Simulated Annealing: Taking the Cooler Path in the Travelling Salesman Problem

Sahaana Vijay[a,1]

[a]*Ashoka University*

Prof. Somendra Bhattacharjee

Abstract—The Travelling Salesman Problem (TSP) is a classic and one of the most-studied NP-hard problems related to combinatorial optimisation. The key idea here – very similar to the famous Königsberg bridge problem – is that given N cities, we try to find the shortest route one can traverse, passing through each city exactly once and returning to the origin. It is essentially a minimisation problem. There have been various optimisation algorithms to find the shortest route such as genetic algorithm, ant colony optimisation and the Djikstra algorithm, however, here, we would be looking at *simmlated annealing* – a computational method that simulates solid annealing.This document aims to serve as a guide to the code, almost like an instructions manual.

Keywords—*Travelling Salesman Problem, combinatorial optimisation, shortest route, annealing*

1. Introduction

The Travelling Salesman Problem (TSP) was first formulated in 1930 and aims to find the shortest path one can take to visit N cities exactly once and return to the origin. Let the cities have the coordinates (x_i, y_i) where $i = 1, 2, 3, ..., n$. Each point X in the state space represents a possible order in which the N cities can be visited. To calculate the objective function $f(X)$, we find the total length of the tour described by the vector X, which shows the order of the cities visited. We then add up the distances between each pair of consecutive cities in the tour, plus the distance from the last city back to the first one

$$f(X) = \sum_{i=1}^{n-1} d(X_i, X_{i+1}) + d(X_n, X_1) \tag{1}$$

Here, X_i is the city visited at step i. If $X_i = k$ and $X_{i+1} = l$, the distance between those two cities is

$$d(X_i, X_{i+1}) = \sqrt{(x_l - x_k)^2 + (y_l - y_k)^2} \tag{2}$$

The final term $d(X_n, X_1)$ ensures that that the tour returns to the starting city, making it a complete loop. To minimise this distance, we will be using the *simulated annealing* algorithm.

2. What is Simulated Annealing?

In the early 1980s, Kirkpatrick, Gelatt, and Vecchi at IBM introduced the idea of *simulated annealing* (SA), drawing inspiration from the annealing process used in metallurgy to solve complex combinatorial optimisation problems [2]. When a metal is heated to a high temperature, its particles move randomly in a disordered liquid state. If it is cooled *slowly* enough, the particles gradually arrange themselves into a highly ordered, low-energy configuration. This cooling process, known as *annealing*, reaches the global minimum energy state.

This two-step process can be summarised as:

1. Heating the material until it enters a liquid state, where particles are free to arrange themselves into various configurations.
2. Cooling the material following a specific temperature schedule to settle into a minimum-energy structure.

If the cooling is rushed, the system may get trapped in a *metastable* (non-optimal) state – a process referred to as *quenching* or *hardening*.

The roots of this approach go back further, to the Metropolis algorithm, developed in 1953 by Metropolis, Rosenbluth, Teller, and colleagues [1]. Their goal was to accurately simulate the physical annealing process. They employed Monte Carlo techniques to generate a sequence of configurations (states) of a material. Starting from an initial state i with energy E_i, a new state j with energy E_j is proposed by making a small modification (for example, swapping two particles' positions).

The accepting rule is

- If $E_j < E_i$, the new state is accepted.
- If $E_j \geq E_i$, the new state is accepted with a probability given by the *Metropolis criterion*:

$$P = \exp\left(-\frac{E_j - E_i}{k_B T}\right) \tag{3}$$

where k_B is the Boltzmann constant and T is the system's temperature.

In the context of the TSP:

- Each **state** corresponds to a candidate solution (a permutation of cities in the TSP),
- The **energy** corresponds to the objective function value (the total distance travelled),
- The **temperature** is a control parameter that governs the acceptance of bad solutions.

The simulated annealing algorithm initially explores the solution space extensively by allowing worse routes at high temperatures. This mimics the high-energy state of a physical system where particles can move freely. As the temperature decreases, the algorithm becomes more selective, gradually favouring only lower-energy (or better) solutions. This cooling allows the system to escape local minima and improves the likelihood of converging towards a global minimum [4].

Algorithm Used

The algorithm here is essentially the Metropolis algorithm wherein the Metropolis criterion is (3).

1. Choose an initial solution x (a random route for TSP) and set the number of iterations. Also, set the initial temperature T_0 and a cooling rate, typically $T_{k+1} = \gamma T_k$, where $0 < \gamma < 1$.
2. Generate a new candidate solution y by applying a small random change to x (here, swapping cities).
3. Calculate the change in the objective function $\Delta E = f(y) - f(x)$
   - If $\Delta E < 0$, accept the new solution y.
   - Else, accept y with probability given by the Metropolis criterion (3)
   $$P = \exp\left(-\frac{\Delta E}{T}\right)$$
4. Update the current best path x accordingly.
5. Reduce the temperature using γ. Repeat this until the specified number of iterations has been completed.

3. Code: Initialisation

Before running the algorithm, the following data structures were set up:

1. `N_cities`: Specifies the number of cities involved in the TSP. Each city is uniquely indexed from 0 to N.

2. `city_coord`: A dictionary used to store the 2D coordinates of each city. The keys are city indices, and the values are tuples of the form (x, y) representing their positions on a grid.
3. `dist_list`: An empty list used to keep track of the total distance of the current tour at every iteration. This is useful for monitoring convergence and generating plots of distance versus iteration number.
4. Additinally, the positions of the cities are generated randomly within a 20×20 grid using `np.random.randint(0, 20)` for both x and y coordinates. This simulates cities being scattered arbitrarily in a 2D space.

4. Code: Functions Used

4.1. euc_dist(x_1, x_2, y_1, y_2)

This function basically calculates the Euclidean distance between any two points (cities) give (x_i, y_i) using equation (2).

```
def euc_dist(x_1, x_2, y_1, y_2):
    '''
    This function calculates the distance between any two points (x_i, y_i)
    '''
    return np.sqrt((x_1 - x_2)**2 + (y_1 - y_2)**2)
```

Code 1. Function euc_dist()

4.2. total_dist(city_coord)

This function calculates the total distance of a closed path that visits all cities exactly once and returns to the starting point. The function iterates through each city and computes the Euclidean distance to the next city in the sequence. To ensure the route forms a loop, the last city is connected back to the first using the modulo operation:

$$\texttt{next\_i} = (i + 1) \bmod N \tag{4}$$

where N is the total number of cities. The total distance is accumulated in a variable `distance`, which is returned at the end of the function. This also serves as the objective function in the optimisation process, with the goal being to minimise it.

Parameters:

- `city_coord`: A dictionary where each key corresponds to a city index and the associated value is a tuple representing the (x, y) coordinates of the city.

```
def total_dist(city_coord):
    '''
    This function returns the total distance covered by the walker while goimg through each of the cities
    '''
    distance = 0
    for i in range(0, len(city_coord)):
        next_i = (i + 1) % len(city_coord) # to handle wrap-around (last city to the first city)
        distance += euc_dist(city_coord[i][0], city_coord[i][1], city_coord[next_i][0], city_coord[next_i][1])
    return distance
```

Code 2. Function total_dist()

4.3. swap(city_coord)

This function proposes a new solution for the TSP by modifying the current combination of cities. It does so by randomly swapping the positions of two cities in the tour, excluding the starting city to maintain a fixed reference point. The function performs the following:

1. Creates a copy of the original city dictionary to avoid modifying the original input.
2. Selects two distinct random indices in the range $[1, N-2]$, where N is the total number of cities. This range excludes index 0, which is treated as the fixed starting city.
3. Swaps the positions of the two selected cities.
4. Returns the modified dictionary representing the new proposed path.

Parameters:

- `city_coord`: A dictionary mapping city indices to their (x, y) coordinates.

```
def swap(city_coord):
    '''
    This function first generates random indices between the range 1 (we are not considering the starting point) and the last index of city_coord. Then, for the x
    and y coordinates, it chooses a random index from those set of indices, and using those, we can swap the values of the dictionary.
    '''
    cities_new = city_coord.copy()
    index = range(1, len(city_coord) - 1)

    i = np.random.randint(index[0], index[-1])
    j = np.random.randint(index[0], index[-1])

    while i == j:
        j = np.random.randint(index[0], index[-1])

    cities_new[i], cities_new[j] = cities_new[j], cities_new[i]

    return cities_new
```

Code 3. Function swap()

4.4. cooling(temp, gamma)

This function implements the cooling schedule in a SA algorithm. It progressively reduces the temperature parameter, which controls the probability of accepting worse solutions as the algorithm proceeds. The function returns the updated temperature by multiplying the current temperature by the cooling rate γ. This results in an exponential decay of the temperature over iterations. The temperature controls how likely the algorithm is to accept worse solutions:

- At higher temperatures, the algorithm is more likely to accept uphill moves (worse solutions), allowing it to escape local minima.
- As the temperature decreases, the algorithm becomes more conservative, focusing on refining the current best solution.

```
def cooling(temp, gamma):
    '''
    After every iteration, we are decreasing the temperature (in our case some way to quantify the randomness)
    '''
    return temp * gamma
```

Code 4. Function cooling()

4.5. checking(temp, new_f, old_f)

This function determines whether a newly proposed solution (path) should be accepted in the context of the Simulated Annealing algorithm, particularly when it is worse than the current solution. If the proposed solution is worse (i.e., has lower fitness), it is accepted

with a probability given by $P = \min\left(1, e^{-\frac{\text{new\_f}-\text{old\_f}}{\text{temp}}}\right)$. This probability decreases as:

- The difference in the old and new objective function value increases (worse new solution).
- The temperature decreases (as the algorithm progresses).

A uniformly distributed random number $r \in [0, 1]$ is generated. If $r < P$, the new solution is accepted. This probabilistic acceptance criterion allows the algorithm to escape local minima early on, when temperature is high, and gradually focus on local search as temperature lowers.

Parameters:

- `temp`: The current temperature in the annealing schedule.
- `new_f`: The fitness of the proposed (new) solution, i.e., the new total distance.
- `old_f`: The fitness of the current (old) solution, i.e., the old total distance.

```
def checking(temp, new_f, old_f):
    '''
    If the distance is less than the shortest distance recorded so far, the new path is accepted with a probability e^(-(new distance - old distance)/temperature).
    Returns True if accepted, else False.
    '''
    prob = min(1, np.exp(-(new_f - old_f)/temp))
    if prob > np.random.uniform(0, 1):
        return True
    else:
        return False
```

Code 5. Function checking()

4.6. main(temp, gamma, iters, plot_int = 200)

This is the main driver function that implements the SA algorithm to solve the TSP. It integrates all perviously mentioned helper functions to iteratively search for the shortest tour connecting a set of cities. This function is where we are implementing the overall algorithm. This function animates the entire process and every `plot_int` iterations (or at the final iteration), updates a live plot to visualise the current state of the tour. The plot dynamically shows the current tour and the total distance at that iteration.

Parameters:

- `temp`: The initial temperature of the system, which controls the acceptance probability of worse solutions. Higher temperatures allow more exploration.
- `gamma`: The cooling rate. After each iteration, the temperature is updated by multiplying it with `gamma` ($0 < \gamma < 1$), gradually reducing randomness.
- `iters`: The total number of iterations the algorithm will run.
- `plot_int`: The interval at which the current tour is visualised using an animated plot.

```
def main(temp, gamma, iters, plot_int = 200):
    '''
    This part combines all the previous functions together to generate the best path. Parameters:
    temp: The "Temperature" of the system is essentially a measure of randomness with withich changes are made to the path. It is a control parameter.
    gamma: The rate at which the system is "cooling", or settling to a global minimum (best path in our case)
    iters: The number of iterations
    plot_int: The time interval for the animation part
    '''
    global best_tour, final_coord, best_dist
    cur_tour = city_coord.copy()
    cur_dist = total_dist(cur_tour)

    best_tour = cur_tour.copy()
    best_dist = cur_dist

    fig, ax = plt.subplots()
    plt.ion() # interactive mode

    for i in range(iters):
        dist_list.append(cur_dist)
        prop_tour = swap(cur_tour)
        prop_dist = total_dist(prop_tour)

        if prop_dist < cur_dist:
            cur_tour = prop_tour
            cur_dist = prop_dist
        else:
            if checking(temp, prop_dist, cur_dist) == True:
                cur_tour = prop_tour
                cur_dist = prop_dist

        if cur_dist < best_dist:
            best_tour = cur_tour.copy()
            best_dist = cur_dist

        temp = cooling(temp, gamma)

        # Animating paths
        if i % plot_int == 0 or i == iters - 1:
            ax.clear()
            final_coord = list(cur_tour.values())
            x_coord = [i[0] for i in final_coord]
            y_coord = [i[1] for i in final_coord]

            ax.plot(x_coord + [x_coord[0]], y_coord + [y_coord[0]], 'o-',c = 'teal', label = f'Iter: {i}\nDist: {round(cur_dist, 3)}')
            # ax.scatter(x_coord, y_coord, c = 'midnightblue')
            ax.set_title('TSP Paths with Simulated Annealing')
            ax.set_xlabel(r'$x$')
            ax.set_ylabel(r'$y$')
            ax.legend()
            ax.grid(True)
            plt.pause(0.001)

        plt.ioff()
```

Code 6. Function main()

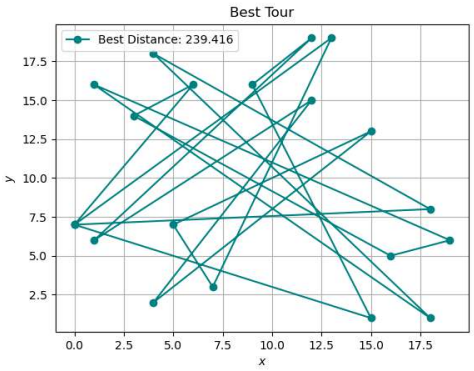
4.7. best_path(best_tour, best_dist)

This function is used to display the final plot of the shortest distance for the TSP. In a second figure, the evolution of the total distance across all iterations was plotted using the global list `dist_list`. This second plot shows how the algorithm converges over time and gives insight into the optimisation using SA – it is very similar to the actual plot for solid annealing.

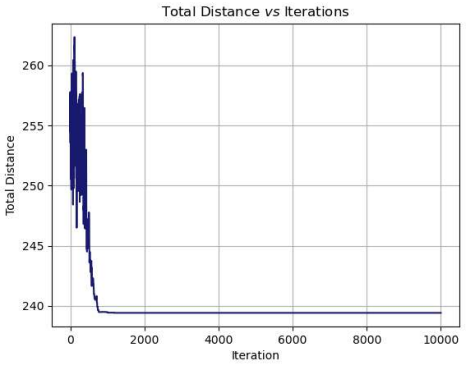
Parameters:

- `best_tour`: A dictionary representing the best tour found by the algorithm, where the keys correspond to city indices and the values are 2D coordinates.
- `best_dist`: The total distance of the best tour.

```
def best_path(best_tour, best_dist):
    '''
    This function plots the final best path and how the total distance decreases with the iterations
    '''
    final_coord = list(best_tour.values())
    x_coord = [i[0] for i in final_coord]
```



(a) Final best path for $N = 20$ cities



(b) Plotting total distance versus iteration number for the Simulated Annealing algorithm applied to the TSP.

Figure 1. Results for TSP with simulated annealing with $N = 20$ cities. In the second plot, a rapid decrease in distance is observed during the initial iterations, indicating effective exploration of the solution space. The curve subsequently flattens, signifying convergence to a near-optimal solution. This behaviour reflects the gradual reduction in randomness as the system cools, leading to stabilisation of the tour.

```
    y_coord = [i[1] for i in final_coord]

    plt.figure()
    plt.plot(x_coord + [x_coord[0]], y_coord + [y_coord[
     ↪ 0]], 'o-',c = 'teal', label = f'Best Distance: {
     ↪ round(best_dist, 3)}')
    plt.title('Best Tour')
    plt.xlabel(r'$x$')
    plt.ylabel(r'$y$')
    plt.legend()
    plt.grid(True)
    plt.show()

    plt.figure()
    plt.plot(dist_list, c = 'midnightblue')
    plt.title(r'Total Distance $vs$ Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Total Distance')
    plt.grid(True)
    plt.show()
```

Code 7. Function best_path()

5. Results and Future Work

The entire code can be found here: https://github.com/LoopyNoodle/TSP-with-Simulated-Annealing. The code was executed for $N = 20$ cities, and the results can be seen in Fig. (1). In the future, the code also aims to explore how the information entropy varies with the number of iterations as well as introducing other complications such as what happens when there is a river passing through as seen in [3].

References

[1] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines", *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953. DOI: 10.1063/1.1699114.

[2] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing", *Science*, vol. 220, no. 4598, pp. 671–680, 1983. DOI: 10.1126/science.220.4598.671.

[3] J. Walker, *Simulated annealing: The travelling salesman problem*, Accessed: 2025-05-07, 2018. [Online]. Available: https://www.fourmilab.ch/documents/travelling/anneal/.

[4] D. Delahaye, S. Chaimatanan, and M. Mongeau, "Simulated annealing: From basics to applications", in *Handbook of Metaheuristics*, ser. International Series in Operations Research Management Science (ISOR), M. Gendreau and J.-Y. Potvin, Eds., vol. 272, Springer, 2019, pp. 1–35, ISBN: 978-3-319-91085-7. DOI: 10.1007/978-3-319-91086-4_1.