Appunti di Algoritmica 2

Tutti e nessuno
Corso di Laurea Magistrale in Informatica - Università di pisa

06/12/2013

Contents

Chapter 1

Esecizio 1: Sottografi K4 e C4

Descrivere un algoritmo per elencare e contare dato un grafo non orientato, tutti i suoi sottografi di tipo C_4 e K_4. Valutare e motivare la complessitá della soluzione proposta

1.1 Svolgimento

1.1.1 Elenchiamo i C4

I C_4 sono tutti i cicli di vertici che non sono collegati da altri archi tra di loro:

---

1 $C_4(G)$
2 $\quad \forall v_1, v_2 \in V$
3 $\quad\quad N_\cap = N(v_1) - v_2 \cap N(v_2) - v_1$
4 $\quad \forall v_3, v_4 \in N_\cap, v_3 \neq v_4$
5 `return` v_1, v_3, v_4, v_2

---

ovvero $\forall$ coppia di vertici v_1 ev_2, consideriamo l'intersezione fra gli insiemi dei vicini di ognuno (eliminando l'altro vertice da ogni insieme dei vicini) e $\forall v_3$ e v_4 appartenenti a tale insieme, ritorno la quadrupla di nodi.
Analzziamo la complessitá:

- $\forall v_1, v_2 \in V$ ha un costo pari a $O(n^2)$
- $N_\cap = N(v_1) - v_2 \cap N(v_2) - v_1$ ha un costo pari a $O(\triangle)$

Il costo totale dell'algoritmo é $O(n^2\triangle)$ dove $\triangle = max_{u \in V} d_u$.

1.1.2 Elenchiamo i C4 - Michele

```
1 ∀(v1, v2) ∈ E st : v1 < v2
2     ∀v3 ∈ (N(v2) − {v1}) st : v2 < v3, (v1, v3) ∉ E
3         ∀v4 ∈ (N(v1) − {v2}) st : v3 < v4, (v2, v4) ∉ E
4             if(v3, v4) ∈ E
5                 Out: (v1, v2, v3, v4)
```

Utilizzando un ordinamento sui vertici si evita la ripetizione.
Scorrere tutti gli archi costa $O(m)$, scorrere una lista di adiacenza costa invece $O(\Delta)$, i vari controlli su ordinamento e appartenenza ad E si possono fare in tempo costante. In totale quindi il costo è $O(m\Delta^2)$.

1.1.3 Elenchiamo i K4

Il K_4 é un grafo completo di 4 vertici. L'idea é quella di sfruttare gli HeavyHitters e il calcolo dei triangoli K_3.

DEFINIZIONE : Un nodo $u \in V$ é detto HeavyHitter se $d_u \geq \sqrt{m}$. In un grafo il numero di HH é $\leq 2\sqrt{m}$.

Prima di implementare l'algoritmo é necessario specificare che si possono creare due situazioni diverse:
1) Nel grafo $\exists$ almeno un nodo non hitter $u \in V - H$
2) Nel grafo tutti i nodi sono hitter $u \in H$

Implementiamo l'algoritmo nel primo caso:

```
1 ∀T_i triangolo a, b, c
2 d_u = min(d_a, d_b, d_c)//dove d_u sono i figli di u
3 ∀l ∈ d_u
4 if(l, a) ∈ E ∧ (l, b) ∈ E ∧ (l, c) ∈ E
5 k++
```

La complessitá é $O(m\sqrt{m}\sqrt{m}) = O(m^2)$ dove $m\sqrt{m}$ é il costo per il calcolo di ciascun triangolo e $\sqrt{m}$ sono i nodi derivanti da d_u .

In pratica il procedimento si basa sulla seguente osservazione: ogni K_4 include un K_3 , pertanto possiamo utilizzare questi ultimi per il nostro obiettivo, prendendo di volta in volta il nodo di grado minore (diminuendo il costo di computazione). Controlliamo quindi per ognuno dei suoi figli se il nodo in oggetto é vicino a tutti i vertici del triangolo, in tal caso troviamo sicuramente un K_4 .

Implementiamo l'algoritmo nel secondo caso:

```
1 ∀v1, v2, v3, v4 ∈ H where v1 ≺ v2 ≺ v3 ≺ v4
2         do
3                 if ( (v1, v2) ∈ E, (v1, v3) ∈ E, (v1, v4) ∈ E, (v2, v3) ∈ E, (v2, v4) ∈
  E, (v3, v4) ∈ E)
4                         K4++
```

- il numero di HH é al massimo $|H| \leqslant 2\sqrt{m}$
- il numero dei possibili sottoinsiemi di 4 nodi é $\binom{|H|}{4} \leq \frac{(2\sqrt{m}^4}{4!} = O((\sqrt{m})^4) = O(m^2)$
- sapendo che il controllo di un arco costa $O(1)$ utilizzando un heap

La complessitá totale é $O(m^2)$.

Chapter 2

Esercizio 2 : Clique Massimali

Una clique massimale é un sottoinsieme S dei vertici del grafo tale che S induce un grafo completo mentre per ogni altro vertice u $\notin$ S vale che S $\cup\{u\}$non induce un grafo completo (massimalitá). Descrivere un algoritmo che elenchi tutte le clique massimali di un grafo non orientato con un costo che dipende dalla dimensione dell'uscita (output-sensitive): se C é l'insieme delle clique massimali del grafo con n vertici e m archi, il costo temporale deve essere limitato superiormente da |C| $\times$ (m + n). Suggerimento: dati gli insiemi R e P, le chiamate ricorsive generano tutte le clique massimali che contengono sicuramente R e possibilmente alcuni vertici di P. Nell'albero della ricorsione, il nodo α che corrisponde a tale chiamata con R e P avrá tali clique nelle foglie discendenti. Sia k il numero di tali foglie, ossia il numero di clique massimali generate a partire da α durante il ciclo for sui vertici in P . Possiamo pagare $O(k(m+n))$ tempo per decidere se una chiamata ricorsiva in tale ciclo for porterá sicuramente a una nuova clique massimale: basta vedere se R unito a ció che rimane di P non é coperto dalle clique generate fino a quel momento dal ciclo for.

2.1 Svolgimento

Chapter 3

Esercizio 3: Cammini tra due vertici

Prendendo il metodo della partizione binaria visto a lezione per elencare tutti i cammini tra due vertici distinti s e t di un grafo non orientato, mostrare come modificarlo in modo da garantire che ogni chiamata ricorsiva conduca ad almeno un nuovo cammino elencato (vogliamo quindi evitare chiamate ricorsive che non generino alcun cammino da elencare). Valutare e motivare la complessitá della soluzione proposta.

3.1 Svolgimento

L' algoritmo di ricerca di un cammino tra due vertici distinti s e t in un grafo non orientato si comporta in questo modo:

GeneraCammini(G,s,t) inizia con una BFS chiamata sul nodo finale t, memorizza tutti i nodi in un insieme *C(t)= {tutti i nodi raggiungibili da t secondo una visita in ampiezza}* e considera solo i figli di s tale per cui esiste un cammino da t a qualche $u \in N(s)$ (un certificato di esistenza di un cammino), dunque $\forall u \in N(s) \cap C(t)$;
Su questi elementi si richiama ricorsivamente con parametri d'ingresso (G- { s }, u, t).

Analizziamo la complessitá :
L' algoritmo di base tende ad avere un costo di $O(n^n)$, mentre la versione modificata costa

$$O(|PATH|n(m+n)) = O(|PATH|nm)$$

questo perchè invocare la BFS ad ogni chiamata ricorsiva mi costa O(m+n), e il numero di chiamate ricorsive (dato che evito chiamate su nodi che non generano cammini) é al piú n, tutto moltiplicato per il numero di cammini

|PATH| generati.
Il fatto di avere |PATH| come componente del costo fa si che l'algoritmo modificato sia proprio *output sensitive*.

```
1 GeneraCammini(G, s, t)
2       Y = Y  ∪ {s }
3       C(t) = BFS (t, G -  {s})
4        ∀u ∈ N(s) ∩ C(t)
5       GeneraCammini (G  - {s}  , u, t)
```

Chapter 4

Esercizio 4

[Cammini più brevi tra due vertici] Descrivere un algoritmo per elencare tutti i cammini più brevi (shortest path) tra due vertici distinti s e t di un grafo non orientato (volendo, pesato o meno). Valutare e motivare la complessità della soluzione proposta.

4.1 Svolgimento

Rappresentiamo il grafo G con le liste di adiacenza, che sappiamo avere costo di rappresentazione pari a O(n+m). Per il nostro algoritmo ci interessa sapere che il costo per cancellare un arco dal grafo è costante, O(1), in quanto utilizziamo le liste di adiacenza con cross over edge. Sfruttiamo la BFS da u a v per capire la distanza minima fra i due nodi (tale informazione ci arriva dall'altezza, la profondità dell' sottoalbero)

- chiamo la BFS con radice u, ottenendo un albero di ricerca che mostra tutti i nodi del grafo e come essi siano collegati mediante qualche cammino con u. A partire da queste informazioni creo un array contenente il livello di ogni nodo dell'albero ordinati secondo la prima occorrenza; tale informazione mi quantifica la distanza fra u e un qualsiasi nodo x raggiungibile da u e, ovviamente, se esiste un cammino tra u e v sarà sicuramente incluso nell'albero per una particolare x e l'altezza dello stesso, denotata con h, alla prima occorrenza di v è la distanza del cammino minimo.
  Att. E' probabile che qualche nodo x compaia più di una volta nell' array, e ciò significa che esiste una loop region. Possiamo sfruttare questa informazione tenendo traccia dei nodi visitati in quel momento e impedire di visitarli nuovamente.

Costo O(n+m) (il costo della BFS)

- A questo punto faccio una BFS su v ottenendo un altro array che contiene tutte le distanze minime tra v e x $\forall x \in V$ e, come prima, in tale insieme ci sarà anche u.

  Costo O(n+m) (il costo della BFS)

  Oss.: A questo punto sappiamo che il cammino minimo tra u e v ha lunghezza h, ma siccome vogliamo trovare tutti gli shortest path di tale lunghezza, dobbiamo verificare per quali nodi x la distanza da u sommata alla distanza da v è proprio h e includere quei nodi in un sottografo G'. Da questo, sappiamo che tutti i nodi vengono attraversati da un qualche cammino minimo tra u e v, quindi basterà listarli tutti.

- Applichiamo l'algoritmo sui due array ottenuti precedentemente, con costo **O(nΔ)**

```
1  ∀x ∈ V − {u, v}
2        ∀y ∈ N(x)
3              if ( d(x,u) + d(x,v) == h ) AND
4                     if ( d(y,u) + d(y,v) == h ) AND
5                           if ( d(u,x) + d(x,y) + d(y,v) = d(u,v)
6                                 add x,y in G'
7                           else
8                                 add x,y in G'
9                                 E' = E - (x,y)
10                    else
11                          add x in G'
```

- ora all'interno di G' inseriamo anche u e v, applico nuovamente la BFS su u (NECESSARIO??? mmm no) che produrrà un albero di ricerca che mostrerà tutti i possibili modi di arrivare a v. Basterà quindi listare tutte le possibili diramazioni dell'albero per ottenere la soluzione con l'algoritmo

GeneraCammini Classico.

Costo O(n+m)

Il costo totale della procedura è **3 * O(n+m) + O(nΔ) = O(nΔ)** che nel caso pessimo converge a **O(n^2)**

Chapter 5

Esercizio5 : Minimal feedback vertex set

Nell'algoritmo per elencare i minimal feedback vertex set (MFVS) visto a lezione, dimostrare che il meta-grafo risultante é fortemente connesso, ossia, per ogni due meta-vertici esiste sempre un cammino diretto che li collega, dove ogni arco é rappresentato dalla funzione successore μ: questa proprietá serve a garantire che da qualunque meta-vertice partiamo, riusciamo a raggiungere tutti gli altri meta-vertici. Dimostrare se la funzione successore μ sia iniettiva o meno; in base a ció, mostrare come elencare tutti i MFVS attraverso una visita di tutti i meta-vertici senza peró costruire e memorizzare esplicitamente il meta-grafo. Valutare e motivare la complessità della soluzione proposta. Suggerimento: non riuscendo a dimostrare se μ sia iniettiva o meno, descrivere come generare tutti i MFVS ipotizzando che (i) μ sia iniettiva e (ii) μ non sia iniettiva (qui usare un dizionario e una coda di prioritá per memorizzare i MFVS/meta-vertici).

5.1 Svolgimento

5.1.1 Non-Iniettività di μ

La funzione $successore$ è definita come $\mu : (MFVS \times V) \rightarrow FVS$, quindi vale:

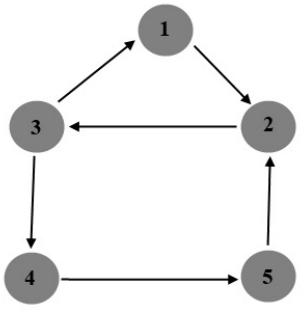
$$\mu \ \ iniettiva \Leftrightarrow [\forall i, j, v \in S_i, w \in S_j \ . \ S_i \neq S_j \wedge v \neq w \Rightarrow \mu(S_i, v) \neq \mu(S_j, w)]$$

Se, ad esempio, volessimo provare a dimostrare la **non iniettività** di μ, potremmo negare la precedente definizione come segue:

$$\mu \ \ non \ \ iniettiva \Leftrightarrow [\exists i, j, v \in S_i, w \in S_j \ . \ S_i \neq S_j \ \wedge \ v \neq w \ \wedge \ \mu(S_i, v) = \mu(S_j, w)]$$

Poiché ci basta trovare un esempio che soddisfi la precedente affermazione, scegliamo, nel grafo mostrato nella figura sottostante, i nodi $v = 1$, $w = 5$ e i MFVS $S_i = \{1, 4\}$ e $S_j = \{1, 5\}$. Avremo, applicando l'algoritmo di minimizzazione ai FVS ottenuti tramite μ, la seguente situazione:

$\mu(\{1,4\},1) = \{4\} \cup \{2\} = \{2\}$
$\mu(\{1,5\},5) = \{1\} \cup \{2\} = \{2\}$



Siamo quindi riusciti a dimostrare che esiste una scelta di u, w, S_i, S_j, diversi tra loro, tale che $\mu(S_i, v) = \mu(S_j, w)$, cioé che la funzione successore μ puó mappare elementi differenti del dominio in elementi uguali del codominio, e che quindi **non é iniettiva**, cvd.

5.1.2 Metagrafo M fortemente connesso sol Nicoletti

Vogliamo verificare se dati due MFVS F e $F*$ esiste sempre un cammino che li collega, ossia:

$$F = F_0, ..., F_s = F*$$

I passi dell'algoritmo sono i seguenti:

- Computiamo un ordine topologico su $G - F*$, escludiamo $F*$ in modo da ottenere un grao aciclico per cui valga l'ordinamento topologico;
- $F_0 = F$, $k = 0$;
- finché $F_k \neq F*$
- scegli v_k come il piú piccolo vertice secondo l'ordine topologico contenuto in $F_k \cap (V - F*)$, ossia il v_k scelto deve appartenere a F_k ma non a $F*$, e prendiamo il piú piccolo in modo tale da poter avere dei nodi uscenti che prima o poi ci condurranno in $F*$.

  ATT: $F_k \cap (V - F*) = \oslash$ sse $F_k = F*$

- $F_{k+1} \subseteq F_k - \{v_k\} + N^+(v_k)$, sappiamo che in un simile insieme esiste sempre e comunque un MFVS e siccome lo otteniamo da F_k eliminando un suo vertice vale che:

  $$F_{k+1} = \mu_G(F_k, v_k)$$

  ergo esiste un arco che collega $F - k$ a F_{k+1}

- In output avremmo $(F_0, ..., F_k)$

Quindi eliminando di volta in volta dei MFVS in cui transitano i vertici v_k piú piccoli e non appartenenti a $F*$, alla fine non possono che rimanerci i vertici apaprtenenti a $F*$.

5.1.3 Visita implicita del Metagrafo - Michele

Sia Q una coda che contiene i MFVS di cui vanno visitati i successori, sia D un dizionario che usiamo per tenere traccia dei MFVS che abbiamo già visitato.

- Calcolo un qualsiasi MFVS F_0, ad esempio con un approccio greedy partendo da tutti i nodi.
- Aggiungo F_0 sia alla coda che al dizionario.
- Eseguo il ciclo

```
1  while(! Q.empty){
2        F = Q.pop();
3        foreach v in F{
4              F_v = μ(F, v)
5              if(!D.contains(F_v)){
6                    D.add(F_v)
7                    Q.add(F_v)
8                    out(F_v)
9                    }
10       }
11 }
```

Complessità alg. greedy Dato un insieme non minimale F di k nodi l'algoritmo greedy utilizzato sia al passo 1 che dalla funzione successore μ itera il seguente procedimento:

- toglie un nodo dall'insieme

- verifica se si ha ancora un FVS mediante una visita BFS sul grafo residuo $G \setminus F$, se la visita incontra un nodo due volte significa che c'è un ciclo e F non è un FVS. Il costo di tale visita è $O(n - |F|)$, dopo tale numero di passi se c'è un ciclo si incontra sicuramente un nodo già visitato.

La complessità di tutto il procedimento è quindi limitabile da:

$$O(\sum_{i=0}^{k-1}(n - (k - i)) = O(nk)$$

Il primo passo ha quindi complessità $O(n^2)$.

Complessità ciclo Nella coda Q ogni MVFS viene inserito ed estratto una volta, il costo dell'estrazione è $O(1)$ quindi basta considerare la complessità del corpo del $foreach$ per $|MFVS| = \sum_{f \in MFVS} |f|$ volte.
Calcolare μ, viste le considerazioni precedenti, costa $O(n(massimaDimensioneMFVS + \Delta)) \subseteq O(n^2)$.
Tutte le operazioni su coda e dizionario hanno costo costante $O(1)$.
Tutto l'algoritmo costa quindi:

$$O(n^2 + |MVFS| * n^2) = O(|MFVS| * n^2)$$

Chapter 6

Esercizio 6 : Cammino/ciclo euleriano

Descrivere un algoritmo per trovare, in tempo lineare, un cammino o un ciclo euleriano in un grafo non orientato e connesso (suggerimento: utilizzare il fatto che nei vertici di grado pari è possibile sempre uscire una volta che si è entrati per trovare un cammino parziale, che viene esteso attraverso cicli che devono essere collegati al cammino perchè il grafo è connesso).

6.1 Svolgimento Maurizio

Assumiamo che valga il seguente teorema sul grafo di input G:
Un grafo ha un cammino euleriano sse tutti i suoi nodi sono di grado pari eccetto due nodi di grado dispari
Rappresentiamo il grafo con le liste di adiacenza, il cui costo di rappresentazione è O(n+m).

6.1.1 Algoritmo

```
1 Sia x ∈ V t.c d(x)%2 ≠ 0 nodo di partenza
2 TempPath = GeneraCammini (E,x)
3 ∀x ∈ TempPath t.c N(x)≠ ∅
4 Sia x nodo di partenza
5 Loop=GeneraCammini(E',x)
6 EulPath=extendPath(TempPath,Loop)
7 return EulPath
8
9 GeneraCammini(E,x)
10 TempPath= TempPath ∪ x
11 if N(x)≠ ∅
12       Sia y primo nodo ∈ N(x)
13       E' = E - (x,y)
14       GeneraCammini ( E', y )
15 return TempPath
```

6.1.2 Complessitá

- Alla riga 1 del codice il costo è 0(n+m), poiché per trovare un nodo di grado dispari al caso pessimo devo scorrere tutte le mie liste di adiacenza
- La cancellazione alla riga 13 dell'arco (x,y) dalle liste di adiacenza hanno il costo pari a quello dell'accesso, ovvero O(1) (ovviamente dalla lista devo cancellare sia x,y che y,x, ma questo è garantito poiché gli elementi nella lista hanno dei puntatori reciproci)
- GeneraCammini (E,x) mi costa O (m) e nel caso ottimo basta una singola chiamata della funzione nella riga 2.
  Nel caso in cui si abbia bisogno delle chiamate successive, qualunque sia il numero dato dal ciclo **for** della riga 3, il costo sará sempre e comunque O(m), dal momento ché GeneraCammini lavora solo sugli archi residui.

La complessitá totale dell'algoritmo é **O(n+m + m) = O(n+m)**

Chapter 7

Esercizio 7: Ricoprimento minimo di vertici

Per il problema del minimum vertex cover (MVC), trovare un controesempio che mostri come la strategia greedy di scegliere sempre il vertice di grado massimo e iterando sul grafo residuo ottenuto cancellando tale vertice e i suoi vicini, non garantisca una 2- approssimazione. Generalizzare tale argomento per mostrare che tale strategia greedy non garantisce una r-approssimazione per una costante $r > 1$.

7.1 Svolgimento

L'algoritmo goloso si comporta in questo modo:

```
1 GreedyVerCov(G) {
2       Seleziono il nodo u di grado massimo
3       Elimino tutti gli archi che hanno u nella coppia
4       Aggiungo u alla soluzione
5       GreedyVerCov(G-{u})
6       return soluzione }
```

Il controesempio alla strategia greedy per il MVC é applicare l'algoritmo a un grafo bipartito.
Il grafo G ha n vertici suddivisi in due sottoinsiemi disgiunti, L e R, tale che:

- |L|+|R|=k + $\sum_{i=1}^{k} \frac{k}{i}$ = n, ció implica che L ha k vertici e R ha k sottoinsiemi disgiunti, dove ogni R_i contiene $\lfloor \frac{k}{i} \rfloor$ vertici $\forall i = 1, .., k$
- ogni vertice di R_i ha un arco che va a i vertici di L
- non necessariamente $\exists\ (u, v)$ dove $u \in L$, $v \in R_i$

- due vertici di R_i non hanno lo stesso vicino in L

La soluzione greedy considera prima tutti i vertici di R ossia $\sum_{i=1}^{k} \frac{k}{i}$, mentre la soluzione ottima considera prima tutto il gruppo L ossia k vertici. Dunque

$$\sum_{i=1}^{k} \frac{k}{i} = k \sum_{i=1}^{k} \frac{1}{i} \cong kH_k \cong klogk$$

dove H_k e la k-esimo numero armonico, o anche detto serie armonica troncata su k, e la serie armonica diverge da qualunque k.

Nel caso $k = 6$ abbiamo che la soluzione ottima include $|L| = 6$ vertici mentre la soluzione greedy include $|R| = 14$ vertici, quindi $|R| > 2 * |L|$, per cui l'algoritmo greedy non e una 2-approssimazione nel caso del grafo bipartito come definito sopra.
Vediamo come viene raffigurato il grafo in questione:

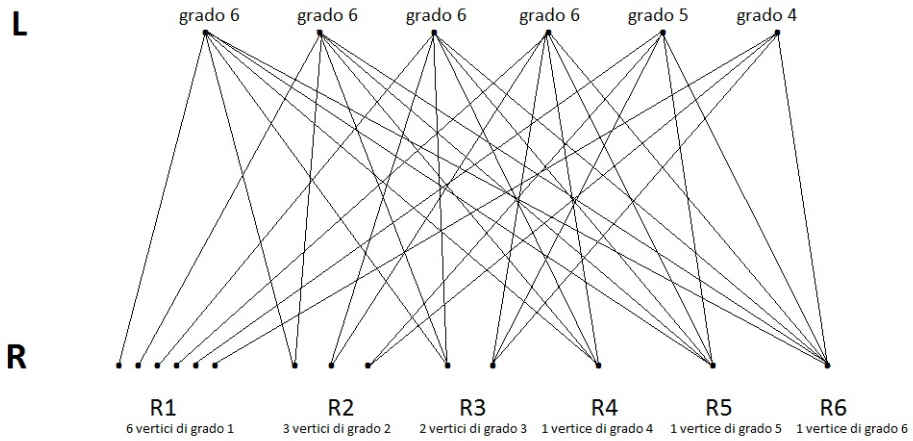


Fig. 7.1: Grafo bipartito con k = 6

Nel caso generale, fissati k e r, con $r > 1$, la soluzione Greedy sarà sempre maggiore di r* | L | , ovvero $H_k > r * |L|$, per cui l'algoritmo greedy non e r-approssimabile per i MVC.

Chapter 8

Esercizio 8: Approssimazione per MAX-SAT

Per il problema MAX-SAT della soddisfacibilità di una formula booleana, si consideri il seguente algoritmo di approssimazione per massimizzare il numero di clausole soddisfatte in una data formula: Sia F la formula data, $x_1, x_2, \ldots, x_n$ **le variabili booleane in essa contenute, e** $c_1, c_2, \ldots, c_m$ **le sue clausole. Scegli i valori booleani casuali** $b_1, b_2, \ldots, b_n$**, ossia ciascun** $b_i \in \{0, 1\}$ $(1 \leq i \leq n)$**. Calcola il numero** m_0 **di clausole soddisfatte dall'assegnamento tale che** $x_i := b_i (1 \leq i \leq n)$**. Calcola il numero** m_1 **di clausole soddisfatte dall'assegnamento tale che** $x_i := \bar{b_i} (1 \leq i \leq n)$**, dove** $\bar{b_i}$ **indica la negazione di** b_i**. Se** $m_0 > m_1$**, restituisci l'assegnamento** $x_i := b_i (1 \leq i \leq n)$**; altrimenti, restituisci l'assegnamento** $x_i := \bar{b_i} (1 \leq i \leq n)$**.**

Dimostrare che il suddetto algoritmo è una r-approssimazione per MAX-SAT, indicando anche il valore di r > 1 (e motivando l'utilizzo di tale valore). Discutere se, in generale, la scelta di $b_1, b_2, \ldots, b_n$ possa influenzare o meno il valore di r, motivando le argomentazioni addotte. Facoltativo: creare un'istanza di MAX-SAT in cui il suddetto algoritmo ottiene un costo che è r volte più piccolo del costo ottimo per una data scelta dei valori di $b_1, b_2, \ldots, b_n$.

8.1 Svolgimento

Una formula F è in **forma normale congiuntiva** (CNF - Conjunctive Normal Form) quando è della forma:

$$\bigwedge_{j=1}^{C} \left(\bigvee_{k=1+j}^{L_i} l_{j,k-j} \right)$$

dove:

- un $letterale\ l$ può essere una variabile negata ($l = \bar{x}$) o affermata ($l = x$);
- una $clausola\ c$ è un insieme di k letterali l_i $(1 \leq i \leq k)$ tali che:

$$\bigvee_{i=1}^{k} l_i$$

- C : è il numero di clausole;
- L_i : è il numero di letterali della clausola i-esima.
- $l_{\{i,k\}}$: è il k-esimo letterale della i-esima clausola.

Sia F una formula in CNF e τ un certo assegnamento dei suoi letterali.
In generale, se una clausola di F con assegnamento τ è **vera** grazie a un sottoinsieme $L^{'} \subseteq L$ di letterali presenti in essa, la stessa clausola con assegnamento $\bar{\tau}$ sarà **vera** grazie agli altri $L - L^{'}$, ad esempio

$$\begin{aligned} F &= c_1 \wedge c_2 = (x_1 \vee x_2) \wedge (x_2 \vee x_3) \\ \tau &\equiv [x_1 = T, x_2 = F, x_3 = T] \\ \bar{\tau} &\equiv [x_1 = F, x_2 = T, x_3 = F] \end{aligned}$$

$$\begin{aligned} &\rightarrow c_1\ vera\ con\ \tau\ grazie\ a\ L^{'} = \{x_1\} \\ &\rightarrow c_1\ vera\ con\ \bar{\tau}\ grazie\ a\ L - L^{`} = \{x_1, x_2\} - \{x_1\} = \{x_2\} \\ &\rightarrow c_2\ vera\ con\ \tau\ grazie\ a\ L^{'} = \{x_3\} \\ &\rightarrow c_2\ vera\ con\ \bar{\tau}\ grazie\ a\ L - L^{`} = \{x_2, x_3\} - \{x_3\} = \{x_2\} \end{aligned}$$

Quindi, una clausola di F che è **falsa** con l'assegnamento τ (resa falsa da tutti i letterali che vi compaiono), sarà sicuramente **vera** con l'assegnamento $\bar{\tau}$ (resa vera da tutti i letterali che vi compaiono).

Sia n il numero di clausole presenti in F, allora n è banalmente il numero massimo di clausole soddisfacibili.

Possiamo distinguere due casi, usando S^* per indicare il numero di clausole restituito dall'algoritmo ottimo:

- se con τ abbiamo $r > \frac{n}{2}$ clausole **non** soddisfatte, allora con $\bar{\tau}$, per quanto detto prima, avremo almeno $r > \frac{n}{2} \geq \frac{S^*}{2}$ clausole soddisfatte;

- se con τ abbiamo $r \leq \frac{n}{2}$ clausole **non** soddisfatte, allora in τ stesso avremo $(n-r) \geq \frac{n}{2} \geq \frac{S^*}{2}$ clausole soddisfatte.

Di conseguenza, poichè l'algoritmo utilizza τ oppure $\bar{\tau}$ per ottenere sempre almeno la metà del numero totale di clausole soddisfacibili per una certa formula F, abbiamo dimostrato che è una $2 - approssimazione$.

```
1 2-approx_Max-Sat(F)
2 {
3         τ  = randomAssignment();
4       return ( (F(τ).Count ≥ n/2) ? τ : τ̄);
5 }
```

8.2 Svolgimento (Grossi)

Siano:

- a il numero di clausole soddisfatte sia da τ che da $\bar{\tau}$;
- b il numero di clausole soddisfatte solo da τ;
- c il numero di clausole soddisfatte solo da $\bar{\tau}$.

Quindi τ soddisfa un numero di clausole pari ad $a + b$, mentre $\bar{\tau}$ ne soddisfa $a + c$, ergo $\max\,(a + b,\, a + c) = a + max(b,\, c) \geq \frac{1}{2}(a + b + c)$ e quindi è una $2 - approssimazione$.

8.3 Svolgimento Maurizio

Nel problema del Max-Sat la soluzione ottima dipende sempre dall'istanza in questione. Prendiamo s come limite superiore della soluzione ottima e dimostriamo che con tale algoritmo otteniamo una soluzione contenente almeno $\frac{s}{2}$ clausole soddisfatte. Data una qualsiasi assegnazione b, distinguiamo due casi:

- Il numero di clausole soddisfatte da b é $m_0 \geq \frac{s}{2}$, dunque abbiamo raggiunto l'obiettivo.
- Il numero di clausole soddisfatte da b é $m_0 < \frac{s}{2}$, allora per comé strutturato l'algoritmo verrá restituito l'assegnazione b^c, e per complementarietá $m_1 \geq \frac{1}{2}$, dimostrando che l'algoritmo é una 2-approssimazione di Max-Sat. Per le suddette argomentazioni, la scelta di b_1, b_2,..., b_n non influenza il valore di r, poiché comunque otterremmo l'obiettivo prefissato.

8.4 Svolgimento Nicoletti

L'algoritmo risolvente é il seguente:

- Scegli casualmente $\beta = (b_1, ..., b_n)$ con $b_i \in \{0, 1\}$
- m_0 il numero di clausole soddisfatte dall'assegnamento t.c. $x_i = b_i$
- m_1 il numero di clausole soddisfatte dall'assegnamento t.c. $x_i = \overline{b_i}$
- massimo = max $\{m_0, m_1\}$
- return massimo

Vogliamo dimostrare che tale algoritmo é una r-approssimazione per un qualche $r > 1$:

Ogni clausola $c_1, ..., c_m$ viene sempre soddisfatta o da b oppure da $\overline{b}$ oppure da entrambe; non esiste che una clausola non sia soddisfatta né con l'assegnazione b né con l'assegnazione $\overline{b}$.

In virtú di questa osservazione il caso pessimo di clausole non soddisfatte si ha quando ogni clausola é soddisfatta in maniera esaustiva da b oppure da $\overline{b}$; e siccome prendiamo il massimo di queste due quantitá il caso pessimo si ha quando metá delle clausole é soddisfatta da b e l'altra metá da $\overline{b}$.

Quindi se la soluzione ottima generata da un'algoritmo ottimo é al piú pari a $m = m_0*$ quella peggiore calcolata dalla nostra soluzione é almeno $\frac{m}{2} = m_A*$:

$$\Rightarrow max(m_0, m_1) \geq \frac{m}{2}$$

$$\Rightarrow m_0* \leq m \Leftrightarrow \frac{m}{2} \geq \frac{m_0*}{2}$$

$$\Rightarrow max(m_0, m_1) \geq \frac{m_0*}{2}$$

ossia il nostro algoritmo é una 2-approssimazione e questo é l'r migliore possibile. Inoltre qualsiasi sia la scelta fatta sulle quantitá $b_1, ..., b_n$ le invarianti di tale dimostrazione non cambiano.

Chapter 9

Esercizio 9 : Approssimazione per knapsack

Nell'algoritmo greedy di approssimazione per il problema dello zaino (knapsack) visto a lezione, mostrare che quando vale la condizione $\bar{a_j} = b$ (vedere Teorema 2.1 nel capitolo disponibile sul sito) tale algoritmo ottiene la soluzione ottima.

9.1 Svolgimento Brundu

Sia G l'insieme degli indici della soluzione Greedy, O quelli della soluzione ottima, e I quelli della soluzione intermedia t.c. $G \cap O = \oslash$ Dimostreremo per assurdo che:

$$\sum_{i \in G+I} p_i < \sum_{i \in O+I} p_i$$

Le ipotesi che abbiamo sono:

- $G + I = (1, ..., j-1)$ t.c. $\frac{p_i}{a_i} > \frac{p_{i+1}}{a_{i+1}}$
- $O + I = (j, ..., n)$
- $\sum_{i \in G+I} a_i = b$
- $\sum_{i \in O+G} a_i \leq b$

9.1.1 Dimostrazione

Come detto dobbiamo dimostrare per assurdo che

$$\sum_{i \in G+I} p_i < \sum_{i \in O+I} p_i$$

Eliminiamo in $i \in G + I$ e in $i \in O + I$ la I in quanto sono elementi in comune e si semplificano.
Poiché $I = G \cap O = \emptyset$posso eliminare I dagli indici di sommatoria.
ossia che l'ottimo ottiene un profitto maggiore, e quindi migliore del greedy.

$$\sum_{i \in G} p_i * \frac{a_i}{a_i} < \sum_{i \in O} p_i * \frac{a_i}{a_i}$$

Sia

$$k = arg_m in_{i \in G} \frac{p_i}{a_i}$$

e

$$k' = arg_m ax_{i \in O} \frac{p_i}{a_i}$$

quindi possiamo riscrivere

$$\frac{p_k}{a_k} * \sum_{i \in G} a_i < \frac{p_{k'}}{a_{k'}} * \sum_{i \in O} a_i$$

Divido e moltiplico entrambi i membri per la quantita $\sum_{i \in O} a_i$:

$$\frac{p_k}{a_k} * \frac{\sum_{i \in G} a_i}{\sum_{i \in O} a_i} < \frac{p'_k}{a'_k} * \frac{\sum_{i \in O} a_i}{\sum_{i \in O} a_i}$$

Chiamiamo la quantitá $\frac{\sum_{i \in G} a_i}{\sum_{i \in O} a_i}$ = d avremo:

$$\frac{p_k}{a_k} * d < \frac{p'_k}{a'_k}$$

Quest'ultimo passaggio significa che k' per il nostro ordinamento fatto nelle ipotesi ($\frac{p_i}{a_i} > \frac{p_{i+1}}{a_{i+1}}$) precede k, MA $k' \in O$ ed é assurdo che precede $k \in G$ quindi non é possibile che l'ottimo ottiene una soluzione migliore del greedy.

9.2 Svolgimento Giuseppe

Per ipotesi sappiamo che, quando l'algoritmo Greedy considera il j-esimo elemento della sequenza di input X, dove $|X| = n$, tale elemento non viene inserito nello zaino poiché quest'ultimo è saturo a causa dell'aggiunta di tutti gli elementi precedentemente incontrati (aventi indice minore di j):

$$\bar{a_j} = \sum_{i=1}^{j-1} a_i = b$$

Poichè, inizialmente, l'algoritmo ordina gli elementi in modo descrescente rispetto al rapporto $\frac{p_i}{a_i}$ $(1 \leq i \leq n)$, lo zaino contiene gli elementi con

rapporto maggiore.

A questo punto, supponiamo di sostituire un sottoinsieme Z di elementi dello zaino con un sottoinsieme T di elementi non presenti nello zaino, successivi nell'ordinamento.

Per poter effettuare tale sostituzione, il peso totale degli elementi che vogliamo aggiungere allo zaino può essere al massimo pari al peso totale di quelli che vogliamo togliere dallo stesso, altrimenti si andrebbe oltre la capacità massima b:

$$\sum_{x \in T} a_x \leq \sum_{y \in Z} a_y$$

Inoltre, gli elementi $x \in Z$ avranno sicuramente un rapporto $\frac{p_x}{a_x}$ $(1 \leq x \leq j-1)$ maggiore o uguale a quello $\frac{p_y}{a_y}$ $(y \in [j, n])$ degli elementi $y \in T$, altrimenti l'ordinamento decrescente basato sul rapporto stesso avrebbe dato precedenza agli elementi di T piuttosto che a quelli di Z.

Poichè abbiamo visto che gli elementi di Z hanno un peso totale che è sempre maggiore o uguale al peso totale degli elementi di T, allora, per avere la precedenza nell'ordinamento basato sul rapporto, dovranno avere un profitto totale obbligatoriamente maggiore o uguale rispetto al profitto totale degli elementi di T:

$$\sum_{y \in Z} p_y \geq \sum_{x \in T} p_x$$

Abbiamo, quindi, dimostrato che sotto l'ipotesi $\bar{a_j} = b$, effettuando una qualunque sostituzione, non possiamo mai migliorare la soluzione trovata dall'algoritmo greedy, ma al massimo possiamo eguagliarla effettuando una sostituzione che pareggia sia lo spazio occupato che il profitto. Ciò implica che la soluzione ottenuta dall'algoritmo greedy è ottima e che quindi, nel caso discusso, coincide con quella restituita dall'algoritmo ottimo.

Chapter 10

Esercizio 10: MAX-CUT

Il problema MAX-CUT è NP-hard ed è definito come segue per un grafo non orientato $G = (V, E)$. Una partizione di nodi $(C, V - C)$ con $C \in V$ si chiama cut, o taglio. Un arco $e = (v, w)$ con $v \in C$ e $w \in V - C$ si chiama arco di taglio (ricordando che (v, w) e (w, v) denotano lo stesso arco in un grafo non orientato). Il numero di archi di taglio definisce la dimensione del cut $(C, V - C)$. Poichè cambiando taglio, può cambiare la sua dimensione, il problema richiede di trovare il taglio di dimensione massima e quindi gli archi di taglio corrispondenti. Dimostrare che il seguente algoritmo randomizzato è una 2- approssimazione in valore atteso, ossia che il numero medio di archi di taglio così individuati è in media almeno la metà di quelli del taglio massimo. (1) Per ogni nodo v $\in$ V , lancia una moneta equiprobabile: se viene testa, inserisci v in C; altrimenti (viene croce), inserisci v in V - C. (2) Inizializza T all'insieme vuoto. Per ogni arco (v,w) $\in$ E, tale che v $\in$ C e w $\in$ V - C, aggiungi (v, w) all'insieme T . Restituisci C e T come soluzione approssimata.

10.1 Svolgimento

Per ogni nodo $v \in V$, lancia una moneta equiprobabile: se viene testa, inserisci v in C; altrimenti (viene croce), inserisci v in $(V - C)$.

Successivamente viene inizializzato T all'insieme vuoto. Per ogni arco $(v, w) \in E$, tale che $v \in C$ e $w \in V - C$, aggiungi (v, w) all'insieme T. Restituisci C e T come soluzione approssimata.

Definiamo la variabile indicatrice: $X(u, v) = \begin{cases} 1, & \text{se } (u, v) \in T \\ 0, & \text{altrimenti} \end{cases}$

Allora avremmo che:

$$Pr[X(u, v) = 1] = Pr[u \in C \wedge v \in V - C] + Pr[u \in V - C \wedge v \in C] = \frac{1}{2}$$

da cui

$$E[X(u,v)] = \frac{1}{2}$$

Definiamo poi $X = \sum_{(u,v)\in E} X_{(u,v)}$. Allora:

$$E[X] = \sum_{(u,v)\in E} E[X_{(u,v)}] = \frac{1}{2}|E|$$

Detto $\tilde{T}$ il taglio ottimo, si ha ovviamente che:

$$|\tilde{T}| \leq |E|$$

da cui

$$E[|T|] = E[X] = \frac{1}{2}|E| \geq \frac{1}{2}|\tilde{T}|$$

C.V.D

Chapter 11

Esercizio 11: Funzioni Hash Uniformi

Mostrare che la famiglia di funzoni hash $H = h(x) = ((ax + b)modp)modm)$ (quasi) pairwise independent, dove $a, b \in [m]$ con $a \neq 0$ e p un numero primo sufficientemente grande $(m + 1 \leq p \leq 2m)$. La pairwise independent la k-wise independent con $k = 2$.

11.1 Svolgimento

Dobbiamo mostrare che preso $h \in H$ t.c. $h : [n] \to [m]$, date due chavi $x, y \in [n]$ t.c. $x \neq y$, e dati due valori $u, v \in [m]$ vale che:

$$Pr[h(x) = u \wedge h(y) = v] \approxeq Pr[h(x) = u] * Pr[h(y) = v] \approxeq \frac{1}{m} * \frac{1}{m} \cong \frac{1}{m^2}$$

ossia pairwise independent.

Dimostrazione.

Contiamo quante sono le chiavi x t.x. $h(x) = u$ (ottenendo cosí i casi favorevoli alla $Pr[h(x) = u]$) e y t.c. $h(y) = v$ (ottenendo cosí i casi favorevoli alla $Pr[h(y) = v]$).

Siano $r, s \in (0, ..., p - 1)$, siccome $x \neq y$, l'equazioni:

$$ax + b = r(modp)$$

$$ay + b = s(modp)$$

hanno un unica soluzione x in $[p]$ e y in $[p]$ in quanto $a \neq 0$ ed sicuramente invertibile.

Ora ci chiediamo quante scelte di x e quindi di r mi portano a una tale uguaglianza: $r = u(modm)$ e quante scelte di y ossia di s mi portano a: $s = v(modm)$. Le possibili scelte per r sono almeno $\lfloor \frac{p}{m} \rfloor$ e al più $\lceil \frac{p}{m} \rceil$, stessa cosa vale per s.

Sia t il numero dei possibili valori per x t.c. $h(x) = u$ abbiamo:

$$\lfloor \frac{p}{m} \rfloor \leq t \leq \lceil \frac{p}{m} \rceil$$

altrettanti per y affinchè $h(y) = v$.

Quindi per quanto riguarda la chiave x avremo:

$$\frac{\lfloor \frac{p}{m} \rfloor}{p} \leq Pr[h(x) = u] \leq \frac{\lceil \frac{p}{m} \rceil}{p}$$

stessa cosa per la chiave y .

In totale le coppie (x, y) possibili sono $p(p-1)$ (p per x e $p-1$ per y):

$$\frac{(\lfloor \frac{p}{m} \rfloor)^2}{p(p-1)} \leq Pr[h(x) = u \wedge h(y) = v] \leq \frac{(\lceil \frac{p}{m} \rceil)^2}{p(p-1)} \cong \frac{1}{m^2} \leq Pr[h(x) = u \wedge h(y) = v] \leq \frac{1}{m^2}$$

da cui

$$Pr[h(x) = u \wedge h(y) = v] \cong \frac{1}{m^2}$$

■

Chapter 12

Esercizio 12: Count min sketch: Prodotto Scalare

Mostrare come usare il paradigma del count-min sketch per approssimare il prodotto scalare ($\sum_{k=1}^{n} F_a[k] * F_b[k]$).

12.1 Svolgimento altri (molto meglio)

12.1.1 Utilizzo del count-min sketch

Ricordiamo che il $prodotto\ scalare$ (o $inner\ product$) tra due vettori di interi a e b di lunghezza n è definito come

$$a \cdot b = \sum_{i=1}^{n} a[i] \cdot b[i]$$

Nell'ambito del $Data\ Streaming$, non possiamo memorizzare gli elementi dei due array sui quali vogliamo calcolare il prodotto scalare, quindi abbiamo bisogno di un approssimazione che mappa il problema del prodotto scalare su quello del $count\text{-}min\ sketch$:

- vediamo i due array di input a e b come due frequence array F_a e F_b, il cui sketch degli elementi è memorizzato tramite due tabelle di contatori T_a e T_b, entrambe di $r \times c$ colonne;
- per indicizzare T_a e T_b utilizziamo, come per il $count\text{-}min\ sketch$, delle funzioni hash scelte uniformemente e indipendentemente da una famiglia di funzioni $H\ pairwise\text{-}independent$ e associamo r celle ad ogni elemento degli array.

Se manteniamo l'invariante che $\forall i \in [0, n)\ .\ a[i] \geq 0 \wedge b[i] \geq 0$, possiamo avere un approssimazione del prodotto scalare data da

$$a \tilde{\cdot} b = \min_{1 \leq j \leq r} RowProd_j$$

dove $RowProd_j$ è il prodotto scalare locale tra la j-esima riga di a e la j-esima riga di b, ovvero

$$RowProd_j = \sum_{k=1}^{c} T_a[j][h_j(k)] \cdot T_b[j][h_j(k)]$$

12.1.2 Analisi

Per l'approccio sopra descritto, possiamo dimostrare che valgono i seguenti lowerbound e upperbound

$$a \cdot b \leq a \tilde{\cdot} b \leq a \cdot b + \varepsilon \|a\| \|b\| \text{ , } con\, probabilità \geq 1 - \delta$$

Prima di tutto, notiamo che l'approssimazione $a \tilde{\cdot} b$, come accadeva nel count-min sketch, possiede un certo quantitativo di "spazzatura" X, dovuto alle collisioni delle funzioni hash a partire da elementi diversi, durante l'indicizzazione delle colonne delle due tabelle. Avremo quindi

$$a \tilde{\cdot} b = a \cdot b + X = \sum_{i=1}^{n} a[i] \cdot b[i] + \sum_{u,v} I_{juv} \cdot a[u] \cdot b[v]$$

con

$$I_{juv} = \begin{cases} 1 & \text{se } u \neq v \wedge h_j(u) = h_j(v) \\ 0 & \text{altrimenti} \end{cases}$$

Dividiamo, quindi, la doppia disuguaglianza in due parti, in modo da schematizzare la dimostrazione.

Parte 1: $a \cdot b \leq a \tilde{\cdot} b$

Poichè per ipotesi $\forall i \in [0, n) \ . \ a[i] \geq 0 \wedge b[i] \geq 0$, allora

$$X \geq 0 \Rightarrow a \cdot b \leq a \tilde{\cdot} b = a \cdot b + X$$

cvd.

Parte 2: $a \tilde{\cdot} b \leq a \cdot b + \varepsilon \|a\| \|b\| \text{ , } con\, probabilità \geq 1 - \delta$

Vogliamo dimostrare che

$$\mathcal{P}\left(a \tilde{\cdot} b \leq a \cdot b + \varepsilon \|a\| \|b\|\right) \geq 1 - \delta$$

Utilizzando il complementare abbiamo

$$\begin{aligned}[\mathcal{P}\left(a \tilde{\cdot} b \leq a \cdot b + \varepsilon\|a\|\|b\|\right)]^C &= \mathcal{P}\left(a \tilde{\cdot} b > a \cdot b + \varepsilon\|a\|\|b\|\right) \\ &= \mathcal{P}\left(a \cdot b + X > a \cdot b + \varepsilon\|a\|\|b\|\right) \\ &= \mathcal{P}\left(X > \varepsilon\|a\|\|b\|\right) \\ &= \mathcal{P}\left(\sum_{u,v} I_{juv} \cdot a[u] \cdot a[v] > \varepsilon\|a\|\|b\|\right)\end{aligned}$$

che possiamo boundare tramite la **disuguaglianza di Markov**:

$$\mathcal{P}\left(X > \varepsilon\|a\|\|b\|\right) \leq \frac{\mathbb{E}\left[X\right]}{\varepsilon\|a\|\|b\|}$$

Abbiamo quindi bisogno di calcolare il valore atteso di X, e quindi di I_{juv}:

$$\begin{aligned}\mathbb{E}\left[I_{juv}\right] &= \mathcal{P}\left(u \neq v \wedge h_j(u) = h_j(v)\right) \\ &= \bigcup_{a=1}^{c} \mathcal{P}\left(u \neq v \wedge h_j(u) = a \wedge h_j(v) = a\right) \\ &= \sum_{a=1}^{c} \underbrace{\mathcal{P}\left(u \neq v \wedge h_j(u) = a \wedge h_j(v) = a\right)}_{h_j \text{ uniformi e pair-wise indip.}} \\ &\leq \sum_{a=1}^{c} \frac{1}{c^2} = \frac{c}{c^2} = \underbrace{\frac{1}{c} = \frac{\varepsilon}{e}}_{c = \frac{e}{\varepsilon}}\end{aligned}$$

che usiamo in

$$\begin{aligned}\mathbb{E}\left[X\right] &= \mathbb{E}\left[\sum_{u,v} I_{juv} \cdot a[u] \cdot b[v]\right] = \sum_{u,v} \mathbb{E}\left[I_{juv}\right] \cdot a[u] \cdot b[v] \\ &= \frac{\varepsilon}{e} \sum_{u,v} a[u] \cdot b[v] \leq \frac{\varepsilon}{e} \|a\|\|b\|\end{aligned}$$

Usando il risultato ottenuto nella disuguaglianza di Markov, avremo:

$$\mathcal{P}\left(X > \varepsilon\|a\|\|b\|\right) \leq \frac{\mathbb{E}\left[X\right]}{\varepsilon\|a\|\|b\|} = \frac{\varepsilon\|a\|\|b\|}{e\,\varepsilon\|a\|\|b\|} = \frac{1}{e} < \underbrace{\frac{1}{2^r} = \delta}_{r = log_2 \frac{1}{\delta}}$$

Ovvero

$$\mathcal{P}\left(X > \varepsilon\|a\|\|b\|\right) < \delta \Leftrightarrow \mathcal{P}\left(X \leq \varepsilon\|a\|\|b\|\right) \geq 1 - \delta$$

cvd.

12.2 Svolgimento nostro

Dati due vettori $a[i]$ e $b[i]$ il loro prodotto vettoriale è definito come $a \odot b = \sum_{i=1}^{n} a[i]b[i]$. Creiamo dunque due count-min sketch, uno per il vettore a e uno per il vettore b, effettuiamo il prodotto vettoriale fra le righe e poi scegliere il minimo fra questi valori:

$$P_j = \sum_{i=1}^{c} T_a(j,i)T_b(j,i)$$

$$\widetilde{(a \odot b)} = \min_{0 \leq j \leq r} P_j$$

Possiamo dimostrare che vale

$$a \odot b \leq \underbrace{\widetilde{a \odot b} \leq a \odot b + \varepsilon||a|| \cdot ||b||}_{\Pr > 1-\delta}$$

Dimostrazione.
Riscriviamo l'*inner product* come

$$\widetilde{(a \odot b_j)} = \sum_{i=1}^{n} a_i b_i + \sum_{p \neq q, h_j(p) = h_j(q)} a_p b_p$$

Dove il secondo termine rappresenta la solita "sporcizia" vista nella dimostrazione del count-min sketch.

Calcoliamo adesso il seguente termine:

$$\mathbb{E}(\widetilde{a \odot b_j} - a \odot b) = \sum_{p \neq q} \Pr[h_j(p) = h_j(q)] a_p b_p \leq \sum_{p \neq q} \frac{\varepsilon a_p b_p}{e} \leq \frac{\varepsilon||a|| \cdot ||b||}{e}$$

Notiamo che le ultime due approssimazioni sono effettuate utilizzando fatti già noti: il termine $\Pr[h_j(p) = h_j(q)]$ lo approssimiamo con la speranza della variabile indicatrice $I_{x,j,k}$ introdotta nella dimostrazione del count-min sketch; le norme vengono invece introdotte perchè si nota che stiamo sommano su tutti gli elementi dei vettori a meno di elementi di indici uguali.

$$E[I_{xjk}] = Pr[p \neq q \wedge h_j(p) = h_j(q)] = Pr[\cup_{a=1}^{c} p \neq q \wedge h_j(p) = h_j(q)] =$$

$$= \sum_{a=1}^{c} Pr[p \neq q \wedge h_j(p) = h_j(q)] \leq \sum_{a=1}^{c} \frac{1}{c} = \frac{1}{c} = \frac{\epsilon}{e}$$

Adesso possiamo utilizzare il termine appena calcolato nella Markov's inequality:

$$\Pr[\widetilde{a \odot b_j} - a \odot b > \varepsilon ||a|| \cdot ||b||] < \frac{\varepsilon ||a|| \cdot ||b||}{e\varepsilon ||a|| \cdot ||b||} = \frac{1}{e} < \delta$$

■

Chapter 13

Esercizio 13: Count min sketch: Interval Query

Mostrare e analizzare come usare il paradigma del count-min sketch per rispondere in modo approssimato alla interval query (i, j) per calcolare $\sum_{k=1}^{j} F[k]$. Notare che per avere una probabilitá al piú δ che $\sum_{l=1}^{t} c_l > \sum_{k=1}^{j} F[k] + 2\epsilon log n||F||$, non basta dire che é al piú δ la probabilitá di errore di ciascun contatore c_l: occorre infatti vedere ciascun contatore come il vero valore piú il residuo, come visto nel count-min sketch di base, e sommare assieme i t valori ottenendo V e i t residui ottenendo X e poi procedere con la disuguaglianza di Markov per ottenere che δ é un limite superiore alla probabilitá che $V + X > \sum_{k=1}^{j} F[k] + 2\epsilon log n||F||$ (cioé vedere la somma dei contatori come fosse un unico meta-contatore).

13.1 Svolgimento

Si vuole fare la somma di una sottosequenza di uno stream F a partire dall'elemento i fino all'elemento j.

Definiamo il vettore $a[i]$ il nostro stream in ingresso e il vettore $b[i]$ un vettore che contenga tutti 0 tranne nelle posizioni comprese nell'intervallo $[i, j]$ dove sono presenti degli 1.

Si presenta però un problema, infatti la disequazione:

$$\widetilde{a \odot b} \leq a \odot b + \varepsilon ||a|| \cdot ||b||$$

può avere il termine a destra arbitrarimanete grande, dato che $||b|| = j - i + 1$.

Per risolverlo possiamo allora dividere l'intervallo utilizzando i *dyadic ranges* $[x2^y + 1, ..., (x + 1)2^y]$ con $y \in [0, \log_2 n]$ ed ogni volta che ci arriva un dato andiamo a incrementare tutti ii count min sketch negli intervalli corriispondenti.

Assumiamo che il nostro intervallo in questione sia lungo $n = 1024$ bit. Possiamo definire $\log_2 n = 10$ insiemi di *dyadic ranges*:

- 1024 intervalli di lunghezza 1,
- 512 intervalli di lunghezza 2,
- 256 intervalli di lunghezza 3,
- ...

Vediamo che un generico punto nell'intervallo appartiene a $\log_2 n$ *dyadic ranges*, uno per ogni y nell'intervallo $[0, \log_2 n]$. Un generico intervallo $[i, j]$ può essere ridotto in al più $2 \log_2 n$ *dyadic ranges*.

Operiamo dunque in questo modo: realizziamo $\log_2 n$ count-min sketch, uno per ogni insieme di *dyadic ranges*, che aggiorniamo ad ogni update di elementi.

Quando si vuole calcolare un range query, si riduce l'intervallo nei $2 \log_2 n$ *dyadic ranges*, si calcolano tante query, una per ogni *dyadic range* al relativo sketch associato (se un *dyadic range* è lungo 2^i allora prendiamo il i-esimo count-min sketch). Calcoliamo infine la somma di tutti i valori e la restituiamo

Definiamo con $a[l, r]$ il risultato effettivo del range query, e con $\tilde{a}[l, r]$ il risultato approssimato con il procedimento visto prima; possiamo dimostrare che con probabilità superiore a $1 - \delta$ vale:

$$\tilde{a}[l, r] \leq a[l, r] + 2\varepsilon \log n ||a||$$

Dimostrazione. Ricordiamo che definiamo $\tilde{a}[l, r] = a[l, r] + X_{l,r}$ dove $X_{l,r,j}$ rappresenta la "sporcizia", che deriva dai fattori additivi di ogni singolo count-min sketch che viene considerato (possono essere al massimo $2 \log n$) quando si esegue una query.

Il valore atteso della "sporcizia" risulta allora:

$$\mathbb{E}(l, r, j) = 2 \log n \frac{\varepsilon}{e} ||a||$$

Notiamo che il termine

$$||a||$$

deriva da un'approssimazione, dato che stiamo facendo la somma su un range di a, lo estendiamo a tutto a e ne calcoliamo dunque la norma.

Adesso possiamo utilizzare la Markov inequality per dimostrare la proprietà:

$$\Pr[\tilde{a}[l, r] \geq a[l, r] + 2\varepsilon \log n ||a||] =$$

$$= Pr[\tilde{a}[l, r] - a[l, r] > 2\varepsilon \log n ||a||] \leq \frac{2 \log n \frac{\varepsilon}{e} ||a||}{2\varepsilon \log n ||a||} \leq \frac{1}{e} \leq \delta$$

■

Chapter 14

Esercizio 14: Count min sketch estensione

[Count-min sketch: estensione] Estendere l'analisi vista a lezione permettendo di incrementare e decrementare i contatori con valori arbitrati. Invece di avere le operazioni F[i]++ e F[i]$--$, l'elemento generico dello stream contiene una coppia (i,v) dove i é una item e v é un intero qualsiasi: l'operazione diventa F [i] = F [i] + v. Notare che l'incremento e il decremento unitari di F [i] possono essere ora visti come (i, 1) e (i, -1).

14.1 Svolgimento altri (molto meglio)

14.1.1 Riepilogo del problema

Nel problema del count-min sketch **point query**, vogliamo contare il numero di occorrenze degli elementi su uno stream che ne contiene n, dove n é un numero intero molto grande. Intuitivamente, potremmo memorizzare un array F di n contatori tale che, ogni volta che incontriamo un elemento (i, v) nello stream, eseguiamo l'operazione $F[i] = F[i] + v$. Purtroppo, peró, essendo nel contesto del $Data\ Streaming$ non é possibile memorizzare F (abbiamo un quantitativo di memoria pari a $polylog(n)$), quindi possiamo soltanto approssimare il risultato della point-query, utilizzando:

- una tabella $T^{r \times c}$ di contatori;
- r funzioni hash $h_1, h_2, \ldots, h_r$ scelte in modo uniforme e indipendente da una famiglia $H\ pairwise\ independent$;
- associando r celle della tabella T ad ogni elemento i dello stream, dove ogni cella é indicizzata nel seguente modo

$$T[1][h_1(i)], T[2][h_2(i)], \ldots, T[r][h_r(i)]$$

- aggiungendo, ad ogni occorrenza dell'elemento (i, v) sullo stream, il valore v a tutte le r celle associate ad i

$$\forall j \in [1, r].\ T[j][h_j(i)] = T[j][h_j(i)] + v$$

L'approssimazione del numero di occorrenze di un generico elemento i dello stream é quindi ottenuta nel modo seguente:

$$\tilde{F}[i] = \min_{1 \leq j \leq r} T[j][h_j(i)]$$

Nel caso in cui valga l'invariante $\forall i \in [1, n].\ F[i] \geq 0$, valgono i seguenti lowerbound e upperbound per la soluzione approssimata

$$F[i] \leq \tilde{F}[i] \leq F[i] + \varepsilon \|F\|,\ \textit{con probabilità} \geq 1 - \delta$$

Rimuovendo l'invariante, modifichiamo l'approssimazione per il generico elemento i dello stream, la quale diventa

$$\tilde{F}[i] = median_{1 \leq j \leq r}\ T[j][h_j(i)]$$

con i rispettivi lowerbound e upperbound

$$F[i] - 3\varepsilon \|F\| \leq \tilde{F}[i] \leq F[i] + 3\varepsilon \|F\|,\ \textit{con probabilità} \geq 1 - \delta^{\frac{1}{4}}$$

14.1.2 Analisi

Dimostriamo quindi che

$$\mathcal{P}\left(F[i] - 3\varepsilon \|F\| \leq \tilde{F}[i] \leq F[i] + 3\varepsilon \|F\|\right) \geq 1 - \delta^{\frac{1}{4}}$$

Innanzitutto, sappiamo che

$$\tilde{F}[i] = F[i] + X_{ji}$$

con X_{ji} che rappresenta la "spazzatura" dovuta alle collisioni con quegli altri elementi dello stream di input che vanno ad incrementare il contatore della stessa cella della tabella (ottenuta tramite la stessa funzione hash h_j). Detto formalmente:

$$X_{ji} = \sum_{k=1}^{n} I_{jik} F[k]$$

con

$$I_{jik} = \begin{cases} 1 & \textit{se } i \neq k \wedge h_j(i) = h_j(k) \\ 0 & \textit{altrimenti} \end{cases}$$

Possiamo allora riscrivere ciò che vogliamo dimostrare come

$$\begin{aligned}\mathcal{P}\left(F[i] - 3\varepsilon\|F\| \leq \tilde{F}[i] \leq F[i] + 3\varepsilon\|F\|\right) &= \mathcal{P}\left(F[i] - 3\varepsilon\|F\| \leq F[i] + X_{ji} \leq F[i] + 3\varepsilon\|F\|\right) \\ &= \mathcal{P}\left(-3\varepsilon\|F\| \leq X_{ji} \leq 3\varepsilon\|F\|\right) \\ &= \mathcal{P}\left(|X_{ji}| \leq 3\varepsilon\|F\|\right)\end{aligned}$$

Considerando ora il complementare della probabilità appena descritta, possiamo trovargli un upperbound utilizzando la **disuguaglianza di Markov**:

$$\left[\mathcal{P}\left(|X_{ji}| \leq 3\varepsilon\|F\|\right)\right]^C = \mathcal{P}\left(|X_{ji}| > 3\varepsilon\|F\|\right) \leq \frac{\mathbb{E}\left[|X_{ji}|\right]}{3\varepsilon\|F\|}$$

per il quale, però, abbiamo bisogno di una stima di $\mathbb{E}\left[|X_{ji}|\right]$.

Per stimare $\mathbb{E}\left[|X_{ji}|\right]$ abbiamo bisogno di

$$\begin{aligned}\mathbb{E}\left[I_{jik}\right] &= \mathcal{P}\left(i \neq k \wedge h_j(i) = h_j(k)\right) \\ &= \bigcup_{a=1}^{c} \mathcal{P}\left(i \neq k \wedge h_j(i) = a \wedge h_j(k) = a\right) \\ &= \sum_{a=1}^{c} \underbrace{\mathcal{P}\left(i \neq k \wedge h_j(i) = a \wedge h_j(k) = a\right)}_{h_j \text{ uniformi}} \\ &\leq \sum_{a=1}^{c} \frac{1}{c} = \frac{c}{c^2} = \underbrace{\frac{1}{c} = \frac{\varepsilon}{e}}_{c = \frac{e}{\varepsilon}}\end{aligned}$$

in quanto

$$\begin{aligned}\mathbb{E}\left[|X_{ji}|\right] &= \mathbb{E}\left[\left|\sum_{k=1}^{n} I_{jik} F[k]\right|\right] \\ &\leq \sum_{k=1}^{n} \mathbb{E}\left[|I_{jik}|\right] |F[k]| = \frac{\varepsilon}{e} \sum_{k=1}^{n} |F[k]| = \frac{\varepsilon}{e}\|F\|\end{aligned}$$

Sostituendo ora il valore ottenuto nella disuguaglianza di Markov, possiamo avere un lowerbound numerico per la probabilità complementare vista precedentemente

$$[\mathcal{P}(|X_{ji}| \leq 3\varepsilon\|F\|)]^C = \mathcal{P}(|X_{ji}| > 3\varepsilon\|F\|)$$
$$\leq \frac{\mathbb{E}[|X_{ji}|]}{3\varepsilon\|F\|} = \frac{\varepsilon\|F\|}{3e\varepsilon\|F\|} = \frac{1}{3e} < \frac{1}{8}$$

Definiamo adesso la variabile indicatrice

$$Y_j = \begin{cases} 1 & \text{se } |X_{ji}| > 3\varepsilon\|F\| \\ 0 & \text{altrimenti} \end{cases}$$

dove

$$Y = \sum_{j=1}^{r} Y_j$$

Quindi Y sarà uguale al numero di elementi i che possiedono un quantitativo di "spazzatura" $|X_{ji}|$ che è superiore a $3\varepsilon\|F\|$.

Notiamo che, per definizione, il mediano $\tilde{F}[i]$ è un valore approssimato corretto per un elemento dell'input i solo se esistono, per lo stesso i, **meno** di $\frac{r}{2}$ celle $T[j][h_j(i)]$ per le quali $|X_{ji}| > 3\varepsilon\|F\|$ (ovvero $Y < \frac{r}{2}$) , cioè se per almeno metà delle celle associate ad i vale $|X_{ji}| \leq 3\varepsilon\|F\|$.

Considerando il caso sfavorevole per cui il numero di celle di i che possiede $|X_{ji}| > 3\varepsilon\|F\|$ sia superiore o uguale alla metà del numero delle celle stesse (ovvero $Y \geq \frac{r}{2}$) e applicando gli **Chernoff's Bounds** con parametri $\mu = \mathbb{E}[Y] = rp$ e $(1+\lambda)\mu = \frac{r}{2}$, otteniamo:

$$\mathcal{P}\left(Y \geq \frac{r}{2}\right) = \mathcal{P}(Y \geq (1+\lambda)\mu) < \left(\frac{e^\lambda}{(1+\lambda)^{1+\lambda}}\right)^\mu = \frac{1}{e^\mu}\left(\frac{e}{1+\lambda}\right)^{(1+\lambda)\mu}$$
$$= \frac{1}{e^\mu}\left(\frac{e}{1+\lambda}\right)^{(1+\lambda)\mu} = \frac{1}{e^{rp}}(2ep)^{r/2} \leq \underbrace{\frac{1}{2^{r/4}} = \delta^{\frac{1}{4}}}_{r = log_2 \frac{1}{\delta}}$$

Passando al reciproco della disequazione ottenuta, abbiamo:

$$2^{\frac{r}{4}} \leq \underbrace{e^{rp}}_{\geq 1} \frac{1}{(2ep)^{r/2}}$$

Quindi ci basta prendere $\frac{1}{2pe} > \sqrt{2} \Leftrightarrow p < \frac{1}{2\sqrt{2}e}$ in quanto $p < \frac{1}{8}$ e $2\sqrt{2}e \approxeq 7.67$ per far si che

$$\begin{aligned}\mathcal{P}\left(Y < \frac{r}{2}\right) &= \mathcal{P}\left(|X_{ji}| \leq 3\varepsilon\|F\|\right) \\ &= \mathcal{P}\left(F[i] + |X_{ji}| \leq F[i] + 3\varepsilon\|F\|\right) \\ &= \mathcal{P}\left(\tilde{F}[i] \leq F[i] + 3\varepsilon\|F\|\right) \geq 1 - \delta^{\frac{1}{4}}\end{aligned}$$

cvd.

14.2 Svolgimento

E' possibile rimuovere l'invariante sul count-min sketch che avevamo imposto in precendenza, ovvero $F[i] \geq 0$

Così facendo dobbiamo però ridefinire $\tilde{F}[i]$ come il mediano fra i valori dei contatori ovvero

$$\tilde{F}[i] = \text{median}_{0 \leq j \leq r}\ T(j, h_j(i))$$

Rimuovendo l'invariante possiamo dimostrare, usando i Chernoff's Bound che vale la seguente relazione:

$$F[i] - 3\varepsilon||F|| \leq \tilde{F}[i] \leq F[i] + 3\varepsilon||F||$$

$$\text{con } p \geq 1 - \delta^{1/4}$$

che riscrivendo $\tilde{F}[i] = F[i] + X_{ij}$ equivale a dire che:

$$-3\varepsilon||F|| \leq X_{ij} \leq +3\varepsilon||F||$$

ovvero

$$|X_{ij}| \leq 3\varepsilon||F||$$

Dimostrazione. Applichiamo dapprima la Markov inequality:

$$\Pr[|X_{ij}| \geq 3\varepsilon||F||] < \frac{\overbrace{\mathbb{E}[|X_{ij}|]}^{\varepsilon/e||F||}}{3\varepsilon||F||} < \frac{1}{3e} < \frac{1}{8}$$

Definiamo:

$$Y_j = \begin{cases} 1 & \text{se } |X_{ij}| > 3\varepsilon||F|| \text{ con } p < 1/8 \\ 0 & \text{altrimenti} \end{cases}$$

come una variabile indicatrice che vale 1 se il contatore selezionato non rispetta il vincolo che stiamo cercando di dimostrare. Ci verrá ritornato un mediano che rispetta il nostro vincolo $|X_{ij}| \leq 3\varepsilon||F||$ se, prese r righe ci sono meno di $r/2$ contatori i' per cui vale $|X_{i'j}| > 3\varepsilon||F||$.

Per calcolare dunque la probabilità di errore consideriamo il caso in cui il numero di righe errate è maggiore di $1/2$ che equivale a dire

$$Y \geq \frac{r}{2}$$

Applichiamo adesso i Chernoff's Bound con parametro $\mu = rp$ e $(1 + \lambda)\mu = r/2$:

$$\Pr[Y \geq (1+\lambda)\mu] < \left(\frac{e^{\lambda}}{(1+\lambda)^{1+\lambda}}\right)^{\mu} = \frac{1}{e^{\mu}}\left(\frac{e}{1+\lambda}\right)^{(1+\lambda)\mu}$$

Adesso maggioriamo:

$$\frac{1}{e^{\mu}}\left(\frac{e}{1+\lambda}\right)^{(1+\lambda)\mu} = \frac{1}{e^{rp}}(2ep)^{r/2} \leq \frac{1}{2^{r/4}} = \delta^{1/4}$$

Invertiamo la disequazione:

$$2^{\frac{r}{4}} \leq \underbrace{e^{rp}}_{\geq 1}\frac{1}{(2ep)^{r/2}}$$

Quindi ci basta prendere $\frac{1}{2pe} > \sqrt{2} \;\Leftrightarrow\; p < \frac{1}{2\sqrt{2}e}$ in quanto $p < \frac{1}{8}$ e $2\sqrt{2}e \approxeq 7.67$ ■

Chapter 15

Esercizio 15:Ordinamento in memoria esterna

[Ordinamento in memoria esterna] Nel modello EMM (external memory model), mostrare come implementare il k-way merge ((k+1)B $\leq$ M), ossia la fusione di k sequenze individualmente ordinate e di lunghezza totale N, con costo I/O di O(N/B) dove B é la dimensione del blocco. Minimizzare e valutare il costo di CPU. Analizzare il costo del merge sort (I/O complexity, CPU complexity) che utilizza tale k-way merge.

15.1 Svolgimento

Per ordinare una sequenza S di dimensione N in EMM, possiamo allora spezzettare la sequenza andando a creare un numero di run $n = \lceil \frac{N}{M} \rceil$ ciascuno di dimensione M. Ciascuno di questi run viene caricato in memoria principale e ordinati individualmente con qualunque algoritmo di sorting per RAM. Il numero n di run é decisivo per il passo successivo ossia il k-way MergeSort. Dobbiamo quindi implementare il k-way Merge con $M \geq (k+1)B \Leftrightarrow k \leq \frac{M}{B} - 1$. Ossia nella memoria RAM di dimensione M vi sono k buffer di input e 1 buffer di output ciascuno di dimensione B. Prima di procedere con il k-way Merge dobbiamo verificare 2 casi:

- se $n \leq k$ allora il k-way Merge verrà eseguito una sola volta: in un'unico passo di fusione verrá creato in memoria esterna un'unico run contenente tutti gli elementi ordinati.
- se invece $n > k$ allora non sarà possibile effettuare il merge in un solo passo, ad ogni passo i run verranno confrontati a gruppi di k, e ciasucn gruppo verrà fuso in un'unico run, e via dicendo fino a che non verrá creato un unico run di dimensione N.

Analizziamo ora il k-way merge sort:

L'idea é quella di servirsi di una coda di prioritá per minimizzare i costi della CPU, si potrebbe quindi implementare attraverso un Min Heap in modo da avere un costo logaritmico ad ogni passo. L'algoritmo che implementeremo prende in input n sottosequenze alla volta individualmente ordinate e le trasferisce nella RAM. Ogni run è composta da $\frac{M}{B}$ blocchi che saranno fusi in un'unica run. Ricordiamo che n puó essere maggiore o minore del numero di buffer. L'idea é la seguente:

- Viene associato ogni run ad uno dei buffer, caricando il primo blocco di B elementi di ogni run nel corrispondente buffer. Quest'ultimi vengono marcati come ATTIVO.
- Memorizzo il primo elemento di ogni run all'interno dell'heap.
- Dall'min heap prelevo il minimo e lo inserisco nel buffer in output.

  Dopo ogni estrazione dall'heap si cerca sempre di inserire in essa il prossimo elemento del run S_i a cui apparteneva quello appena estratto
- Ogni volta che il buffer di output si riempie copio il blocco sul disco e ne inizializzo uno nuovo. I blocchi che carico sul disco corrispondenti a k run formeranno un'altro run.
- Ogni volta che un buffer in input si riempie, viene copiato un nuovo blocco dalla corrispondente sottosequenza, se ció non é piú possibile il buffer viene marcato come INATTIVO.
- Quando alla fine rimane una sola sottosequenza attiva, viene copiato il suo contenuto su disco.

Analizziamo ora l'algoritmo del **k-way MERGE in EMM**

```
1  if (N ≤ M) then
2        Risolvi il problema in RAM con MergeSort tradizionale
3  else
4        Carica S_1^0,...,S_k^0 in RAM S_i^j il jesimo blocco del run S_i
5        Marca ogni S_i come attivo
6        Crea un MinHeap H in RAM
7        OUT ← buffer di output
8        for i ← 1 to k do
9              Inserisci in H la prima chiave m di S_i^0
10       while (ci sono almeno due buffer attivi) do
11             Estrai da H la chiave minima m
12             Inserisci m in OUT
13             Inserisci in H la prossima chiave di S_i^j
14             if (OUT contiene B elementi)
15                   Copia OUT nel disco
16                   Svuota OUT in RAM
17             if (ho raggiunto il termine di S_i^j) then
18                    Carica S_i^{j+1} in RAM
19                    H ← m ∈ S_i^{j+1}
20             else S_i^j é inattivo
21       Copia l'ultimo run attivo nel disco
```

Analizziamo il costo in termini di I/O, nella fase di merge, ogni blocco di ciascun run S_i e ogni buffer OUT viene rispettivamente letto da disco o scritto su disco una sola volta ad ogni passo quindi abbiamo 2 trasferimenti a $O(1)$, quindi la complessitá é $O(\frac{N}{B})$.

E' piú delicato valutare il costo in CPU:

Avendo implementato la coda di prioritá con un Min Heap dei primi k elementi paghiamo a ogni passo costo logaritmico $O(logk)$ per riordinarlo ogni volta che viene inserito un nuovo elemento. L'estrazione dallo heap costa $O(1)$. Dato che ognuno dei N elementi entrerà e usciràè dallo heap, in totale avremo un costo di $O(Nlogk)$.

A ogni esecuzione il numero di run che carico nella RAM si riduce di un fattore k, dunque viene eseguito $log_k \frac{N}{M}$ volte; di conseguenza il MERGE-SORT che fa uso del k-way MERGE ha un costo di:

$$O(\frac{N}{B} log_k \frac{N}{M}) = O(\frac{N}{B} log_{\frac{M}{B}}(\frac{N}{M} * \frac{M}{B})) = O(nlog_m n) I/Os$$

che sappiamo essere ottimo.

Chapter 16

Esercizio 16:Limite inferiore per la permutazione

Estendere l'argomentazione usata per il limite inferiore del problema dell'ordinamento in memoria esterna a quello della permutazione: dati N elementi $e_1, e_2, ..., e_N$ e un array π contenente una permutazione degli interi in $[1, 2, ..., N]$, disporre gli elementi secondo la permutazione in π.

Dopo tale operazione, la memoria esterna deve contenerli nell'ordine e $e_\pi[1], e_\pi[2], ..., e_\pi[N]$.

16.1 Svolgimento

Data una sequenza di elementi N, divisa in blocchi di dimensione B, vogliamo calcolare il numero di possibili permutazioni di N elementi tale che il numero di operazioni I/O sia minimo. Mostreremo che il problema presenta un lowerbound $\Omega(min\{N, \frac{N}{B} log_{\frac{M}{B}} \frac{N}{B}\})$.

L'idea é la seguente: consideriamo una sequenza di t operazioni di input, e contiamo quante permutazioni diverse possiamo ottenere. Trovando il minimo t che ci garantisca la possibilitá di ottenere tutte le permutazioni distinte avremo allora il lower-bound desiderato.

Consideriamo una generica operazione di input: il numero di modi possibili per scegliere un blocco nella memoria esterna é al piú N. Distinguiamo ora due casi:

- Se la pagina che ho scelto viene caricata in memoria per la prima volta (e questo avviene $\frac{N}{B}$ volte), posso permutare i suoi elementi in $B!$ modi, e disporli tra gli elementi giá in memoria in al piú $\binom{M}{B}$ modi.
- Se la pagina era giá stata caricata in memoria in precedenza, i suoi elementi sono giá permutati nell'ordine desiderato, dunque devo solo

scegliere gli $\binom{M}{B}$ modi in cui disporli rispetto agli elementi giá in memoria

Unendo i due casi, dopo una sequenza di t I/O posso produrre al piú

$$N^t \cdot (B!)^{\frac{N}{B}} \cdot \binom{M}{B}^t$$

permutazioni distinte; ovviamente vogliamo che questo numero risulti maggiore di $N!$.
Passando ai logaritmi, dev'essere

$$t * logN + \frac{N}{B} log(B!) + t * log\binom{M}{B} \geq log(N!)$$

da cui, usando le approssimazioni $log(x!) \backsim x * log(x)$ e $log\binom{n}{k} \backsim k * log(\frac{n}{k})$,

$$t * log(N) + N * log(B) + t * B * log(\frac{M}{B}) \geqslant N * log(N)$$

$$t(log(N) + B * log(\frac{M}{B})) \geqslant N(log(N) - log(B))$$

$$t \geqslant \frac{N * log(\frac{N}{B})}{log(N) + B * log(\frac{M}{B})}$$

Distinguiamo ancora due casi:

- Se $log(N) \leqslant B * log(\frac{M}{B})$ dev'essere

$$t \geqslant \frac{N * log\frac{N}{B}}{2 * B * log\frac{M}{B}} = \Omega(\frac{N}{B} log_{\frac{M}{B}}(\frac{N}{B}))$$

- Se invece $log(N) \geqslant B * log(\frac{M}{B})$ allora

$$t \geqslant \frac{N * log(\frac{N}{B})}{2 * log(N)} \geqslant \frac{N * log(N) - N * log(B)}{2 * log(N)} =$$

$$\frac{1}{2}(N - N * \frac{logB}{logN}) \geqslant \frac{1}{2}(N - \frac{1}{2} * N) = \Omega(N)$$

il che ovvio purché $\frac{log(B)}{log(N)} \leqslant \frac{1}{2}$, ossia $B \leqslant \sqrt{N}$ relazione vera in tutti i casi d'interesse.

Dall'unione dei due casi segue il lowerbound desiderato.

Chapter 17

Esercizio 17:Limite inferiore per la ricerca

Mostrare che la ricerca mediante confronti di una chiave in un insieme ordinato di N elementi richiede $\Omega(log_B N)$ I/O (trasferimenti di blocchi di taglia B nel modello EMM.)

17.1 Svolgimento

Sia k la chiave da ricercare, dati B elementi da una lista ordinata ci sono due possibilitá:

- k é uno dei B elementi.
- k potrebbe appartenere a una delle $B + 1$ sotto-sequenze indotte dai B elementi separatori.

Utilizzando un'argomentazione avversariale possiamo ignorare il primo caso; supponiamo di scegliere una chiave *difficile da trovare*, ovvero che sia sempre nella sotto-sequenza piú grande (finché ce ne sono).
La dimensione minima della sotto-sequenza piú grande si ha quando tutte le sequenze sono uguali, in tal caso lo spazio di ricerca si riduce quindi di un fattore $B + 1$.

passo 0 Dopo aver confrontato k con i B elementi la ricerca continua in una delle sotto-sequenze di dimensione

$$\Omega(\frac{N - B}{B + 1})$$

passo t Ad ogni passo k viene confrontata con B elementi, non potendo fare assunzioni sull'algoritmo questi potrebbero anche non essere tutti

nuovi. Quale che siano i valori la chiave da ricercare cadrá nella sotto-sequenza piú grande e quindi al generico passo t la sotto-sequenza avrá dimensione:

$$\Omega(\frac{N-B}{(B+1)^t}) = \Omega(\frac{N}{(B+1)^t} - \frac{B}{(B+1)^t})$$

Essendo $N >> B$ o anche perché asintoticamente $\frac{B}{(B+1)^t}$ tende a 0:

$$= \Omega(\frac{N}{(B+1)^t})$$

La ricerca finisce quando la dimensione della sotto-sequenza piú grande si riduce a 1

$$\frac{N}{(B+1)^t} = 1$$

$$\Rightarrow t = \Omega(\log_B N)$$

Ad ogni iterazione viene letto un blocco dalla memoria esterna, il numero di blocchi letti é quindi pari alle iterazioni. $\Omega(\log_B N)$.

Chapter 18

Esercizio 18:Esecuzione della permutazione nella memoria esterna

Dati due array A e π, dove A contiene N elementi (non importa quali) e π contiene una permutazione di $\{1, ..., N\}$, descrivere e analizzare nel modello EMM un algoritmo ottimo per costruire un terzo array C di N elementi tale che $C[\pi[i]] = A[i]$ per $1 \leq i \leq N$.

18.1 Svolgimento

Dall'analisi svolta sul Lower Bound della permutazione, sappiamo che il costo di I/O é dato da $min\{N, sort(n)\}$, dove $sort(n) = O(\frac{N}{B} * log_{\frac{M}{B}}(\frac{N}{B}))$

Algoritmo: Permutazione in memoria esterna

```
1 minimo=min{N, sort(n)}
2 if (minimo == N)
3       then itera sugli elementi e scrivi la permutazione
4       else esegui paradigma scnan& sort
```

Si può notare che l'algoritmo banale di iterare sugli elementi e scrivere la permutazione é eseguito solo in poche circostanze, ad esempio quando la dimensione B del blocco é molto piccola. Di seguito é mostrato soltanto l'algoritmo con il paradigma scan & sort :

1 Creare le coppie $(i, \pi[i])$, ovvero le coppie che mettono in relazione una posizione del vettore π con l'indice di permutazione.

   Tale passo esegue una lettura di$N$ elementi e una scrittura di N coppie. Da qui si deduce che il costo di I/O é $O(\frac{N}{B})$.

2 Ordina le coppie in base alla seconda componente (l'indice di permutazione).

   Il costo di questa operazione è uguale a quello di un ordinamento che, utilizzando ad esempio il k-way merge-sort é $O(sort(n))$

3 Sostituisci nelle coppie $\pi[i]$ con $C[\pi[i]]$.

   Poiché le coppie sono ordinate per $\pi[i]$, si accede al vettore C in modo sequenziale, quindi il costo di I/O é $O(\frac{N}{B})$

4 Ordina le coppie per la prima componente.

   Ancora l'ordinamento ha un costo di I/O pari a $O(sort(n))$

5 Scrivi nel vettore A le seconde componenti delle coppie.

   Il costo di I/O é $O(\frac{N}{B})$ in quanto sono letti e scritti N sequenzialmente.

In conclusione il costo di I/O dell'algoritmo appena esposto é $O(sort(n))$ ossia $sort(n) = O(\frac{N}{B} * log_{\frac{M}{B}}(\frac{N}{B}))$ é quindi ottimo.

Chapter 19

Esercizio 19:MAP REDUCE

Utilizzare il paradigma **scan and sort** (per esempio, mediante la MapReduce) per calcolare la distribuzione dei gradi in ingresso delle pagine Web, ipotizzando di avere giá tali pagine a disposizione.

19.1 Svolgimento

Vogliamo avere in uscita la distribuzione del grado in ingresso di un insieme di pagine Web cioé una lista del tipo: $< 1, 3609 >, < 2, 2898 >, < 3, 54 >, < 4, 56 >$ ..... che interpretiamo come 3609 pagine hanno grado in ingresso 1, 2898 pagine hanno grado di ingresso 2 ecc.

In input avremo un insieme di url: $< url1, url2, url3..... >$

Per prima cosa dobbiamo calcolare il grado in ingresso di ogni pagina. Per fare ció usiamo un passaggio di map e uno di reduce.

Alla funzione map passiamo in input la lista degli url delle pagine W, per ogni pagina Map effettuerá un parsing degli url contenuti e restituirá un numero di coppie $< url, 1 >$ pari al numero di link contenuti in tale pagina. La costante 1 serve a indicare che é presente un link con quel identificativo.

Funzione MAP: Creazione coppie dei link

```
1 function MAP(W)
2       for each url in W
3             links ← PARSER(url)
4             for each url in links do
5                   create [url, 1]
```

A questo punto l'insieme delle coppie $< url, 1 >$ viene poi ordinato secondo la chiave "url" in questo modo alla fine dell'ordinamento otteniamo una lista cosí fatta:
$< url1, < 1, 1, 1....., 1 >>, < url2, < 1, 1, 1....., 1 >>$ .....

A questo punto se volessimo avere il grado in ingresso di ogni pagina non dovremmo fare altro che dare in pasto a Reduce la lista ottenuta da Map e farli computare la somma degli "1" contenuti nella lista associata ad ogni url

Funzione REDUCE

```
1 function REDUCE (key,list) //key → url ,list →< 1,1,1,...,1 >
2         sum ← 0
3        for each i in list do
4               sum + +
5         create [key, sum]
```

Per ottenere la distribuzione dei gradi in ingresso é necessario modificare la reduce precedente cambiando l'ultima riga in :

$$create[sum, 1]$$

Questa modifica fará si che la lista ottenuta dopo l'ordinamento sará nella forma, $< 1, < 1, 1, 1, ...1, >>, < 2, < 1, 1, 1, 1....., 1 >>, < 3, < 1, 1, 1, 1....., 1 >>$ .....

Ora riapplicando una funzione reduce come la seguente otteremo il nostro risultato

Funzione REDUCE

```
1 function REDUCE (key,list)
2         sum ← 0
3        for each i in list do
4               sum + +
5         create [key, sum]
```

In questo caso la lista sará della forma:
$list \rightarrow < 1, 1, 1, ..., 1 >$ con tanti "1" quante pagine hanno tale grado.

In conclusione bisogna eseguire una Map per trovare i link presenti nelle pagine Web e due funzione Reduce, una per contare il numero di link che ha in ingresso una pagina e una per calcolare la distribuzione dei gradi in ingresso.

Chapter 20

Esercizio 20:Ricerca nel suffix array in memoria esterna

Ipotizzando di avere giá il testo T e il suo suffix array SA in memoria esterna, progettare e analizzare algoritmi efficienti per rispondere alle seguenti query di una stringa pattern P che viene fornita ogni volta on-line da un utente (dove P puó variare ad ogni query mentre T rimane lo stesso e possiamo quindi sfruttare SA):

1. verifica se P occorre in T (un booleano)
2. conta il numero di occorrenze di P in T (un intero non negativo)
3. elenca tutte le posizioni di T in cui P occorre (una lista di interi non negativi)

20.1 Svolgimento

20.1.1 find booleana

Ricerca binaria adattata per trovare un pattern P all'interno di un testo T, avendo testo e suffix array SA in memoria esterna.
Sketch dell'algoritmo:

- carica dal SA il blocco centrale dell'intervallo di ricerca
- confronta P col testo puntato dagli elementi agli estremi del blocco centrale
  - se $P < left$: itera sulla metà sinistra dell'intervallo
  - se $P > right$: itera sulla metà destra
  - se $left < P < right$: esegui un ricerca binaria sugli elementi puntati dal blocco

```
find(key){
      intervallo = [0, N]
      while(intervallo.length > 1){
            pivot_block = load_block_SA(intervallo.center)

            cmp_left = compare_with_EM(key, pivot_block[0])

            if(cmp_left == 0)
                  return true;
            else if(cmp_left < 0)
                  intervallo = [intervallo.left, intervallo.center-1]
            else if(cmp_left > 0){
                  cmp_right = compare_with_EM(key, pivot_block[B-1])
                  if(cmp_right == 0)
                        return true;
                  else if(cmp_right > 0)
                        intervallo = [intervallo.center+1,
intervallo.right]
                  else if(cmp_right < 0)
                        return find_single_block(pivot_block, key)
            }
      }
      if(intervallo.length == 1){
            return find_single_block(load_block_SA(intervallo.center),
key)
      }
      return false
}
```

Dove:

- *compare_with_EM* confronta la chiave con il testo in memoria esterna a partire dal puntatore che gli viene passato.
- *find_single_block* esegue una ricerca binaria sugli elementi puntati da un singolo blocco, analoga a quella fatta sui blocchi

Ad ogni iterazione la dimensione dell'intervallo viene dimezzata, quindi dopo $O(\log_2 \frac{N}{B})$ iterazioni la ricerca termina. Ogni iterazione richiede le seguenti operazioni di IOs:

- carica blocco centrale del SA $O(1)$
- confronta la chiave con il testo puntato dal primo elemento del blocco $O(\frac{|P|}{B})$

- eventualmente confronta la chiave anche con il testo puntato dall'ultimo elemento $O(\frac{|P|}{B})$

Inoltre se si arriva ad avere un intervallo di dimensione uno c'è il costo della ricerca binaria all'interno del singolo blocco $O(\log_2(B) * \frac{|P|}{B})$.
In totale:

$$O(\log_2 \frac{N}{B} * \frac{|P|}{B} + \log_2(B) * \frac{|P|}{B})$$

Il secondo termine però sparisce in quanto $\log_2(B) << \log_2 \frac{N}{B}$

$$O(\log_2 \frac{N}{B} * \frac{|P|}{B})$$

20.1.2 2 count

per contare le occorrenze di P è sufficiente che una volta trovata la prima si proceda linearmente in entrambe le direzione con un costo di $O(\frac{|P|}{B} * \#occorrenze)$, in totale:

$$O(\frac{|P|}{B}(\log_2 \frac{N}{B} + \#occorrenze))$$

20.1.3 3 find all

se oltre a contare si vogliono trovare tutte le posizioni dove occorre il pattern è sufficiente tenerne traccia durante la ricerca in una lista che verrà poi restituita senza costi di IOs aggiuntivi.

20.1.4 Implicazione problemi

$3 \Rightarrow 2$, data la soluzione di 3 si ricava la soluzione di 2 contando gli elementi della lista.
$2 \Rightarrow 1$, sia n la soluzione di 2, 1 ha soluzione positiva se e solo se $n > 0$.

Chapter 21

Esercizio 21: Costruzione del suffix array in memoria esterna

21.1 Svolgimento giusto

Data una stringa $T[0,n) = t_0, t_1, ...t_{n-1}$; denotiamo con S_i il suffisso di $T[i,n) = t_i, t_{i+1}, ..., t_{n-1}$ e con $S_c = \{S_i\}$.

L'obbiettivo del Dc3 é quello di ordinare $S_[0,n]$ di T:

$$S_{SA[0]} < S_{SA[1]} < .... < S_{SA[n]}$$

Carattere	b	a	n	a	n	a	$
$t[i]$	0	1	2	3	4	5	6
Sample	0	1	2	0	1	2	0

21.1.1 Passo 0: Costruire i sample

Per prima cosa dobbiamo costruire i sample per $k = 0, 1, 2$ definiamo:

$$B_k = \{i \in [0,n] | imod3 = k\}$$

Quindi nel nostro esempio avremo:

$$B_0 = (0, 3, 6)$$
$$B_1 = (1, 4)$$
$$B_2 = (2, 5)$$

Definiamo $C = B_1 \odot B_2$ = $(1, 4, 2, 5)$ quindi S_C é l'insieme dei suffissi sample.

21.1.2 Passo 1:Ordinare i Sample

Per $k = 1, 2$ costruisco $R_k = [t_k, t_{k+1}, t_{k+2}], ..., [t_{maxB_k}, t_{maxB_{k+1}}, t_{maxBk+2}]$ i cui caratteri sono triple.

$$R = R1 \odot R2$$

$$R = ([ana], [na\$]) \odot ([nan], [a\$\$])$$

R	[ana]	[na$]	[nan]	[a$$]
S_i	[123]	[456]	[234]	[567]

Ordinando i suffissi di R otteniamo l'ordine dei sample suffix di S_C. Per l'ordinamento userò il RADIX SORT dei caratteri di R e li rinomino con il loro rank nell'ordinamento, otterró R'.

SE i caratteri di R' sono tutti diversi, l'ordine dei caratteri da direttamente l'ordine dei suffissi, **ALTRIMENTI** è necessario riapplicare ricorsivamente l'algoritmo DC3 su R'.
Le regole per assegnare il rank sono le seguenti:

- $rank(S_i)$ nell'ordinamento se $i \in C$
- $rank(S_i) = \perp$ se $i \in B_0$
- $rank(S_{n+1}) = rank(S_{n+2}) = 0$

R	[ana]	[na$]	[nan]	[a$$]
S_i	[123]	[456]	[234]	[567]
R'	[a$$]	[ana]	[na$]	[nan]
rank	1	2	3	4

Di conseguenza i rank sono:

Carattere	b	a	n	a	n	a	$	$	$
$t[i]$	0	1	2	3	4	5	6	7	8
Rank	$\perp$	2	4	$\perp$	3	1	$\perp$	0	0

A questo punto ottengo il sample suffix ordinato $SA'_R = [5, 1, 4, 2]$:
Il rank 1 corrisponde ad "a" il cui indice é 5
Il rank 2 corrisponde ad "a" il cui indice é 1
Il rank 3 corrisponde ad "n" il cui indice é 4
Il rank 4 corrisponde ad "n" il cui indice é 2

21.1.3 Passo 2: Ordinare i non sample

$$\forall S_i \in S_{B_0}$$
$$(ti, rank(S_{i+1})$$

Ricordiamo che $B_0 = (0, 3, 6)$ quindi:

- $i = 0$, ($t[0] = b \mid rank(S_1) = 2$) = $(b, 2)$
- $i = 3$, ($t[3] = a \mid rank(S_4) = 3$) = $(a, 3)$
- $i = 6$, ($t[6] = \$ \mid rank(S_7) = 0$) = $(\$, 0)$

$$\forall i, j \in B_0$$

$$S_i \leq S_j \Leftrightarrow (t_i, rank(S_{i+1})) \leq (tj, rank(S_{j+1}))$$

$$(\$, 0) < (a, 3) < (b, 2)$$

Quindi $SA_{B_0} = [6, 3, 0]$

21.1.4 Passo 3:Merge

una volta ottenuti SA'_R e SA_{B_0}, insiemi ordinati di suffissi li andiamo a fondere usando lo standard dei confronti.

Dato $S_i \in S_C$, e $S_j \in S_{B_0}$ allora:

$$\Rightarrow i \in B_1 : S_i \leq S_j \Leftrightarrow (t_i, rank(S_{i+1})) \leq (t_j, rank(S_{j+1}))$$

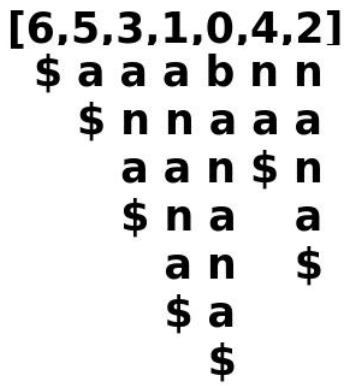
$$\Rightarrow i \in B_2 : S_i \leq S_j \Leftrightarrow (t_i, t_{i+1}, rank(S_{i+2})) \leq (t_j, t_{j+1}, rank(S_{j+2}))$$

Nel nostro esempio con $SA_{B_0} = [6, 3, 0]$ e $SA'_R = [5, 1, 4, 2]$:

- $i = 5 \rightarrow \in B_2$, $j = 6 \Rightarrow (a, \$, 0) \leq (\$, \$, 0) \Rightarrow 6$
- $i = 5 \rightarrow \in B_2$, $j = 3 \Rightarrow (a, \$, 0) \leq (a, n, 1) \Rightarrow 5$
- $i = 1 \rightarrow \in B_1$, $j = 3 \Rightarrow (a, 4) \leq (a, 3) \Rightarrow 3$

- $i = 1 \rightarrow \in B_1$, $j = 0 \Rightarrow (a, 4) \leq (b, 2) \Rightarrow 1$
- $i = 4 \rightarrow \in B_2$, $j = 0 \Rightarrow (n, a, \perp) \leq (b, a, 4) \Rightarrow 0$
- $\Rightarrow 4$
- $\Rightarrow 2$

Il SA finale é :



21.2 Analisi

la complessitá dell'algortimo DC3 in EMM é data da $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{M})$ in quanto la ricorsione viene fatta su una stringa di lunghezza $2/3n$ divisa in B blocchi, ossia $[\frac{2*N}{3*B}]$, inoltre il MergeSort ha costo $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{M})$.
Quindi abbiamo:

$$T[N] = T[\frac{2 * N}{3 * B}] + O(MergeSort) = O(MergeSort)$$

Chapter 22

Esercizio 22:avigazione implicita in vEB

Dato un albero completo memorizzato secondo il layout cache-oblivious (CO) di van Emde Boas (vEB) in modo implicito, ossia senza l'ausilio di puntatori (come succede nel classico heap binario implicito), trovare la regola per navigare in tale albero durante la ricerca di una chiave senza usare puntatori espliciti.

22.1 Alternativa1

Per consentire la navigazione nel layout implicito é necessaro costruire, durante la fase di archiviazione dell'albero, una tabella di dimensione $O(logn)$, in cui per ogni profonditá d, memorizziamo:

- $B[d]$: la dimensione di ogni bottom tree avente radice a profonditá d
- $T[d]$: la dimensione del corrispondente top tree;
- $D[d]$: la profonditá della radice di tale top tree

Inoltre quando effettuiamo una ricerca (a partire dalla radice) di un nodo v di profonditá d, teniamo traccia di:

- i: la posizione che v occuperebbe in una memorizzazione BFS;
- $Pos[j]$: la posizione nella memorizzazione vEB del particolare nodo z a profonditá j che ho incontrato durante la ricerca.

Consideriamo la rappresentazione binaria di i : letti da sinistra verso destra, i suoi bit rappresentano le svolte effettuate nella ricerca di v nella rappresentazione BFS (secondo la regola: 0 svolta a sinistra nella BFS, 1

svolta a destra (il bit piú a sinistra ovviamente vale sempre 1, e indica il fatto che si parte con la radice)). Consideriamo una suddivisione in cui il nodo v cercato é radice di un bottom tree.

Allora se il corrispondente top tree é alto k , cioé $T[d] = 2^k - 1$, i k bit piú a destra di i rappresentano l'indice del bottom tree cercato tra tutti quelli della suddivisione in questione. Poiché tutti i bit di $T[d]$ (in totale k) valgono 1, posso calcolare questo indice come i & $T[d]$.

Vale allora:

$$Pos[d] = Pos(D[d]) + T[d] + (i\&T[d])B[d]$$

ossia:

Parto da $Pos(D[d])$, posizione della radice del top tree; dopodiché avanzo di:

- Tanti nodi quanti ve ne sono nel top tree, cioé $T[d]$;
- Tanti nodi quandi ve ne sono in ogni bottom tree, cioé $B[d]$, e faccio questo tante volte quanti sono i bottom tree che precedono quello di radice v ossia, poiché gli indici partono da 0, $i\&T[d]$.

Usando ripetutamente questa formula é dunque possibile effettuare la ricerca di un qualsiasi nodo v.

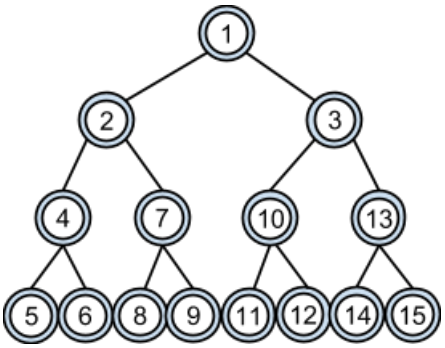
22.2 Alternativa 2

Usiamo il seguente pseudo codice per navigare il vEB senza l'uso di puntatori:

```
1 access(h, u, sd)
2       if (h = 2)
3               return (u + sd + 1, sd)
4       else
5               (f1, m) = access(h/2, u, sd)
6               u' = u + (1 + 2 * m + sd')(2^(h/2) - 1)
7               (f2, m') = access(h/2, u'sd'')
8               return (f2, (2m + sd') (2^(h/2 - 1)) + m')
```

nella quale:

- u é la radice
- h é l'altezza dell'albero
- sd é una procedura booleana che dice se andare al figlio sinistro o al figlio destro con valori O o 1



basic-miktex-2.6-beta-3.exe Consideriamo il seguente esempio: I valori dentro ai nodi corrispondono agli indici di memorizzazione. Supponiamo che le funzioni sd e sd' ritornino valori t.c. il nodo di destinazione sia il numero 12. La procedura, al di la del caso base, ha 2 chiamate ricorsive che intuitivamente possiamo capire agiranno rispettivamente una nella parte alta dell'albero e una nella parte bassa. L'assegnamento di u' tra le 2 funzioni ricorsive ha lo scopo appunto di spostarci dalla parte alta alla parte bassa per poter richiamare poi la funzione ricorsiva.

Vediamo il comportamento dell'algoritmo:

- access($4, 1, sd$)
- $h \neq 2$ quindi si passa al ramo **else**
- chiamata ricorsiva access($2, 1, sd'$)
- $h = 2$ return $(10 + 1 + 1, 1)$ cioé sd' dice di andare a destra $(= 1)$
- ritorno chiamata (f_2, m')=$(12, 1)$
- return$(12, 2 * 1 * (2^1) + 1)$=$(12, 5)$ dove 12 rappresenta il nodo finale/-parziale (nel caso in cui l'albero sia piú grande) mentre 5 indica i nodi che sono alla sinistra del solito livello dei nodi ritornati

Chapter 23

Esercizio 23: LCP

Estendere la costruzione del suffx array SA basata sul mergesort e sulla tecnica DC3 vista a lezione, in modo da calcolare anche l'array LCP tale che $LCP[i]$ contiene il prefisso comune piú lungo tra i due suffssi indicati da $SA[i]$ e $SA[i+1]$, per $1 \leq i \leq N$. La complessitá asintotica deve rimanere la stessa del mergesort. Nota: é sufficiente la complessitá nel modello RAM (ma va bene anche nell'EMM, per chi vuole).

23.1 Svolgimento

LCP puó essere calcolato in due momenti diversi:

1 Durante il RADIX SORT di R creiamo l'LCP per R'

2 Durante la fase di merge servendosi dell'LCP precedentemente calcolato creiamo l'LCP definitivo per il SA.

Durante il radix sort mentre faccio i confronti conto anche il numero di occorrenze uguali che ho ossia, dato $R = ([ana], [na\$], [nan], [a\$\$])$ il radix sort confronta la prima occorrenza di ogni carattere di R e le ordina, confronta le seconde occorrenze, le ordina, e cosi via... Ad ogni passo del radixsort mi memorizzo in array temporanei il numero di occorrenze uguali che incontro dopo l'ordinamento su una colonna di occorrenze:

Abbiamo cosí ottenuto l'LCP per $R' = [5, 1, 4, 2]$ ossia $[1, 0, 2]$:

- $(5, 1) \Rightarrow 1$
- $(1, 4) \rightarrow 0$
- $(4, 2) \Rightarrow 2$

Non ci resta che definirci l'LCP finale come:

```
1 LCP(i, j)
2       if (i, j) è gia definito da LCP'_R
3               return (i, j).val
4        else if  T[i] = T[j]
5               return 1+ LCP(i + 1, j + 1)
6        else
7               return 0
```

Le coppie (i, j) in considerazione sono esattamente nell'ordine in cui le analizzo durante la fase di merge, nella quale $i \in SA'_R$ e $j \in SA_{B_0}$.

- $(5, 6) \longrightarrow (a, \$) \rightarrow 0$
- $(5, 3) \longrightarrow (a, a) \rightarrow 1$
- $(1, 3) \longrightarrow (a, a) \rightarrow (2, 4) \rightarrow 1 + 2 = 3$
- $(1, 0) \longrightarrow (a, b) \rightarrow 0$
- $(4, 0) \longrightarrow (n, b) \rightarrow 0$
- $(4, 2) \longrightarrow (n, n) \rightarrow (5, 3) \rightarrow 1 + 1 = 2$

LCP finale é $[0, 1, 3, 0, 0, 2]$

Chapter 24

Esercizio 24

24.1 Svolgimento Michele

Premesse

Sia m il numero di archi nell'albero, n il numero di nodi e c il numero di caratteri totali che etichettano gli archi. Un algoritmo è lineare nella dimensione dell'albero se ha costo $O(n + m + c)$.
I nodi dell'albero sono strutture dati di due tipi, *intermedi* se fanno solo da snodo all'interno dell'albero e *foglia* se sono per l'appunto foglie e contengono il puntatore al testo; ogni nodo inoltre contiene: un puntatore al padre e un contatore che indica la lunghezza del prefisso dalla radice fino a quel nodo.

Algoritmo

Tenendo traccia del nodo radice $Root$ e dell'ultimo aggiunto $Last$, scorro linearmente SA e per ogni elemento s con indice s_i:

- Se il suffisso aggiunto precedentemente non ha caratteri in comune con questo, ovvero se $LCP[s_i - 1] = 0$, aggiungo un nodo foglia alla radice, sul cui arco c'è tutto il prefisso puntato da s e al cui interno c'è proprio s.
- Nel caso in cui $LCP[s_i - 1] \neq 0$ si deve cercare l'antenato comune risalendo la catena di puntatori ai padri fintanto che $parent.length > LCP[s_i - 1]$.
  - Nel caso che il nodo trovato sia proprio il padre, ovvero $parent.length = LCP[s_i - 1]$, basta aggiungere un nodo foglia al padre, con testo a partire da $s + LCS[s_i - 1]$. Se il padre è a sua volta un nodo foglia[1] va prima trasformato in un nodo intermedio generando

[1] Accade solo se il l'ultimo suffisso aggiunto è anche suffisso di s

un figlio sul carattere \$ e spostando il puntatore.

- Nel caso invece che l'antenato sia *troppo antenato* occorre inserire un nodo intermedio che spezza l'etichetta dell'arco in due parti. Sia $parent$ l'antenato trovato con la visita e f il figlio da cui lo abbiamo raggiunto (ne teniamo traccia durante la visita).
  Creato il nodo intermedio $padre$, ridirezioniamo l'arco $(parent, f)$ verso $padre$ terminando l'etichetta dopo $LCS[s_i{-}1]{-}parent.length$ caratteri. La parte restante dell'etichetta la copiamo in un nuovo arco $(padre, f)$.
  Fatto tutto questo possiamo creare la nuova foglia figlia di $padre$, sul cui arco c'è il suffisso che parte da $s + LCS[s_i - 1]$.

Complessità

Tutto il ciclo viene eseguito per $|SA|$ volte, essendoci un nodo foglia per ogni suffisso questo è maggiorabile da $O(n)$.
Ogni volta che si aggiunge un nodo, oltre a varie operazioni di costo costante, bisogna scrivere l'etichetta dell'arco che lo attacca al padre. Nessun carattere viene mai cancellato durante l'algoritmo e nessun nodo viene rimosso o aggiunto più volte: quindi gli inserimenti apportano un costo $O(c)$.
Durante tutto l'algoritmo non si scende mai verso i figli già generati, quindi la fase di risalita per cercare un antenato non attraverserà mai due volte lo stesso arco: $O(m)$.
Quando si spezza un arco la parte inferiore del prefisso viene copiata in un nuovo arco, non ridiscendendo mai verso i figli ogni carattere viene copiato al più una volta è quindi influisce sul costo complessivo di $O(c)$.
Le restanti operazioni all'interno del ciclo su SA sono tutte costanti, quindi il costo complessivo è

$$O(n + 2c + m) = O(n + c + m)$$

Che è lineare.