Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 2, 842 16 Bratislava 4

Zadanie 3 – Travelling Salesman Problem with Simulated Annealing

Olivér Izsák

Študijný program:Aplikovaná Informatika
Ročník: 2

Krúžok: Utorok 16:00
Predmet: Umelá inteligencia
Cvičiaci: Ing. Boris Slíž
Ak. rok: 2021/2022

Assignment description

Original:

Úloha

Obchodný cestujúci má navštíviť viacero miest. V jeho záujme je minimalizovať cestovné náklady a cena prepravy je úmerná dĺžke cesty, preto snaží sa nájsť najkratšiu možnú cestu tak, aby každé mesto navštívil práve raz. Keďže sa nakoniec musí vrátiť do mesta z ktorého vychádza, jeho cesta je uzavretá krivka.

Zadanie

Je daných aspoň 20 miest (20 – 40) a každé má určené súradnice ako celé čísla X a Y. Tieto súradnice sú náhodne vygenerované. (Rozmer mapy môže byť napríklad 200 * 200 km.) Cena cesty medzi dvoma mestami zodpovedá Euklidovej vzdialenosti – vypočíta sa pomocou Pytagorovej vety. Celková dĺžka trasy je daná nejakou permutáciou (poradím) miest. Cieľom je nájsť takú permutáciu (poradie), ktorá bude mať celkovú vzdialenosť čo najmenšiu.

Výstupom je poradie miest a dĺžka zodpovedajúcej cesty.

Simulované žíhanie (simulated annealing)

Simulované žíhanie patrí do skupiny algoritmov, ktoré využívajú na hľadanie riešenia v priestore možných stavov lokálne vylepšovanie. Zároveň sa algoritmus snaží zabrániť uviaznutiu v lokálnom extréme. Z aktuálneho uzla si algoritmus klasicky vytvorí nasledovníkov. Potom si jedného vyberie. Ak má zvolený nasledovník lepšie ohodnotenie, tak doň na 100% prejde. Ak má nasledovník horšie ohodnotenie, môže doň prejsť, ale len s pravdepodobnosťou menšou ako 100%. Ak ho odmietne, tak skúša ďalšieho nasledovníka. Ak sa mu nepodarí prejsť do žiadneho z nich, algoritmus končí a aktuálny uzol je riešením. Pre nájdenie globálneho extrému je dôležitý správny rozvrh zmeny pravdepodobnosti výberu horšieho nasledovníka. Pravdepodobnosť je spočiatku relatívne vysoká a postupne klesá k nule.

Problém je opäť reprezentovaný vektorom, ktorý obsahuje index každého mesta v nejakom poradí (nejaká permutácia miest). Nasledovníci sú vektory, v ktorých je vymenené poradie niektorej dvojice susedných uzlov.

Dôležitým parametrom tohto algoritmu je rozvrh zmeny pravdepodobnosti výberu horšieho nasledovníka. Príliš krátky (rýchly) rozvrh spôsobí, že algoritmus nestihne obísť lokálne extrémy, príliš dlhý rozvrh natiahne čas riešenia, lebo bude obiehať okolo optimálneho riešenia. Je potrebné nájsť vhodný rozvrh.

Dokumentácia musí obsahovať opis konkrétne použitého algoritmu a reprezentácie údajov. Dôležitou časťou dokumentácie je zhodnotenie vlastností vytvoreného systému a opis závislosti jeho vlastností na rozvrhu. Použite aspoň dva rôzne počty miest (napr. 20 a 30).

The realization of the problem

The travelling salesman problem is one of the most popular problems in graph theory, in theoretical computer science and in combinatorial optimization.
Generally to solve the problem one would have to try all combinations of cities which can take a long time to compute since its time complexity is O(n!), which in the case of 20 cities mean 20!. So in computation a lot of algorithms have been created to solve this problem more efficiently.One of

these algorithms is called Simulated Annealing which is a metaheuristic algorithm that can be used for finding the global optimum solution

The reason why this Simulated Annnealing (SA) is useful is that it can avoid being stuck in a local optimum in order to find the global optimum.
This is thanks to the way SA algorithm work, which is it always takes the path with the minimum length, but sometimes randomly it will take a path that is greater than then the current minimum length. This way it can reach other permutations which might have an even smaller length.

The algorithm has 3 important variables the Temperature, cooling factor, and probability. The temperature is being reduced everytime a path has been found, the reduction rate is descried by the cooling factor. Meanwhile the probability is responsible for calculating the chance whether to replace the previous path which is lesser than the currently found path. This probability is calculated using the temperature, the difference between the current and the previous path.
As the temperature gets lower and lower, the chance for switching a good path to a worst one gets lower. Also the greater the path difference the chances of replacing is also expontentially lower.

For my implementation I used 3 classes for my solution:

Coordinates

These are the coordinates for the cities.
Contains x and y coordinates and a string containing the name of the city.
This class also has a method called calculate distance which uses the pythagoras theorem to calculate the distance between the current city and another one specified in the arguments of the function.

Sequence

This method contains a single function which takes an arraylist of coordinates as argument and calculates the overall length of the cities in a specific permutation. (The order of the coordinates in the array is the specific permutation.)

Main

This class is responsible for starting the program, it contains a basic UI which asks the user to choose from 3 options. First one is a specific test with 30 cities, second is a specific test with 20 cities and the third one is randomized and the user can input the number of cities and number of repetitions. The only real difference between the 1,2 and the third is that for option 1 and 2 I checked the actual best solution on through the use of an online TSP simulator.
(https://www.lancaster.ac.uk/fas/psych/software/TSP/TSP.html).
So for these test was easier to check the correctness of the solution.

Simulated Annealing implementation

When is comes to SA, the most important part of the algorithm is the temperature, cooling factor and the probability. First I tried to create my own probability formula, but that didn't really work out. The solutions were rarely statisfactory. So instead of my own formula I implemented the one that is used in a lot of SA algorithms, which is :

exp(- (e' – e) / T)

Where e' is the length of the currently found permutation, and e is the previous one found, T is the temperature.
The actions of the algorithm can be described as the following:
1.It takes the starting permutation of the cities and calculates the length of the path.
3.Then it goes through each city one by one, and switches them with its right neighbour.
If the new permutation is not better than the current best(starting) then it goes back to the original permutation and goes to the next city and then switches it with its right neighbour etc..Eg.:(abcd->bacd->acbd->abdc)
If the new permutation is better then the current best(starting), then it takes that combination of cities as the new permutation or starting point.
Meanwhile each time a new best solution is found, it is added to a variable that keeps track of the absolute best solution found so far.
Eg.:(abcd->bacd->bcad->badc).
4.Sometimes by chance even if the newly found permutation is worst then the current best solution it will take it and make it the current best to avoid being stuck in local minimum.

To make my algorithm work I've given specific value for each of my variables.
The best results I got when my cooling factor was 0,0001 and my temperature 100, and the way the cooling factor works, is by substracting its value from the temperature everytime the minDistance variable is changed. (Which is the variable that holds the current best solution).
Meanwhile to make sure my algorithm doesn't take too much time to run or to make sure it doesn't get stuck. I implemented the loopLength variable which looks like this:

```
double loopLength = (temperature / coolingFactor);
```

Here I take the temperature divide it by the cooling factor, and the result of this formula times the number of cities equals the amount of time the loop will run.
If we want to change the search time of the algorithm all we need to adjust is the cooling factor.

An additional feature I implemented for my SA algorithm is another way to come out of a local minimum. There is a slight chance that my algorithm finds the best solution at an early stage of the search, but changes it to a path that is worse, and then can't get back to the best solution again. So this avoid this problem I added an if statement which around 50%, 75% and 95% of the completion of the search it changes back the current solution to the one that was the shortest path throughout the runtime of the program.
This way we can fix the deteriorating of the algorithm to a worst solution, when he already found the shortest solution once, but just replaced it by chance.

Testing,results and evaluation

I did the following tests for my implementation:

1.Testing for 20 cities and x repetition. (Not randomized)

2.Testing for 30 cities and x repetition. (Not randomized)

3.Testing for y amount of cities, x repetition, randomized coordinates.(between x:0-200,y:0-200)

The test results explained:

First the programs writes out the starting permutation with coordinates.
Then the sum of the starting permutation.
Then it writes out the path with the cities that had absolute minimum length throughout the runtime of the program, below that is the length of that path.Then next comes the execution time of the search, and below that is the permutation and length of the solution.
Most of the time the absolute minimum and the solution is the same.
But there is a small possibility for them to differ, if in the last 5% of the runtime the best solution is changed to a worst one, and from that it gets stuck in a local minimum.
The „Did it help“ part just shows whether mine anti-stuck if statement didn't work.(If the number after the „Did it help“ is not the same as the solution then it didn't help, and the algorithm with a really low probability changed from a best solution to a worst one in that last 5% of runtime.
The last part of the output is the average of all the solutions throughout x repetitions, average of runtime and the minimal solution found in the x repetitions.

20 cities tests:

cooling factor 0.01

```
a[3,2],b[4,7],c[6,3],d[8,5],e[12,3],f[17,13],g[22,4],h[6,19],i[9,7],
Sum :355.8527 km
The permutation with the minimum length: nohiremqtgkdbacsfljp
Length: 266.1371 km
Solution:
Execution time: 0.124 seconds
Permutation: pjiertqmdcabhgksflon
Length: 266.1371 km
Did it help? 266.1371
The average length for 10 execution: 229.31238 km
Average time for 10 executions: 0.0805 seconds
Minimal solution found: 183.0208 km

Process finished with exit code 0
```

cooling factor 0.001

```
a[3,2],b[4,7],c[6,3],d[8,5],e[12,3],f[17,13],g[22,4],h[6,19],i
Sum :355.8527 km
The permutation with the minimum length: pnoqmhfrbidacegksljt
Length: 191.22749999999996 km
Solution:
Execution time: 0.084 seconds
Permutation: onptmfkridbhjsaecglq
Length: 245.4061 km
Did it help? 0.0
The average length for 10 execution: 187.09897999999998 km
Average time for 10 executions: 0.8834 seconds
Minimal solution found: 167.06319999999997 km
```

cooling factor 0.0001

```
a[3,2],b[4,7],c[6,3],d[8,5],e[12,3],f[17,13],g[22,4],h[6,19],i[9,7
Sum :355.8527 km
The permutation with the minimum length: jtpnoqmhbacdiersfkgl
Length: 172.90489999999997 km
Solution:
Execution time: 12.642 seconds
Permutation: jtpnoqmlfskgreidcabh
Length: 172.90489999999997 km
Did it help? 172.90489999999997
The average length for 10 execution: 169.56214 km
Average time for 10 executions: 11.520599999999998 seconds
Minimal solution found: 167.06319999999994 km

Process finished with exit code 0
```

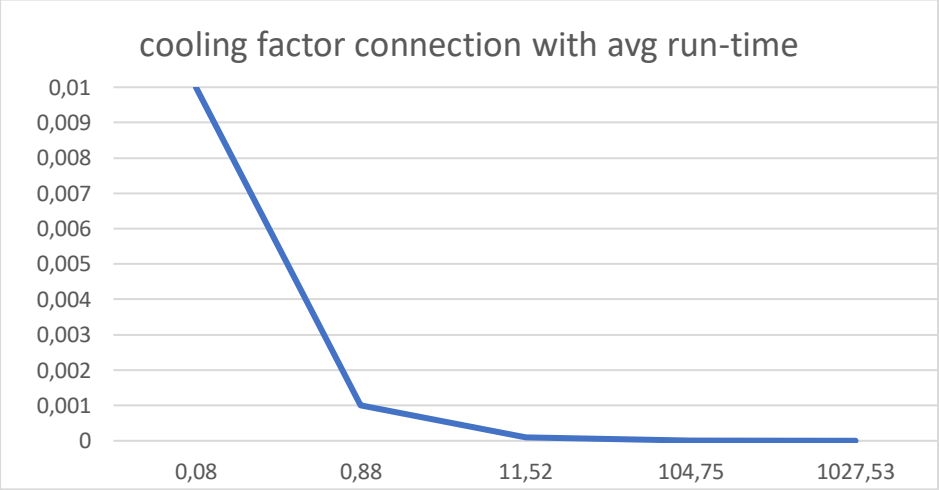
cooling factor 0.00001

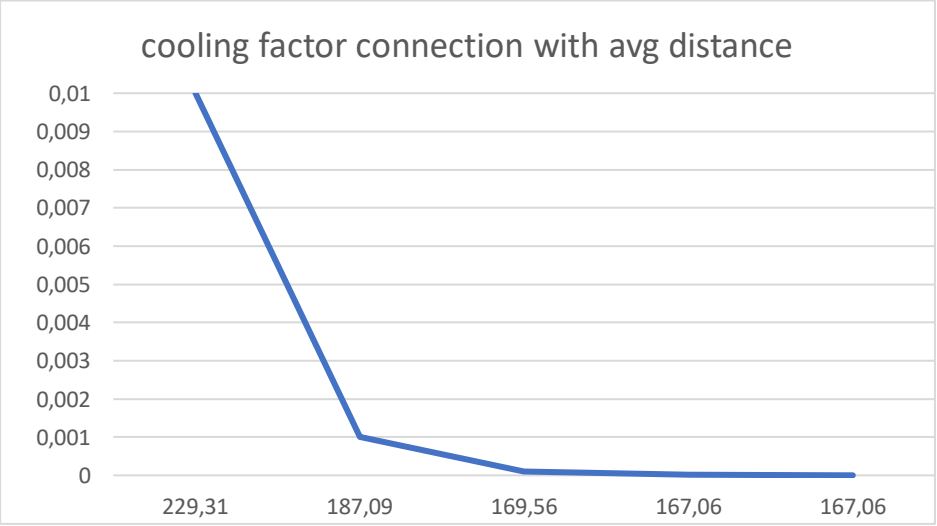
```
a[3,2],b[4,7],c[6,3],d[8,5],e[12,3],f[17,13],g[22,4],h[6,19],i[9,7]
Sum :355.8527 km
The permutation with the minimum length: pnoqmhbacdiergksfljt
Length: 167.06319999999997 km
Solution:
Execution time: 103.201 seconds
Permutation: pnoqmlksfhbacdiergjt
Length: 167.06319999999997 km
Did it help? 167.06319999999997
The average length for 10 execution: 167.0632 km
Average time for 10 executions: 104.74749999999999 seconds
Minimal solution found: 167.06319999999994 km

Process finished with exit code 0
```

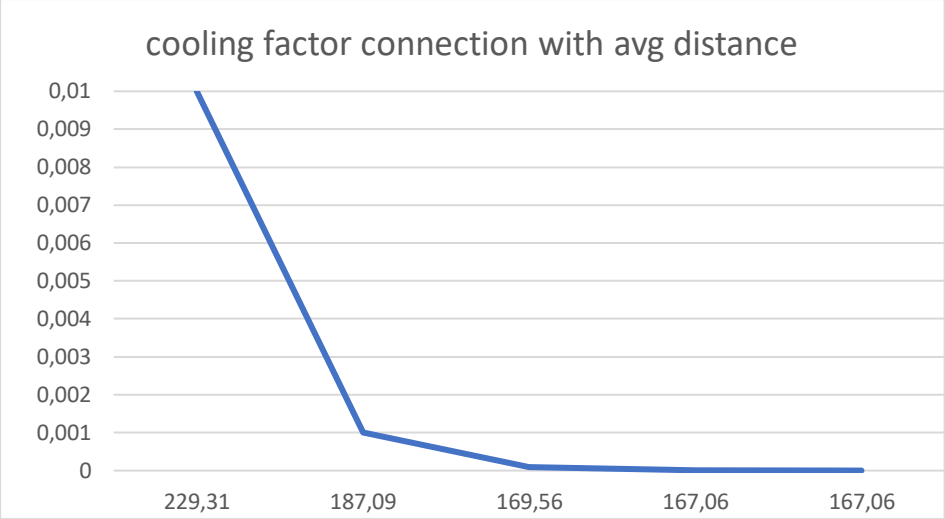
cooling factor 0.000001

```
a[3,2],b[4,7],c[6,3],d[8,5],e[12,3],f[17,13],g[22,4],h[6,19],i[9,7
Sum :355.8527 km
The permutation with the minimum length: ksfljtpnoqmhbacdierg
Length: 167.06319999999994 km
Solution:
Execution time: 1019.767 seconds
Permutation: cabhmqonptjlfskgreid
Length: 167.06319999999997 km
Did it help? 167.06319999999997
The average length for 10 execution: 167.0632 km
Average time for 10 executions: 1027.5369 seconds
Minimal solution found: 167.06319999999994 km

Process finished with exit code 0
```

Graphs:







From the above graphs we can see exponential decay in all three cases.
As we change the cooling factor to a smaller number the running time increases 10 fold. Meanwhile the average min distance solution is getting more and more to the actual value. The actual solution for these coordinates is 167.06. (Source: https://www.lancaster.ac.uk/fas/psych/software/TSP/TSP.html)
So from these test we can include if we want to test for 20 cities and we want to get an answer that is appoximately the shortest solution but in a short amount of time, then its enough run the algorithm with the 0.0001 cooling factor. However if we want to get the shortest path with a 100% certainty then it is enough to run it with 0.00001 cooling factor, which is still okay, since the execution time is only 100 seconds.
However it is not worth using the 0.000001 cooling factor, since it gives the same average as the one before but with 10 times increased execution time.

30 cities test:

cooling factor: 0.01

```
a[3,2],b[4,7],c[6,3],d[8,5],e[12,3],f[17,13],g[22,4],h[6,19],i[9,7],j[31,21]
Sum :625.4608000000002 km
The permutation with the minimum length: bxvqftápznoóúyuhwacdiesméjlkgr
Length: 391.39940000000007 km
Solution:
Execution time: 0.029 seconds
Permutation: bxqvftápznoóúyuhwieadcsméjlkgr
Length: 393.79210000000006 km
Did it help? 0.0
The average length for 10 execution: 423.8699500000001 km
Average time for 10 executions: 0.18370000000000003 seconds
Minimal solution found: 375.5872 km

Process finished with exit code 0
```

cooling factor: 0.001

```
a[3,2],b[4,7],c[6,3],d[8,5],e[12,3],f[17,13],g[22,4],h[6,19],i[9,7],j[31,21],
Sum :625.4608000000002 km
The permutation with the minimum length: évtmqúóonpzylsariwdcbhxfjágkeu
Length: 411.7637000000001 km
Solution:
Execution time: 1.94 seconds
Permutation: rklnoúózpátxsgihuqymdcabévjfwe
Length: 411.7637000000001 km
Did it help? 411.7637000000001
The average length for 10 execution: 379.59924 km
Average time for 10 executions: 1.9894000000000003 seconds
Minimal solution found: 332.31059999999997 km
```

cooling factor: 0,0001

```
a[3,2],b[4,7],c[6,3],d[8,5],e[12,3],f[17,13],g[22,4],h[6,19],i[9,7],j[31,21],k[19,9],
Sum :625.4608000000002 km
The permutation with the minimum length: noóúqubacwiderglksfxhmyévtjápz
Length: 294.44269999999995 km
Solution:
Execution time: 20.981 seconds
Permutation: noóúqubacwiderglksfxhmyévtjápz
Length: 294.44269999999995 km
Did it help? 294.44269999999995
The average length for 10 execution: 297.90733 km
Average time for 10 executions: 20.8374 seconds
Minimal solution found: 276.92060000000004 km

Process finished with exit code 0
```

cooling factor 0.00001

```
a[3,2],b[4,7],c[6,3],d[8,5],e[12,3],f[17,13],g[22,4],h[6,19],i[9,7],j[31,21],k[19,9]
Sum :625.4608000000002 km
The permutation with the minimum length: lskgiabwdrechuévyqúóonzpátjmxf
Length: 309.35810000000004 km
Solution:
Execution time: 214.894 seconds
Permutation: cabhqéjlfskgdiwumyvúóonzpátxre
Length: 309.35810000000004 km
Did it help? 309.35810000000004
The average length for 10 execution: 287.17948999999993 km
Average time for 10 executions: 217.05649999999997 seconds
Minimal solution found: 270.4162 km

Process finished with exit code 0
```

cooling factor 0.000001

```
a[3,2],b[4,7],c[6,3],d[8,5],e[12,3],f[17,13],g[22,4],h[6,19],i[9,7],j[31,
Sum :625.4608000000002 km
The permutation with the minimum length: ksxfljtázpnoóúqvéymuhwbacdierg
Length: 257.65409999999997 km
Solution:
Execution time: 2128.11 seconds
Permutation: bwhsljyétápznoóúvqumxfkgreidca
Length: 257.65409999999997 km
Did it help? 257.65409999999997
The average length for 10 execution: 263.0374699999999 km
Average time for 10 executions: 2136.549 seconds
Minimal solution found: 254.2633999999999 km

Process finished with exit code 0
```

Graphs:

Figure 2.1

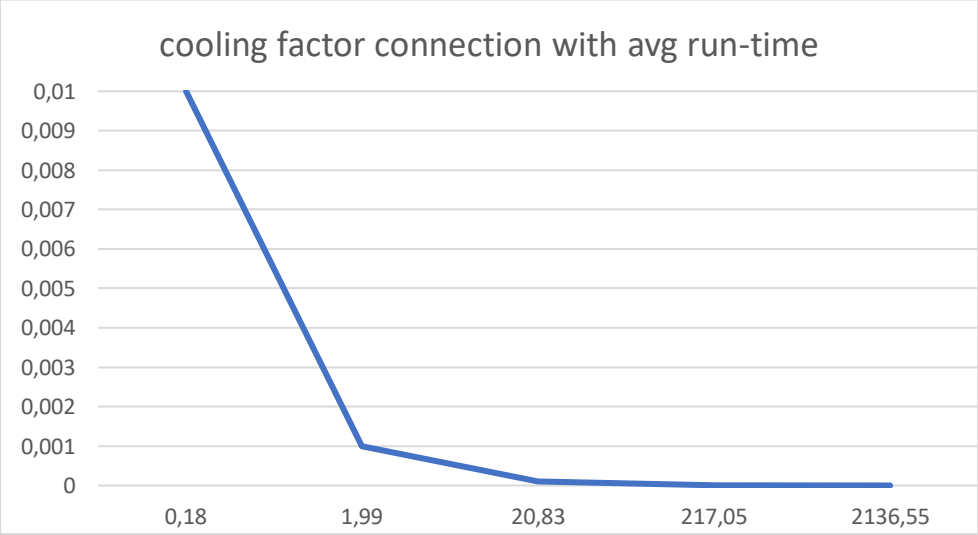


Figure 2.2

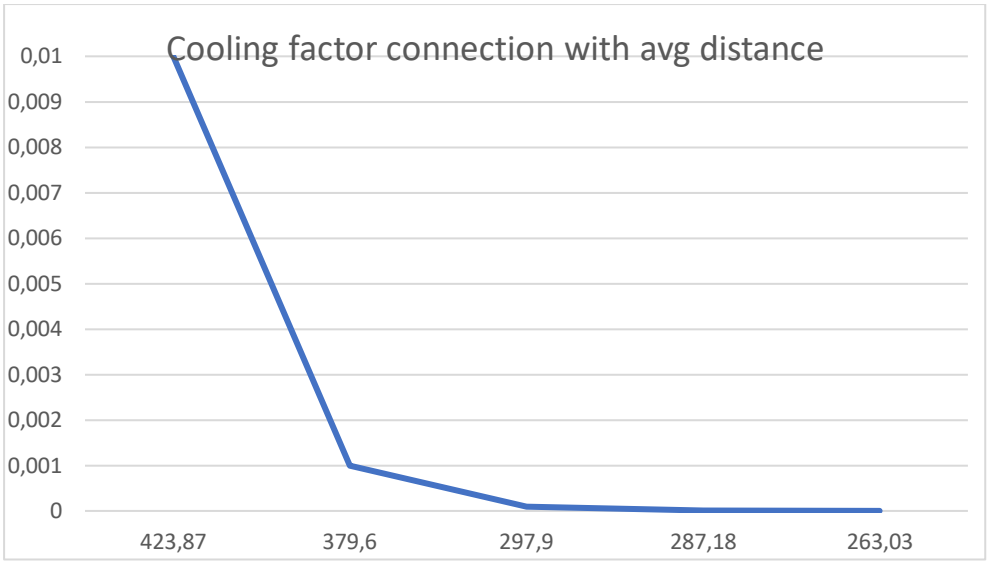
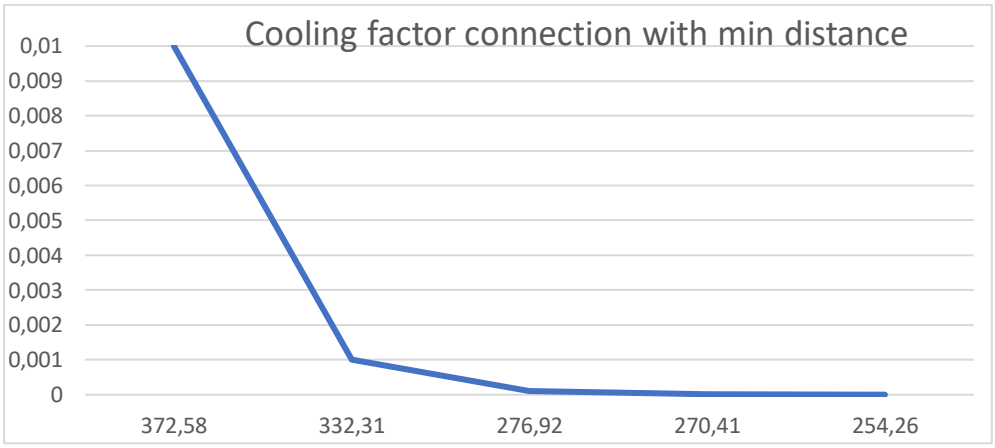


Figure 2.3



From figure 2.1 we can clearly see that for each time we divide the cooling factor with 10, the execution time is multiplied by 10.
The actual solution for these coordinates is 254.26. (Source: https://www.lancaster.ac.uk/fas/psych/software/TSP/TSP.html)
So we can conclude that for 30 cities my SA algorithm with 0.00001 cooling factor can get an acceptable solution, but not the best. However looking at the time difference between 0.00001 and 0.0001 cooling factor execution time and average length times, it is more worth to use the 0.0001 cooling factor. Since its 10 times faster, and the solution different is negligible. Meanwhile if we want to get absolute minimum path for the problem, we have to use the 0.000001 cooling factor. But it will take a lot of time. (Atleast on my laptop 10 repetition took around 6 hours to finish, however from that 10 repetitions 5 were actually the shortest solution)

Bonus test for 40 cities:

```
Sum :4336.863200000001 km
The permutation with the minimum length: 19-13-15-12-7-16-23-8-32-35-22-34-6-24-10-17-4-39-37-11-30-3-2-25-21-33-0-14-20-38-26-36-18-5-27-9-1-28-29-31-
Length: 1148.0189 km
Solution:
Execution time: 3394.915 seconds
Permutation: 28-15-12-7-8-34-6-24-22-35-10-17-30-3-2-25-11-39-4-16-23-32-37-14-0-21-33-20-27-5-18-36-38-26-9-1-13-19-31-29-
Length: 1148.0189 km
Did it help? 1148.0189
The average length for 1 execution: 1148.0189 km
Average time for 1 executions: 3394.915 seconds
Minimal solution found: 1148.0189 km

Process finished with exit code 0
```

Out of curiosity I also tried for 40 cities with the lowest cooling factor (0.000001), which takes the most amount of time to execute but gives the most accurate solution. As we can see it took me 3400 seconds just to execute 1 repetition.
The answer was not the shortest path, but it was really close.(1109 was the shortest for these cities.)

Conclusion

In conclusion, I think my algorithm performs statisfactory with enough time it can find solution for both 20 and 30 cities, if it has enough time. I also avoided the problem of being stuck on longer path when the algorithm has already found a shorter path but changed it thanks to the probability factor.

Some other way to make the SA algorithm would have been to instead of switching the places of neighbouring cities we used random sequences from the current combination of cities and replace som parts of it, or switch them up. This would give more variety to the permutations and it might even reduce search time.
Another thing that could have been done to reduce search time is when my execution reaches 50% and 75% of its search time, and puts the best solution that was found so far into the current best one. (if it had been changed to something worse). And if both at 50% and 75% the same value was put back into the current.
Then the chances of the algorithm finding a better solution that in the remaining 25% of the programs execution time is very slim. So this could reduce search time by 25%.