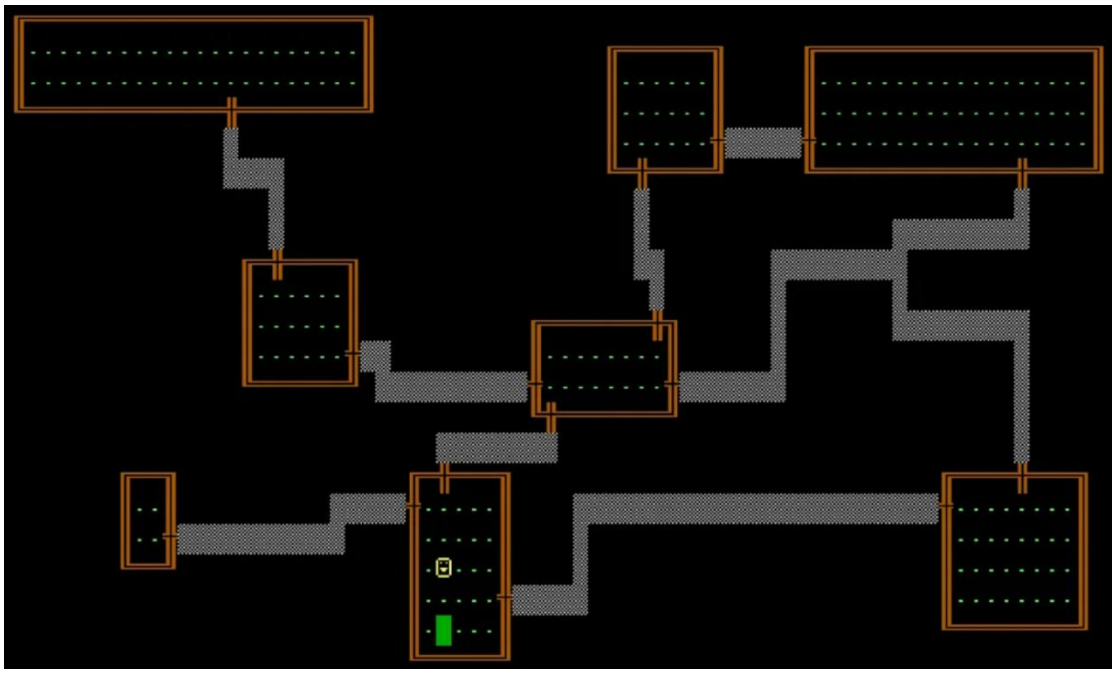
Xi'an Jiaotong-Liverpool University
西交利物浦大学

Report for OOP Project: Design of the Intelligent Rogue Game based on Graph Algorithms



CPT204: Advanced OO Programming

25th May 2024

Group 98

Zheyuan Cao 2141805

Yanqing Fei 2143430

Object-Oriented Programming (OOP) Principles

- **Factory Pattern**

Factory pattern, a creational design pattern that defines an interface for creating objects without the need to specify concrete classes, postponing the process of class instantiation to subclasses. The pattern decouples the object creation from its usage, making the code more flexible and convenient to maintain while enables new algorithm implementations to be added easily without modifying existing codes [1], which is in align with the Open Closed Principle (open for extension but closed for modifications) in the OOP. Specifically, in this project, the factory pattern showed in sketch below is implemented by *AlgorithmFactory* class which receives a game object and returns a concrete algorithm instance through the method *createAlgorithm()*. Such design pattern ensures that algorithms such as BFS, DFS and so on can be explored and extended while guarantees the freedom of type selection is left to players like monster and rogue, separating the algorithm construction from the instantiation.

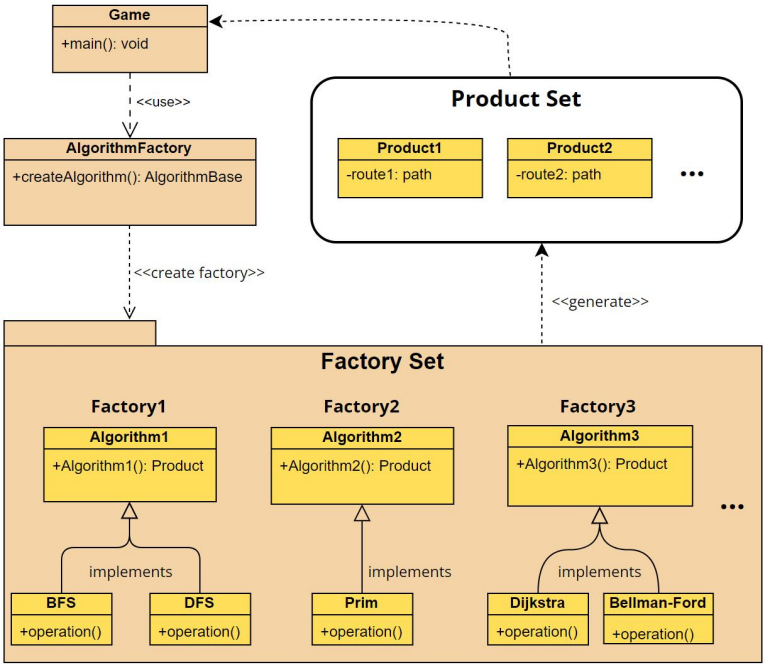


Figure 1: Factory Pattern Sketch

- **Encapsulation**

Encapsulation is defined as one of the fundamental principles of OOP, referring to the bonding of states (attributes) and behaviors (methods) of the object while

hiding the corresponding internal implementation details [2], which reduces the mutual effects of each component so that promotes the modularity of the program. External entities can only access and modify certain object properties through public access methods such as getters and setters, which restricts direct access to object internal components so that ensures the safety and prevent accidental modification. Specifically for this project, application of encapsulation for main classes is outlined as follows, along with the overall class diagram.

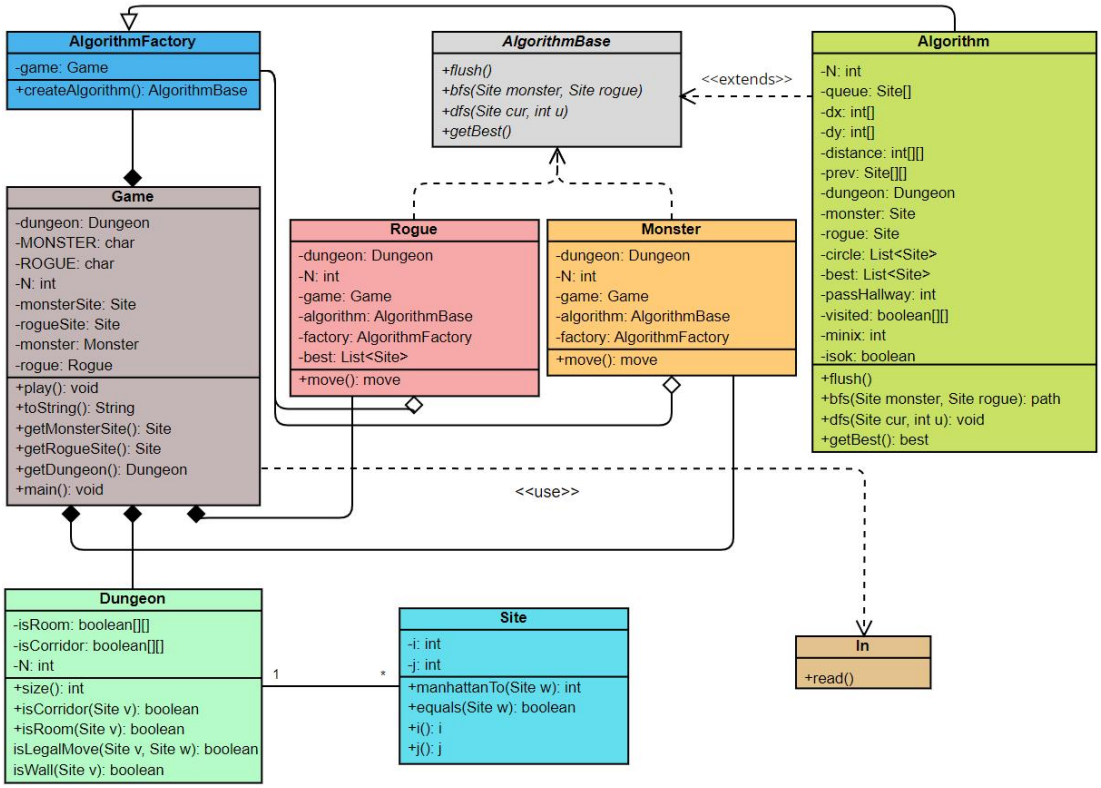


Figure 2: Entity Relation Diagram

1. *AlgorithmFactory* class encapsulates the construction logic of a specific algorithm, separating the algorithm instantiation from utilization. It has a private attribute *game* that cannot be accessed externally and can only be initialized through public constructors. Moreover, implementation details for algorithm instantiation are hidden from users, which simplifies the deployment process.
2. *AlgorithmBase* abstract class defines and encapsulates the basic operations and structure of an algorithm such as state refreshing *flush()*, algorithm logic like *bfs()* or *dfs()* and optimal paths generation *getBest()*.

3. *Algorithm* object encapsulates the concrete logic and implementation details for specific algorithms such as BFS and DFS via a series of private variables like *circle* and *dungeon*. Furthermore, the internal state *best* is the core field of this function to discover potential optimal routes, which cannot be modified arbitrarily and thus the only way to interact is via public get method interface *getBest()*.
4. *Monster* and *Rogue* are two characters that encapsulate the behavior and movement strategies during the game. Specifically, each class maintains a factory object to instantiate the BFS and DFS algorithms while the *move()* method defines the logic of escape or chase, which provides the interface for external entities to modify the player position.
5. *Site* class encapsulates the coordinates and relevant operations on such positions. For example, the get methods *i()* and *j()* are employed to return the private x and y positions and functions like *manhattanTo()* and *equals()* are offered to external objects to calculate and estimate the equality of distances.
6. *Dungeon* encapsulates the layout of the dungeon and query methods for specific properties such as the location of rooms and corridors. Specifically, it provides services through methods like *isCorridor(Site v)* and *isWall(Site v)* to return internal private states. For instance, the *isLegalMove(Site v, Site w)* facilitates the judgement on the correctness of movements and *size()* is applied to pass the structure of the dungeon to other objects.
7. *In* encapsulates the ways to read the input files, which provides public interfaces such as *readLine()* for external entities to invoke, facilitating the process of dungeons inputting and game structure foundation constructing.
8. *Game* is the main class which encapsulates the overall operational logic of the project. It maintains the states for objects in the game such as monster, rogue and dungeon while codes corresponding public methods to obtain such private data, guaranteeing the limited access and enhancing the security.

- **Inheritance**

  Inheritance is an OOP discipline that allows a class (subclass or derived class) to inherit and modify the fields and methods form another class (superclass or base class), enhancing the code reusability, promoting the program scalability and facilitating the maintenance process. For example, *AlgorithmBase* is a superclass that defines the basic structure and methods such as *bfs()* and *dfs()* for algorithms in this project. For corresponding subclass *Algorithm*, it not only inherits and implements those functions in the base class but also extends other methods like its own specific constructor *Algorithm()*.

- **Polymorphism**

  Polymorphism refers to the behavior variances on various object states [3], which means that the same interface can be extended to different implementations in programming. This concept possesses two primary mechanism for achievement, namely compile-time overloading and run-time overriding. For instance, in the *Monster* and *Rogue* classes, the *algorithm* instance is of type *AlgorithmBase*. However, it actually produces a reference to an *Algorithm* instance when utilizing the *bfs()* and *dfs()* methods which are overridden respectively in the *Algorithm* class and thus demonstrates the polymorphism. Such application improves the flexibility and substitutability through various subclass-specific implementation.

- **Abstraction**

  Abstraction is a characteristic that focuses on defining high-level and necessary class attributes and functionalities while ignoring underlying implementation details [4] to reduce programming complexity. This process simplifies the system design and maintenance, along with the enhancement of extensibility and flexibility. An implementation of abstraction in this project manifests in the construction of an abstract class. Specifically, the *AlgorithmBase* class is an abstract class that outlines a contract for its derived classes, albeit not offering tangible implementations for all of its abstract methods such as *flush()*, *bfs()*, *dfs()*, and *getBest()*. These methods are expected to be instantiated by the

subclasses they belong to. This abstraction operates as a blueprint for the class *Algorithm* that extends it, dictating a structure while allowing flexibility in the actual algorithm implementations.

Monster Algorithm

- **Strategy Brief Introduction**

  To optimize the Monster strategy in the game, the Breadth-First Search (BFS) algorithm is employed to determine the shortest path between two points on the map, ensuring that the Monster can efficiently close the distance to the Rogue. The BFS algorithm is particularly suitable for this task due to its ability to explore all possible routes level by level, guaranteeing the shortest path to be discovered in an unweighted graph.

- **Correctness**

  The correctness of the BFS algorithm lies in its systematic exploration of all possible paths in a breadth-first manner, which means that once the rogue site in the dungeon is reached, the path found is always the shortest. Furthermore, the two dimensional distance array also helps record and backtrack from the rogue position to the monster starting point. Such combination of level-based traversal and backtracking guarantees the correctness of the optimal shortest path reconstruction.

- **Optimality**

  BFS implements the optimality in unweighted grids, as it always chooses the next node to explore based on its distance from the start [5], ensuring the minimal number of moves to reach the rogue. This level-by-level search characteristic guarantees that the first time the position of the rogue is encountered, the path calculated is indeed the shortest.

- **Strengths**

  The BFS algorithm systematically explores the graph level by level, guaranteeing the discovery of minimum length if exists, which provides the simplicity, reliability and completeness of finding the shortest path in unweighted

environments. It is well-suited for real-time pathfinding in such grid-based games where the map topology is static while all movements have uniform cost.

- **Weaknesses**

  BFS is memory-sensitive, which leads to its high space complexity especially in large dungeons, as it needs to store all possible positions in the queue and the distance array and therefore constitutes the main weakness of the algorithm. Moreover, performance and efficiency of visiting all reachable nodes in complex map is another issue that impedes the BFS implementation.

- **Non-trivial Dungeons Proof**

  Based on the BFS, the following initial positions and dungeon terrain significantly impact the effectiveness of the monster algorithm in catching the rogue.

  1. **Central Monster, Edge or Corner Rogue – Dungeon P, R**

     According to the algorithm above, the monster will search radially in the graph to discover the potential shortest path to the rogue. Starting from a central position allows the monster to efficiently navigate towards any location in the dungeon. If the rogue begins near the edges or corners, the central initial location reduces the overall distance the monster needs to travel, making it easier to intercept the rogue before it finds an escape route. Below displays two examples of such situation where the dungeon P has a rogue initially sited near the edge while the rogue in R stays in the corner of the dungeon. It is noticed that in both terrains the monster exploited the correctness and optimality of the algorithm and managed to intercept the rogue within relative limited steps.

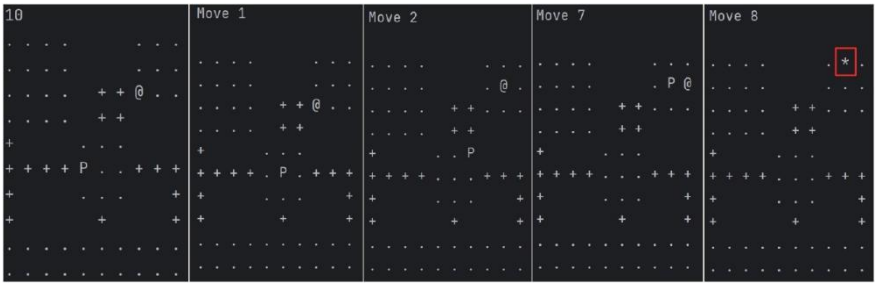


Figure 3: Dungeon P Results

  2. **Close Proximity in Open Space – Dungeon N**

Another potential capture can occur in a situation where the monster and rogue are in open space with a close proximity. Specifically, when the monster starts very close to the rogue in an open terrain, the monster can directly move towards the rogue without any obstacles blocking its path. The minimal initial distance reduces the time needed for the monster to reach the rogue, increasing the chances of a quick capture. The dungeon N is an input where the distance of characters is only two units while the layout is open and simple. The algorithm imagined the ease of movement of the monster in such circumstance, which was indicated by the following rapid chasing process, complying with the optimization assumption based on the algorithm analysis

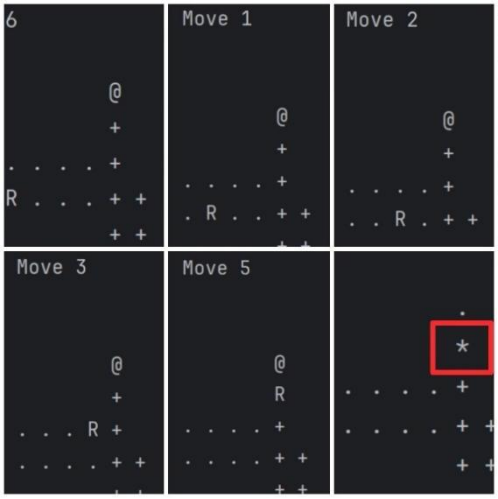


Figure 4: Dungeon R Results

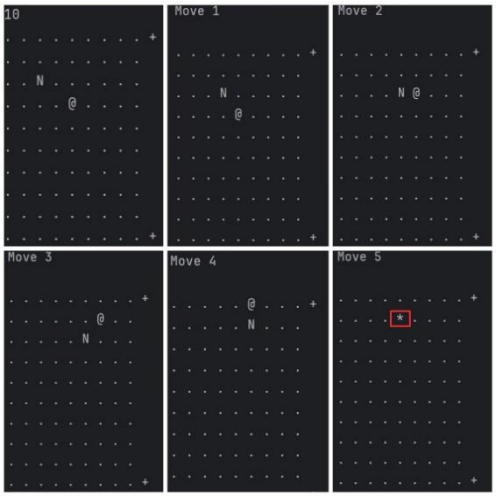


Figure 5: Dungeon N Results

3. Narrow Corridor with Limited Exits – Dungeon D

The layout of the dungeon also has impacts on the success of rogue capture. For instance, narrow corridors with few exits limit the rogue movement options, funneling it into predictable paths. Sine the rogue has fewer directions to escape, such circumstance therefore facilitates the process of cornering and catching the rogue for monster. The BFS-based algorithm can help the monster quickly navigate these confined spaces, leveraging the restricted movement to its advantage. For example, the dungeon D has a spiral corridor with two exits in the room and center respectively. The rogue is initially located in the room site connected to the corridor. However, it was

subsequently forced to step into the narrow corridor to escape from the monster and eventually caught in the central exit, which proved the correctness and optimality of the Monster algorithm.

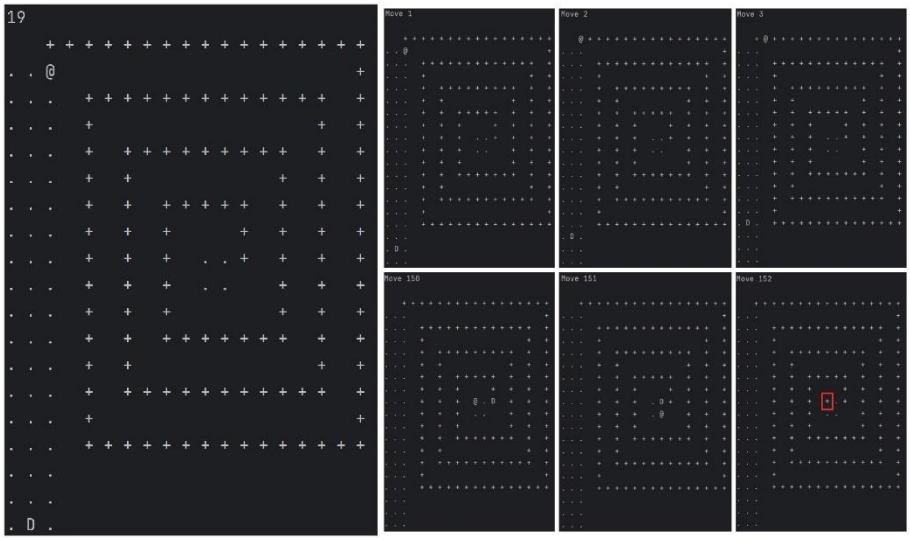


Figure 6: Dungeon D Results

4. Separate Space – Dungeon M

An inevitable failure of chase occurs in the terrain where the dungeon consists of two or more isolated rooms. The BFS algorithm is designed to find the shortest path in an unweighted grid by exploring all possible moves. If no connecting paths exist, the algorithm will exhaust its search space and determine that the rogue is unreachable, leading to the monster remaining stationary. This behavior highlights the limitation of the BFS algorithm in environments where parts of the dungeon are completely independent from each other. The example dungeon M below shows that from the initial state to the 19810th movements, the position of the monster never changed, which demonstrates the assumption above.

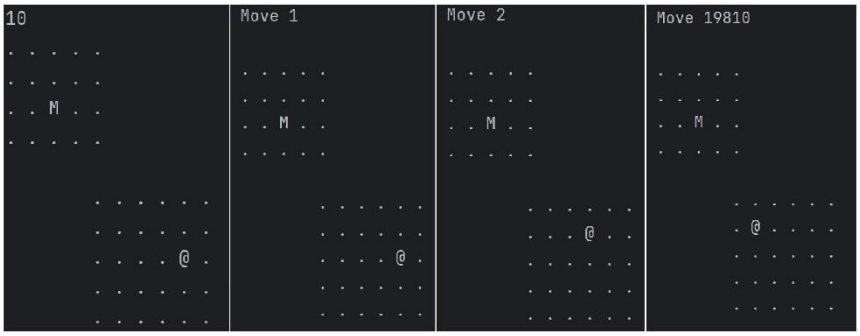


Figure 7: Dungeon M Results

Rogue Algorithm

- **Strategy Brief Introduction**

  The Rogue algorithm employs a dual-strategy approach using Breadth-First Search (BFS) and Depth-First Search (DFS) to navigate and evade threats within dungeon environments. Specifically, BFS is utilized to find the shortest paths, ensuring efficient and optimal navigation through the dungeon while DFS is applied to detect cycles and alternative routes, providing the rogue with additional evasion strategies. This combination allows the rogue to adapt dynamically to different dungeon layouts, maintaining a strategic advantage over monster chasing.

- **Correctness**

  According to the analysis on BFS above, this graph algorithm allows the rogue to identify shortest path to certain sites such as the corridor entrance or exit and the monster position, guaranteeing the efficiency and correctness to enter an infinite loop or evade from the monster. Moreover, for the application of DFS, it provides the rogue with the accurate cycle detection, validation and correct backtracking. Specifically, the DFS explores the dungeon by depth to discover potential cycles before backtracking and actual move. Furthermore, the algorithm utilizes variables like *cur* and *rogue* to compare the position of the starting point with the terminus, ensuring that cycles are valid loop. In addition, methods such as *isCorridor()* are also invoked to guarantee that circle to be passed contains at least one corridor, which is essential for rogue strategic use in evasion. Upon detecting one potential path, recursive approach enables all records and states to be updated and refreshed, providing correct backtracking.

- **Optimality**

  Combination of the BFS with the DFS in the Rogue algorithm achieves the optimization for adaptive strategy, dynamic decision-making and pruning efficiency. For instance, the BFS always maintains the optimal route with a minimum distance from the rogue to the monster or corridor exits while the DFS

identifies cycles that the rogue can exploit for infinite evasion, adding another layer of optimality by providing multiple strategic options. Apart from that, the adoption of pruning mechanism makes it possible for the algorithm to realize the optimization for efficiency. Specifically, the variable *u* not only limits the maximum feasible recursion depth but also provides timely comparison with the current minimum length to avoid excessive computation for non-promising paths, thus achieving the optimization pruning. Additionally, examination on state of the route discovery instance *isOk*, along with the pruning for detecting potential movement process which involves four directions (N, S, E, W) identification rather than eight, eliminating the equivalent redundancy effectively. The pruning therefore reduces the search space while enhances the performance by focusing on promising paths so that helps the algorithm reaches the optimization.

- **Strengths**

  The application of BFS benefits in determining the shortest path while maintaining a safe distance from the rogue to monster to facilitate the rogue survival. For DFS, the algorithm employs it to manage to explore possible routes thoroughly and discover corridor-connected cycles for infinite rogue escape. During the graph algorithm implementation, pruning non-promising paths and cycles reduces the computational load, making the algorithm faster and more efficient. The techniques utilized above allow the rogue to adopt flexible strategies, which means that it can dynamically switch between the straightforward pathfinding and exploiting cycles based on the real-time situation of the dungeon, demonstrating the strengths of the algorithm.

- **Weaknesses**

  Resource intensity is considered as one of the most significant factors that impede the performance of the algorithm. Either the level-based BFS or depth-based DFS is memory-intensive, especially in large dungeons which requires a relative deep recursion to generate strategies. Another key issue is the circle detection dependence on the dungeon structure. For terrains that are too

complex, DFS results in a high computational load when discovering cycles in a large dungeon or a dungeon with too many spiral corridors. However, for simple dungeon layouts with fewer circles, the effectiveness of the DFS may be diminished. These weaknesses may lead to potential latency and hamper the system performance.

- **Non-trivial Dungeons Proof**

1. Immediate Access to Cycles – Dungeon Test 1, J

Circle is the main tool used to escape from the monster infinitely in this game. When the rogue is initially placed near the entrance to cycles, it can rapidly loop into these interconnected paths via the DFS while maintain a safe distance from the monster through BFS, making it difficult for the monster to catch it in such a dynamic and continuous environment. Due to the limitation of the distance interval and cyclicity of the circle, the rogue can thus manage to stay out of the monster reach. For instance, the left figure below takes the independently created dungeon test 1 as an example where the rogue is initially sited at the entrance to the cycle while the monster is set to the corner. The rogue evaluates potential moves using BFS to ensure that each move increases the distance from the monster or keeps it at a safe distance, while using DFS to identify a safer zone. Similarly for the right one, the rogue is one step closer to the cycle entrance than the monster, which enables it to start continuous cyclic escape before capture.

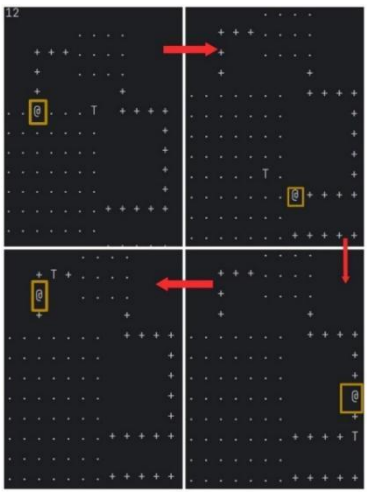


Figure 8: Dungeon Test 1 Results

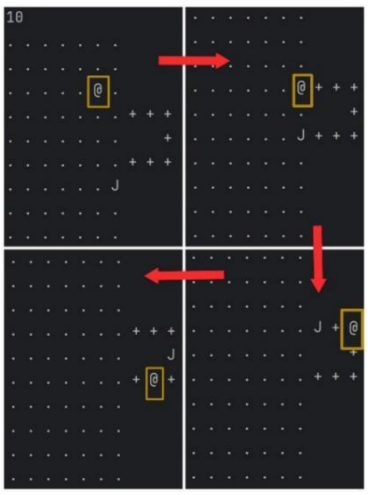


Figure 9: Dungeon J Results

2. Initially Distant and Complex Terrain – Dungeon Test 2, Test 3

In scenarios where the monster and rogue start far apart in a complex terrain that equipped with a few circles, the rogue can apply the DFS to identify all possible cycles to escape in case any of them is intercepted. The intricate layout increases the possibility of the circle generation, facilitating the cycle detection process for the rogue. Moreover, initial positions with a far distance provides the rogue with sufficient space to discover the best path if exists. For example, two self-designed dungeon tests with their corresponding results are shown below. Test 2 displays a complicated structure where the rogue chose to stay stationary in the corner to keep a safe distance and escaped cyclically upon the arrival of the monster. Test 3 is designed with two corridor-based circles and illustrates the partial looping procedure of the chasing.

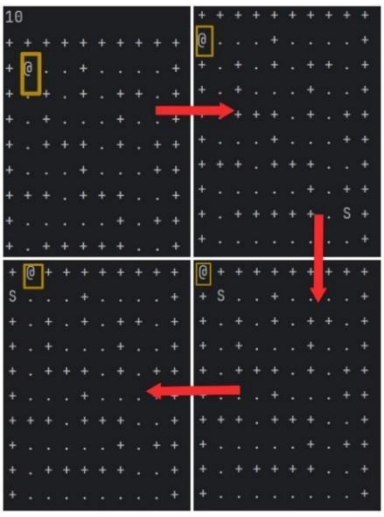


Figure 10: Dungeon Test 2 Results

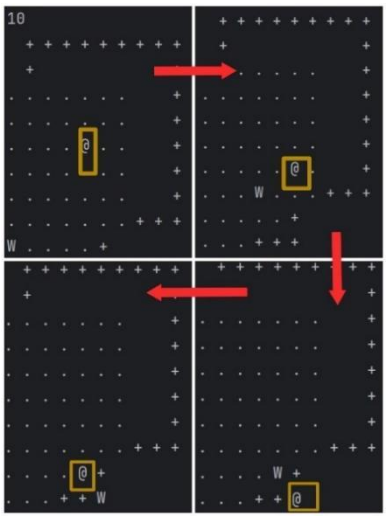


Figure 10: Dungeon Test 3 Results

3. Monster Location Near the Cycle Entrance – Dungeon L

The purpose of the rogue is to discover infinite cycle using DFS, however, it requires the rogue to approach the monster when moving towards the cycle entrance, which increases the risk of capture. Furthermore, the rogue employs the avoidance algorithm that may lead it away from the cycle. This kind of scenario may result in the death of the rogue before entering into the circle. The dungeon L describes a such process where the monster is

located in the right of the cycle entrance while the rogue starts form the bottom. Although there existed a northern circle in the dungeon, the rogue was caught by the monster during its attempt to reach it.

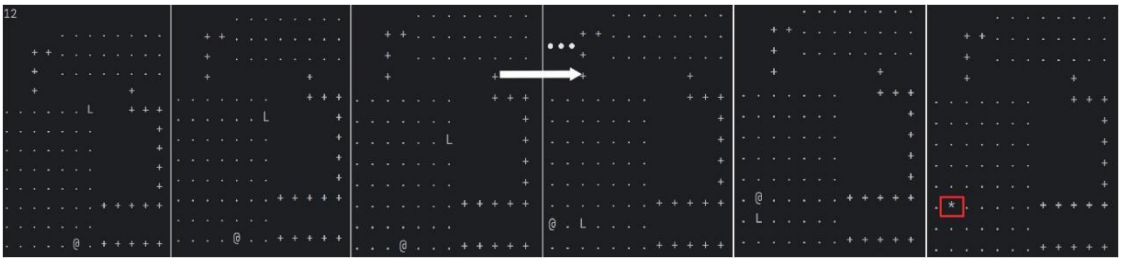


Figure 11: Dungeon L Results

Monster Algorithm Analysis

- **Algorithm Implementation**

The Monster algorithm is designed based on a typical graph algorithm: Breadth-First Search (BFS) to determine the shortest chase path towards the Rogue in the unweighted dungeon. The implementation procedure of the BFS, along with corresponding extension, the Monster algorithm, are summarized as the pseudo codes that are listed below. Specifically, the BFS algorithm is implemented through an array-based queue simulation with the starting position of the Monster. Moreover, it also maintains a 2-dimension array to record distances from the start to each cell, initially marking all sites as unvisited. The algorithm proceeds by dequeuing the current position and exploring all possible moves in eight directions (up, down, left, right, and diagonals). For each valid move, including coordinates that are within the dungeon bounds, not a wall, and unvisited, the new position is enqueued with its distance is updated. The process repeats until the queue is empty or the Rogue position is reached. The first occurrence where the Monster position coincides with that of the Rogue signifies the attainment of the minimum path length, which represents the ideal strategy for the Monster and the first movement in the path is considered as the next behavior for the Monster.

Algorithm 1 BFS Algorithm

Require: *monster*, *rogue*, *dungeon*
Ensure: Shortest path from *monster* to *rogue* if one exists

```
Initialize queue q with monster
Initialize distance array d with −1 for all cells
Set d[monster] to 0
Initialize prev array with null for all cells
while queue is not empty do
    site ← dequeue from q
    for each direction in possible directions (8 directions) do
        x ← site.x + direction.x
        y ← site.y + direction.y
        move ← (x, y)
        if dungeon.isLegalMove(site, move) and d[x][y] == −1 then
            d[x][y] ← d[site.x][site.y] + 1
            prev[x][y] ← site
            Enqueue move into q
        end if
    end for
end while
path ← empty list
x, y ← rogue.x, rogue.y
if d[x][y] ≠ −1 then
    Append rogue to path
    while x ≠ monster.x or y ≠ monster.y do
        t ← prev[x][y]
        if t.x ≠ monster.x or t.y ≠ monster.y then
            Prepend t to path
        end if
        x, y ← t.x, t.y
    end while
end if
return path
```

Algorithm 1: Breadth-First-Search (BFS)

Algorithm 2 Monster Algorithm

Require: *game*, *algorithm*
Ensure: Next move for the monster

```
monster ← game.getMonsterSite()
rogue ← game.getRogueSite()
move ← monster
path ← algorithm.bfs(monster, rogue)
steps ← length of path
if steps > 0 then
    move ← path[0]
end if
return move
```

Algorithm 2: Monster Algorithm

- **Efficiency Analysis**

1. Time Complexity for BFS

A. Initialization: Initializing the visiting order queue or distance array takes O(V), where V is the number of cells in the grid, N*2N.

B. Traversal: In the while loop, queue management such as the comparison and updates of pointers to the head and tail requires a total of O(n^2)

operations. For each site, the number of edges is represented by potential movements and checking eight possible directions is a constant number of operations, O(1). Then the time for searching for all edges is the product of nodes and directions, which is O(n^2). All relevant processing for legal movements such as increasing the distance record and setting the previous nodes also take a work O(1).

C. Path Reconstruction: Tracing back the rogue-to-monster path involves following the recorded predecessors, which in the worst case is O(n^2).

Combining these, the overall time complexity of BFS is O(n^2).

2. Time Complexity for Monster

According to the Monster pseudo codes above, it is noticed that the most significant operation of the *move()* is to invoke the BFS algorithm, which consumes O(n^2) in total. Other behaviors like obtaining initial object positions or determining the next movements simply needs a constant time O(1) and therefore the total time complexity is O(n^2).

3. Space Complexity for BFS

Variables in the algorithm such as *N, head,* and *tail* only requires unit spaces in the memory. However, the space complexity is determined by the most crucial components such as the visiting queue, previous nodes array and distance records array, which takes O(N*2N) in total. Therefore the space complexity of the BFS is O(n^2).

4. Space Complexity for Monster

Similar to the BFS, fields for storing locations like *monster* and *rogue* simply takes O(1) memory while the BFS component will utilize O(n^2) space for its queue and arrays and thus the total complexity is same as the BFS, O(n^2).

Rogue Algorithm Analysis

- **Algorithm Implementation**

The Rogue algorithm employs both BFS and DFS to generate dynamic escape strategy. The former focuses on determining the shortest path between certain

points such as the monster and the rogue or the rogue and the cycle entrance. For the latter algorithm, it mainly aims to detect cycles that contain corridors to form infinite loop. The BFS implementation is similar to the application in the Monster algorithm, pseudo codes below detail the core logic of DFS and Rogue.

Algorithm 3 DFS Algorithm

Require: *current_site*, *steps*
Ensure: Updates the solution if found
  if *steps* $>$ 50 **then**
    mark_solution_as_found()
    return
  end if
  if solution_found **then**
    return
  end if
  if *current_site* is the rogue and passed through hallway and *circular_path.length* $>$ 2 **then**
    if *steps* $<$ *minimum_steps* **then**
      update_best_solution()
    end if
    return
  end if
  if *current_site* is the rogue and *steps* $\neq$ 1 **then**
    return
  end if
  for each possible direction **do**
    next_site $\leftarrow$ *calculate_next_site*(*current_site*, *direction*)
    if *next_site* is legal and not visited **then**
      mark_as_visited(*next_site*)
      if *next_site* is hallway **then**
        increment_hallway_counter()
      end if
      add_to_path(*next_site*)
      dfs(*next_site*, *steps* + 1)
      if *next_site* is hallway **then**
        decrement_hallway_counter()
      end if
      remove_from_path(*next_site*)
      mark_as_unvisited(*next_site*)
    end if
  end for

Algorithm 3: Depth-First Search (DFS)

The DFS is implemented using recursion and backtracking to explore as far as possible in the dungeon. Specifically, it detects four directions with the judgement on legality, visiting status and category of each specific site. If a position satisfies all conditions, then it will be updated and added to the potential best path while the DFS will continue extending to the next depth, otherwise the algorithm will backtrack to the previous depth and withdraw relevant marks on that node. Moreover, the pruning is utilized to optimize the searching efficiency. For example, in the base situation, comparison of searching

depth with the limitation 50 or the current path length with previous recorded best route distance facilitates the cycle detection procedure.

Algorithm 4 Rogue Algorithm

```
Require: Current positions of rogue and monster
Ensure: Determine the next move for the rogue
  Get current positions of rogue and monster
  if rogue is in a corridor then
    Mark rogue as in a circle
  end if
  if first iteration and (not in a circle or self-circling) then
    if dungeon has corridors then
      Run DFS from rogue to find best path
      Retrieve best path
    end if
  end if
  if best path is found then
    Process the best path
    if rogue is in a room and (not in a circle or self-circling) then
      Identify path endpoints
      Adjust path if needed
    end if
    if path to monster is longer from the next step in best path or safe path
    exists then
      Update move position to the next step in best path
    else
      Find the longest path away from the monster as a fallback move
    end if
  else
    Find the longest path away from the monster as a fallback move
  end if
  return move position
```

Algorithm 4: Rogue Algorithm

The Rogue strategy with pseudo codes above is designed to balance the gap between avoiding corner capture (if consider escaping as far as possible only) and ensuring access to cyclical paths without being intercepted (if consider direct routing to the cycle only). Specifically, the rogue assumes a loop exists if it is in a corridor, enabling effective retreat. Otherwise, the algorithm searches for corridors and stops at the first found and employs the DFS to explore the loop and generate the best path. Upon best path discovery, the strategy involves checking the rogue position: if in a room, it identifies entrance and exit to the loop and adjusts the path to maximize distance from the monster if necessary. Furthermore, the rogue moves based on whether it increases or maintains a safe distance from the monster via BFS. If the conditions are not met, which means that approaching the loop path brings the rogue closer to the monster, the strategy opts instead for evasion. This process is iteratively checked and updated

within the game loop, ensuring the rogue always escapes in the optimal direction to maintain safety. According to the above analysis on both Monster and Rogue strategy, the overall logic flow of the game is illustrated in the activity diagram below.

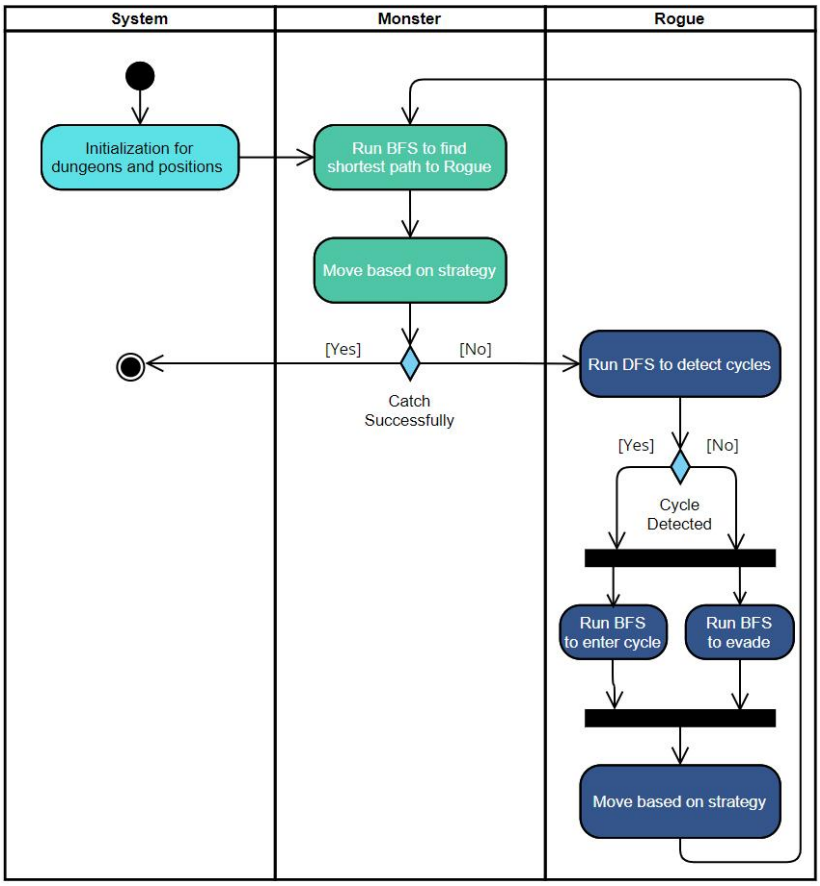


Figure 12: Overall Game Logic

- **Efficiency Analysis**

1. Time Complexity for DFS and Rogue

The time complexity of the DFS is subject to the most significant and complicated operations which refer to the recursive call *dfs()* in this algorithm. Each recursion explores 4 possible directions, leading to a branching factor of 4. Pruning conditions such as *isOk* or equality of certain site and the rogue position were applied to improve the algorithm efficiency, however, to calculate the overall time complexity, the running time in the worst case should be considered as the standard, which is O(4^(N*2N)). Similarly, the *move()* function dominates the time complexity of the Rogue

algorithm, which utilizes both BFS and DFS. The asymptotic time O(n^2) of BFS is illustrated above, which is faster than that of DFS. Therefore, both DFS and Rogue has the time complexity of O(2^n)

2. Space Complexity for DFS and Rogue

In DFS, instances and local variables such as *passHallway*, *minix* and *cur* require unit spaces for storing. However, similar to the time efficiency, the space complexity is proportional to the memory usage of complex data structure such as queue or list. Specifically for DFS, the visiting status boolean array *st* is set to N*2N to record site states of the whole dungeon, which leads to the space complexity O(n^2). Such growth is asymptotically equal to the value of BFS, as the Rogue combines two graph algorithms with same space complexity, therefore the overall space complexity of the Rogue is O(n^2).

Reference

1. Korunović, A., & Vlajic, S. (2023). Example of Integration of Java GUI Desktop Technologies Using the Abstract Factory Pattern for Education Purposes. *ETF Journal of Electrical Engineering*.
2. Wang, S., Ding, L., Shen, L., Luo, Y., Du, B., & Tao, D. (2024). OOP: Object-Oriented Programming Evaluation Benchmark for Large Language Models. *ArXiv, abs/2401.06628*.
3. Issariyakul, T., & Hossain, E. (2012). A Review of the Polymorphism Concept in OOP.
4. P.E., N., F.U, O., A.U., D., J.S., I., & N.H., O. (2023). Detailed Study of the Object-Oriented Programming (OOP) Features in Python. *British Journal of Computer, Networking and Information Technology*.
5. Aliyan, M., Hasan, M.Z., Qayoom, H., Hussain, M.Z., Nosheen, S., Mustafa, M., Chuhan, S.H., Atif Yaqub, M., Awan, R., & Bilal, A. (2024). Analysis and Performance Evaluation of Various Shortest Path Algorithms. *2024 3rd International Conference for Innovation in Technology (INOCON)*, 1-16.

My Java Code

- **AlgorithmFactory**

```
// Factory pattern for algorithm desgin
public class AlgorithmFactory {
	private Game game;
	public AlgorithmFactory(Game game) {
		this.game = game;
	}
	// Instantiate the algorithm
	public AlgorithmBase createAlgorithm(){
		return new Algorithm(game);
	}
}
```

- **AlgorithmBase**

```
import java.util.List;
// Basic abstract class for algorithm to implement
public abstract class AlgorithmBase {
	public abstract void flush(); // Refresh the game state
	public abstract List<Site> bfs(Site monster, Site rogue); // BFS to detect the shortest path
	public abstract void dfs(Site cur,int u); // DFS to find cycles
	public abstract List<Site> getBest(); // Return the optimal route
}
```

- **Algorithm**

```
import java.util.ArrayList;
import java.util.List;

public class Algorithm extends AlgorithmBase{
	Site[] visitOrder; // Sites to be visited
```

```
int[] dx = {1, 0, 0, -1, 1, 1, -1, -1}, dy = {0, 1, -1, 0, 1, -1, 1, -1}; // Eight directions
public int[][] distance; // Distance to the monster
public Site[][] prev; // Previous node of each site
int n; // Dungeon size
private Dungeon dungeon;
Site monster;
Site rogue;
private List<Site> circle = new ArrayList<>(); // Circle path
private List<Site> best = new ArrayList<>(); // Optimal route
int passHallway = 0; // The number of passed corridors
boolean[][] visited; // Visiting status of each site
int minix = 0x3f3f3f3f; // Pivot for comparison of the shortest path
boolean found = false; // Whether find the best route

public Algorithm(Game game){
    this.dungeon = game.getDungeon();
    this.n = dungeon.size();
    distance = new int[n][n];
    prev = new Site[n][n];
    visitOrder = new Site[n*n];
    visited = new boolean[n][n];
    monster = game.getMonsterSite();
    rogue = game.getRogueSite();
    minix = 0x3f3f3f3f;
    found = false;
    circle = new ArrayList<>();
    best = new ArrayList<>();
}

// Refresh the game state
```

```
    public void flush(){
        minix = 0x3f3f3f3f;
        found = false;
        circle = new ArrayList<>();
        best = new ArrayList<>();
    }

    // BFS implementation
    public List<Site> bfs(Site monster, Site rogue){
        int head = 0, tail = 0; // Position of the head and tail of the visiting queue
        distance = new int[n][n];
        prev = new Site[n][n];
        visitOrder = new Site[n*n];
        visitOrder[tail++] = monster; // Start visiting from the monster position
        // Initialize the distance
        for (int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                distance[i][j] = -1;
            }
        }

        distance[monster.i()][monster.j()] = 0;

        while(head < tail){ // If queue is empty
            Site site = visitOrder[head++]; // Extract the site to be visited and move the pointer
            // Traverse eight directions
            for(int i = 0; i < 8; i++){
                int x = dx[i] + site.i(), y = site.j() + dy[i];
                Site move = new Site(x,y);
```

```
                if (dungeon.isLegalMove(site,move) && distance[x][y] == -1){
                    distance[x][y] = distance[site.i()][site.j()] + 1;
                    prev[x][y] = site;
                    visitOrder[tail++] = move;
                }
            }
        }
        // Backtracking from the rogue to monster to record the path
        List<Site> path = new ArrayList<>();
        int x = rogue.i(), y = rogue.j();
        if(distance[x][y] != -1){
            path.add(0, rogue);
            while(x != monster.i() || y != monster.j()){
                Site pre = prev[x][y];
                if(pre.i() != monster.i() || pre.j() != monster.j()){
                    path.add(0, pre);
                }
                x = pre.i();
                y = pre.j();
            }
        }

        return path;
}

// DFS implementation
public void dfs(Site cur, int depth){
        // Pruning for too deep recursion
        if(depth > 50){
            found = true;
```

```
        best = new ArrayList<>();
        return;
    }
    // Find the best path successfully
    if(found){
        return;
    }
    // Update the best cyclic path
    if(cur.equals(rogue) && passHallway > 0 && circle.size() > 2){
        // Pruning for best path update
        if(depth < minix){
            minix = depth;
            best.addAll(circle);
            found = true;
        }
        return;
    }
    // Pruning for redundant traversal: only detect four directions
    for(int i = 0; i < 4; i++){
        int x = cur.i() + dx[i];
        int y = cur.j() + dy[i];
        Site move = new Site(x,y);
        if ((dungeon.isLegalMove(cur,move) && visited[x][y] == false)){
            if(dungeon.isCorridor(move)){
                passHallway++;
            }
            visited[x][y] = true;
            circle.add(move);
            // Recursive call to detect by depth
            dfs(move,depth+1);
```

```
                    // Backtracking
                    if(dungeon.isCorridor(move)){
                        passHallway--;
                    }
                    circle.remove(circle.size() - 1);
                    visited[x][y] = false;
                }
            }
        }

        // Return the optimal path
        public List<Site> getBest(){
            return this.best;
        }
}
```

- **Monster**

```
import java.util.List;

public class Monster{
        private Game game;
        private Dungeon dungeon;
        private int N;
        private AlgorithmBase algorithm;
        private AlgorithmFactory factory;
        public Monster(Game game) {
            factory = new AlgorithmFactory(game);
            this.game = game;
            this.dungeon = game.getDungeon();
            this.N = dungeon.size();
            algorithm = factory.createAlgorithm();
```

```
    }

    // Take an optimal move towards the rogue
    public Site move() {
        Site monster = game.getMonsterSite();
        Site rogue = game.getRogueSite();
        Site move = monster;

        List<Site> path = algorithm.bfs(monster,rogue);
        int steps = path.size();
        if (steps > 0){
            move=path.get(0);
        }

        return move;
    }
}
```

- **Rogue**

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Rogue{
    private Game game;
    private Dungeon dungeon;
    private int N;
    private AlgorithmBase algorithm;
    private AlgorithmFactory factory;
    List<Site> best = null;
    private static boolean inCircle=false;
```

```
public Rogue(Game game) {
        factory = new AlgorithmFactory(game);
        this.game = game;
        this.dungeon = game.getDungeon();
        this.N = dungeon.size();
        algorithm = factory.createAlgorithm();
}
// The position of the rogue in the circle
private static int position = 0;
// If the dungeon has a corridor-based circle
private static boolean selfCircle = false;
// Take an optimal move
public Site move() {
        Site monster = game.getMonsterSite();
        Site rogue = game.getRogueSite();
        Site move = rogue;
        // Assume the corridor is a part of the circle, which can avoid the bug
through size judgement
        if(dungeon.isCorridor(rogue)){
                inCircle = true;
        }
        boolean circleFound = false;
        algorithm.flush();
        if(position == 0 && (inCircle == false || selfCircle == true)){
                for (int i = 0; i < N; i++) {
                        for (int j = 0; j < N; j++) {
                                Site site = new Site(i, j);
                                // Assume the dungeon has a circle if it has corridors
                                if (dungeon.isCorridor(site)){
                                        circleFound = true;
```

```
                    }
                }
                if(circleFound){
                    break;
                }
            }

            if(circleFound){
                algorithm.dfs(rogue, 1);
                best = algorithm.getBest();
                System.out.println(best.size());
            }else{
                best = new ArrayList<>();
            }
        }

        // The path exists
        if(best.size()>0){
            List<Site> newBest = new ArrayList<>();
            boolean inCorridor = false;
            boolean loopFormed = false;
            // Entrance and exit of the corridor
            Site in = null;
            Site out = null;
            if(dungeon.isRoom(rogue) == true && (inCircle == false || selfCircle ==
true)){
                for(Site site : best){
                    if (dungeon.isRoom(site) && inCorridor == true &&
loopFormed == false){
                        if(algorithm.bfs(newBest.get(newBest.size() - 1),
```

```
in).size() == 1){
                                selfCircle = true;
                                break;
                        }
                        out = site;
                        newBest.add(site);
                        // Replace the original path with the optimal route
from the rogue to the cycle with its entrance and exit
                        newBest.addAll(algorithm.bfs(site, rogue));
                        inCorridor = false;
                        break;
                }
                if(dungeon.isCorridor(site) && inCorridor == true){
                        newBest.add(site);
                }
                if (dungeon.isCorridor(site) && inCorridor == false){
                        newBest.addAll(0,algorithm.bfs(rogue, site));
                        in = site;
                        inCorridor = true;
                }
        }
        // If the distance to the exit is less than that to the entrance,
reverse it
        if(out != null && algorithm.bfs(rogue, out).size() <
algorithm.bfs(rogue, in).size()){
                Collections.reverse(newBest);
                newBest.add(newBest.get(0));
                newBest.remove(0);
        }
        best = newBest;
```

```
                position = 0;



            }

            // The best path is legal and reliable
            if((algorithm.bfs(best.get(position%best.size()), monster).size() >= algorithm.bfs(rogue, monster).size() && algorithm.bfs(rogue, monster).size() > 1) || algorithm.bfs(best.get(position%best.size()), monster).size() > 1){
                move = best.get(position%best.size());
                position++;
            }else{
                // Evade as far as possible otherwise
                position = 0;
                List<Site> dist = algorithm.bfs(rogue, monster);

                int[] dx = {-1, 0, 1, 0, 1, 1, -1, -1}, dy = {0, 1, 0, -1, 1, -1, 1, -1};
                int remote = dist.size();
                for(int i = 0; i < 8; i++){
                    int x = dx[i] + rogue.i(), y = rogue.j() + dy[i];
                    Site t = new Site(x,y);
                    if(dungeon.isLegalMove(rogue, t)){
                        int dist_t = algorithm.bfs(t, monster).size();
                        if (dist_t >= remote){
                            remote = dist_t;
                            move = t;
                        }
                    }
                }
            }
```

```
            }else {
                // If the best path does not exist, evade
                List<Site> dist = algorithm.bfs(rogue, monster);
                int[] dx = {-1, 0, 1, 0, 1, 1, -1, -1}, dy = {0, 1, 0, -1, 1, -1, 1, -1};
                int remote = dist.size();
                for(int i = 0; i < 8; i++){
                    int x = dx[i] + rogue.i(), y = rogue.j() + dy[i];
                    Site t = new Site(x,y);
                    if(dungeon.isLegalMove(rogue, t)){
                        int dist_t = algorithm.bfs(t, monster).size();
                        if (dist_t >= remote){
                            remote = dist_t;
                            move = t;
                        }
                    }
                }
            }

            return move;
        }
}
```

- **Site**

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

    // Initialize board from file
    public Site(int i, int j) {
        this.i = i;
        this.j = j;
```

```
    }

    public int i() { return i; }
    public int j() { return j; }

    // Manhattan distance between invoking Site and 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);
    }

    // Equality of two sites
    public boolean equals(Site w) {
        return (manhattanTo(w) == 0);
    }

}
```

- **Dungeon**

```
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

    // 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] == '+'){
                    isCorridor[i][j] = true;
                }
            }
        }
    }

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

    // 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();
        if (i1 < 0 || j1 < 0 || i1 >= N || j1 >= N) return false;
        if (i2 < 0 || j2 < 0 || i2 >= N || j2 >= N) return false;
        if (isWall(v) || isWall(w)) return false;
        if (Math.abs(i1 - i2) > 1)  return false;
        if (Math.abs(j1 - j2) > 1)  return false;
        if (isRoom(v) && isRoom(w)) return true;
        if (i1 == i2)               return true;
        if (j1 == j2)               return true;

        return false;
    }
}
```

- **In**

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
```

```
class In {
    private BufferedReader br;

    // Read from the file input
    public In(String filename) {
        try {
            FileReader fr = new FileReader(filename);
            br = new BufferedReader(fr);
        } catch (FileNotFoundException e) {
            System.err.println("The file is not found: " + filename);
            e.printStackTrace();
        }
    }

    // Return rest of line as string and return it, not including newline
    public String readLine() {
        String s = null;
        try { s = br.readLine(); }
        catch(IOException ioe) { ioe.printStackTrace(); }
        return s;
    }
}
```

- **Game**

```
import java.util.Scanner;

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 Monster monster;         // The monster
private Rogue rogue;             // The rogue

// 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);
            }
        }
```

```
        }
        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;   }

// Return the dungeon structure
public Dungeon getDungeon()  { return dungeon;     }

// Play until the monster catches the rogue
public void play() {
        for (int t = 1; true; t++) {
                System.out.println("Move " + t);
                System.out.println();

                // Monster moves
                if (monsterSite.equals(rogueSite)) break;
                Site next = monster.move();
                if (dungeon.isLegalMove(monsterSite, next)) monsterSite = next;
                else throw new RuntimeException("Monster caught cheating");
                System.out.println(this);

                // Rogue moves
                if (monsterSite.equals(rogueSite)) break;
                next = rogue.move();
                if (dungeon.isLegalMove(rogueSite, next)) rogueSite = next;
```

```
            else throw new RuntimeException("Rogue caught cheating");
            System.out.println(this);
        }

        System.out.println("Caught by monster");

    }

    // String representation of game state
    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;
    }


    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Please enter the absolute path of the file： ");
        String filePath = scanner.nextLine();
        In in = new In(filePath);
        Game game = new Game(in);
        game.play();
    }
}
```