Network Optimization
Local Search for TSP

ROBERTA SABATINI, FILIPPO SPINELLI,
VAHID FARJOOD CHAFI, MOHAMMAD RAOOF HOJATJALALI

May 2020

The Travelling Salesman Problem
Given a set of cities and the distances between each pair of cities, what is the shortest possible tour that visits each city exactly once, and returns to the starting city?

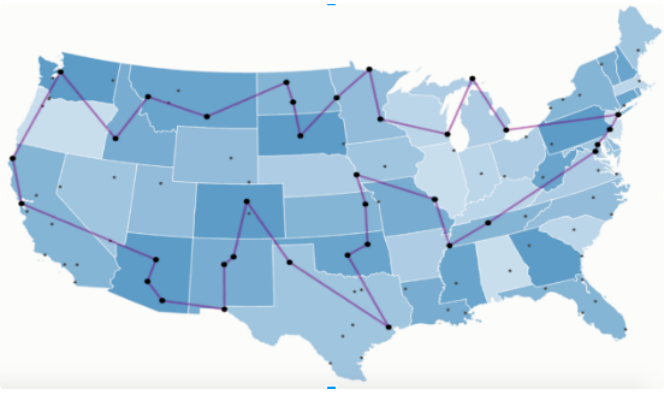

Figure 1: Travelling Salesman Problem

Objective:
Implementation of the Local Search Algorithms for TSP and also some speed-up techniques in order to make the algorithm faster:
1) Nearest Neighbor and Random Path Algorithm
2) 2-OPT Algorithm
3) 2-OPT Algorithm with DLB
4) 2-OPT Algorithm with Pruning (p-nearest neighbours)
5) 2-OPT Algorithm with Pruning (fixed radius)
6) 3-OPT Algorithm

Introduction

TSP is one of the most studied optimization problems of recent decades. The problem concerns the definition of a path at minimal cost that connects all the nodes of a given constellation, visiting each node once only.
Unfortunately, the computational time and the computation itself, due to the search for the optimal solution, are strongly affected by the number of nodes which compose the map.
Given this, over the years, many conjectures have been conceived and numerous heuristics have been developed for the discovery of an admissible solution to the problem. Among the various heuristics, Local Search represents a widespread technique, based on iterative imporovement, which solution represents a good compromise in terms of computational time and fidelity to the optimum.
As far as possible, in terms of "computing power" we have considered some of these heuristics concerning Local Search and we have implemented the respective algorithms in order to test them and compare the results. The Local Search heuristics examined are the *2-opt*, two different mtehods to speed-up the 2-opt algortithm, in particular *Don't Look Bits* (DLB) and *Neighbourhood pruning*, and finally the *3-opt*.
For the testing phase we have chosen to use datasets of different sizes, including some real constellations of nodes and others randomly generated.

Local Search for TSP

Class City and distance functions

- *Class City*:
  In order to associate x and y coordinate to each city, we have defined a class called *City*. So, with that class we can easily calculate and get the coordinate corresponding to each city.

- *getDistance*:
  This function is used whenever we need to calculate the distance between two specific cities. We used the Euclidian Distance method for this propose.

- *getTotalDistance*:
  This is the function which we used to calculate the total distance of the path. We loop through all the cities to get the distance between one city and the next one and, finally, we also add the distance between the last node and the starting one.

Initial solution

Since an improvement heuristic, as in the case of Local Search, requires an initial solution to be iteratively improved, it must start from a feasible initial condition.
In the specific case of the Travelling Salesman Problem, an initial feasible solution is represented by a Hamiltonian cycle.
The choice of an admissible initial solution can be achieved through a simple constructive heuristic or randomly generated by an algorithm. In both cases,

since the optimality of the solution of these algorithms is not their main goal, the computationl time required is really small and this is due to the semplicity of the algorithms themselves.
In our case, the heuristic chosen for the construction of the initial tour is the *Nearest Neighbour*.
Let's see in detail the functions which implement the two chosen algorithms related to the initial solution for the TSP.

Nearest Neighbour

In Nearest Neighbour algorithm — function called *getPathFromNearestCity* in the code —, once selected a starting node, at each iteration we simply search for the closest node to the current one, called *currentCity* and we append it to the end of the *path*, thus increasing the list containing the tour. Moreover, at each iteration, we also take care of removing the node *currentCity* just added to the path from *TemporaryCities*, which is the list of cities still available because not yet visited; after finding the nearest city to the current one, called *nextCity*, we also update the *currentCity* setting it to *nextCity*.
At the end, in order to form a cycle, we add an arc that join the first node with the last one of the path.
In any case, this type of heuristic approach allows us to obtain an initial solution that already has a good quality; however, there still remains room for improvement.

Random initial solution

In the algorithm that produces a random initial solution — function called *getRandomPath* in the code —, at each iteration, we append a node belonging to the *TemporaryCities* list to the end of the increasing current path; this list contains the cities still avalaible because not yet visited. Therefore, at each iteration, we remove from the *TemporaryCities* list a city placed in a position randomly generated by the *randint* function; moreover, we update the number of available cities by decreasing it by one, because it corresponds to the size of the TemporaryCities list.

Observations

In order to compare the quality of the different Local Search algorithms, we have decided to evaluate these algorithms also strating from different initial solutions. Therefore, we have test the Local Search algorithms analyzed using for them, as initial condition, both solutions produced by *getPathFromNearestCity* algorithm and by *getRandomPath* algorithm.
During the testing phase, a strong sensitivity of the final solution to the different initial conditions was observed.

Note that the length of the final path of the Nearest Neighborhood algorithm depends on the choice of the first node. Among all the possible initial solutions, due to diffirent first nodes, we have chosen to use any one, since very consistent datasets can be used to test the Local Search algorithms implemented, and it was impossible, or at least not useful, to average the lengths related to all the possible solutions initials.
Obviously the initial node, although chosen randomly, remains always the same.

The same can be said for the initial solution given by *getRandomPath* function. Note that the route is generated randomly, but the seed is fixed. Therefore, although random, among the almost infinite possibilities, also this initial solution remains always the same.

2-opt Algorithm

In order to implement the 2-opt algorithm we have used two functions. One for finding the optimal path called *getPathFromTwoOpt* and another one which is used for exchanging the arcs called *twoOptSwap*. They work as expained below.

- *getPathFromTwoOpt*: first of all, the function receives an initial path. Then, we define a variable called *improved* to evaluate whether it's possible to find any improvement or not and it is initially set to be `True` in order to go inside the while loop, which is the main loop.

Inside the while loop, we first have *minDistance* variable which calculates the overall path distance by the function *getTotalDistance* of the initial path. We update this variable each time we find an improvement in the path by comparing it with *newDistance* variable.
In order to compare all edge exchanges, we use two nested loops to find an improvement. The outer loop counts from the first city in the path upto the last two city; instead, the inner loop counts from the next city to the i-th upto the last one city. Then we investigate all the possible edge exchanges for each specific city comparing to the other edges. The comparison is done by *Delta Evaluation* method. So, if *Delta Evaluation* finds any improvement, then *twoOptSwap* function is called for swapping those founded arcs; otherwise it checks for the next outer loop counter.
In case of improvement, we first computes the total distance and update *newDistance*, then we compare it with *minDistance*; if *newDistance* is less than *minDistance*,then we update the path and also the value of *minDistance* respectively.
Finally, we update the variable *improved* to be `True` in order to have the next iteration of the while loop.
The while loop stops in case we don't find any improvement in overall calculation.

- *twoOptSwap*: This function receives the path with two extra arguments, the i-th and the j-th cities respectively. First we define a variable called *new_route*, which contains cities from the first city of the path up to the i-th city. Then, it extends the variable by reversing the path from [i] city up to [j +1] city. Finally, the remaining path is added to the variable and *new_route* is returned by the function.

Speed-up of the 2-opt Algorithm

The 2-Opt algorithm gives us a good improvement in terms of quality of distance but, on the other side, the time increases dramatically. So, we have decided to use some techniques to speedup the running time of our algorithm.

Don't Look Bits

First we have used *Don't Look Bits* technique in order to make the algorithm faster. The function — called *getPathFromTwoOptWithDLB* in the code — works as follow.
In *getPathFromTwoOptWithDLB* function we use a special flag *DontLook* for each of the cities.
Initially all the flags are turned off, which means that we allow searching for improving moves starting from a given city. When a search for improving move starting from a city C fails (in the outer loop), then the bit for this city is turned on DontLook[i] = `True` . When a move is performed in which this city is involved as one of the endpoints of the exchanged edges (in the inner loop), then the bit for this city is turned off DontLook[j] = `False`.

Pruning technique

The second technique of speed-up is *Neighbourhood pruning*. In particular, this technique consists in applying the 2-opt algorithm considering, for each node, only the nearest cities.
In our case, it has been decided to estimate the closest neighbours according to two different parameters. The first filter is to consider only a given number *p-nearest* neighbours and the second one, instead, only the nodes placed at a distance less than a *fixed radius*. Let's see them both in detail.

- **P-nearest neighbours**

  The aim of this function is to return the list of neighbours of each city, *city1*, that are in the neighborhood whose size is given by the parameter *NumP*, that is the number of the p-nearest neighbours. The value of *NumP* is estimated according to the parameter p, given as input to the function.
  First of all, we keep track of the distance between each *city1* at issue and all the other cities of *cities* list in a dictionary-like structure, *MapDistance*. The key of each element is represented by the index city2 which allows us to iterate between all the cities in *cities* list; the value,

instead, is the distance between *city1* and each of *city2*. The map is sorted according to the value in increasing order and then, the dictionary structure is converted into a list containing only the keys, namely the cities (now sorted in increasing order of distance from *city1*).
At this point, we take only the p-nearest neighbours, overwriting the list *NearestNeighbours*.
At the end of each iteration, the list *NearestNeighbours* is appended to list *NearestNeighboursList* and then we proceed considering the next *city1*. Finally, the list *NearestNeighboursList* is returned by the function.

- **Fixed radius**

  To implement this criterion, we have resorted two functions; one — called *EstimationFixedRadius* in the code — for estimating the value of the fixed radius and the other — called *getFixedRadiusNeighbours* in the code — for finding neighbours located at a distance lower than this radius.

  - *EstimationFixedRadius*: in this function, for each city, we calculate the distance between the city itself and all the other nodes. The aim is to find the minimum maximum distance, *GetMinMaximum*; we initially set an upper bound for this parameter. Therefore, for each city, after sorting the distances in increasing order, we keep track of the minimum distance. At each iteration, if a lower *GetMaxMinimum* is found, we take care to update the value of this parameter.
    At the end, we estimate the value of *FixedRadius*, which then is returned by the function. This measure is calculated as a percentage of the *GetMinMaximum*, according to the input parameter p received as input by the function.
  - *getFixedRadiusNeighbours*: as previously said, the purpose of this function is to find and return, for each city, the list of neighbours which are in a neighborhood delimited by the radius.

Therefore, first the function calls *EstimationFixedRadius* function to calculate the parameter *FixedRadius*. Then, for each *city1* of *cities*, we find the list FixedRNeighbours.
At the end of each iteration, this list is appended to list *Nearest-NeighboursList*, which finally is returned by the function.

2-opt Algorithm with Pruning

Now that we have seen the two methods to determine the closest neighbours, let's see the general function that realizes the 2-opt algorithm with pruning, regardless of the filter criterion applied.
In particular, for each node of *initial_path* list, which corresponds to the initial solution that the function itself receives in input, we calculate the list of nearest neighbours cities calling one of the two pruning methods, given as an input parameter. This list is assigned to *localCities list*.
At this point, the same reasoning already seen for the 2-opt algorithm is repeated, except for the fact that the two nested loops allow us to iterate only over the *localCities* list. This means that the 2-opt is no longer done iterating over the whole tour, but only over the *localCities* list.
If *Delta Evaluation* finds any improvement, then *twoOptSwap* function is called for swapping those founded arcs belonging to the whole path. Then, if the *newDistance*value relative to *newPath* is lower than the current *minDistance*, we do the swapt also within the *localCities* list and we update *path* list by overwriting it with *newPath*.

3-opt Algorithm

In 3-opt algorithm we try to improve the length of the path by removing 3 connections; this let us to create 7 new combinations (excluding the original one that we had) which we can analyse to find the optimum in order to improve the tour. The algorithm — called *getPathFromThreeOpt* in the code — works as follow.
In function *getPathFromThreeOpt* we try first to find three candidate edges to apply the algorithm. So, we do this by nested iterating through the path in 3 levels:

```
for i in range(len(path) - 3):
   for j in range(i + 1, len(path) - 2):
       for k in range(j + 1, len(path) - 1):
```

Then we try to create new combinations with "reversing" and "substitution" paths between these three nodes.
Let's assume that our original path is like:

```
     A                B                  C                 D
path[0:i] -> path[i:j + 1] -> path[j + 1:k + 1] -> path[k + 1:]
```

One combination would be to substitute between path B and path C:

```
     A                C                  B                 D
path[0:i] -> path[j + 1:k + 1] -> path[i:j + 1] -> path[k + 1:]
```

Another combination can be create by reversing path C

```
     A                reverse C                    B                D
path[0:i] -> reversed(path[j + 1:k + 1]) -> path[i:j + 1] -> path[k + 1:]
```

In this way we can create 7 new combinations; for each one of them, we get a new path of which we can calculate a new total distance. In case of improvement, we choose it as our new optimum and we continue doing this for all of our new combinations.
At the end, we can find out if our three candidate edges give us an improvement or not.
We do this for all three edges in the original path and, if we find any improvement, we reiterate this algorithm again.