

# Contents

1 [Randomized selection]	2
2 [Randomized min-cut algorithm]	4
3 [External memory mergesort]	7
4 [Family of uniform hash functions]	9
5 [Deterministic data streaming]	11
6 [Special case of most frequent item in a stream]	13
7 [Count-min sketch: extension to negative counters]	14
8 [Count-min sketch: range queries]	15
9 [Succinct data structure for range queries]	16
10 [External memory implicit searching]	17
11 [Number of splits for $(a, b)$ -trees]	21
12 [1-D range query]	25
13 [External memory permuting]	26
14 [Lower bound for searching]	29
15 [Cache-oblivious selection]	30
16 [Implicit navigation in vEB layout]	31
17 [MapReduce indegree distribution]	32
18 [MapReduce prefix sums]	34
19 [Suffix sorting in EM]	36
20 [Euler tour]	39
21 [Wrong greedy for minimum vertex cover]	42
22 [Polynomial reduction from NAE-3-SAT to MAX-CUT]	44
23 [Approximation for MAX-SAT]	45
24 [Greedy 2-approximation for MAX-CUT on weighted graphs]	48
25 [Randomized 2-approximation for MAX-CUT]	51

# 1 [Randomized selection]

Consider the **randomized quicksort**, analyzed with the indicator variables, discussed in class.

- a. Show how to modify the randomized quicksort so that, given an array  $A$  and an integer  $1 \leq k \leq |A|$ , it finds the  $k^{th}$  largest element in  $A$  without fully sorting  $A$ .

---

**Algorithm 1** Randomized Selection

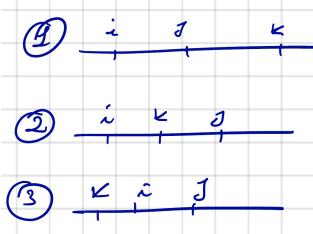
---

```
RandSelect(array A, length n, element k){  
    if (n=1) return A[0];  
    choose (uniformly at random) pivot p from A  
    partition A around p  
        /** everything bigger than the pivot to the right,  
         everything smaller than the pivot to the left **/  
    let j = index of p  
    if (j = k)  
        return p; //  
    if (j > k)  
        return RandSelect(A.left , j-1, k);  
    if (j < k)  
        k <- k - j -1  
        return RandSelect(A.right , n-1-j , k);  
}  
}
```

---

- b. Consider the analysis with indicator variables seen in class, and adapt it to show that the selection algorithm thus obtained requires linear expected time. [hint: since the algorithm has  $A$  and  $k$  as input, define an indicator variable  $X_{ijk}$  for  $z_i$  and  $z_j$ , where  $i < j$ , with the knowledge that  $z_k$  is looked for. The probability will have three cases, according to the relative order of  $z_k$  with respect to  $z_i$  and  $z_j$ .]

- sia  $X_{ijk} = \begin{cases} 1 & \text{se } z_i \text{ e } z_j \text{ sono confrontati quando viene trovato } z_k \\ 0 & \text{altrimenti} \end{cases}$
- considerando che  $P_r[z_i \text{ e } z_j \text{ sono confrontati quando viene trovato } z_k] \leq P_r[\text{uno dei due è il pivot}] = \frac{2}{|blocks|}$
- Dividiamo in tre casi il problema in questione (i casi coprono tutte le possibilità e non si sovrappongono):



la  $P_r [z_i \text{ e } z_j \text{ sono confrontati quando viene trovato } z_k]$  vale  $\begin{cases} \frac{2}{k-i+1} & \text{se } k > j \\ \frac{2}{j-i+1} & \text{se } i < k < j \\ \frac{2}{j-k+1} & \text{se } k < i \end{cases}$

$$1. \sum_{i=1}^{k-2} \sum_{j=i+1}^{k-1} \frac{2}{k-i+1} \approx \sum_{i=1}^{k-2} \sum_{j=i+1}^{k-1} \frac{1}{k-i+1}$$

(a) sostituisco  $j' = j - i$ , ottenendo  $\sum_{i=1}^{k-2} \sum_{j'=1}^{k-i} \frac{2}{k-i+1}$

(b) sostituisco  $i' = k - i$ , ottenendo  $\sum_{i'=1}^{k-2} \sum_{j'=1}^{i'} \frac{2}{i'+1}$

(c) considerando che l'argomento della sommatoria non dipende più da  $j'$  si

$$\text{ottiene } \sum_{i'=1}^{k-1} \frac{2i'}{i'+1} \leq \sum_{i'=1}^{k-1} \frac{i'}{i'} \rightarrow O(n)$$

$$2. \sum_{i=1}^k \sum_{j=k+1}^n \frac{2}{j-i+1} \approx \sum_{i=1}^k \sum_{j=k+1}^n \frac{1}{j-i+1}$$

(a) sostituisco  $j' = j - i$ , quindi  $j = j' + i$ , ottenendo  $\sum_{i=1}^k \sum_{j'=k-i+1}^{n-i} \frac{1}{j'+1}$

(b) a questo punto è necessario ragionare in termini geometrici:

$$3. \sum_{i=k+1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-k+1} \approx \sum_{i=k+1}^{n-1} \sum_{j=i+1}^n \frac{1}{j-k+1}$$

(a) Se  $j$  va da  $i+1$  con  $i$  che parte da  $k+1$  allora  $j$  in realtà assumerà valori compresi tra  $k+2$  e  $n$ .

(b) Quindi è possibile maggiorare:  $\sum_{i=k+1}^{n-1} \sum_{j=i+1}^n \frac{1}{j-k+1} \leq \sum_{i=k+1}^j \sum_{j=k+2}^n \frac{1}{j-k+1}$

(c) L'argomento della sommatoria non dipende più da  $i$  quindi la prima sommatoria verrà ripetuta  $j - (k+1)$  volte:  $\sum_{j=k+2}^n \frac{j-(k+1)}{j-k+1} \leq \sum_{j=k+2}^n \frac{j-k}{j-k}$

## 2 [Randomized min-cut algorithm]

- a. Consider the **randomized min-cut algorithm** discussed in class. We have seen that its probability of success is about  $\frac{2}{n^2}$ .
- b. Describe how to implement the algorithm when the graph is represented by adjacency lists, and analyze its running time.
- Una lista di adiacenza è una modalità di rappresentazione di grafi, secondo la quale ad ogni vertice  $V$  viene associata una lista contenente tutti e soli i vertici  $W$  tali che esista l'arco da  $V$  a  $W$ .
  - Problemi riscontrati nell'algoritmo considerando questa rappresentazione:
    - Aumento della complessità, che non è più  $O(n^2)$ , ma  $O(n^3)$  poiché, durante la contrazione, l'aggiornamento delle liste di adiacenza (dovuto alla creazione del nuovo nodo) costa  $O(n \cdot \text{grado})$  con  $O(\text{grado}) = O(n)$
    - Uno dei metodi per risolvere questi problemi è quello di considerare gli elementi della lista composti da due campi: indice del nodo e puntatore ad esso. In questo modo durante la contrazione, l'aggiornamento di tutte le liste dei nodi adiacenti ai due nodi contratti può essere effettuato in  $O(n)$  perché durante la scansione delle liste dei nodi contratti vengono aggiornate direttamente tutte le altre liste, tramite il puntatore al nodo.
    - Ripetere  $n$  volte
- c. A weighted graph has a weight  $w(e)$  on each edge  $e$ , which is a positive real number. The min-cut in this case is meant to be min-weighted cut, where the sum of the weights in the cut edges is minimum. Describe how to extend the algorithm to weighted graphs, and show that the probability of success is still  $\frac{2}{n^2}$ . [hint: define the weighted degree of a node]
- weighted degree of a node  $n \sum_{e \in \text{edges}(n)} w(e)$
  - un bad edge è anche in questo caso un arco che appartiene a tutti i min cut. Infatti possiamo vedere questo problema come quello generale senza pesi sugli archi in cui consideriamo il multigrafo che per ogni arco di peso  $w$  considera  $w$  archi con peso 1. Quindi contrarre un bad edge nel caso pesato equivale a contrarre  $w$  bad edge nel caso generale.
  - Data questa considerazione, è possibile riformulare la dimostrazione della probabilità di successo del randomized min-cut:
    - Sia  $W = \sum_{e \in E(C)} w(e)$  allora ci sono  $\leq W$  bad edge sul taglio, infatti considerando il caso limite in cui tutti gli archi del taglio hanno peso 1, il numero di archi è al più  $\sum_{e \in E(C)} w(e)$  -> quindi non possono esserci più bad edge di quanti sono gli archi del taglio.
    - A differenza del caso non pesato, la scelta non è uniforme, ma la probabilità di scelta di un arco viene rapportata al peso dell'arco stesso. (+ pesante -> + probabile sceglierlo)

- $pr(arco\ cattivo) \leq \frac{\#\text{weight\ badedges}}{\#\text{total\ weight}} \leq \frac{W}{\#\text{total\ weight}}$  con  $\#\text{total\ weight} \geq \frac{nW}{2}$   
poiché il grafo ha  $n$  nodi, ognuno dei quali può avere  $W$  come grado pesato.

- d. Show that running the algorithm for  $N = cn^2 \ln(n)$  times, for a constant  $c > 0$ , and taking the minimum among the  $N$  min-cuts thus produced, the probability of success can be made at least  $1 - \frac{1}{n^c}$  (hence, with high probability). [hint: use the fact that  $(1 - \frac{1}{x})^x \approx e^{-1}$  for big  $x$ .]

- The RandMin-Cut is not very good. Fortunately, there's a simple method for increasing our chances of finding the minimum cut: run the guessing algorithm many times and return the smallest guess.
- Eseguendo l'algoritmo  $N$  volte invece è possibile migliorare il RandMin-Cut.
- Abbiamo dimostrato che la probabilità che un arco sia cattivo è data da  $pr(arco\ cattivo) \leq \frac{2}{n}$ ; quindi la probabilità che un arco sia buono è  $pr(arco\ buono) \geq 1 - \frac{2}{n} \geq \frac{n-2}{n}$
- La probabilità che l'algoritmo di random min-cut ha successo (cioè che faccia sempre la scelta giusta),  $pr(RandMin - Cut\ ha\ successo)$ , equivale alla probabilità che esso abbia successo su ognuno degli  $n$  nodi di cui è composto il grafo:  $pr(algo\ ha\ successo\ su\ n\ nodi) \rightarrow$  che scriverò per semplicità come  $pr(n)$ .
- La probabilità che l'algoritmo abbia successo su  $n$  nodi è data dalla probabilità che un arco sia buono per la probabilità che l'algoritmo abbia successo su  $(n-1)$  nodi:
  - $pr(n) \geq pr(arco\ buono) \cdot pr(n-1) \geq \frac{n-2}{n} \rightarrow$  sostituendo:  $pr(n) \geq \frac{n-2}{n} \cdot pr(n-1)$
- e così via, cioè la probabilità che l'algoritmo abbia successo su  $(n-1)$  nodi è data dalla probabilità che un arco sia buono per la probabilità che l'algoritmo abbia successo su  $(n-2)$ :  $pr(n-1) \geq pr(arco\ buono) \cdot pr(n-2)$  e così via
- $p(n) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \dots \cdot 1 \rightarrow \prod_{i=3}^n \frac{i-2}{i} = \frac{\prod_{i=3}^{n-2} i}{\prod_{i=3}^n i} = \frac{\prod_{j=1}^{n-2} j}{\prod_{i=j}^n i} = \frac{1 \cdot 2 \cdot 3 \dots (n-2)}{3 \cdot 4 \dots (n-2) \cdot (n-1) \cdot n} = \frac{2}{(n-1) \cdot n}$
- $Pr(RandMin - Cut\ sbaglia) \leq \left(1 - \frac{2}{(n-1) \cdot n}\right)$
- RandMin-Cut ripetuto  $N$  volte sbaglia quando sbaglia  $N$  volte RandMin-Cut semplice:  $\left(1 - \frac{2}{(n-1) \cdot n}\right)^N$
- $Pr(RandMin - Cut\ ripetuto N\ corretto) \geq \left(1 - \left(1 - \frac{2}{(n-1) \cdot n}\right)^N\right)$
- Se  $N = cn^2 \ln(n)$  allora  $1 - \left(1 - \frac{2}{(n-1) \cdot n}\right)^{cn^2 \ln(n)}$ ; considero  $\frac{2}{(n-1) \cdot n} \approx \frac{1}{n^2}$ , quindi per  $n \rightarrow \infty$ ,  $\left(1 - \frac{1}{n^2}\right)^{n^2} \rightarrow e^{-1}$ .
- Infine la  $Pr(RandMin - Cut\ ripetuto N\ corretto) \geq 1 - \frac{1}{e^{cn(n)}} = 1 - \frac{1}{n^c}$

- e. Optional: when the graph becomes smaller, the probability of hitting a bad edges is higher. What if the algorithm is stopped when the resulting graph contains half of the original vertices? Let us run this four times independently starting from the same graph  $G$ , obtaining four graphs  $G_1, G_2, G_3, G_4$ , each with  $\frac{n}{2}$  vertices each. Apply recursively this to each  $G_i$  independently. At the end, the best min-cut thus found among all those generated. Show what is the time complexity.

### 3 [External memory mergesort]

In the external-memory model (hereafter EM model),

- a. show how to implement the k-way merge (where  $(k + 1)B \leq M$ ), namely, how to simultaneously merge  $k$  sorted sequences of total length  $N$ , with an I/O cost of  $O(\frac{N}{B} = n)$  where  $B$  is the block transfer size. Date  $k$  sequenze individualmente ordinate, dette run, il k-way Merge mantiene in memoria principale  $k$  buffer di input e un buffer di output, tutti di dimensione  $B$ .

Sotto la condizione  $(k + 1) \times B \leq M$  il k-way merge può essere effettuato mantenendo in Memoria Principale i  $k$  blocchi correnti dei runs che si vogliono mergare e il blocco di output corrente.

A ogni passo seleziona fra i buffer di input attivi l'elemento più piccolo e lo sposta nel buffer di output; quando questo risulta pieno il suo contenuto viene copiato in memoria esterna. Ogni volta che un buffer di input è arrivato al termine l'algoritmo copia un nuovo blocco dal corrispondente run; se questo non è più possibile il buffer viene marcato come inattivo e non viene più considerato nella selezione dell'elemento più piccolo. Quando infine rimane un solo run attivo viene copiato il suo contenuto in memoria esterna.

---

#### Algorithm 2 k-way merge EM model

---

```
Inizializza k buffer di input contenenti ciascuno il primo blocco  
del corrispondente run e marca ogni buffer come attivo  
  
while (ci sono almeno due buffer di input attivi) do  
    trova il più piccolo elemento nei buffer attivi  
    sposta questo elemento nella prima posizione libera del buffer  
    di output  
  
    if (buffer di output pieno) then  
        svuotalo in memoria esterna e reinizializzalo  
  
    if (buffer di input da cui preleva l'elemento è terminato) then  
        copia in tale buffer il prossimo blocco  
        del run corrispondente  
  
    if (run terminato) then  
        marca quel buffer come inattivo  
  
copia l'ultimo run attivo in memoria esterna
```

---

- **I/O Complexity** =  $Blocchi\ letti + Blocchi\ scritti = \frac{N}{B} + \frac{N}{B} = O\left(\frac{N}{B}\right)$

- b. Also, try to minimize and analyze the CPU time cost (nel solo caso di k-merge)

- Nella terza riga dello pseudo codice troviamo ripetutamente il minimo con una scansione lineare dei primi elementi di ogni buffer; in questo modo paghiamo a ogni passo  $O(k)$  confronti, per un costo totale di  $O(N \cdot k)$ .
- Per una ricerca efficiente della chiave minima è possibile mantenere la prima chiave di ogni run all'interno di un albero (MinHeap). In questo modo il costo per prelevare la chiave minima nel complesso è:
  - **CPU Complexity** = Numero di inserzioni · Costo + Numero di estrazioni.  

$$\text{Costo} = N \cdot \log(k) + N \cdot \log(k) = O(N \cdot \log_2 k)$$

c. Using the above k-way merge, show how to implement the EM mergesort and analyze its I/O complexity and CPU complexity.

- Il costo di un algoritmo merge sort facente uso del k-way merge prevede la lettura di sequenze di dimensione  $M$  in memoria principale e l'ordinamento di tale sequenze utilizzando un algoritmo in memoria principale. Successivamente viene applicata la procedura di k-way merge che provvede ad unire  $k$  sequenze in 1 singola sequenza; tale procedimento viene applicato fino ad ottenere un'unica sequenza che consiste nell'array ordinato.
- Calcolo dell'I/O complexity:
  - La fase di lettura, ordinamento e scrittura di  $\frac{N}{M}$  sequenze ha costo  $O\left(\frac{N}{B}\right)$ .
  - La fase di k-way merge unisce  $k = \frac{M}{B}$  sottosequenze in una e, come detto in precedenza, ha costo  $O\left(\frac{N}{B}\right)$ .
  - La seconda fase viene ripetuta fino ad ottenere un'unica sequenza, ossia  $\log_k\left(\frac{N}{M}\right)$  volte.
- **I/O Complexity** =  $O\left(\frac{N}{B}\right) + \log_k\left(\frac{N}{M}\right) \cdot O\left(\frac{N}{B}\right) = O\left(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{M}\right)$ .

## 4 [Family of uniform hash functions]

Show that the family of hash functions  $H = \{h(x) = ((ax + b) \text{ mod } p \text{ mod } m)\}$  is (almost) “pairwise independent”, where  $a, b \in [m]$  ( $m$  è il numero degli slot della hash table) with a  $a \neq 0$  and  $p$  (dimensione dell'universo delle chiavi) is a sufficiently large prime number ( $m + 1 \leq p \leq 2m$ ).

The notion of pairwise independence says that:

for any  $x_1, x_2$  and  $c1, c2 \in [m]$ , we have that

$$\Pr_{h \in H} [h(x_1) = c_1 \wedge h(x_2) = c_2] = \Pr_{h \in H} [h(x_1) = c_1] \cdot \Pr_{h \in H} [h(x_2) = c_2].$$

In other words, the joint probability is the product of the two individual probabilities.

### Soluzione

- Dobbiamo mostrare che, presa una funzione hash nella famiglia ( $h \in H$ ), prese due chiavi  $k, l$  con  $k \neq l$  e due valori  $u, v \in [m]$ 
  - vale  $\Pr_{h \in H} [h(k) = u \wedge h(l) = v] \approx \Pr_{h \in H} [h(k) = u] \cdot \Pr_{h \in H} [h(l) = v]$ .
- Per farlo, mostreremo che (considerando solo la prima parte della funzione hash, ovvero  $r(k) = ak + b \text{ mod } p$ ).
  1.  $\Pr_r [r(k) = u] \approx P[r(l) = v] \approx \frac{1}{m}$
  2. e che  $\Pr_r [r(k) = u \wedge r(l) = v] \approx \frac{1}{m^2}$ ; dalla definizione di pairwise independent.

Voglio, in sostanza, dimostrare che  $r_{a,b}(k) = ak + b$  è pairwise independent.

- Per dimostrare il primo, basta calcolare  $\Pr_r [r(k) = u]$  che è data dal numero di soluzioni di  $ak + b \equiv u \pmod{p}$  diviso il numero di possibili scelte per  $a$  e  $b$ .
  - Le possibili soluzioni di  $ak + b \equiv u \pmod{p}$ , posso trovarle notando che in  $b \equiv u - ak \pmod{p}$ , le possibili scelte di  $a$  sono  $m$ .
  - Il numero totali di scelte di  $a$  e  $b$  è dato da  $m \cdot (m - 1) \approx m^2$
  - Quindi  $\Pr_{h \in H_r} [r(k) = u] \approx \frac{m}{m^2} \approx \frac{1}{m}$
- Per dimostrare il secondo e quindi calcolare  $\Pr_{h \in H} [r(k) = u \wedge r(l) = v]$  invece, è necessario considerare il seguente sistema:
  - $\begin{cases} u \equiv ak + b \pmod{p} \\ v \equiv al + b \pmod{p} \end{cases}$
  - Noto che  $(u \equiv ak + b) \text{ mod } p \neq (v \equiv al + b) \text{ mod } p$ .

- \* questa congruenza vale poiché se fosse  $ak + b \equiv al + b \pmod{p}$  allora  $a(k - l) \equiv 0 \pmod{p}$ , ma questo non può succedere poiché  $p$  è primo e  $a \neq 0$  e  $(k - l) \neq 0$  dato che  $k \neq l$ .
- \* quindi  $u \neq v$

Il sistema ha un' unica soluzione (dal Chinese Remainder Theorem,  $u - v \equiv a(k - l) \pmod{p}$ ).

- La probabilità è di nuovo data dal numero di soluzioni diviso il numero totali di scelte per  $a$  e  $b$ 
  - Il numero di soluzioni è uno visto che c'è solo una soluzione
  - Il numero di scelte totali per  $a$  e  $b$  è dato da  $m \cdot (m - 1) \approx m^2$
  - $P_r[r(k) = u \wedge r(l) = v] = \frac{1}{m \cdot (m - 1)} \approx \frac{1}{m^2}$

A questo punto è necessario tener conto della seconda parte della funzione hash, ovvero del  $\pmod{m}$ .

- Notando che  $P_r[h(k) = u] = Pr[r(k) = u] + Pr[r(k) = u + m]$ 
  - nel caso in cui  $u + m < p$  quindi  $u < p - m$  (altrimenti il valore  $u + m$  viene eliminato dall'operazione di modulo in  $p$ )
    - \* in questi due casi si ha che  $Pr[h(k) = u] = \frac{1}{m} + \frac{1}{m} = \frac{2}{m}$  poiché i valori di  $r(k)$  che vengono mappati in  $u$  sono  $u$  e  $u + m$ .
  - Infine è necessario mostrare che  $Pr[h(k) = u \wedge h(l) = v] = Pr[h(k) = u] \cdot Pr[h(l) = v]$  e per far ciò è necessario considerare il caso in cui  $u < p - m$  e  $v < p - m$  e i seguenti sistemi lineari generati:

$$1. \begin{cases} r(k) = u \\ r(l) = v \end{cases}$$

$$2. \begin{cases} r(k) = u + m \\ r(l) = v \end{cases}$$

$$3. \begin{cases} r(k) = u \\ r(l) = v + m \end{cases}$$

$$4. \begin{cases} r(k) = u + m \\ r(l) = v + m \end{cases}$$

ognuno dei quali ha una soluzione.

Il risultato finale è  $P_{h \in H}[h(k) = u \wedge h(l) = v] \leq \frac{4}{m \cdot (m - 1)} \approx \frac{1}{m^2}$ .

## 5 [Deterministic data streaming]

Consider a stream of  $n$  items, where items can appear more than once in the stream. The problem is to find the most frequently appearing item in the stream (where ties broken arbitrarily if more than one item satisfies the latter). Suppose that only  $k = O(\log^c n)$  items can be stored, one item per memory cell, where the available storage is  $k + O(1)$  memory cells.

a. Show that the problem cannot be solved deterministically under the following rules:

- the algorithm can access only  $O(\log^c n)$  information for each of the  $k$  items that it can store,
- can read the next item of the stream;
- you, the adversary, have access to all the stream, and the content of the  $k$  items stored by the algorithm,
- you can decide what is the next item that the algorithm reads (please note that you cannot change the past, namely, the items already read by the algorithm) .

**Hint:** it is an adversarial argument based on the  $k$  items chosen by the hypothetical determinist streaming algorithm, and the fact that there can be a tie on  $> k$  items till the last minute.

### Soluzione

Gli algoritmi deterministici posso essere definiti in termini di macchine a stati: uno stato descrive cosa una macchina fa in un particolare istante di tempo.

In questo caso particolare, se  $k$  sono le celle di memoria e  $k$  sono i bit memorizzati dall'algoritmo, allora gli stati che l'algoritmo può assumere sono  $2^k$ .

Supponiamo per assurdo che un algoritmo deterministico, per trovare l'algoritmo più frequente, esista.

- Se diamo a questo algoritmo come input  $2^k+1$  volte lo stesso elemento, l'algoritmo si troverà necessariamente due volte nello stesso stato (stato replicato) in due momenti diversi.
- Chiamo  $A_i$  e  $A_j$  i rispettivi stati dell'algoritmo A dopo che ha ricevuto l'elemento per la  $i$ -esima e per la  $j$ -esima volta rispettivamente (con  $A_i$  stato precedente rispetto ad  $A_{j-1}$ , posso assumere giusto per semplicità di notazione che  $i$  e  $j$  siano anche le iterazioni dell'algoritmo quindi  $i < j - 1$ ). ( $A_i$  e  $A_j$  sono quindi quello stato replicato menzionato prima)
- A questo punto diamo ad A in input  $(i + 1)$  volte un elemento diverso dal precedente.
- Gli algoritmi risponderanno in modo diverso riguardo l'elemento più frequente poiché:

- l'algoritmo nello stato  $i$  ( $A_i$ ) ha visto  $i$  volte il primo elemento e  $i + 1$  volte il secondo, →ha visto più volte il secondo
- invece l'algoritmo nello stato  $j$  ( $A_j$ ) ha visto  $j$  volte (con  $j > i + 1$ ) il primo elemento ma solo  $i + 1$  volte il secondo →ha visto più volte il primo

Rimane da discutere il caso  $i = j - 1$ , in cui i due stati assunti sono consecutivi nel tempo, ma in realtà sono lo stesso stato della macchina (note che l'algoritmo ha input sempre lo stesso elemento).

Se  $j$  è lo stato successivo ad  $i$  significa che esiste la transizione  $(A_i, x) \rightarrow (A_j)$ , è quindi possibile notare che l'algoritmo resterà ancora nello stesso stato ricevendo in input altre volte lo stesso elemento. In particolare  $A_{i+2} = \delta(\delta(A_i, x), x) = \delta(A_i, x) = A_i$  (cioè è uguale allo stato  $A_i$ ); con  $\delta(A_i, x)$  uguale alla transizione  $(A_i, x) \rightarrow (A_j)$ . Con questa considerazione, ci ritroviamo nella stessa condizione descritta precedentemente dove  $j > i + 1$ .

## 6 [Special case of most frequent item in a stream]

Suppose to have a stream of  $n$  items, so that one of them occurs  $> \frac{n}{2}$  times in the stream. Also, the main memory is limited to keeping just two items and their counters, plus the knowledge of the value of  $n$  beforehand. Show how to find deterministically the most frequent item in this scenario.

[Hint: since the problem cannot be solved deterministically if the most frequent item occurs  $\leq \frac{n}{2}$  times, the fact that the frequency is  $> \frac{n}{2}$  should be exploited.]

### Soluzione

Un algoritmo che deterministicamente può determinare l'elemento più frequente (utilizzando una cella di memoria per l'elemento più frequente e una per il contatore) prevede il mantenimento di un contatore che viene decrementato e incrementato come segue:

- decrementato ogni volta che l'elemento successivo è diverso dal corrente
- incrementato ogni volta che l'elemento successivo è uguale al corrente

Come viene aggiornato l'elemento più frequente?

1. Ogni volta che arriva un elemento, se è un nuovo elemento (diverso da quello nella cella), viene memorizzato nella cella come elemento più frequente;
2. Se l'elemento è sempre uguale allora nella cella viene chiaramente lasciato il solito elemento.

Quando il contatore raggiunge lo zero, la cella viene lasciata vuota finché non arriva il nuovo elemento (a questo punto ci ritroviamo nel primo caso).

Il nuovo elemento verrà infatti memorizzato nella cella come elemento più frequente (vengono quindi “eliminati” entrambi gli elementi precedenti).

Da ipotesi l'elemento più frequente è presente più di  $\frac{n}{2}$  volte, ma allora il caso più sfavorevole (possiamo discutere questo caso senza perdita di generalità) è quando ci sono  $\frac{n}{2} - 1$  scontri fra elementi distinti di cui uno è il più frequente (caso in cui sono alternati il più frequente con un altro elemento).

Sotto questa condizione quindi il contatore verrà decrementato  $\frac{n}{2} - 1$  volte, ma essendo presente almeno  $\frac{n}{2} + 1$  volte, almeno nell'ultimo confronto (il contatore avrà raggiunto il valore zero), verrà scelto come elemento più frequente.

Ovviamente il ragionamento vale anche in tutti i casi diversi dal caso pessimo; infatti è sufficiente notare che gli elementi diversi dal più frequente vengono annullati ad ogni confronto.

## 7 [Count-min sketch: extension to negative counters]

Check the analysis seen in class, and discuss how to allow  $F[i]$  to change by arbitrary values read in the stream.

Namely, the stream is a sequence of pairs of elements, where the first element indicates the item  $i$  whose counter is to be changed, and the second element is the amount  $v$  of that change ( $v$  can vary in each pair). In this way, the operation on the counter becomes  $F[i] = F[i] + v$ , where the increment and decrement can be now seen as  $(i, 1)$  and  $(i, -1)$ .

### Soluzione

Per il count min sketch a valori arbitrari vale:

$$P_r \left[ F[i] - 3\epsilon \|F\| \leq \tilde{F}[i] \leq F[i] + 3\epsilon \|F\| \right] \geq 1 - \delta^{\frac{1}{4}}$$

$$\text{con } \tilde{F}[i] = \text{median} \{T[h_j(i)]\}$$

In realtà la diseguaglianza vale anche in questa variante. La differenza sta nella definizione di  $F$ , stavolta anzichè mantenere in  $F[i]$  le frequenze dell'elemento  $i$  – esimo (cioè il numero di volte che  $i$  appare nello snapshot corrente dello streaming S di dati),  $F[i]$  può essere definito come un accumulatore. Ovvero in esso viene mantenuta comunque la frequenza dell'elemento, ma considerando anche il valore.

Si può facilmente notare che l'unico termine che subisce una modifica è proprio la  $\|F\| = \sum_k F[k]$  che stavolta avrà un valore maggiore rispetto al semplice contatore considerato nel caso del min sketch a valori arbitrari.

Ma è un termine che non incide nella dimostrazione, costruita ad hoc, utilizzando la diseguaglianza di Markov, per eliminarlo.

## 8 [Count-min sketch: range queries]

Show and analyze the application of count-min sketch to range queries  $(i, j)$  for computing  $\sum_{k=i}^j F[k]$ .

Hints:

- Reduce the latter query to the estimate of just  $t \leq 2 \cdot \log n$  counters  $c_1, c_2, \dots, c_t$  .
- Note that in order to obtain a probability at most  $\delta$  of error (i.e. that  $\sum_{l=1}^t c_l > \sum_{k=i}^j F[k] + 2 \cdot \log n ||F||$ ),

it does not suffice to say that it is at most  $\delta$  the probability of error of each counter  $c_l$  :

while each counter is still the actual wanted value plus the residual as before, it is better to consider the sum  $V$  of these  $t$  wanted values and the sum  $X$  of these residuals, and apply Markov's inequality to  $V$  and  $X$  rather than on the individual counters.

### Soluzione

Devo stimare la somma delle frequenze con cui sono apparsi gli elementi dell'intervallo richiesto.

Costruiamo gli intervalli diadici per  $n$ , e teniamo  $\log n$  CM-Sketch (tabelle), una per ogni dimensione degli intervalli diadici.

Quando arriva un elemento nell'intervallo  $(l, r)$ , è necessario aggiornare **tutte** le tabelle, ma adesso, ad eccezione del caso base con intervalli diadici di dimensione uno, le tabelle avranno come elementi i vari intervalli di ogni livello.

(e.g nel secondo livello gli intervalli diadici sono due:  $[1, \frac{n}{2}]$  e  $[\frac{n}{2} + 1, n]$ , quindi un elemento appartenente al primo insieme viene considerato come “elemento 1” di quella tabella. In particolare, la tabella in questione ha  $r$  righe e due colonne ).

Sappiamo che  $(l, r)$  può essere suddiviso in al più due intervalli per ogni livello, quindi  $2 \cdot \log n$  intervalli diadici per  $(l, r)$ .

Una query  $Q(l, r)$  implica che per ogni intervallo diadico è necessario interrogare tutte le tabelle sui relativi intervalli, ottenendo al più due stime per ogni livello. Quindi ottengo  $t \leq 2 \cdot \log n$  contatori  $c_i$ .

I contatori vengono presi dalla stessa riga  $j$  di ogni tabella (può essere migliorato, ma per lo scopo basta così, inoltre la dimostrazione è più semplice).

Voglio mostrare che la probabilità di errore è minore di  $\delta$ , i.e  $Pr(\sum_{k=1}^t c_k > \sum_{i=l}^r F[i] + 2\epsilon \log n ||F||) < \delta$  .

Osserviamo che ogni  $c_i$  relativo ad un intervallo  $[a, b]$  contiene  $\sum_{k=a}^b (F[k]) + X_i$ , con  $X_i$  errore su quel determinato intervallo, essendo gli intervalli diadici **disgiunti**.

Inoltre, siano  $\mathbf{d}$  gli intervalli diadici,  $\sum c_i = \sum_d \sum_{k=a_d}^{b_d} (F[k]) + X_d = \sum_{k=l}^r F[k] + X$ , con  $X$  errore complessivo.

Notiamo che la sommatoria sui valori esatti si può eliminare dalla disequazione, ottenendo  $P_r(X > 2 \cdot \epsilon \log n \|F\|) < \delta$ .

Applicando la diseguaglianza di Markov si ottiene che  $P_r(X > 2 \cdot \epsilon \log n \|F\|) \leq \frac{E[X]}{2 \cdot \epsilon \cdot \log n \|F\|}$ .

Per la linearità del valore atteso si ottiene invece  $P_r(X > 2 \cdot \epsilon \log n \|F\|) \leq \frac{\sum_{i=1}^{2 \cdot \log n} E[X_i]}{2 \cdot \epsilon \cdot \log n \|F\|}$ , e poichè  $E[X_i] \leq \frac{\epsilon}{e} \|F\|$  allora  $P_r(X > 2 \cdot \epsilon \log n \|F\|) \leq \frac{2 \cdot \epsilon \cdot \log n \|F\|}{2 \cdot \epsilon \cdot \log n \|F\|} \cdot \frac{1}{e} = \frac{1}{e}$ . Questa è la probabilità che ci sia un errore sulla riga  $j$ .

Visto che abbiamo scelto da tutte le tabelle la riga che minimizza la somma dei contatori (è come avere un'unica tabella Count-Min-Sketch), per avere certamente errore deve esserci errore su **tutte** le righe, per cui, visto che le righe sono  $\mathbf{r}$ , otteniamo  $\frac{1}{e^r} = \delta$ .

Un miglioramento potrebbe essere quello di prendere la riga migliore per ogni tabella ogni volta.

## 9 [Succinct data structure for range queries]

Borrowing the idea of dyadic intervals employed in the above solution for the count-min sketch for range queries, design a data structure that uses few additional bits to preprocess a bitvector  $B$  of length  $n$ , such that  $B[i]$  is the bit in position  $i$  for  $0 \leq i < n$ . After that, the data structure must support any query of the form  $\otimes(i, j)$  to return the bitwise exclusive or of the bits  $B[i], B[i+1], \dots, B[j]$  for  $0 \leq i \leq j < n$ .

### Soluzione

Considerando che lo xor di  $n$  bit restituisce 0 se il numero di bit uguali a 1 è pari, altrimenti restituisce 1 se il numero di bit uguali a 1 è dispari.

Risolvere il problema equivale dunque a calcolare  $(\otimes(1, j)) - (\otimes(1, i))$ , in quanto dobbiamo stabilire se il numero di bit uguali a 1 in  $(i, j)$  è pari o dispari.

- Basta dunque generare un array di  $n$  posizioni, dove

- $A[i]$ , con  $0 \leq i \leq n$ , corrisponde a  $\otimes(1, j)$ .

- c) Analyze the complexity of the proposed solution.

La soluzione richiede  $O(n)$  memoria aggiuntiva e  $O(1)$  per rispondere alla query.

## 10 [External memory implicit searching]

Given a static input array A of N keys in EM, describe how to organize the keys inside A by suitably permuting them during a preprocessing step, so that any subsequent search of a key requires  $O(\log_B N)$  block transfers using just  $O(1)$  blocks of auxiliary storage (besides those necessary to store A). Clearly, the CPU complexity should remain  $O(\log N)$ . Discuss the I/O complexity of the above preprocessing, assuming that it can use  $O(\frac{N}{B})$  blocks of auxiliary storage (differently from the search).

## ESERCIZIO 2

#528075

MARIA CHIARA GECCHI

Voglio ottenere che il costo dello ricerca di una chiave sia pari a  $O(\log N)$  operaz I/O.

Supponiamo c'è un array A già ordinato e lo riconosciamo come un B-tree.

L'idea è di scongiare A uno volta solo, leggendo via via blocchi di dimensione B e rappresentandoli sulle foglie dell'albero. Quando leggerò il  $(B+1)$ -esimo elemento lo inserisco come nodo padre.

N.B. Se il padre contiene più une chiavi, vado al livello superiore - bedre del padre.

Le  $(B+1)$ -esime elementi nel padre dunque se ricerco mi consente di stabilire se continuare nel nodo figlio e SX oppure sulle chiavi seguenti del nodo stesso.

Il pre-processing è dunque di costo  $O\left(\frac{N}{B}\right)$  I/O

algoritmo: // input : A  
// output : C = array che rappresenta un B-tree

J=0, h=0

while (J < N) {

J = J +  $(B+1)^h * B$ ;

}     h++;

① calcolare  
altezza che  
avrà il B-tree  
 $(h = \lceil \log_{B+1} N \rceil)$

~~for ( i=0 ; i < h ; i++ ) {~~

    indici. odd  $\left( \frac{(B+1)^i - 1}{B} \right)$ ;

② calcolo  
indici  
iniziali  
dell'array C

③ scorro A e leggo B+1 elen. per volta  
 indice = 0;  
 while (indice < N) {  
 livello = h - 1 // passo delle foglie  
 leaf = READ(A, indice, indice + B);  
 WRITE(c, leaf, indice leaf);  
 indice leaf = indici [livello]; // i indici  
 if (indice > N - 1)  
 break;
 }

|| scava  
 || B  
 || elementi  
 || de A  
 i indici  
 l'elenco  
 lista celco  
 lista di  
 foglie ②

Adesso devo inserire il B+1 elemento ed ho 2 casi:  
 - se il padre ha meno di B chiavi  $\Rightarrow$  aggiungo qui semplicemente la nuova chiave  
 - se il padre ha B chiavi, scrivo il padre nell'array C e scavo di nuovo di un livello per poter inserire la chiave nel padre, se anch'esso è pieno ripeto le procedure

Key = READ (A, indice);  
 while (livello > 0) {  
 livello --;  
 nodo = READ(c, indici [livello], indici [livello + B]);  
 if (nodo. len < B) { nodo. odd(key);  
 else  
 WRITE(c, nodo, indici [livello] + 1);  
 }
 }

un po' oscura quest'ultima parte

La ricerca di un elemento  $m$  C può essere fatta odosso in  $O(\log_B N)$ .

Se voglio ricercare la chiave  $x$ , si parte ricercando la radice "r" dell'albero, cioè i primi  $B$  elementi di A) in menu prima.

L'ideale sarebbe avere " $r$ " sempre qui in HC perché è il punto di partenza di ogni ricerca.

A tal punto confrontiamo " $x$ " con le  $B$  chiavi, al più avremo  $B+1$  scelte su cui procedere con la ricerca.

Se  $x < k_1 \Rightarrow$  controllo 1° figlio di  $r$

Se  $k_1 < x < k_2 \Rightarrow$  4 2° "

e così via fino al caso  $x > k_B$

Per capire dove si trova effettivamente il figlio ~~scelto~~ usi le formule

Indice =  $(n \cdot B + n + 1) \rightarrow$  i-esimo figlio di  $r$

↓  
posiz del nodo in cui si trova il figlio

Nel caso peggiore della ricerca, quando NON trovo " $x$ ", vengono eseguite al max  $\log_B N$  lettura dello  $\infty N \Rightarrow$  cioè uno scogl.  $\vee$  l'elenco dell'albero.

## 11 [Number of splits for $(a, b)$ -trees]

Consider the  $(a, b)$ -trees with  $a = 2$  and  $b = 3$ . Describe an example of an  $(a, b)$ -tree with  $N$  keys and choose a value of the search key  $k$  such that performing a sequence of  $m$  operations  $\text{insert}(k)$ ,  $\text{delete}(k)$ ,  $\text{insert}(k)$ ,  $\text{delete}(k)$ ,  $\text{insert}(k)$ ,  $\text{delete}(k)$ , etc. . . , produces  $\Theta(m \cdot H)$  split and fuse operations, where  $H = O(\log_a \frac{N}{a})$  is the height. Try to make the example as general as possible. After that, consider the  $(a, b)$ -trees with  $a = 2$  and  $b = 8$ : produce some examples to check that the above situation cannot occur. Try to guess some properties from the examples using the fact that  $a = \frac{b}{4}$ : if they are convincing, try to prove and use them to show that the situation mentioned above cannot occur, and that the number of split and fuse operations is  $\Theta(m)$ .

## Soluzione

es 11

Number of splits for (a,b) trees

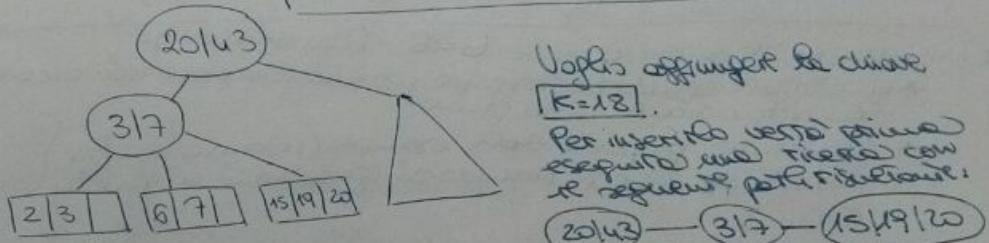
(a,b) tree with  $a=2$  and  $b=3$

- descrivere un esempio di un (a,b) tree con  $N$  chiavi
- scegliere un valore di ricerca  $K$  che esegue una sequenza di un operazioni (tree inserimenti e cancellazioni)
- produce  $\Theta(\log H)$  split e fusioni

dove  $H = \log \frac{N}{a}$  è l'altezza.

(2,3) tree

Suppongo di avere un (a,b) tree con  $a=2$  e  $b=3$  così formato:



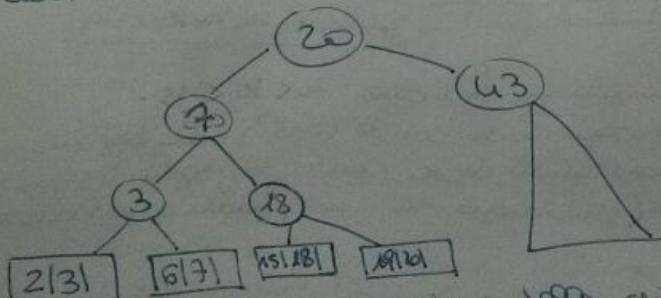
Voglio aggiungere le chiavi

$K=18$ .

Per inserire la posta prima  
eseguiamo una ricerca con  
le seguenti parti rispettive:

20/43 — 317 — 15/19/20

Tutti i nodi incontrati nel percorso hanno  $(b-1)$  rating key, perché sono pieni. La foglia ha già  $b$  chiavi, quindi anche essa è piena. Aggiungono una  $K=18$  provoca uno split, per ogni nodo nel percorso. → causando l'aggiunta di un nuovo livello delle liste.



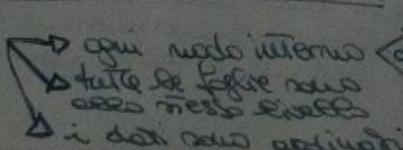
L'aggiunta di  $K$  ha  
prodotto  $\Theta(H)$  split.

Seguendo l'eliminazione delle chiavi  $K=18$  ecco che ottengo  
che si contrarre. Dopo l'eliminazione, la foglia in cui mi trovo  $K$   
contiene  $a+1$  chiavi. Si inizia una fusione che procederà fino  
alla radice. Ritorniamo alla struttura precedente dell'elenco.

Quindi dopo un operazione  $\langle \text{INSERT}(18), \text{DELETE}(18) \rangle$  si ottengono  $\Theta(\log H)$  operazioni di  
divisione e fusione

Note.

(2,3) tree



(uno chiave)

operazioni di  
divisione e fusione

(due chiavi)

ha un elemento e due figli

ha due chiavi e tre figli

i dati sono ordinati

Quindi per un qualsiasi  $(2,3)$  tree avere un cammino di ricerca di nodi pieni, è sufficiente aggiungere un elemento alle foglie di questo cammino per portare una sorta di split fino alla radice.

L'eliminazione di questa雾a ci darà comporterà l'eliminazione delle foglie ormai (attualmente le chiavi appunto).

Vediamo quindi fatti il node padre con il nodo fratello e così da qui ricordare l'albero di fork-join (prima dell'affunghi).

Per conclusione.

Per ogni operazione di inserimento e cancellazione vengono effettuate  $\Theta(H)$  operazioni nell'albero.

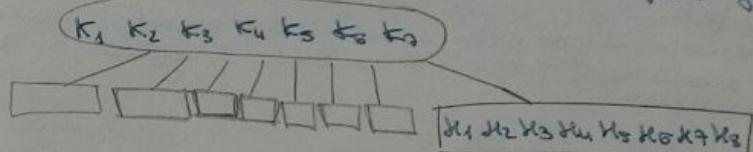
Quindi per un operazione  $\Theta(n \cdot H)$  op. nell'albero.

Considerare il caso  $a=2$  e  $b=8$  (con  $b=4a$ )

→ esibire qualche esempio per dimostrare che non creare la ritrovare visto prima

→ generare proprietà dell'esempio (usando  $a=b/4$ )  
dimostrare che il numero di operazioni in questo caso è  $\Theta(n)$ .

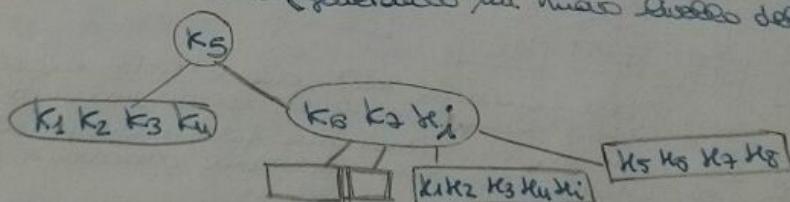
Sappiamo di avere un  $(2,8)$  tree di altezza 2 di questo genere:



Vogliamo aggiungere un elemento nel fatto che  $K_4 < K_5$ .

Il search path è completo (solo nodi con  $b$  figli) e ha foglie

contiene  $b$  chiavi. L'affunghi di chi cerca va split di ogni modo fino alla radice (generalmente non livello dell'elenco).



A questo punto però l'eliminazione di chi non produrrebbe nessun merge. Le foglie infatti dopo la rimozione avrebbero 2a elementi (4); per cui le merge dovrebbero essere eseguite altre 3 eliminazioni. Non verrebbe comunque realizzato un merge fino alla radice poiché il padre contiene solo  $2 > a-1$  chiavi.

Gli operatori e più formidabili.

Se  $b=4\omega$ , una nuova legge si crede da uno split  
in fase di inserimento di una chiavi  
contiene  $\frac{4\omega}{2} = 2\omega$  chiavi approssimativamente.  
  
poiché  $\frac{b+1}{2} \geq 2\omega$

→ quindi un'operazione di ribilanciamento si ottiene  
solo dopo un'operazione di eliminazione o 2e inserimenti.

per avere un ribilanciamento,  $O(n)$  operazioni  
sono necessarie più tardi

Se  $b=4\omega$ , dopo una operazione di fusione di un nodo  $n$ ,  
durante un ribilanciamento dovuto ad una  
eliminazione, si contiene un numero di  
chiavi  $K$

$$\underline{\omega-1+\omega} \leq K \leq \underline{\omega-1+4\omega}$$

↓  
il min numero  
di chiavi del nodo  
che viene fuso con n

↓  
numero mass  
di chiavi del  
nodo che deve  
fusione con n

$$2\omega-1 \leq K \leq 5\omega-1$$

Splitando n se contiene  $3\omega$  elementi, possono fornire che  
se legge coinvolta in una operazione di ribilanciamento  
(durante operazione di eliminazione) contiene un numero di  
chiavi compreso tra  $[3/2\omega \text{ e } 3\omega]$ .

→ un'operazione di ribilanciamento si ottiene dopo  
 $(\frac{1}{2}\omega)$  operazioni di eliminazione o 2e inserimenti  
per avere un ribilanciamento :  $O(n)$  operazioni  
sono necessarie

Gli esempi i casi per avere un ribilanciamento sono necessari  
 $O(n)$  operazioni di inserimento e eliminazione; se non si ottiene  
il ribilanciamento è ovviamente più tardi.

Se sono necessarie  $O(n)$  operazioni per un ribilanciamento  
una sequenza di  $m$  operazioni di inserimento e  $m$  eliminazioni

## 12 [1-D range query]

- a. Describe how to perform a one-dimensional range queries in  $(a, b)$ -trees with  $a, b = \Theta(B)$ .

Given two keys  $k_1 \leq k_2$ , the query asks to report all the keys  $k$  in the  $(a, b)$ -tree such that  $k_1 \leq k \leq k_2$ .

Give an analysis of the cost of the proposed algorithm, which should be output-sensitive, namely,  $O(\log_B(N) + \frac{R}{B})$  block transfers, where  $R$  is the number of reported keys.

- La query può essere effettuata, eseguendo separatamente la query per la chiave  $k_1$  e per la chiave  $k_2$ . (L'array viene mantenuto sequenzialmente in memoria esterna.)
  - Una volta ottenuti i blocchi di  $k_1$  e  $k_2$  eseguiamo uno scan dal blocco che contiene  $k_1$  fino al blocco che contiene  $k_2$ , prendendo tutti gli elementi compresi.
  - La complessità è data:
    - \* dalla ricerca delle due chiavi  $k_1$  e  $k_2$  :  $2 \cdot O(\log_B N)$
    - \* dalla lettura dei blocchi (scan) contenuti tra i nodi visitati durante la ricerca (i quali contengono chiavi di routing e chiavi da restituire come output)  $O(\frac{R}{B})$ .

quindi **I/O Complexity** =  $O(\log_B N + \frac{R}{B})$

- b. After that, for a given set of  $N$  keys, describe how to build an  $(a, b)$ -tree for them using  $O(sort(N))$  block transfers, where  $sort(N) = \Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$  is the optimal bound for sorting  $N$  keys in external memory.

Possiamo costruire l'albero in  $O(sort(N))$  I/Os, seguendo i seguenti passi:

1. ordinando l'array (costo  $\Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ )
2. eseguendo la costruzione dell'albero di ricerca livello per livello dalle foglie alla radice (bottom-up) ( $\sum_{i=1}^H \frac{N}{B^i} = O(\frac{N}{B})$ ).

Quindi il costo dell'ordinamento domina la complessità.

## 13 [External memory permuting]

- a. Given two input arrays  $A$  and  $\pi$ , where  $A$  contains  $N$  elements and  $\pi$  contains a permutation of  $\{1, \dots, N\}$ , describe and analyze an optimal external-memory algorithm for producing an output array  $C$  of  $N$  elements such that  $C[\pi[i]] = A[i]$  for  $1 \leq i \leq N$ .

es. 13 | External memory permuting

Dati due array in input  $A$  e  $\pi$ , dove  $A$  contiene  $N$  elementi e  $\pi$  contiene una permutazione  $\{1, \dots, N\}$  descrivere ed analizzare un algoritmo per produrre un array  $C$  di  $N$  elementi con  $C[\pi[i]] = A[i] \quad \forall i \in \{1, \dots, N\}$

1. Considero un array  $A'$  di coppie  $(A[i], \pi[i])$ , ovvero le coppie che mettono in relazione l'elemento in posizione  $i$ -esima di  $A$  e una posizione del vettore  $\pi$ . Questo passo esegue una lettura di  $N$  elementi e scrittura di  $N$  coppie  $\rightarrow$  costo di I/O:  $O(\frac{N}{B})$
2. Applicando il k-way mergesort, ordiniamo le coppie rispetto alla componente  $\pi[i]$ . Questo passo ha costo  $O(\text{sort}(N))$ , utilizzando un k-way merge sort è  $O(\frac{N}{B} \log_{B/B} \frac{N}{B})$
3. A questo punto scrivo nel vettore  $C$  nelle posizioni indicate da  $\pi[i]$ , il primo elemento delle coppie. Questo passo ha costo  $O(N/B)$  poiché letture e scritture in modo sequenziale.  
\* Il costo complessivo è dominato dal costo dell'ordinamento.

- b. After that, extend the lower bound argument given for the sorting problem in the EM model to the permutation problem:

- given  $N$  input elements  $e_1, e_2, \dots, e_N$  and an input array  $\pi$  containing a permutation of the integers in  $[1, 2, \dots, N]$ , rearrange in EM the elements according to the permutation in  $\pi$ , so that they appear in the order  $e_{\pi[1]}, e_{\pi[2]}, \dots, e_{\pi[N]}$ .

Data una sequenza di elementi  $N$ , suddivisa in blocchi di dimensione  $B$ , vogliamo calcolare il numero di possibili permutazioni degli  $N$  elementi tale che il numero di operazioni I/O sia minimo.

Consideriamo una sequenza di  $t$  operazioni di input, e contiamo quante permutazioni diverse è possibile ottenere. Trovando il minimo  $t$  che definisce tutte le possibili permutazioni, si ottiene il lower-bound desiderato.

Si consideri una generica operazione di input (il numero di modi possibili per scegliere un blocco nella memoria esterna è al più  $N$ , si considera il numero massimo di scelte possibili) e si distinguano i due casi:

1. La pagina viene caricata in memoria per la prima volta (questo caso può avvenire  $n = \frac{N}{B}$  volte, poiché ci sono  $N$  elementi disposti in  $B$  blocchi), posso permutare i suoi elementi in  $B!$  modi e si possono disporre in memoria in al più  $\binom{M}{B}$  modi ( $B$  sottoinsiemi di  $M$ ).
2. La pagina era già stata caricata in memoria, i suoi elementi sono già permutati nell'ordine desiderato (non si considerano tutte le permutazioni ripetute perché sono già state viste) quindi si considera solo come possono essere disposti in memoria gli elementi cioè in  $\binom{M}{B}$  modi.

Unendo i due casi, dopo una sequenza di  $t$  I/O, si possono produrre al più  $N^t \cdot (B!)^{\frac{N}{B}} \cdot \binom{M}{B}^t$  permutazioni distinte. Il numero di permutazioni deve risultare maggiore di  $N!$ , così indica che tutte le permutazioni sono state prese in considerazione.

Passando ai logaritmi si ottiene:  $t \cdot \log(N) + \frac{N}{B} \cdot \log(B!) + t \cdot \log(\binom{M}{B}) \geq \log(N!)$ .

Con l'approssimazione di Stirling  $\log(x!) \approx x \cdot \log(x)$  otteniamo:

$$t \cdot \log(N) + B \cdot \frac{N}{B} \log(B) + t \cdot \log(\binom{M}{B}) \geq N \cdot \log(N)$$

con l'approssimazione  $\log(n)_k \approx k \cdot \log \frac{n}{k}$  otteniamo:

$$t \cdot \log(N) + N \cdot \log(B) + B \cdot t \cdot \log\left(\frac{M}{B}\right) \geq N \cdot \log(N),$$

infine, raccogliendo  $t$  e  $N$ , si ottiene:

$$t \cdot \log(N) + B \cdot t \cdot \log\left(\frac{M}{B}\right) \geq N \cdot \log(N) - N \cdot \log(B) \rightarrow t \cdot (\log(N) + B \log\left(\frac{M}{B}\right)) \geq N \cdot (\log(N) - \log(B))$$

quindi:

$$t \geq \frac{N \cdot \log\left(\frac{N}{B}\right)}{\log(N) + B \log\left(\frac{M}{B}\right)}.$$

A questo punto distinguo due casi:

1. se  $\log N \leq B \cdot \log \frac{M}{B}$  allora  $t \geq \frac{N \cdot \log(\frac{N}{B})}{B \cdot \log(\frac{M}{B})} = (\frac{N}{B}) \cdot \frac{\log(\frac{N}{B})}{\log(\frac{M}{B})} = \Omega \left( \frac{N}{B} \cdot \log_{\frac{M}{B}} \left( \frac{N}{B} \right) \right)$
2. se  $\log N > B \cdot \log \frac{M}{B}$  allora  $t \geq \frac{N \cdot \log(\frac{N}{B})}{\log(N)} = \frac{N \cdot \log(N) - N \cdot \log(B)}{\log(N)} = \left( N - N \cdot \frac{\log(B)}{\log(N)} \right) \geq \left( N - \frac{1}{2} \cdot N \right) = \Omega(N)$ , considerando  $\frac{\log(B)}{\log(N)} \leq \frac{1}{2}$  ossia  $B \leq \sqrt{N}$

Quindi riarrangiare  $N$  elementi rispetto ad una data permutazione richiede  $\Omega(\min\{N, n \cdot \log_m n\})$ .

## Soluzione2

Permutare gli elementi di un array  $A$  può essere fatto utilizzando l'algoritmo merge sort con una particolare relazione di ordinamento  $\prec$ , tale che:

$$A[i] \prec A[j] \iff \pi[i] < \pi[j].$$

In questo caso è necessario tenere in memoria principale una porzione di  $A$  e la porzione di  $\pi$  ad essa associata, ma ai fini della valutazione in memoria esterna non ci sono variazioni.

$$\text{I/O Complexity} = O\left(\frac{N}{B} \cdot \log \frac{N}{M}\right)$$

Con questa soluzione, il lower bound è identico a quello del sorting, perché la permutazione è un ordinamento.

## 14 [Lower bound for searching]

For the best possible comparison-based searching algorithm in a sorted array of  $N$  elements stored in external memory, prove that each search requires  $\Omega(\log_B N)$  I/Os in the worst case.

### Soluzione

Consideriamo il problema della ricerca in memoria interna (dove il miglior *comparison-based search algorithm* è il binary search).

In questo caso per determinare un lower bound si definisce formalmente l'albero delle comparazioni come un albero binario.

Per calcolare con un albero di confronto, si confrontano i due elementi nella radice, ramo a sinistra o a destra a seconda di quale dei due elementi è più piccolo, e si continua ricorsivamente fino a raggiungere una foglia. Il valore memorizzato nella foglia rappresenta il numero di elementi più piccoli rispetto alla query  $x$ .

Dato che ci sono  $N+1$  possibili output (l'elemento  $x$  si può trovare in una delle posizioni dell'array o in nessuna(failure)), l'albero di comparazione ha almeno  $N + 1$  foglie, e quindi ha profondità  $\log_2(N + 1)$ .

Si può seguire lo stesso ragionamento anche nel caso della memoria esterna, sotto l'assunzione che vengano confrontate solo coppie di elementi in memoria interna, inclusa la query  $x$ .

Con questa assunzione è possibile modellare ogni algoritmo come un albero di decisione, ma con grado maggiore. In ogni nodo dell'albero, si carica un blocco di elementi dal disco ( $B$  elementi) e poi si collegano i nodi in accordo con il rank di  $x$  nel blocco.

Come nel caso della memoria interna, l'albero delle decisioni ha  $N + 1$  possibili (foglie) risultati e ogni nodo ha almeno  $B + 1$  figli, quindi la profondità dell'albero è di almeno  $\log_{(B+1)}(N + 1)$  e quindi un qualsiasi algoritmo di ricerca deve compiere almeno  $\log_{(B+1)}(N + 1)$  passi per raggiungere la soluzione, cioè  $\Omega(\log_B N)$ .

## 15 [Cache-oblivious selection]

Consider the linear-time deterministic selection discussed in class.

- Recall:
  1. Concettualmente partiziona l'array in  $\frac{N}{5}$  quintuple di 5 elementi (adiacenti) ciascuno e ricorsivamente calcola la mediana dei gruppi calcolati al passo precedente (con  $O(1)$  confronti).
  2. Calcola ricorsivamente la mediana delle mediane calcolate al passo precedente. Sia  $M$  la mediana ottenuta da queste chiamate ricorsive.
  3. Usando  $M$  come pivot, partiziona gli elementi dell'array in due gruppi: gli elementi maggiori o uguali della mediana  $M$  e quelli minori.
  4. Conta il numero di elementi in ogni gruppo e effettua una chiamata ricorsiva nel gruppo che contiene gli elementi del rank desiderato.
- a. Prove that this algorithm is cache oblivious with complexity  $O(\frac{N}{B})$  block transfers (i.e. cache misses) for  $N$  elements stored in an array and any unknown block size  $B$ .

Per rendere l'algoritmo studiato cache-oblivious è sufficiente specificare come ogni singolo passo lavora in termini di memory layout e scansioni.

Il linear-time deterministic selection fa  $O(N)$  confronti per trovare il  $k$ -esimo elemento più piccolo in un array di dimensione  $n$ .

Proof:

$T(N, k)$  denota il caso pessimo per trovare il  $k$ -esimo elemento più piccolo e sia  $T(N) = \max_k T(N, k)$ .

Step 1 impiega  $O(N)$ , perché determinare la mediana di 5 elementi impiega tempo costante.

Step 2 impiega al più  $T(\frac{N}{5})$ .

Step 3 impiega  $O(N)$ .

Assumendo che almeno  $\frac{3}{10}$  degli elementi dell'array sono più piccoli del pivot  $M$  e almeno  $\frac{3}{10}$  sono più grandi o uguali del pivot, allora - > **questo va provato**

lo Step 4 impiega al più  $T(\frac{7N}{10})$ , e otteniamo quindi la ricorrenza:

$T(N) \leq c \cdot N + T(\frac{N}{5}) + T(\frac{7N}{10})$ , per qualche costante  $c$ .

- In questo particolare caso, in cui l'algoritmo è cache-oblivious, le scansioni costano  $c \cdot \frac{N}{B}$ , a differenza del caso lineare deterministico in cui costano  $c \cdot N$ . Quindi si ottiene la seguente ricorrenza nel numero di trasferimenti in memoria  $T(N)$ :

$$- T(N) \leq T\left(\frac{N}{5}\right) + T\left(\frac{7}{10}N\right) + c \cdot \frac{N}{B} = c' \frac{N}{5B} + c' \frac{7}{10} \frac{N}{B} + c \frac{N}{B} = \frac{9c'N + 10cN}{10B} = \frac{9c' + 10c}{10} \frac{N}{B}$$

Quindi:

$$\frac{9c' + 10c}{10} \leq c' \iff 9c' + 10c \leq 10c'$$

- Per qualsiasi  $c$  posso trovare un  $c'$  che soddisfa la relazione. L'algoritmo impiega  $O(\frac{N}{B})$

## 16 [Implicit navigation in vEB layout]

Consider  $N = 2^h - 1$  keys where  $h$  is a power of 2, and the implicit cache-oblivious vEB layout of their corresponding complete binary tree, where the keys are suitably permuted and stored in an array of length  $N$  without using pointers (as it happens in the classical implicit binary heap but the rule here is different). The root is in the first position of the array.

- a. Find a rule that, given the position of the current node, it is possible to locate in the array the positions of its left and right children.

Per cercare i figli di n

1. Calcolo la posizione dell'elemento n nel sottoalbero “minimo con altezza 2”  $n \% 3$ 
  - (a) se il risultato è 0 allora STOP ( $posfigliosx = n + 1$   $posfigliodx = n + 2$ )
  - (b) se il risultato è 1 o 2 goto 2
2. Calcolo k, il numero di sottoalberi di altezza  $\frac{h}{2}$  prima del sottoalbero di altezza  $h/2$  in cui si trova n, partendo da 1:  $k = \left\lfloor \frac{n}{2^{\frac{h}{2}-1}} \right\rfloor$  con  $2^{\frac{h}{2}} - 1$ .
3. Controllo k
  - (a) se  $k = 0$  sono in **TOP**,  $nf = 0$ ;  $step = 2^{\frac{h}{2}} - 1$ <sup>1</sup>,
  - (b) altrimenti ( $k > 0$ ), calcolo la radice del nuovo sottoalbero su cui ricorro:  $root = root + k \cdot (2^{\frac{h}{2}} - 1)$ 
    - i. normalizzo la posizione di n rispetto alla radice:  $n = n \% (2^{\frac{h}{2}} - 1)$
    - ii. aggiorno il numero totale di nodi a sinistra di n :  $nf = nf + (k - 1) \cdot (2^{\frac{h}{2}-1})$
4. Se  $h > 2$  calcolo la nuova altezza  $h = \frac{h}{2}$  goto 2
  - (a) altrimenti (mi sono ricondotto al caso minimo)
    - i. aggiorno il numero totale di nodi a sinistra aggiungendo l'eventuale fratello sinistro di n:  $nf = nf + (n \% 3) - 1$
    - ii. A questo punto posso calcolare le posizioni dei figli cercate  $posfigliosx = nf \cdot 2 \cdot step + root + step$  infatti ogni foglia (conteggiata in nf) aggiunge due sottoalberi che hanno  $step$  nodi ( $nf \cdot 2 \cdot step$ ) e l'aggiunta di root mi riporta alle posizioni del sottoalbero di n  $posfigliodx = posfigliosx + step$
  - b. Discuss how to apply this layout to obtain
    - (b') a static binary search tree
    - (b'') a heap data structure, discussing the cache complexity.

---

<sup>1</sup> $nf$  è il numero di nodi a sinistra di n (che generano sottoalberi di dimensione  $2^{\frac{h}{2}} - 1$  che precedono i figli di n) conteggiati per calcolare l'offset e  $step$  è la dimensione dei sottoalberi che precedono i figli di n nel vEB.

## 17 [MapReduce indegree distribution]

Use the basic cache-oblivious operations of scan and sort as building blocks for computing the degree distribution of a graph (e.g. a web graph), assuming that this graph is already given and represented by adjacency lists: the output lists all possible indegree values  $d$  and, for each such  $d$ , the number of nodes having indegree  $d$ . Equivalently, see this computation in terms of MapReduce/Hadoop: just sketch the code for map and reduce, leaving the rest.

### Fase 1

Ogni arco della lista di adiacenza  $(i,j)$  contribuisce ad incrementare l'indegree di  $j$ .

Assunzione (valida): la lista di adiacenza è memorizzata contiguamente.

Mappo il procedimento che farei su map-reduce sulle operazioni cache-oblivious:

1. Scan su tutta la lista di adiacenza per costruire un array ( $O(\frac{N^2}{2})$  elementi) in cui inserisco  $< j, 1 >$  se sto leggendo l'arco  $(i, j)$ .
2. Sort sull'array appena costruito sul nome del nodo (i.e  $j$ ).
3. Scan sull'array. Adesso elementi con stesso nodo destinazione saranno contigui nell'array, dunque posso tenere un contatore e ogni volta che finisco di leggere elementi dell'array riguardanti un certo nodo, scrivo in un file  $< j, \# \text{occorrenze} >$ .  $\# \text{occorrenze}$  corrisponde all'indegree del nodo  $j$ .

**Fase 2: poichè devo calcolare la indegree distribution**, eseguo un procedimento analogo per calcolare quanti nodi hanno in-degree “ $k$ ” per esempio.

1. Sort sul nuovo file sulla base dell'indegree degli elementi  $< \text{dest}, \# \text{occorrenze} >$ .
2. Scan dell'array. Finché trovo nodi con stesso  $\# \text{occorrenze}$ , tengo un contatore. Quando l'indegree cambia (i.e sto iniziando a guardare nodi con indegree maggiore), scrivo in un nuovo file  $< \# \text{occorrenze}, \# \text{nodi} >$ .

---

**Algorithm 3** Descrizione dell'algoritmo per la prima fase

---

```
map( String node){  
    emit( node , 1)  
}  
  
reduce( String node , Iterable<Integers> values ){  
    int indegree = 0;  
    for ( Integer val : values )  
        indegree++;  
    writeToFile( indegree , node );  
    /** Assumo che nella Fase 2  
        Sort sia fatto su primo elemento */  
}
```

---

---

**Algorithm 4** Descrizione dell'algoritmo per la seconda fase

---

```
map( Integer indegree , String node ){  
    emit( indegree , node )  
}  
  
reduce( Integer indegree , Iterable<Integers> values ){  
    int numNodes = 0;  
    for ( Integer val : values )  
        numNodes++;  
    writeToFile( indegree , numNodes ); // Scrivo in un nuovo file  
}
```

---

## 18 [MapReduce prefix sums]

Given a huge array A of N entries, define the *i*th prefix sum of A as  $S_i = \sum_{j=1}^i A[j]$ .

Show how to compute simultaneously all  $S_i$ , for  $i = 1, 2, \dots, N$ , using overall  $O(\log N)$  calls to scan/sort/MapReduce/Hadoop.

## Soluzione

18 |

$2^0$

$2^1$

$2^2$

(64)

— — —

map (index, value) {  
 emit (index, value);  
 if (index < N-2<sup>h</sup>)  
 emit (index + 2<sup>h</sup>, value)  
 }  
 reduce (index, values) {  
 sum = values.sumAll(); →  
 emit (index, sum)  
 }

Ogni elemento  $A[i]$   
 è scambiato con  
 $A[i+2^h]$

Abbiamo bisogno  
 di eseguire  $\log_2 n$   
 passi, poiché ad  
 ogni passo le somme  
 da eseguire diminuiscono  
 di un fattore 2.

Note.  $h$  è globale

Complexità:  
 $O(\text{sort}) \cdot \log_2 N = \frac{N}{B} \log_2 \frac{N}{B} \log_2 N$

A  
 Mappiamo gli elementi visto mappati con le sue  
 stesse posizione e con le posizioni successive  
 calcolate come  $i+2^h$ .  
 Questa costruzione permette solo map di utilizzare  
 gli elementi da dare in posso delle reduce, che  
 a questo punto controllano le stesse somme parziali.

## 19 [Suffix sorting in EM]

Using the DC3 algorithm seen in class, and based on a variation of mergesort, design an EM algorithm to build the suffix array for a text of N symbols. The I/O complexity should be the same as that of standard sorting, namely,  $O(N/B \log M/B N/B)$  block transfers.

Volendo estendere l'algoritmo DC3 per la memoria esterna, è necessario modificare opportunamente l'algoritmo.

In primo luogo è necessario sostituire il processo di ordinamento basato sul radix-sort, con il k-way merge sort ( $k = M/B$ ).

Per prima cosa si creano i trigrammi di tipo 1 e 2, come nel DC3 originale, creando un array di coppie, associando ad ogni trigramma la posizione nel testo in input T.

Le coppie avranno quindi la forma  $\langle 1/2\text{-gramma}, i \rangle$  dove  $i = [0, |T|]$ .

Il costo dell'operazione è nell'ordine di  $\text{scan}(N) = O(N/B)$  operazioni di I/O.

Si procede quindi con il sort lessicografico in memoria esterna delle coppie, in base alla colonna relativa al trigramma, assegnando a ciascun trigramma il suo rank. Il costo è pari a  $\text{sort}(N) = O(N/B \log_{M/B} N/B)$ .

Nel caso vi siano duplicati nei rank, è necessario eseguire una ricorsione sulla stringa  $T'$  così ottenuta:

- Si ordinano le tuple in base alla loro posizione i
- Si genera quindi una stringa contenente i ranking degli 1/2-grammi, concatenando tutti i rank degli 1-grammi con tutti i ranking dei 2-grammi.

Sia  $SA'$  il risultato della ricorsione su  $T'$ . Per convertire  $SA'$  in  $SA^{12}$  utilizziamo la seguente formula:

Definiamo  $n_1$  come il numero di 1-grammi di T

- se  $SA'[i] < n_1 \rightarrow SA^{12}[i] = 1 + 3 * SA'[i]$
- altrimenti  $SA^{12}[i] = 2 + 3(SA'[i] - n_1)$

A questo punto  $SA^{12}$  è il suffix array degli 1/2-grammi. Consideriamo ogni elemento di  $SA^{12}$  come una coppia così composta  $\langle SA^{12}[i], i+1 \rangle$ , dove  $i \in [0, 2n/3]$ ;

$i+1$  rappresenta il rank del trigramma in posizione  $SA^{12}[i]$ .

Per ottenere il suffix array inverso  $ISA^{12}$ , ordiniamo le coppie per la prima componente (costo:  $\text{sort}(N)$ )

Adesso procediamo con la costruzione di tuple; più precisamente creiamo:

- Una quintupla per gli **0-grammi** così formata:  
 $\langle T[i], T[i+1], r', r'', i \rangle$

Dove:

$T[i]$  e  $T[i+1]$  sono i primi 2 caratteri dello 0-gramma;  
 $r'$  ed  $r''$  sono rank tali che  $\langle i+1, r' \rangle$  e  $\langle i+2, r'' \rangle \in ISA^{12}$ ;

i è la posizione dello 0-gramma;

- Una quadrupla per gli **1-grammi** così formata:  
 $\langle T[i], r, r', i \rangle$

Dove:

$T[i]$  è il primo carattere dell'1-gramma;

r ed  $r'$  sono rank tali che  $\langle i, r \rangle$  e  $\langle i+1, r' \rangle$  ∈ ISA<sup>12</sup>;

i è la posizione dell'1-gramma;

- Una quintupla per i **2-grammi** così formata:  
 $\langle T[i], T[i+1], r, r'', i \rangle$

Dove:

$T[i]$  e  $T[i+1]$  sono i primi 2 caratteri del 2-gramma;

r ed  $r''$  sono rank tali che  $\langle i, r \rangle$  e  $\langle i+2, r'' \rangle$  ∈ ISA<sup>12</sup>;

i è la posizione del 2-gramma;

Il costo della costruzione delle tuple è pari a O(N/B) operazioni di I/O, in quanto per crearle è stato scorso il testo T (costo: O(N/B)), e il ISA<sup>12</sup> (costo: O(N/B)).

Adesso ordiniamo gli 0-grammi sulla base della prima componente della tupla ( $T[i]$ ) e, a parità di valore, le ordiniamo secondo la terza componente ( $r'$ ).

Eseguiamo un merge delle tuple relative agli 1/2-grammi, ordinandole secondo il rank r.

Possiamo quindi ora eseguire un merge delle due liste, ordinandole in base ai casi seguenti:

i                  j

0-gramma VS 1-gramma -> confrontiamo  $T[i]$  con  $T[j]$ , a parità di valore confrontiamo  $r'i$  con  $r'j$

0-gramma VS 2-gramma -> confrontiamo  $T[i]$  con  $T[j]$ , a parità di valore confrontiamo  $T[i+1]$

con  $T[j+1]$ , a parità di valore confrontiamo  $r''i$  con  $r''j$

La lista risultante sarà il suffix array SA della stringa originale T.

Si può notare come il costo totale rimane nell'ordine del sort(N), in quanto la complessità è dominata dal costo del sorting delle varie tuple usate nell'algoritmo.



## 20 [Euler tour]

20

### EULER TOUR

dato grafo connesso  $G$ ,  $n$  vertici su archi,

dove ogni nodo ha grado pari.

costruzione un algoritmo per costruire un

Euler tour per  $G$  in tempo  $O(n+mn)$

Partiziona offrendo che il nostro  $G$  ha grado

pari per ogni nodo, allora ha un ciclo Euleriano.

Cio' vole per le proprietà:

un grafo NON orientato ha un ciclo Euleriano

$\uparrow$   
 (ogni vertice ha grado pari)  $\Rightarrow$  Abbiamo assunto grafo  
 $\wedge$  pari  $\forall$  nodo

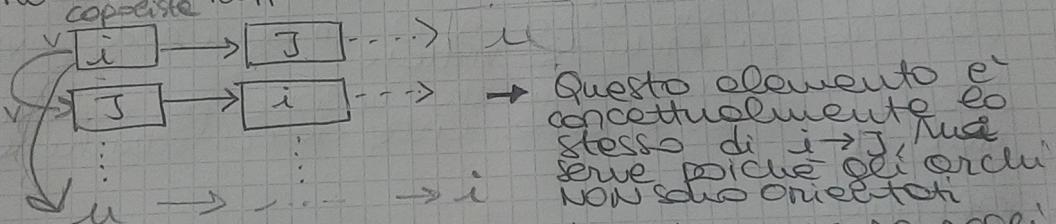
tutti i vertici con grado  $> 0$   
 appartengono ad una singola  
 componente连通

$\Rightarrow$  Abbiamo assunto  $G$  connesso.

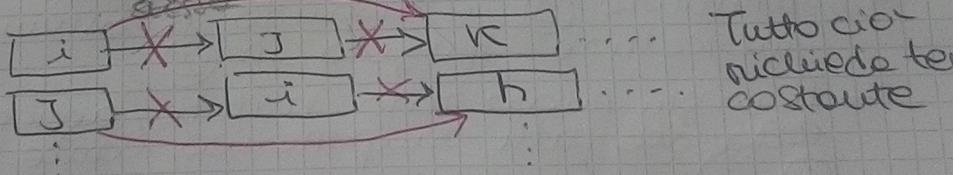
INTRO: Daffronteremo il grafo usando liste di  
 adiacenza, avendo costo  $O(n+mn)$  spazio

### STRUTTURE DATI E LORO USO!

1) liste di adiacenza: rappresentano i nodi e le  
 relazioni tra essi dettate dagli archi.  
 dati i nodi "i" e "j", dato l'arco tra essi  $(i, j)$   
 avrò una rappresentazione come segue



2) Quando visito un arco lo dico "muore nel prossimo elemento della lista, vediamo come:  
 - muore  $i \rightarrow j$   
 - muore  $j \rightarrow i$   
 - in particolare con "muore" intendiamo spostare il puntatore del capo lista al prossimo elemento della lista (anche al prossimo arco)



3) Bit vector: ha n elementi t.c.  $A[i] = 1$   
 se ho visitato almeno una volta  
 nodo "i".

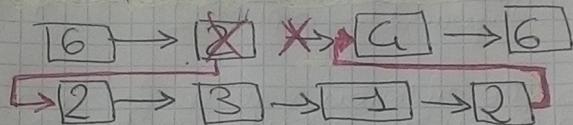
Mi permette di sapere, in tempo costante, se un  
 è già stato attraversato.

4) Liste : ci servirà per mantenere il path temporaneo, in particolare consentendo di salvare ogni ciclo che incontrerò.

FS : Supponendo di aver trovato fin'ora il ciclo che ha toccato i nodi  $(6, 2, 4, 6)$ ; prosegua la visita dei miei nodi e per esempio partendo da 2, mi risulta un nuovo ciclo  $(2, 3, 1, 2)$ .

Basterà unire il nuovo nuovo ciclo alla mia lista, come segue:

- aggiungi alla lista il nuovo ciclo, inserendolo al posto dell'elemento iniziale del ciclo stesso
- Quindi c'è l'idea di estendere il path corrente, innestando dentro la lista il nuovo ciclo trovato.



5) Liste : semplicemente mantiene i nodi visitati, senza tener traccia delle ripetizioni. Per effettuare l'inserimento in tale lista, evitando duplicati, controllo prima il Bit Vector.

6) Infine è importante mantenere, per ogni nodo, un nodo predecessor. Quando questo sarà a  $\emptyset$  in ogni nodo avrà esaurito gli archi da visitare.

Possiamo salvare questo valore, per esempio, negli elenchi in testa alle liste delle strutture dati ①.

### ALGORITMO:

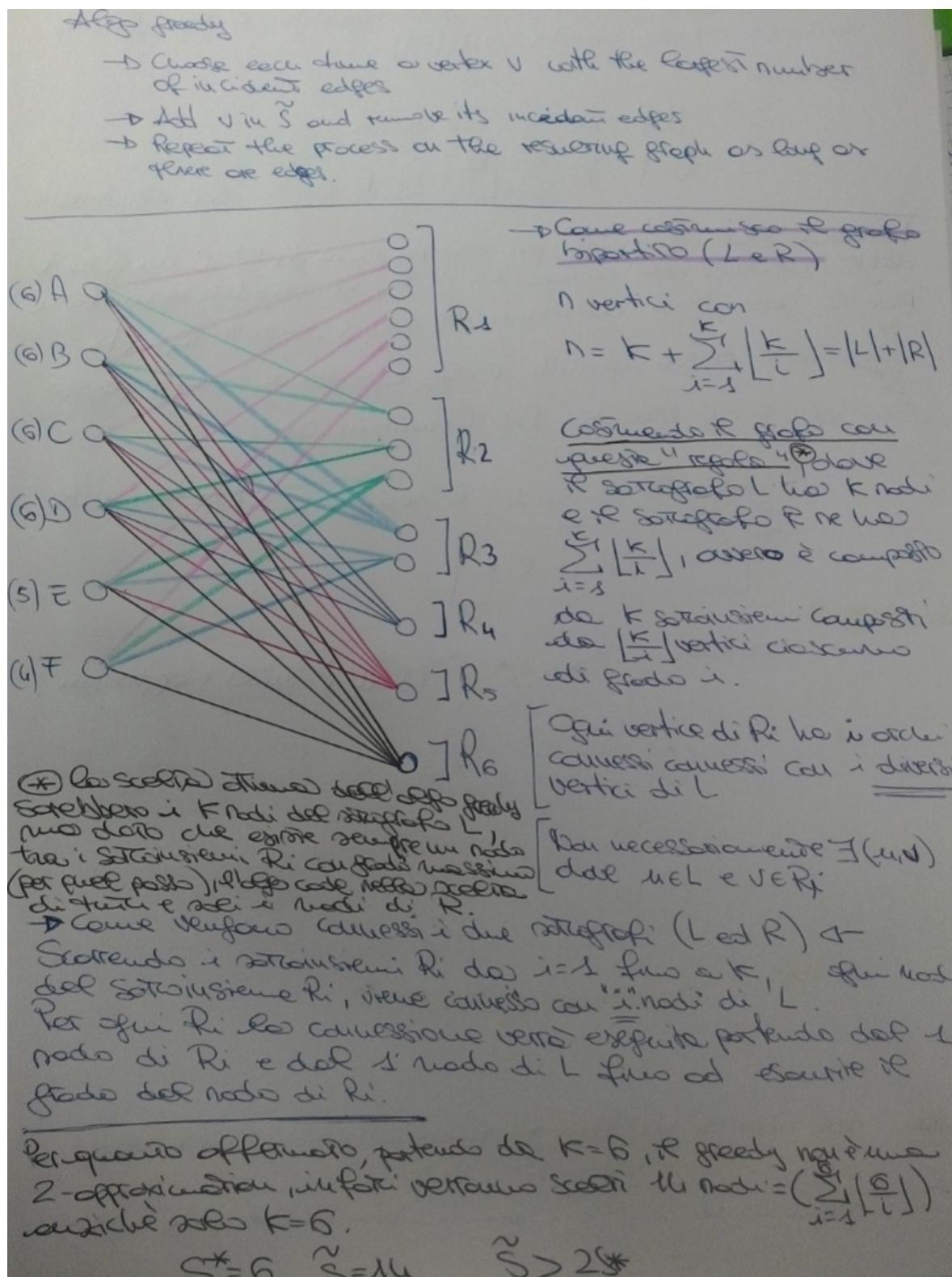
- Scegli nodo 1 del grafo per iniziare
- Finché lo nodi con grado  $\neq \emptyset$ 
  - Scegli un nuovo nodo dal grafo
  - Da esso visita gli archi, mantenuti nella struttura ①, e usando il sistema di marcatura spiegato in ②
  - Per ogni nuovo nodo visitato aggiornare:
    - lista path temporaneo come spiegato in ④.
    - Setto l'elemento del Bit Vector  $A[i] = 1$  (
    - usando il Bit Vector aggiornare la lista dei nodi visitati ⑤).
  - La lista ④ mi aiuterà a capire se ho appena trovato un nuovo ciclo. (vedi es. 2312)

- In tal caso, mi comporto come segue:
  - Scorro la lista dei nodi visitati finché trovo un elemento con grado  $i = \emptyset$
  - Tale elemento sarà il punto da cui farò ripartire il mio algoritmo con la visita degli archi del punto
- FINE algoritmo, tutti i nodi hanno grado residuo  $= \emptyset$ . Restituisco la lista del path che adesso sarà il tour definitivo.

Per la complessità, la lista dei nodi visitati viene scansionata una sola volta poiché il puntatore punta sempre al primo elemento con grado residuo diverso da zero. In più lavoriamo su tutta la lista di adiacenza che è composta da  $2m$  elementi. Le operazioni hanno costo costante per come sono state costruite le strutture dati. Quindi  $O(n + m)$ .

## 21 [Wrong greedy for minimum vertex cover]

Soluzione



Generalizzando:

Per come viene definito il grafico bipartito, l'algo prende in considerazione tutti i vertici di  $R(R)$

Quindi

$$S^* = |L| = k \quad \text{Sol. ottima}$$

$$\tilde{S} = |R| = k \cdot \sum_{i=1}^k \left[ \frac{1}{x_i} \right] \approx k \cdot \log k \quad \text{Sol. approssimata}$$

Notiamo che  $\tilde{S}$  cresce più velocemente di  $S^*$  di un fattore  $\log k$ . Quindi all'aumentare di  $k$ , le considerazioni fanno per  $r$  così  $k=5$  continuano a valere, così il rapporto

$\frac{\tilde{S}}{S^*} = \log k$  cresce, quindi il algo approssima sempre meglio la soluzione ottima. Più formalmente

dato  $\frac{k}{r} = r > 1$ , è sempre possibile trovare  $k$  per cui:

$$\boxed{S > r \cdot |L| = r \cdot k}$$

quindi  $\tilde{S} \geq r \cdot S^*$  (non è una  $r$ -approssimazione)

Note. nel caso di problemi di minimizzazione, deve valere

$$\tilde{C}/C^* \leq r$$

nel caso dei problemi di massimizzazione deve valere

$$C^*/\tilde{C} \leq r$$

## 22 [Polynomial reduction from NAE-3-SAT to MAX-CUT]

### Soluzione

**Claim.** NAE 3-SAT reduces in polynomial time to MAX CUT.

*Proof.* Given a NAE 3-SAT instance  $\varphi$ , we will construct an equivalent MAX CUT instance  $(G, c)$ . For every variable  $x_i$  of  $\varphi$ , we will add two vertices to  $G$  labeled by  $x_i$  and  $\neg x_i$  and we will connect the two vertices by an edge. We assign capacity  $M = 10 \cdot m$  to each of these “variable” edges. (Here,  $m$  is the number of clauses in  $\varphi$  and  $n$  is the number of variables.) For every clause  $C$  in  $\varphi$ , we will add a “clause” triangle between the vertices corresponding to the terms in  $C$ . We assign capacity 1 to each of these “clause” edges.<sup>4</sup>

We claim that  $G$  contains a cut with capacity at least  $n \cdot M + 2 \cdot m$  if and only if  $\varphi$  is satisfiable.

Suppose  $\varphi$  is satisfiable and consider any satisfying assignment. This assignment corresponds to a cut in  $G$ . (One side of the cut consists of all vertices labeled by terms that evaluate to 1 in the assignment. The other side of the cut consists of all vertices labeled by terms that evaluate to 0 in the assignment.) Since exactly one of terms  $x_i$  and  $\neg x_i$  evaluate to 1 in an assignment, all variable edges go across the cut, which contributes  $n \cdot M$  to the capacity of the cut. Since the assignment satisfies  $\varphi$ , exactly two edges in every clause triangle go across the cut, which contributes  $2 \cdot m$  to the capacity of the cut. In total the capacity of the cut is equal to  $n \cdot M + 2 \cdot m$ .

On the other, suppose that  $G$  contains a cut with capacity at least  $n \cdot M + 2 \cdot m$ . First, we claim that all variable edges go across this cut. The reason is that any cut that misses at least one of the variable edges has capacity at most  $(n - 1) \cdot M + 3 \cdot m = n \cdot M + 3 \cdot m - 10m$ , which is strictly smaller than  $n \cdot M + 2m$ . Next, we claim that exactly two edges of every clause triangle go across the cut. The reason is that no cut can separate three edges of a triangle and therefore if a cut separates fewer than two edges in one of clause triangles, then its capacity is strictly smaller than  $n \cdot M + 2m$ . Since all variable edges go across, this cut corresponds to an assignment for  $\varphi$ . Furthermore, since the cut separates exactly two edges per clause triangle, the corresponding assignment satisfies all clauses of  $\varphi$ .

## 23 [Approximation for MAX-SAT]

### Soluzione

**23] Approximation for MAX-SAT**

Problema MAX-SAT della soddisfattibilità di una formula booleana

FORMULA VARIABILI      VALORI BOOLEANI      CLAUSES

$F_i(x_1, \dots, x_n) = b_1 \dots b_n, C_1 \dots C_n$        $b_i \in \{0, 1\}$  con  $1 \leq i \leq n$

Calcolare il numero  $m_1$  di clausole soddisfatte dell'ottienimento  
tale che:  $k_i := b_i$

Calcolare il numero  $m_2$  di clausole soddisfatte dell'ottienimento  
tale che:  $k_i := \bar{b}_i$  con  $1 \leq i \leq n$

Se  $m_2 > m_1$  return  $k_i := b_i$   
altrimenti return  $k_i := \bar{b}_i$

Dimostrare che questo algoritmo è una  $\frac{1}{2}$ -approssimazione di MAXSAT

In seguito all'ottienimento dei valori  $k_i$  booleani per tutte le variabili  $x_i$ , separiamo le variabili nel seguente modo:

- inseriamo in A le variabili  $x_i$  se  $b_i = 1$   
altrimenti inseriamo  $\bar{x}_i$
- inseriamo in B le variabili  $x_i$  se  $b_i = 0$   
altrimenti inseriamo  $\bar{x}_i$

$A$  [A] contiene solo variabili con valori pari ad 1.  
 $B$  contiene solo variabili con valori pari a 0.] quindi l'algoritmo calcola  $m_1$

[Viceversa] quando l'algoritmo calcola  $m_2$   
(visto che vengono considerate le variabili complementate)

Verifichiamo con P il numero di clausole formate solo da variabili  
di A  $\rightarrow$  verificate finiti punti in calcolo  $m_1$ .

Verifichiamo con N il numero di clausole formate solo da variabili  
di B  $\rightarrow$  verificate finiti punti in calcolo  $m_2$ .

Verifichiamo con C il numero di clausole formate da variabili d  
A  $\cup$  B (verificate finiti punti sia quando si controlla  $m_1$  che  $m_2$ )

L'obiettivo si minimizza rispetto a  $\tilde{S}$

$$\tilde{S} = \max\{m_1, m_2\} = \max\{P+C, N+C\} \Rightarrow$$
$$\boxed{\tilde{S} = C + \max(P, N)}$$

$$S^* = \text{val ottimale} \quad S^* \geq \tilde{S} = C + \max(P, N)$$

La sol approssimata è il max uguale a quella di  $S^*$  opt<sup>(1)</sup>.  
Inoltre  $S^* \leq C + P + N$  (numero totale di clausole che compareano  $\neq$ )

Quindi è vero anche  $\frac{S^*}{2} \leq \frac{C+P+N}{2}$

$$\Delta \frac{S^*}{2} \leq C + \frac{P+N}{2} \quad \text{poiché } \frac{C}{2} \leq C$$

Dato che il  $\max(a, b) \geq \frac{a+b}{2}$  otteniamo:

$$\frac{S^*}{2} \leq C + \frac{P+N}{2} \leq C + \max(P, N)$$

Quindi: (con  $\tilde{S} \leq S^*$ )

$$\frac{S^*}{2} \leq C + \frac{P+N}{2} \leq \underbrace{C + \max(P, N)}_{\tilde{S}} \leq S^* \Rightarrow \frac{S^*}{2} \leq \tilde{S} \leq S^*$$

LA SOL APPROSSIMATA È COMPRESA TRA LA META' DEL VALORE OTTIMALE ED IL VALORE OTTIMALE MEDIO

L'obiettivo è una 2-approssimabile

note.

(1) Infatti nel corso delle sol. ottenute tramite l'obiettivo di approssimazione si può verificare determinate sostanzialmente le clausole della soluzione ottima

Il valore di  $r$  non dipende dall'assegnamento dei  $b_1, b_2 \dots b_n$   
valori binari, in quanto se un assegnamento è possibile  
per  $m_4$ , sarà buon per  $m_5$  (e viceversa).

Il corso preferito è quindi metà delle cause che non verificano  
e metà delle loro negazioni si verifica. (\*)

Esempio

$$F = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_4 \vee x_5)$$

$$(1 \vee 0 \vee 1) = 1 \quad \wedge \quad (0 \vee 1) = 1 \quad \Rightarrow 1$$

$$S^* = 2 \quad (x_1 = 1, x_2 = \emptyset, x_3 = 1, x_4 = \emptyset, x_5 = 1)$$

Supponiamo che l'algo crei queste assegnazioni:

$$\begin{array}{l} x_1 = 1 \\ x_2 = \emptyset \\ x_3 = \emptyset \\ x_4 = 1 \\ x_5 = \emptyset \end{array} \quad \left[ \begin{array}{l} A = \{ \bar{x}_1, \bar{x}_2, \bar{x}_3, \underline{x_4}, \underline{x_5} \} \quad P = 1 \quad (2^2) \\ B = \{ \bar{\bar{x}}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4, \bar{x}_5 \} \quad N = 1 \quad (1^2) \\ C = \emptyset \end{array} \right]$$

$$C + \min(P, N) = 1 = \left( \tilde{S} = \frac{1}{2} S^* \right)$$

(\*) Suggerito meglio: Se un assegnamento è possibile per  $m_4$   
ma che che nessuna clausola è  
soddisfatta con questi assegnamenti

## 24 [Greedy 2-approximation for MAX-CUT on weighted graphs]

### 2.1.2 Max-Cut using Greedy Algorithm

Fix an ordering on the vertices  $v_1, v_2, \dots, v_n$ . Start with two empty bins  $S, \bar{S}$ . Pick the first vertex  $v_1$ , put it in  $S$ . For each subsequent vertex, put it in the bin such that  $|E(S, \bar{S})|$  is maximized.

**Theorem 2.1.4** *The greedy algorithm is a 2-approximation algorithm for MAX-CUT.*

**Proof:** Consider any edge in the graph. One vertex must come first in the ordering. Call the later vertex *responsible* for the edge.

Let  $r_i$  be the number of edges that  $v_i$  is responsible for. Since every edge has exactly one responsible vertex,  $\sum_{i=1}^n r_i = m$ .

Claim: When  $v_i$  is added, at least  $\frac{r_i}{2}$  edges are added to the cut. Because: each of the other vertices adjacent to the  $r_i$  edges that  $v_i$  is responsible for have already been assigned to either  $S$  or  $\bar{S}$ . The set which contains the most endpoint vertices of these  $r_i$  edges must contain at least  $\frac{r_i}{2}$  endpoint vertices (i.e. it cannot be that both sets contain less than the average number of vertices per set). By the greedy algorithm,  $v_i$  will be added to the other set, thus adding at least  $\frac{r_i}{2}$  edges to the cut.

$$\begin{aligned}\text{Number of edges in final cut} &= \sum_{i=1}^n \text{number of edges added by each responsible vertex} \\ &\geq \sum_{i=1}^n \frac{r_i}{2} \\ &\geq \frac{m}{2}\end{aligned}$$

Since  $\text{opt}(I) \leq m$ , the algorithm is a 2-approximation. ■

## 24) Greedy 2-approximation for MAX-CUT on weighted graph

Steps of the algorithm:

- Per ogni vertice  $v \in V$  fissare un ordinamento  
 $v_1, v_2, \dots, v_n$

Funziona con due insiami nodi  $S, \bar{S}$

- Prendi il primo vertice  $v_1$  e mettelo in  $S$ .
- Per tutti i vertici successivi, inserisceli in un insieme

dato per cui  $|E(S, \bar{S})|$  è maximizzato.

PSEUDO CODE

for  $i = 1 \dots n$

    if  $|E(S + \{v_i\}, \bar{S})| > |E(S, \bar{S} + \{v_i\})|$   
     then      $S = S + \{v_i\}$

    else      $\bar{S} = \bar{S} + \{v_i\}$

return  $S$  (as  $\bar{S} \rightarrow$  approx of  $S$ )

Della 2-approximation. Dimostrazione:

Definizioni utili per la dimostrazione:

- Reference node: nell'ordinamento dei nodi ce n'è uno che viene prima. Per ogni arco fuori c'è un nodo che viene dopo, questo nodo è il reference-node di quell'arco.



con  $i < j \Rightarrow j$  è il reference node di quell'arco.

- Si  $\rightarrow$  il numero di archi per cui  $i$  è un reference node.
- Quindi  $\sum_i \ell_{ij}$  è estremamente un reference node, quindi:

$$\sum_{i=1}^n \ell_{ij} = m$$

Per dimostrare che l'algoritmo greedy è una 2-approximation  
notiamo che:

- ogni volta che un nodo vi viene aggiunto, almeno  $\frac{r_i}{2}$  archi vengono aggiunti al figlio.

Infatti se così non fosse quel nodo verrebbe aggiunto nell'altro vicino. Quindi gli archi che viene aggiunto, vengono sicuramente aggiunti più di  $\frac{r_i}{2}$  archi al figlio.

Quindi:

$$\text{Numero di archi nel set finale} \geq \sum_{i=1}^n \frac{r_i}{2} \geq \frac{m}{2} \leq \tilde{S}$$

Sappiamo che  $\sum_{i=1}^n r_i = m$

$$\boxed{\begin{array}{l} S^* \leq m \\ \tilde{S} \geq \frac{m}{2} \end{array}} \rightarrow \boxed{\frac{\tilde{S}}{S^*} = \frac{m}{2m} = \frac{1}{2}} \quad \boxed{\frac{S^*}{2} \leq \tilde{S} \leq S^*}$$

Questo dimostrazione vale anche nel caso di grafi pesati, infatti noi utilizzando una rappresentazione del grafo pesato come multigrado (qui con dei pesi, ma poi sarà tra i nodi) noi considerando semplicemente la modifica dell'algoritmo, otteniamo lo stesso risultato.

Nel secondo caso, dobbiamo notare che se però delle costituzioni degli archi di  $S$  e  $\tilde{S}$  ( $\rightarrow |E(S) \cup \tilde{S}|$ ) possono considerare le somme dei pesi degli archi. Vorrei comunque l'ottimizzazione per cui facendo il più aggiunto, almeno metà degli archi di cui è referente vengono aggiunti al figlio, quindi almeno metà del peso totale di questi archi viene aggiunto.

## 25 [Randomized 2-approximation for MAX-CUT]

### 25 | Randomized 2-approximation for MAX-CUT

Alego:

1. Per ogni vertice  $v \in V$ , lancia una moneta equiprobabile
  - se esce testa, inserisci  $v$  in  $C$
  - altrimenti, inserisci  $v$  in  $V \setminus C$
2. Moltiplica  $T = \{v\}$   
 Per ogni arco  $(v, w) \in E$  tale che  $v \in C, w \in V \setminus C$   
 aggiungi  $(v, w)$  in  $T$   
 ritorna  $|T| = \tilde{S}$ .

Per dimostrare che l'algoritmo è una 2-approximazione definiamo  
 una variabile indicatoria  $X(v, w)$

$$X(v, w) = \begin{cases} 1 & \text{se } (v, w) \in T \\ 0 & \text{altrimenti} \end{cases}$$

La variabile  $\boxed{X(v, w)}$  vale 1 quando  $v \in C$  e  $w \in V \setminus C$ ,  
 oppure quando  $v \in V \setminus C$  e  $w \in C$   $\star$

Per come è descritto l'algoritmo, la probabilità che un vertice venga inserito in un insieme anziché in un altro è  $1/2$ .  
 Quindi:

$$\Pr[X(v, w) = 1] = \underbrace{\Pr[v \in C \wedge w \in V \setminus C] + \Pr[v \in V \setminus C \wedge w \in C]}_{\text{i due eventi sono incompatibili}} =$$

$$= \Pr[v \in C] \Pr[w \in V \setminus C] + \Pr[v \in V \setminus C] \Pr[w \in C] = \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \boxed{\frac{1}{2}}$$

pertanto  $E[X(v, w)] = \frac{1}{2}$  in quanto la media di una variabile indicatoria indicatrice è pari alla probabilità che essa sia 1.

Per come è definita  $X$ , la somma di tutte le variabili indicatorie, ovvero gli archi in  $T$ . Definiamo

$$X = \sum_{(v, w) \in E} X(v, w)$$

Abbiamo che

$$\mathbb{E}[X] = \mathbb{E}\left(\sum_{(u,\omega) \in E} X(u,\omega)\right) = \sum_{(u,\omega) \in E} \mathbb{E}(X(u,\omega)) = |E| \cdot \frac{1}{2}$$

Così  $|E|$  = numero di archi del grafo.

Sic  $S^*$  lo sottografo ottimale.  $\rightarrow |S^*| \leq |E|$  (1)

Note (1), otteniamo

$$\mathbb{E}[|\bar{T}|] = \mathbb{E}\left[\sum_{(u,\omega) \in \bar{T}} X(u,\omega)\right] = \frac{1}{2} \cdot |E| \geq \frac{1}{2} |S^*|$$

Il numero di archi in  $\bar{T}$  è  
proprio il numero di tutte  
le vertici selezionati (per come  
sono definite).

Quindi

Il numero medio di archi tagliati dalla sottoset e opposta  
 $|\tilde{S}| = |\bar{T}|$  è almeno pari alla metà di quelli dello sottografo  
ottimale.

$$\boxed{\mathbb{E}[|\bar{T}|] \geq \frac{1}{2} |S^*|}$$