Optimization of the Traveling Salesman Problem Using Simulated Annealing and Parallel Tempering

Melina Karlgren

Autumn 2024

Contents

1 Abstract

This report implemented two optimization algorithm on the Traveling Salesman Problem, Simulated Annealing and Parallel Tempering. The objective was to compare the two methods in terms of performance and computational efficiency. A combination of the two methods was also implemented and tested. The results showed that Simulated Annealing performed best in terms of both performance and efficiency. The results of the combined method were similar to those of the Parallel Tempering method. It is also concluded that the Parallel Tempering method requires fine tuning, and could possibly be tuned to improve the results. Simulated Annealing is the most intuitive and least demanding method which also gives good results at least for simulations with a limited number of cities. Optimization with a large number of cities proved to be difficult and we could not be sure the optimal solution was found when the number of cities was increased.

2 Introduction and Problem Description

The Traveling Salesman Problem is a classical optimization problem. Given a set of cities, we want to compute the shortest path possible that allows us to visit all cities and return to our home city. Even though the idea of the problem sounds very simple, the Traveling Salesman Problem is computationally difficult, with an increasing number of cities, it becomes computationally unfeasible to calculate the exact solution.

The cities are placed into a plane of unit length, where N cities are positioned randomly on the 2D square. The distance between two cities is calculated using the Euclidian norm, where the function to be minimized is the total distance when visiting every city and returning home, hence

$$H = \sum_{n=1}^{N} \|\mathbf{r}_i - \mathbf{r}_{i+1}\|,$$

where $\mathbf{r}_i$ are the city locations, where city $i+1$ is visited after city i, and $\mathbf{r}_{N+1} = \mathbf{r}_1$.

In order to find the exact solution to the TSP by comparing values of H, we need to compare every possible permutation of order of cities visited. The order of which the cities can be visited grows as $N!$, which means the number of permutations quickly becomes large and highly computationally demanding. Therefore, for large N, we can not find the exact solution by this method, but we instead use optimization to find a minimum of H. In this project, two different optimization techniques where used, Simulated Annealing and Parallel Tempering.

3 Theory

3.1 Simulated Annealing

Simulated Annealing is an optimization method to find the global optimum in a large search space. In the simulation, we start with a random order of cities, we then introduce Monte Carlo trial moves that are accepted with a Metropolis acceptance rate. A trial move will perturb the order of cities and thereby change the value of H.

If $H_{\mathrm{i}+1} < H_{\mathrm{i}}$, the trial move is accepted. To avoid getting stuck in a local minimum of H, we must also accept trial moves where $H_{\mathrm{i}+1} > H_{\mathrm{i}}$. To do this, we use a Metropolis acceptance criterion relying on a parameter called the temperature. The Metropolis acceptance criterion is defined as

$$p = \min\left(1, e^{\frac{H_{i+1}-H_i}{T}}\right),$$

where H_{i+1} is H for the new order after the trial move, H_i is H for the previous order before the trial move, and T is the temperature. The trial move is accepted if $p \geq r$, where r is a uniformly distributed random number between 0 and 1, $r \in U[0,1]$. If the trial move is accepted, the new order of cities is saved and $T = T \times c$, where c is the cooling rate.

In the start of the simulation, we have a high temperature, $T = 1$, meaning that Monte Carlo moves will be accepted to a great extent. With each iteration, we lower the temperature by a cooling rate, c, resulting in the temperature decreasing exponentially. By doing this sufficiently slow *i.e* choosing an appropriate cooling rate, the system will be able to first explore all possible orders and then slowly converge to the optimal solution.

3.2 Parallel Tempering

Parallel Tempering is another method that similarly to Simulated Annealing proposes a solution to not get stuck in a local minimum by letting the system explore a larger variety of configurations through variation of the temperature parameter used in the Metropolis acceptance criterion.

In Parallel Tempering, the approach is quite different. We simulate several systems in parallel at different temperatures, $T_1 < T_2, ..., < T_N$. These systems are called replicas. After all replicas are initiated and have a unique temperature, we in every iteration perform two Monte Carlo algorithms.

1. For every replica, a trial move that perturbs the order of cities in the replica is introduced. This trial move is accepted or rejected based on the same Metropolis criterion that was defined for the Simulated Annealing algorithm.
2. After all replicas have had the chance to perturb its order of cities by a trial move, for some of the iterations (within intervals defined), a new Monte Carlo algorithm is introduced that attempts to swap the temperature between two replicas.

   This algorithm chooses a random replica index and an adjacent neighbor index to that replica. The algorithm attempts to swap the temperatures between these two neighboring replicas by the use of a Metropolis acceptance criterion defined as

   $$p = \min\left(1, e^{(H_i - H_j)\left(\frac{1}{kT_i} - \frac{1}{kT_j}\right)}\right),$$

   where $k = 1$, the H_i is H for replica i, and H_j is H for replica j, T_i is the temperature of replica i and T_j is the temperature of replica j.

   The temperature swap is accepted if $p \geq r$, where r is a uniformly distributed random number between 0 and 1, $r \in [0, 1]$.

4 Method

4.1 Trial moves

In this project, we worked with three different trial moves that perturbed the order of cities

- **Swap**: Changes the order of two randomly chosen cities.
- **Two-opt**: Selects two random non adjacent city-indices and reverses the order of cities between the indices.
- **Three-opt**: Selects three random indices, and rearranges the order of cities by reversing the city order between the segments.

We started by analyzing which combination of trial moves should be used by computing the mean H for simulations with different sets of trial moves, and also the standard deviation of H and the convergence rate *i.e* how many iterations it took for the simulations to converge to a final path length value.

4.2 Simulated Annealing

The coordinates for the cities where initially selected randomly on a 2D plane of unit length, but were then saved for us to be able to test the same configurations multiple times. Each city was represented by an x-

and y-coordinate, and the order of cities was represented by the order the cities were placed in the cities-list. The initial temperature was set to 1, the minimum temperature for when reached, the simulation would stop was chosen as $T_{\min} = 0.0001$. A cooling rate was chosen. Then the updating of the cities-list started. The updating of the cities-list stops when the temperature goes below $T_{\min}$. By then, the H obtained is calculated and stored to be compared to more individual simulations. The code for the Simulated Annealing algorithm can be viewed in Appendix 7.2.

The Simulated Annealing algorithm was first tested for six cities, for which the optimal H was also computed by testing all possible permutations of paths. The algorithm was ran 100 times for different cooling rates, and the distribution of final path lengths *i.e* H was plotted in a histogram.

The same process was then repeated for twenty cities, where the optimal H could not be computed by testing all permutations of paths, meaning we did not have a reference true optimal order to compare our results to. Instead, we had to rely on distribution width and cooling the system slower to try and obtain an optimal path.

The width of the distributions of Hs when varying the cooling rate where plotted using the standard deviation of final path lengths for each cooling rate. This was also done similarly when varying the number of cities used in the simulations while keeping the same cooling rate.

4.3 Parallel Tempering

The code for the Parallel Tempering algorithm can be seen in Appendix 7.3. The algorithm was implemented on the same configuration of six and twenty cities as was done for the Simulated Annealing algorithm. The random walks in temperature was also plotted for each case.

4.4 Combined Simulated Annealing and Parallel Tempering

We also wanted to combine the two optimization methods implemented by introducing a cooling rate to the Parallel Tempering algorithm. This will lead all replicas to have a decrease in temperature while they simultaneously will be able to swap temperatures with each other. The step for when the cooling rate is added is also presented in Appendix 7.3.

5 Results and Analysis

5.1 Simulated Annealing

5.1.1 Comparing performance of trial moves

The results of comparing the different trial moves are presented in Table 1, where we used 20 cities, 1000 simulations and cooling rate 0.99. The trial move for each run in a simulation was chosen with equal probability between the present options. We can see from the results that the most optimal set of trial moves is the swap and three-opt. This set of trial moves results in the lowest average H-value and also has the fastest convergence rate. We could see that the worst set of trial moves was the swap and the two-opt. The advantage with having the two-opt and three-opt is that it lets the system explore a wide range of configurations fast, but for when the system is close to its optimum, these trial moves can perturb the order of cities too much, leading away from the optimal solution. Since the three-opt will change the configuration more than the two-opt, it is less likely to be accepted at low temperatures, but can instead work as a stop for the swap moves to get the system away from an optimal solution. The two-opt move is more likely to be accepted even at lower temperatures, which is why the swap and three-opt is the optimal set of trial moves and will be used here after.

Table 1: Comparing trial moves.

	Mean H	Std H	Convergence iteration
Swap	7.46	0.38	499
Swap and two-opt	7.55	0.39	512
Swap and three-opt	7.42	0.42	430
Swap, two-opt and three-opt	7.45	0.44	475

5.1.2 Simulated Annealing performance

In Figure 1 and Figure 2, we plotted the H values over the iteration steps when applying the Simulated Annealing algorithm to 6 and 20 cities respectively. We can see that the algorithm works as expected by firstly exploring many different orders of cities visited until we reach a minimum value of H. In the case of 6 cities, we know the minimum H value since the amount of possible permutations of cities are sufficiently few for the computer to test every permutation and find the optimal H. For our configuration of cities, we found that for six cities, $H = 2.218$. This computation was not feasible for the 20 city setup since the number of permutations grow as $N!$ which becomes very computationally costly.

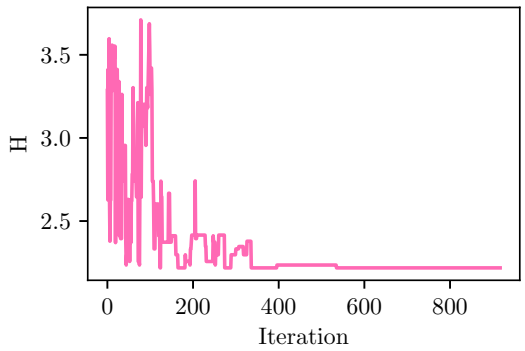

Figure 1: H values plotted over the iteration steps for 100 simulations of 6 cities with cooling rate 0.99.

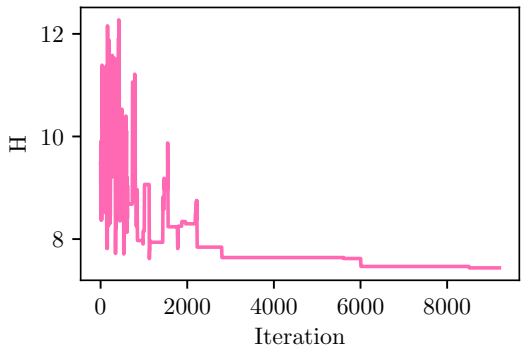

Figure 2: H values plotted over the iteration steps for 100 simulations of 20 cities with cooling rate 0.999.

To get a sense of how the algorithm performs dependent on if we have a start configuration with few cities or more, we plotted the final path lengths (H) in histograms over 100 simulations for both the 6 city setup and the 20 city setup, which can be seen in Figure 3 and Figure 4 respectively. We decided to use a cooling rate of 0.98 for the six city setup since it was found that we reached the minimum H in the absolute majority of simulations. For the 20 city setup, since we do not know the actual optimal H, it is more difficult to determine a sufficient cooling rate. The cooling rate used for 20 cities was 0.999 since it was slower than 0.99, but still made the simulation time quite short.

As we can see in Figure 3, for the 6 cities, the Simulated Annealing algorithm performs really well, only leading $\sim 1\%$ of the simulations to a final path length that is not optimal. For the 20 city setup in Figure 4, the distribution of final path lengths become a lot wider, and our final lengths tend towards a more Gaussian distribution, which is not desired since we want an exponential distribution with its maximum density at the minimum H. This distribution makes it really hard to know if we have actually found the optimal H or not during the simulation.

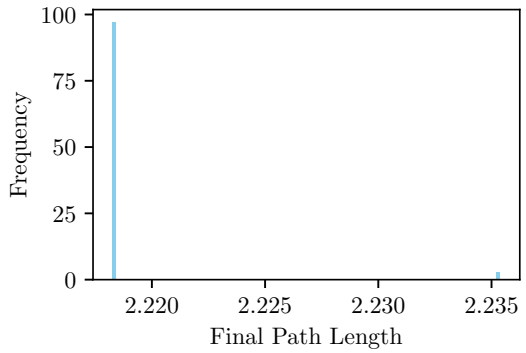

Figure 3: Final path lengths plotted for 100 simulations of 6 cities with cooling rate 0.98.

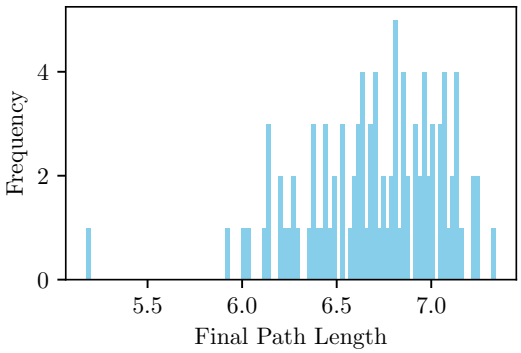

Figure 4: Final path lengths plotted for 100 simulations of 20 cities at cooling rate 0.999.

To investigate how the cooling rate impacts the width of distribution of H-values, we varied the cooling rates between 0.9 and 0.999 for both the 6 cities in Figure 5 and the 20 cities in Figure 6. Agreeing with the previous results, we can see that for the 6 cities, the standard deviation is quite low even for faster cooling rates, and even converges to 0 for slower cooling rates which means we have reached the minimum H-value for all simulations. For the 20 cities, we have a significantly higher standard deviation of final path values, and we can see that the standard deviation decreases with a slower cooling rate, but it is still remaining quite high, in the order of ten times larger than for the 6 city setup. This clearly shows the challenge with the traveling salesman problem.

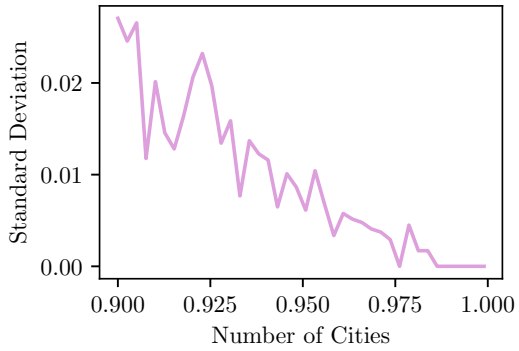

Figure 5: Standard deviation of final path lengths values for simulation of 6 cities.

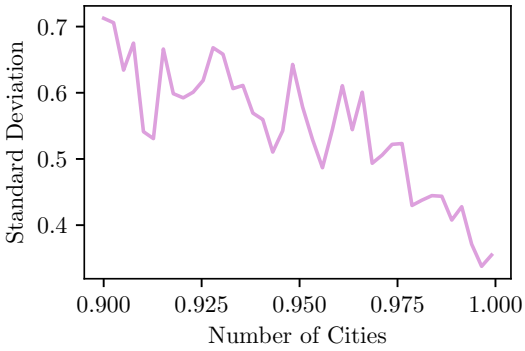

Figure 6: Standard deviation of final path lengths values for simulation of 20 cities.

We also wanted to plot how the standard deviation varied with the number of cities in our simulation for a fixed cooling rate. This was done for a cooling rate of 0.99 with number of cities ranging from 3 to 100 as can be seen in Figure 7, where for every number of cities, we performed 100 independent simulations with a new random configuration of cities. In Figure 8, we did the same procedure, but the city coordinates where fixed for each number of cities tested.

In Figure 7, we can see that the standard deviation of H follows an exponential increase with number of cities. Up to ~ 10 cities, the standard deviation is close to 0, meaning all solutions end up within a narrow band of H. This shows that for up to ~ 10 cities, the algorithm is very precise and trustworthy. For larger amount of cities, we have a broader distribution of H, this in itself does not mean we can not find the optimum solution to be the minimum H found within one distribution from one simulation, the problem is that with a broad distribution, it is hard to tell if the minimum value obtained is the actual optimum value or just the minimum within the range of Hs found. In Figure 7, we can also see that the standard deviation

becomes constant after $\sim$ 40 cities, this is due to the restriction of space for the cities to placed within a unit cell. After $\sim$ 40 cities, adding another city will not change the value of H much since we use random configurations of cities in each run.

In Figure 8, we fix the city coordinates for each run, which generally makes the standard deviation decrease, which is intuitive. We can also see here that the standard deviation increases more slowly and linearly after $\sim$ 20 cities. We do not get the standard deviation to become constant since we are not exploring the whole unit cell by randomizing city configurations for each run. We can see that the standard deviation is close to zero up until $\sim$ 7 cities, meaning we are almost guaranteed to find the optimal solution for configurations with up to 7 cities.

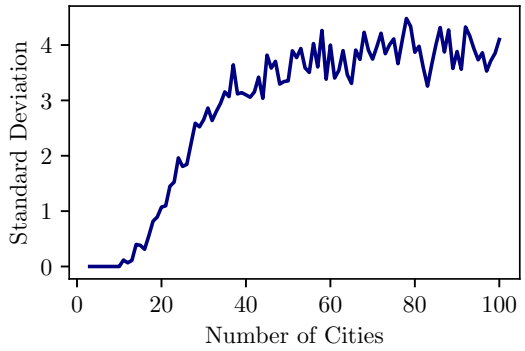

Figure 7: Standard deviation of final path lengths for 100 independent simulations with random city locations performed at number of cities ranging from 3 to 100 with a cooling rate of 0.99.

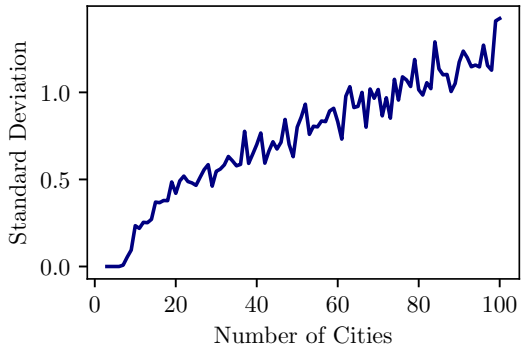

Figure 8: Standard deviation of final path lengths for 100 independent simulations with fixed city locations performed at number of cities ranging from 3 to 100 with a cooling rate of 0.99.

5.2 Parallel Tempering

We started by using the same 6 cities with the Parallel Tempering algorithm and 2000 iterations with 50 replicas and the possibility to swap two replica temperatures every 5th iteration. Every 5th iteration, we attempted to swap 30 replica temperatures simultaneously. The results are shown in Figure 9, where we see the distributions of final path lengths over 100 Parallel Tempering simulations. We can see that the method performs worse than the Simulated Annealing algorithm. It manages to find the optimal H for the majority of simulations, but still, some simulations end up at less optimal final path lengths. To illustrate how the temperatures swap between replicas, we included a plot of the temperature for every fourth replica over the iterations for one simulation in Figure 10. We can see that the replicas swap temperatures quite frequently, but the majority still tend to stay in the same temperature range as they where initiated in. We can see that some replicas get to explore quite a wide range of temperatures. The algorithm could be further tuned to optimize the temperature exchanges between different replicas.

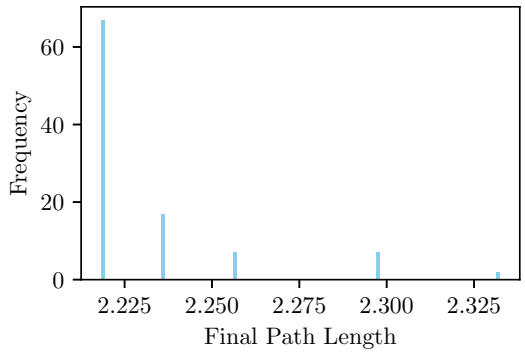

Figure 9: Final path lengths plotted for 100 simulations of 6 cities with the Parallel Tempering algorithm.

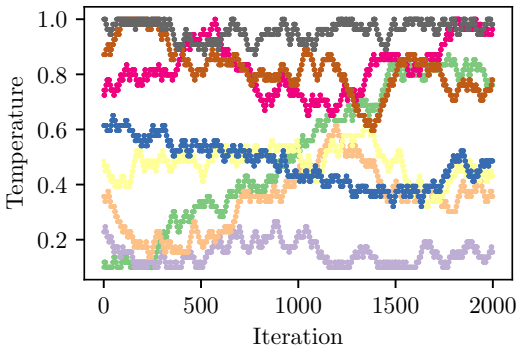

Figure 10: Temperatures for every seventh replica plotted over the iteration steps for 6 cities with the Parallel Tempering algorithm.

To investigate how this algorithm performs for larger amount of cities, we also applied the algorithm to the twenty cities previously used with the Simulated Annealing algorithm. We kept the parameters the same as for the six cities case. As we can see, the distribution of final path lengths seen in Figure 11 looks similar as to what it did for the Simulated Annealing algorithm. The minimum H found with the Parallel Tempering algorithm is higher than the minimum H found with the Simulated Annealing algorithm. Therefore, it is concluded that with the parameters used in this simulation, Parallel Tempering performs worse than Simulated Annealing also for a higher number of cities. The temperatures for every seventh replica is shown in Figure 12, where we see a similar result as what was discussed for the six cities case.

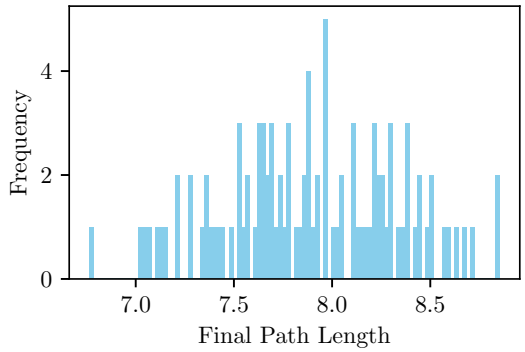

Figure 11: Final path lengths plotted for 100 simulations of 20 cities with the Parallel Tempering algorithm.

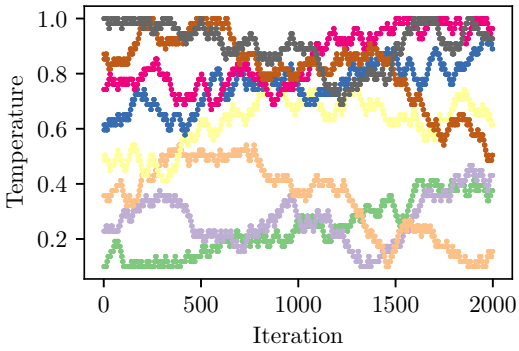

Figure 12: Temperatures for every seventh replica plotted over the iteration steps for 2p cities with the Parallel Tempering algorithm.

5.3 Combined Simulated Annealing and Parallel Tempering

We here introduce a cooling rate of 0.99 to the Parallel Tempering algorithm, using the same 6 cities and 2000 iterations with 50 replicas and the possibility to swap two replica temperatures every 5th iteration. Every 5th iteration, we attempted to swap 30 replica temperatures simultaneously. The results of final path lengths over 100 simulations can be seen in Figure 10. The plot of the temperatures for every seventh replica is also included to illustrate how this combined method works in Figure 11. We saw a slight improvement from the regular Parallel Tempering algorithm, which indicates that including a cooling rate could be sufficient, but no large improvements where made by this combined method.

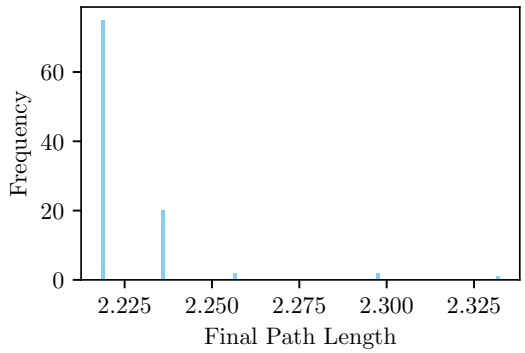

Figure 10: Final path lengths plotted for 100 simulations of 6 cities with the combined Simulated Annealing and Parallel Tempering algorithm.

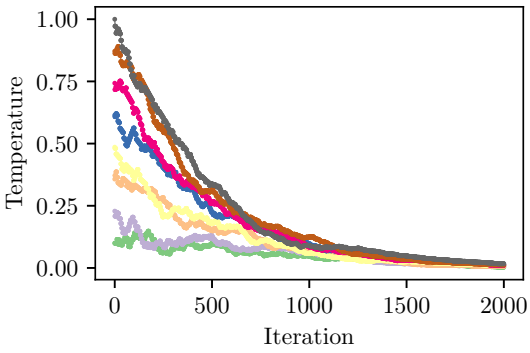

Figure 11: Temperatures for every sixth replica plotted over the iteration steps for 6 cities with the combined Simulated Annealing and Parallel Tempering algorithm.

We did the same for 20 cities, but here used a cooling rate of 0.999. The results can be seen in Figure 12, where we can see that the results are similar to the regular Parallel Tempering algorithm. The temperatures for every seventh replica is plotted in Figure 11, where we see that this slower cooling rate leads to a less drastic drop in temperatures. The system would need more iterations to cool down further, but since the algorithm did not perform better than the regular Parallel Tempering, this was not tested.

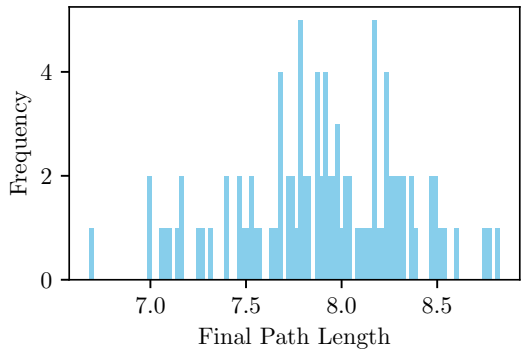

Figure 12: Final path lengths plotted for 100 simulations of 20 cities with the combined Simulated Annealing and Parallel Tempering algorithm.

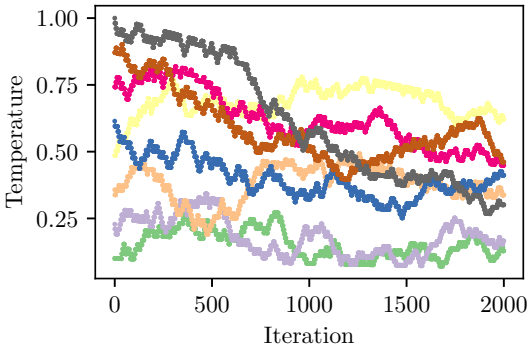

Figure 13: Temperatures for every sixth replica plotted over the iteration steps for 20 cities with the combined Simulated Annealing and Parallel Tempering algorithm.

6 Discussion and Conclusion

When comparing Simulated Annealing with Parallel Tempering, for the cases tested in this project, the Simulated Annealing algorithm is preferred over the Parallel Tempering algorithm due to several reasons. A large factor is the computational efficiency of Simulated Annealing and the intuitiveness of the algorithm. But most important, we also found better results when using Simulated Annealing than Parallel Tempering.

Parallel Tempering has potential, but it is harder to know how to tune the parameters to obtain the best results with the algorithm. The computational cost is very large since there are so many replicas to store and update. If one was able to use even more replicas than tested in this project and maybe use better computers or being able to run replicas simultaneously on different computers, Parallel Tempering could be more useful.

We could see that the performance of Simulated Annealing was very good for a few cities and a slow cooling rate, but introducing larger number of cities quickly made the results hard to interpret since the distribution of Hs found went from the desired exponential distribution to a Gaussian distribution which did not tell us a lot about if we had actually found the optimal city-order or not.

7 Appendix: Code Listings

7.1 Trial moves

```
def swap(cities):
    # Select two random indices
    i, j = np.random.choice(len(cities), size=2, replace=False)
    # Swap the cities at the chosen indices
    cities[i], cities[j] = cities[j], cities[i]
    return cities


def two_opt(cities):
    i = np.random.randint(0, len(cities) - 1)
    j = np.random.randint(i + 1, len(cities))  # Ensure j > i

    # Reverse the portion of the route between i and j
    cities[i:j + 1] = cities[i:j + 1][::-1]

    return cities


def three_opt(cities):
    # Select 3 random indices
    i, j, k = np.random.choice(len(cities), size=3, replace=False)
    i, j, k = sorted([i, j, k])  # Ensure i < j < k for easier slicing

    # Rearrange sections of the route (reverse between the segments)
    new_cities = cities[:i] + cities[i:j+1][::-1] + cities[j+1:k+1][::-1] + cities[k+1:]

    return new_cities
```

7.2 Simulated Annealing algorithm

```
nbOfRuns = 100 # How many independent simulations to run
finalLengths = [] # List to store the final path length after each simulation
nbOfCities = 6

for i in range(nbOfRuns):
    cities = get_cities(nbOfCities)
    # Function returns a list of cities [[x_1,y_1], [x_2, y, 2], ... , [x_6, y_6]]
    # The first city in cities is the first visited and so on
    norm = [calc_norm(cities)] # calc_norm calculates H for the cities
    T_min = 0.0001
    T_initial = 1
    T_list = [T_initial]
    coolRate = 0.99
    it = int(np.log(T_min / T_initial) / np.log(coolRate))
    # We calculate how many iterations is needed to reach T_min

    for i in range(it+1):
        whatmove = np.random.randint(0, 3) #randomly select type of trial move
        if whatmove == 0:
            tempCities = two_opt(cities)
        if whatmove == 1:
            tempCities = three_opt(cities)
        if whatmove == 2:
            tempCities = swap(map)

        tempCitiesNorm = calc_norm(cities)
```

```
        error = tempCitiesNorm - norm[-1]
        # Error is difference in H between new and old cities

        # Metropolis criterion
        if error < 0 or np.random.rand() <= np.exp(-error / T_list[-1]):
            # Trial move accepted
            cities = tempCities
            norm.append(calc_norm(cities))
        else:
            # Trial move rejected
            norm.append(norm[-1])  # If no change, repeat the last norm value

        T_list.append(T_list[-1]*coolRate) # Decrease the temperature after each run

    finalLengths.append(norm[-1]) # Add the H of the iteration to the list of H-values
```

7.3 Parallel Tempering algorithm

```
num_steps = 2000  # How many steps we are simulating for
swap_interval = 10 # How often replicas can swap temperatures

nbOfCities = 6
minimum_lengths = []

nbOfRuns = 100 # How many independent simulations to run

for i in range(nbOfRuns):
    finalLengths = []
    cities_list = []
    temp_order = []
    acceptance_frac = []
    # Parameters
    t_min = 0.1  # Start temperature
    t_max = 1  # End temperature
    n = 20  # Number of temperature points
    temperature_list = [[] for _ in range(n)]

    # Calculate common ratio
    r = (t_max / t_min) ** (1 / (n - 1))

    # Generate temperatures and ensure they do not exceed t_max
    temperatures = np.linspace(t_min, t_max, n).tolist()
    norms = [[] for _ in range(n)]
    for i in range(n):
        cities_list.append(get_cities(nbOfCities))
        random.shuffle(cities_list[i])
        norms[i].append(calc_norm(cities_list[i]))
        temperature_list[i].append(temperatures[i])
        temp_order.append(i)

    succSwaps = 0
    swap_attempts = 10

    for step in range(num_steps):
        # Perform local updates
        for i in range(n):

            whatmove = np.random.randint(0, 3)
            if whatmove == 0:
                tempCities = two_opt(cities_list[i])
            if whatmove == 1:
                tempCities = three_opt(cities_list[i])
            if whatmove == 2:
                tempCities = swap(cities_list[i])
            tempCitiesNorm = calc_norm(tempCities)
            error = tempCitiesNorm - calc_norm(cities_list[i])

```

```
            if error < 0 or np.random.rand() <= np.exp(-error / temperatures[i]):
                cities_list[i] = tempCities
                norms[i].append(calc_norm(cities_list[i]))
            else:
                norms[i].append(calc_norm(cities_list[i]))  # If no change, repeat the last
                    norm value


        # Attempt swaps at intervals
        if step % swap_interval == 0:
            succSwaps = 0 # Possibility to track acceptance rate
            swapped_temps = []  # List of pairs to attempt swaps

            for i in range(swap_attempts):
                # Select a random index for `i`
                i = random.randint(0, n - 1)
                index = temp_order.index(i)

                # Ensure `j` is a neighbor of `i` (directly adjacent)
                if index == 0:
                    j = temp_order[1]  # Swap with the next neighbor
                elif index == len(temp_order)-1:
                    j =  temp_order[-2]  # Swap with the previous neighbor
                else:
                    # Swap with either the previous or the next neighbor
                    j = temp_order[index+1] if random.random() < 0.5 else temp_order[index
                        -1]

                index2 = temp_order.index(j)

                # Compute the swap acceptance probability
                delta = (norms[temp_order[index2]][-1] - norms[temp_order[index]][-1]) * (1
                    / temperatures[temp_order[index2]] - 1 / temperatures[temp_order[index
                    ]])
                P_swap = np.exp(delta)

                # Attempt the swap
                if np.random.rand() <= P_swap and temp_order[index] not in swapped_temps and
                     temp_order[index2] not in swapped_temps:
                    temperatures[temp_order[index]], temperatures[temp_order[index2]] =
                        temperatures[temp_order[index2]], temperatures[temp_order[index]]  #
                         Swap temperatures
                    temp = temp_order[index]
                    temp_order[index] = temp_order[index2]
                    temp_order[index2] = temp
                    swapped_temps.append(temp_order[index])
                    swapped_temps.append(temp_order[index2])
                    succSwaps += 1

            # coolRate can be introduced here for the combined Simulated Annealing and
                Parallel Tempering algorithm
            coolRate = 1
            for i in range(n):
                temperatures[i] = temperatures[i] * coolRate

        # Append the temperatures to track them
        for i in range(n):
            temperature_list[i].append(temperatures[i])

    for i in range(n):
        finalLengths.append(norms[i][-1])

    # We want to use the minimum length for each simulation
    minimum_lengths.append(np.min(finalLengths))
```