VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE & ENGINEERING

——————— * ———————

TRAVELLING SALESMAN PROBLEM'S sOLUTION AND EXPLAINATION

Instructor: Trần Tuấn Anh
—o0o—
Student: Nguyễn Tiến Phát__2452946

0.1 Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is an optimization problem where a salesperson must visit a given set of cities exactly once, starting and ending at the same city. The goal is to find the shortest possible route that covers all the cities and returns to the starting point.

TSP is an NP-hard problem, meaning there is no known efficient solution for large datasets, but various algorithms can provide exact or approximate solutions. This is one of the classical problems in Computer Science.

Define weighted graph G = (V, E, ω) as a weighted Hamiltonian graph, where V is a set of vertices, E is a set of edges and ω represents weight of edges. If G is undirected, the TSP is called symmetric TSP (sTSP). If G is directed, the TSP is called asymmetric TSP (aTSP). If G is undirected, the TSP is called symmetric TSP (sTSP). If G is directed, the TSP is called asymmetric TSP (aTSP).

0.2 Approach

There are two main ways to address the question posed by the travelling salesman problem, Exacts algorithm and Heuristic algorithms. Below are some popular algorithms:

Exacts algorithm:

- Brute Force
- Nearest Neighbor
- Brach & Bound
- Held-karp

Heuristic algorithm:

- Ant Colony Optimization
- Particle Swarm Optimization
- Lin-Kernighan
- African Buffalo Optimization

In this case we will use Held-Karp algorithm and Nearest Neighbor algorithm to solve the asymmetric TSP. Since Help-Karp algorithm only suitable for small number of cites (15-20 cities). For larger number of cities, we will use Nearest Neighbor algorithm.

0.3 Bitmasking

In computer programming, the process of modifying and utilizing binary representations of numbers or any other data is known as bitmasking.

A binary digit is used as a flag in bitmasking to denote the status or existence of a feature or trait. To accomplish this, certain bits within a binary number are set or reset to reflect a particular state or value.

Some common bitwise operations:

- OR (|) - sets a bit to 1 if either of the corresponding bits in the operands is 1.
- AND (&) - sets a bit to 1 if both the corresponding bits in the operands are 1.
- XOR (ˆ) - sets a bit to 1 if the corresponding bits in the operands are different.
- NOT (˜) - flips the bits in the operand, i.e., sets 0 bits to 1 and 1 bits to 0.
- Left Shift («) - shift every bits to the left.
- Right Shift (») - shift every bits to the right.

0.4 Held-Karp Algorithm

Held-Karp Algorithm is a dynamic programming approach, which breaks down the problem into smaller subproblems and use already computed results of smaller subtasks to solve larger subtasks. The idea is to use a 2d array to store minimum cost of visiting a set of cities. The set of cities can be represented as a bitmask, which is a binary number where each bit represents a city. The algorithms can be divided into 5 essential steps:

1. **Initialization:** Set up a dynamic programming table to store the optimal distances between subproblems.
2. **Base Case:** Set the initial condition for the smallest subproblems (with only two cities).
3. **Recursive Step:** Iterate through each subproblem, calculating the optimal distances by considering all possible intermediate cities.
4. **Memoization:** Store the computed results in the dynamic programming table to avoid recomputation.
5. **Backtracking:** Construct the optimal tour by tracing back through the dynamic programming table.

The Held–Karp algorithm has exponential time complexity $\Theta(2^n n^2)$, significantly better than the superexponential performance $\Theta(n!)$ of a brute-force algorithm. Held–Karp, however, requires $\Theta(n2^n)$ space to hold all computed values of the function $g(S,e)$, while brute force needs only $\Theta(n^2)$ space to store the graph itself.

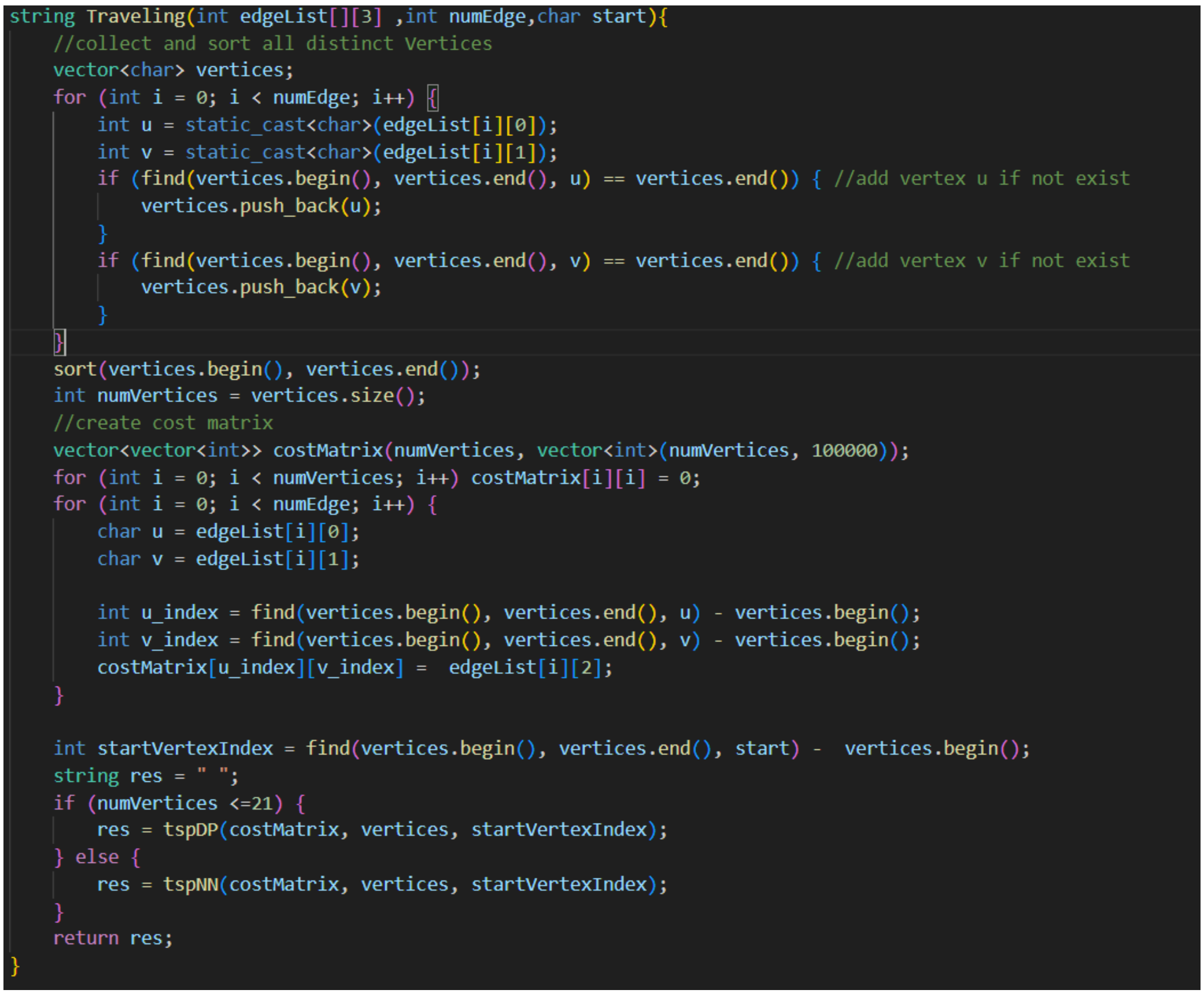
0.5 Nearest Neighbor Algorithm

Nearest Neighbor Algorithm is a greedy algorithm that starts at a random city and repeatedly visits the nearest city until all have been visited. The algorithm quickly yields a short tour, but usually not the optimal one. These are 5 steps of the algorithms:

1. Initialize all vertices as unvisited.
2. Select an arbitrary vertex, set it as the current vertex **u**. Mark **u** as visited.
3. Find out the shortest edge connecting the current vertex **u** and an unvisited vertex **v**.
4. Set v as the current vertex **u**. Mark **v** as visited.
5. If all the vertices in the domain are visited, then terminate. Else, go to step 3.

   The sequence of the visited vertices is the output of the algorithm. The Nearest Neighbor Algorithm has time complexity $\Theta(n^2)$ and space complexity $\Theta(n)$.

0.6 Implement

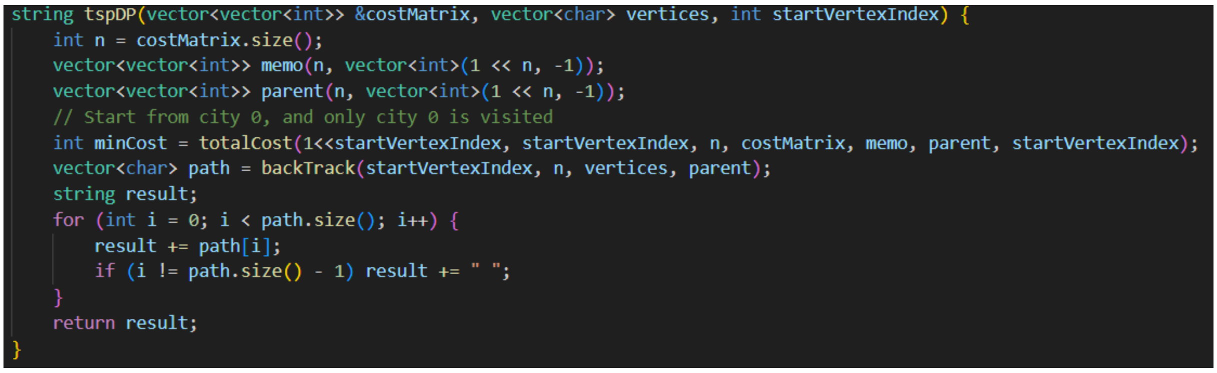


```
string Traveling(int edgeList[][3] ,int numEdge,char start){
    //collect and sort all distinct Vertices
    vector<char> vertices;
    for (int i = 0; i < numEdge; i++) {
        int u = static_cast<char>(edgeList[i][0]);
        int v = static_cast<char>(edgeList[i][1]);
        if (find(vertices.begin(), vertices.end(), u) == vertices.end()) { //add vertex u if not exist
            vertices.push_back(u);
        }
        if (find(vertices.begin(), vertices.end(), v) == vertices.end()) { //add vertex v if not exist
            vertices.push_back(v);
        }
    }
    sort(vertices.begin(), vertices.end());
    int numVertices = vertices.size();
    //create cost matrix
    vector<vector<int>> costMatrix(numVertices, vector<int>(numVertices, 100000));
    for (int i = 0; i < numVertices; i++) costMatrix[i][i] = 0;
    for (int i = 0; i < numEdge; i++) {
        char u = edgeList[i][0];
        char v = edgeList[i][1];

        int u_index = find(vertices.begin(), vertices.end(), u) - vertices.begin();
        int v_index = find(vertices.begin(), vertices.end(), v) - vertices.begin();
        costMatrix[u_index][v_index] =  edgeList[i][2];
    }

    int startVertexIndex = find(vertices.begin(), vertices.end(), start) -  vertices.begin();
    string res = " ";
    if (numVertices <=21) {
        res = tspDP(costMatrix, vertices, startVertexIndex);
    } else {
        res = tspNN(costMatrix, vertices, startVertexIndex);
    }
    return res;
}
```

string Traveling(int edgeList[][3], int numEdge, char start)

- The purpose of this function is to collect all unique vertices into a vector and create a cost matrix, which will be used as a parameter for $tspDP()$ and $tspDP()$. The code determine what algorithm to use based on complexity (number of cities).

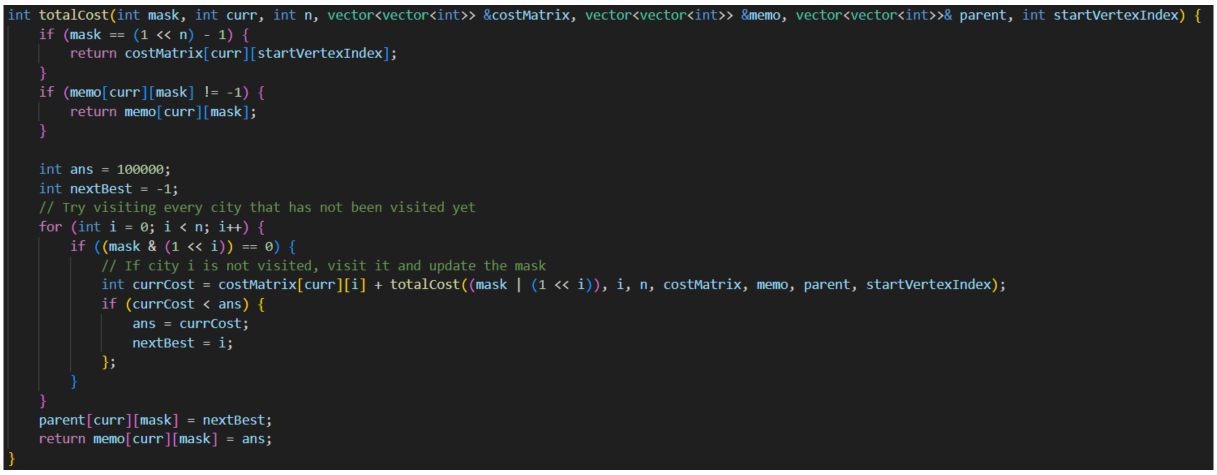
0.6.1 Held-Karp Algorithms



```cpp
string tspDP(vector<vector<int>> &costMatrix, vector<char> vertices, int startVertexIndex) {
    int n = costMatrix.size();
    vector<vector<int>> memo(n, vector<int>(1 << n, -1));
    vector<vector<int>> parent(n, vector<int>(1 << n, -1));
    // Start from city 0, and only city 0 is visited
    int minCost = totalCost(1<<startVertexIndex, startVertexIndex, n, costMatrix, memo, parent, startVertexIndex);
    vector<char> path = backTrack(startVertexIndex, n, vertices, parent);
    string result;
    for (int i = 0; i < path.size(); i++) {
        result += path[i];
        if (i != path.size() - 1) result += " ";
    }
    return result;
}
```

string tspDP(vector<vector<int» &costMatrix, vector<char> vertices, int startVertexIndex)

- n is the numbers of vertices.
- Initialize 2d vector $memo[n][1 << n]$ with initial value of -1. This will stores the minimum cost of visiting all vertices represented by a bitmask and ending at vertex i.
- Initialize 2d vector $parent[n][1 << n]$ with initial value of -1. This will stores the most optimal path, which will later be used for backtracking.
- The purpose of this function is to called $totalCost()$, which is the Held-Karp Algorithm's implementation and $backTrack()$, which will backtrack the most optimal path and return a vector of character. The vector is stored in variable $path$ to be concatenated into a string.



```cpp
int totalCost(int mask, int curr, int n, vector<vector<int>> &costMatrix, vector<vector<int>> &memo, vector<vector<int>>& parent, int startVertexIndex) {
    if (mask == (1 << n) - 1) {
        return costMatrix[curr][startVertexIndex];
    }
    if (memo[curr][mask] != -1) {
        return memo[curr][mask];
    }

    int ans = 100000;
    int nextBest = -1;
    // Try visiting every city that has not been visited yet
    for (int i = 0; i < n; i++) {
        if ((mask & (1 << i)) == 0) {
            // If city i is not visited, visit it and update the mask
            int currCost = costMatrix[curr][i] + totalCost((mask | (1 << i)), i, n, costMatrix, memo, parent, startVertexIndex);
            if (currCost < ans) {
                ans = currCost;
                nextBest = i;
            };
        }
    }
    parent[curr][mask] = nextBest;
    return memo[curr][mask] = ans;
}
```

int totalCost(int mask, int curr, int n, vector<vector<int» &costMatrix, vector<vector<int» &memo, vector<vector<int»& parent, int startVertexIndex)

- Base case: if all cities are visited, return the cost to return to the starting city.
- The next condition will return the stored value if already computed the result.
- Initialize $ans := \infty$ and $nextBest := -1$.
- Loop through every vertices i and checked if that vertex is unvisited (mask & (1 « i) == 0):

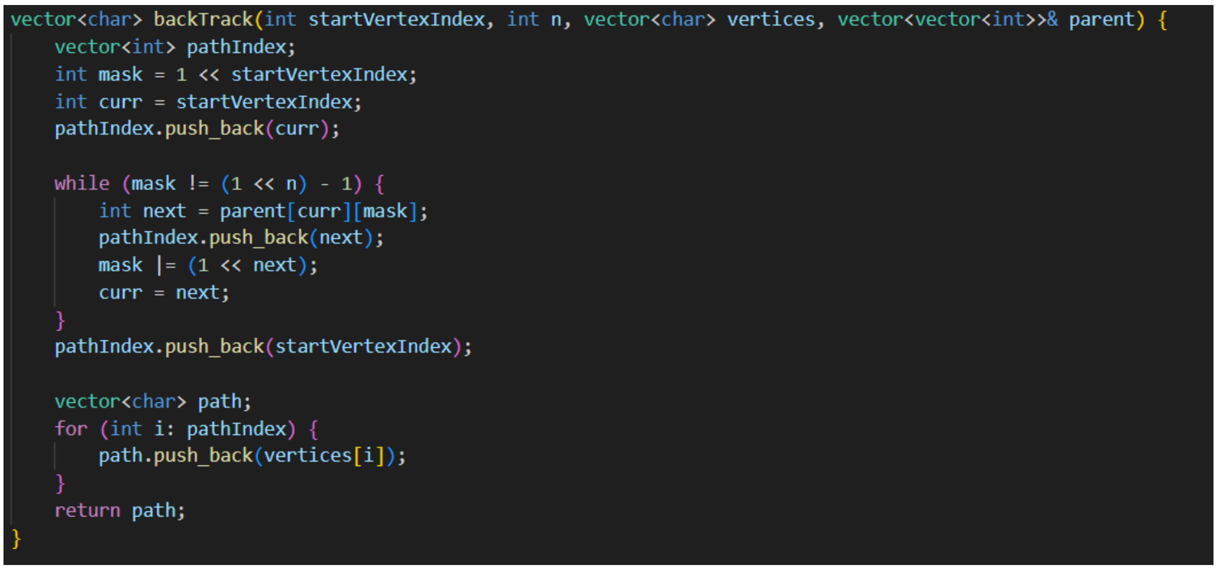
+ If that vertex is unvisted, calculate the cost from $curr \to i$ by recursively computed the best path starting from i + current path's cost (currCost = costMatrix[curr][i] + totalCost((mask | (1 « i)), i, n, costMatrix, memo, parent)).

+ If the new path is better than *ans*. update *ans* to current best value ($ans := currCost$) and *nexBest* to next best vertex ($nextBest := i$).

- Store the next best vertex in parent[curr][mask] ($parent[curr][mask] = nextBest$) which will later be used to trace path.

- Store the best cost in memo[curr][mask] ($memo[curr][mask] = ans$) for future calculation.

- Return best cost for this state.



```
vector<char> backTrack(int startVertexIndex, int n, vector<char> vertices, vector<vector<int>>& parent) {
    vector<int> pathIndex;
    int mask = 1 << startVertexIndex;
    int curr = startVertexIndex;
    pathIndex.push_back(curr);

    while (mask != (1 << n) - 1) {
        int next = parent[curr][mask];
        pathIndex.push_back(next);
        mask |= (1 << next);
        curr = next;
    }
    pathIndex.push_back(startVertexIndex);

    vector<char> path;
    for (int i: pathIndex) {
        path.push_back(vertices[i]);
    }
    return path;
}
```

vector<char> backTrack(int startVertexIndex, int n, vector<char> vertices, vector<vector<int»& parent)

- Initialize vector pathIndex ($vector < int > pathIndex$) which store the vertex's index in the optimal path order.

-Initialize mask contain the first vertex ($int\ mask = 1 << startVertexIndex$) and its current index ($int\ curr = startVertexIndex$)

- Insert first vertex's index into $pathIndex$ ($pathIndex.push\_back(curr)$)

- Iterate until traverse through every vertices:

+ Get next vertex from $parent[curr][mask]$, which we already have from previous function ($int\ next = parent[curr][mask]$).

+ Insert this vertex into $pathIndex$ ($pathIndex.push\_back(next)$).

+ Update current mask to mark visited vertices ($mask \mid = (1 << next)$).

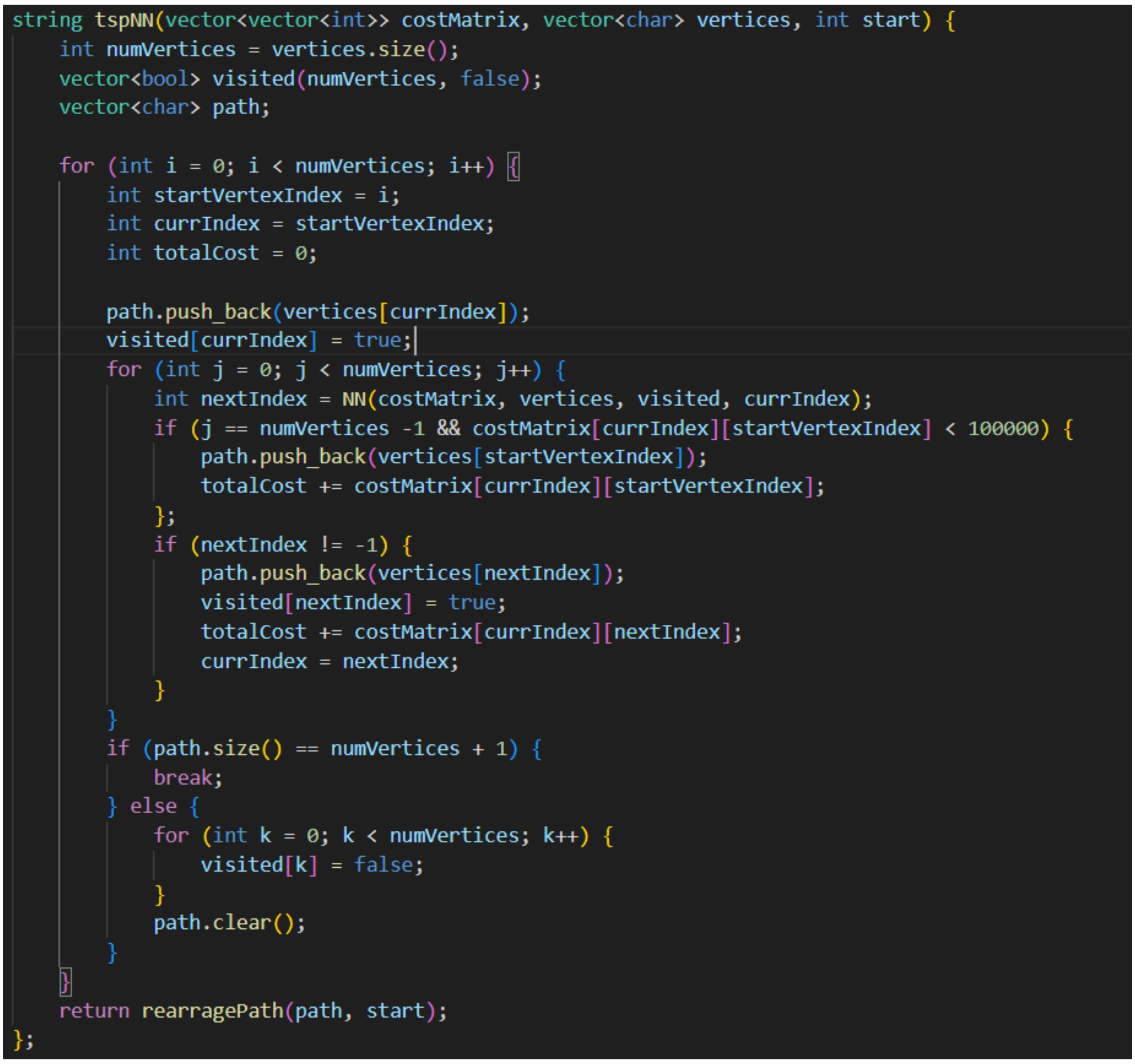
+ Update next vertex as current vertex ($curr = next$).

- Insert first vertex's index into $pathIndex$ ($pathIndex.push\_back(curr)$) as the last element to complete the path.

- Convert all the indices to ASCII character by mapping indices to $vector < char > vertices$

- Return the path.

0.6.2 Nearest Neighbor Algorithm



```
string tspNN(vector<vector<int>> costMatrix, vector<char> vertices, int start) {
    int numVertices = vertices.size();
    vector<bool> visited(numVertices, false);
    vector<char> path;

    for (int i = 0; i < numVertices; i++) {
        int startVertexIndex = i;
        int currIndex = startVertexIndex;
        int totalCost = 0;

        path.push_back(vertices[currIndex]);
        visited[currIndex] = true;
        for (int j = 0; j < numVertices; j++) {
            int nextIndex = NN(costMatrix, vertices, visited, currIndex);
            if (j == numVertices -1 && costMatrix[currIndex][startVertexIndex] < 100000) {
                path.push_back(vertices[startVertexIndex]);
                totalCost += costMatrix[currIndex][startVertexIndex];
            };
            if (nextIndex != -1) {
                path.push_back(vertices[nextIndex]);
                visited[nextIndex] = true;
                totalCost += costMatrix[currIndex][nextIndex];
                currIndex = nextIndex;
            }
        }
        if (path.size() == numVertices + 1) {
            break;
        } else {
            for (int k = 0; k < numVertices; k++) {
                visited[k] = false;
            }
            path.clear();
        }
    }
    return rearragePath(path, start);
};
```

string tspNN(vector<vector<int» costMatrix, vector<char> vertices, int start)

- Initialize vector visited ($vector < bool > visited(numVertices, false)$) which store visited cities.

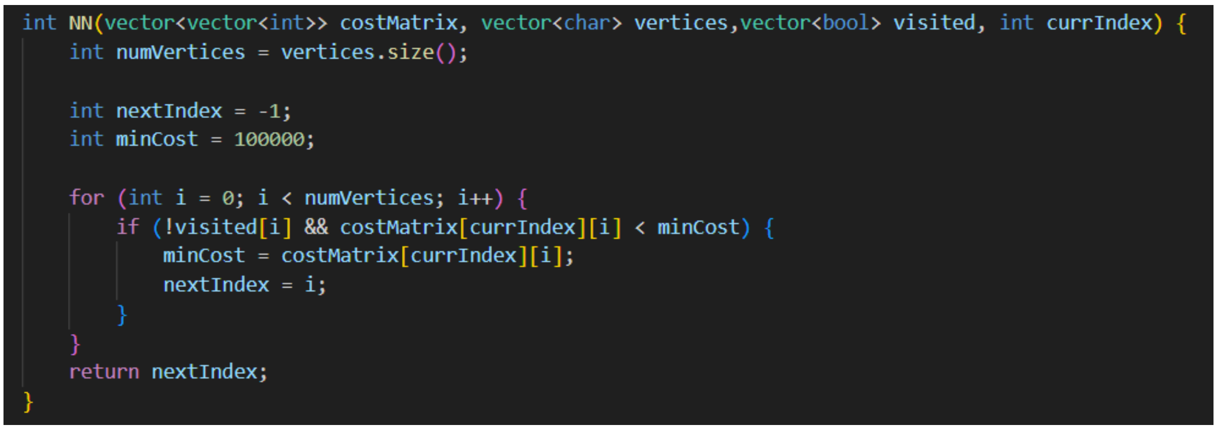
- Initialize vector path ($vector < char > path$) which store the final path.

- Loop through every cities and use each city as starting city to find the path (NN may not go through all the cities because of incomplete graph)

+ If found a valid path, escape the loop and keep the path.

+ If no valid path exist, delete the path.

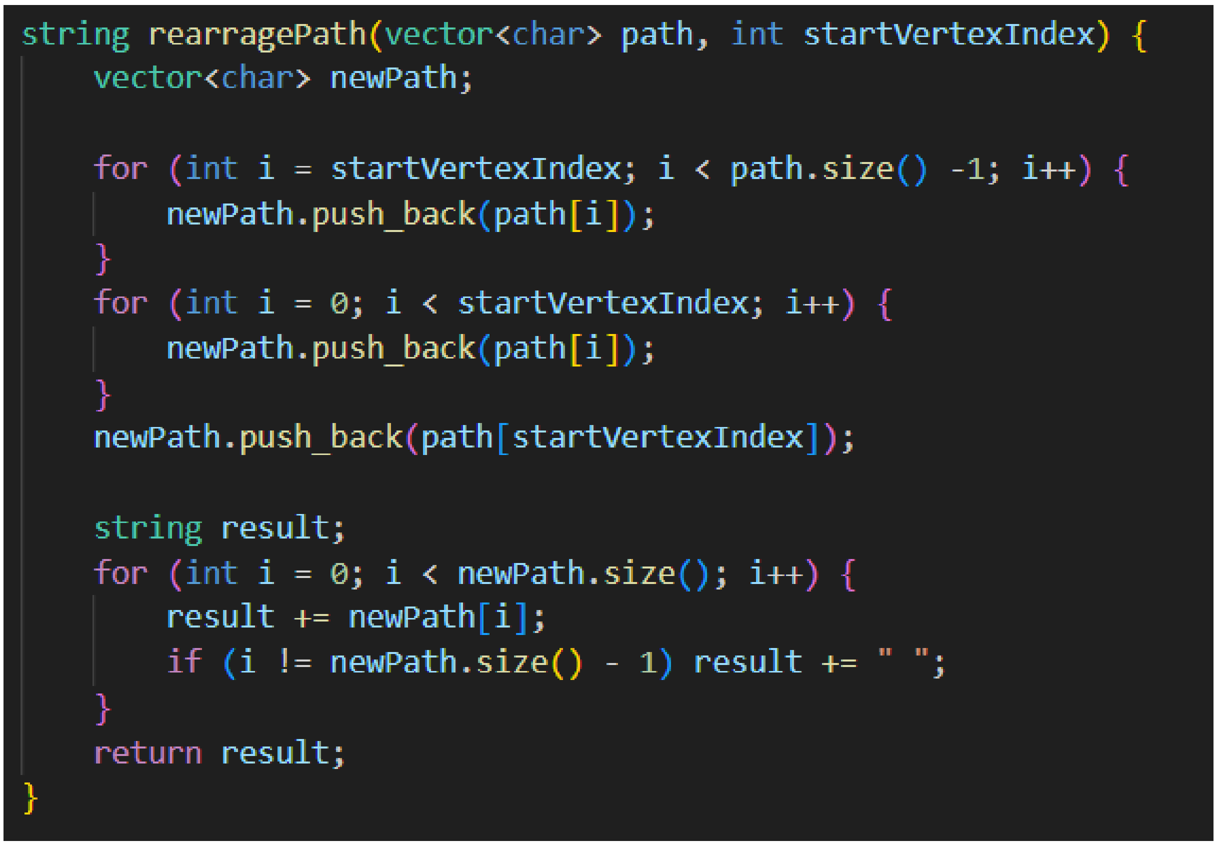
- Return the rearraged path.



```
int NN(vector<vector<int>> costMatrix, vector<char> vertices,vector<bool> visited, int currIndex) {
    int numVertices = vertices.size();

    int nextIndex = -1;
    int minCost = 100000;

    for (int i = 0; i < numVertices; i++) {
        if (!visited[i] && costMatrix[currIndex][i] < minCost) {
            minCost = costMatrix[currIndex][i];
            nextIndex = i;
        }
    }
    return nextIndex;
}
```

int NN(vector<vector<int» costMatrix, vector<char> vertices,vector<bool> visited, int currIndex)

- This function return the index of the nearest unvisited city.



```
string rearragePath(vector<char> path, int startVertexIndex) {
    vector<char> newPath;

    for (int i = startVertexIndex; i < path.size() -1; i++) {
        newPath.push_back(path[i]);
    }
    for (int i = 0; i < startVertexIndex; i++) {
        newPath.push_back(path[i]);
    }
    newPath.push_back(path[startVertexIndex]);

    string result;
    for (int i = 0; i < newPath.size(); i++) {
        result += newPath[i];
        if (i != newPath.size() - 1) result += " ";
    }
    return result;
}
```

string rearragePath(vector<char> path, int startVertexIndex)

- This function take the path and change its starting city (since the path is a Hamiltonian cycle, the path can start at anywhere and keep the same cost of the path).

0.7 References

1. https://www.geeksforgeeks.org/what-is-bitmasking/
2. https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm
3. https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
4. https://www.math.nagoya-u.ac.jp/~richard/teaching/s2020/Quang1.pdf