CECS 451-01 AI

Solving and Visualizing Traveling Salesman Problem with Hill Climbing and Simulated Annealing Heuristic Algorithms

By: Darius Koroni, Jett Sonoda, Bryan Tran, Tien Nguyen, Nhi Pham

California State University Long Beach, Spring 2023

I. Abstract

This study aimed to compare the performance of Hill Climbing and Simulated Annealing as heuristic algorithms for solving the Traveling Salesman Problem (TSP). The experiments were conducted on TSP instances of various sizes, ranging from small to large. The results of the study showed that for larger TSP instances (n = 300), Simulated Annealing outperformed Hill Climbing in terms of solution quality, achieving a 10% improvement over Hill Climbing. However, for smaller TSP instances (n = 50, 100), Hill Climbing was both more accurate and faster than Simulated Annealing.

The study concluded that Hill Climbing is a suitable approach for TSP instances with few or no local minima, as it can quickly converge to the global minimum. However, for TSP instances with complex city layouts that have many local minima, Simulated Annealing is more appropriate, as it can explore the search space more thoroughly and escape local optima. Therefore, the choice of algorithm depends on the specific problem instance and its characteristics, of which city size is the largest determining factor.

Group 10: Darius Koroni, Jett Sonoda, Bryan Tran, Tien Nguyen, Nhi Pham

II. Introduction

Our assignment for the final project was to implement the Traveling Salesman Problem, which aims to find the shortest possible route that visits every city exactly once and returns to the starting location, given a set of cities and the distance between every pair of cities. We implemented two solutions to this problem using Simulated Annealing and Hill Climbing algorithms. We then compared the results from both to determine which algorithm is better in solving the Travelling Salesperson Problem for varying amounts of cities (n = 50, 100, 150, 200, 250, 300).

III. Background

The Traveling Salesman Problem is a problem first considered in 1930 and is one of the most studied problems in computer science, so much so that it serves as a benchmark for many optimization methods. A large part of its importance derives from its classification as an NP-hard problem: a problem which is at least as hard as the hardest problems in NP, a class of problems which can be verified in polynomial time but for which no algorithm has been discovered which can solve them in polynomial time. Should a polynomial-time solution exist for TSP, it would prove that P = NP and be a major breakthrough in computational complexity theory. However in this study we are only looking at heuristics for finding an estimated solution for TSP and not arriving at a deterministic solution.

Group 10: Darius Koroni, Jett Sonoda, Bryan Tran, Tien Nguyen, Nhi Pham

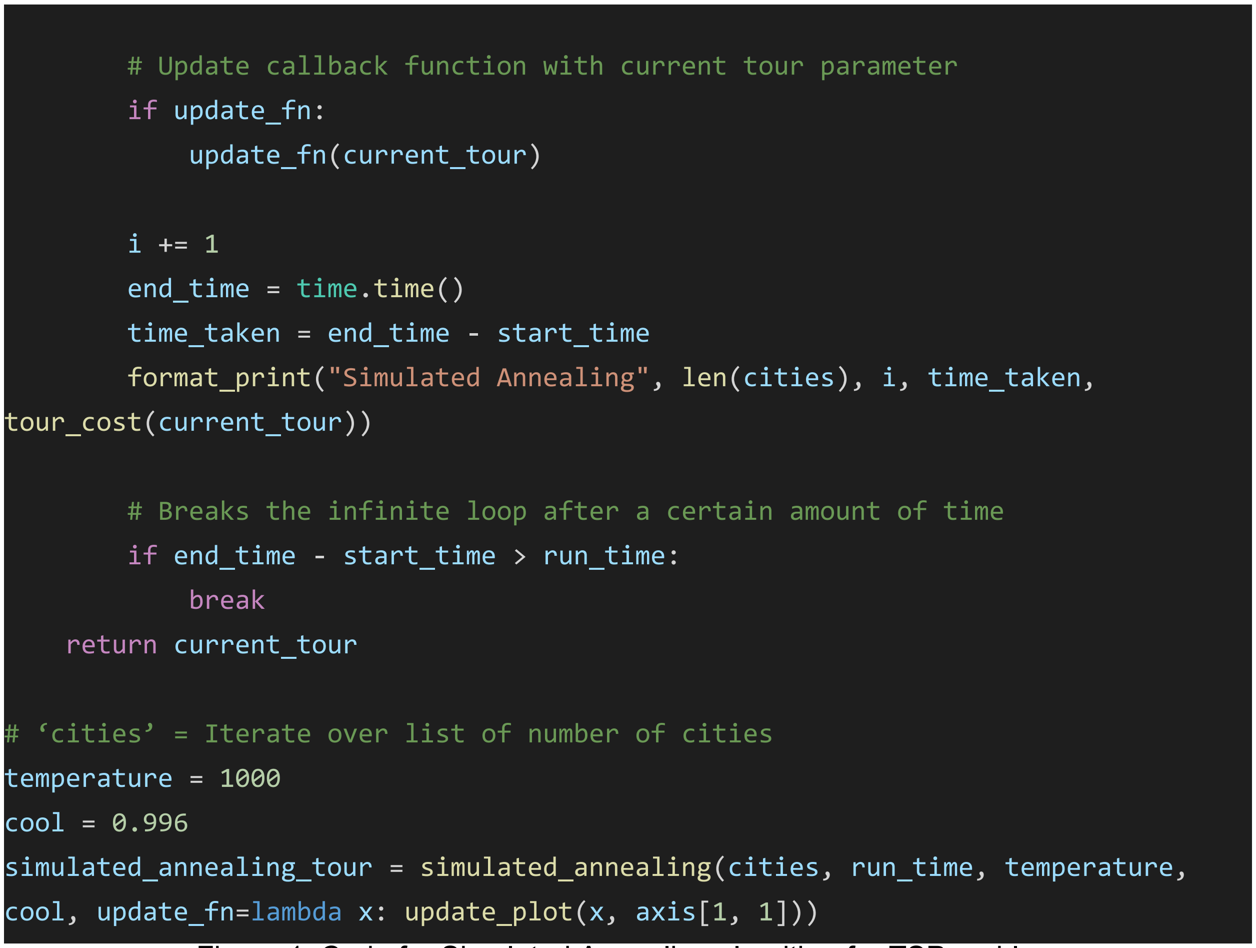
IV. Simulated Annealing

Simulated annealing is a heuristic algorithm which starts by creating a random initial tour and an alternative tour by randomly swapping two cities. The cost of the current tour and the alternative tour are compared to determine if we should accept the new solution. If the alternative tour reduces the length of the current tour, it is always accepted; if not, it may be accepted with a particular probability dependent upon a temperature parameter that declines with each new iteration. The temperature parameter is used to control the chance we accept the worse solution. The algorithm concentrates more on exploring the present solution as the temperature drops, which is conventionally proportional to the average cost of formulating a new move or in this case, a tour.

```python
# The Simulated Annealing algorithm
def simulated_annealing(cities, run_time, temperature, cool, update_fn=None):
    global time_taken
    # Generate a random tour
    current_tour = random.sample(cities, len(cities))
    # Loop through for a specific amount of time
    start_time = time.time()

    i = 0
    format_print("Simulated Annealing", len(cities), i, 0,
tour_cost(current_tour))
    while temperature > 1:
        # Create a neighbor tour by swapping two cities
        neighbor = current_tour.copy()
        j, k = random.sample(range(len(cities)), 2)
        neighbor[j], neighbor[k] = neighbor[k], neighbor[j]
        # Calculate the cost of the current and neighbor tours
        current_cost = tour_cost(current_tour)
        neighbor_cost = tour_cost(neighbor)

        # Decrease the temperature to lower risk of next iteration
        temperature *= cool
        # Determine whether to accept the neighbor tour
        cost = neighbor_cost - current_cost
        probability = math.exp(cost*-1 / temperature)
        if neighbor_cost < current_cost or probability > random.uniform(0, 1):
            current_tour = neighbor
```



```
        # Update callback function with current tour parameter
        if update_fn:
            update_fn(current_tour)

        i += 1
        end_time = time.time()
        time_taken = end_time - start_time
        format_print("Simulated Annealing", len(cities), i, time_taken,
tour_cost(current_tour))

        # Breaks the infinite loop after a certain amount of time
        if end_time - start_time > run_time:
            break
    return current_tour

# ‘cities’ = Iterate over list of number of cities
temperature = 1000
cool = 0.996
simulated_annealing_tour = simulated_annealing(cities, run_time, temperature,
cool, update_fn=lambda x: update_plot(x, axis[1, 1]))
```

Figure 1: Code for Simulated Annealing algorithm for TSP problem

V. Hill Climbing

Hill climbing is a greedy algorithm which initially takes a similar approach to simulated annealing in regards to its implementation. It starts with randomly generating an initial tour and swapping two cities in the tour to obtain a new possible solution. After that, it compares the cost of the initial tour to the alternative tour and accepts whichever has the shorter tour length. It continues until it is no longer able to find a swap which generates a lower tour cost. The key difference between hill climbing and simulated annealing is that hill climbing always chooses the locally optimal choice and does not have a way to consider the globally optimal solution, unlike simulated annealing does with its temperature parameter. Therefore, the hill climbing algorithm is prone to be stuck in a local optima while simulated annealing can escape local optima and converge toward global optima.

```
# The Hill Climbing Algorithm
def hill_climb(cities, run_time, update_fn=None):
    global time_taken
    # Generate a random tour
    current_tour = random.sample(cities, len(cities))
    # Loop through for a specific amount of time
    start_time = time.time()

    i = 0
    format_print("Hill Climbing", len(cities), i, 0, tour_cost(current_tour))
    while True:
        # Create a list of neighboring tours
        neighbors = []
        for j in range(len(cities)):
            for k in range(j+1, len(cities)):
                neighbor = current_tour.copy()
                neighbor[j], neighbor[k] = neighbor[k], neighbor[j]
                neighbors.append(neighbor)
        last_tour_cost = tour_cost(current_tour)
        # Evaluate the neighbors and select the best one
        best_neighbor = min(neighbors, key=lambda x: tour_cost(x))
        if tour_cost(best_neighbor) < tour_cost(current_tour):
            current_tour = best_neighbor

        # Update callback function with current tour parameter
        if update_fn:
            update_fn(current_tour)

        i += 1
        end_time = time.time()
        time_taken = end_time - start_time
        format_print("Hill Climbing", len(cities), i, time_taken,
tour_cost(current_tour))

        # Hill climbing is prone to getting stuck in local minima
        # We end the loop early to save time in these cases
        if last_tour_cost == tour_cost(current_tour):
            break

        # Breaks the infinite loop after a certain amount of time
        if end_time - start_time > run_time:
            break
    return current_tour

# 'cities' = Iterate over list of number of cities
hill_climb_tour = hill_climb(cities, run_time, update_fn=lambda x: update_plot(x,
axis[0, 1]))
```

Figure 1: Code for Hill Climbing algorithm for TSP problem

To better improve our hill climbing and simulated annealing implementations, a timer is added to stop the algorithms at the last iteration step in order to give both functions a fair amount of time to run. The hill climbing algorithm finds the local minimum relatively quick when provided with low numbers of cities. Hence, a break within its loop that stops the algorithm by comparing the last tour cost of one iteration over the next tour cost of the next iteration to see if they are equal. By implementing these methods, the amount of runtime of both algorithms is reduced by removing unnecessary and redundant iteration cycles.

VI. Results

Here are the configurations for the max runtime in seconds and the list of number of cities to run our algorithms.

```python
# Variables to configure
# List of number of cities to run
num_cities_list = [50, 100, 150, 200, 250, 300]
run_time = 60
```

Figure 3: Code for Initial Variables used

The results of running our algorithms for each number of cities in the list are shown below:

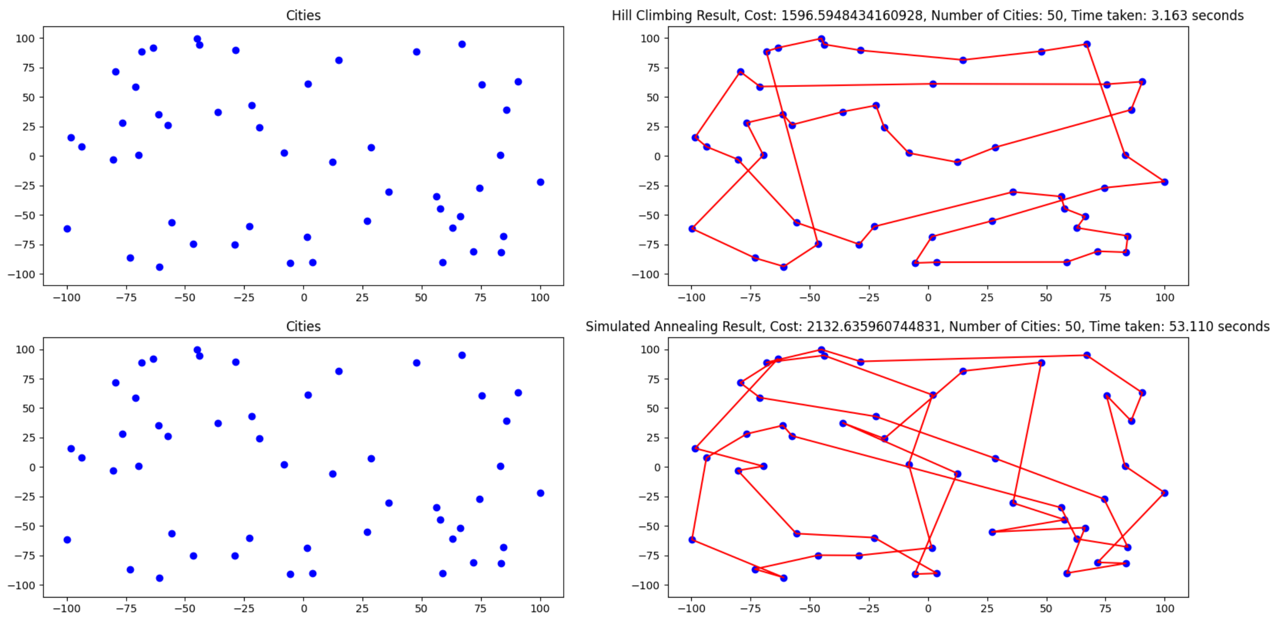


Figure 4: Tour cost visualizations for 50 cities

Group 10: Darius Koroni, Jett Sonoda, Bryan Tran, Tien Nguyen, Nhi Pham

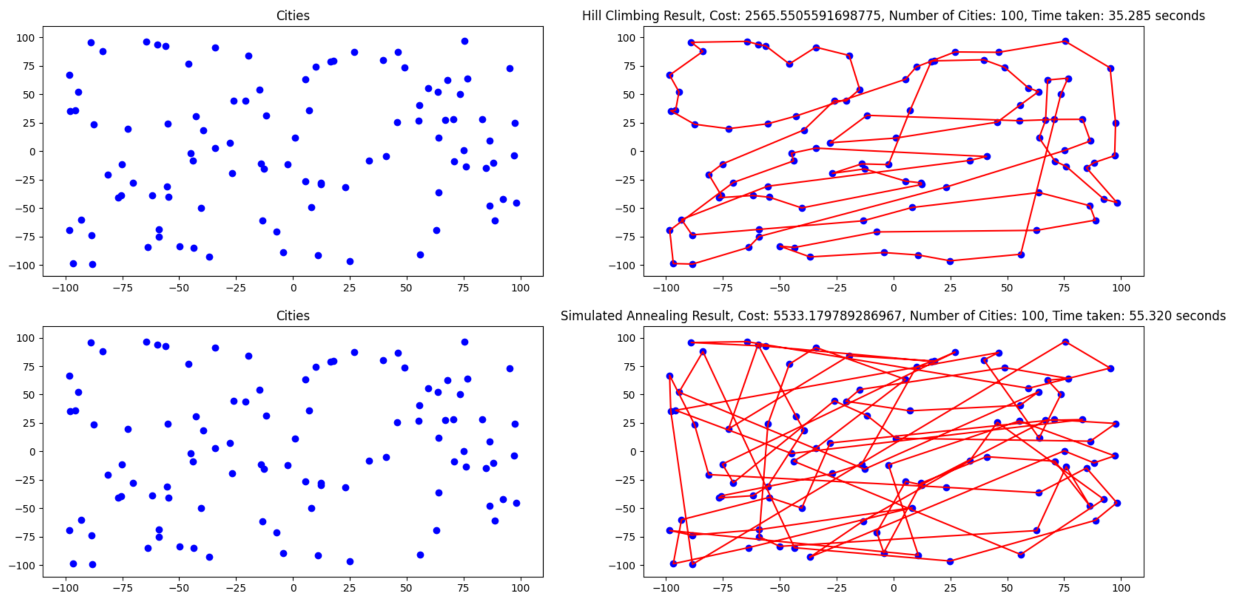


Figure 4: Tour cost visualizations for 100 cities

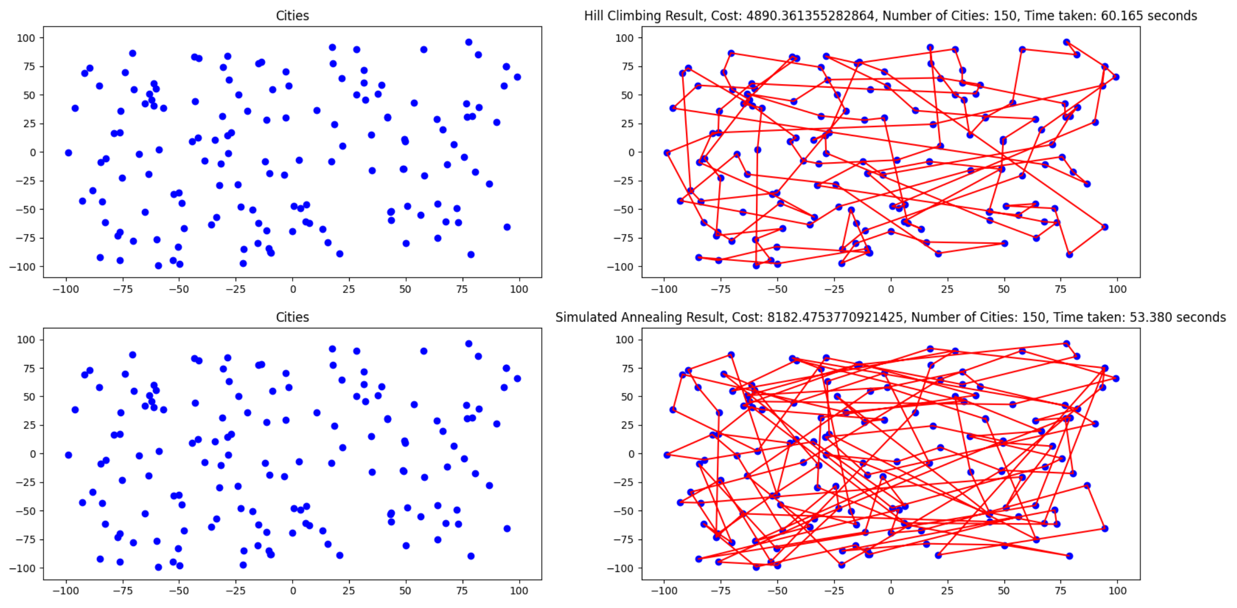


Figure 5: Tour cost visualizations for 150 cities

Group 10: Darius Koroni, Jett Sonoda, Bryan Tran, Tien Nguyen, Nhi Pham

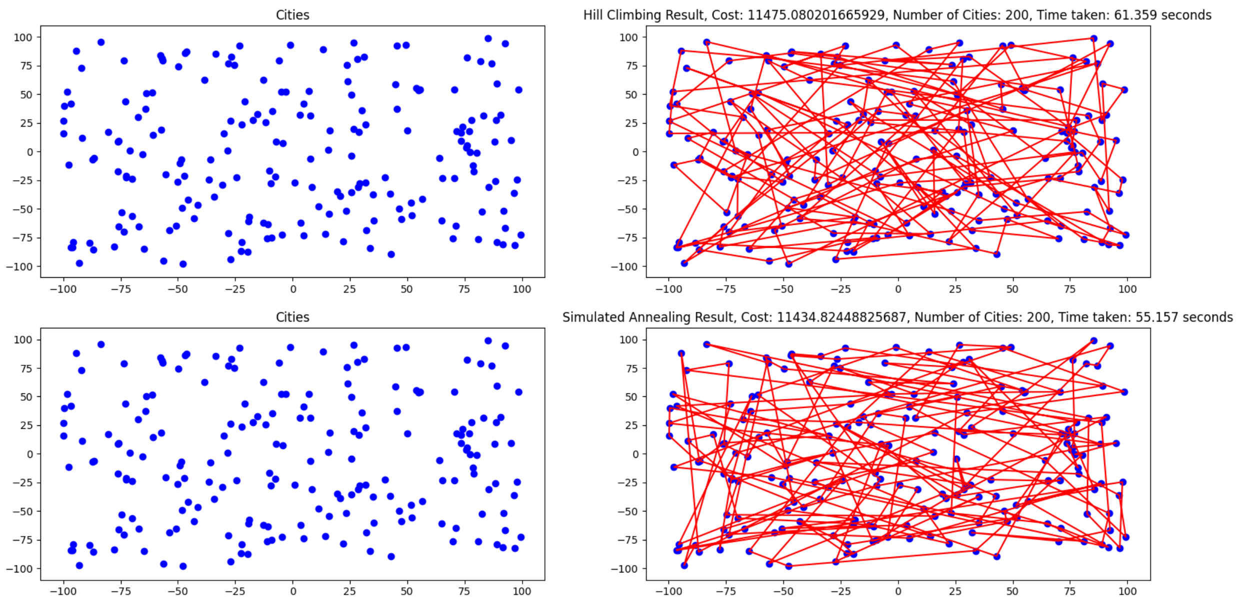


Figure 4: Tour cost visualizations for 200 cities

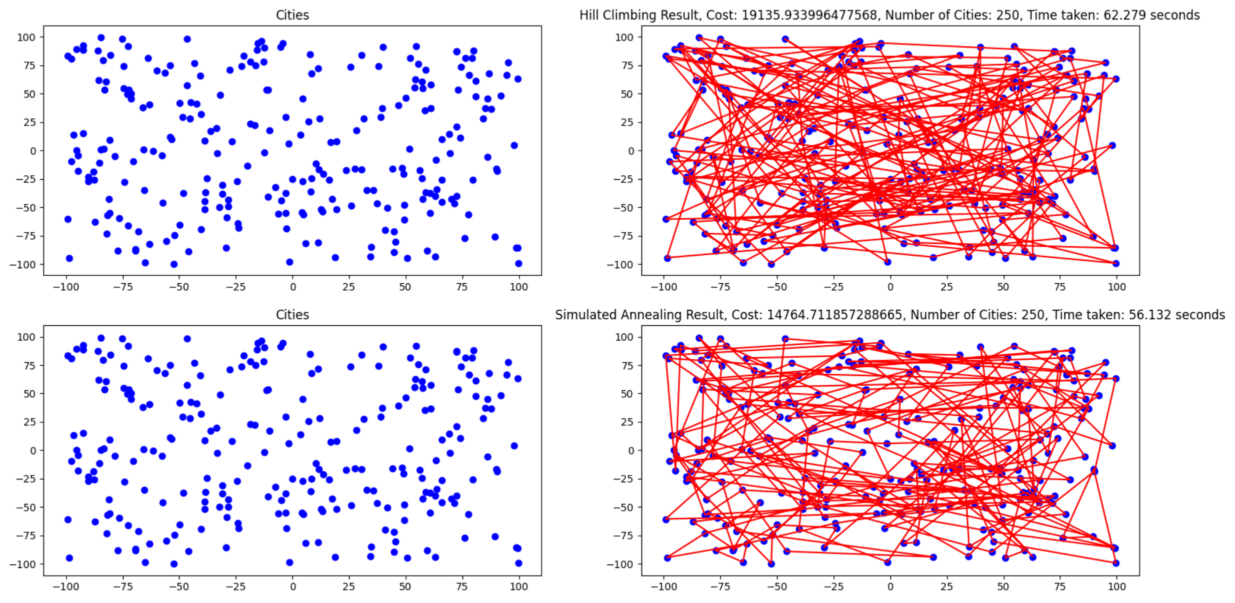


Figure 4: Tour cost visualizations for 250 cities

Group 10: Darius Koroni, Jett Sonoda, Bryan Tran, Tien Nguyen, Nhi Pham

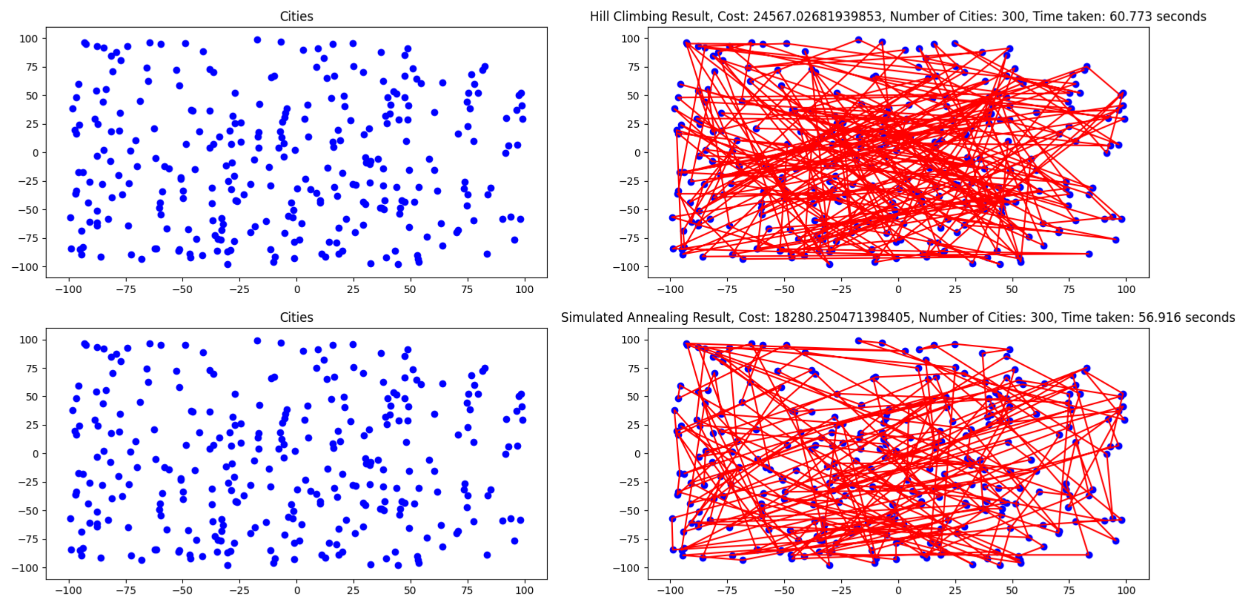


Figure 4: Tour cost visualizations for 300 cities

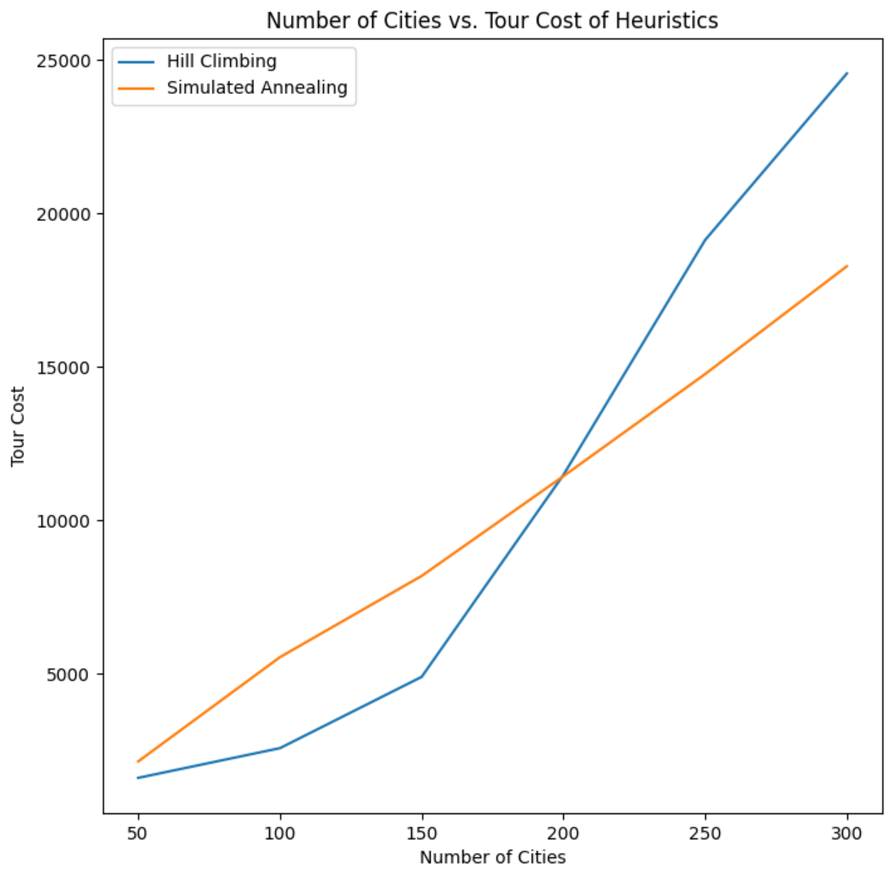


Figure 4: Number of Cities vs. Tour Cost of Heuristics

After running both algorithms, the simulated annealing is seen to perform exponentially better compared to the hill climbing algorithm over an increasing amount of cities. The hill climbing algorithm is simpler than simulated annealing as it requires less computations, thus, runs faster. However, most of the time it is stuck at the local optima and terminates early based on our program. Hill climbing beats simulated annealing in some scenarios where the number of cities is small. In contrast, the simulated annealing algorithm takes longer to explore the search space, but it always produces much better routes.

Dynamic programming algorithms for TSP involve constructing a table that stores the cost of the shortest path from each city to every other city. The method starts with the first city and iteratively builds up the answer by taking into account all potential routes to the other cities. This approach has a time complexity of O(n^2 * 2^n), where n is the number of cities. The main difference between using heuristic search techniques and dynamic programming is the time complexity and the guarantee of optimality. Heuristic algorithms have a lower time complexity but may not discover the ideal solution, whereas dynamic programming techniques have a greater time complexity but ensure that the optimal solution will be found. While dynamic programming typically works better for smaller TSP instances, heuristic algorithms can be faster and more useful for larger TSP instances.

VII. Conclusion

In conclusion, Simulated Annealing and Hill Climbing are two heuristic algorithms that can be used to solve the Travelling Salesman Problem. Simulated Annealing is more effective in finding the global optimum, but it requires more computing power and parameter optimization. On the other hand, Hill Climbing is simpler and faster, but it has the potential to get stuck in a local optimum. The performance of both algorithms is influenced by the problem size and the quality of the initial solution.

VIII. References

- Github: https://github.com/dkoroni/CECS451-SimulatedAnnealing-vs-Hill Climbing
- Simulated annealing lab: ttps://colab.research.google.com/drive/1i3U1oapRF256xO1hNyJtePJjGCPoL8RW