Travelling Salesman Problem

Pankaj Kukreja
Siva Satvik

Introduction

Problem

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

We can see that the problem given falls under **NP complete** class. So, we tried approaching the solution with a few algorithms, and in them two ways(algorithms) seem to be quite fast and efficient(trade-off). They are:

- Genetic Algorithm.
- .....

We multi-threaded them and determined the heuristics for both of the algorithms with and without multi-threading and the results are as shown in the graphs after each section.

Genetic Algorithm

As the brute force algorithm for the problem will give us a complexity of $O(n!)$, this algorithm approaches the problem in an interesting way. This algorithm can be divided into following sections:

- Choosing few an aribitrary solution
- *"Breeding"* the chosen solutions

- Choosing the best out of the solutions in the above section.

This algorithm initially chooses random solutions for the given graph(Done by the function **initialPopulation()**). It most certainly won't be an optimal solution. The number of solutions it chooses can be given as an input from the user. This value when at a range of lesser values tends to give optimal solutions. After we get the solution set, we sort it according to their cost. And as we go ahead in the algorithm, this set essentially will contain the best solutions for the problem. And the top of this set will have the most optimal solutions in the set we have.
Now, after we get the set of solutions for the problems(not necessarily optimal ones), we randomly choose two of the solutions from the above set and apply what we call **crossOver()** on them. This is essentially breeding the two solutions chose. How this works is given as follows,

```
Example:
        parent1: 1 2 3 4 5
        parent2: 1 2 4 5 3

        substring in parent1: 2 3 4
        substring in parent2: 2 4 5

        substring inverted in parent1: 4 3 2
        substring inverted in parent2: 5 4 2

        child1: 1 5 4 2 5
        child2: 1 4 3 2 3
```

Here, let's introduce the concept of **genes**. These are essentially the vertices that are inherited by the children from the two parents we chose to breed.
We will have two random points one of them less than the other and in the range of number of vertices in the graph. Now, for each child, we innherit genes from their respective parents from start till first point and from second point till the end. For the middle part, as shown above, we inherit genes from the other parent in reverse order. This goes for both of the chidren inheriting. The children we get in this might or might not be the solutions for the problem, hence we check them before having them in the set of solutions we are forming. After we see that the children we got are indeed the solutions for the problem, we insert them in the set we created at the top using binary search. Hence maintaining the sorted order of the set. Everytime we insert into the set, we discard the bottom most solution for the problem since it clearly is not an optimal solution i.e. in a sorted set, the bottom most solution is the worst, hence we remove it.

We do the above mentioned ***breeding*** user defined amount of times(Taken care by the function **run()**). Here, we introduce the concept of ***generations***. This is essentially, how many times do you want to breed the solution set we got to get the optimal solution. This is analogous to the real life human genetic lineage. Hence the name **Genetic Algorithm.**

Multi-threading

Analysing the above approach, we reached to the conclusion that a lot of overhead we are getting is when the algorithm is running the breeding said number of times(on a scale of 1000) and attempted to parallelize that and sure enough, we saw good results from doing that. Obvioulsy, we couldn't just parallelize the loop for breeding as there are insertions and deletions and readings in the solution set by different threads which will lead to invalidation of the correctness of the algoritm. Hence, we approached the problem of parallelizing it in a different way. We tried the following approaches.

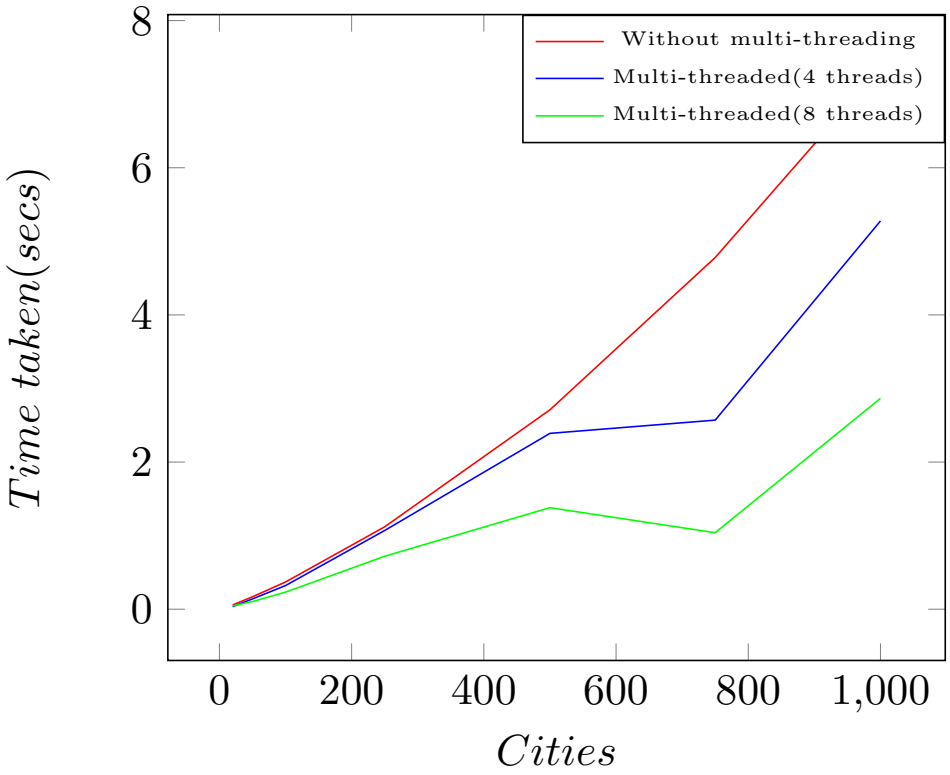
- Introducing mutex/shared locks
- Each thread computing on it's own to find the best result and among the results getting the best one.

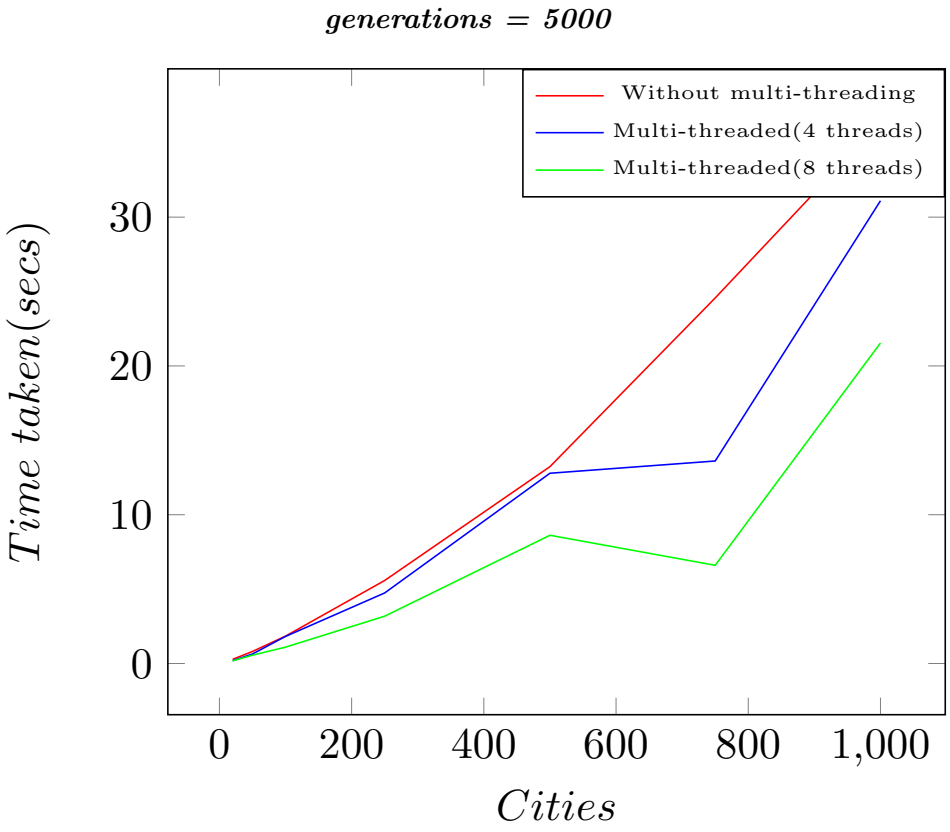
We tried various ways of introducing mutex locks and shared locks but none of them seemed to give good results as the overhead for achieving the locks is accountable as the solution set is being accessed a good amount of times. We couldn't get good results using this method, hence we implemented the below algorithm.

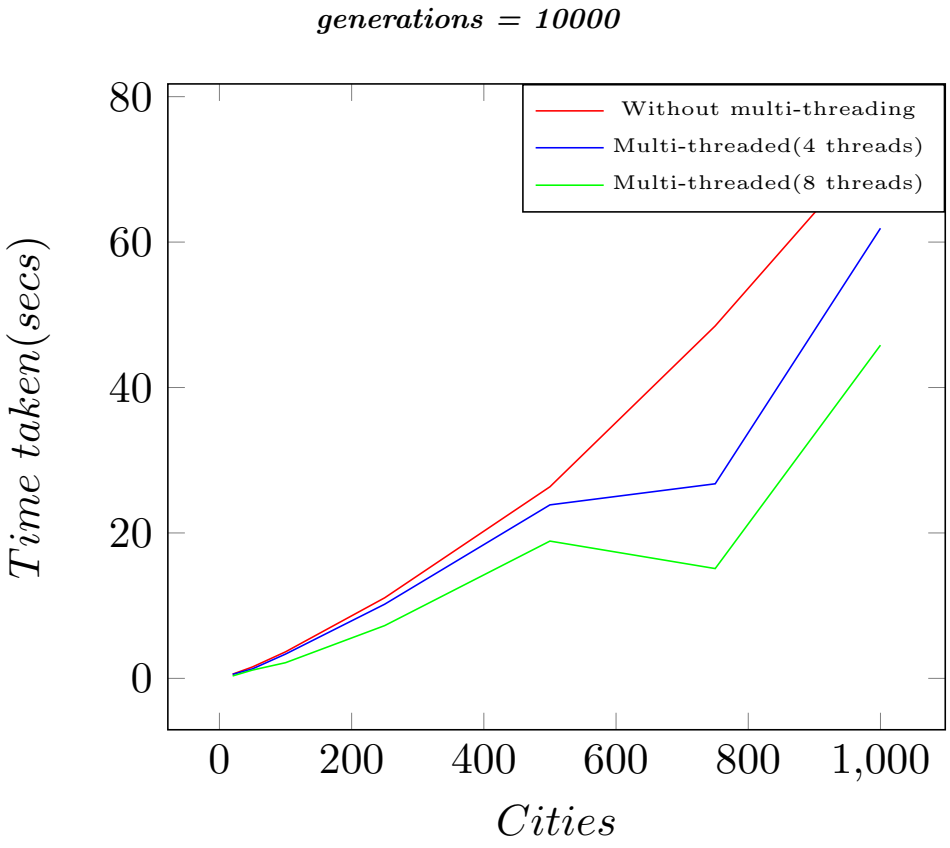
Coming the second method, since anyway we have a number of threads, we thought what if we make each thread compute it's own solution set and give us its best result and among the best results of all the threads, we get the very best one? And surely this approach fetched us good results. So in this implementation, each thread has it's own solution set and will work on that solution set to get optimal results. Essentially, we will be parallelizing the **run()** function.
The above mentioned algorithm is **Lock-free** and **Wait-free** implementation for the threads.
Taking into account the number of processors on the CPU, we gave 4 and 8 threads for testing of the algorithm.

Results

generations = 1000




generations = 5000
Without multi-threading
Multi-threaded(4 threads)
Multi-threaded(8 threads)
Time taken(secs)
30
20
10
0
0
200
400
600
800
1,000
Cities


generations = 10000
Without multi-threading
Multi-threaded(4 threads)
Multi-threaded(8 threads)
Time taken(secs)
80
60
40
20
0
0
200
400
600
800
1,000
Cities