Algorithm Design - Homework 1

Sapienza University of Rome

Marco Costa, 1691388

December 5, 2019

Exercise 1

Problem

The problem consists in finding the expected optimal reward playing the game "*Open the Boxes and keep the Best!*". The input is composed by the number of boxes k, the number of different rewards n an the cost c_i of every box. Two algorithms are required. The first one's complexity must be $\boldsymbol{O}(n^2 \cdot k)$, while the second one's complexity must be $\boldsymbol{O}(n \cdot k)$

Solution

In order to solve the problem, we can define a matrix M composed of k rows and n columns, where the element $M[i, j]$ represents the expected value by opening the box i and having as current maximum reward j.
To populate the matrix we can use **backward induction**, proceeding first considering the last box and choosing what to do for all the possible rewards. Using this information, we can then determine what to do at the penultimate box. This process continues backwards until we have determined the best action for every possible reward for each box. So the element $\mathbf{M}[\mathbf{0}, \mathbf{0}]$ is the expected optimal reward.
Let's explain the mathematical formula:
- $j \cdot \frac{j+1}{n+1}$ (or $M[i+1, j] \cdot \frac{j+1}{n+1}$) is the probability of obtaining a reward $r \leq j$;
- $\frac{1}{n+1} \sum_{h=j+1}^{n} h$ (or $\frac{1}{n+1} \sum_{h=j+1}^{n} M[i+1, j]$) is the probability of obtaining a reward $r > j$;
- $c[i]$ is the cost to pay opening the i^{th} box.
- the sum of the previous values is the expected value opening the box i having as current maximum reward j

To improve the complexity to $\boldsymbol{O}(n \cdot k)$ we can eliminate the summations, since in the worst case we have a sum over n ($\boldsymbol{O}(n)$) in nested cycles ($\boldsymbol{O}(n \cdot k)$) having a total cost of $\boldsymbol{O}(n^2 \cdot k)$. To do this we must distinguish the two cases:

- ***Last box***: we can remove this summation using a mathematical trick: $\sum_{i=j+1}^{n} i = \sum_{i=0}^{n} i - \sum_{i=0}^{j} i = \frac{n \cdot (n+1)}{2} - \frac{j \cdot (j+1)}{2} = \frac{n \cdot (n+1) - j \cdot (j+1)}{2}$;
- ***Other boxes***: we can use an auxiliary vector in which we keep track of the sum of the elements of the matrix related to the box next to the current one.

Here there are the two ***Algorithms***:

Algorithm 1 Get optimal expected value ($\boldsymbol{O}(n^2 \cdot k)$)

$M[k, n+1] \leftarrow 0$
for i in $(k-1, ..., 0)$ **do**
  for j in $(n, ..., 0)$ **do**
    $r \leftarrow 0$
    if $i = k-1$ **then**
      $r \leftarrow j \cdot \frac{j+1}{n+1} + \frac{1}{n+1} \sum_{h=j+1}^{n} h - c[i]$
    else
      $r \leftarrow M[i+1, j] \cdot \frac{j+1}{n+1} + \frac{1}{n+1} \cdot \sum_{h=j+1}^{n} M[i+1, h] - c[i]$
    $M[i, j] \leftarrow \max(j, r)$
return $M[0, 0]$

Algorithm 2 Get optimal expected value ($\boldsymbol{O}(n \cdot k)$)

$M[k, n+1] \leftarrow 0$
$sums[n+1] \leftarrow 0$
for i in $(k-1, ..., 0)$ **do**
  for j in $(n, ..., 0)$ **do**
    $r \leftarrow 0$
    if $i = k-1$ **then**
      $r \leftarrow j \cdot \frac{j+1}{n+1} + \frac{1}{n+1} \cdot \frac{n \cdot (n+1) - j \cdot (j+1)}{2} - c[i]$
    else
      $r \leftarrow M[i+1, j] \cdot \frac{j+1}{n+1} + \frac{sums[j]}{n+1} - c[i]$
    $M[i, j] \leftarrow \max(j, r)$
    if $j = n$ **then**
      $sums[j] \leftarrow 0$
    else
      $sums[j] \leftarrow sums[j+1] + M[i, j+1]$
return $M[0, 0]$

Exercise 2

First problem

The problem is to find the complete graph $\boldsymbol{G}$ of minimum weight given a weighted tree $\boldsymbol{T}$, such that $\boldsymbol{T}$ is the unique minimum spanning tree of $\boldsymbol{G}$. Algorithm's run time must be polynomial in n.

Solution

Insert edges that are not in the tree so as to obtain the complete graph. These edges must have a greater weight than those of the tree, so that $\boldsymbol{T}$ is the only $\boldsymbol{MST}$ of $\boldsymbol{G}$. Since *Kruskal*'s algorithm uses the *Union-Find* structures for representing the cuts, we use this structures to solve the problem. Let's define with $\boldsymbol{V}$ the set of nodes of $\boldsymbol{T}$ and with $\boldsymbol{E}$ the set of edges of $\boldsymbol{T}$.

Algorithm 3 Find complete graph

```
1:  for v ∈ V do                                                    (O(n))
2:      v.initializeUnionFindSingleton()
3:  T.sortEdgesByAscendingWeights()                                 (O(n log n))
4:  G ← ∅
5:  for e ← (u, v) ∈ E do                                           (O(n^3))
6:      G.addEdge(e)
7:      set_1 ← u.findComponents()
8:      set_2 ← v.findComponents()
9:      for u ∈ set_1 do
10:         for v ∈ set_2 do
11:             ê ← (u, v)
12:             if ê ∉ G then
13:                 ê.setWeight(e.getWeight() + 1)
14:                 G.addEdge(ê)
15:     union(set_1, set_2)
16: return G
```

Cost: $\boldsymbol{O}(|V|\log|V| + |E|\cdot|V|^2) = \boldsymbol{O}(n\log n + (n-1)\cdot n^2) = \boldsymbol{O}(n\log n + n^3 - n^2) = \boldsymbol{O}(n^3)$, given by the three nested cycles over n.

Second problem

Find the total weight of the complete graph $\boldsymbol{G}$ of minimum weight given a weighted tree $\boldsymbol{T}$, such that $\boldsymbol{T}$ is the unique minimum spanning tree of $\boldsymbol{G}$. Algorithm's complexity must be $\boldsymbol{O}(n\log n)$.

Solution

This problem is quite similar to the previous one, but in this case there is no need to create all the edges of the graph.

Algorithm 4 Find weight of the complete graph

```
1:  for v ∈ V do                                                    (O(n))
2:      v.initializeUnionFindSingleton()
3:  T.sortEdgesByAscendingWeights()                                 (O(n log n))
4:  w_total ← 0
5:  for e ← (u, v) ∈ E do                                           (O(n log n))
6:      w_total += e.getWeight()
7:      set_1 ← u.findComponents()
8:      set_2 ← v.findComponents()
9:      w_total += (set_1.size() × set_2.size() − 1) × (e.getWeight() + 1)
10:     union(set_1, set_2)
11: return w_total
```

Cost: $\boldsymbol{O}(|V|\log|V| + |E|\log|V|) = \boldsymbol{O}(n\log n + n\log n) = \boldsymbol{O}(n\log n)$ (*union* costs $\boldsymbol{O}(\log n)$).

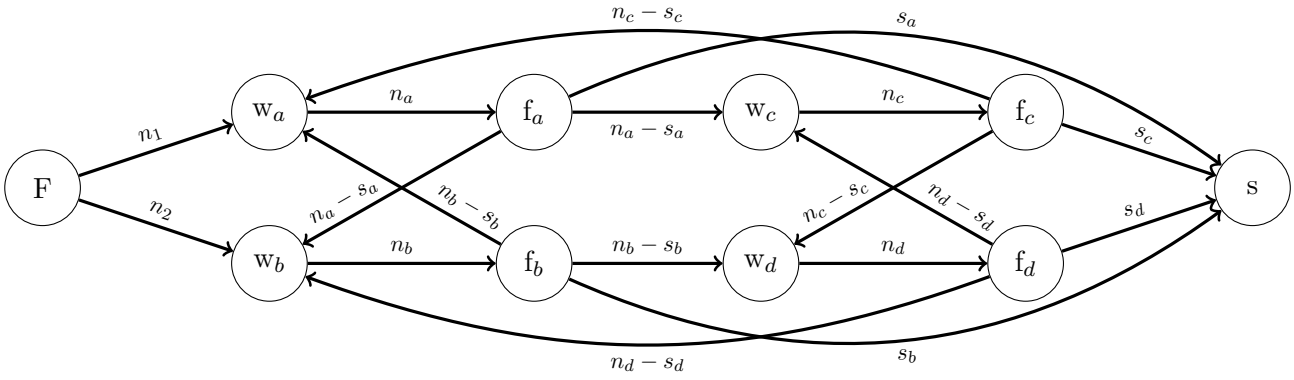
Exercise 3

First Problem

The problem consists in modeling Federico's business as a flow problem in a graph $\boldsymbol{G}$, only consisting of regular vertices, one source, one sink, and capacitated edges to find out how many chocolates he should make every week. An example with $|F| = 4$ is required.

Solution

The model consists in a source node F that represents Federico, a sink node S that represents the costumers and different nodes f_i representing Federico's friends. Moreover, for each friend there is a node w_i that represents the friend's "warehouse". These nodes help us better understand that the amount of total chocolates received by a friend is exactly n_i. Each friend can receive the chocolate from Federico (represented with an edge of path from Federico to the friend and capacity n_i) or from another friend (represented with an edge of path from the new friend to the first one and capacity $n_i - s_i$). Notice that when a friend meets another, they can exchange chocolates for each other (represented with two edges: the first one has the path from the first friend to the warehouse of the second one and capacity $n_{i1} - s_{i1}$, while the second one has the path from the second friend to the first one and capacity $n_{i2} - s_{i2}$). The chocolates sold by each friend is represented with an edge having the path from the friend to the costumers and capacity s_i. Notice that for each friend $s_i \leq n_i$ and the total amount of chocolates sold is $\sum_{i \in |F|} s_i$. It's possible to find the *Maximum Flow* in order to find out how many chocolates Federico should actually make every week. Let's show the example with $|F| = 4$:

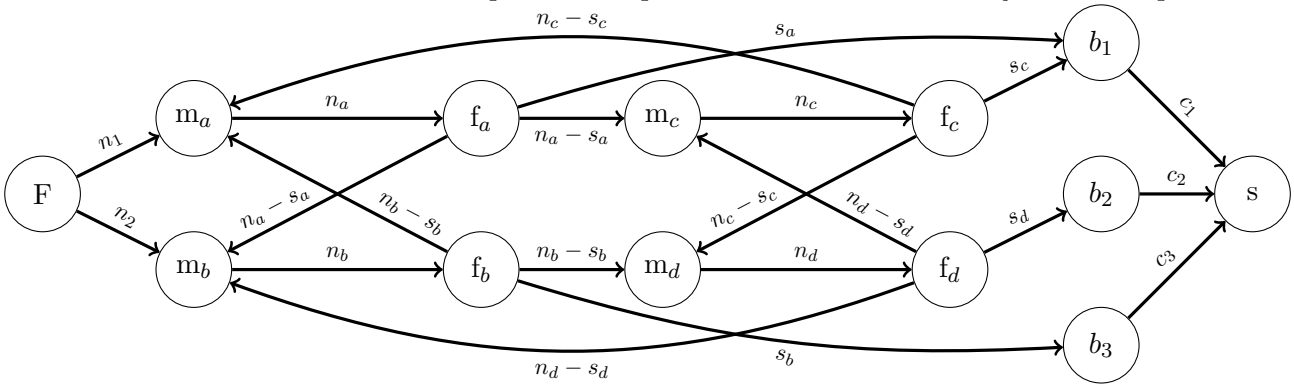


Second Problem

Adjust the network knowing that each friend will have an associated building $b_f \in B$ and find out if and how this impacts Federicos business.

Solution

In this case we must modify a bit the previous model adding some new nodes b_i that represent the buildings. Now each friend can sell chocolates only in his associated building, so we have the edges with capacity s_i with path from the friend to his associated building, and another edge with capacity c_i with path from the building to the costumers. The main difference is that now the total amount of chocolate sold is $\sum_{i \in B} c_i$, and we have that $\sum_{i \in |B|} c_i \leq \sum_{i \in |F|} s_i$. It means that Federico could make less chocolates than before because of the possible drop of sales. Let's show the adjusted example:



Exercise 4

Problem

The problem consists in showing that it is not possible to solve the problem of scheduling n tasks with quick algorithm. Each job j of the n jobs has an earliest time s_j to start task, a deadline d_j when it has to be finished, and the length $l_j \in \mathbb{N}$ that specifies the time needed to complete it. There is also an upper bound time k and $\sum_{j \in J} l_j \leq k$.

Solution

In order to prove that this problem is not solvable with quick algorithm we need to prove that it is ***NP-Hard***. We can do it reducing another *NP-Hard* problem to the current problem. Let's define our problem as Θ, we need to find a well known *NP-Hard* problem Ω such that:

- $\omega \in \Omega \implies \theta \in \Theta$
- $\theta \in \Theta \implies \omega \in \Omega$ (or equivalently $\neg\omega \in \Omega \implies \neg\theta \in \Theta$)

Observation: The earliest start time s_j indicate only at what time the job j is available (it does not need to be done at that moment, but it does have to be done before its deadline d_j).

Observation: Our scheduling problem is solvable if and only if each job j meets its deadline d_j and $\sum_{j \in J} l_j \leq k$ (we know that this last is satisfied).

We can use ***Subset Sum*** for our purpose (Ω = *Subset Sum*).
Subset sum: Given a set of positive integers and an integer k, is there any non-empty subset whose sum to k.

1. Prove that $\omega \in \Omega \implies \theta \in \Theta$:
   Let's suppose we start with a solvable instance of *Subset Sum* Problem. Given $X = \{x_1, ..., x_n\}$ a set of positive integers to which there is a subset X' such that its elements add up to k and the sum of the whole set is λ ($\sum_{x_i \in X'} x_i = k$ and $\sum_{x_i \in X} x_i = \lambda$). Let's define jobs $\{j_1, ..., j_n\}$ such that $s_i = 0$, $l_i = x_i$ and $d_i = \lambda + 1$ for every j_i (earliest start time, duration of the job and deadline respectively). This instance solves the scheduling problem because it's possible to arrange the jobs in any order and they will always meet their deadline.

2. Prove that $\theta \in \Theta \implies \omega \in \Omega$:
   Let's suppose we start with a solvable instance of our *Scheduling* problem. Given the set of jobs $\{j_1, ..., j_n, j_{n+1}\}$ where the first n jobs are defined the same as before and job j_{n+1} having $s_{n+1} = k$, $l_{n+1} = 1$ and $d_{n+1} = k + 1$. The extra job must be done in $[k, k + 1]$, but there would still be λ units of time available in the time interval $[0, \lambda + 1]$. Since this instance is solvable, we have $|X'| = k$ jobs that can be executed before the job j_{n+1} and the remaining $\lambda - k$ jobs after the job j_{n+1}. The set X' solves the *Subset Sum* problem.

We proved that our scheduling problem is *NP-Hard*, so we can say that it is not solvable with a quick algorithm.