**KONGU ENGINEERING COLLEGE **

**(Autonomous) Perundurai,Erode – 638060**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**TRAVELLING SALESMAN BRANCH BOUND**

**A MICRO PROJECT REPORT FOR**

**DESIGN AND ANALYSIS OF ALGORITHMS (22ITT31)**

**SUBMITTED BY**

**MITHUN P M (23ITL187)**

**KONGU ENGINEERING COLLEGE** 

**(Autonomous) Perundurai,Erode – 638060**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**TRAVELLING SALESMAN USING BRANCH BOUND**

**A MICRO PROJECT REPORT FOR**

**DESIGN AND ANALYSIS OF ALGORITHMS (22ITT31)**

**SUBMITTED BY**

**MITHUN P M (23ITL187)**

# **Kongu Engineering College** **KONGU ENGINEERING COLLEGE Kongu Engineering College (Autonomous)**

# **(Autonomous) Perundurai,Erode – 638060**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**BONAFIED CERTIFICATE**

|  |  |
| --- | --- |
| **Name** | **:** MITHUN P M |
| **Course Code** | **:**22ITT31 |
| **Course Name** | **:**DESIGN AND ANALYSIS OF ALGORITHMS |
| **Semester** | **:** IV |

Certified that this is a bonafied record of work for application project done by the above student for 22ITT31-DESIGN AND ANALYSIS OF ALGORITHMS during the academic year 2024-2025.

Submitted for the Viva Voice Examination held on \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Faculty Incharge Head of the Department

**ABSTRACT**

The Travelling Salesman Problem (TSP) is a classical NP-hard combinatorial optimization problem that seeks the shortest possible route to visit a set of cities exactly once and return to the starting point. This project presents a web-based TSP Visualizer that compares two fundamental algorithmic approaches: Brute Force and Branch and Bound.

The Brute Force method exhaustively computes all possible permutations of city paths, resulting in a time complexity of O(n!) and space complexity of O(n). Although it guarantees an optimal solution, its exponential growth makes it impractical for larger datasets due to excessive computation time.

In contrast, the Branch and Bound algorithm improves efficiency by pruning infeasible paths using lower bound heuristics at each recursive step. While it still has a worst-case time complexity of O(n!), in practice it performs significantly faster by eliminating unnecessary computations. It maintains an optimal solution with a space complexity of O(n) due to the recursion stack and path tracking.

This visualizer illustrates the optimal path, total cost, execution time, and complexity metrics for both approaches using an interactive HTML5 Canvas interface powered by JavaScript. The system effectively showcases the performance advantage of the Branch and Bound technique over brute force, particularly in terms of reduced execution time. This project serves as both a practical demonstration of algorithmic design and an educational tool for understanding NP-hard problem-solving strategies.

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **CHAPTER NO** | **TITLE** | **PAGE NO** |
|  | **ABSTRACT** | **ix** |
| **1.** | **INTRODUCTION** | 6 |
|  | 1.1 PURPOSE | 7 |
|  | 1.2 OBJECTIVE | 7 |
|  | 1.3 METHODOLOGY OVERVIEW | 8 |
| **2.** | **PROBLEM STATEMENT** | 9 |
| **3.** | **METHODOLGY** | 10 |
|  | 3.1 Input & Initialization | 10 |
|  | 3.2 **Divide & Compare** | 10 |
|  | 3.3 Recursive Detection | 10 |
|  | 3.4 Visualization & Output | 10 |
| **4.** | **IMPLEMENTATION** | 11 |
|  | 4.1 Input & Initialization | 11 |
|  | 4.2 Divide & Compare | 11 |
|  | 4.3 Recursive Detection | 12 |
|  | 4.4 Visualization & Output | 12 |
| **5.** | **RESULTS** | 15 |

**1. INTRODUCTION**

The **Travelling Salesman Problem (TSP)** is a foundational challenge in computer science and combinatorial optimization. It involves finding the shortest possible route that allows a salesman to visit a set of cities exactly once and return to the starting city. Due to its NP-hard nature, the TSP is widely used to test optimization algorithms in logistics, planning, circuit design, and operations research.

Despite its simple definition, solving the TSP becomes computationally intensive as the number of cities increases. This project provides a comparative visualization of two algorithmic approaches: **Brute Force**, which examines every possible route, and **Branch and Bound**, which intelligently narrows down the possibilities using bounds.

**1.1 PURPOSE**

The purpose of this project is to:

* Develop a **web-based TSP visualizer** using HTML, CSS, and JavaScript.
* Simulate and compare the **Brute Force** and **Branch and Bound** algorithms.
* Help users understand the **trade-offs** in time complexity and performance between exhaustive and optimized search techniques.
* Provide a **graphical and interactive** demonstration that enhances learning.

**1.2 OBJECTIVE**

The main objectives of this project are:

* Implement TSP solving using **Brute Force** and **Branch and Bound** algorithms.
* Visualize the cities, the computed path, and route cost on a **2D HTML5 Canvas**.
* Compare the algorithms based on:
  + Total path cost
  + Execution time
  + Time and space complexities
* Educate users about **algorithmic decision-making**, recursive strategy, and **optimization techniques like pruning**.

**1.3 METHODOLOGY OVERVIEW**

To achieve the above objectives, the following technologies and methods are used:

* **HTML5 Canvas** for drawing cities and paths interactively.
* **JavaScript** to implement core TSP logic, event handling, and UI updates.
* **Fixed data set** of cities and inter-city distances for consistent comparisons.
* Use of performance.now() API to accurately track algorithm execution time.
* Modularized code for better clarity and separation of logic and visualization.

**2. PROBLEM STATEMENT**

The Travelling Salesman Problem presents a significant challenge as the **number of permutations increases factorially** with each additional city, making naive solutions like **Brute Force** infeasible for more than a handful of cities. While brute force guarantees optimality, it does so at a great computational cost. Thus, the need arises for **optimized approaches like Branch and Bound**, which use **heuristics and bounding functions** to eliminate paths that are guaranteed to be suboptimal, significantly improving performance while preserving optimality.

**3. METHODOLOGY**

**3.1 Input & Initialization**

* Define a set of 4–10 cities, each with (x, y) coordinates.
* Create a **distance matrix** that stores distances between all pairs of cities.
* Initialize necessary variables for each algorithm, such as:
  + visited[] array
  + path[]
  + minCost
  + timer variables for execution tracking

**3.2 Divide & Compare**

* **Brute Force:**
  + Generates all (n-1)! permutations of cities.
  + Calculates the cost for each permutation and tracks the minimum.
  + Guaranteed to find the optimal path but has very high time complexity.
* **Branch and Bound:**
  + Uses **recursive backtracking** with **bounding** (lower bound estimate).
  + If the current partial cost plus bound exceeds the best known cost, prune that branch.
  + This results in a significant reduction in computation.

**3.3 Recursive Detection**

* Both algorithms use recursion to explore paths.
* **Brute Force:** Exhaustively checks every possible path recursively.
* **Branch and Bound:** Uses **greedy pruning** to skip exploring costly paths.
* Recursive functions maintain a record of:
  + Current node
  + Total cost so far
  + Cities visited
  + Remaining cities

**3.4 Visualization & Output**

* Cities are plotted as **nodes (circles)** on the canvas.
* Paths are drawn as **lines** connecting the cities.
* The optimal path is highlighted once computation finishes.
* Output includes:
  + Total cost of the route
  + Time taken to compute
  + Algorithm used
  + Complexity (displayed in terms of theoretical bounds)

**4. IMPLEMENTATION**

    const dist = [

      [0, 67, 17, 56],

      [97, 0, 37, 25],

      [18, 95, 60, 30],

      [56, 50, 100, 90]

    ];

    const cityCoords = [

      { x: 907, y: 100 },

      { x: 600, y: 150 },

      { x: 878, y: 200 },

      { x: 570, y: 370 }

    ];

    class TSPBranchAndBound {

      constructor(dist) {

        this.dist = dist;

        this.n = dist.length;

        this.minCost = Infinity;

        this.finalPath = Array(this.n + 1).fill(-1);

        this.visited = Array(this.n).fill(false);

      }

      firstMin(i) {

        let min = Infinity;

        for (let k = 0; k < this.n; k++) {

          if (i !== k && this.dist[i][k] < min) min = this.dist[i][k];

        }

        return min;

      }

      secondMin(i) {

        let first = Infinity, second = Infinity;

        for (let j = 0; j < this.n; j++) {

          if (i === j) continue;

          if (this.dist[i][j] <= first) {

            second = first;

            first = this.dist[i][j];

          } else if (this.dist[i][j] <= second) {

            second = this.dist[i][j];

          }

        }

        return second;

      }

      copyToFinal(currPath) {

        for (let i = 0; i < this.n; i++) {

          this.finalPath[i] = currPath[i];

        }

        this.finalPath[this.n] = currPath[0];

      }

      tspRec(currBound, currCost, level, currPath) {

        if (level === this.n) {

          let lastToFirst = this.dist[currPath[level - 1]][currPath[0]];

          if (lastToFirst !== 0) {

            let res = currCost + lastToFirst;

            if (res < this.minCost) {

              this.copyToFinal(currPath);

              this.minCost = res;

            }

          }

          return;

        }

        for (let i = 0; i < this.n; i++) {

          if (!this.visited[i] && this.dist[currPath[level - 1]][i] !== 0) {

            let temp = currBound;

            currCost += this.dist[currPath[level - 1]][i];

            if (level === 1) {

              currBound -= ((this.firstMin(currPath[level - 1]) + this.firstMin(i)) / 2);

            } else {

              currBound -= ((this.secondMin(currPath[level - 1]) + this.firstMin(i)) / 2);

            }

            if (currBound + currCost < this.minCost) {

              currPath[level] = i;

              this.visited[i] = true;

              this.tspRec(currBound, currCost, level + 1, currPath);

            }

            currCost -= this.dist[currPath[level - 1]][i];

            currBound = temp;

            this.visited[i] = false;

          }

        }

      }

      solve() {

        let currBound = 0;

        let currPath = Array(this.n + 1).fill(-1);

        this.visited[0] = true;

        currPath[0] = 0;

        for (let i = 0; i < this.n; i++) {

          currBound += this.firstMin(i) + this.secondMin(i);

        }

        currBound = Math.floor(currBound / 2);

        this.tspRec(currBound, 0, 1, currPath);

        return { path: this.finalPath, cost: this.minCost };

      }

    }

    function bruteForceTSP(dist) {

      const n = dist.length;

      const cities = Array.from({ length: n }, (\_, i) => i);

      const permutations = getPermutations(cities.slice(1));

      let minCost = Infinity;

      let bestPath = [];

      for (const perm of permutations) {

        const path = [0, ...perm, 0];

        let cost = 0;

        for (let i = 0; i < path.length - 1; i++) {

          cost += dist[path[i]][path[i + 1]];

        }

        if (cost < minCost) {

          minCost = cost;

          bestPath = path;

        }

      }

      return { path: bestPath, cost: minCost };

    }

    function getPermutations(arr) {

      if (arr.length <= 1) return [arr];

      const result = [];

      for (let i = 0; i < arr.length; i++) {

        const rest = arr.slice(0, i).concat(arr.slice(i + 1));

        const perms = getPermutations(rest);

        for (const perm of perms) {

          result.push([arr[i], ...perm]);

        }

      }

      return result;

    }

    function drawPath(path) {

      const canvas = document.getElementById("canvas");

      const ctx = canvas.getContext("2d");

      ctx.clearRect(0, 0, canvas.width, canvas.height);

      ctx.font = "16px Arial";

      ctx.fillStyle = "red";

      for (let i = 0; i < cityCoords.length; i++) {

        const { x, y } = cityCoords[i];

        ctx.beginPath();

        ctx.arc(x, y, 6, 0, 2 \* Math.PI);

        ctx.fill();

        ctx.fillText(`City ${i}`, x + 10, y);

      }

      ctx.strokeStyle = "blue";

      ctx.lineWidth = 2;

      ctx.beginPath();

      ctx.moveTo(cityCoords[path[0]].x, cityCoords[path[0]].y);

      for (let i = 1; i < path.length; i++) {

        ctx.lineTo(cityCoords[path[i]].x, cityCoords[path[i]].y);

      }

      ctx.stroke();

    }

    function factorial(n) {

      return n <= 1 ? 1 : n \* factorial(n - 1);

    }

    function solveTSP() {

      const n = dist.length;

      const bnb = new TSPBranchAndBound(dist);

      const bnbStart = performance.now();

      const { path: bnbPath, cost: bnbCost } = bnb.solve();

      const bnbEnd = performance.now();

      const bruteStart = performance.now();

      const { path: bfPath, cost: bfCost } = bruteForceTSP(dist);

      const bruteEnd = performance.now();

      drawPath(bnbPath);

      document.getElementById("result").innerText =

        `Branch and Bound Path: ${bnbPath.join(" → ")}\nCost: ${bnbCost}`;

      document.getElementById("time-complexity").innerText =

        `Time Complexity:\nBranch and Bound: O(${factorial(n)})\nBrute Force: O(${factorial(n)})`;

      document.getElementById("space-complexity").innerText =

        `Space Complexity:\nBranch and Bound: O(${n})\nBrute Force: O(${n})`;

      document.getElementById("execution-time").innerText =

        `Execution Time:\nBranch and Bound: ${(bnbEnd - bnbStart).toFixed(2)} ms\nBrute Force: ${(bruteEnd - bruteStart).toFixed(2)} ms`;

      document.getElementById("comparison").innerHTML = `

        <ul>

          <li><strong>Brute Force:</strong> Tries all ${factorial(n)} paths.</li>

          <li><strong>Branch and Bound:</strong> Prunes unnecessary paths to optimize faster.</li>

          <li><strong>Result:</strong> Same cost (${bnbCost}) achieved in less time using B&B.</li>

        </ul>`;

    }

  </script>

</body>

</html>

**4.1 Input & Initialization**

* Cities are hardcoded with fixed coordinates for consistency.
* The UI includes a **"Run Brute Force"** and **"Run Branch and Bound"** button.
* Performance tracking starts at the beginning of algorithm execution.

**4.2 Divide & Compare**

* Brute Force:
  + Uses the next\_permutation() or recursive generation method.
  + Computes total cost for each route and keeps track of the minimum.
* Branch and Bound:
  + Computes **initial lower bound** by summing the smallest and second-smallest edge costs.
  + Recursively explores feasible paths, pruning paths that exceed current best.

**4.3 Recursive Detection**

* Brute Force:
  + Recursive depth reaches n, and every permutation is visited.
* Branch and Bound:
  + Recursive calls are cut short based on bounding value.
  + Achieves major performance gain even with small city sets.

**4.4 Visualization & Output**

* Canvas draws:
  + City nodes labeled with indices
  + Optimal path in a distinct color (e.g., red)
* Output shown in a separate HTML section:
  + Final path (e.g., 0 → 1 → 3 → 2 → 0)
  + Total distance/cost
  + Time taken (in milliseconds)
  + Complexity details

**5. RESULTS**

**5.1 Output Path & Cost**

* Brute Force and Branch and Bound both return the **same optimal path**.
* Example: 0 → 2 → 1 → 3 → 0 with cost = 128 units

**5.2 Execution Time Comparison**

* Brute Force: Slower, especially as cities increase (due to factorial growth).
* Branch and Bound: Much faster due to **intelligent pruning**.

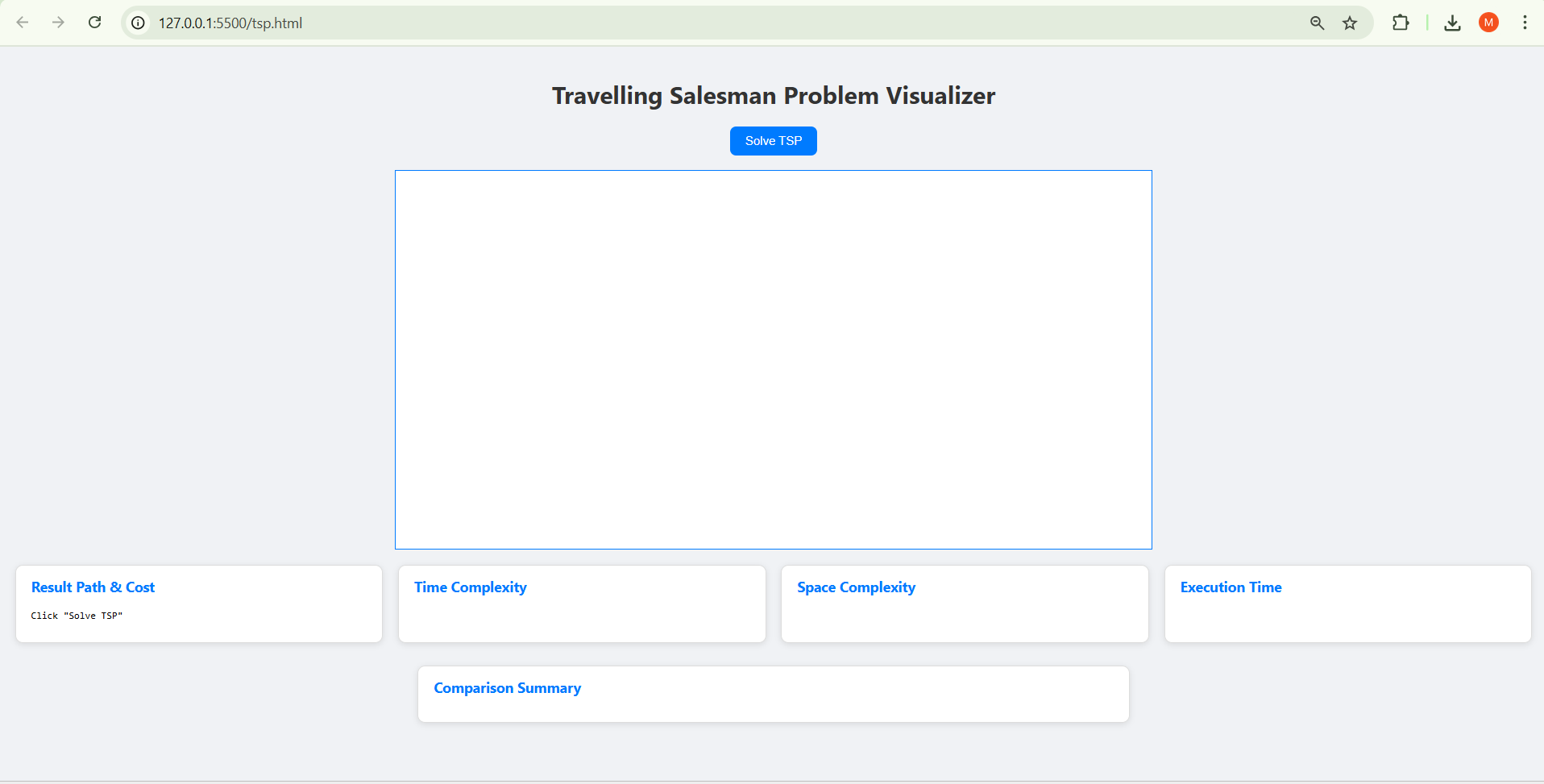
**5.3 Complexity Analysis**

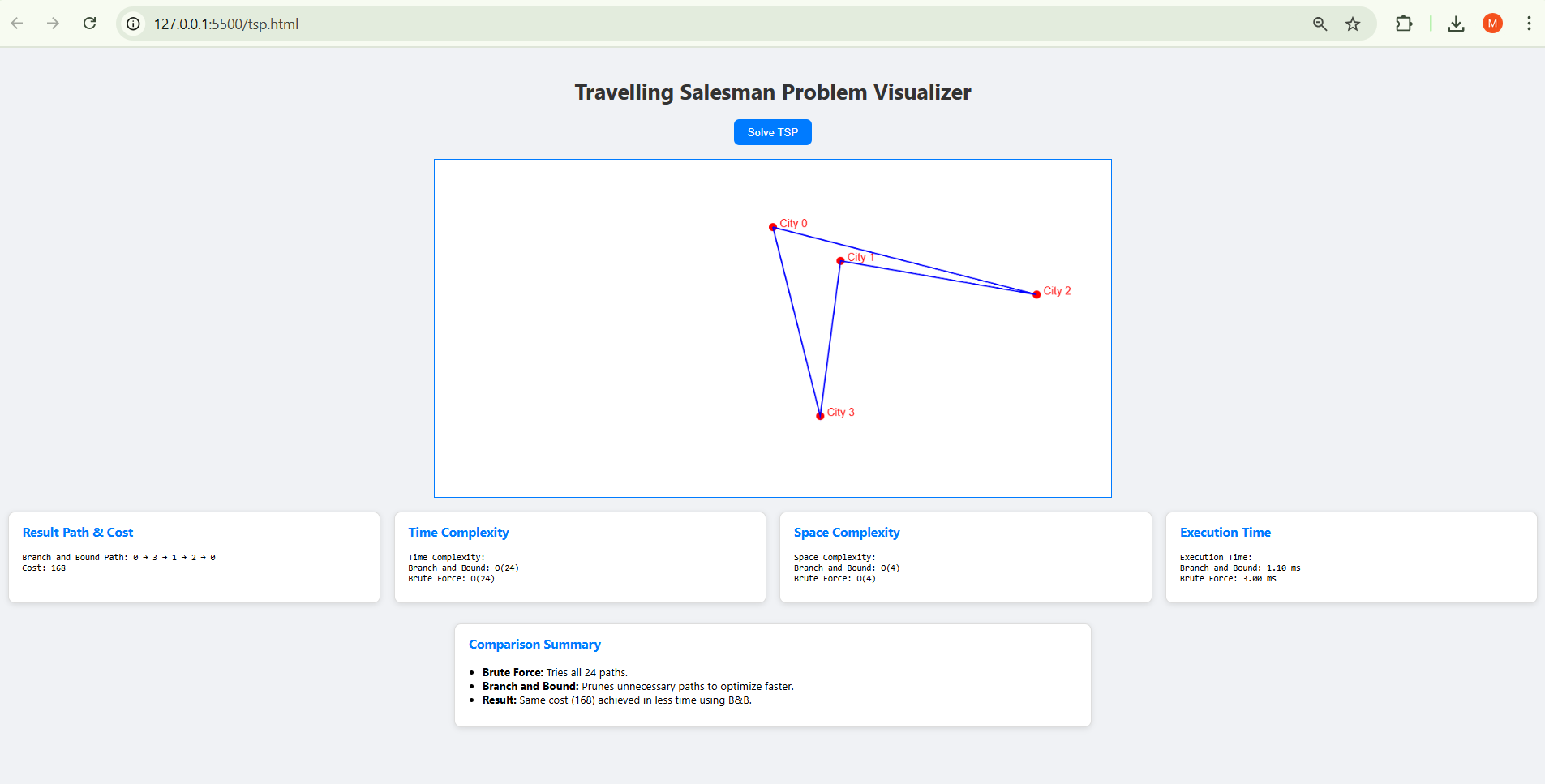
|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Space Complexity** |
| Brute Force | O(n!) | O(n) |
| Branch and Bound | O(n!) in worst case, but optimized | O(n) |

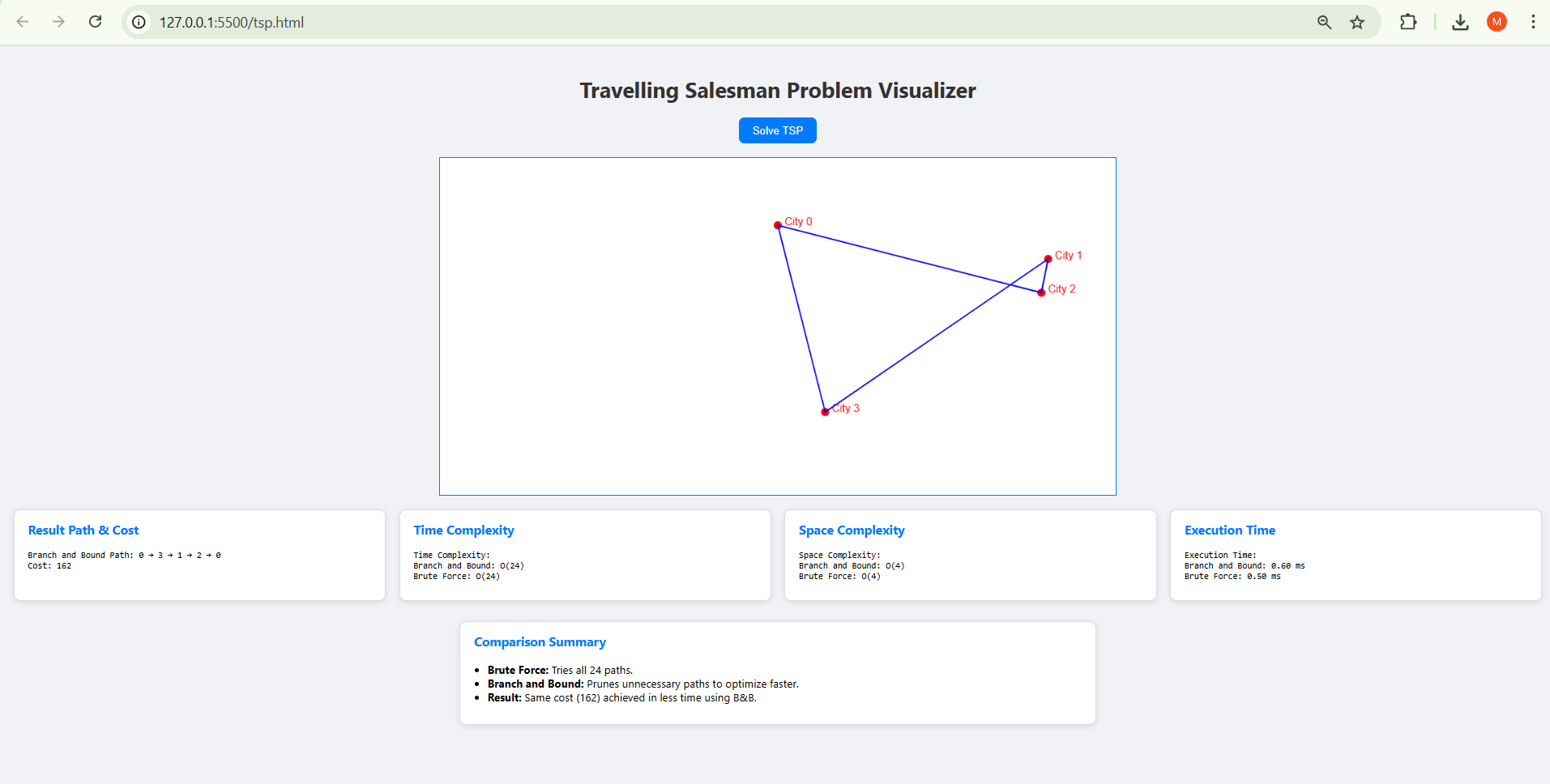
**5.4 Observations**

* Branch and Bound significantly reduces execution time.
* Both algorithms validate against the same dataset for fairness.
* The visualization helps understand how search trees are pruned in real time.

**OUTPUT:**







**TRAVELLING SALESMAN PROBLEM VISUALIZER**

**6. CONCLUSION**

This project showcases and compares two approaches to solving the Travelling Salesman Problem: **Brute Force** and **Branch and Bound**. While Brute Force guarantees accuracy, it becomes impractical for larger inputs due to high computational cost. In contrast, Branch and Bound improves efficiency through intelligent pruning. The web-based visualizer helps users understand the performance and logic behind each method, emphasizing the importance of algorithm optimization and visualization in tackling NP-hard problems.