



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

UCCD3073 Parallel Programming

Assignment 2

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

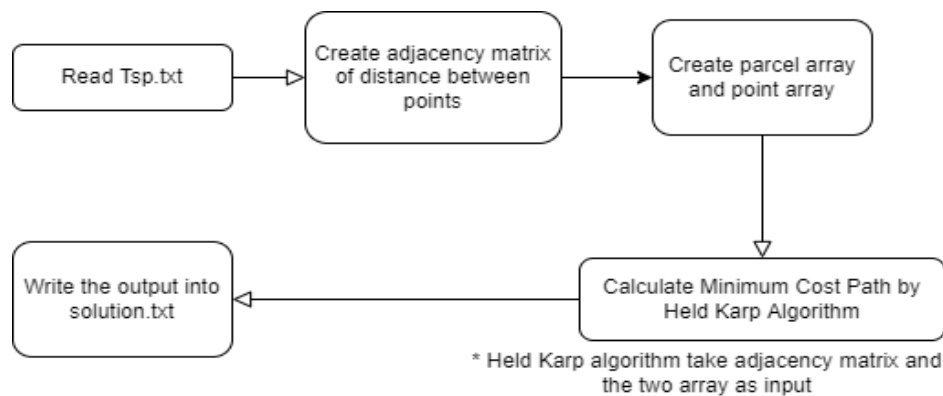
Introduction

On pervious assignment we discuss about what travelling salesman problem and type. With the rise of multi-core processer parallel programming are but fancy stuff but can be implement with different library. We are here to implement parallelism on our algorithm which is Held Karp in order make use the resource of multicore processer to speed up our program.

Problem Statement

This TSP Variant problem is given set of green and red point and we have to find the route with minimum distance. The route must start at the depot and go back to depot and other requirements is before pass through a red point a green point must be meet first. The problem is no changing but with additional requirement to speed up or parallelized the algorithm. Previously, we purposed Held-Karp algorithm to solve this TSP variant problem. Our pervious results are very successful, we're able to solve 27 points (9 Red, 18 Green, in 2.7 minutes). The objective of this assignment is to make the program faster that the pervious program.

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.

The program takes most of the time at the calculate minimum cost with help Karp by help Karp function process. This process will be given attention to implement parallelism

Function Explanation

There are two function we implement parallelism and others does not change and same as the pervious assignment. This function explanation will focus on explain how we do parallelism on the two functions. Other function explanation will document at the appendix.

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; j < end; j++) {
            a[i] = j;
            b[i] = col_list[j];
        }
    }

    return combs;
}
```

Combination Function is used to generate unique combinations based on a vector of integers. For example, given a vector of [1, 3, 5] and size of 2, it will generate a 3C_2 number of subsets. The subsets in the combs will be {[1, 3], [1, 5], [3, 5]}.

HeldKarpMultiThreadedParallelWindows

Normally, for a parallel implementation of an algorithm, it would be efficient to try to parallelize for loops, rather than refactor our coding to make it parallelizable. Therefore, we shall be focusing our attention on for loops in our algorithm. After a deep thorough analysis of our algorithm, we have discovered two parts of our algorithm where we could parallelize without any issue.

The difficulty of parallelism is the codependence of our code. We cannot do our future task, if we do not have the past task at hand. Some coding in for loops have codependence. The current value of the for loop is dependent on the previous value. To make a parallel algorithm, we have to first identify sub tasks that are repeatable and not related. Therefore, for some for loops, we cannot run them concurrently otherwise the value of our function will not be identical to its serializable counterpart.

After analyzing our Held Karp algorithm, we have discovered two parts where we could parallelize efficiently. The parts that are parallelizable in our algorithm are:

1. Generating Candidate Subsets for our possible paths
2. Calculating Minimum Value among the subsets in the group

After numerous testings, we have discovered that Microsoft PPL library performs much better than async future. The following code below is how we successfully convert our single threaded code into multi-threaded

Generate Candidate Subset

For generating candidate subsets in single threaded, we can observe that there are a lot of for loops. These for loops can be parallelized. However, not all for loops can be parallelized, because parallelism will cause function overhead. Excessive parallelism will not improve the time nor make our program more efficient, but instead increase execution time instead. Therefore, we only parallelize the last for loops, because the number of combinations needed is the size of the green subset array times the size of the red subset array.

Single Threaded

The code below is our single threaded implementation of generating candidate subset.

```
vector<vector<vector<int>>> parcel_comb_arr(rN);
for (int k = 1; k < rN + 1; k++) {
    parcel_comb_arr[k - 1] = combinations(parcel_arr, k);
}

vector<vector<vector<int>>> point_comb_arr(rN);
for (int k = 1; k < rN + 1; k++) {
    point_comb_arr[k - 1] = combinations(point_arr, k);
}

for (vector<int> gSubset : parcel_comb_arr[0]) {
    for (vector<int> rSubset : point_comb_arr[0]) {
        subset_arr[subset_ind++] = SubsetObj(gSubset, rSubset);
    }
}

for (int k = 2; k < rN + 1; k++) {
    for (vector<int> gSubset : parcel_comb_arr[k - 1]) {
        for (vector<int> rSubset : point_comb_arr[k - 2]) {
            subset_arr[subset_ind++] = SubsetObj(gSubset, rSubset);
        }
    }

    for (vector<int> gSubset : parcel_comb_arr[k - 1]) {
        for (vector<int> rSubset : point_comb_arr[k - 1]) {
            subset_arr[subset_ind++] = SubsetObj(gSubset, rSubset);
        }
    }
}
```

Multi-Threaded:

The code below is our implementation of generating candidate subset in parallel.

```
vector<vector<vector<int>>> parcel_comb_arr(rN);
for (int k = 1; k < rN + 1; k++) {
    parcel_comb_arr[k - 1] = combinations(parcel_arr, k);
}

vector<vector<vector<int>>> point_comb_arr(rN);
for (int k = 1; k < rN + 1; k++) {
    point_comb_arr[k - 1] = combinations(point_arr, k);
}

subset_ind = 0;
subset_arr[0].resize(gN * rN);
t_gN = parcel_comb_arr[0].size();
t_rN = point_comb_arr[0].size();
subset_size = t_gN * t_rN;
for (vector<int> gSubset : parcel_comb_arr[0]) {
    for (vector<int> rSubset : point_comb_arr[0]) {
        subset_arr[0][subset_ind++] = SubsetObj(gSubset, rSubset);
    }
}

for (int k = 2; k < rN + 1; k++) {
    t_gN = parcel_comb_arr[k - 1].size();
    t_rN = point_comb_arr[k - 2].size();
    subset_size = t_gN * t_rN;
    subset_arr[2 * k - 3].resize(subset_size);

    parallel_for(int(0), subset_size, [&](int ij) {
        vector<int> gSubset = parcel_comb_arr[k - 1][ij / t_rN];
        vector<int> rSubset = point_comb_arr[k - 2][ij % t_rN];
        subset_arr[2 * k - 3][ij] = SubsetObj(gSubset, rSubset);
    });
}

for (int k = 2; k < rN + 1; k++) {
    t_gN = parcel_comb_arr[k - 1].size();
    t_rN = point_comb_arr[k - 1].size();
    subset_size = t_gN * t_rN;
    subset_arr[2 * k - 2].resize(subset_size);

    parallel_for(int(0), subset_size, [&](int ij) {
        vector<int> gSubset = parcel_comb_arr[k - 1][ij / t_rN];
        vector<int> rSubset = point_comb_arr[k - 1][ij % t_rN];
        subset_arr[2 * k - 2][ij] = SubsetObj(gSubset, rSubset);
    });
}
```

Our parallel approach for implementation of generating candidate subset is a success. We are able to generate the exact same candidate subset with less execution Time.

Calculate Minimum Value Among Candidate Subsets

For calculating minimum subset value in candidate subset, we can observe that the entire process is just a single for loop. Luckily, calculating minimum subset value is a single subtask that is not related to previous or next object in the subset array. This indicates that this for loop is parallelizable. The code below shows our implementation in parallelizing the for loop.

Single Threaded

```
// Iterate subsets of increasing length and store
// intermediate results in classic dynamic programming manner
for (SubsetObj obj : subset_arr) {
    if (obj.parcel_arr.size() == obj.point_arr.size()) {
        // The last node FROM THE PREVIOUS SUBSET will be a green point
        subset2 = obj.parcel_arr;
    }
    else {
        // The last node FROM THE PREVIOUS SUBSET will be a red point
        subset2 = obj.point_arr;
    }

    subset1 = vector<int>(obj.size);

    ind = 0;

    for (int elem : obj.parcel_arr) {
        subset1[ind++] = elem;
    }

    for (int elem : obj.point_arr) {
        subset1[ind++] = elem;
    }

    bits = 0;
    for (int elem : subset1) {
        bits |= ((long long) 1 << elem);
    }

    // Find the lowest cost to get to this subset
    for (int k : subset1) {
        prev = bits & ~((long long)1 << k);

        opt = inf;
        parent = -1;
        flag = false;

        for (int m : subset2) {
            key = prev * 100 + m; //key as string

            // Check if Key exist in our Map
            if (umap.find(key) != umap.end()) {
                flag = true;

                tmp_opt = umap.at(key).first + adj_mat[m][k];

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

        if (flag) {
            key = bits * 100 + k;
            umap[key] = make_pair(opt, parent);
        }
    }
}
```


Multi-Threaded:

To parallelize our for loop, we could put the contents of the for loop into a separate function for easier implementation. Therefore, we create the function `parallelHeldKarp` that calculates the minimum subset value based on the subset object passed into the function.

parallelHeldKarpSingle Function

```
void parallelHeldKarpSingle(SubsetObj obj) {
    vector<int> subset1, subset2;
    long long bits, prev, key;
    bool flag;

    double opt = inf, tmp_opt = 0;
    int parent = 0, ind = 0;

    if (obj.parcel_arr.size() == obj.point_arr.size()) {
        // The last node FROM THE PREVIOUS SUBSET will be a green point
        subset2 = obj.parcel_arr;
    }
    else {
        // The last node FROM THE PREVIOUS SUBSET will be a red point
        subset2 = obj.point_arr;
    }

    subset1 = vector<int>(obj.size, 0);

    ind = 0;
    bits = 0;

    for (int elem : obj.parcel_arr) {
        subset1[ind++] = elem;
        bits |= ((long long)1 << elem);
    }

    for (int elem : obj.point_arr) {
        subset1[ind++] = elem;
        bits |= ((long long)1 << elem);
    }

    // Find the lowest cost to get to this subset
    for (int k : subset1) {
        prev = bits & ~((long long)1 << k);

        opt = inf;
        parent = -1;
        flag = false;

        for (int m : subset2) {
            key = prev * 100 + m; // key prev

            if (c_hash_map.find(key) != c_hash_map.end()) { // Not Thread Safe
                flag = true;
                tmp_opt = c_hash_map.at(key).first + adj_mat[m][k];

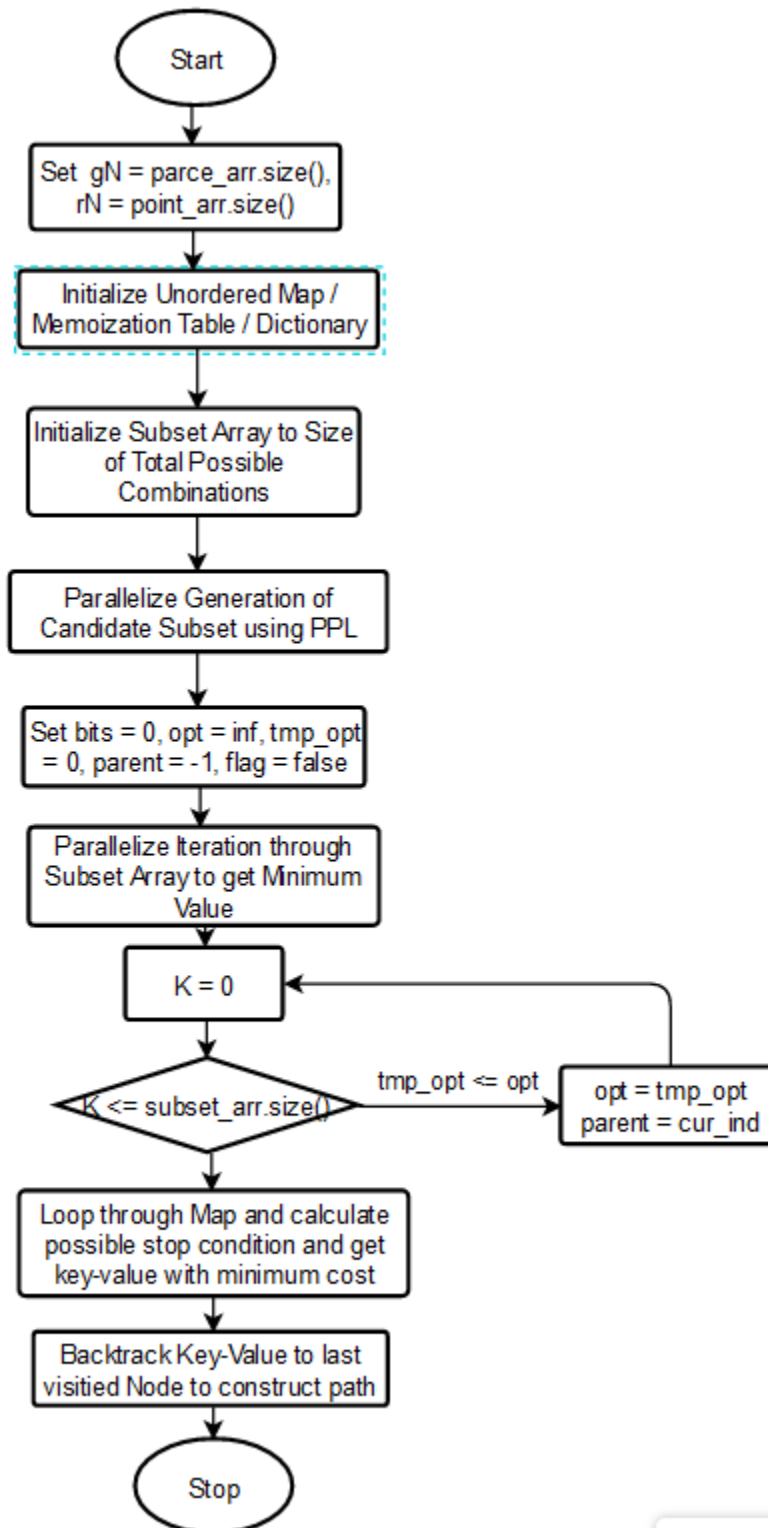
                // Replace Value if Current Cost is smaller than Best Cost
                if (tmp_opt <= opt) {
                    opt = tmp_opt;
                    parent = m;
                }
            }
        }

        if (flag) {
            key = bits * 100 + k;
            c_hash_map[key] = make_pair(opt, parent); // Not Thread Safe
        }
    }
}
```

Parallel Implementation

```
for (vector<SubsetObj> arr : subset_arr) {  
    subset_size = arr.size();  
    parallel_for(int(0), subset_size, [&](int i) {  
        parallelHeldKarpSingle(arr[i]);  
    });  
}
```

The flowchart diagram showcases the flow of the entire parallel held Karp algorithm

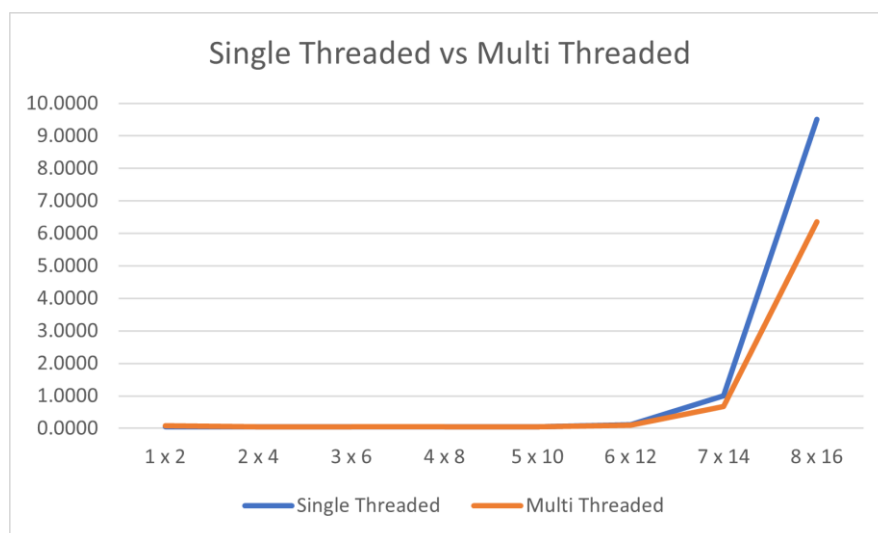


Result Analysis

To perform our experiment, we have tested our single threaded implementation and multi-threaded implementation on 1*2, 2*4, 3*6, 4*8, 5*10, 6*12, 7*12, and 8*16 respectively. Both of our Single threaded and multi-threaded implementations achieve a time of less than 2 minutes in all of the above test cases. Our multi-threaded approach is implemented using Microsoft PPL library. These results are conducted using the x64 compiler of C++14 version, release mode.

Red Points	Green Points	Total Points (Include Depot)	Single Threaded Time (ms)	Multi Threaded Time (ms)	Efficiency
1	2	4	0.0432	0.0796	0.54x
2	4	7	0.0465	0.0473	0.98x
3	6	10	0.0440	0.0470	0.94x
4	8	13	0.0465	0.0470	0.99x
5	10	16	0.0530	0.0543	0.98x
6	12	19	0.1191	0.1036	1.15x
7	14	22	1.0031	0.6761	1.48x
8	16	25	9.5099	6.3536	1.50x

Below is a graph of the above table



As you can observe, the number of calculations needed increases exponentially, therefore, the time spent also increases exponentially. From the above graph, we can see that our multi-threaded implementation is much faster than our single threaded implementation. Both graphs have a graph of $2^n * n^2$, just that, our multi-threaded graphs have a much smaller size. This indicates that our multi-threaded implementation is working as intended.

From our calculations, we can observe that we did indeed complete our assignment target of creating a parallel implementation that is 1.3x faster than our single threaded approach. Our parallel implementation is approximately 1.5x faster for $7*14$ and $8*16$ points.

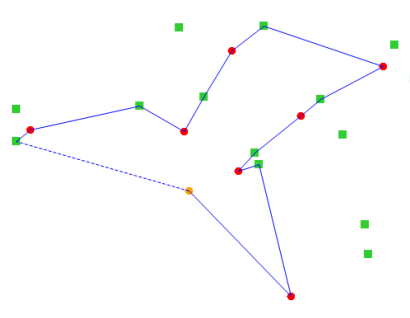
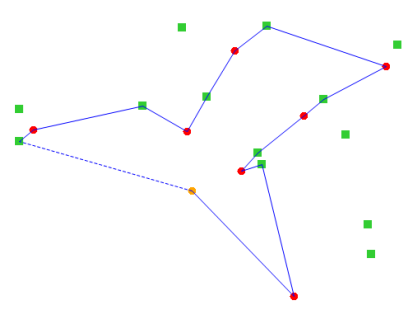
Although our parallel implementation is much faster, we did not do testing for points above $8*16$. This is because our parallel implementation is still limited to memory issue. Parallel implementation does not mean that our program has extra memory, it just means that our program will execute faster. Therefore, testing more points on our parallel implementation will be meaningless.

These experiments are conducted using machine:

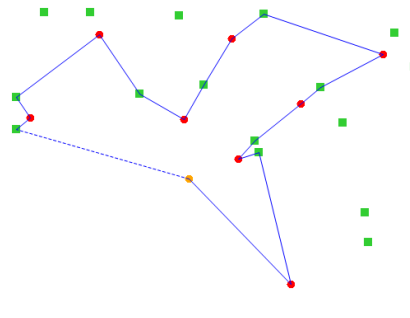
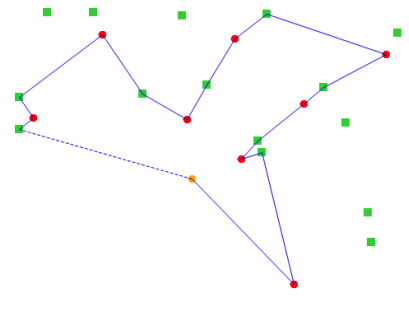
Processor: AMD RYZEN 5600x

Memory:16GB RAM

7 red points * 14 green points (22 points)

Single Threaded	Multi-Threaded
<p>App Name : Tsp-Solver-SingleThreaded Checksum : 352BCFD3FA98E6269EB2D0438CA787F6 Distance : 1342.8062 Total Execution Time: 1.0031 s Total # Thread: 1 Per CPU Usage: 96.5799 % Verification : True</p> 	<p>App Name : TSP-Solver-MultiThreaded-PPL Checksum : 451A1550E7362EA33DAB16957E871013 Distance : 1342.8062 Total Execution Time: 0.6761 s Total # Thread: 1 Per CPU Usage: 628.5944 % Verification : True</p> 

8 red points * 16 green points (25 points)

Single Threaded	Multi-Threaded
<p>App Name : Tsp-Solver-SingleThreaded Checksum : 352BCFD3FA98E6269EB2D0438CA787F6 Distance : 1450.4058 Total Execution Time: 9.5099 s Total # Thread: 1 Per CPU Usage: 99.7313 % Verification : True</p> 	<p>App Name : TSP-Solver-MultiThreaded-PPL Checksum : 451A1550E7362EA33DAB16957E871013 Distance : 1450.4058 Total Execution Time: 6.3536 s Total # Thread: 1 Per CPU Usage: 773.4273 % Verification : True</p> 

Conclusion

In conclusion, our multi-threaded approach did indeed achieve a faster speed of 1.3x faster than our single threaded approach. Thanks to parallelizing two sections of our single threaded approach, we can achieve a speedup of 1.5x. The route and cost value are identical which indicates that our multi-threaded held Karp is working as intended. Hopefully, for future work, we can design an even better parallel approach.

Appendix

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. Red Point to Red Point
3. Green Point to Green Point
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 Function

```
// 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

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.

```
Total Number of Points: 22
Depot: 1
Number of Red Points: 7
Number of Green Points: 14
Total Number of Calculations: 316,752
```