Advanced Algorithms Project
(2018 - 2019)

The Traveling Salesman Problem (TSP)

Master CPS2/DSC/MLDM
Jean Monnet University
Saint Etienne

Group members:
Abdelhamid ALAOUI
Ezukwoke Ifeanyi KENNETH
Othmane BELLAGH
Walid OUCHTITI

Professors:
Amaury HABRARD
Léo GAUTHERON

Table of Contents:

1. Introduction:

1.1 Objective and problem statement

The objective of this project is to implement some algorithms to solve the Traveling Salesman Problem (TSP), to provide an experimental study of their running time and the quality of the solutions found.

The traveling salesman problem (TSP) is a known mathematics problem that asks for the most efficient possible trajectory (tour), given a set of points (cities) and distances, that must all be visited just once.

In computer science, this problem is an NP-complete problem that can be applied to the most efficient (less costly) route, considering a set of data to travel between various nodes (cities).

The inputs for the algorithms are in the form of a list containing different locations -nodes- and the distance information. Algorithms work on the process of identifying the optimal possible paths between the locations starting from a given city; this can be done through exhaustive search, the process of elimination or other methods.

1.2 Applications of the TSP

The traveling salesman problem has applications in identifying network or hardware optimization methods. For example, in the vastly complex global Internet, the traveling salesman problem can be used to work out the most efficient trajectories for data packets being routed anywhere in the system. The same holds true for designing telecommunications networks, computing DNA sequences and scheduling order-pickers in warehouses etc.…

Our project group implemented the following algorithms:

- The brute-force approach exploring all the possible solutions in a systematic way.
- The branch-and-bound version.
- The version adding and removing edges.
- The approximation based on the minimum spanning tree.
- The version considering at each step the nearest unvisited choice (called the greedy approach).
- The dynamic approach.
- The randomized approach

1.3 Guide to Programs execution

Programming language: **Java**
Development environment: **Eclipse IDE**
The project's structure:

- A **Main.java** file will allow running all the algorithms for a given single instance.
- All algorithms source codes are placed in the package **tsp_project** and can be called individually with its own main class.
- The interface **Constants** contains all the global variables for our programs:
    - **Number of cities** (we define the number of cities to use when we like to generate our data input randomly)
    - **Minimum and maximum distances** to set when we are using the random data generator, in order to define the distance interval between the cities.
    - **Starting point** : defining the index of the starting city from where the salesman will start his route and go back to at the end (ie: index 0 for C0 (city with index 0))
    - **Use database** , if this variable is set to the value 1 , it means that we are using the external files as an input data for the cities and distances.
    - **Database** variable , has the path of the experimental data file.

Program's execution:
The program can be called by running **Main.java** file or any othe Main file.

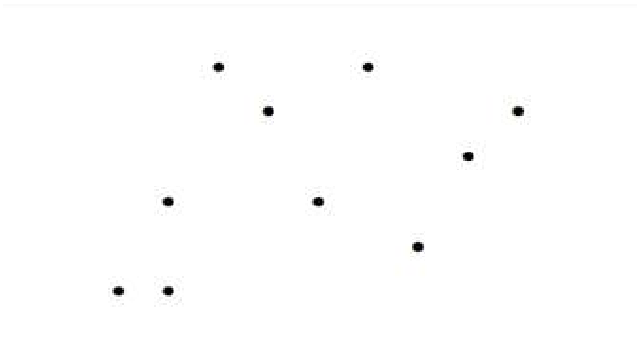
Input: Data set of cities and distances between each city, we can either:
1- Use a random generator of tsp instances
2- Parse a given **txt** file containing artificial data.

Output:

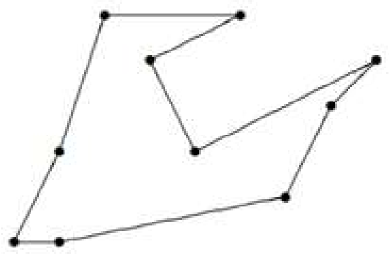
- The optimal paths for some algorithms and a reasonable path for the others.
- The cost of each resulted path (the total weight)
- Execution time of each algorithm.

2. A Traveling Salesman Problem instance

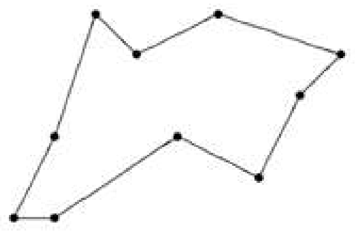
Let's take this example of ten-randomly placed cities.



The figure below is an example of a reasonable tour (path) ;



While this one is the optimal tour (path) for a given distances between the 10-cities



2.1 The Brute Force Approach

The brute-force search also known as exhaustive search, is a very general problem-solving approach that consists on systematically enumerating all the possible candidates for the solution and checking if each candidate satisfies the problem's constraints.

In our problem, the algorithm is supposed to examine all possible permutations of cities, and keep the one that –given a starting city- has the optimal cost (shortest).

How the algorithm works:

- Make a list of all possible Hamiltonian circuits
- Calculate the weight of each Hamiltonian circuit by summing the weights of its edges.
- Choose the Hamiltonian circuit with the least total cost.

The Brute-Force Algorithm is optimal: it guarantees finding the optimal solution. On the other hand, the algorithm is inefficient because it has to examine all **(n)!** Hamiltonian circuits, considering **n** the number of cities, which can be costly with a big amount of data.

Time complexity for this approach: O(n!) *// n: #vertices*
Space complexity for this approach: **O(n!)**
Simulation:

A simple demo with three cities A, B and C.
Let us assume that to go:
From A to B the weight is 2,
A to C the weight is 3,
B to A the weight is 9,
B to C the weight is 4,
C to A the weight is 2,
C to B the weight is 1.

The result after executing the program would be as follows:
The optimal path is: ABCA
The minimal cost to reach your destination is 8.0
Note that this is the optimal solution.

<u>2.2 The Branch and Bound Approach</u>

Brute Force approach and Branch and Bound are Similar, they both use stage based tree. The difference here is that the Brute Force approach is Breadth First Search (BFS), which means that it is costlier than the B&B which uses the Depth First Search (DFS).

Branch and Bound algorithm uses a tree search approach to enumerate all possible paths for a given instance of the problem, applying cutting rules to eliminate branches that cannot lead to a better solution to minimize our search space. In order to optimally solve the problem, Branch and Bound algorithm divides the whole set possible solutions –Routes- into exhaustive subsets. In every moment of its work, the algorithm stores the best found solution so far and its value called upper bound and the set of subsets not yet evaluated. Branch and Bound does not search these subsets to which it is sure that they do not contain a better solution than the best found so far, so it is much more efficient than an exhaustive search approach-Brute Force-.

If for a certain subset, the lower bound is equal or greater than the value of the best solution found so far, such subset is removed from the set of subsets not yet analyzed and will be no more considered. A good lower bound is a basic requirement for an efficient Branch and Bound minimization procedure.

Time complexity for this approach: O(n!) *// n: #vertices*
Space complexity for this approach: **O(n!)**

<u>Simulation:</u>
A simple demo with three cities A, B and C.
Let us assume that to go:
From A to B the weight is 390,
A to C the weight is 122,
B to A the weight is 239,
B to C the weight is 211,
C to A the weight is 221,

C to B the weight is 347.

The result after executing the program would be as follows:
The optimal path is: ACBA
The minimal cost to reach your destination is 708

Note that this is the optimal solution.

2.3 Greedy Approach (considering at each step the nearest unvisited choice)

The Greedy algorithm is an approach that seeks for the local optimal choice in each step of its execution in hope of finding the global optimum as a result.
In general, solving a particular problem using the greedy approach does not necessarily produce the optimal solution, nevertheless it gives a reasonable output in a short amount of time.

This approach requires much less computing resources, it is much faster to execute, its only disadvantage being that it does not always reach the global optimum.

Time complexity for this approach: O(n^2 x log(n)) *// n: #vertices*
Space complexity for this approach: **O(n)**
Simulation:
A simple demo with three cities A, B and C.
Let us assume that to go:
From A to B the weight is 390,
A to C the weight is 461,
B to A the weight is 188,
B to C the weight is 321,
C to A the weight is 305,
C to B the weight is 132.

The result after executing the program would be as follows:
The path is: ACBA
The cost to reach your destination is 437

Note that this solution may not be the optimal.

2.4 Dynamic programming

It is a technique for efficiently computing recurrences by storing partial results and re-using them when needed. It is well known that dynamic-programming recursions can be expressed as shortest-path problems in the case of travelling salesman.

This approach is supposed to outputs the optimal solution(the shortest possible path) for a given set of cities and distances between every city.

The implementation was proposed by Richard M. Karp [1962] in a publication titled '**A Dynamic Programming Approach to Sequencing Problems**'.

For a subset of cities $S \subseteq \{1,2,3, . . . , n\}$ that includes 1, and $k \in S$, let Cost(S, k) be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at k.

When $|S| > 1$, we define C(S, 1) = Inf since the path cannot both start and end at 1. Now, let's express C(S, k) in terms of smaller subproblems. We need to start at 1 and end at k; what should we pick as the second-to-last city? It has to be some $i \in S$, so the overall path length is the distance from 1 to i, namely, $C(S - \{k\},i)$, plus the length of the final edge, d ik . We must pick the best such i:

```
The subproblems are ordered by |S|
C({1}, 1) = 0
for s = 2 to n:
    for all subsets S ⊆ {1,..., n} of size s and containing 1:
        C({S}, 1) = Inf
        for all k ∈ S, k != 1:
            C({S}, 1) = min{C(S − {k},i) + dik : i ∈ S,i != k}
return min_k C({1,....,n}, k) + d_k1
```

Time complexity for this approach: O(n^2 x *2^n*)
Space complexity for this approach: **O(n^2)**
simulation:

2.5 Randomized approach

The randomized algorithm is an approach that is also used to solve the Travel Salesman Person. this approach is random based, which means that given N cities, the distances between each location and a starting point, the algorithm checks all the possible destinations (unvisited cities) and picks one randmally, which will be added to the visited list and can not be visited again, the algorithm iterate until having the complete tour. therefore, the nature of this algorithm is in some way greedy, which means that the space and time complexity for the randomized approach are as for the greedy approach.

Time complexity for this approach: O(n^2 x log(n))
Space complexity for this approach: **O(n)**
Simulation:
A simple demo with three cities A, B and C.
Let us assume that to go:
From A to B the weight is 390,
A to C the weight is 441,
B to A the weight is 188,
B to C the weight is 321,
C to A the weight is 305,

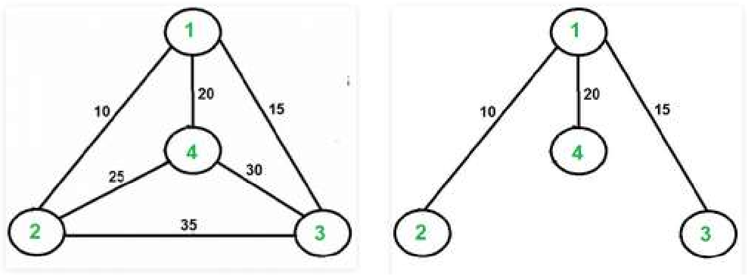
C to B the weight is 132.

The result after executing the program would be as follows:
The path is: ACBA
The cost to reach your destination is 417

Note that this solution may not be the optimal (in general).

2.6 The approximation based on the minimum spanning tree

The minimum spanning tree is an approach that is used to provide a reasonable (to approximate) solution that is close to the optimum solution (the shortest path). assuming that we have N cities, the distances between each location and a starting point, this approach is about to apply the Prim's algorithm or the Kruskal's algorithm to construct the minimal spanning tree *(it is a minimum cost tree that connects all vertices)* taking into account the starting point as the root. after constructing the MST, it lists the visited vertices in preorder walk of the constructed MST and add the starting city at the end of the path.



(source: geeks for geeks website)

Let *T* be this minimum spanning tree (in the right) and imagine performing a traversal of *T* and recording the list of vertices as we encounter them. This gives us a list of vertices in which the number of times a particular vertex *v* appears is equal to its degree, deg(*v*).

In the example, in order to cross all the vertices, the sequence must be as follows: S = 1-2-1-4-1-3, since we have to visit all the vertices just once, we remove all but the first occurence of each vertex in his sequence (except the starting point), which gives us the update of S = 1-2-4-3 and we add the starting vertex in the last which gives us the path as follows: S = 1-2-4-3-1.

The result of the algorithm is optimal in this case, but it may not produce the shortest path on all cases.

The cost of the best possible Travelling Salesman shortest path (the optimum) is never less than the cost of MST.

Time complexity for this approach: O((E+n)*log(n)) *where n: # vertices, E: # edges*
Space complexity for this approach: **O((E+n)*log(n))**

Simulation:

A simple demo with three cities A, B and C.
Let us assume that to go:
From A to B the weight is 451,
A to C the weight is 242,
B to A the weight is 232,
B to C the weight is 365,
C to A the weight is 75,
C to B the weight is 87.

The result after executing the program would be as follows:

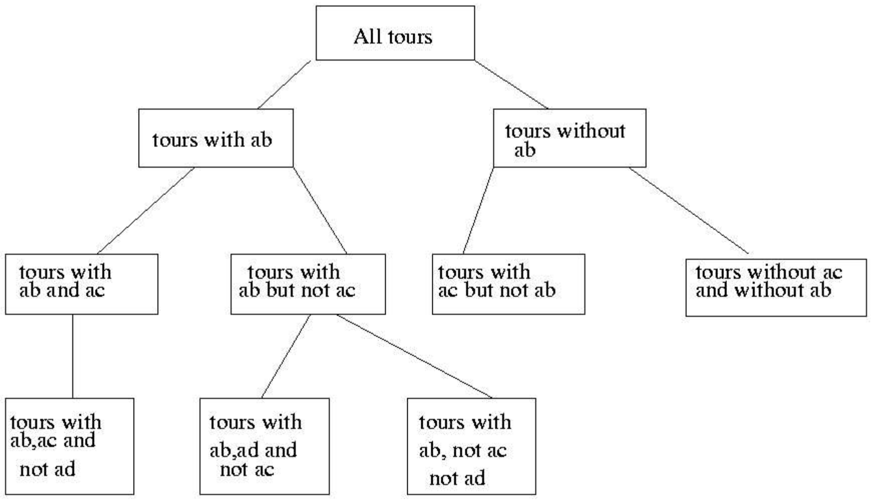
```
Edges    Weights
C - B     87
A - C     242

the path is : ACBA
the cost of this path is :328
```

Note that this solution may not be the optimal.

2.7 Adding and removing edges approach

The adding and removing edges approach is a technique that could be used to find the optimal solution for the travel salesman problem. the way this algorithm is working is in some ways similar to the branch and bound approach. the principle is as follows:
we consider a **binary** search tree where each node is an edge. the left child corresponds to take the edge in the tour, while the right child is a tour without this edge. The idea of this version is to chose the edges considered such that the solutions that might cost more are placed in the right child to be cut. (see example below)

Note: while setting edges to not be on the tour, at some point there will not be enough edges to finish the tour. Because of this, we require to determine if it is possible to do a tour given the list of remaining edges, and if not, to stop the exploration.


All tours
tours with ab
tours without ab
tours with ab and ac
tours with ab but not ac
tours with ac but not ab
tours without ac and without ab
tours with ab,ac and not ad
tours with ab,ad and not ac
tours with ab, not ac not ad

To implement this algorithm, we referred to the following article (http://lcm.csa.iisc.ernet.in/dsa/node187.html)

Time complexity for this approach: O(2^n)
Space complexity for this approach: O(2^n)

3. Planning

Date	Task	difficulties
November 5, 2018	analysis of the algorithms	New, recently learned approaches
November 9, 2018	Classification and task assignment	Problem to define realistic timeframes
November 10 -29, 2018	implementation, planning, meetings	different opinions between member of group, different coding methodologies
December 01, 2018	integration data, work with the given databases	merge all different methodologies, ensure correctness and required level of complexity
December 9, 2018	review	identify potential defects

During the beginning of this project, we had some struggles with the management, in terms of unrealistic time frames and tasks, and that was due to the lack of experience in project management. as a result, we ended up with inaccurate deadlines and wrong estimations, and a poor progress tracking, in order to find a solutions for this , we had to use a shared repository where we could communicate better our progress . In addition, during our work we experienced some lacks in project planning ; the main reason behind this was deficiency of effective communication, we tried to solve this issue with scheduling more meetings to keep track on the progress, help each other with the difficulties we faced and it helped us to improve productivity.

4. Conclusion

Finding a solution for the Traveling Salesman Problem is complex and time consuming especially when we are dealing with a large number of cities; we can adopt various approaches to solve it, either using algorithms that assure having the optimum , or others that will give us a reasonable solution depending on our constraints.

The greedy approach gives a reasonable solution but not exact, on the other hand it can be better in terms of complexity, comparably to other methods (branch and bound, brute force).

We can also conclude from our experiments that dynamic programming algorithm compared to the brute force algorithm could be very time saving.

After an experimental analysis of the computation of the TSP, The order of complexity of the different algorithms applied on different kind of data would yield:

O(approximation of greedy) < O(randomized approach) < O(MST approximation) < O(dynamic programming) < O(adding removing edges) < O(Branch and Bound) < O(Brute Force)

5. Reference

[1] Michael Held and Richard M. Karp, A Dynamic Programming Approach to Sequencing Problems. Journal for Society for Industrial and Applied Mathematics. 1962, Vol. 10, No. 1, pp. 196-210.