Homework Algorithm Design

Daniele Cantisani 1707633
Daniele Iacomini 1706790

5 December 2019

1 Ex1

The goal of this exercise is to design an algorithm to find the expected optimal value that can be obtained by participating in the game "Open the Boxes and keep the Best". We decide to consider the dynamic programming to solve the problem: this means that the resolution of a sub-problem not specifically focused on the search for the expected optimal value will help us in the final resolution and the discovery of the required value.
The idea is to create a matrix N * K where N are the maximum reward obtainable in the game and K are the number of boxes that can be opened. Starting from the last column (the last box) we iterate the matrix by column, calculating the gain that we expect to find if, having reward n (maximum value reached at k-th cell), we decide to open the box.
The algorithm is therefore focused on the application of the following formula for each matrix cell:

$$max\left(n,\ \sum_{q=0}^{N} E\left[R_{n,k+1}\right]\left[V_n = q\right] \times Prob\left[V_n = q\right]\right)$$

In the case of the column K, the expected value ($E\left[R_{n,k+1}\right]\left[V_n = q\right]$) is simply the maximum reward obtained at that time. In fact, if I have no boxes to open and I have a reward n in my hand, then the expected value is just n.
To this formula it is necessary to integrate the cost of the open box that must be subtracted from the calculation of the expected gain.

Algorithm Computing Matrix

```
1: Given N, K and C[] array containing the cost of each box
2: procedure COMPUTE_MATRIX(N, K, C[])
3:     initialize M[N][K] ← 0
4:     for each box i = K to 1 do
5:         for each reward j = 0 to N do
6:             rmax ← 0
7:             res ← 0
8:             for each reward q = 0 to N do
9:                 res ← res + E[R_{j,i+1}][V_j = q] × Prob[V_j = q]
10:            end for
11:            res ← res − C[i]
12:            M[j][i] ← res
13:        end for
14:    end for
15: end procedure
```

After constructing the matrix we are able to recover the desired value then, the expected optimal reward that will be found in our example at the first element of the matrix. In fact, the optimum reward is exactly the reward we expect to find when we have not yet opened any boxes and therefore we have not started playing.
The main instruction is executed N * N * K times so the cost of the algorithm is O ($N^2 * K$).

The second goal of the exercise is to find the same result by improving the algorithm in O (N * K).
The idea is to remove the last for loop from the previous algorithm by finding mathematical relationships between the cells of the matrix so as to avoid the additional cost of iterating from 0 to N for each cell.
Reasoning in this way we point out three cases:
Case 1: We notice that the first cell that we compute (M[0][K]) it is the sum of $\frac{i}{n}$ for each $i \in [0, N]$. So we can calculate the same cell like this: $\frac{((N+1)*N)}{2} * \frac{1}{N}$ that becomes $\frac{N+1}{2}$.
Case 2: The remaining cells of the matrix in the first row (M[0][k] with $k \in [1, K-1]$) are nothing more than the combination of the expected rewards of the column computed previously, divided by the number of rewards N. Therefore, during the calculation of the previous column we can sum the result in a variable ($columnsum$). When all the cells in the column have been calculated and we move to the next column we make $\frac{columnsum}{N}$ and then we reset the variable as it will contain the sum of the values of the new column whose first element is just what we calculated.
Case 3: DA FARE

2 Ex2

The objective of the second exercise is to construct a complete minimum graph having an MST as input. We can handle the problem request using the Kruskal algorithm and the cut property of a graph. To benefit of the Kruskal algorithm, we must use a Disjoint-set data structure. This data structure partitions every node of the graph as many disjoint subgroups. It also allows us to perform basic Find and Union operations at a low cost.

Algorithm Find Complete Graph G

```
1: Given An MST T of |V| vertices
2: procedure FIND_COMPLETE_GRAPH(Graph T)
3:     initialize Disjoint-Set of |V| elements
4:     G ← T
5:     Q ← SortedList of Edges of T in ascending order
6:     while Q is not empty do
7:         edge(vertex_1, vertex_2) ←Q.RemoveMin()
8:         set_1 ← Find(vertex_1) in Disjoint Set
9:         set_2 ← Find(vertex_2) in Disjoint Set
10:        for each vertex v_1in set_1 do
11:            for each vertex v_2in set_2 do
12:                create edge e(v_1, v_2)   only if different from edge(vertex_1, vertex_2)
13:                weight_e(v_1, v_2) ← weight_edge(vertex_1, vertex_2) + 1
14:                add edge in G
15:            end for
16:        end for
17:    end while
18:    return G
19: end procedure
```

The main instruction is the creation of additional arcs to form the complete graph. The cost of the total algorithm is O($|E| + |V| * log|V|$), where $|E|$ is the number of added edges; So given that $|E| = \frac{|V|*(|V|-1)}{2}$ then the final cost is O($|V|^2$).
PER QUANTO RIGUARDA he key statement in which the arcs are added The cost of the algorithm

Algorithm Sum of the Edges of the Complete Graph G

```
1: Given A MST T of |V| vertices
2: procedure FIND_COMPLETE_GRAPH(Graph T)
3:     initialize Disjoint-Set of |V| elements
4:     sum_total ← 0
5:     Q ← SortedList of Edges of T in ascending order
6:     while Q is not empty do
7:         edge(v_1, v_2) ←Q.RemoveMin()
8:         set_1 ← Find(v_1) in Disjoint Set
9:         set_2 ← Find(v_2) in Disjoint Set
10:        dimension_1 ← set_1.size()
11:        dimension_2 ← set_2.size()
12:        sum_total ← sum_total + weight_edge(v_1, v_2)
13:        Union(set_1, set_2)
14:        sum_total ← sum_total + (dimension_1 * dimension_2 − 1) * (weight_edge + 1)
15:    end while
16:    return sum_total
17: end procedure
```

depends on the ordering function of the arcs of the MST and its inclusion in the list Q. The final cost of the algorithm is therefore O ($|V| * log|V|$).

3 Ex3

4 Ex4

In this exercise, prove that the problem, in general, can't be solved with a quick algorithm, is like proving that the above question is an NP-Complete problem. But a problem is NP-Complete if (and only if) it is both in NP and NP-Hard.So let's start showing that our problem is NP-Hard.
First of all, we have to formalize the problem.
Giovanni has only k hours available, so given n tasks in the task set J, for each task j we have an earliest time s_j, that represents time to start a task, a deadline d_j, that represents the time-frame available to complete a task, and l_j, that represents how long he will take to do task j.
Tasks, once started, can't be stopped and we know that $t(s_j) \leq d_j$. To prove that our problem is an NP-hard, we need to prove that any instance of the NP-hard problem, known as Subset Sum Problem, can be solved. This known problem say that, given a set of integers E = $\{e_1, e_2, ..., e_n\}$ and a target w, Subset Sum Problem verifies the existence of a subset of E, whose sum of the elements is exactly w.
To verify the existence of this subset, let us take an instance of our problem with n + 1 tasks. For each of the first n tasks $j_1, ..., j_n$ we set:
$s_{j_k} = 0$, $d_{j_k} = 1 + \sum_{i=1}^{n} a_i$ and $l_{j_k} = i\text{-}th$ element of the set e_i. The last task j_{n+1}, which we added to verify, will have: $s_{j_n+1} = w$, $d_{j_n+1} = w + 1$ and $l_{j_n+1} = 1$.
We can see that the last task j_{n+1} can be completed in the time span between w and $w+1$, but this means that before w, we have to complete a number of tasks that $\sum_{j=1}^{n} lj = w$, and this means that the solution of our question contains the solution of the Subset Sum Problem.
Now we need to prove that our problem is NP, and we know that, a problem is "in NP" if, given a potential solution, you can verify that it is correct or incorrect in polynomial time.
Given an instance of the problem, a certificate that is verifiable would be a data that represents the start time of each task.
Then we could check that each task runs between its release time s_j and deadline d_j. We can also prove that the solution runs in polynomial time, in fact it's like cycling on all the tasks.
So, we have shown that the problem is NP and HP-hard, in conclusion we can say that our problem is NP-complete.