**BIPARTITE MAX MATCHING (in PO)**

Input: **G(V,E)** bipartito

Output: M sottoinsieme E s.t. arco sia unico

Costo: |M|

Tipo: Max

*Cammino aumentante (CA) per M:*

Esistono due tipi di lati, quelli occupati (incide un lato di M) e quelli non occupati.

Un cammino aumentante parte e arriva su un vertice non occupato, alternando lati occupati e non occupati.

*Teorema: Se esiste CA per M, M non è massimo*

**Dimostrazione:**

Individuato un CA esisteranno lati in M e lati non in M, per la definizione di CA quelli non in M sono > di quelli in M. Quindi fare lo switch.

*Teorema: Se M non è massimo, esiste CA*

**Dimostrazione:**

|M’| > |M|, <delta> = M intersezione M’ = (M \ M’) U (M’ \ M)

* Su ogni vertice incidono al massimo due lati di X se no M’ o M non sarebbero matching.
* I cicli sono di lunghezza pari, e hanno i lati alternati in (M \ M’) e (M’ \ M), la stessa quantità viene rimossa da M’ e da M.
* Quindi a questo punto, in M’ ci deve essere un CA, quindi deve avere più lati di M. E si puo’ notare come questo inizia e finisce in vertici non occupati non appartenenti a M.

**LOAD BALANCING (NPO-comp, APX)**

Input: **t\_o, …, t\_n** tasks, **m** macchine

Output: insieme degli assegnamenti dei tasks alle macchine

Ammissibili: scelte di assegnamenti di tutti i task alle macchine in modo coerente

Costo: Ogni macchina alla fine dell’esecuzione ha un carico L\_i, L = max(L\_i)

Tipo: Min

*Teorema: GreedyBalance è 2-APX per load balancing*

Lemma 1:

Sia L\* il costo della soluzione ottima, il suo costo è >= alla somma dei task / il numero di macchine. (distribuzione perfetta)

Lemma 2:

L\* >= del max processo

**Dimostrazione:**

Sia î la macchina con carico max

* prima dell’ultimo task era: L’î = Lî - tj; notiamo che L’î è la più scarica tra tutte le macchine, perchè l’algoritmo è greedy.
* dopodichè consideriamo tutti gli i ottenendo m disequazioni, ognuna da 1…m(Lî - tj) <= somma di tutti task
* Dividendo tutto per m, otteniamo Lî - tj = (somma di tutti i task)/m <= L\* per Lemma 1
* Ovviamente, L = Lî = (Lî - tj) + tj→ L\* >= tj per Lemma 2
* L <= L\* + tj <= L\* +L\*<=2L\*

*Teorema: ∀ε > 0 esiste un input su cui Greedy Balance produce una soluzione L t.c. 2−ε ≤ L/L\* ≤ 2*

**Dimostrazione:**

* n. di macchine *m*  > 1/ε; n. di task = *m(m-1) +1;*
* lunghezza task = 1 tranne ultimo = m;
* Se assegno per ultima quella di lunghezza m:

L = = 2m -1

* Se assegno prima m:

L = m

Quindi L = 2m-1. L/L\* = (2m-1)/m = 2 - 1/m >= 2 - ε perchè m>1/ε.

*Teorema: Sorted Greedy Balance fornisce una 3/2-approssimazione per Load Balancing*

**Dimostrazione:**

Assumiamo che numero di task (che partono da 0)> numero macchine, in caso contrario abbiamo sempre L\*.

* L\*>= 2t\_m, sapendo che l’algoritmo assegna sempre alla macchina più scarica, dopo m+1 passi 2 task saranno assegnati alla stessa macchina ed essendo in ordine decrescente vale quanto affermato.
* considerando l’ultimo task da assegnare alla macchina, sappiamo che sarà tj <= 2t\_m <= L\* → tj <= t\_m <=½ L\*
* dimostrazione precedente: L = Lî = Lî - tj + tj

L\* + ½ L\* = 3/2 L\*

**CENTER SELECTION (NPO-comp, APX)**

Input: **S** punti appartenenti a uno spazio metrico, **K** centri da selezionare

Output: C sottoinsieme di S

Ammissibili: |C| <= K

Costo: ρ(C) = max d(s, C(s))

Tipo: Min

*Teorema: CenterSelection plus, sia r una stima del raggio di copertura e ρ la distanza massima tra un punto e il suo centro alla fine dell’esecuzione*

* *se l'algoritmo emette un C, l'approssimazione è ≤ 2r/ρ\**
* *se r ≥ ρ∗, l'algoritmo emette un output*
* *se il risultato è impossibile, allora r < ρ\**

**Dimostrazione:**

* Ogni punto in S sia <= 2r perchè quelli maggiori li ho rimossi, per cui p(C) <= 2r → ρ(C)/ρ\* <= 2r/ρ\* (rapporto di approssimazione)
* Consideriamo s1 ed s2 due punti che si riferiscono al centro C\*(s): utilizziamo la disuguaglianza triangolare e constatiamo che la distanza tra i due sia <= 2ρ\* <= 2r.

se r >= ρ\*, l’algoritmo emette una soluzione con approssimazione 2r/ρ\*

se r <= ρ\*/2, l’algoritmo non produce soluzioni

se ρ\*/2 <= r <= ρ\*, puo’ produrre entrambe le soluzioni

*Teorema: Greedy Center Selection è 2-appr. per Center Selection*

**Dimostrazione**:

Per assurdo, supponiamo che esista un output ρ(C) > 2ρ\*, ossia esiste s (un punto) la cui distanza dal suo centro è >= 2ρ\*; all’i-esima iterazione sia c\_i il centro che viene aggiunto, la sua distanza dai centri è >= di qualsiasi altro punto, e ovviamente anche di s e quindi di 2ρ\*.

Questo coincide con le prime k iterazioni di CenterSelectionPlus con r=2ρ\*, la d(s, C) è > 2ρ\* e quindi il punto s non è stato ancora rimosso, quindi deve ancora fare almeno un’iterazione e quindi emetterà “impossibile”, ma questo è assurdo perchè con r >= 2ρ\* l’algoritmo deve terminare.

*Teorema: Non esiste un algoritmo α-approssimante per Center Selection con α<2*

**Dimostrazione:**

* Riduco polinomialmente il problema a Dominating Set (NP completo): esso considera dei nodi in un grafo, in cui ciascuno di essi o fa parte di un insieme D o è vicino di un nodo che ne fa parte.

Input: **G** = (V,E), **K** in N

Output: {0,1} = 2

Problema: Esiste un D sottoinsieme di V s.t. la sua cardinalità è <= K.

Un dominating set è un insieme di vertici tale s.t. dolp esiste un y vicino di x.

* Traduciamo l’input per CenterSelection, ossia che i vertici di DS costituiscono i punti di CenterSelection:

d(x,y) = 0 se x = y

d(x,y) = 1 se x != y e (x,y) in E;

2 altrimentiyt

viene rispettata la disuguaglianza triangolare

* supponiamo che esista un algoritmo che α approssima CS con α<2:
  + 1 <= ρ / ρ\* <= 2 - e

Dato che ρ puo’ assumere solo due valori, o 1 o 2

* + p\*(s,K) = 1, allora 1<=p\*(s,K) <=2-e: esiste un DS
  + p\*(s,K) = 2, allora 2<=p\*(s,K) <=4-2e: non esiste un DS

Non posso calcolare in tempo polinomiale (se P diverso da NP) in un problema NPO completo

**Tecniche di pricing**

**MINIMUM SET COVER (NPO-comp, f**-**APX)**

Input: **S1,**...**,Sm** in **U** s.t. l’unione dei sets è = **U**, per ogni i = 1, …, m un peso w\_i nei Reali

Output: Insieme di insiemi scelti (o indici), C sottoinsieme di {1,...,m}

Ammissibili: C s.t. l’unione copri l’universo

Costo: w = Somma dei pesi degli insiemi scelti

Tipo: Min

Funzione armonica:

Sommatoria con i che va da 1 a n di 1/i

Proprietà: ln(n+1) <= H(n) <= 1+ln(n)

Osservazione 1:

Il costo della soluzione equivale a

w = Sommatoria di s in U di c\_s // somma dei costi degli insiemi scelti

**Dimostrazione:**

Quando aggiungo un elemento nell’insieme di quelli scelti il suo costo è w\_i / |S\_i instersecato R| (rapporto tra prezzo è copertura dell’universo)

Osservazione 2:

Per ogni k, il costo degli elementi in sk, è uguale

Somm s∈Sk Cs ≤ H(|Sk|) · Wk

**Dimostrazione:**

* Sia Sk un insieme di subset tra quelli da scegliere con gli elementi indicizzati in ordine di copertura (elementi: da 1 a d);
* Consideriamo l’istante in cui si copre l’elemento sj tramite un altro insieme, Sh(questo è l’insieme al momento j-esimo, alla fine diventerà Sk → Sh <= Sk) → visto che gli elementi di Sk sono in ordine di copertura (elementi rimanenti: da j a d);
* Il costo dell’elemento j equivale a il rapporto tra
  + Wk’/|Sh intersecato R| <= Wk / |Sk intersecato R| <= Wk/ d -j +1 (denominatore più piccolo, rapporto più grande);
  + Consideriamo ora tutti gli elementi di Sk (elementi con j che vanno da 1 a d): notiamo che “sviluppando” la sommatoria, possiamo raccogliere e ottenere Wk(1+½…+1/d) = WkH(d) = WkH(|Sk|)

*Teorema: Price Set Cover fornisce una H(M)-approssimazione per Set Cover, dove M = max\_i |Si| (massimo numero di elementi di un insieme)*

**Dimostrazione**:

* Consideriamo il peso della soluzione ottima:

w\* = Somm (Si appart. a C\*) wi

* per l'osservazione 2, sappiamo che w\_i >= Somm dei costi / H(|S\_i|) >= Somm dei costi / H(|M|)

Consideriamo il valore ottenuto sommando, per ogni insieme della copertura ottima C\*, la somma dei costi dei singoli elementi in ogni insieme. Questo valore è maggiore o uguale alla somma dei costi degli elementi nell’U, poichè nel primo valore potremmo sommare più volte il costo di alcuni elementi che compaiono in più insiemi scelti rispetto all’universo.

Per l’osservazione 1 la seconda sommatoria è uguale al costo della soluzione ottima.

Applicando le definizioni

w\* = Somm w\_i >= Somm in S\* Somm in S\_i cu / H(M) <= w / H(M)

w/w\* <= H(M)

Inoltre possiamo osservare che M <= H(n) = O(ln(n))

**Corollario:** Greedy Set Cover è O(ln(n))-apx

Osservazione 3:

L’analisi è tight, non esiste un algoritmo migliore.

**Dimostrazione:**

Consideriamo un input S:

1. Due insiemi grandi n/2 che uniti coprono tutti gli elementi di U, di costo 1-*e*
2. Un insieme che copre n/4 elementi degli insiemi 1 e 2 al punto 1, di costo 1
3. Un insieme che copre n/8 elementi degli insiemi 1 e 2 al punto 1, di costo 1

* …

Al primo passo viene preferito l'insieme al punto 2, visto che il suo costo è 1 / (n/2) = 2/n, mentre quelli al punto 1, 1+*e*/n/2 = (2+2*e)*/n

…

Il costo che si ottiene è w = log(n). La soluzione ottima sarebbe quella di prendere al primo passo i primi due insieme di costo 2+2e.

**Vertex Cover (NPO-comp, f**-**APX)**

Input: G = (V,E) non orientato con vertici pesati, w0, ...,wm

Output: Insieme vertici scelti

Ammissibili: X sottoinsieme di V s.t. ogni arco ha una delle due estremità in un vertice della soluzione

Costo: w = Somma dei pesi dei vertici scelti

Tipo: Min

*Teorema: Vale che SetCover è riducibile polinomialmente a VertexCover.*

Dato (G = (V,E), wi, w\_max) input per VertexCover, si puo’ trasformare in una istanza di SetCover (S={S1,...,Sn}, wi, w\_max) dove ogni Si per ogni vertice i è uguale ai lati che incidono su di esso.

Osservazione 3: VertexCover è H(D)-apx

**PriceVertexCover**

L’idea si basa sul fatto che gli archi sono intenzionati a comprare uno dei due estremi a un certo prezzo. Un nodo si vende, se la somma delle offerte degli archi incidenti arriva al suo Wi (prezzo).

Ovvero, se le offerte Pe degli archi che incidono sul vertice i non superano il suo costo, ovviamente devono raggiungerlo per comprare il vertice.

Lemma 1: Se Pe è equo, allora Somm e in E Pe ≤w∗

**Dimostrazione:**

* + La definizione di equità implica che la somma dei prezzi dei lati incidenti su quel vertice sia <= al Wi di quel vertice
  + Estendiamo per tutti i vertici appartenenti a S\*; arriviamo a dire che <= W\*

Lemma 2: Pe è tight, aw Somm e in E Pe = wi

Lemma 3: Price Vertex Cover crea un peso: W ≤ 2 Somm. e∈E Pe

**Dimostrazione:**

* viene considerato il doppio in quanto i lati possono essere considerati una o due volte.

*Teorema: Price Vertex Cover è 2-approssimante per Set Cover*

**Dimostrazione:**

* Sostituendo w/w\* con gli utili due lemmi dimostrati qui sopra, <= 2.

**Disjoint Paths**

Input: G = (V,E) orientato, s1,...,sk sorgenti, t1,...,tk destinazioni

Output: Insieme di archi (coppie) s.t. non sono utilizzate più di C volte

Costo: |I|

Tipo: Max

L’idea di GreedyDisjointPaths è quella di allungare i lati per scoraggiarne l’utilizzo.

β > 1 è il fattore di allungamento.

Considerando l’evolversi dei cammini dell’algoritmo, un cammino puo’ essere:

* Inutile, se unisce una sorgente e una destinazione già collegate
* Utile ma ostruito, ovvero collega una sorgente e una destinazione sconnesse, ma passa per una arco utilizzato da più di C cammini

Lemma 1: Un cammino A è corto rispetto a una certa lunghezza l sse l(A) < β^C, lungo altrimenti.

**Dimostrazione:** ovvio in quanto io moltiplico al max β^C volte

Lemma 2: Se i in I∗ \ I, allora l(Π∗i ) ≥ βc, ovvero, se una certa coppia i appartiene alla soluzione ottima ma non alla nostra, vuol dire che questo cammino è maggiore o uguale a β^c.

**Dimostrazione**: Se valesse l(Π∗i ) < β^c, il cammino Π∗i sarebbe corto e utile, in quanto collega una coppia ancora non collegata, dato che non appartiene a I. Quindi, non esisterebbe motivo per non selezionarlo.

Lemma 2: Vale che Somm.e∈E l(e) ≤ β^(c+1) |I| + m

m = #archi

|I| = iterazioni dell’algoritmo

**Dimostrazione:** Il valore iniziale della sommatoria è uguale a m (#archi); dopo aver selezionato un cammino, la funzione di lunghezza assumerà due valori, ossia invariata o moltiplicata per il fattore β se l’arco è stato considerato in quella iterazione; considerando la differenza delle lunghezze, (quando la funzione la funzione è invariata la somma è uguale a 0)

la sommatoria delle lunghezze sul cammino è minore o uguale di β^(c+1) per il lemma precedente moltiplicato per il numero di iterazioni fatte, più il valore iniziale m.

*Teorema: Greedy Disjoint Paths con β = m^(1/c+1) fornisce una (2cm^(1/c+1) +1)-approssimazione.*

**Dimostrazione**: Mettendo insieme i lemmi precedenti e ponendo beta = m ^ (1/c+1), notiamo che il tasso di approssimazione |I\*|/|I| <= cβ +cβ^-c + 1 al variare di C varia il grado di approssimazione dell’algoritmo.

L’algoritmo non è buono, il fattore di approssimazione è pessimo.

**Tecniche di arrotondamento**

**Vertex Cover con ILP**

Input Vertex Cover: G = (V,E) non diretto con vertici pesati, w0, ...,wm, e Q

Ci si chiede se esiste X sottoinsieme dei V s.t. per ogni e in E una delle due estremità è in V, e la somma dei pesi della soluzione sia <= Q.

*Sia x\_i una variabile binaria, 0 se non è stato scelto, 1 se è stato scelto.*

Costruiamo un ILP:

* x\_i + x\_j >= 1 per ogni coppia i,j in E
* 0 <= x\_i <= 1, precisamente o 0 o 1.

Funzione obiettivo: min(sommatoria per ogni V del peso del vertice \* x\_i)

Lemma 1: w\*\_LP <= w\*\_ILP

Se trasformiamo in un problema di programmazione lineare (non intera) è possibile risolverla in tempo polinomiale, e chiamiamo questa soluzione **r\**.***

*Sia r\_i una variabile binaria, 0 se x\_i < 1/2, 1 se x\_i >= ½*

Lemma 2: La soluzione **r** è ammissibile per ILP

**Dimostrazione**

Deve essere vero che per ogni coppia i,j in E allora r\_i + r\_j > 1.

Per assurdo supponiamo r\_i + r\_j < 1, questo vuol dire che sono entrambi a 0, poichè la somma se no sarebbe 1.

Quindi entrambi sono < ½, ma questo è impossibile perchè nel problema di programmazione lineare c’è il vincolo che la somma deve essere >= 1, in altre parole non sarebbe una soluzione.

Lemma 3:

Per ogni vertice r\_i <= 2x\*\_i

Se r\_i = 0 ovvio

Se r\_i = 1 allora x\*\_i >= ½ e quindi 2x\*\_i >= 1 = r\_i

Ora dimostriamo il primo Lemma:

**Dimostrazione**

Per ogni vertice:

Sum w\_i\*r\_i <= 2 \* Sum w\_i\*x\*\_i

L’utilizzo di LP porta a una 2-apx per vertex cover.

**TSP**

Lemma 1: Un multigrafo ammette un circuito Euleriano sse tutti i vertici hanno grado pari.

Dimostrazione: grafica; nel caso in cui avessi un solo lato come grado, non avrei in alcun modo la possibilità di uscire, per cui ritornerei su un arco già visititato.

Lemma 2: In ogni grafo non orientato il numero di nodi con grado dispari è pari (lemma delle strette di mano).

Dimostrazione: per mantenere la parità, bisogna che il numero di gradi dispari sia pari (in quato dispari + dispari = pari). I numeri di grado pari non mutano la parità.

Lemma 3: Il peso dello spanning tree è al più uguale al peso della soluzione ottima: δ(T) ≤ δ∗

* + Dimostrazione: Rimuovendo qualunque lato dal circuito hamiltoniano, si ottiene un albero, che è >= del MST, che a sua volta è <= del peso della soluzione ottima perchè considera un lato in meno.
* ll costo del perfect matching è al massimo la metà della soluzione ottima: δ ( M ) ≤ 1/2 δ ∗
  + Dimostrazione: consideriamo due matching, M1 e M2: i loro pesi sono >= del peso del perf matching M ciascuno perchè non sono minimi→ se sommiamo i pesi di m1 e m2 avremo almeno il doppio del costo di M, ma a loro volta la somma può essere al massimo la sol. ottima, perchè al massimo potrebbe corrispondere al cammino ottimo. Otteniamo perciò: 2δ(M) ≤ δ(M1) + δ(M2) ≤ δ∗ → δ ( M ) ≤ 1/2 δ ∗

Passi dell’algortimo di Christofides:

* Input: clique con pesi che formano una metrica
* Trovo MST
* D = TrovaGradiDispari(MST)
* Restringo il grafo sui nodi di D → G[D]
* M = MinPerfectMatching(G[D])
* H = Unione tra T e M
* TT =TrovaEuleriano(H)
* TT = TrovaHamiltoniano(TT) –> output
* L'algoritmo di Christodes è 3/2 -approssimante per TSP metrico su clique.
  + Dimostrazione: il costo è determinato sia dal perf. matching che dal MST, per cui otteniamo δ(Π ̃) ≤1/2δ∗ + δ∗ ≤ 3/2δ∗
* L'analisi di Christodes è stretta.
  + Dimostrazione: vedi [appunti](https://boldi.di.unimi.it/Corsi/AlgComp2022/Appunti/Alg_26_Oct_2022.pdf) prof per le disposizione dei nodi e le due soluzioni: se rapportiamo la prima soluzione con la seconda, quella ottima (rapp. di approssimazione), risulta che il limite da n→ inf e eps →0 sia 3/2.
* Non esiste un α > 1 tale che TSP (generale) sia α-approssiamante in tempo polinomiale.
  + Dimostrazione: abbastanza chiara dagli appunti

**Minimum Partition**

Il problema minimum partition consiste nel dividere un insieme in due sottoinsiemi s.t. il massimo della somma degli elementi dei due insiemi sia minimizzato. Si puo’ vedere come Load Balance con due macchine.

**PTAS 2-LOADBALANCE**

L’algoritmo è una specie di SortedBalance, ma dopo l’ordinamento dei processi si assegnano i primi k (valore calcolato in base ad epsilon; se epsilon >=1 si assegna tutto alla prima macchina) processi in modo ottimale. I restanti n-k processi vengono assegnati con Greedybalance

*Teorema: L'algoritmo proposto fornisce una (1 + ε)-approssimazione per 2-loadbalance*. In tempo polinomiale in **n** ma esponenziale in e.

**Dimostrazione:**

Consideriamo L la metà del peso complessivo dei task:

* w\* >= L perché, se fosse il contrario, il peso ottimo non raggiungerebbe il numero di task complessivo (2L). Vediamo i casi rispetto ad *ε*:
  + *ε* >=1:

peso macchina y1 = 2L (vedi algoritmo) <=2w\* (come detto prima)<=(1+*ε*)w\* (in quanto *ε* >=1) → rispetta l’approssimazione

* *ε*<1:

supponiamo che peso y1 >= peso y2, h l’indice dell’ultimo task della prima macchina y1 e k il numero di task già assegnati alle due macchine

* Caso 2A: h<=k → ho già trovato la soluzione ottima in quanto i primi **k** task sono assegnati in modo ottimale.
* Caso 2B: h>k (parte greedy)→ consideriamo i pesi delle due macchine prima di assegnare task h:
  + w(y1) − wth ≤ w(y2)′ ≤ w(y2) → questo perchè GreedyBalance assegnerà il task alla macchina più scarica in quel momento
  + sommiamo il peso w1 ad entrambi i membri → otteniamo che il peso y1 è <= di wh/2 + L
  + Inoltre 2L = somma di tutti i task

>= wh(k+1) (come se avessi tutti i task di peso come quello successivo, perchè sono in ordine decrescente)

* Sfuttando per proprietà si arriva a dire che minimum partition e quindi 2-loadbalance è 1+e-APX.
* Tempo => O(2^1/e + nlogn)

**Knapsack (NPO-comp - FPTAS)**

Input: n oggetti con valori v1,...,vn-1 e con pesi w0,...,wn-1 e una capacità W

Output: Sottoinsieme di oggetti

Ammissibili: Insieme di oggetti che non eccedono il peso dello zaino W

Costo: Somma del valore degli oggetti scelti

Tipo: Max

**Dynamic programming**

Sia DP[i,w] il massimo valore che riesco ad ottenere con i primi i oggetti e se il peso dello zaino fosse w.

Vale 0 se i = 0 e w = 0:

max(DP[i-1,w] e DP[i-1, w - wi] + vi) otherwise: prendo il massimo tra il valore precedente (non vale la pena aggiungere l’oggetto *i*) e l’oggetto *i* (scarto quello precedente).

Pseudopolinomiale O(nK) => perchè per tenere in memoria K ci servono 2^|W| bit

**Dynamic programming rivisitato**

Cambia il contenuto della matrice.

Sia DP[i,v] il minimo **peso** che devo avere, per portare a casa *i* oggetti con un valore complessivo >= v.

Vale 0 se v = 0

min(DP[i-1,v],DP[i-1,v-vi]+wi) otherwise

Pseudopolinomiale O(n^2\*v\_max) => per il secondo termine

Osservazione 1: Si puo’ restringere la matrice in larghezza, arrotondando i valori e introducendo una approssimazione nel valore ma non nella soluzione.

**Dynamic rescaling**

Sia t = e\*v\_max / 2n s.t. 0<=e<=1 and n numero di oggetti

Consideriamo i 3 problemi (i valori di P’ e P’’ sono tutti approssimati per eccesso):

P = (v,w,W) s.t. S\*

P’ = (v’,w,W) s.t. S’\*

P’’ = (v’’/t,w,W) s.t. S’’\*

v’ = v[i]/t \* t => quindi S’\* = S’’\*

v’’ = v[i]/t (arrotondati per eccesso per ottenere i valori interi utili al DP)

vi’=t\*v’’

*Teorema: Sia S una soluzione ammissibile di P, allora: somm. (i app.* S\**)vi <= (1+eps.) somm. (i app.* S’\**)vi: qualsiasi soluzione di knapsack è minore o uguale a quella ottima calcolata in DP per (1 + eps)*

**Dimostrazione:**

Tenendo conto delle considerazioni precedenti:

* sappiamo che la sommatoria di tutti i valori v di S\* è <= della sommatoria con i v’ valori per l’arrotondamento per eccesso.
* questa stessa sommatoria è <= dei v’ valori appartenenti ad S’\* (in quanto essendo ottimo, si presume che si massimizza l’output possibile)
* L’insieme S’\* equivale per definizione a quello S’’\*, quindi la sommatoria dei valori v’ p <= a v’ + t. Posso estrarre il valore t dalla sommatoria e moltiplicare per n.
* Infine otteniamo la **somm. (i app. a S’\*) v’ + (e\*v\_max)/2**
* Consideriamo ora l’insieme con solamente il valore massimo. Confrontandolo col risultato ottenuto precedentemente e ponendolo <= “allo stesso ma senza e.” (in quanto e <=1), avremo:
  + v\_max <= somm. (i app. S’\*)v’ + v\_max/2
    - quindi: **somm. (i app. S’\*)v’ >= v\_max/2**

Mettendo insieme otteniamo che:

somm. (i app. a S\*) v’ <= somm. (i app. a S’\*) v’ + (e\*v\_max)/2

<= somm. (i app. a S’\*) v’ + e\*somm. (i app. a S’\*) v’

<= (1+e) somm. (i app. a S’\*) v’

Corollario: Risolvendo il problema P’ si ottiene una soluzione il cui valore per P è (1 + ε)-approssimazione di Knapsack in tempo e spazio O(n^2/e ).

Al variare di e si aumenta o si abbassa il tempo al completamento dell’algoritmo.

FPTAS

**(Global) Minimum Cut**

Input: G = (V,E)

Output: Insiemi vertici scelti V1 e V2

Ammissibili: {xy in E s.t. x in V1 & y in V2} (numero di vertici che hanno una estremità in V1 e l’altra in V2)

Costo: |E| (numero di lati toccati nel taglio)

Tipo: Min

Il problema di min cut globale cerca di individuare un taglio in un grafo, in modo da minimizzare il numero di archi che incidono sulle due partizioni create.

**PROBABILISTIC ALGORITHMS**

**KARGER**

Lemma 1: Se esiste un vertice di cardinalità d, allora esiste un E\* di cardinalità <= d.

Contrazione: Operazione che consiste nell’eliminare un arco e unire le due estremità in un unico nodo.

Osservazioni: Sia ora E\_S∗ un taglio ottimo e k∗ = |E\_S∗|. Sia Gi il grafo dopo *i* passi dell’algoritmo.

1. Gi ha n−i vertici e numero di lati ≤m−i

2. Ogni taglio di Gi corrisponde a un taglio di G

3. Il grado minimo di un vertice in Gi è sempre ≥ k∗, se così non fosse, Gi avrebbe un taglio < k∗, quindi anche G, per il punto 2

4. Il punto 1 e 3 fanno in modo che il numero di lati sia ≥ k∗(n−i)/2

Lemma 2: Vale che: m−i≥ k∗(n−i)/2

**Dimostrazione:**

Sappiamo per definizione (handshaking), che il doppio dei lati in un grafo equivale alla somma dei gradi di tutti i vertici; estendendo l’osservazione 3, equivale a dire che la somma dei gradi sia >= di k\*(cardinalità dei lati). Per l’oss. 1, sappiamo che il numero di vertici equivale a (n-i), per cui otterremo che il numero dei lati (m-i) sia >= di k\* per (n-i)

Lemma 3: Vale che per ogni i: P(Ei|E1,...,Ei−1)≥ **(n-i-2)/(n-i)**

Sia E\_i l’evento che all’i-esima iterazione dell’algoritmo di Karger non contraggo nessun lato di E\*\_s

**Dimostrazione:**

Consideriamo la probabilità P(Ei|E1,...,Ei−1) come l’inverso, cioè 1 meno la probabilità opposta a P(Ei|E1,...,Ei−1). Essa equivale al numero di lati ottimo k\* diviso il numero dei lati del grafo per quell’indice *i.* I lati di Gi sono >= come detto nel lemma precedente a (K\*(n-i))/2. Semplificando ed eseguendo i calcoli otteniamo il lemma.

*Teorema: L'algoritmo di Karger trova l'ottimo con probabilità ≥ 1 /(n 2)*

**Dimostrazione:**

La soluzione ottima coincide con la probabilità dell’intersezione di tutti gli eventi fino al terzultimo (in quanto l’algoritmo termina quando rimangono gli ultimi due vertici dopo le contrazioni). L’intersezione (o AND) equivale al prodotto delle probabilità di ciascun evento dati quelli precedenti (P[E1]·P[E2|E1]·P[E3|E1,E2]...). Essi sono maggiori uguali, a seconda dell’indice: **(n-i-2)/(n-1)** (dal lemma precedente) ciascuno. La produttoria sia al numeratore che al denominatore di questi termini equivale a dire (n-2)!/(n!/2). Sviluppando i calcoli otteniamo il reciproco del coefficiente binomiale 1 / (n 2).

Corollario 1: L'esecuzione dell'algoritmo di Karger (n 2) ln n volte e prendendo il taglio minimo, l'ottimo di ottiene con probabilità ≥ 1 − 1/n

**Dimostrazione:**

L’ottimo non si trova con probabilità <= 1 - 1/(n 2). Se eleviamo per il numero di volte che dobbiamo eseguire l’algoritmo, **(n 2) ln n volte**, sviluppando i calcoli otteniamo 1/n (che è la probabilità di non trovare mai l’ottimo), per cui l’ottimo è 1 - 1/n

**SET COVER PROBABILISTICO**

* L'algoritmo proposto produce una soluzione ammissibile con probabilità ≥ 1 − e^−k.
  + Dimostrazione: sia v^ la sommatoria delle m soluzioni pesate con un certo wi. Inoltre sappiamo che la probabilità di una soluzione ammissibile equivale ad 1 meno la probabilità della soluzione errata. Nel nostro caso la soluzione errata è l’insieme degli insiemi non scelti per la copertura (Eu). L’unione di questi eventi è >= della somma delle probabilità di ciascuno di questi eventi per la disuguaglianza di Boole. La probabilità di Eu può essere vista anche come la produttoria degli elementi con probabilità con i non appart. ad I (insieme in cui aggiungo x^i nell’algoritmo secondo una certa probabilità) → probabilità inversa → sommatoria della produttoria di 1-x^i ripetuto k+lnn volte (sempre per l’algoritmo). Sappiamo poi che e^x >= 1 +x → e ^-x<=1-x → produttoria di esponenziale alla -x moltiplicato per k + lnn → sposto la produttoria all’esponente così da avere la sommatoria di tutti gli x^i: ricordando che x^i è la probabilità di scelta per ciascun sottoinsieme, da PL sappiamo che la somma di tutti gli x\_i deve essere >= 1 (per avere una soluzione ammissibile)→ 1− Somm u appart. U. e^−(k+lnn) → card. U = n → 1 - n(e^-k\*e-^lnn) → dopo semplificazioni e calcoli → 1 - e^-k
* Per ogni α > 0, sia R il fattore di approssimazione dell'algoritmo. P[R≥α(k+lnn)]≤ 1/α, Ovvero è poco probabile avere un fattore di approssimazione brutto.
  + Dimostrazione: Per Boole, la probabilità che Si venga scelto, quindi la probabilità dell’unione di tutti gli eventi possibili, è <= di (k + lnn)x^i, ovvero la somma di tutti gli eventi possibili (in quanto nell’algoritmo si itera k + lnn volte). Per cui, consideriamo l’evento v = Somm. di wi\*la probabilità che Si sia scelto → Boole → <= somm. wi\*(k+lnn)\*x^i. Ciò avevamo detto nella dimostrazione precedente ad essere uguale a v^, per cui avremo v^\*(k+lnn)<=v\* moltiplicato per (k+lnn). Passiamo al rapporto di approssimazione, ed in particolare consideriamo ciò che dice il teorema in questione, ovvero: P[v/v∗ ≥α(k+lnn)] → Markov→ P[v/v∗ ≥α(k+lnn)] <= E[v/v\*]/α(k+lnn) → v<=v\*(k+lnn) → v\*(k+lnn)/v\* per α \*(k+lnn) =1/α.

Quindi possiamo affermare che il nostro algoritmo trova una soluzione spesso (1-e^-k volte) e la probabilità che quando la trova sia una buona soluzione è alta (α(k+ln(n))).

* Con k = 3 c'è almeno il 45% di probabilità che l'algoritmo emetta una soluzione ammissibile con fattore di approssimazione al più 6 + 2ln n
  + Dimostrazione: Consideriamo gli eventi che non ammettono una soluzione ammissibile o non ottima:
    - E1 (non ammissibile)>= inv(1 -e ^-k) (I teorema)<=e ^ -k (k=3) = 0.049
    - E2 (non ottimo) <= (teorema 2: P[R≥2(3+lnn)]≤ 1/2) 1/alfa = ½
    - La probabilità che l’ottimo o l’ammissibile accadano è uguale ad 1- la probabilità che non accadano
    - Per Boole, la probabilità dei due eventi è <= alla somma della probabilità di ciascuno, per cui: 1-(½ + 0.049) = 1 - 0.55=0.45 = 45%

**MaxEkSat**

* Un assegnamento casuale delle variabili, porta ad una soluzione che soddisfa in valore atteso ≥ ((2^k −1)/2^k)\*t clausole, ovvero, sia T la variabile aleatoria che rappresenta il numero di clausole soddisfatte: E[T] = ((2^k −1)/2^k)\*t
  + Dimostrazione: definiamo:
    - x1, . . . , xm le variabili che compaiono nella formula e c1,...,ct le clausole.
    - Variabili aleatorie:
      * X1,...,Xm ∼Unif({0,1}), doveXi è il lvalore assegnato a xi
      * C1, . . . , Ct ognuna delle quali vale 1 sse la clausola ci è soddisfatta
      * T numero di clausole soddisfatte, T = Somm. i=1 a t Ci
    - Il valore atteso, che chiamiamo E[T], è dato dalla sommatoria dell’evento E dati i valori delle variabili aleatorie X\_i moltiplicato per la probabilità dell’evento E, che a sua volta può essere riscritto come il prodotto della probabilità di ciascuna variabile aleatoria. Essendo tutte equivalenti ad ½ (o 0 o 1) le raccogliamo come (1/2)^n. T è il numero di clausole soddisfatte, per cui lo sostituiamo con Somm. i=1 a t Ci. Per linearità portiamo la sommatoria all’esterno ed esaminiamo E[Ci|X1 = b1,...,Xn = bn]. Dato che per ciascuna clausola ho k variabili, esiste un solo assegnamento delle k che rende la clausola stessa falsa (OR) (=0), per cui avremo 2^(n-k), dove n indica il numero di variabili totali. Il numero di clausole vere sarà perciò 2^n - 2^(n-k) (totale meno quelle false). A questo punto, proseguendo con l’equazione, sostituiamo e otterremo: ½^n\*t(2^n-2^n-k)=...=t\*(2^k-½^k).
    - “Derandomizziamo il teorema”, ossia consideriamo le prime j variabili aleatorie (0<=j<=n) conoscendone esattamente i valori → j indica il numero di variabili assegnate deterministicamente. Per induzione, arriviamo a dire che uno dei due valori atttesi è maggiore di *risultato*, per cui dimostriamo il >=.
* Algoritmo derandomizzato: Itero sugli indici delle variabili e dichiaro delta0 e delta1, che sono rispettivamente di quanto aumenta il valore atteso se assegno xi=0 o 1, e i due insiemi in cui aggiungerò il numero di clausole rese vere per xi=0 o 1. Faccio un ciclo for annidato che itera per il numero di clausole (parentesi) in cui se l’indice compare nell’insieme delle clausole già assegnate vado alla prossima iterazione, altrimenti mi calcolo *h* che mi dice il numero di variabili presenti nella clausola maggiori dell’indice *i* che sto considerando. Infine aumento o diminuisco di un fattore 1/(2^h) in base se xi sia negata o meno nella clausola. Alla fine del ciclo annidato (ho analizzato tutte le clausole secondo una certa varibaile con indice i), confronterò quale dei due insiemi delta sia massimo, tale da aggiungerlo all’assegnamento finale in output.

**MaxEkSat inapprossimabilità**

* **l’algoritmo derandomizzato per MaxEkSat è ottimo.**
* Scegliamo un linguaggio L in NP-complete (quindi L è in PCP[O(log\_n), O(1)]) e consideriamo una macchina v in PCP[O(log\_n), O(1)] e consideriamo un input z in 2\*.

A questo punto generiamo una sequenza R di bit random r(|z|) (ovviamente tutte le possibili sequenze sono |RR| = 2^r(|z|))) e il verificatore v produrrà q query all’oracolo (i1R…iqR). Per ognuna di queste query il verificatore otterrà delle risposte dall’oracolo che indicano se l’input verrà accettato oppure no (w\_i1R…w\_iqR).

Adesso definiamo una funzione fR(w\_i1R…w\_iqR) che date tutte le risposte ci dice se l’input verrà accettato oppure no.

L’idea è quella di descrivere il comportamento di fR come una CNF quindi introduciamo un numero di variabili booleane pari al numero di query create dal verificatore (x\_1, x\_2, …, x\_|w| => |w| := |q|), quindi in una clausola ci possono essere al massimo |q| letterali. A questo punto la funzione fR si puo’ scrivere come una formula logica che chiamiamo *phizR* che utilizza queste variabili booleane. Allora il comportamento del verificatore è definito come *phiz=*(congiunzione)AND\_{da R a RR} di *phizR.*

*phiz* ha una sottoformula per ogni possibile stringa random, quindi ci sono |RR| sottoformule con al più 2^q clausole, per un totale di 2^r(|z|))\*2^q = O(|z|) clausole (al massimo).

In secondo luogo, possiamo notare che:

* se z è in L il verificatore deve accettare con probabilità 1 per qualche w^ che quindi rende vera *phiz* (tutte le *phizR* ) => è soddisfacibile.
* se z non è in L per ogni w si soddisfa al più, meno della metà delle *phizR,* ossia non esiste un w che soddisfi la metà o più delle clausole.

IMPORTANTE: Questo ci dice che delle |RR|\*2^q clausole di cui *phiz* è costituita, nel secondo caso(z non in L) ogni w rende vere al massimo un numero di clausole <= (|RR|/2)\*(2^q) + (|RR|/2)\*(2^q - 1) .

**Dimostrazione:**

Sia L in NP-comp. Per questo linguaggio esiste una specifica funzione r(|n|) in O(log(n)) e una specifica q nei Naturali s.t. L è in PCP[r,q].

Ora fissiamo e^ = 1/2^(q+1), e supponiamo che MaxSat sia (1+e^)-appr. possiamo notare che (come prima):

* se z è in L -> t(*phiz*) >= t\*(*phiz*) / 1+e^ := |R|\*2^q / 1+e^ = A
* se z non è in L -> t(*phiz*) <= t\*(*phiz*) := |R|\*2^q - |R| / 2 = B

A - B > 0! Ma questo è **impossibile** perchè potremmo usare tutta la catena polinomiale di calcoli per DECIDERE se z è in L.

ASSURDO se P != NP

**Max Indipendent Set inapprossimabilità**

* Teorema: Per ogni e > 0 INDIPENDENT SET non è (2-e)-appr. in tempo polinomiale se P!=NP

**Dimostrazione:**

Sia L in NP-comp. Per questo linguaggio esiste una specifica funzione r(|n|) in O(log(n)) e una specifica q nei Naturali s.t. L è in PCP[r,q].

Per ogni z in 2\* si consideri l’insieme R\_z sequenze di bit random di cardinalità |R\_z| = 2^r(|z|) e per ogni possibile R in R\_z tutte le possibili QzR possibili risposte dell'oracolo di cardinalità 2q. Quindi definiamo l’insieme C\_z = l’unione per tutti gli R in R\_z {R} \* QzR dei possibili input al probabilistic checker (sono una quantità polinomiale, ovviamente), ognuno dei quali risponderà **si** o **no.**

Definiamo A\_z un sottoinsieme di C\_z di tutte le configurazioni accettate nella forma (R, {(i1R:v1)…(iqR:vq)}) e abbiamo che |A\_z| <= |C\_z| = 2^r(|z|) \* 2q = **O(|z|).**

Ora creiamo un grafo G\_z sulle configurazioni accettanti e inseriamo un arco tra (R, {(i1R:v1)…(iqR:vq)}) → (R’, {(i1R’:v’1)…(iqR’:v’q)}) sse le due configurazioni sono **incompatibili** (R = R’ oppure se nella stessa posizione ci sarebbero due v diverse). Se non trovo archi avrò un Indipendent Set.

Come in MaxEkSat possiamo notare che:

* se z è in L, allora G\_z ha un Indipendent Set di cardinalità >= 2^r(|z|)

**Dimostrazione:**

Se z è in L, allora esiste una w (stringa dell’oracolo) in 2^q tale che il verificatore accetta con probabilità 1. Questo vuol dire che tutte le configurazioni ottenute al variare delle possibili stringhe random sono compatibili con w, e sono 2^r(|z|). Nota che non ci sarebbero archi in G\_z => sono un **Indipendent Set**

* se z non è in L, allora G\_z ha un Indipendent Set di cardinalità <= 2^r(|z|) - 1 (= ½ \* 2^r(|z|))

**Dimostrazione:**

Se z non è in L, e supponendo che ne esista un Indipendent Set di cardinalità maggiore, allora esisterebbe una w tale che tutte le simulazioni verrebbero verificate positivamente. Ma questo non è possibile perchè avremmo la più della metà delle simulazioni verificate (probabilità > 1/2), in other words non è possibile perchè siamo nel caso in cui z non è in L quindi la probabilità deve essere < ½.

Se ora esistesse un algoritmo A che dà una approssimazione migliore di 2 (quindi la e del teorema iniziale > 0), riusciremmo a distinguere i due casi e di conseguenza decidere se x è in L in tempo polinomiale.

ASSURDO se P != NP

**Jacobson Rank**

*Four Russian Trick*

Dato un vettore **b** di dimensione n:

* Si divide in superblocchi di dimensione (log n)^2
* I superblocchi si dividono in blocchi di dimensione ½\*log n

Memorizzando la tabella di rank per un certo blocco avremmo ½\*log n righe ognuna con un valore che occupa log(½\*log n). Se questo lo facessimo per ogni tipo blocco avremmo una complessità spaziale di (½\*log n \* log(½\*log n) \* 2^(½\*log n) bit = o(n).

Per tipo di blocco si intende come è composto lo stesso, se un blocco è composto da k bit allora i possibili tipi di blocchi sono 2^k.

**Costruzione:**

1. per ogni superblocco, il numero di 1 prima del superblocco => S[i]
2. per ogni blocco, il numero di 1 tra l’inizio del superblocco che lo contiene, e l’inizio del blocco => B[i]
3. la tabella appena discussa => TAB

**Calcolo del rank:**

Sia p la posizione che vogliamo cercare, allora il rank di questa posizione è dato da:

*rank(p) =* S[p / (log n)^2] + B[p / ½ log n] + TAB[t, p mod ½ log n]

Per sapere quanti 1 ci sono fino alla posizione p dobbiamo:

* calcolare il superblocco di appartenenza e quanti 1 in S(p)
* calcolare il blocco di appartenenza e quanti 1 ci sono in B(p)
* a questo punto ci manca solo quanti 1 ci sono dall’inizio del blocco a p => FRT usando la tabella

Il t in TAB equivale al tipo di blocco in cui si trova p.

La primitiva è ottenuta in tempo costante.

Struttura succinta e stessa efficienza della struttura naive.

**Clarke Selection**

Dato un vettore b di dimensione n:

*select(k)* = posizione del k-esimo 1

**Primo livello:**

Si memorizzano le posizioni degli 1 multipli di log(n)log(log(n)).

*P[i]* = select(i \* log(n)log(log(n)))

Questo vettore occupa (n / log(n)log(log(n))) \* log(n) = **o(n)** dove n sono il numero di 1 in b, nel caso peggiore sono n.

**Secondo livello:**

Per le posizioni che non sono multiple di log(n)log(log(n)), si costruisce il secondo livello che è costruito differentemente in base ad un indice calcolato per ogni P[i].

*r[i]* = P[i+1] - P[i]

Se la distanza tre le due posizioni è 1 allora r[i] varrà log(n)log(log(n)), altrimenti sarà maggiore.

In base alla densità degli 1 distinguiamo due casi.

*Caso Sparso: r[i]* >= (log(n)log(log(n)))^2

Questo significa che in **b** ci sono molti 0 tra gli 1 nelle posizioni P[i+1] e P[i] (sempre multiple di log(n)loglog(n)).

* Sia S[i] la lista delle posizioni degli 1 tra le due posizioni (come differenza da P[i]).
* La grandezza di S[i] = log(n)log(log(n)) perchè stiamo contando esattamente gli 1 tra le due posizioni.
* Ad ognuno di essi si associa un numero, che è <= log(*r[i]*)

Space Complexity

log(n)log(log(n)) \* log(*r[i]*) <= *r[i] /* log(log(n)) <= *n /* log(log(n)) = **o(n)** bit

*Caso Denso: r[i]* <= (log(n)log(log(n)))^2

Questo significa che in **b** ci sono pochi 0 e 1 tra le posizioni P[i+1] e P[i].

* Si memorizzano le posizioni degli 1 multipli di log(*r[i]*)log(log(n)) (come differenza da P[i])
* Sj[i] = select(j \* log(*r[i]*)log(log(n))) - P[i]

Space Complexity

log(n)log(log(n)) / log(*r[i]*)log(log(n)) \* log(*r[i]*) <= *r[i] /* log(log(n)) <= *n /* log(log(n)) = **o(n)** bit

**Terzo livello:**

Se nel Secondo livello si è nel caso denso si utilizza questo livello, che permette di memorizzare la posizione degli 1 che non sono state salvate nel secondo livello.

*r[i,j]* = S[i][j + 1] - S[i][j]

In base alla densità degli 1 distinguiamo due casi.

*Caso Sparso: r[i,j]* >= log(r[i])log(log(n))^2 \* log(*r[i,j])*

Questo significa che in **b** ci sono molti 0 e 1 tra le posizioni S[i][j + 1] e S[i][j].

* Si memorizzano le posizioni degli 1 tra S[i][j + 1] e S[i][j], questo vuol dire che
* La grandezza è log(r[i])log(log(n)) perchè siamo contando esattamente gli 1 tra le due posizioni.
* Ad ognuno di essi si associa un numero, che è <= log(*r[i,j]*)

Space Complexity

log(r[i])log(log(n)) \* log(*r[i,j]*) <= *r[i,j] /* log(log(n)) <= *n /* log(log(n)) = **o(n)** bit

*Caso Denso: r[i,j]* <= log(r[i])log(log(n))^2 \* log(*r[i,j])*

Questo significa che in **b** ci sono pochi 0 e 1 tra le posizioni P[i+1] e P[i].

Si utilizza il *Four Russian Trick.*

Osservazione 1: log(*r[i,j])* <= log(*r[i])* <= log(log(n)log(log(n)))^2 <= 4log(log(n))

Osservazione 2: *r[i,j] <=* log(r[i])log(log(n))^2 \* log(*r[i,j]) <=* 16(log(log(n)))^4

Four Russian Trick space complexity:

* 2^*r[i,j]* enumerazioni di sottovettori, ossia la dimensione tra S[i][j + 1] e S[i][j]
* Per ognuna di queste enumerazioni è necessario salvare la posizione di *r[i,j]* utilizzando una memoria pari a log(*r[i,j])*

2^*r[i,j]* \* *r[i,j]* \* log(*r[i,j]) =* 16(log(log(n)))^4 \* 2 ^ 16(log(log(n)))^4 \* log(16(log(log(n)))^4) = o(n)

Nel worst case scenario la space complexity è di:

* o(n) primo livello
* o(n) secondo livello
* o(n) terzo livello

Quindi in totale è n + o(n) e ha accesso in tempo costante, quindi è succinta. Ma non è usabile in quanto loglog(n)^4\*16 è enorme.

**Binary Tree**

**Corollario 5.7.** ∀𝑛 log2(𝐶𝑛) = 2𝑛 + 𝑂(log2(𝑛)) *Dimostrazione.* Utilizzando l’approssimazione di Stirling

𝑥! ≈ √2𝜋𝑥(𝑥𝑒)𝑥

si ha che

(che può essere dimostrato asintoticamente corretto). Questo significa che  
log2(𝐶𝑛) = 𝑛 log2(4) − 12 log2(𝜋𝑛3) = 2𝑛 − 32 log2(𝑛) − 12 log2(𝜋) = 2𝑛 + 𝑂(log2(𝑛))

**Corollario 5.8.** *Per memorizzare alberi binari con* 𝑛 *nodi interni sono necessari* 𝑍𝑛 = 2𝑛 + 𝑂(log2(𝑛)) *bit*

L’ADT che vogliamo costruire per un albero binario è definito a partire da una definizione di nodi interni 𝐼, nodi interni 𝐸, e relazioni genitore-figlio. Le operazioni che vogliamo eseguire su questi oggetti sono:

∀𝑛∈(𝐼∪𝐸) **is**\_**leaf**(𝑛)=1 ⟺ 𝑛∈𝐸

∀𝑛 ∈ (𝐼 ∪ 𝐸) **left**\_**child**(𝑛) = {𝑛𝑙|𝑛𝑙 è il figlio sinistro di 𝑛} ∀𝑛 ∈ (𝐼 ∪ 𝐸) **right**\_**child**(𝑛) = {𝑛𝑙|𝑛𝑙 è il figlio destro di 𝑛}

∀𝑛∈(𝐼∪𝐸) **parent**(𝑛)={𝑝|𝑛 è il figlio di 𝑝}

Per capire come implementare l’ADT utilizzando il vettore **v**, ci poniamo nella situazione di dover trovare, dato un nodo numerato 𝑝 in un albero binario 𝑇, i suoi due figli, i quali sono numerati 𝑞 e 𝑞 + 1. Sia quindi 𝑇′ un sottoalbero di 𝑇, radicato nella stessa radice di 𝑇, che contiene tutti i nodi di 𝑇 numerati fino al nodo precedente al nodo 𝑞,

𝑞 = |𝑖𝑛𝑡(𝑇′)| + |𝑒𝑥𝑡(𝑇′)| = 2|𝑖𝑛𝑡(𝑇′)| + 1 = 2|𝐸𝛵,𝑝| + 1 = 2**rankv**(𝑝) + 1

Per risalire al genitore a partire da un figlio, ossia da 𝑞 risalire a 𝑝, basta osservare che 𝑝 è un nodo tale per cui

**rank**(𝑝)+1=𝑞 **rank**(𝑝)=𝑞−1 ⟹{**v** 22⟹{**v** 22

2**rank**(𝑝)+1=𝑞 se𝑞èilfigliodisinistra {**v**

2**rankv**(𝑝)+2=𝑞 se𝑞èilfigliodidestra ma possiamo concludere che

**rankv**(𝑝)+1= 𝑞2 **rankv**(𝑝)= 𝑞2 −1 **rankv**(𝑝) = ⌊𝑞2 − 12⌋

siccome a seconda che 𝑞 sia pari o dispari l’equazione sarà sempre verificata, ossia questa uguaglianza è vera se e solo se è almeno una delle due precedenti; se 𝑞2 − 12 è intero, allora è vera la prima equazione, mentre se non è intero si ottiene 𝑞2 − 1, verificando la seconda. Applichiamo ad entrami i membri l’operazione di **select**:

**selectv**(**rankv**(𝑝)) = **selectv**(⌊𝑞2 − 12⌋) ⟹ 𝑝 = **selectv**(⌊𝑞2 − 12⌋) **Complessità in spazio**

Per rappresentare un albero con 𝑛 nodi interni utilizziamo un vettore 𝑏 che ha tanti bit tanti quanti sono i nodi dell’albero, ossia 2𝑛 + 1. Oltre a questo, si utilizza lo spazio utilizzato dalle strutture di rank e select, ossia

𝐷𝑛 =2𝑛+1+𝑜(2𝑛+1)=2𝑛+1+𝑜(𝑛) con un risultato per 𝑍𝑛 pari a 2𝑛 + 𝑂(log1(𝑛)), la differenza è

𝐷𝑛 − 𝑍𝑛 = 𝑜(𝑛)

pertanto la struttura è succinta con accesso in tempo costante.

**Struttura di Elias-Fano per sequenze monotone**

Dati x[0],...,x[n] >= 0 ordinati in modo crescente s.t. esiste **u** (dimensione universo) e ogni x[i] < **u**.

L’ADT richiede una primitiva che, dato un indice i, restituisca l’i-esimo elemento in questa posizione.

project(i) = x[i]

**Rappresentazione:**

La rappresentazione banale sarebbe di usare un vettore di n numeri e per ognuno usare log(u) bit. Ma la rappresentazione di Elias-Fano utilizza meno spazio:

*l* = max{0, log(u/n)}

gli *l* bits meno significativi di ogni xi vengono memorizzati esplicitamente.

* Le parti inferiori vengono estratte definendo per ogni i, *l[i] =* x[i] mod 2^l, utilizzando n\*l bits in totale
* La restante parte superiore che è x[i] / 2^l, è rappresentata per differenza dal numero precedente in unario.

s[i] = x[i] / 2^l - x[i-1] / 2^l s.t. x[-1] = 0 (termini arrotondati per difetto)

La sequenza così creata occupa:

n + **sum da 0 a n - 1** { x[i] / 2^l - x[i-1] / 2^l } → n + u / 2^log(u/n) bit

* Se u/n è una potenza di 2:

n + u/ (u/n) = 2n bits

* Se u/n non è una potenza di 2:

n + 2n/ (u/n) = 3n bits

Considerando entrambe le parti la struttura occupa: (l + 2)n o (l + 3)n bits, sapendo che l = log(u/n) concludiamo che occupa (log(u/n) + 2)n + o(n). o(n) è lo spazio occupato dalle strutture di rank e select.

**Lower bound:**

Si dovrebbero considerare tutte le sequenze monotone s.t. 0 <= x[0] <= … <= x[n-1] <= u

Fissati n e u, quante sono?

* Esse sono in biiezione con i multinsiemi di cardinalità n sottoinsiemi di {0,1,...,n-1}
* Faccio le combinazioni con ripetizione
* Uno di questi multinsiemi si puo’ vedere come: c[0],...,c[u-1], dove ogni c[i] è il numero di occorrenze del valore i nel multinsieme ossia il numero di soluzioni intere non negative dell’equazione

c[0] + … + c[u-1] = n

A questo punto per contare e possibili soluzioni usiamo il metodo ***stars and bars*** (in quanti modi possiamo mettere u-1 bars tra n stars) => log(u + n - 1 **su** u - 1)= log(u+n-1 su n) bit

Introducendo la regola di approssimazione: (a su b) ≈𝑏log (𝑎/b)+(𝑎−𝑏)log ( 𝑎/𝑎-b ) e la regola 𝑥 ≈ ln(1 + 𝑥) siamo in grado attraverso numerosi calcoli di ottenere ≈ nlog(u/n) bit → spazio → 𝐷𝑛 = 2𝑛 + 𝑛⌈log2(𝑢/𝑛)⌉ + 𝑜(𝑛) = O(Z𝑛) (Appunti)

Per cui per raffinare l’analisi è necessario considerare il lower bound sullo specifico elemento, che è 𝑍̄𝑛

≈ log(u/n) → 𝐷𝑛 = 2𝑛 +⌈log2(𝑢/𝑛)⌉ + 𝑜(𝑛) = 𝑍̄𝑛 +O(1)

Altre assunzioni fatte per questi calcoli sono n << u e n <= rad. quadrata di u

DA CONTINUARE IN UN MOMENTO DI FELICITA’ :’D

**Struttura per parentesi ben bilanciate**

**Linguaggi di Dyck**

Un linguaggio di Dyck L è un linguaggio fissato sull’alfabeto D = {(,)} s.t. L sottoinsieme di D\* e un stringa appartenente a D\* appartiene anche a L iif:

1. #(w = #)w
2. w = w1w2 => #(w1 >= #)w2

Funzione di eccesso di una parola di Dyck, il numero di parentesi aperte con posizione minore di “i” - il numero di parentesi chiuse “i” compresa:

Ew(i) = |{j s.t. j<i e wj = “(”} - {j s.t. j<=i e wj = “)”}|

La funzione inizia a 0, e se termina con 0 la parola è nel linguaggio altrimenti no.

*Fortemente bilanciata:* se lo 0 è solo all’inizio e alla fine

*Debolmente bilanciata:* se tocca lo 0 anche in mezzo, in questo caso esiste un punto di spezzamento in cui la parola è divisibile in due parole.

**ADT stringa bilanciata**

Assumiamo di leggere la parola da sx -> dx

Per ogni p in N:

* find\_open(p) = trova parentesi aperta corrispondente alla chiusa in posizione p
* find\_close(p) = trova parentesi chiusa corrispondente all’aperta in posizione p
* enclose(p) = trova prima parentesi aperta che racchiude la parentesi in posizione p

**Rappresentazione**

Dividiamo la parola in blocchi di lunghezza l, creando k = n/l blocchi.

Notiamo che in ogni blocco ci sono parentesi chiuse e aperte:

* parentesi **vicina** se la corrispondente è nello stesso blocco
* parentesi **lontana** se la corrispondente non è nello stesso blocco
* parentesi **pioniera** se è la prima aperta lontana, per ogni blocco (sono le parentesi aperte di quelle lontane)

Dato:

* w, la nostra parola
* |w| = n

Memorizziamo oltre alla parola stessa:

* un vettore **p** di n bit, con 1 nelle posizioni delle pioniere
* un vettore **E**, che per ogni i-esimo blocco ci dice l’eccesso (quante “)” mancano per essere una parola in L)
* un vettore **M**, che per ogni i-esimo blocco mantiene la posizione della parentesi corrispondente all’i-esima pioniera
* un vettore **O**, che per ogni i-esimo blocco mantiene la prima aperta avente eccesso x-1 s.t. x è il minimo eccesso del blocco

Definiamo l = log(n)

*Teorema: Se ci sono k blocchi ci sono al massimo 2k - 3 coppie di pioniere*

**Dimostrazione:**

Dato un G = (V,E) dove V sono i blocchi ed esiste un lato tra un blocco x e y iif x contiene un pioniera che ha la sua corrispondente in y. Dimostriamo per induzione di k:

1. se l’insieme dei blocchi è separabile, la parola è debolmente bilanciata ed è scomponibile in 2 parole in L. Per ipotesi induttiva nella prima ci sono 2p - 3 pioniere e nella seconda 2(k-p+1) - 3 => 2k - 4 <= 2k -3
2. se l’insieme dei blocchi non è separabile, la parola è fortemente bilanciata. Si prende la prima coppia di parentesi lontane e si rimuovono da w, quindi il numero delle pioniere in w è la somma di quelle in w’ più quella rimossa => 2k - 4 + 1 = 2k -3

Lo spazio occupato quindi è:

* w := n
* p := n + o(n)
* E := klog(n)
* O := klog(n)
* M := pioniere \* log(n) < 4klog(n)

Quindi lo spazio è: 2n + o(n) + 6klog(n) = 8n + o(n) bit

**Implementazione:**

* **find\_close(p)**

1. calcolo l’eccesso in ogni posizione del blocco a partire da E
2. si scopre se p è vicina
3. se p è lontana allora M[rank(p)], trovo il blocco in cui si chiude la pioniera di questo blocco e scorrendo indietro cerco quella che ha eccesso uguale a p.

Tempo richiesto log(n), uguale per find\_open(p)

* **enclose(p):**

1. Assumiamo di aver trovato la corrispondente p^ di p, e l’eccesso di entrambe sia *e*
2. Si cerca alla sx della posizione della parentesi aperta se c’è una posizione con eccesso *e*-1, che segna l’aperta precedente
3. Se non si trova, si cerca alla dx della posizione della parentesi chiusa se ce una posizione con eccesso *e*-1, che segna la chiusa successiva
4. Se non si trova a questo punto, basta usare il vettore O per trovare la posizione della parentesi cercata

**Lower bound:**

Una foresta ordinata è una sequenza ordinata (per numero di nodi) di alberi ordinati.

Induttivamente:

* <>lista vuota di alberi, allora <> è una foresta ordinata
* <T1, …, Tk> sono alberi, allora <T1, …, Tk> è una foresta ordinata
* se F è una foresta ordinata, tree(F) è un albero radicato in un nuovo nodo che ha come figli tutti gli alberi di F

**Isomorfismo tra foreste ordinate e alberi binari**

Definiamo una funzione y che associa ad ogni foresta ordinata un preciso albero binario.

Induttivamente:

* y(<>) = .
* y(<tree(F), T1, … , Tk>) = (y(F), y(<T1, … , Tk>))

Le foreste ordinate sono tante quante gli alberi binari: y ‘manda’ una foresta ordinata con n nodi interni in un albero binario con altrettanti nodi interni.

**Isomorfismo tra foreste ordinate e parole di Dyck**

Definiamo X: D\_2n -> F\_n un isomorfismo tra parole ben parentesizzate di lunghezza 2n e foreste ordinate con n nodi.

Induttivamente:

* X(*e*) = <>
* X(w1, …, wk) = <X(w1), …, X(wk)>

X(*w*) = <albero radicato con figli tutti e soli gli alberi della foresta y(w)>

E sappiamo, grazie al numero di Catalano, essere circa D\_n = 2n + o(log(n))

Le parole di Dyck di lunghezza 2n hanno come lower bound Z\_n = n + o(log(n)) bit:

D\_n - Z\_n = 8n + o(n) - n - o(log(n)) = O(n)

La struttura è compatta e non succinta.

**Hash Minimale perfetto**

*Il pdf è più dettagliato, mi sto basando principalmente sugli appunti del prof*

La funzione di hash mappa gli elementi di un universo U con m bucket→ h: U → m

1. h calcolabile in tempo costante
2. h occupi spazio costante
3. Si possa estrarre uniformemente qualsiasi funzione di hash (con U e m fissi) → full randomness assumption

**Peeling sequence**

Si supponga di avere un grafo G= (V,E) non orientato. Una peeling sequence è una sequenza di coppie di archi e vertici nel quale compaiono tutti i lati e per ciascuno un vertice incidente a tale lato (hinge). L’hinge selezionato non deve essere già stato scelto per altri lati.

**Ipergrafo**

Un r-ipergrafo è G=(V,E) di vertici e *iperlati* dove ogni iperlato è un insieme di r vertici, ossia E contenuto (V su r). Anche qui vale la definizione di peeling sequence (ma per gli iperlati).

**Tecnica MWHC**

Definiamo funzione statica una funzione che mappa un elemento in un sottoinsieme X contenuto nell’universo (chiavi) con 2^r (concetto di dizionario in Python).

Per la rappresentazione, fissiamo un *m* intero e si scelgono uniformemente due funzioni di hash *h1* e *h2* diverse tra loro. Inoltre il grafo considerato deve essere aciclico. Costruiamo il grafo con vertici pari a *m* ed i lati corrispondono agli elementi dell’insieme X. Questo stesso grafo viene poi trasformato in un sistema di equazioni in cui ogni vertice è una variabile w[0],w[1]...,w[m-1] e ad ogni lato corrisponde l’equazione:

* *(Wh1(x) + Wh2(x))mod 2^****r*** *= f(x) per ogni x*

Essendo un grafo aciclico con una sequenza di peeling si possono ordinare le equazioni in modo che una delle variabili non sia comparsa prima, assegnando il valore che rende vera l’equazione alla variabile che non è ancora apparsa. Una volta memorizzate le soluzioni del sistema si calcola f(x) con:

* *f(x) =(Wh1(x) + Wh2(x))mod 2^****m***

La scelta di *m* è determinata dal teorema in cui se *m* supera 2.09 |X|, la procedura di selezione di h1 e h2 termina con probabilità 1 (quasi sempre grafo aciclico) con circa 2 tentativi per generarne le coppie.

**Spazio**

* Vettore di *m* elementi con *r* bit per cella: *mr*
* Con card. di X = *n*, si ha *mr >= 2.09nr per grafo*
* Per k-ipergrafi, la costante m deve essere circa > di 1.23*nr*

La variante compressa ridefinisce il vettore w sfruttando un vettore ausiliario b secondo cui le variabili diverse da zero vengono memorizzate con 1 , 0 altrimenti. Dopodichè ridefinisco il vettore w sfruttando il rank per i valori di b[i]=1. *Spazio → (r+1.23)n bit → (r+1.23)n <1.23rn → r>5*

*Con r=2 avremmo circa 2.46 bit per elemento.*

**ITLB**

Per capire che tipo di struttura stiamo costruendo, dobbiamo calcolare l’information-theoretical lower bound. Stiamo memorizzando una funzione da un insieme 𝑋 fissato ad un insieme 2^𝑟. Le funzioni di questo tipo sono 2^𝑟|𝛸| = 2^𝑟𝑛, definendo come prima |𝑋| = 𝑛. Quindi, l’information-theoretical lower bound è

𝑍𝑛 =log2(2^𝑟𝑛)= 𝑟𝑛 bit e la struttura che abbiamo descritto occupa:

𝐷𝑛 =1.23𝑛𝑟=𝑂(𝑍𝑛)bit

pertanto la struttura è compatta.