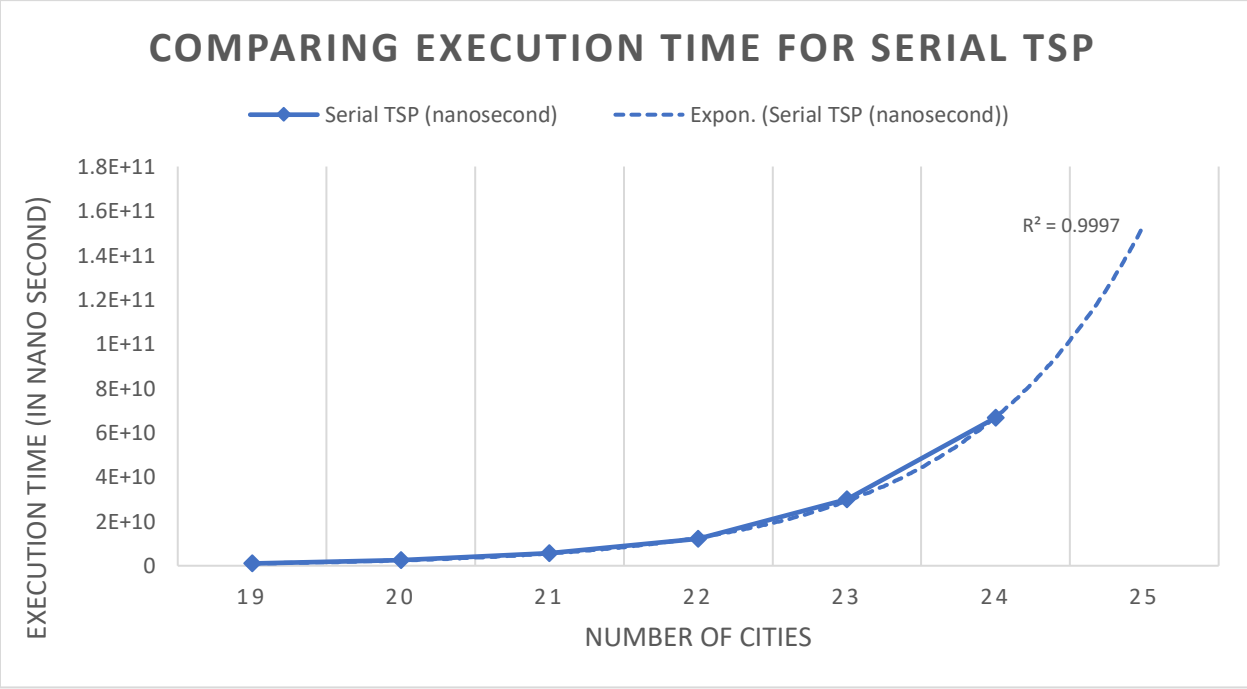
**Assignment 1 Report**

**Serial TSP**

SerialTsp.java asks for the number of cities from the user. For n number of cities, the SerialTsp.java create three random values (2 integers and 1 double) for the x coordinate, y coordinate, and infection rate. Next, we call the SerialTsp.java constructor an empty weighted adjacency matrix. Next, we created random coordinates with the x coordinate and y coordinate and calculate weight based on the formula. Once the weight is calculated for a coordinate, we calculate the weighted adjacency matrix. Then, we send the distance matrix to SerialTsp.getTour (). The getTour() function runs a while loop less than or equal to r and finds all possible combinations to get the minimum distance. Then it returns the Tour path by the nodes. The complexity of the algorithm is O(n!) to O(n^2 * 2^n). The algorithm is not efficient for a larger number of cities. Here is a sample chart of the number of cities and the amount of time it takes the algorithm to execute the program:

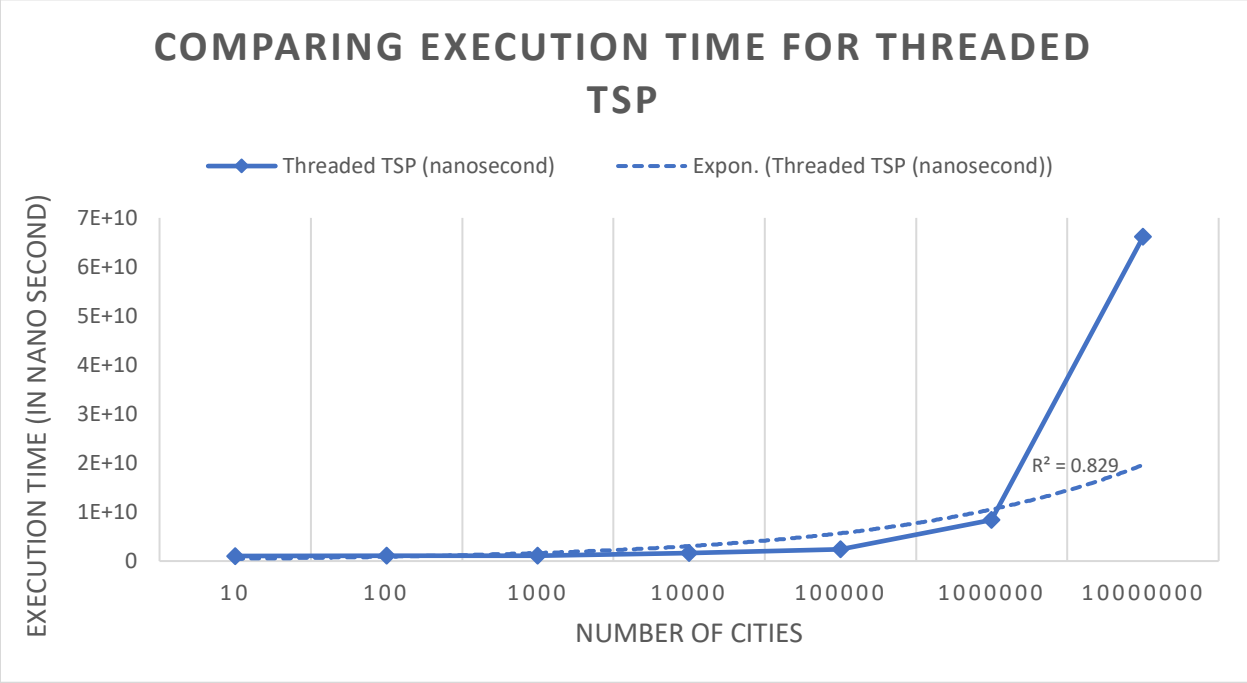
| n | Serial TSP (nanosecond) |
|---|---|
| 19 | 1026476200 |
| 20 | 2492251601 |
| 21 | 5642363900 |
| 22 | 12107982300 |
| 23 | 29969632601 |
| 24 | 66730497699 |
| 25 | Breaks! |



**Threaded TSP**

The ThreadedTsp.java like the SerialTsp.java implements the solve() function and it uses an iterative approach to find the TSP. Similar to the last step, we calculate the weighted adjacency matrix from three random values. Then we ask for arg1 thread and arg2 cityPerThread argument from the user. The algorithm then creates arg1 threads and each thread creates a weighted adjacency matrix with arg2 coordinates. Then the threads run in parallel. A global ArrayList adds the individual tsp from the threads once each thread is done. We used a very simple stitching algorithm where we just simply add the tsp of all threads in the ArrayList. Then we use the Inversion algorithm to detect inversion and handle it. First. We detect the crossing of the two edges by using the Line2D library. The library then uses the current and next three coordinates to see if there are any crossings on the two closest edges. For reducing time complexity, we only compared neighbor edges for inversion. Ideally, each edge needs to be compared with all the other edges to find all possible inversions. Once an inversion is found, we swap the second node (city) with the third node (city).

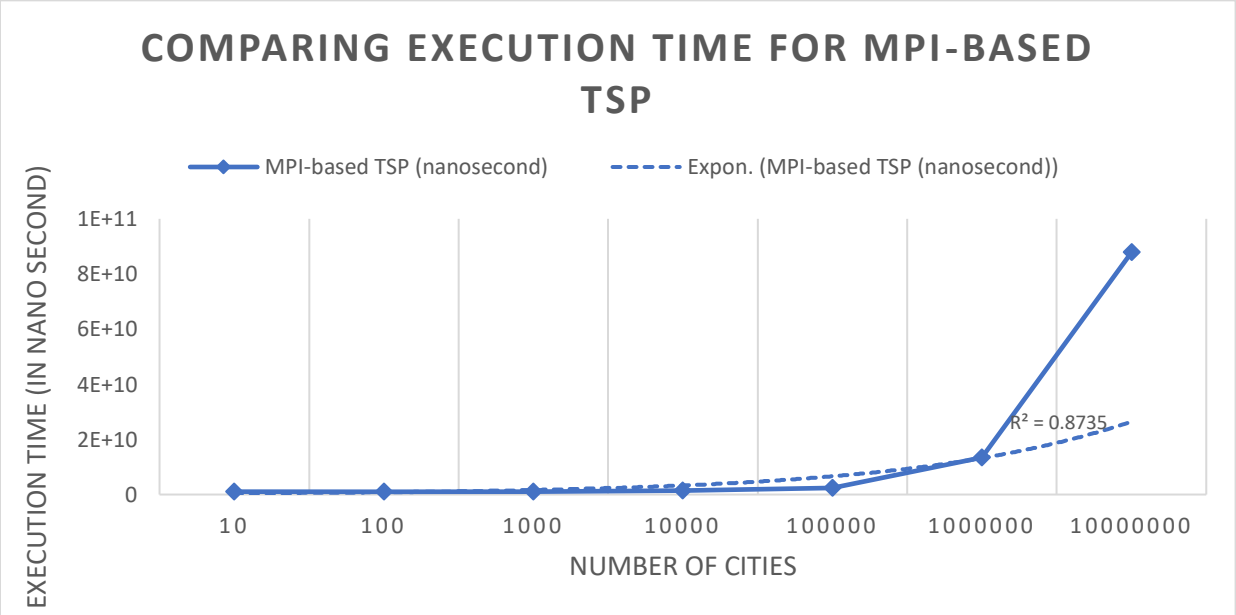
| N (Number of Block x Block Size) | Threaded TSP (nanosecond) |
|---|---|
| 1 X 10 | 1030834200 |
| 10 X 10 | 1080946800 |
| 100 X 10 | 1041580600 |
| 1000 X 10 | 1611682100 |
| 10000 X 10 | 2353034800 |
| 100000 X 10 | 8362454400 |
| 1000000 X 10 | 66091837600 |



**MPI-based TSP**

The MPITsp.java like the SerialTsp.java and ThreadedTsp.java implements the solve() function and it uses an iterative approach to find the TSP. Similar to the last step, we calculate the weighted adjacency matrix from three random values. Then we ask for the arg1 thread and arg2 cityPerThread argument from the user. The algorithm then creates arg1 threads and each thread creates a weighted adjacency matrix with arg2 coordinates. The master process is the main thread and the n-1 threads are the slave threads. The master processes and the n-1 process run in parallel. The master process has an ArrayList that collects the tsp from the slave process when the slave process is done. When the salve processes are done the thread is automatically destroyed. The master process then stitches the TSPs by just adding them arbitrarily (specifically whichever slave process finishes tsp first gets added first). Similar to ThreadedTsp, we use the Inversion algorithm to detect inversion and handle it. First. We detect the crossing of the two edges by using the Line2D library. The library then uses the current and next three coordinates to see if there are any crossings on the two closest edges. For reducing time complexity, we only compared neighbor edges for inversion. Ideally, each edge needs to be compared with all the other edges to find all possible inversions. Once an inversion is found, we swap the second node (city) with the third node (city).

| N (Number of Block x Block Size) | MPI-based TSP (nanosecond) |
|---|---|
| 1 X 10 | 1008484500 |
| 10 X 10 | 1005727800 |
| 100 X 10 | 1048154200 |
| 1000 X 10 | 1391129300 |
| 10000 X 10 | 2467518200 |
| 100000 X 10 | 13380199600 |
| 1000000 X 10 | 87808032500 |



**Observation:**

For the SerialTsp, the execution time is very higher but 100% accurate in finding tsp. However, over 24 cities the SerialTsp fails, which is an issue for finding tsp with a higher dataset. But the threaded algorithm is more time efficient in execution but less accurate in finding tsp. We can do an inversion to make it more accurate but it is still not 100% accurate in finding tsp. Since the blocks are running on threads, there are fewer combinations of paths (for small n value) which reduces time complexity. MPITsp is similar to ThreadedTsp regarding time efficient but more accurate than ThreadedTsp. Time efficiency depends on the number of cities in the block. More cities in the block lead to less time efficiency but more tsp accuracy. Since in MPITsp has a specific number of cities in every slave process, the accuracy is not as arbitrary as ThreadedTsp. The inversion also significantly improves the accuracy of the MPIthread.