DD2440 TSP Report
Submission id: 9928195
Score: 41.207387

[redacted] Isak Nyberg, [redacted], [redacted]

November 2022

1 Introduction

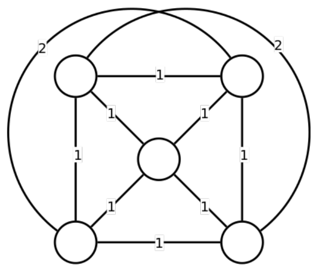
Travelling salesperson problem is a famous NP-hard problem. Basically, it can be described as given a list of cities and the distances between each city, how to find a minimum cycle that visits each city exactly once and goes back to the starting city.

In this project, we applied Christofides algorithm to find an initial tour. After that, optimizations like 2-opt and 3-opt are adopted to increase the performance. Finally, we can achieve a 41.2 on Kattis.

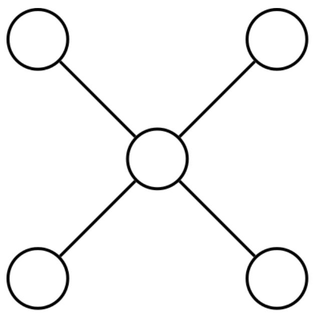
2 Global Optimization (score: 5)

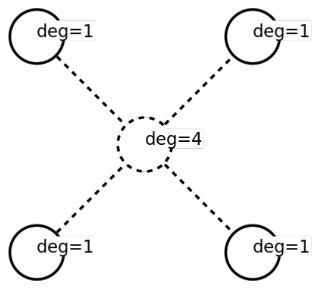
In [1], four heuristics are put forward to solve TSP problem, which are Nearest neighbor, Greedy, Clarke-Wright and Christofides respectively. Christofides is an algorithm within 1.5-approximation in polynomial time while others have much worse performances. Christofides has running time $O(n^3)$ and it is not too bad compared with other heuristics. Therefore, we choose Christofides as our implementation.

Basically, Christofides can be divided into the following parts [2]. First, it finds the minimum spanning tree T of the graph G. Second, it finds the vertices that have odd degree in the minimum spanning tree (MST). Third, it finds a perfect matching M of these odd vertices. Fourth, it combines T and M together to form a new graph G' where each vertex has even degree. Fifth, it finds an Eulerian path in G'. Sixth, it finds a Hamiltonian path, which is the result, from the Eulerian path. The following figure (Figure 1) illustrates the whole process.

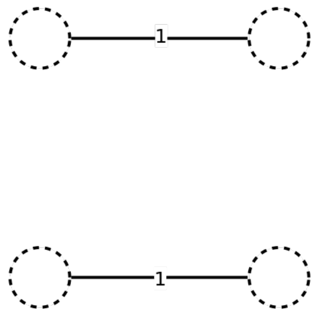


(a) Original Graph

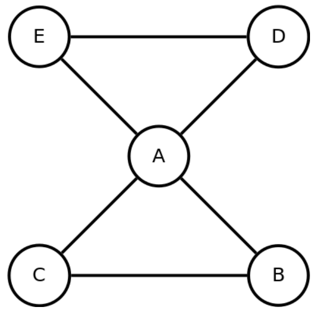

(b) MST



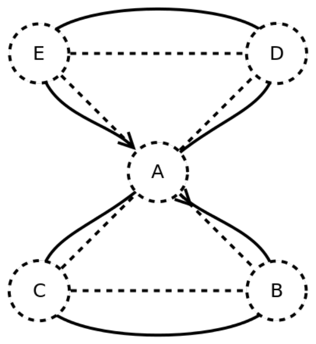
(c) Odd Vertices



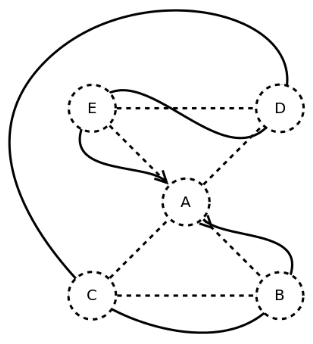
(d) Perfect Matching



(e) Each Vertex with Even Degree



(f) Eulerian Path



(g) Hamiltonian Path

Figure 1: Christofides Algorithm [3]

2.1 Minimum Spanning Tree

There are two famous algorithms for constructing a minimum spanning tree: Kruskal's algorithm and Prim's algorithm [4]. Both of them take a greedy approach. Kruskal's algorithm adds edges one by one with least weight in each iteration till a spanning tree is formed. In dense graphs, the number of edges is very high, and Kruskal's algorithm may not perform well. Therefore, we choose to use Prim's algorithm.

We use n to denote the number of vertices in this graph. We use three arrays (used, parent and key) to maintain the information needed. "used" is initialized with 0 as every vetex is not used now. "parent" is initialized with 0 as the parent of each vertex is unknown. But parent[0] $= -1$ as the first vertex has no parent. "key" is initialized with MAX as we don't know the minimum weight between MST and each vertex. But key[0] $= 0$ as the first vertex will be added to the MST.

Then we find a not used vertex (idx) that is closest to the MST and add it to the MST. For each not used vertex (j) that has connection with idx, we judge whether it is colser to the MST after adding idx to the MST. If yes, we update the parent of j (parent[j]) and the minimum weight (key[j]) between j and

MST. We repeat this process again and again until all the vertices are in the MST. With the parent of each vertex, we can easily construct a MST.

Algorithm 1 Prim's algorithm

```
1: $used \leftarrow [0, ..., 0]$ // array of size $n$ to store whether one vertex is in MST or not
2: $parent \leftarrow [-1, 0, ..., 0]$ // array of size $n$ to store the parent of this vertex
3: $key \leftarrow [0, MAX, ..., MAX]$ // array of size $n$ to store minimum weight between this vertex and MST
4: for $i = 0$ to $n - 2$ do
5:   idx ← a neareset neighbor of MST that has not been used
6:   $used$[idx] ← 1 // mark it as used
7:   for $j = 0$ to $n - 1$ do
8:     if $j$ is not idx and $j$ is not used and the distance between idx and $j$ is smaller than $key[j]$ then
9:       $parent[j]$ = idx // update the parent of $j$
10:      $key[j]$ = the distance between idx and $j$ // update the minimum weight between $j$ and MST
11:    end if
12:  end for
13: end for
```

2.2 Odd Vertices

With the parent of each vertex, we can construct a data structure "vector<vector<int>> *neighbors*" in C++ to store the adjacent vertices to the current vertex i in the MST, e.g., *neighbors*[i] contains the adjacent vertices to i. Then, we can go through each vertex i to get the size of *neighbors*[i]. If it is odd, then vertex i is an odd vertex.

2.3 Perfect Matching

For perfect matching, we also take a greedy approach. We use n to denote the number of odd vertices (n is always even according to [2]). Starting from the first odd vertex x_1, we go through the other not used odd vertices to find the closest one, e.g. vertex x_2. Then standing at x_2, we find the next not used closest vertex x_3. If we do this until all the vertices are used, we can get an array [x_1, x_2, ..., x_n]. Finally, we can make pairs like (x_1, x_2), (x_3, x_4), ..., (x_{n-1}, x_n).

Algorithm 2 Perfect matching

```
1: $odd$ // array of size $n$ for storing the odd vertices
2: $used \leftarrow [1, 0, ..., 0]$ // array of size $n$ to store whether this odd vertex is used or not. We start from the
   first one, so it is marked as used.
3: $tour$ ← [odd[0],0,...,0] // array of size $n$ to store the sequence, the first vertex will be placed in it before
   starting.
4: for $i = 1$ to $n - 1$ do
5:   temp ← $-1$ // vaule for odd vertex
6:   idx ← $-1$ // vaule for the index of the odd vertex in array $odd$
7:   for $j = 0$ to $n - 1$ do
8:     if $used[j] == 0$ and (temp = $-1$ or distance between $odd[j]$ and $tour[i - 1]$ < distance between
       temp and $tour[i - 1]$) then
9:       temp ← $odd[j]$ // update the temp by currently nearest vertex
10:      idx ← $j$ // update the idx by the index of currently nearest vertex in array $odd$
11:    end if
12:  end for
13:  $tour$[i] ← $odd[j]$ // $odd[j]$ is placed into array $tour$
14:  $used$[j] ← 1 // $j$ becomes used now
15: end for
16: return constructed pairs based on $tour$
```

2.4 Creating a New Graph

With the pairs from section 2.3, one can update the *neighbors* in section 2.2 by just pushing new adjacent vertices back to *neighbors* accordingly.

2.5 Eulerian Path

After creating the new graph, each vertex should have even degree by now, so an Eulerian path should exist. In an undirected graph, Eulerian is a walk that uses each edge exactly once [5]. Therefore, a depth-first search (DFS) algorithm will be of help when finding such a path.

First, "map<pair<int, int>, int> edge" is created to store the number of each edge. Then we start from one vertex and do DFS to find the Eulerian path. For each DFS iteration, if the number of edge is > 0, we can continue from the other vertex of the edge and decrease the corresponding edge number. We do this until all the edges have number 0.

Algorithm 3 DFS

```
1: $edge \leftarrow$ a map that stores the number of each edge // Note: ($i$, $j$) and ($j$, $i$) are treated as different
   edges as **pair** is order-sensitive.
2: $neighbors \leftarrow$ the updated neighbors in section 2.4
Input: $x$ // a vertex to start the DFS
Output: $ans$ // an array that returns the order of visiting
3: for each $i$ in $neighbors[x]$ do
4:    if number of edge($i$, $x$) is $> 0$ then
5:       decrease the number of edge($i$, $x$) and edge($x$, $i$)
6:       DFS($i$) // continue DFS starting from $i$
7:    end if
8: end for
9: $ans$.push_back($x$)
```

2.6 Hamiltonian Path

Eulerian path may visit one vertex several times as it covers all edges. In this part, we generally remove this repition so that each vertex will be visited only once. And we can start from the back of Eulerian path as "vector<int>" in C++ supports "pop_back()". We also maintain an array that stores whether it is visited for each vertex.

We just go through the Eulerian path from back to front. For each vertex, if it is not visited before, we put in *ans* until we have exhausted Eulerian path.

Algorithm 4 Hamiltonian

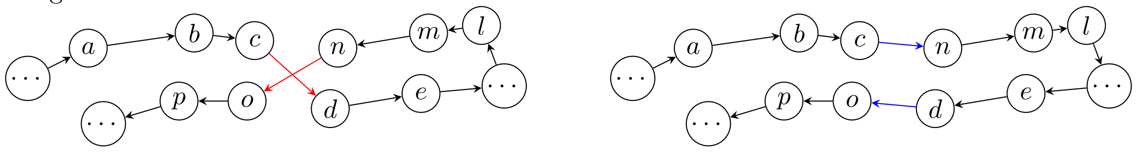
```
Input: $eulerPath$
Output: $ans$: Hamiltonian path
1: $used \leftarrow [0, ..., 0]$ // all vertices are currently unvisited
2: while eulerPath is not empty do
3:    cur ← back of eulerPath
4:    if cur not visited then
5:       $ans$.push_back(cur)
6:       mark cur as visited
7:    end if
8:    eulerPath.pop_back()
9: end while
```

.

3 Local Optimization

3.1 2-OPT (Score: 23)

The basic idea of two opt is that two edges in the path are removed to create two disjoint paths, the two paths are then reconstructed in a way such that the two new edges that are added will have the smallest total length.



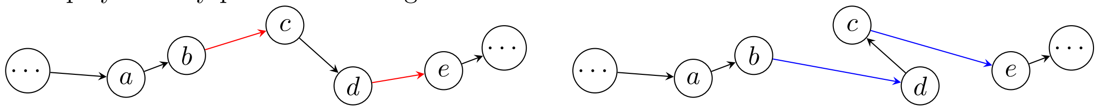
In the example above, the could would calculate the following lengths $(c, d), (c, n), (n, o), (d, o)$ and if $|(c, n)| + |(d, o)| < |(c, d)| + |(n, o)|$ it would mean that a unit of length would be saved if the alternative right hand path was used instead and thus the two paths could be replaced accordingly. It is also important to note that after these edges are changed, each of the edges $(n, m), (m, l), \ldots, (e, d)$ would need to be reversed in order to still keep the direction correct. To run this algorithm the following code was used: [6]

```
int opt2(short path[], short N) {
    int improvement = 0;
    for (short i=0; i < N-1; i++) {
        int a = DISTANCE(path[i], path[i+1]);                    // c->d

        for (int j=i+2; j < N-1; j++) {
            int b = DISTANCE(path[j], path[j+1]);                // n->o

            int c = DISTANCE(path[i], path[j]);                  // c->n
            int d = DISTANCE(path[i+1], path[j+1]);              // d->o

            int original_dist = a + b;                           // c->d + n->o
            int new_dist = c + d;                                // c->n + d->o

            if (original_dist > new_dist) {
                reverse(path, i+1, j, N);
                a = DISTANCE(path[i], path[i+1]);
                improvement += original_dist - new_dist;
            }
        }
    }
    return improvement;
}
```

From the two *for* loops it is evident that the code will run in $\mathcal{O}(n^2)$ time, which still is not great but since the number of vertices in the graph is limited to 1000 each run will at most take, 1 000 000 iterations. In this case the variable i would represent the vertex c and the variable j would be vertex d. Similarly to how we then calculate the distances in the example we find the distance with respect to $i + 1$ and $j + 1$ and reverse parts of the path accordingly. It is also important that the improvement is kept track of since it can easily be calculated and returned to keep track of the overall length of the path, instead of having to check for it linearly every time, this is important later. Some improvements were also made to start the variable i to skip some values depending on where we are aware of changes. This function would be run multiple times until the improvements would reach 0. In the end we reached a score of about 23 with this code.

3.2 Avoiding Local Minima (Score: 41)

After a few iterations of 2-OPT the length of the path would stop improving and the function would return 0. This would indicate that the path had reached a local minima and that 2-OPT was no longer sufficient to improve the path any further. Some experiments were done with the so called 3-OPT algorithm, however due to 3-OPT having a complexity of $\mathcal{O}(n^3)$ and in our experimentation did not yield enough improvements to justify the time taken we used a different approach. Instead of improving the graph any further we did random shuffling to the path, making the path longer temporarily with the hope that further usage of 2-OPT on this scrambled graph would give a better optimization than before. As trivial as this sounds it played a key part in obtaining a score above 40.



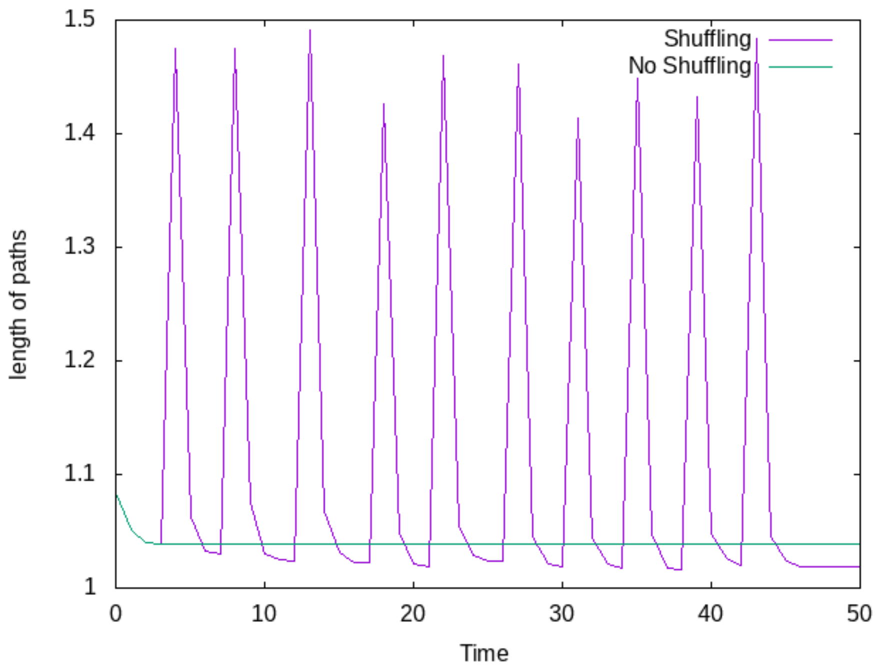
In the example above the edge from b to c and d to e are replaced with b to d and c to e. The edge c to d is also reversed. While this seems like it would be a lot of operations, in reality the only difference is that the position in the path of c and d are swapped. This can be seen in the following code.

```
int small_change(int path[], int N) {
    int path_dist_change = 0;
    short num_changes = int(N * 0.5);
    for (short i = 0; i < num_changes; i++) {
        short r = (rand() % (N-3)) + 1;
        int old_edge_length = DISTANCE(path[r-1], path[r]) + DISTANCE(path[r+1], path[r+2]);
        int new_edge_length = DISTANCE(path[r-1], path[r+1]) + DISTANCE(path[r], path[r+2]);
        path_dist_change += new_edge_length - old_edge_length;

        // switch places of r and r+1
        short temp = path[r];
        path[r] = path[r+1];
        path[r+1] = temp;
    }
    return path_dist_change;
}
```

This combined with the 2-OPT algorithm lead to the following main-loop, reserved for small optimizations.

```
// path is an array of cities
// path_dist is the current length of path in terms of distance
while (clock() < DEADLINE) {
    int improvement = opt2(path, N);
    path_dist -= improvement;

    if (improvement == 0) {
        int change = small_change(path, N);
        path_dist += change;
    }
}
```

Running the code, the impact of the random shuffle is quite substantial. In the real example there is a number of minor optimization in terms of the number of shuffles but the code examples provide the essence of what the code does.



The graph above shows the length of the current path at the end of each loop in the main-loop as a multiple the best path found at the end. The graph only shows the first 50 loops but in reality within the 2 seconds time limit the loop actually runs upwards of 300 times. The green line is the result of running 2-OPT until there is no further improvement, while the purple line shuffles the path slightly when 2-OPT returns 0 and then runs 2-OPT again. It is obvious that the path becomes a lot worse initially going up to 1.5 times the length of the best path found, however when it eventually plateaus again it sometimes finds an even shorter path than before due to the added randomization. This means we can successfully avoid getting stuck in local minima. At the end of every loop if a shorter path is found, it is saved and eventually the best path will be printed out when the time is up.

3.3 Conclusion

After a lot of experimentation with more complex ways of finding ways of reorganizing and try to find rearrangements to the path, the most impact was to simply ignore the complex algorithms in favours of the much faster 2-OPT combined with randomization.

References

[1] Johan Hastad, Notes for the Course Advances Algorithms, Jan. 2000.

[2] Nicos Christofides, Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem, Feb. 1976.

[3] Wikipedia, Christofides Algorithm, Accessed in Nov. 2022.

[4] Gurasis Singh, Algorithms: Minimum Spanning Tree, Accessed in Nov. 2022.

[5] Wikipedia, Eulerian Path, Accessed in Nov. 2022.

[6] Wikipedia, 2-opt, Accessed in Nov. 2022.