Traveling Salesman

Armande CIROT, Titouan POQUILLON, Jiri RUZICKA

January 20, 2020

1 Introduction

The travelling salesman problem is the study of the following question : "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"

We will try to answer this NP-hard problem by finding the shortest path visiting evert node of a non-oriented graph. This path is called a Hamiltonian path.

1.1 Exercise 1. Exact solution

We represent a weighted, non-oriented graph with an adjacency list. We also assume that a graph is always connected, i.e. that it is always possible to reach a node from any other node.

Number of Hamiltonian paths in a complete graph of size n Among a set N of n nodes, we can describe each Hamilton path as an ordered combinaison of all elements of N. It's a sequence of nodes. In a complete graph, all nodes are connected so every sequence n describes a Hamiltonian path.The number of possible sequences $A_n^n = n!$.

The number of Hamiltonian paths in a complete graph of size n is $n!$.

Enumerating all Hamiltonian paths starting from a node i in a graph of size n and returning the shortest one. We build a recursive function that returns the list of all existing Hamiltonian paths.

[1]:
```
import networkx as nx
```

[2]:
```
def HamiltonRecursiveLister(PathList_,Path,Graph,Currentnode):
    Path.append(Currentnode)#C=O(1)
    if len(Path)==Graph.number_of_nodes():#C=O(1)
        PathList_.append(Path.copy()) #C=O(n)
        del Path[-1]#C=O(1)
    else:
        for node in Graph.neighbors(Currentnode):
↪#C=O(n-1*C(HamiltonRecursiveLister(Graph of size n-1)))
            if node not in Path:
                PathList_=HamiltonRecursiveLister(PathList_,Path,Graph,node)␣
↪#C=O(HamiltonRecursiveLister(Graph of size n-1))
```

```
        del Path[-1]#C=O(1)


    return PathList_#C=O(1)
```

For a graph of size N, the time complexity is $N(N-1)! = N!$ because in the worst case, we have $(N-1)!$ HP. Our recursive function ends $(N-1)!$ times and each end have a complexity of N dues to the copy.

We build a test to check if our function works properly. One of its shortest Hamilton path is 1, 3, 4, 2 with a length of 3.

[3]:
```
import matplotlib.pyplot as plt
testG= nx.Graph()
testG.add_nodes_from([1,2,3,4])
testG.add_weighted_edges_from([(3, 1,1),(3,2,2),(4,2,1),(3,4,1)])
nx.draw_networkx(testG)
```

[4]:
```
print(HamiltonRecursiveLister([],[],testG,1),HamiltonRecursiveLister([],[],testG,2),
HamiltonRecursiveLister([],[],testG,3),HamiltonRecursiveLister([],[],testG,4))
```

```
[[1, 3, 2, 4], [1, 3, 4, 2]] [[2, 4, 3, 1]] [] [[4, 2, 3, 1]]
```

We can find mannualy that these are all the correct Hamiltonian paths.

We now build a recursive function that enumerates all Hamiltonian paths starting from a node i in a graph of size n, and returns the shortest one.

[5]:
```
#HamiltonRecursiveFinder. Takes in param a graph of size N and a node i
def HRF(Graph,Curentnode,BestPath=[],Bestlength=-1,Pathlength=0,CurentPath=[]):
    CurentPath.append(Curentnode) #C=O(1)

    #the end of our recursive function
    if len(CurentPath)==Graph.number_of_nodes(): #C=O(1) (we assume this number␣
 ↪is not calculated)
        if Bestlength>Pathlength or Bestlength==-1: #C=O(1)
            BestPath=CurentPath.copy() #C=O(N)
            Bestlength=Pathlength #C=O(1)
        del CurentPath[-1]

    else:
        for node in Graph.neighbors(Curentnode): #C=(n-1)*C(HRF(graph of size␣
 ↪n-1))
            if node not in CurentPath:#C=O(1)
                length=Pathlength+Graph[Curentnode][node]['weight']#C=O(1)
                if length < Bestlength  or Bestlength==-1:#C=O(1)
```

```
                    BestPath,␣
↪Bestlength=HRF(Graph,node,BestPath,Bestlength,length,CurentPath)#C=O(HRF(graph␣
↪of size n-1)
        del CurentPath[-1] #C=O(1)


    return BestPath, Bestlength
```

We test it for our graph with a known shortest Hamilton Path:

[6]:
```
print(HRF(testG,1)==([1, 3, 4, 2], 3))
```

```
True
```

It works.

For a graph of size N, the time complexity is $N(N-1)! = N!$ because in the worst case, we have $N-1!$ HP each shorter than the previous one so our recursive function end $(N-1)!$ times and each end have a complexity of N dues to the copy.

Solving the Traveling Salesman Problem We just need to apply our HRF function to every node of the graph and find the best solution.

[7]:
```
#Traveling Salesman Problem Solver C=O(n*n!)
def TSPS(Graph):#C=O(1)
    Path=[]#C=O(1)
    length=-1#C=O(1)
    for node in Graph.nodes(): #C=O(n*n!)
        P,d= HRF(Graph,node) #C=O(n!)
        if (d<length or length==-1)and d!=-1:
            Path=P
            length=d
    return Path,length
```

[8]:
```
TSPS(testG)
```

[8]:
```
([1, 3, 4, 2], 3)
```

1.2 Exercise 2. The Nearest Neighbor heuristic

Implementing the Nearest Neighbor heuristic

[9]:
```
def NNH(Graph,node):
    Path={node}#C=O(1)
    N=len(Graph)#C=O(1)
    while len(Path)<N:#C=O(n^2ln(n))
        successor=-1 #C=O(1)
        d=-1 #C=O(1)
```

```
        for nd in Graph.neighbors(node): #C=O(n ln(n))
            if nd not in Path and (Graph[nd][node]['weight']<d or d==-1):␣
↪#C=O(ln(n))
                successor =nd #C=O(1)
                d=Graph[nd][node]['weight'] #C=O(1)
        if successor==-1: return []
        node=successor #C=O(1)
        Path.add(node) #C=O(1)
    return Path
```

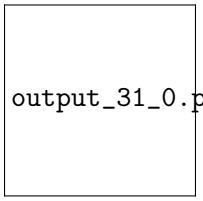
[10]:
```
NNH(testG,1)
```

[10]:
```
{1, 2, 3, 4}
```

Time complexity As we can see in our code's comments, we have a time complexity C = O(n^2).

Traping the NNH We build a graph that will trap the NNH. To do so, we ensure that the last edges that will take the NNH have an extremly high weight.

[11]:
```
trapG= nx.Graph()
trapG.add_nodes_from([1,2,3,4])
trapG.add_weighted_edges_from([(1, 2,1),(1,3,3),(2,3,2),(2,4,1),(3,4,8),(1,4,5)])
pos = nx.spring_layout(trapG)
nx.draw_networkx_nodes(trapG, pos)
nx.draw_networkx_labels(trapG, pos)
for edg in trapG.edges():
    nx.draw_networkx_edges(trapG, pos, edgelist=[edg],
                           width=2*trapG[edg[0]][edg[1]]["weight"])
```



[12]:
```
print("shortest Hamiltonian path : ",HRF(trapG,1),", solution with NNH:
↪",(NNH(trapG,1)))
```

```
shortest Hamiltonian path :  ([1, 3, 2, 4], 6) , solution with NNH: {1, 2, 3, 4}
```

We found a graph for which this heuristic does not compute the shortest Hamiltonian path.

1.3 Exercise 3. The Minimum Spanning Tree heuristic

A spanning tree of a graph G = (V, E) is a subset F of its edges such that (V, F) is a tree. A minimum spanning tree (MST) of a graph is a spanning tree that minimizes the sum of the weights of its edges.

Here is an algorithm that computes a minimum spanning tree (Prim's algorithm).

Data: Graph G = (V, E)

Result: (V, F) a MST

F ←

W ← x 0 an arbitrary element

while W != V do

```
Let (x, y)  E the shortest edge such that x  W and y / W

W ← W  y

F ← F  (x, y)
```

end

Result: (V, F)

Time complexity If we use a heap queue and the adjacency list of the graph of size n (and in the worst case, n), we can reduce the time complexity to $C = O(nln(n))$.

Proof : for each node of the queue ($n - 1$ nodes, because the root is not in the queue) we :

- get the closest one from graph (the one at the top of the queue) $C = O(1)$
- remove it from the queue $C = O(1)$
- update the distances from the graph for all remaining nodes in the queue. $C = O(n * ln(n))$
- sort the queue to have the new closest one from the graph at the top of the queue. $C = O(n * ln(n))$

So the final complexity is $C = O(nln(n))$

Proof that Prim's algorithm returns a spanning tree. We use a proof based on induction.

Mathematical induction:

Base case: W=x0, F=, is a tree: (WF) is Acyclic and Connected

Induction step: lets assume that the subgraph (W, F) is a tree, we show that (W U y,F (x, y)) is a tree, with (x, y) E the shortest edge such that x W and y / W

If we remove any edge e of F, (W,F) is disconnected (definition of a tree). It forms two connected subgraphs W1,W2. xW1 (resp W2). y/W1 , y/W2 and y is connected to W1 (resp W2). (W U y =W1 U y + W2 ,F (x, y) /e) is disconnected. If we remove the edge (x,y), there is, by construction,

no edge between y and W. (W U y,F) is disconnected. If we remove any edge from (W U y,F (x, y)), it is disconnected. Thus (W U y,F (x, y)) is a tree.

Conclusion: during each iteration of the while loop, the subgraph (W, F) is a tree.

At the end of Prim's algorithm (W, F) is a tree, V=W and FE. Prim's algorithm returns a spanning tree of Graph (V,E).

Prim's algorithm returns a MST Let assume that prim's algorithm returns a tree PT that is longer than the MST of a graph G (but because the first node does not matter for the Prims'algorithm, we assume they have the same root). They both have the same number of edges (fixed for a tree of n nodes). Lets walk through the nodes of PT and MST until we find different edges (they start from the same node). e1=(x-y) and e2=(x-y'). Let X and Y resp Y' be the disconnected tree of PT-{(x-y)} resp PT-{(x-y')} with x X and y / X ,y'/X , y Y and Y', y' Y' and Y. By construction of PT, e1 e2. Because both b and b' are not in A. X U Y' +{e1} is a ST of G: (it contains all nodes of G and if we remove any egde of X or Y', they are disconnected because they are trees and if we remove e1, X and Y' are disconnected so it is a tree). Moreover, the length of X U Y' +{e1} = length(MST)+e1-e2 length(MST). By definition of the MST, X U Y' +{e1} = length(MST) and X U Y' +{e1} is a MST.

Implementing Prim's algorithm We build two different methods, one with high time complexity, but we are sure it is working. The other one with lower time complexity (the same as suggested in the time complexity part). We use the first one to check if the second one works.

[13]:
```
import numpy as np
def Prims(Graph):
    V=set(Graph.nodes)
    edges=list(trapG.edges())
    sort=np.argsort([trapG[edg[0]][edg[1]]["weight"]for edg in edges]) #sort␣
 ↪edges by length.
    E=[edges[s] for s in sort]
    F=[]
    W={list(Graph.nodes())[0]}
    while len(W)<len(V):#C= O(n^3ln(n)) #for each edge we find the shortest edge␣
 ↪(x,y) such that x  W and y / W
        for edge in E:#C=O(len(edge)*ln(len(n))) = O(n^2ln(n))
            if (edge[0] not in W and edge[1] in W): #C=O(ln(len(W))) =O(ln(n))
                F.append(edge) #C=O(1)
                W.add(edge[0]) #C=O(1)
                break
            elif (edge[1]not in W and edge[0] in W):#C=O(ln(len(W)) =O(ln(n))
                F.append(edge)#C=O(1)
                W.add(edge[1])#C=O(1)
                break
    return (W,F)
```

[14]:
```
import math as m
import heapq as hp
```

```
def Prims2(Graph):

    #Initialisaton

    Node=1 #we assume that our graph hase one node called "1" as our root
    W=[list(Graph.nodes())[0]] #W is the list of nodes of our tree
    F=[] # F is the list of edges of our tree
    Hqueue=[] #Hqueue is the list of nodes not in our tree sorted by their␣
↪distance to it:
                      #The smallest weight of the edges that connect them to the␣
↪tree
                      # inf if the node is not connected to the tree

    # Initialisation of the Queue
    for node in Graph.nodes:
        if node != Node:
            if node in Graph[Node]:
                Hqueue.append((Graph[Node][node]["weight"],(node,Node)))
            else:
                Hqueue.append((m.inf,(node,Node)))
    hp.heapify(Hqueue) # C=O(n ln(n)) (sort algorithme)

    # Main loop of the algorithm.

    while len(Hqueue)!=0: #for every iteration a node leaves the queue so there␣
↪are n-1 iteration C(while)=O(n²ln(n))
        # For every iteration we pop the closest node to the graph (ie, the one␣
↪at the beggining of the Hqueue)
        Pop=hp.heappop(Hqueue) #C=O(1)
        edge=Pop[1] #C=O(1)
        Node=Pop[1][0] #C=O(1)
        F.append(edge) #C=O(1) We add the edge to F
        W.append(Node) #C=O(1) We then add the node to W

        #We then update the distances to Graph for each node:
        for i in range(len(Hqueue)): #n iteration max, C(for)=O(n ln(n)
            if Hqueue[i][1][0] in Graph[Node]: #search in Dict: C=O(ln(n))
                if(Graph[Node][Hqueue[i][1][0]]["weight"]<Hqueue[i][0]):
                    ␣
↪Hqueue[i]=(Graph[Node][Hqueue[i][1][0]]["weight"],(Hqueue[i][1][0],Node))␣
↪#C=O(1)
                                                                        ␣
↪#(Hqueue[i],Node) as one node in W, one out

        hp.heapify(Hqueue) # we sort the heapqueue C=O(n ln(n)) (sort algorithm)
    return W,F
```

[15]:
```
Prims(trapG), Prims2(trapG)
```

[15]:
```
(({1, 2, 3, 4}, [(1, 2), (2, 4), (2, 3)]),
 ([1, 2, 4, 3], [(2, 1), (4, 2), (3, 2)]))
```

Hamiltonian cycle deduced from the MST The triangular inequality is the following inequality: x, y, z V, w(x, z) w(x, y) + w(y, z), where w(x, y) is the weight of edge x $\rightarrow$ y (more direct paths are shorter). We assume that a graph verifies the triangle inequality.

1- A Hamiltonian cycle of the graph is a spanning tree of the graph. So by definition, the length of the MST is shorter or equal than the shortest Hamiltonian cycle SHP of the graph.

2- Let W be full walk (visiting all the nodes) of the MST. W as a length of at most twice the length of the MST. Because in the worst case, you have to go through every edge twice. W may not be a Hamilton path because of duplicats. (if it is, W=MST=SHP and the MST is the shortest Hamilton path (1-) and length(MST) 2 length(MST)) To build the Hamiltonian path from W, we read W node by node. Every time a node N is seen for the second time, we remove it from the walk. W(...-N-...-A-N-B-...) becomes W'(-N-...-A-B-...) Because of the triangular inequality, w(A-B) w(A-N) + w(N-B) , so the length of W' length of W 2 length MST .

When all duplicats have been removed, the path is an Hamiltonian path (every node is in the path exactly once) and its length is shorter than twice the length of the MST. Lets call this path HMST. Its the Hamiltonian cycle obtained by visiting the MST

because of (1-) we have: 2 length(MST) 2 length(SHP) because of (2-) we have: length(H) 2 length(MST) In conclusion, length(H) 2 length(S)

Assuming that the graph verifies the triangle inequality, the length of the Hamiltonian cycle obtained by visiting the MST is less than twice the length of the shortest Hamiltonian cycle of the graph.