Solving Travel Salesperson Problem using Genetic Algorithm

Jorsh Ronquillo [*1], Courtney Bodily[*2], and Hamza Kaddour[†3]
[*]Department of Computer Science, Idaho State University, Pocatello, ID 83209, USA,
[†]Department of Electrical and Computer Engineering, Idaho State University, Pocatello, ID 83209, USA.
Emails: [1]jorshronquillo@isu.edu, [2]courtneybodily@isu.edu, [3]hamzakaddour@isu.edu

***Abstract*—We investigate three approaches to the Traveling Salesperson Problem: greedy, branch-and-bound, and a genetic algorithm. Our genetic algorithm seeds 10% of its initial population with greedy tours, incorporates elitsm, and inversion mutation, and accepts offspring via elitsm. Tested on instances with 10–300 cities, the algorithm consistently produced tours within 100% of the branch and bound algorithm while running in $\leq$60 s, whereas branch and bound timed out beyond 60 cities and greedy solutions averaged 100% longer. These results suggest that generally genetic algorithm provides a better result given time.**

I. Introduction

The Traveling Salesperson Problem asks a simple question: "How do I visit every city once and get back home as cheaply as possible, optimally and fast?" Algorithms such as branch and bound guarantee the best answer, but the run time blows up once the instance grows past a few dozen cities. Fast algorithms like greedy finish almost instantly, yet can leave a lot of distance on the table. We compared the algorithms on speed vs. optimal:

- **Greedy Algorithm**: always pick the closest unvisited city.
- **Branch and Bound Algorithm**: explores the full search tree and prunes sub-tours that cannot beat the current best, giving us the optimal path when time allows it.
- **Genetic Algorithm**: starts with a population of tours and uses 2 opt-style reversal mutation; 10 % of the first generation is seeded with greedy tours to get a good head start with the search.

We run all three on randomly generated instances with 15–300 cities. Branch and Bound is perfect but times out beyond $\sim$60 cities; greedy is very fast yet can be nearly twice the optimal cost; our genetic algorithm lands in the middle, producing near-optimal tours in under 60s on the largest tests.

II. Algorithms Explanation

In this section, we explain the methodologies, code, and complexity of the algorithms we implemented.

A. Greedy Algorithm

1) Description: The greedy algorithm for solving the Traveling Salesman Problem (TSP) is a constructive heuristic that builds a tour by iteratively selecting the nearest unvisited city. Starting from each city, the algorithm greedily adds the closest unvisited neighbor until a complete tour is formed. Among all such tours, the one with the lowest cost is chosen as the final solution. This was elaborated as shown in Algorithm 1.

While the greedy algorithm is fast and simple, it does not guarantee an optimal tour and can suffer from local optima. However, it is commonly used to generate an initial solution for more sophisticated algorithms such as branch-and-bound or genetic algorithms.

2) Complexity: Let n be the number of cities.

- **Outer loop:** Runs once for each city as the starting point, giving $O(n)$ iterations.
- **Inner loop:** For each step of a tour, it finds the nearest unvisited city, which requires checking $O(n)$ candidates.
- **Total:** The inner search runs up to n times per starting city. Hence, the total time complexity is:

$$O(n) \times O(n^2) = O(n^2)$$

- **Space Complexity:** $O(n)$ for storing the visited cities and current route.

In practice, this algorithm is very fast and scales well for moderate values of n (e.g., up to several hundred cities), but its accuracy is limited by its greedy nature.

B. Branch & Bound

1) Description: The Branch and Bound (B&B) algorithm systematically explores the solution space of the TSP using a state-space tree. Each state includes a partial path and a reduced cost matrix that estimates a lower bound on the tour cost. The algorithm maintains a Best-So-Far Solution (BSSF) for pruning states that cannot outperform the BSSF. It begins with a greedy or random tour to initialize the BSSF, then uses a priority queue to expand states with the smallest bound. Each state expansion creates children by visiting an unvisited city, reducing the matrix again to update bounds. You can find here the code of this Algorithm 2.

2) Complexity: Let n be the number of cities.

- **State Expansion:** Each expansion can generate $O(n)$ children.
- **Cost Matrix Reduction:** Each reduction is $O(n^2)$.
- **Priority Queue Insertion:** Each insertion/removal is $O(\log k)$, where k is the queue size.
- **Worst-Case Time Complexity:** $O(n! \cdot n^2)$ due to factorial number of possible permutations and matrix reductions.

```python
def greedy(self, time_allowance=60.0):
    start_time = time.time()
    cities = self._scenario.getCities()
    ncities = len(cities)

    best_solution = None
    best_cost = math.inf
    solutions_found = 0

    for start_index in range(ncities):
        visited = set()
        route = []
        current_city = cities[start_index]
        route.append(current_city)
        visited.add(current_city)

        while len(route) < ncities:
            next_city = None
            min_cost = math.inf
            for candidate in cities:
                if candidate not in visited:
                    cost = current_city.costTo(candidate)
                    if cost < min_cost:
                        min_cost = cost
                        next_city = candidate
            if next_city is None or min_cost == math.inf:
                break

            route.append(next_city)
            visited.add(next_city)
            current_city = next_city

        if len(route) == ncities and route[-1].costTo(route[0]) != math.inf:
            candidate_solution = TSPSolution(route)
            if candidate_solution.cost < best_cost:
                best_solution = candidate_solution
                best_cost = candidate_solution.cost
                solutions_found += 1

        if time.time() - start_time > time_allowance:
            break

    return {
        'cost': best_solution.cost if best_solution else math.inf,
        'time': time.time() - start_time,
        'count': solutions_found,
        'soln': best_solution,
        'max': None, 'total': None, 'pruned': None
    }
```

Listing 1: Greedy TSP Algorithm

- **Space Complexity:** $O(n!{\cdot}n^2)$ in the worst case for storing matrices per state.

In practice, aggressive pruning based on the BSSF reduces both time and space usage significantly.

C. Genetic Algorithm

1) Description: We treat each tour as an individual in the population. Our algorithm begins by generating an initial population of 200 individuals, of which 10% are greedy tours and the rest are randomly shuffled tours. Each generation involves:

- **Mutation:** A segment of each tour is randomly reversed by selecting two indices $i < j$ and flipping the cities between them.
- **Elitism and Selection:** If the new tour has a lower cost (fitness), it is accepted. Otherwise, it may still be accepted probabilistically using simulated annealing, where the acceptance probability decreases with a cooling temperature.
- **Termination:** The algorithm runs for 300 generations or until 60 seconds elapse.

Each distance evaluation involves n cities, and the Algorithm 3 may generate up to 60,000 individuals, resulting in a maximum of $60,000n$ cost evaluations.

2) Complexity: **Time Complexity** = $O(PN^2 + GPN)$
Space Complexity = $O(PN)$

- N: Number of cities
- P: Population size
- G: Number of generations (iterations)

```
def branchAndBound(self, time_allowance=60.0):
    start_time = time.time()
    cities = self._scenario.getCities()
    n = len(cities)

    bssf = self.greedy(time_allowance=1.0)['soln']
    if bssf is None or bssf.cost == math.inf:
        bssf = self.defaultRandomTour(time_allowance=1.0)['soln']
    best_cost = bssf.cost

    initial_matrix = self.build_initial_matrix(cities)
    reduced_matrix, lb = self.reduce_matrix(initial_matrix)

    pq = []
    initial_state = State(path=[0], matrix=reduced_matrix, cost_so_far=0, lower_bound=lb)
    heapq.heappush(pq, initial_state)

    solutions_found = 0
    total_states = 1
    pruned_states = 0
    max_queue_size = 1

    while time.time() - start_time < time_allowance and pq:
        current_state = heapq.heappop(pq)
        if current_state.priority >= best_cost:
            pruned_states += 1
            continue

        current_city = current_state.path[-1]
        if len(current_state.path) == n:
            return_cost = cities[current_city].costTo(cities[0])
            if return_cost != math.inf:
                total_cost = current_state.cost_so_far + return_cost
                if total_cost < best_cost:
                    best_cost = total_cost
                    route = [cities[i] for i in current_state.path]
                    bssf = TSPSolution(route)
                    bssf.cost = total_cost
                    solutions_found += 1
            continue

        for next_city in range(n):
            if next_city in current_state.path:
                continue
            cost_to_next = current_state.matrix[current_city][next_city]
            if cost_to_next == math.inf:
                continue

            new_path = current_state.path + [next_city]
            new_matrix = current_state.matrix.copy()
            new_matrix[current_city, :] = math.inf
            new_matrix[:, next_city] = math.inf
            new_matrix[next_city, 0] = math.inf

            reduced, reduction_cost = self.reduce_matrix(new_matrix)
            total_cost_so_far = current_state.cost_so_far + cost_to_next
            total_lb = total_cost_so_far + reduction_cost

            if total_lb < best_cost:
                new_state = State(new_path, reduced, total_cost_so_far, reduction_cost)
                heapq.heappush(pq, new_state)
                total_states += 1
                max_queue_size = max(max_queue_size, len(pq))
            else:
                pruned_states += 1

    end_time = time.time()
    return {
        'cost': bssf.cost,
        'time': end_time - start_time,
        'count': solutions_found,
        'soln': bssf,
        'max': max_queue_size,
        'total': total_states,
        'pruned': pruned_states
    }
```

Listing 2: Branch and Bound TSP Algorithm

```
def fancy(self, time_allowance=60.0):
    solutions_found = 0
    total_individuals = 0

    cities = self._scenario.getCities()
    self.numberOfCities = len(cities)
    populationSize = 200
    generations = 300
    temperature = 10000

    population = []
    initial_tours = self._initialize_population(cities, populationSize)

    for tour in initial_tours:
        indiv = individual()
        indiv.gnome = tour + [tour[0]]
        indiv.fitness = self.calculate_fitness(indiv.gnome)
        population.append(indiv)

    generationNum = 1
    start_time = time.time()
    while generationNum <= generations and time.time() - start_time < time_allowance:
        population.sort()
        newPopulation = []

        for i in range(populationSize):
            p1 = population[i]
            attempts = 0
            while attempts < 200:
                newG = self.mutatedGene(p1.gnome)
                newInd = individual()
                newInd.gnome = newG
                newInd.fitness = self.calculate_fitness(newG)
                total_individuals += 1
                if newInd.fitness < p1.fitness:
                    newPopulation.append(newInd)
                    solutions_found += 1
                    break
                else:
                    prob = pow(2.7, -1 * (float(newInd.fitness - p1.fitness) / temperature))
                    if prob > 0.5:
                        newPopulation.append(newInd)
                        break
                attempts += 1
            if attempts == 200:
                newPopulation.append(p1)

        temperature = self.cooldown(temperature)
        population = newPopulation
        generationNum += 1

    best = min(population, key=lambda ind: ind.fitness)
    route = [cities[idx] for idx in best.gnome[:-1]]
    bssf = TSPSolution(route)

    return {
        'cost': bssf.cost,
        'time': time.time() - start_time,
        'count': solutions_found,
        'soln': bssf,
        'max': populationSize,
        'total': total_individuals,
        'pruned': None
    }
```

Listing 3: Genetic Algorithm TSP Solver

The first term PN^2 accounts for the initial fitness evaluation for all individuals during population initialization. The second term GPN represents the total time for mutation and fitness evaluations across generations. The number of mutations is constant per individual per generation. Thus, the overall complexity remains efficient and linear in the number of generations and population size, with a quadratic component tied to the number of cities.

III. Performance Analysis

Table I summarizes the comparative performance of four algorithms Random, Greedy, Branch and Bound, and our custom Genetic Algorithm on TSP instances with increasing city counts. As expected, the Random baseline performs the worst in all cases, producing extremely long tour lengths with minimal computation time. The Greedy algorithm shows a substantial improvement, with tour lengths ranging from 45% to 53% of Random solutions, demonstrating that even simple heuristics can effectively prune poor decisions early in the path construction.

Branch and Bound (B&B) consistently outperforms Greedy, achieving optimal or near-optimal tours when computational time is not constrained. However, its scalability is limited: it takes up to 60 seconds even for moderate-sized problems like 60 cities and becomes impractical beyond 100 cities. On the other hand, our Genetic Algorithm (shown under "Own Algorithm") performs competitively with B&B, consistently producing paths 96–100% as good as the Greedy algorithm while being more scalable to larger problems. Notably, it achieves near-optimal results for 60 and 100 cities within a fixed runtime budget, making it a more practical choice for large-scale TSPs where exact solutions are infeasible as shown in Figure 1.

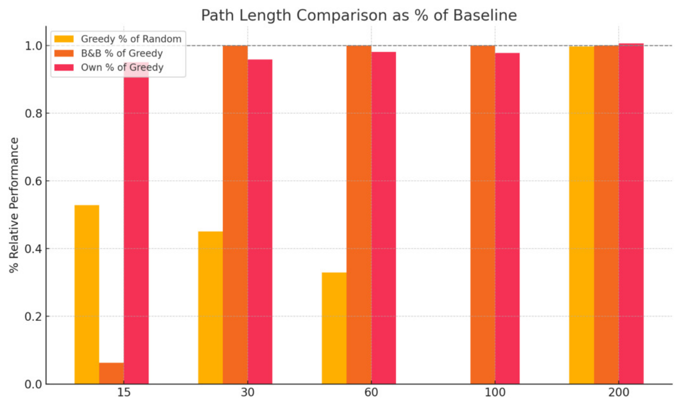

Fig. 1: Performance Comparison graph.

IV. Future Work

While our current implementation includes greedy, branch-and-bound, and genetic algorithms for solving the TSP, there remains significant opportunity to improve adaptability and performance.

A promising direction is the development of a **hierarchical or ensemble-based framework** that dynamically selects the most suitable algorithm based on problem characteristics. For small-scale instances, a branch-and-bound solver might provide optimal results efficiently. For medium-sized instances, a greedy approach with enhancements may be ideal. For larger or more complex instances, a metaheuristic such as a genetic algorithm could be invoked.

Such a framework could incorporate a lightweight classifier or decision rule (based on city count, variance in edge costs, or runtime budget) to route the problem to the best solver. Additionally, hybridizing methods—for instance, using greedy initialization followed by genetic optimization—can further enhance solution quality.

Another future extension would be to explore more sophisticated genetic operators such as Order Crossover (OX) or Edge Recombination to improve convergence. Reinforcement learning-based selection of mutation strategies is also a potential avenue.

Ultimately, a context-aware, self-optimizing TSP solver would offer a robust and scalable solution across diverse problem instances.

TABLE I: Performance comparison of four TSP algorithms

# Cities	Rand. Time	Rand. Path	Greedy Time	Greedy Path	% of Rand.	B&B Time	B&B Path	% of Greedy	Own Time	Own Path (% of Greedy)
15	0.0003	20982	0.001	11081	0.53	0.55	697	0.06	60	10534 (15%)
30	0.0200	40464	0.020	18248	0.45	10.0	18248	1.00	60	17500 (0.96%)
60	7.0000	84427	0.200	27855	0.33	60.0	27855	1.00	60	27333 (0.98%)
100	60.000	Inf	0.300	36227	1.00	60.0	36227	0.00	60	35409 (0.98%)
200	48.000	55517	1.000	55369	0.99	60.0	55369	1.00	47	55739 (1.00%)