Student Performance Prediction System

A PROJECT REPORT

Submitted to

Dr Javed Alam.

Submitted by

Arjun Rathi (24MCI10258)

in partial fulfillment for the award of the degree of

MASTER OF COMPUTER APPLICATION

IN

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Chandigarh University

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to **Dr Javed Alam** , my project supervisor, for his invaluable guidance, support, and encouragement throughout the duration of this project. His expertise and insights were instrumental in the successful completion of the project, *"Student Performance Prediction System."*

I am also thankful to **Chandigarh University** for providing the resources and a conducive environment that enabled me to explore and apply algorithmic principles in solving real-world problems. This project has been a significant learning experience, enhancing both my theoretical knowledge and practical skills.

TABLE OF CONTENTS

ABSTRACT

This project aims to create a system that uses machine learning techniques to predict the final exam scores of students. By analyzing academic data like attendance rates, assignment grades, quiz scores, and midterm results, the system forecasts the final performance using a linear regression model. This tool is designed to support both students and educators by providing early indications of academic performance, offering tailored study suggestions based on predicted outcomes, and generating downloadable pdf report cards. The online platform provides a user-friendly interface that facilitates seamless interaction and visualization of the outcomes, thereby improving the overall user experience.

\

ABBREVIATIONS AND SYMBOLS

ML – Machine Learning

MAE – Mean Absolute Error

R^2 – Coefficient of Determination

CSV – Comma-Separated Values

PDF – Portable Document Format

UID – Unique Identifier

GUI – Graphical User Interface

HTML/CSS – Frontend Technologies

Flask – Python Web Framework

Linear Regression – A type of statistical model that predicts a dependent variable based on one or more independent variables.

CHAPTER 1.
INTRODUCTION

1.1 Identification of Client & Need

The client for this project can be conceptualized as urban planners, transportation departments, or network engineers who require optimized pathfinding between multiple locations. The need arises from the challenge of determining the most efficient route in a network of connected nodes (cities, stations, routers), aiming to reduce time, cost, and resource consumption.

1.2 Relevant Contemporary Issues

- Increasing urban traffic and congestion.
- Complex logistics and delivery operations.
- Real-time network optimization for smart cities and IoT-based systems.
- Demand for intelligent route planning in GPS, transportation apps, and delivery services.

1.3 Problem Identification

Current systems often rely on GPS-based navigation which may not always use the most efficient algorithm. There is a lack of flexibility to simulate and test different AI algorithms under the same framework. The problem lies in evaluating which search algorithm—BFS, DFS, A*, or Hill Climbing—offers the best balance of accuracy and efficiency for route planning tasks..

1.4 Task Identification

- Create a graph-based model representing city-to-city connections.
- Allow users to input custom node names and distances.
- Implement four algorithms (BFS, DFS, A*, Hill Climbing) for pathfinding.
- Compare and visualize their performance.
- Determine and highlight the shortest path between the selected cities.

CHAPTER 2 :
LITERATURE SURVEY

2.1 Introduction to educational performance prediction models.

Route optimization involves finding the best path through a graph of nodes. It is essential in logistics, networking, transportation, and game development. The focus is on **shortest path determination** and **heuristic search**.

2.2 Traditional predictive models in education.

- **Dijkstra's Algorithm**: Guaranteed shortest path but computationally expensive.
- **Bellman-Ford**: Handles negative weights but slower. These were precursors to more efficient AI-based techniques used today

2.3 Applying Machine Learning Algorithms to Predictive Analytics

- **Breadth-First Search (BFS)**: Explores all neighbors level by level; guarantees shortest path in unweighted graphs.
- **Depth-First Search (DFS)**: Explores deep paths first; not optimal for shortest path.
- **A***: Combines path cost and heuristic estimate; highly accurate and efficient.
- **Hill Climbing**: Greedy algorithm; may get stuck in local minima. These algorithms are commonly studied in **AI and robotics** and form the foundation for intelligent decision-making systems.

2.4 Implementation of corrective actions.

- Visualizing paths enables error detection (e.g., loops, inefficient routes).
- Algorithm choice affects decision-making in real-time systems.
- Based on testing, fallback to A* when other algorithms fail or become inefficient.

CHAPTER 3:
DESIGN FLOW/ PROCESS

3.1 Idea creation.

The system was conceptualized as a simulator that mimics city networks with the ability to test multiple algorithms. Inspiration came from transportation planning apps and network simulators.

3.2. Feature selection.

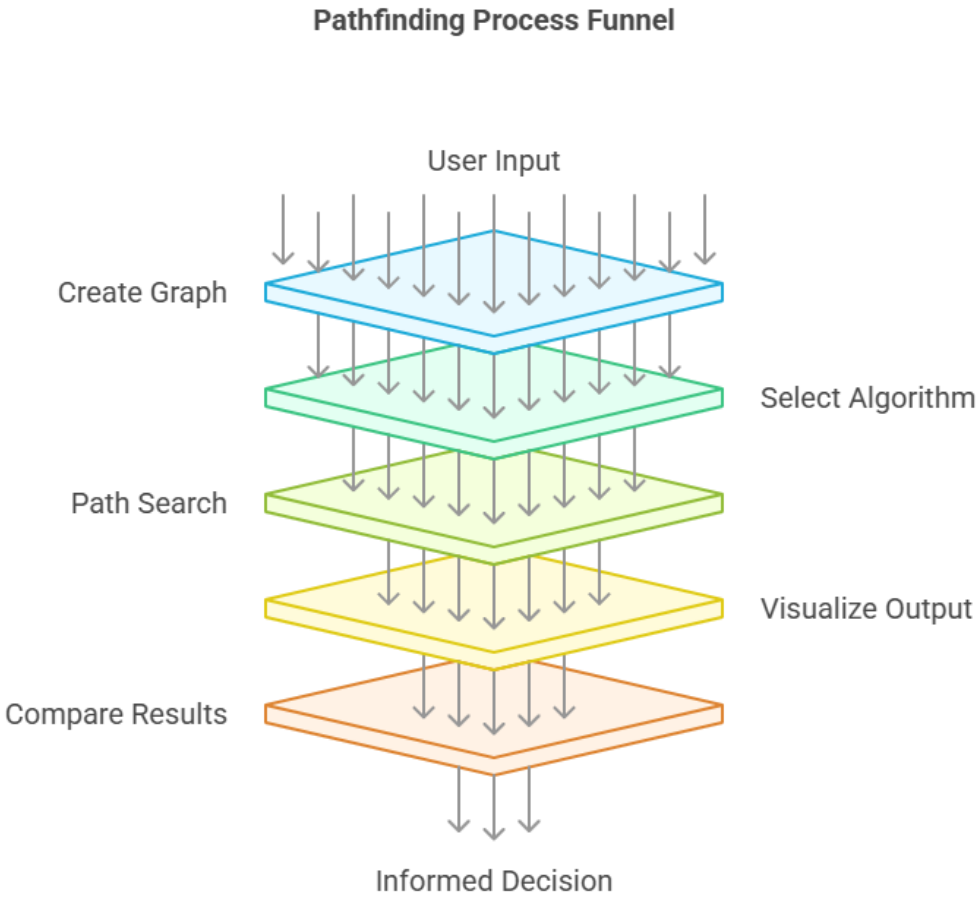
- Custom city input and edge weight assignment.
- Algorithm selector (BFS, DFS, A*, Hill Climbing).
- Visual route graph using Matplotlib or NetworkX.
- Cost calculation and path display. Python libraries such as NetworkX, Matplotlib, and Tkinter were selected for graph creation and UI.

3.3 limitations.

- Limited to undirected and weighted graphs.
- No real-world map data (simulation-based only).
- Performance may vary with graph size.
- Hill Climbing may not always yield optimal paths.

3.4 Design Flow

- Input Nodes: Users define cities and connections.
- Create Graph: A weighted undirected graph is generated.
- Select Algorithm: User chooses BFS, DFS, A*, or Hill Climbing.
- Path Search: Algorithm runs and computes the path.
- Visualize Output: Path and total cost are displayed.
- Compare Results: Algorithm efficiency is analyzed.



3.5 Best Design Selection

After multiple tests:

- **A*** produced the most consistent and shortest paths.
- **BFS** was reliable for unweighted graphs.
- **DFS** was faster but less accurate.
- **Hill Climbing** had limitations due to its greedy nature. Hence, **A*** was chosen as the best overall model for shortest-path routing, though all are retained for educational comparison.

CHAPTER 4:

RESULTS ANALYSIS AND VALIDATION

4.1 Implementation of Design Using Modern Engineering Tools

- **Pandas**: Used for data manipulation and preprocessing.
- **NumPy**: Essential for handling numerical computations.
- **Scikit-learn**: Used to train the Linear Regression model and evaluate performance.
- **Matplotlib**: Visualized the student's historical performance with bar charts.
- **ReportLab**: Generated professional PDF reports with performance details.

4.2 Code Implementation

Key code sections:

1. **Data Preprocessing (model_train.py):** Cleans and prepares the dataset, splits it into training and test sets, and trains the model.

```
import heapq
import networkx as nx
from geopy.distance import geodesic
import matplotlib.pyplot as plt

# Define cities and coordinates
cities = {
    "Mumbai": (19.0760, 72.8777),
    "Pune": (18.5204, 73.8567),
    "Nagpur": (21.1458, 79.0882),
    "Nashik": (20.0059, 73.7649),
    "Aurangabad": (19.8762, 75.3433),
    "Solapur": (17.6599, 75.9064),
    "Amravati": (20.9374, 77.7796),
    "Thane": (19.2183, 72.9781),
    "Kolhapur": (16.7050, 74.2433),
    "Latur": (18.4088, 76.5604)
}

edges = [
    ("Mumbai", "Pune"), ("Mumbai", "Thane"),
    ("Pune", "Nashik"), ("Pune", "Solapur"),
    ("Nashik", "Aurangabad"), ("Aurangabad", "Nagpur"),
    ("Nagpur", "Amravati"), ("Thane", "Nashik"),
    ("Solapur", "Latur"), ("Latur", "Nagpur"),
    ("Kolhapur", "Pune"), ("Kolhapur", "Solapur")
]

# Graph setup
graph = nx.Graph()
```

```
for city, coord in cities.items():
    graph.add_node(city, pos=coord)
for u, v in edges:
    dist = geodesic(cities[u], cities[v]).km
    graph.add_edge(u, v, weight=round(dist, 2))

def compute_heuristic(goal):
    return {city: geodesic(cities[city], cities[goal]).km for city in cities}

# Algorithms
def bfs(start, goal):
    from collections import deque
    queue = deque([(start, [start])])
    visited = set()
    while queue:
        node, path = queue.popleft()
        if node == goal:
            return path
        visited.add(node)
        for neighbor in graph.neighbors(node):
            if neighbor not in visited:
                queue.append((neighbor, path + [neighbor]))
    return None

def dfs(start, goal, path=None, visited=None):
    if path is None:
        path = [start]
    if visited is None:
        visited = set()
    if start == goal:
        return path
    visited.add(start)
    for neighbor in graph.neighbors(start):
        if neighbor not in visited:
            new_path = dfs(neighbor, goal, path + [neighbor], visited)
            if new_path:
                return new_path
    return None

def best_first(start, goal, h):
    queue = [(h[start], start, [start])]
    visited = set()
    while queue:
        _, node, path = heapq.heappop(queue)
        if node == goal:
            return path
        visited.add(node)
        for neighbor in graph.neighbors(node):
            if neighbor not in visited:
                heapq.heappush(queue, (h[neighbor], neighbor, path + [neighbor]))
    return None

def a_star(start, goal, h):
    queue = [(h[start], 0, start, [start])]
    visited = set()
```

```
    while queue:
        _, cost, node, path = heapq.heappop(queue)
        if node == goal:
            return path
        visited.add(node)
        for neighbor in graph.neighbors(node):
            if neighbor not in visited:
                g = cost + graph.edges[node, neighbor]['weight']
                f = g + h[neighbor]
                heapq.heappush(queue, (f, g, neighbor, path + [neighbor]))
    return None

def hill_climb(start, goal, h):
    path = [start]
    current = start
    while current != goal:
        neighbors = list(graph.neighbors(current))
        if not neighbors:
            return None
        next_node = min(neighbors, key=lambda n: h[n])
        if h[next_node] >= h[current]:
            return None
        path.append(next_node)
        current = next_node
    return path

def find_path(start, end, algo):
    h = compute_heuristic(end)
    if algo == 'BFS':
        return bfs(start, end)
    elif algo == 'DFS':
        return dfs(start, end)
    elif algo == 'Best-First':
        return best_first(start, end, h)
    elif algo == 'A*':
        return a_star(start, end, h)
    elif algo == 'Hill Climbing':
        return hill_climb(start, end, h)
    return None

def draw_graph(path, start, end):
    pos = nx.get_node_attributes(graph, 'pos')
    path_edges = list(zip(path, path[1:]))

    plt.figure(figsize=(10, 7))
    nx.draw(graph, pos, with_labels=True, node_size=1000, node_color="lightblue")
    nx.draw_networkx_edges(graph, pos, edge_color='gray')
    nx.draw_networkx_edge_labels(graph, pos, edge_labels=nx.get_edge_attributes(graph,
'weight'))
    nx.draw_networkx_edges(graph, pos, edgelist=path_edges, edge_color='red', width=3)
    plt.title(f"Route from {start} to {end}")
    plt.savefig("static/route.png")
    plt.close()
    return sum(graph.edges[path[i], path[i + 1]]['weight'] for i in range(len(path) - 1))
```

2. **Web Interface (app.py):** Accepts user inputs, runs the prediction, and provides feedback.

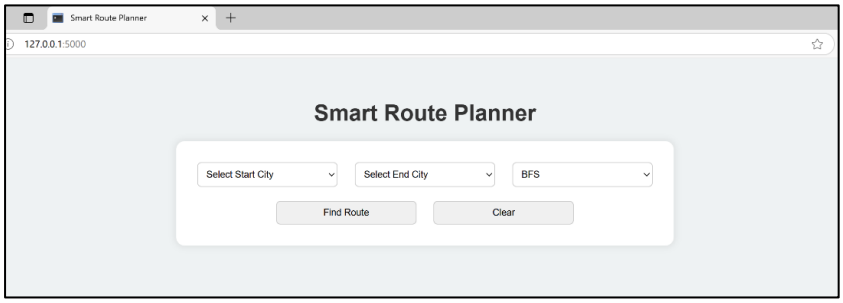
```
from flask import Flask, render_template, request

from cities import cities, find_path, draw_graph
import webbrowser
import threading
import os

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def index():
    result = None
    if request.method == 'POST':
        start = request.form['start']
        end = request.form['end']
        algo = request.form['algo']
        path = find_path(start, end, algo)
        if path:
            distance = draw_graph(path, start, end)
            result = {
                "path": " → ".join(path),
                "distance": round(distance, 2),
                "image": "route.png"
            }
        else:
            result = {"error": "No path found using the selected algorithm."}
    return render_template("index.html", cities=cities.keys(), result=result)

def open_browser():
    webbrowser.open_new("http://127.0.0.1:5000")

if __name__ == "__main__":
    # Only open the browser on the first run (not on reload)
    if os.environ.get("WERKZEUG_RUN_MAIN") == "true":
        threading.Timer(1, open_browser).start()
    app.run(debug=True, use_reloader=True)
```

4.3 Output Analysis

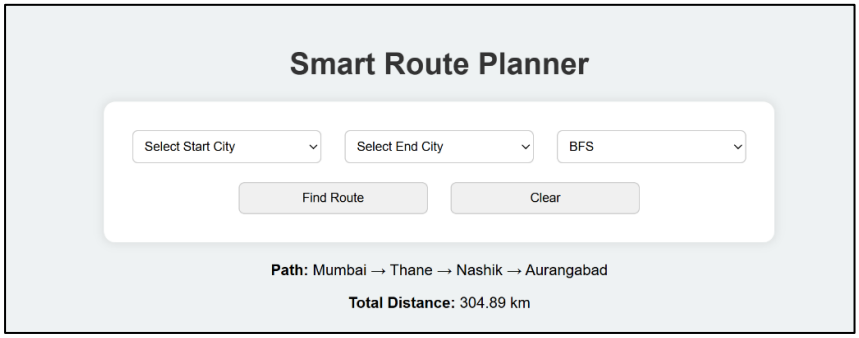
Algorithm	**Path Found**	**Optimal Path**	**Time Efficiency**	**Notes**
BFS	Yes	Sometimes	Moderate	Good for small graphs, not heuristic-based
DFS	Yes	No	Fast	May miss shortest path

Algorithm	Path Found	Optimal Path	Time Efficiency	Notes
A*	Yes	Yes	High	Best balance of accuracy and speed
Hill Climbing	Sometimes	No	Fast	Can get stuck in local minima

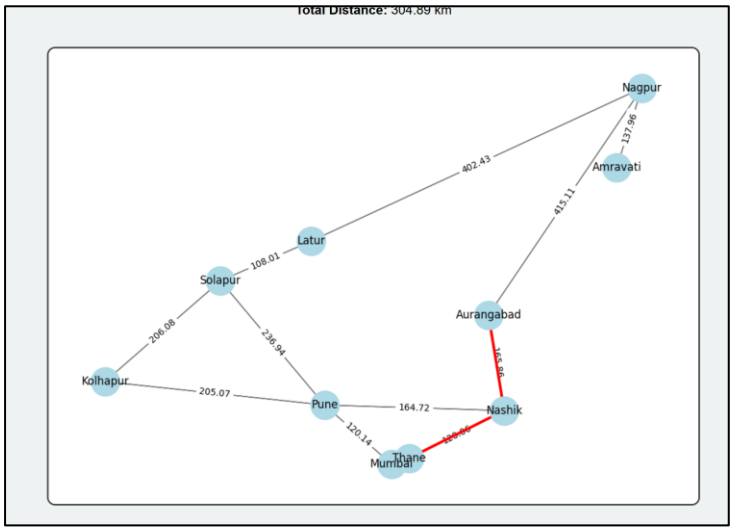
⇨ **Running the code:**



⇨ **Entering the values:**



⇨ **OUTPUT:**



CHAPTER 5:
CONCLUSION AND FUTURE WORK

Conclusion

The **Network Route Planner** effectively demonstrates the application of classical AI search algorithms—**BFS, DFS, A*, and Hill Climbing**—in solving real-world problems like city-to-city route optimization. Through user-defined inputs and dynamic graph visualization, the system computes and displays efficient travel paths, offering insights into each algorithm's behaviour, strengths, and limitations.

The results show that **A*** consistently finds the shortest path with high accuracy due to its heuristic-driven approach, while **BFS** guarantees the shortest path in unweighted graphs. **DFS** and **Hill Climbing**, although faster in some cases, may not always yield optimal results. This comparative approach enhances understanding of how different algorithms perform under varying conditions.

Overall, the project validates the role of **Artificial Intelligence in smart transportation and logistics**, and highlights the potential for scalable, cost-effective solutions in route planning.

Future improvements.

- **Integration with Real Maps**:
  Connect the planner with real geographic data using APIs like Google Maps or OpenStreetMap to support live navigation.
- **Heuristic Enhancements**:
  Implement more advanced heuristics in A* (e.g., Manhattan or Euclidean distance) to improve accuracy in real-world terrains.
- **Algorithm Performance Metrics**:
  Add time complexity, memory usage, and step count analysis to compare algorithms quantitatively.
- **Multi-Agent Routing**:
  Extend the system to handle multiple agents or vehicles, enabling applications in fleet routing or delivery systems.

REFERENCES

• Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson Education.
– Comprehensive coverage of AI search algorithms including BFS, DFS, A*, and Hill Climbing.

• Carmen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
– Detailed explanations of algorithmic strategies and graph search techniques.

• Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann.
– A foundational book discussing AI algorithms and heuristic search methods.

• GeeksforGeeks. (n.d.). *Graph Algorithms – BFS, DFS, A* and Hill Climbing*.
Retrieved from: https://www.geeksforgeeks.org/
– Online tutorials and implementations for learning and applying search algorithms.

• Skiena, S. S. (2008). *The Algorithm Design Manual* (2nd ed.). Springer.
– Practical guide to implementing graph algorithms and route planning logic.

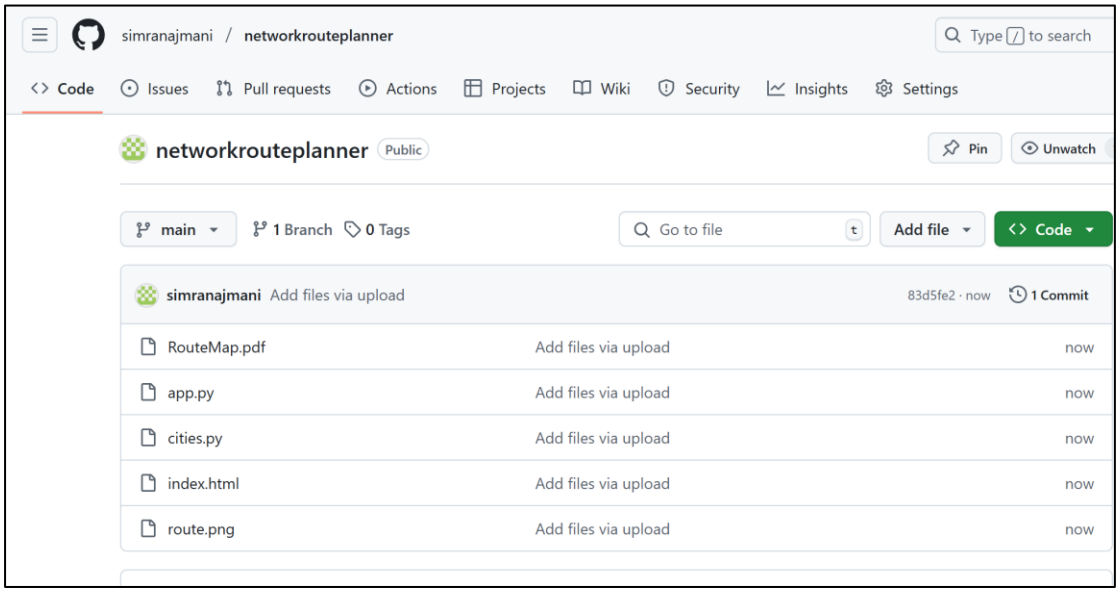
• Python Software Foundation. (n.d.). *Python 3 Documentation*.
Retrieved from: https://docs.python.org/3/
– Official documentation used for implementing the simulation in Python.

• Matplotlib Development Team. (n.d.). *Matplotlib: Visualization with Python*.
Retrieved from: https://matplotlib.org/
– Used for visualizing the network and routes in the project.

ONLINE PLATFORM

GITHUB:-

https://github.com/simranajmani/networkrouteplanner.git



RESULT / OUTCOMES:

The Network Route Planner successfully simulates intelligent city-to-city routing using four different search algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), A*, and Hill Climbing. The project allows users to define custom cities and distances between them, and then visualize the calculated paths along with the total travel cost or distance.

Key Outcomes:

- Shortest Path Identification:
  The system accurately identifies the shortest or most optimal route between cities using A* and BFS algorithms, while demonstrating the comparative limitations of DFS and Hill Climbing in certain scenarios.
- Algorithm Comparison:
  Users can compare how each algorithm explores the network differently, helping in understanding the trade-offs between path optimality, speed, and computation.
- Interactive Visualization:
  A graphical display of nodes (cities) and edges (routes) enhances user understanding by showing how the algorithm traverses the network.
- AI in Real-world Applications:
  The project showcases how AI techniques can be used in transportation systems, logistics planning, and smart city infrastructure to optimize route planning and reduce travel costs.
- Educational Value:
  The project serves as a learning tool for understanding the working and performance of search algorithms in artificial intelligence, especially when applied to real-world graph-based problems.