UCCD3073 Parallel Programming

Assignment Part 1 (January 2022 Trimester)

Name	Student ID	Course	Practical Group
Tan Jing Jie	18ACB04560	CS	P2
Jacynth Tham Ming Quan	18ACB01600	CS	P2

Section A: Definitions and Abbreviations

TSP	Travelling Salesman Problem
n	Total number of points, including Parcels, Cities and Depot
INF	Holds the infinity value – numeric_limits<double>::infinity()
Parcel	Throughout the report, all "green points" will be referred to as Parcels
City	Throughout the report, all "red points" AKA "Point" in the question given, will be referred to as Cities
Depot	The starting/ending point of the modified TSP problem
Point	Represents all points, including Parcels, Cities and Depot

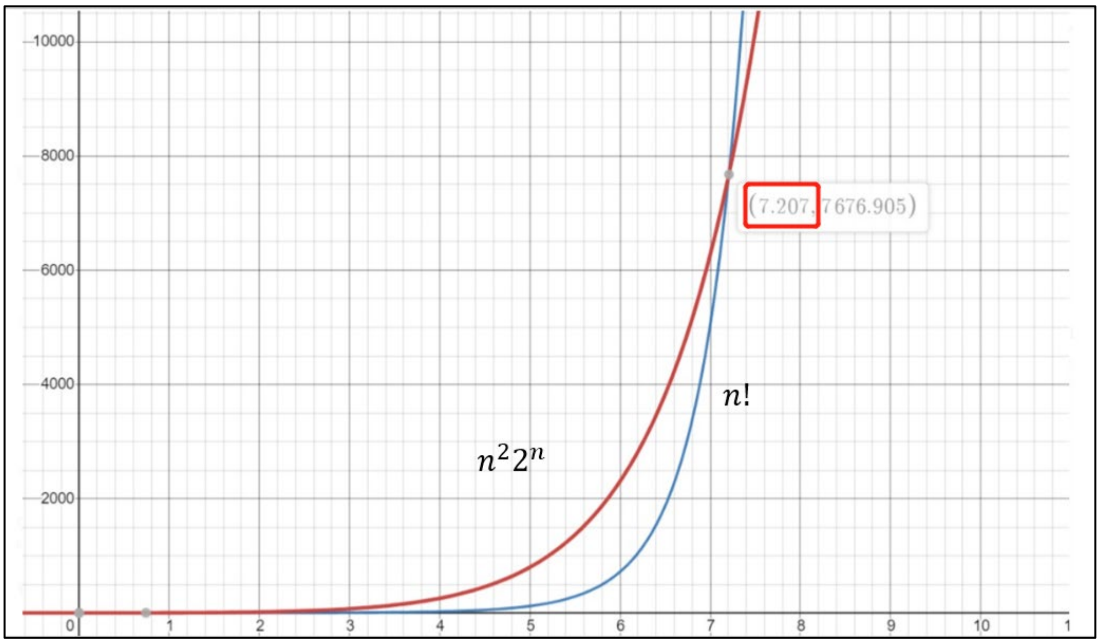
Section B: Introduction

Question*:* *"Given a set of locations, find the minimum distance of tour that visits each red point exactly once using brute-force method with the constraints that before travelling to each of the red point, a green point must be visited first. The total distance, including the green point must be the shortest. If there are more green points than the red points, ignore them."*

The problem given in the assignment question is a variation of the Travelling Salesman Problem (TSP). The Travelling Salesman Problem, on its own, is an algorithmic problem that challenges programmers to find the shortest route between a set of points in which all points must be visited once. TSP is categorized as an NP-hard optimization problem because there are no "quick" solutions to this problem. This is due to the increasing complexity of determining the optimal path as more and more points are added to the problem. In the given question, one additional criteria had been included – that is to traverse the red points and green points in alternate. In other words, the implemented solution must be able to find the minimum distance of a tour that visits every red point (Point) once with the exception that a green point (Parcel) must be visited before travelling to every red point.

Over the past decades, many exact and estimated algorithms have been proposed to solve the TSP problem, but to no avail. Some of these include Branch and Bound, Dynamic Programming, Brute Force and the algorithm that is explored deeper in this solution – Held-Karp. This solution program selects Held-Karp + Pure Brute-Force to solve the given problem described above because of the following reasons:

- The Held-Karp algorithm uses a bottom-up dynamic programming approach to reduce the runtime of the problem through enumerating through all sets of function calls (computes smaller tasks and use the results for larger tasks).
- The Held-Karp algorithm runs in exponential time instead of factorial time, which is theoretically faster.
- The Held-Karp algorithm is able to produce an exact solution (not estimate) to the given TSP problem.
- The Held-Karp algorithm has a time complexity of $O(n^2 2^n)$ whereby Brute-Force has a time complexity of $O(n!)$. However, theoretically, when n (the total number of points) is 7 or less, Brute-Force produces results much faster than the Held-Karp algorithm. Held-Karp is only able to beat Brute-Force when there are 8 points and above. The graph below illustrates this theory:



Section C: Brain-Storming

The aim of the brain storming session was to find the most efficient algorithm to solve the modified TSP problem given. In order to simplify the process, the team have decided to solve smaller problems first and ultimately use the solutions to the problems solved earlier to solve upcoming problems. Therefore, the overall problem to be solved in the question had been split into a series of smaller problems, as listed below:

1. First, a working solution for the original TSP problem (assume that all points need to be visited, regardless of color) is needed.
2. Secondly, the solution has to be able to traverse Parcels and Cities alternatively, and that each City can only be traversed after a Parcel has been visited.
3. Finally, the solution has to be able to handle scenarios in which there are more Parcels than Cities.
4. At this point, performance of the program can be considered and enhanced.

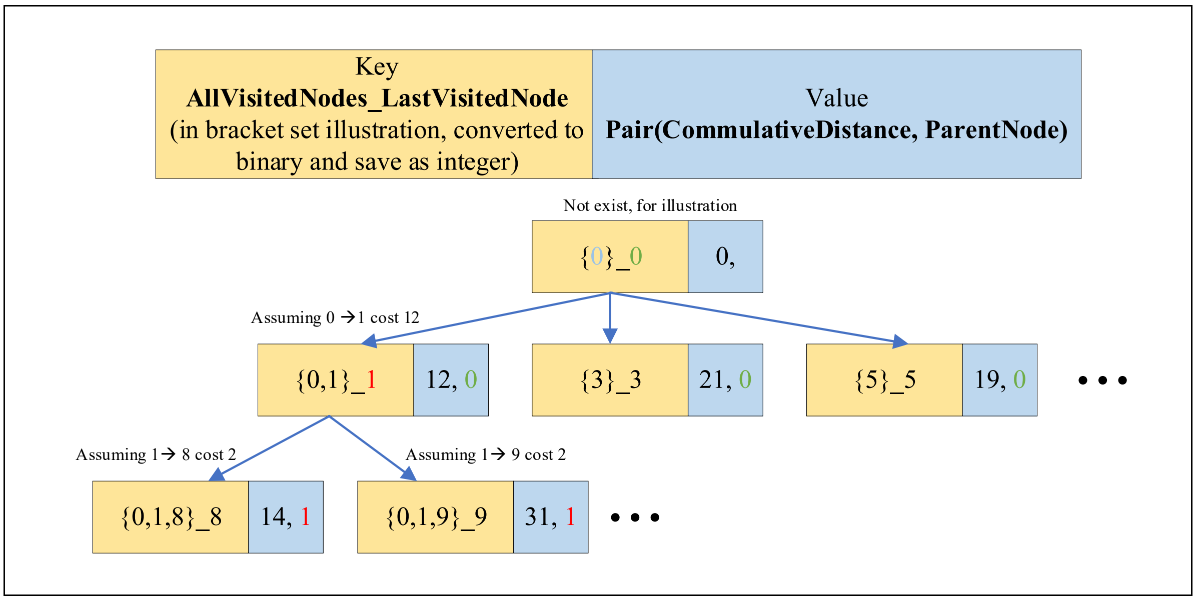
Having determined the small tasks that needed to be solved, the team had to think of the main algorithm to be used. During the brain storming session between the team members, the first solution implemented was by using Brute Force. However, with the Brute Force method, one major disadvantage was that this solution could only handle a maximum of 7 red points and 7 green points. Therefore, other solutions were sought out. Out of all the possible exact solutions that were explored, the Held-Karp seemed promising because it utilizes the same logic as the brainstorming session – splitting a larger task into multiple smaller tasks, with the idea that to solve problem n, the solution to problem $n - 1$ can be used.

Having looked up the time complexities of both Brute-Force and Held-Karp, which are O(n!) and O($n^2 2^n$) respectively [1], it was discovered that for 7 points and less, Brute-Force actually produces results much faster (theory obtained through calculation). Therefore, the implemented solution will take the best of both worlds, which will result in a combination of Held-Karp + Brute-Force algorithm.

Once the algorithms to-be-used have been determined, the implementation of the solution algorithm can begin. The following section explains the implementation of the solution and the results obtained.

Section D: Modified Held-Karp Concept

Held-Karp is a dynamic programming algorithm that is able to utilize space to reduce time. In the Held-Karp used in the implemented solution, all possible path combinations are first generated. Then, all the possible distances between points are generated. While computing the latter, Held-Karp also makes an effort to store the calculated distance as well as the cumulative distance of the parent node. This way, the child nodes can utilize the distance from the parent node when attempting to calculate the minimum distance. By using unordered map, this can reduce the finding complexity from $O(n)$ to $O(1)$. The diagram below illustrates this theory using an unordered map:


Key
AllVisitedNodes_LastVisitedNode
(in bracket set illustration, converted to binary and save as integer)
Value
Pair(CommulativeDistance, ParentNode)
Not exist, for illustration
{0}_0
0,
Assuming 0 → 1 cost 12
{0,1}_1
12, 0
{3}_3
21, 0
{5}_5
19, 0
Assuming 1→ 8 cost 2
Assuming 1→ 9 cost 2
{0,1,8}_8
14, 1
{0,1,9}_9
31, 1

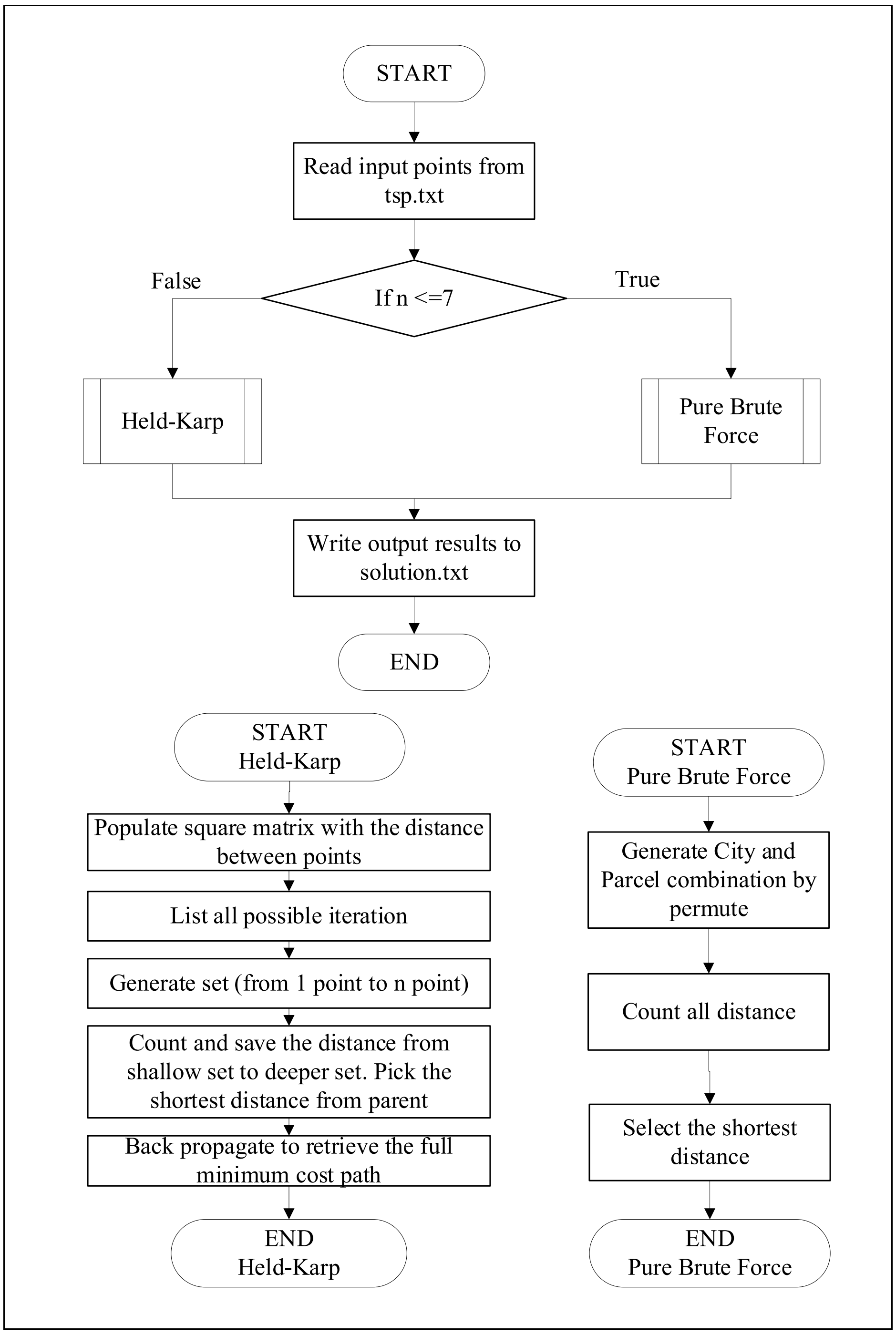
The diagram above shows how Held-Karp is able to build different path combinations along with its corresponding cumulative distance efficiently. The topmost node does not exist but is present in the drawing to show that all nodes eventually span from a common parent, or Depot, in this case. In the unordered map above, the needed information are stored as key-value pairs. The keys hold the string representation of all the visited nodes (represented in binary and converted to an integer) and the last visited node. Taking the leftmost node as an example, {0,1,8} are all the nodes that the algorithm had passed through. If the full set of nodes in this example is {0,1,2,3,4,5,6,7,8}, then {0,1,8} will yield 10000011 in binary, which is 385 when converted to integer. The last visited node in {0,1,8} is 8, therefore, the second part of the key will hold the value 8. The final key will be 385_8, in this example.

Furthermore, the value of each corresponding key stores a pair data type which consists of two values – the cumulative distance from all the parent nodes up to the current node, as well as the index of the parent node itself. This makes it easier to retrieve the parent of a given node, which is especially useful when trying to use back propagation to retrieve the full minimum cost path later on. Using the previous example of the leftmost node, it is assumed that the distance from 1 → 8 is 2. Therefore, the cumulative distance will be the previous cumulative distance (12) plus 2, which will yield 14.

The next section describes the full Held-Karp + Pure Brute Force algorithm used in the implemented solution.

Section E: Held-Karp + Pure Brute-Force Implementation

The diagram below shows a summarized flow of the solution's implementation:


START
Read input points from tsp.txt
If n <=7
False
True
Held-Karp
Pure Brute Force
Write output results to solution.txt
END
START Held-Karp
Populate square matrix with the distance between points
List all possible iteration
Generate set (from 1 point to n point)
Count and save the distance from shallow set to deeper set. Pick the shortest distance from parent
Back propagate to retrieve the full minimum cost path
END Held-Karp
START Pure Brute Force
Generate City and Parcel combination by permute
Count all distance
Select the shortest distance
END Pure Brute Force

Step 1: Read input points from tsp.txt

In this step, the input points are obtained from tsp.txt using a while loop and categorized into Parcel, City and Depot. At this point, the *parcel_num* array and *city_num* array are also populated with the index of the input points. After that, a separated variable, *all*, is also populated to store the Depot, all Cities and all Parcels in that order. Next, if the total number of points is 7 or less, the implemented solution uses Brute-Force(permutation) to find the shortest path. On the other hand, if the total number of points is 8 or more, then the implemented solution uses Held-Karp to find the optimal path.

Step 2ai: Populate distance matrix (Start of Held-Karp)

Next, the distance (adjacency) matrix, roadmap, is populated with the distance between each point (Depot, Parcel and City). In order to reduce the number of path combinations later on, the "impossible paths" are marked with INF, which holds the maximum value that a double data type can hold. The conditions to determine "impossible paths" are listed below:

a. The program cannot travel from Depot to Point.
b. The program cannot travel from Parcel to Depot.
c. The program cannot travel from itself to itself.

If all these conditions are fulfilled, then the program calculates the distance between the two given points. The table below illustrates the final outlook of roadmap:

	Depot	**City1**	**City2**	**Parcel1**	**Parcel2**	**Parcel3**
Depot	INF	INF	INF	distance	distance	distance
City1	distance	INF	INF	distance	distance	distance
City2	distance	INF	INF	distance	distance	distance
Parcel1	INF	distance	distance	INF	INF	INF
Parcel2	INF	distance	distance	INF	INF	INF
Parcel3	INF	distance	distance	INF	INF	INF

Step 2aii: Held-Karp Implementation

The steps to the Held-Karp Implementation are listed below:

1. First, an unordered map is initialized to store key value pairs, such what each key represents allVisitedNodes_lastVisitedNode and each corresponding value is a pair of two numbers – cumulative distance between all previously visited nodes and the parent node. The example below shows how one entry in the unordered map looks like:

key: 6_3	→ 6 in binary is 111, which means that three points have been visited.
value: pair(80, 3)	→ 80 is the cumulative distance and 3 is the parent node.

2. Next, pairset lists of parcel-to-city sets are generated and create a fundamental entry of start from parcel in a for loop.
   - Pairset is defined as 1 set parcel + 1 set city. For example:
     Input: City {1,2,3} and Parcel {4,5,6}
     Output: (Umap): [1_0, Pair(distance, 1), 3_0, Pair(distance, 1),...]]
     (Pairset): ([{1},{4}], [{1},{5}], [{1},{6}],[{2},{4}],...,[{3},{6}])

   - This pairset list is needed for the later steps, where the cumulative distance needs to be calculated. As of this point, the pairset list only houses the city-parcel combinations.
   - In this step, a key will be created with the format: allVisitedNodes_parentNode. For example, if {4,1,5} have been visited, the last visited node is 4, then the key will be 25_4 (because {4,1,5} will yield 11001 in binary, which equals 25).
   - Next, pairs of (distance of Depot to Parcel, 0) are stored into an unordered map (umap) with the key created above. 0 here represents Depot, which means that this is the first node and that there are no nodes before this.

3. The next step is to generate all possible routes from a single point or from multiple points
    - The main purpose of this step is to generate pairsets and store in pairset list.

      Input: City set and Parcel set

      Output: (routes): All possible routes from start parcel to end city (without Depot) will be stored in *routes* **(complete** routes only).

      (pairset): Generate possible combination (multiple Cities and Parcels)
    - In this step, a function called generate_combinations is used to generate the combination list. An example is provided below:

      Input of generate_combinations: {1,2,3,4} and size = 4

      Output of generate_combinations: ([{1},{2},{3},{4}]

      [{1,2},{1,3},{1,4},{2,3},{2,4},{2,5}...{3,4}]

      [{1,2,3},{1,2,4},…,{2,3,4}]

      [{1,2,3,4}])
    - Pairset can be generated by iterating the input Parcel set and City set in the pairset list and performing the following:
        1. First, visited parcel and city combinations are obtained using generate_combinations function.
        2. Then, with the integer obtained, the parcel sets and city sets will be combined so that in the output set (combset), both Cities and Parcels are included.
        3. After combset is obtained, the sizes of Parcel set and City set are compared. If both the sizes are equal, means the next node to visit must be a parcel, then set nextSet as Parcel set. On the other hand, if the two sizes are different, means the next node to visit is a City, then set nextSet as City set.
    - Meanwhile, when the pairset matches a valid route, it also saved into a route (number of Cities == number of Parcels), it also saved into a route array. To determine whether a route is complete, the number of visited nodes is checked against 2* number of Cities. If the two numbers are equal, means the route is valid. For example, if there are {1,2,3,4,5,6} points and {1,2,3,6} are visited, then the corresponding binary string will be 111001, which converts to 57 as an

integer (same as in generating first part of umap key). Each of these integers is then stored in a vector<int> named routes.

4. Next, the distances for all complete routes are computed.
    - Input: Pairset
    - Output: Calculate the minimum cumulative distance and populate umap
    - In this step, combset is looped and one point will be discarded in order to find the minimum parent path from umap. For example, for one turn of the for loop, if combset holds 110011 (exp value in binary), then for this round of looping, the implemented solution will look for 10011 from the umap. If this key exists, then combset will be created under this new parent (10011) in the umap. The new key will be {11011}_4 if the optimum point in nextset is 4.
    - At this point, the minimum cost path is already known by the program. This is saved under the key with the parent node (last node of previous node).
    - Hence, the next step is to find a way back to the Depot.

5. Moving on, all the possible paths from City to Depot are looped to cumulate the distance to return (no need to save into umap, since can directly know the best route and start backward propagation in next step).
    - This is done by calculating the temporary_min value by using the existing cumulative distance in the umap and adding the new distance (City → Depot) to see which return path is the minimum.
    - If the temporary_min is more than the min, then the min will be updated to temporary_min.
    - When this happens, the parent (which is the last city point) of the minimum cost return path is noted down too, to be able to retrieve the full path starting from this parent in the final step.

6. Once the full path is obtained, the implemented solution uses backwards propagation to find the full minimum cost path.

- If the umap figure was used to illustrate this point, the current position of the program now is at the bottom of the “tree”, and the goal is to back propagation to the root of the “tree” (First parcel) to get the full path.
- This is done by using the last obtained parent from Step 5 and using it to find the previous node.
- Every round, the parent will be updated with the parentNode of the current node (by removing the leftmost bit). This way, by repeatedly getting the parent node of the current node, eventually, the full path is obtained.

Step 2b: Pure Brute Force

The Pure Brute Force algorithm is used sparsely – only when there are up to 3 Cities and 3 Parcels, 1 City and 5 Parcels, or 2 Cities and 4 Parcels. All the combinations mentioned earlier will lead to a total of 7 points (including Depot). For total number of points beyond this, the Held-Karp algorithm earlier will be called. The implementation of the Pure Brute Force algorithm is not so significant, but yields faster results for 7 points and below only, when compared to the Held-Karp algorithm.

The steps of the Pure Brute Force process are listing below, assuming that the point input step had already been completed:

1. The distance from the depot to each possible parcel point is calculated (Tabulated in 1-D vector array).
2. The distance from each possible city point to depot is calculated (Tabulated in 1-D vector array).
3. The distance between each parcel to each possible city is calculated (Tabulated in 2-D vector array).
4. Do permute for all possible city and parcel into a list. For city, count of result is formulate by ${}^{CITY\_SIZE}P_{CITY\_SIZE}$ and for parcel is ${}^{PARCEL\_SIZE}P_{CITY\_SIZE}$. By using the set, City{1,2} and Parcel {1,2,3} the list is:
    - City = [{1,2},{2,1}]
    - Parcel = [{1,2},{2,1},{1,3},{3,1},{2,3},{3,2}]

5. After that, the summation of all possible paths are calculated using 3-nested for loops. The first 2 for loop is running for iterate the city and parcel combination. The summation can be done by adding an inner for loop. This can be visualized as seen in the table below (only yellow have to be calculated).

	City1	City2	City3	City4
Parcel1	Depot to Parcel1 Parcel1 to City1			
Parcel2	City1 to Parcel2	Parcel2 to City2		
Parcel3		City2 to Parcel 3	Parcel 3 to City3	
Parcel4			City3 to Parcel 4	Parcel 4 to City4 City4 to Depot

- In the table above, an example is given with 4 cities and 4 parcels. Since it is known that a Parcel has to be visited before each City, the minimum cost path should look like the path illustrated in the table above.
- Therefore, in the submation process, the precalculated distances of the yellow-highlighted path (obtained from earlier steps) will be totaled up to get the total distance of the minimum cost path. The purpose of this is to reduce repetition of redundant calculations.

6. Meanwhile, the path with the shortest path is always record and the order of points are written to the output file at the end.

Step 3: Write output results to solution.txt

Once the minimum cost path has been found from either the Held-Karp algorithm or the Pure Brute Force algorithm, the labels (Parcel1, Depot, Point2) will be written into the output file, solution.txt.

Section F: Testing

During the testing phase, the implemented solution was able to find the minimum cost paths of a maximum of 9 Cities and 16 Parcels. Several other extreme test cases were thrown to the implemented solution to see if the results were desirable. In the end, the implemented solution was able to deal with scenarios where:

- Normal cases
- There are more Parcels than Cities
- The Parcel and Cities are towards the edge of the screen (larger distance, but solution did not overflow)
- There is only one City (Regardless of number of Parcels)

The specification of testing computer is:

- Processor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1800 Mhz
- RAM: 11GB available memory

The following table shows the maximum results obtained by benchmarking using the TSP program (time limit = 3 minutes):

Number of Cities	**Number of Parcels**	**Total Points (Including Depot)**	**Time Taken to Complete Execution**
12	12	25	120.62 seconds
11	13	25	119.88 seconds
10	14	25	97.96 seconds
9	16	26	115.63 seconds
8	16	26	34.38 seconds
8	19	28	136.24 seconds
7	14	22	6.96 seconds
7	23	31	139.72 seconds
6	12	19	0.53 seconds

Refer to Appendix for screenshot of benchmarking.

When city point less than or equal 8, the TSP program able to resolve the problem (with maximum doubled parcel) within the time (3 minutes). Besides, when city point less than 6, the TSP program can resolve the problem like instantly (<1 seconds).

Worth to mentioned that if the number of city is 9 and parcel is 16. The total execution time exceed 3 minutes, but it cannot complete and had been automatically terminated.

Section G: Final Wordings and Future Work

The implemented Held-Karp + Pure Brute Force algorithm has been proven to be effective when used to solve the modified Travelling Salesman Problem given in the assignment question. However, Held-Karp itself is sufficient to solve the TSP problem when there are 7 or less points. If there are size concerns with the implemented solution, the Pure Brute Force part can be omitted in the solution.

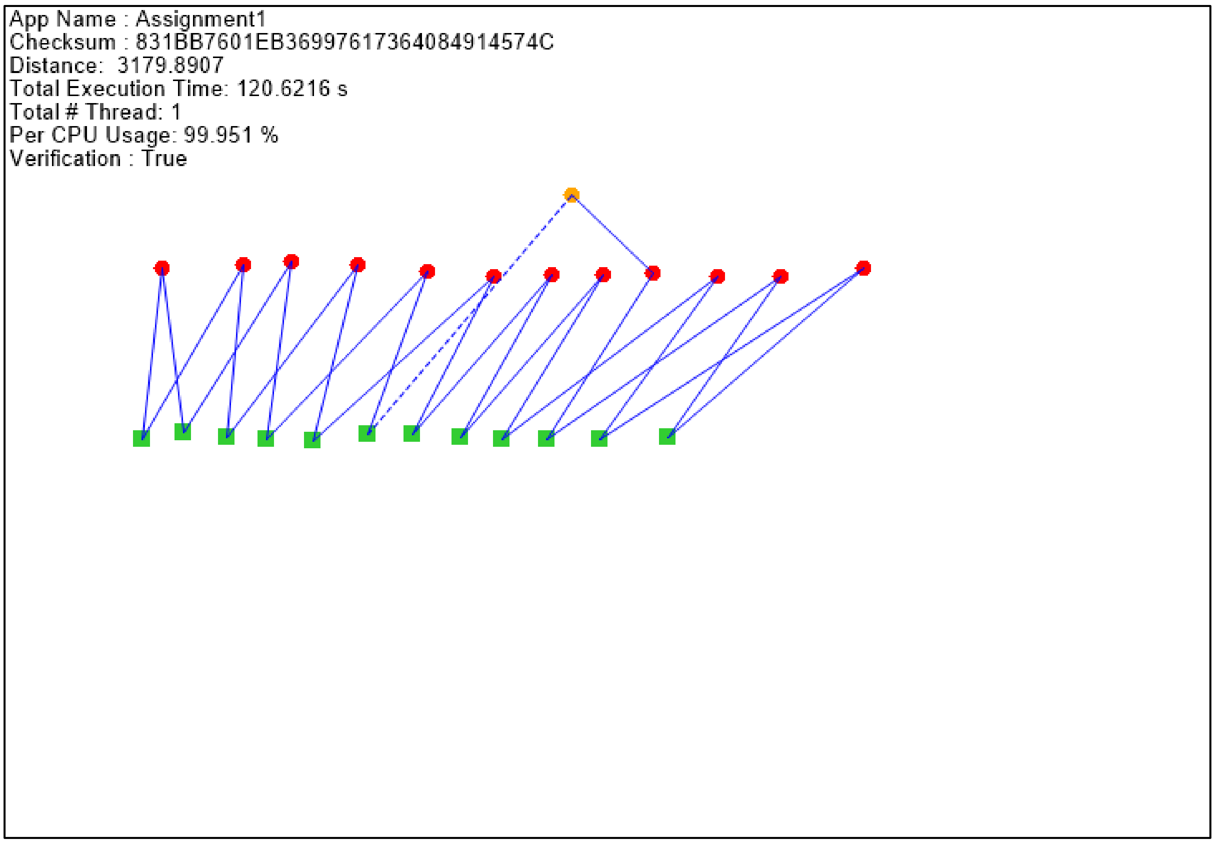
Having fully tested the implemented solution thoroughly, it is proven that the combination of Held-Karp and Pure Brute Force is viable. In the upcoming assignment, the ultimate goal is to parallelize the implemented solution in this assignment. By doing this, the work can be split across threads to yield faster results.

Section H: References

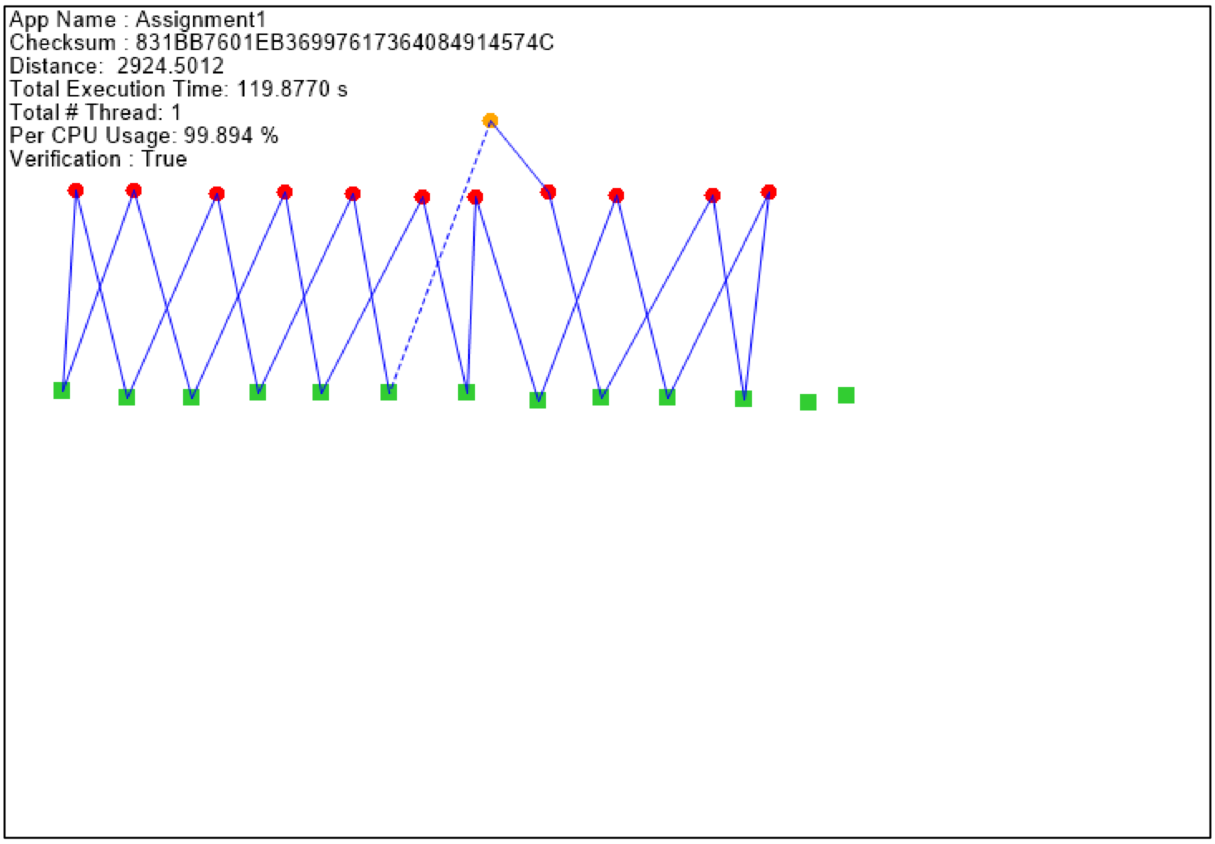
[1] N. Quang, "Travelling Salesman Problem and Bellman-Held-Karp Algorithm", Math.nagoya-u.ac.jp, 2020. [Online]. Available: http://www.math.nagoya-u.ac.jp/~richard/teaching/s2020/Quang1.pdf. [Accessed: 10- Mar- 2022]

[2] "HeldKarp/heldkarptsp.cpp at master · dcruzf/HeldKarp", GitHub, 2022. [Online]. Available: https://github.com/dcruzf/HeldKarp/blob/master/code/heldkarptsp.cpp. [Accessed: 12- Mar- 2022].

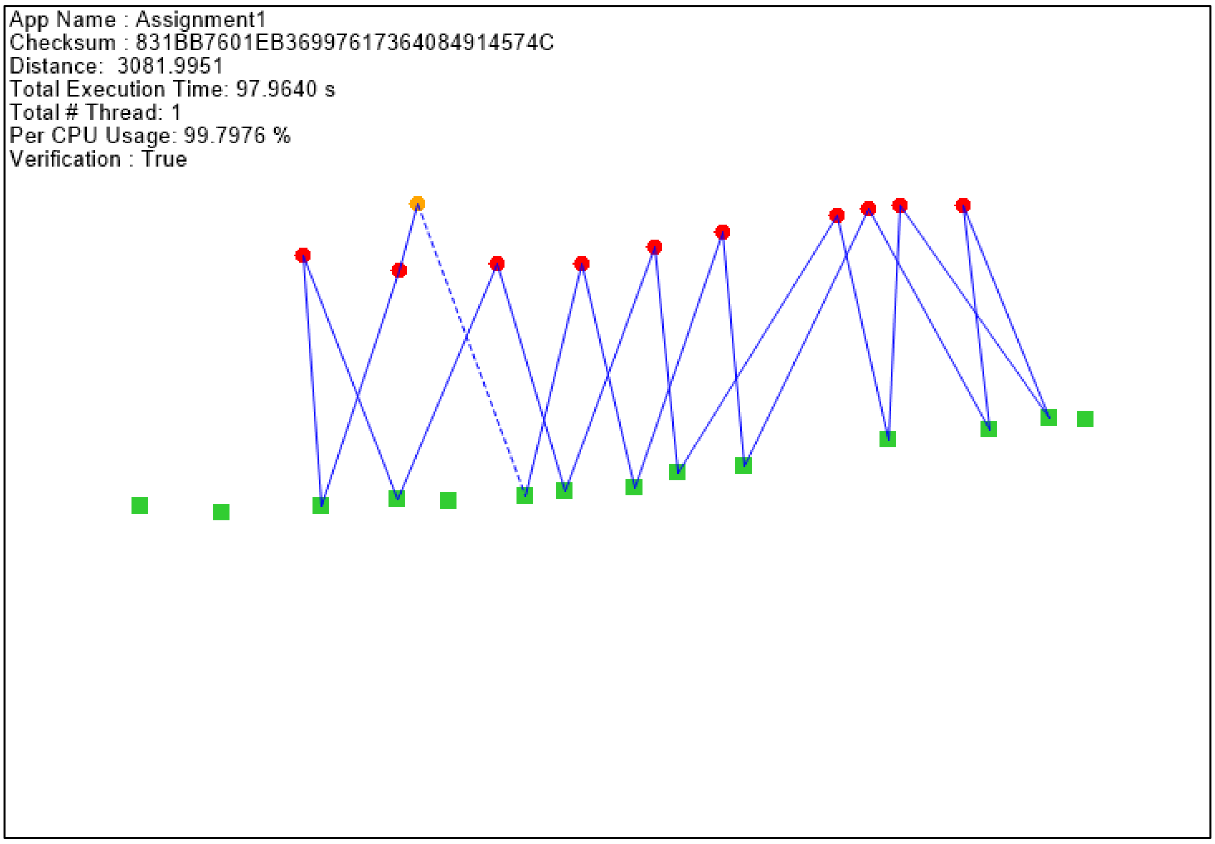
Section I: Appendix



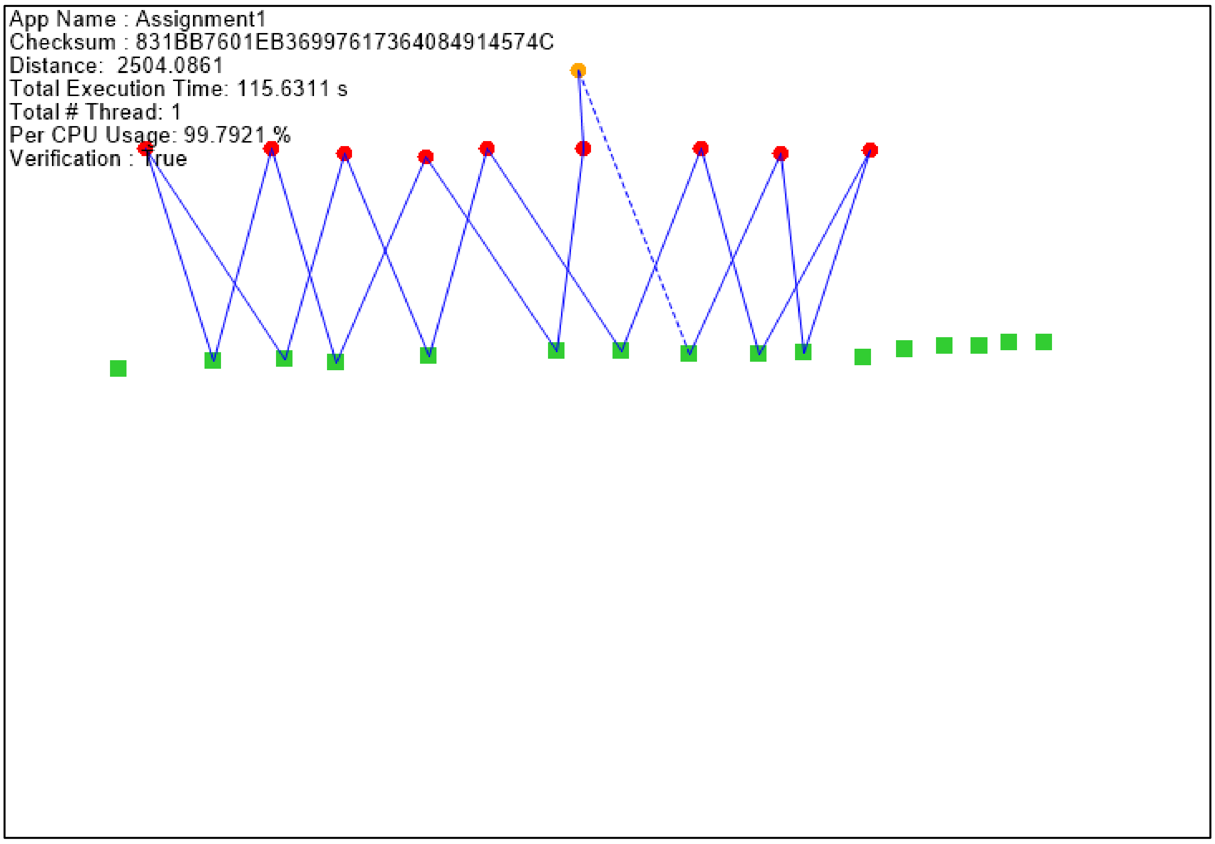
```
App Name : Assignment1
Checksum : 831BB7601EB36997617364084914574C
Distance:  3179.8907
Total Execution Time: 120.6216 s
Total # Thread: 1
Per CPU Usage: 99.951 %
Verification : True
```



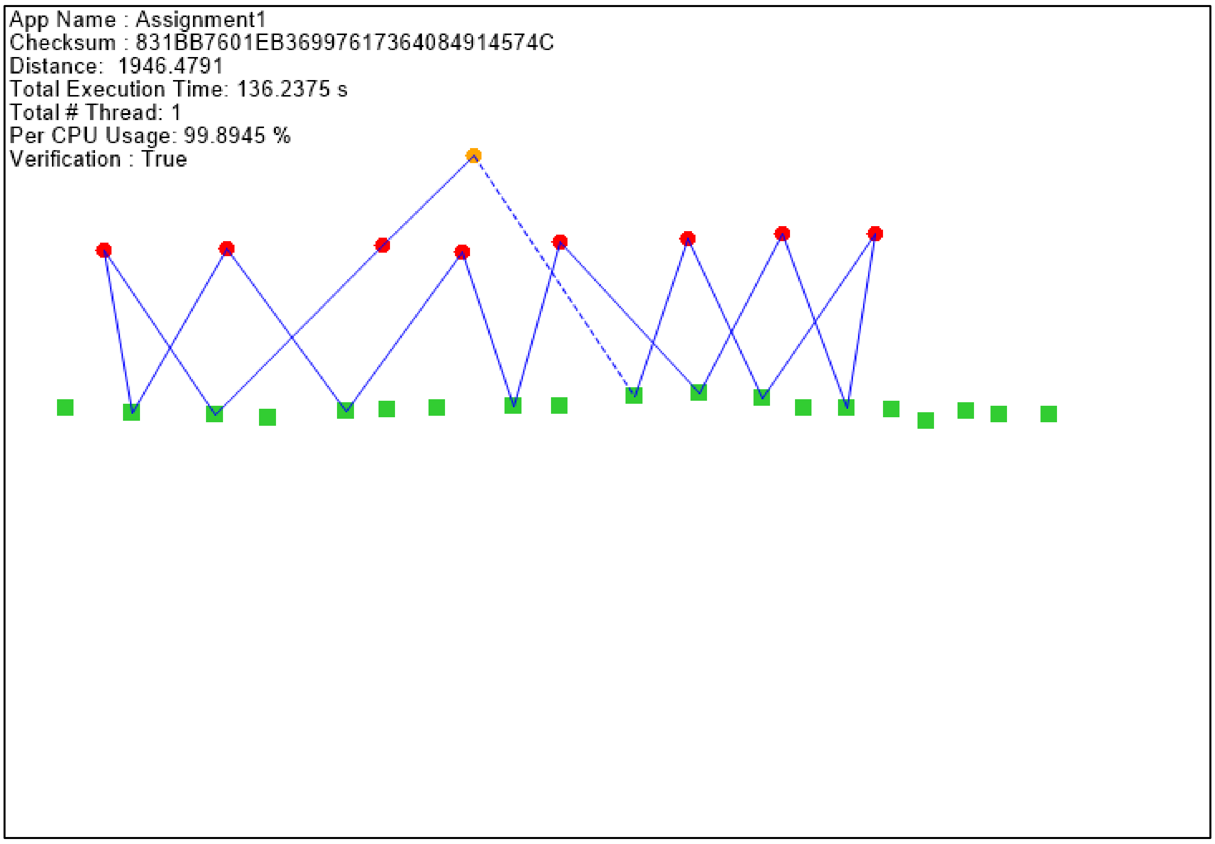
```
App Name : Assignment1
Checksum : 831BB7601EB36997617364084914574C
Distance:  2924.5012
Total Execution Time: 119.8770 s
Total # Thread: 1
Per CPU Usage: 99.894 %
Verification : True
```



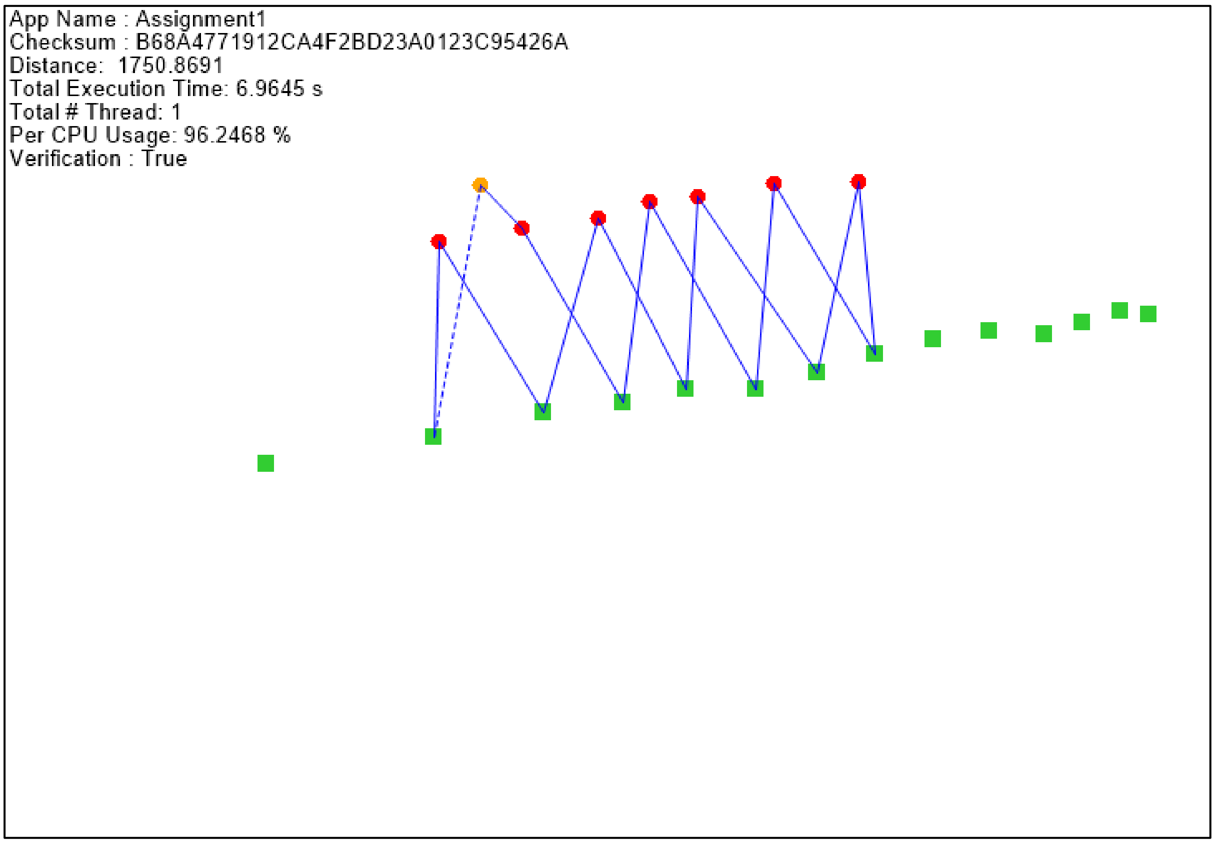
```
App Name : Assignment1
Checksum : 831BB7601EB369976173640849145 74C
Distance:  3081.9951
Total Execution Time: 97.9640 s
Total # Thread: 1
Per CPU Usage: 99.7976 %
Verification : True
```



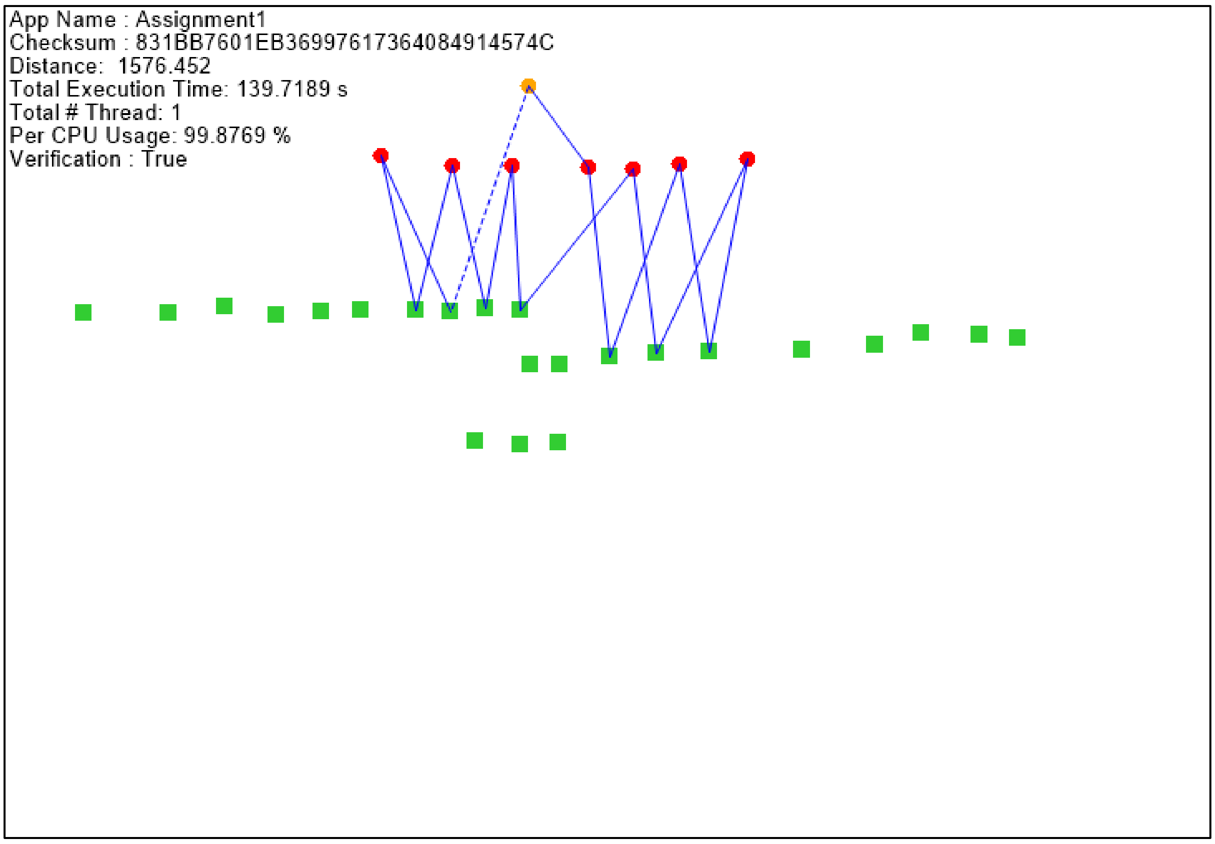
```
App Name : Assignment1
Checksum : 831BB7601EB369976173640849145 74C
Distance:  2504.0861
Total Execution Time: 115.6311 s
Total # Thread: 1
Per CPU Usage: 99.7921 %
Verification : True
```



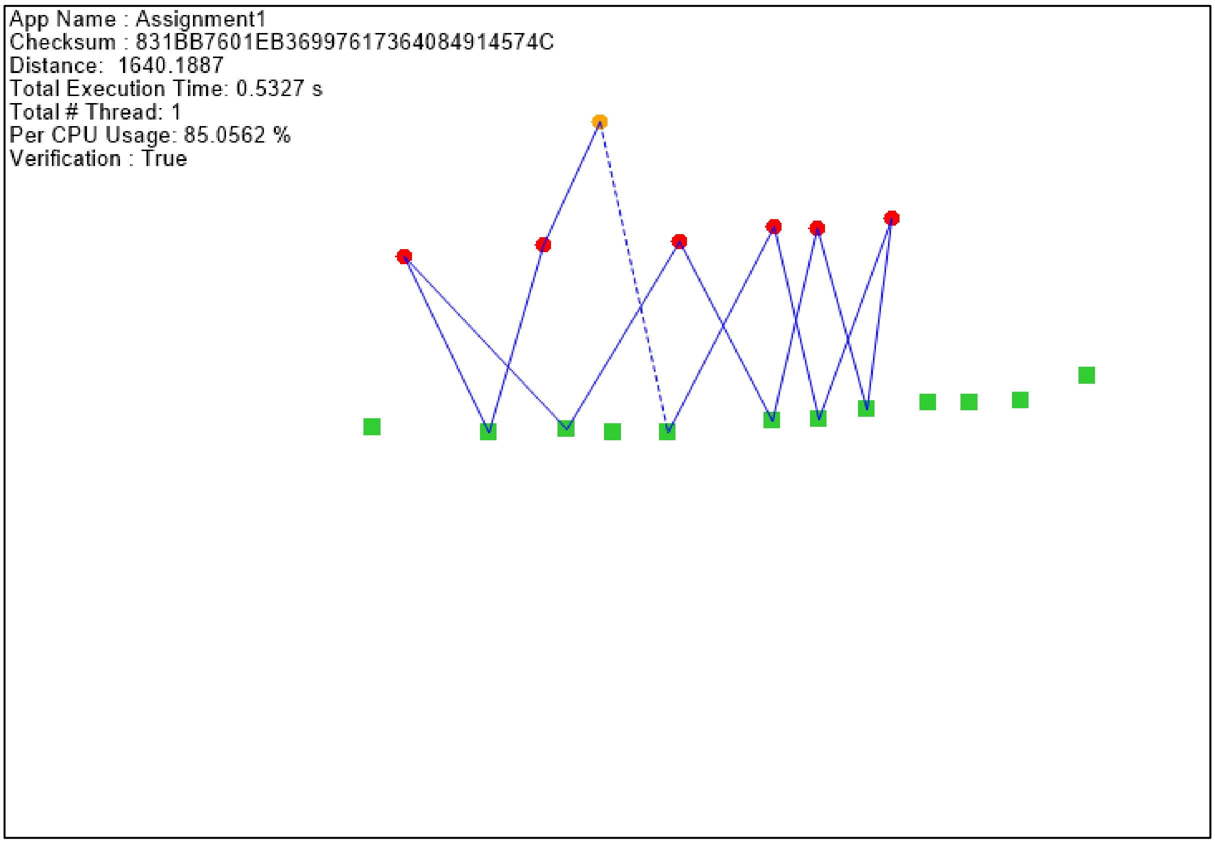
```
App Name : Assignment1
Checksum : 831BB7601EB36997617364084914574C
Distance:  1946.4791
Total Execution Time: 136.2375 s
Total # Thread: 1
Per CPU Usage: 99.8945 %
Verification : True
```



```
App Name : Assignment1
Checksum : B68A4771912CA4F2BD23A0123C95426A
Distance:  1750.8691
Total Execution Time: 6.9645 s
Total # Thread: 1
Per CPU Usage: 96.2468 %
Verification : True
```



```
App Name : Assignment1
Checksum : 831BB7601EB369976173640849145 74C
Distance:  1576.452
Total Execution Time: 139.7189 s
Total # Thread: 1
Per CPU Usage: 99.8769 %
Verification : True
```



```
App Name : Assignment1
Checksum : 831BB7601EB369976173640849145 74C
Distance:  1640.1887
Total Execution Time: 0.5327 s
Total # Thread: 1
Per CPU Usage: 85.0562 %
Verification : True
```