Travelling salesman problem considering the weight of vertices

Kristie Harris
Rutgers University
Piscataway, NJ, USA
Email: kf.harris@rutgers.edu

Xuenan Wang
Rutgers University
Piscataway, NJ, USA
Email: xuenan.roderick.wang@rutgers.edu

Zhenyuan Zhang
Rutgers University
Piscataway, NJ, USA
Email: zzy.zhang@rutgers.edu

***Abstract*— In our daily life, we always try to find the optimal route to save time and energy. That is why GPS navigator is necessary when travelling. There is so-called classical travelling salesman problem (TSP) discussing how to find the optimal route that include all vertices once and finally goes back to the starting point. Sometimes, however, not only do we consider the distance cost, we might even care more about the energy cost coming from the mass that is carried. For example, a UPS driver needs to consider the total mass of packages on the truck as well as the distance of different routes. That is why in this paper we try to modify the TSP algorithm so that it can be applied on a model in which not only edges but also vertices themselves are weighted.**

I. Project Description

In this paper, we modify TSM algorithms for real life applications. The model used here is undirected, connected, weighted graph. All the edges are weighted positively(representing distance). Also vertices are weighted as well. The weight at each vertex could be positive or negative depending on different situations. In garbage collection, for example, the weight will be the mass of garbage collected at each location and is positve. For delivery men, on the other hand, the weight will then be the mass of packages delivered to each location and is negative. Our purpose is to find the optimal route to visit all vertices considering the weight of both vertices and edges. The main difference from original weighted graph is that, the weight of edges only adds into the cost when edges are visited. But the weight of vertices defined in this project will continuously raise the total cost during the rest of tour. This project can be used in real life to save the cost of garbage collection, package delivery, etc. This is a NP-Hard problem and it's not easy for un-highly-educated man to solve and that's the value of out project, which is help to solve this complicate problem by simply need user provide basic information like how far from A to B and what's the weight of P1. The outcome of this project will not need any basic of computer and simple typing skill will do.

The project has four stages: Gathering, Design, Infrastructure Implementation, and User Interface.

A. Stage1 - The Requirement Gathering Stage.

- The general system description: Package delivery has always been a tricky task. Delivery men's job is to deliver packages as soon as possible. At the meantime, their employers expect them to use as few resources like gasline and vehicle abrasion as possible so that they can actually make more money. This is very similar to the traveling salesman problem (TSP). The TSP originally ask 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?" Isn't that similar with the package delivery problem as we just mentioned? The difference of these two problems is that each city in TSP doesn't actually affect later cost, but in package delivery problem, it does. To understand the difference, we need to figure out what exactly is this package delivery problem we have been talking about. Assume a delivery man leaves from package warehouse and has 20 packages (P1 P20) with him. He needs to go to 7 places (A G) to deliver these packages. There are some roads (R1 Rn) he can choose and those roads are not the same length. Considering the delivery man needs to deliver these packages as soon as possible, what is his best way to chose the route? Or even more, if we consider the extra package weight will cause vehicle consuming more gas, and delivery man needs to make sure he can deliver packages fast and economical as well. What route is best for this scenario?
- The real world scenarios:
  - Scenario1 description: Package delivery arrangement and optimization.
  - System Data Input for Scenario1: Package weight (P1 P20), Road length (R1 Rn)
  - Input Data Types for Scenario1: Matrix
  - System Data Output for Scenario1: A functional road selection collection
  - Output Data Types for Scenario1: Array
  - Scenario2 description: Garbage collection arrangement and optimization.
  - System Data Input for Scenario2: Garbage weight (P1 P20), Road length (R1 Rn)
  - Input Data Types for Scenario2: Matrix
  - System Data Output for Scenario2: A functional road selection collection
  - Output Data Types for Scenario2: Array
- Project Time line and Divison of Labor. This project will be divided into 3 parts. First one is to try to build a

working map to simulate the delivery progress. Secondly, we need to try to figure out what is the best strategy to deliver packages without considering gas consumption. Last part, we take package weight into consideration and try to analyze data to come up with a functional resolution. Each part will take about one week to finish. For now, we are thinking Xuenan and Zhenyuan will be in charge of coding and algorithm design, Kristie will be in charge of report writing and also algorithm improvements. Both three of us will be participating in presentation preparation.

B. Stage2 - The Design Stage.

- Short Textual Project Description.
  Given a complete directed graph, we first select which algorithm to use depending on graph size and time complexity. Our input has weight on both edges and vertices. We can choose to consider distance on edges only or consider mass cost on vertices as well. After achieving the optimal path and its total cost, we could compare all three different methods and make analysis on the result.
- Flow Diagram.

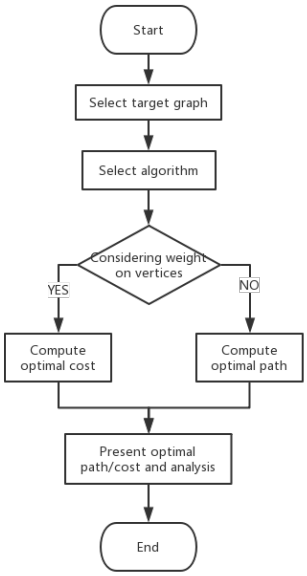

Fig. 1. System Flow Diagram

- High Level Pseudo Code System Description.
  Initially, a complete directed graph is given. It has weight on edges and vertices.(distance cost and mass cost)

  1. For small n, we can use the easiest method which is exhaustion:

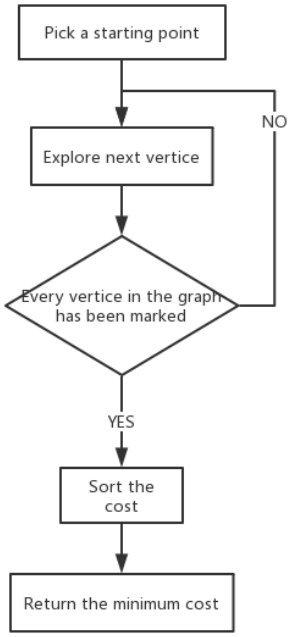

Fig. 2. Algorithm 1: Exhaustion

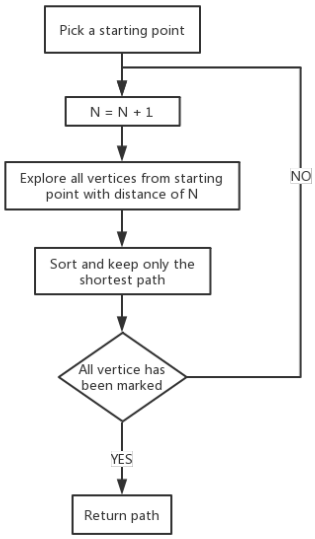

Fig. 3. Algorithm 2: Dynamic Programming

#Set the starting point
#Generate all permutations of visiting n vertices
#Keep track of the minimum cost considering weight of both edges and vertices
#Return the minimum cost and its path

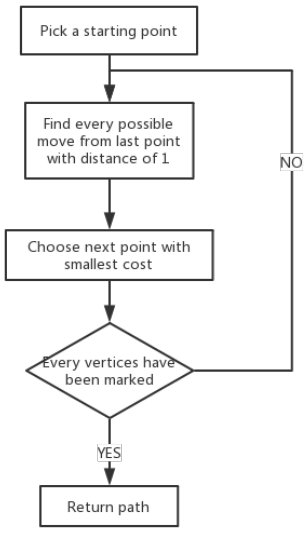

Fig. 4. Algorithm 3: Greedy Algorithm

This method has time and space complexity of O(n!) since there are n! permutations in total(That's why it only works for small n). This method will be used to verify our result using the other two methods.

2. A more realistic and effective way is dynamic programming:

#Set the starting point
#Start with visiting v=2 vertices in total.(1 starting point and 1 other point)
#Calculate the cost of each path and store the result.(All vertices visited already, the last vertex visited, total distance cost, total mass cost, mass carried)
#If more than one path share the same visited vertices and last vertex, only keep the one with minimum cost
#Continue with v=3 using the result of v=2
#Repeat the previous steps for v=4, 5, 6... until v=n
#The last step is adding the edge from the end point back to the starting point and find the minimum cost
#Return the minimum cost and its path

This method is much faster than the first one and it should give the correct final answer. But still it is not suitable for n that is too large.

3. An even faster method is greedy algorithm:

#Set the starting point and let end point to be the same point
#Among all the edges coming out of the end point, find the one with minimum total cost. Include that edge into the path and set the other node of the edge to be new end point
#Repeat the last step with the new end point until all vertices are visited
#Return the cost and the path

This method does not necessarily return the global minimum cost. But with some assumptions, it is still a good approximation and it is really fast!

- Algorithms and Data Structures.
  1. Brute-force approach: explore every possible permutation to find the best solution. Easy but time-consuming.
  Data Structures: Graph(Adjacency Matrix), Tree, Array

  2. Dynamic programming: Divide the problem into subproblems. Start with 1 point in the path and continue adding more points until all vertices are visited. Exponential time complexity.
  Data Structures: Graph(Adjacency Matrix), Cost Matrix, Binary integer, Array

  3. Approximate algorithm: Greedy algorithm always find local minimum cost. It might not give global best solution. But it is fast.
  Data Structures: Graph(Adjacency Matrix), Tree, Array
- Flow Diagram Major Constraints. Please insert here the integrity constraints:
  - Input Constraint. Input graph must be fully connected directed graph.
  - Distance Constraint. Distance must be positive integer.
  - Weight Constraint. Weight must be positive integer.
  - Output Constraint. Output must be an array of path.

C. Stage3 - The Implementation Stage.

We used python 3.7 language The deliverables for this stage include the following items:

- Sample small data snippet.
  Input:
  Use Matrix= [
  [0, 2, 0, 6, 1],
  [1, 20.4, 4, 4, 2],
  [5, 3, 200.2, 1, 5],
  [4, 7, 2, 100.1, 1],
  [2, 6, 3, 6, 80]
  ] as the graph we need.
  Here Matrix represents the graph of TSP. Matrix[i][j] is the distance between two places when i¡¿j and is the weight that needs to be carried when i==j

- Sample small output
  Output:
  ratio: 0
  optimal path: [1, 3, 4, 5, 2, 1]
  minimum cost: 9.0
  totalweight: 700.6999999999999

The result is correct!
And if we set a non-zero ratio, result becomes very different(depending on the value of ratio and the graph itself):
ratio: 0.1
optimal path: [1, 4, 5, 3, 2, 1]
minimum cost: 322.20000000000005
totalweight: 700.6999999999999

- Working code
ModifiedTSP(dynamic).py: see attachment
import sys
import copy
Matrix= [
[0, 4, 7, 12, 6],
[5, 20.4, 11.2, 14.7, 12],
[6.3, 9.9, 200.2, 7, 9.2],
[10, 11, 12.1, 100.1, 10.2],
[8.5, 16.2, 3.5, 20, 80]
]
For simplicity, here the Matrix is set to be 5x5 and is fixed. In usage, the Matrix could be larger and can be replaced
n=len(Matrix)
ratio = float(input("ratio: "))
state = []
new=[(1,[1],0.0,0.0)]
state.append(new)
end=0
visited=[]
cost=0.0
weight=0.0
point=list(range(1, n+1))
for j in range(0,n-1):
new=[]
state.append(new)
for k in range(len(state[j])):
end=state[j][k][0]
visited=state[j][k][1]
cost=state[j][k][2]
weight=state[j][k][3]
for nextend in point:
if nextend not in visited:
nextvisited=copy.deepcopy(visited)
nextcost=cost+Matrix[end-1][nextend-
1]*(1+ratio*weight)
nextweight=weight+Matrix[nextend-1][nextend-1]
nextvisited.append(nextend)
new=[nextend,nextvisited,nextcost,nextweight]
q=0
for m in range(len(state[j+1])):
if sorted(nextvisited) == sorted(state[j+1][m][1]) and nextend==state[j+1][m][0]:
q=1
if nextcost¡state[j+1][m][2]:
state[j+1][m][2]=nextcost
state[j+1][m][1]=nextvisited
if q==0:
state[j+1].append(new)
j=n-1
new=[]
state.append(new)
for k in range(len(state[j])):
end=state[j][k][0]
visited=state[j][k][1]
nextvisited=copy.deepcopy(visited)
cost=state[j][k][2]
weight=state[j][k][3]
nextvisited.append(1)
totalcost=cost+Matrix[end-1][0]*(1+ratio*weight)
new=[end,nextvisited,totalcost,weight]
state[j+1].append(new)
path=[]
Mincost=float("inf")
j=n
for k in range(len(state[j])):
visited=state[j][k][1]
cost=state[j][k][2]
weight=state[j][k][3]
if cost ¡Mincost:
Mincost=cost
path=visited
Totalweight=weight
print("optimal path:",path,' '"minimum cost:",Mincost,' '"totalweight:",Totalweight)

- Demo and sample findings
Matrix= [
[0, 2, 0, 6, 1],
[1, 20.4, 4, 4, 2],
[5, 3, 200.2, 1, 5],
[4, 7, 2, 100.1, 1],
[2, 6, 3, 6, 80]
]
This example is from online so the answer without diagonal terms is known: Tour 1-3-4-5-2-1 has lowest cost of 9.
In our case, if we set ratio to be 0, the problem becomes the same as traditional TSP, here is the result:
ratio: 0
optimal path: [1, 3, 4, 5, 2, 1]
minimum cost: 9.0
totalweight: 700.6999999999999
The result is correct!
And if we set a non-zero ratio, result becomes very different(depending on the value of ratio and the graph itself):
ratio: 0.1
optimal path: [1, 4, 5, 3, 2, 1]
minimum cost: 322.20000000000005
totalweight: 700.6999999999999
It makes sense because if we consider the cost coming from carrying weight, the optimal path will change.
Next step will be comparing the other 2 algorithms to

check our new result and see how the greedy algorithm can approximate the result in a faster time.

- Data size will be in the order of n!, n^2*2^n and n^2 for algorithm 1, 2 and 3.
- We noticed that in greedy algorithm, when considering the effect of weight, we could kind of make a prediction about how far in the future we need to carry the weight and find the 'effective' local best solution for the problem.

D. Stage4 - Result Analysis.

Our system does not have a complex user interface. So here we focus more on the anaysis of the performance of our system. First of all, our system is designed for finding the optimal route for package delivery. For example, one vehicle is responsible for n delivery points, each point having some mass to deliver. Also the distance between each two points is known. Our purpose is to find the best route to minimize the total cost.

In real use, Users only need to provide the mass at each point and all the distance for the system to construct a map. For simplicity, all these variables are randomized to demonstrate the system. The output will give the result of 3 different algorithms we use.(For different map size, users can choose different algorithm)

Here is one example: (n=4)
[[0. 3. 9. 9.]
[9. 0. 3. 2.]
[7. 3. 0. 4.]
[2. 5. 6. 0.]]
node_weight: [0, 0, 0, 0]
self.permutations_list: [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
cost_array: [12.0, 18.0, 16.0, 27.0, 24.0, 27.0]
*********Brute Force********
minimum cost without node weights: 12.0
optimized sequence is: [1, 2, 3]

---

node_weight: [0, 6, 3, 8]
self.permutations_list: [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
cost_array: [18.823529411764707, 23.352941176470587, 28.41176470588235, 41.05882352941177, 36.1764705882353, 40.23529411764706]
*********Brute Force********
minimum cost with node weights: 18.823529411764707
optimized sequence is: [1, 2, 3]

---

node_weight: [0, 0, 0, 0]
*********Dynamic********
minimum cost without node weights: 12.0
optimized sequence is: [1, 2, 3]

---

node_weight: [0, 6, 3, 8]
*********Dynamic********
minimum cost with node weights: 18.823529411764707
optimized sequence is: [1, 2, 3]

---

node_weight: [0, 0, 0, 0]
node_visited_weight: 0
node_visited_weight: 0
*********Greeedy********
minimum cost without node weights: 18.0
optimized sequence is: [1, 3, 2]

---

node_weight: [0, 6, 3, 8]
node_visited_weight: 6
node_visited_weight: 14
*********Greeedy********
minimum cost with node weights: 23.352941176470587
optimized sequence is: [1, 3, 2]

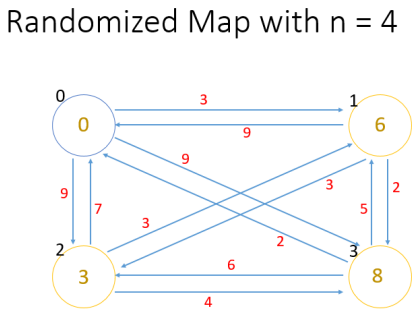

Fig. 5. Example with n=4

Our definition of total cost is: Sum of [distance + (carried mass/total mass)*distance]
For path [0, 1, 2, 3, 0], cost[0, 1]=3+3*17/17=6; cost[1, 2]=3+3*11/17=84/17; cost[2, 3]=4+4*8/17=100/17; cost[3, 0]=2. Total cost = 8+184/17 = 18.8235
As shown in the result, path [0, 1, 2, 3, 0] is the optimal path. Algorithm 1 and 2(Brute Force and Dynamic) always give the right answer while Greedy algorithm gives approximate answer.
The program is attached in the folder. Changing different n will randomize a new map with size n.
data with different n is stored in the data.txt.

Next, we will show the following:

- The approximation of Greedy algorithm
- The n dependence of time consumption for 3 algorithms
- Analysis of result

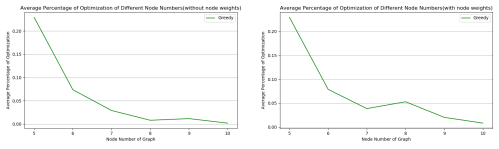

Fig. 6. Top percentage of greedy algorithm

It is very clear that the result of greedy algorithm is among the best 20 percent of all permutations and is getting better and better with larger n. I have to say this is not necessarily true in real use if distance varies a lot from edge to edge. However, considering the fast speed, greedy algorithm is a very good approximation method in our system.

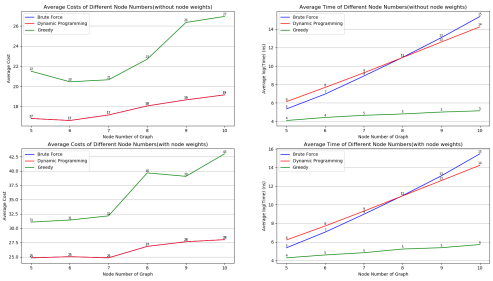

Fig. 7. Time consumption

Figure 7 shows that Algorithm 1 and 2(Brute Force and Dynamic) always give accurate answer and Algorithm 3 usually gives a higher total cost.
On the other hand, however, Algorithm 3 has the least time comsumption. The red linear curve in the 'Average Time' figure indicates the exponential time complexity of Dynamic algorithm. The blue curve grows even faster, agreeing with n! time complexity. The green curve is the best and is supposed to be polynomial time.

It is worth mentioning that for comparison, here we also show the result for not considering the weight on vertices. In that case, the problem will be exactly the same with traditional TSP problem. As can be seen in the data.txt, in most cases, the optimal path of traditional problem in TSP problem is not the best path for our new problem.
Also, in this example, we solve the problem of package delivery where the mass carried on the vehicle becomes less and less. The system could also work for garbage collection for example, where the mass carried will accumulate along the route.

In conclusion, our system provides a useful tool for users like delivery conpanies or garbage collection companies to plan a best route for their vihecles to save total cost. And it is easy to use, the only information we need is the distance and weight of goods for each position. Considering it is a NP-hard problem, users can also choose to use approximation algorithm when they have big sample size.