

*Green University of Bangladesh*

#### Department of Computer Science and Engineering (CSE) Semester: (Spring, Year: 2025), B.Sc. in CSE (Day)

**Path Finding and Navigation System**

#### Course Title: Algorithm Lab Course Code: 232 D9

#### Section: CSE 206

### Students Details

|  |  |
| --- | --- |
| **Name** | **ID** |
| Rukonuzzaman Topu | 232002280 |

#### Submission Date: 30.04.2025

#### Course Teacher’s Name: Farjana Akter Jui

### [For teachers use only: Don’t write anything inside this box]

**Signature:**

**Date:**

**Marks:**

**Comments:**

**Lab Project Status**

**Contents**

1. [**Introduction**](#_3bwgu63zw3i3) **3**
   1. [Overview](#_vhrbgemsn1cm) 3
   2. [Motivation](#_7u6x6ida673u) 3
   3. [Problem Definition](#_t7kloktsctf) 3
      1. [Problem Statement](#_4r2anmuca0s8) 3
   4. [Design Goals/Objectives](#_okhw6d97mphx) 4
   5. [Application](#_nhd19fjcznos) 4
2. [**Design/Development/Implementation of the Project**](#_g5s4tsu1fkrm) **5**
   1. [Introduction](#_913j12p26fqn) 5
   2. [Project Details](#_nl0bwkjs5qqt) 5
      1. [Key Features](#_7o64if4alua1) 5
      2. [Components](#_xseukyrznnqd) 6
      3. [Functionality Breakdown](#_gx9x2alw0uki) 6
   3. [Implementation](#_efyos1vhibzs) 6
      1. [The workflow](#_ge0hc3brfew3) 6
      2. [Tools and Libraries](#_wwx0udad5nod) 7
   4. [Algorithms](#_b800qtjeadwo) 8
3. [**Performance Evaluation**](#_pc2jmid1gci5) **9**
   1. [Results Analysis/Testing](#_gezksmsefgku) 9
      1. [Result\_portion\_1](#_kd4d3ebevpx9) 9
4. [**Conclusion**](#_gr655m4t2ybm) **11**
   1. [Discussion](#_gtsqoiz696gd) 11
   2. [Limitations](#_onptxkojnjs0) 11
   3. [Scope of Future Work](#_wkqkfqk5yjm4) 12
5. [**Reference**](#_bvmzchsz3ksc) **13**

**Chapter 1 Introduction**

# Overview

This project implements a path-finding system that determines the shortest route between two points on a map by leveraging four classic graph-search algorithms—Dijkstra’s algorithm, A\* search, breadth-first search (BFS), and depth-first search (DFS). It begins by constructing a graph-based representation of the map, then develops efficient, modular code for each algorithm to explore and evaluate possible paths. Finally, these algorithms are integrated into an interactive user interface that visualizes each search in real time, allowing users to observe how walls and open spaces influence the search process and to compare the performance and behavior of the different methods.

# Motivation

I picked this project to broaden my knowledge of graph algorithms and how they are used in practical settings like navigational systems. I can explore complex problem- solving while improving my coding skills by creating a pathfinding tool. The project’s significance lies in its capacity to develop effective routing solutions, which could have an impact on GPS technology, gaming, logistics, and other industries.

# Problem Definition

## Problem Statement

This project develops a robust path-finding and navigation system that identifies the most efficient routes between any two points on a map by leveraging four classic graph-search algorithms—Dijkstra’s algorithm, A\* search, breadth-first search (BFS), and depth-first search (DFS). By representing the environment as a weighted graph, the system addresses real-world navigation challenges across industries such as gaming, logistics, and transportation. The primary objective is to minimize travel time and resource consumption while providing a flexible framework for comparing algorithm performance, handling dynamic obstacles, and enhancing the user’s overall navigational experience.

# Design Goals/Objectives

My projects objectives are given below:

* Effective Pathfinding: Create a system that can determine the shortest path be- tween any two points on a map in a timely manner .
* User-Friendly Interface: Provide a user-friendly interface so that users can inter- act with the system, enter beginning and ending locations, see routes, and com- prehend how to navigate.
* Algorithmic Accuracy: Verify that the algorithm computes optimal routes accu- rately, taking into account edge cases, obstacles, or map modifications without sacrificing efficiency.
* Testing and Validation: To confirm the system’s precision, effectiveness, and de- pendability in identifying the best routes, carry out thorough testing across a range of map configurations and scenarios.

# Application

This pathfinding system has a wide range of real-world uses. It can optimize delivery and shipping by finding the fastest truck or van routes; power GPS and mobile navigation to guide drivers and pedestrians turn by turn; drive video games and virtual worlds by moving characters or objects along smart paths; improve computer networks by routing data packets efficiently; enable robots and self-driving vehicles to navigate factory floors or roads; assist urban planners and emergency services in mapping out the best roads for new developments, ambulances, or fire trucks; and even support hospital wayfinding, helping patients and staff find clinics, wards, or labs indoors.

**Chapter 2**

**Design/Development/Implementation of the Project**

# Introduction

This Java console application displays a map as a grid of characters and lets you find the shortest path between two points. When you run it, you:

1. **Enter grid size** (rows and columns)
2. **Set obstacle density** (a number from 0 to 1 to control how many walls # appear)
3. **Choose an algorithm** (1 = Dijkstra, 2 = A\*, 3 = BFS, or 4 = DFS)
4. **Provide start and finish coordinates** on the grid.

The program builds the grid with walls (#) and empty spaces (.), then prints it. As the chosen algorithm runs, it marks each visited cell with +. Once the finish is reached, it traces back the best route, marking that path with \*, and prints the final grid. This simple text interface makes it easy to see exactly how each algorithm explores the grid and finds the shortest path.

# Project Details

This Java console application provides a simple text-based interface for exploring pathfinding algorithms. When you run it, you’re prompted to enter the grid dimensions and an obstacle density (a value between 0 and 1), then choose between Dijkstra, A\*, BFS, or DFS. After you supply the start and finish coordinates, the program builds a two-dimensional array of node objects—marking walls as # or open spaces as .—and prints the initial grid. As the selected algorithm runs, each visited cell turns into +, and once the finish is reached the optimal route is traced back with \*. Finally, the complete map with walls, visited cells, and the shortest path is printed to the console, giving you a clear, step-by-step view of how each method explores and solves the grid.

## Key Features

**Algorithm Selection via Console** Pick 1 for Dijkstra, 2 for A\*, 3 for BFS, or 4 for DFS when prompted.

**Grid Setup Inputs** Enter the number of rows and columns, then a decimal between 0 and 1 for obstacle density to build a random map of # (walls) and . (open spaces).

**Start/Finish Coordinates** Type in the row/column for S (start) and F (finish) when asked.

**Textual Visualization**

* Initial Grid is printed with # and .
* As the search runs, visited cells turn into +
* Once the path is found, it’s marked with \*
* Finally the grid shows S, F, all +, and the \* path

**Scenario Re-runs** To try different maps or densities, simply rerun the program with new inputs.

## Components

**PathFindingConsole** – The main class that drives the program.  
 – Reads user input (grid size, obstacle density, algorithm choice, start/finish coords).  
 – Calls methods to build the grid, run the chosen search, and print both the initial and final maps.

**Node** – A nested static class representing each cell on the grid.  
 – Holds x, y coordinates, a type (S, F, #, ., +, or \*), cost fields (gCost, hCost), a parent pointer, and a visited flag.

**Algorithm** – A nested static class with four static methods: dijkstra, aStar, bfs, and dfs.  
 – Includes helpers for generating neighbors, computing the Manhattan heuristic, marking visited cells, and reconstructing the final path.

**Utility Methods (in PathFindingConsole)** – initializeGrid(double density): fills the grid with walls or empty spaces.  
 – placeEndpoints(Scanner): reads and sets the start (S) and finish (F) nodes.  
 – printGrid(Node[][]): prints the current grid to the console.  
 – markPath(List<Node>): back‐tracks from finish to start marking \* for the final route.

## Functionality Breakdown

* + - * **Initialization** Read user inputs for grid size, obstacle density, algorithm choice, and start/finish coordinates.
      * **Grid Creation** Fill a 2D Node[][] array with walls (#) or open spaces (.) according to the density value.
      * **Endpoint Placement** Place the start node (S) and finish node (F) at the user-specified coordinates.
      * **Algorithm Execution** Run the chosen search (Dijkstra, A\*, BFS, or DFS). As it explores, mark visited cells as +.
      * **Textual Visualization** Print the initial grid, then show + for every visited cell, and finally trace the shortest path with \* before re-printing the completed grid.

# Implementation

## The workflow

**Initialization & Input**

* The program starts in main, prompting the user (via Scanner) to enter grid size (rows, cols), obstacle density (0–1), algorithm choice (1 = Dijkstra, 2 = A\*, 3 = BFS, 4 = DFS), and start/finish coordinates.

**Grid Creation**

* initializeGrid(density) builds a 2D Node[][] array. Each cell becomes a wall (#) or empty space (.) based on the random check against the density value.

**Endpoint Placement**

* placeEndpoints(sc) reads the start and finish positions and marks them as S and F in the grid.

**Initial Display**

* printGrid(grid) prints the starting map of # and . (plus the S/F markers) to the console.

**Algorithm Execution**

* The chosen method (Algorithm.dijkstra, .aStar, .bfs, or .dfs) runs on the grid. Each visited cell’s type is updated to + in memory.

**Path Reconstruction**

* Once the finish node is reached, reconstructPath(...) backtracks via each node’s parent pointer, and markPath(path) marks the shortest route with \*.

**Final Display**

* A second call to printGrid(grid) outputs the complete map showing walls (#), visited cells (+), the final path (\*), and the S/F endpoints.

## Tools and Libraries

**Java Standard Library**

* java.util.Scanner for reading user inputs (grid size, density, algorithm choice, coordinates).
* java.util.Random to generate walls based on the obstacle density.
* System.out.print / System.out.println for all grid and status output to the console.

**Java Collections & Utilities**

* PriorityQueue (with Comparator) for Dijkstra and A\* open sets.
* LinkedList as a Queue for BFS.
* Stack for DFS.
* ArrayList / List for building paths and neighbor lists.
* Collections for path reversal.



Figure 2.1: Java Language

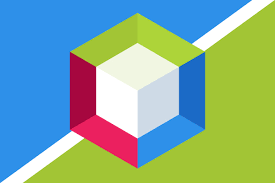


Figure 2.2: Netbeans

# Algorithms

**Algorithm 1:** Sample Algorithm

**Input:** Your Input **Output:** Your output **Data:** Testing set *x*

**1** function Dijkstra(Graph, source): dist[source] := 0 for each vertex v in Graph: if v source dist[v] := infinity add v to Q

**2** while Q is not empty: // The main loop v := vertex in Q with min dist[v] remove v from Q

**3** for each neighbor u of v: // where neighbor u has not yet been removed from Q. alt := dist[v] + length(v, u) if alt < dist[u]: // A shorter path to u has been found dist[u] := alt // Update distance of u

**4** return dist[] end function

##### Implementation details (with screenshots and programming codes)

****

Figure 2.3: Implementation Dijkstra Algorithum



Figure 2.4: Implementation-A \*



Figure 2.4:

Figure 2.5: Implementation-BFS

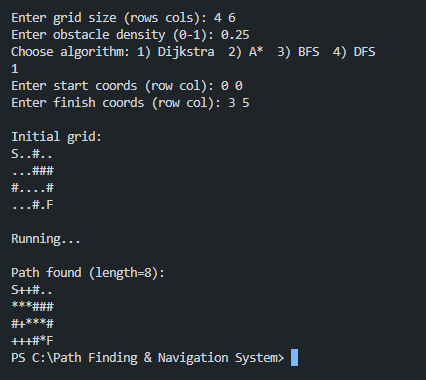
Figure 2.6: Implementation-DFS

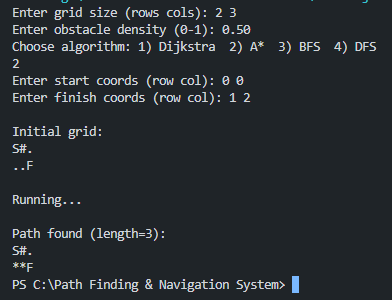
Chapter 3

Performance Evaluation

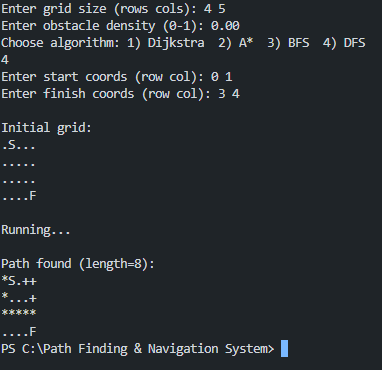
# Results Analysis/Testing

## Result\_portion\_1

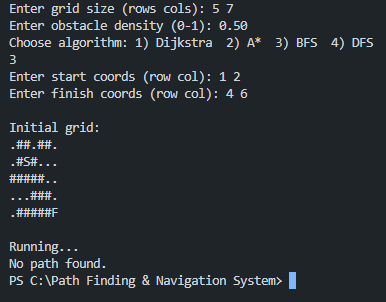
****

** Figure 3.1: Output 1**

**Figure 3.2: Output 2**



**Figure 3.3: Output 3**



**Figure 3.4: Output 4**

**Chapter 4 Conclusion**

# Discussion

This console-based Java program gives a clear, text-only demonstration of how four pathfinding algorithms—Dijkstra, A\*, BFS, and DFS—navigate a grid. After you enter the grid size, obstacle density, algorithm choice, and start/finish coordinates, the program prints an initial map of walls (#) and open spaces (.). As the chosen search runs, each visited cell changes to +, and when the finish node is reached the optimal route is backtracked and marked with \* before the final map is displayed. By comparing before-and-after grids, you can directly see how each method explores and solves the maze of obstacles.

In practice, A\* and Dijkstra always find the shortest path, but A\*’s heuristic often lets it visit fewer cells and finish more quickly on larger or sparser grids. BFS also guarantees an optimal route on an unweighted grid, although it may flood large open areas level by level and mark many more cells than A\*. DFS can sometimes reach the finish fastest on very sparse maps, yet it doesn’t ensure the shortest route and can even miss a path entirely if the grid is too dense. As obstacle density increases, all algorithms must visit more cells and may fail entirely when walls completely block the way—densities above roughly 0.6 often produce no path. Likewise, growing the grid dimensions multiplies the search space: even with low density, very large maps generate a flood of + marks before a path appears. This simple character-based visualization, though less flashy than a GUI, offers hands-on insight into each algorithm’s behavior, performance trade-offs, and sensitivity to map complexity.

# Limitations

This console-based system provides a straightforward, text-only view of how Dijkstra, A\*, BFS, and DFS operate on a uniform grid, but it has several constraints. First, the grid model treats every move as equal cost and allows only orthogonal (up/down/left/right) steps, so it cannot represent varied terrain costs or diagonal movement. Second, obstacles are static and randomly generated at startup—there’s no support for dynamic barriers that appear or disappear during a search. Third, the ASCII visualization (printing #, ., +, and \*) offers limited insight into the algorithms’ decision-making; you can’t pause, step backward, or inspect node costs and heuristics in real time. Finally, because every scenario requires restarting the program and re-entering parameters, it’s not well suited for rapid experimentation or large-scale performance testing. Extending the system to support weighted grids, dynamic updates, richer visual or interactive controls, and built-in logging of metrics would broaden its practical and educational value.

# Scope of Future Work

In the future, I want to make this pathfinding tool even more useful and easy to learn. First, I will add more ways to find a path—like Breadth-First Search and Depth-First Search—along with smarter versions of A\* (for example, A\* that uses less memory). I’ll also run simple tests that measure how fast each method is, how much memory it uses, and how good the path is. That way, anyone can see which method works best for different sized maps and obstacle amounts.

Next, I will let the grid show different kinds of ground—like grass, water, or hills—each with its own “cost” to move through. This will help users see how Dijkstra’s and A\* handle tougher routes. I’ll also add obstacles that can move or disappear while the search is running, so the program can show how to re-plan a route on the fly, just like a robot or a game character would need.

To make it easier to follow what’s happening, I will build a simple animation mode that highlights each step: which square is being checked, how the queue or stack changes, and how the final path is traced back. I’ll add sliders or fields where users can tweak settings—like how much weight the A\* heuristic uses—and instantly watch how those changes affect the search.

Finally, I plan to support much bigger maps and even real-world map files, so people can test on real city streets or hiking trails. I’ll wrap all this in easy-to-read tutorials with clear examples and include quick quizzes that ask questions like “Why did the algorithm pick this path?” Together, these updates will turn a simple demo into a hands-on learning tool and a flexible way to try out pathfinding ideas in real scenario

**Chapter 5 Reference**

<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/> <https://www.programiz.com/dsa/dijkstra-algorithm> [https://www.simplilearn.com/tutorials/artificial-intelligence-tuto](https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm#%3A~%3Atext%3DThe%20A%2A%20algorithm%20is%20widely%2Cobstacles%20and%20find%20optimal%20paths)rial/

[a-star-algorithm#:~:text=The%20A\*%20algorithm%20is%20widely,](https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm#%3A~%3Atext%3DThe%20A%2A%20algorithm%20is%20widely%2Cobstacles%20and%20find%20optimal%20paths)obstacles% [20and%20find%20optimal%20paths.](https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm#%3A~%3Atext%3DThe%20A%2A%20algorithm%20is%20widely%2Cobstacles%20and%20find%20optimal%20paths)

<https://www.geeksforgeeks.org/a-search-algorithm/>