Wholly owned by UTAR Education Foundation
(Co. No. 578227-M)
DU012(A)

UCCD3073 Parallel Programming

Assignment 1

Name	Student ID	Programme
Tan Xi En	19ACB04098	CS
Lee Hao Jie	18ACB01544	CS

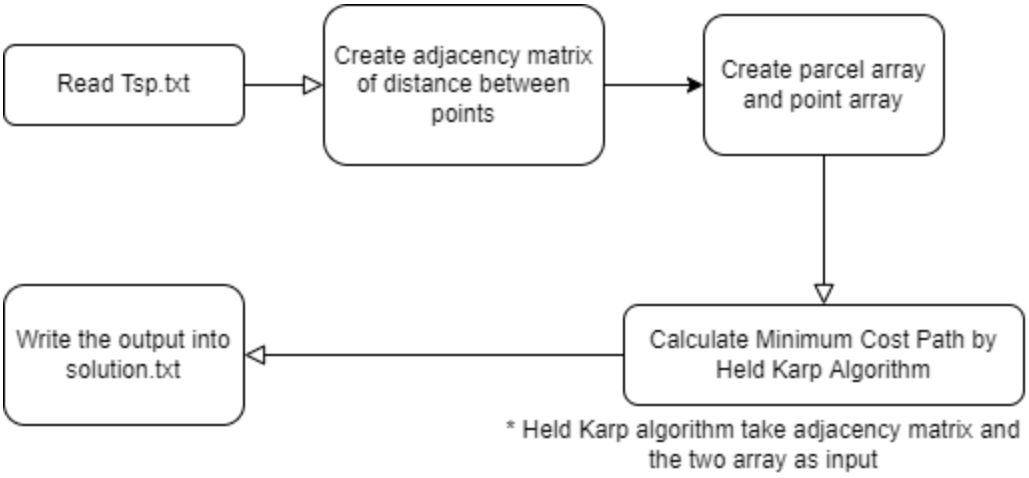
Introduction

The traveling salesman problem is one of the most well-known algorithmic problems in the fields of computer science and operations research. It asks the following question: “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?”. Although the traveling salesman problem was mathematically formulated in the 19th century, it is still continuously being researched by many leading experts in computer science and operation research. This is because TSP is an NP-hard problem, which means that there is no solution that can solve the problem in polynomial time. Good solutions (Heuristic solutions) have been continually proposed to solve TSP problem, however, not much progress have been done for exact solver solutions algorithms. Today, we are tasked with finding an exact solver for a TSP variant, where the salesman must pass at least 1 parcel points before passing each city. Is it possible to find a best route that travel all cities and return to starting point?

Problem Statement

For this TSP Variant problem, we are given a set of locations, comprising of n green points and m red points. We must find the minimum distance of tours that visits each red point once. The catch is before traveling to each red point, a green point must be visited first. The total distance, including the green points must be the shortest. Although it looks simply, this TSP problem have a time complexity of O(n!). The complexity of the problem will skyrocket with the increase of each point. To address this problem, the most common approach is to use depth first search with backtracking to go through each possible route and calculate the shortest path. However, the issue with this solution is that it is too computationally slow. Therefore, we propose to use a dynamic programming algorithm, known as Held-Karp algorithm to solve this TSP variant problem. Our results are very successful, we’re able to solve 27 points (9 Red, 18 Green, in 2.7 minutes.

Activity diagram



The above diagram is the activity diagram for the whole problem, we read tsp.txt as input and create an adjacency matrix. To calculate the cost of the tour, the Euclidean distance between two points is calculated and used as the cost value. To improve the performance of our algorithm, we assume the following distance as infinity value:

- Depot to Red Points
- Red Points to Red Points
- Green Points to Green Points
- Green Points to Depot

This is because the following route are impossible. By setting such a condition, we can improve the performance of our algorithm.

Once we have successfully created our adjacency matrix, the parcel points array and point points array are created to store green points and red points respectively. The adjacency matrix, parcel points array and point points array are passed into our modified held Karp algorithm to get the minimum cost and minimum cost path value.

To finish our program, we store the results of the cities passed into solution.txt. When the txt file is generated, the program shall end execution.

FUNCTION Explanation

For this section, I shall focus on only 4 functions, that is combinations(), binomialCoeff(), gen_total_calculation() and HeldKarpSingleThreaded(). This is because I believe the rest of the functions are self-explanatory. The explanation for the rest of the function will be included at abstract.

Combinations Function

```
// Combination Without Duplicate => Self Written
vector<vector<int>> combinations(vector<int> col_list, int r) {
    vector<vector<int>> combs;

    int end = col_list.size();

    if (r > end) return combs;

    vector<int> a(r), b(r);
    for (int i = 0; i < r; i++) {
        a[i] = i;
        b[i] = col_list[i];
    }

    while (true) {
        combs.push_back(b);

        int i = r - 1;

        while (i >= 0 && a[i] >= end - 1 - (r - 1 - i)) {
            i--;
        }

        if (i < 0) {
            break;
        }

        for (int j = a[i] + 1; i < r; j++, i++) {
            a[i] = j;
            b[i] = col_list[j];
        }
    }

    return combs;
}
```

Combination Function is used to generate unique combinations based on a vector of integer. For example, given vector of [1, 3, 5] and size of 2, it will generate 3C2 number of subsets. The subsets in the combs will be {[1, 3], [1, 5], [3, 5]}.

BinomialCoeff Function

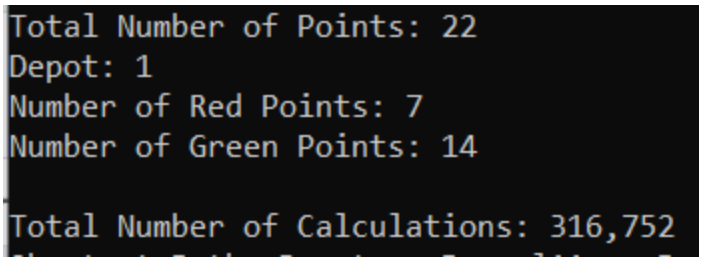
```
// Dynamic Binomial Coefficient to get the total number of combinations and Subset
int binomialCoeff(int n, int k)
{
    vector<int> C(k + 1, 0);
    C[0] = 1; // nC0 is 1
    for (int i = 1; i <= n; i++){
        for (int j = min(i, k); j > 0; j--) {
            C[j] = C[j] + C[j - 1];
        }
    }
    return C[k];
}
```

BinomialCoeff Function is used to generate nCk value. It parses in N and K, and returns an int value. For example, binomialCoeff(3, 2) returns 3C2. This function is important for generating the number of possible subset combination.

Gen_total_calculation Function

```
// This formula computes the total number of calculations needed
// This formula constructs the bottom root of the tree to the final subset
int gen_total_calculation(int gN, int rN) {
    int ans = 0;

    // 1 x 1
    ans += gN * rN;

    // 2 * 1, 2 * 2, 3 * 1, .... k * (l - 1) , k * l
    for (int k = 2; k < rN + 1; k++) {
        for (int l = k - 1; l < k + 1; l++) {
            ans += binomialCoeff(gN, k) * binomialCoeff(rN, l);
        }
    }
    return ans;
}
```

Gen_Total_Calculation Function generates the total number of combinations needed for the current parcel point and point points array. For example, given 7 red and 14 green, this function will output a value of 316,752.



You can consider this value as the number of ram needed to perform each calculation. By my machine standards, 1 possible subset requires 1 KB. So 316 752 requires at least 300 MB in my RAM. Having this calculation allows me to know whether the TSP question generated is possible for my machine.

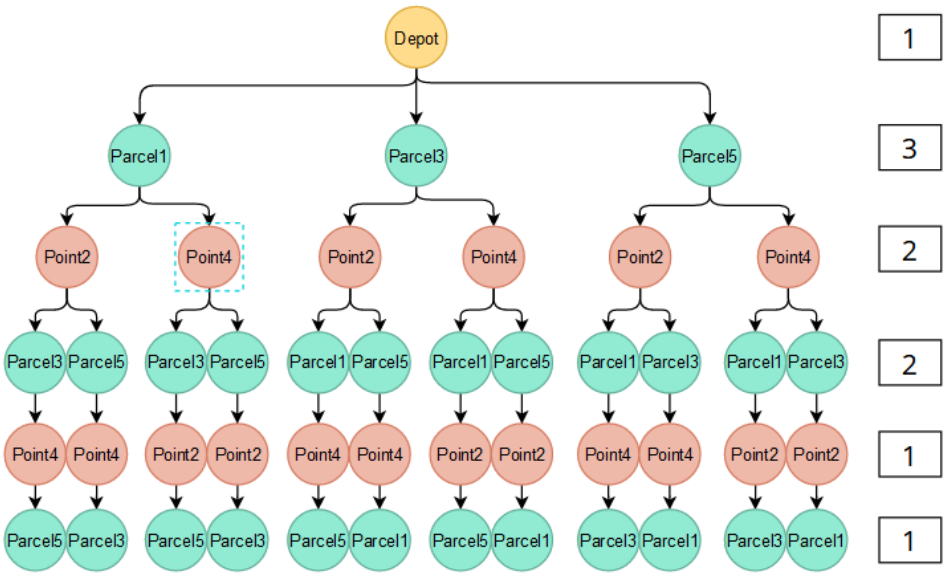
HeldKarpSingleThreaded Function

```
Path HeldKarpSingleThreaded(vector<vector<double>>, vector<int>, vector<int>, vector<string>);
```

Held-Karp algorithm, also called Bellman-Held-Karp algorithm, is a dynamic programming algorithm proposed in 1962 independently by Bellman, Held, Karp to solve the traveling salesman problem. The input for this algorithm is a distance matrix between a set of cities. This algorithm finds an exact solution to this problem in exponential time of O(N^2 * 2^N).

To understand why we chose the Held Karp algorithm, let's first explain how to solve this TSP variant problem using a brute force method, the Depth First Search algorithm. By constructing our TSP variant into an adjacency matrix, we are able to use Depth-First-Search to traverse all possible paths and get the minimum cost and minimum cost path respectively.

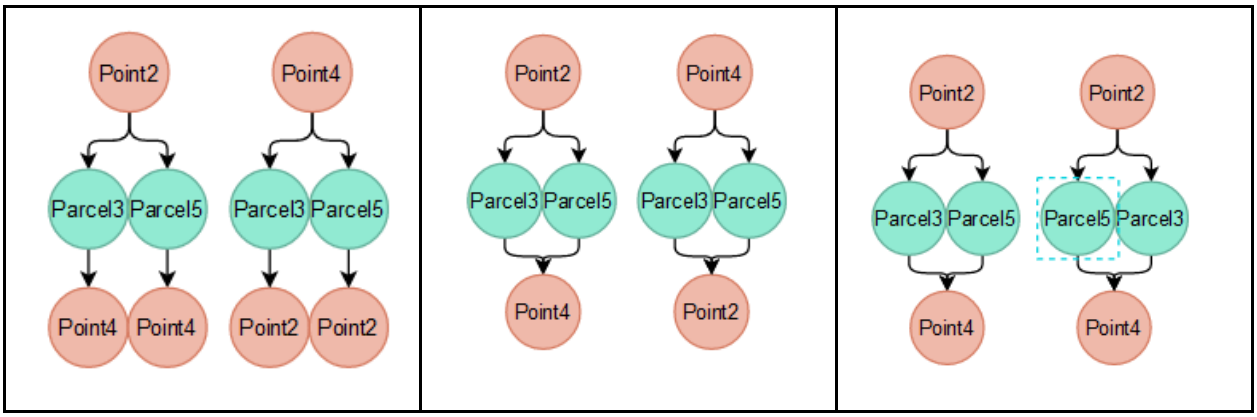
Given an example, where our parcel array (Green Points) is 1, 3, 5 and our point array (Red Point) is 2, 4, we would generate the following Depth-First-Search tree.



The following possible is 12 possible routes. This combination is derived from

1 * 3 * 2 * 2 * 1 * 1 = 12

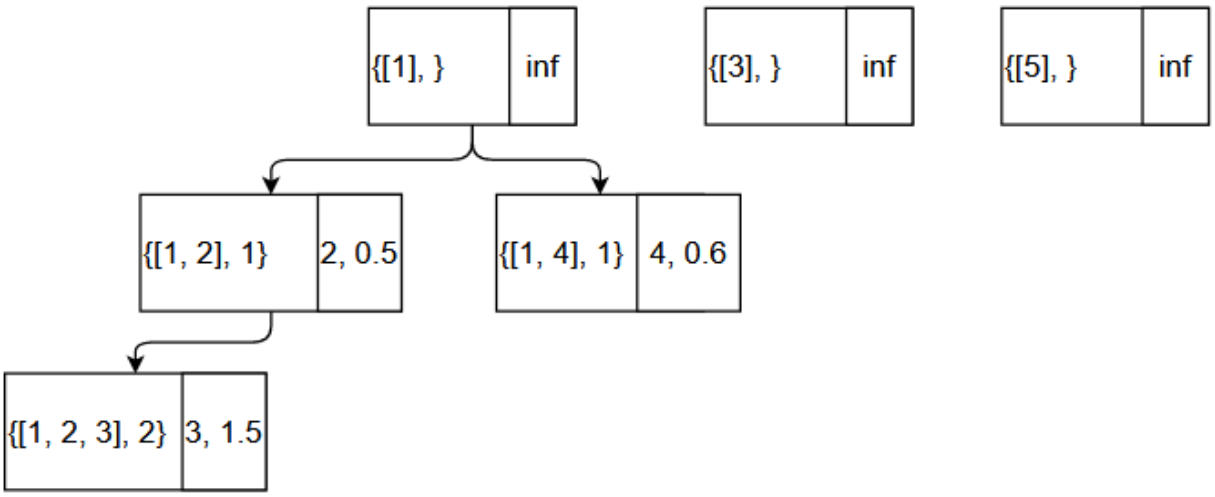
For small problems, it seems to be simple enough. However, DFS is computationally expensive. For this TSP variant, its time complexity is N!. Using gN as the size of Parcel array and rN as the size of Point array, the equation for the DFS complexity is derived using gN P rN * rN!, or mPn * n!.The reason why DFS is computationally expensive is because it overlaps between many common subsets. These common subsets are repeated calculations, just that they are in different order. The image below shows two common subsets that are just in different order.



The above two subset are similar, just that it is in a different order. If we can cache these repeating patterns using some sort of dynamic array, for example a dictionary, our algorithm would perform less calculations and compute a faster performance. To cache these results, we used a dynamic programming approach, to calculate the number of subsets needed and find the minimum cost value between each subsets. The formal name for this approach is known as the Held-Karp algorithm.

To cache our subsets, we must first declare the key and value stored in our dictionary / memoization table / hash map. We use an unordered map to cache our key-value pair, because unordered map has the fastest lookup time with complexity of O(1). For a dynamic programming with large calculations, the fast lookup time is necessary.

For our TSP variant problem, our key will be (visited subset, last visited node), our value will be (last node from current node, current cost). The binary tree below shows how our memoization method works.



To initialize our cache array, we initialize the key by using the index from the parcel array, and give it a value of infinity. Parcel array is used to initialize each key as the only possible point that can be traversed by depot is parcel points. This is to fulfill our problem requirements of traversing at least one parcel point before each red points. Rather than initializing our dictionary with each points, selectively initializing with parcel points helps to reduce the calculation needed.

The code to initialize our memory cache using parcel array is given below:

```
// Set transition cost from initial state
for (int k : parcel_arr) {
    key = to_string(1 << k) + to_string(k); //key as string
    umap[key] = make_pair(adj_mat[0][k], 0);
}
```

To improve our key index, rather than storing our visited subset as an array, we can store it as an integer. To compute this value, as indexes are integers, we may derive the sum of the subset using the sum of exponent 2 of each index. This method is known as bitmask programming. For example, given a route of 1, 2, 3, 4,5, to track whether 1, 3, 4 have been visited, we write the notation as 01101.. The sum of the current subset will be 2^0 + 2^2 + 2^3 = 13. Storing the subset as an integer number rather than an array, means faster retrieval and less memory usage.

Furthermore, in our TSP variant, all paths will follow a specific order, that is Depot -> Green -> Red -> Green -> Red …. -> Depot. After traversing every green point, we will traverse to another red point and vice versa. Understanding these concepts allows us to simplify our calculation, as we know our next last node will be the opposite of our current node. For example, if our current last node is green, our next last node will be red. Therefore, we create a struct called SubsetObj to store each combination of green points and each combination of red points respectively. SubsetObj allow us to switch between green points or red points based on the current last node.

The concept above is defined in the code below. The code section below is used to generate all possible combinations needed for the subsets.

```
vector<SubsetObj> subset_arr; // Generate Subset Array
// Generate All Possible Combinations and populate subset Array
for (int k : parcel_arr) {
    vector<int> gSubset = { k };
    for (int l : point_arr) {
        vector<int> rSubset = { l };
        subset_arr.emplace_back(SubsetObj(gSubset, rSubset));
    }
}

for (int k = 2; k < rN + 1; k++) {
    vector<vector<int>> gSubsetArr = combinations(parcel_arr, k);
    for (vector<int> gSubset : gSubsetArr) {
        for (int l = k - 1; l < k + 1; l++) {
            vector<vector<int>> rSubsetArr = combinations(point_arr, l);
            for (vector<int> rSubset : rSubsetArr) {
                subset_arr.emplace_back(SubsetObj(gSubset, rSubset));
            }
        }
    }
}
```

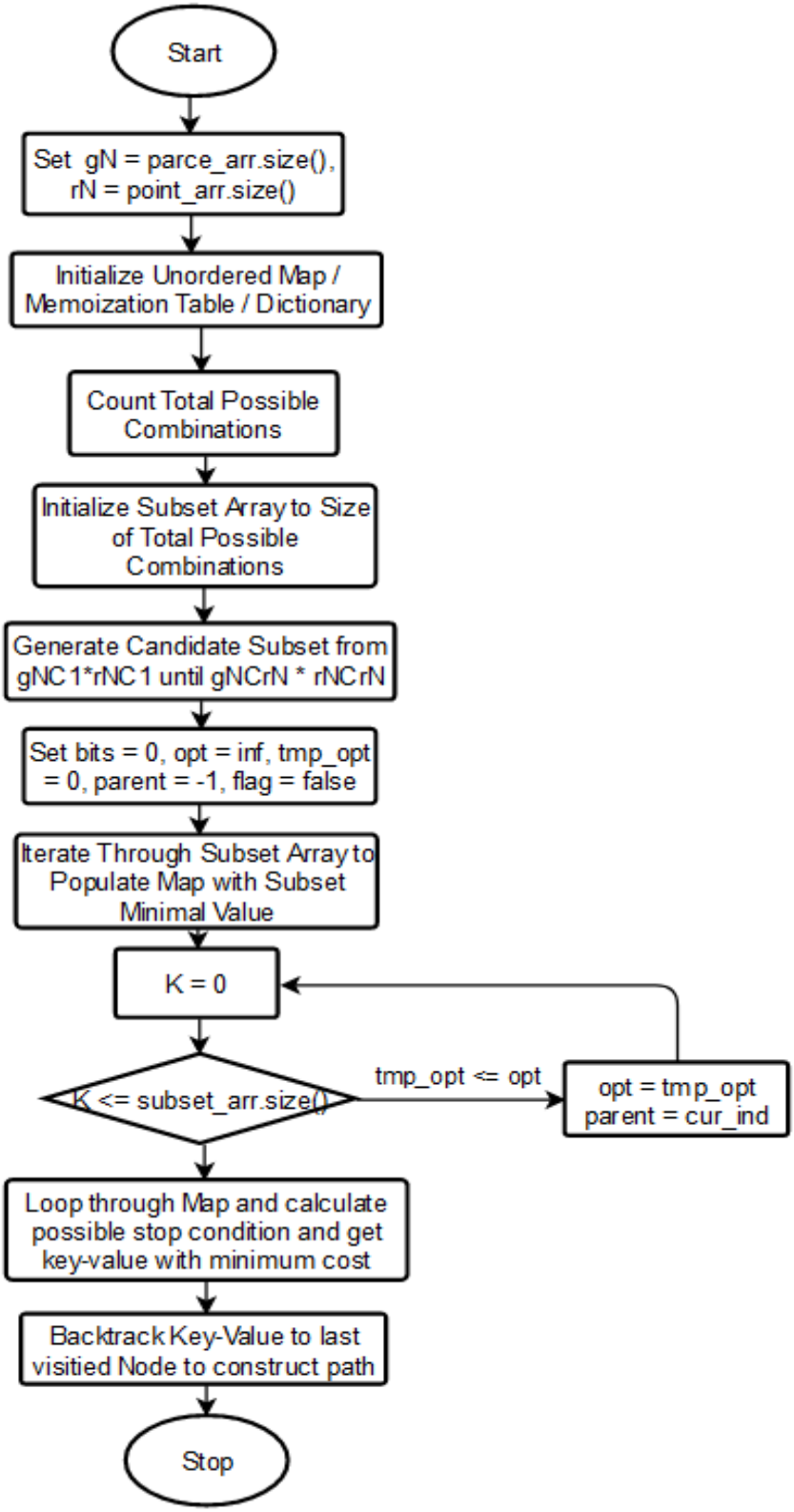
After we have filled up our memory cache array, with all possible subset and minimum cost, we must now compute the shortest path. Following our above example of green parcel, 1, 3, 5 and red parcel, 2, 4, we know that the final possible path will be 1234, 1245 and 2345. This is because these subsets traverse through all possible red points, that is 2 and 4. Therefore, for the above example, there will only be 3 possible subset paths. To look up the minimum cost for each subset, we first compute the bit value of these 3 subsets, which will be 20, 54 and 60 respectively. We put these values into our route_arr and declare these values as stop condition. We then iterate through all the keys and get the minimum cost of the possible values in the route_arr. We then store the minimum cost value into a new path object.

```
// Get All Possible Final Paths (E.g. [1 2 3 4], [1 2 4 5], [2 3 4 5])
for (int route_bits : route_arr) {
    for (int k : point_arr) {

        key = to_string(route_bits) + to_string(k); // Prev Key
        tmp_opt = umap[key].first + adj_mat[k][0];

        // Replace Value if Current Cost is smaller than Best Cost
        if (tmp_opt <= opt) {
            opt = tmp_opt;
            parent = k;
            bits = route_bits;
        }
    }
}
```

Finally, to get the shortest path from Depot to the current last node, we just need to backtrack through the last node from each key. We know the final possible path will have a length of 2 * size of red point array + 2, where 2 * size of red point array is derive from green -> red -> green -> red …, and +2 is from Depot at start and Depot at end. Using the memory cache as a form of linked list, we get the last node from the current key and backtrack upwards until we hit Depot.

```
// Backtrack to find full path
vector<string> path(2 * rN + 2, "Depot");

for (int i = 2 * rN; i > 0; i--) {
    // Map Path Index-Element to Col List Index-Element
    path[i] = name_arr[parent];
    key = to_string(bits) + to_string(parent);
    bits = bits & ~(1 << parent);
    parent = umap.at(key).second;
}
```

The flowchart diagram showcases how the flow of the entire algorithm



Results and Analysis

To perform our experiment, we have tested our code on 4*8, 5*10, 6*12, 7*12, 8*16 and 9*18 respectively. Using our modified HeldKarp algorithm, we can achieve a time of less than 3 minutes in each of these subsets. These results are conducted using x64 compiler of C++14 version, release mode.

The experimental results and analysis is provided in the table below.

Red Points	Green Points	Total Points (Include Depot)	Num of Calculation	Time (s)
4	8	13	1,222	0.0942
5	10	16	7,787	0.0700
6	12	19	49,583	0.2320
7	14	22	316,752	2.2108
8	16	25	2,031,518	21.3216
9	18	28	13,079,333	214.199

As you can observe, the number of calculations needed increases exponentially, therefore, the time spent also increases exponentially. The results overall are very promising, as we observe that our program can solve 25 points in under 21s, albeit selectively! The program however is limited due to hardware issues. 25 points requires a whopping among of 2GB free ram, which is quite a heavy workload on our machine. Perhaps with a better machine with more ram, we can benchmark more successfully.

There is also an issue with 9*18 points. The number of combinations is 13, 079, 333, which is approximately equivalent to 13GB Ram. Due to the large number of RAM, and limitations in our machine, we are unable to get the accurate result as The code stops executing when it hits 11 GB Ram.

These experiments are conducted using machine:

Process:AMD RYZEN 5600x

Memory:16GB RAM

Conclusion

In conclusion, our HeldKarpSingleThreaded Algorithm is able to perform as expected and derive the shortest possible tour that visits all red points once, a green point before each red points and return to depot successfully. The program returns the minimum cost and minimum tour with no issues. For our next assignment, we shall use a multi-threaded approach to see if we're able to optimize our algorithm even further.

APPENDIX (More Function explanation)

We use structure to define custom data types to store point and path information. Point structure includes the 3 variables which are name , x and y-coordinates. In the path structure which has the cost and the dynamic array (vector) to store the path array.

minPath Function

```
Path minPath(Path pth1, Path pth2) {
    return (pth1.cost < pth2.cost) ? pth1 : pth2;
}
```

This function will compare two path and return the path with smaller cost

distance Function

```
double distance(Point p1, Point p2) {
    return sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2));
}
```

This function will return the distance between two point by the formula

Split Function

```
vector<string> split(string str, string delim) {
    vector<string> tmp_arr;
    int ind = 0;
    string token = "";
    while ((ind = str.find(delim)) != string::npos) {
        token = str.substr(0, ind);
        tmp_arr.push_back(token);
        str.erase(0, ind + delim.length());
    }
    tmp_arr.push_back(str);
    return tmp_arr;
}
```

This function is use to split the line by delimiter for example we use this function to split “Depot 161 132” into three string by space delimiter so we can get the point information

containsSubStr Function

```
bool containsSubStr(string str, string str2) {
    return str.find(str2) != string::npos;
}
```

This function is use to find the substr , we use it to find the point name is include “Parcel” or “Point”

create_adj_mat Function

```
vector<vector<double>> create_adj_mat(vector<Point> pt_arr) {
    int x_ind(0), y_ind(0), n(pt_arr.size());

    vector<vector<double>> adj_mat(n, vector<double>(n, inf));

    bool flag = false;

    for (Point pt : pt_arr) {
        string name = pt.name;

        y_ind = 0;

        for (Point pt2 : pt_arr) {
            flag = pt.name == pt2.name
                || (containsSubStr(pt.name, "Depot") && containsSubStr(pt2.name, "Point"))
                || (containsSubStr(pt.name, "Point") && containsSubStr(pt2.name, "Point"))
                || (containsSubStr(pt.name, "Parcel") && containsSubStr(pt2.name, "Parcel"))
                || (containsSubStr(pt.name, "Parcel") && containsSubStr(pt2.name, "Depot"));
            if (!flag) adj_mat[x_ind][y_ind] = distance(pt, pt2);
            y_ind++;
        }

        x_ind++;
    }
    return adj_mat;
}
```

This function is used to build the Adjacency Matrix of the distance of different points. We initialize all distance to infinity and calculate distance of points except 4 condition which is

1. Depot to Red Point
2. 2. Red Point to Red Point
3. 3.Green Point to Green Point
4. 4.Green Point to Depot

Create_parcel_arr Function

```
// Create Parcel Array
vector<int> create_parcel_arr(vector<Point> pt_arr) {
    vector<int> parcel_arr;

    for (int i = 0; i < pt_arr.size(); i++) {
        if (containsSubStr(pt_arr[i].name, "Parcel")) {
            parcel_arr.push_back(i);
        }
    }

    return parcel_arr;
}
```

This function is use to build the parcel array

Create_parcel_arr Funcrtion

```
// Create Point Array
vector<int> create_point_arr(vector<Point> pt_arr) {
    vector<int> point_arr;

    for (int i = 0; i < pt_arr.size(); i++) {
        if (containsSubStr(pt_arr[i].name, "Point")) {
            point_arr.push_back(i);
        }
    }

    return point_arr;
}
```

This function is use to build point array

formatLargeNum Function

```
string formatLargeNum(int v, string delim) {
    string s = to_string(v);

    int end = (v >= 0) ? 0 : 1; // Support for negative numbers
    for (int n = s.length() - 3; n > end; n -= 3) {
        s.insert(n, delim);
    }

    return s;
}
```

This function is use to make large number has better look