Algorithm Design - Homework 1

Academic year 2017/2018

Tufa Alexandru Daniel 1628927

10 December 2017

1 Exercise 1

1.1 Request

Given an array A of n positive real numbers we need to find a subarray A[i,j], i.e. starting at A[i] and ending at A[j], such that $\prod_{k=i}^{j} A_k$ is maximized. The used algorithm has to compute only the value of the max and must run in linear time ($O(n)$).

1.2 Solution

To find the solution to this problem we did a linear reduction from our problem that regards finding the maximum product in a subarray to the well known problem of finding the maximum sum in a subarray [1], more precisely Kadane's algorithm that runs in linear time. To do this we would need a function that transforms products in sums and rational numbers between 0 and 1 into negative numbers. This function is the logarithm function. The other useful property of the logarithm is that it's a monotonic function. For simplicity we will use $\log_{10}$. The main property of the logarithm that we'll use is the following:

Definition 1.1.

$$log_{10}(ab) = log_{10}a + log_{10}b$$

Let's suppose that we have the algorithm for calculating the largest sum contiguous subarray. We can make a linear reduction and compute the maximum product in the following way.

```
def maxProd(array):
    for i in range(len(array)):
        array[i] = math.log10(array[i])
    optimumMax = maxSum(array)
    return math.pow(10,optimumMax)
```

Theorem 1.1. *Reduction is correct.*

Proof. We transform the original array into a new array by applying the logarithm function to each element. When we receive the result of the maximum contiguous sum we compute $10^{optimumSum}$ to obtain the answer to the original problem.
$maxSum \Rightarrow maxProd$: Asssuming that we know maxSum we will prove by induction that maxProd derives from it thanks to our transformation.

- Induction Principle: Assume for n maxProd = 10^{maxSum}.
- Base Case: If n = 1 then

$$maxSum = log_{10}(A[0])$$
$$maxProd = 10^{maxSum}$$
$$maxProd = 10^{log_{10}(A[0])}$$
$$maxProd = A[0]$$

- Inductive Step:WLOG we assume that A[n+1] belongs to the maximum contiguous sum. We have that

$$maxSum' = maxSum + log_{10}A[n+1]$$
$$maxProd' = 10^{maxSum'}$$
$$maxProd' = 10^{maxSum+log_{10}(A[n+1])}$$
$$maxProd' = 10^{log_{10}(A[i])+..+log_{10}(A[n])+log_{10}(A[n+1])}$$
$$maxProd' = 10^{log_{10}(A[i]*..*A[n]*A[n+1])} \text{ by def 1.1}$$
$$maxProd' = A[i] * .. * A[n] * A[n+1]$$
$$maxProd' = maxProd * A[n+1]$$

□

$maxProd \Rightarrow maxSum$: We can reuse the same reasoning from the previous demonstration and suppose that are we given an array with the maximum product calculated. Instead of applying the logarithm function to each element we apply $10^{A[i]}$.

- Induction Principle: Assume for n maxSum = $log_{10}maxProd$
- Base Case: If n = 1 then

$$maxProd = 10^{A[0]}$$
$$maxSum = log_{10}maxProd$$
$$maxSum = log_{10}10^{A[0]}$$
$$maxSum = A[0]$$

- Inductive Step: WLOG we assume that A[n+1] belongs to the maximum contiguous product. We have that:

$$maxProd' = maxProd * 10^{A[n+1]}$$
$$maxSum' = log_{10}maxProd'$$
$$maxSum' = log_{10}(maxProd * 10^{A[n+1]})$$
$$maxSum' = log_{10}maxProd + log_{10}(10^{A[n+1]})$$
$$maxSum' = maxSum + A[n+1]$$

Theorem 1.2. *Reduction is linear.*

Proof. We compute a constant number of operations for each index in the array, so the total number of operations will be executed once depending on the size of the array. □

2 Exercise 2

2.1 Request

Given a graph G=(V,E) and assuming it's a tree we need to find the hub conductor. The hub conductor is a node v, such that upon removing v, the induced subgraph G' = $\{V\backslash\{v\}, E \cap (V\backslash\{v\} \times V\backslash\{v\})\}$ consists of connected components of size at most $\frac{|V|}{2}$. This algorithm has to run in linear time *(O(|V|+|E|))*.

2.2 Solution

To solve this problem we'll need a few ingredients.

Theorem 2.1. *Given a node u, if we remove it and by doing so we generate a connected component of size* $> \frac{|V|}{2}$*, then the connected component is unique.*

Proof. Let's suppose, by contradiction, that, after removing a certain node u, the generated connected components of size $> \frac{|V|}{2}$ are more than one. If this is the case, we call k the first such connected component and i all the other connected components formed after removing u. We will define as:

- $|V|$ the size of the graph, in terms of number of nodes
- $|k|$ the sum of all the nodes in the connected component that we called k with size $> \frac{|V|}{2}$
- $\sum_{i \neq k} |i|$ the sum of all the nodes in the connected components generated after removing u and not considering the nodes that belong to k

Now we can write that:

$$|V| = |k| + \sum_{i \neq k} |i| + 1$$

$$|k| = |V| - \sum_{i \neq k} |i| - 1 > |V|/2$$

$$\sum_{i \neq k} |i| < \frac{|V|}{2} - 1$$

But this is a contradiction because the sum of all the connected components, without considering k, can't be $< \frac{|V|}{2}$ if there exists at least another connected component with size $> \frac{|V|}{2}$. □

Corollary 2.2. *Given node u and node v adjacent to it that belongs to a connected component of size* $> \frac{|V|}{2}$*, the connected component to whom u belongs to is unique and has size* $< \frac{|V|}{2}$*.*

Theorem 2.3. *Given any node v, by always choosing a node u adjacent to it that belongs to a connected component of size* $> \frac{|V|}{2}$*, I will eventually find the hub conductor.*

Proof. Suppose, by contradiction, that we'll never find the hub conductor. If this is the case then, for every node, if we remove it, it will always generate at most one connected component of size $> \frac{|V|}{2}$ by theorem 2.1. By following our hypothesis, given a node v, I will always choose a node u adjacent to it that belongs to a connected component of size $> \frac{|V|}{2}$. By iterating our reasoning, sooner or later, we'll reach an external vertex. Let's call this external vertex u while the vertex adjacent to it v. By definition of external node, the connected component that is generated after removing this external node has size $|V|$-1, but this is a contradiction because, by hypothesis, given that I always choose a node u adjacent to v that belongs to a connected component of size $> \frac{|V|}{2}$ this violates corollary 2.2 which states that the connected component to whom v belongs is unique and of size $< \frac{|V|}{2}$. This contradiction indicates us that we would have found the hub conductor before reaching this external node. □

Now we can write the algorithm that will resolve our problem:

1. Given a graph G, choose any vertex
2. Check if the chosen node is a hub conductor
3. If not, choose an adjacent node u such that the connected component to whom u belongs to has size $> \frac{|V|}{2}$
4. Go to point 2

Theorem 2.4. *Algorithm is correct.*

Proof. Because the algorithm follows strictly the hypothesis given by theorem 2.3, it will find the hub conductor. □

Theorem 2.5. *Cost of the algorithm is* $O((|V| + |E|))$.

Proof. As we have seen in the proof of theorem 2.3, starting from any vertex and excluding the trivial cases, the search for the hub conductor will never lead us to an external vertex. With this assumption we can use as a weak upper bound $O(|V| + |E|)$. □

To simplify our implementation we will transform the graph given in input from an adjacency list data structure to a tree data structure where every node, starting from the root, has a list of children. This can be done because we assumed that the graph given in input is connected and acyclic. The transformation from the graph data structure do a tree data structure can be done with one DFS by choosing any vertex, in our case the first one, as root, so the cost of the transformation is O(|V|+|E|). The implementation has been done in Java as follows:

```
public Tree(Graph g) {
    List<Vertex> vertices = g.getVertices();
    if(vertices.size() == 0) {
        this.root = null;
        this.size = 0;
        this.children = null;
    } else {
        Vertex root = vertices.get(0);
        this.root = "";
        this.children = new ArrayList<>();
        DFS(this,root);
    }
}

public static void DFS(Tree tree, Vertex v) {
    tree.setRoot(v.getName());
    v.setVisited(true);
    for(Edge e: v.edges) {
        if(!e.getVisited()) {
            Vertex w = e.opposite(v);
            if(!w.getVisited()) {
                Tree child = new Tree(w.getName());
                tree.insertChild(child);
                e.setVisited(true);
                DFS(child,w);
            }
        }
    }
}
```

To find the hub conductor we'll need to calculate, for each subtree, its size. With the term subtree we refer to the nodes contained in the tree rooted at one of the children of a given node u. This can be done with a DFS of cost O(|V|) as follows:

```
public static int size(Tree tree) {
    if(tree.children.size() == 0) {
        tree.setSize(1);
        return 1;
    }
    int total = 1;
    for(Tree child:tree.children) {
        total += size(child);
    }
    tree.setSize(total);
    return total;
}
```

The implementation of the algorithm for finding the hub conductor is the following:

```
public static Tree hubConductor(Tree tree,int V) {
    for(Tree child: tree.children) {
        if(child.getSize() > V / 2) {
            return hubConductor(child,V);
        }
    }
    return tree;
}
```

Theorem 2.6. *Cost of finding the hub conductor is* $O(|V| + |E|)$.

Proof. The transformation of the data structure from adjacency list to tree data structure has the same cost of a DFS, $O((|V| + |E|))$. To calculate the size of each subtree, because we do a DFS in a tree data structure, has cost $O(|V|)$ as we can see in the following paper [2] by using a recurrence relation. The algorithm for finding the hub conductor, given that we still do a DFS on a tree data structure, has cost $O(|V|)$. Therefore the total process of finding the hub conductor, from the original graph, has cost $O(|V| + |E|)$. □

3 Exercise 3

3.1 Request

Given a graph G = (V,E) show that the following two problems are NP-hard:

1. G has a spanning tree where every node has at most k neighbors and k is part of the input.
2. G has a spanning tree where every node has at most 5 neighbors.

3.2 Solution

Definition 3.1. Given problem X thas is NP-complete and problem Y, we define problem Y to be NP-hard if we can perform a polynomial reduction from problem X to Y.

Definition 3.2. A polynomial-time reduction is a method of solving one problem by means of a hypothetical subroutine for solving a different problem that uses polynomial time.

Theorem 3.1. *Hamiltonian Path* $\equiv_p$ *2-bounded spanning tree*

Proof. These two problems are intrinsically equivalent:
Ham-Path $\Leftarrow$ 2-BST: if a graph G contains a spanning tree, given that it's spanning, it respects also the Hamiltonian condition[1]. Because it has degree ≤ 2 it cannot branch so by definition it's a path.
Ham-Path $\Rightarrow$ 2-BST: if a graph G contains a Ham-Path, this path must be a spanning tree since the path visits every node and a path trivially is a tree. □

Theorem 3.2. *2-BST* $\leq_p$ *k-BST*

Proof. Given a graph G, to build our polynomial reduction from 2-BST to k-BST we require the use of a gadget. From the original graph we build a new graph G' where, for every node in the original one, we add k-2 nodes to it . By doing this, every internal node of the new spanning tree will have precisely 2 + k - 2 = k neighbors, with the exception of the 2 external nodes of the tree that will have 1 + k - 2 = k - 1 neighbors.
$\Rightarrow$ if G has a 2-BST then, after we have added k-2 nodes to every vertex, by construction G' will have a spanning tree where every node has a maximum of k neighbors, more precisely k for the internal nodes and k-1 for the external ones.
$\Leftarrow$ if G' contains a k-bounded spanning tree, by removing the added k-2 nodes for each vertex, we obtain G that contains the k-BST where, for every node that originally had a maximum of k neighbors, we removed k-2 neighbors, so now every vertex of G has k - (k - 2) = 2 maximum neighbors. This is the case because, for every internal vertex, we'll remove k-2 external vertices which are the added ones, so the remaining graph will necessarily be G given that we have removed

[1] A Hamiltonian Path must cover all vertices

the added vertices. This graph will contain our 2-BST because, from any k-BST, by removing only the external vertices, we'll be left with the internal ones that still form a bounded spanning tree, but instead of k neighbours each vertex will have a maximum of 2 neighbours.
The action of adding and removing k-2 vertices to every node in the original graph is done in polynomial time with respect to the size of the input because we simply traverse all the graph once. □

Theorem 3.3. *k-bounded spanning tree is NP-hard.*

Proof. Because 2-BST is NP-Complete, by theorem 3.1, and because we have done a polynomial time reduction from 2-BST to k-BST, by theorem 3.2, using definition 3.1 of NP-hardness, the k-bounded spanning tree problem is NP-hard. □

Corollary 3.4. *Because we have proved that defining if a graph contains a k-bounded spanning tree is an NP-hard problem with k being whatever number, by simply using 5 instead of k we can see that the 5-BST problem is also NP-hard.*

4 Exercise 4

4.1 Request

Given n jobs with processing times p_1,..,p_n and m machines we need to schedule the jobs on the machines such that the completion time C_{max} of the final job is minimized. We also know that a job cannot be executed simultaneously on two different machines and also that our machines support interruptions. The algorithm must also run in linear time *(O(n))*. Once we have done this, we must prove that this problem is NP-hard even on 2 machines if interruptions are not allowed.

4.2 Solution

For the sake of simplicity we will use some particular simbols.[2] Because we need to minimize the completion time of the final job on a set of m machines, this can happen only when all machines have the same completion time. If one of the machines has a completion time that is smaller than t, then another one will necessarily have a completion time longer than t, so t is the optimum searched. Because a job cannot be executed simultaneously on two different machines we can have the particular case that a single p_j can be larger than our t, so the optimum completion time in that case becomes the highest p_j.

Theorem 4.1. $C_{max} = max(t, p_{longest})$ *is the optimum completion time.*

Proof. By contradiction we suppose there exists a $t_{opt} < C_{max}$. We will analyze the two subcases of t described above:

- C_{max}= t = T/m:
  We have defined T as the sum of all processing times and in this case the completion time is equal for all the machines so $T = m * t_{opt}$. We have that

$$t_{opt} < C_{max} = t$$

$$m * t_{opt} < m * t$$

$$T < T \quad Contradiction$$

[2] T = sum of all processing jobs
m = number of machines
t = T / m

- $C_{max} = p_{longest}$, C_{max} is the longest processing time so we may have a machine that has completion time $< C_{max}$. We will prove that if there exists $t_{opt} < C_{max}=p_{longest}$, then we have the same process running on 2 machines concurrently. Let's call s_i the starting time of process p_i and f_i the finishing time of process p_i. Our hypothesis imposes us that, for a given process, $s_i < f_i$. Let's suppose

  $$s_{p_{longest}} = x$$

  where with $s_{p_{longest}}$ we intend the starting time of the process with the longest processing time. If the processing time is higher than t_{opt} then the process must be run on another machine and the remaining part will be $x + p_{longest} - t_{opt}$. By the non concurrency condition we must have that

  $$x + p_{longest} - t_{opt} < x$$

  $$p_{longest} < t_{opt}$$

  But this is a contradiction because we have assumed $t_{opt} < p_{longest}$.

□

If C_{max} is the optimum completion time, then we can assign jobs to a machine until its processing time reaches C_{max}. If this time is reached and the job has still additional units of computation then the remaining processing time is assigned to the next machine. We will never have a job running on two machines because this can only happen if its processing time is higher than the single processing time of a machine, but we already covered this particular case.

Alg:
1. Compute C_{max} = max($p_{longest}$, T / m)
2. For each machine schedule all jobs until C_{max}
3. If a machine is full and the process has not finished its processing time, schedule it on the next machine

We will see an implementation of this algorithm in Python to better understand the underlying structure. We will consider the input as an array A of Jobs, made of their index and the processing time, and an array M of machines where every index containes a set of Jobs, made of their index and the processing time on that particular machine.

```
class Machine:
    def __init__(self):
        self.jobs = []

    def addJob(self, job):
        self.jobs.append(job)

class Job:
    def __init__(self,index,processingTime):
        self.index = index
        self.processingTime = processingTime

class Scheduler:
    def __init__(self,jobs,machines):
        self.jobs = jobs
        self.machines = machines

    def schedule(self):
        t = 0
        T = 0
        tMax = 0
        machineCounter = 0
        singleMachine = 0
        for job in self.jobs:
            T += job.processingTime
            if(job.processingTime > tMax):
```

```
                tMax = job.processingTime

        t = T / len(self.machines)
        if(tMax >= t):
            t = tMax

        print("Cmax is:" + str(t))
        for job in self.jobs:
            if(job.processingTime > t - singleMachine):
                spaceOnThisMachine = t - singleMachine
                self.machines[machineCounter].addJob(Job(job.index,
                                                spaceOnThisMachine))
                machineCounter += 1
                singleMachine = job.processingTime - spaceOnThisMachine
                self.machines[machineCounter].addJob(Job(job.index,singleMachine))
            else:
                singleMachine += job.processingTime
                self.machines[machineCounter].addJob(job)

                if(singleMachine == t):
                    singleMachine = 0
                    machineCounter += 1
        return
```

As we can see we use only two *for* cycles, one for searching for the job with the highest processing job and for T, and one cycle for assigning jobs to machines. Because the number of operations inside the cycles are constant with respect to the input and the number of iterations are linear with respect to the size of the array of jobs we can conclude that our algorithm runs in linear time *(O(n)))*.

Theorem 4.2. *Algorithm is correct.*

Proof. We have proved that the optimum value for each machine is $C_{max} = \max(p_{longer}, T/m)$ by theorem 4.1 and because our algorithm, by definition, cannot make machines have a completion time greater than C_{max} , our algorithm is correct. □

To prove that the problem is NP-hard on 2 machines if interruptions are not allowed we will need some instruments.

Definition 4.1. Load-balancing problem is a decision problem where we want to know if the optimum completion time on two machines where no interruptions are allowed is equal to the equivalent optimum completion time on machines where we allow interruptions. This completion time, in the case of 2 machines, is equal for both of them and is

$$C_{\max} = C1 = C2 = \frac{\sum_{p_i \in jobs} p_i}{2}$$

C_{max} will never be the longest processing time of a process because we don't allow overlapping of the same job.

Definition 4.2. The partition problem is the task of deciding whether a given multiset S of positive integers can be partitioned into two subsets S1 and S2 such that the sum of the numbers in S1 equals the sum of the numbers in S2. This problem is known to be NP-complete.

Theorem 4.3. *Partition* $\leq_p$ *Load-Balancing*

Let's define the partition problem and load-balancing in a mathematical way.
Partition:

$$S = \{s_i | s_i \in \mathbb{N}, \exists\ s_i = s_j, \text{for some i} \neq j\}$$
$$S1, S2 \quad s.t. \quad \sum_{s_i \in S_1} s_i = \sum_{s_j \in S_2} s_j$$

Load-balancing:

$$J = \{p_i | p_i \text{ is the processing time of job i}\}$$

$$M_1, M_2 \text{ are 2 machines with no interruptions}$$

$$\sum_{p_i \in M_1} p_i = \sum_{p_j \in M_2} p_j$$

We do the following transformation:

$$\forall s_i \in S, \text{ create a job s.t } p_i = s_i$$

$$S1 \to M1 \quad S2 \to M2$$

This transformation is polynomial because we scan all the numbers belonging to the superset S once.

Proof. Partition ⇒ Load-Balancing: Let's suppose, by, contradiction that the load-balacing problem doesn't have a solution, while the partition problem does. According to definition 4.1

$$\sum_{p_i \in M_1} p_i \neq \sum_{p_j \in M_2} p_j$$

and because of the transformation we applied, we get that

$$\sum_{s_i \in S_1} s_i \neq \sum_{s_j \in S_2} s_j$$

but this is a contradiction because we assumed that the partition problem has a solution.

Load-Balancing ⇐ Partition: Let's suppose, by contradiction, that the partition problem doesn't have a solution, while the load-balancing problem does. According to definition 4.2

$$\sum_{s_i \in S_1} s_i \neq \sum_{s_j \in S_2} s_j$$

but because of the transformation we applied, we get that

$$\sum_{p_i \in M_1} p_i \neq \sum_{p_j \in M_2} p_j.$$

This is a contradiction because we assumed that the load-balancing problem has a solution. □

Theorem 4.4. *Load-Balancing on two machines with no interruptions is NP-hard.*

Proof. Because the Partition problem is NP-complete, by definition 4.2, and because we have done a polynomial time reduction from the Partition problem to our problem, by theorem 4.3, using definition 3.1 of NP-hardness, the Load-Balacing problem on two machines with no interruptions is NP-hard. □

5 Exercise 5

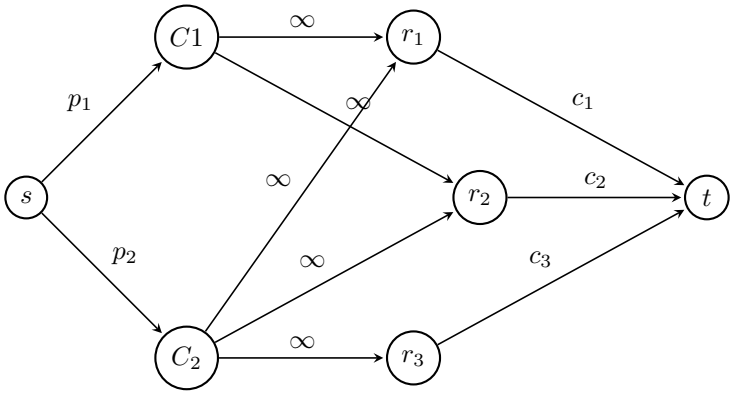
5.1 Request

Given n challenges C_i, $i \in \{1, .., n\}$ and m resources r_j, $j \in \{1, .., m\}$ of cost c_j where every challenge C_i offers p_i points when the game player has acquired all the resources required $(r_i, .., r_{\mathrm{n_i}})$ to complete it, we need to compute an algorithm that maximizes the profit, defined as the sum of acquired points minus the required costs of resources for those particular challenges. Every resource is bought only once and can be used for all challenges.

5.2 Solution

To find the solution to this problem we will build a network flow graph where challenges and resource cards represent nodes. The graph will be designed in the following way:

- We connect the challenge nodes C_i to the source node s. To each edge connecting that challenge to the source we assign as capacity the points p_i offered by that particular challenge
- We connect the resource nodes r_j to the sink t and assign as capacity the cost of using that particular resource card
- We connect the challenge nodes with the resource nodes in the following way: for each challenge C_i, $i \in \{1, .., n\}$ we add an edge directed to each resource r_j that the challenge in consideration uses. Because, for every challenge, we know that it uses all the resources from his own last one to the first one, we connect them in such way. It will never happen that a challenge C_i is connected to resource r_i, but not to r_{i-1} or previous resources until the first one. Every such edge is assigned with an infinite capacity.



On this graph we will calculate the min-cut, but in order to do this propertly we need some useful ingredients.

Definition 5.1. A st-cut is a partition (A,B) of the vertices with $s \in A$ and $t \in B$.
We define

$$cap(A, B) = \sum_{\text{e out of A}} cap(e)$$

as the sum of the capacities of the edges from A to B. The cut with minimum capacity is called min-cut.

Theorem 5.1. *To enforce the precedence constraints[3], the edges connecting the challenge nodes to the resource nodes have infinite capacity.*

Proof. Suppose by contradiction that I have a min-cut where, for challenge C_i, resource r_{i_j} is not included. Because every challenge is connected with an edge of infinite capacity to all its corresponding resources, we would have two nodes, $C_i \in A$ and $r_{i_j} \in B$, connected by an edge with infinite capacity. By definition 5.1 this would generate a min-cut with infinite capacity which is a contradiction. □

Definition 5.2. We define

$$P = \sum_{i=1}^{n} p_i$$

as the total revenue of the challenges and

$$C = \sum_{j=1}^{m} c_j$$

as the total cost of the cards.

[3] Every challenge, in order to be used, requires some specific resources.

By this definition we can see that the capacity of the cut ($\{s\}$,V\$\{s\}$) is P, so the maximum-flow value in this network is at most P.

Definition 5.3. Given a cut (A,B), it's capacity will be

$$cap(A,B) = \sum_{i|C_i \notin A} p_i + \sum_{j|r_j \in A} c_j$$

that is the sum of the capacities of the edges outgoing from the source to the challenges that don't belong to the cut plus the sum of the capacities of the edges outgoing from the resources to the sink that belong to the cut.

Definition 5.4. Given a cut (A,B), we define

$$profit(A) = \sum_{i|C_i \in A} p_i - \sum_{j|r_j \in A} c_j$$

as the sum of the capacities of the edges outgoing from the source to the challenges that belong to the cut minus the sum of the capacities of the edges outgoing from the resources to the sink that belong to the cut. By using the meaning that we gave to this capacities, we can interpret this value as the sum of the revenue given by the challenges that belong to A minus the sum of the cost of the resources that belong to A, that is exactly the notion of profit if the chosen challenges are the ones that belong to A.

Theorem 5.2. *The capacity of the cut (A,B), where A is the set of chosen challenges together with the needed resources, is*

$$cap(A,B) = P - profit(A)$$

Proof. Using definition 5.3 we know that any cut (A,B) has capacity

$$cap(A,B) = \sum_{i|C_i \notin A} p_i + \sum_{j|r_j \in A} c_j$$

We can write the first term as

$$\sum_{i|C_i \notin A} p_i = P - \sum_{i|C_i \in A} p_i$$

so the total capacity becomes

$$cap(A,B) = P - \sum_{i|C_i \in A} p_i + \sum_{j|r_j \in A} c_j$$

By using definition 5.4 we have proven that

$$cap(A,B) = P - profit(A)$$

□

Theorem 5.3. *If (A,B) is a minimum cut, then the set $A' = A - \{s, r_j\}$, where r_j are the resources that belong to A, is the set of optimum challenges.*

Proof. If cap(A,B) is a cut, by theorem 5.2, we know that

$$cap(A,B) = P - profit(A)$$

By being a minimum cut we know that

$$cap(A,B) = P - profit(A) = minValue$$

Because P is a constant by definition 5.2, we get that *profit(A)* is maximized. In fact we have that

$$profit(A) = P - minValue$$

Standing to definition 5.4, this value is the total revenue, so by finding the min-cut we have maximized it. The set $A' = A - \{s, r_j\}$ is the set of optimum challenges because it derives from the min-cut which, if chosen differently, would have given us a smaller value for the profit. We will prove this by contradiction and using as counter-example the graph that we drew before. Let's suppose that the optimum challenge is C1, while the needed resources are r1 and r2. Let's pretend, by contradiction, that the min-cut is made of:

- challenges that are optimum, C1, with resources that are not needed, r3. If this is the case, the min-cut is traversed by the edge (C2,r3) of ∞ capacity.
- challenges that are not optimum, C2, with resources that are needed by the optimum, r1 and r2. In this case the edge (C1,r1) with ∞ capacity would traverse the cut.
- challenges that are not optimum, C2, with resources that are not needed by the optimum, r3. The edge (C2,r1) with ∞ capacity traverses the cut.

In al three cases, by theorem 5.1, we would have a min-cut with infinite capacity which is a contradiction. □

Established all this key points, we can easily compute the min-cut using the Ford-Fulkerson algorithm. If we receive as input two lists: C that is the list of challenges and R that is the list of resources, we can use the following algorithm:

1. Build the graph as described in the beginning
2. Run Ford-Fulkerson on the graph
3. Retrieve the min-cut from the final residual graph
4. Remove the source and the resources from the min-cut

Theorem 5.4. *Algorithm is optimal and correct.*

Proof. By theorem 5.3. □

References

[1] Maximum subarray problem. https://en.wikipedia.org/wiki/Maximum_subarray_problem.

[2] Depth-first search in a tree. http://www.cs.yale.edu/homes/aspnes/pinewiki/DepthFirstSearch.html.