Simple Custom Route Planning Application for Organizing Room or City with Javafx and A Algorithm*

Abdullah Mubarak - 13522101
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522101@std.stei.itb.ac.id

***Abstract*—This paper presents a Simple Custom Route Planning Application designed for organizing rooms or cities using JavaFX and the A* algorithm. The application features an intuitive graphical user interface built with Scene Builder and managed through Maven, allowing users to draw paths, place obstacles, and compute optimal routes efficiently. Nodes within a grid pane can be interactively marked, connected, or erased, with paths dynamically adjusting in response to these changes. Users can also place various types of obstacles, such as furniture or buildings, which the A* algorithm navigates around to find the shortest path. (*Abstract*)**

Keywords—A* Algorithm; Route Planning; JavaFX; Path Finding; Organizing Room; Organizing City.

I. Introduction

Efficient navigation and spatial organization are crucial aspects in various domains, ranging from personal living spaces to urban planning. With the increasing complexity of environments and the necessity for optimized paths, developing effective pathfinding solutions has become an essential task. Although still lacking in many aspects, this paper presents a simple custom pathfinding application tailored for organizing rooms or city layouts, leveraging JavaFX for the graphical user interface and the A* algorithm for pathfinding.

The A* algorithm, known for its efficiency and accuracy in finding the shortest path in a weighted graph, serves as the core of my application. It combines the benefits of Dijkstra's algorithm and greedy best-first search, making it a popular choice for real-time pathfinding. With its heuristic function, A* can receive multiple variables to define the most suitable path. By integrating A* with JavaFX, a powerful and versatile Java library for building rich internet applications, we create a user-friendly platform that allows users to visualize and manage spatial arrangements effectively.

My application aims to provide an intuitive interface for users to define spaces, obstacles, and destinations, enabling automatic generation of optimal paths. This functionality can be applied to various scenarios, such as arranging furniture in a room, planning evacuation routes, or designing urban infrastructure.

II. Theory and Concepts

A. *JavaFX*

JavaFX is next generation client application platform build on Java. JavaFX is a powerful Java library for building cross-platform graphical user interfaces (GUIs) and rich internet applications (RIAs). It offers a suite of graphics and media tools that allow developers to create, test, debug, and deploy complex client applications that work seamlessly across various platforms.

JavaFX supports the best practice of building good graphical user interfaces (GUI), that is using *model-view-controller* to separate view or user interface (UI) design with controller. View contains visual/graphical attribute component of the interface whereas the controllers handle interaction and the action of each component of the interface.

JavaFX controllers are written in Java language whereas views are written in FXML - an XML-based markup language used to describe user interface. An integral part of the FXML format is a possibility of declaring an associated controller class and exposing to it UI elements, and event handler hooks. The controller is then responsible for reacting on the events and updating the view accordingly.

FXML can be edited in text editor called Scene Builder. Scene Builder enables to quickly design JavaFX application by dragging a UI component from its library and dropping it into a content view area. Scene Builder also provide quick setting for layout and properties such as padding, layout, margins.

B. *Route Planning*

Route planning is the process of computing the effective method of transportation or transfers through several stops. It uses a weighted graph to represent the stops (nodes), the path

(side), and the cost of the path (weight). There are several types of algorithms for route planning, including: Breadth-First Search (BFS), Depth-First Search (DFS), Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), A-Star (A*).

Breadth-First Search (BFS) is an algorithm that explores all neighbors' nodes at the present depth prior to moving on to nodes at the next depth level. It ensures that the shallowest node is expanded first, guaranteeing the shortest path in terms of the number of edges traversed. This method is typically used when finding the shortest path in an unweighted graph.

Depth-First Search (DFS) explores as far as possible along each branch before backtracking. It's often implemented recursively and is useful for traversing or searching tree or graph structures. While it doesn't necessarily find the shortest path, it's memory efficient and can be used for tasks like topological sorting, cycle detection, and maze generation.

Uniform Cost Search (UCS) is an algorithm that expands the least cost node, preferring cheaper paths. It's optimal for finding the shortest path in weighted graphs where edge costs can vary. UCS is similar to BFS but takes into account the cost of the path rather than just the number of edges.

Greedy Best-First Search (GBFS) selects the node which is closest to the goal based on a heuristic function. It's not guaranteed to find the shortest path but can be very efficient, especially in large search spaces. GBFS is suitable for problems where an approximate solution is acceptable or when the entire search space is too large to explore exhaustively.

A-Star (A*) is a variance of Dijkstra algorithm that combines the advantages of UCS and GBFS by considering both the cost of the path from the start node and the estimated cost to reach the goal node (heuristic function). It uses a heuristic function to guide the search towards the most promising nodes, resulting in an optimal path if certain conditions are met.

C. A Algorithm*

A* algorithm is one of the best paths finding (or in this case route planning) algorithm that can calculate the best path with certain conditions (can be maximum cost or minimum cost). Like other path finding algorithms, A* has evaluation functions to determine the best path. The evaluation function:

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

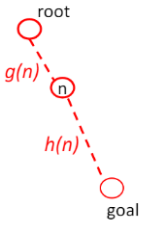

Fig. 1. The image of connection between f(n), g(n), h(n)

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf)

g(n) is the total cost that accumulates from the start node to the n node, whereas h(n) is the heuristic value that represents the estimated cost from current node to the goal node. It can be very different in each case.

One of the most frequently used heuristic function is the Manhattan distance. Manhattan distance is used to calculate the value of the absolute axis distances of two nodes in geometric metric space. The value of Manhattan distance in the 2-dimensional plane is the value between two points, calculating from the vertical axis and the distance on the horizontal axis and add these two distances.

The green line is the Euclidean distance, which is used to calculate the straight-line distance between two points. The red line calculates the Manhattan distance by adding two values from the vertical axis and the horizontal axis [1].

$$d(a, b) = |ax - bx| + |ay - by|$$

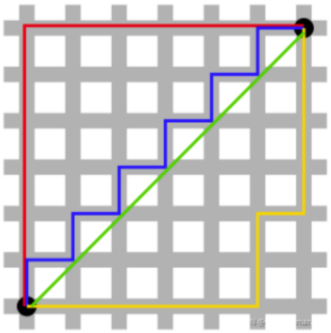

Fig. 2. The image of calculating the Manhattan Distance (Source: [1])

Different heuristic function can result in very different result. For a heuristic function to be considered admissible for the problem, it must have better or same cost than value of the g(n) (lower if minimum or higher if maximum).

Step-by-step implementation of A* algorithm:

1. **Initialize**:

   Create two lists: open_list (initially containing the start node) and closed_list (initially empty).

   Define the start node with *g(start)=0* and calculate f*(start)=g(start)+h(start).*

2. **Loop Until Goal is Reached**:

   Select the node with the lowest f-value from the open_list and set it as the current node. Move the current node from open_list to closed_list.

3. **Generate Successors**:

   For each neighbor of the current node: If the neighbor is in the closed_list, ignore it else calculate g and f for the neighbor.

If the neighbor is not in the open_list, add it and record its f-value.

If the neighbor is in the open_list but with a higher g-value, update its g-value and re-calculate f.

4. **Repeat**:

Continue the process of selecting nodes, calculating costs, and updating lists until the goal node is reached or the open_list is empty.

5. **Path Reconstruction**:

Once the goal is reached, reconstruct the path by tracing back from the goal node to the start node using recorded parent nodes.

III. Application Implementation

A. A Mapping*

1. Solution Space

Solution space of this application can be represented in a vector with n-tuple sized:

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

with $x_1, x_2, . \ . \ , x_n \in$ {UP, DOWN, RIGHT, LEFT}

or

with $x_1, x_2, . \ . \ , x_n \in \{\{1, 0\}, \{0, 1\}, \{-1, 0\}, \{0, -1\}\}$

This structured representation of the solution space helps to found the best paths from the start node to the goal node, considering all potential movements at each step.

2. Bounding Function

The bounding function in this application plays a crucial role in ensuring the validity of the solution space by limiting the xxx values so that they remain within the bounds of the grid. This function checks whether the coordinates resulting from any movement direction (UP, DOWN, RIGHT, LEFT) stay within the grid's predefined limits. This prevents the algorithm from considering invalid moves that would lead to positions outside the grid, which could result in errors or infinite loops.

In addition to ensuring the movements stay within the grid, the bounding function also checks the walkability of the new position. This means that the function not only ensures the new position is within bounds but also that it is a traversable node (i.e., not an obstacle). By combining these checks, the bounding function helps in maintaining the feasibility of the generated path, ensuring that all considered paths are within the valid and walkable regions of the grid. This mechanism is essential for the proper functioning of the A* algorithm, as it systematically prunes infeasible paths and focuses computational resources on exploring valid and promising routes.

3. Node

The code defines a Node class representing each point on the grid, with attributes for coordinates, cost from the start node (*g*), heuristic cost to the goal (*h*), and the parent node for path reconstruction. The Node class includes methods for calculating the total cost (*getF()*), comparison for priority queue ordering, and equality checks.

Here is the code implementation for class Node:

```
class Node implements Comparable<Node> {

    public int row, col;

    public int g, h;

    public Node parent;

    public Node(int row, int col) {

        this.row = row;

        this.col = col;

    }

    public int getF() {

        return g + h;

    }

    @Override

    public int compareTo(Node other) {

        return Integer.compare(this.getF(),
other.getF());

    }

    @Override

    public boolean equals(Object obj) {

        if (this == obj) return true;

        if (obj == null || getClass() !=
obj.getClass()) return false;

        Node node = (Node) obj;

        return row == node.row && col == node.col;

    }

    @Override

    public int hashCode() {

        return Objects.hash(row, col);

    }

}
```

4. Generating Function

The generating function in the A* algorithm is responsible for producing successor nodes from the current node based on the possible movement directions. For each node being evaluated, the function considers all potential movements (UP, DOWN, RIGHT, LEFT) by adding the corresponding direction vectors to the current node's coordinates. It then checks each new coordinate using the bounding function to ensure it is within the grid's limits and walkable. If the new coordinate is valid, a new successor node is created with updated g-values (cost from the start node to this successor node), and heuristic h-values (estimated cost from this successor node to the goal). These successor nodes are then added to the open set for further evaluation.

5. Total Cost

The total cost function in the A* algorithm is the key to determining the most efficient path from the start node to the goal node. It is computed using the evaluation function:

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

where *g(n)* is the cumulative cost from the start node to the current node *n*, and *h(n)*is the heuristic estimate of the cost from *n* to the goal.

he *g(n)* value represents the actual cost incurred to reach node *n*, considering the sum of all the movements taken so far. The heuristic *h(n)* often implemented as the Manhattan distance in grid-based pathfinding, estimates the remaining cost to reach the goal, guiding the algorithm by prioritizing nodes that seem closer to the goal. The total cost *f(n)* thus combines the known cost with the heuristic estimate, allowing A* to balance between exploring the shortest known path and the most promising routes based on the heuristic. This approach ensures that A* finds an optimal path efficiently, minimizing the total traversal cost from start to goal.

B. A Implementation*

In this application the AStar class encapsulates the algorithm, maintaining the grid's dimensions and walkability. It defines directions for possible movements (up, down, left, right). The findPath method implements the A* algorithm, using a priority queue (openSet) to explore nodes with the lowest total cost first. It also uses a set (closedSet) to keep track of visited nodes.

The algorithm begins by initializing the start node's costs and adding it to the open set. It then enters a loop where it processes the current node (the one with the lowest f value), checks if it's the goal, and explores its neighbors. For each neighbor, the algorithm calculates a tentative g cost and updates the neighbor's costs and parent if the new path is better. The heuristic used is the Manhattan distance, suitable for grid-based pathfinding.

If the goal node is reached, the path is reconstructed by tracing back from the goal to the start node using parent pointers. If no path is found, the method returns an empty list. This straightforward implementation demonstrates the essential steps of the A* algorithm, ensuring efficient and optimal pathfinding in a grid environment.

Here is the code implementation for class A*:

```
private static final int[][] DIRECTIONS = {{1, 0}, 
{0, 1}, {-1, 0}, {0, -1}};

private final int rows, cols;

private final List<List<Boolean>> walkable;

public AStar(int rows, int cols, List<List<Boolean>> 
walkable) {

    this.rows = rows;

    this.cols = cols;

    this.walkable = walkable;

}

public List<Node> findPath(Node start, Node goal) {

    PriorityQueue<Node> openSet = new 
PriorityQueue<>();

    Set<Node> closedSet = new HashSet<>();

    start.g = 0;

    start.h = heuristic(start, goal);

    openSet.add(start);

    while (!openSet.isEmpty()) {

        Node current = openSet.poll();

        if (current.equals(goal)) {

            return reconstructPath(current);

        }

        closedSet.add(current);

        for (int[] direction : DIRECTIONS) {

            int newRow = current.row + direction[0];

            int newCol = current.col + direction[1];

            if (isValid(newRow, newCol) && 
walkable.get(newRow).get(newCol)) {

                Node neighbor = new Node(newRow, 
newCol);

                if (closedSet.contains(neighbor)) 
continue;
```

```
                int tentativeG = current.g + 1;

                if (tentativeG < neighbor.g ||
!openSet.contains(neighbor)) {

                    neighbor.g = tentativeG;

                    neighbor.h = heuristic(neighbor,
goal);

                    neighbor.parent = current;

                    if (!openSet.contains(neighbor))
{

                       openSet.add(neighbor);

                    }

                }

            }

        }

    }

    return Collections.emptyList(); // No path found

}



private int heuristic(Node a, Node b) {

   return Math.abs(a.row - b.row) + Math.abs(a.col -
b.col);

}

private boolean isValid(int row, int col) {

   return row >= 0 && col >= 0 && row < rows && col
< cols;

}

private List<Node> reconstructPath(Node node) {

    List<Node> path = new ArrayList<>();

    while (node != null) {

        path.add(node);

        node = node.parent;

    }

    Collections.reverse(path);

    return path;

}
```

C. Application Implementation

In this JavaFX application, Scene Builder plays a pivotal role in streamlining the graphical user interface (GUI) development process. Author use Scene Builder's intuitive drag-and-drop interface to ease designs and constructs the application's GUI elements. The grid pane attribute serves as the primary canvas for visualizing the main area, with each node represented by a distinctive blue color.

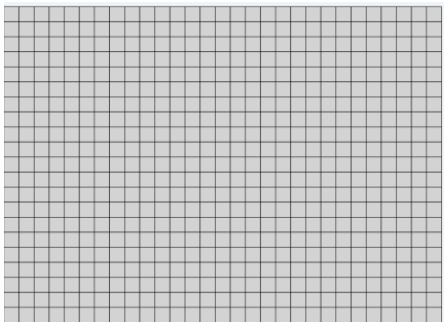

Fig. 3. The image of grid pane
(Source: author documentation)

There are a few features implemented:

1. Draw path

One of the notable features of the application is the ability for users to draw paths by marking nodes within the grid. This interactive feature enables users to define routes or sequences of waypoints by simply clicking on the grid cells. The flexibility of this functionality allows for the creation of multiple sets of nodes, with each set delineated by a separate draw actio. Additionally, users have the convenience of erasing nodes by clicking on them again, facilitating iterative refinement of route plans.

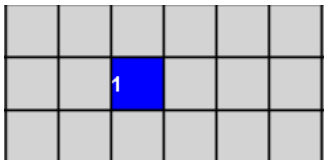

Fig. 4. The image of grid that have been clicked
(Source: author documentation)

2. Place Obstacle

Another significant feature of the application is the capability to place obstacles within the environment, simulating real-life elements such as furniture in a room or buildings in a city layout. These obstacles are categorized into two types: manual and shape-based. The shape-based obstacles, just circles (for now), offer user the flexibility to customize dimensions such as radius and can be added to grid simply by click on the wanted posotion.

Furthermore, users can opt for manual placement of obstacles, providing precise control over obstacle placement by selecting specific grid cells. The visual representation of obstacles, distinguished by darker shades of grey within the grid, facilitates clear identification and manipulation. Additionally, user can remove obstacles by clicking on them.

D. *Further Reference*

Further reference about the application are presented in the author's repository on github:

https://github.com/b33rk/MakalahStima

IV. Integration Testing

1. Draw Path Without Obstacle

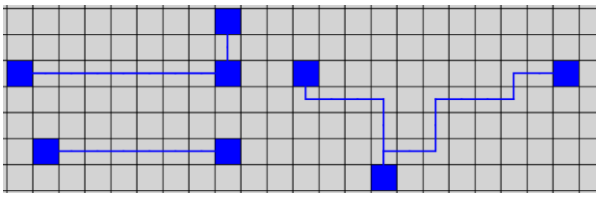

Fig. 6. Path without Obstacle
(Source: author documentation)

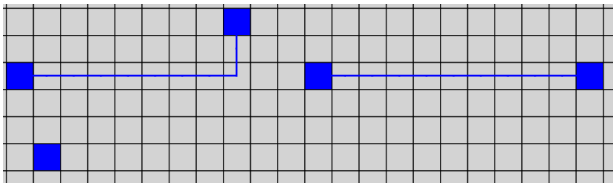

Fig. 7. Path fig. 6 after removing one node
(Source: author documentation)

As can be seen in two images above, the application can do one of the intended feature (draw path). The auto connect feature when node is removed also successfully implemented.

2. Place Obstacle

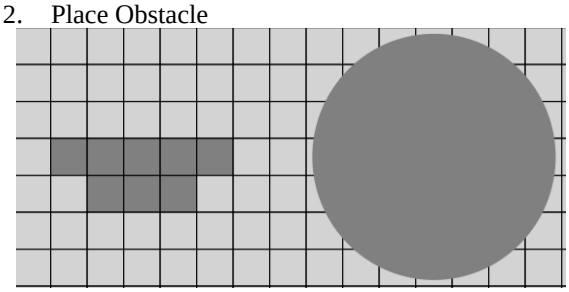

Fig. 8. The image of obstacles
(Source: author documentation)

The application also successfully implements feature that enables users to place obstacles effectively within the grid. Obstacles can represent various real-life elements such as furniture in a room or buildings in a city layout. Users can choose between manual placement and shape-based obstacles. Shape-based obstacles, such as circles and squares, can be customized in size and position, enhancing the application's versatility. Manual obstacles allow for precise placement by clicking on specific grid cells, with darker grey shades indicating the presence of obstacles. This feature provides a clear visual distinction between traversable and non-traversable areas, aiding users in realistic scenario planning and route adjustments.

3. Draw Path with Obstacles

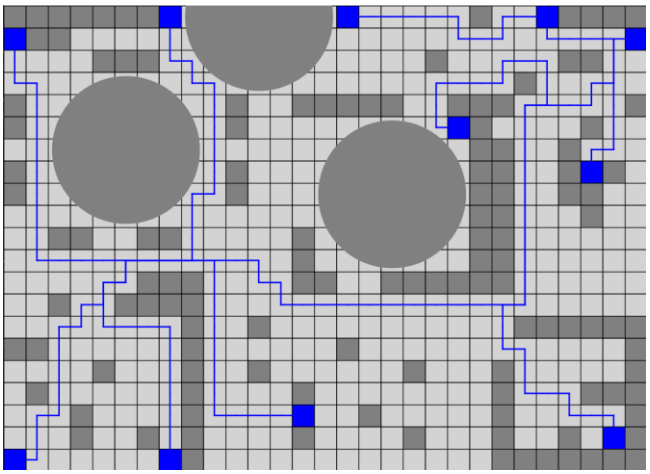

Fig. 9. The image of obstacles + path finding
(Source: author documentation)

As illustrated in the image above, the A* algorithm can compute the shortest paths that navigate around placed obstacles. This feature is critical for real-world applications where paths must be planned around fixed structures or impassable areas. The application dynamically adjusts the computed route to find the optimal path that avoids obstacles, ensuring that the path remains efficient.

V. Conclusion and Suggestion

A. *Conclusion*

In this paper, author presented a simple custom route planning application for organizing rooms or cities, leveraging JavaFX and the A* algorithm. The application provides a user-friendly interface for drawing paths, placing obstacles, and finding optimal routes. Utilizing Scene Builder and Maven, the application's design and functionality were streamlined, allowing users to interactively plan and adjust routes in real-time. The integration of features such as automatic node connection, dynamic obstacle placement, and efficient pathfinding demonstrates the application's functionality. Through comprehensive testing, the application has shown to handle various scenarios effectively, showing possibility to be used for both room organization and city planning purposes.

B. *Suggestion*

Author realises that this application still lacking in many aspects to be used in real work and implementation. But, author believe, with proper enhancement, this application can be more useful and play a bigger role. Future enhancements that author suggest focussing on expanding include:

1. **Enhanced Obstacles**: Introduce more complex obstacle shapes and types, including irregular polygons and dynamic obstacles that can change position over time.

2. **Multiple Floors/Levels**: Extend the application to support multi-floor buildings or multi-level city

structures, allowing users to plan routes that navigate across different levels.

3. **Integration with Real-World Data**: Incorporate real-world data such as traffic information, pedestrian flow, or room occupancy to provide more accurate and realistic route planning.

4. **User Customization**: Enable users to save and load custom layouts, allowing them to reuse and share their configurations easily.

5. **Improved Heuristics**: Experiment with different heuristic functions to enhance the efficiency and accuracy of the A* algorithm, especially for larger and more complex grids.

6. **Mobile Compatibility**: Develop a mobile version of the application to increase accessibility and usability in various environments, making it convenient for on-the-go planning.

7. **Collaboration Features**: Add features that allow multiple users to collaborate in real-time, facilitating team-based planning and decision-making processes.

By implementing these suggestions, the application can become even more versatile and valuable, catering to a wider range of use cases and user needs.

ACKNOWLEDGMENT

The first and foremost, I serve my gratitude to the Almighty god, that through Him I can write this paper and build this application. Secondly, thanks to my parents that always support me throughout my life, especially during learning in Institut Teknologi Bandung. And lastly, to my teacher, Mr. Rinaldi Munir, thank you for all your teaching and guidance, I appreciate it a lot.

REFERENCES

[1] Yan, Yumeng, "Research on the A Star Algorithm for Finding Shortest Path,". Beijing: Beijing-Dublin International College, 2023, pp. 157.

[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf. Accessed on 11 May 2024

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Juni 2024

Abdullah Mubarak

Abdullah Mubarak
13522101