ESILV LÉONARD DE VINCI | ÉCOLE D'INGÉNIEURS PARIS-LA DÉFENSE

2016-2017

Traveling salesman problem

GROUPE IBO1

FELICIEN GAZON- ABDELLAH SABRY

This document discuss some algorithms and their complexity used to approximate the longest path of the traveling salesman problem applied to cities in France. We use an exact solution and two heuristic approximations for this NP problem. Finally we create a spanning tree with some indicated modifications and calculate its cost.

Note: The text in red discuss the complexity, in black explain and discuss the algorithms.

Data structures

For modeling the cities, we use a list of structure “city” which have 4 attributes:

- Key that identify a city
- Name of the city
- Longitude of the city
- Latitude of the city

For stocking the distances between cities, we use an adjacent matrix such as the element M[i][j] is the distance between the city with key “i” and the city with key “j”. If the distance between city I, and city j is closer than 100 kilometers, we stock INFINITY in their places.

The purpose of this problem is to find the maximum length path of the trip of traveling salesman problem starting from "PARIS" and visits each city exactly once and returning to "PARIS” using an exact method and two heuristic methods.

Algorithms

a) Using an exact method to find an exact solution.

Given:

- An array with the name of cities
- Length of this array
- The limit
- The adjacent matrix
- The List of cities
- The source city which is PARIS

Output: display the optimum combinations of the cities (that have the longest distance) in each iteration

Algorithm:

This algorithm is a recursive function. The algorithm will display the optimum result (the longest path between cities) for each iteration.
The array token (on the given), is the array containing the cities name after deleting “PARIS” the source city. Limit is the index where we begin our combinations, the adjacent matrix is the matrix that contains all distances between cities and the source city (PARIS).

So what our algorithm does, is all the combinations possible (without PARIS, which is added in the end of the function to begin and the end of the list), then check if this path is possible (it doesn’t contains any distance equals to infinity).

To do that, we iterate a loop from limit (which is equals to 0 in the beginning) to the length of the array. Complexity O (n)
In each iteration we:

- swap the two elements: array [limit] and array [index] in the array, Complexity O (1)
- re-call the same function (Combination) replacing the limit with limit+1 and for others it takes the same values Complexity O (n-1), because this function will contain the same loop, but will begin at 1 to length.
- Finally we re-swap the same elements to let it as it was. Complexity O (1)

We'll keep doing this until (length-limit) is equals to 1 (This means that we are at the last element of the array) Complexity O (n*n-1*n-2*…*1). Because every time we will re-call the recursive function the complexity decrease by 1.

When we arrive to that step we:

A. convert the array to List of cities Complexity O (n)
B. add "PARIS" at the begin and the end of this List Complexity O (n)
C. compute the total path between the cities in the List Complexity O (n)
D. Compare it to the optimum distance, if and only if the new distance is greater than the optimum distance we change the result List and the value of the optimum distance. Complexity O (n) if we change ,otherwise Complexity O (1)

Complexity discuss

The complete complexity of this algorithm is O (n!). To create the combination of the cities, we need to 0 (n!). Then when we have to convert the array to a list of cities that take O(n).
So O(n)+O(n!) = O(n!). That's why our algorithm have a complexity result of O(n!) and its just impossible to run it on 72 cities. We run it using 10 cities and we had to wait a moment before the result was displayed.

b) Using an heuristic algorithm Lin-Kernighan (2-OPT)

Function `two OPT`:
Given:

- The adjacent matrix
- The List of cities
- The city source with is PARIS

Output: return the cities List of optimum combinations of the cities (that have the longest distance)

Algorithm:

This algorithm is an iterative function. The algorithm will return an approach of the optimum result (the longest path between cities).
To apply this algorithm, we need to have a possibility that works, not the optimal solution but just a solution. To create this solution we edit the List of cities such as we delete "Paris" from the List, then we re-insert it in top and the bottom of the List. Complexity O (n) We travel the list to delete PARIS O(n) + then we re-insert it in the begin 0(n)+ then we insert it in the end O(n)=O(n).

So what our algorithm does, is swapping the cities that are one after the other (when it doesn't create any distance equals to infinity), in purpose to create the longest path possible.

We travel the edited list of cities, to begin swapping. Complexity O(n)

- Before looking to the new "promising" edges, we make sure that if the old edges isn't equals to infinity, and the new one aren't, we just swap it without other conditions. Complexity O (1) because we get the distance directly in the adjacent matrix.
- Then we make sure that the new "promising" edges exists (that they aren't equals to infinity), if its equals we just skip those cities. Same complexity of the last instruction.
- Finally if the other conditions aren't accomplished, we check if the new promised edges is bigger than the old one. If it's bigger, we swap those two cities Complexity O (n) in the worst case because we can travel all cities to swap those two cities.
  The result of the complexity of this bloc is $O(n^2)$.The O(n) of traveling all cities and then the swap that add a O(n) complexity for each iteration in worst case.

We travel the cities as long as there are modification in the result list, but if we exceed 72 (number of cities) we just break the function to not having a bigger complexity. Complexity of this single loop O (n), but we have a result of O(n^2) complexity in the previous bloc, so the complexity goes to O(n^3)

Finally we just make sure that everything is fine:

- Check if there are any edge which distance is equals to infinity (there shouldn't be an edge there, so the modified given list in parameters wasn't a good solution from the beginning. Complexity O (n) because we travel the list of cities
- The last city of the List constructed have a connection to PARIS, if the distance isn't infinity we have a good approach, otherwise we aren't able to approximate the path using this method Complexity O (1) we get the distance on the matrix directly

Complexity Discuss
We have a complexity of O(n) because we travel all the cities (and for each the complexity is O(n) for travelling the list of cities and for each there are another O(n) for all what happens when we swap cities) . So the complete complexity is O(n)+ O(n^3) + O(n) = O(n^3)

c) Using an constructive heuristic algorithm Local Search (The farthest neighbor)

Function Local Search:
Given:

- The adjacent matrix
- The original List of cities
- The city source with is PARIS

Output: return the cities List of optimum combinations of the cities (that have the longest distance)

Algorithm:

This algorithm is an iterative function. The algorithm will return an approach of the optimum result (the longest path between cities).
We edit the List of cities: we delete "Paris" from the List, then we re-insert it in top and the bottom of the modified List of cities. Complexity O (n)

We create a List of cities (lets name it constructed) that contains only the source city (PARIS), then we search for the farthest city from PARIS, we add it to the List constructed. Complexity O (n) because we travel all cities to find the farthest one
Then we check for that new element the farthest city from it, we add it to constructed and then we check for the newest one, etc... We keep doing this until constructed contains the same number of cities that we had on the List of cities (token on given) Complexity O (n) because we travel all cities. So the result of this bloc is O(n^2);we travel the list of cities, then we search for each one the farthest one.

Finally we just connect the last city of constructed to PARIS, if it's possible and the distance isn't infinity we have a good approximation otherwise we aren't able to approach it with this method Complexity O (1).

Complexity discuss
The complete complexity is O(n) + O(n^2)+ O(1)=O(n^2)

Comparison between the 3 methods.

```
current result:
1 :PARIS position :(48.87;2.33)
5 :ANNECY position :(45.90;6.12)
4 :ANGERS position :(47.48;0.53)
2 :SAINT GEORGES position :(3.91;51.81)
7 :AUXERRE position :(47.80;3.57)
0 :AGEN position :(44.20;0.63)
6 :ARRAS position :(50.28;2.78)
3 :AIX EN PROVENCE position :(43.52;5.45)
1 :PARIS position :(48.87;2.33)
longestPath:17641.05 kilometres.
```

The exact method using 8 cities.

```
1 :PARIS position :(48.87;2.33)
2 :SAINT GEORGES position :(3.91;51.81)
0 :AGEN position :(44.20;0.63)
3 :AIX EN PROVENCE position :(43.52;5.45)
4 :ANGERS position :(47.48;0.53)
5 :ANNECY position :(45.90;6.12)
6 :ARRAS position :(50.28;2.78)
7 :AUXERRE position :(47.80;3.57)
1 :PARIS position :(48.87;2.33)
16700.55 Kilometres
```

The result using Lin Kernighan

```
1 :PARIS position :(48.87;2.33)
2 :SAINT GEORGES position :(3.91;51.81)
4 :ANGERS position :(47.48;0.53)
3 :AIX EN PROVENCE position :(43.52;5.45)
6 :ARRAS position :(50.28;2.78)
0 :AGEN position :(44.20;0.63)
5 :ANNECY position :(45.90;6.12)
7 :AUXERRE position :(47.80;3.57)
1 :PARIS position :(48.87;2.33)
17310.26 Kilometres
```

The result using local search

3. The purpose of the third question is to create a spanning tree with constraint that PARIS->SAINT GEORGES

Function Kruskal:
Given:

- The adjacent matrix
- The original List of cities
- Two cities constrained to be added to the tree

Output: display the minimum spanning tree, having the connection between the two cities constrained and the cost of this tree.

Algorithm:

This algorithm is an iterative function. The algorithm will display the tree with its cost.
We create a list of edges that contains all possible edges of the cities, and a dictionary than contains keys and values. The name of the city is the key on the dictionary, and the value is arbitrary (each city should have a different value), so we just choose to make it the key of the city. Complexity $O(n^2)$ to populate the edges List, and O(n) to create the dictionary. The result is $O(n^2)$ then.

As we have a constraint to add a certain edge (the edge between the two cities token on the given), we add it before we run the usual kruskal algorithm. We did it in the beginning to be sure that this edge is present in the spanning tree result. Complexity O(n) because we travel the List of cities to get the keys of the two cities.

The difficult part of this algorithm, is detecting when the edge added to result create a cycle. For this we use a simple method using the dictionary created previously. When we add an edge between city A and city B in the spanning tree, we edit the value of A in the dictionary to be equals to the value of B. Complexity O(n).

Let's take an example. Suppose we had cities A, B, C and D, the dictionary will be defined like : <A;0>,<B;1>,<C;2>,<D;3>. When we add the edge between A and B in the spanning tree, the dictionary will be modified to be <A;1>,<B;1>,<C;2>,<D;3>. So when we look to the dictionary, we see which cities are already connected and which are just single.

So logically, we cannot add edges which connect city A and city B, if the value of the city A and city B (in the dictionary) are equals. With this method, we can detect if an edge will create a cycle in the spanning tree or not but also we can see whenever the spanning tree is completed. When the dictionary contains same value for all keys, that means that all cities are already connected with each other and the tree is completed. The complexity of those instructions the complexity is $O(n^2)$ because the dictionary length is equals to n (number of cities) , so when we check if its completed or we just edited the worst case will alwaysbe O (n).To get the name of the city given its index we need another O(n) complexity to accomplish it. So the final complexity of this part is O (n^2).

Complexity discuss
The complete complexity is O(n*v) , the n is the number of cities, and v is the number of edges that are possible. Because we travel all the edges (O(v)) and then we modify the dictionary (the worst case is $O(n^2)$ like seen previously) so the complexity of my algorithm is $O(n^2*v)$. Normally in other languages it would be O(n*v) only.