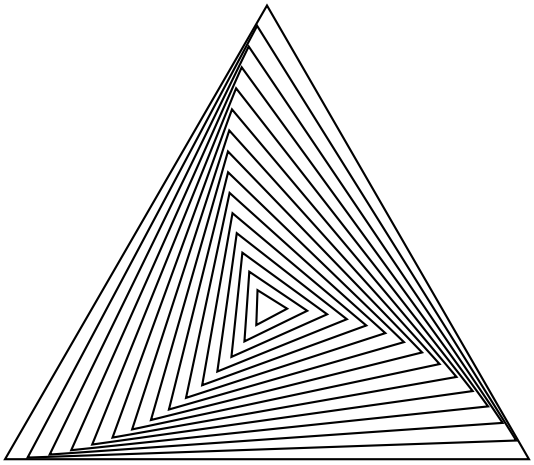
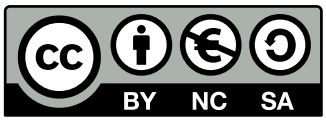
Algoritmi e complessità

Appunti delle lezioni tenute dal Prof. Paolo Boldi

Marco Cutecchia, Edoardo Marangoni

Università Statale di Milano
Dipartimento di Informatica
15 aprile 2022





Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

Questo corso esplora alcune tra le classi di complessità non trattate nei corsi di base di algoritmi; tratteremo gli algoritmi di ottimizzazione, gli algoritmi randomici e le strutture succinte. L'obiettivo è studiare delle aree importanti per gli informatici, presentando "oggetti" alla base delle tecniche informatiche. Iniziamo con un'introduzione alle notazioni matematiche utilizzate.

1.1 Notazione matematica

1.1.1 Insiemi

Useremo insiemi numerici come i numeri naturali $\mathbb{N}$, i numeri interi $\mathbb{Z}$ e i numeri reali $\mathbb{R}$ e le rispettive "versioni positive" $\mathbb{N}^+$, $\mathbb{Z}^+$ e $\mathbb{R}^+$.

1.1.2 Monoidi e stringhe

Avremo spesso a che fare con i **monoidi liberi**. Essi sono delle strutture algebriche che rispettano gli assiomi di **chiusura** (un monoide è un *gruppoide*), di **associativit**□ (un monoide è un *semigruppo*) e di esistenza dell'elemento neutro. Per ciò che ci interessa, "istanzieremo" i monoidi su degli alfabeti finiti Σ, costruendo un monoide *libero* Σ^* generato da Σ, ossia un insieme dotato di un'operazione binaria associativa $\cdot$ e un elemento neutro ϵ: indicheremo il monoide con la tripla $(\Sigma^*, \cdot, \epsilon)$[1]. Data una stringa $w \in \Sigma$, possiamo indicarne la lunghezza con $|w|$ e definiamo $w = w_0 w_1 \cdots w_n$.

1.1.3 Funzioni

Dati due insiemi A e B si definisce

$$B^A = \{f | f : A \to B\}$$

l'insieme di tutte le funzioni che hanno dominio A e codominio B. Se k è un numero intero, si definisce, introducendo una lieve ambiguità,

$$k = \{0, 1, \cdots, k-1\}$$

l'insieme con cardinalità k contenente i naturali da 0 a $k-1$; ad esempio, $0 = \emptyset$, $1 = \{0\}$ e così via.

Risulta quindi che 2^* è il monoide libero basato su tutte le stringhe binarie - solitamente equipaggiato con $\cdot$ come operazione di concatenazione e ϵ la stringa vuota. Definiamo

$$2^A = \{f | f : A \to \{0, 1\}\}$$

[1] Per un'introduzione alla relazione tra strutture algebriche e linguaggi (e la loro chiusura di Kleene) consultare [Sako9].

Possiamo interpretare i valori delle immagini di queste funzioni come dei booleani che descrivono l'appartenenza ad A: per esempio, se

$$\forall a \in A \ f_A(a) = 1$$

f_A è la *funzione caratteristica* dell'insieme A. Allargando il ragionamento a tutte le $f \in 2^A$, possiamo definire quest'ultimo come l'insieme delle funzioni caratteristiche di tutti i possibili sottoinsiemi di A.

Seguendo la definizione, abbiamo inoltre che

$$A^2 = \{f | f : \{0, 1\} \to A\}$$

ogni funzione associa a 0 un elemento di A e a 1 un elemento di A - a meno di isomorfismi, l'insieme delle immagini delle funzioni in A^2 è $A \times A$. Come ultimo esempio, abbiamo che 2^{2^*} è l'insieme di tutti i linguaggi binari.

1.2 Problemi

1.2.1 Definizione formale

Prima di definire cosa siano gli *algoritmi*, è necessario definire formalmente cosa sia un *problema*; un problema Π è definito da:

- l'insieme degli input del problema I_Π;
- l'insieme degli output del problema O_Π; e
- una funzione $Sol_\Pi : I_\Pi \to \{2^{O_\Pi} \setminus \emptyset\}$, interpretata come una funzione che associa ad ogni input i relativi output corretti per il problema - in altre parole, la funzione calcola un sottoinsieme non vuoto di O_Π che risolve il problema per il dato input[2].

Esempi

1
NPrime
Input: $\mathbb{N}$
Output: $\{0, 1\} = 2$
Problema: $n \in \mathbb{N}$ è primo?
è un problema di **decisione**.

2
MCD
Input: $\mathbb{N} \times \mathbb{N}$
Output: $\mathbb{N}$
Problema: Trova il massimo comun divisore tra due numeri.

3
SAT
Input: CNF ben formate
Output: $\{0, 1\} = 2$
Problema: È possibile soddisfare la formula in input?
è nuovamente un problema di decisione.

[2] La funzione ha come codominio un insieme di funzioni, di conseguenza si può vedere come una curryficazione di $Sol : I \times O \to 2$, che indica se, effettivamente, un output sia valido un certo input.

1.3 Rappresentazioni

In modo da poter definire formalmente gli algoritmi prendendo come modello di riferimento le macchine di Turing, assumiamo

$$I_{\Pi} \subseteq 2^2$$

e

$$O_{\Pi} \subseteq 2^2$$

Assumiamo di dover scrivere in binario i due numeri 3 e 5, ossia 11 e 101. Per dare i due input "in pasto" alla macchina di Turing non possiamo semplicemente concatenare le due stringhe: 11101 sarebbe un altro numero (29, in particolare). Possiamo utilizzare un trucco, ossia raddoppiamo ogni bit: 1111, 110011. Si nota facilmente che non compare mai la coppia 01 o 10 leggendo i bit due a due; si potranno utilizzare quindi questi marker per segnalare la fine di un numero e l'inizio di un altro.

Avremo spesso a che fare con input più complicati (si pensi al TSP: matrici di incidenza, liste di adiacenza, ...); se ci sono molti modi diversi per **codificare** l'input, parlare informalmente dei problemi causa problemi nel definire (e implementare, ovviamente) gli algoritmi, addirittura arrivando a cambiare la complessità dell'algoritmo in base alla codifica utilizzata.

Per il livello di dettaglio al quale noi vogliamo scendere nello studio della complessità, sarà sufficiente non dare molto peso in termini di differenza di complessità alle rappresentazioni, introducendo una leggera imprecisione. In termini pratici, come *regola del pollice*, si può notare che la distanza indotta, in termini di complessità, dal cambio di rappresentazione è polinomialmente limitata: si prenda l'esempio della rappresentazione a matrici di incidenza per i grafi sparsi; nonostante essi siano *meno efficienti* delle liste di adiacenza, ci si accorge facilmente che la distanza è limitata polinomialmente.

Tuttavia, non è sempre questo il caso: si prenda per esempio la rappresentazione binaria di un numero, e.g. `10100`, e la sua rappresentazione unaria `00000000000000000001`. È chiaro che il rapporto tra le due rappresentazioni non sia polinomiale: il confine tra polinomiale e "probabilmente non polinomiale" contiene dei problemi che hanno una complessità esponenziale se l'input è in rappresentazione binaria e "diventano" polinomiali se l'input è unario.

Gonfiando artificialmente l'input, il costo in tempo dell'algoritmo - che è rappresentato in termini della lunghezza dell'input - necessariamente decresce. In questo senso, se l'algoritmo è polinomiale nel *valore* dell'input ma non necessariamente nella sua lunghezza, esso è detto **pseudo-polinomiale**.

1.4 Algoritmi

Un algoritmo A per un problema Π è una funzione

$$A : I_{\Pi} \to O_{\Pi}$$

O, localmente,

$$x \mapsto y$$

tale che $y \in Sol_{\Pi}(x)$ (o, alternativamente, $Sol_{\Pi}(x)(y) = 1$), ossia una soluzione *corretta* per x.

In termini formali, un algoritmo rappresenta una *macchina di Turing*. Tuttavia, non scenderemo mai ad un livello di dettaglio tale per cui dovremo descrivere, effettivamente, un programma in termini di MdT, che richiederebbe uno sforzo non indifferente; utilizzeremo invece una notazione relativamente informale basata sullo *pseudocodice*.

Ciò che ci interessa degli algoritmi è studiare la loro **complessit□**. Quando si parla di complessità, si possono adottare due accezioni: complessità **algoritmica** e complessità **strutturale**.

1.4.1 Complessit□ algoritmica

Chiedersi se un determinato problema Π può essere risolto non è banale, affatto (basta seguire un qualsiasi corso di informatica teorica per rendersene conto) - e anche per una semplice argomentazione di cardinalità[3] ci possiamo rendere conto che esiste una quantità più che numerabile di problemi che può essere risolta da un numero numerabile di algoritmi. La teoria della calcolabilità è l'area che si occupa di questi problemi.

Per quanto riguarda il nostro corso, ci terremo dall'altra parte della barricata, ossia tratteremo solo problemi che sappiamo essere calcolabili, ossia problemi Π per i quali esiste un algoritmo A che lo risolve. Ma non tutti gli algoritmi sono equivalenti: ci interessa infatti studiare "quanto costa" un algoritmo, e dovremo di conseguenza adottare una misura di costo (spazio sul nastro, istruzioni eseguite,...)

Costo

Definiamo quindi una funzione di costo:

$$T_A : I_\Pi \to \mathbb{N}$$

che dipende da ciò che vogliamo calcolare; così com'è, però, è difficile da utilizzare con l'obiettivo di confrontare due algoritmi. È preferibile lavorare per "taglia", ossia per dimensione dell'input; definiamo quindi una funzione

$$t_A : \mathbb{N} \to \mathbb{N} \quad \text{dove} \quad t_A(n) = max\{T_A(x) | x \in I_\Pi \wedge |x| = n\}$$

che è chiamata *semplificazione del caso peggiore*; chiaramente, si ha che t_A è una valutazione pessimista del costo: per esempio, se $t_A(100) = 7500$ significa che su input di grandezza 100 il costo *massimo* è 7500. La complessità **algoritmica** utilizza proprio queste funzioni per confrontare due algoritmi. Dati A_1 e A_2 possiamo disegnare t_{A_1} e t_{A_2} come in figura 1.1.

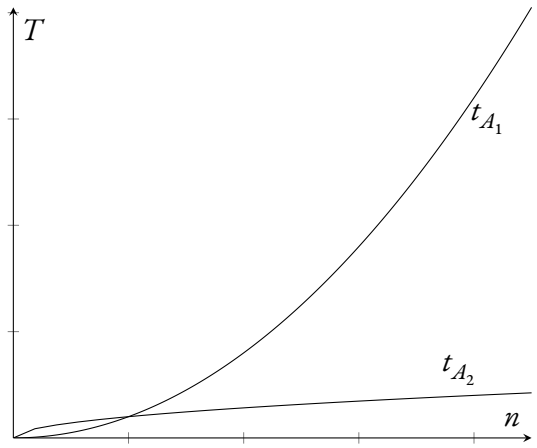


Figura 1.1: Complessità algoritmica semplificata nel caso peggiore per due funzioni t_{A_1} e t_{A_2}.

A questo punto, possiamo interessarci alle fasce di grandezza e capire, in un certo range, quale algoritmo scegliere, oppure fare un'assunzione asintotica scegliendo l'algoritmo che asintoticamente cresce di meno.

Upper e lower bound, ottimalit□ Supponiamo di aver trovato un algoritmo A la cui complessità è $O(n^{2.37})$. Questo valore è un *upper bound*. Siamo sicuri di non poter fare di meglio? Raramente si è certi: qui interviene un tipo di ragionamento completamente diverso, il cui obiettivo è dimostrare che più di tanto per un certo problema non si può fare, dimostrando quindi dei *lower bound*. In questo contesto si cercano dimostrazioni, non algoritmi. Trovando, per esempio, che il lower bound teorico per il problema è $\Omega(n^2)$, non si sanulla

[3] Tra tanti, due testi che trattano la calcolabilità sono [HU79] e [KMA82].

del range tra n^2 e $n^{2.37}$ e, idealmente, si continuerà a cercare un algoritmo finché si arriva ad un algoritmo $O(n^2)$, che è (asintoticamente) ottimale. Pochissimi problemi godono di un algoritmo ottimale: uno dei pochi è l'ordinamento di array, che ha complessità ottimale $\Theta(n \cdot log(n))$. Tuttavia, questo non significa che *heapsort*, per esempio, sia l'algoritmo *migliore* in toto: la pratica spesso smentisce queste possibilità. Un esempio è l'algoritmo di Danzig, in teoria esponenziale e in pratica migliore di altri algoritmi polinomiali.

1.4.2 Complessit□ strutturale

L'obiettivo finale della complessità algoritmica è trovare un algoritmo ottimo per ogni problema. Questo obiettivo è, chiaramente, quasi sempre irraggiungibile. Immaginiamo, per un momento, di conoscere tutte le complessità ottimali dei problemi: la complessità strutturale parte dal presupposto che per ogni problema si possa definire la *sua* complessità, in modo da poter collocare ogni problema in una precisa **classe**.

Classi di complessit□

Solitamente, la complessità strutturale si occupa esclusivamente di problemi di decisione, i quali sono analoghi al problema dell'appartenenza di una *parola* ad un *linguaggio*; pertanto tutti i problemi di decisione sono dei sottoinsiemi di 2^{2^*} e un sottoinsieme in particolare è l'insieme di *tutti i problemi decidibili in tempo polinomiale* **P**.

Per molti problemi vorremmo sapere se esso appartiene o meno a **P** ma, al momento, non abbiamo modo di saperlo: un esempio è SAT. Allo scopo di arrivare ad una risposta a questa domanda, è stato "inventata" la classe di complessità **NP**[4].

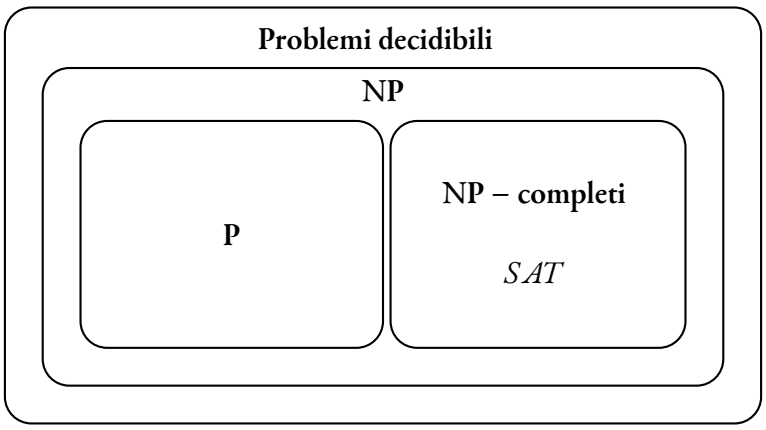


Figura 1.2: Classi di complessità strutturale.

Riducibilit□

Il concetto di **riducibilit□** (in tempo polinomiale) gioca una parte fondamentale nella teoria della complessità strutturale: si dice che un problema Π_1 è polinomialmente riducibile ad un problema Π_2 se e solo se

$$\exists f : 2^* \rightarrow 2^*$$

tale che:

- f è calcolabile polinomialmente
- $\forall x \in I_{\Pi_1}\ f(x) \in I_{\Pi_2}$

[4] Per approfondire queste tematiche consultare [AB09] e [AB08].

- $\forall x \; Sol_{\Pi_2}(x) = 1 \iff Sol_{\Pi_1}(x) = 1$

Questa definizione ha come conseguenza il seguente lemma.

Lemma 1.1. *Se $\Pi_2 \in \mathbf{P}$ e $\Pi_1 \leq_p \Pi_2$, ossia Π_1 è riducibile polinomialmente a Π_2, allora $\Pi_1 \in \mathbf{P}$.*

La classe dei problemi **NP − completi** è quindi definita come

$$\mathbf{NP - completi} = \{\Pi \in \mathbf{NP} | \forall \Pi' \in \mathbf{NP} \; \Pi' \leq_p \Pi\}$$

Teorema 1.2 (di Cook)**.** $SAT \in \mathbf{NP - completi}$.

CAPITOLO 2

Problemi di ottimizzazione

2.1 Introduzione

Un problema di **ottimizzazione** è caratterizzato da:

1. è l'insieme degli input I_{Π};
2. è l'insieme degli output O_{Π};
3. una funzione che ad ogni input associa una famiglia di output: $Amm_{\Pi} : I_{\Pi} \to \{2^{O_{\Pi}} \setminus \emptyset\}$;
4. il tipo del problema $Tipo_{\Pi} \in \{Min, Max\}$; e
5. una funzione da una coppia input-output ai naturali

$$c_{\Pi} : I_{\Pi} \times O_{\Pi} \to \mathbb{N}$$

che formalizza il concetto di **costo** di una soluzione - per un problema di minizzazione, l'obiettivo sarà scegliere l'output con costo minore.

Le proprietà 3, 4 e 5 formalizzano ulteriormente l'insieme Sol_{Π} definito in precedenza.

2.1.1 Esempi

MaxSat

MaxSAT	
Input:	CNF ben formate
Output:	$\mathbb{N}$
Problema:	Qual è il numero massimo di clausole che si possono verificare?
Ammissibili:	Assegnamenti di valori di verità coerenti
Tipo:	Max
Costo:	Numero di clausole rese vere

Le istanze di questo problema sone formule in forma normale congiunta ben formate (ossia CNF); le soluzioni ammissibili sono assegnamenti di valori di verità; il costo (o *funzione obiettivo*) è il numero di clausole rese vere. MaxSat ha chiaramente $Tipo_{\Pi} = Max$, in quanto l'obiettivo è massimizzare il numero di clausole verificate.

In alcuni frangenti potrebbe causarsi una certa ambiguità: l'algoritmo cerca *il valore* della soluzione ottimale o *la soluzione* ottimale stessa? Possiamo affermare che cerchiamo la soluzione stessa (in quanto il suo valore è calcolabile con la funzione di costo) e la indicheremo con la notazione $y^*(x)$; inoltre, indicheremo il costo della soluzione ottimale con $c^*(x)$.

2.2 Rapporto di prestazioni

Dato un input $x \in I_\Pi$ e $y \in Amm_\Pi$, possiamo sempre affermare che

$$\begin{cases} c_\Pi(x,y) \geq c_\Pi(x,y^*(x)) = c^*_\Pi(x) & \text{per i problemi di minimo} \\ c_\Pi(x,y) \leq c_\Pi(x,y^*(x)) = c^*_\Pi(x) & \text{per i problemi di massimo} \end{cases}$$

2.2.1 Rapporto di approssimazione

Definiamo **rapporto di approssimazione** la quantità

$$R_\Pi(x,y) = \max\{\frac{c_\Pi(x,y)}{c_\Pi(x,y^*(x))}, \frac{c_\Pi(x,y^*(x))}{c_\Pi(x,y)}\}$$

questo valore ci permette di dimenticare se stiamo trattando un problema di minimizzazione o massimizzazione, in quanto sarà sempre $R_\Pi \geq 1$.

α-approssimazione

Se, per esempio, $R_\Pi = 1$, la soluzione y è in realtà $y = y^*(x)$; se $R_\Pi = 2$, per un problema di minimo significa che il costo di y è il doppio del costo di $y^*(x)$, mentre per un problema di massimo significa che il costo di y è la metà del costo di $y^*(x)$. In generale, dato un problema di approssimazione tenteremo di progettare un algoritmo che preso un input $x \in I_\Pi$ produca un output $y(x) \in Amm_\Pi$. Se si riesce a dimostrare che l'algoritmo costruito trova una soluzione che, per ogni input x, è tale per cui $R(x,y(x)) \leq \alpha$ si definisce l'algoritmo α-approssimato.

2.3 Classi di complessit□ per l'ottimizzazione

Considereremo sempre algoritmi che terminano in tempo polinomiale; ovviamente, vorremmo trovare un α il più piccolo possibile - l'obiettivo quindi non sarà più migliorare il polinomio, bensì migliorare il grado di approssimazione α trovando il più piccolo possibile.

La classe dei problemi approssimabili in modo esatto ($\alpha = 1$) in tempo polinomiale è chiamata **PO**; si noti che non è una classe ristretta ai problemi di decisione - questa è infatti l'analogo della classe **P** rispetto ai problemi di ottimizzazione. Allo stesso modo possiamo definire una classe dei problemi di ottimizzazione risolvibili con approssimazione 1 in tempo nondetermistico polinomiale: **NPO** è la classe definita dai problemi $\Pi = (I_\Pi, O_\Pi, Amm_\Pi, c_\Pi)$ tali per cui

1. esiste un polinomio Q tale che $\forall x \in I_\Pi \forall y \in Amm_\Pi(x) \ \ |y| \leq Q(|x|)$,
2. dato $x \in I_\Pi$ e $y \in 2^*$ con $|y| \leq Q(|x|)$ è decidibile in tempo polinomiale se $y \in Amm_\Pi$, e
3. c_Π è calcolabile in tempo polinomiale

Nonostante questa classe non sia definita in termini di MdT con modulo nondetermistico (in quanto questo modello è "démodée": la teoria della complessità moderna utilizza al loro posto il concetto di *testimoni*) la definizione è completamente analoga e riconducibile alle definizioni che ne fanno uso.

2.3.1 Classe di problemi NPO − completi

Tra la classe **PO** e **NPO** sussiste la stessa relazione che c'è tra **P** e **NP** - effettivamente, possiamo anche definire i problemi **NPO − completi**. Per arrivare a questa definizione occorre definire la nozione di *problema di decisione associato*: dato un problema di ottimizzazione Π, definiamo un problema di decisione $\hat{\Pi}$ associato a Π definendo

$$I_{\hat{\Pi}} = I_{\Pi} \times \mathbb{N}$$

che formalizza la *richiesta* "esiste una soluzione ammissibile per x con costo minore o uguale a k?" (o maggiore o uguale per i problemi di massimizzazione).

Teorema 2.1.

$$\Pi \in \mathbf{PO} \implies \hat{\Pi} \in \mathbf{P}$$
$$\Pi \in \mathbf{NPO} \implies \hat{\Pi} \in \mathbf{NP}$$

Analogamente, la classe dei problemi **NPO − completi** è

$$\mathbf{NPO - completi} = \{\Pi \in \mathbf{NPO} | \hat{\Pi} \in \mathbf{NP - completi}\}$$

Ed è corretto aspettarsi che il problema di inclusione di **NP** in **P** venga mantenuto anche per i problemi di ottimizzazione:

Teorema 2.2. *Se* $\Pi \in$ **NPO − completi**, *allora* $\Pi \notin$ **PO** *a meno che* **P** = **NP**.

Dimostrazione. Assumiamo $Tipo_{\Pi} = Max$. Per assurdo, supponiamo $\Pi \in$ **PO**. dato un input $(x, k) \in I_{\Pi} \times \mathbb{N}$ calcoliamo la soluzione ottima $y^*(x)$ in tempo polinomiale usando il fatto che $\Pi \in$ **PO**. Se $k \leq c_{\Pi}(x, y^*(x))$ rispondiamo *sì*, altrimenti rispondiamo *no*. Questo algoritmo funziona in tempo polinomiale e decide il problema di decisione associato a Π; in quanto $\hat{\Pi} \in$ **NP − completi**, concludiamo **P** = **NP**. □

2.3.2 Altre classi di complessit□

In base al rapporto di approssimazione e al comportamento dell'algoritmo dati gli input e il rapporto di approssimazione stesso è possibile definire ulteriori classi di complessità. Utilizziamo ora la notazione A_{Π} per denotare un algoritmo che risolve il problema Π.

La classe APX

La classe dei problemi *approssimabili* in tempo nondeterministico polinomiale:

$$\mathbf{APX} = \{\Pi | \exists A_{\Pi}, \alpha : A_{\Pi} \text{ è } \alpha\text{-approssimante per } \Pi\}$$

Abbiamo che **APX** ⊊ **NPO**: vi sono infatti alcuni problemi che non sono approssimabili.

La classe PTAS

La seguente classe è parametrizzata dal valore del rapporto di approssimazione scelto:

$$\mathbf{PTAS} = \{\Pi | \exists A_{\Pi}, (x, \rho) \in I_{\Pi} \times \mathbb{Q}^{\geq 1} : A_{\Pi}(x) = y \in Amm_{\Pi}(x) \text{ in tempo } poly(x) \wedge R_{\Pi}(x, y) \geq \rho\}$$

Abbiamo che **PTAS** ⊊ **APX**, infatti vi sono problemi che non possono essere approssimati al più di un certo valore. Si noti che i problemi in **PTAS** sono risolti in tempo polinomiale *nell'input* ma non nel valore di approssimazione stesso.

La classe FPTAS

Stringendo la restrizione di polinomialità anche sul rapporto di approssimazione otteniamo la seguente classe:

$$\textbf{FPTAS} = \{\Pi | \Pi \in \textbf{PTAS} \wedge A_\Pi \text{ termina in tempo } poly(x, \rho)\}$$

Abbiamo che **FPTAS** $\subsetneq$ **PTAS**, infatti vi sono problemi che possono essere approssimati arbitrariamente solo utilizzando un tempo non polinomiale nel valore di approssimazione.

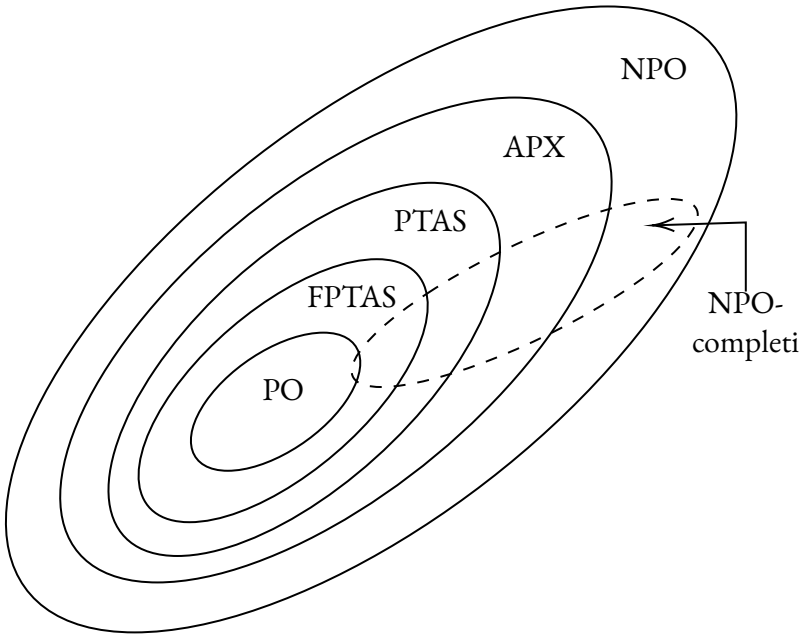


Figura 2.1: Rappresentazione insiemistica delle classi di complessità.

2.4 Terminologia riguardante i problemi

2.4.1 Grafi

I grafi non orientati sono $G = (V, E)$ (vertici e lati), dove

$$E \in \binom{V}{2}$$

Il **grado** di un vertice x $d(x)$ è il numero di lati incidenti su tale vertice. Il numero di vertici è $n = |V|$, mentre $m = |E|$. In un grafo non orientato un **cammino** di lunghezza k

$$\pi = v_1, v_2, \cdots, v_k$$

tale che $\forall v_i \in \pi : \exists \{v_i, v_{i+1}\} \in E$. Un cammino senza ripetizioni di vertici è chiamato *semplice*, altrimenti è definito *non semplice*. Un **circuito** è un cammino semplice chiuso di lunghezza ≥ 3. La **connessione** tra due vertici x e y è denotata $x \rightsquigarrow y$ e sussiste se esiste un cammino da x a y (e, di conseguenza, da y a x); questa nozione è inoltre una relazione di equivalenza (totale), gode infatti di riflessività, transitività e simmetria. Gli insiemi di vertici mutuamente connessi formano le **componenti connesse** di un grafo.

CAPITOLO 3

Algoritmi deterministici

Come anticipato, in questo corso tratteremo inizialmente gli algoritmi deterministici.

3.1 Problema del matching bipartito massimale

BipartiteMaxMatching
Input: $G = (V, E)$ *bipartito*
Output: Insieme di lati
Problema: Qual è il *matching* più ampio?
Ammissibili: Scelte di insiemi di lati che siano *matching*
Tipo: Max
Costo: Cardinalità dell'insieme

Un grafo *bipartito* è un grafo non orientato in cui l'insieme dei vertici è diviso in due parti; tutti i lati hanno un'estremità che incide su una parte e un'estremità che incide sull'altra, come rappresentato nella figura 3.1.

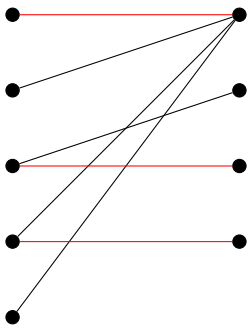


Figura 3.1: Esempio di un grafo bipartito. I lati colorati rappresentano un possibile *matching*.

Le soluzioni ammissibili sono dei *matching*, ossia una scelta di lati tale che nessun vertice abbia più di un lato incidente. L'obiettivo è trovare il matching più grande possibile, ossia quello col numero maggiore di lati. La funzione di costo è quindi la cardinalità dell'insieme di matching, e il problema è di massimo.

Questo problema si può risolvere polinomialmente. Esiste anche un algoritmo che risolve questo problema anche sui grafi generali, ma sono molto complessi.

3.1.1 Algoritmo basato su cammini aumentanti

Dato un grafo G e un matching M possiamo definire *occupati* i lati presenti nel matching. Un vertice *esposto* è un vertice su cui non incidono lati occupati. Nei vertici non esposti può incidere al massimo uno ed un solo lato occupato (altrimenti non sarebbe un matching). A partire da questa definizione, possiamo definire

i **cammini aumentanti**: essi sono cammini che alternano lati liberi e lati occupati, iniziando e terminando su un vertice esposto.

Dato il cammino aumentante, possiamo rimuovere dal matching tutti i lati *occupati* dal cammino aumentante e possiamo inserivi tutti i lati che non erano presenti. Questa operazione si chiama **switch del cammino aumentante**: alcuni lati sono stati inseriti, altri sono stati rimossi. Il matching risultante può essere più grande o più piccolo, ma dovremo comunque controllare se effettivamente ciò che risulta è un matching.

(a) Prima dello switch. (b) Dopo lo switch.

Figura 3.2: Esempio di un cammino aumentante in un grafo bipartito. I lati colorati rappresentano i lati selezionati nel matching.

Teorema 3.1. *Esiste un cammino aumentante per M se e solo se M non è massimo.*

Dimostrazione. $\Longrightarrow$ banale dall'operazione di switch.

$\Longleftarrow$ Ipotizziamo, per assurdo, che M non sia massimo e non esistano cammini aumentanti. Sia M' un matching tale che $|M'| > |M|$. Sia $X = M \triangle M' = (M \setminus M') \cup (M' \setminus M)$.

Su ogni vertice incidono al massimo due lati di X, ossia in X vi saranno alcuni vertici con grado 0, 1 oppure 2 (se ci fossero con grado 3, allora uno dei due matching non sarebbe davvero un matching).

Se si trova un ciclo in X con un lato in $M \setminus M'$, quello seguente dovrà essere in $M' \setminus M$, quello seguente ancora sarà in $M \setminus M'$ e così via; i cicli hanno pertanto per forza lunghezza pari. Ogni ciclo, quindi, "cancella" una quantità uguali di lati dalle due metà $(M \setminus M', M' \setminus M)$.

Ma deve essere per forza $|M' \setminus M| > |M \setminus M'|$, poiché M' è un matching più ampio di M. Oltre ai cicli si devono considerare anche i cammini: vi deve essere un cammino in X che ha più lati in $M' \setminus M$: i lati saranno alternati, nuovamente, tra lati in $M \setminus M'$ e lati in $M' \setminus M$: i vertici estremi devono essere esposti in M, altrimenti ci sarebbe un lato di M che incide su di esso, che però non può stare in M'. Pertanto, questo cammino è un cammino aumentante per M.

□

L'algoritmo è definito come segue.

Algoritmo 1: BipartiteMaxMatching

Input: grafo $G = (V, E)$
Output: Matching massimale per G

```
1 M = ∅
2 while ∃ cammino aumentante π per M do
3     esegui uno switch su π
```

Dimostriamo ora che se G è bipartito, calcolare se esiste un cammino aumentante ha costo $O(nm)$, rendendo la complessità in tempo

$$O(n^2 m)$$

Dimostrare che esiste un cammino aumentante esiste richiede una visita; procediamo, quindi, con un breve intermezzo riguardo la visita dei grafi.

Visite di grafi

Le visite dei grafi (*graph traversal*) sono algoritmi nei quali i vertici del grafo, in ogni istante, possono rientrare in tre categorie:

- **bianchi**: sono vertici sconosciuti e non esplorati.
- **grigi**: sono vertici conosciuti ma non esplorati. Vengono anche chiamati *nodi di frontiera*. Essi sono contenuti in un'apposita struttura dati chiamata *dispenser*, che effettivamente classifica la visita.
- **neri**: sono vertici conosciuti e visitati.

L'algoritmo generale di visita è l'algoritmo 2.

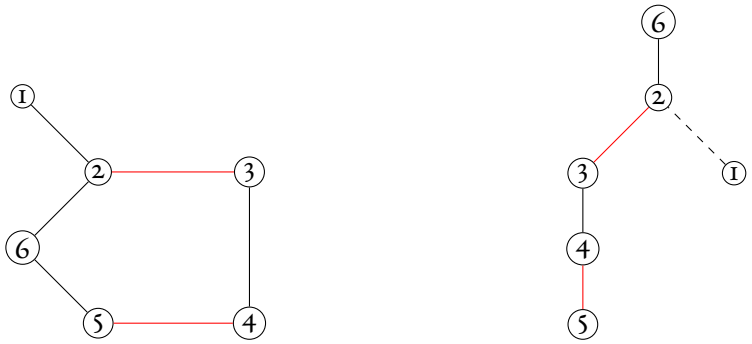
Algoritmo 2: GenericGraphVisit

```
   Input: grafo G = (V, E), vertice v_0
1  for v ∈ V(G) do
2      v.color = bianco
3  F = v_0                          /* F è un dispenser di vertici, e.g. stack */
4  v_0.color = grigio
5  while !F.empty() do
6      v = F.get()
7      visit(v)
                                        /* N(v) estrae i successori di v */
8      for w : N(v) do
9          if w.color = bianco then
10             F.insert(w)
11             w.color = grigio
12     v.color = nero
```

La visita dipende dalla struttura F: se F è una coda, la visita sarà in ampiezza, mentre se è uno stack la visita sarà in profondità. Chiaramente, non è detto che al termine dell'algoritmo tutti i vertici siano stati visitati, infatti, se non esiste un cammino tra il nodo seme ed un altro nodo, quest'ultimo non sarà mai visitato (pertanto, le visite possono essere utilizzate anche per trovare le componenti connesse.) Ogni visita richiede tempo $O(m)$, poiché di tutti i nodi si guardano i vicini una volta sola.

Un cammino aumentante è un cammino che comincia e termina in un vertice esposto e alterna lati nel matching a lati esterni. Inizialmente, possiamo limitarci a cercare tali cammini tra quelli che partono da un vertice esposto; eseguiamo quindi una visita in ampiezza (ossia F è una coda) alternando i lati liberi ai lati occupati. Se nel corso della visita si trova un vertice esposto. Questo algoritmo funziona solo su grafi bipartiti: si supponga di cominciare la visita dal nodo 6 in figura 3.3. I vicini di 6 sono 2 e 5: supponiamo di scegliere il lato $6 \to 2$. I vicini di 2 sono 3 e 1. Selezioniamo l'unico lato coperto, $2 \to 3$. I vicini di 3 sono 4 e 2 (che è nero); selezioniamo l'unico lato non coperto, $3 \to 4$. Procediamo con $4 \to 5$. La visita termina. Il cammino aumentante esiste: $6 \to 5 \to 4 \to 3 \to 2 \to 1$.

Teorema 3.2. *BipartiteMaxMatching* $\in$ **PO**.



(a) Situazione ipotetica in cui solo due lati, quelli rossi, sono inseriti nel matching.

(b) Visita per la creazione del cammino aumentante a partire dal nodo 6.

Figura 3.3: Esemplificazione dell'algoritmo per trovare cammini aumentanti.

Corollario 3.3. *Il problema del* PerfectMatching *(dato un grafo, decidere se esiste un matching che incide su tutti i vertici) è in* **P**.

Questo deriva dal fatto che se esiste un matching perfetto per un grafo, il matching massimo avrà cardinalità $n/2$.
Tutti gli altri problemi che vedremo saranno in **NPO – completi**.

3.2 Problema del bilanciamento del carico

LoadBalancing
Input: n task $t_0, t_1, \cdots, t_{n-1} \in \mathbb{N}^+$, m macchine
Output: Insieme di assegnamenti di task a macchine
Problema: Qual è l'assegnamento con costo massimo minore?
Ammissibili: Scelte di assegnamenti di tutti i task alle macchine in modo coerente
Tipo: *Min*
Costo: Ogni macchina ha carico $L_i = \sum_{j \text{ assegnato a } i} t_j$ la funzione obiettivo è $L = \max_i L_i$

Per esempio, si supponga di avere 8 task e 3 macchine. I task hanno costo $\{3, 2, 3, 1, 1, 4, 5, 1\}$; vi sono diversi modi per assegnare le task:

- m_0: $\{5, 1, 1\}$ con $L_0 = 7$
- m_1: $\{4, 3\}$ con $L_1 = 7$
- m_2: $\{2, 3, 1\}$ con $L_2 = 6$.

Pertanto $L = 7$.

Teorema 3.4. *LoadBalancing* $\in$ **NPO – completi**.

Algoritmo 3: GreedyBalance

Input: m macchine, n task con costi t_i

```
for i : 0..n − 1 do
    L_i = 0
for j : 0..n − 1 do
    i = arg min(L_i)
    M_i.addTask(t_j)
    L_i = L_i + t_j
```

3.2.1 Algoritmo greedy balance

Un algoritmo banale per risolvere approssimativamente il problema LoadBalancing è l'algoritmo 3. Questo algoritmo ha complessità $O(n \log(n))$.

Teorema 3.5. *GreedyBalance è un algoritmo 2-approssimante per LoadBalancing.*

Prima di procedere con la dimostrazione, sono necessarie due osservazioni:

Lemma 3.6.

$$L^* \geq \frac{1}{m} \sum_j t_j$$

Dimostrazione. Si supponga di avere la soluzione ottima. Si ha che

$$\sum_{i=0}^{m-1} L_i^* = \sum_{j=0}^{n-1} t_j$$

e pertanto

$$L^* = \max\{L_i^*\}_i \geq \frac{1}{m} \sum_j t_i$$

□

Lemma 3.7.

$$L^* \geq \max_j t_j$$

Dimostrazione. Banale. □

Passiamo alla dimostrazione del Teorema 3.5.

Dimostrazione. Eseguiamo GreedyBalance; sia $\hat{i}$ la macchina più carica al termine dell'esecuzione, ossia $L = L_{\hat{i}}$. Sia $\hat{j}$ l'ultimo task assignato alla macchina $\hat{i}$. Prima di ricevere $\hat{j}$, il suo carico è tale per cui

$$\forall i = 1..m \; L_{\hat{i}} - t_{\hat{j}} \leq L_i' \leq L_i$$

dove L_i' è il carico della macchina i quando il task $\hat{j}$ è deve essere assegnato. Sommiamo quindi su tutti gli i:

$$m \cdot (L_{\hat{i}} - t_{\hat{i}}) \leq \sum_{i=1}^{m} L_i = \sum_{j=0}^{n-1} t_j$$

poiché tutte le task sono state assegnate. Grazie al Lemma 3.6 possiamo affermare

$$L_{\hat{i}} - t_{\hat{i}} \leq \frac{1}{m} \sum_i L_i \leq L^*$$

È ovvio che $L = L_{\hat{i}} = (L_{\hat{i}} - t_{\hat{j}}) + t_{\hat{j}}$ e sappiamo anche che $t_{\hat{j}} \leq L^*$ dal Lemma 3.7; possiamo quindi concludere

$$L = (L_{\hat{i}} - t_{\hat{j}}) + t_{\hat{j}} \leq L^* + t_{\hat{j}} \leq L^* + L^* = 2L^*$$

□

Abbiamo dimostrato che GreedyBalance è 2-approssimato, ma ci sono davvero degli input per cui fa così male? Potremmo trovare un'approssimazione migliore? Quando vogliamo arrivare a questa conclusione, si accompagna un *teorema di strettezza*: c'è uno specifico input per cui GreedyBalance fa davvero così male.

Teorema 3.8. *Per ogni $\epsilon > 0$ esiste un input per cui GreedyBalance fornisce una soluzione L tale che*

$$2 - \epsilon \leq \frac{L}{L^*} \leq 2$$

Dimostrazione. Sia $m \geq \lceil \frac{1}{\epsilon} \rceil$. Sia $n = m(m-1) + 1$ dove i primi $m(m-1)$ compiti hanno durata (o costo) 1, mentre l'ultimo ha durata (o costo) m. Questa è la soluzione ottima:

```
0  [1][1]...[1]  (m volte)
1  [1][1]...[1]  (m volte)
2  [1][1]...[1]  (m volte)
3  [1][1]...[1]  (m volte)
 ..[1][1]...[1]  (m volte)
m [      m      ]
```

GreedyBalance conclude l'esecuzione in questa configurazione:

```
0   [1][1]  ...  [1]  [     m     ]
1   [1][1]  ...  [1]  (m-1 volte)
2   [1][1]  ...  [1]  (m-1 volte)
..  [1][1]  ...  [1]  (m-1 volte)
m   [1][1]  ...  [1]  (m-1 volte)
```

Concludendo con $L = 2m - 1$. $L/L^* = (2m-1)/m = 2 - \frac{1}{m} \geq 2 - \epsilon$ □

Corollario 3.9. *LoadBalancing* $\in$ **APX**.

3.2.2 Algoritmo sorted balance

La dimostrazione precedente suggerisce un modo diverso per assegnare i task: partendo dal task più lungo per decidere gli assegnamenti si crea l'algoritmo SortedBalance. L'algoritmo 4 ha complessità $O(m \log(m) +$

Algoritmo 4: SortedBalance

Input: m macchine, n task con costi t_i

```
1 sortedTasks = sortDescending(t_i)
2 GreedyBalance(m, sortedTasks)
```

$n \log(m)$) e migliora il tasso di approssimazione.

Lemma 3.10. *Supponiamo che vi siano più task che macchine. Allora, considerando i task ordinati dal più al meno costoso, vale*

$$L^* \geq 2t_m$$

Ossia il valore ottimale è maggiore o uguale di due volte il costo dell'm-esimo task.

Dimostrazione. Banale: siccome prima di t_m altri m task sono stati assegnati (da t_0 a t_{m-1}) ognuno dei quali ha costo maggiore o uguale a t_m, t_m verrà necessariamente assegnato ad una macchina che avrà carico maggiore o uguale a t_m. □

Teorema 3.11. *SortedBalance è un algoritmo $\frac{3}{2}$-approssimato.*

Dimostrazione. Se $n \leq m$, SortedBalance (ma anche GreedyBalance) trova la soluzione ottimale: se ci sono meno task che macchine il costo finale sarà il costo della task più lunga, che è il lower bound del costo ottimale.

Assumiamo quindi $n > m$. Eseguiamo SortedBalance e sia $\hat{i}$ l'indice della macchina con carico massimo $L = L_{\hat{i}}$. Se la macchina $\hat{i}$ ha una sola task assegnata, la soluzione è ottima. Assumiamo, quindi, che $\hat{i}$ abbia almeno due task assegnate; sia $\hat{j}$ l'ultima task assegnatale. È evidente che $\hat{j} \geq m$ e questo significa che $t_{\hat{j}} \leq t_{\hat{m}} \leq \frac{1}{2}L^*$. Quindi

$$L = L_{\hat{i}} = (L_{\hat{i}} - t_{\hat{j}}) + t_{\hat{j}} \leq \frac{3}{2}L^*$$

□

Un risultato di Graham afferma che SortedBalance in realtà è $\frac{4}{3}$-approssimante, mentre un alro risultato di Hochbaum et al. dell'88 dimostra che il problema del bilanciamento del carico appartiene a **PTAS**, ossia esiste un algoritmo in grado di approssimare arbitrariamente la soluzione ottimale ed è anche stato dimostrato che il problema non sia in **FPTAS** (ammesso che **P** $\neq$ **NP**).

3.3 Problema della selezione del centro

Si supponga di avere un'azienda che ha vari uffici sparsi per la città S. Gli uffici hanno delle **distanze** definite tra loro. Al momento, un solo ufficio è dotato di magazzino. Questo è molto poco efficiente: i manager, dopo un calcolo, scoprono di avere a disposizione il budget per creare nuovi k magazzini. Ognuno degli uffici rimanenti si rivolgerà al magazzino più vicino. Dove inserire i nuovi magazzini in modo tale che i loro *bacini di utenza* abbiano la massima distanza ufficio-magazzino minima?

CenterSelection

Input:	S di punti, d una *metrica*, k il numero di punti selezionabili
Output:	Una selezione di punti $C \subseteq S$
Problema:	Quali punti selezionare in modo che la distanza tra i selezionati e non selezionati sia minima?
Ammissibili:	La selezione è di al più k punti: $\lvert C\rvert \leq k$
Tipo:	Min
Costo:	$\rho(C) = \max_{x \in S} d(x, C)$

Perché

$$d : S \times S \to R^+$$

definisca una metrica (o uno spazio metrico), deve essere

- $\forall x \ d(x, x) = 0$
- $\forall x, y \ d(x, y) = d(y, x)$
- $\forall x, y, z \ d(x, y) \leq d(x, z) + d(z, y)$ (disuguaglianza triangolare)

Teorema 3.12. *CenterSelection* $\in$ **NPO – completi**.

3.3.1 Algoritmo center selection plus

Questo algoritmo non è realistico e richiede anche un input in più, oltre a S, $d : S \times S \to R$ e k, anche $r \in R^{>0}$, il raggio di copertura ottimo. Chiaramente, r è a priori sconosciuto!

Algoritmo 5: CenterSelectionPlus

```
Input: S, k, r, d
C = ∅
while S ≠ ∅ do
    ĉ = extractRandomNode(S)
    C = C ∪ {ĉ}
    S = S \ {x | d(x, ĉ) ≤ 2r}
if |C| ≤ k then
    return C
else
    print("Impossible!")
```

L'algoritmo è descritto nel listato 5; diamo delle proprietà di questo algoritmo che dipendono da r.

1. Se CenterSelectionPlus emette una soluzione C, allora C è una soluzione ammissibile e $\rho(C) \leq 2r$.

2. Se $r \geq \rho^*$ CenterSelectionPlus emette un output valido diverso da "impossibile". Si consideri una soluzione ottima C^*: Ogni volta che l'algoritmo inserisce $\hat{c}$ in uno dei bacini di C^* tutti i punti a distanza minore o uguale di $2r$ vengono cancellati: un qualunque punto s che sta nello stesso bacino di $\hat{c}$ ha distanza $d(\hat{c}, s) \leq d(\hat{c}, c^*) + d(s, c^*)$, che sono anche $\leq \rho^*$, quindi

$$d(\hat{c}, s) \leq d(\hat{c}, c^*) + d(s, c^*) \leq 2\rho^* \leq 2r$$

   Ad ogni iterazione rimuoviamo quindi un bacino intero, pertanto dopo al più k iterazioni non ci saranno più punti in S, e il ciclo termina; di conseguenza $|C| \leq k$.

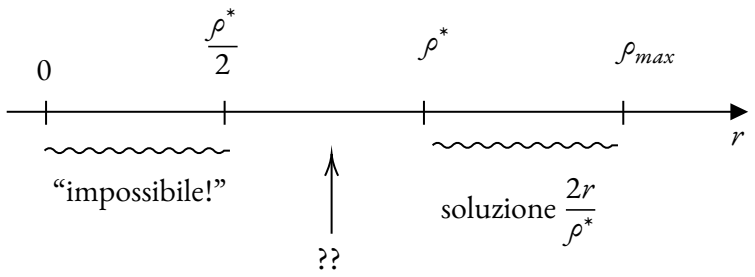


Figura 3.4: Comportamento di CenterSelectionPlus.

Quindi, ciò che succede dipende da r: se si sceglie maggiore o uguale a ρ^* l'algoritmo produce una soluzione ammissibile con approssimazione $\frac{2r}{\rho^*}$. Se $r \leq \frac{\rho^*}{2}$, l'algoritmo non produce soluzioni. Quando la scelta di r è $\rho^*/2 < r < \rho^*$ l'algoritmo ha un comportamento sconosciuto, ogni tanto produce soluzioni ammissibili e ogni tanto no. Noi non sappiamo il valore di r, ma sappiamo che è sicuramente $r \in [0, \rho_{max}]$, dove ρ_{max} è la distanza massima tra due punti. Una tecnica dicotomica (scegli un valore vicino a 0, se non produce nessun output alza la soglia minima e procedi con un valore vicino a ρ_{max}, ripeti) può portare dei risultati.

3.3.2 Algoritmo greedy center selection

L'algoritmo *greedy* per CenterSelection agisce come descritto nell'algoritmo 6.

Algoritmo 6: GreedyCenterSelection

Input: S, k, d
if $|S| \leq k$ **then**
  return S
$\hat{c} = extractRandomNode(S)$
$C = \{\hat{c}\}$
while $|C| \leq k$ **do**
  $\hat{c} = \arg\max_S d(s, C)$
  $C = C \cup \{\hat{c}\}$
return C

Teorema 3.13. *L'algoritmo GreedyCenterSelection è 2-approssimante.*

Dimostrazione. Per assurdo, supponiamo che C sia $\rho(C) \geq 2\rho^*$, ossia esiste un $\hat{s} \in S$ tale che $d(\hat{s}, C) \geq 2\rho^*$.

Consideriamo l'i-esima iterazione dell'algoritmo: sia $C_{\hat{i}}$ l'insieme dei centri all'inizio dell'i-esima iterazione e $c_{\hat{i}}$ il nuovo centro da inserire. Possiamo affermare che

$$\forall s \;\; d(c_{\hat{i}}, C_{\hat{i}}) \geq d(s, C_{\hat{i}})$$

e, in particolare, questo vale per $\hat{s}$:

$$d(c_{\hat{i}}, C_{\hat{i}}) \geq d(\hat{s}, C_{\hat{i}}) \geq d(\hat{s}, C) > 2\rho^*$$

il che implica che i k cicli sono una delle esecuzioni possibili dei primi k cicli dell'algoritmo, pertanto valgono tutte le proprietà che valevano per l'altro algoritmo per $r = \rho^*$. Siccome la distanza $d(\hat{s}, C) > 2\rho^*$ il punto $\hat{s}$ non è ancora stato rimosso da S, quindi l'algoritmo deve ancora fare almeno un'iterazione, e allora l'algoritmo emetterà "impossibile". Questo è assurdo poiché per la proprietà 2 di CenterSelectionPlus se $r \geq \rho^*$ allora l'algoritmo termina. □

Teorema 3.14. *Ammesso che* $\mathbf{P} \neq \mathbf{NP}$*, non esiste alcun algoritmo α-approssimante per CenterSelection con $\alpha < 2$*

Dimostrazione. Per dimostrare questo teorema introduciamo brevemente il problema del DominatingSet, un famoso problema di decisione NP-Completo.

DominatingSet
Input: $G = (V, E)$ non orientato, $k \in \mathbb{N}$
Output: $\{0, 1\} = 2$
Problema: Esiste $D \subseteq V$ con $|D| \leq k$ tale che sia un dominating set?

Diciamo che, dato un grafo $G = (V, E)$, un sottoinsieme $D \subseteq V$ è un DominatingSet di G se e solo se $\forall x \in (V \setminus D)$ esiste un $y \in D$ vicino di x.

Per assurdo, supponiamo che esista un algoritmo α-approssimanete per CenterSelection con $\alpha < 2$. Sia $(G = (V, E), k)$ una istanza di DominatingSet, costruiamo una istanza di CenterSelection dove $S = V$, $k = k$ e la funzione distanza è definita in questo modo:

$$d(x, y) = \begin{cases} 0 & \text{se } x = y \\ 1 & \text{se } x \neq y \text{ e } (x, y) \in E \\ 2 & \text{altrimenti} \end{cases}$$

È facile dimostrare che questa funzione rispetta le proprietà di uno spazio metrico: osservando la disuguaglianza triangolare abbiamo sul lato sinistro un valore che può essere 1 o 2, dal lato destro invece abbiamo la somma di due valori che possono essere entrambi anche loro 0 1 o 2, dunque sempre maggiore o uguale al lato sinistro

$$\underbrace{d(x,y)}_{1\,0\,2} <= \underbrace{\underbrace{d(x,z)}_{1\,0\,2} + \underbrace{d(z,y)}_{1\,0\,2}}_{2\,0\,3\,0\,4}$$

In una istanza di DominatingSet convertita in CenterSelection dunque abbiamo che ρ^* può assumere solo due valori: 1 o 2. Se $\rho^* = 1$ è possibile dimostrare che D, l'insieme di centri scelti, è un dominating set nell'istanza originale del problema, tramite una semplice catena di implicazioni

$$\begin{aligned}\rho^* = 1 &\leftrightarrow \forall x \in (V \setminus D) d(x, D) = 1 \\ &\leftrightarrow \forall x \in (V \setminus D), \exists y \in D \text{ tale che } d(x,y) = 1 \\ &\leftrightarrow \forall x \in (V \setminus D), \exists y \in D \text{ tale che } (x,y) \in E \\ &\leftrightarrow D \text{ è un DominatingSet}\end{aligned}$$

Eseguiamo il nostro algoritmo α-approssimante per CenterSelection, con $\alpha < 2$ per ipotesi, sull'istanza convertita da DominatingSet. Abbiamo dunque che

$$1 \leq \frac{\rho}{\rho^*} \leq 2 - \varepsilon$$
$$\rho^* \leq \rho \leq (2 - \varepsilon) * \rho^*$$

Dato che ρ^* può essere solamente 1 o 2, allora il nostro ρ può ricadere in uno solo di due casi disgiunti:

$$1 \leq \rho \leq 2 - \varepsilon \implies \text{esiste DominatingSet}$$
$$2 \leq \rho \leq 4 - 2\varepsilon \implies \text{non esiste DominatingSet}$$

Osservando quale delle due disequazioni il nostro ρ soddisfa, potremmo decidere in tempo polinomiale se esiste o meno un DominatingSet per la nostra istanza. Questo sappiamo essere impossibile, dunque concludiamo che non possiamo approssimare di più CenterSelection. □

3.4 Problema della copertura d'insiemi

Prima di passare a descrivere il problema SetCovering è necessario introdurre alcune proprietà delle funzioni armoniche.

3.4.1 Funzioni armoniche

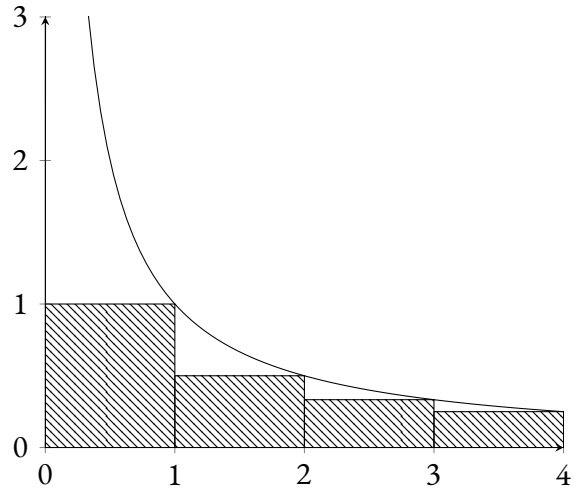
Una funzione armonica è una funzione

$$H : \mathbb{N}^+ \to \mathbb{R}$$

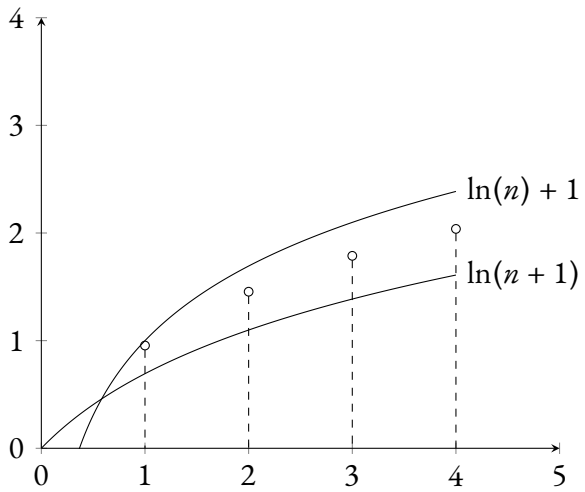
ed è definita

$$H(n) = \sum_{k=1}^{n} \frac{1}{k}$$

per esempio $H(3) = 1 + 1/2 + 1/3$.
Le funzioni armoniche hanno delle proprietà:



(a) Grafico di $f(x) = \frac{1}{x}$ verso aree delle somme in $H(4)$.



(b) Grafico di $f(x) = \ln(x) + 1$ e $v(x) = \ln(x + 1)$ verso i valori di $H(x)$ per i naturali $x = 1, 2, 3, 4$.

Figura 3.5: Proprietà di funzioni armoniche

Lemma 3.15.

$$H(n) \leq 1 + \int_1^n \frac{1}{x} dx = 1 + \ln(x)|_1^n = 1 + \ln(n) - 0 = 1 + \ln(n)$$

Lemma 3.16. *In quanto*

$$\int_t^{t+1} \frac{1}{x} dx \leq \int_t^{t+1} \frac{1}{t} dx = \frac{1}{t}$$

allora

$$H(n) = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{t} \geq \int_1^2 \frac{1}{x} dx + \int_2^3 \frac{1}{x} dx + \cdots + \int_n^{n+1} \frac{1}{x} dx =$$
$$= \int_1^{n+1} \frac{1}{x} dx = \ln(x)|_1^{n+1} = \ln(n+1)$$

quindi $H(N) \geq \ln(n+1)$*, e in conclusione*

$$\ln(n+1) \leq H(N) \leq 1 + \ln(n)$$

Torniamo a descrivere il problema SetCover:

SetCover

Input:	$S_1, S_2, \cdots, S_m \subseteq U$ tali che $\cup_{i=1}^m S_i = U$ e pesi $\forall i = 1, \cdots, m \;\; w_i \in \mathbb{R}^{>0}$
Output:	Insieme di insiemi scelti (o indici) $C \subseteq \{1, \cdots, m\}$
Problema:	Quali sono gli insiemi da scegliere per coprire tutti gli elementi di U col costo minore possibile?
Ammissibili:	C tale che $\cup_{i \in C} S_i = U$
Tipo:	Min
Costo:	$w = \sum_{i \in C} w_i$

3.4.2 Algoritmo greedy set cover

Enunciamo e dimostriamo alcune proprietà dell'Algoritmo 7; inizialmente notiamo che ogni elemento $s \in U$ viene inserito in qualche iterazione j dell'algoritmo che sceglie un qualche $\hat{S}_j = S_{i_j}$. Definiamo quindi

$$\forall u \in U \;\; c_u = \frac{w_{i_j}}{|S_{i_j} \cap R_j|}$$

Algoritmo 7: GreedySetCover

Input: S_i, U

$R = U$
$S = \emptyset$
while $R \neq \emptyset$ **do**
  $\hat{S} = \arg\min_{S_i} \{\frac{w_i}{|S_i \cap R|}\}$
  $S = S \cup \{\hat{S}\}$
  $R = R \setminus \hat{S}$
return S

come il costo di ogni singolo elemento di U coperto grazie alla scelta di S_{i_j} durante la j-esima iterazione.

Lemma 3.17.

$$w = \sum_{u \in U} c_u$$

Dimostrazione. Supponiamo che la scelta $S = \{S_{i_1}, S_{i_2}, \cdots, S_{i_k}\}$ produca un costo

$$w = w_{i_1} + w_{i_2} + \cdots + w_{i_k}$$

con ogni w_{i_j} il costo di ogni S_{i_j} scelto alla j-esima iterazione. Gli elementi coperti alla j-esima iterazione sono esattamente i $u \in S_{i_j} \cap R_j$, che hanno

$$c_u = \frac{w_{i_j}}{|S_{i_j} \cap R_j|}$$

ed essendo in numero proprio $|S_{i_j} \cap R_j|$, si ottiene

$$\sum_{u \in S_{i_j} \cap R_j} c_u = |S_{i_j} \cap R| \cdot \frac{w_{i_j}}{|S_{i_j} \cap R|} = w_{i_j}$$

da cui si ottiene facilmente l'enunciato. □

Lemma 3.18.

$$\forall k \sum_{u \in S_k} c_u \leq H(|S_k|) \cdot w_k$$

Dimostrazione. Sia

$$S_{i_j} = \{u_1, u_2, \cdots, u_d\}$$

dove gli u_i sono elencati in ordine di copertura, ossia all'inizio quelli che vengono coperti prima nell'algoritmo e alla fine quelli che verranno coperti dopo. Nell'iterazione j in cui verrà coperto un certo $u_k \in S_j$ sarà

$$\{u_k, \cdots, u_d\} \subseteq R_j$$

e quindi deve necessariamente essere

$$|S_{i_j} \cap R_j| \geq d - k + 1$$

ossia rimangono almeno tanti elementi da coprire quanti quelli non ancora coperti di S_{i_j} sin questo momento; pertanto

$$\begin{aligned} c_{u_k} &= \frac{w_{j'}}{|S_{j'} \cap R_j|} && \text{con } j' \text{ potenzialmente diverso da } i_j \\ &\leq \frac{w_{i_j}}{|S_{i_j} \cap R_j|} \leq && \text{poiché l'algoritmo sceglie il minimo} \\ &\leq \frac{w_{i_j}}{d-k+1} \end{aligned}$$

Sommando per ogni c_{u_i} si ha

$$c_{u_1} + \cdots + c_{u_d} \leq \frac{w_{i_j}}{d-1+1} + \frac{w_{i_j}}{d-2+1} + \cdots + \frac{w_{i_j}}{d-d+1} = w_{i_j}(1 + \frac{1}{2} + \cdots + \frac{1}{d}) = w_{i_j} \cdot H(|S_{i_j}|)$$

□

Teorema 3.19. *Sia $M = \max |S_i|$. L'algoritmo GreedySetCover è una $H(M)$-approssimazione per il problema SetCover.*

Dimostrazione. Sia $w^* = \sum_{S_i \in S^*} w_i$. Allora, grazie al Lemma 3.18, abbiamo

$$w_i \geq \frac{\sum_{u \in S_i} c_u}{H(|S_i|)} \geq \frac{\sum_{u \in S_i} c_u}{H(M)}$$

Consideriamo il valore ottenuto sommando, per ogni insieme della copertura ottima S^*, la somma dei costi dei singoli elementi in ogni insieme. Questo valore è maggiore o uguale alla somma dei costi degli elementi nell'universo U, poichè nel primo valore potremmo sommare più volte il costo di alcuni elementi che compaiono in più insiemi scelti. Per il Lemma 3.17 la seconda sommatoria è uguale al costo della soluzione ottima.

$$\sum_{S_i \in S^*} \sum_{u \in S_i} c_u \geq \sum_{u \in U} c_u = w$$

Applicando queste due osservazioni alla definizione di costo della soluzione ottima otteniamo

$$w^* = \sum_{S_i \in S^*} w_i \geq \frac{\sum_{S_i \in S^*} \sum_{u \in S_i} c_u}{H(M)} \geq \frac{w}{H(M)} \tag{3.1}$$

$$\implies \frac{w}{w^*} \leq H(M) \tag{3.2}$$

Inoltre, osservando che $M \leq |U| = n$, possiamo anche dire che:

$$H(M) \leq H(n) = O(\ln(n))$$

□

Corollario 3.20. *GreedySetCover è un algoritmo $O(\ln(n))-$approssimante.*

Si può inoltre dimostrare che non esiste nessun algoritmo $(1 - O(1)) \ln(n)$-approssimante per il SetCoverProblem, ammesso che $\mathbf{P} \neq \mathbf{NP}$.

Teorema 3.21. *Il lower bound $O(\log(n))$ è stretto.*

Dimostrazione. Il teorema si dimostra creando un input "cattivo": □

La particolarità tecnica interessante risiede nell'analisi di complessità: abbiamo attribuito ad ogni singolo elemento un costo; questa tecnica, che si chiama *pricing*, è a volte utile anche per il design degli algoritmi.

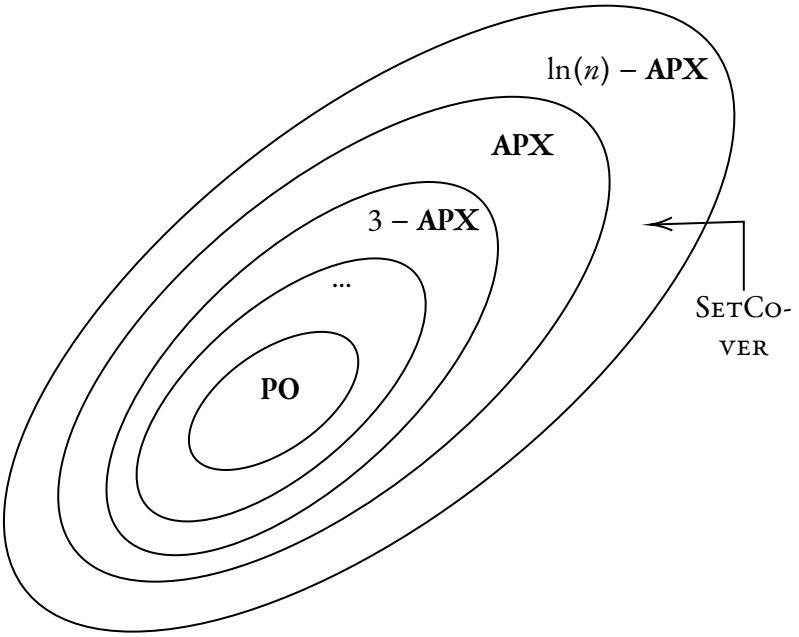


Figura 3.6: SetCover ∈ ln(n) − **APX**.

3.5 Problema della copertura dei vertici

VertexCover

Input:	$G = (V, E)$ non diretto, con pesi $\forall i \in V\ w_i \in \mathbb{Q}^{>0}$
Output:	Insieme di vertici $X \subseteq V$
Problema:	Quale è il numero minimo di vertici da selezionare per coprire ogni lato?
Ammissibili:	$X \subseteq V$ tale che $\forall e \in E\ e \cap X \neq \emptyset$
Tipo:	Min
Costo:	$w = \sum_{i \in X} w_i$

3.5.1 Relazione tra vertex e set cover

Teorema 3.22. *Ogni istanza del problema di decisione associato $\hat{\Pi}_{VC}$ del problema VertexCover è polinomialmente riducibile ad una istanza del problema di decisione associato $\hat{\Pi}_{SC}$ di SetCover, ossia*

$$\hat{\Pi}_{VC} \leq_p \hat{\Pi}_{SC}$$

Dimostrazione. Supponiamo di avere un'istanza $(G = (V, E), w_i, w_{max})$ di $\hat{\Pi}_{VC}$; si può trasformare nell'istanza $(S = \{S_1, S_2, \cdots, S_n\}, w_i, w_{max})$ di $\hat{\Pi}_{SC}$ definendo l'universo $U = E$ e creando un S_i per ogni vertice $i \in V$ tale che $S_i = \{e \in E | e \text{ incide su } i\}$. □

Di fatto, se abbiamo un algoritmo per risolvere SetCover con un fattore di approssimazione, allora possiamo risolvere anche VertexCover con lo stesso fattore; questo, ovviamente, non è necessariamente il caso generale: dati due problemi $\hat{\Pi}_1 \leq_p \hat{\Pi}_2$ e un algoritmo che risolve con un certo fattore di approssimazione $\hat{\Pi}_2$, non è detto che esso risolva *con lo stesso fattore di approssimazione* un'istanza di $\hat{\Pi}_1$.

Teorema 3.23. *VertexCover è $H(D)$-approssimabile (dove D è il grado massimo) utilizzando la riduzione polinomiale verso SetCover.*

3.5.2 Algoritmo basato su pricing

Sia l'istanza di VertexCover formata da $G = (V, E)$ un grafo e $\langle w_i \rangle_{i \in V}$ un'insieme di costi definiti sui vertici; diciamo che un *assegnamento di prezzi* sui lati $\langle P_e \rangle_{e \in E}$ è **equo** se e solo se

$$\forall i \in V \sum_{e \text{ incidente su } i} P_e \leq w_i$$

e l'assegnamento si definisce *stretto* su un vertice i se

$$\sum_{e \text{ incidente su } i} P_e = w_i$$

Lemma 3.24. *Se $\langle P_e \rangle_{e \in E}$ è un sistema di prezzi equo, allora*

$$\sum_{e \in E} P_e \leq w^*$$

dove w^* il costo ottimo per l'istanza di VertexCover.

Dimostrazione. Per l'equità sappiamo che

$$\forall i \in V \sum_{e \text{ incidente su } i} P_e \leq w_i$$

Sia quindi $X^* \subseteq V$ una soluzione ottima, sommiamo tutti i prezzi degli archi incidenti sui vertici di X^*:

$$\sum_{i \in X^*} \sum_{e \text{ incidente su } i} P_e \leq \sum_{i \in X^*} w_i = w^*$$

Dato che X^* è una soluzione, tutti i lati del grafo incidono su almeno uno dei suoi vertici, possiamo dunque dire

$$\sum_{e \in E} P_e \leq \sum_{i \in X^*} \sum_{e \text{ incidente su } i} P_e$$

Osservando i lati della disuguaglianza completa otteniamo il lemma:

$$\sum_{e \in E} P_e \leq \sum_{i \in X^*} \sum_{e \text{ incidente su } i} P_e \leq \sum_{i \in X^*} w_i = w^*$$

□

Algoritmo 8: PricedVertexCover

Input: $G(V, E), w_i$

```
for e ∈ E do
    P_e = 0
while ∃ē = {ī, j̄} t.c. P_ē non è stretto né su ī né su j̄ do
    Δ = min{w_ī − Σ_{e incidente su ī} P_e, w_j̄ − Σ_{e incidente su j̄} P_e}
    P_ē = P_ē + Δ
S = {v ∈ V | ⟨P_e⟩ è stretto su v}
return S
```

Lemma 3.25. *Al termine dell'esecuzione, per l'Algoritmo 8 vale*

$$w \leq 2 \sum_{e \in E} P_e$$

Dimostrazione. Abbiamo che

$$w = \sum_{i \in S} w_i \text{ e}$$

inoltre, dato che l'insieme S dato in output dall'algoritmo contiene per definizione solo vertici stretti, vale anche

$$\forall i \in S \; w_i = \sum_{e \text{ inc. su } i} P_e$$

quindi

$$w = \sum_{i \in S} \sum_{e \text{ inc. su } i} P_e$$

Per come è fatta la somma, ogni e appare 1 o 2 volte in base a se solo uno o entrambi dei suoi vertici sono stretti. Approssimando per eccesso dunque possiamo dire che

$$w \leq 2 \sum_{e \in E} P_e$$

□

Teorema 3.26. *PricedSetCover è un algoritmo 2-approssimante per VertexCover.*

Dimostrazione.

$$\frac{w}{w^*} \underset{\text{Lemma 3.25}}{\leq} \frac{2 \sum_{e \in E} P_e}{w^*} \underset{\text{Lemma 3.24}}{\leq} \frac{2 \sum_{e \in E} P_e}{\sum_{e \in E} P_e} = 2$$

□

3.5.3 Algoritmo basato sull'arrotondamento

Per poter utilizzare questa tecnica, è necessario introdurre alcune nozioni aggiuntive.

Programmazione lineare

LinearProgramming

Input:	Una matrice $A \in \mathbb{Q}^{m \times n}$ che rappresenta gli m vincoli per n variabili, un vettore $\mathbf{b} \in \mathbb{Q}^m$ che rappresenta
Output:	Un vettore $\mathrm{x} \in \mathbb{Q}^n$
Problema:	Qual è il vettore x che implica il costo minore?
Ammissibili:	$\mathrm{x} \in \mathbb{Q}^n$ tale che $A\mathrm{x} \geq \mathbf{b}$
Tipo:	Min
Costo:	$\mathbf{c}^T \mathrm{x}$

Programmazione lineare intera

IntegerLinearProgramming

Input:	Una matrice $A \in \mathbb{Q}^{m \times n}$, un vettore $\mathbf{b} \in \mathbb{Q}^m$ e un vettore $\mathbf{c} \in \mathbb{Q}^b$
Output:	Un vettore $\mathrm{x} \in \mathbb{Z}^n$
Problema:	Qual è il vettore x che implica il costo minore?
Ammissibili:	$\mathrm{x} \in \mathbb{Z}^n$ tale che $A\mathrm{x} \geq \mathbf{b}$
Tipo:	Min
Costo:	$\mathbf{c}^T \mathrm{x}$

È chiaro che, per lo stesso input, una soluzione per IntegerLinearProgramming è peggiore o uguale ad ogni soluzione per LinearProgramming, in quanto una soluzione per quest'ultimo potrebbe utilizzare anche lo spazio di valori frazionari per ottenere una soluzione migliore. La trasformazione di un problema da IntegerLinearProgramming in un problema in LinearProgramming è detto *rilassamento*.

Per lungo tempo non si è saputo se LinearProgramming fosse risolvibile in tempo polinomiale: oggi, si sa che è risolvibile in tempo polinomiale nella dimensione dell'input e nel numero di vincoli.

Teorema 3.27. *LinearProgramming* ∈ **PO**.

Uno degli esempi dei metodi polinomiali è il *metodo del punto interno* di Karmarkar. Questi metodi sono molto complicati da implementare e benché siano dimostrabilmente polinomiali, spesso sono meno efficienti di algoritmi che non sono dimostrabilmente polinomiali come l'**algoritmo di Dantzig**.

Per quanto riguarda ILP, invece, la situazione è diversa.

Teorema 3.28. *IntegerLinearProgramming* ∈ **NPO − completi**.

Dimostrazione. Per assurdo, assumiamo che esista un algoritmo che risolva IntegerLinearProgramming in tempo polinomiale, e di conseguenza anche il suo problema di decisione associato. Possiamo dare una riduzione in tempo polinomiale di una istanza del problema di decisione VertexCover in una istanza del problema di decisione IntegerLinearProgramming.

Dati n nodi con costi $v_0, \cdots, v_n$ e m archi, vogliamo scegliere il numero massimo di nodi tale che il costo sia minimo e tutti gli archi siano coperti. Per tradurre questo in un problema di programmazione lineare intera, generiamo una variabile binaria per ogni nodo:

$$x_i = \begin{cases} 0 & \text{il vertice } i \text{ non è stato scelto} \\ 1 & \text{il vertice } i \text{ è stato scelto} \end{cases}$$

Imponiamo i seguenti vincoli sulle variabili x_i:

$$\begin{cases} x_i + x_j \geq 1 & \forall i, j \in E \\ x_i \geq 0 & \forall i \in V \\ x_i \leq 1 & \forall i \in V \end{cases}$$

La funzione obiettivo per il problema IntegerLinearProgramming(G, w) così costruito sarà

$$\sum_{i \in V} w_i x_i$$

Se riuscissimo a risolvere tutti i problemi di IntegerLinearProgramming in tempo polinomiale allora potremmo risolvere anche VertexCover in tempo polinomiale. Dato che VertexCover ∈ **NP − Completi** concludiamo che anche IntegerLinearProgramming ∈ **NP − Completi** e dunque anche IntegerLinearProgramming ∈ **NPO − Completi**. □

Essendo nell'insieme **NPO** − *completi*, ogni problema è polinomialmente riducibile ad un'istanza di IntegerLinearProgramming; ciò che a volte accade è che anche la versione rilassata, ossia un'istanza di LinearProgramming, abbia una relazione col problema originale: questo è ciò che faremo con VertexCover.

Data un'istanza Π_{VC}, una soluzione per la sua riduzione Π_{ILP} è una soluzione **esatta** anche per l'istanza originale (per la **NP**−completezza di quest'ultimo problema). Se, ora, si rilassa il vincolo di interezza, ossia

$$\forall x, x_i \in \mathbb{Q}$$

mantenendo le condizioni

$$\begin{cases} x_i + x_j \geq 1 & \forall i, j \in E \\ x_i \geq 0 \\ x_i \leq 1 \end{cases}$$

si ottiene un'istanza *rilassata*, ossia LinearProgramming(G, w), che è risolvibile polinomialmente (ma non è detto che la soluzione sia esatta anche per il problema originale).

Abbiamo, da quanto detto precedentemente

Lemma 3.29.

$$w^*_{LP} \leq w^*_{ILP}$$

Per sfruttare la programmazione lineare per risolvere un'istanza di VertexCover, è necessario innanzitutto trasformarla in un problema di programmazione lineare (non intera); a questo punto è possibile risolvere l'istanza in tempo polinomiale ottenendo una soluzione x^*; definiamo la soluzione per l'istanza iniziale **r** come segue:

$$\forall i \in V \; r_i = \begin{cases} 1 & x_i \geq \frac{1}{2} \\ 0 & x_i < \frac{1}{2} \end{cases}$$

Lemma 3.30. *La soluzione* **r** *è ammissibile per* IntegerLinearProgramming*.*

Dimostrazione. Deve essere $\forall i, j \in E \; r_i + r_j \geq 1$. Per assurdo, assumiamo che $\exists i, j \in E$ tali che $r_i + r_j < 1$. Allora deve necessariamente essere che $r_i = r_j = 0$, poiché altrimenti la somma sarebbe già uguale a 1. Quindi

$$\begin{aligned} & x^*_i \leq \frac{1}{2} \wedge x^*_j \leq \frac{1}{2} \\ \implies & x^*_i + x^*_j < 1 \end{aligned}$$

Ma questo è impossible, perché nel problema di programmazione lineare c'è il vincolo che per ogni arco la somma delle due variabili corrispondenti fosse maggiore o uguale a 1 – in altre parole, x^* stessa non sarebbe una soluzione ammissibile per LinearProgramming. □

Lemma 3.31.

$$\forall i \in V \; r_i \leq 2x^*_i$$

Dimostrazione. Se $r_i = 0$ la disuguaglianza è ovvia; se $r_i = 1$ allora, $x^*_i \geq \frac{1}{2}$ e $2x^*_i \geq 1 = r_i$. □

Lemma 3.32.

$$\sum_{i \in V} w_i r_i \leq 2 \sum_{i \in V} w_i x^*_i = 2w^*_{LP} \leq 2w^*_{ILP}$$

Dimostrazione. Diretta dal Lemma 3.29. □

Teorema 3.33. *L'utilizzo di* LinearProgramming *per risolvere problemi di* VertexCover *porta ad un algoritmo* 2*-approssimante.*

Dimostrazione. Diretta dal Lemma 3.32. □

3.6 Problema dei cammini disgiunti

DisjointPaths

Input:	$G = (N, A)$ orientato, k coppie $(s_1, t_1), \cdots, (s_k, t_k) \in \mathbb{N} \times \mathbb{N}$ e $c \in \mathbb{N}$
Output:	Un insieme di cammini
Problema:	Quante coppie possono essere collegate senza utilizzare nessun lato più di c volte?
Ammissibili:	$I \subseteq \{1, \cdots, k\}$ e, per ogni $i \in I$, un cammino $\Pi_i : s_i \rightsquigarrow t_i$ tale che nessun $a \in A$ sia usato da più di c cam
Tipo:	Max
Costo:	Cardinalità dell'insieme I, $\lvert I \rvert$

Dato un grafo orientato $G = (N, A)$, il problema consiste nel collegare più coppie possibili senza usare nessun lato più di c volte. Anche in questo caso potremo usare un'algoritmo basato sulla tecnica del pricing: ogni lato avrà, nell'esecuzione di questo algoritmo, un prezzo che aumenterà durante l'esecuzione (più un lato è congestionato, più vogliamo che venga scelto raramente). Il problema è di massimo, ossia vogliamo collegare più coppie possibili.

3.6.1 Algoritmo basato su pricing

Oltre al grafo G, le k coppie e il numero c di utilizzi massimi per ogni lato, l'algoritmo utilizza un parametro β e una funzione di costo

$$l : A \to \mathbb{R}^{>0}$$

estensibile ai cammini, ossia dato un cammino

$$\pi = \langle x_1, x_2, \cdots, x_k \rangle$$

si definisce

$$l(\pi) = l(x_1, x_2) + l(x_2, x_3) + \cdots$$

Il listato è esposto nell'Algoritmo 9. Per dimostrare che esso è polinomiale, ricordiamo che Dijkstra richiede tempo $O(m \log(n))$; ripetuto k volte il costo di PricedDisjointPaths è $O(km \log(n))$.
Con lo scopo di esplorare alcuni aspetti dell'algoritmo, diamo alcune definizioni: fissata la funzione $l(\cdot)$, definiamo un cammino π **corto** se e solo se $l(\pi) \leq \beta^c$; inoltre, un cammino π è definito **utile** *in un determinato momento dell'esecuzione dell'algoritmo* se e solo se collega una coppia $i \notin I$.

Lemma 3.34. *Durante l'algoritmo, finché esistono cammini utili e corti, l'algoritmo sceglie uno di essi.*

Dimostrazione. Siccome l'algoritmo sceglie tra tutti i cammini utili quello più corto, è chiaro che se c'è un cammino di costo minore di β^c l'algoritmo non andrà mai a sceglierne uno lungo. □

Inoltre, come osservazione marginale, durante la prima fase dell'esecuzione dell'algoritmo possiamo evitare di cancellare gli archi: si supponga di non cancellare gli archi finché l'algoritmo ha modo di scegliere i cammini corti; si supponga quindi che per errore venga scelto un cammino *corto* che passa per un arco già utilizzato c volte. Questo è impossibile, poiché se quel cammino contiene un arco già utilizzato c volte, il suo costo è maggiore o uguale a β^c, pertanto non è corto.

Quando la disponibilità di cammini corti termina, l'algoritmo si ferma oppure comincia a scegliere dei cammini lunghi. A noi interessa proprio lo stato del sistema in questo momento: sia quindi c_t l'insieme dei cammini *utili* e *corti* all'inizio della t-esima iterazione. È chiaro che sia

$$c_0 \supseteq c_1 \supseteq \cdots \supset c_t$$

poiché un cammino utile e corto può diventare (o essere rimpiazzato da un altro, a causa dell'eliminazione degli archi) inutile o lungo. Può accadere quindi che ad una certa iterazione $\bar{t}$ che sia $c_{\bar{t}} = \emptyset$; chiamiamo

Algoritmo 9: PricedDisjointPaths

Input: coppie (s_i, t_i), β, $G = (N, A)$, l

```
1  I = ∅                                                    /* insieme di coppie */
2  P = ∅                                                    /* insieme di cammini */
3  for a ∈ A do
4      l(a) = 1
5  while true do
       /* trova il più corto cammino π_i rispetto a l che connette delle
          coppie (s_i, t_i) tali che i ∉ I                                        */
6      π_i = MinPath(l, (s_i, t_i)) if !π_i then
7          return I, P
8      I = I ∪ {i}
9      P = P ∪ {π}
       /* aggiorna i costi che pesano su ogni arco: se superano il massimo,
          rimuovili in modo che non vengano più scelti                            */
10     for a ∈ π_i do
11         l(a) = l(a) · β
12         if l(a) = β^c then
13             delete a
```

quindi $\bar{l}$ la funzione l a tale iterazione (o, se non accade, al termine dell'esecuzione). Sia quindi $\bar{P} \subseteq P$ l'insieme dei cammini $\bar{l}$-corti nell'output e $\bar{I} \subseteq I$ l'insieme delle coppie collegate da tali cammini.

Lemma 3.35. *Sia $i \in I^* \setminus I$. Si ha $\bar{l}(\pi_i^*) \geq \beta^c$; in altre parole il costo di un cammino ottimo per una coppia che non è stato incluso nella soluzione dall'algoritmo è, all'istante $\bar{t}$, maggiore o uguale di β^c.*

Dimostrazione. Se così non fosse, allora π_* sarebbe corto e, in quel momento, utile, pertanto verrebbe selezionato, contraddicendo l'ipotesi iniziale. In altre parole, π_i^* resta utile fino alla fine; se fosse $\bar{l}(\pi_i^*) < \beta^c$, allora π_i^* sarebbe corto nell'istante $\bar{t}$. □

Lemma 3.36.

$$\sum_{a \in A} \bar{l}(a) \leq \beta^{c+1} |\bar{I}| + m$$

Dimostrazione.

- per $t = 0$, $\sum_{a \in A} l_0(a) = \sum_{a \in A} 1 = m$
- consideriamo il j-esimo passo, in cui si modificano i pesi di alcuni archi modificando la funzione l_j creando l_{j+1} in questo modo:

$$l_{j+1}(a) = \begin{cases} l_j(a) & \text{se } a \notin \pi_i \\ \beta \cdot l_j(a) & \text{se } a \in \pi_i \end{cases}$$

Possiamo quindi calcolare la differenza di prezzi degli archi nel passaggio tra iterazioni:

$$\sum_{a \in A} l_{j+1}(a) - \sum_{a \in A} l_j(a) = \sum_{a \in A} (l_{j+1}(a) - l_j(a)) =$$
$$\underbrace{\sum_{a \in A \setminus \pi} 0}_{\text{nessun cambio di peso negli archi non in } \pi} + \sum_{a \in \pi} (\beta l_j(a) - l_j(a)) =$$
$$= \sum_{a \in \pi_i} (\beta - 1) l_j(a) \leq \beta \sum_{a \in \pi_i} l_j(a)$$

Sappiamo dal Lemma 3.35 che tutti i π aggiunti fino al passo corrispondente a $\bar{l}$ sono cammini utili e corti, dunque abbiamo che

$$\beta \underbrace{\sum_{a \in \pi_i} l_j(a)}_{\leq \beta^c \text{ dato che è un cammino utile}} \leq \beta^{c+1}$$

Abbiamo dunque che alla prima iterazione il peso totale degli archi è m mentre in quelle successive abbiamo un incremento nel peso totale degli archi di un valore $\leq \beta^{c+1}$. Sappiamo che ad ogni iterazione corrisponde l'aggiunta di un singolo cammino $\pi \in \bar{I}$, dunque $|\bar{I}|$ è anche il numero di iterazioni fatte dall'algoritmo quando la funzione di costo è $\bar{l}$. Mettendo insieme queste osservazioni concludiamo che

$$\sum_{a \in A} \bar{l}(a) \leq \beta^{c+1} |\bar{I}| + m$$

□

Corollario 3.37. *Dal Lemma 3.35 abbiamo*

$$\sum_{i \in I^* \setminus I} \bar{l}(\pi_i^*) \geq \beta^c |I^* \setminus I|$$

Corollario 3.38.

$$\sum_{i \in I^* \setminus I} \bar{l}(\pi_i^*) \leq \sum_{i \in I^*} \bar{l}(\pi_i^*) \leq c \sum_{a \in A} \bar{l}(a) \qquad \textit{poiché } I^* \textit{ è una soluzione ogni arco è usato al più } c \textit{ volte}$$

$$\leq c(\beta^{c+1} |\bar{I}| + m) \qquad \textit{dal Lemma 3.36}$$

Teorema 3.39. *PricedDisjointPaths è un algoritmo $O(1 + 2cm^{\frac{1}{c+1}})$−approssimante per DisjointPaths.*

Dimostrazione. Dalle proprietà degli insiemi è possible affermare che

$$\begin{aligned}
&\beta^c |I^*| = \beta^c |I^* \cap I| + \beta^c |I^* \setminus I| \leq \\
&\leq \beta^c |I^* \cap I| + \sum_{i \in I^* \setminus I} \bar{l}(\pi_i^*) && \text{dal Corollario 3.37} \\
&\leq \beta^c |I| + \sum_{i \in I^* \setminus I} \bar{l}(\pi_i^*) \leq \beta^c |I| + c(\beta^{c+1} |\bar{I}| + m) && \text{dal Corollario 3.38}
\end{aligned}$$

Siccome $\bar{I} \subseteq I$, possiamo continuare affermando

$$\begin{aligned}
&\beta^c |I| + c(\beta^{c+1} |\bar{I}| + m) \leq \beta^c |I| + c(\beta^{c+1} |I| + m) \\
&\implies \beta^c |I^*| \leq \beta^c |I| + c(\beta^{c+1} |I| + m) \qquad (3.3)
\end{aligned}$$

dividendo entrambi i membri dell'Eq. (3.3) per β^c si ottiene

$$|I^*| \leq |I| + c\beta |I| + c\beta^{-c} m$$

sostituiamo $c\beta^{-c} m$ con $c\beta^{-c} m |I|$ per semplificare i calcoli, possiamo farlo perchè $|I| \geq 1$. Otteniamo

$$|I^*| \leq |I| + c\beta |I| + c\beta^{-c} m |I|$$

dividendo entrambi i membri per $|I|$ si ottiene

$$\frac{|I^*|}{|I|} \leq 1 + c\beta + c\beta^{-c} m = 1 + c(\beta + \beta^{-c} m)$$

Scegliamo quindi $\beta = m^{\frac{1}{c+1}}$:

$$\frac{|I^*|}{|I|} \leq 1 + c(m^{\frac{1}{c+1}} + m^{\frac{-c}{c+1}} m) = 1 + c(m^{\frac{1}{c+1}} + m^{\frac{-c+c+1}{c+1}}) = 1 + 2cm^{\frac{1}{c+1}}$$

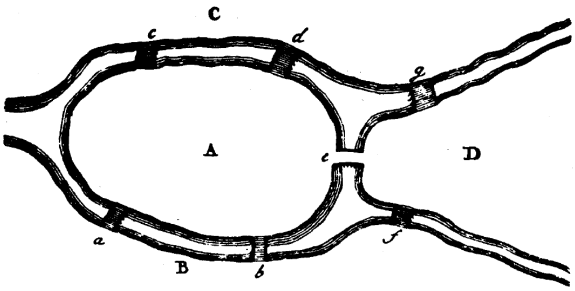
□

3.7 Problema del commesso viaggiatore

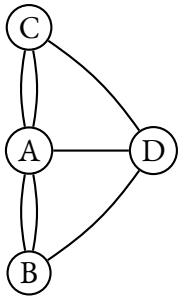
Il problema del commesso viaggiatore, o TravelingSalesman (problem), è uno dei problemi più famosi della teoria dei grafi. Prima di affrontarlo, è utile introdurre altre nozioni.

3.7.1 Problema dei sette ponti di Ko□nisberg

Könisberg, un tempo facente parte della Prussia Orientale e oggi odierna Kaliningrad, Russia, è percorsa dal fiume Pregel, e le aree della città sono collegate da sette ponti.



(a) Rappresentazione dei ponti come descritta da Eulero.



(b) Rappresentazione come multigrafo.

Figura 3.7: I ponti di Könisberg.

Un antico problema chiedeva: è possibile partire da un punto qualsiasi della città e attraversare tutti i ponti esattamente una ed una sola volta? Per studiare questo problema, Eulero pensò di trasformare questa mappa in un grafo, dove i vertici rappresentano le zone A, B, C, D e i lati sono i sette ponti della città. I grafi fatti in questo modo sono chiamati, oggi, *multigrafi*, ossia grafi i cui lati non sono un insieme ma un *multinsieme*, insiemi con ripetizioni distinte: formalmenti, sono rappresentati con una mappa che associa agli elementi il numero di ripetizioni. In termini di teoria dei grafi, il problema si traduce come segue: G ha un circuito (o ciclo) che passi esattamente una volta per ogni lato, ossia un **circuito euleriano**?

La risposta, in questo caso specifico, è no.

Teorema 3.40. *Esiste un circuito euleriano se e solo se tutti i vertici di un grafo connesso hanno grado pari.*

Dimostrazione. $\Longleftarrow$ (Se tutti i vertici di un grafo hanno grado pari, allora esiste un circuito euleriano.) Sia G un grafo in cui tutti i vertici hanno grado pari. Partendo da un vertice a caso e seguendo un cammino formato da lati non ancora scelti (ossia si tiene traccia di quelli già "consumati"), non può accadere che ci sia un arco non ancora scelto per planare sul nodo ma non un altro per uscirne, poiché questo significherebbe che il grado di tale nodo sia dispari.

In questa costruzione succede, ad un certo punto, che si torna su uno dei vertici già visitati. Anche in questa situazione deve esistere un arco che permette di uscire da tale vertice: si segue quindi il lato non ancora utilizzato e si continua il percorso. Prima o poi, con questo ragionamento, si tornerà al vertice dal quale si è partiti, e questo è l'unico modo per costruire un circuito, che non è detto che sia euleriano, poiché non è detto che visiti tutti i lati. Tuttavia, si può ricominciare la visita partendo da un lato non ancora visitato: siccome il grafo è connesso, ci sarà modo di ricongiungersi al circuito iniziale.

□

Definiamo, invece, **circuito hamiltoniano** un circuto che passa esattamente una volta su ogni vertice del grafo.

Un lemma utile è il seguente:

Lemma 3.41 (Handshaking lemma). *In ogni grafo, il numero di vertici di grado dispari è pari.*

Dimostrazione. Deve essere

$$\sum_{x \in V} d(x) = 2m$$

ma la parità di una sommatoria dipende solo dai numeri dispari, infatti gli addendi pari non cambiano la parità. Se tale somma è pari, è necessario che il numero di addendi dispari sia pari. □

Possiamo ora tornare al problema del commesso viaggiatore.

TravelingSalesman

Input:	un grafo $G = (V, E)$ e un costo $\forall e \in E \delta_e$
Output:	Insieme ordinato di lati
Problema:	Qual è il circuito hamiltoniano di minor costo?
Ammissibili:	Insieme ordinato di lati che formi un circuito hamiltoniano
Tipo:	Min
Costo:	$\sum_{e \in \pi} \delta_e$

3.7.2 Algoritmo di Christofides

TSP su clique

Si noti che non è necessario che esistano delle soluzioni ammissibili! Per facilitare l'analisi e ottenere risultati migliori specializzeremo il problema in un certo modo: analizzeremo il TSP su *clique* (cricche), ossia un grafo $G = (V, \binom{V}{2})$.

In quanto non è necessariamente vero che il grafo sia una cricca, supponiamo di avere un grafo pesato non completo: lo trasformiamo in un grafo completo

$$G = (V, E), \delta_e \quad \rightsquigarrow \quad K = (V, \binom{V}{2}), \bar{\delta}_e$$

definendo

$$\bar{\delta}_e = \begin{cases} \delta_e & e \in E \\ 1 + \sum_{e \in E} \delta_e & e \notin E \end{cases}$$

Se si trova una soluzione per K che non utilizza nessun lato fittizio, chiaramente tale soluzione è valida anche per G ed è anche ottima, poiché nessun circuito hamiltoniano può costare più di anche solo un lato fittizio. In altre parole, la soluzione ottima coinvolge un lato fittizio se e solo se per K non vi sono soluzioni ammissibili.

TSP metrico su clique

Tuttavia, anche sulle clique il TSP è un problema estremamente complesso da risolvere e, in generale, non è approssimabile a meno di una costante. Con un ulteriore rilassamento riusciremo ad approssimare TSP, ossia imponendo che le distanze formino una *metrica* su G: richiediamo che G sia una cricca e δ_e sia una metrica, ossia

$$\delta_{ij} \leq \delta_{ik} + \delta_{kj}$$

Prima di designare l'algoritmo risolvente, introduciamo brevemente due problemi che saranno utili.

Minimo albero ricoprente

MinimumSpanningTree
Input: $G = (V, E)$ *bipartito*
Output: Insieme di lati
Problema: Qual è l'insieme di archi che copre i vertici con un costo minore?
Ammissibili: L'insieme di lati è un albero, ossia un grafo connesso e aciclico
Tipo: Min
Costo: Cardinalità dell'insieme di lati

Questo problema è risolvibile esattamente dall'algoritmo di Kruskal in tempo $O(m \log(n))$.

Matching perfetto a costo minimo

MinimumWeightPerfectMatching
Input: $G = (V, E)$ con un numero pari di vertici
Output: Insieme di lati
Problema: Esiste un matching perfetto?
Ammissibili: Insieme di lati che formano un matching, ossia nessun vertice compare più di una volta, perfetto, ossia
Tipo: Min
Costo: Somma dei pesi degli archi scelti

Anche questo problema è risolvibile in tempo polinomiale: un algoritmo famoso è l'algoritmo *dell'infiorescenza* che ha complessità $O(m \log(n))$.
Possiamo ora passare all'algoritmo per risolvere istanze di TravelingSalesman su grafi completi dotati di una distanza metrica.

Algoritmo 10: ChristofidesTSP

Input: grafo $G = (V, \binom{V}{2})$ con pesi δ_e che formano una metrica

```
1  T = FindMST(G)
   /* D è l'insieme dei vertici di grado dispari nel minimo albero
      ricoprente T. Per il Lemma 3.41, è |D| mod 2 = 0.            */
2  D = FindOddDegreeVertices(T)
   /* G[D] è il grafo ristretto sui nodi di D                      */
3  G[D] = G(V ∩ D, ···)
4  M = FindPerfectMatching(G[D])
   /* È possibile che lo stesso lato appaia due volte, rendendo H un
      multigrafo. Tutti i vertici in H hanno grado pari, poiche' quelli
      che in D hanno grado dispari hanno un nuovo lato.            */
5  H = T ∪ M
6  π = FindEulerianWalk(H)
7  R = FindRepeatingVertices(π)
8  for v : R do
       /* per ogni vertice v ripetuto nel cammino si cancellano due lati
          (uno entrante e uno uscente) e, siccome il grafo è una cricca, si
          inserisce un nuovo lato che collega i due vertici disconnessi
          (quello che portava a v e quello raggiunto da v)         */
9      π = RemoveAndReplace(π, v)
10 return π
```

Lemma 3.42. *Il costo dell'albero T su G è minore o uguale del costo ottimale del cammino hamiltoniano su G metrico e completo:*

$$\delta(T) \leq \delta^*$$

Dimostrazione. Sia π^* un circuito hamiltoniano ottimo. Sia e un qualunque lato che compare in π^* e si consideri $\pi^* \setminus e$: il risultato è uno spanning tree (possibilmente minimo). Pertanto,

$$\delta(T) \leq \delta(\pi^* \setminus e) \leq \delta^*$$

poiché T è *un* minimo albero ricoprente. □

Lemma 3.43.

$$\delta(M) \leq \frac{1}{2}\delta^*$$

Dimostrazione. Sia π^* un circuito hamiltoniano ottimo. Dal Lemma 3.41 sappiamo che un numero pari di vertici appare in D come costruito nell'algoritmo: sia quindi π' un qualunque circuito sui vertici di D. Chiaramente, siccome δ è una metrica,

$$\delta(\pi') \leq \delta(\pi^*)$$

Dividiamo i lati di π' in due insiemi M_1 e M_2, in modo che si alternino nel cammino: essi sono due perfect matching su D. Allora

$$\begin{aligned} &\delta(M_1) \geq \delta(M) \wedge \delta(M_2) \geq \delta(M) \\ &\implies \delta^* \geq \delta(\pi') = \delta(M_1) + \delta(M_2) \geq 2\delta(M) \end{aligned}$$

□

Teorema 3.44. *L'algoritmo di Christofides è una $\frac{3}{2}$-approssimazione per il problema del commesso viaggiatore su grafi completi con distanza metrica.*

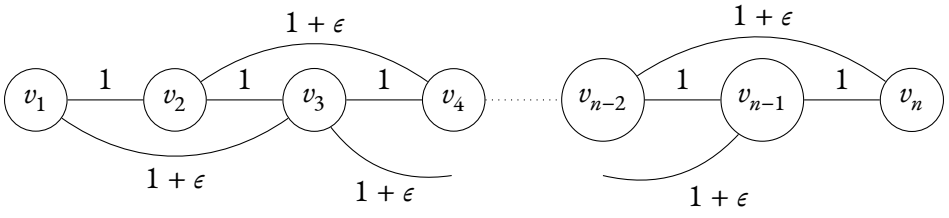
Dimostrazione. Siano $\tilde{\pi}$ il cammino hamiltoniano e π il cammino euleriano costruiti dall'algoritmo. Allora deve essere $\delta(\tilde{\pi}) \leq \delta(\pi)$: π passa per tutti gli archi di H esattamente una volta:

$$\delta(\pi) = \sum_{e \in H} \delta(e) = \delta(M) + \delta(T) \leq \underbrace{\frac{1}{2}\delta^*}_{\text{Lemma 3.43}} + \underbrace{\delta^*}_{\text{Lemma 3.42}} = \frac{3}{2}\delta^*$$

□

Teorema 3.45. *L'analisi di approssimazione di TSP metrico su clique con Christofides è stretta.*

Dimostrazione. Dato n pari ed $\epsilon \in (0, 1)$, esibiamo il seguente grafo:



Tutti i lati mancanti hanno costo pari al costo del cammino minimo tra i due vertici del lato. L'algoritmo di Christofides seleziona il MinimumSpanningTree T, ossia il cammino composto da lati di costo 1,

quindi $\delta(T) = n - 1$. I nodi di grado dispari in T sono i due estremi, quindi $D = \{v_1, v_n\}$. Il cammino minimo che li collega è un singolo arco che ha peso $\delta(M) = (1+\epsilon)\frac{n}{2} + 1$.

Al termine dell'algoritmo, il costo del circuito hamiltoniano ottenuto è

$$\delta = n - 1 + (1+\epsilon)\frac{n}{2} + 1 = \frac{3}{2}n + \frac{\epsilon n}{2}$$

Il costo del cammino ottimo è

$$\delta^* = (1+\epsilon)\frac{n}{2} + (1+\epsilon)\frac{n}{2} + 2$$

quindi

$$\frac{\delta}{\delta^*} = \frac{\frac{3}{2}n + \frac{\epsilon n}{2}}{(1+\epsilon)n + 2} = \frac{\frac{3}{2}n + \frac{n}{2}\epsilon}{n + 2 + \epsilon n} = \frac{3}{2}$$

per $n \to \infty$, indipendentemente da ϵ. □

3.7.3 Inapprossimabilit□ di TSP

La situazione non è altrettanto positiva per il caso generale del TSP:

Teorema 3.46. *Decidere se un grafo contiene un cammino hamiltoniano è un problema in* **NP − completi**.

Teorema 3.47. *Non esiste alcun α tale che* TravelingSalesman *sia α-approssimabile a meno che* **P** ≠ **NP**.

Dimostrazione. (per assurdo.) Sia $G = (V, E)$ un grafo che si completa creando G': su questo grafo definiamo una nozione di distanza

$$d(x,y) = \begin{cases} 1 & x, y \in E \\ \lceil \alpha n \rceil + 1 & x, y \notin E \end{cases}$$

Se G ammette un circuito hamiltoniano, in G' quel circuito ha costo n, poiché tocca tutti i vertici concludendo il circuito. Se G non ammette un circuito hamiltoniano su di esso, possiamo concludere che in G' tutti i circuiti hamiltoniani passano per almeno un lato di costo $\lceil \alpha n \rceil + 1$, quindi il circuito del costo hamiltoniano minimo è almeno $\lceil \alpha n \rceil + 1$. Se G ha un circuito hamiltoniano l'algoritmo α approssimante per trovare cammini hamiltoniani in G' (che, per assurdo, assumiamo esistere), troverà un circuito di costo minore o uguale a αn (poiché è α-approssimante); se G non ammette un circuito hamiltoniano, troverà un circuito di costo maggiore di $\lceil \alpha n \rceil + 1$. È impossibile che $\alpha < \lceil \alpha n \rceil + 1$, altrimenti sapremmo decidere se G ammette circuiti hamiltoniani. Ossia, deve essere

$$\alpha > \frac{\lceil \alpha n \rceil + 1}{n} \geq \frac{\alpha n + 1}{n} = \alpha + \frac{1}{n}$$

ossia $\alpha \geq \alpha + \frac{1}{n}$, impossibile. Concludiamo che TravelingSalesman $\notin$ **APX**. □

3.8 Problema del 2-carico

In LoadBalancing, l'input era composto da $t_0, t_1, \cdots, t_{n-1} \in \mathbb{N}^+$ tasks e un numero m di macchine. L'obiettivo era costruire degli assegnamenti tali per cui il carico massimo di una macchina è il minimo possibile. La versione 2-LoadBalancing è una specializzazione in cui $m = 2$.

Algoritmo 11: PartitionBalance

Input: $m_1, m_2, t_0, ..., t_{n-1}, \epsilon$
1 **if** $\epsilon > 1$ **then**
2 $m_1.tasks = \{t_0, \cdots, t_{n-1}\}$
3 **return**
4 $tasks = [t_0, \cdots, t_{n-1}].nonDecrescingSort()$
5 $k = \lceil \frac{1}{\epsilon} - 1 \rceil$
/* Esegui l'algoritmo esaustivo sui primi k task */
6 $optPartition = findOptimalPartition(tasks[0 \cdots k-1])$
7 $GreedyBalance(m_1, m_2, tasks[k-1 \cdots])$

3.8.1 Algoritmo PTAS

Due algoritmi per risolvere LoadBalancing sono stati proposti: greedy (Algoritmo 3) o con ordinamento iniziale delle task (Algoritmo 4). Ora, faremo molto meglio descrivendo un algoritmo che porta 2-LoadBalancing in **PTAS**: daremo quindi un tasso di approssimazione vincolante per la soluzione trovata - tuttavia l'algoritmo risulterà esponenziale in tale tasso.

Teorema 3.48. *L'Algoritmo 11 è polinomiale in n (ma non in ϵ) e produce una $1+\epsilon$ approssimazione per 2-LoadBalancing.*

Dimostrazione. Se $\epsilon \geq 1$, assegnare tutte le task ad una sola macchina non può essere peggio del doppio del costo ottimale. Altrimenti, proseguiamo seguendo l'esecuzione dell'algoritmo. I primi k task vengono assegnati in modo ottimale. I seguenti $n-k$ task vengono assegnati in maniera greedy. Assumiamo, senza perdita di generalità, che $w(m_1) > w(m_2)$. Sia h l'indice dell'ultimo task assegnato alla macchina m_1. Abbiamo due casi:

- $h < k$. Tutti i task assegnati in maniera greedy appartengono alla macchina m_2. Siccome i task assegnati a m_1 sono assegnati in modo ottimale, il costo massimo $w(m_1)$ è ottimale.
- $h \geq k$. Dopo la fase ottima la macchina m_1 riceve altri task. Sia $L = \sum_i t_i$.

$$
\begin{aligned}
& w(m_1) - t_h \geq w(m_2) \text{ nel momento in cui si assegna } h \\
& \implies 2 * w(m1) - t_h \leq w(m_1) + w(m_2) \implies 2w(m_1) - t_j \leq 2L \\
& \implies w(m_1) - \frac{t_h}{2} \leq L \\
& 2L = 2 * (t_0 + t_1 + \cdots + t_k + t_h + \cdots + t_{n-1}) \geq t_h(k+1) \\
& \implies \frac{w(m_1)}{w^*} \leq \frac{w(m_1)}{L} \leq \frac{\frac{t_h}{2} + L}{L} = 1 + \frac{t_h}{2L} \leq 1 + \frac{t_h}{t_h \cdot (k+1)}
\end{aligned}
$$

Ma $k = \lceil \frac{1}{\epsilon} \rceil - 2$, quindi $k + 1 \geq \frac{1}{\epsilon}$:

$$
\frac{\frac{t_h}{2} + L}{L} = 1 + \frac{t_h}{2L} \leq 1 + \frac{t_h}{t_h \cdot (k+1)} \leq 1 + \frac{1}{1+k} \leq 1 + \frac{1}{\frac{1}{\epsilon}} = 1 + \epsilon
$$

□

Teorema 3.49. *L'algoritmo ha tempo d'esecuzione $O(n \log n + 2^{\frac{1}{\epsilon}})$.*

Corollario 3.50. *Essendo uguale a 2-LoadBalancing, MinimumPartition $\in$* **PTAS**.

3.9 Problema dello zaino

KNAPSACK	
Input:	n oggetti con valori $v_0, \cdots, v_{n-1} \in \mathbb{N}$ e pesi $w_0, \cdots, w_{n-1} \in \mathbb{N}$ e una capacità $W \in \mathbb{N}$
Output:	Insieme di oggetti S
Problema:	Qual è l'insieme di oggetti di valore maggiore che si può scegliere senza eccedere la capacità W?
Ammissibili:	Scelta di oggetti che non eccedono W: $\sum_{i \in S} w_i \leq W$
Tipo:	Max
Costo:	Valore degli oggetti in S: $\sum_{i \in S} v_i$

Teorema 3.51. $KnapsackProblem \in$ **NPO – completi**.

3.9.1 Algoritmo esponenziale basato su programmazione dinamica

Come solitamente accade quando si desidera trovare un algoritmo basato sulla *programmazione dinamica*, suddividiamo il problema in problemi più piccoli: costruiamo una matrice

$$vOPT[i, w] = \text{ massimo valore di } i \text{ oggetti con zaino di capacità } w$$

con $i \leq n$ e $w \leq W$. Ovviamente, ciò che ci interessa è $vOPT[n, W]$, ossia il valore massimo ottenibile considerando tutti gli n oggetti e con capacità W. In quanto il valore ottenibile scegliendo 0 oggetti è 0, abbiamo che, per qualsiasi capacità, $vOPT[0, \_] = 0$ - analogamente, siccome nessun oggetto può essere scelto se la capacità è 0, deve essere $vOPT[\_, 0] = 0$.

L'entry della $i+1$-esima riga nella $w+1$-esima colonna si costruisce decidendo se inserire o meno l'i-esimo oggetto:

$$vOPT[i+1, w] = \begin{cases} vOPT[i, w] & w_i > w \\ vOPT[i, w - w_i] + v_i & w_i \leq w \end{cases}$$

Questo algoritmo, ovviamente, non può essere polinomiale (altrimenti sarebbe **P** = **NP**) – è vero che il numero di entry nella matrice è $n \cdot w$, ma l'algoritmo non è polinomiale nella lunghezza binaria dell'input W, bensì è esponenziale, rendendo quindi l'algoritmo *pseudopolinomiale*.

3.9.2 Algoritmo FPTAS basato su programmazione dinamica

Per cercare di ovviare al problema della pseudopolinomialità del metodo precedente, scomponiamo il problema in termini di oggetti e valore (invece che peso):

$$wOPT[i, v] = \text{minimo peso necessario per i primi } i \text{ oggetti con valore } \geq v$$

In $wOPT$ le colonne rappresentano valori tra $[0, \sum_i v_i]$ - in realtà, approssimiamo questo range con $[0, n \cdot v_{max}]$, con $v_{max} = \max_i v_i$.

Sull'ultima riga troveremo il minimo peso necessario per scegliere n oggetti; potrà accadere che per molte colonne $wOPT[i, v] > W$, che rappresentanos soluzioni non accettabilil; dovremo quindi cercare la prima entry che non sfora la capacità W. La prima colonna sarà $wOPT[\_, 0] = 0$, mentre, inizialmente, si imposta $wOPT[0, \geq 1] = \infty$.

La regola di riempimento che definiamo è

$$wOPT[i+1, v] = \min(wOPT[i, v], wOPT[i, \max(v - v_i, 0)] + w_i)$$

Benché apparentemente sembra non ci sia alcun vantaggio, in questo frangente possiamo operare delle modifiche sulla matrice: l'idea è quella di "schiacciare" le colonne, operando una divisione o un cambio di

misura, nonostante venga in questo modo introdotta un'approssimazione dei valori. Introduciamo, quindi, un *valore di scala*:

$$\theta = \frac{\epsilon v_{max}}{2n}$$

e l'obiettivo finale sarà avere una $1+\epsilon$-approssimazione. Sia quindi $X = (v_i, w_i, W)$ l'input del problema; siano

$$\bar{v}_i = \lceil \frac{v_i}{\theta} \rceil \cdot \theta, \quad \hat{v}_i = \lceil \frac{v_i}{\theta} \rceil$$

ai quali associamo i relativi problemi $\bar{X} = (\bar{v}_i, w_i, W)$ e $\hat{X} = (\hat{v}_i, w_i, W)$ che avranno delle soluzioni ottime $v^*, \bar{v}^*$ e $\hat{v}^*$, derivanti da insiemi $S^*, \bar{S}^*$ e $\hat{S}^*$.

Osservazione 3.52. *Banalmente,*

$$\bar{v}^* = \theta \hat{v}^*$$

In altre parole, risolvere $\hat{X}$ o risolvere $\bar{X}$ restituisce le stesse soluzioni, pertanto

$$\bar{S}^* = \hat{S}^*$$

Lemma 3.53. *Sia S una soluzione ammissibile per il problema. Allora*

$$(1+\epsilon) \sum_{i \in \hat{S}^*} v_i \geq \sum_{i \in \bar{S}^*} v_i$$

Dimostrazione.

$$\begin{aligned}
&\sum_{i \in S} v_i \leq \sum_{i \in S} \bar{v}_i \text{ grazie all'arrotondamento per eccesso} \\
&\leq \sum_{i \in \bar{S}^*} \bar{v}_i \text{ poiché è la soluzione ottima} \\
&= \sum_{i \in \hat{S}^*} \bar{v}_i \text{ poiché } \hat{S}^* = \bar{S}^* \text{ da Osservazione 3.52} \\
&= \sum_{i \in \bar{S}^*} \bar{v}_i \leq \sum_{i \in \hat{S}^*} (v_i + v) \leq \sum_{i \in \hat{S}^*} v_i + n\theta = \sum_{i \in \hat{S}^*} v_i + n\frac{\epsilon v_{max}}{2n}
\end{aligned}$$

quindi

$$\sum_{i \in S} v_i \leq \sum_{i \in \hat{S}^*} v_i + \frac{\epsilon v_{max}}{2}$$

In particolare, questo vale per la soluzione composta solamente dall'oggetto con valore massimo $S = \{max\}$, da cui segue

$$\begin{aligned}
&v_{max} \leq \sum_{i \in \hat{S}^*} v_i + \frac{\epsilon v_{max}}{2} \leq \sum_{i \in \hat{S}^*} v_i + \frac{v_{max}}{2} \text{ poiché } \epsilon \leq 1 \\
&\implies \sum_{i \in \hat{S}^*} v_i \geq \frac{v_{max}}{2} \\
&\implies \sum_{i \in S} v_i \leq \sum_{i \in \hat{S}^*} v_i + \frac{\epsilon v_{max}}{2} \leq \sum_{i \in \hat{S}^*} v_i + \epsilon \sum_{i \in \hat{S}^*} v_i = (1+\epsilon) \sum_{i \in \hat{S}^*} v_i
\end{aligned}$$

□

Teorema 3.54.

$$(1+\epsilon) \sum_{i \in \hat{S}^*} v_i \geq \sum_{i \in S^*} v_i = v^*$$

Risolvendo il problema $\hat{X}$ si ottiene una soluzione il cui valore per il problema originale è $\frac{1}{1+\epsilon}$ volte l'ottimo.

Algoritmo 12: FPTASKnapsack

Input: $X = (v_i, w_i, W), \epsilon$

1 $\hat{X} = getFrom(X, \epsilon)$
/* La soluzione così trovata è una $(1+\epsilon)$−approssimazione */

2 **return** $solveWithWOpt(\hat{X})$

Dobbiamo ora convincerci che Algoritmo 12 termini in tempo polinomiale: l'ultima colonna sarà $n\hat{v}_{max}$; sappiamo che

$$v_{max} = \lceil \frac{v_{max}}{\theta} \rceil = \lceil \frac{v_{max} n}{\epsilon v_{max}} \rceil = \lceil \frac{n}{\epsilon} \rceil$$

pertanto il numero di colonne

$$n\hat{v}_{max} \leq \frac{n^2}{\epsilon} + n$$

polinomiale nell'input e in ϵ.

CAPITOLO 4

Algoritmi probabilistici

Fino ad ora abbiamo considerato il modello di calcolo delle macchine di Turing per scrivere algoritmi per problemi di ottimizzazione, in particolare algoritmi di approssimazione: gli algoritmi sono quindi **determi-**

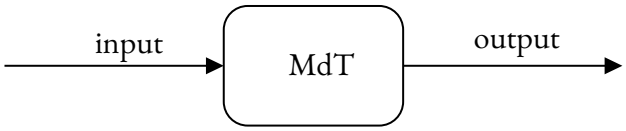


Figura 4.1: Macchina di Turing deterministica

nistici, nonostante in alcune situazioni gli algoritmi possano fare scelte "arbitrarie", che formalizziamo con la nozione di gradi di libertà - abbiamo tuttavia dimostrato che le proprietà sono indipendenti da queste scelte arbitrarie.

Dobbiamo riservare il termine di **non determinismo** per definire classi di complessità per modelli di calcolo *non realistici*. Estendiamo ora il modello: abbiamo una MdT che è anche in grado di leggere un nastro su cui sono scritti dei valori casuali, chiamato **sorgente aleatoria**.

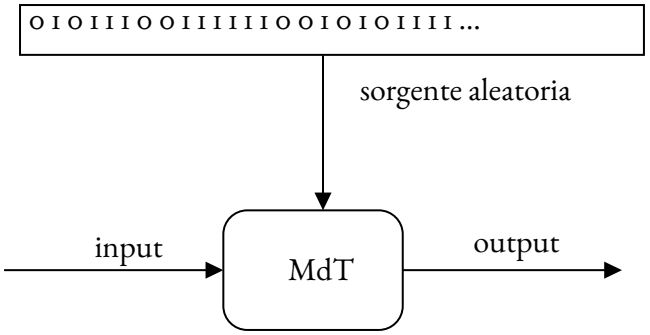


Figura 4.2: Macchina di Turing probabilistica

Un algoritmo così costruito è definito **probabilistico**, in quanto l'output sarà in funzione dell'input e del seme casuale. L'algoritmo possiede quindi una certa distribuzione associata

$$P[Y = y|X = x]$$

ossia la probabilità di avere l'output y per un input x. Gli algoritmi probabilistici si dividono in due famiglie: gli algoritmi **Monte-Carlo** in cui l'output è probabilistico e gli algoritmi **Las Vegas**, in cui l'output è deterministico, ma il tempo di esecuzione è probabilistico. In particolare, studieremo la prima famiglia.

Questo modello di calcolo si può applicare sia a problemi di decisione che di ottimizzazione; per applicare questi algoritmi, vi sono due varianti: nel primo caso l'algoritmo mira ad ottenere l'ottimo con una certa probabilità, idealmente alta - essi possono tuttavia fallire arbitrariamente male. Alternativamente, l'algoritmo può mirare a ottenere un'*approssimazione* dell'ottimo con una certa probabilità.

4.1 Problema del taglio minimo

MinimumCut
Input: $G = (V, E)$
Output: Sottoinsieme $X \subseteq V$
Problema: Qual è il *taglio* minore?
Ammissibili: $X \subseteq V$ tale che il numero di lati che hanno un vertice in X e un vertice in X^c (*tagliati*) è minimo; definia
Tipo: Min
Costo: $|E_x|$

Le soluzioni banali sono $X = V$ e $X = \emptyset$; inoltre, una soluzione sempre possibile è scegliere $X = \{v\}$ per un qualsiasi $v \in V$.

Teorema 4.1. *MinimumCutProblem* ∈ **NPO – completi**

4.1.1 Algoritmo di Karger

L'algoritmo di Karger utiliza l'operazione di *contrazione*: dato un grafo G, l'operazione $G \downarrow e$ su un lato $e = \{u, v\} \in E$ unisce i due vertici u e v, rimuovendo e.

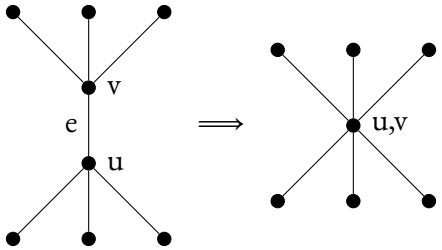


Figura 4.3: Contrazione $G \downarrow e$.

È necessario che G sia un multigrafo, poiché unendo u e v è possibile che un terzo vertice y fosse collegato sia a u che a v; rimarranno quindi dei lati paralleli. In caso il lato contratto fosse uno dei lati paralleli tra u e v anche i lati paralleli rimanenti vengono contratti.

Algoritmo 13: KargerMinimumCut

```
   Input: G = (V, E)
1  if ¬G.isConnected() then
2      return findConnectedComponent(G)
3  while |V| > 2 do
                /* Estrai un lato con una distribuzione uniforme e contrailo */
4      e = uniformExtraction(E)
5      G = G ↓ l
                                    /* Restituisci uno dei due vertici rimanenti */
6  return chooseOne(V)
```

Chiaramente, l'output dipende dalla scelta dei lati da contrarre, che è casuale. Sia quindi S^* il taglio minimo, k^* il numero di lati tagliati da S^* e la serie $G_1, \cdots, G_i$ la sequenza di grafi ottenuti per ogni contrazione operata dall'algoritmo.

Osservazione 4.2. $\forall i \ |G_i(V)| = n - i + 1 \wedge |G_i(E)| \leq m - i + 1$

Osservazione 4.3. *Per ogni i, ogni taglio in G_i è un taglio in G dello stesso costo.*

Osservazione 4.4. *Il grado minimo in G_i è maggiore o uguale a k^*.*

Dimostriamo l'Osservazione 4.4.

Dimostrazione.

$$2(m - i + 1) \geq 2 * |G_i(E)| = \sum_{v \in G_i(V)} d_{G_i}(v) \geq k^*(n - i + 1) \implies m - i + 1 \geq \frac{(n - i + 1) * k^*}{2}$$

□

Lemma 4.5. *Sia E_i l'evento "al passo i-esimo un lato $e \notin E_{S^*}$ viene contratto".*

$$\forall i \ P[E_i | E_1, \cdots, E_{i-1}] \leq \frac{n - i - 1}{n - 1 + 1}$$

Dimostrazione.

$$\begin{aligned} P[E_i | E_1, \cdots, E_{i-1}] &= 1 - P[\neg E_i | E_1, \cdots, E_{i-1}] \\ &\leq 1 - \frac{2 \cdot k^*}{(n - i + 1)k^*} = 1 - \frac{2}{n - i + 1} = \frac{n - i - 1}{n - i + 1} \end{aligned}$$

□

Teorema 4.6. *L'algoritmo di Karger emette l'ottimo con probabilità $p \geq \frac{1}{\binom{n}{2}}$.*

Dimostrazione.

$$\begin{aligned} &P[E_1 \wedge E_2 \wedge \cdots \wedge E_{n-2}] = P[E_1] \cdot P[E_2 | E_1] \cdot \cdots \cdot P[E_{n-2} | E_{n-3}, \cdots, E_1] \\ &= \frac{n - 1 - 1}{n - 1 + 1} \cdot \frac{n - 2 - 1}{n - 2 + 1} \cdot \cdots \cdot \frac{n - (n - 2) - 1}{n - (n - 2) + 1} \\ &= \frac{n - 2}{n} \cdot \frac{n - 3}{n - 1} \cdot \cdots \cdot \frac{1}{3} = \frac{\Pi_{i=1}^{n-2} i}{\Pi_{i=3}^{n} i} = \frac{2 \cdot 1}{n(n - 1)} = \frac{2}{n(n - 1)} = \frac{1}{\binom{n}{2}} \end{aligned}$$

□

Osservazione 4.7. *Si esegua l'algoritmo di Karger $\binom{n}{2} \log(n)$ volte. L'ottimo si ottiene con probabilità $p \geq 1 - \frac{1}{n}$.*

Serve una proprietà delle funzioni esponenziali:

$$\forall x \geq 1 \ \frac{1}{4} \leq (1 - \frac{1}{x})^x \leq \frac{1}{e}$$

Dimostrazione. La probabilità di non trovare l'ottimo in nessuna esecuzione è

$$p \leq (1 - \frac{1}{\binom{n}{2}})^{\binom{n}{2} \log(n)} \leq \frac{1}{e}^{\log(n)} = \frac{1}{n}$$

□

4.2 Problema della copertura d'insiemi

Abbiamo già definito il SetCoverProblem: dato una serie di insiemi

$$S_1, \cdots, S_m \subseteq U$$

con pesi

$$w_1 \cdots, w_m \in \mathbb{Q}^+$$

definiamo $n = |U|$; vogliamo trovare un $S \subseteq m$ tale che

$$\cup_{i \in S} S_i = U$$

e il suo costo $\sum_{i \in S} w_i$ sia minimo.

4.2.1 Algoritmo probabilistico basato sull'arrotondamento

Il problema può essere trasposto in un problema di programmazione lineare intera: creiamo delle variabili

$$x_1, \cdots, x_m$$

tali che

$$\begin{cases} x_j \leq 1 & \forall j = 1, \cdots, m \\ x_k \geq 0 & \forall j = 1, \cdots, m \\ \sum_{i:u \in S_i} x_i \geq 1 & \forall u \in U \end{cases}$$

Chiamiamo questo problema associato a Π Π_{IPL}. È importante richiamare ora alcune proprietà statistiche.

Teorema 4.8 (Disuguaglianza di Markov). *Per ogni variabile aleatoria X non negativa e per ogni $\alpha > 0$*

$$P[X \geq \alpha] \leq \frac{E[X]}{\alpha}$$

Teorema 4.9 (Union bound o disuguaglianza di Boole).

$$P[\cup_i E_i] \leq \sum_i P[E_i]$$

Algoritmo 14: ProbabilisticRoundingSetCover

Input: $S_1, \cdots, S_m, w_1, \cdots, w_m$ e un intero k

1 $\hat{x_1}, \cdots, \hat{x_m} = solve(\Pi_{PL})$
2 $S = \emptyset$
3 **for** $t = \{1, \lceil k + \log(n) \rceil\}$ **do**
4 **for** $i = 1, ..., m$ **do**
5 $S = probInsert(S, i, \hat{x_i})$ `/* Inserisci` i `in` S `con probabilità` $\hat{x_i}$ `*/`
6 **return** S

L'Algoritmo 14 potrebbe trovare una soluzione non ammissibile. Dimostriamo ora che *spesso* è ammissibile e che quando è ammissibile è una soluzione molto buona.

Teorema 4.10. *La probabilità che l'Algoritmo 14 produca una soluzione ammissibile è*

$$p \geq 1 - e^{-k}$$

parametrica in k, che determina il numero di tentativi di inserimento.

Dimostrazione.

$$P[\text{soluzione ammissibile}] = 1 - P[\text{almeno un elemento dell'universo non è coperto}]$$

chiamiamo B_u l'evento per cui u non è coperto nella soluzione. Allora

$$P[\text{soluzione ammissibile}] = 1 - P[\cup_{u \in U} B_u]$$

Possiamo quindi usare il Teorema 4.9:

$$1 - P[\cup_{u \in U} B_u] \geq 1 - \sum_{u \in U} P[B_u]$$

L'elemento u non è coperto quando nessuno degli elementi che contentono u non è stato scelto:

$$1 - \sum_{u \in U} \Pi_{i:u \in S_i} P[S_i \text{ non è stato scelto}] = 1 - \sum_{u \in U} \Pi_{i:u \in S_i} (1 - \hat{x_i})^{\lceil k + \log(n) \rceil} \geq$$
$$\geq 1 - \sum_{u \in U} \Pi i : u \in S_i e^{-\hat{x_i}(k + \log(n))} = 1 - \sum_{u \in U} \exp(-(k + \log(n)) \sum_{i:u \in S_i} \hat{x_i})$$

Siccome S_i è ammissibile, deve essere $\sum_{i:u \in S_i} \hat{x_i} \geq 1$, quindi

$$\geq 1 - \sum_{u \in U} e^{-(k + \log(n))} = 1 - \sum_{u \in U} \frac{e^{-k}}{n} = 1 - e^{-k} \frac{1}{n} |U| = 1 - e^{-k}$$

□

Teorema 4.11.

$$\forall \alpha \ 0 \leq \alpha \leq 1 \ P[\frac{v_{out}}{v^*} \geq \alpha(k + \log(n))] \leq \frac{1}{\alpha}$$

Dimostrazione. Abbiamo che $\hat{v} = \sum_i w_i \hat{x_I} \leq v^*$; inoltre, la probabilità che S_i venga scelto è

$$P[S_i \text{ sia scelto}] = P[\cup_t S_i \text{ sia scelto durante l'iterazione } t]$$
$$\leq \sum_t P[S_i \text{ sia scelto durante l'iterazione } t] = (k + \log(n)) \hat{x_I}$$

Per poter utilizzare il Teorema 4.8 calcoliamo il valore atteso di v_{out}:

$$E[v_{out}] = E[\sum_{i \in S} w_i] = \sum_i w_i P[S_i \text{ sia nell'output}] \text{ (per la linearità del valore atteso)}$$
$$\leq \sum_i w_i (k + \log(n)) \hat{x_i} = \hat{v}(k + \log(n))$$

$$P[\frac{v_{out}}{v^*} \geq \alpha(k + \log(n))] \leq \frac{E[v_{out}]}{\alpha(k + \log(n)) v^*} \leq \frac{v^*(k + \log(n))}{\alpha(k + \log(n)) v^*} = \frac{1}{\alpha}$$

□

Osservazione 4.12. *Se si esegue l'Algoritmo 14 con $k = 3$ c'è il 45% di probabilità di ottenere una soluzione ammissibile con fattore di approssimazione $\frac{v_{out}}{v^*} \leq 6 + 2\log(n)$.*

Dimostrazione. Sia $E_{ammissibile}$ l'evento per cui si ottiene una soluzione ammissible e E_{buona} l'evento per cui l'output sia entro $6 + 2\log(n)$ dall'ottimo. Ci interessa

$$P[E_{ammissibile} \wedge E_{buona}] = 1 - P[\neg E_{ammissible} \vee \neg E_{buona}] \geq 1 - P[\neg E_{ammissibile}] - P[\neg E_{buona}]$$
$$\geq 1 - e^{-3} - \frac{1}{2} \approx 0.45 \quad \text{(dal Teorema 4.11)}$$

□

4.3 Problema MaxEkSat

MaxEkSat è la versione k-indicizzata di MaxSat: date n clausole di k letterali ciascuna, l'obiettivo è massimizzare il numero di clausole soddisfatte.

Definiamo $x_1, \cdots, x_n$ le formule che compaiono nella formula e $c_1, \cdots, c_t$ le clausole della formula.

Teorema 4.13. $MaxEkSat \in$ **NPO − completi** *per* $k \leq 3$.

4.3.1 Algoritmo probabilistico

Teorema 4.14. *Un assegnamento casuale soddisfa, in media,*

$$E[T] = \frac{2^k - 1}{k} t$$

clausole.

Per la dimostrazione definiamo

$$X_i \; Unif\{0, 1\}$$

il valore assegnato ad ogni x_i probabilisticamente;

$$C_i = \begin{cases} 0 & C_i \text{ non soddisfatto} \\ 1 & \text{altrimenti} \end{cases}$$

mentre T è il numero di clausole soddisfatte. E l'algoritmo si svolge, banalmente, estraendo un valore per ogni x_i e restituendo il numero di clausole soddisfatte. Prima di procedere con la dimostrazione, ricordiamo la seguende legge statistica:

Teorema 4.15. *Sia $\{\mathcal{E}_i\}_{i=1..k}$ una partizione dell'universo degli eventi. Allora*

$$E[X] = \sum_{i=1}^{k} E[X|\mathcal{E}_i]P[\mathcal{E}_i]$$

Dimostrazione. La partizione $\{\mathcal{E}_i\}_i$ nel contesto di MaxEkSat è l'insieme dei possibili assegnamenti $X_1 = b_1, \cdots, X_n = b_n$; pertanto

$$\begin{aligned}
E[T] &= \sum_{b_1 \in 2} \cdots \sum_{b_n \in 2} E[T | X_i = b_1, \cdots, X_n = b_n] P[X_1 = b_1, \cdots, X_n = b_n] \\
&= \sum_{b_1 \in 2} \cdots \sum_{b_n \in 2} E[T | X_i = b_1, \cdots, X_n = b_n] P[X_1 = b_1] \cdots P[X_n = b_n] \\
&= \frac{1}{2^n} \sum_{b_1 \in 2} \cdots \sum_{b_n \in 2} E[T | X_i = b_1, \cdots, X_n = b_n] \\
&= \frac{1}{2^n} \sum_{j=1}^{t} (\sum_{b_1 \in 2} \cdots \sum_{b_n \in 2} E[C_j | X_i = b_1, \cdots, X_n = b_n]) \\
&= \frac{1}{2^n} \sum_{j=1}^{t} (2^n - 2^{n-k}) = \frac{2^n - 2^{n-l}}{2^n} t = \frac{2^k - 1}{k} t
\end{aligned}$$

□

4.3.2 Algoritmo derandomizzato

Teorema 4.16. *Per ogni $j = 0, \cdots, n$ esistono $b_1, \cdots b_j \in 2$ tali che*

$$E[T | X_1 = b_1, \cdots, X_j = b_j] \geq \frac{2^k - 1}{2^k} t$$

Questo significa che non solo il numero atteso di clausole è *alto*, ma possiamo inoltre fissare le prime j variabili in modo che questa proprietà continui ad essere preservata.

Dimostrazione. Per induzione su j. Base: per $j = 0$ verificato dal Teorema 4.14. Passo induttivo: per ipotesi induttiva vale

$$E[T | X_1 = b_1, \cdots, X_{j-1} = b_{j-1}] \geq \frac{2^k - 1}{k}$$

Per il Teorema 4.15 vale

$$\begin{aligned}
&E[T | X_1 = b_1, \cdots, X_{j-1} = b_{j-1}] = \\
&= E[T | X_1 = b_1, \cdots X_{j-1} = b_{j-1}, X_j = 0] P[X_j = 0] + E[T | X_1 = b_1, \cdots X_{j-1} = b_{j-1}, X_j = 1] P[X_j = 1] \\
&= \frac{1}{2} E[T | X_1 = b_1, \cdots X_{j-1} = b_{j-1}, X_j = 0] + \frac{1}{2} E[T | X_1 = b_1, \cdots X_{j-1} = b_{j-1}, X_j = 1] \\
&= \frac{1}{2} \alpha_0 + \frac{1}{2} \alpha_1
\end{aligned}$$

Deve essere che per qualche $i \in 2$

$$\alpha_i \geq \frac{2^k - 1}{2^k} t$$

supponiamo α_0 e α_1 minori: allora

$$\frac{1}{2} \alpha_0 + \frac{1}{2} \alpha_1 < \frac{1}{2} \frac{2^k - 1}{2^k} t + \frac{1}{2} \frac{2^k - 1}{2^k} t = \frac{2^k - 1}{2^k} t$$

contraddicendo l'ipotesi induttiva. □

Algoritmo 15: DerandomMaxEkSat

```
1  D = ∅                                          // indici delle clausole già determinate
2  for i = 1, ··· , n do
3      Δ_0 = 0
4      Δ_1 = 0
5      ΔD_0 = ∅
6      ΔD_1 = ∅
7      for j = 1, ··· , t do
8          if j ∈ D then
9              continue
10         if x_i ∉ c_j then
11             continue
12         h = findGeq(i, c_j)                    // variabili con indice ≥ i in c_j
13         if x_i ∈ c_j ∧ x_i = 1 then
14             Δ_0 = Δ_0 − 1/2^h
15             Δ_1 = Δ_1 + 1/2^h
16             ΔD_1 = ΔD_1 ∪ {j}
17         else
18             Δ_0 = Δ_0 + 1/2^h
19             Δ_1 = Δ_1 − 1/2^h
20             ΔD_0 = ΔD_0 ∪ {j}
21     u = arg max_{0,1}{Δ_0, Δ_1}
22     x_i = u
23     D = D ∪ ΔD_u
```

Teorema 4.17. *L'Algoritmo 15 trova un assegnamento che soddisfa*

$$\frac{2^k - 1}{2^k} t$$

clausole.

Dimostrazione. Banale dal Teorema 4.16. □

Lemma 4.18. *L'Algoritmo 15 garantisce che quando l'i-esima variabile viene assegnata vale*

$$E[T | X_1 = x[1], \cdots, X_{i-1}[x_{i-1}]] \geq \frac{2^k - 1}{2^k} t$$

Dimostrazione. Per $i = 0$, questo è vero per il Teorema 4.16. Per il passo induttivo, supponiamo $E[T | X_1 = x_1, \cdots, X_{i-1} = x_{i-1}] \geq \frac{2^k-1}{2^k} t$. Che valore assegnare a X_i? Supponiamo che la clausola alla quale appartiene X_i abbia h variabili non assegnate X_j con $j \geq i$ (le rimanenti hanno assegnato il valore 0); ci sono allora 2^h possibili assegnamenti: di questi 2^h, soltanto uno (tutte le X_j false) rende falsa la clausola, mentre $\frac{2^h-1}{2^h}$ rendono la clausola vera. Se si assegna 1 a X_i allora avrà contributo 1. L'incremento di valore atteso sarà

$$\Delta E[T] = 1 - \frac{2^h - 1}{2^h} = \frac{1}{2^h}$$

Viceversa, se si assegna a X_i il valore 0 la clausola rimarrà incerta e le variabili libere si riducono di uno: le possibili combinazioni che renderanno vera la clausola saranno $2^{b-1} - 1$. In questo caso, l'incremento nel valore atteso è

$$\Delta E[T] = \frac{2^{b-1} - 1}{2^{b-1}} - \frac{2^b - 1}{2^b} = -\frac{1}{2^b}$$

□

Corollario 4.19. *L'Algoritmo 15 fornisce una* $\frac{2^k}{2^k-1}$*-approssimazione per* MaxEkSat*.*

Dimostrazione. Sia t^* il numero ottimo di clausole soddisfatte. Si ha ovviamente $t^* \leq t$. Abbiamo, allora

$$\frac{t^*}{t_{algo}} \leq \frac{t}{t_{algo}} \leq \frac{t}{\frac{2^k-1}{2^k}t} = \frac{2^k}{2^k - 1}.$$

□

4.4 Il teorema PCP

Un punto fondamentale nell'analisi degli algoritmi proposta in questo corso sarà l'analisi dell'inapprossimabilità di alcuni problemi. Per poter analizzare la questione sarà necessario introdurre uno dei più importanti teoremi della teoria della complessità dal teorema di Cook: il teorema PCP, *probabilistically checkable proofs*. Il punto di partenza per arrivare all'enunciato di PCP è descrivere un'estensione delle macchine di Turing deterministiche.

4.4.1 Macchine di Turing oracolari

Le MdT oracolari ricevono in input un certo $x \in 2^*$ e in output restituiscono un valore booleano, una risposta $True$ o $False$ per un certo problema di decisione. Queste MdT hanno però anche accesso ad una *stringa* o *nastro dell'oracolo* $w \in 2^*$; se vuole accedere alla stringa dell'oracolo, la MdT ha un nastro speciale, definito *nastro di query* sul quale all'occorrenza può scrivere un numero binario; una volta scritto tale numero, la macchina entra in uno stato speciale che "aspetta" la risposta dell'oracolo: la MdT userà il numero scritto sul nastro di query come posizione in cui leggere w, che conterrà un 1 o uno 0. La MdT passerà ad uno stato relativo al numero trovato in w:

$$\text{stato speciale di interrogazione: } q? \begin{cases} w[n] = 1 & \to q_1 \\ w[n] = 0 & \to q_0 \end{cases}$$

Le MdT con oracolo sono il modo moderno per definire le classi nondeterministiche di macchine di Turing.

Teorema 4.20. *Un linguaggio binario* $L \subseteq 2^*$ *appartiene a* **NP** *se e solo se esiste una MdT oracolare* v *tale che:*

- $v(x, w)$ *termina in un numero polinomiale nella lunghezza* $|x|$*; e*
- $\forall x \in 2^* \; \exists w \in 2^* : v(w, x) = True$ *se e solo se* $x \in L$*.*

4.4.2 Probabilistic checkers

Un'estensione delle MdT oracolari sono i *probablistic checkers*: anch'essi hanno accesso ad un oracolo e, in più, possono accedere ad una *sorgente di bit casuali* forniti su un nastro apposito. Nuovamente, questo verificatore emetterà un valore tra $True$ e $False$; il suo comportamento, ovviamente, dipenderà da x, w, e $r \in 2^*$, la stringa casuale.

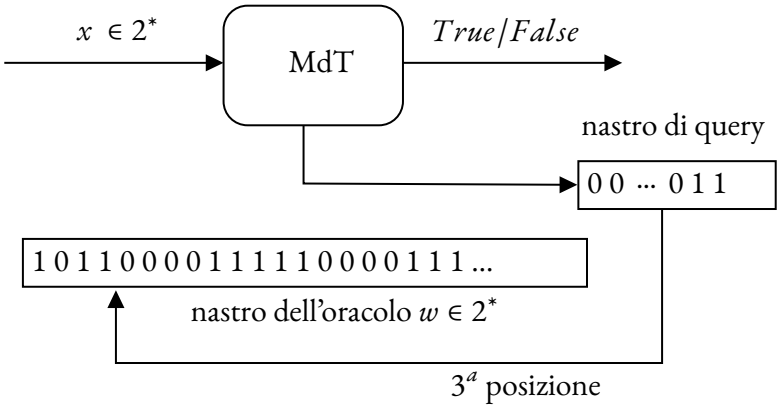


Figura 4.4: La MdT è dotata di un nastro di query sulla quale scrive un numero quando necessario e, in base al numero scritto, accede al nastro dell'oracolo.

Sottoclassi di PCP

In particolare, ci interessano i PC che effettuano un numero massimo di accessi alla stringa casuale e all'oracolo: definiamo $\mathbf{PCP}[r, q]$ come la classe dei verificatori che leggono al massimo r bit random ed effettuano al massimo q query all'oracolo. Utilizziamo inoltre la stessa notazione per identificare i linguaggi accettabili dalle macchine così definite: un linguaggio L è in $\mathbf{PCP}[r, q]$ se e solo se esiste una macchina $v \in \mathbf{PCP}[r, q]$ tale che

1. $v(x, R, W)$ funziona in tempo polinomiale in $|x|$,
2. $v(x, R, W)$ effettua al massimo un numero proporzionale a q e $|x|$ query,
3. $v(x, R, W)$ legge al massimo un numero proporzionale a r e $|x|$ bit casuali, infine
4. se $x \in L$ esiste una $w \in 2^*$ tale che v accetta x con probabilità 1 – cioè $v(x, -, W) = True$. Viceversa, se $x \notin L$, v rifiuta con probabilità $\geq \frac{1}{2}$ per qualunque w.

In altre parole, fissando x e w, a seconda di quale delle $2^{r(|x|)}$[1] possibili stringhe casuali è a disposizione la macchina v accetta o rifiuta: la probablità di accettazione è il numero di sequenze random per cui per gli specifici x e w si accetta sul numero totale di sequenze possibli.

Alcune sottoclassi sono interessanti: $\mathbf{PCP}[0, 0]$ è una normale macchina deterministica, pertanto la classe di linguaggi $\mathbf{PCP}[0, 0]$ è $\mathbf{P}$. $\mathbf{PCP}[0, Poly]$ è una macchina nondeterministica senza stringhe casuali che riconosce i linguaggi $\mathbf{PCP}[0, Poly]$, ossia $\mathbf{NP}$.

4.4.3 Enunciato di PCP

Teorema 4.21 (Arora, Safra 1998: PCP). $\boldsymbol{NP} = \boldsymbol{PCP}[O(log(n)), O(1)]$.

Dimostrazione. Omessa. □

In pratica, dato un $L \in \mathbf{NP}$, si può costruire un probabilistic checker v che necessita una quantità logaritmica in $|x|$ di bit casuali e che accede ad una stringa oracolare di lunghezza finita che funziona "come promesso": se $x \in L$ esiste un w per il quale v accetterà con probabilità 1, mentre se $x \notin L$ la macchina

[1] Accadrà di utilizzare la notazione $r(|x|)$ o $q(|x|)$ nonostante r e q siano state definite come costanti e non come funzioni: l'interpretazione è considerarle come funzioni che restituiscono un naturale proporzionale sia a x che a r (o q).

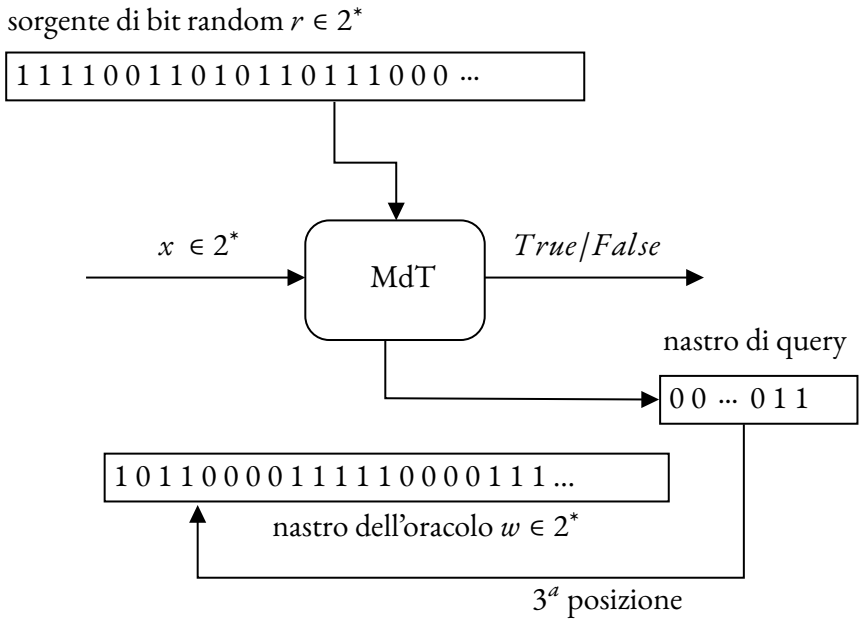


Figura 4.5: I probabilistic checkers hanno accesso alla sorgente casuale e alle informazioni dell'oracolo.

rifiuterà con probabilità almeno $\frac{1}{2}$. Questo determina il *tradeoff* tra casualità e nondeterminismo: è più utile avere informazione casuale piuttosto che la stessa "quantità" di nondeterminismo.

Si può inoltre notare che questo significa che l'albero di query nondeterministiche ha un'altezza finita e nota aprioristicamente.

Verificatori in forma normale

Assumeremo, senza perdita di generalità, che v usi *esattamente* $r(|x|)$ (che sarà sempre $O(\log n)$) bit random e che effettui *esattamente* $q \in \mathbb{N}$ query all'oracolo, ossia i probabilistic checker che esamineremo saranno $v \in \mathbf{PCP}[r(n), q]$ con $r(n) \in O(\log n)$. Inoltre, assumeremo anche che:

- v estrae tutti gli $r(|x|)$ bit random all'inizio;
- v, dopo aver estratto i bit random, effettua tutte le q query all'oracolo; le query, pertanto, non potranno essere adattive e il verificatore dovrà effettuare, in caso, tutte le 2^q possibili chiamate all'inizio.

Un verificatore che si comporta in questo modo è definito *verificatore in forma normale*.

Esemplificazione dei probabilistic checkers

Sia $L \in \mathbf{PCP}[r(n), q(n)]$. Analizziamo cosa accade, secondo il Teorema 4.21, per una qualsiasi $x \in L$; ipotizziamo che $r(|x|) = 17$, quindi vi saranno 2^{17} possibili r e $q(|x|) = 15$, quindi vi saranno 2^{15} possibili w.

$x \in L$ In questo caso, tra le 2^{15} possibili, deve esistere una stringa w tale per cui v, compiute le 15 richieste, accetta con probabilità 1. Per ognuna di queste esistono 2^{17} possibili stringhe random[2] e per ogni w ognu-

risposte dell'oracolo	$000\cdots000$	$\cdots$	$i_0i_1i_2\cdots i_{14}i_{15}i_{16}$	$\cdots$	$000\cdots000$
spazio dei bit random		$\cdots$		$\cdots$	

Tabella 4.1: Rappresentazione delle associazioni *w-r*: nelle aree di probabilità, la parte nera rappresenta le stringhe r che causano l'accettazione, mentre la parte bianca rappresenta le stringhe r che, tra tutte le 2^{17} possibili, causano la non accettazione.

na di queste può conferire una diversa probabilità di accettazione, tranne nel caso di "*quella*" stringa w che conferisce la probabilità di accettazione 1.

$x \notin L$ In questo caso, tra le 2^{15} possibili, non può esistere una stringa w tale per cui v, compiute le 15 richieste, accetta con probabilità maggiore di $1/2$.

risposte dell'oracolo	$000\cdots000$	$\cdots$	$i_0i_1i_2\cdots i_{14}i_{15}i_{16}$	$\cdots$	$000\cdots000$
spazio dei bit random		$\cdots$		$\cdots$	

4.5 Inapprossimabilit□

4.5.1 MaxEkSat

La prima applicazione del Teorema 4.21 che affronteremo è l'inapprossimabilità di MaxEkSat. La conclusione alla quale arriveremo è che l'Algoritmo 15 derandomizzato per MaxEkSat è *ottimo*, ossia non si può fare meglio di così. Il punto di partenza per questa dimostrazione è scegliere un linguaggio $L \in$ **NP − completi** (quindi anche $L \in$ **PCP**$[O(\log(n), O(1)]$ se **P** $\neq$ **NP**). Consideriamo la macchina $v \in$ **PCP**$[O(\log(n)), O(1)]$ e consideriamo un certo input $z \in 2^*$, per il quale genereremo una sequenza di $r(z)$ bit random – le sequenze possibili, che denotano lo *spazio probabilistico* $\mathcal{R}$ su z, sono $2^{r(z)}$. Per ogni specifica sequenza $R \in \mathcal{R}$, il verificatore produrrà q query all'oracolo:

$$i_1^R, i_2^R, \cdots, i_q^R$$

Per ognuna delle query il verificatore otterrà delle risposte in base alle quali l'input verrà accettato o meno. Definiamo quindi

$$f^R(w_{i_1^R}, w_{i_2^R}, \cdots, w_{i_q^R})$$

la funzione che, dati i bit alle posizioni i_j^R, restituisce *True* se v accetta dati $r = R \in \mathcal{R}$ e w con le relative query oppure *False* in caso contrario:

$$f^R(w_{i_1^R}, w_{i_2^R}, \cdots, w_{i_q^R}) = True \iff v(z, r, w) = True$$

L'idea è quindi descrivere il comportamento di f (e di conseguenza di v) come una formula booleana. In questo modo si dimostra che si può ridurre una qualsiasi istanza del problema del riconoscimento $z \in L$ per un arbitrario L in **NP − completi** ad un'istanza di MaxEkSat; introduciamo quindi $|w|$ variabili booleane $x_1, x_2, \cdots x_{|w|}$. La funzione f^R si può descrivere con una formula logica, che chiamiamo φ_z^R che utilizza come letterali proprio le variabili x_i che compongono w:

$$\varphi_z^R = (x_1 = 1 \vee x_2 = 0 \cdots) \wedge (x_1 = 1 \vee x_4 = 1 \vee x_8 = 0 \vee X_9 = 1) \wedge \cdots$$

[2]Descrivendo la forma normale abbiamo definito il comportamento del checker nella maniera esattamente opposta, ossia prima si estraggono informazioni dalla sorgente casuale e poi si effettuano le richieste all'oracolo; per ora, a scopo illustrativo, ipotizziamo tacitamente l'opposto.

è importante notare che la CNF così descritta ha clausole con al massimo q letterali e possiamo assumere che sia esattamente così, ossia ogni clausola abbia esattamente q letterali. Complessivamente, il comportamento del verificatore è esprimibile come la congiunzione di tutte le possibili φ_z^R:

$$\Phi_z = \bigwedge_{R \in \mathcal{R}} \varphi_z^R$$

Φ_z ha una sottoformula per ogni possibile stringa random, quindi ci sono $|\mathcal{R}|$ sottoformule ognuna con al più 2^q clausole, per un totale di $|\mathcal{R}|2^q = 2^{r(|z|)}2^q = 2^{r(|z|)+q} = 2^{O(\log(|z|)+q} = O(|z|)$ clausole (al massimo).

In secondo luogo, notiamo che se $z \in L$ il verificatore deve accettare con probabilità 1 per un qualche $\bar{w}$ che quindi rende vera Φ_z e di conseguenza anche ogni φ_z^R - il che significa che Φ_z è soddisfacibile. Al contrario, se $z \notin L$ per ogni possibile w si soddisfa al più meno della metà delle φ_z^R, ossia non è possibile che esista un w che soddisfi la metà o più delle possibili φ_z^R. Questo significa inoltre che delle $|\mathcal{R}|2^q$ clausole di cui Φ_z è costituita, nel caso $z \notin L$, ogni w rende vere al massimo un numero di clausole minore o uguale a

$$\frac{|\mathcal{R}|}{2}2^q + \frac{|\mathcal{R}|}{2}(2^q - 1)$$

Il seguente teorema conduce all'inapprossimabilità di MaxEkSat.

Teorema 4.22. *Esiste $\epsilon > 0$ tale che MaxSat non è $(1+\epsilon)-$approssimabile in tempo polinomiale a meno che* $\mathbf{P} = \mathbf{NP}$.

Dimostrazione. Sia $L \in \mathbf{NP}-\mathbf{completi}$. Per questo linguaggio esisterà una specifica funzione $r(|n|) \in O(\log(n))$ e uno specifico $q \in \mathbb{N}$ tale che $L \in \mathbf{PCP}[r, q]$. Definiamo

$$\bar{\epsilon} = \frac{1}{2^{q+1}}$$

e supponiamo che MaxSat sia $(1+\epsilon)-$approssimabile.

Per ogni input $z \in 2^*$ possiamo costruire Φ_z sulla quale potremo eseguire l'algoritmo $(1+\epsilon)$-approssimabile per MaxSat, il quale termina in tempo polinomiale. Se $z \in L$ sappiamo che Φ_z è soddisfacibile, cioè il vero valore calcolato risolvendo l'istanza di MaxSat è $t^*(\Phi_z) = |\mathcal{R}|2^q$, mentre se $z \notin L$ sappiamo che il valore restituito risolvendo l'istanza di MaxSat è $t^*(\Phi_z) = \frac{|\mathcal{R}|}{2}2^q + \frac{|\mathcal{R}|}{2}(2^q - 1) = 2^q|\mathcal{R}| - \frac{|\mathcal{R}|}{2}$.

Se $z \in L$ allora

$$t(\Phi_z) \geq \frac{t^*(\Phi_z)}{1+\bar{\epsilon}} = \frac{2^q|\mathcal{R}|}{1+\frac{1}{2^{q+1}}} = A$$

mentre se $z \notin L$

$$t(\Phi_z) \leq t^*(\Phi_z) \leq 2^q|\mathcal{R}| - \frac{|\mathcal{R}|}{2} = B$$

Calcoliamo $A - B$:

$$\begin{aligned}
A - B =& \frac{2^q|\mathcal{R}|}{1+\frac{1}{2^{q+1}}} - 2^q|\mathcal{R}| + \frac{|\mathcal{R}|}{2} \\
=& |\mathcal{R}| \frac{2^{q+1} - 2^{q+1}(1+\frac{1}{2^{q+1}}) + (1+\frac{1}{2^{q+1}})}{2(1+\frac{1}{2^{q+1}})} \\
=& |\mathcal{R}| \frac{2^{q+1} - 2^{q+1} - 1 + 1 + \frac{1}{2^{q+1}}}{2(1+\frac{1}{2^{q+1}})} \\
=& |\mathcal{R}| \frac{\frac{1}{2^{q+1}}}{2(1+\frac{1}{2^{q+1}})} \\
>& 0
\end{aligned}$$

che implica $A > B$. Quindi tutta la catena polinomiale di calcoli può essere utilizzata per *decidere* se z appartiene o meno a L:

$$z \in 2^* \stackrel{?}{=} L \in \textbf{NP} - \textbf{completi} \rightarrow \text{riduci } (L \rightsquigarrow \Phi_z) \rightarrow \text{risolvi MaxSat su } \Phi_z \rightarrow t(\Phi_z) \begin{cases} > A & z \in L \\ \leq B & z \notin L \end{cases}$$

Assurdo se $\mathbf{P} \neq \mathbf{NP}$. □

Teorema 4.23. *$MaxE3Sat$ non è $(\frac{8}{7} - \epsilon)-$approssimabile per un qualche $\epsilon > 0$.*

Dimostrazione. Omessa. □

Corollario 4.24. *L'algoritmo randomizzato per $MaxE3Sat$ è ottimo.*

4.5.2 Problema dell'insieme indipendente

IndependentSet

Input:	$G = (V, E)$
Output:	Sottoinsieme $X \subseteq V$
Problema:	Qual è l'insieme indipendente maggiore?
Ammissibili:	$X \subseteq V$ tale che X è un insieme indipendente, ossia tale che $\forall i, j \in X \ (i, j) \notin E$
Tipo:	Max
Costo:	$\lvert X \rvert$

Teorema 4.25. *Per ogni $\epsilon > 0$ IndependentSet non è $(2 - \epsilon)-$approssimabile in tempo polinomiale se $\mathbf{P} \neq \mathbf{NP}$.*

Dimostrazione. Sia $L \in \textbf{NP} - \textbf{completi}$; per Teorema 4.21 è anche $L \in \textbf{PCP}[r(n), q]$ con $r(n) \in O(\log(n))$ e $q \in \mathbb{N}$.

Per ogni $z \in 2^*$ si consideri l'insieme $\mathcal{R}_z$ sequenze di bit random di cardinalità $|\mathcal{R}_z| = 2^{r(|z|)}$ e, per ogni possibile $R \in \mathcal{R}$ tutte le Q_z^R possibili risposte dell'oracolo di cardinalità $|Q_z|^R = 2^q$; costruiamo quindi l'insieme $\mathcal{C}_z = \cup_{R \in \mathcal{R}_z} \{R\} \times Q_z^R$ dei possibili input al probabilistic checker, ognuno dei quali porta ad una risposta *sì* o *no*. Definiamo quindi $\mathcal{A}_z \subseteq \mathcal{C}_z$ l'insieme delle configurazioni accettanti nella forma

$$c = (R, \langle i_1^R : v_1, i_2^R : v_2, \cdots, i_q^R : v_q \rangle)$$

abbiamo inoltre che

$$|\mathcal{A}_z| \leq 2^{r(|z|)} 2^q = 2^{O(\log(|z|))} 2^q = O(|z|)$$

Costruiamo un grafo $G_z = (\mathcal{A}_z, E_{\mathcal{A}_z})$ sulle configurazioni accettanti e inseriamo un arco tra due configurazioni

$$(R, \langle i_1^R : v_1, i_2^R : v_2, \cdots, i_q^R : v_q \rangle) \rightarrow (R', \langle i_1^{R'} : v'_1, i_2^{R'} : v'_2, \cdots, i_q^{R'} : v'_q \rangle)$$

se e solo se le configurazioni sono *incompatibili*, ossia

$$R = R' \vee \exists k, k' : i_k^R = i_{k'}^{R'} \wedge v_k \neq v'_{k'}$$

definiamo questi lati come *lati di incompatibilità*.

Osservazione 4.26. *Se $z \in L$, G_z ha un insieme indipendente di cardinalità maggiore o uguale a $2^{r(|z|)}$.*

Dimostrazione. Siccome $z \in L$, deve esistere $\bar{w} \in 2^q$ tale che il verificatore accetta con probabilità 1 - questo significa che tutte le configurazioni ottenute al variare delle possibili stringhe random in cui la seconda parte è compatibile con $\bar{w}$ sono accettanti: la quantità di queste configurazioni accettanti è $2^{r(|z|)}$, ossia l'enumerazione di tutte le possibili stringhe random compatibili con $\bar{w}$. □

Osservazione 4.27. *Se $z \notin L$ ogni insieme indipendente di G_z ha cardinalità $\leq 2^{r(|z|)-1}$.*

Dimostrazione. Si immagini un insieme di configurazioni compatibili: nelle configurazioni, la parte delle query può contenere posizioni richieste più volte con risultati diversi, altrimenti non sarebbero compatibili (e di conseguenza l'insieme non sarebbe indipendente). Dato un insieme indipendente, si può costruire una stringa w che è compatibile con tutte le risposte ottenute, dove nelle posizioni di query che non appaiono nell'insieme indipendente si possono inserire valori arbitrari; tuttavia non ci possono essere contraddizioni. Qualunque w è adatta per far accettare z dal probabilistic checker; se ci fosse un insieme indipendente in G_z di cardinalità maggiore di $2^{r(|z|)-1}$ verrebbe contraddetto il Teorema 4.21. □

Dall'Osservazione 4.26 e dall'Osservazione 4.27 si arriva alla dimostrazione del Teorema 4.25: qualunque algoritmo che sia in grado di dare un'approssimazione migliore di 2 riesce a distinguere i due casi, sapendo quindi decidere se $z \in L$ o meno in tempo polinomiale. □

CAPITOLO 5

Strutture succinte

5.1 Abstract data types

Gli *abstract data type* sono tipi di dati descritti dal loro comportamento: un esempio è l'ADT `stack<T>`, il quale è dotato di alcune operazioni inerenti al tipo stesso, chiamate **primitive**:

```
bool   isEmpty()
T      top()
void   pop()
void   push(T)
```

Il comportamento cosa facciano i metodi si può descrivere in molti modi: si può utilizzare un metodo discorsivo, spiegando a parole, o utilizzare un metodo analitico:

$$\forall S, s.push(x).top() = x$$

(nonostante la notazione impropria dovuta alla signatura delle funzioni);

$$\forall S, s.isEmpty() \implies S.push(x).pop().isEmpty()$$

Una volta descritto un ADT è necessario implementarlo, ossia costruire effettivamente una struttura che implementa le primitive rispettandone la descrizione. Chiaramente, vi sono molte implementazioni diverse che soddisfano le richieste ma hanno *costi* diversi, sia in tempo che in spazio. Siamo interessati ad alcuni ADT e relative implementazioni che utilizzano poco tempo e spazio. Ogni ADT ha associato un concetto di *taglia*, che rappresenta genericamente la grandezza di un'istanza: nel caso dello stack, la taglia sarebbe il numero di elementi presenti sulla pila.

5.1.1 Teoria dell'informazione

Per poter caratterizzare le implementazioni degli ADT in base allo spazio che occupano è necessario introdurre alcuni concetti della teoria dell'informazione, i quali discendono dai Teoremi di Shannon, sommariamente riassumibili nel seguente teorema.

Teorema 5.1 (della codifica della sorgente). *Per codificare v valori servono in media* $\log_2(v)$ *bit.*

Per esempio, immaginiamo di dover codificare un'immagine 100×100 pixel in bianco e nero. Le immagini possibili sono 2^{10000}: per codificare queste immagini servono in media 10000 bit. In effetti, la rappresentazione banale che rappresenta ogni pixel, utilizza esattamente 10000 bit e non potrebbe usarne di meno! Usandone, per esempio, solo 9000, alcune immagini diverse avrebbero la stessa rappresentazione.

Questo teorema vale anche per rappresentazioni di dimensione variabile, ossia vale anche per codifiche: si supponga di avere un algoritmo in grado di comprimere tre immagini ognuna in 100 bit. La conseguenza di questo teorema è che ci saranno delle altre immagini che utlizzeranno più di 10000 bit, in modo che la media rimanga 10000.

In generale, dati v valori rappresentabili con $x_1, x_2, \cdots, x_v$ bit rispettivamente; il Teorema afferma che

$$\frac{\sum_i x_i}{v} \geq \log_2(v)$$

di così e che esiste un sistema di compressione che riesce a rappresentare in realtà il Teorema 5.1 dice di più, ossia che non si può fare meglio i v valori utilizzando un numero di bit medio tra $[\log_2(v), 1 + \log_2(v))$, assumendo che tutti i v valori siano equiprobabili[1].

Ci confronteremo spesso con questo **information-theoretical lower bound**: immaginiamo tutte le possibili istanze v_i di ADT di taglia i; per esempio, uno stack con valori in $\{0, 1, \cdots, 9\}$; lo stack di taglia 0 è lo stack vuoto, gli stack di taglia 1 sono le 10 istanze stack che contengono solo 1, solo 2, e così via, mentre gli stack di taglia 2 sono 10^2; in generale uno stack di taglia n ha 10^n valori. Il Teorema afferma che in media servono $\log_2 10^n = n \log_2(10) \approx 4n$ bit per rappresentare uno stack con valori in $\{0, 1, \cdots, 9\}$: sappiamo quindi che *nessuna implementazione* può utilizzare, in media, meno di $Z_n = n \log_2(10)$ bit, l'information-theoretical lower bound per rappresentare stack di taglia n con valori in $\{0, 1, \cdots, 9\}$.

Ipotizziamo di avere una struttura che utilizza in media $D_n \geq Z_n$ bit: esiste un tradeoff tra quanto *compatta* è la struttura e quanto tempo è necessario per eseguire le funzioni primitive. Esistono sistemi di compressione che ignorano completamente il problema: ad esempio, comprimendo un oggetto con l'algoritmo `tz2`, non si può utilizzare l'oggetto compresso come la sua rappresentazione non compressa per eseguire le primitive su di esso! Noi siamo interessati a strutture compresse - rappresentano i dati in maniera efficiente rispetto allo spazio - e con primitive efficienti tanto quanto un'implementazione non compressa.

Definiamo quindi delle classificazioni delle implementazioni in base al rapporto tra l'effettivo utilizzo di spazio (in media) e l'indice Z_n: un'implementazione dell'ADT è chiamata **implicita** se occupa un numero di bit $D_n = Z_n + O(1)$, **succinta** se occupa un numero di bit $D_n = Z_n + o(Z_n)$ e **compatta** se $D_n = O(Z_n)$; tutto questo sempre notando che le primitive devono essere efficienti tanto quanto quelle definite su strutture non compresse.

5.2 Strutture di rango e selezione

Questi ADT sono definiti da un vettore $\mathbb{b} \in 2^n$ con due primitive:

$$\mathbf{rank}_b : \mathbb{N} \to \mathbb{N}$$

$$\mathbf{select}_b : \mathbb{N} \to \mathbb{N}$$

tali che:

$$\forall p \leq n \ \ \mathbf{rank}_b(p) = |\{i | i < p \wedge b_i = 1\}|$$

$$\forall k \leq n \ \ \mathbf{select}_b(k) = \max\{p | \mathbf{rank}_b(p) \leq k\}$$

Quindi, per esempio, per un $\mathbf{b} = [0110101]$ si hanno due tabelle di rank e select come in Tabella 5.1.

Rank e select sono funzioni inverse in un senso molto stretto, ciò vale che

$$\forall k \ \ \mathbf{rank}_b(\mathbf{select}_b(k)) = k$$

[1] [Sha48] è il lavoro di C. Shannon che ha dato vita al campo della teoria dell'informazione; un approccio più moderno è [CT06].

p	$\mathbf{rank}_b(p)$
0	0
1	0
2	1
3	2
4	2
5	3
6	3
7	4

(a) $\mathbf{rank}_b(p)$

k	$\mathbf{select}_b(p)$
0	1
1	2
2	3
3	6
4	7
5	7
...	7

(b) $\mathbf{select}_b(p)$

Tabella 5.1: Tabelle per **b**.

mentre l'inverso, ossia applicare select a rank, si ottiene una proprietà diversa:

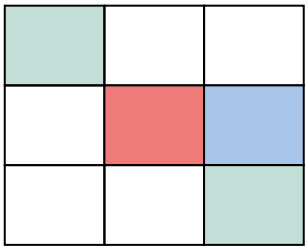
$$\forall p \ \ \mathbf{select}_b(\mathbf{rank}_b(p)) \geq p$$

proprio grazie a quest'ultima proprietà è possibile dedurre la struttura sottostante, nel senso che è possibile capire dove siano gli 0 e gli 1 in **b**.

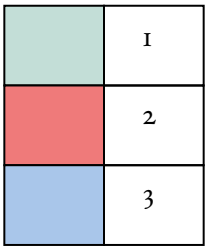
5.2.1 Struttura di Jacobson per il rango

Four-russians trick

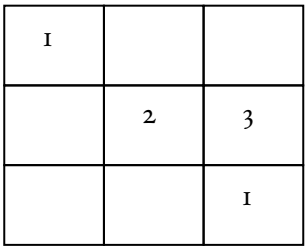
L'implementazione dell'ADT rank di Jacobson utilizza il "four-russians trick". Si immagini di voler rappresentare una matrice binaria: un modo per farlo potrebbe essere dividere la matrice in blocchi chiamati *piastrelle* ed enumerare le possibili piastrelle. Chiaramente, se nella matrice appare ogni possibile combinazione di piastrella, il guadagno del trucco sarà nullo. Se, invece, la matrice è molto ripetitiva, le piastrelle possibili da ricordare saranno poche e basterà utilizzare il numero associato alla piastrella per rappresentare l'intera matrice. Un esempio è in Fig. 5.1.



(a) Tabella iniziale. Ogni riquadro contiene 5 bit.



(b) Enumerazione di sottomatrici.



(c) Matrice risultante dopo l'applicazione del *four-russians trick*.

Figura 5.1: Trucco dei quattro russi.

Implementazione

Il vettore **b** di n bit viene quindi diviso in blocchi della stessa lunghezza, chiamati *superblocchi*, di lunghezza $\log_2(n)^2$. Ogni superblocco viene diviso a sua volta in blocchi più piccoli, di lunghezza $\frac{1}{2}\log_2(n)$, come rappresentato in Fig. 5.2. Per esempio, se $n = 256$, i superblocchi avranno lunghezza $\log_2(256)^2 = 8^2 = 64$ bit e saranno $256/64 = 4$, mentre i blocchi interni saranno $\frac{1}{2}\log_2(256) = \frac{1}{2}8 = 4$ bit e saranno $64/4 = 16$

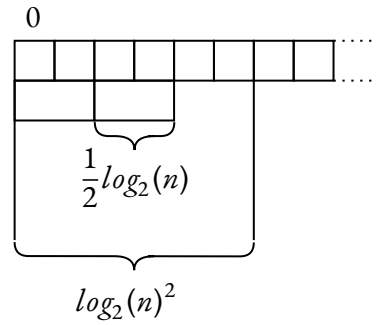


Figura 5.2: Divisione di **b** in superblocchi e blocchi.

per superblocco, $16 * 4 = 64$ in totale. In questo esempio, i possibili blocchi sono $2^4 = 16$ e, in generale, siccome i blocchi hanno lunghezza $\frac{1}{2}\log_2(n)$, sono

$$2^{\frac{1}{2}\log_2(n)} = (2^{\log_2(n)})^{\frac{1}{2}} = \sqrt{n}$$

Se volessimo memorizzare la funzione di rank per un singolo blocco, costruiremmo una tabella di $\frac{1}{2}\log_2(n)$ righe e per ognuna di queste bisognerebbe salvare il numero di 1 presenti nel blocco fino a quel punto, utilizzando per ogni rank uno spazio $\log_2(\frac{1}{2}\log_2(n))$. Interamente, quindi, la tabella di rank per un singolo blocco occupa spazio

$$\frac{1}{2}\log_2(n) \cdot \log_2(\frac{1}{2}\log_2(n)) \text{ bit}$$

e, volendo memorizzare la tabella per ogni tipo di blocco, si consuma uno spazio

$$2^{\frac{1}{2}\log_2(n)} \cdot \frac{1}{2}\log_2(n) \cdot \log_2(\frac{1}{2}\log_2(n)) = \sqrt{n} \cdot \frac{1}{2}\log_2(n) \cdot \log_2(\frac{1}{2}\log_2(n)) \leq \sqrt{n}\frac{1}{2}\log_2(n) \cdot \log_2(\log_2(n)) = o(n) \text{ bit}$$

che significa che si può definire una tabella che enumera i tipi di blocco e per ognuno di essi, come mostrato nella Tabella 5.2.

0000	p	0	1	2	3	4
	rank(p)	0	0	0	0	0
0001	p	0	1	2	3	4
	rank(p)	0	0	0	0	1
...						
1111	p	0	1	2	3	4
	rank(p)	0	1	2	3	4

Tabella 5.2: Rank per ogni possibile blocco di lunghezza 4.

Tutta questa struttura, benché sembra molto grande, è in realtà memorizzabile in $o(n)$, ossia in una quantità di spazio che cresce meno rapidamente rispetto a n. La prima idea è memorizzare queste strutture di rank per i blocchi. Dopo di che, per ogni superblocco si memorizzano gli 1 prima del superblocco, ossia per ogni superblocco i si definisce

$$S_i = \textbf{rank}(i[0])$$

come rappresentato nella Fig. 5.3. Per ogni blocco l afferente al superblocco i si definisce

$$B_l = \textbf{rank}(l[0]) - S_i$$

come rappresentato nella Fig. 5.4.

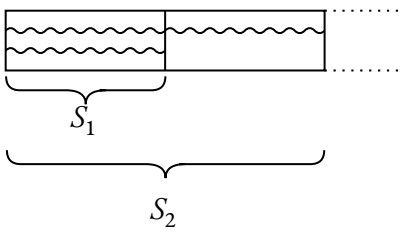


Figura 5.3: S_2 è il numero di 1 presenti in **b** 'sotto' la traccia più lunga, mentre S_1 è il numero di 1 sotto quella più corta. Va notato che $S_0 = 0$ benché non sia mostrato nella figura.

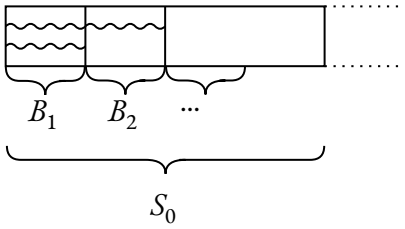


Figura 5.4: B_2 è il numero di 1 presenti in **b** 'sotto' la traccia più lunga, mentre B_1 è il numero di 1 sotto quella più corta. Va notato che B_0, in questo frangente, è 0, poiché S_0 è il primo superblocco di **b**.

Quindi, gli S_i sono tanti quanti sono i superblocchi, ossia $\frac{n}{log_2(n)^2}$ e occupano spazio

$$\frac{n}{\log_2(n)^2} \underbrace{\log_2(n)}_{\text{spazio di un } S_I} = \frac{n}{\log_2(n)} = o(n) \text{ bit}$$

mentre i B_l sono tanti quanti sono i blocchi, ossia $\frac{n}{\frac{1}{2}\log_2(n)}$ e occupano spazio

$$\frac{n}{\frac{1}{2}\log_2(n)} \underbrace{\log_2(\log_2(n)^2)}_{\text{al massimo}} = \frac{2n}{\log_2(n)} 2\log_2(\log_2(n)) = o(n) \text{ bit}$$

Se si vuole conoscere il rango di uno specifico bit in un blocco bisogna recuperare S_i e B_i e calcolare il quale sia effettivamente il rango, che è stato memorizzato in una tabella Tab utilizzando il four-russians trick enumerando i possibili $\sqrt{(n)}$ blocchi e memorizzando il rango di ogni bit del blocco. Complessivamente, quindi, tutte le tabelle necessarie occupano $D_n = o(n)$ bit.

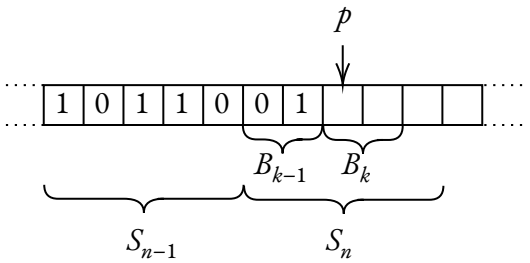


Figura 5.5: Calcolo di $\mathbf{rank_b}(p)$.

Quanto tempo impiega un'implementazione basata su queste strutture per calcolare il rango di una posizione p? Per sapere quanti 1 ci sono prima della posizione p, bisogna innanzitutto calcolare il superblocco di appartenenza di p, che è il superblocco numero $\frac{p}{\log_2(n)^2}$. Vogliamo sapere quanti 1 ci sono prima di quel superblocco - valore che abbiamo memorizzato nei S_i. A questo punto, è necessario calcolare il numero di 1 dall'inizio del superblocco all'inizio del blocco al quale appartiene p, valore salvato in B_i. Rimane, quindi,

da capire quanti 1 ci sono dall'inizio del blocco al quale appartiene p fino alla posizione p stessa. Per questo si può utilizzare la tabella Tab del four-russians trick:

$$\mathbf{rank_b}(p) = S_{\frac{p}{\log_2(n)^2}} + B_{\frac{p}{\frac{1}{2}\log_2(n)}} + Tab[t, p \mod \frac{1}{2}\log_2(n)]$$

dove t è il tipo blocco a cui p appartiene secondo l'enumerazione dei blocchi di Tab, calcolabile data la posizione d'inizio del blocco di p

$$x = \lfloor \frac{p}{\frac{1}{2}\log_2(n)} \rfloor (\frac{1}{2}\log_2(n))$$

accedendo il vettore $\mathbf{b}$ e leggendo i valori del blocco[2]:

$$t = \mathbf{b}[x, x+1, \cdots, x + \frac{1}{2}\log_2(n) - 1]$$

Il calcolo del rank avviene quindi in tempo lineare, poiché si tratta di accedere a 3 tabelle e la struttura quindi occupa lo spazio di $n + o(n)$ bit, poiché è necessario mantenere b e risponde alle query in tempo $O(1)$. Rispetto alla nostra classificazione, questa struttura è **succinta** e ha la stessa efficienza della struttura naïve.

5.2.2 Struttura di Clarke per la selezione

La struttura di Clarke per la selezione è succinta teoricamente, ma in pratica è raramente utilizzata perché la sua implementazione è molto complessa. L'obiettivo è, dato un vettore $\mathbf{b}$ di valori binari fissato, calcolare la funzione di selezione

$$\mathbf{select_b}(k) = \text{ posizione del } k-\text{esimo } 1$$

La struttura di Clarke utilizza dei *livelli*, simili a quelli utilizzati dalla struttura di Jacobson.

Primo livello

Il primo livello della struttura di Clarke si è un insieme di valori che rappresentano le posizioni degli 1 di ordinalità multipla di $\log_2(n) \cdot \log_2(\log_2(n))$, ossia

$$P_i = \mathbf{select_b}(i \cdot \log_2(n) \cdot \log_2(\log_2(n)))$$

ossia la posizione del $(i \cdot \log_2(n) \cdot \log_2(\log_2(n)))$-esimo 1.

Memoria La grandezza di questa famiglia, ossia il numero di P_i, dipende dal numero di i che ci sono in $\mathbf{b}$, ma nel caso peggiore, il vettore è composto unicamente da 1 e vi saranno $\frac{n}{\log_2(n)\cdot\log_2(\log_2(n))}$ membri. Ad ognuno di essi va associato un elemento, il quale richiede $\log_2(n)$ bit; in totale, questo livello occupa

$$\frac{n}{\log_2(n) \cdot \log_2(\log_2(n))} \cdot \log_2(n) = o(n) \text{ bit}$$

[2] In termini pratici, come avviene questo accesso? Se avviene leggendo, uno per uno, tutti i valori del blocco al quale appartiene p per poi accedere alla tabella Tab, non ha più senso leggere i valori da $\mathbf{b}[x]$ fino a $\mathbf{b}[p-1]$ contando gli 1 visti?

Secondo livello

Per le posizioni che non sono multiple di $\log_2(n) \cdot \log_2(\log_2(n))$ si utilizza un secondo livello, che è costruito differentemente in base ad un indice calcolato per ogni P_i:

$$\forall i \; r_i = P_{i+1} - P_i$$

che rappresenta la distanza fra l'$(i \cdot \log_2(n) \cdot \log_2(\log_2(n)))$-esimo 1 e il $((i+1) \cdot \log_2(n) \cdot \log_2(\log_2(n)))$-esimo. Questo valore, r_i, sarà esattamente uguale a $\log_2(n) \cdot \log_2(\log_2(n))$ se e solo se tra P_i e P_{i+1} ci sono unicamente 1, e sarà maggiore se invece le due posizioni sono più lontane. I due casi che si considerano dipendono dal valore di r_i.

Caso sparso Se $r_i \geq (\log_2(n) \cdot \log_2(\log_2(n)))^2$, significa che in **b** ci sono pochi 1 tra le posizioni P_i e P_{i+1}; in questo caso definiamo S_i come la lista esplicita delle posizioni di tutti gli 1 in **b** tra le due posizioni rappresentate come differenza tra P_i.

In questo caso, gli S_i costruiti sono esattamente $\log_2(n) \cdot \log_2(\log_2(n))$, in quanto stiamo contando gli 1 in **b** tra il $(i \cdot \log_2(n) \cdot \log_2(\log_2(n)))$-esimo 1 e il $((i+1) \cdot \log_2(n) \cdot \log_2(\log_2(n)))$-esimo 1, e ad ognuno di essi si associa un numero che in grandezza è minore o uguale a $\log_2(r_i)$, concludendo che per memorizzare un S_i sono necessari

$$\begin{aligned}
\log_2(n) \cdot \log_2(\log_2(n)) \cdot \log_2(r_i) &= \frac{(\log_2(n) \cdot \log_2(\log_2(n)))^2}{\log_2(n) \cdot \log_2(\log_2(n))} \cdot \log_2(r_i) \\
&\leq \frac{r_i}{\log_2(n) \cdot \log_2(\log_2(n))} \cdot \log_2(r_i) \\
&\leq \frac{r_i}{\log_2(n) \cdot \log_2(\log_2(n))} \cdot \log_2(n) \\
&\leq \frac{r_i}{\log_2(\log_2(n))} \\
&\leq \frac{n}{\log_2(\log_2(n))} = o(n) \text{ bit}
\end{aligned}$$

Caso denso Se $r_i \leq (\log_2(n) \cdot \log_2(\log_2(n)))^2$, si memorizzano gli 1 multipli di $\log_2(r_i)\log_2(\log_2(n))$, ossia partendo dalla posizione P_i si salvano le posizioni degli $j \cdot \log_2(r_i)\log_2(\log_2(n))$-esimi 1 come differenze da P_i, ossia

$$S_j^i = \mathbf{select}_{\mathbf{b}}(j \cdot \log_2(r_i)\log_2(\log_2(n))) - P_i$$

In questo caso, gli 1 memorizzati sono $\frac{\log_2(n) \cdot \log_2(\log_2(n))}{\log_2(r_i) \cdot \log_2(\log_2(n))}$. Ad ognuno di questi si associa un valore in grandezza $\log_2(r_i)$, quindi lo spazio utilizzato per memorizzare tutti i valori S_j^i sono

$$\frac{\log_2(n) \cdot \log_2(\log_2(n))}{\log_2(r_i) \cdot \log_2(\log_2(n))} \log_2(r_i) = \frac{\log_2(n) \cdot \log_2(\log_2(n))}{\log_2(\log_2(n))} \leq \frac{r_i}{\log_2(\log_2(n))} \leq \frac{n}{\log_2(\log_2(n))} = o(n) \text{ bit}$$

Terzo livello

Se nel secondo livello ci si trova nel caso denso, si utilizza un terzo livello. Questo livello viene utilizzato esclusivamente per gli 1 le cui posizioni non sono state salvate nel secondo livello nel caso denso; pertanto, l'assunto è che

$$r_i < (\log_2(n)\log_2(\log_2(n)))^2$$

Per ognuna di queste posizioni calcoliamo la differenza tra S_j^i:

$$\forall j \bar{r}_j^i = S_{j+1}^i - S_j^i$$

Come nel caso precedente, abbiamo due possibilità in base al valore di $\bar{r}_j^i$, il quale gode comunque della proprietà

$$\forall i, j \ \bar{r}_j^i \geq \log_2(r_i) \log_2(\log_2(n))$$

Caso sparso Nel caso in cui $\bar{r}_j^i \geq \log_2(\bar{r}_j^i) \log_2(r_i) \log_2(\log_2(n))^2$, si memorizzano esplicitamente tutte le posizioni degli 1 tra S_j^i e S_{j+1}^i con dei valori $T_{j,k}^i$ come differenze tra S_j^i. In questo caso, il consumo di memoria è

$$(\log_2(r_i) \cdot \log_2(\log_2(n)) \cdot \log_2(\bar{r}_j^i) \leq \frac{\log_2(r_i) \cdot \log_2(\log_2(n))^2 \log_2(\bar{r}_j^i)}{\log_2(\log_2(n))} \leq \frac{\bar{r}_j^i}{\log_2(\log_2(n))} = o(n) \text{ bit}$$

Caso denso Nel caso in cui $\bar{r}_j^i \leq \log_2(\bar{r}_j^i) \log_2(r_i) \log_2(\log_2(n))^2$, significa che ci sono pochi 0 tra S_j^i e S_{j+1}^i e si utilizza il four-russians trick. Inizialmente osserviamo quanto segue.

Osservazione 5.2.

$$\begin{aligned}\log_2(\bar{r}_j^i) \leq \log_2(r_i) &\leq \log_2(\log_2(n) \cdot \log_2(\log_2(n)))^2 \\ &= 2\log_2(\log_2(n)) + 2\log_2(\log_2(\log_2(n))) \\ &\leq 4\log_2(\log_2(n))\end{aligned}$$

Osservazione 5.3.

$$\bar{r}_j^i < \log_2(\bar{r}_j^i) \cdot \log_2(r_i) \cdot (\log_2(\log_2(n))^2 \leq 16(\log_2\log_2(n))^4$$

Lo spazio necessario per utilizzare il four-russians trick è quanto segue. Servono $2^{\bar{r}_j^i}$ enumerazioni di 'sottovettori', ossia la dimensione tra S_j^i e S_{j+1}^i, che è la parte che va memorizzata esplicitamente; per ognuna di queste enumerazioni è necessario salvare la posizione di $\bar{r}_j^i$ 1 utilizzando memoria al massimo $\log_2(\bar{r}_j^i)$:

$$\begin{aligned}2^{\bar{r}_j^i} \cdot \bar{r}_j^i \cdot \log_2(\bar{r}_j^i) &\leq 2^{16(\log_2\log_2(n))^4} \cdot 16(\log_2\log_2(n))^4 \cdot \log_2(16(\log_2\log_2(n))^4) \\ &= 16(\log_2\log_2(n))^8 \log_2(16(\log_2\log_2(n))^4) = o(n)\end{aligned}$$

Complessit□ totale in spazio

In totale, per memorizzare **b** e i possibili tre livelli di struttura, sono necessari $o(n)$ bit per il primo livello e, in base a come è fatto il secondo livello

$$\sum^{\frac{n}{\log_2(n)\cdot\log_2(\log_2(n))}} \frac{P_{i+1} - P_i}{\log_2(\log_2(n)} = \frac{P_n - P_0}{\log_2(\log_2(n))} \leq \frac{n}{\log_2(\log_2(n))} = o(n) \text{ bit}$$

più $o(n)$ bit per il terzo livello. Quindi, la struttura di Clarke occupa spazio $n + o(n)$ e ha accesso tempo di accesso costante, pertanto è una struttura **succinta**.

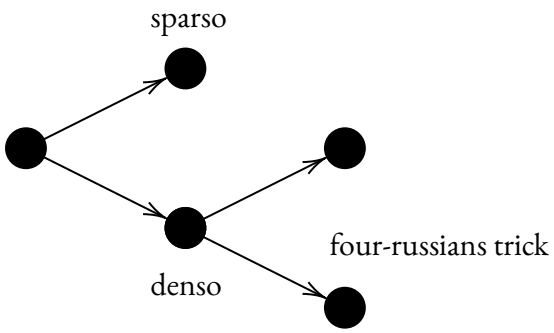
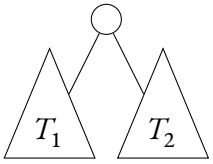


Figura 5.6: Struttura di Clarke per la selezione.

5.3 Struttura per alberi binari

Gli *alberi* possono essere visti in due modi: per i matematici, essi sono dei grafi connessi aciclici mentre per gli informatici sono una *struttura dati* radicata che ha una *radice*. In particolare, un albero *binario* è un albero tale per cui ogni nodo ha al più due figli; formalmente, si definiscono induttivamente: l'albero con un solo nodo (che è anche la radice), denotato $\emptyset$, è un albero binario; se T_1 e T_2 sono due alberi binari, allora anche l'albero radicato in un nuovo nodo che ha come figli T_1 e T_2, denotato (T_1, T_2) è un albero binario. In questa definizione, ogni nodo ha o 0 o 2 figli. I nodi che non hanno figli sono chiamati **nodi esterni** o **foglie**, mentre gli altri sono chiamati **nodi interni**.

(a) L'albero binario $\emptyset$.



(b) L'albero binario (T_1, T_2).

Figura 5.7: Definizione induttiva di albero binario.

Ora stiamo descrivendo alberi *vuoti*, ossia che non contengono dati: alternative sono gli alberi **ancillari**, i quali contengono dati solo nei nodi interni o solo nelle foglie. Al netto di queste distinzioni, definiamo E l'insieme dei nodi esterni, I l'insieme dei nodi interni di un albero binario e n il numero di nodi interni, ossia $n = |I|$. Inoltre, 'equipaggiamo' gli alberi binari di due funzioni *ext* e *int*, che denotano l'insieme delle foglie interne e l'insieme delle foglie interne di un albero binario.

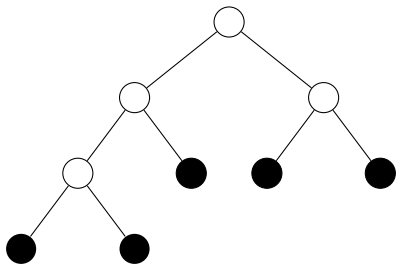


Figura 5.8: L'albero binario $(((\emptyset, \emptyset), \emptyset), (\emptyset, \emptyset))$.

Teorema 5.4. *In ogni albero binario, il numero di foglie in un albero binario è uguale al numero di nodi esterni più uno:*

$$|E| = |I| + 1$$

Dimostrazione. Per induzione.

- **base**: $ext(\emptyset) = 1$ e $int(\emptyset) = 0$.
- **induzione**: siano L, R due alberi binari. Allora

$$ext((L,R)) = ext(L) + ext(R) = int(L) + 1 + int(R) + 1 = int((L,R)) + 1$$

□

Corollario 5.5. *Ogni albero con n nodi interni ha in totale $2n + 1$ nodi.*

Teorema 5.6 (di Catalan)**.** *Il numero di alberi binari con n nodi interni è*

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

Dimostrazione. Omessa. □

Corollario 5.7. $\forall n \ \log_2(C_n) = 2n + O(\log_2(n))$

Dimostrazione. Utilizzando l'approssimazione di Stirling

$$x! \approx \sqrt{2\pi x}(\frac{x}{e})^x$$

si ha che

$$C_n = \frac{1}{n+1}\frac{(2n)!}{n!(2n-n)!} = \frac{1}{n+1}\frac{(2n)!}{(n!)^2} \approx \frac{1}{n+1}\frac{\sqrt{4\pi n}(\frac{2n}{e})^{2n}}{2\pi n(\frac{n}{e})^{2n}} = \frac{1}{n+1}\frac{1}{\sqrt{\pi n}}2^{2n} \approx \frac{4^n}{\sqrt{\pi n^3}}$$

(che può essere dimostrato asintoticamente corretto). Questo significa che

$$\log_2(C_n) = n\log_2(4) - \frac{1}{2}\log_2(\pi n^3) = 2n - \frac{3}{2}\log_2(n) - \frac{1}{2}\log_2(\pi) = 2n + O(\log_2(n))$$

□

Corollario 5.8. *Per memorizzare alberi binari con n nodi interni sono necessari*

$$Z_n = 2n + O(\log_2(n)) \text{ bit}$$

5.3.1 L'ADT albero binario

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

$$\forall n \in (I \cup E) \ \textbf{is\_leaf}(n) = 1 \iff n \in E$$

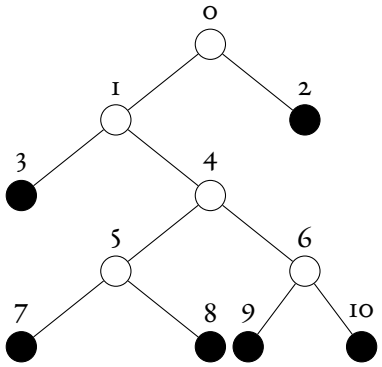
$$\forall n \in (I \cup E) \ \textbf{left\_child}(n) = \{n_l | n_l \text{ è il figlio sinistro di } n\}$$

$$\forall n \in (I \cup E) \ \textbf{right\_child}(n) = \{n_l | n_l \text{ è il figlio destro di } n\}$$

$$\forall n \in (I \cup E) \ \textbf{parent}(n) = \{p | n \text{ è il figlio di } p\}$$

Rappresentazione

Per rappresentare un albero binario con n nodi interni, numeriamo seguendo una visita in ampiezza i nodi dell'albero, come rappresentato in Fig. 5.9a. I numeri assegnati andranno da 0 a $2n$ e introducono una biiezione tra l'insieme $\{0, \cdots, 2n\}$ e i nodi, realizzato come una funzione $node : \{0, \cdots, 2n\} \to (E \cup I)$.



(a) L'albero binario $((\emptyset, ((\emptyset, \emptyset), (\emptyset, \emptyset))), \emptyset)$ numerato.

$\mathbf{b} =$

0	1	2	3	4	5	6	7	8	9	10
1	1	0	0	1	1	1	0	0	0	0

(b) Vettore **v** associato.

Figura 5.9: Un albero binario e il suo vettore associato.

La concreta rappresentazione dell'albero si realizza con un vettore di bit di lunghezza $2n + 1$ **v** tale che

$$\forall i \in \{0, \cdots, 2n\} \ \mathbf{v}[i] = \begin{cases} 1 & node(i) \in I \\ 0 & node(i) \in E \end{cases}$$

ottenendo quindi esattamente n '1' nel vettore, come mostrato in Fig. 5.9b.

Implementazione

Per capire come implementare l'ADT utilizzando il vettore **v**, ci poniamo nella situazione di dover trovare, dato un nodo numerato p in un albero binario T, i suoi due figli, i quali sono numerati q e $q + 1$. Sia quindi T' un sottoalbero di T, radicato nella stessa radice di T, che contiene tutti i nodi di T numerati fino al nodo precedente al nodo q, come rappresentato in Fig. 5.10.

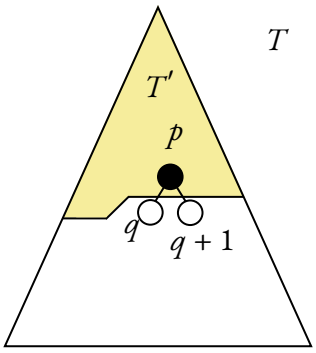


Figura 5.10: Sottoalbero T' di T utilizzato per calcolare q e $q + 1$.

Definiamo l'insieme dei nodi interni di T con numerazione strettamente minore di p

$$E_{T,p} = \{n \in int(T) | n < p\}$$

e, intuitivamente, deve essere

$$q = |int(T')| + |ext(T')| = 2|int(T')| + 1 = 2|E_{T,p}| + 1) = 2\mathbf{rank_v}(p) + 1.$$

poiché $|E_{T,p}|$ è uguale al numero di '1' presenti in $\mathbf{v}$ con indice strettamente minore a p, concludendo, chiaramente se e solo se $\mathbf{is\_leaf}(p) = \mathbf{v}[p] = 1$, che[3]

$$\forall p \in \{0, \cdots, 2n\} \ \mathbf{left\_child}(p) = 2\mathbf{rank_v}(p) + 1$$

$$\forall p \in \{0, \cdots, 2n\} \ \mathbf{right\_child}(p) = 2\mathbf{rank_v}(p) + 2$$

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

$$\begin{cases} 2\mathbf{rank_v}(p) + 1 = q & \text{se } q \text{ è il figlio di sinistra} \\ 2\mathbf{rank_v}(p) + 2 = q & \text{se } q \text{ è il figlio di destra} \end{cases} \implies \begin{cases} \mathbf{rank_v}(p) + \frac{1}{2} = \frac{q}{2} \\ \mathbf{rank_v}(p) + 1 = \frac{q}{2} \end{cases} \implies \begin{cases} \mathbf{rank_v}(p) = \frac{q}{2} - \frac{1}{2} \\ \mathbf{rank_v}(p) = \frac{q}{2} - 1 \end{cases}$$

ma possiamo concludere che

$$\mathbf{rank_v}(p) = \lfloor \frac{q}{2} - \frac{1}{2} \rfloor$$

siccome a seconda che q sia pari o dispari l'equazione sarà sempre verificata, ossia questa uguaglianza è vera se e solo se è almeno una delle due precedenti; se $\frac{q}{2} - \frac{1}{2}$ è intero, allora è vera la prima equazione, mentre se non è intero si ottiene $\frac{q}{2} - 1$, verificando la seconda. Applichiamo ad entrami i membri l'operazione di **select**:

$$\mathbf{select_v}(\mathbf{rank_v}(p)) = \mathbf{select_v}(\lfloor \frac{q}{2} - \frac{1}{2} \rfloor) \implies p = \mathbf{select_v}(\lfloor \frac{q}{2} - \frac{1}{2} \rfloor)$$

Complessit□ in spazio

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

$$D_n = 2n + 1 + o(2n + 1) = 2n + 1 + o(n)$$

con un risultato per Z_n pari a $2n + O(\log_1(n))$, la differenza è

$$D_n - Z_n = o(n)$$

pertanto la struttura è succinta con accesso in tempo costante.

Alberi binari con dati

Se i dati si trovano su ogni tipo di nodo dell'albero (sia interni che interni) i dati ancillari si possono mantenere in un ulteriore vettore della stessa lunghezza. Se, invece, i dati ancillari si trovano esclusivamente sui nodi interni, la situazione è leggermente più complicata: il vettore dei dati avrà lunghezza pare ai al numero di nodi interni e sarà necessario ottenere il numero ordinale del nodo interno, utilizzando quindi la funzione rank. Alternativamente, se si vogliono salvare dati sulle foglie, si dovrà utilizzare una tecnica in grado di contare il `rank` per gli 0.

[3] In quanto segue si omette la biiezione *node*, dando per implicito il rapporto tra enumerazione e nodi.

5.4 Struttura di Elias-Fano per sequenze monotone

La struttura (o codifica, rappresentazione) di Elias-Fano che andiamo a presentare occupa sulle sequenze monotone di interi, ossia una sequenza di n numeri interni

$$0 \leq x_0, x_1, \cdots, x_{n+1}$$

ordinati, ossia

$$\forall i, j \; i < j \implies x_i < x_j$$

e tali per cui

$$\exists u \; \forall i \; x_i < u$$

dove u è chiamato *dimensione dell'universo*. L'ADT che vogliamo rappresentare richiede una primitiva che, presentato un indice i, restituisce l'i-esimo elemento della sequenza, ossia

$$\forall i \; \textbf{project}(i) = x_i$$

La rappresentazione di seqenze monotone di interi è molto utile. Per esempio, per rappresentare un grafo, si può attribuire ad ogni nodo un numero: l'idea classica è quella di utilizzare poi, per esempio, liste di adiacenza per elencare i nodi raggiungibili da un certo noto considerato. I nodi adiacenti possono essere ordinati in ordine crescente, ottenendo una sequenza esattamente monotona. Un altro esempio di utilizzo di sequenze monotone è negli indici inversi.

5.4.1 Rappresentazione

La rappresentazione banale sarebbe utilizare un vettore di n numeri utilizzando per ognuno $\log_2(u)$ bit; la rappresentazione di Elias-Fano utilizza meno spazio basandosi sulla seguente idea: si definisce

$$l = \max\{0, \lfloor \log_2(\frac{u}{n}) \rfloor\}$$

e gli l bit meno significativi di ogni x_i vengono salvati esplicitamente, mentre i rimanenti bit vengono rappresentati in un altro modo. Consideremo sempre il caso in cui $l \neq 0$, in quanto $l = 0 \iff u < n$, facendo quindi una moderata assunzione di sparsità.

```
0 0 0 1  parte superiore
. . . .
1 1 1 1
-------  limite l
0 1 1 1
. . . .
1 0 0 1  parte inferiore
```

Tabella 5.3: Divisione di ogni x_i.

Implementazione e complessit□ in spazio

Le parti inferiori vengono 'estratte' definendo

$$\forall i \in \{0, \cdots, n\} \; l_i = x_i \mod 2^l$$

utilizzando $n \cdot l$ bit in totale. Ciò che rimane è la parte superiore, ossia $s_i = \lfloor \frac{x_i}{2^l} \rfloor$; definiamo una sequenza di differenze

$$d_i = \lfloor \frac{x_i}{2^l} \rfloor - \lfloor \frac{x_{i-1}}{2^l} \rfloor$$

assumendo $x_{-1} = 0$. Stiamo sfruttando il fatto che la sequenza sia non decrescente, quindi $\forall i \; d_i \geq 0$.

Ogni differenza della sequenza viene rappresentata in unario, utilizzando d_i bit '0' seguiti dal bit '1' per rappresentare il valore d_i; la sequenza così rappresentata viene salvata in un vettore **d** equipaggiato con funzioni di rango e selezione. Questa struttura occupa in spazio una quantità al più

$$\sum_{i=0}^{n-1} (d_i + 1) = n + \sum_{i=0}^{n-1} d_i = n + \sum_{i=0}^{n-1} (\lfloor \frac{x_i}{2^l} \rfloor - \lfloor \frac{x_{i-1}}{2^l} \rfloor) \text{ bit}$$

che identifica una serie telescopica; pertanto, il consumo in spazio è

$$n - (\lfloor \frac{x_{n-1}}{2^l} \rfloor - \lfloor \frac{x_{-1}}{2^l} \rfloor) \leq n + \frac{u}{2^l} = n + \frac{u}{2^{\lfloor \log_2(\frac{u}{n}) \rfloor}} \text{ bit}$$

Se u/n è una potenza di 2, allora il consumo è

$$n + \frac{u}{(u/n)} = 2n \text{ bit}$$

altrimenti è al più

$$n + \frac{u}{2^{\log_2(\frac{u}{n})-1}} = n + \frac{u}{2^{\log_2(u/n)} 1/2} = n + \frac{2n}{u/n} = 3n \text{ bit}$$

Considerando entrambe le parti, questa struttura occupa in spazio $l \cdot n$ bit per la parte inferiore e $2n$ o $3n$ bit per la parte superiore; in totale, quindi, si consumano $(l + 2)n$ o $(l + 3)n$ bit. Ricordando che stiamo considerando sempre $l = \lfloor \log_2(u/n) \rfloor$, consideriamo

$$\lceil \log_2(u/n) \rceil = \begin{cases} l & u/n \text{ è una potenza di 2} \\ l + 1 & \end{cases}$$

e concludiamo

$$D_n = (2 + \lceil \log_2(u/n) \rceil)n \text{ bit}$$

che però non tiene conto dello spazio occupato dalla struttura di rango e selezione, che infatti sono necessarie: supponiamo di volere la posizione dell'i-esimo 1 in **d**:

$$\textbf{select}_{\mathbf{d}}(i) = d_0 + d_1 + \cdots + d_{i-1} + i$$

quindi

$$\textbf{select}_{\mathbf{d}}(i) - i = \sum_{j=0}^{i} \lfloor \frac{x_j}{2^l} \rfloor - \lfloor \frac{x_{j-1}}{2^l} \rfloor = \lfloor \frac{x_i}{2^l} \rfloor$$

di conseguenza

$$x_i = \lfloor \frac{x_i}{2^l} \rfloor 2^l + (x_i \bmod 2^l) = (\textbf{select}_{\mathbf{d}}(i) - i)2^l + l_i$$

Contando anche le strutture di rank e select, che quindi sono necessarie, si occupa uno spazio

$$D_n = (2 + \lceil \log_2(u/n) \rceil)n + o(n) \text{ bit}$$

5.4.2 Lower bound per le strutture di Elias-Fano

La parte difficile dello studio di questa struttura è il calcolo del lower bound: dobbiamo considerare tutte le sequenze monotone

$$0 \leq x_0 \leq \cdots \leq x_{n-1} \leq u$$

Quante sono, una volta fissati n e u? Esse sono in biiezione con i multinsiemi di cardinalità n sottoinsiemi di $\{0, 1, \cdots, u-1\}$. Uno di questi multinsiemi si può vedere come

$$c_0, c_1, \cdots, c_{u-1}$$

dove ogni c_i è il numero di occorrenze del valore i nel multinsieme, ossia il numero di soluzioni intere non negative dell'equazione

$$c_0 + c_1, \cdots, c_{u-1} = n$$

e possiamo calcolare il numero di sequenze monotone calcolando il numero di possibili soluzioni per questa equazione fissati n e u.

Per esempio, si immagini di avere un universo di dimensione $u = 7$ e $n = 5$ numeri estratti da $\{0, 1, \cdots, 7\}$: la sequenza

$$0 \leq 1 \leq 3 \leq 3 \leq 5 \leq 6 \leq 7$$

la cui equazione associata è

$$c_0 + c_1 + c_2 + \cdots + c_6 = 5$$

è rappresentabile, in termini di c_i, come

$$c_0 = 0, c_1 = 1, c_2 = 0, c_3 = 2, c_4 = 0, c_5 = 1, c_6 = 1$$

che è una soluzione dell'equazione precedente.

Metodo stars and bars

Per contare le possibili soluzioni si può utilizzare la tecnica *stars and bars*, in cui si utilizzano delle stringhe costruite utilizzando n stelline e $u-1$ barrette, piazzate in qualsiasi posizione:

$$|\star||\star\star||\star|\star$$

Il numero di soluzioni è uguale al numero di stringhe costruite in tale modo: i valori nella sequenza sono n e vanno 'distribuiti' in u spazi, delineati dalle $u-1$ barrette. L'interpretazione della stringa appena mostrata è $c_0 = 0, c_1 = 1, c_2 = 0, c_3 = 2, c_4 = 0, c_5 = 1, c_6 = 1$, come nell'esempio precedente. I simboli utilizzabili sono $(u-1)+n$, pertanto le possibili stringhe costruibili con questi simboli sono $\binom{u+n-1}{u-1}$. Questo fornisce il lower bound desiderato:

$$Z_n = \log_2 \binom{u+n-1}{u-1} = \log_2 \binom{u+n-1}{n} \text{ bit}$$

e introduciamo l'approssimazione

$$\binom{a}{b} \approx b \log_2(\frac{a}{b}) + (a-b) \log_2(\frac{a}{a-b})$$

che ci permette di scrivere

$$Z_n = \log_2 \binom{u+n-1}{u-1} \approx n \log_2(\frac{u+n-1}{n}) + (u-1)\log_2(\frac{u+n-1}{u-1}) =$$
$$= n \log_2(\frac{u+n-1}{n}) = n \log_2(\frac{u}{n}(1 + \frac{n}{u} - \frac{1}{u}))$$
$$= n \log_2(\frac{u}{n}) + n \log_2(1 + \frac{n}{u} - \frac{1}{u}) \text{ bit}$$

Un'altra proprietà che utilizziamo è

$$x \approx \ln(1+x)$$

quindi

$$Z_n = n \log_2(\frac{u}{n}) + n \ln(1 + \frac{n}{u} - \frac{1}{u})\frac{1}{\ln(2)} \approx n \log_2(\frac{u}{n}) + n(\frac{n}{u} - \frac{1}{u})\frac{1}{\ln(2)}$$
$$= n \log_2(\frac{u}{n}) + \frac{1}{u}\frac{1}{\ln(2)} - \frac{n}{u}\frac{1}{\ln(2)} \approx n \log_2(\frac{u}{n}) \text{ bit}$$

L'utilizzo in spazio calcolato precedentemente è

$$D_n = 2n + n\lceil \log_2(\frac{u}{n}) \rceil + o(n)$$

In questo frangente, per raffinare l'analisi, conviene considerare la differenza col lower bound non sull'intera struttura, bensì sul singolo elemento. Definiamo, quindi, il lower bound per l'elemento

$$\bar{Z}_n \approx \log_2(\frac{u}{n})$$

e

$$\bar{D}_n = 2 + \lceil \log_2(\frac{u}{n}) \rceil + o(n) = \bar{Z}_n + O(1)$$

la struttura è 'quasi' implicita, dove l'incertezza deriva dal numero di approssimazioni fatte e sull'assunzione di sparsità; in pratica, ciò che succede in pratica è che questa uguaglianza vale quando $n \leq \sqrt{u}$.

5.5 Struttura per parentesi ben formate

5.5.1 Linguaggi di Dyck

Un linguaggio di Dyck L è un linguaggio sull'alfabeto $D = (,)$ tale che $L \subseteq D^*$ e una stringa $w \in D^*$ appartiene a L se e solo se

1. $|w|_(= |w|_)$; e
2. $\forall w_1, w_2 \; w = w_1 w_2 \implies |w_1|_(\geq |w_2|_)$

Un modo interessante per studiare una parola w di Dyck è studiarne la *funzione di eccesso*, definita

$$E_w(i) = |\{j | j \leq i \; w_j = (\}| - |\{j | j \leq i \; w_j =)\}|$$

Questa funzione parte da 0, può tornare a 0 e, se la parola è effettivamente nel linguaggio, termina a 0. Ci sono due tipi di stringhe di Dyck: quelle per cui la funzione non raggiunge nessuno 0 tranne quello

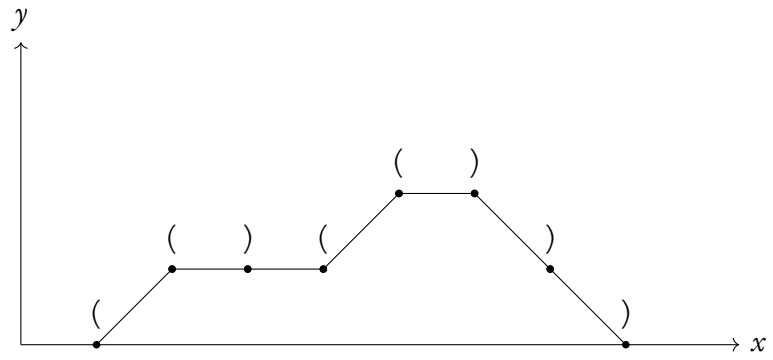


Figura 5.11: Funzione di eccesso per la parola (()(())).

iniziale e quello finale, chiamate *fortemente bilanciate*, e quelle per cui questa funzione raggiunge uno 0 diverso da quello iniziale e quello finale, quindi $w = w_1 w_2$ è *debolmente bilanciata* e tale per cui almeno uno tra w_1 e w_2 è fortemente bilanciata.

Ci sono molti motivi per cui le sequenze di parentesi sono interessanti. In particolare, le espressioni ben parentesizzate sono in biiezione sia con le foreste di alberi binari che con gli alberi non binari. In qualche modo, esattamente come abbiamo visto una rappresentazione per gli alberi binari in precedenza, ora stiamo analizzando una struttura per alberi generali. Le sequenze di parentesi aperte e chiuse le penseremo come stringhe di bit, cioè una stringa

(()(()())(()))

è rappresentata come una stringa

11011010011000

la cui funzione di eccesso è rappresentata in Fig. 5.12.

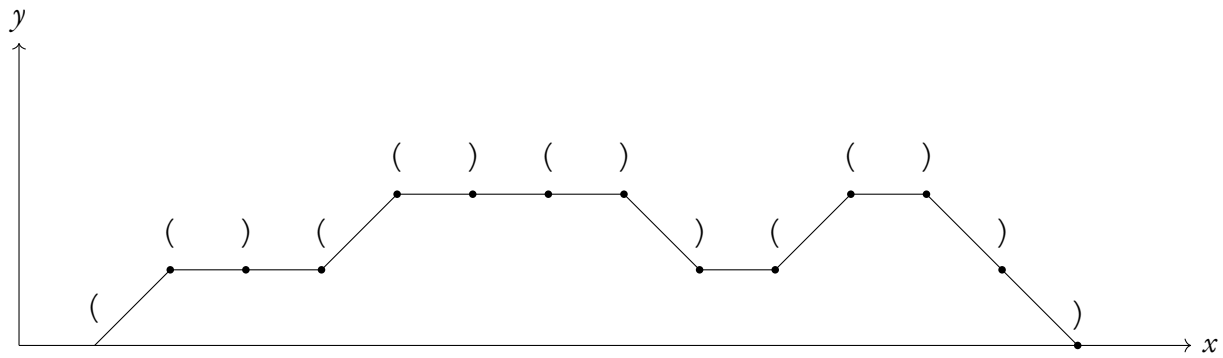


Figura 5.12: Funzione di eccesso per la parola (()(()())(())).

5.5.2 L'ADT stringa bilanciata

L'ADT che vogliamo descrivere per le stringhe di parentesi ben bilanciate ha le primitive

$\forall p \in \mathbb{N}$ **find_open**(p) = parentesi aperta corrispondente alla chiusa in posizione p

$\forall p \in \mathbb{N}$ **find_close**(p) = parentesi chiusa corrispondente all'aperta in posizione p

$\forall p \in \mathbb{N}$ **enclose**(p) = prima parentesi aperta che racchiude la parentesi in posizione p

Pensando ad una soluzione banale, una **find_open** in una qualche posizione si può calcolare innanzitutto realizzando che la parentesi aperta corrispondente ad una parentesi chiusa è sicuramente prima della chiusa stessa; basta quindi cercare all'indietro fermandosi alla prima parentesi aperta che ha la stessa funzione di eccesso. Analogamente si fa per **find_close**. La cosa più semplice, quindi, sarebbe calcolare anticipatamente la funzione di eccesso e usarla facendo ricerche lineari, utilizzando uno spazio $n \log_2(n)$. Faremo meglio creando una struttura succinta con tempo di interrogazione logaritmico.

Rappresentazione

La prima operazione sulle parentesi da eseguire nella costruzione della struttura è dividere, nuovamente, la parola in blocchi di lunghezza l, creando $k = \lceil n/l \rceil$ blocchi. In ogni blocco ci sono parentesi chiuse e aperte e, in alcuni casi, le parentesi hanno la loro corrispondente dentro il blocco stesso: in questa situazione, definiamo la parentesi *vicina*, mentre tutte le altri sono definite *lontane*; in generale, tutte le parentesi aperte lontane si chiuderanno in un qualche blocco successivo.

In ogni blocco ci sono alcune parentesi vicine e varie parentesi lontane. Può succedere che una o più parentesi lontane si chiudano nello stesso blocco: definiamo *pioniera* la lontana aperta (benché si possa fare lo stesso ragionamento, a specchio, per le chiuse) che è la prima del blocco al quale appartiene a chiudersi in un qualsiasi altro blocco successivo. Un esempio di queste divisioni è in Fig. 5.13, dove le parentesi gialle sono le vicine, quelle blu sono lontane e quelle rosse sono pioniere.

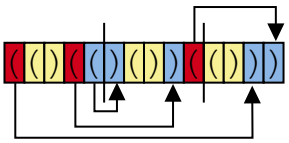


Figura 5.13: Divisione in blocchi di una stringa parentesizzata.

Assumiamo che w sia la parola da rappresentare e assumiamo che $|w| = n$. Per rappresentare la parola memorizziamo, oltre a w stessa, un vettore **p** di n bit con 1 nelle posizioni delle pioniere; il vettore **E**, che per ogni blocco i da l'eccesso all'inizio del blocco e ha un elemento per ogni blocco; il vettore **M**, che per ogni blocco i mantiene la posizione della parentesi corrispondente all'i-esima pioniera e, infine, il vettore **O** che per ogni blocco i mantiene la posizione della prima aperta a sinistra dell'inizio del blocco avente eccesso $x-1$, dove x è il minimo eccesso del blocco. Per la rappresentazione, definiamo $l = \log_2(n)$.

Teorema 5.9. *Se ci sono k blocchi, vi sono al massimo $2^k - 3$ coppie di pionieri.*

Dimostrazione. Costruiamo un grafo $G = (V, E)$ dove V sono i blocchi ed esiste un lato tra un blocco x e y se e solo se x contiene una pioniera cha ha in y la sua corrispondente. Dimostriamo per induzione su k: vi sono due casi.

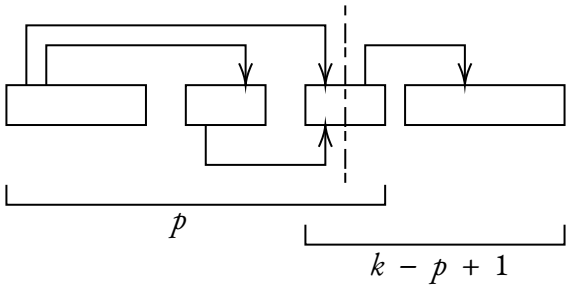
- se l'insieme di blocchi è separabile, ossia esiste una posizione nella parola sulla quale "non passano archi", la parola è debolmente bilanciate ed è scomponibile in due diverse parole ben formate. Allora, per ipotesi induttiva, il numero di pioniere nella prima parte è al più $2p - 3$ e il numero di pioniere nella seconda parte è al più $2(k - p + 1) - 3$; allora il numero di pioniere è al più

$$2p - 3 + 2k - 2p + 2 - 3 = 2k - 4 \leq 2k - 3$$

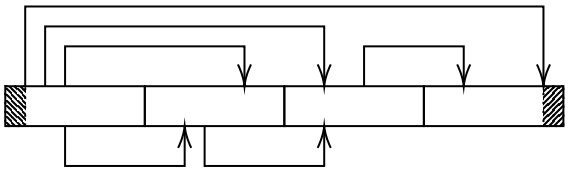
- se l'insieme di blocchi non è separabile, ossia la parola è fortemente bilanciate e non è scomponibile in due parole diverse, si prende la prima coppia di parentesi lontane che si trova in w e si rimuove. La parola risultante, che chiamiamo w', gode della proprietà che si vuole dimostrare. Il numero di pioniere in w è quindi al più la somma delle pioniere in w' e la pioniera rimossa

$$2k - 4 = 2k - 3$$

□



(a) Caso 1.



(b) Caso 2.

Figura 5.14: Dimostrazione induttiva del numero di pioniere in una parola ben parentesizzata.

Quindi, lo spazio occupato da queste strutture è:

w	n
p	$n + o(n)$ (sarà necessario introdurre la struttura di rango)
E	$k \log_2(n)$
O	$k \log_2(n)$
M	pioniere $\cdot \log_2(n) < 2(2k-3)\log_2(n) = (4k-6)\log_2(n) \leq 4k \log_2(n)$

Sommando, lo spazio occupato è $D_n = 2n + o(n) + 6k \log_2(n) = 2n + 6n + o(n) = 8n + o(n)$ bit.

Implementrazione

Per implementare la funzione **find_close** che, trovata una parentesi aperta in posizione p, restituisce la posizione in cui si chiude, si deve:

1. calcolare gli eccessi di tutte le posizioni del blocco al quale appartiene p con E (tempo logaritmico);

2. se p è vicina, ossia esiste un'altra posizione nel blocco con lo stesso eccesso, la funzione è completa. Altrimenti, p è la posizione di una parentesi lontana e $j = \mathbf{rank_p}(p)$ è l'indice della pioniera che precede p; si può usare $M[j] = p'$ per calcolare la posizione in cui si chiude la pioniera che precede p, che diciamo essere in un blocco b. Scorrendo indietro, per trovare la chiusa corrispondente, basta trovare la posizione con eccesso uguale all'eccesso di p.

Tutto questo richiede tempo $\log_2(n)$ e analogamente per l'implementazione di **find_open**. Per l'implementazione di **enclosed**, che cerca la prima parentesi aperta che racchiude la parentesi in posizione p, ipotizziamo di aver già trovato la corrispondente $\bar{p}$ con una find e ammettiamo che l'eccesso di entrambe sia e. Inizialmente, si cerca alla sinistra della posizione della parentesi aperta se c'è una posizione con eccesso $e - 1$, che segna l'aperta precedente. Se, nel blocco al quale appartiene l'aperta, tutte le posizioni hanno eccesso $\geq e - 1$, si cerca per una chiusa di eccesso $e - 1$ nella parte a destra del blocco in cui giace la chiusa. Se anche in questo caso non si trova, significa che la parentesi che si cerca è di eccesso $e - 1$ che giace nel blocco a sinistra a quello della parentesi aperta, che ha come valore di eccesso minore di tale blocco. In tal caso, basta usare il vettore **O** per trovare la posizione della parentesi cercata.

5.5.3 Lower bound per parentesi ben bilanciate

Foreste ordinate

Una foresta ordinata è una sequenza ordinata (per numero di nodi) di alberi ordinati (corrispondenti ad alberi radicati). Formalmente, definiamo induttivamente una foresta ordinata:

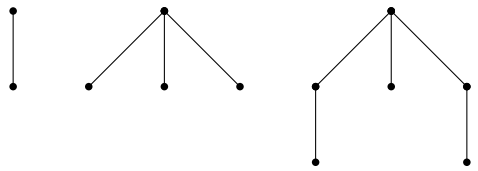


Figura 5.15: Foresta ordinata di alberi.

- $\langle\rangle$ indica la lista vuota di alberi ed è una foresta ordinata;
- se $\langle T_1, \cdots, T_k\rangle$ sono alberi, allora $\langle T_1, \cdots, T_k\rangle$ è una foresta ordinata
- se F è una foresta ordinata, $tree(F)$ è un albero radicato in un nuovo nodo che ha come figli tutti gli alberi di F.

Isomorfismo tra foreste ordinate e alberi binari

Definiamo una funzione ϕ che associa ad ogni foresta ordinata un preciso albero binario. Induttivamente,

$$\phi(\langle\rangle) = \cdot$$

ossia ϕ di una foresta vuota è un albero costituito da una singola radice.

$$\phi(\langle tree(F), T_1, \cdots, T_k >\rangle) = (\phi(F), \phi(\langle T_1, \cdots, T_k\rangle))$$

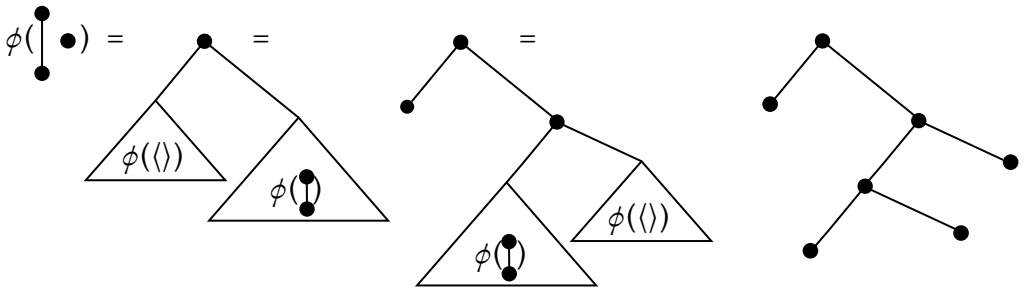


Figura 5.16: Esempio di calcolo isomorfismo tra foreste e alberi binari.

Ciò che abbiamo dimostrato è che le foreste ordinate sono tante quante gli alberi binari: ϕ 'manda' una foresta ordinata con n nodi interni in un albero binario con altrettanti nodi.

Isomorfismo tra foreste ordinate e parole di Dyck.

Definiamo $\psi : D_{2n} \to F_n$ un isomorfismo tra parole ben parentesizzate di lunghezza $2n$ e foreste ordinate con n nodi induttivamente:

$$\psi(\epsilon) = \langle\rangle$$

$$\psi(w_1 \cdots w_k) = \langle\psi(w_1), \cdots, \psi(w_k)\rangle$$

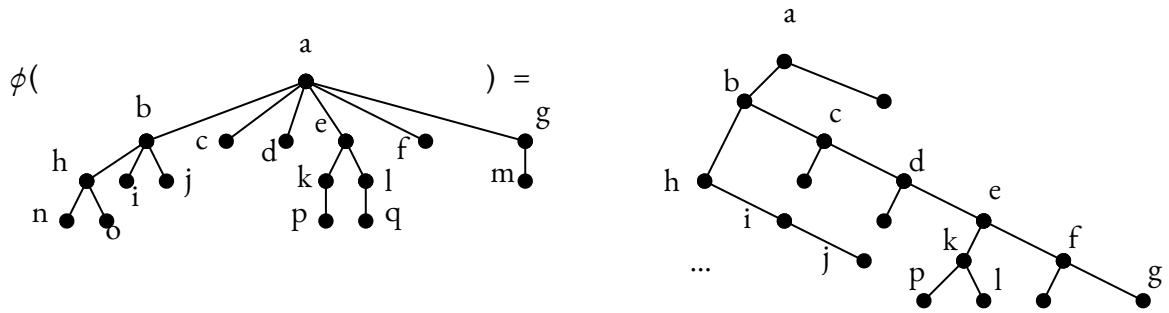


Figura 5.17: Funzionamento, in generale, dell'isomorfismo tra foreste ordinate e alberi binari con la tecnica 'first-child next-sibling'.

e

$$\psi((w)) = \langle \text{ albero radicato con figli tutti e soli gli alberi della foresta} \phi(w) \rangle$$

Sappiamo che $D_{2n} \cong F_n \cong B_n$ e sappiamo quanti sono i possibili alberi binari con n nodi sono $|B_n| = C_n \approx 2n + o(\log_2(n))$; di conseguenza, le parole di Dyck di lunghezza $2n$ hanno come information-theoretical lower bound

$$Z_n = n + o(\log_2(n)) \text{ bit}$$

Quindi

$$D_n - Z_n = 8n + o(n) - n - o(\log_2(n)) = 7n - o(n) - o(\log_2(n)) = O(n) \implies D_n \approx Z_n + O(Z_n)$$

pertanto la struttura descritta è compatta.

5.6 Struttura per hash minimali perfetti

5.6.1 Funzioni di hash

Le funzioni di hash compaiono in molti contesti diversi e un'applicazione 'famose' sono le *tabelle di hash*. Nel modo più generale, dato un universo U infinito o *molto grande* e un numero $m \in \mathbb{N}$, che è il *numero di bucket*, una funzione di hash è

$$h : U \to m$$

le funzioni di hash, che denominiamo $H_{U,m}$ è infinito se U è infinito, mentre è finito se U è finito, in particolare $|H_{U,m}| = m^{|U|}$. La funzione h deve avere alcune proprietà: prendiamo come esempio proprio le tabelle di hash, utilizzate per memorizzare un sottoinsieme $S \subseteq U$, per esempio delle stringhe su un certo alfabeto $\Sigma = \{a, b, c, d\}$. Fissato un $m \in \mathbb{N}$, la funzione è

$$h : \Sigma^* \rightsquigarrow m$$

vedendo i simboli in Σ come un'enumerazione $a = 0, b = 1, c = 2, d = 3$, una funzione di hash potrebbe semplicemente sommare i simboli di una stringa e operarne il modulo in m, in modo che il valore risultante sia $0 \leq h \leq m - 1$.

Lo scopo è mantenere una tabella, spesso realizzata come un vettore di list di elementi, i cui indici sono esattamente gli hash possibili che la funzione h calcola. In questo modo, è possibile accedere in modo (quasi) diretto alle parole nella tabella[4]. In termini pratici, partendo da una tabella vuota, quando si vuole inserire una stringa $s = "foo"$ nella tabella, si utilizza la funzione di hash per calcolare $h(s) = h_1$. Il valore calcolato sarà l'indice della tabella dove inserire il riferimento alla stringa.

[4]Vi sono principalmente due modi per costruire tabelle di hash: *open addressing* e *separate chaining*. In questo frangente stiamo analizzando la versione con separate chaining.

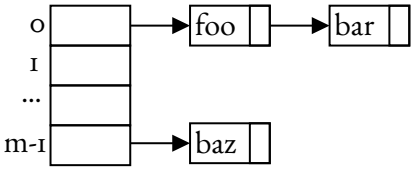


Figura 5.18: Esempio di una tabella di hash con separate chaining.

Possono accadere dei conflitti, ossia due stringhe s_1 e s_2 tali che $h(s_1) = h(s_2)$; per ovviare a questo problema si descrive M come una mappa di liste, ossia ad un indice h_1 di M corrisponde una lista di stringhe.

Requisiti

La tabella M funziona con qualsiasi funzione di hash h (anche $h(\cdot) = 0$), ma l'efficienza di M cambia al suo variare, fino al denegenerare in una lista. Per far funzionare "bene" M, h deve essere

- h sia veloce da calcolare; e
- h divide l'universo U in *buckets* in modo tale che le controimmagini siano più o meno *grandi uguali*. In altri termini, se guardiamo Σ^* e guardiamo come h ne partiziona gli elementi, vorremmo che i blocchi (o bucket) della partizione siano più o meno grandi uguali e non ci sia qualche blocco che contiene molti più elementi degli altri.

Questi requisiti sono esprimibili formalmente ovviamente, ma solitamente oltre a questi requisiti di base ve ne sono altri che dipendono dall'utilizzo che si vuole fare della funzione; nel nostro caso, facciamo le seguenti assunzioni: la prima, chiamata *full randomness assumption*, è che possiamo estrarre uniformemente una funzione h dall'insieme $\mathcal{H}_{U,m}$.

La seconda è che h sia calcolabile in tempo e spazio costante, inoltre occupa spazio costante in termini di codice (non usa array arbitrariamente grandi, ...). Questa assunzione è normalmente inattuabile: si pensi al caso in cui $U = \Sigma^*$: si dovrebbero considerare tutte le possibili funzioni dalle stringhe ai naturali minori di m, che sono infinite e tra le quali ce ne sono anche di non calcolabili.

Rappresentazione

Ciò che si può fare è invece considerare un universo limitato con un upper bound dipendende da un qualche intero k. Chiaramente si perde della 'randomness', ma il vantaggio è che molto spesso i risultati che si ottengono sotto l'assunzione di completa casualità si possono portare anche sotto U così definito, magari con qualche approssimazione.

Supponiamo quindi di avere $U = \Sigma^{\leq k}$; vogliamo scrivere delle funzioni $h : U \to m$ e un modo per descriverle è il seguente. Si usa un array $\mathbf{p}$ chiamato *array dei pesi* contenente k valori, inizializzandolo a valori pseudocasuali nell'insieme $\{0, \cdots, m-1\}$. Quando si vuole calcolare l'hash di una stringa $s = "foo"$ si considera il valore di ogni lettera della stringa e si moltiplica per il peso di quel carattere in senso posizionale, ossia il primo peso è associato al primo carattere della stringa, il secondo peso al secondo carattere e così via; quindi si sommano i risultati e si computa il modulo m. Per esempio, se

$$a = 0, b = 1, c = 2, \cdots, f = 5, \cdots, o = 14, \cdots$$

e

$$\mathbf{p} = [15, 86, 90, \cdots]$$

Il calcolo dell'hash è

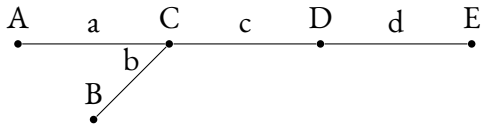
$$h("foo") = ((5 * 15) + (5 * 86) + (14 * 90)) \mod m$$

ottendo un valore che si calcola in un tempo lineare nella lunghezza della stringa diverso per ogni possibile scelta dei pesi. Cioè, non stiamo scegliendo tra tutte le possibili funzioni di hash, bensì scegliamo tra le funzioni di hash caratterizzabili dal vettore **p**, ossia in m^k modi. Queste funzioni non occupano molto spazio e si calcolano in un tempo non costante ma logaritmico nella dimensione dell'universo.

5.6.2 Relazione con i grafi

Sequenza di peeling di un grafo

Si supponga di avere un grafo $G = (V, E)$ non orientato. Una **sequenza di peeling** è una sequenza di coppie di archi e vertici in cui appaiono tutti i lati e uno dei due vertici incidenti a tale lato. Ogni vertice che appare è chiamato *hinge* della sequenza. La sequenza deve essere tale per cui nessun vertice hinge x_i è apparso nei lati che precedono i.



$$a = \{(A, C), A\}$$
$$b = \{(B, C), B\}$$
$$c = \{(C, D), D\}$$
$$d = \{(D, E), E\}$$

Figura 5.19: Un grafo e una sequenza di peeling valida.

Prendendo come esempio il grafo in Fig. 5.19, per verificare una sequenza di peeling si deve innanzitutto verificare che tutti i lati appaiano nella sequenza; quindi, si verifica che ogni vertice non sia mai comparso prima: per esempio, avendo scelto il lato (A, B) invece di (A, C), non si sarebbe potuto scegliere il vertice B da associare al lato (B, C).

Non tutti i grafi ammettono una sequenza di peeling:

Teorema 5.10. *Un grafo G ammette una sequenza di peeling se e solo se è aciclico.*

Dimostrazione. $\implies$ Per assurdo, si supponga che $\langle \{e_0, x_0\}, \cdots, \{e_{m-1}, x_{m-1}\} \rangle$ sia una sequenza di peeling e che esista un ciclo sui vertici $y_1, y_2, \cdots, y_k$ e i relativi lati $\{y_1, y_2\}, \cdots, \{y_{k-1}, y_k\}$. Sia $\bar{i}$ l'indice massimo della sequenza del ciclo. Inserendo tutti i lati del ciclo nella sequenza, quando si arriva ad inserire l'ultimo lato del ciclo, non ci sarà modo di scegliere un nodo che ancora non appare nella sequenza.

$\impliedby$ Per induzione su $|E|$. Omessa. □

Ipergrafi

Vogliamo ora generalizzare la nozione di sequenza di peeling agli ipergrafi.

Definizione Un r-ipergrafo è $G = (V, E)$ di vertici e *iperlati* dove ogni lato è un insieme di r vertici, ossia $E \subseteq \binom{V}{r}$. Un esempio è in Fig. 5.20.

Non esiste una nozione di aciclicità per ipergrafi, mentre esiste una nozione di sequenza di peeling: una sequenza di peeling per un r-ipergrafo è una sequenza dei suoi iperlati ai quali si associa un hinge in modo tale che non sia mai apparso negli iperlati precedenti. Per questo motivo non si generalizza la nozione di aciclicità bensì quella di sequenza di peeling; si dice che un ipergrafo è aciclico se e solo se ammette una sequenza di peeling.

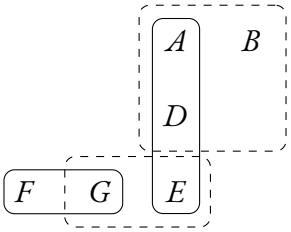


Figura 5.20: Esempio di ipergrafo.

5.6.3 Tecnica MWHC

Funzioni statiche

Il nostro obiettivo è memorizzare funzioni statiche. Dato un universo U, un sottoinsieme fissato $X \subseteq U$ e $r \in \mathbb{N}$ vogliamo memorizzare una funzione

$$f : X \to 2^r$$

di nuovo, si immagini U come l'insieme dei caratteri ASCII; una funzione f può essere come quella in Tabella 5.4, in cui X è l'insieme di quelle tre stringhe: a noi non interessa memorizzare stringhe diverse da quelle.

x	$f(x)$	
Paolo Boldi	00111	7
Anna Zuppi	101000	20
Giovanni Galli	101110	23

Tabella 5.4: Esempio di funzione statica.

Memorizzare una funzione significa che vogliamo ricavare una struttura dati D tale che permette di valutare un input in modo da ottenere il valore di $f(x)$ ossia, per esempio, $"PaoloBoldi" \mapsto 00111$ e così via, mentre il comportamento inteso per gli input non appartenenti all'insieme X che vogliamo memorizzare è irrilevante. La funzione f si definisce *statica* poiché è paragonabile alla struttura *dizionario* nei linguaggi di programmazione come Python, benché questa struttura sia immodificabile e non ci sia modo di sapere se una certa chiave è presente o meno.

Rappresentazione

Per costruire una rappresentazione succinta si utilizza la tecnica MWHC[5]. Inizialmente si fissa un m intero e si scelgono uniformemente due funzioni hash

$$h_1, h_2 : U \to m$$

Assumiamo che $\forall x \in X\ h_1(x) \neq h_2(x)$. Costruiamo un grafo i cui vertici sono i numeri $0, \cdots, m-1$ e i lati corrispondono agli elementi dell'insieme X, con l'idea che alla chiave x corrisponde il lato $(h_1(x), h_2(x))$.

Seguendo l'esempio in Tabella 5.4, immaginiamo che le due funzioni calcolino l'hash delle stringhe in X come descritto in Tabella 5.5, supponendo $m = 17$ e $r = 5$. Si può ora costruire un grafo costruito come

[5] Majewski, Worwald, Havas, Czech

in Fig. 5.21: grazie all'assunzione precedente, ossia la differenza dei due hash per ogni chiave, sappiamo che nel grafo non vi sono cappi. In caso accada, si generano due nuove funzioni h_1 e h_2. Inoltre, vogliamo che il grafo sia aciclico e che a nessun lato corrispondano due o più chiavi diverse: anche in questi casi si possono generare due nuove funzioni hash.

x	$h_1(x)$	$h_2(x)$
Paolo Boldi	3	7
Anna Zuppi	14	13
Giovanni Galli	13	2

Tabella 5.5: Calcolo di h_1 e h_2 sulle stringhe nell'insieme X.

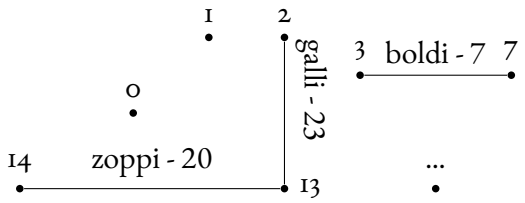


Figura 5.21: Grafo associato per la costruzione di una struttura per funzioni statiche.

Trasformeremo ora il grafo in un sistema di equazioni: ogni vertice è una variabile $w_0, w_1, \cdots, w_{m-1}$ e ad ogni lato corrisponde l'equazione

$$\forall x \in X \ (w_{h_1(x)} + w_{h_2(x)}) \mod 2^r = f(x)$$

Se il grafo è aciclico (che è un'assunzione) il sistema è risolvibile. L'esistenza di una sequenza di peeling (che esiste se e solo se il grafo è aciclico) significa che si possono ordinare le equazioni in modo che una delle due variabili non sia mai comparsa prima: questo significa che la soluzione si può trovare ordinando le equazioni e assegnando il valore che rende vera l'equazione alla variabile che non è ancora apparsa. Quindi, per esempio

$$\begin{cases} (w_{14} + w_{13}) \mod 2^5 = 20 \\ (w_2 + w_{13}) \mod 2^5 = 23 \\ (w_3 + w_7) \mod 2^5 = 7 \end{cases}$$

Usando come hinge $w_{14} = 20$, $w_2 = 23$ e $w_7 = 7$ si risolve il sistema.

Implementazione

Una volta memorizzate le soluzioni w_i del sistema in un vettore $\mathbf{w}$, si può calcolare $f(x)$ per ogni $x \in X$ come

$$f(x) = (w_{h_1(x)} + w_{h_2(x)}) \mod 2^m$$

Bisogna però decidere la grandezza di m, ossia il numero di vertici (o il numero di bucket, o il numero massimo che può risultare da un hash): è facile vedere che se m è "troppo piccolo" è molto imporbabile che riescano a soddisfare le condizioni, ossia si troveranno cicli, collisioni e così via. Se si sceglie "troppo grande", la struttura che si memorizza occupa molto spazio. Precisamente, questo tradeoff dipende da quanto è probabile che il grafo generato sia aciclico.

Teorema 5.11. *Se $m > (2.09 \cdot |X|)$, il grafo è "quasi sempre" aciclico. Il numero atteso di tentativi di generazione di coppie di funzioni hash è circa* 2.

Dimostrazione. Omessa. □

Naturalmente si può anche calcolare la funzione di un qualcosa che non è tra gli input desiderati e qualcosa verrà prodotto, ma non avrà un senso ben inteso.

Spazio

Il vettore che dobbiamo memorizzare ha m elementi, ognuno dei quali contiene r bit: in tutto, il vettore occupa spazio $m \cdot r$ bit. Definendo $|X| = n$, dal teorema precedente si ha $m \cdot r \geq 2.09nr$; lo spazio totale si trova aggiungendo lo spazio per memorizzare le funzioni di hash h_1 e h_2.

Tutto questo processo può essere eseguito non solo per i grafi (costruiti con 2 funzioni hash) ma anche per gli ipergrafi, costruendo un r-ipergrafo utilizzando r funzioni hash. Ci si può quindi chiedere quale sia la costante che rende ipergrafi con $r > 2$ "quasi sempre" aciclici:

Teorema 5.12. *Per ogni k-ipergrafo esiste una costante γ_k tale che se $m > \gamma_k n$ allora l'ipergrafo ammette quasi sempre una sequenza di peeling.*

Dimostrazione. Omessa. □

In effetti, le costanti γ_k hanno un minimo in $k = 3$: il meglio che si può ottenere è quindi utilizzando 3 funzioni di hash. Il numero di bit che consuma $\mathbf{w}$ è quindi $1.23nr$ bit.

k	γ_k
2	2.09
3	1.23
···	> 1.23 ···

Tabella 5.6: Costanti γ_k per k-ipergrafi.

Lower bound

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

$$Z_n = \log_2(2^{rn}) = rn \text{ bit}$$

e la struttura che abbiamo descritto occupa

$$D_n = 1.23nr = O(Z_n) \text{ bit}$$

pertanto la struttura è compatta.

Struttura succinta

Si può osservare che nel vettore $\mathbf{w}$ molte entry sono uguali a 0: il numero di entry diverse da zero è pari al numero di hinge nella sequenza di peeling che risolve il sistema generato dal grafo, quindi $\mathbf{w}$ che (assumendo l'utlizzo di 3 funzioni di hash) ha esattamente $m = 1.23n$ elementi, al più n elementi sono non nulli.

Quindi, si possono memorizzare unicamente gli elementi non nulli in $\tilde{\mathbf{w}}$ utilizzando un ulteriore array $\mathbf{b}$ di m bit tale che

$$\mathbf{b}[i] = \begin{cases} 1 & \mathbf{w}[i] \neq 0 \\ 0 & \mathbf{w}[i] = 0 \end{cases}$$

definendo quindi il vettore

$$\mathbf{w}[i] = \begin{cases} 0 & \mathbf{b}[i] = 0 \\ \tilde{\mathbf{w}}[\mathbf{rank}_{\mathbf{b}}[i]] & \mathbf{b}[i] = 1 \end{cases}$$

In questo modo, la struttura (che è comunque compatta) in totale occupa

$$D_n = nr + m = nr + 1.23n = (r + 1.23)n \text{ bit}$$

e si ha quindi

$$(r + 1.23)n < 1.23rn \implies (r + 1.23) < 1.23r \implies r > 5$$

Inizialmente, la tecnica MWHC era stata pensata per un uso ben più specifico, ossia memorizzare una funziona di hash minimale perfetta, ossia una funzione che mappa n chiavi in n bucket privi di collisioni. Tuttavia questa tecnica è abbastanza generale da memorizzare qualsiasi funzione e si può utilizzare questa tecnica come intesa inizialmente semplicemente associando $f(x)$ diversi da 0 a $n - 1$ per ogni elemento di X.

APPENDICE A

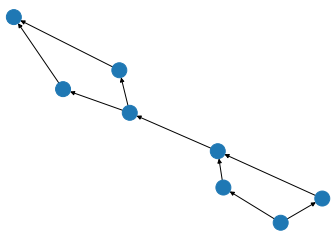
Laboratorio 1: Cammini disgiunti tramite algoritmo basato su pricing

```
import networkx as nx
import math
from networkx.algorithms.shortest_paths.weighted import dijkstra_path
```

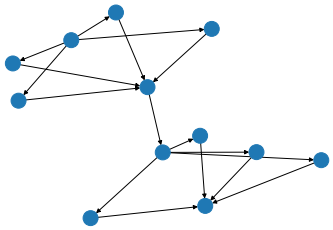
Definiamo una funzione per generare grafi in una forma precisa, ossia a *doppio ventaglio*:

```
def doubleFan(k):
    G = nx.DiGraph()
    G.add_nodes_from(['s', 't', 'x', 'y'])
    G.add_nodes_from(['a' + str(i) for i in range(k)])
    G.add_nodes_from(['b' + str(i) for i in range(k)])
    G.add_edges_from([('s', 'a' + str(i)) for i in range(k)])
    G.add_edges_from([('a' + str(i), 'x') for i in range(k)])
    G.add_edges_from([('y', 'b' + str(i)) for i in range(k)])
    G.add_edges_from([('b' + str(i), 't') for i in range(k)])
    G.add_edge('x', 'y')
    return G
```

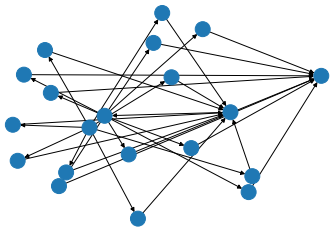
```
nx.draw(doubleFan(2))
```



```
nx.draw(doubleFan(4))
```



```
nx.draw(doubleFan(8))
```

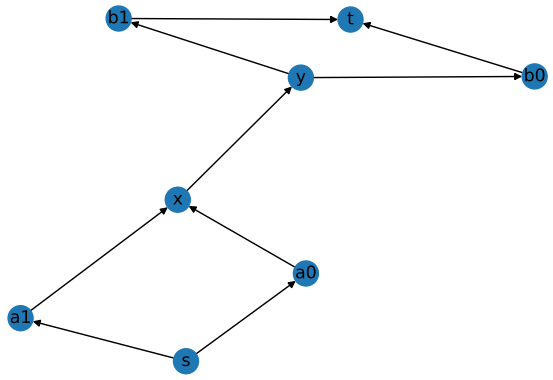


Definiamo, quindi, una funzione che implementa PricedDisjointPaths:

```
def greedyDisjointPaths(G_original, sourceTargetPairs, c = 1):
    G = G_original.copy()
    result = []
    beta = math.pow(G.number_of_edges(), 1 / (c + 1))
    # Set all lengths to 1 and all congestion to 0
    for u,v,d in G.edges(data = True):
        d['length'] = 1
        d['congestion'] = 0
    # Main cycle
    while True:
        minPath = None
        for pairIndex in range(len(sourceTargetPairs)):
            try:
                source = sourceTargetPairs[pairIndex][0]
                target = sourceTargetPairs[pairIndex][1]
                path = dijkstra_path(G, source, target, 'length')
            except:
                pass
            else:
                pathLength = 0
                for i in range(len(path) - 1):
                    pathLength += G[path[i]][path[i+1]]['length']
                if minPath == None or pathLength < minPathLength:
                    minPath = path
                    minPathLength = pathLength
                    minPathIndex = pairIndex
        if minPath == None:
            break
        result.append(minPath)
        sourceTargetPairs.pop(minPathIndex)
```

```
        for i in range(len(path) - 1):
            x1 = path[i]
            x2 = path[i+1]
            G[x1][x2]['length'] *= beta
            G[x1][x2]['congestion'] += 1
            if G[x1][x2]['congestion'] == c:
                G.remove_edge(x1, x2)
    return result
```

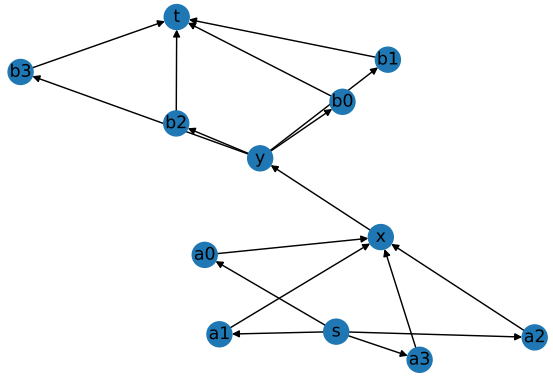
```
g = doubleFan(2)
nx.draw(g, with_labels = True)
```




```
greedyDisjointPaths(g, [('s', 't')]*11, c = 10)
```

```
[['s', 'a0', 'x', 'y', 'b0', 't'],
 ['s', 'a1', 'x', 'y', 'b1', 't'],
 ['s', 'a0', 'x', 'y', 'b0', 't'],
 ['s', 'a1', 'x', 'y', 'b1', 't'],
 ['s', 'a0', 'x', 'y', 'b0', 't'],
 ['s', 'a1', 'x', 'y', 'b1', 't'],
 ['s', 'a0', 'x', 'y', 'b0', 't'],
 ['s', 'a1', 'x', 'y', 'b1', 't'],
 ['s', 'a0', 'x', 'y', 'b0', 't'],
 ['s', 'a1', 'x', 'y', 'b1', 't']]
```

```
g = doubleFan(4)
nx.draw(g, with_labels = True)
```




```
greedyDisjointPaths(g, [('s', 't')]*11, c = 10)
```

```
[['s', 'a0', 'x', 'y', 'b0', 't'],
 ['s', 'a1', 'x', 'y', 'b1', 't'],
 ['s', 'a2', 'x', 'y', 'b2', 't'],
 ['s', 'a3', 'x', 'y', 'b3', 't'],
 ['s', 'a0', 'x', 'y', 'b0', 't'],
 ['s', 'a1', 'x', 'y', 'b1', 't'],
 ['s', 'a2', 'x', 'y', 'b2', 't'],
 ['s', 'a3', 'x', 'y', 'b3', 't'],
 ['s', 'a0', 'x', 'y', 'b0', 't'],
 ['s', 'a1', 'x', 'y', 'b1', 't']]
```

APPENDICE B

Laboratorio 2: il problema dello zaino

```
import numpy as np
import random
```

Definiamo una funzione helper per creare istanze arbitrarie del problema:

```
# n is the number of objects to generate
# maxv is the maximum value
def generateInstance(n, maxv = 100):
    w = random.sample(range(1, maxv), n)
    v = random.sample(range(1, maxv), n)
    wBound = max(int(random.random() * maxv), max(w))
    return (list(zip(w, v)), wBound)
```

Definiamo quindi il primo metodo basato sulla matrice dei valori $vOPT$.

```
# wv is a list of pairs (w,v)
# wBound is the capacity of the knapsack
# returns a pair (I, v) where v is the optimal value of a solution
# and I is the solution (set of indices)
def knapsackVopt(wv, wBound):
    n = len(wv)
    vOpt = np.zeros((n + 1, wBound + 1), int)
    for i in range(1, n + 1):
        vOpt[i][0] = 0
        for w in range(1, wBound + 1):
            currentItemWv = wv[i-1]
            currentW = currentItemWv[0]
            currentV = currentItemWv[1]
            if w >= currentW:
                vOpt[i][w] = max(vOpt[i-1][w], vOpt[i-1][w-currentW]+currentV)
            else:
                vOpt[i][w] = vOpt[i-1][w]
    I = []
    i = n
    w = wBound
```

```
    while i > 0:
        if vOpt[i-1,w] != vOpt[i, w]:
            I.append(i-1)
            w -= wv[i-1][0]
        i -= 1
    return (I,vOpt[n][wBound])
```

```
a = generateInstance(5, 10)
print(a)
print(knapsackVopt(*a))
```

```
([(3, 4), (9, 8), (4, 9), (2, 6), (1, 3)], 9)
([3, 2, 0], 19)
```

Definiamo il secondo metodo, basato sulla matrice di pesi $wOPT$.

```
# wv is a list of pairs (w,v)
# wBound is the capacity of the knapsack
# returns a pair (I, v) where v is the optimal value of a solution
# and I is the solution (set of indices)
def knapsackWopt(wv, wBound):
    n = len(wv)
    v = [wv[i][1] for i in range(n)]
    w = [wv[i][0] for i in range(n)]
    vMax = max(v)
    nvMax = n * vMax
    # Dynamic programming matrix
    wOpt = np.zeros((n + 1, nvMax + 1))
    print("Dimensione tabella: ", wOpt.nbytes)
    # Initialization
    for a in range(1, nvMax + 1):
        wOpt[0][a] = float('inf')
    # Filling
    for i in range(1, n + 1):
        wOpt[i][0] = 0
        for a in range(1, nvMax + 1):
            wOpt[i][a] = min(
                    wOpt[i - 1][a],
                    wOpt[i - 1][max(a - v[i - 1], 0)] + w[i - 1])
    # Find the solution value a (on the last row)
    for a in range(nvMax, 0, -1):
        if wOpt[n][a] <= wBound:
            break
    # Reconstruct the solution
    finalV = a
    I = []
    i = n
    b = a
    while i > 0:
```

```
        if wOpt[i-1][b] != wOpt[i][b]:
            I.append(i-1)
            b -= v[i-1]
        i -= 1
    return (I,finalV)
```

L'algoritmo **FPTAS** utilizza proprio quest'ultimo algoritmo appena definito:

```
from math import ceil
def knapsackFPTAS(wv, wBound, epsilon):
    n = len(wv)
    v = [wv[i][1] for i in range(n)]
    w = [wv[i][0] for i in range(n)]
    vMax = max(v)
    theta = max(epsilon * vMax / (2 * n), 1.0)
    vHat = [int(ceil(v[i] / theta)) for i in range(n)]
    print("Prima dell'arrotondamento: ", wv, wBound)
    print("Dopo l'arrotondamento: ", list(zip(w, vHat)), wBound)
    I, opt = knapsackWopt(list(zip(w, vHat)), wBound)
    vOpt = sum([v[i] for i in I])
    return I, vOpt
```

```
a = generateInstance(15, 10000)
Iexact, vexact = knapsackWopt(*a)
I, v = knapsackFPTAS(*a, 0.3)
print(Iexact, vexact)
print(I, v)
```

```
Dimensione tabella:  17850368
Prima dell'arrotondamento:  [(1042, 3555), (9804, 9297), (376, 2780),
(2421, 277), (6697, 5289), (1202, 4197), (7569, 1164), (1896, 1082),
(1183, 7622), (1546, 1473), (7089, 6509), (3244, 4382), (8391, 1358),
(1373, 7811), (1454, 8170)] 9804
Dopo l'arrotondamento:  [(1042, 39), (9804, 100), (376, 30), (2421,
3), (6697, 57), (1202, 46), (7569, 13), (1896, 12), (1183, 82), (1546,
16), (7089, 71), (3244, 48), (8391, 15), (1373, 85), (1454, 88)] 9804
Dimensione tabella:  192128
[14, 13, 11, 8, 5, 0] 35737
[14, 13, 11, 8, 5, 0] 35737
```

Bibliografia

[AB08] Scott Aaronson and Chris Bourke. The complexity zoo, 2008. [Online; accessed 2022-01-24].

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, USA, 2006.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[KMA82] Assaf J. Kfoury, Robert N. Moll, and Michael A. Arbib. *A Programming Approach to Computability*. Texts and Monographs in Computer Science. Springer, 1982.

[Sak09] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[Sha48] Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.