ALGORITMI E COMPLESSITÀ

Luca Cappelletti
Simone Sinigaglia
Prof. Sebastiano Vigna

6 CFU

2019
Informatica Magistrale
Università degli studi di Milano
Italia
2 febbraio 2020

Indice

1 Algoritmi di approssimazione

Definizione 1.1 | Algoritmo approssimato

Un algoritmo ρ-approssimato risolve istanze arbitrarie di un problema NP in tempo polinomiale identificando una soluzione che è al più a un fattore ρ dalla soluzione ottima. Il problema principale con questa categoria di algoritmi sta nel dimostrare che la soluzione prodotta è **vicina** all'ottimo, senza però conoscere quale sia il valore ottimo.

1.1 Load balancing problem

Problema 1.1 | Load balancing

Dato un insieme di m macchine $M_1, \dots, M_m$ ed un insieme di n lavori con associato un tempo di elaborazione t_j. L'obbiettivo è assegnare ogni lavoro a una macchina in modo tale che il tempo di esecuzione sia il più bilanciato il possibile: vogliamo minimizzare il tempo di esecuzione assegnato a una macchina massimo (**makespan**).

Esempio 1.1 | Greedy Balance

Un possibile algoritmo molto semplice per approcciare l'LBP è un algoritmo che itera lungo la lista dei lavori e assegna il lavoro j-esimo alla macchina il cui carico è minore sino ad ora.

Analisi 1.1 | Lowerbound per Greedy Balance

Sia T il **makespan** dell'assegnamento ottenuto dall'algoritmo. Vogliamo mostrare che T non è troppo maggiore del makespan ottimo T^*. Per fare questo confronto però dobbiamo identificare il valore T^*, che non possiamo calcolare. Ne cerchiamo pertanto un **minorante**, e un buon candidato è il **makespan medio**, dato che almeno una delle m macchine dovrà svolgere una frazione $1/m$ del lavoro.

$$T^* \geqslant \frac{1}{m} \sum_j t_j$$

In alcuni casi esistono però tempi legati a lavori che sono maggiori della media, e pertanto il minorante identificato precedentemente risulterebbe non sufficientemente forte. Si va ad aggiungere quindi un ulteriore minorante: $T^* \geqslant \max_j t_j$

Lemma 1.1 | Approssimazione del Greedy Balance

L'algoritmo Greedy Balance produce un assegnamento dei lavori alle macchine con makespan $T \leqslant 2T^*$.

Dimostrazione 1.1 | Approssimazione del Greedy Balance

Consideriamo una macchina M_i cui viene assegnato il carico massimo T. Quale è stato l'ultimo lavoro j assegnato a M_i? Se t_j non è eccessivamente più grande degli altri lavori, potremmo usare il minimale ottenuto tramite la media, altrimenti quello ottenuto tramite il massimo.

Quando assegniamo il lavoro j alla macchina M_i, la macchina M_i ha il carico minore di tutte le altre macchine ed era pari a $T_i - t_j$. Ne segue che:

$$T_i - t_j \leqslant \frac{1}{m} \sum_k T_k$$

Ma il valore $\sum_k T_k = \sum_j t_j$, per cui il valore a destra coincide con il lower bound della media:

$$T_i - t_j \leqslant T^*$$

Considerando ora il lower bound del massimo, che dice che $t_j \leqslant T^*$:

$$T_i = (T_i - t_j) + t_j \leqslant 2T^*$$

Da cui la tesi.

Esempio 1.2 | Greedy Sorted Balance

Si tratta di un algoritmo migliore del Greedy Balance, che procede quasi come il precedente ma i lavori sono assegnati utilizzando una lista ordinata in ordine decrescente.

Analisi 1.2 | Identificare un lowerbound per Greedy Sorted Balance

Avendo m macchine, se esistono meno di m lavori la soluzione identificata dall'algoritmo sarà banalmente ottima, se invece abbiamo più di m lavori possiamo introdurre un altro valore minimale.

Consideriamo i primi $m+1$ lavori ordinati: ognuno richiede almeno tempo t_{m+1}. Siccome esistono $m+1$ lavori e solo m macchine, ci deve essere una macchina cui viene assegnato due lavori, e questa macchina avrà tempo di calcolo $2t_{m+1}$.

Lemma 1.2 | Approssimazione del Greedy Sorted Balance

L'algoritmo Greedy Sorted Balance produce un assegnamento dei lavori alle macchine con makespan $T \leqslant \frac{3}{2} T^*$.

Dimostrazione 1.2 | Approssimazione del Greedy Sorted Balance

Consideriamo la macchina M_i con carico massimo. Se ad essa è stato assegnato un solo lavoro allora la soluzione è ottimale, altrimenti la macchina ha almeno due lavori.

Sia t_j l'ultimo lavoro assegnato alla macchina. Il lavoro deve essere $j \geqslant m+1$ dato che i primi m lavori sono stati assegnati ad m macchine distinte. Di conseguenza, $t_j \leqslant t_{m+1} \leqslant \frac{1}{2}T^*$. In precedenza usando i minimali avevamo ottenuto le disuguaglianze $T_i - t_j \leqslant T^*$ e $t_j \leqslant T^*$, ma in questo caso la seconda disuguaglianza può essere resa più stringente: $t_j \leqslant \frac{1}{2}T^*$.

Sommando le due disuguaglianze otteniamo:

$$T_i \leqslant \frac{3}{2}T^*$$

1.2 Set Cover

Problema 1.2 | Set Cover

Considerando un insieme di n elementi U ed una lista $S_1, S_2, \ldots, S_m$ di sottoinsiemi di U. Una **set cover** è una collezione di insiemi la cui unione è uguale a U. Nella versione pesata, ad ogni insieme $S_1, S_2, \ldots, S_m$ è assegnato un peso $w_i \geqslant 0$, ed il nostro scopo è identificare una **set cover** di peso minimo.

Definizione 1.2 | Algoritmo greedy per il Set Cover

L'algoritmo greedy costruisce la **set cover** un insieme per volta, scegliendo ad ogni iterazione l'insieme di peso minimo che copre più elementi. Se chiamiamo R l'insieme parziale di elementi ancora non coperti, definiamo il peso assegnato ad ogni insieme come:

$$\frac{w_i}{S_i \cap R}$$

Ogni volta che viene aggiunto un set S_i andiamo a rimuoverne gli elementi da R e l'algoritmo termina la sua iterazione quando R risulta vuoto.

1.3 Il problema dello zaino

Problema 1.3 | Il problema dello zaino

Si vogliono mettere n oggetti in uno zaino di capacità finita W, ogni oggetto ha un peso w_i ed un valore v_i: vogliamo massimizzare il valore totale degli oggetti trasportati senza superare la capacità massima.

Definizione 1.3 | Approssimazione polinomiale del problema dello zaino

Dopo aver scelto un parametro di arrotondamento b, il cui valore fissiamo in base al valore dell'approssimazione $1+\epsilon$ che vogliamo raggiungere:

$$b = \frac{\epsilon}{2n} \max_i v_i$$

Calcoliamo i pesi arrotondati:

$$\tilde{v}_i = \left\lceil \frac{v_i}{b} \right\rceil \cdot b$$

Procediamo quindi a risolvere il problema dello zaino con i pesi arrotondati. Il problema dello zaino con i valori $\tilde{v}_i$ hanno lo stesso insiemi di soluzioni ottime, i valori ottimi differiscono di un fattore b e i valori scalati sono interi.

1.4 Center selection problem

Problema 1.4 | Center selection problem

Dato un insieme S composto da n punti (i luoghi), vogliamo scegliere un insieme C di k punti che risultino **centrali** rispetto ai punti di S. Diciamo che l'insieme C forma una r-cover se ogni punto si S si trova a una distanza al più pari a r da uno dei centri. Il valore minimo di r per cui C è una r-cover viene chiamato **raggio di copertura** di C: l'obbiettivo è selezionare i k punti di C in modo tale da minimizzare r.

Definizione 1.4 | Approccio Greedy semplice per center selection problem

Un algoritmo Greedy molto semplice procederebbe scegliendo come primo punto il migliore possibile se dovesse scegliere un solo centro, quindi procederebbe ad aggiungere centri in modo da ridurre ad ogni step il massimo possibile il **raggio di copertura**.

Osservazione 1.1 | Limiti di un approccio Greedy semplice

Prendiamo in considerazione un'istanza con due luoghi in S e $k=2$. Sia d la distanza tra s e z: un algoritmo greedy semplice inizierebbe scegliendo il punto a metà tra i due ed il raggio di copertura sarebbe $\frac{d}{2}$. Ma ora qualsiasi altra posizione venga scelta per il secondo centro, uno dei due punti di S lo avrebbe più vicino e il **raggio di copertura non migliorerebbe**. Chiaramente per un'istanza di questo tipo la soluzione ottima sarebbe selezione i due punti stessi come centri, raggiungendo **raggio di copertura zero**.

Osservazione 1.2 | In che modo conoscere il raggio ottimo aiuta?

Supponiamo di conoscere il raggio ottimo r^* di una soluzione C^* e di dover individuare un insieme di centri C che non abbia un raggio troppo maggiore di r: Consideriamo un qualsiasi luogo $s \in S$. Deve esistere un centro $c^* \in C^*$ che copre s. Andremmo quindi a scegliere questo luogo s come centro per la nostra soluzione, non sapendo quale sia c^*, e per coprire tutti i luoghi che c^* copre nella soluzione sconosciuta C^* andremmo a raddoppiare il raggio da r^* a $2r^*$ (per diseguaglianza triangolare).

Ogni insieme di centri C identificati da questo algoritmo ha raggio di copertura $r \leqslant 2r^*$.

Lemma 1.3 | Esistenza di soluzione ottima con raggio r^*

Se l'algoritmo greedy che conosce il raggio ottimo r^* seleziona più di k centri, allora per ogni soluzione C^* di dimensione al più k, il raggio di copertura è $r(C^*) > r^*$.

Dimostrazione 1.3 | Esistenza di soluzione ottima con raggio r^*

Ragioniamo per assurdo che esista una soluzione C^* di al più k centri con raggio di copertura $r(C^*) \leqslant r^*$. Ogni centro $c \in C$ selezionato dall'algoritmo è uno dei luoghi $s \in S$, e l'insieme C^* ha raggio di copertura al più pari a r^*, per cui deve esistere un centro $c^* \in C^*$ che è al più a distanza r da un centro $c \in C$.

Vogliamo dimostrare che non esiste un centro $c^* \in C^*$ che risulti vicino a due centri della soluzione C: questo implicherebbe che ogni centro $c \in C$ è vicino a un centro ottimo c^*, ed ognuno di questi deve essere distinto e pertanto $|C^*| \geqslant |C|$ e siccome $|C| > k$ si contraddirebbe l'assunzione che C^* contiene al più k centri.

Procediamo quindi a mostrare che nessun centro ottimo $c^* \in C^*$ può essere vicino a due centri $c, c' \in C$. Ogni coppia di centri $c, c' \in C$ è separata da un distanza maggiore di $2r^*$ per costruzione e quindi se il punto c^* fosse a distanza al più r^* da ognuno dovrebbe violare la disuguaglianza triangolare.

Osservazione 1.3 | Come procediamo senza conoscere il raggio ottimo?

Possiamo iniziare da un valore scelto arbitrariamente per r, che sappiamo essere tra 0 e la distanza massima tra due luoghi qualsiasi $r_{\max}$. Un possibile valore è $r = \frac{r_{\max}}{2}$. In base all'algoritmo o possiamo identificare una soluzione di k centri con raggio di copertura al più pari a $2r$ o possiamo concludere che non esiste una soluzione con raggio di copertura al più r. Nel primo caso possiamo iterare ed abbassare la nostra approssimazione per r^*, mentre nel secondo caso la dovremo aumentare.

Si procede quindi ad iterare dicoticamente per identificare una stima sempre migliore del raggio di copertura, procedendo fino a che la distanza tra il raggio ottimo approssimato a uno step non cambia troppo dall'iterazione precedente.

La soluzione ottenuta quindi risulta essere una 2-approssimazione della soluzione ottima.

Definizione 1.5 | Algoritmo greedy per il problema della selezione dei centri

Procedendo in un modo simile all'algoritmo proposto, ma senza iterare sui possibili valori ottimali, dopo aver scelto come centro iniziare un punto s a caso, possiamo scegliere il luogo s più distante dal centro selezionato precedentemente. Se esiste un luogo a distanza $2r$ da tutti gli altri centri scelti allora il punto s più distante deve essere parte dei centri.

Lemma 1.4 | Approssimazione dell'algoritmo greedy per il problema della selezione dei centri

L'algoritmo greedy identifica un insieme di k punti C tale che $r(C) \leqslant 2r(C^*)$, dove C^* è un insieme ottimale di k punti.

Dimostrazione 1.4 | Approssimazione dell'algoritmo greedy per il problema della selezione dei centri

Sia r^* il raggio di copertura minimo per un insieme di k centri. Procediamo assumendo per assurdo di dover identificare un insieme di k centri C con $r(C) > 2r^*$. Sia s un luogo che è a distanza maggiore di $2r^*$ da un qualsiasi centro in C. Consideriamo qualche iterazione intermedia dell'algoritmo, in cui son stati selezionati alcuni centri in C'. Supponiamo di procedere ad aggiungere c' in questa iterazione: riteniamo che c' è distante almeno $2r^*$ da qualsiasi centro in C', dato che il luogo s è distante più di $2r$ da tutti i centri del set C, e quindi selezioniamo il luogo c che è il più distante da tutti gli altri centri scelti.

Ne segue che l'algoritmo greedy è una implementazione corretta per le prime k iterazioni dell'algoritmo proposto precedentemente che conosceva il raggio ottimale r^*: ad ogni iterazione aggiungiamo il punto a distanza maggiore di $2r^*$ da ogni centro selezionato precedentemente.

Ma l'algoritmo precedente avrebbe $S' \neq \emptyset$ dopo aver selezionato k centri dato che $s \in S'$, e quindi procederebbe a scegliere più di k centri concludendo alla fine che k centri non possono avere raggio di copertura r^*. Questo contraddice la nostra scelta di r, e la contraddizione prova che $r(C) \leqslant 2r^*$.

Problema 1.5 | Problema dell'insieme dominante

Un **insieme dominante** per un grafo $G = (V, E)$ è un sottoinsieme D di V tale che ogni vertice non in D è adiacente ad almeno un membro di D. Il **numero di dominazione** è il numero di vertici nel più piccolo **insieme dominante** di G.

Il **problema dell'insieme dominante** consiste nel determinare se per un dato grafo è possibile trovare un insieme dominante di cardinalità inferiore a un dato k. Si tratta di un problema NP-completo.

Teorema 1.1 | Inapprossimibilità del Center Selection problem

A meno che **P** = **NP**, non esiste nessuna ρ-approssimazione per il problema di Center Selection con $\rho < 2$.

Dimostrazione 1.5 | Inapprossimibilità del Center Selection problem

Procediamo per assurdo assumendo che sia possibile realizzare un algoritmo con $(2-\epsilon)$-approssimazione per **center selection**: un tale algoritmo potrebbe essere utilizzato per risolvere il **problema dell'insieme dominante** (un noto problema NP-Completo) in tempo polinomiale.

Sia $G = (V, E)$ e $k \in \mathbb{N}$ un'istanza di **problema dell'insieme dominante**. Costruiamo un'istanza G' di **center selection** i cui siti sono i vertici del grafo e le distanze sono definite come:

$$\text{dist}(u, v) = 1 \text{ se } (u, v) \in E$$
$$\text{dist}(u, v) = 2 \text{ se } (u, v) \notin E$$

Il grafo G ha un **insieme dominante** di dimensione k se e solo se esistono k centri C^* con $r(C^*) = 1$. Di conseguenza, se G ha un insieme dominante di dimensione k, una algoritmo di $2-\epsilon$-approssimazione per **center selection** potrebbe trovare una soluzione C^* con $r(C^*) = 1$, dato che non può usare nessun arco di distanza 2.

1.5 Il metodo del Pricing: Vertex Cover

Problema 1.6 | Vertex Cover

Un **vertex cover** in un grafo $G = (V, E)$ è un insieme $S \subseteq V$ tale che ogni lato abbia almeno un termine in S. Nella versione pesata ad ogni vertice $i \in V$ viene assegnato un peso $w_i \geq 0$, e vorremmo trovare il **vertex cover** a peso minimo.

Lemma 1.5 | Approssimare Vertex Cover con Set Cover

È possibile utilizzare l'algoritmo di approssimazione del **Set Cover** per raggiungere un algoritmo con $H(d)$-approssimazione per la versione pesata del **Vertex Cover**, dove d è il grado massimo del grafo.

Dimostrazione 1.6 | Approssimare Vertex Cover con Set Cover

Consideriamo un'istanza di **Vertex Cover pesato**, specificata da un grafo $G = (V, E)$. Definiamo un'istanza di **Set Cover** con insieme U pari a E e per ogni nodo i definiamo il set S_i consistente di tutti i lati incidenti a un nodo i e diamo a questo insieme peso w_i. Le **set cover** che coprono U corrispondono quindi precisamente a **vertex cover**. La massima dimensione degli insiemi S_i così definito coincide con il grado massimo d.

Ne segue che è possibile utilizzare l'algoritmo di approssimazione per Set Cover per trovare una soluzione al problema del **vertex cover pesato** il cui peso è una $H(d)$-approssimazione.

Osservazione 1.4 | Quanto è buona l'approssimazione del Vertex Cover col Set Cover?

L'approssimazione è piuttosto buona quando d è piccolo, ma peggiora mano a mano che d diventa più grande, arrivando a un limite logaritmico nel numero dei vertici.

Lemma 1.6 | Minimale del costo totale del pricing per Vertex Cover

Per qualsiasi vertex cover S^*, e ogni insieme di prezzi non negativi p_e, si ha che:

$$\sum_{e \in E} p_e \leq w(S^*)$$

Dimostrazione 1.7 | Minimale del costo totale del pricing per Vertex Cover

Consideriamo una vertex cover S^*: per definizione di **equità del costo**, abbiamo che:

$$\sum_{e=(i,j)} p_e \leq w_i \quad \forall i \in S^*$$

Aggiungendo queste disuguaglianze su tutti i nodi in S^* otteniamo:

$$\sum_{i \in S^*} \sum_{e=(i,j)} p_e \leq \sum_{i \in S^*} w_i = w(S^*)$$

L'espressione sulla sinistra è una somma dei costi dei lati. Siccome S^* è una vertex cover, ogni lato e contribuisce al più un termine p_e al lato sinistro. Potrebbe infatti contribuire più di una copia di p_e alla somma, siccome potrebbe essere coperto da ambo i lati in S^*. Ma i prezzi sono non-negativi, per cui la somma dei termini a sinistra è grande almeno quanto la somma di tutti i prezzi p_e:

$$\sum_{e \in E} p_e \leq \sum_{i \in S^*} \sum_{e=(i,j)} p_e$$

Combinando questa disuguaglianza con quella ottenuta precedentemente otteniamo:

$$\sum_{e \in E} p_e \leq w(S^*)$$

Definizione 1.6 | Algoritmo greedy per Vertex Cover

Diciamo che un nodo i è **pagato** se risulta che $\sum_{e=(i,j)} p_e = w_i$. Inizializziamo tutti i pesi p_e a zero, quindi iteriamo fintanto che esiste un lato di cui nessuno dei due vertici sono **pagati**: selezioniamo questo lato ed aumentiamo il prezzo p_e senza violare l'**equità del costo**. Quando il ciclo termina, la nostra vertex cover sarà l'insieme dei vertici **pagati**.

Lemma 1.7 | Equità dell'algoritmo greedy per Vertex Cover

La **vertex cover** ed i prezzi P individuati dall'algoritmo greedy per Vertex Cover soddisfano l'uguaglianza:

$$w(S) \leqslant 2 \sum_{e \in E} p_e$$

Dimostrazione 1.8 | Equità dell'algoritmo greedy per Vertex Cover

Dato che tutti i nodi in S sono **pagati**, si ha che $\sum_{e=(i,j)} p_e = w_i \quad \forall i \in S$. Sommando tutti i nodi in S otteniamo:

$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e$$

Un lato $e = (i, j)$ può essere incluso nella somma del lato destro al più due volte (nel caso in cui sia i che j risultano nella **vertex cover** S), per cui si ottiene:

$$w(S) = \sum_{i \in S} \sum_{e=(i,j)} p_e \leqslant 2 \sum_{e \in E} p_e$$

Lemma 1.8 | Algoritmo greedy per Vertex Cover è 2-approssimato

L'insieme S ritornato dall'algoritmo greedy per Vertex Cover è una **vertex cover** ed il suo costo è al più due volte il costo ottimo.

Dimostrazione 1.9 | Algoritmo greedy per Vertex Cover è 2-approssimato

Se S per assurdo non fosse una **vertex cover** non andrebbe a coprire almeno un lato $e = (i, j)$. Questo implicherebbe che ne i ne j sarebbero **pagati** ma questo impedirebbe all'algoritmo di uscire dal ciclo iterativo.

Sia p l'insieme dei prezzi per l'algoritmo e sia S^* la vertex cover ottima. Siccome i prezzi sono **equi** vale che $2 \sum_{e \in E} p_e \geqslant w(S)$ e siccome S è una **vertex cover** a prezzi non negativi vale che $\sum_{e \in E} p_e \leqslant w(S^*)$. Ne segue che:

$$w(S) \leqslant 2 \sum_{e \in E} p_e \leqslant 2w(S^*)$$

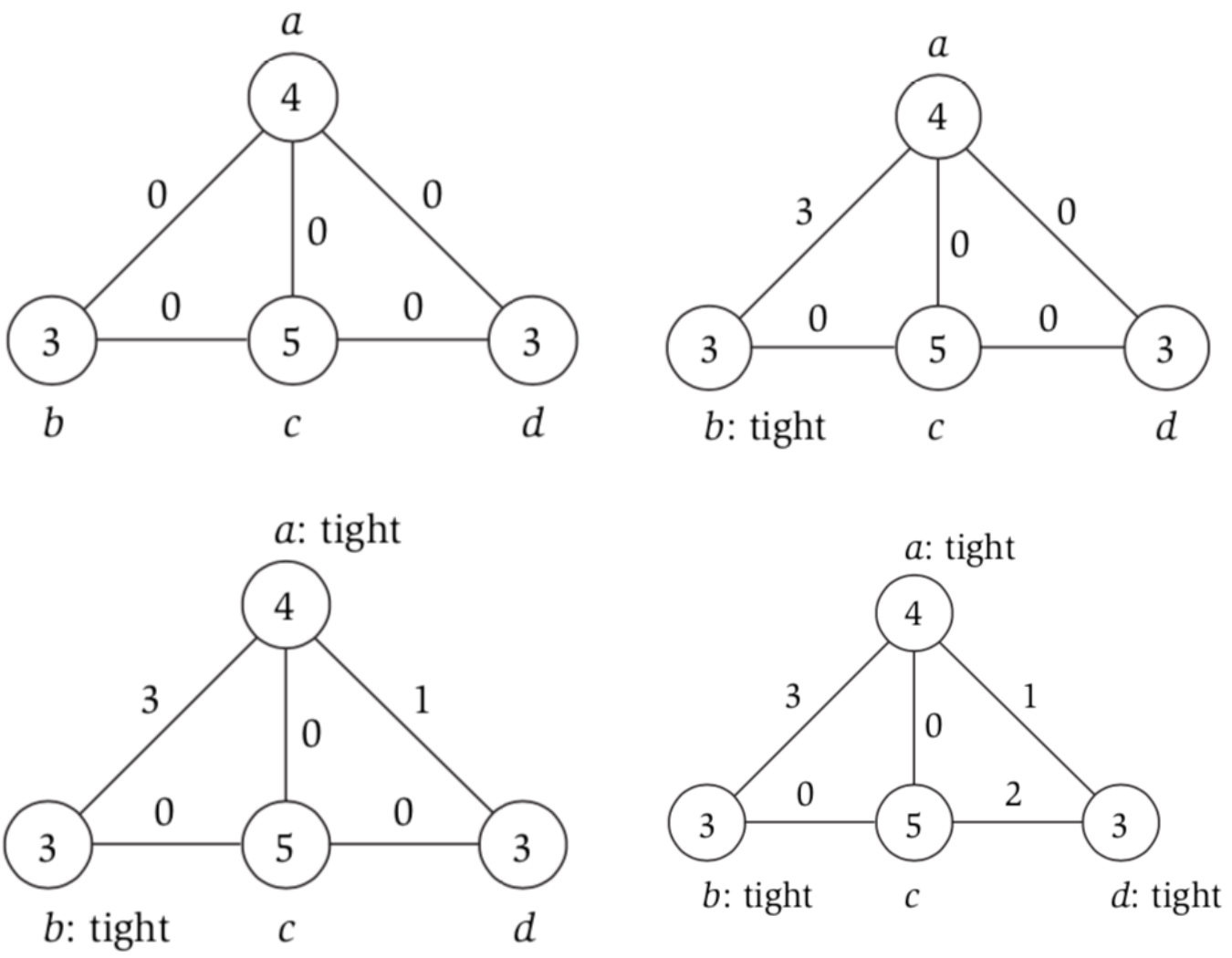

Figura 1.1: Algoritmo greedy per Vertex Cover

1.5.1 Approccio con programmazione lineare

Definizione 1.7 | Problema di PLI per vertex cover

$$\begin{aligned} \min \quad & \sum_{i \in V} w_i x_i \\ s.t. \quad & x_i + x_j \geqslant 1 \quad (i,j) \in E \\ & x_i \in \{0,1\} \quad i \in V \end{aligned}$$

Lemma 1.9 | Massimale del peso di Vertex Cover come problema di PL

Sia S^* una vertex cover di peso minimo. Allora $w_{LP} \leqslant w(S^*)$

Dimostrazione 1.10 | Massimale del peso di Vertex Cover come problema di PL

La **vertex cover** di G corrisponde alle soluzioni intere del problema di programmazione intera associato, pertanto il minimo di $\min\left(w^t x : \vec{1} \geqslant x \geqslant 0, Ax \geqslant 1\right)$ su tutti i vettori interi x corrisponde alla vertex cover di peso minimo. Per ottenere il minimo del problema di programmazione lineare, consentiamo ad x di assumere valori frazionari (rilassamento continuo), per cui il minimo del problema di programmazione lineare non risulta più grande di quello ottenuto tramite il problema di programmazione intera.

Lemma 1.10 | Massimale del peso di Vertex Cover come problema di PI

L'insieme S ottenuto tramite arrotondamento della soluzione di programmazione lineare è una **vertex cover** e $w(S) \leqslant w_{\text{LP}}$

Dimostrazione 1.11 | Massimale del peso di Vertex Cover come problema di PI

Iniziamo provando che l'insieme S ottenuto arrotondando la soluzione di PL ottenuta sia una **vertex cover**. Consideriamo un lato $e = (i,j)$: perché S sia una vertex cover almeno uno dei due vertici deve essere in S. Una dei vincoli associati al problema di programmazione lineare intera risulta è $x_i + x_j \geqslant 1$, per cui in ogni soluzione x^* che soddisfa questa disuguaglianza o $x_i^* \geqslant \frac{1}{2}$ o $x_j^* \geqslant \frac{1}{2}$. Pertanto almeno uno di questi due sarà arrotondato e o i o j sarà posto in S.

Consideriamo ora il peso $w(S)$ di questa vertex cover. L'insieme S ha solo vertici con $x_i^* \geqslant \frac{1}{2}$, per cui il problema di programmazione lineare **paga** almeno $\frac{1}{2} w_i$ per il nodo i e paghiamo solo w_i, al più pari al doppio.

Più formalmente vale la seguente catena di disuguaglianze:

$$w_{LP} w^t x^* = \sum_i w_i x_i^* \geqslant \sum_{i \in S} w_i x_i^* \geqslant \frac{1}{2} \sum_{i \in S} w_i = \frac{1}{2} w(S)$$

1.6 Pricing: problema dei passi separati

Problema 1.7 | Maximum Disjoint Paths Problem

Consideriamo un grafo direzionato G, k coppie di nodi $(s_1,t_1),(s_2,t_2),\ldots,(s_k,t_k)$ ed un intero c (capacità). Ogni coppia (s_j,t_j) vuole rappresentare una **richiesta di routing**, che richiede un cammino da s_j a t_j. Una soluzione di questa istanza consiste in un sottoinsieme di richieste soddisfacibili e l'insieme dei cammini che le soddisfano tali che nessun lato viene percorso da due cammini distinti.

Il problema consiste nel soddisfare il maggior numero di richieste possibile. L'intero c rappresenta il massimo numero di lati condivisi che riterremo ammissibili: con $c=1$ viene richiesto che i cammini siano completamente separati, con $c>1$ consentiremo qualche sovrapposizione tra i cammini.

Definizione 1.8 | Algoritmo greedy per Maximum Disjoint Paths Problem

Iniziamo da un insieme di richieste soddisfacibili vuoto e iteriamo finché non può essere identificato un nuovo cammino P_i senza sovrapposizioni. Consideriamo un tale cammino P_i, il più corto possibile, che connette una coppia (s_j,t_j). Aggiungiamo il path P_i all'insieme della soluzione per connettere la richiesta identificata da (s_j,t_j).

Lemma 1.11 | Approssimazione dell'algoritmo greedy per Maximum Disjoint Paths Problem

L'algoritmo greedy per Maximum Disjoint Paths Problem è una $(2\sqrt{m}+1)$-approssimazione, dove m è il numero dei lati.

Dimostrazione 1.12 | Approssimazione dell'algoritmo greedy per Maximum Disjoint Paths Problem

Consideriamo una soluzione ottima: sia I^* il suo insieme di coppie per cui il cammino è stato selezionato in questa soluzione ottima e sia P_i^* l'insieme dei cammini selezionati. Chiamiamo I l'insieme di coppie identificate dall'algoritmo greedy e sia P_i il cammino per ogni coppia identificata.

La chiave della dimostrazione è fare una distinzione tra cammini lunghi e corti e considerarli separatamente. Chiameremo un cammino **lungo** se ha almeno $\sqrt{m}$ lati, **corto** altrimenti. Sia I_S^* l'insieme degli indici in I^* corrispondenti a cammini corti. Il grafo G ha m lati, e ogni cammino lungo usa almeno $\sqrt{m}$ lati, quindi possono esistere al più $\sqrt{m}$ cammini lunghi in I^*.

Consideriamo ora i cammini corti in I^*. Perché I^* contenga molte più richieste soddisfatte di I dovrebbero esserci tante coppie connesse in I^* che non sono connesse in I. Consideriamo quindi coppie che sono connesse nella soluzione ottima utilizzando un cammino corto, ma che non vengono connesse dall'algoritmo greedy. Siccome il cammino P_i^* che connette s_i e t_i nella soluzione ottimale I^* è corto, la soluzione greedy lo avrebbe selezionato se fosse stato disponibile prima di selezionare un cammino più lungo, ma l'algoritmo greedy non lo ha connesso e quindi uno dei lati che compongono P_i^* deve essere in un altro cammino P_j selezionato precedentemente dall'algoritmo greedy. Diremo quindi che il lato e **blocca** il cammino P_i^*.

Ora la lunghezza dei cammini selezionati dall'algoritmo greedy aumenta monotamente, siccome ad ogni iterazione vengono scelti i cammini più corti. Il cammino P_j era selezionato prima considerando P_i^* e quindi deve essere più corto: $|P_j| \leqslant |P_i^*| \leqslant \sqrt{m}$. Ne segue che anche P_j è **corto**. Siccome i cammini usati dalla soluzione ottima sono separati, ogni lato in un cammino P_j può **bloccare** al più un cammino P_i^*. Ne segue che ogni cammino corto P_j blocca al più $\sqrt{m}$ cammini nella soluzione ottima e quindi otteniamo la disequazione:

$$|I_s^* - I| \leqslant \sum_{j \in I_s} |P_j| \leqslant |I_s|\sqrt{m}$$

Consideriamo ora la soluzione ottima come composta da 3 tipi di cammini:

1. Cammini lunghi, di cui ne esistono al più $\sqrt{m}$
2. Cammini che si trovano anche nella soluzione I
3. Cammini corti che non si trovano anche in I

Siccome $|I| \geqslant 1$ se almeno una richiesta risulta soddisfacibile, si può realizzare la disequazione:

$$|I^*| \leqslant \sqrt{m} + |I| + |I_s^* - I| \leqslant \sqrt{m} + |I| + \sqrt{m}|I_s| \leqslant (2\sqrt{m}+1)|I|$$

1.7 Problema del commesso viaggiatore (Christofides)

Problema 1.8 | Problema del commesso viaggiatore

Dato un insieme finito di vertici V e una funzione di distanza $d : V^2 \to \mathbb{R}$ che soddisfa tutte le proprietà di una distanza (**non-negatività**, **simmetria** e **disuguaglianza triangolare**). L'obbiettivo è identificare il ciclo Hamiltoniano di lunghezza minima tra tutti i vertici.

Definizione 1.9 | Christofides per TSP

Prima di tutto costruiamo un **albero ricoprente minimo** M, quindi iniziando da un qualsiasi vertice percorriamo l'albero ricoprente M in senso anti-orario: questo percorso attraverso ogni lato una volta in ogni direzione, ma non è un **percorso Hamiltoniano** perché visita alcuni vertici più volte.

Possiamo eliminare i vertici già visitati come segue: iniziando da un punto arbitrario, percorriamo il ciclo realizzato ma marchiamo i vertici che incontriamo. Quando ne incontriamo uno visitato precedentemente lo saltiamo e passiamo direttamente al vertice successivo, iterando fino ad incontrare uno non visitato o il vertice iniziale. Il risultato è un **percorso Hamiltoniano**, e la sua lunghezza non è più grande del percorso originale grazie alla **disuguaglianza triangolare**.

Sia ora O l'insieme dei vertici **a grado dispari** di M: questo insieme contiene tutte le foglie di M e potenzialmente anche qualche vertice interno. La cardinalità dell'insieme O è sicuramente pari perché la somma dei gradi di tutti i nodi deve essere pari essendo esattamente il doppio del numero di lati. Possiamo pertanto costruire un **matching perfetto** sul grafo completo che ha O come vertici e possiamo trovare un **matching minimo perfetto** P in tempo **polinomiale**.

Consideriamo ora il grafo di lati $E = E(P) \cup E(M)$: questo grafo potrebbe contenere archi multipli, dato che P ed M potrebbero sovrapporsi. Notiamo due aspetti importanti: tutti i nodi hanno grado **pari**, dato che aggiungiamo solo un lato di P incidente ad ogni nodo di O e vale che:

$$d(E) \leqslant 3d\left(T^*\right)/2$$

perché $d(M) \leqslant d(T^*)$ e $d(P) \leqslant d(T^*)/2$. Siccome ogni nodo ha grado pari possiamo costruire un percorso Euleriano (cioè un ciclo che attraversa tutti gli archi esattamente una volta). Il percorso Euleriano è costruito iniziando da un vertice arbitrario e seguendo gli archi finché non si incontra il vertice iniziale nuovamente: se per caso fosse incontrato prima di finire il ciclo Euleriano basta continuare finché non si termina il percorso completamente. Il percorso Euleriano ottenuto ha lunghezza al più $3d(T^*)/2$.

Procediamo ora a comprimere il percorso a un ciclo Hamiltoniano saltando, come fatto precedentemente, i vertici già visitati e per disuguaglianza triangolare otteniamo un ciclo Hamiltoniano di lunghezza $3d(T^*)/2$.

Lemma 1.12 | 1° approssimazione di Christofides

Il primo percorso Hamiltoniano identificato da Christofides (prima di ottenere il matching) è una 2-approssimazione dell'ottimo.

Dimostrazione 1.13 | 1° approssimazione di Christofides

Sia T il ciclo risultante dalla prima approssimazione di Christofides per TSP, e sia T^* il ciclo ottimo. Sia e un lato qualsiasi del ciclo ottimo T^*, quindi:

$$d(T) \leqslant 2d(M) \leqslant 2d\left(T^* - e\right) \leqslant 2d\left(T^*\right)$$

La prima disuguaglianza deriva dal fatto che la lunghezza del ciclo iniziale ottenuto dall'**albero ricoprente minimo** M, dato che ogni lato di M è attraversato esattamente due volte, e il ciclo ottenuto saltando alcuni vertici non è peggiore per la **disuguaglianza triangolare**. La seconda disuguaglianza deriva dal fatto che $T^* - e$ è un albero ricoprente e quindi $d(M) \leqslant d(T^* - e)$, dato che M è un albero ricoprente minimo. Infine la terza disuguaglianza deriva dal fatto che tutte le disuguaglianze sono non-negative.

Lemma 1.13 | 2° approssimazione di Christofides

Il secondo (e ultimo) percorso Hamiltoniano identificato da Christofides (prima di ottenere il matching) è una $\frac{3}{2}$-approssimazione dell'ottimo.

Dimostrazione 1.14 | 2° approssimazione di Christofides

Sia N^* il ciclo Hamiltoniano minimo costruito sull'insieme dei nodi a **grado dispari** O e siano N_1 ed N_2 i due **matching perfetti** su O ottenuti scegliendo alternando i lati di N^*. Allora:

$$d(P) \leqslant \min\left(d(N_1), d(N_2)\right) \leqslant d\left(N^*\right)/2 \leqslant d\left(T^*\right)/2$$

La prima disuguaglianza si ottiene dal fatto che N_1 ed N_2 sono **matching perfetti** per O e P è un **matching minimo perfetto** su O. La seconda disuguaglianza deriva dal fatto che il minimo di $d(N_1)$ e $d(N_2)$ è al più la media tra i due. Per la terza e ultima disuguaglianza, otteniamo il ciclo Hamiltoniano minimo (soluzione del TSP) su O dalla soluzione ottima T^* saltando i vertici pari. Per la **disuguaglianza triangolare**, la lunghezza del percorso ottenuto non è peggiore di $d(T^*)$ e $d(N^*)$ non è peggiore perché è un ciclo ottimo su O.

Consideriamo ora il grafo di lati $E = E(P) \cup E(M)$. Vale che:

$$d(E) \leqslant 3d\left(T^*\right)/2$$

perché $d(M) \leqslant d(T^*)$ e $d(P) \leqslant d(T^*)/2$ siccome costruiamo il ciclo Hamiltoniano selezionando un sottoinsieme degli archi di E risulta chiaramente che il ciclo ottenuto è una $\frac{3}{2}$-approssimazione del ciclo ottimo T^*

2

Algoritmi probabilistici e randomizzati

2.1 Probabilistically Checkable Proofs

Teorema 2.1 | Teorema di Cook

SAT è **NP-completo**.

Definizione 2.1 | Probabilistically checkable proof (PCP)

Una **probabilistically checkable proof (PCP)** è un tipo di dimostrazione che può essere verificata da un algoritmo randomizzato utilizzando una quantità limitata di casualità e leggendo un numero limitato di bit della dimostrazione. L'algoritmo quindi deve procedere ad accettare dimostrazioni corrette e rifiutare dimostrazioni sbagliate con una probabilità molto alta.

Osservazione 2.1 | Perché le PCP sono interessanti?

Le PCP sono interessanti perché esistono alcune PCP che possono essere verificate leggendo solo alcuni bit della dimostrazione, utilizzando la casualità in un modo fondamentale.

Definizione 2.2 | Dimostratore (Prover)

Dato una possibile soluzione x con lunghezza n a un problema L che potrebbe essere false, un **dimostratore** (prover) produce una dimostrazione π in cui si afferma che x risolve il problema L, cioè $x \in L$.

Osservazione 2.2 | Cosa è una dimostrazione per un problema L?

Dato un problema di decisione o linguaggio L con un alfabeto Σ, la dimostrazione realizzata da un dimostratore (prover) è una stringa di caratteri appartenenti all'insieme dato dalla chiusura di Kleene dell'alfabeto Σ^*.

Definizione 2.3 | Verificatore (Verifier)

Un **verificatore** (verifier) è una macchine di Turing V che svolge il ruolo di oracolo randomizzato: essa verifica la dimostrazione π prodotta da un **dimostratore** (prover) e decide se accettare l'affermazione connessa alla dimostrazione, che la soluzione x risolve il problema L (o equivalentemente $x \in L$).

Il verificatore è un algoritmo probabilistico che riceve in input x, una stringa w detta testimone (witness) e una sequenza di bit casuali R: V esegue quindi una computazione deterministica basata sull'input.

Proprietà 2.1 | Verificatore non-adattativo

Un verificatore è detto non-adattativo se esegue tutte le sue query prima di ricevere la risposta di una qualsiasi query precedente.

Proprietà 2.2 | Verificatore $(r(n), q(n))$-ristretto

Un verificatore è detto $(r(n), q(n))$-**ristretto** se, per ogni input x di lunghezza n e ogni w, $V^w(x)$ esegue al più $q(n)$ query su w e usa al più $r(n)$ bit casuali.

Definizione 2.4 | Sistema PCP

Dato un problema di decisione L (o un linguaggio L con un alfabeto Σ), un sistema PCP per L con completezza $c(n)$ e correttezza (soundness) $s(n)$, dove $0 \leqslant s(n) \leqslant c(n) \leqslant 1$, è composto da un **dimostratore** (prover) e un **verificatore** (verifier).

Proprietà 2.3 | Completezza

Per ogni $x \in L$, data la dimostrazione π prodotta dal **dimostratore** (prover) associato al sistema PCP, il **verificatore** (verifier) associato accetta l'affermazione prodotta dal **dimostratore** con probabilità almeno $c(n)$.

Proprietà 2.4 | Correttezza (Soundness)

Per ogni $x \notin L$, per ogni dimostrazione π prodotta dal **dimostratore** (prover) associato al sistema PCP, il **verificatore** (verifier) associato accetta erroneamente (falso positivo) l'affermazione prodotta dal **dimostratore** con probabilità al più $s(n)$.

Definizione 2.5 | Classe $\mathrm{PCPc}(n)$

La classe di complessità $\mathrm{PCPc}(n)$, $s(n)[r(n), q(n)]$ è la classe composta da tutti i problemi di decisione che hanno sistemi di dimostrazioni verificabili probabilisticamente (PCP) dopo il **verificatore è non-adattativo**, viene eseguito in **tempo polinomiale** e ha una complessità della casualità $r(n)$ e una complessità delle query $q(n)$: vale a dire il verificatore associato alla classe è $(r(n), q(n))$-ristretto.

Definizione 2.6 | Classe PCP

La classe di complessità $\mathrm{PCP}[r(n), q(n)]$ indica l'insieme dei problemi di decisione che hanno dimostrazioni verificabili probabilisticamente (PCP) che possono essere verificate in tempo polinomiale usando al più $r(n)$ bit casuali e leggendo al più $q(n)$ bit dalla dimostrazione. Tipicamente, le dimostrazioni corrette dovrebbero sempre essere accettate e quelle sbagliate dovrebbero essere rifiutate con probabilità maggiore di $\frac{1}{2}$.

È una notazione compatta per $\mathrm{PCP}(1), \frac{1}{2}[r(n), q(n)]$.

Definizione 2.7 | Classe PCP

La classe di complessità PCP è definita come $\mathrm{PCP}(1), \frac{1}{2}[O\log n, O(1)]$.

Osservazione 2.3 | P come sottoclasse di PCP

Possiamo scrivere la classe P come vari sottoclassi di complessità di $\mathrm{PCP}[r(n), q(n)]$, per esempio:

1. $\mathrm{PCP}[0,0]$: La sottoclasse definita come priva di casualità e senza accesso a una dimostrazione.
2. $\mathrm{PCP}[O(\log n), 0]$: La sottoclasse che utilizza un numero logaritmico di bit casuali (che non aiuta il **prover** comunque un algoritmo polinomiale può provare tutte le stringhe di lunghezza logaritmica in tempo polinomiale) e senza accesso a una dimostrazione.
3. $\mathrm{PCP}[0, O(\log n)]$: La sottoclasse definita come priva di casualità che produce dimostrazioni di lunghezza logaritmica, che essendo prive di casualità sono stringhe fisse di lunghezza logaritmica. Un algoritmo polinomiale può verificare tutte le stringhe di lunghezza logaritmica.

Definizione 2.8 | Derandomizzazione

Se $r(n) = O(\log(n))$, allora il processo di verifica della dimostrazione può essere **derandomizzato**, cioè il verificatore V può essere simulato in tempo polinomiale da un verificato deterministico che simula la computazione di V su ognuno dei $2^{r(n)} = n^{O(1)}$ possibili input e quindi computa la probabilità che $V^w(x)$ accetti l'esito della computazione. Il verificatore simulante accetta se e solo se la probabilità ottenuta è pari a 1.

Teorema 2.2 | Teorema del PCP

$$\mathrm{NP} = \mathrm{PCP}[O(\log n), O(1)]$$

La classe di complessità dei problemi non polinomiali coincide con la classe di problemi che utilizzano un numero logaritmico di bit casuali e produce dimostrazioni di lunghezza costante.

Teorema 2.3 | Max3SAT è inapprossimabile

Il teorema del PCP implica che non esiste un $\epsilon_1 > 0$ tale che non esiste un algoritmo $1+\epsilon_1$-approssimato in tempo polinomiale per **Max3SAT**, a meno che $\mathrm{P} = \mathrm{NP}$.

Dimostrazione 2.1 | Max3SAT è inapprossimabile

Sia $L \in \mathrm{PCP}[O(\log n), q]$ un problema NP-completo, dove q è una costante e sia V un verificatore per L $\big(O(\log n), q\big)$-ristretto. Procediamo a descrivere una riduzione da L a $Max3SAT$.

Data un'istanza x di L, per dimostrare la tesi del teorema procediamo a costruire una formula 3CNF (forma normale congiunta) φ_x con m clausole tale che per un qualche $\epsilon_1 > 0$ da determinare vale:

$$\begin{aligned} x \in L &\quad\Rightarrow\quad \varphi_x \text{ è soddisfacibile.} \\ x \notin L &\quad\Rightarrow\quad \text{nessun assegnamento soddisfa più di } (1-\epsilon_1)\, m \text{ clausole di } \varphi_x \end{aligned}$$

Procediamo a enumerare tutti gli input casuali R per il verificatore V. La lunghezza di ogni stringa è $r(|x|) = O(\log|x|)$, per cui il numero di queste stringhe è polinomiale in $|x|$. Per ogni R, il verificatore V sceglie q posizioni $i_1^R, \ldots, i_q^R$ e una funzione booleana $f_R : \{0,1\}^q \to \{0,1\}$ e accetta se e solo se $f_R\left(w_{i_1^R}, \ldots, w_{i_q^R}\right) = 1$.

Vogliamo simulare la possibile computazione del verificatore (per valori differenti di bit casuali di input R e diversi testimoni w).

Per ogni R aggiungiamo le clausole che rappresentano il vincolo $f_R\left(x_{i_1^R}, \ldots, x_{i_q^R}\right) = 1$. Questo può essere fatto con una forma normale congiunta di dimensione $\leqslant 2^q$: vale a dire che dovremmo aggiungere al più 2^q clausole se dovessimo scrivere una espressione in forma normale congiunta di q termini. Dobbiamo però anche convertire le clausole di lunghezza q a clausole di lunghezza 3, cosa che può essere realizzata introducendo delle variabili addizionali come nella riduzione standard da kSAT a 3SAT. Tutto considerato, questa trasformazione crea una formula φ_x con al più $q2^q$ clausole 3CNF.

Consideriamo la relazione tra l'ottimo di φ_z come un'istanza di **MaxE3SAT** e la domanda se $z \in L$. Consideriamo i due casi:

Se $z \in L$ Allora deve esiste un testimone w tale che $V^w(z)$ accetta per ogni R. Poniamo $x_i = w_i$ e le variabili ausiliarie di conseguenza, quindi l'assegnamento soddisfa tutte le clausole e φ_z è soddisfacibile.

Se $z \notin L$ Allora consideriamo un assegnamento arbitrario alla variabile x_i e le variabili ausiliare. Consideriamo la stringa w dove w_i è posta uguale a x_i. Il testimone w fa rifiutare al verificatore metà dei possibili $R \in \{0,1\}^{r(|z|)}$ e per ogni dato R, una delle clausole rappresentanti f_R fallisce. Tutto considerato, almeno una frazione $\varepsilon_1 = \frac{1}{2}\frac{1}{q2^q}$ delle clausole fallisce.

Teorema 2.4 | Riduzione verso PCP

Se esiste una riduzione verso **MaxE3SAT** per un problema $L \in \mathrm{NP}$, allora $L \in \mathrm{PCP}[O(\log n), O(1)]$. In particolare, se L è NP-completo allora vale il teorema del PCP.

Dimostrazione 2.2 | Riduzione verso PCP

Descriviamo come costruire un verificatore per L. Il verificatore V su input z si aspetta un testimone w per soddisfare l'assegnamento per φ_z. Il verificatore sceglie $O\left(\frac{1}{\varepsilon_1}\right)$ clausole a di φ_z a caso, e verifica che w le soddisfi tutte. Il numero di bit casuali usato dal verificatore è $O\left(\frac{1}{\varepsilon_1}\log m\right) = O(\log|z|)$. Il numero di bit del testimone che sono letti dal verificatore sono $O\left(\frac{1}{\varepsilon_1}\right) = O(1)$, per cui:

1. Se $z \in L$ allora φ_z è soddisfacibile e ne segue che esiste un testimone w tale che $V^w(z)$ accetti sempre.
2. Se $z \notin L$ allora per ogni testimone w, una frazione delle clausole di φ rimangono insoddisfatte da w, e quindi per ogni testimone w il verificatore $V^w(z)$ rifiuta l'affermazione con probabilità $\geqslant \frac{1}{2}$.

2.2 Da PCP a Indipendent Set

Definizione 2.9 | Configurazione

Sia L un problema NP-completo, V un verificatore che mostra che $L \in \mathrm{PCP}_{c,s}[q(n), r(n)]$. Per un input x, consideriamo tutte le possibili computazioni di $V^w(x)$ su tutte le possibili dimostrazioni w. Una descrizione completa delle computazioni di V è data dalla specifica della casualità usata da V, la lista delle query fatta da V nella dimostrazione e la lista delle risposte.

Per ogni input x fissato, ogni query è determinata da x, dalla casualità e dalle risposte precedenti: pertanto è sufficiente specificare la casualità e le risposte per poter completamente specificare una computazione.

Una tale descrizione è detta **configurazione**.

Osservazione 2.4 | Massimo numero di configurazioni

Il numero totali di **configurazioni** è al più pari a:

$$2^{r(n)} \cdot 2^{q(n)}$$

dove n è la lunghezza di x.

Proprietà 2.5 | Configurazione accettante

Una configurazione accettante è una configurazione il cui verificatore accetta l'affermazione $x \in L$.

Proprietà 2.6 | Configurazioni inconsistente

Due configurazioni c, c' sono dette **inconsistenti** se sia c che c' specificano una query nella stessa posizione e ricevono due risposte differenti alla stessa query.

Definizione 2.10 | Grafo di configurazioni

Per ogni input x definiamo un **grafo di configurazioni** G_x come un grafo che ha un vertice per ogni **configurazione accettante** di x e ha un lato tra due configurazioni se esse sono **inconsistenti**.

Affermazione 2.1 | 13

Se $x \in L$, allora il grafo delle configurazioni associato G_x ha un **insieme indipendente** di dimensione $\geqslant c \cdot 2^{r(n)}$.

Dimostrazione 2.3 | 13

Se $x \in L$, allora esiste un testimone w tale che $V^w(x)$ accetta l'affermazione con probabilità almeno c. Cioè esiste un testimone w tale che esistono almeno $c \cdot 2^{r(n)}$ input casuali tali che $V^w(x)$ accetti. Questo implica che esistono almeno $c \cdot 2^r(n)$ configurazioni mutualmente consistenti nel grafo associato G_x e queste formano un insieme indipendente.

Affermazione 2.2 | 14

Se $x \notin L$, allora ogni **insieme indipendente** del grafo delle configurazioni associato G_x ha dimensione $\geqslant s \cdot 2^{r(n)}$.

Dimostrazione 2.4 | 14

Procediamo per assurdo: assumiamo che esista un insieme indipendente nel grafo delle configurazioni associato G_x di dimensione $\geqslant s \cdot 2^{r(n)}$, e mostriamo che questo implichi che $x \in L$.

Iniziamo definendo un testimone w: per ogni configurazione nell'insieme indipendente, aggiorniamo i bit di w su cui è stata eseguita una query nella configurazione in base alle risposte nelle configurazioni. Andiamo a porre a zero i bit del testimone w su cui non è stata eseguita una query in nessuna configurazione dell'insieme indipendente.

Le $s \cdot 2^{r(n)}$ configurazione dell'insieme indipendente corrispondono ad un numero equivalente di stringhe casuali. Quando $V^w(x)$ sceglie una di queste stringhe casuali la accetta, e pertanto $V^w(x)$ accetta l'affermazione con probabilità almeno s, implicando che $x \in L$, da cui l'assurdo.

2.3 Approssimazione di MaxE3SAT

Problema 2.1 | MaxSAT

MaxSAT chiede di soddisfare il massimo numero di clausole di una formula in forma normale congiunta.

Problema 2.2 | MaxE3SAT

MaxE3SAT chiede di soddisfare il massimo numero di clausole booleane contenenti **esattamente** tre letterali.

Problema 2.3 | MaxEkSAT

MaxEkSAT chiede di soddisfare il massimo numero di clausole booleane contenenti **esattamente** k letterali. Si tratta di una generalizzazione di MaxSAT.

Osservazione 2.5 | Quante clausole sono soddisfacibili dall'approssimazione di MaxE3SAT?

L'approssimazione di **MaxE3SAT** consente di soddisfare in tempo polinomiale almeno $\frac{7}{8}$ delle clausole di un'istanza di **MaxE3SAT**, o più in generale soddisfare $\left(2^k-1\right)/2^k$ clausole di **MaxEkSAT**. Con la stessa tecnica si possono soddisfare $\frac{1}{2}$ delle clausole di un'istanza di **MaxSAT**.

Osservazione 2.6 | Quale è l'idea di base dell'approssimazione di MaxE3SAT?

Si procede a **derandomizzare** un algoritmo randomizzato: l'algoritmo estrae semplicemente un assegnamento a caso.

Dimostrazione 2.5 | Approssimazione di MaxE3SAT

Se le clausole sono composte da 3 letterali, una sola combinazione di letterali su 2^3 può rendere la clausola falsa, quindi estraendo un assegnamento a caso uniformemente, la probabilità che una specifica clausola sia vera è $\frac{7}{8}$. Se denotiamo con $f_i : 2^3 \to 2$ la funzione di verità a tre argomento della clausola i, dove $i \in \{0,\dots,n\}$ ed n è il numero delle clausole, possiamo esprimere il numero di clausole soddisfatte da un certo assegnamento alle variabili della formula come la somma delle f_i, valutate sui valori assegnati alle variabili della clausola i.

La somma delle f_i è quindi una funzione $C : 2^k \to \mathbb{N}$, dove k è il numero delle variabili, che associa a ogni possibile assegnamento il numero di clausole soddisfatte. Se denotiamo con X_i una variabile aleatoria che assume valore 0 o 1 equiprobabilmente, $C(X_0, X_1, \dots, X_{k-1})$ è una variabile aleatoria che ha come valore, per ogni assegnamento possibile, il numero di clausole soddisfate.

Possiamo facilmente calcolare il valore atteso di $C(X_0, X_1, \dots, X_{k-1})$, che **per linearità** coincide con la somma dei valori attesi delle f_i. Ogni f_i assume valore 0 con probabilità $\frac{1}{8}$ e valore 1 con probabilità $\frac{7}{8}$: il valore atteso di $C(X_0, X_1, \dots, X_{k-1})$ risulta pertanto $\frac{7}{8}n$. Il risultato ottenuti ci fornisce un algoritmo randomizzato che ha come valore atteso del risultato l'approssimazione richiesta.

Per **derandomizzare** l'algoritmo utilizziamo il **metodo delle probabilità condizionate**. Supponiamo di aver dimostrato che:

$$\mathbf{E}\left[C(X_0, X_1, \dots, X_{k-1}) \mid X_0 = b_0, X_1 = b_1, \dots X_{j-1} = b_{j-1}\right] \geqslant \frac{7}{8}n$$

per un $0 \leqslant j < k$ e $b_i \in \{1,2\}$. Per $j = 0$ non c'è condizionamento e l'asserto è quello che abbiamo dimostrato. Per definizione di valore atteso condizionato abbiamo che:

$$\begin{aligned}
&\tfrac{7}{8}n \leqslant \mathbf{E}\left[C(X_0, X_1, \dots, X_{k-1}) \mid X_0 = b_0, X_1 = b_1, \dots X_{j-1} = b_{j-1}\right] \\
&= \tfrac{1}{2}\mathbf{E}\left[C(X_0, X_1, \dots, X_{k-1}) \mid X_0 = b_0, X_1 = b_1, \dots X_{j-1} = b_{j-1}, X_j = 0\right] \\
&+ \tfrac{1}{2}\mathbf{E}\left[C(X_0, X_1, \dots, X_{k-1}) \mid X_0 = b_0, X_1 = b_1, \dots X_{j-1} = b_{j-1}, X_j = 1\right]
\end{aligned}$$

Deve esserci quindi un $b_j \in \{1,2\}$ tale che:

$$\mathbf{E}\left[C(X_0, X_1, \dots, X_{k-1}) \mid X_0 = b_0, X_1 = b_1, \dots X_{j-1} = b_{j-1}, X_j = b_j\right] \geqslant \frac{7}{8}n$$

Per ogni clausola, valutiamo i letterali già assegnati. Avremo alcune clausole con valore costante (che avranno valore atteso zero o uno), altre clausole con un letterale rimasto (quindi valore atteso $\frac{1}{2}$), clausole con due letterali rimasti (valore atteso $\frac{3}{4}$) e clausole con tre letterali rimasti (Con valore atteso $\frac{7}{8}$). Sommando i valori attesi, otteniamo il valore atteso condizionato di $C(X_0, X_1, \dots, X_{k-1})$ che ci permette di scegliere b_j.

Quando $j = k-1$ otteniamo un assegnamento $b_0, b_1, \dots, b_{k-1}$ che soddisfa almeno $\frac{7}{8}$ delle clausole. Nel caso di **MaxSAT** possiamo solo garantire la soddisfacibilità di metà delle clausole perché le funzioni di verità delle clausole formate da un solo letterale hanno valore atteso $\frac{1}{2}$ e quindi possiamo solo dire che il valore atteso $C(X_0, X_1, \dots, X_{k-1})$ è maggiore o uguale a $\frac{n}{2}$.

2.4 Arrotondamento aleatorio per Set Cover

Osservazione 2.7 | A cosa serve l'arrotondamento aleatorio?

L'**arrotondamento aleatorio** (randomized rounding) è una tecnica che può essere utilizzata per ottenere soluzioni discrete a partire da un rilassamento lineare.

Definizione 2.11 | Arrotondamento aleatorio per Set Cover

Utilizziamo la seguente **procedura di scelta**: per ogni i, inseriamo l'indice i nella soluzione I con probabilità $\overline{x}_i$, dove $\overline{x}_i$ sono indicatori binari dell'appartenenza dell'insieme S_i alla soluzione.

La procedura di scelta così descritta lascerà molto probabilmente degli elementi di U scoperti: possiamo però controllare la probabilità che uno specifico elemento v rimanga scoperto:

$$\prod_{v \in S_i} (1 - x_i) \leqslant \prod_{v \in S_i} e^{-x_i} \leqslant e^{-\sum_{v \in S_i} x_i} \leqslant e^{-1}$$

Ne consegue che se ripetiamo la procedura $k + \ln n$ volte e consideriamo l'unione degli insieme soluzione così generati, la probabilità che un elemento non sia coperto è controllata da:

$$\left(\frac{1}{e}\right)^{k+\ln n} = \frac{1}{n} e^{-k}$$

e utilizzando il limite per l'unione, la probabilità che ci sia un qualche elemento scoperto è meno di e^{-k}.

Inoltre, se iteriamo la procedura di scelta $k + \ln n$ volte, il valore atteso della funzione obiettivo sull'insieme unione di tutte le $k + \ln n$ scelte generate è minore o uguale a $(k + \ln n)\overline{c}$ per **linearità**, e per la disuguaglianza di Markov sarà maggiore o uguale a $q(k + \ln n)\overline{c}$ con probabilità al più $\frac{1}{q}$ per ogni $q > 1$.

Rimane quindi solo da tarare il valore di k. Se scegliamo per esempio $k = 3$, la probabilità che non copriamo tutti gli elementi è meno di $e^{-3} \leqslant 0.05$, mentre la probabilità di ottenere una soluzione maggiore di $2(3 + \ln n)\overline{c}$ è minore di $\frac{1}{2}$. Messi insieme i due eventi, abbiamo probabilità $\geqslant 0.45$ che l'algoritmo copra tutti gli elementi con un rapporto di prestazioni al più $6 + 2\ln n$.

2.5 Algoritmo di Miller-Rabin

Osservazione 2.8 | A cosa serve l'algoritmo di Miller-Rabin?

L'algoritmo di Miller-Rabin utilizza dei bit aleatori per stabilire se un numero è composto. Se la risposta è positiva questa è sempre corretta, se la risposta è negative (cioè il numero sarebbe considerato primo) c'è una probabilità di errore inferiore a $\frac{1}{2}$.

La probabilità d'errore è sui bit aleatori, non sull'input.

Definizione 2.12 | Anello delle classi di resto modulo n

Denotiamo con $\mathbb{Z}_n$ l'**anello delle classi di resto modulo** n, una struttura algebrica composta dall'insieme di numeri congrui in modulo n su cui sono definite due operazioni binarie, chiamate somma e prodotto.

Definizione 2.13 | Gruppo delle unità

Denotiamo con $\mathbb{U}_n$ il **gruppo delle unità** di un **anello delle classi di resto modulo** n $\mathbb{Z}_n$, cioè il suo gruppo moltiplicativo formato dagli elementi diversi da zero e invertibili.

Definizione 2.14 | Testimone di composizione

Sia n un numero dispari, e $n-1=2^a b$, con b dispari. Dato $t \in \mathbb{Z}_n \setminus \{0,1\}$ consideriamo la sequenza degli a quadrati successivi a partire da b:

$$t^b, t^{2b}, t^{2^2 b}, t^{2^3 b}, \ldots, t^{n-1}$$

Diciamo che t è un **testimone di composizione di** n se $t \not\perp n$, se $t^{n-1} \neq 1$ o se esiste un i tale per cui $t^{2ib} \neq \pm 1$ e $t^{2^{i+1}b} = 1$.

Si tratta di un elemento di $\mathbb{Z}_n$ che ci mostra che n ha un divisore proprio, che $n-1$ non è un multiplo dell'ordine di $\mathbb{U}_n$ o che esistono radici quadrate dell'unità diverse da ± 1. In tutti e tre i casi n non può essere primo.

Teorema 2.5 | Teorema cinese del resto

Se $c \perp d$, allora:

$$\mathbf{Z}_{cd} \cong \mathbf{Z}_c \times \mathbf{Z}_d$$

Teorema 2.6 | Teorema di Miller

Se n è primo, non ha testimoni. Se n è un numero dispari composto che non è una potenza di primo, almeno metà dei $t \in \mathbb{Z}_n \setminus \{0\}$ sono testimoni.

Definizione 2.15 | Algoritmo di Miller-Rabin

Dato un numero intero n:

1. Se n è pari è composto.
2. Altrimenti, per bisezione controlliamo se n è una potenza (basta controllare gli esponenti da 2 a $\log n$, dobbiamo quindi operare $\log n$ ricerche da $\log n$ passi, e ogni passo richiede solo un'esponenziazione), e in caso positivo concludiamo che n è composto.
3. Estraiamo in maniera uniforme un numero casuale $t \in \mathbb{Z}_n \setminus \{0\}$
4. Se $t \not\perp n$ concludiamo che n è composto.
5. Se $t^{n-1} \neq 1$ concludiamo che n è composto.
6. Infine, calcoliamo a e b tali che $n-1=2^a b$ con b dispari e costruiamo la sequenza $t^b, t^{2b}, t^{2^2 b}, \ldots, t^{n-1}$. Se nella sequenza compare una radice dell'unità diversa da ± 1 concludiamo che n è composto.
7. Se n passa tutti i test precedenti concludiamo che è primo.

Ripetendo l'algoritmo k volte con esito negativo, la possibilità di errore nello stabilire che n è primo è al più 2^{-k}.

2.6 Inapprossimabilità del TSP

Dimostrazione 2.6 | Il problema del commesso viaggiatore è NP-completo

Partiamo dal fatto che stabilire se un grafo non orientato $G = \langle V, E \rangle$ possiede un ciclo Hamiltoniano è NP-completo. Mostriamo ora come assumere l'esistenza di un algoritmo polinomiale r-approssimante per il commesso viaggiatore permetterebbe di riconoscere in tempo polinomiale l'esistenza di un ciclo Hamiltoniano.

A partire da G, costruiamo un'istanza G' del commesso viaggiatore con insieme di vertici V in cui:

$$d(x,y) = \begin{cases} 1 & \text{se } \langle x, y \rangle \in E \\ \lceil nr \rceil + 1 & \text{se } \langle x, y \rangle \notin E \end{cases}$$

dove $n = |V|$. Ne segue che G ha un ciclo Hamiltoniano se e solo se G' ha un ciclo Hamiltoniano di peso n. Solo tale ciclo, infatti, usa solo lati di G. Qualunque ciclo Hamiltoniano di G' che utilizza un lato non in G ha peso almeno $\lceil nr \rceil + 1$.

Se potessimo approssimare il commesso viaggiatore meglio di $(\lceil nr \rceil + 1)/n > r$ potremmo decidere se il grafo G possiede un ciclo Hamiltoniano in tempo polinomiale.

2.7 NP-optimization problem (NPO)

Definizione 2.16 | Classe di problemi NPO

NPO è la classe dei problemi di ottimizzazione i cui associati problemi decisione, per qualunque costante, sono in **NP**. La soluzione di un problema in **NPO** è un algoritmo che a fronte di un'istanza x trova una soluzione y ottima, cioè con costo:

$$c_A(x,y) = t\{c_A(x,z) | z \text{ soluzione di } x\}$$

Scriveremo $\text{opt}_A(x)$ il costo di una soluzione ottima per l'istanza x di A.

Definizione 2.17 | NP-optimization problem (NPO)

Un problema A di ottimizzazione in NPO è dato da:

1. Un insieme di **istanze** riconoscibili in tempo polinomiale.
2. Per ogni istanza x, un insieme di soluzioni (accettabili), che devono essere corte (esiste un polinomio p tale che $|y| \leq p(|x|)$ per ogni istanza x e per ogni soluzione y di x) e riconoscibili in tempo polinomiale.
3. Una funzione c_A, calcolabile in tempo polinomiale, che data un'istanza x e una sua soluzione y restituisce il **costo** (intero) di y.
4. Un tipo $t \in \{\max, \min\}$

Definizione 2.18 | Rapporto di prestazioni NPO

Per ogni soluzione y di un'istanza x di un problema A in **NPO** esiste un **rapporto di prestazioni**:

$$R_A(x,y) = \max\left\{\frac{c_A(x,y)}{\text{opt}_A(x)}, \frac{\text{opt}_A(x)}{c_A(x,y)}\right\}$$

che esprime la bontà della soluzione come un numero nell'intervallo $[1 \ldots \infty)$ **indipendentemente** dal tipo di problema (minimizzazione / massimizzazione).

Definizione 2.19 | APX

APX è la sottoclasse di problemi **NPO** che ammettono un algoritmo di approssimazione con rapporto di prestazioni controllato da una costante. Esiste cioè una costante r, indipendente dall'istanza x e dalla soluzione y, tale che $R_A(x,y) < r$.

Osservazione 2.9 | APX $\neq$ NPO

MaxE3SAT è un problema della classe APX, dato che il suo rapporto di prestazioni è minore o uguale a $\frac{8}{7}$, mentre **MaxDisjointPaths** non lo è (dato che ci sono istanze le cui soluzioni hanno rapporto di prestazioni $\Omega(\sqrt{m})$).

Definizione 2.20 | PTAS

PTAS è la sottoclasse di problemi di **APX** che ammettono uno schema di approssimazione polinomiale: esiste cioè un algoritmo che dato r e un'istanza x calcola in tempo polinomiale in $|x|$ (ma non necessariamente in r) una soluzione.

Osservazione 2.10 | PTAS $\neq$ APX

Il problema dello zaino è in **PTAS**, mentre il problema della **Center selection** è in **APX** ma non in **PTAS** (dato che non esistono algoritmi polinomiali con rapporto di prestazione inferiore a 2).

2.8 Problemi di gap con promessa

Definizione 2.21 | Problema di decisione con promessa

Un **problema di decisione con promessa** L è specificato da due insiemi $L_{\text{yes}}, L_{\text{no}} \subseteq 2^*$ che soddisfano $L_{\text{yes}} \cap L_{\text{no}} = \emptyset$. La differenza fondamentale con un normale problema di decisone è che non necessariamente tutte le stringhe in 2^* sono istanze si o istanze no: possono esistere in generale stringhe che non appartengono né a L_{yes}, né a L_{no}, e nella definizione delle classi noi teniamo conto solo delle stringhe $L_{\text{yes}} \cup L_{\text{no}}$.

Intuitivamente **promettiamo** alla macchina di Turing che deve riconoscere L di dare in input solo stringhe che sono istanze. Ogni problema di decisione standard è semplicemente un problema con promessa vuota.

Osservazione 2.11 | A cosa servono i problemi con promessa?

I problemi con promessa permettono di codificare in maniera naturale i risultati di inapprossimabilità tramite gap.

Esempio 2.1 | Problema con promessa applicato a Center Selection

Sappiamo che non è possibile distinguere certe istanze di **Center Selection** che hanno soluzione ottima 1 o 2, a meno che $\text{P} = \text{NP}$. Questo è traducibile nel fatto che il problema con promessa che ha come istanze-sì le istanze con soluzione ottima 1 e ha come istanze-no le istanze con soluzione 2 è NP-difficile.

Definizione 2.22 | Riduzione

Dati due problemi con promessa L e M, una **riduzione** da L a M è una funzione $f : 2^* \to 2^*$, calcolabile in tempo polinomiale, tale che:

$$x \in L_{\text{yes}} \implies f(x) \in M_{\text{yes}}$$
$$x \in L_{\text{no}} \implies f(x) \in M_{\text{no}}$$

La funzione f mappa istanze-si di L a istanze-si di M e istanze-no di L a istanze-no di M.

Osservazione 2.12 | A cosa servono le riduzioni?

Se la riduzione M è trattabile anche L è trattabile, quindi se L è difficile anche la riduzione M è difficile. Possiamo quindi propagare risultati di difficoltà di approssimazione verso nuovi problemi costruendo un problema di gap con promessa opportuno, e una riduzione opportuna.

Esempio 2.2 | Ridurre MaxE3SAT a Maximum Indipendent Set

Un problema di gap con promessa di **MaxE3SAT** si può ridurre a un problema di gap con promessa equivalente di **Maximum Indipendent Set**.

Utilizzando la riduzione da **3SAT** a **Indipendent Set**, che associa a ogni clausola con k letterali una k-cricca i cui vertici sono decorati con i rispettivi letterali, e connette vertici di cricche diverse associati a letterali opposti, è possibile vedere che il numero di clausole soddisfatte di un'istanza di **MaxE3SAT** è esattamente la dimensione del massimo insieme indipendente del grafo associato. Dato che le clausole di **MaxE3SAT** contengono esattamente tre letterali, il grafo associato a una formula con n clausole ha esattamente $3n$ vertici.

Se consideriamo un problema di gap con promessa che chiede di distinguere formule di **MaxE3SAT** soddisfacibili da formule con al più $1-\epsilon$ clausole soddisfacibili (cioè dove esiste un gap $\frac{1}{(1-\epsilon)}$), vediamo subito che le formule vengono mappate in grafi con $3n$ vertici, dove n è il numero di vertici della formula.

Nel caso soddisfacibile, il grafo ha un insieme indipendente di dimensione esattamente n, mentre nel secondo caso non può esistere un insieme indipendente di dimensione superiore a $(1-\epsilon)\,n$. Il gap associato a questa promessa è quindi di nuovo $\frac{1}{1-\epsilon}$, e non è quindi possibile approssimare meglio **Maximum Indipendent Set**. Utilizzando il risultato ottimo di Håstad $\varepsilon = \frac{1}{8}$ abbiamo che il **Maximum Indipendent Set** non è approssimabile meglio di $\frac{8}{7}$, esattamente come **MaxE3SAT**.

Esempio 2.3 | Ridurre Maximum Indipendent Set a Minimum Vertex Cover

Il problema di gap con promessa di **Maximum Indipendent Set** appena descritto può essere ridotto a un problema di gap con promessa per **Minimum Vertex Cover**.

Dato che il complemento di un insieme indipendente è esattamente un insieme di vertici di copertura, la promessa diventa la seguente: si considerano grafi di dimensione $3n$ per qualche $n \geq 1$, e si sa che esiste un insieme di vertici di copertura più piccolo di $3n - n = 2n$, oppure uno maggiore di $3n - (1-\varepsilon)n = 2n + \varepsilon n$. Il gap associato è quindi $(2+\varepsilon)n/2n = 1 + \varepsilon/2$. Dato che $\epsilon < 1$ vale che:

$$1 + \frac{\varepsilon}{2} < 1 + \frac{\varepsilon}{1-\varepsilon} = \frac{1}{1-\varepsilon}$$

Utilizzando il risultato ottimo di Håstad $\epsilon = \frac{1}{8}$ vale che **Minimum Vertex Cover** non è approssimabile meglio di $1 + \frac{1}{16} \approx 1.06$.

2.9 Taglio minimo

Problema 2.4 | Taglio minimo

Il **taglio minimo** di un grafo è una partizione dei vertici di un grafo in due sottoinsiemi disgiunti che risulta minimale in qualche senso: nel caso di grafi pesati può essere il totale del peso degli archi tagliati, e in questo caso può essere risolto in tempo polinomiale dall'algoritmo di Stoer-Wagner. Nel caso in cui i lati non sono pesati, l'algoritmo di Karger fornisce un'approssimazione efficiente per identificare il taglio minimo.

Definizione 2.23 | Algoritmo di Karger

L'algoritmo di Karger iterativamente sceglie un lato a caso e contrae i nodi connessi in un solo nodo: i lati connessi a uno qualsiasi dei due nodi vengono ora considerati connessi al nuovo nodo. L'algoritmo itera fino a che non rimangono solo due nodi: questi rappresentano un taglio del grafo originale.

L'algoritmo viene iterato un numero sufficiente di volte tale che un taglio minimo può essere identificato con alta probabilità.

Analisi 2.1 | Analisi dell'algoritmo di Karger

In un grafo $G=(V,E)$, con $n=|V|$ vertici, l'algoritmi di Karger ritorna un taglio minimo con probabilità $\binom{n}{2}^{-1}$.

Ogni grafo ha $2^{n-1}-1$ tagli, tra i quali al più $\binom{n}{2}$ sono minimi. La probabilità di successo di questo algoritmo è molto migliore della probabilità di identificare un taglio minimo a caso, che è:

$$\binom{n}{2} / \left(2^{n-1}-1\right)$$

Osservazione 2.13 | Quante volte va ripetuto l'algoritmo di Karger?

Ripetendo l'algoritmo di Karger per $T=\binom{n}{2}\ln n$ volte con scelte indipendenti randomiche e ritornando il taglio minimo trovato, la probabilità di non trovare il taglio minimo è:

$$\left[1-\binom{n}{2}^{-1}\right]^{T} \leqslant \frac{1}{e^{\ln n}}=\frac{1}{n}$$

Complessità 2.1 | Complessità di Karger

Il tempo totale per eseguire T ripetizioni per un grafo con n vertici e m lati è:

$$O(Tm)=O\left(n^2 m \log n\right)$$

Analisi 2.2 | Applicazione dell'algoritmo di Karger a un grafo ciclico

In un grafo ciclico di n vertici ha esattamente $\binom{n}{2}$ tagli minimi. L'algoritmo di Karger identifica ognuno di questi tagli con pari probabilità.

Per determinare un massimale della probabilità di successo in generale, sia C l'insieme degli archi di un taglio di dimensione k. Alla prima contrazione, la probabilità che non venga coinvolto un lato appartenente a C è $1-\frac{k}{|E|}$. Il grado minimo di G è almeno k, per cui $|E| \geqslant \frac{nk}{2}$. La probabilità che l'algoritmo, ad ogni data iterazione, scelga un nodo da C è:

$$\frac{k}{|E|} \leqslant \frac{k}{nk/2}=\frac{2}{n}$$

La probabilità p_n che l'algoritmo su un grafo con n vertici ritorni C vale:

$$p_n \geqslant \prod_{i=0}^{n-3}\left(1-\frac{2}{n-i}\right)=\prod_{i=0}^{n-3}\frac{n-i-2}{n-i}$$

$$=\frac{n-2}{n}\cdot\frac{n-3}{n-1}\cdot\frac{n-4}{n-2}\cdots\frac{2}{4}\cdot\frac{1}{3}=\binom{n}{2}^{-1}$$

3

Strutture succinte

3.1 Rango e selezione

Definizione 3.1 | Strutture sistematiche

Le strutture succinte si basano su una ser ie di primitive di base che forniscono informazioni relative a un vettore $\underline{\boldsymbol{b}}$ di n bit utilizzando spazio addizionale $o(n)$.

Tali strutture vengono chiamate **sistematiche**, per distinguerle da quelle **non sistematiche** che invece rappresentano internamente il vettore.

Definizione 3.2 | Rango

L'operatore di **rango** (**rank**) è definito come segue:

$$\text{rank}_b(p) = \left|\{i \in p | b_i = 1\}\right| = \sum_{0 \leqslant i < p} b_i$$

Il rango alla posizione p è il numero di 1 che precedono la posizione p. Se $\underline{\boldsymbol{b}}$ è pensato come la rappresentazione di un sottoinsieme X di n, $\text{rank}_b(p)$ è semplicemente la cardinalità del sottoinsieme di X formato dagli elementi minori di p. Inoltre, $\text{rank}_b(n)$ è semplicemente il numero di bit a uno nel vettore.

Definizione 3.3 | Selezione

L'operazione duale al *rango* è la **selezione** (**select**), definita da:

$$\text{select}_b(k) = \max\{p | \text{rank}_b(p) \leqslant k\}$$

dove $0 \leqslant k < \text{rank}_b(n)$. Detto altrimenti $\text{select}_b(k)$ è la posizione del k-esimo bit a uno nel vettore (dove il primo uno ha indice zero). Se $\underline{\boldsymbol{b}}$ è pensato come la rappresentazione di un sottoinsieme X di n, $\text{select}_b(k)$ è semplicemente il k-esimo elemento di X.

Osservazione 3.1 | Cosa succede concatenando rank e select?

Concatenando le operazioni di **rango** e **selezione** si ottiene:

$$\begin{aligned}
&\text{rank}_b\left(\text{select}_b(k)\right) = k \\
&\text{select}_b\left(\text{rank}_b(p)\right) > p \text{ se } b_p = 0 \\
&\text{select}_b\left(\text{rank}_b(p)\right) = p \text{ se } b_p = 1
\end{aligned}$$

In generale applicare il rango seguito dalla selezione ha l'effetto di calcolare il **successore** di un elemento, dove il successore in $X \subseteq n$ di un elemento $x \in n$ è il minimo $y \in X$ tale che $y \geqslant x$.

Definizione 3.4 | Metodo dei quattro russi

Il metodo dei quattro russi, in generale, prevede di memorizzare in una tabella i risultati per piccoli blocchi dell'input (per esempio le risposte per sotto-matrici più piccole di un problema matriciale).

Definizione 3.5 | Struttura per rango di Jacobson

La struttura per rango di Jacobson consente di determinare il rango in tempo costante ed utilizza il **metodo dei quattro russi**. La struttura prevede di dividere gli n bit del vettore in blocchi di lunghezza $\frac{1}{2}\log n$ e in **superblocchi** di lunghezza $(\log n)^2$. Si noti che una tabella che contiene tutte le risposte possibili per i blocchi è formata da $2^{\frac{1}{2}\log n} \cdot \frac{1}{2}\log n \cdot \log\log n = O(\sqrt{n}\log n \log\log n) = o(n)$ bit bit Per ogni superblocco, memorizziamo il rango alla posizione iniziale del superblocco stesso e qesto richiede $\log n \cdot n/(\log n)^2 = n/\log n = o(n)$ bit. Infine, per ogni blocco memorizziamo il delta di rango rispetto al superblocco, cioè rango alla posizione iniziale del blocco meno il rango alla posizione iniziale del superblocco che lo contiene (il numero di 1 che intercorrono tra l'inizio del superblocco e l'inizio del blocco). Questo numero è minore di $(\log n)^2$, e quindi ha bisogno di $2\log\log n$ bit per essere scritto. Ma i blocchi sono $O(n/\log n)$, per cui i dati dei blocchi usano $O(n\log\log n/\log n) = o(n)$ bit.

Definizione 3.6 | Operazione di rango sulle struttura di Jacobson

Per calcolare il rango in posizione p, recuperiamo il rango all'inizio del superblocco in cui si trova p, sommando il delta di rango del blocco in cui si trova p abbiamo il numero di 1 che precedono l'inizio del blocco che contiene p. A questo punto estraiamo in tempo costante dalla tabella il numero di 1 tra l'inizio del blocco e p, lo sommiamo al numero di 1 fino all'inizio del blocco e restituiamo il valore così calcolato. Abbiamo effettuato un numero costante di operazioni.

Osservazione 3.2 | Le tabelle precalcolate possono essere pesanti!

Sebbene l'occupazione in spazio della struttura sia in teoria asintoticamente evanescente, per valori anche significativi di n è molto grande: per esempio se $n = 10^6$:

$$2^{\frac{1}{2}\log n} \cdot \frac{1}{2}\log n \cdot \log\log n \approx 10^3 \frac{1}{2} \cdot 20 \cdot 5 = 50000$$

e quindi le tabelle precalcolate richiedono il 5% di spazio in più. Se aggiungiamo anche i contatori dei blocchi, abbiamo bisogno di altri $2\log\log n \cdot n/\log n \approx 2 \cdot 5 \cdot 10^6/20 = 500000$ bit, cioè il 50% di spazio in più.

3.2 Alberi binari

Definizione 3.7 | Alberi binari

Un **albero binario** è o un albero vuoto o una coppia $\langle S, D\rangle$ di alberi binari, nel qual caso S è detto sotto-albero **sinistro** e D sottoalbero **destro**.

I nodi di un albero binario hanno o 0 o 2 figli.

Definizione 3.8 | Alberi binari a livelli

Un albero binario a livelli è un albero binario rappresentato da un vettore $\underline{\boldsymbol{b}}$ di bit per **livelli**: si parte dalla radice, si visita l'albero per livelli e si scrive un 1 per ogni nodo interno e uno 0 per ogni nodo esterno. In tutto scriviamo $2n+1$ bit. Il numero di alberi binari con n nodi interni è il **numero di Catalan**:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

definito da $C_0 = 1$ e:

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$$

relazione da cui si ricava il rapporto con il conteggio degli alberi binari. Si noti che utilizzando l'approssimazione di Stirling:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{1}{n+1}\frac{(2n)!}{(n!)^2} \sim \frac{1}{n+1}\frac{\left(\frac{2n}{e}\right)^{2n}\sqrt{2\pi 2n}}{\left(\frac{n}{e}\right)^{2n} 2\pi n} \sim \frac{4^n}{\sqrt{\pi n^3}}$$

da cui segue che $C_n = 2n - O(\log n)$. Quindi, se con $o(n)$ bit addizionali siamo in grado di calcolare il genitore e il figlio di ogni nodo avremo ottenuto una struttura dati succinta.

Dato un nodo in posizione p, è possibile determinare se è interno o esterno guardando se b_p risulta pari a uno o zero.

Se il nodo è interno, il suo primo figlio deve comparire dopo tutti i nodi interni che lo precedono, che sono esattamente $\mathrm{rank}_b(p)$, e dopo tutti i loro figli, che sono $\mathrm{rank}_b(p)+1$: il figlio sinistro ha dunque indice $2\,\mathrm{rank}_b(p)+1$ (mentre quello destro è una posizione dopo).

Se vogliamo calcolare il genitore di un nodo in posizione p, questo è il nodo interno di rango $\lfloor (p-1)/2 \rfloor$. Supponiamo che il nodo in posizione p sia un figlio sinistro. Il padre è il nodo in posizione q tale che $2\,\mathrm{rank}_b(q)+1 = p$, cioè $\mathrm{rank}_b(q) = (p-1)/2$.

Ma dato che $b_q = 1, q = \mathrm{select}_b\big(\mathrm{rank}_b(q)\big) = \mathrm{select}_b((p-1)/2)$.

Un calcolo analogo per il figlio desro mostra che, alla fine, il padre del nodo in posizione p ha posizione $\mathrm{select}_b(\lfloor (p-1)/2 \rfloor)$.

Si noti che se è necessario associare dati ancillari a ogni nodo (interno ed esterno), questo è possibile utilizzando semplicemente un vettore di lunghezza $2n+1$. Se invece si desidera associare dati a nodi interni, basta un vettore di lunghezza n che verrà indicizzato da $\mathrm{rank}_b(p)$, dove p è l'indice di un nodo interno.

3.3 Rappresentazione di Elias-Fano

Definizione 3.9 | Rappresentazione di Elias-Fano

Data una sequenza monotona di numeri non negativi:

$$0 \leqslant x_0 \leqslant x_1 \leqslant \cdots \leqslant x_{n-2} \leqslant x_{n-1} < u$$

dove $u > 0$ è un qualunque limite superiore per l'ultimo valore. La scelta $u = x_{n-1} + 1$ è naturalmente possibile (e ottima) ma può essere costoso e un valore ragionevole per u può essere disponibile tramite informazioni esterne.

Rappresenteremo la sequenza in due vettori di bit:

1. Gli $\ell = \max\{0, \lfloor \log(u/n) \rfloor\}$ bit inferiori di ogni x_i sono memorizzati esplicitamente in maniera contigua nel **vettore dei bit inferiori**.
2. I bit superiori sono memorizzati nel **vettore dei bit superiori** come sequenza di scarti codificati in unario.

Proprietà 3.1 | Bit per elemento della rappresentazione di Elias-Fano

La rappresentazione di Elias-Fano utilizza al più $2 + \lceil \log(u/n) \rceil$ bit per elemento: si può facilmente vedere dal fatto che ogni codice unario usa un bit di stop e per ogni altro bit scritto aumenta il valore dei bit superiori di 2^ℓ: questo non può succedere più di $\lfloor x_{n-1}/2^\ell \rfloor$ volte. Inoltre:

$$\left\lfloor \frac{x_{n-1}}{2^\ell} \right\rfloor \leqslant \left\lfloor \frac{u}{2^\ell} \right\rfloor \leqslant \frac{u}{2^\ell} = \frac{u}{2^{\max\{0, \lfloor \log(u/n) \rfloor\}}} \leqslant 2n$$

quindi, scriviamo al più n 1 e $2n$ zeri, il che implica quanto affermato, dato che $\lceil \log(u/n) \rceil = \lfloor \log(u/n) \rfloor + 1$ a meno che $\frac{u}{n}$ sia una potenza di due, ma in quel caso la disequazione finirebbe con $\leqslant n$, e quindi l'affermazione sarebbe ancora vera.

Dimostrazione 3.1 | La rappresentazione di Elias-Fano è quasi succinta

Il numero di successioni monotone di n elementi in un universo di u elementi è:

$$\binom{u+n-1}{u-1} = \binom{u+n-1}{n}$$

Una tale successione monotona è equivalente a un multi-insieme di cardinalità n su u elementi. Tali insiemi sono in biiezione con le soluzioni non-negative dell'equazione:

$$x_0 + x_1 + \cdots + x_{u-1} = n$$

Le soluzioni possono essere espresse come la composizione di $u-1$ segni + in mezzo a n segni •. Con l'assunzione $n \leqslant \sqrt{u}$ abbiamo allora che:

$$\begin{aligned}
\left\lceil \log \binom{u+n-1}{n} \right\rceil &\approx n \log\left(\frac{u+n-1}{n}\right) \\
&= n \log\left(\frac{u}{n}\left(1 + \frac{n}{u} - \frac{1}{u}\right)\right) \\
&= n \log\left(\frac{u}{n}\right) + n \log\left(1 + \frac{n}{u} - \frac{1}{u}\right) \\
&\approx n \log\left(\frac{u}{n}\right) + \frac{n^2}{u}
\end{aligned}$$

La rappresentazione è molto vicina a essere succinta quando il vettore è sufficientemente sparso siccome il secondo termine tende a 0.

Definizione 3.10 | Selezione su rappresentazione di Elias-Fano

Per ottenere x_i, effettuiamo una selezione dell'i-esimo bit sul vettore dei bit superiori, ottenendo la posizione p: il valore dei bit superiori di x_i è allora esattamente $p-i$. Gli ℓ bit inferiori possono essere estratti con un accesso diretto, dal momento che sono memorizzati in posizione $i\ell$ nel vettore dei bit inferiori.

Osservazione 3.3 | Elias-Fano come struttura non sistematica

La struttura di Elias-Fano può essere vista come una struttura di selezione su vettori di bit **non sistematica**, e cioè che rimpiazza interamente il vettore di bit originale con una rappresentazione **opportunistica** che è funzionale ai vettori sparsi.

Definizione 3.11 | Rango su rappresentazione di Elias-Fano

Il rango, nella rappresentazione di Elias-Fano, non è un'operazione in tempo costante e corrisponde a contare quanti x_i sono minori strettamente di un dato p. Occorre costruire una struttura di selezione di 0 sul vettore dei bit superiori, e dato p trovare il bit di rango $\lfloor p/2^\ell \rfloor$, perché il primo x_i minore di p è associato a un 1 che sta tra la posizione $\text{select}_0(\lfloor p/2^\ell \rfloor - 1)$ e la posizione $\text{select}_0(\lfloor p/2^\ell \rfloor)$.

Scorriamo quindi il vettore all'indietro, cercando gli 1 (che corrispondono a elementi della lista) e estraendo il corrispondente valore finché non è minore o uguale a p. L'indice dell'1 raggiunto è il grado di p.

3.4 Parentesi bilanciate

Definizione 3.12 | Parentesi bilanciate

Sequenze di parentesi bilanciate possono essere utilizzate nella costruzione di strutture succinte. Una sequenza di parentesi può essere rappresentata come un vettore di n bit $\underline{\boldsymbol{b}}$ scrivendo un 1 per ogni parentesi aperta e uno 0 per ogni parentesi chiusa.

Definizione 3.13 | Funzione di eccesso

Definiamo **funzione di eccesso** (aperta):

$$E_b(i) = \left|\{b_j | j < i \wedge b_j = 1\}\right| - \left|\{b_j | j < i \wedge b_j = 0\}\right|$$

che rappresenta l'eccesso di parentesi aperte rispetto a quelle chiuse in posizione i (esclusa).

Proprietà 3.2 | Vettore debolmente bilanciato

Il vettore $\underline{\boldsymbol{b}}$ è **debolmente bilanciato** se la funzione di eccesso è sempre non-negativa, e vale 0 in 0 e in n.

Proprietà 3.3 | Vettore fortemente bilanciato

Il vettore $\underline{\boldsymbol{b}}$ è **fortemente bilanciato** se la funzione di eccesso è sempre strettamente positiva, tranne in 0 e in n, in cui vale 0.

Definizione 3.14 | Parentesi lontane e vicine

Per calcolare le primitive in tempo costante la sequenza di parentesi vengono divise in blocchi: una parentesi è detta **lontana** se la sua corrispondente è al di fuori del blocco, **vicina** altrimenti.

Definizione 3.15 | Pioniere

Una parentesi lontana aperta p è un **pioniere** se la sua parentesi corrispondente giace in un blocco diverso da quello della parentesi corrispondente a quella lontana aperta immediatamente precedente a p.

Lemma 3.1 | Numero dei pionieri

Se esistono k blocchi, ci sono al più $2k-3$ pionieri.

Dimostrazione 3.2 | Numero dei pionieri

Se consideriamo il grafo $G = \langle V, E \rangle$ in cui V contiene tutti i blocchi e $(i, j) \in E$ se il blocco i contiene un pionieri e il blocco j la sua parentesi corrispondente, G è **planare esterno**, e tali grafi hanno al più $2k-3$ lati.

Definizione 3.16 | Findclose

La primitiva $\mathrm{findclose}(p)$ trova la parentesi chiusa corrispondente a quella arteta in posizione p.

Per implementarla svolgiamo i seguenti passi:

1. Controlliamo se la parentesi in posizione p è vicina, scansionando il blocco, e nel caso restituiamo la posizione della parentesi corrispondente.
2. Altrimenti troviamo il pioniere associato (tramite un'operazione di predecessore che possiamo implementare tramite rango e selezione), e se il pioniere stesso è la parentesi restituiamo la parentesi corrispondente.
3. Altrimenti contiamo le parentesi aperte lontane tra il pioniere e p: è facile perché basta misurare la differenza in eccesso aperto al pioniere e in p. A ogni parentesi aperta deve corrispondere una parentesi chiusa lontana nel blocco di arrivo, e sappiamo per definizione che tutte le parentesi chiuse che dovremo contare saranno nello stesso blocco (per definizione di pioniere). Possiamo quindi concludere il calcolo con la scansione di un blocco.

Definizione 3.17 | Endclose

La primitiva $\mathrm{endclose}(p)$ trova la parentesi più vicina q, posto che esista, tale che p sta tra la parentesi aperta in q e la sua parentesi chiusa corrispondente.

Per implementarla teniamo conto in una tabella, per ogni blocco, della prima parentesi aperta avente eccesso aperto di uno inferiore al minimo nel blocco. Data una parentesi aperta in posizione p, con parentesi chiusa associata in posizione q, la parentesi aperta che corrisponde alla coppia che racchiude la coppia $\langle p, q \rangle$ è la prima che precede p con eccesso inferiore di uno a quello di p. Se questa parentesi si trova nel blocco, la possiamo trovare con una scansione. Altrimenti, andiamo a vedere se p è lontana o vicina:

Primo caso (lontana): la funzione di eccesso aperto tra p e q non è mai inferiore al valore che ha in p, e quindi il valore in p è il minimo del blocco a cui p appartiene: possiamo trovare la parentesi che racchiude $\langle p, q \rangle$.

Secondo caso (vicina): scandiamo le parentesi che seguono q nel blocco. Se troviamo una parentesi chiusa di eccesso aperto uguale a p, la corrispondente aperta è quella che cerchiamo, e la possiamo recuperare con una **findopen**. Altrimenti, tutte le parentesi aperte rimanenti nel blocco hanno eccesso aperto maggiore o uguale a p, e ricadiamo nel caso precedente.

Definizione 3.18 | Isomorfismo Left Child - Right Sibling (LC-RS)

L'**isomorfismo Left Child - Right Sibling** è un isomorfismo tra le foreste ordinate (cioè insiemi ordinati di alberi ordinati) e gli alberi binari. Le foreste si possono costruire in modo induttivo:

1. La foresta vuota è una foresta.
2. Date due foreste F e G, possiamo costruire una nuova foresta collegando un nuovo nodo alle radici degli alberi di F, ottenendo così un albero, e concateniamo a questo albero la foresta G.

Specifichiamo l'isomorfismo come segue:

1. All'albero binario vuoto corrisponde la foresta vuota.
2. Dato un albero binario $\langle S, D\rangle$, la foresta corrispondente si ottiene applicando la seconda regola alla foresta associata a S e a quella associata a D

Specifichiamo l'isomorfismo inverso come segue:

1. Alla foresta vuota corrisponde l'albero binario vuoto.
2. Data una foresta generata da F e G usando la seconda regola, se S è l'albero binario associato a F e D l'albero binario associato a G associamo alla foresta l'albero binario $\langle S, D\rangle$.

Definizione 3.19 | Rappresentare foreste tramite parentesi debolmente bilanciate

Una foresta può essere rappresentata scrivendo una serie di parentesi debolmente bilanciate durante una visita in profondità effettuata partendo dalla radice di ogni albero della foresta, nell'ordine naturale.

1. Alla foresta vuota corrisponde l'espressione ben parentesizzata vuota.
2. Data una foresta generata da F e G usando la seconda regola, se f è l'espressione associata a F e g quella associata a G associamo alla foresta l'espressione $(f)g$.

Definizione 3.20 | Rappresentare alberi binari tramite parentesi debolmente bilanciate

Per rappresentare un albero binario lo trasformiamo nella foresta corrispondente e utilizziamo una rappresentazione a parentesi bilanciate. Volendo esplicitare induttivamente:

1. All'albero binario vuoto corrisponde l'espressione ben parentesizzata vuota.
2. Dato un albero $\langle S, D\rangle$, se s è l'espressione associata a S e d quella associata a D, associamo all'albero binario l'espressione $(s)\,d$.

3.5 Funzioni succinte

Esempio 3.1 | Memorizzare hash tramite funzione succinta

Un problema interessante da risolvere in maniera succinta è la memorizzazione di una funzione $f : X \to 2^r$, dove X è un sottoinsieme di n elementi da un universo U di cardinalità u. Il numero di bit necessari a memorizzare una tale funzione è almeno rn.

Prendiamo due funzioni di hash, $h, g : U \to m$ con uno spazio dei valori m un po' più ampio della cardinalità n di X, e consideriamo un vettore $\underline{\boldsymbol{w}}$ di variabili di dimensione m con valori in 2^r. Scriviamo ora il sistema di n equazioni:

$$w_{h(x)} + w_{g(x)} = f(x) \mod 2^r$$

Per rappresentare i vincoli imposti dal sistema costruiamo un grafo G non orientato con m vertici e n lati, in cui per ogni $x \in X$ abbiamo che $h(x)$ è adiacente a $g(x)$ tramite un lato etichettato da $f(x)$. Consideriamo una sequenza di **pelature** del grafo G. La prima pelatura, $F_0 \subseteq m$, è data dall'insieme dei vertici che sono una foglia (hanno grado 0 o 1) nel grafo $G_0 = G$. La seconda, F_1, dall'insieme dei vertici che sono una foglia nel grafo G_1 ottenuto cancellando i vertici in F_0 dal grafo. Continuiamo così finché non arriviamo a una pelatura F_k in cui non ci sono più foglie: G_k è l'insieme vuoto se e solo se il grafo è **aciclico**.

Se G_k è vuoto, possiamo risolvere ora il sistema nel seguente modo: partiamo da F_{k-1}, e per tutti i vertici x che sono di grado 0 in G_{k-1}, assegniamo $w_x = 0$. I vertici x di grado 1 sono invece adiacenti esattamente a un altro vertice y. In genere, w_y è già stato assegnato, dal momento che y fa parte di una pelatura successiva: fanno eccezione i vertici agli estremi di un lato isolato, che possono essere entrambi non assegnati, nel qual caso poniamo $w_y = 0$. Se v è il valore assegnato al lato che collega x e y, poniamo allora:

$$w_x = v - w_y \mod 2^r$$

Possiamo sempre effettuare l'assegnamento perché gli F_i sono partizioni dei vertici del grafo, e quindi non incontreremo mai un vertice già assegnato.

Osservazione 3.4 | Quali grafi consentono di risolvere il problema di memorizzare hash tramite funzione succinta?

Assumendo che le funzioni di hash h e g siano casuali e indipendenti, il grafo che andiamo a costruire è un grafo casuale di m vertici con n archi, e un risultato importante di teoria dei grafi casuali dice che per n sufficientemente grande quando $m > 2.09n$ il grafo è **quasi sempre** privo d cicli. In sostanza, scegliendo bene h e g, e posto che il grafo non risulti degenere, quasi tutti i grafi che otterremo permetteranno di risolvere il sistema.

Osservazione 3.5 | Il caso proposto per risolvere il problema di memorizzare hash tramite funzione succinta è ottimo?

Il caso ora descritto non è ottimo: utilizzando dei 3-ipergrafi causali e generalizzando il processo di pelatura, utilizzando 3 funzioni di hash, portiamo il limite che garantisce l'aciclicità a $m > 1.23n$.

3.6 Hash minimali perfetti

Definizione 3.21 | Hash

Una funzione di **hash** per un insieme di chiavi $X \subseteq U$, dove U è l'universo di tutte le chiavi possibili, è una funzione $f : X \to m$.

Definizione 3.22 | Hash perfetto

Quando la funzione è **iniettiva**, lo **hash** è detto **perfetto**.

Definizione 3.23 | Hash minimale

Quando $|X| = m$, lo **hash** è detto **minimale**.

Definizione 3.24 | Fibra

La **fibra** di un elemento y del codominio di una funzione è l'insieme degli elementi del dominio mappati in y.

Definizione 3.25 | Volume di una partizione

Chiamiamo **volume** di una partizione il numero di insiemi che separa.

Lemma 3.2 | Bit per memorizzare una funzione di hash minimale perfetto

Per memorizzare una funzione di hash minimale perfetto occorrono quindi, se n non è troppo grande, almeno

$$\log e + O(\log n / n) \approx 1{,}44$$

bit per elemento.

Dimostrazione 3.3 | Bit per memorizzare una funzione di hash minimale perfetto

Una funzione di hash con n valori su U corrisponde a una partizione P di U con n parti non vuote: ogni parte è la fibra di un $x \in n$. Una tale funzione è minimale perfetta per un insieme $X \subseteq U$ se P **separa** X, cioè se in ogni parte di P c'è esattamente un elemento di X. Diremo che un insieme di partizioni di U è un **n-sistema** se per ogni $X \subseteq U$ di cardinalità n esiste una partizione dell'insieme che separa X. Per stimare quanti bit sono necessari per scrivere una funzione di hash dobbiamo quindi studiare la minima dimensione $H_U(n)$ di un n-sistema per U. Se avessimo un limite superiore v al volume di una partizione, avremmo immediatamente che:

$$H_U(n) \geqslant \frac{\binom{u}{n}}{v}$$

dato che dobbiamo separare $\binom{u}{n}$ insiemi, e al massimo una partizione ne può separare v. Una partizione ha volume massimo quando le sue parti sono approssimativamente uguali:

$$v \approx \left(\frac{u}{n}\right)^n$$

Pertanto:

$$H_U(n) \geqslant \frac{\binom{u}{n}}{\left(\frac{u}{n}\right)^n} = \frac{\frac{u!}{n!(u-n)!}}{\frac{u^n}{n^n}} = \frac{n^n}{n!} \cdot \frac{u!}{u^n(u-n)!}$$

Assumendo che n non sia troppo grande ($n \leqslant \sqrt[2+\varepsilon]{u}$), si ha:

$$\begin{aligned}
\frac{u!}{u^n(u-n)!} &= \frac{u(u-1)\cdots(u-n+1)}{u^n} \\
&\geqslant \left(\frac{u-n}{n}\right)^n \\
&\geqslant \left(1 - \frac{\sqrt[2+\varepsilon]{u}}{u}\right)^{2+\xi/u} = \left(1 - \frac{1}{u^{\frac{1+\varepsilon}{2+\varepsilon}}}\right)^{u^{\frac{1}{2+\varepsilon}}} \\
&= \left(1 - \frac{1}{u^{\frac{1+\varepsilon}{2+\varepsilon}}}\right)^{\frac{1+\varepsilon}{2+\varepsilon} - \frac{1+\varepsilon}{2+\varepsilon} + \frac{1}{2+\varepsilon}} \sim \left(e^{-1}\right)^{u^{\frac{-\varepsilon}{2+\varepsilon}}} \sim 1
\end{aligned}$$

e quindi:

$$H_U(n) \geqslant \frac{n^n}{n!} \cdot \frac{u!}{u^n(u-n)!} = \Omega\left(\frac{n^n}{n!}\right)$$

Prendendo i logaritmi naturali abbiamo che:

$$\ln H_U(n) \geqslant n \ln n - (n \ln n - n + O(\ln n)) = n + O(\ln n)$$

e cambiando base si ottiene: $\log H_U(n) \geqslant n \log e + O(\log n)$

3.7 Firme

Osservazione 3.6 | A cosa serve una firma?

Uno **hash minimale perfetto** non permette di riconoscere se un elemento fa parte di un insieme X o no, dato che è definita solo sugli elementi di X per definizione. Utilizziamo un insieme di **firme**, quindi, associato all'insieme X delle chiavi.

Osservazione 3.7 | Come vengono usate le firme?

Data una funzione $s : U \to 2^r$ che associ a ogni chiave possibile una sequenza di r bit "casuali", nel senso che la probabilità che $s(x) = s(y)$ se x e y sono presi uniformemente a caso da U è $\frac{1}{2^r}$. Oltre a f, memorizziamo una tabella di n firme a r bit S e mettiamo la firma $s(x)$ di $x \in X$ in S nella posizione $f(x)$.

Per interrogare la struttura risultante su input $x \in U$, agiamo come segue:

1. Calcoliamo $f(x)$ (che è un numero naturale).
2. Recuperiamo $S_{f(x)}$.
3. Se $S_{f(x)} = s(x)$ restituiamo $f(x)$, -1 altrimenti.

La collisione tra firme avverrà con probabilità $\frac{1}{2^r}$.

Osservazione 3.8 | Come si può ridurre lo spazio occupato?

Quando r non è troppo piccolo è possibile utilizzare un **vettore compatto** per memorizzare i valori della soluzione del sistema.

Per come abbiamo descritto il processo di soluzione non è possibile che vengano poste a valori diversi da zero più di n delle variabili.

Se potessimo memorizzare con una piccola perdita di spazio **solo i valori diversi da zero** potremmo ridurre l'occupazione di memoria di nr bit.

Per farlo, consideriamo un vettore di bit $\underline{\boldsymbol{b}}$ che contiene in posizione i un 1 se la variabile corrispondente w_i è diversa da zero, e zero altrimenti. Se mettiamo in un vettore C i valori delle variabili w_i diversi da zero abbiamo:

$$w_i = \begin{cases} 0 & \text{se } b_i = 0 \\ C[\text{rank}_b(i)] & \text{se } b_i \neq 0 \end{cases}$$

La funzione occuperà $nr + 1.23n$ bit, più il necessario per la struttura di rango.

Definizione 3.26 | Autofirma

Un **autofirma** è uno strumento per mantenere efficientemente un dizionario statico in maniera approssimata. Consideriamo un insime $X \subseteq U$ che vogliamo rappresentare e una funzione di firma $s : U \to 2^r$.

Andiamo a realizzare una funzione $f : X \to 2^r$ data da $f(x) = s(x)$ che utilizza $nr + 1.23n$ bit. La funzione f mappa ogni chiave nella propria firma. Il dizionaria viene a questo punto interrogato come segue:

1. Calcoliamo $f(x)$.
2. Se $f(x) = s(x)$ appartiene a X, altrimenti no.

Come detto precedentemente, falsi positivi avveranno con probabilità $\frac{1}{2^r}$.

A

Domande d'esame

1. Elias-Fano e limite teorico delle successioni monotone.

   [RISPOSTA] vedi dispense cap.11 sulle successione monotone. In particolare dimostrare quale è l'Information Theoretical Lower Bound, mostrare costruzione della struttura e descrivere come vengono effettuate le operazioni di recupero degli elementi e di successore.

2. Algoritmo di Karger per la mincut e probabilità di ottenerere la cut ottima.

3. Inapprossimabilità di MaxE3Sat tramite PCP.

4. Approssimazione 3/2 per il problema di Load Balancing.

5. Riduzione diretta da PCP a Max Independent Set con dimostrazione di inapprossimabilità per ogni valore di rapporto di prestazione (in pratica dimostrare che Max Independent Set non è in APX). Questa è stata la mia domanda a scelta.

   [TRACCIA RISPOSTA] L'argomento copriva quasi una lezione intera quindi qui riporto solo la traccia della risposta.

   I punti da seguire per la dimostrazione sono:

   - definire cosa si intende per configurazione del verificatore $(r(n), q(n))$ del PCP (input, bit aleatori, bit estratti dal testimone e ripsota SI/NO) e far vedere che il numero totale è esattamente $2^{r(n)} * 2^{q(n)}$
   - costruire grafo con nodi corrispondenti alle sole configurazioni accettanti e lati tra nodi inconsistenti (cioè che leggono bit diversi nelle stesse posizioni di un testimone).
   - Usando il PCP sappiamo che le istanze SI del problema di decisione scelto (qualsiasi esso sia) vengono mappate in istanze del grafo delle configurazioni che ha un indenpedent set con almeno $2^{r(n)}$ nodi. Questo perchè per definizione del PCP esiste almeno un testimone per cui per ogni stringa aleatoria il verificatore risponde SI e dato che i bit aleatori possibili sono $2^{r(n)}$ e sono consistenti.....
   - Per le istanze NO invece il numero massimo di nodi di un qualsiasi independent set è $2^{r(n)-1}$. ciò si dimostra per assurdo facendo vedere che se esistesse un independent set con più di questi nodi allora il PCP avrebbe una probabilità di errore superiore al 50% ma sappiamo per definizione che ciò non è possibile
   - in questo modo abbiamo costruito un gap di dimensione 2, ma dato che il gap è grande esattamente quanto l'inverso della probabilità di errore ($1/2 => 2, 1/4 => 4$ etc) allora posso ripetere la computazione del PCP diminuendo tale probabilità e incrementando la dimensione del gap. In particolare dato un qualunque rapporto di prestazione r posso costruire un gap grande almeno r. Ciò implica che per ogni r è impossibile esibire un algoritmo approssimato con rapporto di prestazione r ergo Max Independent Set non è in APX.

6. Test di Miller-Rabin con definizione di testimone di composizione.

   [RISPOSTA] vedi Cap.5 delle dispense del prof.

7. Risoluzione del problema di Vertex Cover attraverso la programmazione lineare con dimostrazione r=2. In particolare mi ha chiesto di definire cosa è la PL, di scrivere il mapping del problema (la definizione dei vincoli e la funzione obiettivo) e di dimostrare che l algoritmo è 2-approssimante.

   [RISPOSTA] Nel Cap 11.6 del Tardos è spiegato molto bene e c'è tutto quello che serve sapere.