Algorithm Design: Homework #1

Due on December 5, 2019 at 3:10pm

Professor ***Stefano Leonardi***

Luca Tomei **Andrea Aurizi**

Problem 1

To help Philip to find the optimal strategy and calculate his expected payoff it was decided to use *Dynamic Programming*, an algorithm design technique based on the division of the problem into subproblems and on the use of optimal substructures.
First of all we need to identify the variables of the problem, represented by N, K and C_i, where N is the maximum reward obtainable in the game, K are the number of boxes that can be opened and c_i is the cost that a player pay for opening the box. For to solve the problem, we have build $N \cdot K$ matrix with all zeroes and we have used a bottom-up *DP* approach initially with time complexity of $O(k \cdot n^2)$ and subsequently, through an optimization of the code, it was improved the complexity to $O(n \cdot k)$.

```
class FirstEx:
    def __init__(self, n, k, cost):
        self.n = n
        self.k = k
        self.cost = cost
        self.matrix = [[0 for i in range(k)]  for j in range(n+1)]
        self.res = self.summation = self.diff = self.prevdiff = 0
        self.matrix1 = self.E1_a()
        self.matrix2 = self.E1_b()

    def E1_a(self): # O(N * K)
        for i in range(self.k-1, -1, -1):
            ex_val = 0
            for p in range(0, self.n+1):
                ex_val = 0
                for j in range(0, self.n+1):
                    if j < p:
                        if i == self.k-1: ex_val += p/(self.n+1)
                        else: ex_val += (self.matrix[p][i+1])/(self.n+1)
                    else:
                        if i == self.k-1: ex_val += j/(self.n+1)
                        else: ex_val += (self.matrix[j][i+1])/(self.n+1)
                ex_val -= self.cost[i]
                self.matrix[p][i] = round(max(p - self.cost[i], ex_val),4)
        return self.matrix
```

To calculate the value of an optimal solution for the problem instance, it was decided to write the Bellman equation associated with the dynamic programming solution:

$$OPT(i,j) = \begin{cases} -c_i + (j+1) \cdot \frac{j}{n+1} + \sum_{x=j+1}^{n} \frac{x}{n+1} & \text{if } i = k-1 \\ \max(-c_i + j, -c_i + (j+1) \cdot \frac{OPT(i+1,j)}{n+1} + \sum_{x=j+1}^{n} \frac{OPT(i+1,x)}{n+1}) & \text{otherwise} \end{cases}$$

```
def E1_b(self): #   O(N * K)
        for i in range(self.k-1, -1, -1):
            s = self.summation/(self.n+1)
            self.summation = self.diff = 0
            self.prevdiff = 0
            for p in range(0, self.n+1):
                if i == self.k-1:
                    if p == 0: ex_val = ((self.n*(self.n+1))/2)/(self.n+1)
                    else: ex_val = self.matrix[p-1][i] + (p/(self.n+1)) + self.cost[i]
                else:
                    if p == 0: ex_val = s
                    else:
                        self.diff = (self.matrix[p][i+1]-self.matrix[p-1][i+1])*p + self.prevdiff
                        self.prevdiff = self.diff
                        ex_val = s + (self.diff)/(self.n+1)
                ex_val -= self.cost[i]
                self.matrix[p][i] = round(max(p - self.cost[i] , ex_val),4)
                self.summation += self.matrix[p][i]
        return self.matrix
```

At each iteration the algorithm must be able to decide whether it is better to accept the current gain or try its luck by opening the subsequent boxes by comparing i, the current gain, and the sum calculated with the previous points which represents the expected gain.Arriving at the penultimate box, you need to make a maximum transaction between the two, which is the gain before opening the current case, and the subsequent ones, which is the gain you expect by opening this case and the next.
Applying this reasoning recursively for all the columns of the matrix, in the case of the case $k-1$th the maximum value i will be greater than the sum calculated with the previous points. At this point, instead of going ahead and continuing to open the boxes, it is advisable to keep the current value and not pay the cost of the $k-1$ box.
To visualize the result of the problem, it will be enough to print on screen the content of $matrix[0][0]$.

Problem 2

The second exercise requires the implementation of an algorithm that having an MST in input and returns in output the relative minimum complete graph having as only MST that in input.
For the implementation of the algorithm, it is necessary to draw inspiration from the Kruskal algorithm and the cutting property of a graph. The choice of structure to manage clouds is the list.

The total cost of the algorithm is $O(|E| + |V| \cdot log|V|) = O(|V|^2)$. In fact the dominant operation concerns the creation and consequently also the insertion of the weights of the edges in order to build the complete graph, therefore knowing that:

$$|E| = \frac{|V| \cdot (|V| - 1)}{2} \tilde{=} O(|V|^2). \tag{1}$$

```
import networkx as nx, matplotlib.pyplot as plt
from networkx.algorithms import tree

G = nx.Graph()
tupleList = [(0, 2, 49), (0,4,43), (1, 3, 31), (1, 2, 56)]
for t in tupleList: G.add_edge(t[0],t[1],weight=t[2])

edgelist = list(tree.minimum_spanning_edges(G, algorithm=’kruskal’, data=True))
G_complete=nx.complete_graph(5)
for i in edgelist:  G_complete[int(i[0])][int(i[1])][’weight’]=i[2].get("weight")

nodi=G.nodes
lista_nodi = [];
for nodo in nodi: lista_nodi.append(nodo)

lista_nuvole = [];
for nodo in lista_nodi:
    nuvola = []
    nuvola.append(nodo)
    lista_nuvole.append(list(nuvola))

for i in edgelist:
    node1=int(i[0])
    node2=int(i[1])
    nuvola1 = nuvola2 = []
    for nuv in lista_nuvole:
        if node1 in nuv :   nuvola1 = nuv
        if node2 in nuv :   nuvola2 = nuv
    peso= i[2].get("weight")
    for n1 in nuvola1:
        for n2 in nuvola2:
            if(not(n1 == node1 and n2 == node2)) :
                peso1=G_complete.get_edge_data(n1,n2).get("weight")
                if(not peso1):
                    NewGraph.add_edge(n1, n2, weight=peso+1)
                    G_complete[n1][n2][’weight’]=peso+1
                else:   NewGraph.add_edge(n1, n2, weight=peso)
    nuvola3 = nuvola1.extend(nuvola2)
    nuvola1 = nuvola3
    lista_nuvole.remove(nuvola2)
```

The sum of the weight of the edges of the complete graph is obtained by scrolling the arc list through an instruction for adding to a counter the weight of each arc purchased through the get function.

```
summ=0
for i in G_complete.edges:  summ += G_complete.get_edge_data(i[0],i[1]).get("weight")
edgelist = list(tree.minimum_spanning_edges(G_complete, algorithm=’kruskal’, data=True))
```

Problem 3

The problem can be represented with a Graph G in which there are:

- One Source, represented by Federico
- One Sink, that is the arrival node of the network

For every friend of Federico's, there will be two nodes among which there will be an edge with a capacity equal to the number of chocolates n_f that a friend can handle.
Considering two friends, all the nodes must go from the second node of the first friend to the first node of the second friend and the capacity of the edge must always be n_f, with f the number of the friend of arrival.
The Graph is build in this way:

1. Create Source node "Federico" and one sink node "Clients".
2. For each friend $f_1 \in F$ create two regular nodes f_{iA} and f_{iB}, then add an edge from f_{iA} to f_{iB} with capacity n_i.
3. For each friend f_i that Federico will meet in person add an edge from the source "Federico" to f_{iA} with capacity ∞.
4. For each couple of friends (f_i, f_j) that see each other regularly add two edges from f_{iB} to f_{jA} and from f_{jB} to f_{iA} with capacity ∞.
5. For each friend $f_i \in F$ add an edge from f_{iB} to the sink "Clients" with capacity s_i.

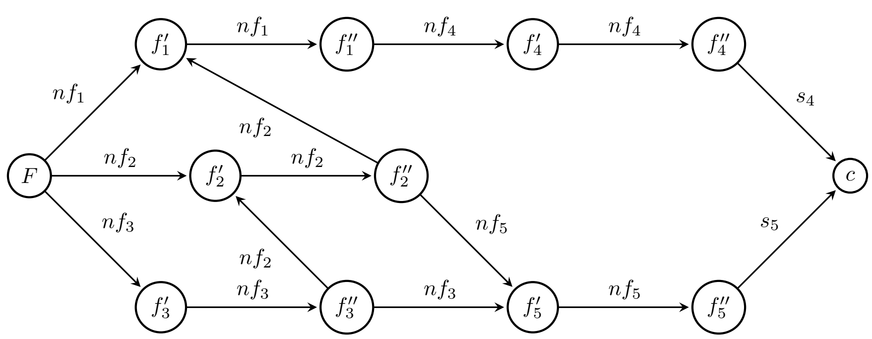

Figure 1: **Example with** $|F| = 5$

....

In the second part of the exercise, are added some restritions that limit the weekly quantity of chocolate sold there in c_B.
To deal with this problem, some changes will be made to the main model adding a node to each building.
For each building, a node with only one outgoing edge of capacity c_B is added to the sink and all other friends who sell associated with this building have an edge towards the node associated with that building of capacity s_f. To add the buildings to the problem formulation just add this definition to the previous graph and to find the number of chocolates to sell, you can use the *Ford Fulkerson* algorithm, which is a max_flow algorithm that allows to find the maximum flow that crosses a graph from one point to another of this.

For the second part of this exercise points 1-4 must be repeated and:

5. For each building in B, create a regular node b_i and an edge from it to the sink "Clients" with capacity c_{b_i}.
6. For each friend $f_i \in F$ create an edge from f_{iB} to the node b_x, that represent his associated building, with capacity s_i.

Problem 4

Manca solo lui