Parameterising Simulated Annealing
for the Travelling Salesman Problem

Gary Sun
University of New South Wales

Nov 2021

Abstract

In this paper, we explore the impact of different values of initial temperatures and cooling rates for Simulated Annealing. These values are benchmarked using the Travelling Salesman Problem, to determine their impact on optimality and runtime. Finally, we determine if there is a correlation between the initial temperature and cooling rate with the input cities, and hence a way to parameterise the temperature and cooling rate.

Contents

1 Introduction

1.1 The Travelling Salesman Problem

The travelling salesman is an example of an NP-hard combinatorial optimisation problem.
It goes as follows:

> Given a weighted graph, starting at a vertex, find the minimal cost to travel to all the vertices of the graph once, before returning to the starting vertex.

It has many applications, both in practice and theory, for example in logistics (determining an optimal route of least cost), job scheduling (determining the best way to schedule a series of jobs on a set of machines) and also benchmarking optimisation algorithms

The specific purpose of this project is to explore the "Symmetric" Travelling Salesman Problem, the graph is undirected (the weight along a path / edge is the same regardless of direction). Whilst this halves the number of solutions as compared to the asymmetric TSP, it still has a super-polynomial running time as the number of solutions for n cities is $(n-1)!/2$.

City Count	Path Configurations
3	1
5	12
10	1.81×10^4
20	6.08×10^{16}
30	4.42×10^{30}

Figure 1: Number of path configurations per number of cities

1.2 Algorithms for the Travelling Salesman Problem

Due to the difficult nature of solving the TSP, there are two main categories of algorithms.

- Exact algorithms
- Approximate algorithms

Exact algorithms have the benefit that they can find the optimal solution. For example, the current record for the largest TSP problem contained 85 900 cities and was solved using an ILP formulation [1]. However, they generally suffer from a longer runtime.

On the other hand, approximate algorithms find a near-optimal, whilst running relatively quickly [2]. Hence, in practice, if the optimal is not required, it is suitable to use an approximate algorithm. An example of a common algorithm that can be applied for finding approximate solutions is SA (Simulated Annealing).

Note TSP can be solved efficiently deterministically through cutting plane algorithms, and approximately through the Lin-Kernighan heuristic. The goal of experimenting with SA is mostly out of curiosity and interest, however may reveal insights into the SA algorithm in general.

2 Simulated Annealing

Simulated Annealing is a stochastic global search optimisation algorithm. Whilst greedy searches may get stuck in a local minimum, Simulated Annealing aims to overcome this by probabilistically accepting worse options, and has a higher chance of finding the global minimum.
In the context of TSP, we start with a random solution, and then try different "similar" solutions. We then choose to accept this new "similar" solution based off the SA algorithm, and continue this process, until we have not found better similar solutions.

2.1 Origin

It is based on annealing in metallurgy, where the structure of the material changes randomly rapidly. when the temperature is high. As a result, in the beginning, SA may choose a solution, even if does not lead to an improvement in the objective function.

However, over time as the temperature drops and loses its internal energy, the structure settles into a more stable form. Hence, as the temperature drops, the SA algorithm will with a higher probability discard solutions that do not improve the objective function and always choose those that improve the objective function.

Finally, the resulting material structure reaches its absolute (global) minimum internal energy configuration. Similarly, the SA algorithm will then find an approximate (or the) optimal solution for maximising the objective function.

Note the temperature and cooling rate has to be carefully selected. If the temperature cools too fast, then the process may get stuck at a local minimum. This is similar to what happens in a greedy method, where the search gets stuck in a local minimum. Hence, the introduction of the temperature, and probabilistic acceptance of worse states allows for a higher chance of finding the global minimum.

2.2 Process

Step 1: Choose a suitable initial temperature T_0, cooling rate $r < 1$, a feasible solution $\mathbf{x}^{(0)}$, iteration counter $k = 0$, and the objective function f.

Step 2: Then select a neighbouring solution $\mathbf{x}^{(k+1)}$ to $\mathbf{x}^{(\mathbf{k})}$, where $\mathbf{x}^{(k)}$ is the current solution at iteration k.

Step 3: Calculate $\Delta f = f(\mathbf{x}^{(\mathbf{k+1})}) - f(\mathbf{x}^{(\mathbf{k})})$. We then choose to take $\mathbf{x}^{(k+1)}$ as the new solution with the following probability $P(\Delta f)$

$$P(\Delta f) = \begin{cases} 1 & \text{if } \Delta f > 0, \\ \exp\left(\dfrac{\Delta f}{T_k}\right) & \text{else} \end{cases}$$

Step 4: If we have reached the max iterations, then stop.

Step 5: Else, set $k = k + 1$, and $T_k = rT_k$, then go to Step 2.

2.3 Cooling Rate and Temperature

The initial temperature of the system determines how likely it is to accept worse solutions. As shown in the selection function $\exp(\frac{\Delta f}{T})$ the temperature of the system should be proportional to the possible loss. Hence, **the temperature should be proportional to the values of the loss function** to retain the same probability of choosing a worse configuration. Typically a temperature is chosen that will result in an acceptance ratio of 0.8 [8].

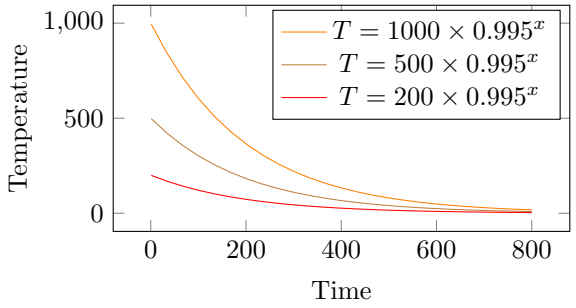


Figure 2: Comparison initial temperatures cooling over time

The cooling rate on the other hand determines the rate of change of the probability that we accept worse configurations. There is research [7] to suggest that a constant temperature can produce optimal more configurations than that with a cooling rate, however this has only been tested for problems with less than 150 cities.

Whilst there are different cooling schedules, geometric cooling tends to perform well compared to other general methods [6]. This is due to high temperatures at the beginning allowing for more searching, whilst low temperatures later allow it to converge to a solution quickly.

Taking inspiration from nature, ignoring other properties, denser materials cool slower. If the TSP behaves like so, **the greater the number of cities, the slower the system should cool** as it is "denser" in the number of cities. Furthermore, if the number of cities is greater, there may also more local minimums, which can be mitigated by slower cooling.

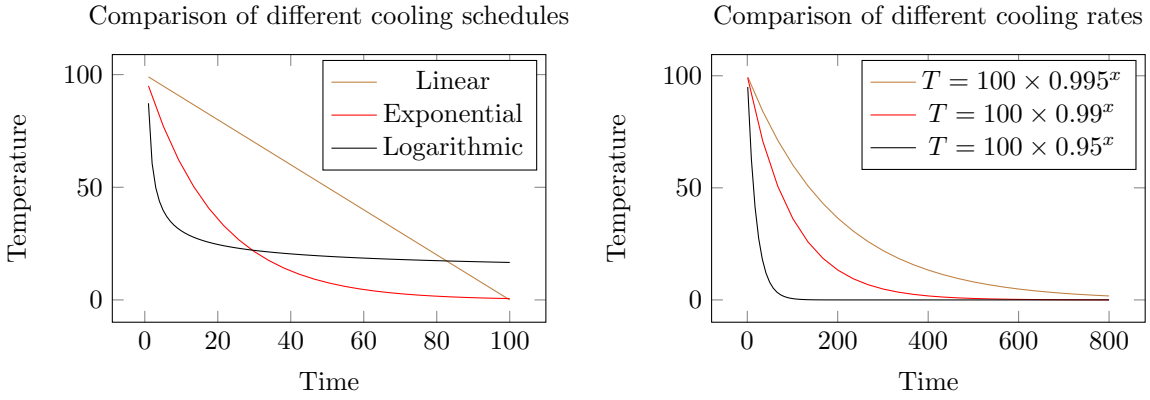


Figure 3: Examples of different cooling schedules and rates where $r = 0.995$

3 Implementation

3.1 Controlled Variables

3.1.1 Objective Function

Given a set of n cities, where $c_{i,j}$ represents the cost of travelling between i and j, we aim to find $O = \langle o_1, o_2, ..., o_n \rangle$ such that we

$$\text{minimise} \left(\sum_{1 \leq i < n} c_{o_i, o_{i+1}} + c_{n,1} \right)$$

where $o_i = j$ means that city j is in position i of the path.

3.1.2 Selection Probability

Although dependent on the temperature, the selection function penalises the selection of a local possible path configuration proportional to how much worse it is than the current path.
To do this, we use the Boltzmann Distribution as proposed by Metropolis [3],

$$p_i \propto \exp\left(\frac{-\epsilon_i}{kT}\right),$$

where

- p_i is the probability of choosing the state i,
- ϵ_i is the energy of the state (in our model, it is the cost of going from our state to the next)
- k is Boltzmann's constant (ignored in our model)
- T is the current temperature of the system

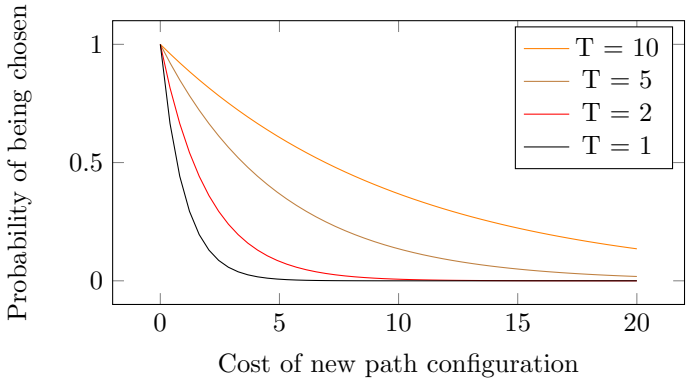


Figure 4: Relationship between change in path cost and probability of selection

We can see that the greater the temperature, the higher the probability of choosing a worse configuration. However, the greater the loss, the exponentially lower the probability of acceptance.

3.1.3 Local Search (2 opt)

It is an important part of the algorithm to determine an algorithm to search for nearby possible solutions. A well-known method is the **two-opt** method. This involves selecting two random cities, re-orders the route between each other so that it does not.

The example below shows before ($A \to B \to D \to C \to E \to F$) on the left, and after reversing the order between nodes B and E ($A \to B \to C \to D \to E \to F$) on the right.

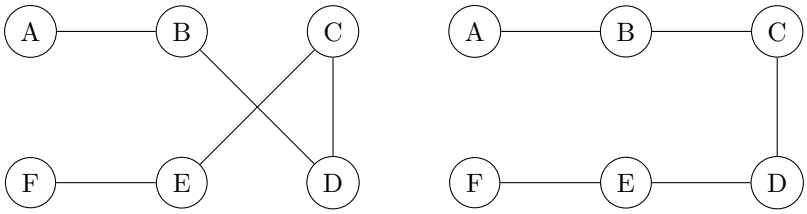


Figure 5: two-opt between B and E

In the above scenario, the "before" distance will be shorter than the "after" distance if

$$\text{distance}(B, D) + \text{distance}(C, E) < \text{distance}(B, C) + \text{distance}(D, E)$$

which can be calculated in $O(1)$, whilst applying the two-opt runs in $O(n)$. Hence, in the implementation, the loss of the new path configuration is calculated first before the path is changed. Note that a more optimal method is the LKH (Lin–Kernighan heuristic), however, this is largely a variation of the two-opt and three-opt methods. However, because of its difficulty in implementation [2], and the main focus is on the temperature and cooling rate, only two-opt been implemented.

3.1.4 Max Iterations

There also has to be a way of determining when the solver has found what it considers the optimal solution. Currently we allows SA to continue until it iterated `max_iterations` times without finding a better solution. From initial tests, it was possible for a slow cooling rate to have a very optimal solution before converging and then not find a better solution for many iterations [1].

```
max_iterations = 50 // (1 - cooling_rate)
```

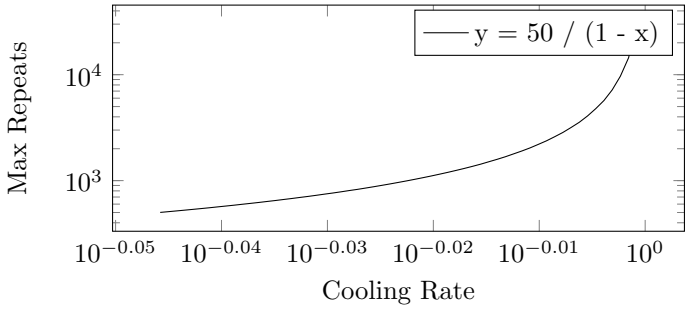


Figure 6: Maximum repeats for each cooling rate

[1] In hindsight, this should have been a function of the number of cities, as it would have allowed for longer iterations for greedy solutions

3.2 Parameterisation

We need to find a measure of the input that can correlate with the Cooling Rate and Temperature. Ideally, these measures can be calculated quickly in polynomial time or better so that they can be found and then used to parameterise the Simulated Annealing algorithm before running it.

3.2.1 Cooling Rate

As the cooling rate allows for more exploration of nearby path configurations in the local search to prevent being stuck in a local minimum, we need a measure of "how likely it is to be stuck in a local minimum".
Whilst there may be other methods, the **number of cities will be used**, and finding it is as simple as the code below, which runs in $O(1)$ in python or at max $O(n)$ in a naive implementation.

```
def get_num_cities(cities: list[tuple[int, int]]) -> float:
    return len(cities)
```

Hence, in the tests, different cooling rates will be tested on maps with different number of cities. Their optimality and iteration count will then be compared to determine a suitable cooling rate based off the number of cities, where the other variables such as the average difference in distance of cities is the same.

3.2.2 Temperature

The temperature is relevant for determining the probability of accepting a worse configuration, as the loss from a worse configuration is divided by the temperature to find the probability. As a result, the temperature needs to be high enough to accept worse configurations so that the search space is reduced, but not too high that the number of iterations is too great.

Hence, it may be possible to parameterise the temperature with the average distance between cities. However, initial tests showed zero better optimality from higher temperatures for higher city distances. This is because it does not take into consideration the loss function:
Given the city coordinates `cities`, the ordering of the path `order` (where the first value is the first city to visit, and the second value the second city to visit, etc), return the change in total path length after applying two-opt between the city `a` and the city `b`. This is implemented as:

```
from math import dist

def loss(cities: list[tuple[int, int]], order: list[int], a: int, b: int) -> float:
    n = len(cities)
    a1, a2 = cities[order[a]], cities[order[(a + 1) % n]]
    b1, b2 = cities[order[b]], cities[order[(b + 1) % n]]
    curr_dist = dist(a1, a2) + dist(b1, b2)
    swap_dist = dist(a1, b1) + dist(a2, b2)
    return swap_dist - curr_dist
```

Hence, the return value of the `loss` function is proportional to the **difference in distances between the cities**.

It has been suggested to take $T_0 = \Delta E_{\max}$ where $\Delta E_{\max}$ is the maximal cost difference in Kirkpatrick et al [4]. Another method described in Kirkpatrick et al [4] is to choose T_0, and then perform a number of transitions, which will result in ratio of accepted probabilities x_0.

A similar formula proposed by Johnson et al is to set the initial temperature $T_0 = \frac{\overline{\Delta E}}{\ln(X_0)}$ where $\overline{\Delta E}$ is an estimation of the cost of positive transitions. Given that X_0 is often to be a value of around $0.8 - 0.9$, and $\ln(0.8)^{-1} \approx 5$, and $\ln(0.9) \approx 10$, the initial temperature may ideally be $\times 5$ to $\times 10$ times larger than the average positive transition cost.

Whilst, there are several methods to calculate the average positive transition cost [8], they rely on samples from running SA multiple times. The following code details how the average difference in distances (which is also the average positive transition cost) can be found in $O(n^2 \log n)$ using a dynamic programming (sliding window) approach:

```
from math import dist

def get_diff_city_dist(cities: list[tuple[int, int]]) -> float:
    n = len(cities)
    d = sorted([dist(cities[i], cities[j]) for i in range(n-1) for j in range(i+1,n)])
    m = len(d)

    prev_diff = sum([x - d[0] for x in d[1:]])
    total_diff = prev_diff
    for i in range(1, m):
        prev_diff = prev_diff - (d[i] - d[i - 1]) * (m - i)
        total_diff += prev_diff

    return total_diff / ((m * (m - 1)) / 2)
```

Hence, in the tests, different temperatures will be tested on maps with different differences in the distances. The maps will be generated with n cities, and their positions scaled to adjust the distances. Their optimality and iteration count will then be compared to determine a whether a suitable temperature based off the number of cities can be chosen.

3.3 Code

The code is viewable online at https://github.com/angary/simulated-annealing-tsp.

For the implementation there main code can be found in the `src/` folder, where the following files have the following purpose:

- `src/solvers.py`

  This contains classes that extend an abstract class `Solver` which contains abstract methods `get_next_order()` which runs one iteration of the solving algorithm, returning the current path it has found (used in the visualisation). It also contains the `SimulatedAnnealing` class, which takes in optional parameters `temperature` and `cooling_rate`.

- `src/main.py`

  This contains the code for running a visualisation of a problem from a file or randomly generated cities. It also has a method `solve()` which runs `get_next_order()` until the algorithm has determined that is has finished.

- `src/benchmark.py`

  This contains the code to benchmark the `SimulatedAnnealing` solver. It can either load in problems from a file or generate random problems (as detailed later in the report) and tests the `SimulatedAnnealing` solver with different options for `temperature` and `cooling_rate`. The results are then stored in the `results/` folder.

- `src/setup.py`

  This contains the code for reading in the cities from a file in TSPLIB format. It also contains functions for determining the metrics such as average difference in city distances used for parameterising the Simulated Annealing algorithm.

- `src/config.py`

  This contains some configuration variables for both the visualisation and the testing. The testing variables are kept here to prevent import path issues when trying to access them from the Jupyter Notebooks.

The `data/` folder contains different problems taken from the TSPLIB library, where `.tsp` files contain problem instances, and `.opt.tour` files contain the optimal tour for the problem.

The `report/` folder contains Jupyter Notebooks for visualising the results and also the source code for this report.

4 Experimentation

4.1 TSP Dataset Test

First, the SA algorithm was benchmarked with TSPLIB [5] (a public library for TSP instances) with different initial temperatures and cooling rates. The problems sets that were used are listed below

```
a280.tsp      berlin52.tsp  ch150.tsp     pcb442.tsp    pr2392.tsp    tsp225.tsp
att48.tsp     ch130.tsp     gr666.tsp     pr1002.tsp    rd100.tsp     ulysses22.tsp
```

To conduct a test, the SA algorithm would be benchmarked, with a temperature and cooling rate of 0. Then it would be run again with different combinations of temperature and cooling rates. All of these tests would then be repeated multiple times as each run of SA can produce very different results. The algorithm was then determined to have "finished" after it did not find a better solution after 1000 iterations.

The following variables were used:

```
TEST_REPEATS = 20
TEMPERATURES = [10, 50, 100, 500, 1_000, 5_000]
COOLING_RATES = [0.999, 0.999_5, 0.999_9, 0.999_95]
```

where `TEST_REPEATS` is the number of times each test was repeated, `TEMPERATURE` is the initial temperature, and `COOLING_RATE` is how much the temperature decreases each iteration.

4.1.1 Results

Cooling Rate	Mean Optimality %	Optimality Std Dev	Mean Iterations
0	0.7206	0.2151	69942
0.999	0.7190	0.2151	71486
0.9995	0.7214	0.2165	73394
0.9999	0.7388	0.2263	94316
0.99995	0.7549	0.2287	125381

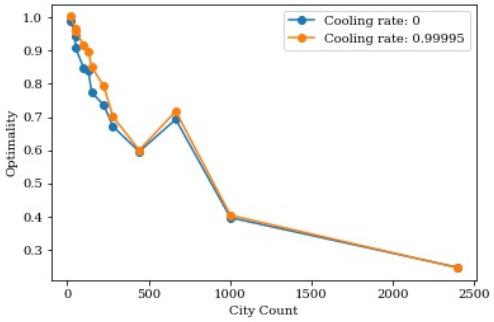
Figure 7: Results of different cooling rates of the TSPLIB problems

Temperature	Mean Optimality %	Optimality Std Dev	Mean Iterations
0	0.7206	0.2151	69942
10	0.7236	0.2171	74344
50	0.7343	0.2207	81645
100	0.7341	0.2230	84811
500	0.7381	0.2233	95380
1000	0.7367	0.2245	99195
5000	0.7343	0.2245	111490

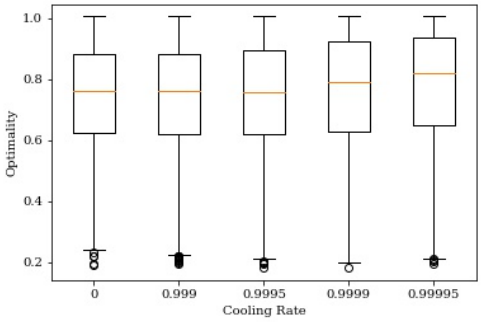
Figure 8: Results of different initial temperatures of the TSPLIB problems

Name	City Count	Avg. City Distance	Avg. Iterations	Avg. Optimality %	Optimality Std. Dev. %
ulysses22.tsp	22	4	84003	0.9991	0.0061
att48.tsp	48	1608	17216	0.9576	0.0224
berlin52.tsp	52	282	29205	0.9342	0.0359
rd100.tsp	100	275	32775	0.8780	0.0446
ch130.tsp	130	177	39815	0.8576	0.0437
ch150.tsp	150	178	40381	0.8087	0.0470
tsp225.tsp	225	91	56350	0.7502	0.0466
a280.tsp	280	61	68501	0.6757	0.0424
pcb442.tsp	442	872	78402	0.5924	0.0439
gr666.tsp	666	44	133338	0.6904	0.0482
pr1002.tsp	1002	3215	156407	0.4064	0.0295
pr2392.tsp	2392	3186	347207	0.2458	0.019

Figure 9: Results of different TSPLIB problems

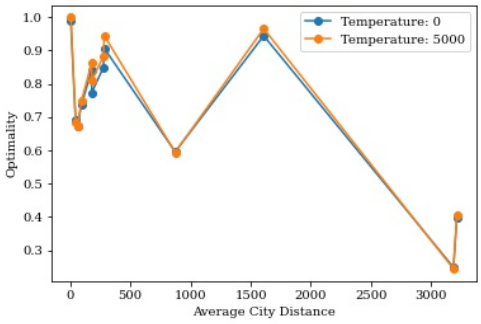


(a) Cooling rate optimality for city count

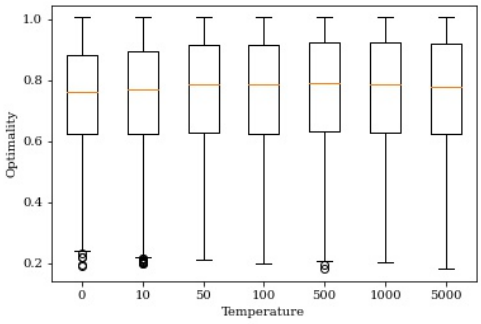


(b) Optimality for each cooling rate

Figure 10: Comparison between optimality, city counts, and cooling rate



(a) Temp optimality for city distance



(b) Optimality for each initial temperature

Figure 11: Comparison between optimality, city distances and temperatures

4.1.2 Discussion

From these results, there are a couple of observations that we can make

- There is a negative exponential correlation between city count and temperature, however, there was a lack of correlation between city distance and optimality
- The higher the cooling rate, the better the optimality, however, there is no benefit to constantly increasing the temperature
- There was a logarithmic trend between cooling rate and iterations, and also a logarithmic trend between initial temperature and iterations

However, some issues need to be addressed

- gr666.tsp is an outlier, is a dense cluster of cities, whilst the rest are sparsely scattered, resulting in higher average optimality as despite perhaps having a bad path configuration - because so many cities were close, the change in distance was small)
- since optimality has a very small correlation with optimality, we cannot tell if a certain temperature increases the optimality for an average city distance unless there is the same number of cities

To get more meaningful results, there needed to be a greater increase in the range of the cooling rate to determine the limit of a higher cooling rate. The problem set used for the TSP problem also needs to be better controlled, as the impact of temperature on optimality for different temperatures could not be determined as optimality was more dependent on the number of cities.

4.2 Random Dataset Test

Rather than experimenting with well known TSP problems such as those found in TSPLIB, the SA was tested using randomly generated cities. The reasoning is that variables such as distances or city count can be controlled well whilst the other values are adjusted.

As a result, specific maps could be generated for testing temperature (where the city count was kept the same, but differences in distance increased) and maps could be generated for testing cooling rate (where differences in distance was kept the same, but the city count was increased).

To conduct the tests, the following variables were used:

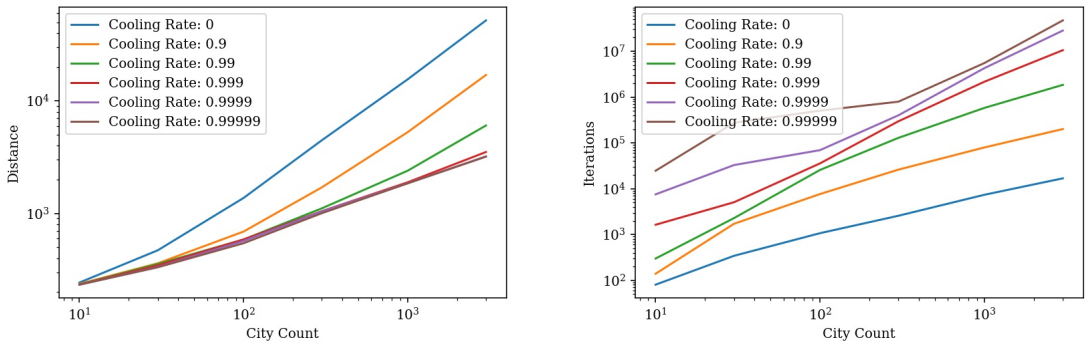
```
MAP_COUNT = 4
TEST_REPEATS = 20
DIST_DIFFS = [1, 10, 100, 1_000, 10_000]
CITY_COUNTS = [10, 30, 100, 300, 1_000, 3_000]
CONST_CITY_COUNT = 400
CONST_DIST_DIFF = 20

TEMPERATURES = [0, 1, 10, 100, 1_000, 10_000]
COOLING_RATES = [0, 0.9, 0.99, 0.999, 0.999_9, 0.999_99]
CONST_TEMPERATURE = 40
CONST_COOLING_RATE = 0.9999
```

where

- `MAP_COUNT` was the number of different generated maps for each test
- `TEST_REPEATS` was how many times a test would be repeated with the same map, initial temperature and cooling rate
- `DIST_DIFFS` is an array of the difference in distances between cities, used with `CONST_CITY_COUNT` to generate `MAP_COUNT` maps for testing temperature
- `CITY_COUNTS` is an array of the different number of cities, used with `CONST_DIST_DIFF` to generate `MAP_COUNT` maps for testing cooling rate
- `TEMPERATURES` is an array of the different initial temperatures used, whilst `CONST_COOLING_RATE` was the cooling rate that was kept constant across the temperature tests
- `COOLING_RATES` is an array of the different cooling rates used, whilst `CONST_TEMPERATURE` was the temperature that was kept constant across the cooling rate tests

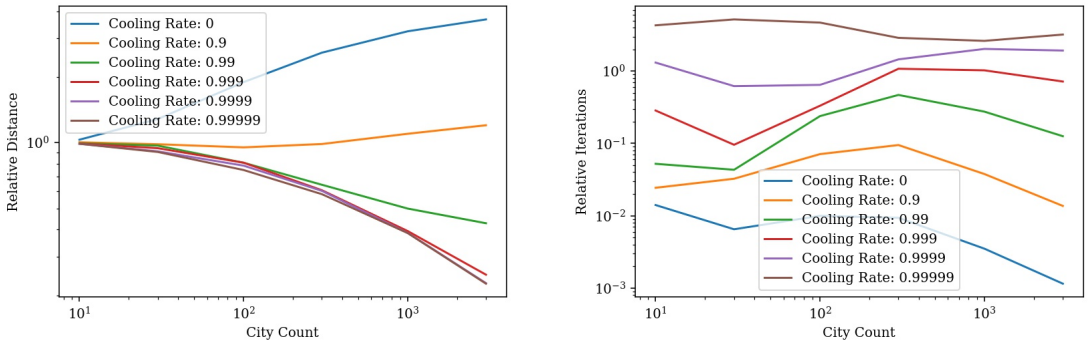
4.2.1 Results

There was a very good trend between the a slower cooling rate and a more optimal solution, and more iterations.



(a) Absolute final distances (b) Absolute iterations

Figure 12: Absolute comparison between final distances, and iterations, for different r

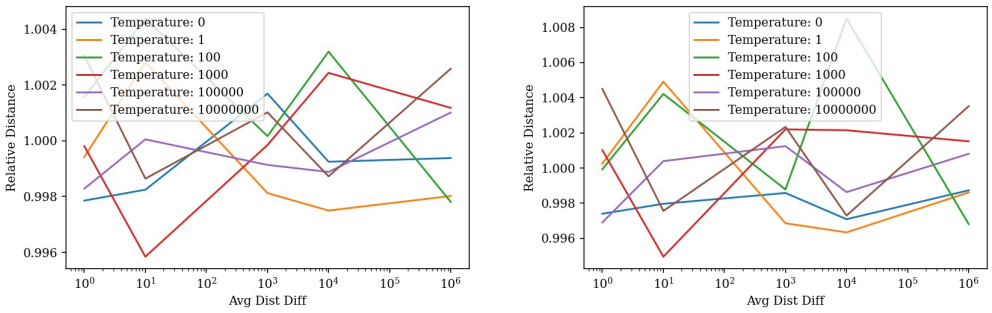


(a) Relative final distance (b) Relative iterations

Figure 13: Relative comparison between final distances, and iterations, for different r

An initial test was conducted for different T_0 and average difference in distances for $r = 0.995, n = 400$. This resulted in little to no trend between the final distance. Hence, another test was conducted where $r = 0.995, n = 800$.

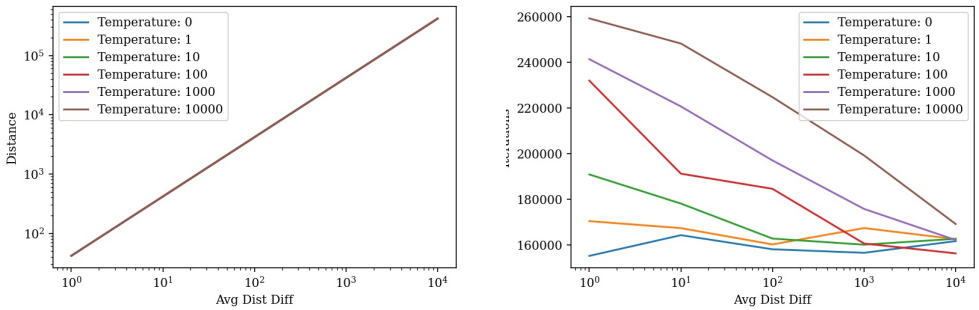
Both these tests resulted in negligible trends, and their results are shown below.



(a) Relative final distance, where $n = 400$ (b) Relative final distance, where $n = 800$

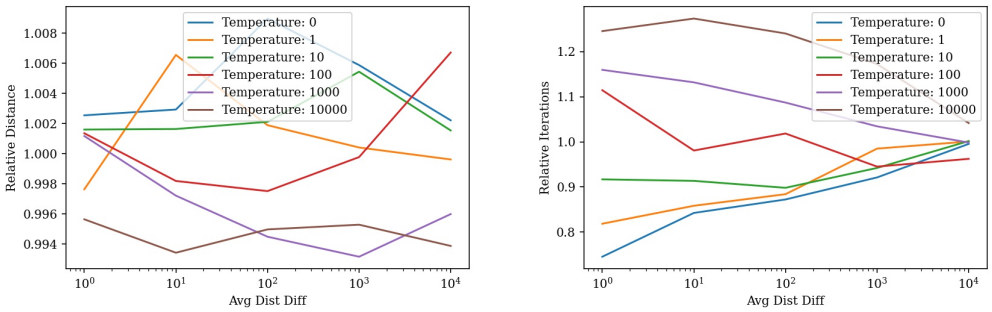
Figure 14: Relative comparison between final distances, and iterations, for different T_0

The reasoning for this was assumed to be the fast cooling rate, meaning that the temperature cooled too fast towards a low level, essentially rendering the initial temperature irrelevant. Hence, another test was conducted, where $r = 0.9999$, and $n = 400$.



(a) Final distance for different (b) Optimality for each cooling rate

Figure 15: Absolute comparison between final distances, and iterations, for different T_0



(a) Final distance for different (b) Optimality for each cooling rate

Figure 16: Relative comparison between final distances, and iterations, for different T_0

4.2.2 Discussion

Cooling Rate

There was a clear trend between a slower cooling rate and a greater relative optimality, and this effect seemed to increase the greater the city count. Note r is the cooling rate and n is the number of cities.

	$n = 10^1$	$n \approx 10^{1.5}$	$n = 10^2$	$n \approx 10^{2.5}$	$n = 10^3$	$n \approx 10^{3.5}$
$r = 0$	1.000	1.000	1.000	1.000	1.000	1.000
$r = 0.9$	1.027	1.304	1.989	2.619	2.946	3.052
$r = 0.99$	1.038	1.323	2.339	4.024	6.480	8.574
$r = 0.999$	1.041	1.360	2.338	4.263	8.213	14.74
$r = 0.9999$	1.044	1.406	2.411	4.292	8.374	16.12
$r = 0.99999$	1.044	1.415	2.525	4.443	8.391	16.21

Figure 17: Table comparison between cooling rate and iterations

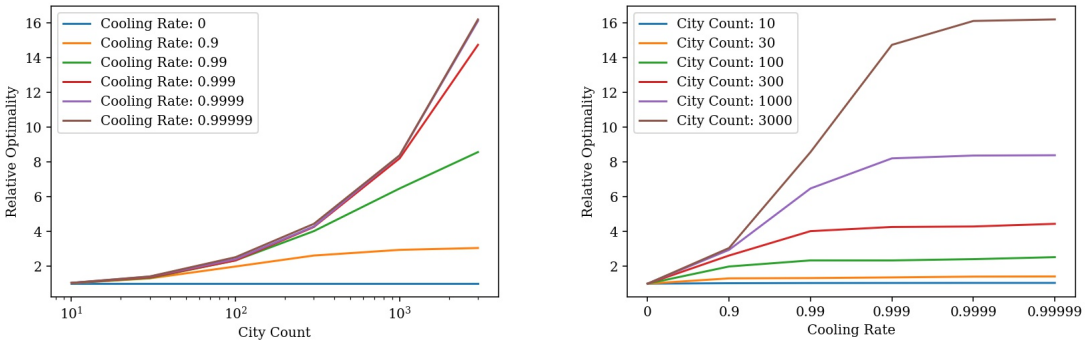


Figure 18: Plot comparison between cooling rate and iterations

Overall, having a slower cooling rate allowed us to find a more optimal solution. This increase in optimality would be even more pronounced for greater city counts - as it was likely that a greedy solution on average would get stuck in a local minimum.

However, there seemed to be a limit of how much better a Simulated Annealing solution could be compared to a greedy method, and this limit would be proportional to the city count.

Overall this trend follows that of a logistic curve, where if we take $r' = -\log_{10}(1-r)$, i.e. if $r = 0.9$, then $r' = 1$, if $r = 0.99$, then $r' = 2$, and so on, the relative optimality could be determined by the function

$$f(r', n) = \frac{L}{1 + e^{-k(r'-r'_0)}}$$

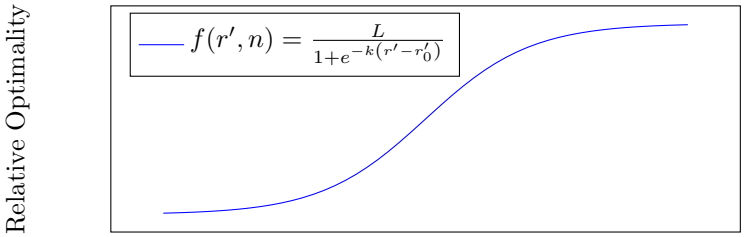


Figure 19: Example of a logistic curve

where

- L is the curve's maximum value.
  Whilst this is an unreachable asymptote, it's value for a city count above 30 can be predicted to be
  $$L \approx 0.45 + 4^{\log_{10}(n)-1.5}$$
- r'_0 is the r' value of the sigmoid point.
  As a result, $r'_0 = r'_1/2$ if at cooling rate r'_1 we reach close to the highest possible optimality (or the asymptote), and so is around
  $$r'_0 \approx \log_{10}(n) - 1$$
- k is the logistic growth rate or steepness of the curve.
  We need to ensure that at a cooling rate of 0 will result in a relative optimality of 1, i.e. $f(r = 0, n) = 1$ and hence

  $$1 = \frac{L}{1 + e^{-k(r'-r'_0)}} \approx \frac{0.45 + 4^{\log_{10}(n)-1.5}}{1 + e^{-k(-\log_{10}(n)-1)}}.$$

  Therefore,

  $$\begin{aligned} 1 + e^{-k(-\log_{10}(n)-1)} &\approx 0.45 + 4^{\log_{10}(n)-1.5} \\ -k(-\log_{10}(n) - 1) &\approx \ln(4^{\log_{10}(n)-1.5} - 0.55) \\ k &\approx \frac{\ln(4^{\log_{10}(n)-1.5} - 0.55)}{\log_{10}(n) - 1}. \end{aligned}$$

This can be pieced together to form a prediction of the relative optimality based off the cooling rate and city count.

More importantly though, this can be used to determine the point at which a slower cooling rate results in a lesser increase in optimality. Whilst the inflection point of the curve can be used (which is also r'_0), it tends to be around half of the optimal value that can be achieved. Another value that can be used is finding the value of r' that produces the lowest value in the second derivative (this value also represents the value of the "maximal growth period").

That being said, these calculations are highly dependent on the experimental results and resulted in very convoluted equations. Something more suitable may have been an approximation that produces around $80 - 90\%$ of the optimal solution. This value tended can be determined as

$$r \approx 1 - 100^{-log_{10}(n)+1}.$$

The number of iterations increased linearly with the city count where,

$$\text{if } r = 0, \text{ iterations} \approx 10n.$$

There was a clear trend between a slower cooling rate and an increase in the number of iterations. This is because with a slower cooling rate, the temperature remains higher for longer. For example, if the starting temperature was T_0, the following number of iterations are required to reach a portion of the initial temperature.

Cooling Rate	Iterations to reach $0.1T_0$	Iterations to reach $0.01T_0$
0.9	22	44
0.99	230	459
0.999	2302	4603
0.9999	23025	46050
0.99999	230258	460515

Figure 20: Number of iterations required to lower the temperature by a certain percent

Cooling Rate	Relative Iterations	Increase from prev.
0	1	N/A
0.9	6	6
0.99	27	4.5
0.999	80	2.96
0.9999	180	2.25
0.99999	516	2.87

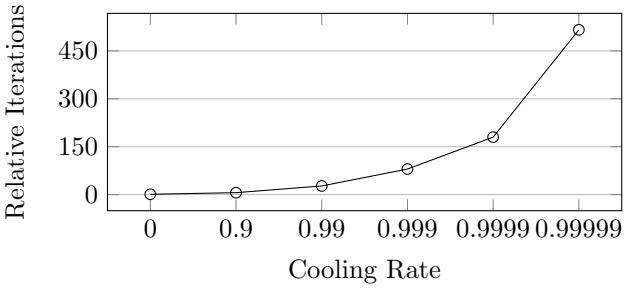


Figure 21: Raw comparison between cooling rate and iterations from experiments

Whilst there aren't enough points to verify, if the relative iterations grow polynomially or exponentially, it can be determined that the increase in iterations scales faster than the optimality.

Hence, at least for cooling rates in the range $0 - 0.99999$, you do not receive a greater increase in optimality relative to the increase in iterations from slowing the cooling rate.

Interestingly, the relative increase in iterations from the experimental results for each cooling rate are lower than those required for the temperature to reach a certain percentage of it's initial temperature. This suggests that the solution starts to converge before at different temperatures for different cooling rates.

Temperature
Relative optimality in the final distance by changing the temperature was mostly negligible compared to changing the cooling rate. However, each T_0 typically produced it's best final distance, if it was equal to the average difference in distances between the cities. Higher temperatures also tended to provide better results, however they also resulted in a greater number of iterations.

	$\Delta d = 10^0$	$\Delta d = 10^1$	$\Delta d = 10^2$	$\Delta d = 10^3$	$\Delta d = 10^4$
$T_0 = 0$	1.0026	1.0029	1.0090	1.0059	1.0022
$T_0 = 10^0$	0.9976	1.0066	1.0019	1.0004	0.9996
$T_0 = 10^1$	1.0016	1.0016	1.0021	1.0054	1.0016
$T_0 = 10^2$	1.0014	0.9982	0.9975	0.9998	1.0067
$T_0 = 10^3$	1.0012	0.9972	0.9945	0.9932	0.9960
$T_0 = 10^4$	0.9957	0.9934	0.9950	0.9953	0.9939

Figure 22: Table comparison between relative final distance and temperature

Overall, a better conclusion may have been determined if there were more tests, though there is already previous research to suggest that a value of T_0 relative to the average loss is ideal [8].

Interestingly, if the temperature was very high relative to the average difference in distance between the cities, it underwent a lot of iterations. However, as the average distance between cities increased, the number of iterations decreased, despite the number of cities remaining the same.

This is because when the temperature is too high, a worse configuration is more or less always accepted. As a result, it does not converge to a solution. By subtracting the iterations from the highest initial temperature $T_0 = 10000$, from the greedy solution, we can determine at which temperature the solution converges.
The table below shows for each column, the temperature after the number of extra iterations it took for the initial temperature to find the solution compared to the greedy solution. Only $T_0 = 10^4$, and $T_0 = 10^3$ was used as these temperatures had the more stable results.

Δd	$T_0 = 10^3$	$T_0 = 10^4$
10^0	0.1805	0.3019
10^1	3.5453	2.2511
10^2	20.409	12.731
10^3	146.05	494.62
10^4	952.85	4725.4

Figure 23: Temperature after which the number of iterations was the same as $T_0 = 0$

The key takeaway is that the point at which the solution started converging was on the same magnitude as the average distance between the cities.
Another note is that the impact of the temperature is amplified with a slower cooling rate, as a slow cooling rate allows for the initial tmemperature to be sustained longer.

5 Conclusion

Whilst there are faster and more optimal algorithms for TSP, we noted that the cooling rate could be parameterised off the number of cities, and the temperature by the average distance between the cities.

Overall the cooling rate had a much bigger impact than the temperature on both the final distance found, and the number of iterations. Slower cooling rates and higher initial temperatures often resulted in lower final distances, however also resulted in high iterations and hence solving time.

Whilst slower cooling rates resulted in better solutions, there was a limit of how much better it could be relative to the greedy solution. Hence, there was a point at which slowing the cooling rate more resulted in a negligible increase in optimality. Whilst not experimented with, this is likely also the case with increasing the temperature, however, there is little purpose for doing so as there is comparatively small benefit from increasing the temperature.

A suitable value of the cooling rate that provided around 90% of the most optimal solution could be calculated as

$$r \approx 1 - 100^{-\log_{10}(n)+1}.$$

A suitable value of the temperature was one that was proportional to the average difference in distance of the cities. This could be calculated in $O(n \log n)$ using a sliding window approach in the following pseudo code:

```
from math import dist

def get_diff_city_dist(cities: list[tuple[int, int]]) -> float:
    n = len(cities)
    d = sorted([dist(cities[i], cities[j]) for i in range(n-1) for j in range(i+1,n)])
    m = len(d)

    prev_diff = sum([x - d[0] for x in d[1:]])
    total_diff = prev_diff
    for i in range(1, m):
        prev_diff = prev_diff - (d[i] - d[i - 1]) * (m - i)
        total_diff += prev_diff

    return total_diff / ((m * (m - 1)) / 2)
```

Unfortunately the runtime was not tested the experimentation as it may have been sensitive to the testing environment. However, it would be interesting to see the runtime relative to the rest of the parameters, as may not scale exactly with iterations. This is because the runtime of an iteration would be $O(n)$ if it accepts a new configuration, else $O(1)$. As a result, this would also give insight into details such as how the probability of accepting a different configuration changes with temperature, and the current optimality of the solution. Another important note discovered towards was that the `max_repeats` has a significant impact on the optimality of a solution.

6 References

[1] Cook, William J. (2012). *In Pursuit of the Traveling Salesman: Mathematics at the Limit of Computation.* Princeton, NJ: Princeton UP

[2] K. Helsgaun, (1998). *An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. DATALOGISKE SKRIFTER (Writings on Computer Science), No. 81* Roskilde University.

[3] N.A. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. (1953) *Equation of state calculations by fast computing machines* J. Chem. Phys, vol. 21, pp. 1087–1092

[4] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, "Optimization by simulated annealing," Science, vol. 220, pp. 671–680, 1983

[5] G. Reinelt, (1991). *TSPLIB - A Traveling Salesman Problem Library",* ORSA J. Comput., 3-4, 376-385.

[6] Mahdi, Walid & Medjahed, Seyyid Ahmed & Ouali, Mohammed. (2017). *Performance Analysis of Simulated Annealing Cooling Schedules in the Context of Dense Image Matching.* Computación y Sistemas. 21. 493-501.

[7] Fielding, M. (2000). *Simulated Annealing With An Optimal Fixed Temperature.* SIAM J. Optim., 11, 289-307.

[8] Ben-Ameur, Walid. (2004). *Computing the Initial Temperature of Simulated Annealing. Computational Optimization and Applications.* 29. 369-385.