Graphs - Traveling Salesman Problem | Nearest Neighbor Heuristic

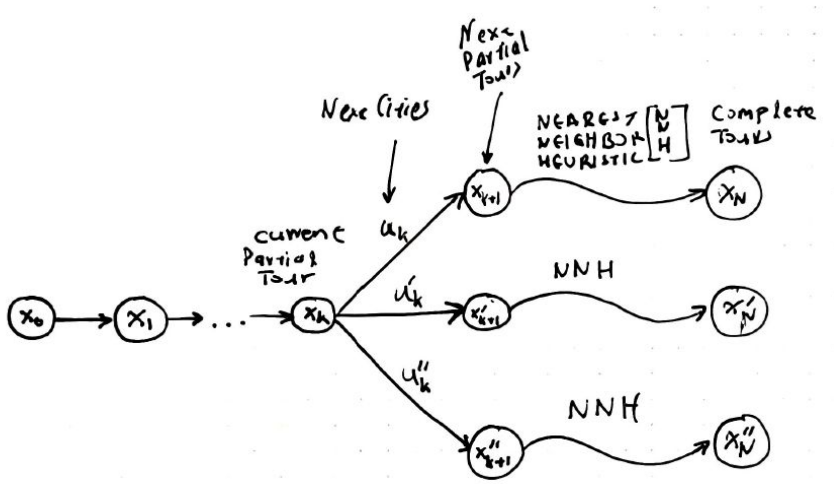
Traveling Salesman Problem: Let us consider a traveling salesman that wants to visit each of N given cities c = 0,...,N-1 exactly once, return back to the city he started from and perform this tour with a minimum cost. For each pair of distinct cities c and c' there is an associated traversal cost g(c,c') which in turn corresponds to the weight of the edge of the respective graph representation. Furthermore, it is assumed that the graph is complete - that is that we can go directly from every city to every other city. The TSP problem is to find a tour, a visit order, that goes through every city only once and the sum of the associated edges cost is minimum.

Nearest Neighbor Heuristic: There are many solutions to the TSP problem and in fact - as discussed - this is a known hard problem of mathematics. An imperfect, approximate, but fairly simple solution is the so-called "Nearest Neighbor" Heuristic (NNH).

NNH starts from a partial tour (i.e., an ordered collection of distinct cities) and proceeds to construct a sequence of partial tours, adding to each partial tour a new city that a) does not close a cycle in the graph, and b) minimizes the cost of the new extended tour. More specifically, given a sequence $\{c_0,c_1,...,c_k\}$ of distinct cities, NNH adds a city c_{k+1} that minimizes the associated cost $g(c_k,c_{k+1})$ over all cities other than those already visited. This in turn forms the sequence $\{c_0,c_1,...,c_k,c_{k+1}\}$. Performing this process incrementally, NNH eventually identifies a sequence of N cities $\{c_0,c_1,...,C_{N-1}\}$ thus resulting in complete tour with total cost:

$$g(c_0,c_1) + \ldots + g(c_{N-2},c_{N-1}) + g\ (C_{N-1},c_0)$$

The associated cost is not the optimal minimum but the other constraints of the TSP problem (e.g., no cycle) are satisfied. It is thus a valid solution. An associated visualization of the incremental methodology that NNH is approximating a solution of TSP is shown below.



Task: You are asked to implement a C++ implementation of NNH which considers an undirected and fully connected graph (e.g., with 4 vertices and edge weights you will specify) and returns the cost of the Nearest Neighbor Heuristic solution.

References:

- Bertsekas, D.P., 2019. Reinforcement learning and optimal control. Athena Scientific.
- Flood, M.M., 1956. The traveling-salesman problem. Operations research, 4(1), pp.61-75.
- https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/ (previously discussed approach using evaluation of permutations)
- http://www.martinbroadhurst.com/nearest-neighbour-algorithm-for-tsp-in-c.html

Indicative Solution

```c
#include <stdlib.h>
typedef struct {
    unsigned int first;
    unsigned int second;
    unsigned int weight;
} weighted_edge;
/* Check if the tour already contains an edge */
static unsigned int tour_contains(const weighted_edge *tour, unsigned int
t,
        const weighted_edge *edge)
{
    unsigned int contains = 0;
    unsigned int i;
    for (i = 0; i < t && !contains; i++) {
        contains = tour[i].first == edge->first
            && tour[i].second == edge->second;
    }
    return contains;
}
/* Find the edge to v's nearest neighbour not in the tour already */
static unsigned int nearest_neighbour_edge(const weighted_edge *edges,
unsigned int size,
        const weighted_edge *tour, unsigned int t, unsigned int v)
{
    unsigned int min_distance = 0;
    unsigned int nearest_neighbour;
    unsigned int i;
    for (i = 0; i < size; i++) {
        if ((edges[i].first == v || edges[i].second == v)
                && (min_distance == 0 || edges[i].weight < min_distance)
                && !tour_contains(tour, t, &edges[i]))
        {
            min_distance = edges[i].weight;
            nearest_neighbour = i;
        }
```

```
    }
    return nearest_neighbour;
}
weighted_edge *nearest_neighbour_tsp(const weighted_edge *edges, unsigned
int size,
        unsigned int order)
{
    unsigned int t, v = 0;
    weighted_edge *tour = malloc(order * sizeof(weighted_edge));
    if (tour == NULL) {
        return NULL;
    }
    for (t = 0; t < order; t++) {
        unsigned int e = nearest_neighbour_edge(edges, size, tour, t, v);
        tour[t] = edges[e];
        v = edges[e].first == v ? edges[e].second : edges[e].first;
    }
    return tour;
}



#include <stdio.h>
#include <stdlib.h>
/* Connect two edges */
void  weighted_edge_connect(weighted_edge  *edges,  unsigned  int  first,
unsigned int second,
        unsigned int weight, unsigned int *pos)
{
    edges[*pos].first = first;
    edges[*pos].second = second;
    edges[*pos].weight = weight;
    (*pos)++;
}
void print_edges(const weighted_edge *edges, unsigned int n)
{
    unsigned int e;
```

```
    for (e = 0; e < n; e++) {
            printf("(%u,  %u,  %u)  ",  edges[e].first,  edges[e].second,
edges[e].weight);
    }
    putchar('\n');
}
int main(void)
{
    unsigned int i = 0;
    const unsigned int size = 15; /* Edges */
    const unsigned int order = 6; /* Vertices */
    weighted_edge *edges = malloc(size * sizeof(weighted_edge));
    weighted_edge *tour;
    weighted_edge_connect(edges, 0, 1, 25, &i);
    weighted_edge_connect(edges, 0, 2, 19, &i);
    weighted_edge_connect(edges, 0, 3, 19, &i);
    weighted_edge_connect(edges, 0, 4, 16, &i);
    weighted_edge_connect(edges, 0, 5, 28, &i);
    weighted_edge_connect(edges, 1, 2, 24, &i);
    weighted_edge_connect(edges, 1, 3, 30, &i);
    weighted_edge_connect(edges, 1, 4, 27, &i);
    weighted_edge_connect(edges, 1, 5, 17, &i);
    weighted_edge_connect(edges, 2, 3, 18, &i);
    weighted_edge_connect(edges, 2, 4, 20, &i);
    weighted_edge_connect(edges, 2, 5, 23, &i);
    weighted_edge_connect(edges, 3, 4, 19, &i);
    weighted_edge_connect(edges, 3, 5, 32, &i);
    weighted_edge_connect(edges, 4, 5, 41, &i);
    tour = nearest_neighbour_tsp(edges, size, order);
    print_edges(tour, order);

    free(tour);
    free(edges);
    return 0;
}
```