**Artificial Intelligence (COSC 6368) (Dr. Eick) – Fall 2017**

**Project 1: Solving Travelling Salesman Problems**

Prepared By

Egorov Maksim

**Introduction:**

In the current project I used few methods to solve Traveling Salesman Problem. First, I implemented simple Random Swapping strategy, Simulated Annealing, Depth First and Breadth First searches. I will explain each strategy in more details.

**Random Swapping:**

Random Swapping is the first search strategy implemented in this project to solve Traveling Salesman Problem. The main idea behind this strategy is creating random route first and then improve cost by swapping two random cities. Since algorithm contains random factor I run algorithm 3 times and produce 3 outputs. Below you can see pseudo code.

The Algorithm Pseudo Code:

1. Choose by shuffling list of cities initial state *S* with initial minimum cost *minCost*.
2. Create new state *S’* by randomly swapping two random cities in *S*.
3. Compute cost for new state *S’*

* If cost(*S’*)<*minCost* then *S=S’*
* Over wise we go back to step 2 not changing *S*

1. Repeat step 2 and 3 keeping track of the best solution until termination condition met.

Termination Condition:

First, as a stopping I chose condition then number of swaps reaches provided MEB. Although, running time of this algorithm not big, noticeable that after certain amount of swaps search getting stuck in local/global maximum and swaps not produces better results than previously discovered. I added another termination condition: I will also periodically check to ensure that the cost of the most optimal state found so far is changing. If it doesn’t change within a period of iterations, the search will be stopped to limit unproductive work. I chose 90 000 as a number of iterations I terminate search if during these iterations most optimal state didn’t change.

Results:

In the following tables we can see average results after 3 runs produced by algorithm for cost functions c1, c2, c3 and number of cities N=10, 30, 60, 120. I didn’t put actual path found by algorithm in this tables, because it will take too much space, but in output file I provide actual shortest path found with other parameters.

Table 1, cost function c1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **10** | **30** | **60** | **120** |
| **Initial cost** | 1091 | 1044 | 2151 | 5418 |
| **End cost** | 427 | 493 | 586 | 790 |
| **# of swaps** | 90071 | 90544 | 95431 | 126186 |
| **# of productive swaps** | 5 | 30 | 78 | 168 |
| **Running time (sec)** | 1.51 | 3.36 | 6.52 | 16.763 |

Table 2, cost function c2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **10** | **30** | **60** | **120** |
| **Initial cost** | 91 | 586 | 1681 | 5685 |
| **End cost** | 60 | 205 | 417 | 815 |
| **# of swaps** | 90067 | 91921 | 98328 | 132778 |
| **# of productive swaps** | 7 | 36 | 80 | 170 |
| **Running time (sec)** | 1.5 | 3.35 | 6.5 | 13.7 |

Table 3, cost function c3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **10** | **30** | **60** | **120** |
| **Initial cost** | 962 | 29224 | 241953 | 1979544 |
| **End cost** | 818 | 25459 | 209514 | 1702449 |
| **# of swaps** | 90167 | 94222 | 113018 | 200000 |
| **# of productive swaps** | 10 | 90 | 280 | 1300 |
| **Running time (sec)** | 1.7 | 3.7 | 8 | 30 |

In these tables, **# of productive swaps** represents number of swaps when algorithm finds better solution and changes current minimum cost. What we can see from these tables, Random Swapping algorithm provides significant performance in a short amount of time even for bigger number of cities.

**Simulated Annealing:**

After implementing first strategy I decided to proceed with some sophisticated strategy that will allow to avoid algorithm being stuck in local maximum. The main different Simulated Annealing (SA) algorithm from previous one, that SA in early stages of search allows to accept bad solutions with some acceptance probability. Same as Random search I run algorithm 3 times and output all results.

Parameter’s description:

* T0 initial temperature.
* Tend ending temperature
* Tk temperature in k algorithm iteration
* α – cooling factor, 0< α<1

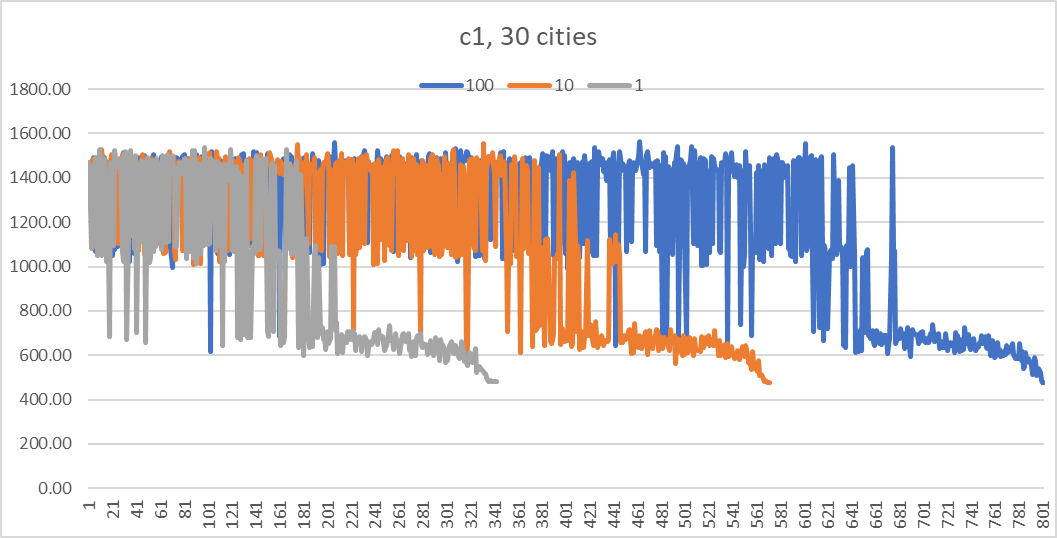
The Algorithm Pseudo Code:

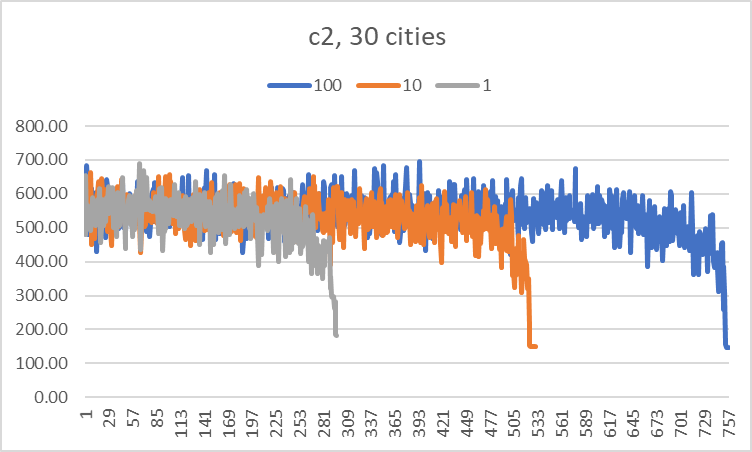
1. Choose by shuffling list of cities initial state *S* with initial minimum cost *minCost*, set T0, Tend, α
2. Create new state *S’* by randomly swapping two random cities in *S*.
3. Compute cost for new state *S’*  and delta = cost(*S’* )-cost(*S*)/cost(*S*)
   1. If delta<=0 , we accept this solution with p=1
   2. Over wise we accept this solution with probability exp(-delta/Tk)=> random(0,1)
   3. Tk+1=αTk
4. Repeat step 2 and 3 keeping track of the best solution until termination condition met.

Termination Condition:

For Simulated Annealing algorithm it is important to choose right termination condition, because if we will stop too soon, we won’t find optimal solution, over wise we will waste time on unnecessary calculations. We will terminate our search if we will reach MEB, reach end temperature or same as in previous strategy check to ensure that the cost of the most optimal state found so far is changing. I check this condition after 10 000 iterations. Less than for Random Swapping because we accept even bad decision with some probability.

The most difficult part was come up with start/end temperature and cooling factor. To do that I ran algorithm few times for couple different values and recorded algorithm performance and average values. First of all, I set end temperature to 0.0001 and cooling factor to 0.9999 so temperature won’t drop so fast. I ran algorithm for 30 cities, for three cost function for initial temperature 100,10,1. In the next charts we can see results:





How we can see from charts, almost for any initial temperature and cost function algorithm finds approximately same optimal solution. The difference is number of iterations needed before algorithm converges and stops. For initial temperature 1 algorithm runs less iterations than for initial temperature 100. But for cost function c2, for initial temperature 100 optimal solution has cost 148, for T=10 cost 151 and for T=1 cost 182. For other cost function difference not so big as for cost function c2. I decided to choose initial temperature equals 10.

To obtain more precisely results deeper research need to be done. At this moment I don’t have enough time to complete research.

Results:

Since I use randomized initial path I ran algorithm three times for each case. In this tables presented average results after running algorithm three times for the same number of cities.

Table 4, cost function c1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **10** | **30** | **60** | **120** |
| **Initial cost** | 961 | 1331 | 2254 | 5564 |
| **End cost** | 426 | 484 | 583 | 760 |
| **# of swaps** | 188253 | 188253 | 188253 | 188253 |
| **Running time (sec)** | 2.43 | 5.38 | 10 | 18 |

Table 5, cost function c2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **10** | **30** | **60** | **120** |
| **Initial cost** | 88 | 575 | 1700 | 5737 |
| **End cost** | 54 | 161 | 356 | 787 |
| **# of swaps** | 188253 | 188253 | 188253 | 188253 |
| **Running time (sec)** | 2.36 | 5 | 9.28 | 17.43 |

Table 6, cost function c3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **10** | **30** | **60** | **120** |
| **Initial cost** | 993 | 29988 | 246050 | 1990342 |
| **End cost** | 818 | 25284 | 209596 | 1712193 |
| **# of swaps** | 188253 | 188253 | 188253 | 188253 |
| **Running time (sec)** | 2.5 | 5.45 | 9.69 | 18.68 |

In the next Table 7,8 and 9 I compared minimal cost founded by Random Swapping and Simulated Annealing algorithms for three cost functions c1, c2 and c3.

Table 7, cost function c1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **10** | **30** | **60** | **120** |
| **Random Swapping** | 427 | 493 | 586 | 790 |
| **Simulated Annealing** | 426 | 484 | 583 | 760 |

Table 8, cost function c2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **10** | **30** | **60** | **120** |
| **Random Swapping** | 60 | 205 | 417 | 815 |
| **Simulated Annealing** | 54 | 161 | 356 | 787 |

Table 9, cost function c3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **10** | **30** | **60** | **120** |
| **Random Swapping** | 818 | 25459 | 209514 | 1702449 |
| **Simulated Annealing** | 818 | 25284 | 209500 | 1702193 |

How we can see, Simulate Annealing algorithm performs better with any cost functions than Random Swapping.

**Depth First Search:**

After, I decide to implement Depth First Search and see how DFS will perform solving traveling salesman problem. As a node in the tree I used a path of cities where left element root node and right element current node (city). This approach helps to store path and node information all together. For example, we have N=4 and we chose root node equals 0. In the picture is what the search tree will approximately look like:

The algorithm efficiency mostly depends on from what city we will start to search our tree. In the project description was specified that if we have random generator we need to run algorithm three times and every time with different root node.

Turns out DFS not efficient solving Traveling Salesman problem because we basically need to search all tree to find all possible paths and their cost. I advanced the strategy by pruning search path if the cost of current incomplete path more than min cost of the path we discovered earlier, because there is no need to continue expanding nodes if we already exceeded min cost. In Table 10 we can see comparison standard DFS and DFS with pruning for Number of cities equals 10.

Table 10

|  |  |  |
| --- | --- | --- |
|  | DFS standard | DFS with pruning |
| Number of nodes expanded | 200000 | 133473 |
| Frontier size | 23 | 0 |
| Min cost found | 426 | 426 |
| Running time (sec) | 8.24 | 5.017 |

Where frontier list contains nodes, we would explore if we continue search.

We can see that pruning decrease number of nodes being stored in memory and being expanded which leads to better performance. Advantages of this methods compare to Random Swapping and Simulated annealing that in the end we certainly will find optimal solution. Disadvantages: the biggest disadvantage is that algorithm performance depends on what start node we are going to choose, also DFS needs more resources and since we have MEB not efficient if the number of cities more than 10. In conclusion, DFS not efficient to solve traveling salesman problem for big number of cities.

Also, noticed that even for 11 cities algorithm runs few times slower than for 10 cities. Need to research deeper outside of this project why such less difference in number of cities so drastically increases execution time.

**Breadth First Search:**

After implementing DFS I also decided to apply Breadth first search for traveling salesmen problem because it is easy if DFS already developed. Unfortunately, MEB value 200 000 not enough to search all tree for number of cities 10 or higher.

**Conclusion:**

In conclusion, from all implemented strategies, the most efficient strategy to solve Traveling Salesman problem is Simulated Annealing, even though Depth First Search certainly provides optimal solution and pruning increases method performance, for bigger numbers of cities DFS still not efficient and can take long time and resources.

The advantage from Random Swapping, Simulated Annealing won’t be stuck in local minimum in early steps. But need to be careful by choosing temperature and cooling factor parameters.