Travelling Salesman Problem

Project for

CPE 593

Applied Algorithms and Data Structures

May 2016

Participants:

Jinglong Wu

Juncheng Li

Xintong Yan

Content

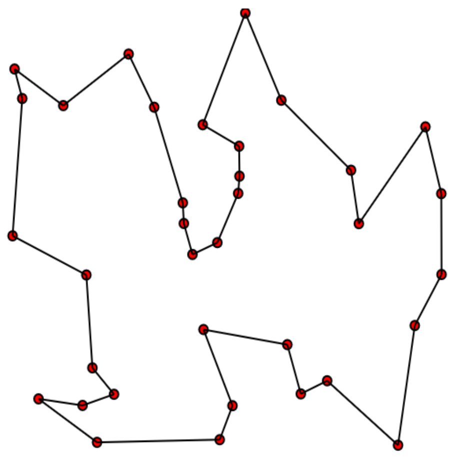
1. Introduction

The Travelling Salesman Problem (TSP) is a deceptively simple combinatorial problem. It can be stated very simply: A salesman spends his time visiting N cities (or nodes) cyclically. In one tour he visits each city just once, and finishes up where he started. In what order should he visit them to minimize the distance traveled? TSP is applied in many different places such as warehousing, material handling and facility planning.

The travelling salesman problem (TSP) can be stated as follows:

Given:

- Complete undirected graph G(V, E)
- Metric edge cost Ce >=0



Problem:

- Find a hamiltonian cycle with minimal cost.

Although optimal algorithms exist for solving the TSP, for large-size TSP, as following Brute Force and Dynamic Programming algorithms show that, it is almost impossible to generate an optimal solution within a reasonable amount of time. Heuristics, instead of optimal algorithms, are extensively used to solve such problems. People conceived many heuristic algorithms to get near-optimal solutions. Here we implement Greedy and Lin-Kernighan (2-opt) and analyze the result.

The algorithms can be implemented by using Visual C++. From these program the results are attained and the results of each algorithms can be compared.

2. Description of Algorithms

The algorithms for TSP can be classified as

1. Construction Algorithms;
2. Improvement Algorithms; and
3. Hybrid Algorithms.

Construction algorithms are those in which the tour is constructed by including points in the tours, usually one at a time, until a complete tour is developed. In improvement algorithms, a given initial solution is improved, if possible, by transposing two or more points in the initial tour. Hybrid algorithms, which are not discussed in our project, use a construction algorithms to obtain an initial solution and then improve it using an improvement algorithm. For example, simulated annealing and neural network fall into this category.

In next section, we will talk about what algorithm we implemented.

2.A Brute Force

When one thinks of solving TSP, the first method that might come to mind is a brute-force method. The brute-force method is to simply generate all possible tours and compute their distances. The shortest tour is thus the optimal tour. To solve TSP using Brute-force method we can use the following steps:

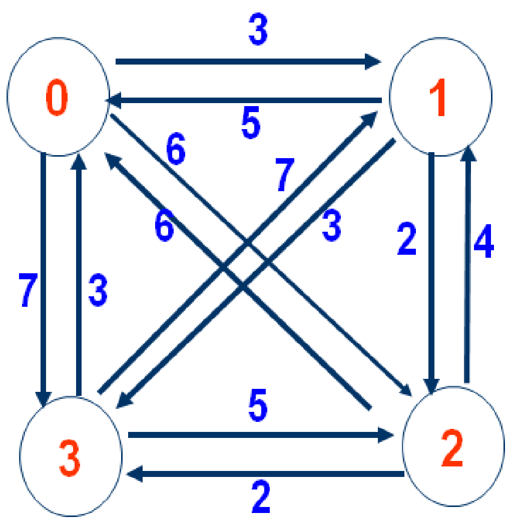
Step 1. calculate the total number of tours.
Step 2. draw and list all the possible tours.
Step 3. calculate the distance of each tour.
Step 4. choose the shortest tour, this is the optimal solution.

For the C++ code for Brute Force, please refer to the GitHub(https://github.com/jli98/Travelling-Salesman-Problem/blob/master/TSP-BruteForce.cpp).

2.B Dynamic Programming

Assume that there is a TSP problem as follow.

There are 4 cities, 0 1 2 3, shown as picture. The adjacency matrix represents the distance between cities.



$$C = \begin{pmatrix} 0 & 3 & 6 & 7 \\ 5 & 0 & 2 & 3 \\ 6 & 4 & 0 & 2 \\ 3 & 7 & 5 & 0 \end{pmatrix}$$

1. Why DP can works?

Assume that {s, s1 ,s2 , … , sp, s} is an optimal circle. if dist(s,s1) is known, then problem becomes to find optimal path from s1 to s. If dist(s1,s2) is known, then problem becomes to find optimal path from s2 to s. Obviously, it’s a optimize substructure problem.

2. DP equation

Assume start from s , d(i, V’) represent the optimal weight of the circle, which go from vertex i, go through all vertices in V’ (a set) once and only once, then go back to s.

There are two scenarios:

① if V’ is empty ,d(i, V’) represents go from i ,go through nothing and go back to s. In this scenario,

d(i, V’)=C(i,s) (distance between i and s).

② if V’ is not empty , it comes to optimize a sub-problem. Try every vertices in set V’, choose the best (minimum) one.

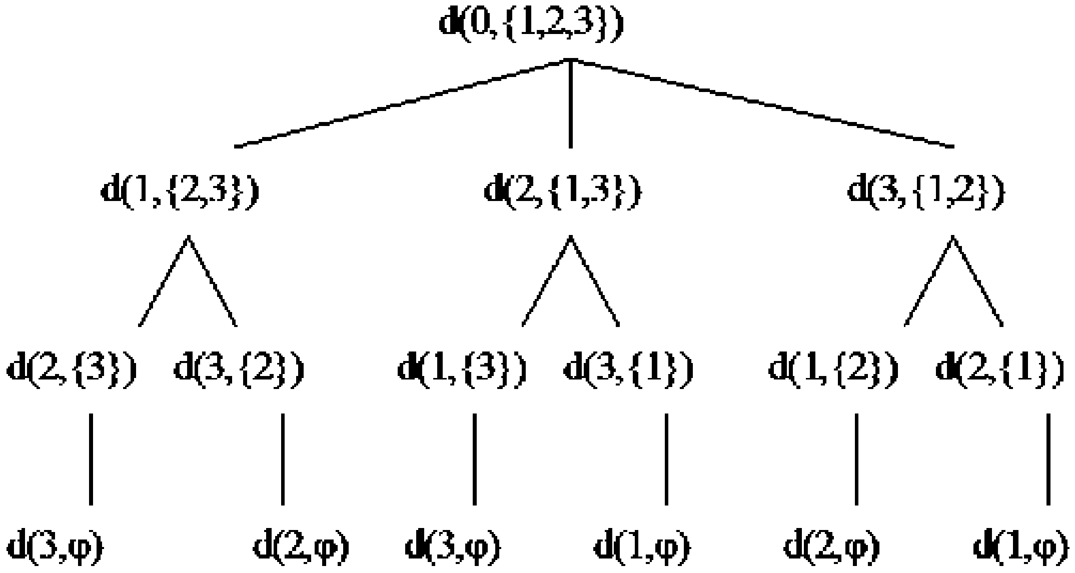
d(i, V’)=min{C(i,k) + d(k, V’-{k})}

Ps : d(k, V’-{k}) is a sub-problem

$$d(i,V')=\begin{cases} c_{is} & V'=\phi, i\neq s \\ \min\limits_{k\in V'}\{c_{ik}+d(k,V'-\{k\})\} & V'\neq\phi \end{cases}$$

3. Case study

Now, use the example mentioned before to explain the process of calculation of DP equation. (Assume start from City/Vertex 0)



①The final result we want is d(0,{1,2,3}) representing the cost of shortest path, which start from City 0, go through City 1/2/3 and go back to City 0.

②d(0,{1,2,3}) can not be calculate immediately and straight forward. As the second level of the “tree” picture shows:

d(0,{1,2,3})=min {

C01+d(1,{2,3})

C02+d{2,{1,3}}

C03+d{3,{1,2}}

}

③the same as d(0,{1,2,3}), d(1,{2,3}) , d(2,{1,3}) , d(3,{1,2})can not be calculate immediately and straight forward either.

d(1,{2,3})=min{

C12+d(2,{3})

C13+d(3,{2})

}

d(2,{1,3}) , d(3,{1,2}) are calculated by same process.

④Based on this equation, only in the last level, the set V’ is empty , Cis can be known directly from adjacency matrix as follow.

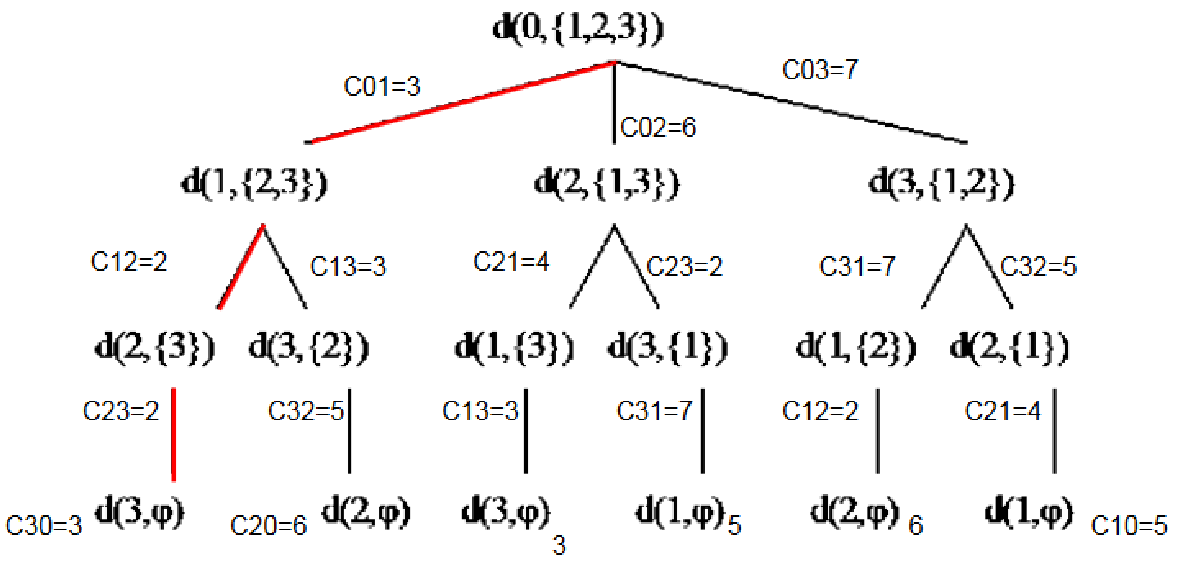
$$C=\begin{pmatrix} 0 & 3 & 6 & 7 \\ 5 & 0 & 2 & 3 \\ 6 & 4 & 0 & 2 \\ 3 & 7 & 5 & 0 \end{pmatrix}$$

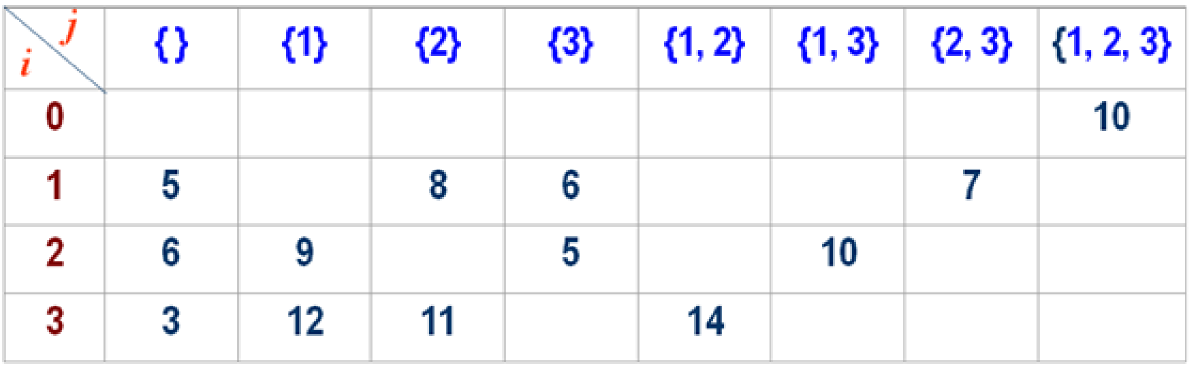
4. Calculation process

Represent d(i, V’) as 2D-table, d[i][j] :

d(0,{1,2,3}) = 3+2+2+3 =10

Shortest path is 0-1-2-3-0.





i \ j	{ }	{1}	{2}	{3}	{1, 2}	{1, 3}	{2, 3}	{1, 2, 3}
0								10
1	5		8	6			7	
2	6	9		5		10		
3	3	12	11		14			

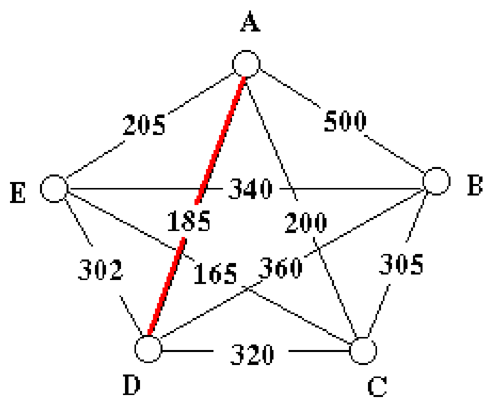
For the C++ code for DP algorithm, please refer to the GitHub(https://github.com/jli98/Travelling-Salesman-Problem/blob/master/TSP-DP.cpp).

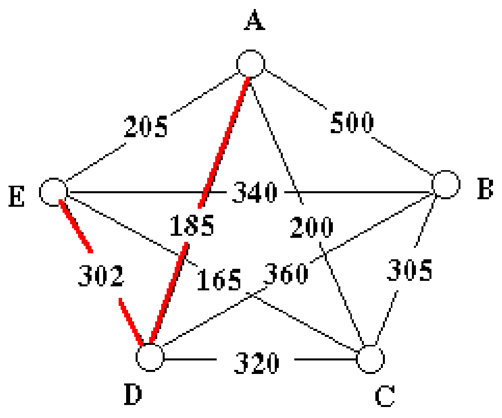
2.C Greedy(Nearest Neighbor)

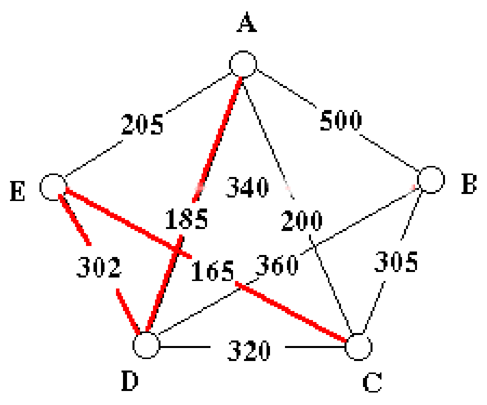
In TSP, we usually need an efficient way to obtain a suboptimal result for other improvement algorithms. This is where Greedy is used.

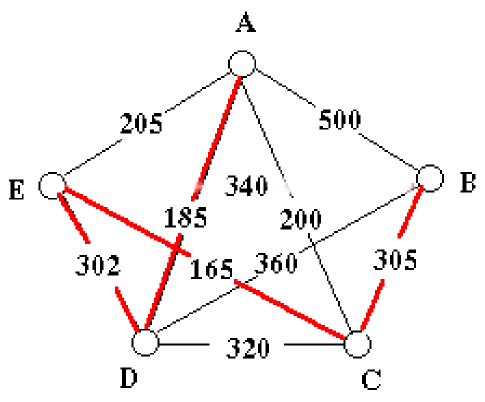
Greedy algorithm is the simplest algorithm. Greedy algorithm is starts with the departure Node 1. Then the algorithm calculates all the distances to other n-1 nodes. Go to the next closest node. Take the current node as the departing node, and select the next nearest node from the remaining n-2 nodes. The process continues until all the nodes are visited once and only once then back to Node 1. When the algorithm is terminated, the sequence is returned as the best tour; and its associated OFV is the final solution.

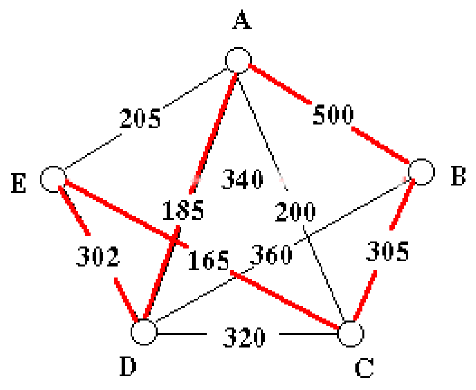
For example,











We start from Node A, then we select a shortest path A-D and go to D point. Repeat this process and after we go through nodes, we return to A point. Finally we get a suboptimal solution A-D-E-C-B-A.

For the JAVA code for greedy algorithm, please refer to the GitHub(https://github.com/jli98/Travelling-Salesman-Problem/blob/master/TSPNearestNeighbour.java).

2.D Lin-Kernighan Heuristic

Lin-Kernighan (LK):

- Most famous and best local search approach for the sym. TSP.
- Developed by Shen Lin and Brian Kernighan in 1973.
- Best exact solver for the TSP is Concorde
- (Applegate, Bixby, Chvatal, Cook).

- Best LK-Code today: Keld Helsgaun.
- Concorde + Code of Helsgaun, 2006: pla85900 solved (world record).
- (Applegate, Bixby, Chvatal, Cook, Espinoza, Goycoolea, Helsgaun)

K-opt neighborhood for tour x:

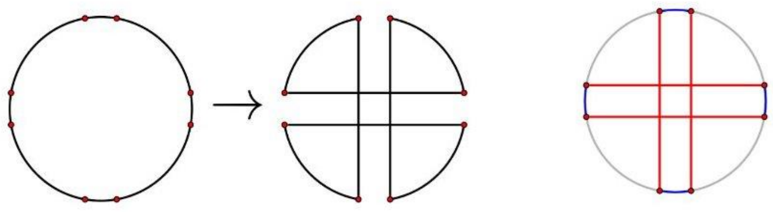
$N_k(x)$ consists of all tours, which can be constructed from X by deleting and adding K edges.

Observation:

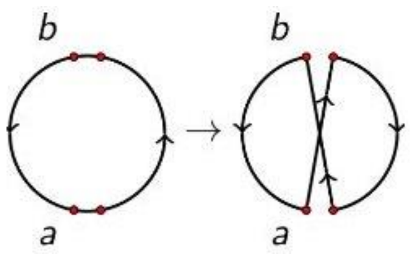
- Two hamiltonian cycles only differ in K edges (2<= k <= n)
- Problem 1: k-optimality in $N_k$can only be tested in $O(n^k)$
- Problem 2: K is unknown.
- Approach: Choose an efficient searchable neighborhood such that K can be chosen dynamically.

Sequential K-opt moves

Double Bridge Move (4-opt):



Flip Operations:

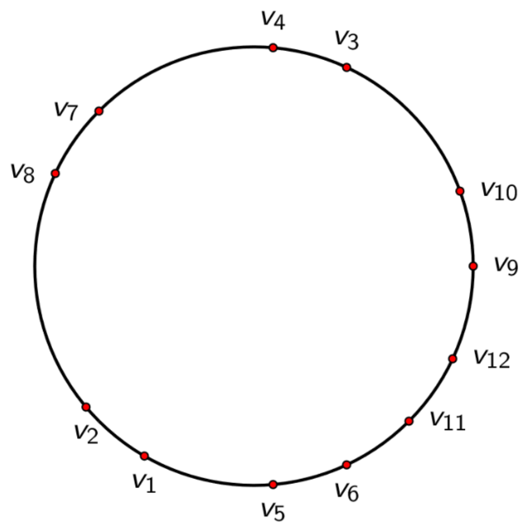


flip(next(a), prev(b))

Gain g_t of flip t:g_t = C(a,next(a)) + C(prev(b), b) - C(next(a),b) - C(a, prev(b))

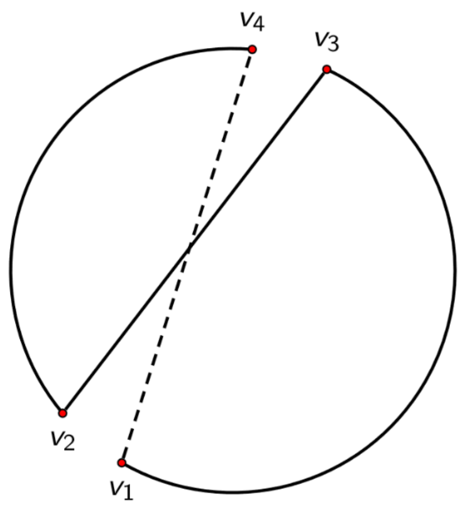
Choose a fixing start node a and construct the alternating path by a sequence of flip operations of the form flip(next(a), prev(b))
The goal within this construction is to obtain

Example:



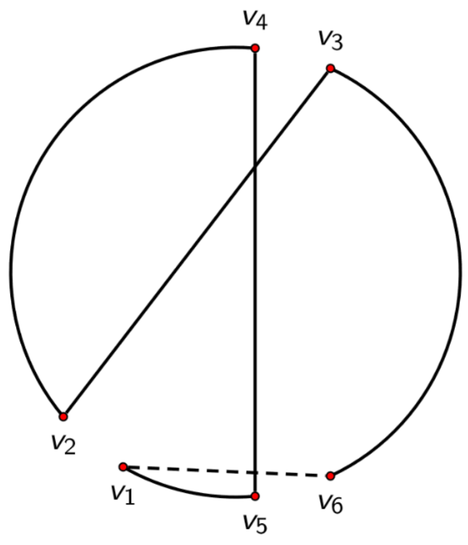
$$v_1 \underline{v_2 - v_8 v_7 - v_4} v_3 - v_{10} v_9 - v_{12} v_{11} - v_6 v_5 -$$

$$\texttt{flip}(v_2, v_4)$$



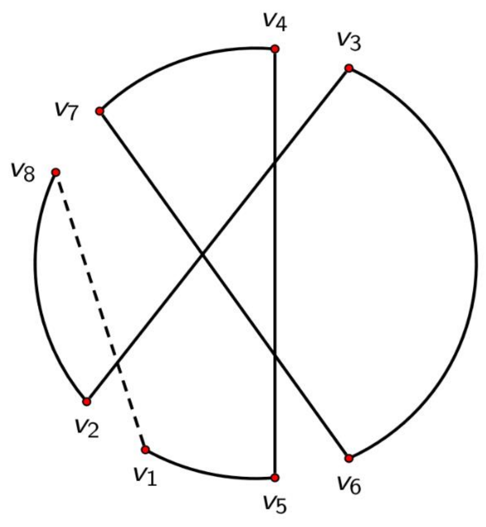
$$v_1 \underline{v_4 - v_7 v_8 - v_2 v_3 - v_{10} v_9 - v_{12} v_{11} - v_6} v_5 -$$

$$\texttt{flip}(v_4, v_6)$$



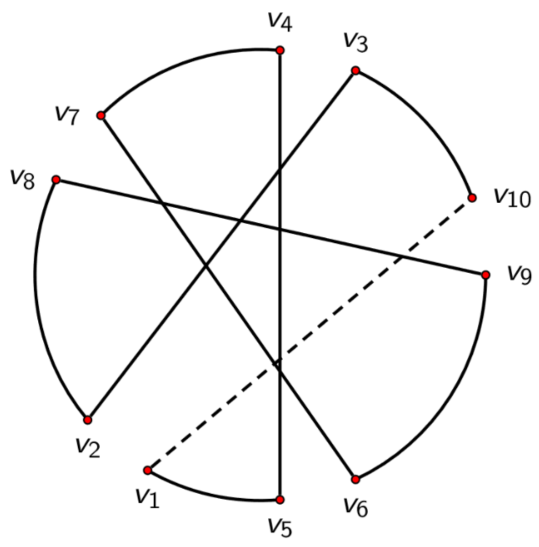
$$v_1 \underline{v_6 - v_{11} v_{12} - v_9 v_{10} - v_3 v_2 - v_8} v_7 - v_4 v_5 -$$

$$\texttt{flip}(v_6, v_8)$$



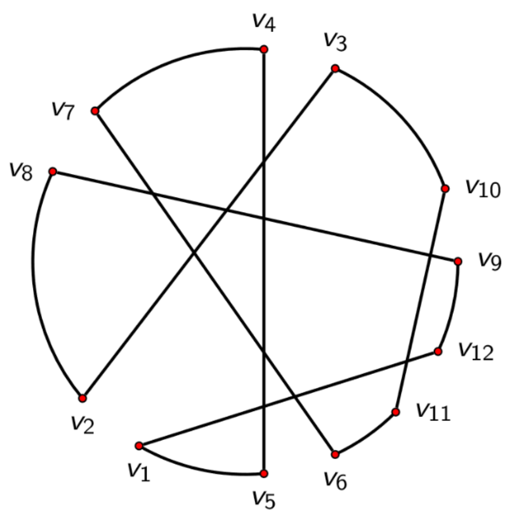
$$v_1\,\underline{v_8 - v_2\,v_3 - v_{10}}\,v_9 - v_{12}\,v_{11} - v_6\,v_7 - v_4\,v_5 -$$

$$\texttt{flip}(v_8, v_{10})$$



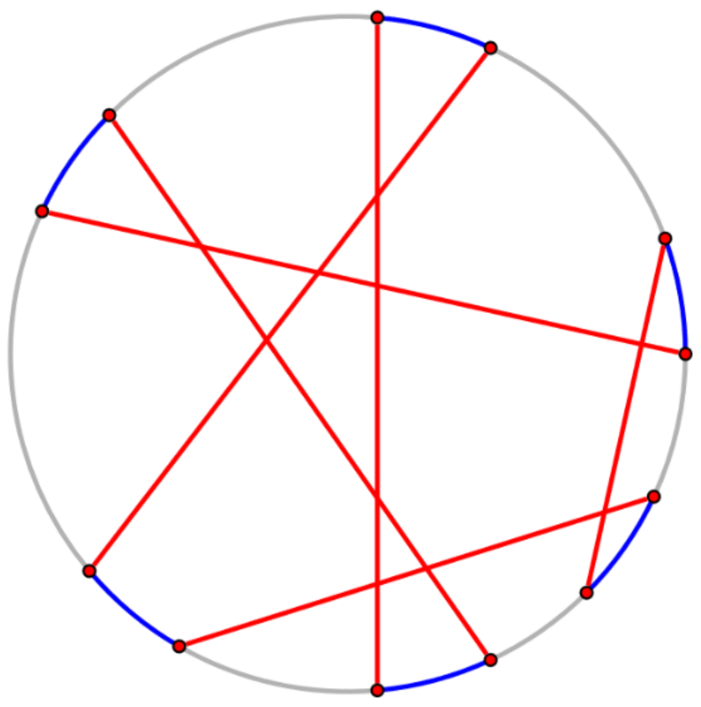
$$v_1\,\underline{v_{10} - v_3\,v_2 - v_8\,v_9 - v_{12}}\,v_{11} - v_6\,v_7 - v_4\,v_5 -$$

$$\texttt{flip}(v_{10}, v_{12})$$



$$v_1 v_{12} - v_9 v_8 - v_2 v_3 - v_{10} v_{11} - v_6 v_7 - v_4 v_5 -$$

Final result:



Our project LK is based on 2-opt. For n nodes in the TSP problem, the 2-Opt algorithm consists of three steps:

Step 1 Let S be the initial solution provided by the user and z its objective function value (In our model, we use greedy algorithm to setup the initial solution and objective function value.). Set S*=s, z*=z, i=1 and j=i+1=2.

Step 2 Consider the exchange results in a solution S’ that has OFV z’<z*, set z*=z’ and S*=S’. If j<n repeat step 2; otherwise set i=i+1 and j=i+1. If i<n, repeat step 2; otherwise go to step 3.

Step 3 If S^{1}S*, set S=S*, z=z*, i=1, j=i+1=2 and go to step 2. Otherwise, output S* as the best solution and terminate the process.

Observe that the 2-opt algorithm considers only pairwise exchange. Initially, the algorithm considers transposition of Nodes 1 and 2. If the resulting solution’s OFV is smaller than that of the initial solution, it is stored as a candidate for future consideration. If not, it is discarded and the algorithm considers transposing of Nodes 1 and 3. If this exchange generates a better solution, it is stored as a candidate for future consideration; if not, it is discarded. Thus, whenever a better solution is found, the algorithm discards the previous best solution. This procedure continues until all the pairwise exchanges are considered. Because each node can be exchanged with n-1 other nodes and there are n nodes in total, there are n(n-1)/2 different exchanges. These n(n-1)/2 exchanges are considered in step 2. The solution retained at the end of step 2 is the one that provides the most improvement in the OFV. Starting with this as the new solution, the algorithm repeats step 2 to find another better solution. At some stage, no improvement in the current best solution is possible, and then the algorithm terminates. The remaining solution is returned to the user as the best one.

For the C++ code for greedy algorithm, please refer to the GitHub(https://github.com/jli98/Travelling-Salesman-Problem/tree/master/TSP-LK).

3. Improvement of Algorithms

Greedy and L-K algorithms above do not eliminate the crossing of routes. Of course, these crossings of routes lengthen the total distance of travel in TSP. We can improved these algorithms by getting rid of these crossings.

This algorithms require coordinator graph representation for TSP problem, because it judge the existence of crossover by calculate the coordinate of vertices.

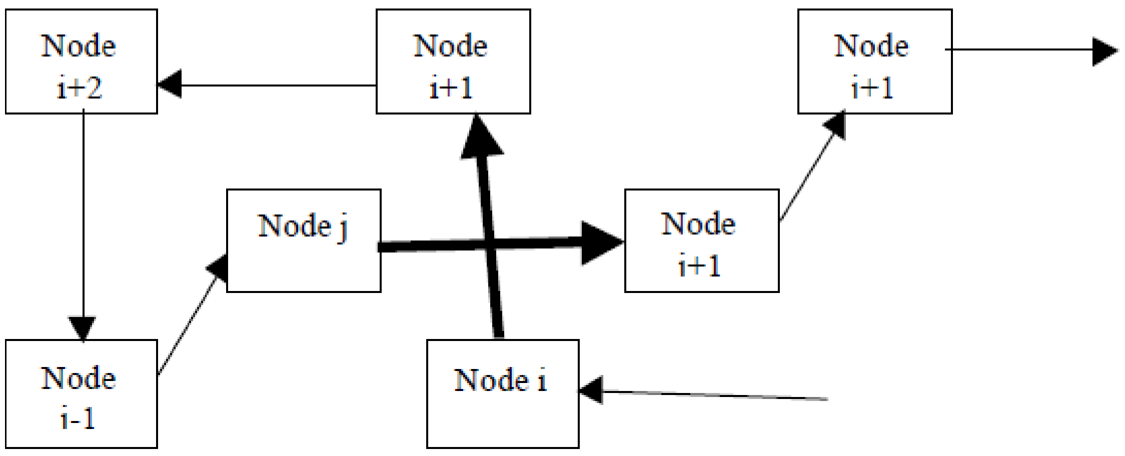
The process is as follows.

First, obtain the solution with any of the preceding algorithms.

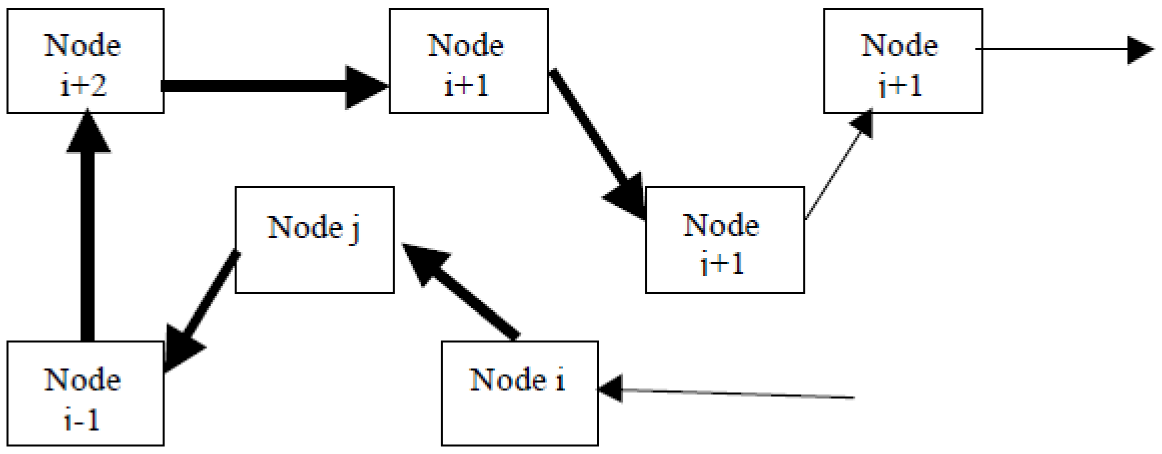
Second, pick up the segment between Node 1 and Node 2, check any other segment against the 1-2 segment to see if these 2 segments intersect each other. If not, pick up the segment 2-3, and check consequent segment against segment 2-3. If the intersection is found for any two distinct segments, re-route the loop. And the process is continued sequentially throughout the network.

The mathematical implementation of both checking and re-routing is as follows: The current 2 segments are $i \to i+1$, $j \to j+1$ ($1 < i < i+1 < j < j+1 < n$). The line of $i \to i+1$ segment can be expressed as $x=x_i+(x_{i-1} - x_i)*t$, $y= y_i +(y_{i+1} - y_i)*t$, and the line of $j \to j+1$ segment can be expressed as $x=x_j+(x_{j-1} - x_j)*s$, $y= y_j +(y_{j+1} - y_j)*s$. Given these parametric expressions and the coordinates of these 4 nodes, the coordinates and associated s, and t can be calculated for the intersection of these lines. If both s and t are 0 < t < 1 and 0 < s < 1, it is determined that these two segments are intersected. Otherwise not. When they are intersected, transpose node i+1 and node j. There would be neither intersection on these 2 segments nor break of the loop.

Since there are N+1 routes for N cities, N(N+1)/2 crossing-checks are necessary. Our model finds these non-crossing methods substantially improved upon these algorithms for large-size problem.



(Original Result)



(Improved Result)

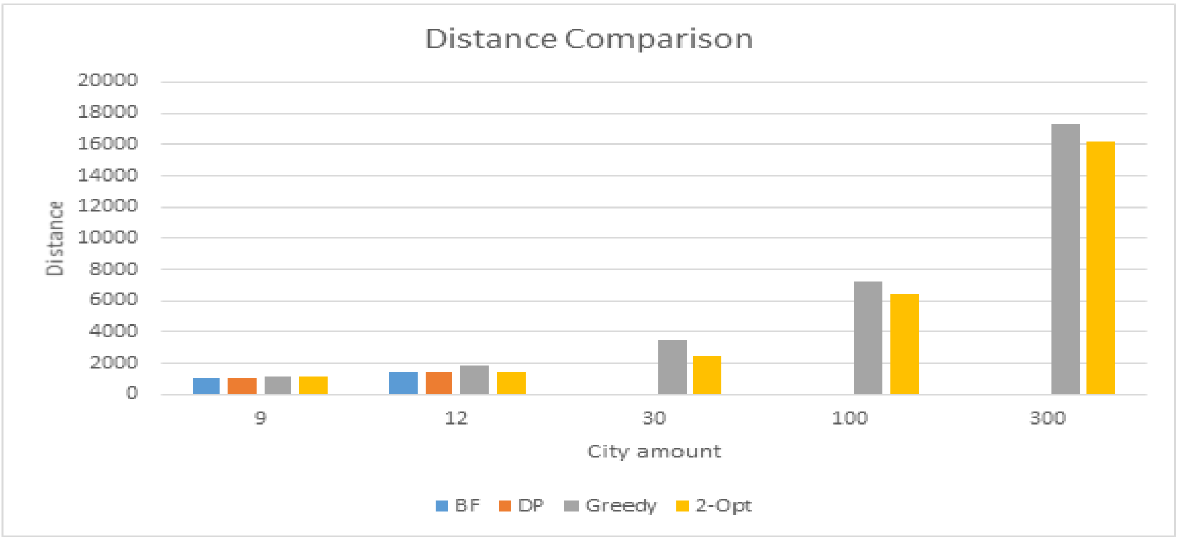
Since we use adjacent matrix as our input data format, we did not actually apply this improvement algorithm in our project

4. Results and Analysis

For the input data, please refer to the GitHub(https://github.com/jli98/Travelling-Salesman-Problem/tree/master/Input%20file). All input data are ramdomly generated by c++ program(TSP_inputfile.cpp).

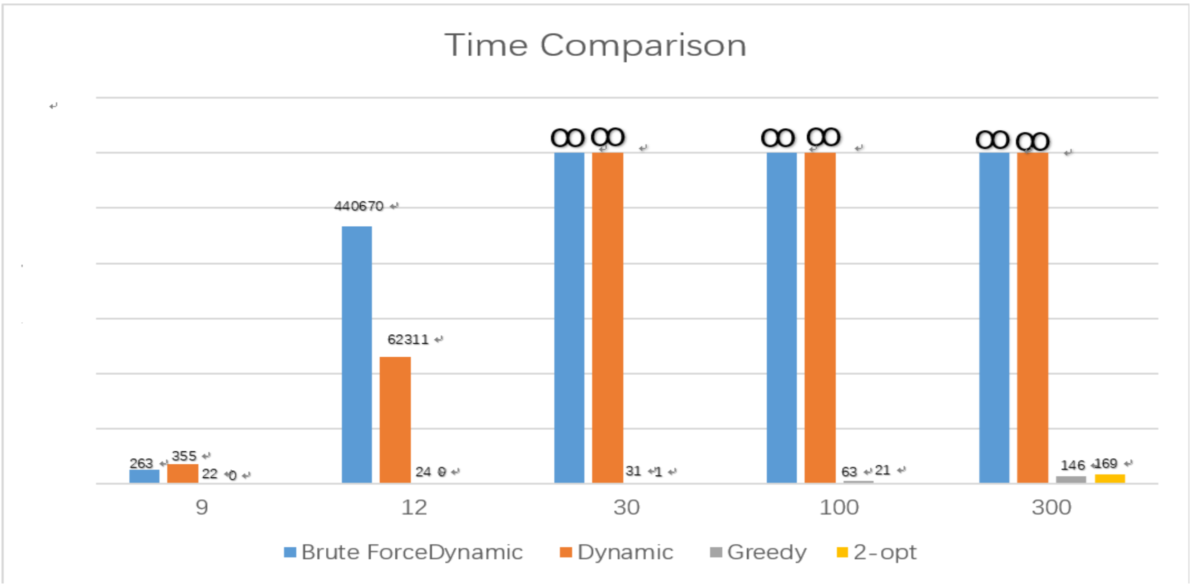
4.A Length Comparison

N / Algo.	9	12	30	100	300
Brute Force	1070	1493	Unknown	Unknown	Unknown
Dynamic Programming	1070	1493	Unknown	Unknown	Unknown
Greedy	1099	1815	3444	7215	17286
2 - Opt (Based on Greedy)	1099	1493	2442	6430	16156



4.B Time Comparison

N / Algo.	9	12	30	100	300
Brute Force	263 ms	440670 ms	>20 min	Unknown	Unknown
Dynamic Programming	355 ms	62311 ms	Unknown	Unknown	Unknown
Greedy (JAVA)	22 ms	24 ms	31 ms	63 ms	146 ms
2-Opt (Based on Greedy)	0 ms	0 ms	1 ms	21 ms	169 ms



According to the comparison table above,

Brute force and dynamic programming can give the optimal results when input n is below 15. When input scale increased, theses two methods are too expensive to obtain a solution.

Dynamic Programming is a better way than Brute Force to get an optimal solution. It cost much less time than Brute Force, but still perform bad when you have more than 12 cities. Sometimes, if N is very small, it even performs worse than BF.

Greedy is better in cost, but the results are so inaccurate that it can not be used to solve TSP.

The improvement algorithm 2-opt is based on the existential result from greedy, it’s a practical way to obtain a nearly optimal solution. Notice that we also set 100 hundred times flip operations as limit, so when input scale is small, it sometime come out with optimal solution, but when input scale is big(e.g. N=300), the limitation of flop operations also limit the performance of improvement. Therefore, the large scale input require more times flip operations to get a better solution.

5. Conclusion

Brute Force is a construction algorithm which can come up with optimal solution. However, it takes too long time to compute, we can’t get a solution when more than 12 cities, because its complexity is O(n*(n-1)*(n-2)*...*2*1)=O(n!)

Dynamic Programming is not a good idea for TSP, because it performs worse than brute force when N is small and it can not handle large scale either as brute force.

Greedy is the fastest way to get an approximate solution because of its complexity is O(n), and most of time it is too inaccurate to satisfy our request.

LK is a practical and efficient way to improve algorithms. But after optimizing, it still could be an inaccurate solution. In manipulation, we need to control the flip operations times to obtain a solution that both accurate enough for distance and practical enough for costing time.

Coordinator representation is most practical way to represent TSP problem, and Eliminate crossover is a useful improvement algorithm, we'll implement them in the future.

6. Reference

1) *Comparison of TSP Algorithms*, Byung-In Kim, Jae-Ik Shim, Min Zhang. December 1998
2) *Exercise 12: Implementing the Lin-Kernighan heuristic for the TSP*, Markus Reuther, January 19, 2012
3) Learn DP for TSP from http://blog.csdn.net/gfaiswl/article/details/4749713
4) Learn DP for TSP from http://blog.csdn.net/zouxinfox/article/details/1917107
5) Learn LK programming from https://github.com/lingz/LK-Heuristic