SUMMER TRAINING PROJECT REPORT

Maze Solver Visualizer

A training report

Submitted in partial fulfillment of the requirements for the award of degree of

Bachelor of Technology

In

Computer Science and Engineering (L.E)

Submitted to

LOVELY PROFESSIONAL UNIVERSITY

PHAGWARA, PUNJAB

From 23/06/25 to 30/07/25

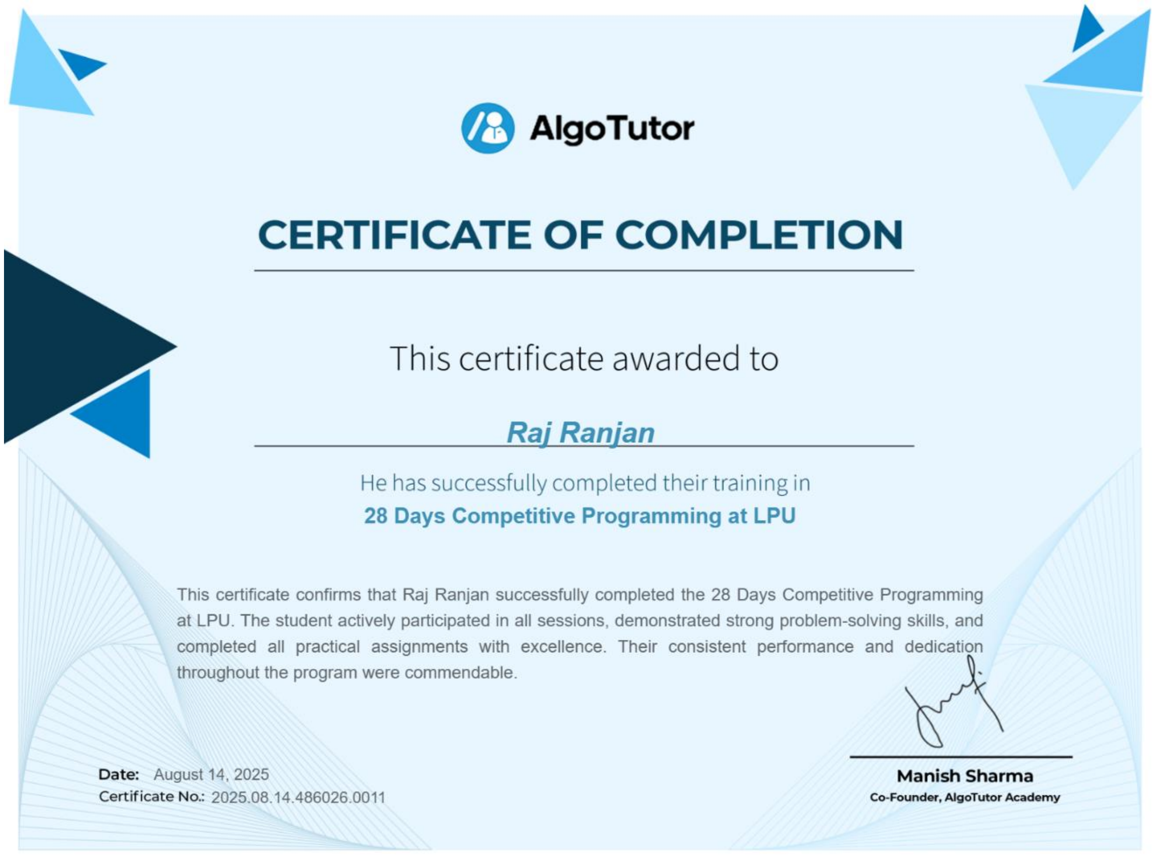
SUBMITTED BY:

Name of student: Raj Ranjan

Registration Number: 12400244

Signature of the student:

CERTIFICATE



AlgoTutor

CERTIFICATE OF COMPLETION

This certificate awarded to

Raj Ranjan

He has successfully completed their training in

28 Days Competitive Programming at LPU

This certificate confirms that Raj Ranjan successfully completed the 28 Days Competitive Programming at LPU. The student actively participated in all sessions, demonstrated strong problem-solving skills, and completed all practical assignments with excellence. Their consistent performance and dedication throughout the program were commendable.

Date: August 14, 2025

Certificate No.: 2025.08.14.486026.0011

[illegible]

Manish Sharma

Co-Founder, AlgoTutor Academy

Student Declaration

To whom so ever it may concern

I, **Raj Ranjan, Reg. No. 12400244**, hereby declare that the work done by me on **"Maze Solver Visualizer"** from **June, 2025** to **July, 2025**, is a record of original work for the partial fulfillment of the requirements for the award of the degree, **Bachelor of Technology in Computer Science and Engineering (L.E)**.

Raj Ranjan (12400244)

Signature of the student:

Dated:

ACKNOWLEDGEMENT

I would like to express my sincere gratitude and profound thanks to all those who have contributed directly or indirectly to the successful completion of my Summer Training/Internship Project Report on the **" Maze Solver Visualizer"**.

First and foremost, I extend my heartfelt appreciation to **Lovely Professional University (LPU)** for providing the invaluable opportunity to undertake this summer training, which significantly enhanced my practical skills and theoretical knowledge in **Competitive Programming**.

I am immensely grateful to my esteemed guide, **Mr. Azher Beg**, our mentor. His continuous encouragement, insightful guidance, constant support, and willingness to share his profound knowledge were instrumental throughout the entire duration of this training and the development of this project. His patient instruction and constructive feedback were invaluable in overcoming challenges and achieving the project objectives.

I would also like to thank the entire faculty and staff of the **School of Computer Science and Engineering** for their excellent teaching, support, and for fostering a conducive learning environment.

Finally, I express my gratitude to my family and friends for their unwavering support and encouragement during this training period.

Name: - Raj Ranjan

Registration No: - 12400244

List of Figures

Figure No.	Figure Name	Page
1.1	HTML Implementation	10
1.2	Event Listeners	11
1.3	BFS Algorithm	11
1.4	DFS Algorithm	11
1.5	Dijkstra’s Algorithm	12
1.6	A* Algorithm	12
1.7	Work Plan	14
1.8	List of Algorithms	14
1.9	Control Panel for User Interactivity	15
1.10	Initial State or Reset State	16
1.11	Final State	17

CHAPTER 1

INTRODUCTION OF THE PROJECT UNDERTAKEN

1.1 Title of the project: Maze Solver Visualizer

1.2 Overview of the project:

- The **Maze Solver Visualizer** is an interactive, browser-based application designed to demonstrate the working of various pathfinding algorithms in a visually intuitive manner. It allows users to generate customizable mazes and observe the step-by-step execution of different algorithms such as **Breadth-First Search (BFS)**, **Depth-First Search (DFS)**, **Dijkstra's Algorithm**, and **A* Search**.

- Users can adjust the **grid size**, **obstacle percentage**, and **visualization speed** according to their preference. The interface supports dynamic user interaction, enabling them to set start and end points, add or remove walls, and select the algorithm they want to visualize.

- The core objective of the project is to make complex pathfinding concepts accessible and engaging for learners and enthusiasts in Data Structures & Algorithms (DSA) and Competitive Programming (CP). By visually representing how different algorithms explore and choose paths, the application bridges the gap between theoretical understanding and practical comprehension.

- The project is implemented using **HTML, CSS (with Tailwind CSS)**, and **JavaScript** for the frontend logic, ensuring platform independence and ease of access through any modern web browser.

1.3 Objectives of this project:

1. **Implement Multiple Pathfinding Algorithms**
   Develop and integrate various algorithms (BFS, DFS, Dijkstra, A*) to compare their efficiency, accuracy, and behaviour in solving mazes.

2. **Provide Interactive Visualization**
   Create an intuitive and dynamic interface where users can watch step-by-step algorithm execution, improving understanding of the underlying logic.

3. **Enable User Customization**
   Allow users to modify grid size, obstacle density, and start/end points to explore different maze configurations.

4. **Control Execution Speed**
   Provide adjustable speed controls so users can observe algorithms at varying paces for better comprehension.

5. **Enhance Learning Through Experimentation**
   Encourage hands-on learning by enabling users to choose algorithms, tweak parameters, and compare results.

6. **Analyse Algorithm Performance**
   Display key metrics like path length, number of visited nodes, and execution time for comparative analysis.

7. **Support Manual Maze Editing**
   Allow users to draw or erase obstacles manually for custom test scenarios.

8. **Improve Problem-Solving Skills**
   Strengthen users' algorithmic thinking by showing real-world applications of graph traversal and shortest path algorithms.

1.4 Importance and Applicability:

- The Maze Solver Visualizer is important because it bridges the gap between theoretical knowledge of algorithms and practical understanding through visualization. It helps students, educators, and programmers see how different pathfinding algorithms behave in real time.
  Its applicability spans:
  - **Education:** Teaching DSA concepts in schools, colleges, and training programs.
  - **Research:** Comparing algorithm performance for AI pathfinding problems.
  - **Gaming Development:** Designing AI movement for maze-based or open-world games.
  - **Robotics:** Simulating navigation algorithms for autonomous robots.
  - **Optimization Problems:** Applying pathfinding techniques to logistics, routing, and network flow problems.

1.5 Scope of the Project:

- The project focuses on implementing multiple pathfinding algorithms with customizable visualizations, enabling interactive learning.
  Scope includes:

  1. Support for BFS, DFS, Dijkstra, and A* algorithms.
  2. Adjustable grid size, obstacle density, and visualization speed.
  3. Real-time color-coded visualization of visited cells, frontier nodes, and the final path.
  4. Manual maze editing to create custom scenarios.
  5. Performance statistics such as path length, number of visited nodes, and execution time.
  6. Comparison of algorithm efficiency under different conditions.

1.6 Relevance of the project:

- The This project is highly relevant to:

  1. **Computer Science Students:** Demonstrates direct application of graph theory, queues, stacks, and priority queues.
  2. **Competitive Programmers:** Offers insights into time complexity, heuristic functions, and optimization strategies.
  3. **Software Developers:** Highlights interactive UI design with efficient algorithm implementation.
  4. **Industry Use Cases:** Forms the basis for navigation systems, AI agents, and automated planning tools. Its real-time visualization approach makes abstract DSA concepts tangible and engaging, fostering deeper comprehension.

1.7 Work Plan:

- This The project will be carried out in the following planned phases:

1. **Requirement Analysis & Research**
    a. Identify suitable pathfinding algorithms (BFS, DFS, Dijkstra, A*).
    b. Study visualization techniques for real-time simulation.
2. **Design Phase**
    a. Design the grid layout using HTML/CSS.
    b. Plan colour schemes for visited nodes, frontier nodes, and final path.
    c. Create UI controls for grid size, obstacle percentage, speed, and algorithm selection.
3. **Algorithm Development**
    a. Implement BFS, DFS, Dijkstra, and A* algorithms in JavaScript.
    b. Ensure each algorithm can run independently with clear visualization steps.
4. **User Interaction & Customization**
    a. Enable manual maze editing with mouse clicks/drags.
    b. Implement obstacle generation based on user-defined percentage.
5. **Visualization & Animation**
    a. Add adjustable speed control to slow down or speed up pathfinding animations.
    b. Use smooth transitions to highlight the algorithm's progress.
6. **Testing & Debugging**
    a. Test various grid sizes, obstacle percentages, and speeds for correctness.
    b. Verify algorithms produce correct paths and handle edge cases.

7. **Documentation & Final Report**

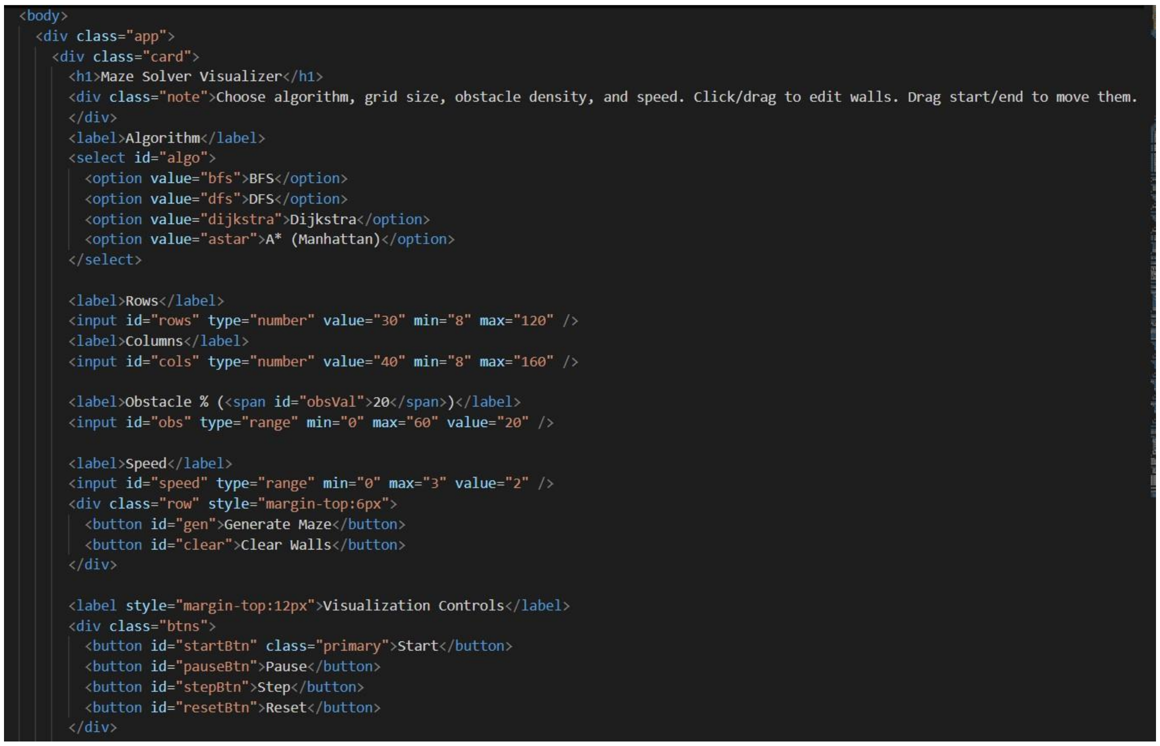
    a. Document objectives, scope, implementation details, and results.

    b. Prepare user instructions for operating the visualizer.

1.8 Implementation:

- The Maze Solver Visualizer is implemented as a **web-based application** using the following technologies:

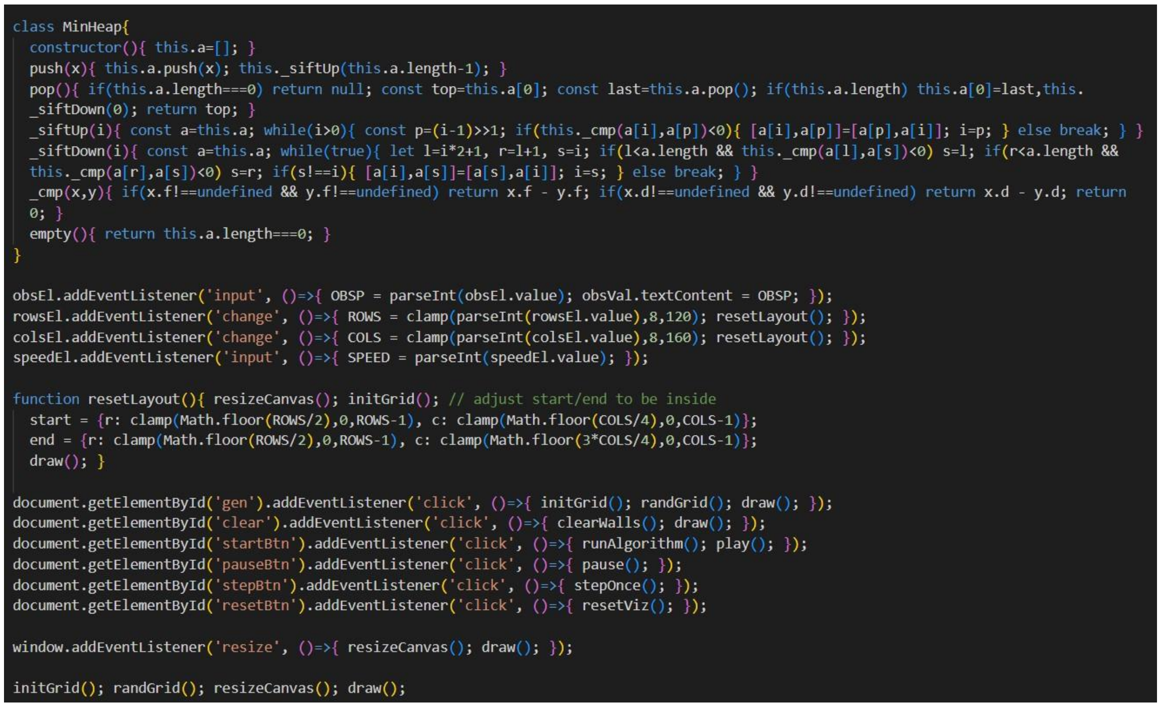
    1. **HTML5 & CSS3** → For grid layout and UI styling.



```html
<body>
  <div class="app">
    <div class="card">
      <h1>Maze Solver Visualizer</h1>
      <div class="note">Choose algorithm, grid size, obstacle density, and speed. Click/drag to edit walls. Drag start/end to move them.
      </div>
      <label>Algorithm</label>
      <select id="algo">
        <option value="bfs">BFS</option>
        <option value="dfs">DFS</option>
        <option value="dijkstra">Dijkstra</option>
        <option value="astar">A* (Manhattan)</option>
      </select>

      <label>Rows</label>
      <input id="rows" type="number" value="30" min="8" max="120" />
      <label>Columns</label>
      <input id="cols" type="number" value="40" min="8" max="160" />

      <label>Obstacle % (<span id="obsVal">20</span>)</label>
      <input id="obs" type="range" min="0" max="60" value="20" />

      <label>Speed</label>
      <input id="speed" type="range" min="0" max="3" value="2" />
      <div class="row" style="margin-top:6px">
        <button id="gen">Generate Maze</button>
        <button id="clear">Clear Walls</button>
      </div>

      <label style="margin-top:12px">Visualization Controls</label>
      <div class="btns">
        <button id="startBtn" class="primary">Start</button>
        <button id="pauseBtn">Pause</button>
        <button id="stepBtn">Step</button>
        <button id="resetBtn">Reset</button>
      </div>
```

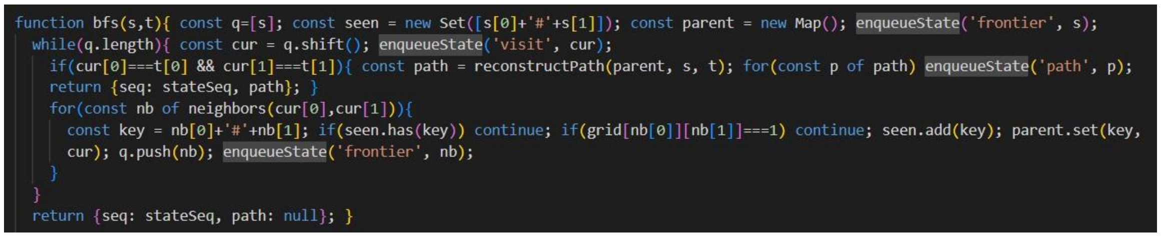
Figure 1.1: HTML Implementation

2. **JavaScript (ES6)** → For implementing algorithms and controlling visualizations.



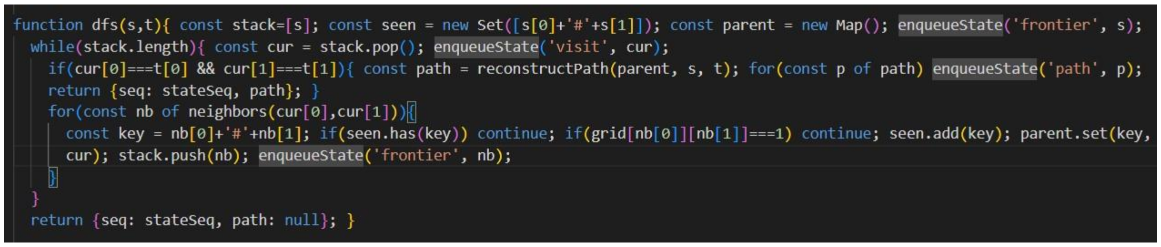
```
class MinHeap{
  constructor(){ this.a=[]; }
  push(x){ this.a.push(x); this._siftUp(this.a.length-1); }
  pop(){ if(this.a.length===0) return null; const top=this.a[0]; const last=this.a.pop(); if(this.a.length) this.a[0]=last,this.
  _siftDown(0); return top; }
  _siftUp(i){ const a=this.a; while(i>0){ const p=(i-1)>>1; if(this._cmp(a[i],a[p])<0){ [a[i],a[p]]=[a[p],a[i]]; i=p; } else break; } }
  _siftDown(i){ const a=this.a; while(true){ let l=i*2+1, r=l+1, s=i; if(l<a.length && this._cmp(a[l],a[s])<0) s=l; if(r<a.length &&
  this._cmp(a[r],a[s])<0) s=r; if(s!==i){ [a[i],a[s]]=[a[s],a[i]]; i=s; } else break; } }
  _cmp(x,y){ if(x.f!==undefined && y.f!==undefined) return x.f - y.f; if(x.d!==undefined && y.d!==undefined) return x.d - y.d; return
  0; }
  empty(){ return this.a.length===0; }
}

obsEl.addEventListener('input', ()=>{ OBSP = parseInt(obsEl.value); obsVal.textContent = OBSP; });
rowsEl.addEventListener('change', ()=>{ ROWS = clamp(parseInt(rowsEl.value),8,120); resetLayout(); });
colsEl.addEventListener('change', ()=>{ COLS = clamp(parseInt(colsEl.value),8,160); resetLayout(); });
speedEl.addEventListener('input', ()=>{ SPEED = parseInt(speedEl.value); });

function resetLayout(){ resizeCanvas(); initGrid(); // adjust start/end to be inside
  start = {r: clamp(Math.floor(ROWS/2),0,ROWS-1), c: clamp(Math.floor(COLS/4),0,COLS-1)};
  end = {r: clamp(Math.floor(ROWS/2),0,ROWS-1), c: clamp(Math.floor(3*COLS/4),0,COLS-1)};
  draw(); }

document.getElementById('gen').addEventListener('click', ()=>{ initGrid(); randGrid(); draw(); });
document.getElementById('clear').addEventListener('click', ()=>{ clearWalls(); draw(); });
document.getElementById('startBtn').addEventListener('click', ()=>{ runAlgorithm(); play(); });
document.getElementById('pauseBtn').addEventListener('click', ()=>{ pause(); });
document.getElementById('stepBtn').addEventListener('click', ()=>{ stepOnce(); });
document.getElementById('resetBtn').addEventListener('click', ()=>{ resetViz(); });

window.addEventListener('resize', ()=>{ resizeCanvas(); draw(); });

initGrid(); randGrid(); resizeCanvas(); draw();
```

Figure 1.2: Event Listeners



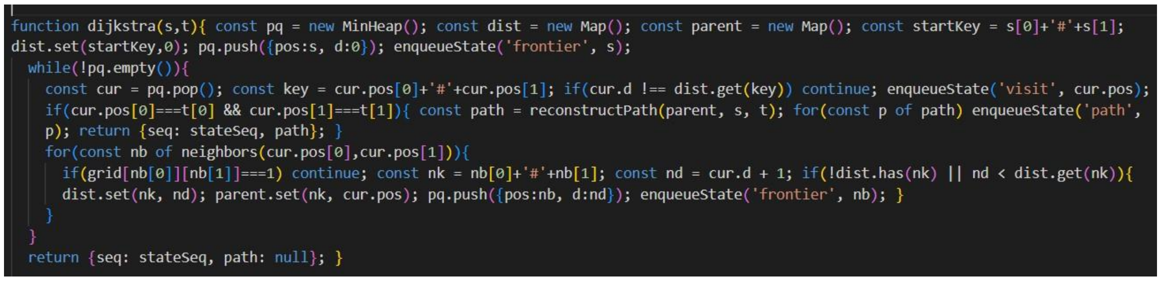
```
function bfs(s,t){ const q=[s]; const seen = new Set([s[0]+'#'+s[1]]); const parent = new Map(); enqueueState('frontier', s);
  while(q.length){ const cur = q.shift(); enqueueState('visit', cur);
    if(cur[0]===t[0] && cur[1]===t[1]){ const path = reconstructPath(parent, s, t); for(const p of path) enqueueState('path', p);
    return {seq: stateSeq, path}; }
    for(const nb of neighbors(cur[0],cur[1])){
      const key = nb[0]+'#'+nb[1]; if(seen.has(key)) continue; if(grid[nb[0]][nb[1]]===1) continue; seen.add(key); parent.set(key,
      cur); q.push(nb); enqueueState('frontier', nb);
    }
  }
  return {seq: stateSeq, path: null}; }
```

Figure 1.3: BFS Algorithm



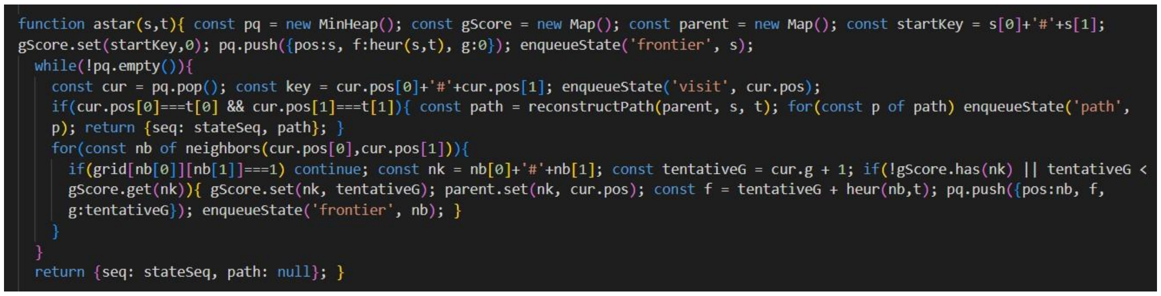
```
function dfs(s,t){ const stack=[s]; const seen = new Set([s[0]+'#'+s[1]]); const parent = new Map(); enqueueState('frontier', s);
  while(stack.length){ const cur = stack.pop(); enqueueState('visit', cur);
    if(cur[0]===t[0] && cur[1]===t[1]){ const path = reconstructPath(parent, s, t); for(const p of path) enqueueState('path', p);
    return {seq: stateSeq, path}; }
    for(const nb of neighbors(cur[0],cur[1])){
      const key = nb[0]+'#'+nb[1]; if(seen.has(key)) continue; if(grid[nb[0]][nb[1]]===1) continue; seen.add(key); parent.set(key,
      cur); stack.push(nb); enqueueState('frontier', nb);
    }
  }
  return {seq: stateSeq, path: null}; }
```

Figure 1.4: DFS Algorithm



```
function dijkstra(s,t){ const pq = new MinHeap(); const dist = new Map(); const parent = new Map(); const startKey = s[0]+'#'+s[1];
dist.set(startKey,0); pq.push({pos:s, d:0}); enqueueState('frontier', s);
  while(!pq.empty()){
    const cur = pq.pop(); const key = cur.pos[0]+'#'+cur.pos[1]; if(cur.d !== dist.get(key)) continue; enqueueState('visit', cur.pos);
    if(cur.pos[0]===t[0] && cur.pos[1]===t[1]){ const path = reconstructPath(parent, s, t); for(const p of path) enqueueState('path',
    p); return {seq: stateSeq, path}; }
    for(const nb of neighbors(cur.pos[0],cur.pos[1])){
      if(grid[nb[0]][nb[1]]===1) continue; const nk = nb[0]+'#'+nb[1]; const nd = cur.d + 1; if(!dist.has(nk) || nd < dist.get(nk)){
      dist.set(nk, nd); parent.set(nk, cur.pos); pq.push({pos:nb, d:nd}); enqueueState('frontier', nb); }
    }
  }
  return {seq: stateSeq, path: null}; }
```

Figure 1.5: Dijkstra's Algorithm



```
function astar(s,t){ const pq = new MinHeap(); const gScore = new Map(); const parent = new Map(); const startKey = s[0]+'#'+s[1];
gScore.set(startKey,0); pq.push({pos:s, f:heur(s,t), g:0}); enqueueState('frontier', s);
  while(!pq.empty()){
    const cur = pq.pop(); const key = cur.pos[0]+'#'+cur.pos[1]; enqueueState('visit', cur.pos);
    if(cur.pos[0]===t[0] && cur.pos[1]===t[1]){ const path = reconstructPath(parent, s, t); for(const p of path) enqueueState('path',
    p); return {seq: stateSeq, path}; }
    for(const nb of neighbors(cur.pos[0],cur.pos[1])){
      if(grid[nb[0]][nb[1]]===1) continue; const nk = nb[0]+'#'+nb[1]; const tentativeG = cur.g + 1; if(!gScore.has(nk) || tentativeG <
      gScore.get(nk)){ gScore.set(nk, tentativeG); parent.set(nk, cur.pos); const f = tentativeG + heur(nb,t); pq.push({pos:nb, f,
      g:tentativeG}); enqueueState('frontier', nb); }
    }
  }
  return {seq: stateSeq, path: null}; }
```

Figure 1.6: A* Algorithm

3. **Canvas / DOM Manipulation** → For dynamic updates to grid cells during execution.

- **Implementation Steps**

  1. **Grid Creation**

     - The grid is represented as a 2D array of nodes, each with properties like isWall, isVisited, and distance.

  2. **Algorithm Modules**

     - Each algorithm (BFS, DFS, Dijkstra, A*) is implemented as a separate function, taking the grid, start, and end points as parameters.

     - During execution, the algorithm pushes visited nodes into a queue for visualization.

3. **Visualization Logic**
   - The visualization engine iterates over the visited nodes array, colouring cells based on their state.
   - The final path is highlighted in a distinct colour (e.g., yellow).
4. **User Controls**
   - Dropdown to select algorithm.
   - Sliders for grid size, obstacle density, and speed control.
   - Buttons to generate random obstacles, clear the grid, and start the algorithm.
5. **Speed Control Implementation**
   - Visualization is delayed using setTimeout or async/await to match user-selected speed.
6. **Event Handling**
   - Mouse events allow the user to place or remove walls.
   - Start and end points can be dragged to new positions.
7. **Testing & Optimization**
   - Each algorithm is tested on multiple configurations to ensure correctness.
   - Optimizations are applied to prevent unnecessary re-renders for large grids.

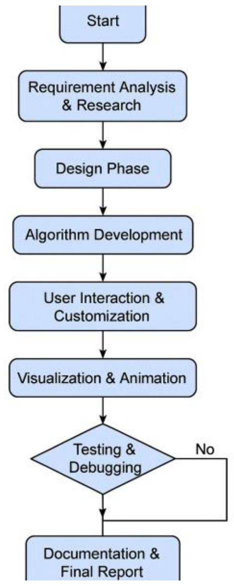

Start
Requirement Analysis & Research
Design Phase
Algorithm Development
User Interaction & Customization
Visualization & Animation
Testing & Debugging
No
Documentation & Final Report

Figure 1.7 Work Plan

1.9 Some Project Snapshots

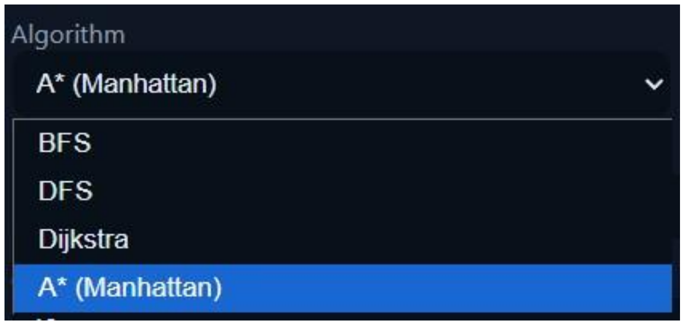
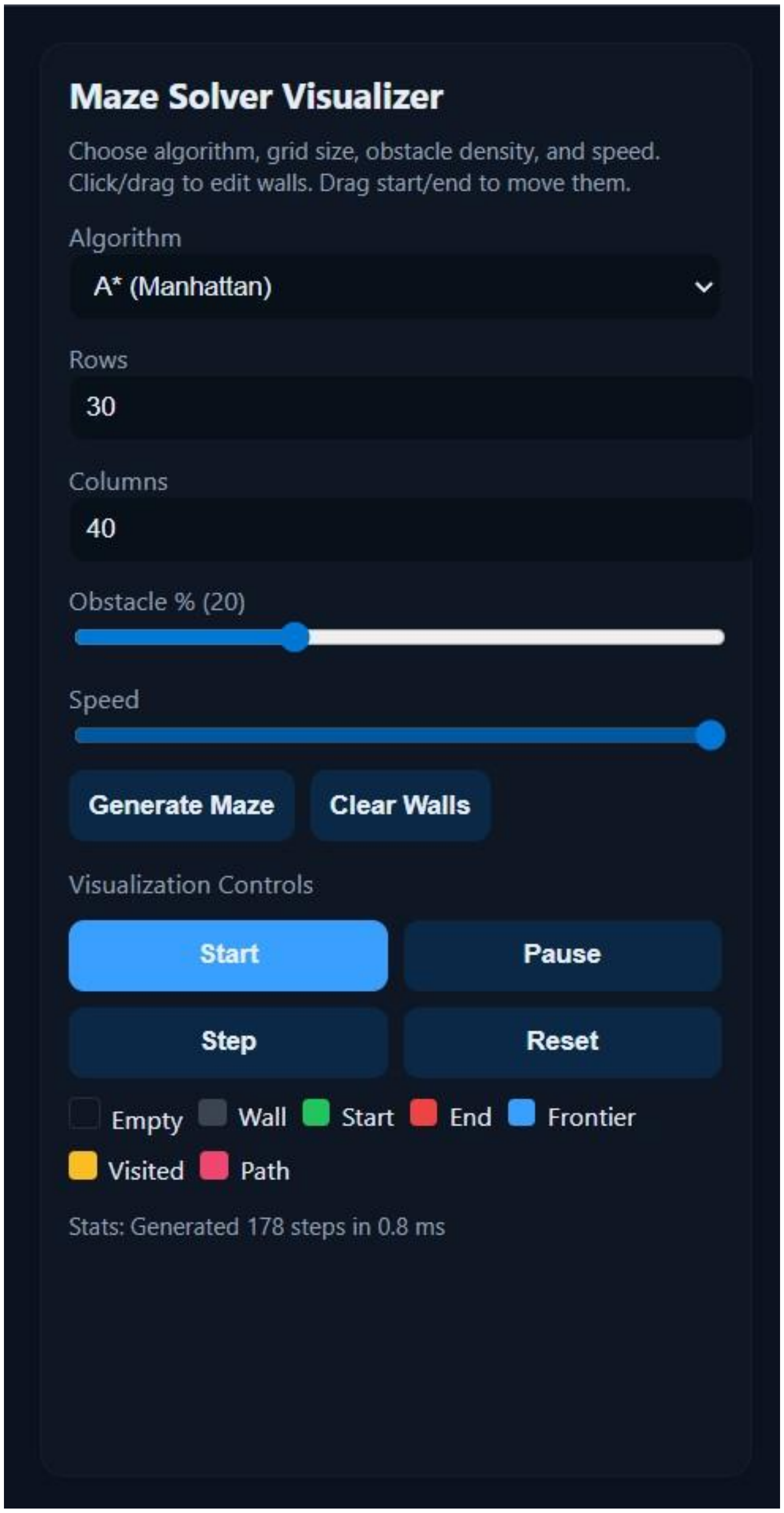


Figure 1.8: List of Algorithms



Maze Solver Visualizer

Choose algorithm, grid size, obstacle density, and speed. Click/drag to edit walls. Drag start/end to move them.

Algorithm: A* (Manhattan)

Rows: 30

Columns: 40

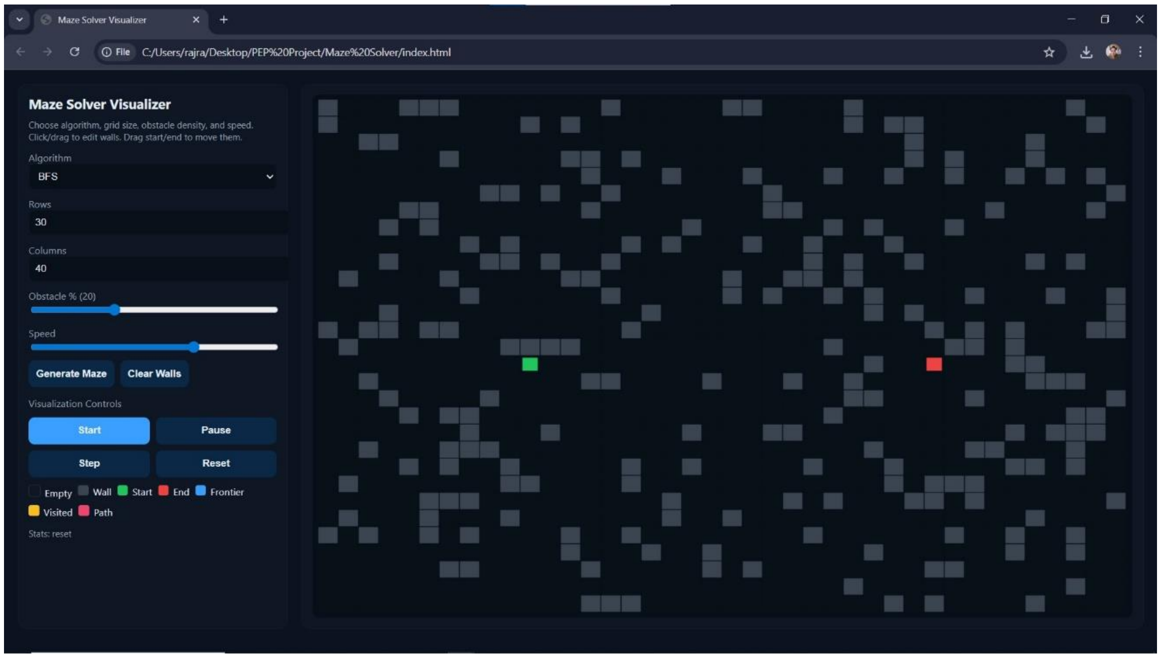
Obstacle % (20)

Speed

⬛ Empty ⬛ Wall 🟩 Start 🟥 End 🟦 Frontier 🟨 Visited 🟥 Path

Stats: Generated 178 steps in 0.8 ms

Figure 1.9: Control Panel for User Interactivity



Maze Solver Visualizer

Choose algorithm, grid size, obstacle density, and speed.
Click/drag to edit walls. Drag start/end to move them.

Algorithm: BFS

Rows: 30

Columns: 40

Obstacle % (20)

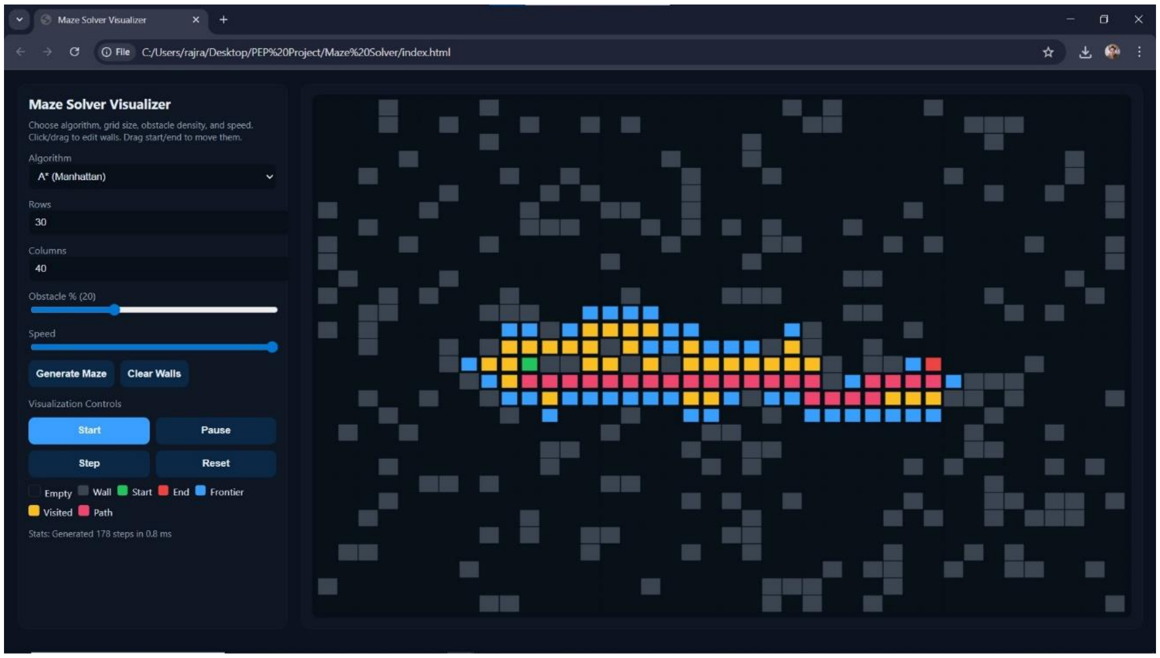
Speed

Visualization Controls

Empty ▨ Wall 🟩 Start 🟥 End 🟦 Frontier
🟨 Visited 🟥 Path

Stats: reset

Figure 1.10: Initial State or Reset State


Maze Solver Visualizer
C:/Users/rajra/Desktop/PEP%20Project/Maze%20Solver/index.html

Maze Solver Visualizer

Choose algorithm, grid size, obstacle density, and speed. Click/drag to edit walls. Drag start/end to move them.

Algorithm: A* (Manhattan)

Rows: 30

Columns: 40

Obstacle % (20)

Speed

▢ Empty ■ Wall 🟩 Start 🟥 End 🟦 Frontier 🟨 Visited 🟥 Path

Stats: Generated 178 steps in 0.8 ms

Figure 1.11: Final State

CHAPTER 2

TRAINING EXPERIENCE

2.1 Overview of the Training

- From 23rd June 2025 to 30th July 2025, I attended a comprehensive Data Structures, Algorithms, and Competitive Programming training program conducted by Algo Tutor. The program was designed to strengthen algorithmic thinking, competitive coding strategies, and mastery of essential data structures, preparing students for technical interviews and high-level programming contests.
- The Competitive Programming and Data Structures Algorithms training aimed to bridge the gap between academic programming knowledge and real-world problem-solving efficiency. The objective was to make participants adept at solving algorithmic challenges under strict time constraints while maintaining clean, optimized code.
- The training focused on strengthening algorithmic problem-solving skills and deepening the understanding of core data structures such as arrays, linked lists, stacks, queues, hash maps, heaps, bit manipulation, string manipulation, trees, dynamic programming and graphs.

2.2 Key Learning Outcomes

- The following were the major outcomes of the training:

    1. **Algorithm Analysis:**
        - Proficient in evaluating time and space complexity for different algorithms, enabling the selection of optimal approaches for competitive problems.
    2. **Search and Sorting Techniques:**
        - Implemented and optimized classic algorithms such as binary search, merge sort, quick sort, and their variations used in contest problems.
    3. **Advanced Data Structures:**
        - Developed in-depth understanding of graphs, heaps, tries, and segment trees, along with traversal and query algorithms like DFS, BFS, and Dijkstra’s.

4. **Dynamic Programming:**

   - Mastered problem-solving techniques for both classical DP problems (Knapsack, LIS, Matrix Chain Multiplication) and advanced optimizations.

5. **Competitive Mindset:**

   - Improved my ability to solve problems under time constraints and developed strategies for tackling unknown problem patterns in live contests.

2.3 Integration with Competitive Programming Platforms

- The knowledge gained during the course was actively applied on platforms such as Codeforces and LeetCode.

1. **Problem Solving Efficiency:**

   - Implemented DSA concepts to solve problems in the minimum possible time and with optimal space usage.

2. **Graph Algorithms:**

   - Applied BFS/DFS in competitive problems involving shortest paths, cycle detection, and connectivity checks.

2.4 Impact of the Training

- The Algo Tutor training elevated my technical problem-solving skills and contest performance:

- Reduced solution implementation time in contests through better template usage and modular coding.

- Strengthened fundamentals that will serve as a base for learning advanced algorithms like flow networks and computational geometry.

- Developed confidence to tackle algorithm-heavy problems in interviews and competitive scenarios

- The training not only elevated my technical skill set but also reinforced the importance of DSA as the backbone of real-world application development.

2.5 Areas covered during training:

- The training program encompassed a wide range of topics essential for competitive programming:
    - **Core Data Structures and Their Applications**
        - Gained proficiency in using fundamental data structures like Arrays, Linked Lists, Stacks, Queues, Heaps, Sets, and Hash Maps.
        - Learned to apply these structures to solve real-world competitive programming problems efficiently.
    - **Mathematical Foundations for Problem Solving**
        - Mastered mathematical tools including GCD, LCM, Prime Factorization, Modular Arithmetic, and Fast Exponentiation.
        - Explored number theory concepts like Euler's Totient Function and implemented prime generation using the Sieve of Eratosthenes.
    - **Efficient Searching and Sorting Techniques**
        - Studied and implemented algorithms like Quick Sort, Merge Sort, and Binary Search.
        - Learned how to integrate sorting and searching with Greedy Algorithms for optimizing solutions.
    - **Prefix Sums and Range Query Optimization**
        - Learned to precompute cumulative sums and use them for fast range sum queries.
        - Solved problems involving Special Index concepts and optimized multiple query processing.
    - **Tree Data Structures and Traversal Algorithms**
        - Explored Binary Trees, Binary Search Trees (BST), and their properties.
        - Implemented Depth-First Search (DFS) and Breadth-First Search (BFS) with variations like Inorder, Preorder, Postorder, and Level Order Traversals.

- **Advanced Stack and Queue Techniques**
    - Understood and implemented Monotonic Stack and Monotonic Queue structures.
    - Applied sliding window techniques to optimize problems involving consecutive elements.
- **Recursion and Dynamic Programming (DP)**
    - Solved problems using recursion and memoization techniques.
    - Practiced DP on 1D, 2D, and 3D arrays, DP on trees, and classic problems like the Knapsack problem.
    - Applied backtracking for exhaustive search problems.
- **Graph Theory and Algorithms**
    - Represented graphs using adjacency lists/matrices and applied algorithms like Disjoint Set Union (DSU).
    - Learned Minimum Spanning Tree construction, Dijkstra’s shortest path algorithm, and Topological Sorting.
- **Segment Trees and Advanced Range Queries**
    - Built segment trees for sum, min, and max queries.
    - Learned update and query operations for dynamic datasets.
- **Practical Problem-Solving on Competitive Programming Classics**
    - Worked on well-known problems including Aggressive Cows, Book Allocation, N-Queen II, Pasha and String, Rat in a Maze, and Reconstructing a Two-Row Binary Matrix.
    - Focused on understanding patterns, constraints, and optimal solution approaches for each problem.

2.6 Weekly summary:

- **Week 1: Fundamentals, Arrays & Mathematical Concepts**
  - The first week focused on building a strong foundation in essential problem-solving tools and techniques. We began with arrays and their operations, then delved into important mathematical foundations such as GCD, LCM, and prime factorization, which are crucial for number theory problems. We explored modular arithmetic and its applications, along with fast exponentiation for handling large powers efficiently. Euler's Totient Function and the Sieve of Eratosthenes were introduced for prime number-related problems. Prefix sums and range queries were covered to handle subarray-based problems efficiently. We also discussed special index problems and implemented sorting algorithms such as Quick Sort and Merge Sort. The week concluded with binary search and greedy algorithms, both fundamental strategies in competitive programming.
- **Week 2: Data Structures & Traversal Techniques**
  - The second week concentrated on data structures and their efficient use in problem-solving. We learned about stacks and monotonic stacks, followed by queues and monotonic queues, essential for range minimum/maximum queries. The sliding window technique was introduced to optimize subarray problems. We also covered linked lists and hash-based structures such as hashmaps and sets for fast lookups. Moving into hierarchical structures, we studied trees and implemented DFS, BFS, inorder, preorder, postorder, and level order traversals. The concepts of binary trees and binary search trees (BSTs) were explained, along with heaps for priority-based processing.
- **Week 3: Recursion, Dynamic Programming & Backtracking**
  - The third week explored problem-solving through recursion and progressively moved to dynamic programming (DP). We implemented DP on 1D, 2D, and 3D arrays, solving problems involving states and transitions. We studied DP on trees for hierarchical problem structures and worked on classical problems like the Knapsack problem. Backtracking techniques were introduced to systematically explore all possibilities in problems like the N-Queens puzzle. This week strengthened optimization

and state management skills essential for advanced competitive programming.

- **Week 4: Graph Algorithms & Advanced Structures**
    - In the fourth week, we entered the domain of graphs and their algorithms. We studied different graph representations, learned Disjoint Set Union (DSU) for connectivity problems, and worked on minimum spanning tree algorithms. The Dijkstra algorithm for shortest paths and topological sorting for directed acyclic graphs were implemented. This week provided the tools to handle connectivity, shortest path, and dependency-based problems.

- **Week 4: Segment Trees & Problem-Solving Practice**
    - The final week was dedicated to segment trees for efficient range queries and updates. We also focused on solving important interview and contest problems, such as:
        - **Aggressive Cows** – Binary search on answer
        - **Book Allocation** – Partitioning with binary search
        - **N-Queen II** – Backtracking with optimization
        - **Pasha and String** – String manipulation and reversal
        - **Rat in a Maze** – Backtracking with pathfinding
        - **Reconstruct a Two Row Binary Matrix** – Matrix construction based on constraints
    - This week emphasized applying learned concepts to real problem scenarios, improving competitive readiness.

Final Chapter

CONCLUSION

- The 28-day Competitive Programming Training Program proved to be an intensive and structured learning experience that systematically built problem-solving skills from the ground up. Starting with the fundamentals in Week 1, the program reinforced the mathematical and algorithmic basics required to approach complex computational challenges. Topics like GCD, LCM, prime factorization, modular arithmetic, and sorting algorithms laid the groundwork for more advanced concepts in later weeks, ensuring that each participant possessed the necessary theoretical and practical tools to progress.

- By Week 2, the focus shifted towards mastering data structures and understanding their real-world applicability. The inclusion of stacks, queues, linked lists, hashmaps, and sets allowed participants to efficiently store, retrieve, and process data. Introducing trees, heaps, and traversal algorithms further enhanced logical thinking and provided insight into how hierarchical data can be navigated and manipulated, an essential skill for both competitive programming and software development.

- Week 3 was pivotal in introducing dynamic programming, recursion, and backtracking—three cornerstones of advanced problem solving. These techniques taught us to break down complex problems into smaller subproblems, reuse computations, and explore all possible solutions within given constraints. Learning to balance time and space complexity through these methods improved efficiency and problem-solving confidence.

- The final phase of the training, spanning Weeks 4 and 5, was dedicated to graph algorithms, advanced data structures, and high-level problem practice. Dijkstra's algorithm, Disjoint Set Union, and segment trees expanded our toolkit for solving competitive programming problems that require speed and precision. The hands-on practice with iconic problems such as Aggressive Cows, N-Queen II, and Rat in a Maze bridged the gap between theoretical understanding and real-time application.

- Overall, the project has not only strengthened algorithmic thinking but has also nurtured analytical reasoning, logical structuring, and optimization skills. It has provided a deep understanding of how different algorithms and data structures interact, how to approach problems methodically, and how to apply solutions efficiently under competitive constraints. This experience serves as a strong foundation for excelling in coding competitions, technical interviews, and real-world software development challenges.

Future Perspective

- The skills and concepts acquired during the **Competitive Programming Training** open up a broad range of possibilities for further growth and application. Competitive programming is not merely an academic exercise—it is a foundation for excelling in technical interviews, hackathons, and large-scale software development projects. In the future, the understanding of algorithms, data structures, and optimization techniques can be extended into domains such as artificial intelligence, data analytics, and high-performance computing. This training also sets the stage for participation in prestigious contests like ICPC, Codeforces, LeetCode Weekly Challenges, and Google Kick Start, where advanced problem-solving speed and efficiency are key to success.
- With continued practice, these skills can further evolve to meet real-world industry demands. Complex systems such as recommendation engines, search algorithms, and network routing solutions often rely on the same principles learned here. Moreover, understanding time and space trade-offs at this level equips learners to design software solutions that are both robust and scalable. This makes the training not just a short-term academic gain but a long-term professional asset.
- For the **Maze Visualizer Project**, the future holds exciting opportunities for enhancement and innovation. Beyond being a visualization tool for pathfinding algorithms, it can be expanded into a fully interactive educational platform for learners to experiment with algorithms like BFS, DFS, A*, and Dijkstra's in real time. Adding features such as weighted and unweighted maze generation, obstacle customization, performance comparisons, and step-by-step execution can transform it into a comprehensive learning aid for computer science students.
- Furthermore, the Maze Visualizer can be integrated into gamified learning experiences where users solve maze challenges under time constraints, encouraging both fun and practical application of algorithmic concepts. By connecting it with competitive programming platforms or using it as a teaching assistant in workshops, its reach can extend beyond academic use into mainstream coding education. With proper deployment on web and mobile platforms, it has the potential to become a widely used resource for both beginners and professionals.
- In conclusion, both the training program and the Maze Visualizer project have substantial future potential. While the training ensures that the participants have a strong algorithmic foundation for any career in technology, the Maze Visualizer serves as a creative bridge between theoretical learning and visual, hands-on understanding. Together, they represent a sustainable path toward continuous skill development, innovation, and impactful contributions to the tech community.