our rogue algorithm, we utilized BFS to find the shortest path in a graph. The BFS algorithm is well-suited for this task due to its inherent characteristics of exploring all nodes at the current depth level before moving on to nodes at the next depth level. This makes it efficient for identifying the shortest path between nodes, as it ensures that the first time a node is reached, it is reached via the shortest possible path. In the graph algorithm's data structure, the monster's BFS uses a queue for level-order traversal, a HashMap to track each node's parent for path reconstruction, and a set to record visited nodes to avoid redundant processing and optimize the search.

4.2. Verification of Correctness and Superiority

To verify the correctness and optimal performance of our Monster algorithm, we chose dungeonB and our custom dungeonAdded2 as test cases due to their abundance of rooms and corridors. Additionally, to improve interpretative efficiency, we employed a manual input module, allowing us to manipulate the confrontation rogue. This approach facilitated the simulation of the most challenging confrontation scenarios and enabled a detailed explanation of the specific steps.

Based on DungeonB's tests conducted in Figure 12, it is evident that the monster (yellow lines) caught rogue before it entered the corridor. Specifically, from #1 to #2, the rogue I controlled (pink lines) attempted to head towards the southeast corridor to get the escape loop. Subsequently, from #2 to #4, whenever the rogue was in a different row than the monster, the monster chose a diagonal path, ultimately catching the rogue at the node just before entering the corridor. If the monster had not chosen a diagonal path, the rogue would have been able to enter the corridor and escape.

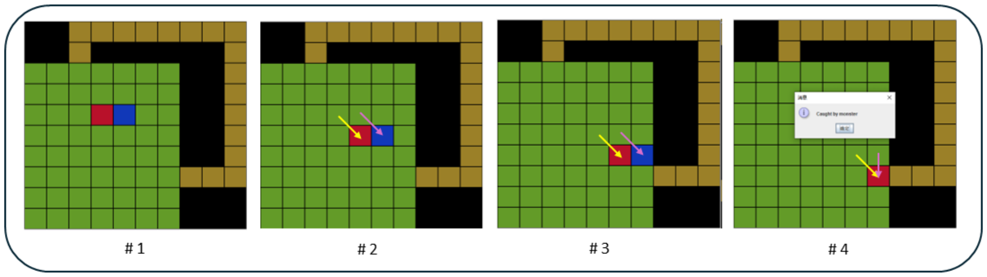


Figure 12: Monster Algorithm Test Case on dungeonB

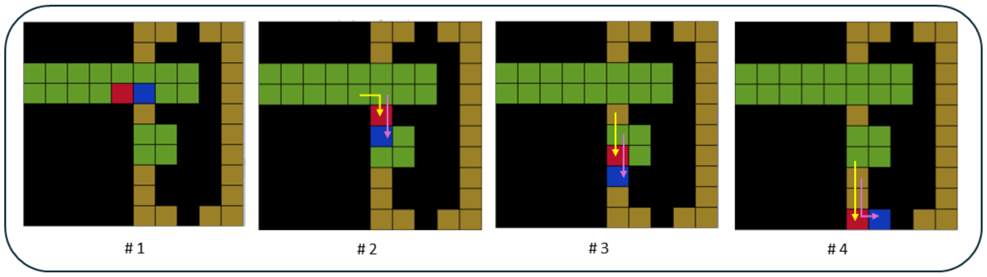


Figure 13: Monster Algorithm Test Case on dungeonAdded2

Likewise, Figure 13 illustrates the dungeonAdded2 test, showing how the monster corners the rogue. Specifically, from #1 to #2, the monster forces the rogue into the southern corridors. Then, from #2 to #4, the monster navigates through the rooms and corridors, finally trapping the rogue at a dead end.

4.3. Evaluation of Weaknesses

The primary weakness of the current monster algorithm is its inability to account for dead ends in maps with corridors. Specifically, the pathfinding logic only identifies the locally optimal solution and fails to dynamically analyze the entire map, including corridor dead ends. Consequently, the monster ignores potential paths that could trap the rogue at a dead end. In the future, we will try to integrate a global pathfinding strategy, such as A*. It can comprehensively evaluate the entire map and identify optimal paths that consider dead ends and potential traps for the rogue.

4.4. Time and Space Complexity Analysis

In the monster algorithm, the primary factors affecting time and space complexity are attributed to the findShortestPath function. For complexity analysis, let V represent the number of nodes in the graph and E denote the number of edges between these nodes.

In the findShortestPath function, the initialization of the data structures (queue, `parent`, and `visited`) occurs in constant time, O(1). For the outer `while` loop, in the worst-case scenario, all nodes are enqueued and dequeued once, resulting in a time complexity of O(V). The two inners `for` loops process each node by taking it from the queue and checking all its neighbors, that is effectively traversing all its edges. Consequently, the complexity of the inner loops is O(E). Therefore, the overall time complexity of the Monster is O(V + E).

The space complexity of the monster algorithm is determined by the storage demands of the employed data structures. The queue list, `parent` map, and `visited` set can each grow to hold up most V elements. In complexity analysis, we consider the highest-order term that dominates the growth rate. Thus, the total space complexity is O(V).

5. Rogue Algorithm Analysis

5.1. Integration with Graph Algorithms

In our rogue algorithm, we utilized BFS twice to support graph traversal. The BFS algorithm is well-suited for our rogue pathfinding due to its inherent characteristics of exploring all nodes at the present depth level before moving on to nodes at the next depth level. This makes it efficient for identifying the node farthest from the monster, as it thoroughly traverses all nodes in the graph. Additionally, when determining the most suitable corridor, BFS ensures that the first encountered path to a corridor is the shortest, thereby meeting the requirement for an efficient solution.

In the graph algorithm's data structure, the rogue's BFS utilizes a queue to manage nodes to be explored, ensuring a level-order traversal of the graph. Additionally, a HashMap is used to keep track of each node's parent, facilitating the reconstruction of the path once the target corridor is found. Furthermore, a set is employed to record visited nodes, preventing redundant processing, and thereby optimizing the search process.

5.2. Verification of Correctness and Superiority

To verify the correctness and superiority of our algorithm, we chose `dungeonI` and our custom dungeonAdded1 as test cases. These were chosen due to their inclusion of loops and the proximity of the monster to the rogue. Besides, to enhance the efficiency of interpretation, we use the manual input module which allows us to manipulate the confrontation monster, simulating the most challenging confrontation scenarios and enabling a detailed explanation of the specific steps.

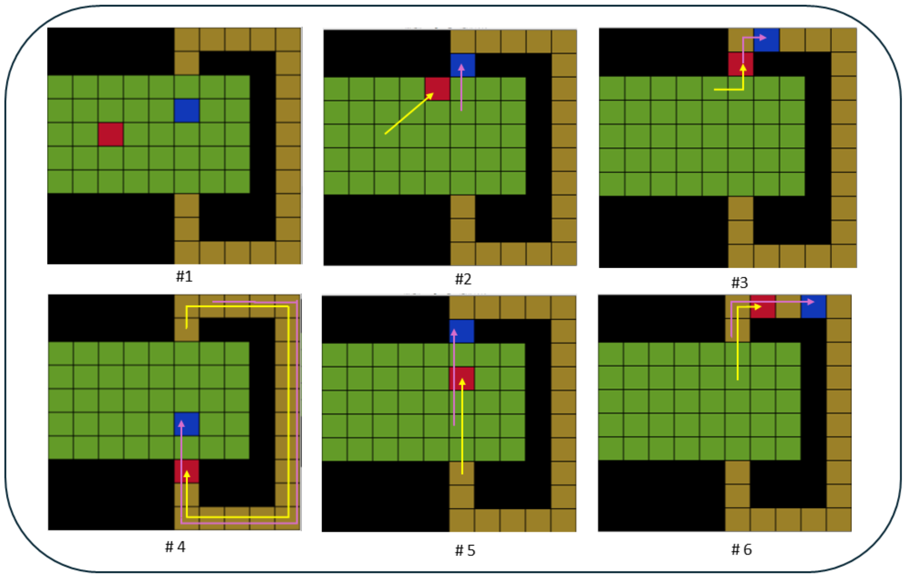


Figure 14: Rogue Algorithm Test Case on dungeonI

Based on `dungeonI`'s tests depicted in Figure 14, it is evident that the rogue (pink lines) enters an escape loop with maximum escape probability. Specifically, from #1 to #2, the monster (yellow lines) follows a diagonal path, which is optimal, while the rogue goes to the nearest corridor. Subsequently, from #2 to #4, the rogue continues to escape down the corridors and exits. Finally, from #4 to #6, as the monster consistently takes the optimal path, the distance between the two characters remains one unit, causing the rogue to re-enter the corridor, thereby creating a permanent escape loop.

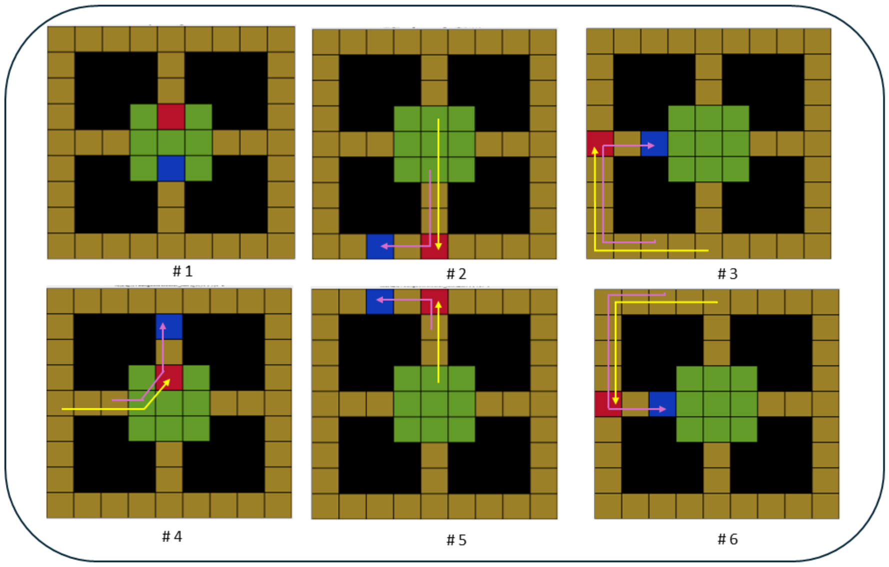


Figure 15: Rogue Algorithm Test Case on dungeonAdded1

Similarly, Figure 15 illustrates the dungeonAdded1 test, demonstrating that the rogue also enters an escape loop. Specifically, from #1 to #3, the monster forces the rogue into the southwest corridors until the rogue is about to return to the central rooms. Subsequently, from #3 to #5, the rogue detects the monster in the left corridor and opts to enter the north corridor entrance. Finally, from #5 to #6, due to the monster always taking the optimal path, the distance between the two characters remains constant at 1. The rogue's position at #6 is the same as at #3, indicating a perpetual escape loop.

5.3. Evaluation of Weaknesses

The primary weakness of the current algorithm lies at the intersection of corridors and rooms, where the rogue often misjudges the next action, resulting in unnecessary lingering. Moreover, although our algorithm is workable in scenarios with corridors, the game's map does not always meet this condition. This limitation arises because our rogue algorithm only considers locations farthest from the monster and avoids those near it in a corridor-less map. Consequently, our rogue's BFS can only predict locally optimal solutions. Specifically, if the monster is centrally located in the path to the farthest place, the rogue will still take the risk to head for the optimal point. However, there may be longer detour paths that could enable the rogue to survive longer. In the future, we will explore the use of adversarial algorithms, such as Minimax, which simulate all possible opponent actions and select strategies that minimize potential path losses.

5.4. Time and Space Complexity Analysis

In the rogue algorithm, the main influence on time complexity and space complexity is `findPathAwayFromMonster1` and `findPathToNearestCorridor`. To facilitate complexity analysis, we define V as the number of nodes in the graph, E as the number of edges between nodes in the graph, and M as the number of represents the number of the monster's neighbors.

For `findPathAwayFromMonster1`, the initialization of the data structures (`queue`, `parent`, and `visited`) is constant time, O(1). Collecting the monster's adjacent positions requires iterating through its neighbors, resulting in constant time, O(1), because the monster's adjacent nodes are constant. The BFS traverses the entire graph, visiting each node and edge exactly once, leading to a time complexity of O(V + E). Finally, determining the farthest node involves iterating through the visited nodes (the number of them is V) and computing distances, contributing an O(V). Therefore, the overall time complexity of this method is O(V + E). Similarly, for `findPathToNearestCorridor`, the initialization of the data structures is also O(1). The primary `while` loop executes until the queue is empty, resulting in O(V). For each vertex, the inner `for` loop checks all its neighbors. In the worst case, each vertex could be connected to (V-1) other vertices, resulting in f O(E). Therefore, the time complexity of this method is also O(V + E). Finally, due to both of them being executed in parallel, the overall complexity remains O(V + E), as it is determined by the maximum of the two complexities

The space complexity of the two algorithms is determined by the storage requirements of the data structures used. The queue list, `parent` map, and `visited` set can each grow to contain at most V elements, resulting in a complexity of O(V). Additionally, the `monsterAdjacent` set stores up to M neighbors of the monster's position, resulting in O(M). For another algorithm, there is no `monsterAdjacent` set, so the space complexity is O(V). Consequently, the overall space complexity of the rogue algorithm is O(V + M).

References

[1] L. Zhang, "Study on comparison of AOP and OOP," in *2011 International Conference on Computer Science and Service System (CSSS 2011) - Proceedings*, 2011, pp. 3596-3599, doi: 10.1109/CSSS.2011.5974835.

[2] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," *SIGPLAN Not.*, vol. 21, no. 11, pp. 38–45, Jun. 1986, doi: 10.1145/960112.28702.

[3] D. Wu, L. Chen, Y. Zhou, and B. Xu, "A metrics-based comparative study on object-oriented programming languages," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Pittsburgh, PA, USA, 2015, vol. 2015-January, pp. 272-277, doi: 10.18293/SEKE2015-064.

[4] K. H. Wai, N. Funabiki, S. T. Aung, X. Lu, Y. Jing, H. H. S. Kyaw, and W.-C. Kao, "Code Writing Problems for Basic Object-Oriented Programming Study in Java Programming Learning Assistant System," in *GCCE 2023 - 2023 IEEE 12th Global Conference on Consumer Electronics*, Nara, Japan, 2023, pp. 5-6, doi: 10.1109/GCCE59613.2023.10315469.

[5] K. K. Zaw, N. Funabiki, E. E. Mon, and W.-C. Kao, "An Informative Test Code Approach for Studying Three Object-Oriented Programming Concepts by Code Writing Problem in Java Programming Learning Assistant System," in *2018 IEEE 7th Global Conference on Consumer Electronics (GCCE)*, Nara, Japan, 2018, pp. 629-633, doi: 10.1109/GCCE.2018.8574687.

6. My Java Code

`Dungeon.java`

```
import java.util.*;

public class Dungeon {
    private boolean[][] isRoom;         // is v-w a room site?
    private boolean[][] isCorridor;     // is v-w a corridor site?
    private int N;                          // dimension of dungeon
    private final Map<Integer, List<Integer>> adjacencyList = new
HashMap<Integer, List<Integer>>();
    private boolean haveCorridor = false;

    // initialize a new dungeon based on the given board
    public Dungeon(char[][] board) {
        N = board.length;
        isRoom     = new boolean[N][N];
        isCorridor = new boolean[N][N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if      (board[i][j] == '.') isRoom[i][j] = true;
                else if (board[i][j] == '+'){
                    haveCorridor = true;
                    isCorridor[i][j] = true;
                }
            }
        }
        generateAdjacencyList();
        System.out.println(adjacencyList);
        System.out.println(findShortestPath(0,99));
    }

    // return dimension of dungeon
    public int size() { return N; }

    // does v correspond to a corridor site?
    public boolean isCorridor(Site v) {
        int i = v.i();
        int j = v.j();
        if (i < 0 || j < 0 || i >= N || j >= N) return false;
        return isCorridor[i][j];
    }

    // does v correspond to a corridor site? (use int)
    public boolean isCorridor(int i, int j) {
        if (i < 0 || j < 0 || i >= N || j >= N) return false;
        return isCorridor[i][j];
    }
    // does v correspond to a room site?
    public boolean isRoom(Site v) {
        int i = v.i();
        int j = v.j();
        if (i < 0 || j < 0 || i >= N || j >= N) return false;
        return isRoom[i][j];
    }

    // does v correspond to a room site?(use int)
    public boolean isRoom(int i, int j) {
        if (i < 0 || j < 0 || i >= N || j >= N) return false;
        return isRoom[i][j];
    }

    // does v correspond to a wall site?
    public boolean isWall(Site v) {
        return (!isRoom(v) && !isCorridor(v));
    }

    //have Corridor
    public boolean haveCorridor() {
        return haveCorridor;
    }

    // does v-w correspond to a legal move?
    public boolean isLegalMove(Site v, Site w) {
        int i1 = v.i();
        int j1 = v.j();
        int i2 = w.i();
        int j2 = w.j();

        // Judgment does not go beyond the boundary
        if (isNotValidCoordinate(i1, j1) || isNotValidCoordinate(i2,
j2)) {
            return false;
        }

        // Judge not to go out of the wal
        if (isWall(v) || isWall(w)) {
            return false;
        }

        // Calculating Manhattan distance
        int manhattanDistance = Math.abs(i1 - i2) + Math.abs(j1 - j2);

        if (manhattanDistance == 1) {
            // 水平或垂直移动
            return (isRoom(v) && isRoom(w)) ||
                    (isCorridor(v) && isRoom(w)) ||
                    (isCorridor(v) && isCorridor(w)) ||
                    (isRoom(v) && isCorridor(w));
        } else if (manhattanDistance == 2) {
            // Move diagonally
            return isRoom(v) && isRoom(w);
        }else return manhattanDistance == 0;
    }

    private boolean isNotValidCoordinate(int i, int j) {
        return i < 0 || j < 0 || i >= N || j >= N;
    }


    private void generateAdjacencyList() {
        int[][] dirs = {
                {-1, 0}, {1, 0}, {0, -1}, {0, 1},{-1, -1}, {-1, 1}, {1,
-1}, {1, 1}
        };

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                adjacencyList.putIfAbsent(i*N+j, new ArrayList<>());
                for (int[] d : dirs) {
                    int ni = i + d[0];
                    int nj = j + d[1];
                    if (ni >= 0 && ni < N && nj >= 0 && nj < N) {
                        if (isLegalMove(new Site(i,j), new Site(ni,nj)))
{
                            adjacencyList.get(i*N+j).add(ni*N+nj);
                        }
                        }
                    }
                }
            }
        }

    // Method to find the shortest path using BFS with priority on
movement direction
    public List<Integer> findShortestPath(int start, int end) {
        if (start == end) {
            return Collections.singletonList(start);
        }

        Queue<Integer> queue = new LinkedList<>();
        Map<Integer, Integer> parent = new HashMap<>();
        Set<Integer> visited = new HashSet<>();

        queue.add(start);
        visited.add(start);

        int startX = start / N;
        int startY = start % N;
        int endX = end / N;
        int endY = end % N;

        boolean prioritizeStraight = (startX == endX) || (startY ==
endY);

        while (!queue.isEmpty()) {
            int current = queue.poll();
            int currentX = current / N;
            int currentY = current % N;

            List<Integer> neighbors =
adjacencyList.getOrDefault(current, Collections.emptyList());

            // Separate straight and diagonal neighbors
            List<Integer> straightNeighbors = new ArrayList<>();
            List<Integer> diagonalNeighbors = new ArrayList<>();
            for (int neighbor : neighbors) {
                int neighborX = neighbor / N;
                int neighborY = neighbor % N;

                if (Math.abs(currentX - neighborX) + Math.abs(currentY -
neighborY) == 1) {
                    straightNeighbors.add(neighbor);
                } else {
                    diagonalNeighbors.add(neighbor);
                }
            }

            // Process neighbors based on priority
            if (prioritizeStraight) {
                neighbors = straightNeighbors;
                neighbors.addAll(diagonalNeighbors);
            } else {
                neighbors = diagonalNeighbors;
                neighbors.addAll(straightNeighbors);
            }

            for (int neighbor : neighbors) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    parent.put(neighbor, current);
                    queue.add(neighbor);
                    if (neighbor == end) {
                        return buildPath(parent, start, end);
                    }
                }
            }
        }
        return Collections.emptyList(); // Return empty list if no path
is found
    }

    // Method to find the path that maximizes the distance from the
monster when no corridor
    public List<Integer> findPathAwayFromMonster1(int rogueStart, int
monsterPosition) {
        Queue<Integer> queue = new LinkedList<>();
        Map<Integer, Integer> parent = new HashMap<>();
        Set<Integer> visited = new HashSet<>();

        queue.add(rogueStart);
        visited.add(rogueStart);

        // Collect monster adjacent positions
        Set<Integer> monsterAdjacent = new HashSet<>();
        for (int neighbor : adjacencyList.getOrDefault(monsterPosition,
Collections.emptyList())) {
            monsterAdjacent.add(neighbor);
        }

        while (!queue.isEmpty()) {
            int current = queue.poll();
            for (int neighbor : adjacencyList.getOrDefault(current,
Collections.emptyList())) {
                if (!visited.contains(neighbor) && neighbor !=
monsterPosition && !monsterAdjacent.contains(neighbor)) {
                    visited.add(neighbor);
                    parent.put(neighbor, current);
                    queue.add(neighbor);
                }
            }
        }

        // Find the farthest node from the monster
        int farthestNode = rogueStart;
        int maxDistance =
calculateManhattanDistance(indexToSite(rogueStart),
indexToSite(monsterPosition));
        for (int node : visited) {
            int distance = calculateManhattanDistance(indexToSite(node),
indexToSite(monsterPosition));
            if (distance > maxDistance) {
                maxDistance = distance;
                farthestNode = node;
            }
        }
        return buildPath(parent, rogueStart, farthestNode);
    }

    public List<Integer> findPathToNearestCorridor(int start, int
monster) {
        Queue<Integer> queue = new LinkedList<>();
        Map<Integer, Integer> parent = new HashMap<>();
        Set<Integer> visited = new HashSet<>();

        queue.add(start);
        visited.add(start);

        while (!queue.isEmpty()) {
            int current = queue.poll();
            int currentX = current / N;
            int currentY = current % N;

            // Check if the current position is a corridor
            if (isCorridor(new Site(currentX, currentY))) {
                return buildPath(parent, start, current);
            }

            for (int neighbor : adjacencyList.getOrDefault(current,
Collections.emptyList())) {
                if (!visited.contains(neighbor)) {
                    // Check if the neighbor is near the monster
                    boolean nearMonster = isNearMonster(neighbor,
monster);

                    // Check if the neighbor is a corridor and not the
monster's position
                    boolean isCorridorNeighbor = isCorridor(new
Site(neighbor / N, neighbor % N)) && neighbor != monster;

                    if (!nearMonster || isCorridorNeighbor) {
                        visited.add(neighbor);
                        parent.put(neighbor, current);
                        queue.add(neighbor);
                    }
                }
            }
        }
        return Collections.emptyList(); // Return empty list if no
corridor is found
    }

    // Helper method to check if a position is near the monster
    private boolean isNearMonster(int position, int monster) {
        int monsterX = monster / N;
        int monsterY = monster % N;
        int posX = position / N;
        int posY = position % N;

        // Check if the position is adjacent to the monster
        return Math.abs(monsterX - posX) <= 1 && Math.abs(monsterY -
posY) <= 1;
    }


    // Helper method to build the path from parent map
    private List<Integer> buildPath(Map<Integer, Integer> parent, int
start, int end) {
        List<Integer> path = new LinkedList<>();
        for (Integer at = end; at != null; at = parent.get(at)) {
            path.add(at);
        }
        Collections.reverse(path);
        if (!path.isEmpty() && path.get(0) == start) {
            path.remove(0); // Remove the start position from the path
        }
        return path;
    }

    private Site indexToSite(int index) {
        int i = index / N;
        int j = index % N;
        return new Site(i, j);
    }
    private int siteToIndex(Site site) {
        return site.i() * N + site.j();
    }

    private int calculateManhattanDistance(Site a, Site b) {
        return Math.abs(a.i() - b.i()) + Math.abs(a.j() - b.j());
    }

}
```

Game.java

```
import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;

public class Game {

    // portable newline
    private final static String NEWLINE =
System.getProperty("line.separator");

    private Dungeon dungeon;     // the dungeon
    private char MONSTER;        // name of the monster (A - Z)
    private char ROGUE = '@';    // name of the rogue
    private int N;               // board dimension
    private Site monsterSite;    // location of monster
    private Site rogueSite;      // location of rogue
    private Role monster;     // the monster
    private Role rogue;         // the rogue
    private int mode;            // game mode
    private GridPanel gridPanel;
    private JFrame gridFrame;
    private boolean waitingForInput;
    private ArrayList<Site> roguePath = new ArrayList<Site>();

    // initialize board from file
    public Game(In in) {
        // read in data
        N = Integer.parseInt(in.readLine());
        char[][] board = new char[N][N];
        for (int i = 0; i < N; i++) {
            String s = in.readLine();
            for (int j = 0; j < N; j++) {
                board[i][j] = s.charAt(2*j);

                // check for monster's location
                if (board[i][j] >= 'A' && board[i][j] <= 'Z') {
                    MONSTER = board[i][j];
                    board[i][j] = '.';
                    monsterSite = new Site(i, j);
                }

                // check for rogue's location
                if (board[i][j] == ROGUE) {
                    board[i][j] = '.';
                    rogueSite  = new Site(i, j);
                    roguePath.add(rogueSite);
                }
            }
        }
        dungeon = new Dungeon(board);
        monster = new Monster(this);
        rogue   = new Rogue(this);
    }

    // return position of monster and rogue
    public Site getMonsterSite() { return monsterSite; }
    public Site getRogueSite()   { return rogueSite;   }
    public Dungeon getDungeon()  { return dungeon;     }

    public void setGridPanel(GridPanel gridPanel) {
        this.gridPanel = gridPanel;
    }

    // play until monster catches the rogue
    public void play(GridPanel gridPanel, JFrame gridFrame, int mode) {
        this.gridPanel = gridPanel;
        this.gridFrame = gridFrame;
        this.mode = mode;
```

```
        // Use SwingWorker to update the grid panel in the background
        SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>() {
            @Override
            protected Void doInBackground() {
                for (int t = 1; true; t++) {
                    updateGridPanel();
                    try {
                        Thread.sleep(200);
                        // Check for end condition
                        if (monsterSite.equals(rogueSite)) break;
                        // Monster moves
                        if (mode == 1) {
                            waitingForInput = true;
                            while (waitingForInput) {
                                Thread.sleep(100); // Wait for user
input
                            }
                        } else {
                            Site next = monster.move();
                            if (dungeon.isLegalMove(monsterSite, next))
{
                                monsterSite = next;
                            } else {
                                throw new RuntimeException("Monster
can't catch rogue anyway");
                            }
                        }
                        updateGridPanel();
                        Thread.sleep(200);
                        // Rogue moves
                        if (monsterSite.equals(rogueSite)) break;

                        if (mode == 2) {
                            waitingForInput = true;
                            while (waitingForInput) {
                                Thread.sleep(100); // Wait for user
input
                            }
                        } else {
                            Site next = rogue.move();
                            if (dungeon.isLegalMove(rogueSite, next)) {
                                rogueSite = next;
                            } else {
                                throw new RuntimeException("Monster
can't catch rogue anyway");
                            }
                        }
                        updateGridPanel();
                        SwingUtilities.invokeLater(() ->
System.out.println(Game.this.toString()));
                        Thread.sleep(500);

                    } catch (InterruptedException e) {
                        System.out.println(e.getMessage());
                    } catch (RuntimeException e) {
                        JOptionPane.showMessageDialog(gridFrame,
e.getMessage());
                    }
                }
                JOptionPane.showMessageDialog(gridFrame, "Caught by
monster");
                return null;
            }
        };
        worker.execute();
    }

    public void processMove(int nextMove) {
        Site next = moveToSite(nextMove);
        if (mode==1&&dungeon.isLegalMove(monsterSite, next)) {
            monsterSite = next;
            waitingForInput = false;
        } else if (mode==2&&dungeon.isLegalMove(rogueSite, next)) {
            rogueSite = next;
            waitingForInput = false;
        } else {
            throw new RuntimeException("Illegal move");
        }
    }
    private void updateGridPanel() {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (dungeon.isCorridor(i, j)) {
                    gridPanel.setCellColor(i, j, new Color(155, 128,
40));
                } else if (dungeon.isRoom(i, j)) {
                    gridPanel.setCellColor(i, j, new Color(99, 155,
40));
                } else {
                    gridPanel.setCellColor(i, j, new Color(0, 0, 0));
                }
            }
        }
        gridPanel.setCellColor(rogueSite.i(), rogueSite.j(), new
Color(17, 56, 184));
        gridPanel.setCellColor(monsterSite.i(), monsterSite.j(), new
Color(184, 17, 42));
        gridPanel.repaint();
    }

    // move(intInput2Site)
    public Site moveToSite(int i) {
        int[][] dirs = {
                {1, -1},  // 1: Southwest
                {1, 0},   // 2: South
                {1, 1},   // 3: Southeast
                {0, -1},  // 4: West
                {0, 0},   // 5: Middle
                {0, 1},   // 6: East
                {-1, -1}, // 7: Northwest
                {-1, 0},  // 8: North
                {-1, 1}   // 9: Northeast
        };
        Site now;
        if(mode==1){
            now = monsterSite;
        }else {
            now = rogueSite;
        }
        int ni = now.i() + dirs[i - 1][0];
        int nj = now.j() + dirs[i - 1][1];
        return new Site(ni, nj);
    }

    // string representation of game state (inefficient because of Site
and string concat)
    public String toString() {
        String s = "";
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                Site site = new Site(i, j);
                if (rogueSite.equals(monsterSite) &&
(rogueSite.equals(site))) s += "* ";
                else if (rogueSite.equals(site)) s += ROGUE + " ";
                else if (monsterSite.equals(site)) s += MONSTER + " ";
                else if (dungeon.isRoom(site)) s += ". ";
                else if (dungeon.isCorridor(site)) s += "+ ";
                else if (dungeon.isRoom(site)) s += ". ";
                else if (dungeon.isWall(site)) s += "  ";
            }
            s += NEWLINE;
        }
        return s;
    }
}
```

GameMenu.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.FilenameFilter;

public class GameMenu {
    private JFrame frame;
    private Image backgroundImage;
    private JPanel mainPanel;
    private JFrame gridFrame;
    private GridPanel gridPanel;
    private Integer mode;

    public GameMenu() {
        // Load the background image
        backgroundImage = new ImageIcon("background.png").getImage();
        initializeUI();
    }

    /**
     * Initializes the user interface for the game menu.
     */
    private void initializeUI() {
        // Create and set up the main window
        frame = new JFrame("Game Menu");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 600);

        // Set the window's location (e.g., centered on the screen)
        Dimension screenSize =
Toolkit.getDefaultToolkit().getScreenSize();
        int x = (screenSize.width - frame.getWidth()) / 2;
        int y = (screenSize.height - frame.getHeight()) / 2;
        frame.setLocation(x, y);

        // Create a custom JPanel to draw the background image
        JPanel backgroundPanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                g.drawImage(backgroundImage, 0, 0, getWidth(),
getHeight(), this);
            }
        };
        backgroundPanel.setLayout(new GridBagLayout()); // Use
GridBagLayout for button layout
        frame.setContentPane(backgroundPanel);

        mainPanel = new JPanel();
```

```
mainPanel.setOpaque(false);
mainPanel.setLayout(new GridBagLayout());
backgroundPanel.add(mainPanel);

// Create buttons
JButton option1 = new JButton("1. rogue vs monster");
JButton option2 = new JButton("2. rogue vs monster (people)");
JButton option3 = new JButton("3. rogue (people) vs monster");
JButton option4 = new JButton("4. exit");

// Create GridBagConstraints to control button layout
GridBagConstraints gbc = new GridBagConstraints();
gbc.insets = new Insets(10, 10, 10, 10); // Set button margins
gbc.gridx = 0;
gbc.gridy = 0;
mainPanel.add(option1, gbc);

gbc.gridy = 1;
mainPanel.add(option2, gbc);

gbc.gridy = 2;
mainPanel.add(option3, gbc);

gbc.gridy = 3;
mainPanel.add(option4, gbc);

// Add button click event handlers
option1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        mode = 0;
        showFileList(mode);
    }
});

option2.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        mode = 1;
        showFileList(mode);
```

```
            }
        });

        option3.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                mode = 2;
                showFileList(mode);
            }
        });

        option4.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });

        // Display the window
        frame.setVisible(true);
    }

    /**
     * Displays a list of files in the "Dungeons" directory.
     *
     * @param mode The game mode selected by the user
     */
    private void showFileList(int mode) {
        File dir = new File("Dungeons");
        if (!dir.exists() || !dir.isDirectory()) {
            JOptionPane.showMessageDialog(frame, "Dungeons Folder does 
not exist");
            return;
        }

        File[] files = dir.listFiles(new FilenameFilter() {
            @Override
            public boolean accept(File dir, String name) {
                return name.toLowerCase().endsWith(".txt");
            }
        });

        if (files == null || files.length == 0) {
            JOptionPane.showMessageDialog(frame, "There are no files in
the Dungeons folder");
            return;
        }

        JList<File> fileList = new JList<>(files);
        fileList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        JScrollPane scrollPane = new JScrollPane(fileList);

        // Create a new window to display the file list
        JFrame listFrame = new JFrame("Dungeon list");
        listFrame.setSize(400, 300);
        listFrame.setLocationRelativeTo(frame);
        listFrame.add(scrollPane);
        listFrame.setVisible(true);

        fileList.addListSelectionListener(e -> {
            if (!e.getValueIsAdjusting()) {
                File selectedFile = fileList.getSelectedValue();
                if (selectedFile != null) {
                    // Get the name of the selected file
                    String fileName = selectedFile.getAbsolutePath();

                    In stdin = new In(fileName);
                    Game game = new Game(stdin);
                    // Display a new window
                    showGridWindow("Current running: " +
selectedFile.getName() + "; The map size is: " +
game.getDungeon().size(), game.getDungeon().size(), game);
                    System.out.println(game);
                    game.play(gridPanel, gridFrame, mode);
                }
            }
        });
    }

    /**
```

```
     * Displays a grid window with the game state.
     *
     * @param message  The message to display at the top of the window
     * @param gridSize The size of the grid
     * @param game     The game instance
     */
    private void showGridWindow(String message, int gridSize, Game game)
{
        gridFrame = new JFrame("Game Window");
        gridFrame.setSize(600, 600);
        gridFrame.setLayout(new BorderLayout());

        // Add a label with the message at the top
        JLabel label = new JLabel(message, SwingConstants.CENTER);
        gridFrame.add(label, BorderLayout.NORTH);

        // Create the grid panel
        gridPanel = new GridPanel(gridSize, game);
        game.setGridPanel(gridPanel);

        gridFrame.add(gridPanel, BorderLayout.CENTER);
        gridFrame.setLocationRelativeTo(null);
        gridFrame.setVisible(true);
    }

    public static void main(String[] args) {
        new GameMenu();
    }
}
```

GridPanel.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

/**
 * GridPanel class extends JPanel and implements KeyListener to handle
```

```
keyboard events.
 * This class creates a grid of cells to represent the game board and
allows user interaction through keyboard input.
 */
class GridPanel extends JPanel implements KeyListener {
    private JPanel[][] gridCells;  // 2D array of JPanel objects
representing the grid cells
    private Game game;  // Reference to the Game object

    /**
     * Constructor for the GridPanel class.
     * Initializes the grid with specified size and sets up the key
listener.
     *
     * @param gridSize The size of the grid (number of rows and columns)
     * @param game The Game object associated with this grid
     */
    public GridPanel(int gridSize, Game game) {
        this.game = game;
        setLayout(new GridLayout(gridSize, gridSize));
        gridCells = new JPanel[gridSize][gridSize];
        for (int i = 0; i < gridSize; i++) {
            for (int j = 0; j < gridSize; j++) {
                JPanel cell = new JPanel();

cell.setBorder(BorderFactory.createLineBorder(Color.BLACK));
                cell.setOpaque(true);
                gridCells[i][j] = cell;
                add(cell);
            }
        }
        // To ensure JPanel can receive keyboard events, it needs to be
focusable
        this.setFocusable(true);
        this.requestFocusInWindow();
        addKeyListener(this);
    }

    /**
     * Sets the color of a specific cell in the grid.
     *
     * @param row The row index of the cell
     * @param col The column index of the cell
     * @param color The color to set the cell background
     */
    public void setCellColor(int row, int col, Color color) {
        if (isValidCell(row, col)) {
            gridCells[row][col].setBackground(color);
        }
    }

    /**
     * Checks if the specified cell position is valid within the grid
bounds.
     *
     * @param row The row index of the cell
     * @param col The column index of the cell
     * @return True if the cell position is valid, otherwise false
     */
    private boolean isValidCell(int row, int col) {
        return row >= 0 && row < gridCells.length && col >= 0 && col <
gridCells[0].length;
    }

    @Override
    public void keyTyped(KeyEvent e) {
        // Do nothing
    }

    @Override
    public void keyPressed(KeyEvent e) {
        int keyCode = e.getKeyCode();
        if ((keyCode >= KeyEvent.VK_1 && keyCode <= KeyEvent.VK_9) ||
                (keyCode >= KeyEvent.VK_NUMPAD1 && keyCode <=
KeyEvent.VK_NUMPAD9)) {
            int nextMove;
            if (keyCode >= KeyEvent.VK_NUMPAD1 && keyCode <=
KeyEvent.VK_NUMPAD9) {
                nextMove = keyCode - KeyEvent.VK_NUMPAD0;
            } else {
                nextMove = keyCode - KeyEvent.VK_0;
            }
            game.processMove(nextMove);
        }
    }

    @Override
    public void keyReleased(KeyEvent e) {
        // Do nothing
    }
}
```

In.java

```
import java.io.*;

/**
 * In class provides functionality to read data from files.
 * It uses BufferedReader to read text from an input stream.
 */
class In {
    private BufferedReader br;  // BufferedReader object to read text
from an input stream

    /**
     * Constructor for the In class.
     * Initializes the BufferedReader to read from the specified file.
     *
     * @param fileName The name of the file to read from
     */
    public In(String fileName) {
        try {
            InputStream is = new FileInputStream(fileName);
            InputStreamReader isr = new InputStreamReader(is);
            br = new BufferedReader(isr);
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }

    /**
```

```
     * Reads a line of text from the input stream.
     * The line read does not include the newline character.
     *
     * @return The line read as a string, or null if an error occurs or
the end of the stream is reached
     */
    public String readLine() {
        String s = null;
        try {
            s = br.readLine();
        } catch(IOException ioe) {
            ioe.printStackTrace();
        }
        return s;
    }
}
```

Monster.java

```
import java.util.List;

public class Monster extends Role{
    private Game game;
    private Dungeon dungeon;
    private int N;

    public Monster(Game game) {
        this.game    = game;
        this.dungeon = game.getDungeon();
        this.N       = dungeon.size();
    }

    // move function for monster
    @Override
    public Site move() {
        int mi = game.getMonsterSite().i();
        int mj = game.getMonsterSite().j();
        int ri = game.getRogueSite().i();
        int rj = game.getRogueSite().j();
        Site rogue   = game.getRogueSite();
        Site move    = null;
        // The monster will move to the nearest shortest path to the
rogue
        List<Integer> path = dungeon.findShortestPath(mi*N+mj,ri*N+rj);
        System.out.println(path);
        if (!path.isEmpty()){
            int loc = path.get(0);
            move = new Site(loc/N,loc%N);
            System.out.println(move);
        }else
            move = rogue;

        return move;
    }

}
```

Rogue.java

```
import java.util.List;

public class Rogue extends Role{
    private Game game;
    private Dungeon dungeon;
    private int N;
    private boolean hasCorridor;

    public Rogue(Game game) {
        this.game    = game;
        this.dungeon = game.getDungeon();
        this.N       = dungeon.size();
        this.hasCorridor = dungeon.haveCorridor();
    }

    //    move function for rogue
    @Override
    public Site move() {
        int mi = game.getMonsterSite().i();
        int mj = game.getMonsterSite().j();
        int ri = game.getRogueSite().i();
        int rj = game.getRogueSite().j();
        Site rogue   = game.getRogueSite();
        Site move    = null;

        List<Integer> path;
        // The rogue will move to the nearest corridor if the dungeon
has a corridor
        if(hasCorridor){
            if(dungeon.isCorridor(ri, rj)){
                // The rogue will move to the nearest corridor if the
rogue is in a corridor
                path =
dungeon.findPathAwayFromMonster1(ri*N+rj,mi*N+mj);
            }else{
                // The rogue will move to the nearest corridor if the
rogue is not in a corridor
                path =
dungeon.findPathToNearestCorridor(ri*N+rj,mi*N+mj);
            }
        }else {
            // Default move
            path = dungeon.findPathAwayFromMonster1(ri*N+rj,mi*N+mj);
        }
        System.out.println(path);
        if (!path.isEmpty()){
            int loc = path.get(0);
            move = new Site(loc/N,loc%N);
            System.out.println(move);
        }else
            move = rogue;

        return move;
    }

}
```

Role.java

```
/**
 * Abstract class Role, representing a character in the game.
 * This class defines an abstract method move(), to be implemented by
specific character classes.
 */
abstract class Role {

    /**
     * Abstract method move, which defines the movement behavior of the
character.
     * Specific character classes must implement this method to provide
their unique movement logic.
     *
     * @return The new position of the character after moving, of type
Site.
     */
    abstract Site move();
}
```

Site.java

```
public class Site {
    private int i;
    private int j;

    /**
     * Initializes a Site with coordinates (i, j).
     *
     * @param i the row coordinate
     * @param j the column coordinate
     */
    public Site(int i, int j) {
        this.i = i;
        this.j = j;
    }

    /**
     * Returns the row coordinate of the site.
     *
     * @return the row coordinate
     */
    public int i() {
        return i;
    }

    /**
     * Returns the column coordinate of the site.
     *
     * @return the column coordinate
     */
    public int j() {
        return j;
    }

    /**
     * Returns a string representation of the site.
     *
     * @return a string in the format "Site{i=x, j=y}"
     */
    @Override
    public String toString() {
        return "Site{" +
                "i=" + i +
                ", j=" + j +
                '}';
    }

    /**
     * Calculates the Manhattan distance between the invoking Site and
the given Site w.
     * The Manhattan distance is the sum of the absolute differences of
their coordinates.
     *
     * @param w the site to which the distance is calculated
     * @return the Manhattan distance between this site and site w
     */
    public int manhattanTo(Site w) {
        Site v = this;
        int i1 = v.i();
        int j1 = v.j();
        int i2 = w.i();
        int j2 = w.j();
        return Math.abs(i1 - i2) + Math.abs(j1 - j2);
    }

    /**
     * Checks if the invoking site is equal to the given site w.
     * Two sites are considered equal if their Manhattan distance is
zero.
     *
     * @param w the site to compare with
     * @return true if the sites are equal, false otherwise
     */
    public boolean equals(Site w) {
        return (manhattanTo(w) == 0);
    }
}
```