CMOR451 Project

Rahul Prakash

December 2024

1 Introduction

Optimization problems are present across a wide range of fields, from operations research and computer science to physics and machine learning. Among the many approaches to tackle these problems, simulated annealing (SA) stands out as a particularly powerful and versatile method. Simulated annealing is a probabilistic technique for approximating the global optimum of a given function. Inspired by the annealing process in metallurgy, where materials are heated and slowly cooled to remove defects and improve structural integrity, SA emulates this process to find optimal or near-optimal solutions in large and complex search spaces.

One of the key strengths of simulated annealing is its ability to escape local optima, a challenge faced by many other optimization algorithms like gradient descent. By allowing occasional acceptance of worse solutions during the search process, SA maintains exploration diversity and increases the likelihood of discovering the global optimum. This ability makes it especially useful for combinatorial optimization problems, where the search space is discrete and riddled with numerous local optima. Simulated annealing has been successfully applied to a variety of real-world applications, including scheduling, circuit design, network optimization, and machine learning model tuning.

In this discussion, we will focus on applying simulated annealing to one of the most well-known and challenging combinatorial optimization problems: the Traveling Salesman Problem (TSP).

2 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a classic problem in combinatorial optimization and graph theory. The premise is simple but profoundly challenging: given a set of cities and the distances between each pair of cities, the objective is to find the shortest possible route that visits each city exactly once and returns to the starting city. Despite its straightforward definition, TSP is classified as an NP-hard problem, meaning that as the number of cities increases, the computational effort required to find the exact solution grows exponentially.

Formally, the TSP can be represented as a complete weighted graph, where the nodes represent cities and the edges represent the distances between them. The goal is to determine a Hamiltonian cycle (a path that visits each node exactly once and returns to the starting node) with the minimum total distance.

3 Simulated Annealing and its Application to the TSP

3.1 The Simulated Annealing Algorithm

Simulated annealing provides an effective framework for addressing the TSP's complexity. By allowing controlled exploration of suboptimal routes, the algorithm increases the chance of finding a high-quality solution. The core idea is to start with an initial solution — which could be a random route through all the cities — and iteratively improve it. At each step, a small change is made to the current solution, such as swapping two cities in the route. If the new route is shorter, it is accepted. If it is longer, it may still be accepted with a certain probability that decreases as the algorithm progresses, mimicking the cooling process in annealing.

The temperature parameter plays a crucial role in controlling the acceptance of worse solutions. At high temperatures, the algorithm is more likely to accept worse solutions, promoting exploration of the search space. As the temperature decreases, the algorithm becomes more selective, focusing on local refinement around promising solutions.

The following outlines the key steps of the Simulated Annealing algorithm for optimizing the Traveling Salesman Problem (TSP). This algorithm explores potential solutions using a probabilistic approach to avoid local minima and converge to a near-optimal solution:

General Steps

1. **Define Inputs**
   - *start* — The starting city for the path.
   - *cities* — A list of all the cities to visit.
   - `max_iterations` — The total number of iterations to run the algorithm.
2. **Initialization**
   - Generate an initial path `current_path` starting from the given city.
   - Compute the distance `current_distance` for this path.
3. **Main Iteration Loop (for n from 1 to `max_iterations`)**
   - **Step 1: Generate a New Neighbor**

- Create a neighboring path `neighbor` by swapping two cities in the current path.
- Compute the distance `distance_neighbor` for this neighbor.

- **Step 2: Determine Whether to Accept the Neighbor**
  - If the new path is better (`distance_neighbor` < `current_distance`), accept it as the new current path.
  - If the new path is worse, compute the acceptance probability using:

$$p_{\text{accept}} = \exp\left(\frac{\Delta}{T}\right), \quad \text{where} \quad \Delta = \texttt{current\_distance} - \texttt{distance\_neighbor}$$

  - Generate a random number $u \in [0, 1]$. If $u < p_{\text{accept}}$, accept the worse path.

- **Step 3: Update the Best Solution**
  - If the current distance is better than the best-known distance, update the best-known path and distance.

- **Step 4: Update Temperature**
  - Update the temperature T for the next iteration using a cooling schedule. One possible schedule is:

$$T = \log(1 + n)$$

4. **Return Results**

- Return the best-known path and its distance

Mathematical Formulation

The acceptance probability p_{accept} is calculated as:

$$p_{\text{accept}} = \begin{cases} 1, & \text{if} \quad \Delta > 0 \\ e^{\frac{\Delta}{T}}, & \text{if} \quad \Delta \leq 0 \end{cases}$$

where Δ is the difference in distance between the current path and the neighboring path, and T is the temperature, which decreases over time according to the schedule $T = \log(1 + n)$.

Notice that when $\Delta \leq 0$, as the neighboring distance increases, Δ becomes more negative, causing the acceptance probability $e^{\frac{\Delta}{T}}$ to approach 0. Consequently, for neighboring distances that are significantly greater than the current distance, the probability of accepting them is very low.

3.2 Code

```python
def simulated_annealing(start, cities, max_iterations=500_000):
    current_path = compute_first_path(start, cities)
    current_distance = compute_distance_path(cities, start, current_path)
    best_path = current_path[:]
    best_distance = current_distance
    best_distances = [current_distance]
    current_distances = [current_distance]

    for n in range(1, max_iterations + 1):
        neighbor = find_neighbor(current_path)
        distance_neighbor = compute_distance_path(cities, start, neighbor)

        if distance_neighbor < current_distance:  # Accept better neighbor
            current_path = neighbor
            current_distance = distance_neighbor
        else:  # Accept worse neighbor probabilistically
            delta = current_distance - distance_neighbor
            temperature = np.log(1 + n)  # Calculate temperature
            acceptance_probability = np.exp(delta / temperature)

            if random.random() < acceptance_probability:
                current_path = neighbor
                current_distance = distance_neighbor

        # Update the best solution found
        if current_distance < best_distance:
            best_path = current_path[:]
            best_distance = current_distance

        # Record distance history
        best_distances.append(best_distance)
        current_distances.append(current_distance)

        # Optional: Print progress every 100000 iterations
        if n % 100 == 0:
            print(f"Iteration {n}, Best Distance: {best_distance}")

    return best_path, best_distance, best_distances, current_distances
```

Figure 1: Code for Simulated Annealing Algorithm

4 Variance Reduction through Adjusted Neighbor Selection

There are a variety of variance-reduction techniques when applying the simulated annealing method to the Traveling Salesman Problem, one of which is altering the way we generate our neighbors. Rather than generating a neighbor by swapping random cities within our current path (and then deciding to accept/reject it), we can instead impose a maximum distance that allows two cities to be swapped only if the distance between them is lower than the maximum distance. This is especially helpful as we begin approaching an optimal solution, where swapping cities far apart from one another is almost always useless and will lead to a distance that is significantly worse than our current distance and as a result is a waste of time. Instead, as we begin approaching an optimal solution, we want to swap neighbors that are closer together as this will reduce the variance in our current distances while still having the same answer as in our traditional method of simulated annealing (where we swap random neighbors). Let's refer to this new method as Method 2 and our original method as Method 1.

Computing Maximum Distance

Let our grid size be represented by $x \in (-x_1, x_1)$ and $y \in (-y_1, y_1)$. The maximum distance between two points in this grid is:

$$\text{Maximum Distance} = \sqrt{(2x_1)^2 + (2y_1)^2}$$

We define the distance threshold k as:

$$k = \frac{\text{Maximum Distance}}{\text{num\_cities}^{1/4}}$$

This threshold ensures that as the number of cities increases, the allowable maximum distance decreases, encouraging local exploration.

As we can see in the graphs below, we are able to maintain and even improve our simulated annealing distances when we use Method 2.

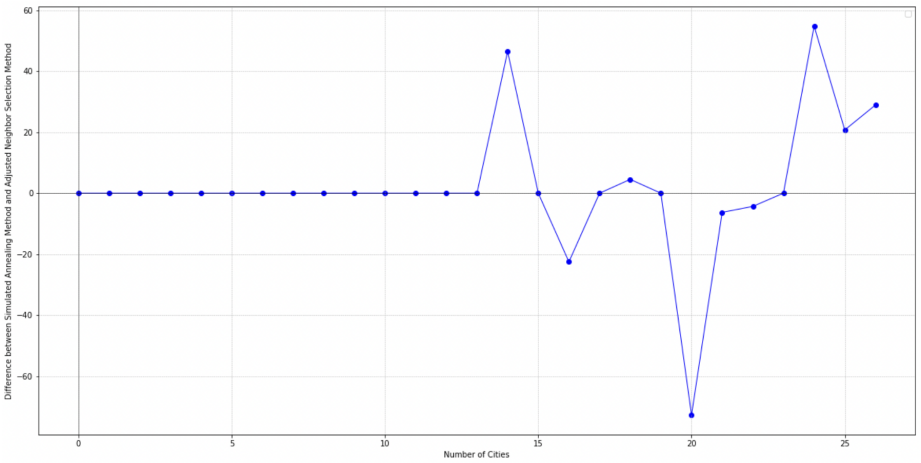

Figure 2: Performance of Method 1 vs Method 2

Negative values represent instances where Method 2 outperformed Method 1 (where neighbors are swapped randomly) and vice versa. As we can see, when the number of cities are small, both methods perform similarly but as the number of cities grow, each method produces different solutions, but it is unclear to tell which one is better based on the results seen above.

However, as we see in the following graph, the variance of the distances of generated neighbors is much lower in Method 2, which makes sense since neighbors are selected that closely resemble the current path. Given that it is unclear to detect which method is better just based off raw distances produced, we can see that there is significant variance reduction in Method 2 and thus is a more optimal method than Method 1.

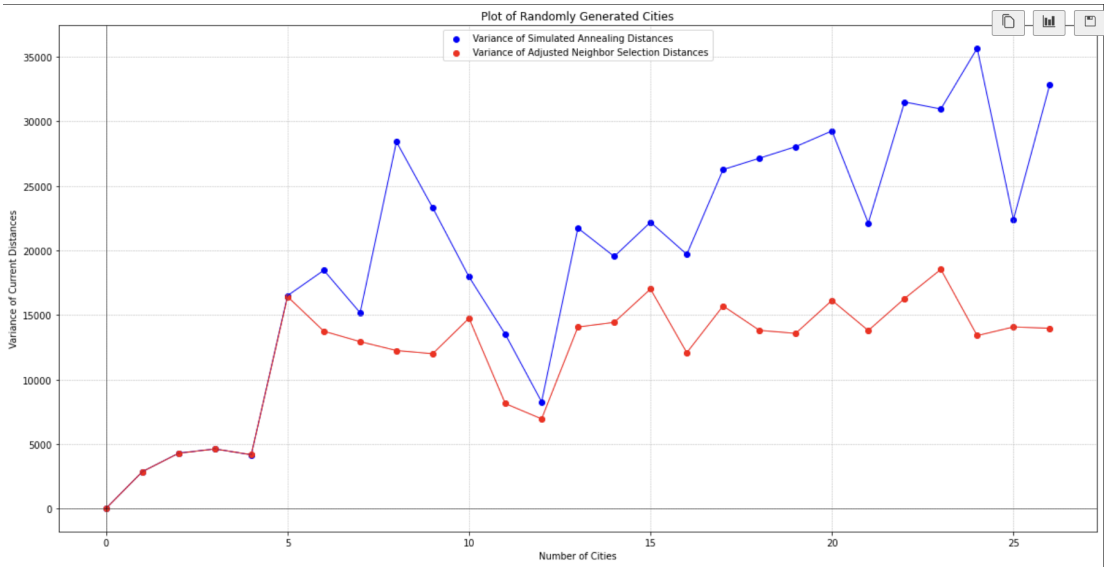

Figure 3: Variance of Neighbor Selection for Method 1 vs Method 2

5 Comparison with Integer Program

There are other methods that exist that can compute the optimal path but typically take much longer to run than simulated annealing, which usually provides a 'good enough' approximation for the optimal path. However, we can analyze the performance of our simulated annealing predictions by comparing our results with the results produced by an integer program. When it comes to the Traveling Salesman Problem, integer programs are almost always optimal.

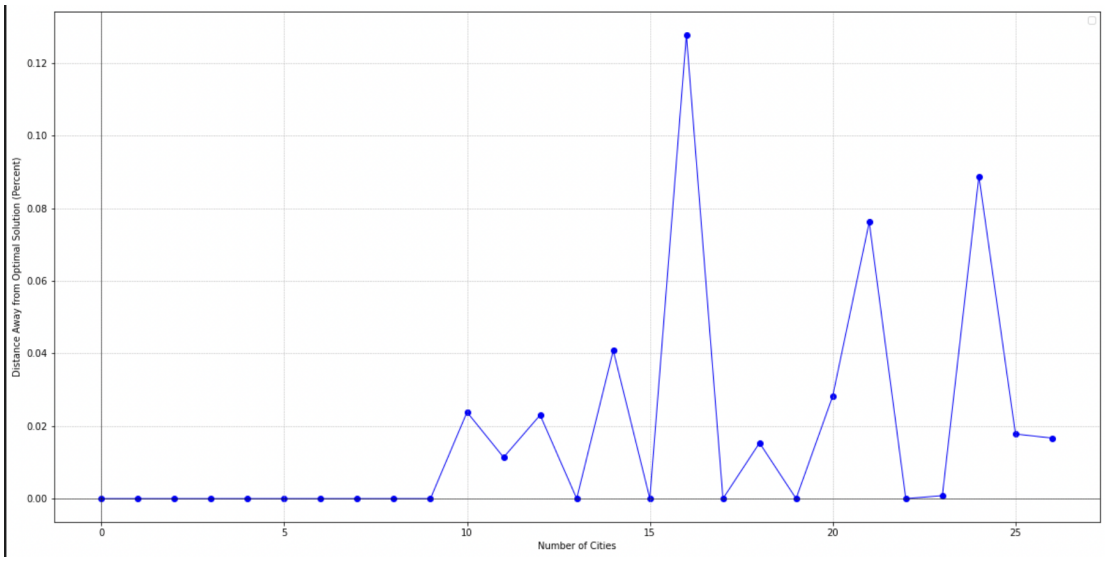

Figure 4: Comparison of Simulated Annealing and Integer Programming Solutions

As we can see, as the number of cities grow, our simulated annealing method begins stray away from the optimal solution, but not by much however. Therefore we can say that simulated annealing provides a good enough approximation for the best path, especially when the number of cities is small.

6 Conclusion

In this project, we explored the application of simulated annealing to solve the Traveling Salesman Problem (TSP), one of the most challenging and widely studied combinatorial optimization problems. By leveraging the flexibility and exploration capabilities of simulated annealing, we demonstrated how it can efficiently search for near-optimal solutions in complex, high-dimensional solution spaces.

Our implementation of simulated annealing included several key enhancements, most notably the introduction of a variance reduction strategy. By adjusting the neighbor selection process to limit swaps to nearby cities, we achieved a significant reduction in variance while maintaining solution quality. This method (Method 2) proved to be more stable and consistent than the traditional approach (Method 1), especially as the number of cities increased. The results clearly showed a marked reduction in variance, allowing for a more controlled and efficient search process.

Furthermore, we compared our results against an integer programming approach, which is known to provide optimal solutions for the TSP. While the integer program produces exact solutions, it comes with a much higher compu-

tational cost. On the other hand, simulated annealing provided a fast, effective approximation with minimal deviation from the optimal path, particularly when the number of cities was small to moderate.

Overall, this project demonstrated the power and versatility of simulated annealing in tackling NP-hard problems like the TSP. Our proposed variance reduction technique not only improved the stability of the solution but also showcased how thoughtful modifications to classical algorithms can yield significant performance gains. Future work could explore hybrid approaches that combine the speed of simulated annealing with the precision of integer programming, or further refine the neighbor selection method to achieve even greater variance reduction. This project highlights the importance of balancing exploration and exploitation in heuristic optimization and provides a strong foundation for future research in combinatorial optimization.