Implementation of A* Algorithm for Exfiltration Route Optimization in DMZ Mode on Koschei Complex Map of Call of Duty: Warzone 2.0

Christopher Brian - 13522106
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522106@std.stei.itb.ac.id

***Abstract*—DMZ is an extraction shooter game mode featured in Call of Duty: Warzone 2.0 where players can complete objectives while surviving from other players and bots, and save their progress after a success exfiltration. This paper will only focus on the Koschei Complex Map of the game mode. Considering the survival element, an optimal route planning for exfiltration that minimizes time is important for minimizing the probability of meeting other players and increasing the success rate of exfiltration. Fortunately, route optimization is both possible and well established with the use of algorithms such as A*. This paper introduces a solution utilizing A* algorithm to find the optimal route in an efficient way.** *(Abstract)*

Keywords—A*; Call of Duty; DMZ; pathfinding; route optimization; Warzone

I. Introduction

DMZ is a mode featured in Call of Duty: Warzone 2.0. It is an extraction mode shooter game with a sandbox experience, giving players a large space for creativity in completing in game objectives and surviving. The mode is available with the release of Warzone 2.0 (coinciding with the release of Call of Duty: Modern Warfare II) in late 2022. DMZ takes place in several locations called “Exclusion Zones”, namely Al Mazrah, Building 21, Ashika Island, Koschei Complex, and Vondel.

Fig. 1. DMZ Game Mode in Call of Duty: Warzone 2.0
(Source: https://www.forbes.com/sites/erikkain/2023/12/01/activision-just-killed-call-of-dutys-dmz-mode-but-you-can-still-sort-of-play-it-for-now/)

In this game mode, players can complete objectives in the forms of various faction missions, as well survive both enemy players and bots in order to fight their way over to exfiltration. Players can also keep valuable loots and items once they successfully exfiltrate.

At the start of the game mode, players, either as a solo or part of a team of maximum three players, will randomly spawn in one of the multiple spawn locations in the map they chose at the start of the matchmaking process. Players will then be given a set time (different for each map) to exfiltrate before the time ends, or otherwise die and lose all their valuable loots. There are multiple exfiltration sites on the map in each match, therefore players are able to choose the nearest exfiltration site. However, players can still choose to exfiltration in any exfiltration sites on the map.

Every map in DMZ is full of enemy players and aggressive bots which will continue to attack and try to kill you once aggravated. While players can choose to play carefully and use stealth to avoid confrontation with enemy players and bots, the random and complex nature of the mode makes it hard for players to play this way in each match. Players can exfiltrate right from the start of the match, but spending more time in the match will give players more chance to complete their objectives and gain valuable loots, as well as increase their chance of meeting enemy players and bots.

Players’ success in DMZ is directly tied to exfiltration success, considering the only other way to end the match is by automatically dying once the set time expired. In order to maximize their success rate of exfiltration, players need to design and implement a robust strategy.

One such strategy is planning an optimal route that maximize players’ survivability until exfiltration. In this game mode, players’ survivability can mostly be attributed to their chance of meeting enemy players and bots. It can be deducted that in order to minimize this chance, players need to go straight to the exfiltration site as soon as possible when they choose to end the match. Therefore, the correct strategy to apply here is a route optimization strategy from the players location to the nearest exfiltration site.

A* algorithm is a popular route-finding algorithm that is used to find the shortest path(s) between two locations. This algorithm considers the cost to reach a specific node, and a heuristic or estimated extra cost to reach the goal node from that specific node to guide the searching process. A* algorithm go through the graph by iteratively choosing the node with the minimum total cost, which is the sum of the cost to reach the node from the start node, and the estimated cost of reaching the end node. A* algorithm is also popularly used in puzzle solving games and artificial intelligence.

In this game mode, A* algorithm is used to find the optimal route from a player's location to the nearest exfiltration site in the map. This paper will explore how A* algorithm is implemented to find the shortest route from a player's location to the nearest exfiltration site.

II. BASIC THEORY

A. *A* Algorithm*

A* Algorithm is one of the most frequently used algorithms for optimal pathfinding. The algorithm could be described as an improvement of the older Dijkstra's algorithm. However, A* algorithm works more efficiently by using heuristics to guide its search. Compared to Dijkstra's algorithm where the goal node is every other node excluding the start node, A* algorithm only has one goal node.

A* algorithm is classified as an informed search, defined as any search algorithm where the information about the end node is known, which allows it to find the optimal route from the start node to the end node in a shorter time. The optimal route is the route with the minimum total cost. The cost used in the algorithm could be in various forms, including but not limited to number of steps, distance, time, and price, depending on the problem being solved. The algorithm works by maintaining a tree of paths, beginning for the start node and expanding each node in an order until it finally reaches the end node.

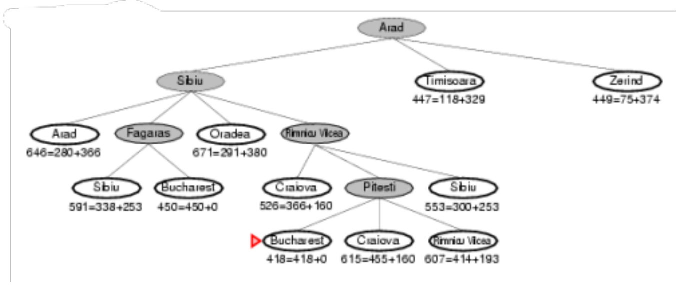


Fig. 2. Illustration of A* Algorithm Path Tree
(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf)

In the case of tree expansion, A* algorithm have to pick the path that should be prioritized to be expanded next. To do this, the algorithm will choose the path that minimizes (or maximizes, depending on the type of the problem) the total cost, which is the sum of the cost from the start node to the node, and the estimated cost from that node to the end goal. Mathematically, it can be expressed by the formula,

$$f(n) = g(n) + h(n)$$

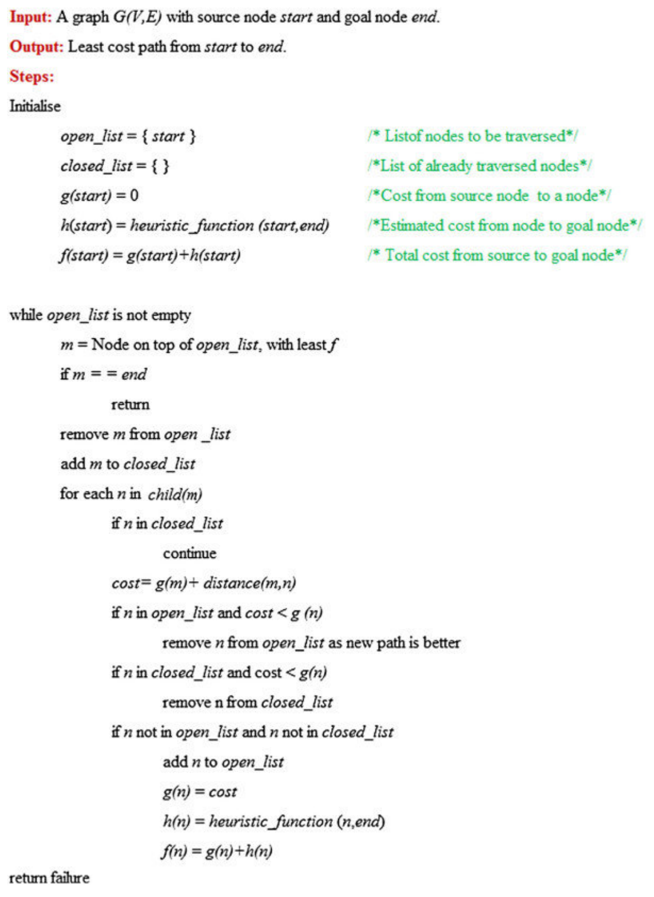
For. 1. Formula for the total cost of a node
(Source: author's documentation)

where g(n) is the cost to reach a node from the start node and h(n) is the heuristically estimated cost to reach the end node from that node. The sum of both is equal to f(n), which will then be minimized or maximized when choosing a node in every step.

In general, the steps of A* algorithm could be expressed as so.

1. Initialize the priority queue.
2. Choose the start node and set it as the current node.
3. Stop search if the current node is a goal node.
4. Find all children node of the current node and insert it to the priority queue according to the total cost.
5. Remove the element with the lowest cost from priority queue and set it as the current node.
6. Repeat steps 3-6 until the optimal path is formed.

Here is an example of a pseudocode for implementing the A* algorithm.



```
Input: A graph G(V,E) with source node start and goal node end.
Output: Least cost path from start to end.
Steps:
Initialise
    open_list = { start }                          /* Listof nodes to be traversed*/
    closed_list = { }                              /*List of already traversed nodes*/
    g(start) = 0                                   /*Cost from source node  to a node*/
    h(start) = heuristic_function (start,end)      /*Estimated cost from node to goal node*/
    f(start) = g(start)+h(start)                   /* Total cost from source to goal node*/

while open_list is not empty
    m = Node on top of open_list, with least f
    if m = = end
        return
    remove m from open _list
    add m to closed_list
    for each n in  child(m)
        if n in closed_list
            continue
        cost= g(m)+ distance(m,n)
        if n in open_list and cost < g (n)
            remove n from open_list as new path is better
        if n in closed_list and cost < g(n)
            remove n from closed_list
        if n not in open_list and n not in closed_list
            add n to open_list
            g(n) = cost
            h(n) = heuristic_function (n,end)
            f(n) = g(n)+h(n)
return failure
```

Fig. 3. A* algorithm pseudocode example
(Source: https://www.researchgate.net/figure/A-search-algorithm-Pseudocode-of-the-A-search-algorithm-operating-with-open-and-closed_fig8_232085273)

B. *Admissible A* Algorithm Heuristic*

In order to from the globally optimal solution, the heuristic function (h(n)) must be admissible, meaning the value of said function should never overestimate the actual cost from a specific node to the end node. Otherwise, it is possible for the algorithm to choose the wrong step when forming the route.

An admissible heuristic function (h(n)) is formally defined as so. For every node n, h(n) satisfies $h(n) \leq h^*(n)$, where $h^*(n)$ is the actual minimum cost from the node n to the end node. Therefore, an admissible heuristic function is a heuristic function that is optimistic in nature.

While an exact heuristic will calculate the exact value of h in some algorithm, such as Dijkstra's algorithm, it is also time consuming. This particular technique includes pre-computing the distance between each node before starting the searching process with the A* algorithm. However, if the problem involves no obstacles between nodes, the exact value can just be easily calculated using formulas such as the Euclidean distance.

On the other hand, approximation heuristic is often less time consuming. Finding the approximation heuristic need not pre-computation process done between each pair of nodes, but instead uses the known information about the end node to estimate the remaining distance or cost between a specific node to the end node. One of the most frequently used formulas to calculate the approximation heuristic is the Manhattan distance.

A Manhattan distance between two nodes is calculated as a sum of absolute values of the differences between the end node's Cartesian coordinates.

```
h = abs(current.x − goal.x) +
      abs(current.y − goal.y)
```

Fig. 4. Pseudocode for calculating Manhattan distance
(Source: https://www.geeksforgeeks.org/a-search-algorithm)

Manhattan distance is ideally used to find the approximation heuristic when the movement is only allowed in four directions only (up, down, right, left).

C. *Exfiltration in DMZ*

DMZ is an extraction shooter game mode where players can move around the map freely and have to exfiltrate before the set time ends to successfully end a match. The game is played in an area of interconnecting paths. Exfiltration is simple, all the player needs to do is to go to the nearest exfiltration site, wait until the exfiltration arrives, and survive until the exfiltration process ends.

In this paper, the author only focused on the Koschei Complex map of the game mode. The reason being the Koschei Complex map is the only map with truly defined paths that connects pairs of intersections in the map. In this map, players can only move in four directions from every start point (up, down, left, right), compared to the other maps where players can generally move freely from any location to any other location on the map.

Fig. 4. Koschei Complex map gameplay
(Source: https://www.charlieintel.com/call-of-duty-warzone/what-is-koschei-complex-in-dmz-warzone-2-area-explained-250087/)

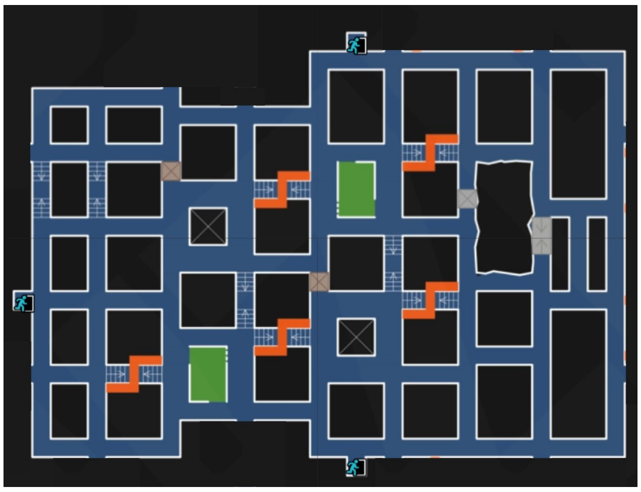


Fig. 5. Koschei Complex map paths and exfiltration sites
(Source: https://wzhub.gg/map/koschei/dmz)

In each match, players will randomly spawn inside the map, given a set time to explore the map and complete objectives, and can choose to successfully end the match by exfiltrating from any of the three exfiltration sites available in the map.

As seen in the image, players are only able to move in four directions, and the paths are well defined. As our goal is to minimize the time to go to the nearest exfiltration site in order to minimize the chance of meeting enemy players and bots, an optimal route for this problem will be the route with the shortest total distance from the player's location to any of the three available exfiltration sites on the Koschei Complex map.

An optimal and robust route would consist of which steps should the player take in order to go from a location to the nearest exfiltration map available in the map. Such strategy would be very crucial in maximizing the success rate of the player and survivability. Considering that if at any time before the exfiltration ends a player dies, all of the progress in that match would be deleted and reset, and the player would have to then start from zero again in the next match. Additionally, if a player is able to successfully exfiltrate in several matches without being killed one time between the matches, the player will gain an extra exfiltration streak which will then reward the player with some more valuable perks.

Fig. 6. An exfiltration site in the Koschei Complex map (Source: https://twinfinite.net/call-of-duty/how-to-complete-bedrock-dmz-mission-warzone-2/)

III. IMPLEMENTATION

A. Mapping into A Algorithm Domain*

1. Solution Space

   The solution space of exfiltration route for DMZ game mode on Koschei Complex map is represented in a vector with n-tuple sized:

   $$X = (x_1, x_2, \dots, x_n)$$

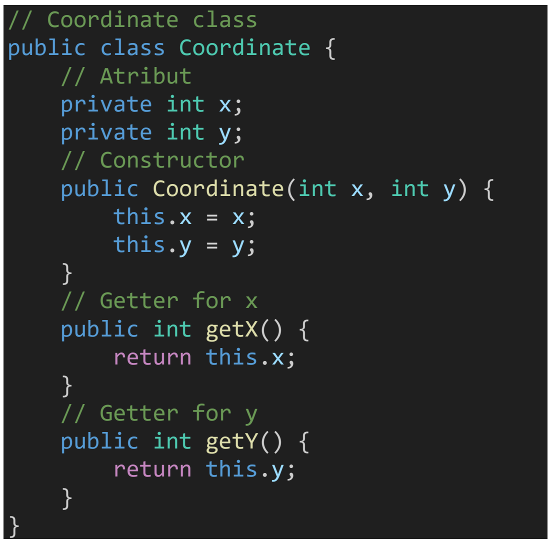
   with $x_1, x_2, \dots, x_n \in$ {list of nodes}

   For. 2. Solution Space of Exfiltration Route for DMZ Game Mode on Koschei Comples Map
   (Source: author's documentation)

   with depth being the number of nodes needed to be traversed in an optimal exfiltration route formed using A* algorithm searching.

2. Coordinate Representation

   Each node will have a cartesian coordinates represented with a Java class as follows.

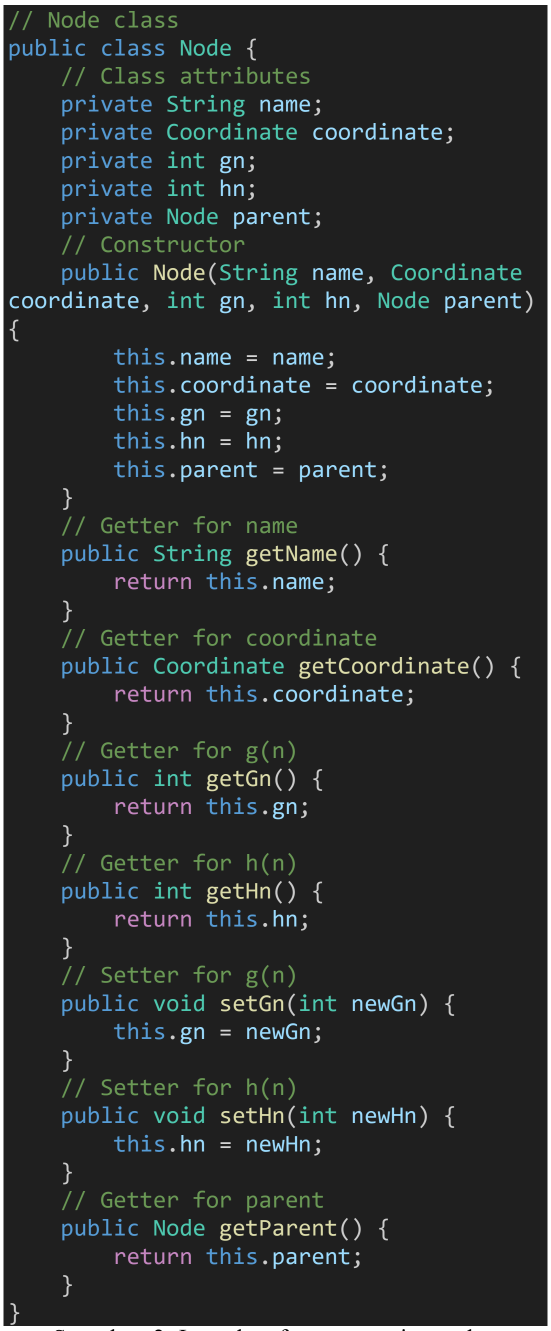


```java
// Coordinate class
public class Coordinate {
    // Atribut
    private int x;
    private int y;
    // Constructor
    public Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }
    // Getter for x
    public int getX() {
        return this.x;
    }
    // Getter for y
    public int getY() {
        return this.y;
    }
}
```

Snapshot. 1. Java class for representing coordinates.
(Source: author's documentation)

3. Node Representation

   In the implementation made, nodes are represented with a Java class as follows.



```java
// Node class
public class Node {
    // Class attributes
    private String name;
    private Coordinate coordinate;
    private int gn;
    private int hn;
    private Node parent;
    // Constructor
    public Node(String name, Coordinate
coordinate, int gn, int hn, Node parent)
{
        this.name = name;
        this.coordinate = coordinate;
        this.gn = gn;
        this.hn = hn;
        this.parent = parent;
    }
    // Getter for name
    public String getName() {
        return this.name;
    }
    // Getter for coordinate
    public Coordinate getCoordinate() {
        return this.coordinate;
    }
    // Getter for g(n)
    public int getGn() {
        return this.gn;
    }
    // Getter for h(n)
    public int getHn() {
        return this.hn;
    }
    // Setter for g(n)
    public void setGn(int newGn) {
        this.gn = newGn;
    }
    // Setter for h(n)
    public void setHn(int newHn) {
        this.hn = newHn;
    }
    // Getter for parent
    public Node getParent() {
        return this.parent;
    }
}
```

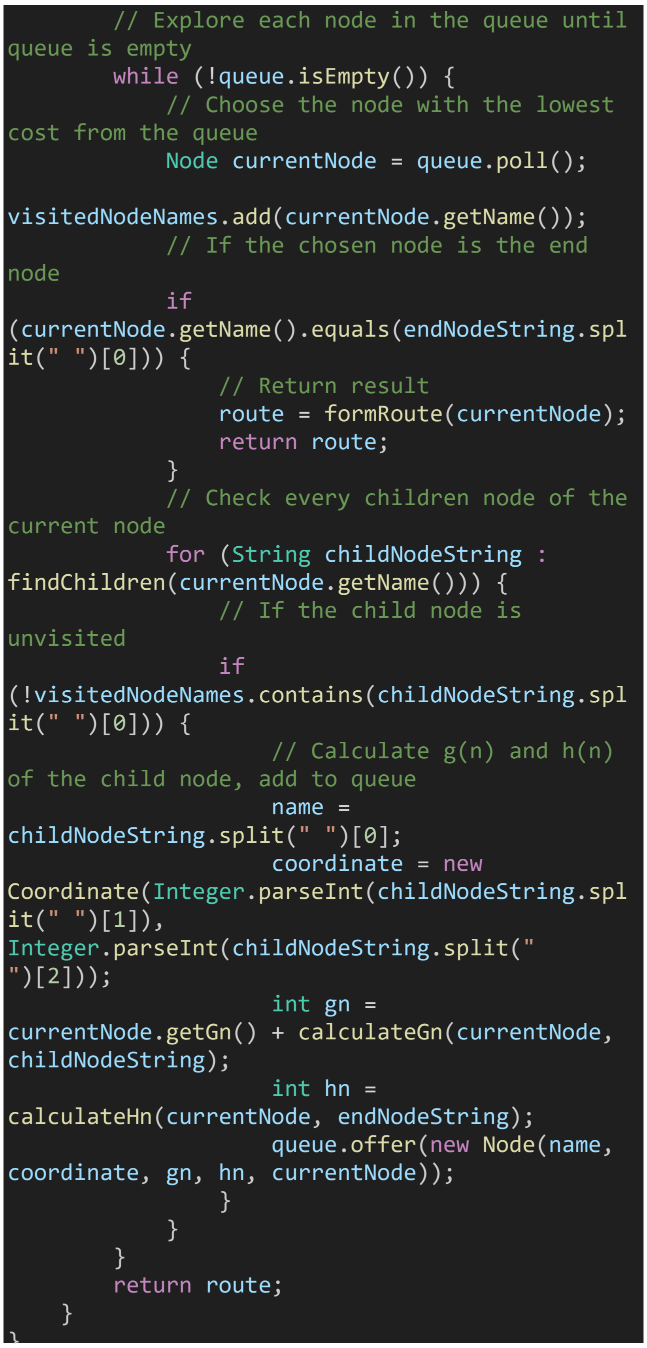
Snapshot. 2. Java class for representing nodes.
(Source: author's documentation)

B. A Main Algorithm*

The A* algorithm implementation will be processed by a Java class as follows.

```java
// Import library
import java.util.*;
// Solver class
public class Solver {
    // Attribute
    private Set<String> map;
    // Constructor
    public Solver (Set<String> map) {
        this.map = map;
    }
    // Function for forming a route from start
to end node
    private List<String> formRoute (Node end)
{
        List<String> route = new
ArrayList<>();
        Node node = end;
        // Form route by recursively calling
the parent node
        while (node != null) {
            route.add(0, node.getName());
            node = node.getParent();
        }
        return route;
    }
    // Function to find all child nodes
    private List<String> findChildren (String
name) {
        List<String> children = new
ArrayList<>();
        String childName;
        // Traverse through the map
        for (String nodeString : map) {
            // The parent node is the part of
the string after the last space
            childName =
nodeString.substring(nodeString.lastIndexOf("
") + 1);
            if (childName.equals(name)) {
                children.add(nodeString);
            }
        }
        return children;
    }
    // Function to calculate g(n), which is
the distance between a parent node and a child
node
    private int calculateGn (Node parentNode,
String childNodeString) {
        int parentX =
```

```java
        int childX =
Integer.parseInt(childNodeString.split("
")[1]);
        int childY =
Integer.parseInt(childNodeString.split("
")[2]);
        return (int) (Math.abs((parentX
- childX)) + Math.abs((parentY -
childY)));
    }
    // Function to calculate heuristic
h(n), which is the Manhattan distance
between a node and the end node
    private int calculateHn (Node
startNode, String endNodeString) {
        int startX =
startNode.getCoordinate().getX();
        int startY =
startNode.getCoordinate().getY();
        int endX =
Integer.parseInt(endNodeString.split("
")[1]);
        int endY =
Integer.parseInt(endNodeString.split("
")[2]);
        return (int) (Math.pow((startX -
endX), 2) + Math.pow((startY - endY),
2));
    }
    // Function for optimal route
formation from the start node to the end
node using A* algorithm
    public List<String> AStar (String
startNodeString, String endNodeString) {
        // Initialize route
        List<String> route = new
ArrayList<>();
        // Initialize priority queue
based on f(n)
        Queue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(
Node -> (Node.getGn() + Node.getHn())));
        // Initialize set for visited
nodes
        Set<String> visitedNodeNames =
new HashSet<>();
        // Add start node to queue
        String name =
startNodeString.split(" ")[0];
        Coordinate coordinate = new
Coordinate(Integer.parseInt(startNodeStr
ing.split(" ")[1]),
Integer.parseInt(startNodeString.split("
")[2]));
        Node tempNode = new Node(name,
coordinate, 0, 0, null);
        queue.offer(new Node(name,
coordinate, 0, calculateHn(tempNode,
endNodeString), null));
```



```
        // Explore each node in the queue until
queue is empty
        while (!queue.isEmpty()) {
            // Choose the node with the lowest
cost from the queue
            Node currentNode = queue.poll();

visitedNodeNames.add(currentNode.getName());
            // If the chosen node is the end
node
            if
(currentNode.getName().equals(endNodeString.spl
it(" ")[0])) {
                // Return result
                route = formRoute(currentNode);
                return route;
            }
            // Check every children node of the
current node
            for (String childNodeString :
findChildren(currentNode.getName())) {
                // If the child node is
unvisited
                if
(!visitedNodeNames.contains(childNodeString.spl
it(" ")[0])) {
                    // Calculate g(n) and h(n)
of the child node, add to queue
                    name =
childNodeString.split(" ")[0];
                    coordinate = new
Coordinate(Integer.parseInt(childNodeString.spl
it(" ")[1]),
Integer.parseInt(childNodeString.split("
")[2]));
                    int gn =
currentNode.getGn() + calculateGn(currentNode,
childNodeString);
                    int hn =
calculateHn(currentNode, endNodeString);
                    queue.offer(new Node(name,
coordinate, gn, hn, currentNode));
                }
            }
        }
        return route;
    }
}
```

Snapshot. 3. Java class for implementing A* algorithm.
(Source: author's documentation)

IV. RESULT

In order to test the A* algorithm implementation thoroughly, three examples of start locations will be used. The program will then generate a route consisting of the nodes a player should take in order to travel the shortest path from the starting location to the nearest exfiltration site available on the map. The resulting optimal path generated by the program will then be examined and checked.

1. First Test

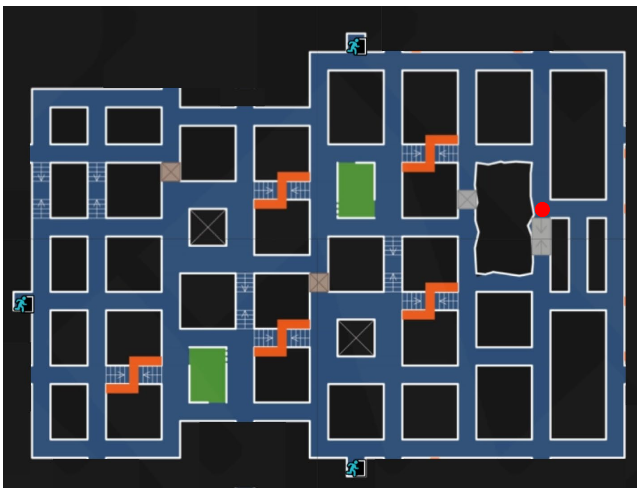


Fig. 7. Player's location for the first test indicated by a red dot.
(Source: Author's documentation)

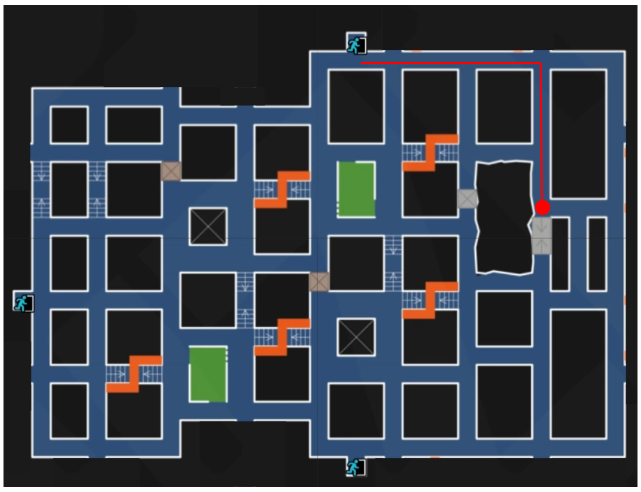


Fig. 8. Route generated by the program for the first test.
(Source: Author's documentation)

2. Second test

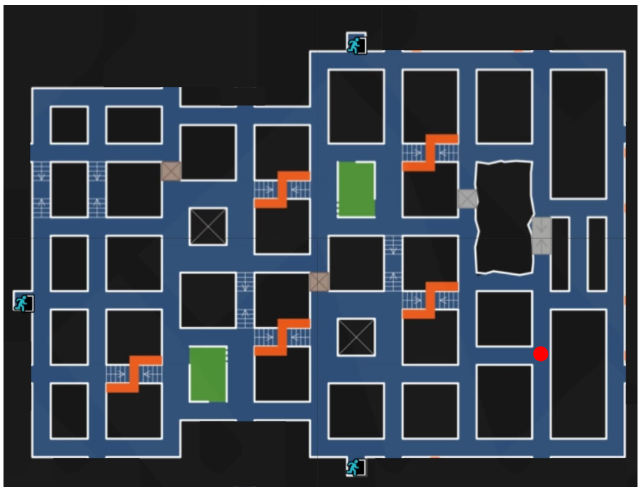


Fig. 9. Player's location for the second test indicated by a red dot.
(Source: Author's documentation)

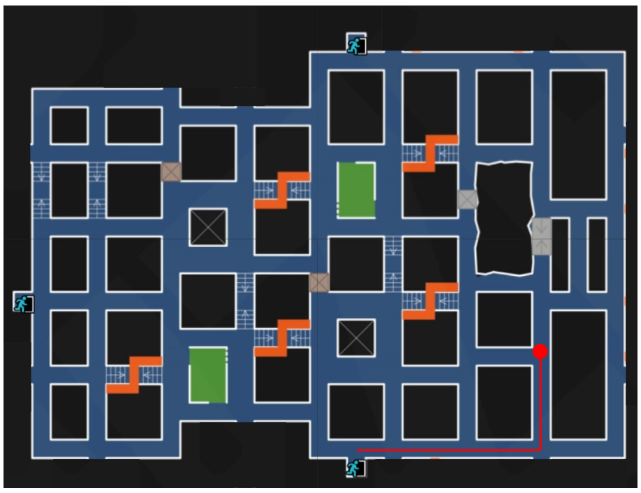


Fig. 10. Route generated by the program for the second test. (Source: Author's documentation)

3. Third Test

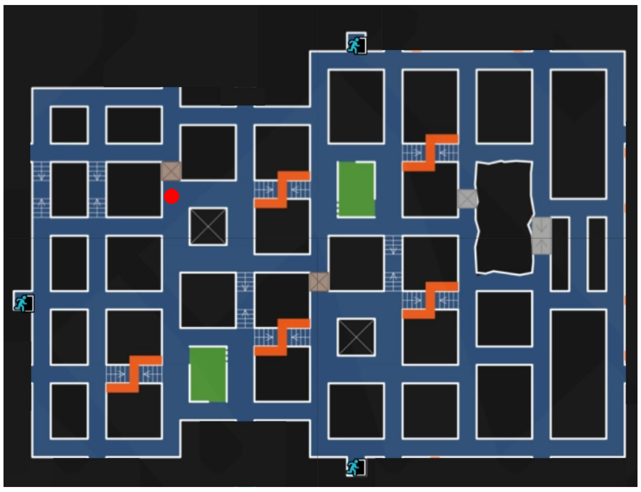


Fig. 10. Player's location for the third test indicated by a red dot. (Source: Author's documentation)

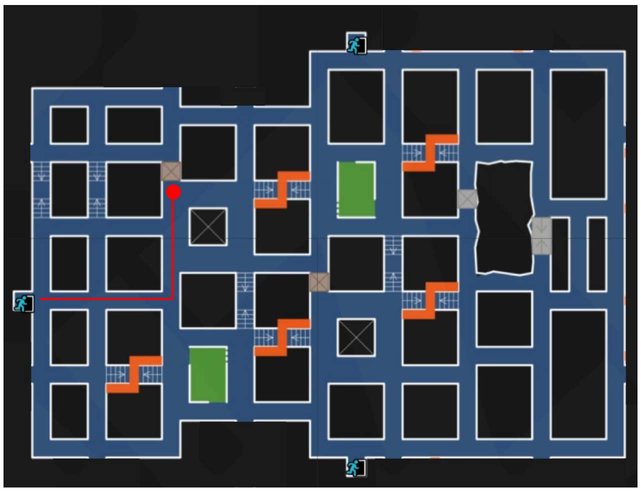


Fig. 12. Route generated by the program for the third test. (Source: Author's documentation)

V. CONCLUSION AND SUGGESTION

A. Conclusion

A* algorithm is a popular route-finding algorithm that is able to efficiently construct the shortest path between any two nodes on a graph and produce globally optimal solutions. It has many applications in various fields, including but not limited to route planning, puzzle games solving, and artificial intelligence.

In the context of route finding, one such example for the usage of A* algorithm for this particular field is finding the optimal exfiltration route on the Koschei Complex Map of DMZ Game Mode in Call of Duty: Warzone 2.0. The result shows that A* algorithm does form the optimal route for this game mode both efficiently and effectively. It succeeds in constructing the optimal path, that is the path with the shortest distance and therefore minimum time to complete

B. Suggestion

The code currently used in the program is still far from perfect. It only considers the distance between nodes on the map for its cost, while in reality, a player's survivability (chance of meeting enemy players and bots) is also related to existence of proper cover positions, openness, and other complex factor.

The current version of the A* algorithm implementation could certainly be improved and expanded to consider more factor and produce a more realistic optimum solution for the problem. For future developments, the author strongly suggests more trial and errors in order to find a more efficient and thorough heuristic and cost calculation.

ACKNOWLEDGMENT *(Heading 5)*

First and foremost, all praise and glory to Jesus Christ, our Lord and Savior, which gives the author strength and knowledge needed to finish this paper. The author would also give a big appreciation to his lecturer on this subject, Dr. Nur Ulfa Maulidevi, S.T, M.Sc., for giving this task on writing a paper related to the subject.

The author feels very grateful being given this task because it has helped him significantly in gaining a deeper understanding of how pathfinding algorithms, in particular the A* algorithm, works and how one can correctly design an implementation of the algorithm to solve specific problems.

The author would also want to thank his family and friends who supported him both academically and mentally through a lot of hard times in this semester.

REFERENCES

[1] Apa itu DMZ? Begini penjelasan detail mode baru Warzone 2.0 [Online]. https://www.oneesports.id/seputar-game/penjelasan-mode-warzone-2-0-dmz/. Accessed on June 11th 2024.

[2] Artificial intelligence: a modern approach. Norvig, Peter (4th ed.). Russell, Stuart J. (2018). Boston: Pearson.

[3] DMZ [Online]. https://callofduty.fandom.com/wiki/DMZ. Accessed on June 11th 2024.

[4] "Engineering Route Planning Algorithms". Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation. Lecture Notes in Computer Science. Delling, D.; Sanders, P.; Schultes, D.; Wagner, D. (2009).

[5] Intoduction to A* [Online]. http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. Accessed on June 12th 2024.

[6] Penentuan Rute (Route/Path Planning) Bagian 2 : Algoritma A* [Online]. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-202 1/Route-Planning-Bagian2-2021.pdf. Accessed on June 11th 2024.

STATEMENT OF ORIGINALITY

I hereby declare that the paper I have authored is an original work, and is not a copy nor a translation from another person's paper, nor is it plagiarized.

Bandung, July 12th, 2024

[illegible]

Christopher Brian
13522106