Algorithm Design - HW1

Edoardo Puglisi - 1649359

5 December 2019

1 Exercise 1

Algorithm 1: Help Philip

Result: Find optimal expected reward
Initialize N, K, costlist[K], M[N+1][K];
for *box* ← *K* **to**-*1* **do**
    for *rew* ← 0 **to** *N+1* **do**
        t = expected(box,rew);
        M[rew][box] = max(t-c[box],rew);
    end
end
return M[0][0];

Creates a table $K \cdot (N+1)$ with all possible expected wins. We start from last box cause is the one without constraints on next box. t represent the expected value given the following boxes (thats the reason why we start from last one). Expected value at box b and reward rew is calculated as following:

$$t = \frac{M[rew][b+1] * (rew+1)}{N+1} + \sum_{i=rew+1}^{N} \frac{M[i][b+1]}{N+1}$$

Note: on last box replace $M[rew][b+1] * (rew+1)$ with rew and $M[i][b+1]$ with i to avoid out-of-bound exception.
The result we want in return is the first cell because it represent our initial state $(box = 0, rew = 0)$.
The cost is $O(n^2 \cdot k)$.

2 Exercise 2

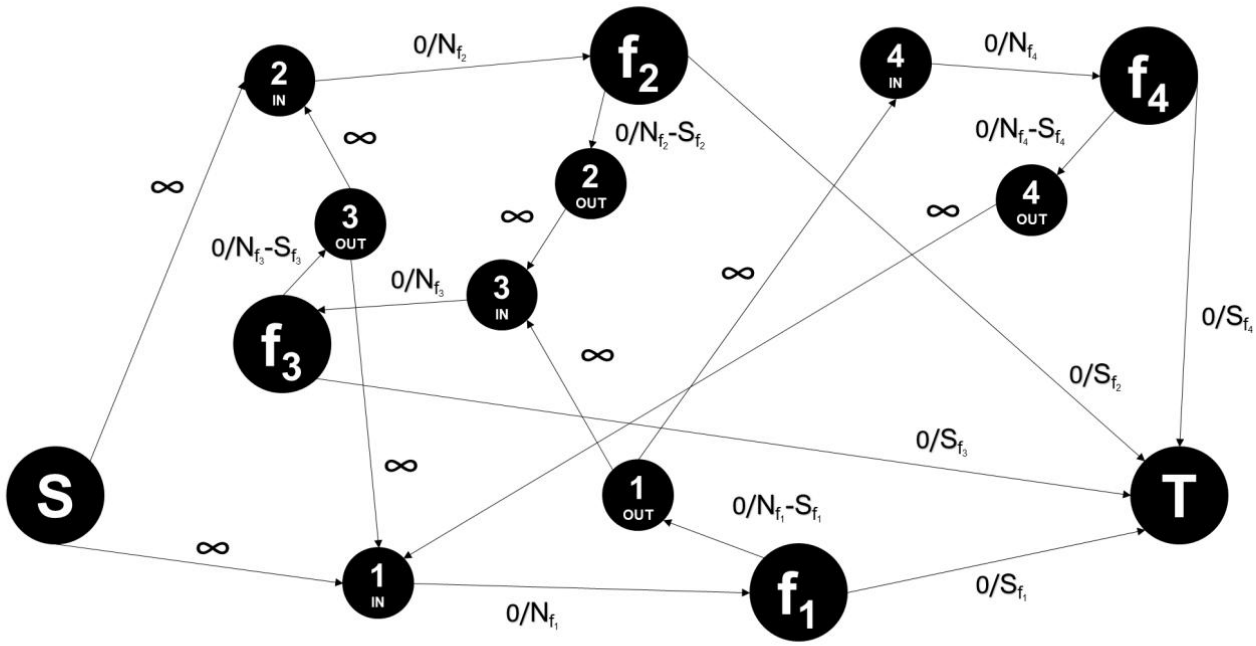
Make use of $UNION - FIND$ structures like in Kruskal's algorithm:
$FIND - SET(v)$ returns the the set containing node v;
$UNION(s1, s2)$ unify the two sets s1 and s2 in a new set of size $|s1| + |s2|$.
Sort edges in crescent order according to their weight. T = total weight of complete graph. For every edge e = (v1,v2) (from min to max weight) compute $FIND-SET(v1) = X$ and $FIND-SET(v2) = Y$ and then $UNION(X, Y)$ to add both to the "*wannabe*" complete graph. We want a complete graph so there must be an edge connecting each pair of node: for this reason, each time we do an union of sets, we should add new edges connecting each node of the two sets X and Y. These edges must be heavier than $e(weight(e) + 1)$ so that the Kruskal's algoritm will avoid to add them in the MST. To calculate the total weight of the graph, at each step:

$$T = T + weight(e) + (|X| \cdot |Y| - 1) \cdot (weight(e) + 1)$$

Creating UNION-FIND structures and ordering the edges in ascending order will cost $O(|V| \cdot \log |V|)$ with a simple merge-sort, iterating over all edges will cost $O(|V|)$, so the total cost will be $O(|V| \cdot \log |V|)$ to find the total minimal weight of the graph.
If we want to compute even the complete graph itself, each time we make the $UNION$ operation, we have to add the edges to the graph: this will cost $|V|^2$ due to the fact that for each vertex in X we iterate over all vertices in Y.

3 Exercise 3

The goal is to compute how much chocolate Federico can sell according to his friends network: for this purpose we model a network flow as follow.
Of course the SOURCE (S) will be Federico with available resources = ∞. We can imagine the SINK (T) instead as the total of possible customers and therefore the MAX-FLOW of the graph will give us the maximum of chocolate that Federico can sell in a particular week. F is the set of vertices representing Federico's friends.
Every vertex $f_i \subseteq F$ is connected with an IN_i vertex with an incoming edge of capacity N_{f_i} representing the max storage capacity of f_i.
Every vertex $f_i \subseteq F$ is connected with an OUT_i vertex with an outgoing edge of capacity $N_{f_i} - S_{f_i}$ representing the max quantity of chocolate that f_i can exchange with other friends.
Every vertex $f_i \subseteq F$ is connected with T with and outgoing edge of capacity S_{f_i} representing the max quantity of chocolate that f_i can sell to customers.
Federico knows every connection between his friends and each other and of course with him: lets call the subset of friends that he meets regularly X. Federico is connected to every $f \subseteq X$ with an edge of capacity ∞ to the corresponding IN_i. Set X represents the only way that Federico can use to put the chocolate into the network, the only direct connection between him and customers. Every $f \subseteq F$ could meet each other and exchange exceeding chocolate $N_f - S_f$: connect every pair of "meeting friends" i, j with two edges $e_1 = (OUT_i, IN_j)$ and $e_2 = (OUT_j, IN_i)$ of capacity equal to ∞. Example following.



In the new semester every $f \subseteq F$ will have an associated university building $b_f \subseteq B$ with a certain number of customers available c_b. To model this scenario we can simply add edges from every f_i to their corresponding b_i with capacity S_{f_i} and then from every b_i to the sink with capacity c_{b_i}. Federico's business will be lowered only if for every b_i the sum of capacity of incoming edges is > than the outgoing edge capacity.

4 Exercise 4

The goal of the exercise is to demonstrate that the problem is NP-complete: we will do this demonstrating that it is NP and NP-hard.

NP demonstration A problem is NP if given an instance of it and a possible solution for it, it requires a poly-time algorithm to tell if it is valid or not. In our case a possible solution is a complete sequence of jobs that Giovanni has to do. To check it we just need to iterate over it and verify, for each job j_i, that, given its starting time t_i :

- $t_i \geq s_i$ to avoid a job starting before its earliest time.
- $t_i + l_i \leq d_i$ to avoid job finishing after its deadline.
- $t_i + l_i \leq t_{i+1}$ to avoid overlapping with next job.

Iteration requires $O(N)$ time $\rightarrow$ the problem is NP.

NP-hard demonstration We will do this reducing Subset-Sum, a well known NP-complete problem, to this problem. Suppose we have n numbers w_i and our goal is to find a subset of sum W. We now create a similar version for scheduling: $K = \sum_{i=1}^{n} w_i$, n jobs j_i each with $s_i = 0$ and $d_i = K + 1$ (same earliest time and deadline) and $l_i = w_i$. In this way we can set them in any order, all finishing in time. Now lets add one more constrain: we want to be able to solve it only by grouping together a subset of jobs whose durations sum up to W. For doing this we define job j_{n+1} with $s_{n+1} = W$, $d_{n+1} = W + 1$ and $l_{n+1} = 1$. In a possible solution, job j_{n+1} can be run only in $[W, W + 1]$ leaving K time between $s_i = 0$ and $d_i = K + 1$: in this way every job must be executed one by one without pauses. Assuming that jobs j_i with $i = 1 \ldots n$ are the ones that run before time W than $w_{i_1} \ldots w_{i_n}$ is the Subset that sum up to W. If we have $w_{i_1} \ldots w_{i_n}$ we can schedule them before j_{n+1} and the remainder after j_{n+1} obtaining a feasible solution.