Homework 1

Algorithm Design

Mariani Matteo
Matricola ID: 1815188

Contents

Exercise 1

Given an array A of n positive numbers, the goal is to find a subarray $A[i, j]$ s.t. the product of its elements is the maximum among all other possible subarrays of A.

$$\prod_{k=i}^{j} A_k = max \tag{1}$$

We are interested only in the value considered as the maximum subproduct, not in the sequence.

Pseudocode

Algorithm 1: Find a subarray of A (of consecutive elements) s.t. the product of its elements is the maximum among all other possible subarrays of A

```
1  maximum sub-product (A, n);
   Input  : Array A and the n length of that array
   Output: max, maximum subproduct among all the subarrays of A
2  max ← A[0]
3  prod ← A[0]
4  for i to n do
5  |  prod ← prod * A[i]
6  |  if prod ≥ max then
7  |  |  max ← prod
8  |  end
9  |  if max ≤ A[i] then
10 |  |  prod ← A[i]
11 |  |  max ← A[i]
12 |  end
13 |  return max
14 end
```

Proof of correctness

Considering the maximum subproduct among all the subarrays of A that are already encountered at the i-th pass (max_i) and the incremental product that it's used to update the maximum one ($prod_i$), then:

$$max_i = \begin{cases} A_0 & \text{if } i = 0 \\ max\{max_{i-1}, prod_i\} & \text{otherwise} \end{cases}$$

and:

$$prod_i = \begin{cases} A_0 & \text{if } i = 0 \\ max\{prod_{i-1} * A_i, A_i\} & \text{otherwise} \end{cases}$$

The proof is done by induction.

- **base step**: $i = 0$ then $max_i = A_0$ and $prod_i = A_0$.

- **inductive step**: considering that at the $i-1$-th step we have (1) max_{i-1} as the maximum subproduct of the array until A_{i-1} and (2) $prod_{i-1}$ as the maximum product of a sequence $A_{j,i-1}$; we have to prove that $prod_i$ is the optimum one until step i ($prod_i = prod_{\{opt\ step\ i\}}$). Note that, we can also not consider the cases for max_i because its value is clearly dependant on the one assumed from $prod_i$.

By contradiction, assume that $\exists$ a subsequence $A_{j,i}$ s.t. $prod_i \neq prod_{\{opt\ step\ i\}}$, in particular $prod_i > prod_{\{opt\ step\ i\}}$. There are two cases and subcases:

- $prod_i = A_i$. (1) If the algorithm above chooses $prod_{\{opt\ step\ i\}} = A_i$ means that $prod_i = A_i > A_i = prod_{\{opt\ step\ i\}}$, but this a contradiction. In the other case, (2) the algorithm willl choose $prod_{\{opt\ step\ i\}} = prod_{i-1} * A_i$. If this happens, means that the algorithm detects that $prod_{i-1} * A_i > A_i$ (so clearly, $prod_{i-1} > 1$). But $prod_i = A_i > prod_{\{opt\ step\ i\}} > A_i$ and this is a contradiction.
- $prod_i = prod_{i-1} * A_i$. (1) The algorithm chooses $prod_{\{opt\ step\ i\}} = prod_{i-1} * A_i$. But if $prod_i > prod_{\{opt\ step\ i\}}$ by hypothesis, means that $prod_i > prod_{i-1} * A_i$ so $\frac{prod_i}{A_i} > prod_{i-1}$, but this is a contradiction. In the other case (2) the algorithm will choose $prod_{\{opt\ step\ i\}} = A_i$. If this happens, means that the algorithm detects that $prod_{i-1} * A_i < A_i$ (so clearly, $prod_{i-1} < 1$). But since $prod_i = prod_{i-1} * A_i$ then must be that $prod_{i-1} > 1$. This is a contradiction.

Complexity

The complexity is $O(n)$ because the procedure of collecting the maximum subproduct requires a time proportional to the lenght of the array A.

Code

ex1.py

```
def maxsubprod(A):
    max = A[0]
    prod = A[0]
    n=len(A)
    for i in range(1,n):
        prod = prod*A[i]
        if(prod>=max):
            max=prod
        if(max<=A[i]):
            prod=A[i]
            max=A[i]
    return max
```

Exercise 2

Given a power network $G = (V, E)$ that we assume to be a tree (a connected graph with no cycle), the goal is to find a node that is a hub conductor. A hub conductor is a node v s.t. upon removing it, the induced subgraph G' consists of connected components of size at most $\frac{1}{2} \cdot |V|$.

Pseudocode

Assuming that first there is a DFS applied to the connected graph G (started from a random node) s.t. can be obtained a tree T in which each node k has a counter that represents the sum between (1) the number of children of the subtree of k and (2) itself; we have:

Algorithm 2: Find the first hub conductor in the tree T of size n, if it exists

```
1  findHubConductor (T, n);
   Input  : Tree T and the n number of nodes
   Output: hub, hub conductor in T
2  current_node ← T.root
3  while not is_leaf(current_node) do
4      max_node ← ⊥
5      max ← 0
6      for all children c of current_node do
7          if c.counter > max then
8              max ← c.counter, where counter is the number of children of the node c + 1 (itself)
9              max_node ← c
10         end
11     end
12     if max ≤ n/2 then
13         return current_node, that is the hub conductor
14     end
15     else
16         current_node ← max_node
17         max ← 0
18         max_node ← ⊥
19     end
20 end
21 return null
```

Proof of correctness

The proof is given by contradiction.
By contradiction let's assume that v is not a hub conductor. The node v is the output if (1) $counter(v) > \frac{n}{2}$ (otherwise, the output would be the father of node v) and (2) if $counter \leq \frac{n}{2}$ for all the children of v. But removing v and its k edges, we have k connected components of which 1 has $n - \frac{k \cdot n}{2} \leq \frac{n}{2}$ nodes and $k - 1$ components with a number of nodes $< \frac{n}{2}$. But this means that v is a hub conductor.

Code

Using Python, I can't use dictionary structure to represent the graph - otherwise the complexity will not be $O(|V| + |E|)$. For this reason, I implemented the graph as the following, using the same base concept

expressed in the pseudocode but with small variants to take better advantage of the language used.

ex2.py

```
class Node(object):
    def __init__(self,name):
        self.value = name
        self.counter = 0 #num of nodes of the "subtree" that has as root the current Node +1
        self.neighbors = []
        self.isVisited = False

#starting from a random node, I increment the counters
def updateCounter(current_node):
    current_node.isVisited = True
    current_node.counter = 1
    for neighbor in current_node.neighbors:
        if neighbor.isVisited:
            continue
        else:
            current_node.counter+=updateCounter(neighbor)
    return current_node.counter

#return the hub conductor
def findHub(node,v):
    hub = node
    #instead of making a 2nd DFS to modify the isVisited values
    #the semantic is inverted: True means that the node is not visited,
    #False otherwise
    node.isVisited = False
    for neighbor in node.neighbors:
        if not neighbor.isVisited:
            continue
        if neighbor.counter > v//2:
            hub = findHub(neighbor, v)
    return hub

#node of the graph
a = Node("a")
b = Node("b")
c = Node("c")
d = Node("d")
e = Node("e")
#cardinality graph
cardin = 5
#build edges
a.neighbors = [b,c]
b.neighbors = [a]
c.neighbors = [d,a,e]
d.neighbors = [c]
e.neighbors = [c]
#random node
start_node = a
#update counters
updateCounter(start_node)
#find hub conductor
hub = findHub(start_node, cardin)
```

Complexity

Considering the implementation, we have to consider the cost of (1) the DFS applied on the graph to update the counters, (2) the optional DFS applied to reset the *isVisited* variable and (3) the last search to find the hub conductor - that is always less than the DFS. For this reasons, the complexity is $O(|V| + |E|)$.

Exercise 3

Exercise 3.1

By definition, a problem A is an NP-Hard problem iff there exists a NP-Complete problem B s.t. $B \leq_p A$. The problem that we need to show as NP-Hard is the *Bounded-Degree Spanning Tree*. The k-bounded spanning tree problem has as input an undirected graph $G(V, E)$ and we have to decide if exists a spanning tree s.t. each node has at most k neighbors.
In order to show the problem is NP-Hard, we have to find an NP-Complete problem that is poly-time riducible to it. For the Bounded-Degree Spanning Tree, the problem is the *Hamiltonian-Path*, a well-known NP-Complete problem.

HAMILTONIAN-PATH $\leq_p$ BOUNDED-DEGREE SPANNING TREE

Considering an appropriate choice for the number k ($k = 2$), we have the following:
Given a graph G exists a Hamiltonian-Path iff exists a spanning tree where each node has at most 2 neighbors.

Proof

- ($\rightarrow$) A Hamiltonian-Path is a path in an undirected or directed graph $G(V, E)$ that visits each node exactly once. Let's assume that s and t are respectively the starting and the ending nodes of this path - where $s, t \in V$. This means that (1) each node in $V \setminus \{s, t\}$ has at most two incident edges and (2) s, t have only one incident edge. For this reason the Hamiltonian-Path is exactly a spanning tree where each node has at most 2 neighbors.
- ($\leftarrow$) Given a spanning tree where each node has at most 2 neighbors, we can clearly see that starting from one of the leaves we can create a path where each node is traversed exactly once - and this is a Hamiltonian-Path.

Exercise 3.2

Let's consider a Hamiltonian-Path problem with a graph $G = (V, E)$ and two nodes s and t as input.

We have to prove that:
A Hamiltonian-Path instance is a yes instance (so it exists) iff exists a Bounded-Degree Spanning Tree of degree at most $k = 5$ on a graph G'.

Proof

Using gadgets, I can create a new graph G' from G as follow:

Starting from the original graph G, I add (1) $k - 2$ new nodes (v') for each node $v \in V$ and (2) other two nodes s' and t'. After that, the first nodes are connected with edges (v', v) for each $v \in V$, and the second ones with the (s', s), (t', t) edges. In our scenario, $k = 5$ so we have to add exactly 3 nodes for each $v \in V$.

- ($\rightarrow$) There exists a Hamiltonian-Path from s to t in G if G' has a spanning tree of degree at most 5. By construction of G', all the new nodes have degree 1, so all the new edges must be in any spanning tree of G' to achieve connectivity. Note that: (1) the nodes s and t can have at most one more incident edge (because they have already three edges from the new three nodes plus the edge from the respective new nodes s' or t') and (2) all the other nodes in $V \setminus \{s, t\}$ can have at most two more incident edges. Since that all the nodes in G had at most 2 neighbors (by exercise 3.1), adding the new ones we get a spanning tree in G' with at most 5 neighbors.
- ($\leftarrow$) There exists a Bounded-Degree Spanning Tree of degree at most 5 on a graph G' if a Hamiltonian-Path instance on G is a true instance. Removing from G' the nodes that were added previously for each $v \in V$, we don't lose connectivity in the new graph since they were leaves of G'. The obtained graph has a spanning tree of at most $k = 2$ neighbors, so exists a Hamiltonian-Path (by exercise 3.1).

Exercise 4

Exercise 4.1

In order to find the best way to schedule all the n jobs in the m machines, the completion time C_{max} of the final job must be minimized. Once the correct C_{max} is found (C_{opt}), the schedule of all the jobs is trivial: we can simply take the first job j in the sequence and put it in the first available machine m, and so on. When a job j_k that has to be scheduled in a machine m_i will exceeds the C_{opt} for that m_i (so the maximum load allowed); the job j_k will be splitted in two parts: the first part will be scheduled as the last job of the machine m_i to achieve the C_{opt}, the remaining part will be scheduled in the machine m_{i+1} as the first job. The tricky part lays in the way to choose the correct C_{max}.

Pseudocode

Algorithm 3: Find the minimum completion time among m machines and n jobs with different processing times.

```
1  findMinCompletionTime (jobs, m);
   Input  : Jobs that have to be scheduled by the m machines
   Output: minimum C_max
2  C_max ← 0
3  total_processing_times ← 0
4  for all jobs j in jobs do
5  |  if j.processing_time > C_max then
6  |  |  C_max ← j.processing_time
7  |  end
8  |  total_processing_times ← total_processing_times + j.processing_time
9  end
10 avg ← total_processing_times /m
11 if C_max < avg then
12 |  C_max ← avg
13 end
14 return C_max
```

Proof of correctness

Considering that the machines support interruptions *but* a job cannot be executed simultaneously on two different machines (so it cannnot overlap on itself), we have to prove that the algorithm finds the optimum C_{max}.

- $m \geq n$, trivial. Because I can schedule all the n jobs in the m machines s.t. the minimum completion time is the finishing time of the longest scheduled job.
- $m < n$, means that at least one m_i machine will have more than one job to schedule. In this case, we have two possibilities:
  - $C_{max} = p_j$, where p_j is the longest processing time among all the jobs.
  - $C_{max} = \sum_{i=1}^{n} \frac{p_i}{m}$

Let's prove the two possibilities.

(1) If $p_j \geq \sum_{i=1}^{n} \frac{p_i}{m}$ then $C_{opt} = p_j$.
By contradiction, assume that C_{opt} is not the optimum. For this reason $\exists$ a schedule with $C < C_{opt}$ that can schedule all the jobs without that a single job is executed simultaneosly on two machines. Besides, $C < p_j$. But p_j is the longest processing time and this means that the job j will overlap with itself even if the machines are empty (simultaneous execution). This is a contradiction.

(2) If $p_j \leq \sum_{i=1}^{n} \frac{p_i}{m}$ then $C_{opt} = \sum_{i=1}^{n} \frac{p_i}{m}$.
By contradiction, assume that $C = p_j$ is the optimum. This means that there exist a schedule that can schedule all the jobs using machines that have at most a load $L = p_j$. But, if a further job k with a $p_k < p_j$ arrives; the schedule will have a machine with a load $L' = L + \epsilon > L$, where ϵ is a part of the job k that has to be scheduled. For this reason, C cannot be the optimum.

Complexity

The problem requires two tasks: (1) determine the minimum C_{max} and (2) schedule all the j jobs in the m machines using the found C_{max}. Both tasks have a $O(n)$ complexity because in each task the jobs are scanned exactly once.

Exercise 4.2

If the interruptions are not allowed, the problem analyzed in 4.1 is NP-Hard, even on 2 machines.
To prove that is NP-Hard we have to find an NP-Complete problem that is ploy-time reducible to it. For the *MultiprocessorScheduling* problem, the *Partition* problem represents a good candidate.

PARTITION $\leq_p$ MULTIPROCESSOR SCHEDULING

Proof

Let's consider a Partition problem and an instance $S = \{p_1, p_2, ..., p_n\}$ of finite n positive integers. The goal is to find if exists a partition of S into two subsests S_1 and S_2 s.t. $sum_{\{p \in S_1\}} p = sum_{\{p \in S_2\}} p$.

To show the reduction correctly, we have to prove the following using a gadget:
there exists a scheduling problem that has $C_{max} = \frac{1}{2} \sum_{i=1}^{n} p_i$ iff the partition instance is a yes instance.

- ($\leftarrow$) Let's assume that the instance of partition is a no instance. This implies that $\nexists\ S_1, S_2$ s.t. $sum_{\{p\ \in\ S_1\}}p = sum_{\{p\ \in\ S_2\}}p$. If the elements in S are exactly the processing times p_i, $i = 1...n$ of the n jobs and the two subsets correspond to the number of machines available, then doesn't exist a way to schedule all the jobs to have C_{max} minimized, if the interruptions are not allowed.
- ($\rightarrow$) $C_{max} = \frac{1}{2}\sum_{i=1}^{n} p_i$. So $\exists$ 2 machines M_1 and M_2 s.t. the completion time is minimized. This means that the n jobs have to be partitioned in the two machines s.t. $sum_{\{p\ \in\ M_1\}}p = sum_{\{p\ \in\ M_2\}}p$. This corresponds exactly to a yes instance of partition.

Since that the algorithm that build the transformation from an instance of Partition into one of the Multiprocessor Scheduling involves a polynomial number of steps, the reduction is proved.

Exercise 5

The problem can be rappresented by a graph and be solved using the max flow/min cut formulation. The graph G has built with the following rules:

- the source node s is connected to all the resource cards nodes r_j through edges of cost c_j;
- the sink node t is connected with all the challenges nodes C_i through edges of cost p_i;
- edges of infinite cost are added to connect each resource cards node r_j to the respective challenges nodes C_i that requires r_j for the completion.

Once G is built, we can apply the *Ford-Fulkerson* algorithm to obtain the max-flow and after that the min-cut (A, B). If we consider the set of nodes $B \setminus \{t\}$, they represent the set of resource cards to buy and challenges to satisfy in order to maximize the revenue.
This formulation can be seen as a *Project Selection* problem, where the resource card and the challenges are the projects, in particular the resource cards are the prerequisites for the callenges. Note that the edges of the graph are inverted w.r.t. the real formulation of Project Selection problem.

Proof

First of all, some properties of the min cut (A, B):

1. the upper bound of the maximum flow in the graph G is $C = \sum_{1 \leq i \leq m} c_j$, which is the sum of all the possible costs deriving from the purchase of all the resources cards r_j;
2. the edges connecting the resource cards and the challenges have infinite values, so they will not be contained in the min cut - otherwise the min cut could have a capacity value greater than the max flow. If a resource card $r_j \in A$ then all the challenges C_i that use r_j belong to A, since no edge with capacity greater than the max flow can belong to the min cut;
3. the capacity of the min cut $c(A, B)$ can be written in two equivalent forms:
   - $C - (\sum_{r_j \in A} c_j - \sum_{C_i \in A} p_i)$.
   - $P - (-\sum_{r_j \in B} c_j + \sum_{C_i \in B} p_i)$, where $P = \sum_{1 \leq i \leq n} p_i$, so the sum of all possible points deriving from the completion of all the challenges C_i in the graph G.

What we want to prove is that:
(A, B) is the min cut iff $B \setminus \{t\}$ is the set of resource cards and challenges that maximizes the net revenue.

- ($\rightarrow$) By contradiction, assume that $B \setminus \{t\}$ is not the optimum set of challenges and resource cards that maximizes the net revenue. Therefore $\exists\ B' \setminus \{t\}$ s.t. $\sum_{C_i \in B'} p_i - \sum_{r_j \in B'} c_j > \sum_{C_i \in B} p_i - \sum_{r_j \in B} c_j$, so B' has a greater revenue than the one obtained by B.
  Considering the two cuts (A, B) and (A', B') for sure we have that the sum of all possible points deriving from the completion of all the challenges is the same $\sum_{C_i \in B} p_i + \sum_{C_i \in A} p_i = \sum_{C_i \in B'} p_i + \sum_{C_i \in A'} p_i = P$.
  Considering the capacities of the cuts we have that $c(A, B) = C - (\sum_{r_j \in A} c_j - \sum_{C_i \in A} p_i)$ and $c(A', B') = C - (\sum_{r_j \in A'} c_j - \sum_{C_i \in A'} p_i)$. From the property (1) we have that $\sum_{r_j \in A} c_j + \sum_{r_j \in B} c_j = \sum_{r_j \in A'} c_j + \sum_{r_j \in B'} c_j = C$ and susbstitute these in the $c(A, B)$ and $c(A', B')$ equations, we have:

  1. $c(A, B) = C - (C - \sum_{r_j \in B} c_j - (P - \sum_{C_i \in B} p_i)) = P - (\sum_{C_i \in B} p_i - \sum_{r_j \in B} c_j)$;
  2. $c(A', B') = C - (C - \sum_{r_j \in B'} c_j - (P - \sum_{C_i \in B'} p_i)) = P - (\sum_{C_i \in B'} p_i - \sum_{r_j \in B'} c_j)$.

  Since we said that $\sum_{C_i \in B'} p_i - \sum_{r_j \in B'} c_j > \sum_{C_i \in B} p_i - \sum_{r_j \in B} c_j$ we can say that $c(A', B') < c(A, B)$. But this is contradiction because we assumed that (A, B) was the min cut.

- ($\leftarrow$) Since $B \setminus \{t\}$ is the optimum set containing all the challenges and resource cards that maximizes the net revenue, we have that $\sum_{C_i \in B} p_i - \sum_{r_j \in B} c_j$ is maximized.
  Considering that (1) $c(A, B) = \sum_{C_i \in A} p_i + \sum_{r_j \in B} c_j$ and (2) $\sum_{C_i \in A} p_i = \sum_{1 \leq i \leq n} p_i - \sum_{C_i \in B} p_i$ we can say that $c(A, B) = \sum_{1 \leq i \leq n} p_i - (\sum_{C_i \in B} p_i - \sum_{r_j \in B} c_j)$.
  Know that the revenue is maximized by hypothesis, the difference is minimized; so (A, B) is the min cut.