Algorithm Design Homework 01

Reference Book: Algorithm Design - Jon Kleingerb Eva Tardos

Eleonora Beatrice Rossi 1749038 - Andrea Dito 1844760

30/11/2022

1 Exercise 1

1.1 The Algorithm

The problem is the following: We have n kids and each of them has a level l, and we want to partition these kids into three groups that have exactly the same aggregated level, so we have to ensure that: $D(AggregatedSum) = \frac{1}{3}\sum_{i=1}^{n} l_i = \frac{sum}{3}$. Surely if the sum of all levels of all children is not divisible by 3, the partition cannot exist. After this check, to find three groups with the same aggregated sum our first greedy idea was to analyze all possible dispositions, but in this way the complexity would have been exponential.
We found a better solution thinking to the problem as a generalization of the knapsack problem from the SubsetSum algorithm (p.266 of the book), so the initial approach was to define a dynamic algorithm which takes in input an initial sequence of n kids $\{k_1, ..., k_n\}$, each with non negative level l_i with $i = 1, ...n$ and a bound D, and returns in output a subset in which the sum of all elements is exactly D.

$$OPT(i, D) = \begin{cases} OPT(i-1, D) & if D < l_i \\ max(OPT(i-1, D), k_i + OPT(i-1, D-l_i)) & \text{otherwise} \end{cases} \quad (1)$$

This problem can be solved in $O(n * D)$.

Notice that in this algorithm whenever an element is not useful it gets discarded. Instead, considering that the final algorithm has to split a sequence in three groups, we figured out that it was not possible to discard the useless elements but we needed instead to pass them to the other groups and check if there is a correct combination using all the elements of the sequence. So the final solution for our problem is:

$$OPT(i, a, b, c) =$$

$$(OPT(i-1, a-l_i, b, c)) \vee (OPT(i-1, a, b-l_i, c)) \vee (OPT(i-1, a, b, c-l_i))$$

Where the algorithm takes in input the kids' indexes and the value of levels for each group, called a,b and c (whose size at the beginning is D for every group) and returns in output a boolean value that is *TRUE* if the partitions exists and *FALSE* otherwise. Notice that the dynamic algorithm that we have just discussed is implemented through the use of a dictionary in the pseudocode 1. It is required in order to keep track of the combination already computed by the previous recursive calls and will also have a huge impact on the temporal complexity of the algorithm. In order to return the elements of each partition it is sufficient to use a list in which we assign to each element an integer (1, 2 or 3) during the recursions steps and print it accordingly to their associated group.

1.2 The complexity

Like previously seen, the algorithm based on a single bucket operates in a pseudo-polynomial time and its cost is based on both the number of the n kids and the researched bound D. When we generalize the algorithm to three buckets we must consider how the number of recursive calls increases. For this reason the complexity is equal to $O(n * (sum/3)^3) = O(n * sum^3)$. We know also that D (the aggregated sum) has to be greater than L (the maximum value that a level can assume) and it has to be less than $\frac{n*L}{3}$ that represents the case in which each kid has a level equal to L:

$$L \leq \frac{sum}{3} \leq \frac{n * L}{3}$$

So we can use the definition of Big-O notation and we can conclude that the complexity is:

$$O(n * sum^3) = O(n * (n * L)^3) = O(n^4 * L^3)$$

1.3 The correctness

Suppose that given a sequence of levels there exists a 3-Partition of it. Now suppose that our algorithm return FALSE, so it didn't find a solution for that sequence. This means that, if from the hypothesis a partition exists, and our algorithm finds three groups with different aggregated sum, therefore there will be at least one group with an aggregated sum greater than the bound D, and at least one with an aggregated sum lower than the bound D.

We can although prove that this situation is not reachable because our implementation before each recursive call does a conditional control on the remaining space in the group and on the value of each level, so that the size of each group can never over-exceed the value $\frac{sum}{3}$, and it is a contradiction from the initial statement. Also our algorithm will always find a possible solution if there is one because it recursively tries all the possible disposition of the sequence of each group.

Algorithm 1 3-Partiton pseudocode

```
1: Levels l_1 ... l_n
2: Kids 1... n
3: D is the sum each level divided by three
4: Dictionary at the beginning empty
5: List for printing the elements (empty at the beginning)
6:
7: 3-Partition(Levels , n, a ,b, c , Dictionary, List){
8:
9: sum = 0
10: for all l_i do
11:     Sum = sum + l_i
12: end for
13: if sum is not divisible by 3 then the partitions don't exist and return FALSE
14: end if
15: if n < 3 then the partitions don't exist and return FALSE
16: end if
17:
18: Base cases:
19: if a = 0 and b = 0 and c = 0 then we have found the partitions then return TRUE
20: end if
21: if n < 0 then we return FALSE
22: end if
23:
24: Recursive Calls:
25: key = (a,b,c,n) //in every recursive call we save the current a,b,c,n values.
26: if key is not in Dictionary then
27:
28:     A = FALSE
29:     if the element l_i fits in a then
30:         list[n] = 1
31:         A = 3-partition(Levels, n-1, a-l_i, b, c, Dictionary, List)
32:     end if
33:
34:     B = FALSE
35:     if not A and l_i fits in b then
36:         list[n] = 2
37:         B = 3-partition(Levels,n-1,a,b-l_i,c,Dictionary,List)
38:     end if
39:
40:     C = FALSE
41:     if If not A and l_i fits in c  then
42:         list[n] = 3
43:         C = 3-partition(Levels,n-1,a,b,c-l_i,Dictionary,List)
44:     end if
45:
46:     Dictionary[key] = A or B or C
47: end if
48:
49: Return dictionary[key]
50:
51: }
52:
53: Main(){
54: result = 3-Partition(Levels, n − 1, D, D, D, Dictionary, List)
55: if not result then the partition does not exist then return FALSE
56: end if
57:
58: print the List of kids for each group
59:
60: }
```

2 Exercise 2

In this problem we have to find a set of advertisers $A \subset N$, such that for each $n \in N$, either $n \in A$ or n is a friend of some $a \in A$. To prove that this problem in *NP*-complete we have to show that: the problem is *NP* and then choose another known *NP*-complete problem and prove that a reduction from the chosen *NP*-complete problem to our problem exists.

2.1 The problem is *NP*

Given a set people, to find if it is a set of advertisers A we have to check if each $n \in N$ is in A or if it has a friend in this group. If we find some n that return false for both conditions we have to stop, choose another group of people and restart the control-loop. The overall algorithm runs in polynomial time, therefore it is *NP*.

2.2 The choice of the NP-Complete known problem

We decided to stick to one of the *NP*-Complete problems seen more in depth during the lectures, in order to be more confident with the reduction step. For this reason at the beginning we have chosen the *SAT* problem but later on we figured out that we could apply a more rigorous reduction through the use of the *3SAT* problem, that is a special case of *SAT*, in which all clauses contain exactly three terms: the reason behind this choice comes from an analysis over the *3SAT*-independent set reduction that we observed in the book (p.460). That observation gave us the hint to visualize our problem as a graph structure where each $n \in N$ is a vertex and each friendship bond is an edge.

2.3 Reduction from 3-SAT (pictures in the following page)

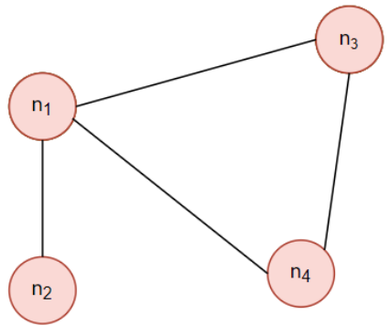
In order to apply a reduction we need first of all to associate the elements of the Advertisers problem to the elements of the chosen *NP*-Complete problem. We decided to represent the n people of our network N as the variables x_i or as its negation $\widehat{x}_i$ of the 3SAT problem. Their friendship link will be instead represented by the clauses of the formula, that will show, for three people at a time, if those variables must be considered as friends or not. We construct a graph G as follows: for each person n we introduce literal vertices x_i and $\widehat{x}_i$ and we connect these three vertices to form a triangle. Also in the graph each clause is represented by a vertex c_i and it is connected to its component terms (i.e the clause x_i, $\widehat{x}_j$, x_k will have an associated vertex c_i and will be connected with the vertices x_i, $\widehat{x}_j$, x_k).
The presence of the clause vertex is mandatory for our example, since it defines in the graph the friendship bond between the various triangle vertices. The creation of such a complex graph accomplishes both the necessities to complete our reduction and to find in an easy way the Advertiser Set A to it associated. In our problem an A set results in fact as the set of vertices elected as $a \in A$ and whose characteristic is the following: the union between A and their neighbours vertices return the entire graph G (it is basically what the problem is asking to us).
Considering the fact that A must not include twice the same person and in particular must not include both itself or its negation, for this reason it's necessary that whenever a clause contains x_i, it must not contains $\widehat{x}_i$ and vice versa.
Notice also the created A is composed only by the x_i variables there are set to true and the $\widehat{x}_i$ variables for each x_i that is false. Considering how it is defined, A will surely contain n vertices of our network and this ensures the fact that the entire graph is covered.

1. If we assume that there is a *3SAT* assignment that is satisfiable this means that:
   - each triangle contains exactly one node that is evaluated TRUE.
   - A contains exactly one node from each triangle.
   - Since every clause is true, this means that each clause-node c_i is adjacent to vertices in the Advertisers Set. Therefore this set must be correct since each vertex of the graph has been covered.

   Whenever instead the 3SAT formula is not satisfiable also A would not be correct.

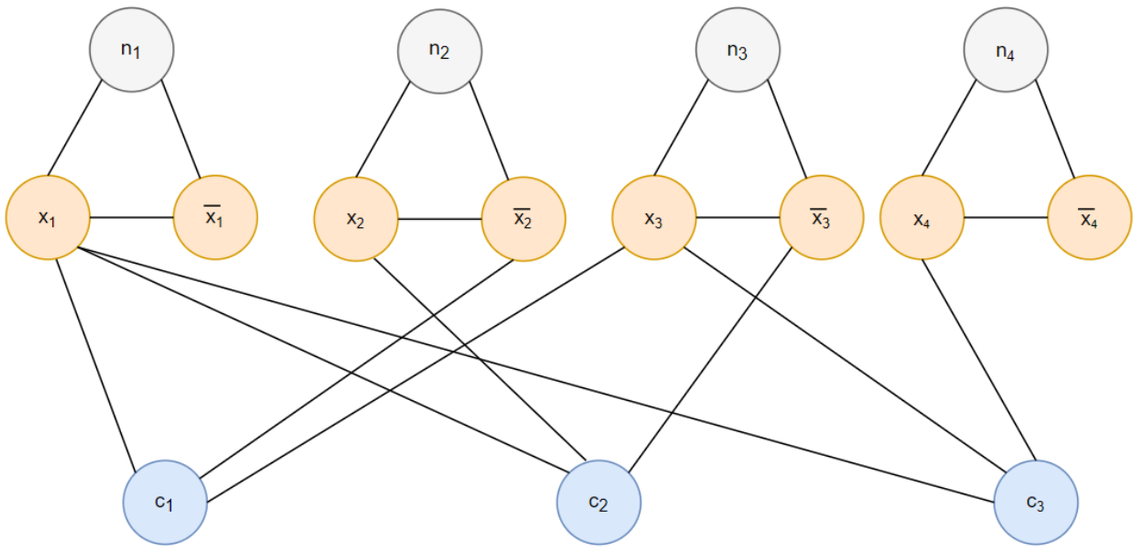
2. Suppose that a correct Advertisers Set, that has a size exactly n, exists, this means that this set must contain exactly one node of each triangle. Each vertex contained in A will be set as *TRUE* in the *3SAT* formula, *FALSE* otherwise. Now consider a clause-node c_i that contains x_i and $\widehat{x}_j$. If x_i is *TRUE*, x_i is in the advertisers set and the clause is *TRUE*, if $\widehat{x}_j$ is true this means that x_j is *FALSE* and the clause is again *TRUE*.

Since that given an Advertisers Set the *3SAT* formula is satisfied and given a *3SAT* satisfied formula the Advertisers set is correct the reduction is complete.



Original Graph

$$c_1 = x_1 \vee \widehat{x}_2 \vee x_3 \qquad c_2 = x_1 \vee x_2 \vee \widehat{x}_3 \qquad c_3 = x_1 \vee x_3 \vee x_4$$



3-SAT Reduction

3 Exercise 3

To solve this exercise we will use some basic probability definitions and theorems: we define the best threshold-strategy as the one that maximizes the expected value of the performance. Given the expected value formula and given the fact that the possible outcomes of the game are within an interval of real numbers we will not consider directly the reward x_i but we consider instead c_i as the central tendency of the interval $[\tau_i\ ,1]$, that is $\frac{\tau_i+1}{2}$, and p_i as the probability of obtaining the reward x_i, therefore the probability of exceeding τ_i.

$$\max E[X] = \max_{0\leq x\leq 1} \sum_{i=1}^{n} x_i p_i = \max_{\tau_i \leq c \leq 1} \sum_{i=1}^{n} c_i p_i$$

In order to maximize the expected value we will do its derivative and identify the best possible thresholds τ_i for our performance.

Dynamic threshold-strategy for n = 2

$$E[X] = c_1 p_1 + c_2 p_2$$

where $p_1 = P(x_1 > \tau_1) = 1 - P(x_1 < \tau_1) = 1 - \tau_1$ is the probability that the reward x_1 is greater than the threshold τ_1, while $p_2 = (1-p_1)P(x_2 > \tau_2) = \tau_1(1 - P(x_2 < \tau_2)) = \tau_1(1-\tau_2)$ is instead the probability of failing the first attempt ($x_1 < \tau_1$) and take the second reward x_2. When we arrive at the second attempt, after failing the first, we necessarily want to take x_2 because it is our last try and we would not obtain any reward otherwise. For this reason the threshold τ_2 is set equal to zero.

$$E[X] = (1-\tau_1)\tfrac{\tau_1+1}{2} + \tau_1(1-\tau_2)\tfrac{\tau_2+1}{2} = (1-\tau_1)\tfrac{\tau_1+1}{2} + \tfrac{\tau_1}{2}$$

Then we apply the derivative over this quantity and impose that is equal to zero and we obtain that the τ_1 that maximizes it is $\frac{1}{2} = 0.5$ and the maximum expected value is $\frac{5}{8}$.

$$E'[X] = \tfrac{-2\tau_1+1}{4} = 0 \quad \Rightarrow \quad \tau_1 = 0.5 \quad \Rightarrow \quad E[X] = (1-0.5)\tfrac{0.5+1}{2} + \tfrac{0.5}{2} = \tfrac{5}{8}$$

Dynamic threshold-strategy for n = 3

$$E[X] = c_1 p_1 + c_2 p_2 + c_3 p_3$$

As before p_1 is the probability of taking the first one, p_2 of failing the first and taking the second, while p_3 is the probability of succeeding the third time after failing twice. In that case we want to take x_3 because it is our last try and we would not obtain any reward otherwise, therefore $\tau_3 = 0$.

$$\begin{aligned} E[X] &= (1-\tau_1)\frac{\tau_1+1}{2} + \tau_1(1-\tau_2)\frac{\tau_2+1}{2} + \tau_1\tau_2(1-\tau_3)\frac{\tau_3+1}{2} \\ &= (1-\tau_1)\frac{\tau_1+1}{2} + \tau_1\left[(1-\tau_2)\frac{\tau_2+1}{2} + \tau_2(1-\tau_3)\frac{\tau_3+1}{2}\right] = (1-\tau_1)\frac{\tau_1+1}{2} + \tau_1\frac{5}{8} \end{aligned}$$

The reason why what is inside the square brackets is equal to $\frac{5}{8}$ is trivially because it is exactly the same calculus previously computed with $\tau_2 = 0.5$. As we will see later on this is a basic example of recursion where we can reuse part of equations already calculated. At this point we can compute the derivative to maximize the reward and find τ_1.

$$E'[X] = \tfrac{-8\tau_1+5}{8} = 0 \quad \Rightarrow \quad \tau_1 = \tfrac{5}{8} \quad \Rightarrow \quad E[X] = (1-\tfrac{5}{8})\tfrac{\frac{5}{8}+1}{2} + \tfrac{5}{8}\tfrac{5}{8} = \tfrac{89}{128} = 0.695$$

Static threshold-strategy for n = 1

For the static strategy the procedure is the same as the dynamic one with the difference that there aren't many τ_i values but there is only one τ. Trivially in the situation with n = 1 we want to take any reward x_1 that may occur, for this reason τ must be equal to zero.

$$E[X] = cp_1 = \tfrac{\tau+1}{2}P(x_1 > \tau) = \tfrac{\tau+1}{2}(1-\tau) = \tfrac{1}{2} = 0.5$$

Static threshold-strategy for n = 2

$$E[X] = cp_1 + cp_2 = \tfrac{\tau+1}{2}P(x_1 > \tau) + (1-p_1)P(x_2 > \tau) = \tfrac{\tau+1}{2}(1-\tau) + \tfrac{\tau+1}{2}\tau(1-\tau)$$

For n=2 the steps are almost identical to the ones computed for the other cases. The main difference comes from the fact that we do not have more τ_i, therefore there is the possibility that we do not take any reward, while in the dynamic strategy it could not happen.

$$E'[X] = -3\tau^2 - 2\tau + 1 \quad \Rightarrow \quad \tau = \tfrac{1}{3} \quad \Rightarrow \quad E[X] = \tfrac{\frac{1}{3}+1}{2}(1-\tfrac{1}{3}) + \tfrac{\frac{1}{3}+1}{2}\tfrac{1}{3}(1-\tfrac{1}{3}) = \tfrac{16}{27} = 0.592$$

Best dynamic threshold as a function of n

For this specific strategy we rely on a dynamic approach, since through the use of recursion we avoid calculating more than once the same calculi done in previous steps. In particular at each step n we compute the expected reward taking x_n plus the failure of the previous attempts whose expected value has been already calculated. Obviously in the case n=1, $\tau_1 = 0$ and $E[1] = 0.5$.

$$\max E[n] = \max((1-\tau_1)c_1 + \tau_1 E[n-1])$$

4 Exercise 4

4.1 The Algorithm

In order to solve the fourth exercise we base the algorithm on a Greedy Approach. The idea is to build one hospital at time and put it on the best possible position: that is the position that will cover the first considered house and that will have the biggest area of effect.
In the specific, the algorithm will look at the first house 0, found in location houses[0], and will build the first hospital at distance d from this house. In this way the algorithm will not only cover the first house but it will also put the hospital on the limit of its "range of effect", reaching as many following houses as possible. If the following houses are covered from this hospital the algorithm will not build a new hospital. Instead, if an house i (in position houses[i]) is located at a distance greater than d from Hospital[0] (in absolute value), a new hospital will be generated with the same principle: its position will be at houses[i] + d. To save the last position in which the last hospital is built we use an auxiliary variable to save that value and for comparing it with the positions of every house. In the end the algorithm returns an array containing integers that identify hospitals and their associated positions.

4.2 Complexity

The time complexity of this function is $O(k)$ because there is only one for-loop and the number of iterations is varying with the value of k, that is the number of houses. Inside the loop there are only statements that give a constant contribution.

4.3 Proof of Correctness

We have built a Greedy Algorithm, therefore we have to give a proof of correctness by comparing ALG vs OPT (our solution vs the optimal one). Let's define an algorithm OPT which builds the minimum number of hospitals for the provided sequence, a number that will be smaller than the one obtained by ALG. We suppose that OPT builds the hospitals $Hospital_1$ and $Hospital_2$ in the same position of our algorithm ALG, but generates for example the hospital $Hospital'_3$ in a different position than $Hospital_3$

$$ALG = \{Hospital_1, Hospital_2, Hospital_3...\}$$
$$OPT = \{Hospital_1, Hospital_2, Hospital'_3...\}$$

Considering how ALG works, we know that $Hospital_3$ is the hospital built at distance d from an house i (which is the first not covered from the previous $Hospital_2$). It will be the furthest possible distance (to the right) that still cover that house.

Therefore the $Hospital'_3$ generated by OPT will be located to the left of $Hospital_3$: it cannot be in the same position of $Hospital_3$ from hypothesis, and not to the right of $Hospital_3$ either because it would not cover the i-th house anymore.

Since it is located on the left of $Hospital_3$, this means either that it could still cover all the following houses covered by $Hospital_3$, generating a solution with the same number of hospitals, or it could instead not cover some of the following houses, making necessary to build a new one ($Hospital'_4$).

Although, if a new hospital is needed, this means that the algorithm OPT provides a solution that generates a number of hospitals greater or equal than the number of hospital generated by ALG and this is a contradiction for the hypothesis.

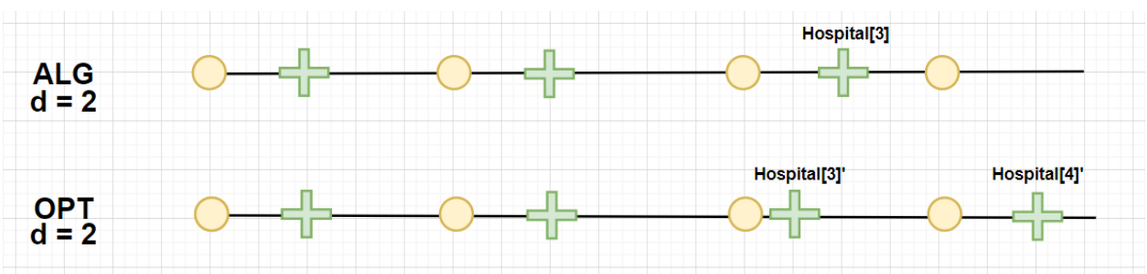


Figure 1: Correctness

Algorithm 2 Build Hospital Code

```
k number of houses
d max distance from an house to an hospital

int* BuildHospitals(k, d, int houses[]){
int hospitals[k]
int lastHospitalBuilt = -1
int l = 0

if k=0 then return hospitals
end if

for int i = 0; i < k; i++  do
    if (lastHospitalBuilt == -1)  then
        hospitals[l] = houses[i] + d;
        lastHospitalBuilt = houses[i] + d;
        l++;
        continue;
    end if
    if abs(houses[i] − lastHospitalBuilt) <= d then
        continue;
    else
        hospitals[l] = houses[i] + d;
        lastHospitalBuilt = houses[i] + d;
        l++;
    end if
end for

Return hospitals
```

5 Exercise 5

To solve this exercise we started from the second statement: the reason behind that is that if we can prove it, automatically the first one results proved as well.

5.1 Proof of claim b

In order to prove the second statement we remind what theory asserts about the max-flow min-cut theorem: *In every flow network, the maximum value of an s-t flow is equal to the minimum capacity of an s-t cut:*

$$\begin{aligned} v(f) &= f^{out}(A^*) - f^{in}(A^*) \\ &= \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e) \\ &= \sum_{e \text{ out of } A^*} C_e - 0 \\ &= C(A^*, B^*) \end{aligned} \tag{3}$$

The theorem basically states that, in a flow network, the maximum amount of flow passing from the source to the sink is equal to the total capacity of the edges in a minimum cut. We can now prove the second claim by contradiction:
We assume that the number that identifies the max-flow is an odd value but in the edge e* (the one having an odd capacity) there is not full flow $f_{e*} \neq c_{e*}$: from this hypothesis we know that e^* does not belong to the min-cut and that each edge present in the min-cut is instead an edge having even capacity. We can now consider that $\forall e \in$ out of $A^* : C_e = 2S_e$ (definition of even number):

$$v(f) = \sum_{e \text{ out of } A^*} C_e = \sum_{e \text{ out of } A^*} 2S_e = 2 * \sum_{e \text{ out of } A^*} S_e$$

$2 * \sum S_e$ is always an even number, so we proved by contradiction that since the max flow is an odd value, therefore we know from the max-flow min-cut theorem that there exists at least an edge in minimum cut whose capacity is an odd number. The exercise states that there is exactly one edge e* having an odd capacity, therefore that edge has to be in the min-cut. This is true because the sum of an even number and an odd number is always odd, so if we define an even number as *2m* and an odd number as *2n+1* where m and n are integers, we can prove this statement with the following equality:

$$v(f) = \sum_{e \text{ out of } A^*} C_e + C_{e^*} = 2m + 2n + 1 = 2(m+n) + 1$$

Due to the fact that m and n are integers we can redefine $(m+n) = k$ where k is an integer so the result is *2k +1* that is by definition an odd number.

Even if the hypothesis says that $f_{e*} \neq c_{e*}$, we demonstrated that e* has to be in the min-cut, therefore that edge must necessarily have a full flow $f_{e*} = c_{e*}$ and we have fallen in a contradiction. The second claim results proved, in fact in every maximum flow, there is a full flow in the edge with an odd capacity.

5.2 Proof of claim a

The first claim results demonstrated as well: since we have proven that in every maximum flow there is a full flow in the edge with an odd capacity, it follows that the flow passing from that edge has to be greater than 0.