The Travelling Thief Problem: Solution

(Group 1)

Pranjal Mathur **1410110296**

Prerna **1410110306**

Saketh Vallakatla **1410110352**

Problem Statement:

Consider a thief who is raiding various cities each having some items with associated profit and weight. The thief has to visit every city exactly once and returns to the city he started from at the end of his raid.

The goal is to maximize the profits obtained by filling up the thief's knapsack, which has a maximum capacity W, while minimizing the total distance travelled, subject to the following constraints:

- The total number of cities are *n*, with matrix D of size n*n defining the direct distance between the two cities (i, j). B[i,j]=infinity, if a direct route doesn't exist between the two.
- Matrix M of size 2*m containing the price and weight of each jewel in each of n cities, where m is the total number of items and M[1,j] represents the weight of the jewel, while M[2,j] represents the corresponding weight.
- Matrix L of size n*x contains the list of items available in each of the cities, where n is the number of cities and x is the maximum number of items a city can have.

The goal of the problem is to maximize the profits obtained by filling up the thief's knapsack of weight W, while minimizing the total distance travelled.

Assumptions:

- The thief either takes the item completely or doesn't take it i.e the status of an object's inclusion in the knapsack is either 0 or 1, wherein 1 denotes he has placed the entire object in the knapsack and 0 denotes he hasn't taken any portion of the object.
- The thief takes the same amount of time in each city, irrespective of the number of items he picks up at the city.
- Similarly, the velocity of the thief remains constant throughout his raid.
- Each city may have 1 or more items and we have prior full knowledge of the price and weight of each item in each city
- The paths between cities are bidirectional and there are no obstacles whatsoever on the path between any cities.

Logic:

In order to achieve the goal, the problem is subdivided into finding:

a) The least cost Hamiltonian circuit, covering all the nodes(cities) exactly once while starting and finishing at the same node.

$$f(\underline{x}) = \sum_{i-1}^{n-1} (t_{xi,xi+1}) + t_{xi,xi+1}, \underline{x} = (x_1, x_2, x_3, \ldots . x_n)$$

Where $\underline{x}$ represents a tour, which contains all of the cities exactly once, $t_{xi,xi+1}$ is the time of the travel between $x_i$and x_{i+1} and it is calculated by:

$$t_{x_i,x_{i+1}} = \frac{d_{x_i,x_{i+1}}}{v_c}$$

where, f is the total time of the tour. The aim is to find x_i which minimizes f.

b) Set of elements which can be incorporated in the knapsack of weight W such that sum of the weights of this subset is smaller than or equal to W. In this problem, there are m items $I_{1,}$ $I_{2,}$ $I_{3,\ldots,}$ I_m, each of which have a value (pi) and a weight (wi)

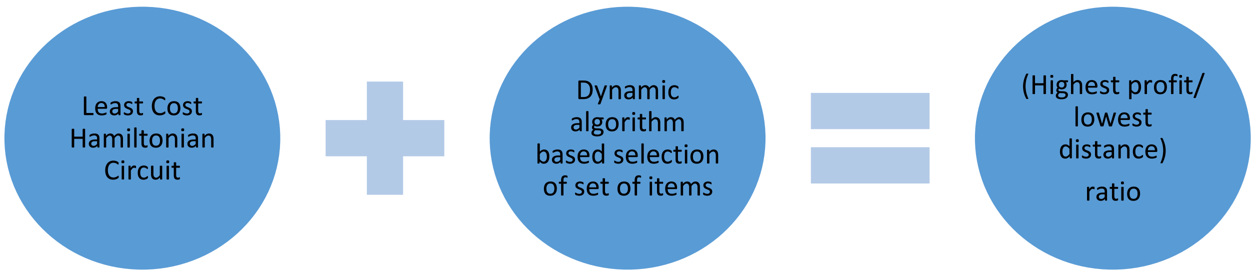
$$maximize\ g(\overline{y}) = \sum_{i=1}^{m} p_i y_i , \overline{y} = (y_1, \ldots, y_m)$$

We propose our hypothesis by considering the problem as an optimization problem, where we try to optimize both (a) and (b) by maximizing the following:

$$maximize\ h(np) = \frac{\underline{Total\ Profit(x)}}{\underline{Total\ Distance\ travelled\ (y)}}$$

where np is the profit per Km, x is the total distance travelled by the thief and y is the total profit.

Algorithm:


Least Cost Hamiltonian Circuit
Dynamic algorithm based selection of set of items
(Highest profit/ lowest distance) ratio

Hamiltonian path:

- Generate all the possible paths that visit every node exactly once, i.e., all Hamiltonian paths of the graphs.
- Calculate the traversal cost of each Hamiltonian path by summing up the edge cost of each edge in the path.
- The minimum cost path is chosen for the actual traversal.

Selection of items from each city:

- The structure of the sub-problem is defined, as per the concepts of Dynamic Programming, as the order of the items does not matter, we run a dynamic programming algorithm on the set of all the items combined into one pool. This gives us the best profit achievable.
- Next, we choose the path with minimum cost and it will be guaranteed that the items can be selected as suggested by the result of the previous step.

Testing on Data:

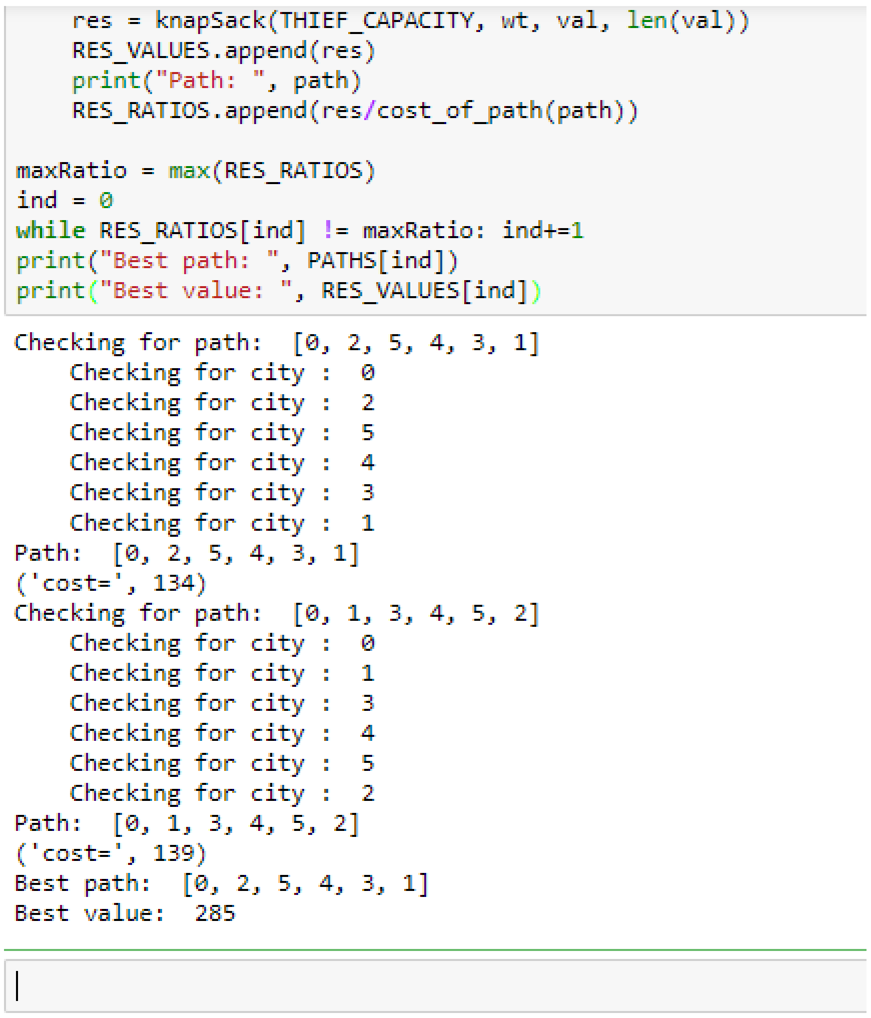
- As a part of our analysis, we used a randomly generated dataset which consisted of integers for edge weights of the path as well as the weight and cost of the individual items:

  Following are the attributes of each of the inputs to the problem:

  1. *B* (Matrix of distances between two cities)
     a. Range of values: 200-1000Km (if direct route), else inf
     b. Tested on n=10, 50 and 100 cities.
     c. Each city is uniquely identified by the index of the array.
  2. *M* (Matrix containing items weights and the corresponding profits)
     a. Range of values: Weight (2-10 Kg), Price (Rs. 1000- Rs. 10000)
     b. Tested for m=10, 50, 100, 180.
     c. Each item is identified by the column index of the array.
  3. *L* (Matrix containing set of items in each city)
     a. Range of values: 1-4 items/ city.
     b. Row numbers represents the City IDs, Items are represented their IDs, derived from array M.

- Actual distances between cities in India (http://www.mapofindia.com/distances/)

Screenshots of results:



```
    res = knapSack(THIEF_CAPACITY, wt, val, len(val))
    RES_VALUES.append(res)
    print("Path: ", path)
    RES_RATIOS.append(res/cost_of_path(path))

maxRatio = max(RES_RATIOS)
ind = 0
while RES_RATIOS[ind] != maxRatio: ind+=1
print("Best path: ", PATHS[ind])
print("Best value: ", RES_VALUES[ind])
```

```
Checking for path:  [0, 2, 5, 4, 3, 1]
    Checking for city :  0
    Checking for city :  2
    Checking for city :  5
    Checking for city :  4
    Checking for city :  3
    Checking for city :  1
Path:  [0, 2, 5, 4, 3, 1]
('cost=', 134)
Checking for path:  [0, 1, 3, 4, 5, 2]
    Checking for city :  0
    Checking for city :  1
    Checking for city :  3
    Checking for city :  4
    Checking for city :  5
    Checking for city :  2
Path:  [0, 1, 3, 4, 5, 2]
('cost=', 139)
Best path:  [0, 2, 5, 4, 3, 1]
Best value:  285
```

Analysis:

Based on our analysis of the problem, we propose the following hypothesis:

Hypothesis: The travelling thief is the most benefitted when the profit per unit of distance travelled is the highest, i.e. the ratio of **profit gained** and the **distance travelled** is the highest.

Here, we assume that the thief incurs an expense for travelling between cities and the end goal is to maximize the final profit which is the difference of the earnings from the items picked up and the expense incurred for travelling between the cities. From this, we inferred that the goal is to maximize the profit per unit distance.

To illustrate the same:

Case A: Let Rs. 1000 (p1) be the price of all the items picked from the cities and 100 Km(d1) be the distance travelled.

Case B: Let Rs. 1200 (p2) be the price of the items picked and 140(d2) be the distance travelled.

Comparing the cases: A and B based on our hypothesis, we found that: p1/d1 > p2/d2

Therefore, the profit earned per kilometer of distance travelled is higher in CASE A and is, therefore preferred by the thief.

However, before finding the optimal subset of items the thief can pickup, it is necessary to ensure that there exists a path/tour that visits all the cities exactly once and returns to the start node. Therefore, in the absence of such a path, the program returns null and terminates.

Conclusion:

Though the goal while defining the problem was to mimic the real-world as closely as possible, there were a few simplifications introduced later. Moreover, the travelling salesman problem, which is a constituent sub problem of the problem that we tackled, is an NP-Hard Problem.

Post the simplifications introduced and analysis of the constraints imposed, the ratio between the earnings from the stolen items and the distance travelled by chosen as the measure that was to be optimized, for this ratio quantifies the two key objectives: Maximize earnings from stolen items and Minimize distance travelled.

Though the proposed algorithm to solve the problem worked satisfactorily, it wasn’t tested out for massive datasets which would require large amounts of storage and processing capabilities for intermediate computations in both the Hamiltonian circuit finding logic as well as the Dynamic Programming logic for the selection of items.

Code:

```
# coding: utf-8

# In[130]:

THIEF_CAPACITY = 60
# city_capacity=[10,20,30,40,50,60]
city_values = [25,86,47,98,32,45]
city_weights  =  [[22,21,14,35],  [24,35,12],  [44,93,12],  [16,31,54,30],
[10,22,63,34,45], [22,33]]
city_values  =  [[50,42,11,33,],  [32,83,51],  [54,70,91],  [22,41,42,31],
[61,82,33,24,83], [25,30]]

edge_weight=[[0,10,12,0,0,0],[14,0,25,41,0,0],[4,9,0,6,0,7],[0,25,36,0,5,1
0],[0,0,0,45,0,78],[0,0,5,35,45,0]]
g=dict()
print((len(edge_weight)))
for i in range(0,len(edge_weight)):
   l=list()
   count=-1
   for eachWeight in edge_weight[i]:
       count+=1
       if(eachWeight>0):
           l.append(count)
   g[i]=l
print(g)

def cost_of_path(path):
   cost=0
   for i in range(0,len(path)-1):
#          print('city contains',city_capacity[path[i+1]])
#          print('edge cost=',edge_weight[path[i]][path[i+1]])
       cost=cost+(edge_weight[path[i]][path[i+1]])
   print(('cost=',cost))
   return cost

def find_all_paths(graph, start, end, path=[]):
   path = path + [start]
   if start == end:
       return [path]
   if start not in graph:
       return []
   paths = []
   for node in graph[start]:
       if node not in path:
           newpaths = find_all_paths(graph, node, end, path)
           for newpath in newpaths:
               paths.append(newpath)
   return paths

def findHamiltons(graph,start):
       temp_list_of_paths=[]
```

```
        paths = []
        for eachNode in graph[start]:
            temp_list_of_paths.append(find_all_paths(graph,start,eachNode))
        for eachPath_outer in enumerate(temp_list_of_paths):
            for eachPath in eachPath_outer[1]:
                if(len(eachPath)>len(edge_weight)-1):
                    print((eachPath))
                    paths.append(eachPath)
                    cost_of_path(eachPath)
        return paths



print(findHamiltons(g,0))
PATHS = findHamiltons(g, 0)


# In[131]:

# sack = []
def knapSack(W , wt , val , n):
   if n == 0 or W == 0 :
       return 0
   if (wt[n-1] > W):
       return knapSack(W , wt , val , n-1)

   else:
       return max(val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1),
                  knapSack(W , wt , val , n-1))
#          if kal == val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1):
#              pack.append(getID(n-1, cities))
#          else:
#              sack.append(knapSack(W , wt , val , n-1, pack))
#          return kal


# In[140]:

RES_VALUES = []
RES_RATIOS = []

for path in PATHS:
   wt = []
   val = []

   print("Checking for path: ", path)
   for i in range(len(path)):
       print("    Checking for city : ", path[i])
       for j in range(len(city_weights[path[i]])):
           wt.append(city_weights[path[i]][j])
           val.append(city_values[path[i]][j])
```

```
#     for i in path:
#         wt.append(city_capacity[i])
#         val.append(city_values[i])
#     print("THIEF CAPACITY = ", THIEF_CAPACITY)
   res = knapSack(THIEF_CAPACITY, wt, val, len(val))
   RES_VALUES.append(res)
   print("Path: ", path)
   RES_RATIOS.append(res/cost_of_path(path))

maxRatio = max(RES_RATIOS)
ind = 0
while RES_RATIOS[ind] != maxRatio: ind+=1
print("Best path: ", PATHS[ind])
```