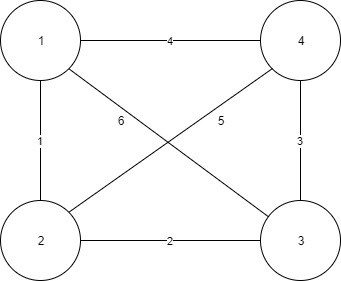
Travelling salesman problem

Abstract

Introduction

Travelling salesman problem: Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. This problem is a famous NP hard problem, which means there is no polynomial time know the solution for the problem. However, when the number of city is very small, we can easily figure out the answer. For example, consider the given graph. There are 4 cities, which is 1,2,3,4. We can quickly figure out that the shortest route would be 1+2+3+4=10. However, if we search deep for how we get this out. The answer is that, we generate all the possible route, which is 1\*2\*3=6 possible ways. What’s more when we are doing this we set 1 as the starting and ending point automatically. Theoretically, the idea is right. Calculate all the possible route and find out the minimum. Although the idea is simple, in real world, there would be much more cities involved. As the number of city goes up, the possible routes would increase like crazy. It will take too much time to calculate it, even we use computer programming. In this case, researchers find some other way to solve this problem, even the answer may not be exactly the right one, it is still very close.

**naive / BF implementation**

There are a lot of solutions for the traveling salesman problem. Naïve/BF is the simplest one. The idea about naïve solution is easy to understand. Consider all the possible routes and compare with other and find the lowest cost. The algorithm of naïve/BF implementation is:

1. Consider a city as the starting and ending point.
2. Generate all (n-1) permutations of cities, which means generate all the possible routes.
3. Calculate cost of every permutation and keep track of min cost route.
4. Return the minimum cost route.

The code below is the pseudocode and flowchart for the BF implementation.

void brute\_force\_solver(**const** Tsp\_map& cities) {

Tour tour(cities.get\_default\_tour());

double best\_score = cities.score(tour);

Tour best\_tour(tour);

cout << tour << " score = " << best\_score << endl;

**while** (next\_permutation(tour.begin() + 1, tour.end())) {

double s = cities.score(tour);

**if** (s < best\_score) {

best\_score = s;

best\_tour = tour;

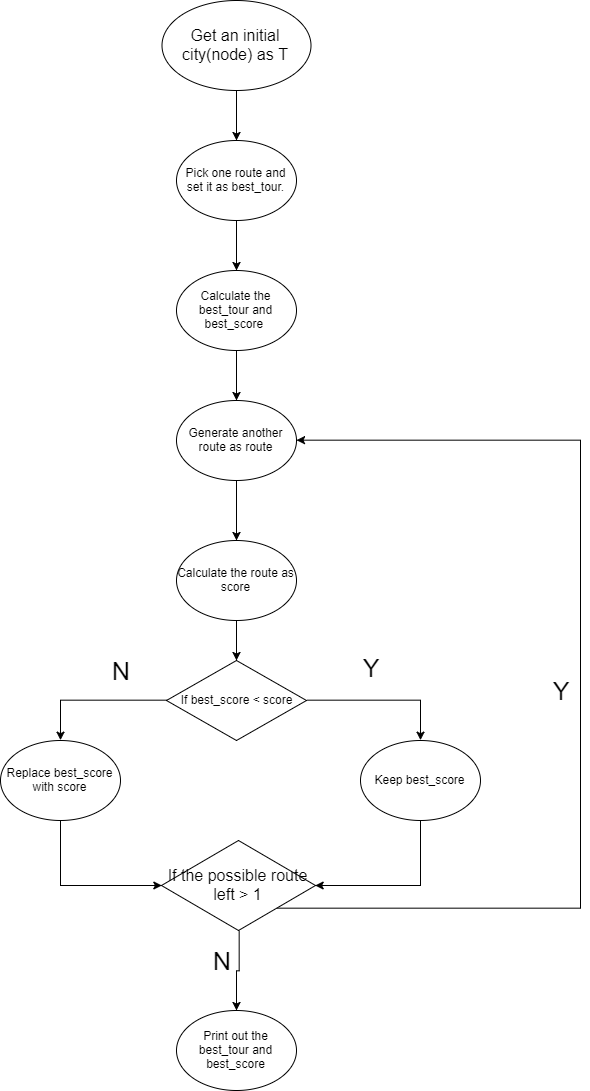
}

cout << tour << " score = " << s << endl;

} *// while*

cout << "best tour: " << best\_tour

<< "**\n**score = " << best\_score << endl;

}

Based on the flowchart and pseudocode, we can generate those advantages about the naïve/BF implementation:

1. Naïve/BF implementation will find the most correct route if we have enough time to run this program. (The idea about most correct route I will declare it later in the paper)
2. Simple algorithm, and easy coding.

However, we can generate some disadvantages as well.

1. It is not efficient.
2. It takes too much time to run the program, no one in the real world would ever take so much time to calculate this. Not good for real problem.