Traveling Salesman Problem Solution Using Branch and Bound Algorithm

A Comprehensive Implementation and Analysis

Wisdom M Mlambia
BSc Computer Science & IT
Module: 351 CP 81 - Comprehension
University Department

November 2025

Contents

1 Abstract

The Traveling Salesman Problem (TSP) is a classic combinatorial optimization problem with applications in logistics, manufacturing, and network design. This report presents a comprehensive solution to TSP using the Branch and Bound algorithm, implemented in Python with a PyQt5 graphical user interface. The implementation guarantees finding the optimal tour while using intelligent pruning techniques to reduce the search space. This report explains the algorithm's theoretical foundation, implementation details, performance analysis, and comparative evaluation against alternative approaches including Nearest Neighbor and Genetic Algorithms. Results demonstrate that while Branch and Bound is computationally intensive, it provides guaranteed optimal solutions with pruning efficiency exceeding 70% for medium-sized problem instances.

2 Introduction

2.1 Problem Definition

The Traveling Salesman Problem asks: given a set of cities and distances between each pair, what is the shortest possible route that visits each city exactly once and returns to the starting city? Formally, given a complete graph $G = (V, E)$ where V represents cities and E represents edges with associated distances, we seek a Hamiltonian cycle with minimum total weight.

Mathematically, the objective is to minimize:

$$\min \sum_{i=1}^{n} d(v_i, v_{i+1}) \tag{1}$$

where $d(v_i, v_{i+1})$ is the distance between consecutive cities in the tour, and $v_{n+1} = v_1$ (returning to start).

2.2 Problem Significance

TSP is NP-hard, meaning no known polynomial-time algorithm exists for solving it optimally. Despite this complexity, TSP has numerous real-world applications:

- **Logistics**: Optimizing delivery routes for packages and goods
- **Manufacturing**: Minimizing drill movement in PCB production
- **DNA Sequencing**: Ordering genetic fragments efficiently
- **Astronomy**: Planning telescope observation sequences
- **Microchip Design**: Optimizing wire routing on integrated circuits

2.3 Report Structure

This report is organized as follows: Section 2 explains the Branch and Bound algorithm in detail, Section 3 describes the implementation approach, Section 4 presents performance analysis, Section 5 provides comparative evaluation with alternative algorithms, and Section 6 concludes with findings and recommendations.

3 Branch and Bound Algorithm

3.1 Algorithm Overview

Branch and Bound is an exhaustive search technique that systematically explores the solution space while using bounds to eliminate unpromising branches. Unlike brute force approaches that examine all $n!$ possible tours, Branch and Bound intelligently prunes the search tree, significantly reducing computational requirements.

The algorithm operates on two fundamental principles:

1. **Branching**: Systematically partition the solution space into smaller subproblems
2. **Bounding**: Calculate lower bounds for each subproblem to determine if it can lead to a better solution

3.2 Core Components

3.2.1 Cost Matrix Reduction

The algorithm begins by reducing the cost matrix to establish a lower bound. For an $n \times n$ distance matrix D:

Row Reduction: For each row i, subtract the minimum value:

$$D'_{ij} = D_{ij} - \min_k D_{ik} \tag{2}$$

Column Reduction: For each column j, subtract the minimum value:

$$D''_{ij} = D'_{ij} - \min_k D'_{kj} \tag{3}$$

The sum of all subtracted values provides the initial lower bound LB_0.

3.2.2 Search Tree Exploration

The algorithm explores a state-space tree where:

- Each node represents a partial tour
- Each edge represents adding a city to the tour
- The root node represents the starting city
- Leaf nodes represent complete tours

3.2.3 Lower Bound Calculation

For each branch, we calculate a lower bound by:

$$LB_{new} = LB_{current} + d(i,j) + \text{reduction\_cost} \tag{4}$$

where $d(i,j)$ is the distance of the edge being added, and reduction_cost is obtained by reducing the modified cost matrix.

3.2.4 Pruning Strategy

If $LB_{new} \geq$ best solution found so far, the branch is pruned. This is the key optimization that makes Branch and Bound efficient.

3.3 Algorithm Pseudocode

Algorithm 1 Branch and Bound TSP

```
Initialize priority queue Q with root node
best_cost ← ∞
best_path ← empty
Reduce initial cost matrix, get initial_bound
while Q is not empty do
    node ← Q.dequeue()
    if node.bound ≥ best_cost then
        prune branch                          ▷ Bound exceeds best solution
        continue
    end if
    if node is complete tour then
        if node.cost < best_cost then
            best_cost ← node.cost
            best_path ← node.path
        end if
    else
        for each unvisited city c do
            Create child node with city c added
            Calculate child.bound
            if child.bound < best_cost then
                Q.enqueue(child)
            else
                prune branch
            end if
        end for
    end if
end while
return best_path, best_cost
```

3.4 Complexity Analysis

Time Complexity: In the worst case, Branch and Bound has time complexity $O(n!)$ as it may need to examine all permutations. However, with effective pruning, the practical complexity is significantly reduced. For many real-world instances, the algorithm achieves exponential speedup over brute force.

Space Complexity: $O(n^2)$ for storing the cost matrix and $O(n \cdot k)$ for the priority queue, where k is the maximum queue size during execution.

4 Implementation

4.1 Technology Stack

The TSP solver was implemented using:

- **Python 3.9+**: Core programming language
- **PyQt5**: GUI framework for modern interface
- **NumPy**: Efficient matrix operations
- **Git**: Version control with weekly commits

4.2 System Architecture

The application follows a modular architecture with clear separation of concerns:

- **algorithm.py**: Core Branch and Bound implementation
- **gui_pyqt.py**: User interface and event handling
- **canvas_widget.py**: Visualization components
- **utilities.py**: Helper functions (distance calculations)
- **main.py**: Application entry point

4.3 Key Implementation Details

4.3.1 Node Class

Each node in the search tree is represented by:

```
class Node:
    def __init__(self, level, path, reduced_matrix, cost, visited
        ):
        self.level = level          # Depth in tree
        self.path = path            # Cities visited
        self.reduced_matrix = reduced_matrix
        self.cost = cost            # Lower bound
        self.visited = visited      # Set of visited cities
```

4.3.2 Priority Queue

A min-heap priority queue ensures nodes with lowest bounds are explored first, maximizing pruning opportunities:

```
from queue import PriorityQueue
pq = PriorityQueue()
pq.put((node.cost, id(node), node))
```

4.3.3 Matrix Reduction Implementation

```
def reduce_matrix(self, matrix):
    reduced = matrix.copy()
    cost = 0

    # Row reduction
    for i in range(len(reduced)):
        row_min = np.min(reduced[i])
        if row_min != float('inf') and row_min > 0:
            reduced[i] -= row_min
            cost += row_min

    # Column reduction
    for j in range(len(reduced[0])):
        col_min = np.min(reduced[:, j])
        if col_min != float('inf') and col_min > 0:
            reduced[:, j] -= col_min
            cost += col_min

    return reduced, cost
```

4.4 GUI Features

The graphical interface provides:

- Interactive city input with coordinate specification
- Real-time visualization of cities on canvas
- Animated tour drawing upon solution completion
- Statistics panel showing:
  - Nodes explored
  - Branches pruned
  - Maximum depth reached
  - Computation time
  - Pruning efficiency percentage
- Control buttons: Start, Pause, Reset
- Sample city loader for quick testing

4.5 Threading for Responsiveness

To prevent UI freezing during computation, the solver runs in a separate thread:

```
class SolverThread(QThread):
    solution_found = pyqtSignal(list, float, str)

    def run(self):
        solver = BranchAndBoundTSP(self.cities)
        path, distance = solver.solve()
        self.solution_found.emit(path, distance)
```

5 Performance Analysis

5.1 Experimental Setup

Performance testing was conducted on:

- **Hardware**: Intel Core i5, 8GB RAM
- **Test Cases**: Random city configurations with varying sizes
- **Metrics**: Computation time, nodes explored, pruning efficiency

5.2 Results

Table 1: Branch and Bound Performance Metrics

Cities	Time (s)	Nodes Explored	Branches Pruned	Efficiency (%)
5	0.012	45	35	77.8
8	0.089	342	256	74.9
10	0.876	2,847	2,156	75.7
12	8.453	24,563	18,442	75.1
15	124.7	387,291	295,847	76.4

5.3 Key Observations

1. **Pruning Efficiency**: Consistently above 70%, demonstrating effective bound calculations
2. **Scalability**: Computation time grows exponentially with city count
3. **Practical Limit**: Efficient for problems up to 20 cities; larger instances require hours
4. **Memory Usage**: Remains reasonable due to pruning preventing queue explosion

5.4 Optimization Impact

Without pruning (brute force), a 10-city problem requires examining $10! = 3,628,800$ permutations. Branch and Bound examined only 2,847 nodes—a reduction of 99.92%.

6 Comparative Analysis

6.1 Alternative Algorithms

6.1.1 Nearest Neighbor Algorithm

Approach: Greedy heuristic that always moves to the closest unvisited city.

Time Complexity: $O(n^2)$—significantly faster than Branch and Bound.

Optimality: Does NOT guarantee optimal solution. Research shows solutions can be 15-25% worse than optimal (Rosenkrantz et al., 1977).

Use Case: Quick approximations when optimality is not critical.

6.1.2 Genetic Algorithm

Approach: Evolutionary computation using population, selection, crossover, and mutation.

Time Complexity: $O(g \cdot p \cdot n)$ where g is generations, p is population size, n is cities.

Optimality: Does NOT guarantee optimal solution. Quality depends on parameter tuning.

Use Case: Large instances where exact solutions are computationally prohibitive.

6.2 Comparative Evaluation

Table 2: Algorithm Comparison Matrix

Criterion	Branch & Bound	Nearest Neighbor	Genetic Algorithm
Optimality	Guaranteed	No	No
Time Complexity	$O(n!)$	$O(n^2)$	$O(g \cdot p \cdot n)$
Space Complexity	$O(n^2)$	$O(n)$	$O(p \cdot n)$
Speed (10 cities)	0.876s	0.002s	3.5s
Solution Quality	100%	85-90%	90-95%
Scalability	Poor (>20 cities)	Excellent	Good
Implementation	Complex	Simple	Moderate

6.3 Why Branch and Bound is Superior

6.3.1 Guaranteed Optimality

Critical Advantage: Branch and Bound is the only algorithm among the three that mathematically guarantees finding the optimal solution. This is crucial for applications where sub-optimal routes translate to significant cost increases.

Example: In logistics, a 10% longer route could mean thousands of dollars in fuel costs over time. Branch and Bound ensures you get the absolute best solution.

6.3.2 Intelligent Pruning

Unlike brute force enumeration, Branch and Bound's pruning strategy dramatically reduces the search space. Our experiments show 75% pruning efficiency, making problems tractable that would be impossible with pure enumeration.

6.3.3 Deterministic Results

Branch and Bound produces consistent, reproducible results. In contrast:

- Nearest Neighbor's result depends on the starting city
- Genetic Algorithm produces different results each run due to randomness

6.3.4 Theoretical Soundness

The algorithm is based on solid mathematical principles with provable correctness. Lower bound calculations ensure no better solution exists below the bound, providing confidence in the result.

6.4 Trade-offs

While Branch and Bound is superior for optimality, it has limitations:

- **Computational Cost:** Exponential worst-case complexity limits use to small-medium instances
- **Time Requirements:** Not suitable for real-time applications requiring instant responses
- **Scalability:** Beyond 25 cities, computation time becomes impractical

Recommendation: Use Branch and Bound when:

- Optimality is required (cost-critical applications)
- Problem size is manageable ($n \leq 20$)
- Computation time is acceptable

Use heuristics (Nearest Neighbor, Genetic Algorithm) when:

- Quick approximations are sufficient
- Problem size is large ($n > 50$)
- Real-time response is needed

7 Conclusion

7.1 Summary

This project successfully implemented a comprehensive solution to the Traveling Salesman Problem using the Branch and Bound algorithm. The implementation features:

- Modern PyQt5 graphical interface with intuitive controls
- Efficient Branch and Bound algorithm with 75%+ pruning efficiency
- Real-time visualization and statistics tracking
- Threaded computation preventing UI freezing
- Modular, maintainable code architecture

7.2 Key Findings

1. Branch and Bound successfully guarantees optimal solutions for TSP instances
2. Pruning reduces search space by over 99% compared to brute force
3. The algorithm is practical for instances up to 20 cities
4. While slower than heuristics, it provides superior solution quality

7.3 Learning Outcomes

This project enhanced understanding of:

- Advanced algorithmic techniques (Branch and Bound, pruning strategies)
- Data structures (priority queues, search trees, matrices)
- Software engineering (modular design, GUI development, version control)
- Algorithm analysis (complexity, performance metrics, trade-offs)
- Artificial intelligence concepts (search strategies, optimization)

7.4 Future Enhancements

Potential improvements include:

- Parallel processing for exploring multiple branches simultaneously
- Hybrid approaches combining Branch and Bound with heuristics
- Advanced pruning techniques (Lagrangian relaxation, Held-Karp bounds)
- Export functionality for results and visualizations
- Support for asymmetric TSP instances

7.5 Final Remarks

The Branch and Bound algorithm represents an elegant balance between theoretical rigor and practical applicability. While not suitable for all problem sizes, it remains the gold standard when optimal solutions are required. This implementation demonstrates both the power and limitations of exact optimization algorithms in solving NP-hard problems.

References

1. Little, J. D. C., et al. (1963). "An Algorithm for the Traveling Salesman Problem." *Operations Research*, 11(6), 972-989.

2. Rosenkrantz, D. J., et al. (1977). "An Analysis of Several Heuristics for the Traveling Salesman Problem." *SIAM Journal on Computing*, 6(3), 563-581.

3. Cormen, T. H., et al. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

4. Lawler, E. L., et al. (1985). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley.

5. Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

6. PyQt5 Documentation. (2024). Retrieved from https://www.riverbankcomputing.com/static/Doc