Algoritmi e strutture dati

Giacomo Fantoni
Telegram: @GiacomoFantoni
Github: https://github.com/giacThePhantom/AlgoritmiStruttureDati

18 aprile 2020

Indice

Capitolo 1

Introduzione

Problema computazionale

Dati un dominio di input e uno di output, un problema computazionale è rappresentato dalla funzione matematica che associa un elemento del dominio di output ad ogni elemento del dominio di input.

Algoritmo

Dato un problema computazionale, un algoritmo è un procedimento effettivo espresso tramite una funzione di passi elementari ben specificati in un sistema formale di calcolo che risolve il problema in tempo finito.

1.1 Descrizione di un algoritmo

Per descrivere un algoritmo si rende necessario utilizzare un linguaggio formale ben definito detto pseudo-codice, indipendente dall'implementazione effettiva ma con dettaglio sufficiente a descrivere i passaggi necessari alla descrizione dell'algoritmo.

1.1.1 Pseudo-codice

- $a = b$.
- $a \leftrightarrow b \equiv tmp = a; a = b; b = tmp$.
- $T[]A =$ **new** $T[1 \dots n]$.
- $T[][]A =$ **new** $T[1 \dots n][1 \dots m]$.
- Tipi in grassetto.
- **and**, **or**, **not**.
- $==, \neq, \geq, \leq$.
- $+, -, \cdot, /, \lfloor x \rfloor, \lceil x \rceil, \log, x^2, \cdots$.
- $iif(condizione, v_1, v_2)$.

- **if** *condizione* **then** *istruzione*.
- **if** *condizione* **then** $istruzione_1$ **else then** $istruzione_2$.
- **while** *condizione* **do** *istruzione*.
- **foreach** *elemento* $\in$ *insieme* **do** *istruzione*.
- **return**
- % commento.

1.2 Valutazione degli algoritmi

1.2.1 Efficienza

Si definisce complessità di un algoritmo l'analisi delle risorse necessarie per la sua risoluzione, in funzione di tipologia e dimensione di input. Le risorse si distinguono in tempo, memoria e banda (per gli algoritmi distribuiti).

Tempo

Il numero di secondi necessari alla risoluzione dell'algoritmo dipende da troppi fattori, si utilizzano pertanto tecniche di analisi che prendono in considerazione il numero di operazioni rilevanti, quelle che caratterizzano lo scopo dell'algoritmo.

1.2.2 Correttezza

Per valutare la correttezza di algoritmi si devono considerare le invarianti:

- Invariante: una condizione che deve rimanere vera sempre in un certo punto del programma.
- Invariante di ciclo: una condizione che deve rimanere vera all'inizio dell'iterazione di un ciclo.
- Invariante di classe: una condizione sempre vera al termine dell'esecuzione di un metodo di una classe.

Invariante di ciclo e algoritmi iterativi

L'invariante di ciclo permette di dimostrare la correttezza degli algoritmi iterativi attraverso il principio di induzione:

- Inizializzazione (caso base): l'invariante è vera prima della prima iterazione.
- Conservazione (passo induttivo): se la condizione è vera prima di un'interazione allora rimane vera al suo termine.
- Conclusione: quando il ciclo termina l'invariante deve rappresentare la correttezza dell'algoritmo.

Capitolo 2

Analisi di algoritmi

Per definire la complessità di un algoritmo occorre definire una funzione che ha come dominio la dimensione dell'input e come insieme immagine il tempo.

Dimensione dell'input

La dimensione dell'input può essere definita secondo due criteri:

- Criterio di costo logaritmico: la taglia dell'input è il numero di bit necessari a rappresentarlo.
- Criterio di costo uniforme: la taglia dell'input è il numero di elementi di cui è costituito.

In molti casi si può assumere che gli elementi siano costituiti da un numero costante di bit, e in tal caso le due misure coincidono a meno di una costante moltiplicativa.

Definizione di tempo

Si definisce il tempo come il numero di istruzioni elementari necessarie al completamento dell'algoritmo. Si definisce elementare una funzione che può essere svolta in tempo costante dal processore.

2.1 Modelli di calcolo

Un modello di calcolo è una rappresentazione astratta del calcolatore. L'astrazione permette di nascondere dei dettagli, il suo realismo permette di riflettere con un certo grado di precisione una situazione reale e la potenza matematica di trarre conclusioni formali sul costo.

2.1.1 Macchina di Turing

Una macchina di Turing è una macchina ideale che manipola i dati contenuti su un nastro di lunghezza infinita secondo un insieme prefissato di regole. Ad ogni passo la macchina di Turing:

- Legge il simbolo sotto la testina.
- Modifica il proprio stato interno.
- Scrive un nuovo simbolo nella cella.

- Muove la testina a destra o a sinistra.

Questo modello è fondamentale per lo studio della calcolabilità ma è troppo dettagliato per l'analisi.

2.1.2 Random Access Machine (RAM)

Memoria

La memoria è costituita da un numero infinito di celle di dimensione finita a cui si accede, indipendentemente dalla posizione, in tempo costante.

Processore (singolo)

Un insieme di istruzioni simili a quelle reali: algebriche, logiche e di salto.

Costo delle istruzioni elementari

Uniforme e ininfluente ai fini dell'analisi.

2.2 Funzioni di costo, analisi asintotica

Per studiare la complessità di un algoritmo si analizza il suo comportamento asintotico, ovvero quando la dimensione del suo input tende a infinito.

2.2.1 Notazione O

Sia $g(n)$ una funzione di costo, si indica con $O(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geq 0 : f(n) \leq cg(n), \forall n \geq m$$

$g(n)$ si dice limite asintotico superiore di $f(n)$, ovvero $f(n)$ cresce al più come $g(n)$.

2.2.2 Notazione Ω

Sia $g(n)$ una funzione di costo, si indica con $\Omega(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geq 0 : f(n) \geq cg(n), \forall n \geq m$$

$g(n)$ si dice limite asintotico inferiore di $f(n)$, ovvero $f(n)$ cresce almeno quanto $g(n)$.

2.2.3 Notazione Θ

Sia $g(n)$ una funzione di costo, si indica con $\Theta(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c_1 > 0, \exists c_2 > 0, \exists m \geq 0 : c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq m$$

$f(n)$ cresce esattamente come $g(n)$ e $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

2.2.4 Proprietà della notazione asintotica

Espressioni polinomiali

Enunciato

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0, a_k > 0 \Rightarrow f(n) = \Theta(n^k)$$

Limite superiore

$$\exists c > 0, \exists m \geq 0 : f(n) \leq cn^k, \forall n \geq m$$

Dimostrazione

$$\begin{aligned} f(n) &= a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \\ &\leq a_k n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n + |a_0| \\ &\leq a_k n^k + |a_{k-1}| n^k + \cdots + |a_1| n^k + |a_0| n^k \qquad \forall n \geq 1 \\ &= (a_k + |a_{k-1}| + \cdots + |a_1| + |a_0|) n^k \\ &\overset{?}{\leq} cn^k \end{aligned}$$

Vera per $c \geq (a_k + |a_{k-1}| + \cdots + |a_1| + |a_0|)$ e per $m = 1$.

Limite inferiore

$$\exists d > 0, \exists m \geq 0 : f(n) \geq dn^k, \forall n \geq m$$

Dimostrazione

$$\begin{aligned} f(n) &= a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \\ &\geq a_k n^k - |a_{k-1}| n^{k-1} - \cdots - |a_1| n - |a_0| \\ &\geq a_k n^k - |a_{k-1}| n^{k-1} - \cdots - |a_1| n^{k-1} - |a_0| n^{k-1} \qquad \forall n \geq 1 \\ &= (a_k + |a_{k-1}| + \cdots + |a_1| + |a_0|) n^k \\ &\overset{?}{\geq} dn^k \end{aligned}$$

Vera per $d \leq a_k - \frac{|a_{k-1}|}{n} - \frac{|a_{k-2}|}{n} - \cdots - \frac{|a_1|}{n} - \frac{|a_0|}{n} > 0 \Leftrightarrow n > \frac{|a_{k-1}+\cdots+|a_0|}{a_k}$.

Dualità

Enunciato

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

Dimostrazione

$$\begin{aligned} f(n) = O(g(n)) &\Leftrightarrow f(n) \leq cg(n), \forall n \geq m \\ &\Leftrightarrow g(n) \geq \frac{1}{c} f(n), \forall n \geq m \\ &\Leftrightarrow g(n) \geq c' = f(n), \forall n \geq m, c' = \frac{1}{c} \\ &\Leftrightarrow g(n) = \Omega(f(n)) \end{aligned}$$

Eliminazione delle costanti

Enunciato

$$f(n) = O(g(n)) \Leftrightarrow af(n) = O(g(n)), \forall a > 0 \tag{2.1}$$

$$f(n) = \Omega(g(n)) \Leftrightarrow af(n) = \Omega(g(n)), \forall a > 0 \tag{2.2}$$

Dimostrazione

$$\begin{aligned} f(n) = O(g(n)) &\Leftrightarrow f(n) \leq cg(n), \forall n \geq m \\ &\Leftrightarrow af(n) \leq acg(n), \forall n \geq m, \forall a \geq 0 \\ &\Leftrightarrow af(n) \leq c'g(n), \forall n \geq m, c' = ac > 0 \\ &\Leftrightarrow af(n) = O(g(n)) \end{aligned}$$

La dimostrazione è analoga per il limite inferiore.

Sommatoria

Enunciato

$$f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \Rightarrow f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$$

$$f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \Rightarrow f_1(n) + f_2(n) = \Omega(\min(g_1(n), g_2(n)))$$

Dimostrazione

$$\begin{aligned} f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) &\Rightarrow \\ f_1(n) \leq c_1 g_1(n) \wedge f_2(n) \leq c_2 g_2(n) &\Rightarrow \\ f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) &\Rightarrow \\ f_1(n) + f_2(n) \leq \max(c_1, c_2)(2\max(g_1(n), g_2(n))) &\Rightarrow \\ f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)) & \end{aligned}$$

La dimostrazione è analoga per il limite inferiore.

Prodotto

$$f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

$$f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdots g_2(n))$$

Dimostrazione

$$\begin{aligned} f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) &\Rightarrow \\ f_1(n) \leq c_1 g_1(n) \wedge f_2(n) \leq c_2 g_2(n) &\Rightarrow \\ f_1(n) \cdot f_2(n) \leq c_1 c_2 g_1(n) g_2(n) &\Rightarrow \\ f_1(n) f_2(n) = O(g_1(n) g_2(n)) & \end{aligned}$$

Simmetria

Enunciato

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

Dimostrazione Grazie alla proprietà di dualità:

$$\begin{array}{lll} f(n) = \Theta(g(n)) \Rightarrow & f(n) = O(g(n)) \Rightarrow & g(n) = \Omega(f(n)) \\ f(n) = \Theta(g(n)) \Rightarrow & f(n) = \Omega(g(n)) \Rightarrow & g(n) = O(f(n)) \end{array}$$

Transitività

Enunciato

$$f(n) = O(g(n)), g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

Dimostrazione

$$\begin{aligned} f(n) = O(g(n)) \wedge g(n) = O(h(n)) &\Rightarrow \\ f(n) \leq c_1 g(n) \wedge g(n) \leq c_2 h(n) &\Rightarrow \\ f(n) \leq c_1 c_2 h(n) &\Rightarrow \\ f(n) = O(h(n)) & \end{aligned}$$

Logaritmi

Enunciato Si vuole provare che $\log n = O(n)$. Si dimostri pertanto per induzione che:

$$\exists c > 0, \exists m \geq 0 : \log n \leq cn, \forall n \geq m$$

Dimostrazione

- Caso base, $n = 1$: $\log 1 = 0 \leq cn = c\dot{1} \Leftrightarrow c \geq 0$.
- Ipotesi induttiva: sia $\log k \leq ck, \forall k \leq n$.
- Passo induttivo: si dimostri la proprietà per $n + 1$.

$$\begin{aligned} \log(n+1) &\leq \log(n+n) = \log 2n && \forall n \geq 1 \\ &= \log 2 + \log n && \log ab = \log a + \log b \\ &= 1 + \log n && \log 2 = 1 \\ &\leq 1 + cn && \text{per induzione} \\ &\overset{?}{\leq} c(n+1) && \text{obiettivo} \end{aligned}$$

$$1 + cn \leq c(n+1) \Leftrightarrow c \geq 1$$

2.2.5 Notazioni o, ω

Notazione o

Sia $g(n)$ una funzione di costo, si indica con $o(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\forall c, \exists m : f(n) < cg(n), \forall n \geq m$$

Notazione ω

Sia $g(n)$ una funzione di costo, si indica con $\omega(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\forall c, \exists m : f(n) > cg(n), \forall n \geq m$$

Significato

Utilizzando il concetto di limite si noti come:

- $\lim_{x\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$.
- $\lim_{x\to\infty} \frac{f(n)}{g(n)} \neq 0 \Rightarrow f(n) = \Theta(g(n))$.
- $\lim_{x\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n))$.
- $f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$.
- $f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$.

2.2.6 Classificazione delle funzioni

Espandendo le relazioni dimostrate è possibile ottenere un ordinamento delle principali espressioni. Si consideri per ogni $r < s,\ h < k,\ a < b$:

$$O(1) \subset O(\log^r n) \subset O(\log^s n) \subset O(n^h) \subset O(n^h \log^r n)$$
$$\subset O(n^h \log^s n) \subset O(n^k) \subset O(a^n) \subset O(b^n)$$

2.3 Ricorrenze

Definizioni

- Equazione di ricorrenza: calcolare la complessità di un algoritmo ricorsivo richiede le creazione di un'equazione di ricorrenza, una formula matematica definita in maniera ricorsiva.
- Forma chiusa: l'obiettivo è partire dall'equazione di ricorrenza e trasformarla in una forma chiusa in modo da comprendere la classe di complessità dell'algoritmo.

2.3.1 Analisi per livelli

Questo metodo di risoluzione delle equazioni ricorsive detto anche metodo dell'albero di ricorsione consiste nell'espandere la ricorrenza in un albero i cui nodi rappresentano i costi ai vari livelli della ricorsione considerando poi il livello più basso in cui tutte le chiamate sono state ricondotte al caso base. Si otterrà pertanto una sommatoria da cui si potrà derivare la forma chiusa e la classe di complessità.

2.3.2 Metodo della sostituzione

Questo metodo di risoluzione delle equazioni ricorsive, detto anche metodo per tentativi, consiste nel cercare di indovinare una soluzione in base alla propria esperienza e di tentare di dimostrare tale soluzione per induzione.

2.3.3 Metodo delle ricorrenze comuni

Questo metodo di risoluzione delle equazioni ricorsive, detto anche metodo dell'esperto, mette a disposizione dei teoremi che permettono di risolvere facilmente ampie classi di equazioni di ricorrenza.

Ricorrenze lineari con partizione bilanciata

Teorema Siano a e b costanti intere tali che $a \geq 1$ e $b \geq 2$ e c e β costanti reali tali che $c > 0$ e $\beta \geq 0$. Sia $T(n)$ l'equazione di ricorrenza nella forma:

$$T(n) = \begin{cases} aT(\frac{n}{b}) + cn^\beta & n > 1 \\ d & n \leq 1 \end{cases}$$

Posto $\alpha = \frac{\log a}{\log b} = \log_b a$ allora:

$$T(n) = \begin{cases} \Theta(n^\alpha) & \alpha > \beta \\ \Theta(n^\alpha \log n) & \alpha = \beta \\ \Theta(n^\beta) & \alpha < \beta \end{cases}$$

Estensione delle ricorrenze lineari con partizione bilanciata

Sia $a \geq 1$, $b > 1$ e $f(n)$ asintoticamente positiva e sia

$$T(n) = \begin{cases} aT(\frac{n}{b}) + f(n) & n > 1 \\ d & n \leq 1 \end{cases}$$

Sono dati tre casi.

- $\exists \varepsilon > 0 : f(n) = O(n^{\log_b a - \varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$.
- $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(f(n) \log n)$.
- $\exists \varepsilon > 0 : f(n) = \Omega(n^{\log_b a + \varepsilon}) \wedge \exists c : 0 < c < 1, \exists m > 0 : af(\frac{n}{b}) \leq cf(n), \forall n \geq m \Rightarrow T(n) = \Theta(f(n))$.

Ricorrenze lineari di ordine costante

Siano $a_1, \ldots, a_n$ costanti intere non negative, con h costante positiva, $c > 0$ e $\beta \geq 0$ costanti reali e $T(n)$ una relazione di ricorrenza nella forma:

$$T(n) = \begin{cases} \sum\limits_{1 \leq i \leq h} a_i T(n-i) + cn^\beta & n > m \\ \Theta(1) & n \leq m \leq h \end{cases}$$

Posto $a = \sum\limits_{1 \leq i \leq h} a_i$ allora:

- $T(n) = \Theta(n^{\beta+1})$ se $a = 1$.
- $T(n) = \Theta(a^n n^\beta)$ se $a \geq 2$.

2.4 Relazione tra complessità di un problema e di un algoritmo

Un problema ha complessità $O(f(n))$ se esiste un algoritmo che lo risolve con complessità $O(f(n))$. Un problema ha complessità $\Omega(f(n))$ se tutti gli algoritmi che lo risolvono hanno complessità $\Omega(f(n))$.

2.4.1 Complessità in tempo di un algoritmo

La quantità di tempo richiesta per input di dimensione n:

- $O(f(n))$: per tutti gli input l'algoritmo costa al più $f(n)$.
- $\Omega(f(n))$: per tutti gli input l'algoritmo costa almeno $f(n)$.
- $\Theta(f(n))$: per tutti gli input l'algoritmo richiede $f(n)$.

2.4.2 Complessità in tempo di un problema computazionale

La quantità di tempo richiesta per input di dimensione n:

- $O(f(n))$: complessità del miglior algoritmo che risolve il problema.
- $\Omega(f(n))$: dimostrare che nessun algoritmo può risolvere il problema in un tempo inferiore a $\Omega(f(n))$.
- $\Theta(f(n))$: algoritmo ottimo.

2.5 Valutare algoritmi in base alla tipologia di input

In alcuni casi gli algoritmi si comportano in maniera diversa in base a caratteristiche dell'input. Conoscerle in anticipo permette di scegliere il miglior algoritmo per la situazione.

2.5.1 Tipologie di analisi

- Analisi del caso pessimo: il tempo di esecuzione è il limite superiore al tempo di esecuzione per un qualisasi input.
- Analisi del caso medio: molto difficile in quanto si deve trovare una distribuzione uniforme degli input.
- Analisi del caso ottimo: ha senso se si conoscono caratteristiche dell'input.

2.5.2 Algoritmi di ordinamento

Il problema di ordinamento è rappresentato da una sequenza $A = a_1, \ldots, a_n$ di valori in input e dà in output una sequenza $B = b_1, \ldots, b_n$ permutazione di B tale per cui $\forall 0 < i < n-1, b_i \leq b_{i+1}$.

Selection sort

Cerco il minimo e lo metto nella posizione corretta, riducendo il problema ai restanti $n-1$ valori.

```
: selectionSort(item [] A, int n)
for i = 1 to n-1 do
    int min = min(A, i, n)
    A[i] ↔ A[min]
```

```
: min(item [] A, int i, int n)
%Posizione del minimo parziale
int min = i
for j = i + 1 to n do
    if A[j] ≤ A[min] then
        %Nuovo minimo parziale
        min = j
    return min
```

Complessità

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = n^2 - \frac{n}{2} = O(n^2)$$

InsertionSort

Algoritmo efficiente per ordinare piccoli insiemi. In cui si prende l'i-esimo elemento e lo si mette nella posizione corretta rispetto agli elementi precedenti, proseguendo fino alla fine.

```
: insertionSort(item [] A, int n)
for i = 2 to n do
    item temp = A[i]
    int j = i
    while j > 1 and A[j-1] > temp do
        A[j] = A[j - 1]
        j -= 1
    A[j] = temp
```

Correttezza e complessità DA COMPLETARE

MergeSort

Il mergeSort si basa sulla tecnica di divide-et-impera in quanto divide il vettore di n elementi in due sottovettori di $\frac{n}{2}$ elementi, chiama il mergeSort ricorsivamente su quei due elementi e unisce (merge) le due sequenze ordinate. Si sfrutta il fatto che i due sottovettori sono già ordinati per ordinare più velocemente.

```
: merge(item [] A, int first, int last, int mid)
int i, j, k , h
i = first
j = mid + 1
k = first
while i ≤ mid and j ≤ last do
    if A[i] ≤ A[j] then
        B[k] = A[i]
        i += 1
    else
        B[k] = A[j]
        j += 1
    k += 1
j = last
for h = mid down to i do
    A[j] = A[h]
    j -= 1
for j = first to k-1 do
    A[j] = B[j]
```

merge()

Costo computazionale $O(n)$

mergeSort()

```
: mergeSort(item [] A, int first, int last)
if first < last then
    int mid = ⌊(first + last) / 2⌋
    mergeSort(A, first, mid)
    mergeSort(A, mid + 1, last)
    merge(A, first, last, mid)
```

Costo computazionale Si assuma per semplificare che $n = 2^k$ in modo che $k = \log n$ e tutti i sottovettori hanno dimensioni di potenze esatte di due. Si ottiene così l'equazione di ricorrenza:

$$T(n) = \begin{cases} c & n = 1 \\ 2T(\frac{n}{2}) + dn & n > 1 \end{cases}$$

Si noti come ad ogni chiamata ricorsiva si svolga un'operazione di merge di costo $O(1)$ e k vari tra 0 e $\log n$. Si ottiene pertanto $O(\sum_{i=0}^{k} s^i \frac{n}{2^i}) = O(\sum_{i=0}^{k} n) = O(kn) \Leftrightarrow O(n \log n)$.

2.6 Analisi ammortizzata

Si intende per analisi ammortizzata una tecnica di analisi di complessità che valuta il tempo per eseguire nel caso pessimo una sequenza di operazioni su una struttura dati. Esistono operazioni più o meno costose e se le operazioni costose sono meno frequenti allora il loro costo può essere ammortizzato da quelle meno costose. A differenza dell'analisi del caso medio è deterministica su operazioni multiple e verifica il caso pessimo.

2.6.1 Metodi per l'analisi ammortizzata

Metodo dell'aggregazione

In questo metodo si calcola la complessità $T(n)$ per eseguire n operazioni in sequenza nel caso pessimo. Grazie alla sequenza si considera l'evoluzione della struttura dati data una sequenza di operazioni, considerando la sequenza pessima e sommando insieme tutte le complessità individuali. Successivamente questo $T(n)$ viene diviso per il numero di operazioni in modo da verificare la complessità ammortizzata di un'operazione nella sequenza.

Metodo degli accantonamenti

Alle operazioni vengono assegnati costi ammortizzati che possono essere minori o maggiori del loro costo effettivo. Questa differenza è dovuta al fatto che le operazioni meno costose vengono caricate di un costo aggiuntivo detto credito: $costo\ ammortizzato = costo\ effettivo + credito\ prodotto$. Questo credito accumulato viene speso dalle operazioni più costose: $costo\ ammortizzato = costo\ effettivo - credito\ consumato$. Si deve dimostrare che la somma dei costi ammortizzato a_i è un limite superiore alla somma dei costi effettivi: $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i$ e che il valore così ottenuto è "poco costoso". Considerando il caso pessimo, la dimostrazione deve valere per tutte le sequenze e il credito dopo la t-esima operazione deve essere sempre positivo: $\sum_{i=1}^{t} a_i \leq \sum_{i=1}^{t} c_i \geq 0$.

Metodo del potenziale

Lo stato del sistema viene descritto attraverso una funzione di potenziale $\Phi(D)$. Le operazioni meno costose devono incrementare $\Phi(D)$ e quelle più costose decrementarlo. Il costo ammortizzato è pari al costo effettivo sommato alla differenza di potenziale: $a_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$. Per una sequenza di operazioni di lunghezza n, se $\Phi(D_n) - \Phi(D_0) \geq 0$ il costo ammortizzato A è un limite superiore al costo reale.

Capitolo 3

Strutture dati

Si intende per dato in un linguaggio di programmazione un valore che una variabile può assumere. Un tipo di dato astratto è una collezione di valori e un insieme di operazioni ammesse su quei valori. Si dicono primitivi i tipi di dati forniti direttamente dal linguaggio di programmazione. La definizione di un tipo di dato astratto si divide nella specifica, una descrizione di alto livello di come può essere utilizzata e nell'implementazione, la realizzazione vera e propria di basso livello. Le strutture dati sono collezioni di dati, caratterizzate dall'organizzazione della collezione stessa più che dal tipo di dati contenuti. Le strutture dati sono caratterizzate da un insieme di operatori che permettono di manipolarne la struttura e un modo sistematico per organizzare l'insieme di dati. Le strutture dati si dividono in:

- Lineari o non lineari rispetto alla presenza o meno di una sequenza.
- Statiche o dinamiche se la dimensione della struttura può variare o meno.
- Omogenee o disomogenee se possono contenere uno o diversi tipi di dati.

3.1 Sequenza

Una sequenza è una struttura dati dinamica, lineare che rappresenta una sequenza ordinata di valori, dove un valore può comparire più di una volta. È importante l'ordine all'interno della sequenza.

Operazioni ammesse

- Data una posizione è possibile aggiungere o eliminare un elemento in quella posizione. Si considerano anche per comodità le posizioni pos_0 e pos_{n+1}.
- È possibile accedere direttamente alla testa e alla coda della sequenza.
- È possibile accedere sequenzialmente a tutti gli altri elementi.

: Sequence

%Restituisce True se la sequenza è vuota
boolean `isEmpty()`

%Restituisce True se p è uguale a pos_0 o a pos_{n+1}
boolean `finished`(Pos p)

%Restituisce la posizione del primo elemento
Pos `head()`

%Restituisce la posizione dell'ultimo elemento
Pos `tail()`

%Restituisce la posizione dell'elemento che segue p
Pos `next`(Pos p)

%Restituisce la posizione dell'elemento che precede$p$
Pos `prev`(Pos p)

%Inserisce l'elemento v di tipo Item nella posizione p
%Restituisce la posizione del nuovo elemento che diventa predecessore di p
Pos `insert`(Pos p, Item v)

%Rimuove l'elemento alla posizione p
%Restituisce la posizione del successore di p
%Che diventa il successore del predecessore di p
Pos `remove`(Pos p)

%Legge l'elemento di tipo Item contenuto nella posizione p
Item `read`(Pos p)

%Scrive l'elemento v di tipo Item nella posizione p
`write`(Pos p, Item v)

Specifica

3.2 Insiemi

Per insieme si intende una struttura dati dinamica, non lineare che memorizza una collezione non ordinata di elementi senza valori ripetuti. L'ordinamento tra i valori sarà dato dall'eventuale relazione d'ordine definita sul tipo di elementi stessi.

Operazioni ammesse

- Operazioni base:
  - Inserimento.
  - Cancellazione.
  - Verifica contenimento.
- Operazioni di ordinamento:
  - Massimo.
  - Minimo.
- Operazioni insiemistiche:
  - Unione.
  - Intersezione.
  - Differenza.
- Iteratori.
  - **foreach** $x \in S$ **do**.

Specifica

3.3 Dizionari

Un dizionario è una struttura dati che rappresenta il concetto matematico di relazione univoca $R : D \to C$ o associazione chiave-valore. L'insieme D è il dominio ed è costituito dalle chiavi, l'insieme C è il codominio ed è costituito dai valori.

Operazioni ammesse

- Ottenere il valore associato ad una particolare chiave se presente altrimenti **nil**.
- Inserire una nuova associazione chiave-valore cancellando eventualmente associazioni precedenti per la stessa chiave.
- Rimuovere un'associazione chiave-valore esistente.

: Set

```
%Restituisce la cardinalità dell'insieme
int size()
%Restituisce True se x è contenuto nell'insieme
boolean contains(Item x)
%Inserisce x nell'insieme se non è già presente
insert(Item x)
%Rimuove x dall'insieme se è presente
remove(Item x)
%Restituisce un nuovo insieme che è l'unione di A e B
Set union(Set A, Set B)
%Restituisce un nuovo insieme che è l'intersezione di A e B
Set intersection(Set A, Set B)
%Restituisce un nuovo insieme che è la differenza tra A e B
Set Difference(Set A, Set B)
```

: Dictionary

```
%Restituisce il valore associato alla chiave k se presente, nil altrimenti
Item lookup(Item k)
%Associa il valore v alla chiave k
insert(Item k, Item v)
%Rimuove l'associazione della chiave k
remove(Item k)
```

Specifica

3.4 Alberi e grafi

3.4.1 Alberi ordinati

Un albero ordinato è dato da un insieme finito di elementi detti nodi, uno dei quali è designato come radice e i rimanenti, se esistono sono partizionati in insiemi ordinati e disgiunti, anch'essi alberi ordinati.

3.4.2 Grafi

La struttura di un grafo è composta da un insieme di elementi detti nodi o vertici e un insieme di coppie (ordinate o no) di nodi detti archi.

3.4.3 Operazioni

Tutte le operazioni su grafi e alberi ruotano intorno alla possibilità di effettuare visite su di essi.

3.5 Lista

Una lista o linked list è una struttura dati contenente una sequenza di nodi contenente dati arbitrari, $1-2$ puntatori all'elemento successivo e/o a quello precedente. La contiguità nella lista è diversa dalla contiguità in memoria e tutte le operazioni su una lista hanno costo $O(1)$.

Possibili implementazioni

Bidirezionale o monodirezionale: in base al numero di puntatori e se puntano a quello successivo e/o a quello precedente. Con sentinella o senza sentinella: in base alla presenza di una sentinella, una struttura che punta sempre al primo elemento della lista. Circolare o non circolare: si dice circolare se il puntatore dell'ultimo elemento punta al primo invece di essere **nil**.

Lista bidirezionale con sentinella

3.6 Pila

Una pila o stack è una struttura dati lineare e dinamica in cui l'elemento rimosso dall'operazione di cancellazione è determinato, ovvero quello che per meno tempo è rimasto nell'insieme (LIFO: last-in, first-out). Possibili implementazioni sono una lista bidirezionale con un puntatore all'elemento top o tramite vettore, con dimensione limitata ma overhead più basso.

Specifica

Lista basata su vettore

3.7 Coda

Una coda è una struttura dati lineare e dinamica in cui l'elemento rimosso dall'operazione di cancellazione è determinato, ovvero quello che per più tempo è rimasto nell'insieme (FIFO: first-in,

: List

```
List pred %Predecessore
List succ %Successore
Item value %Valore
List List()
    List t=new List
    t.pred = t
    t.succ = t
    return t
boolean isEmpty()
    return pred = succ = this
Pos head()
    return succ
Pos tail()
    return pred
Pos next(Pos p)
    return p.succ
Pos prev(Pos p)
    return p.pred
boolean finished(Pos p)
    return (p = this)
Item read(Pos p)
    return p.value
Item write(Pos p, Item v)
    p.value = v
Pos insert(Pos p, Item v)
    List t = List()
    t.value = v
    t.pred = p.pred
    p.pred.succ = t
    t.succ = p
    p.pred = t
    return t
Pos remove(Pos p)
    p.pred.succ = p.succ
    p.succ.pred = p.pred
    List t = p.succ
    delete p
    return t
```

: Stack

```
%Restituisce True se la pila è vuota
boolean isEmpty()
%Inserisce v in cima alla pila
push(Item v)
%Estrae l'elemento in cima alla pila e lo restituisce
Item pop()
%Legge l'elemento in cima alla pila
Item top()
```

```
: Stack
Item [] A %Elementi
int n %Cursore
int m %Dimensione massima
Stack Stack(int dim)
    Stack t = new Stack
    t.A = new int [1...dim]
    t.m = dim
    t.n = 0
    return t
Item top()
    precondition: n > 0
    return A[n]
boolean isEmpty()
    return n=0
Item pop()
    precondition: n > 0
    Item t = A[n]
    n = n - 1
    return t
push(Item v)
    precondition: n < m
    n = n + 1
    A[n] = v
```

first-out, questo tipo di politica si dice fair). Si può implementare attraverso liste monodirezionali con il puntatore head per l'estrazione e il puntatore tail per l'inserimento o tramite array circolari, con dimensione limitata ma overhead più basso. Per i secondi la circolarità è ottenuta attraverso l'operazione modulo e si deve prestare attenzione ai problemi di overflow.

Specifica

```
: Queue
%Restituisce True se la coda è vuota
boolean isEmpty()
%Inserisce v in fondo alla coda
enqueue(Item v)
%Estrae l'elemento in testa alla coda e lo restituisce
Item dequeue()
%Legge l'elemento in testa alla coda
Item top()
```

Coda basata su vettore circolare

: Queue

```
Item [] A %Elementi
int n %Dimensione attuale
int testa %Testa
int m %Dimensione massima
Queue Queue(int dim)
    Queue t = new Queue
    t.A = new int [1 . . . dim - 1]
    t.m = dim
    t.testa = 0
    t.n = 0
    return t
Item top()
    precondition: n > 0
    return A[testa]
boolean isEmpty()
    return n=0
Item dequeue()
    precondition: n > 0
    Item t = A[testa]
    testa = (testa + 1)  mod  m
    n = n - 1
    return t
enqueue(Item v)
    precondition: n < m
    A[(testa + n)  mod  m] = v
    n = n + 1
```

Capitolo 4

Alberi

4.1 Alberi radicati

Un albero radicato può essere definito in due modi.

4.1.1 Definizioni

Definizione chiusa

Un albero radicato consiste in una serie di nodi e un insieme di archi orientati che connettono coppie di nodi con le seguenti proprietà:

- Un nodo dell'albero è designato come radice.
- Ogni nodo n, a parte la radice ha esattamente un arco entrante.
- Esiste un cammino unico dalla radice ad ogni nodo.
- L'albero è connesso.

Definizione ricorsiva

Un albero radicato è dato da:

- Un insieme vuoto.
- Un nodo radice e zero o più sottoalberi ognuno dei quali è un albero, la radice è connessa alla radice di ogni sottoalbero con un arco orientato.

4.1.2 Terminologia

- Il nodo senza archi entranti è detto radice (root).
- Per generare i sottoalberi (subtrees) da un albero si elimina la radice e tutti i suoi archi uscenti.
- Nodi con lo stesso genitore sono detti fratelli (siblings).
- Considerando un arco, il nodo da cui parte è detto genitore (parent) del nodo in cui arriva, detto figlio (child).

- I nodi senza archi uscenti sono detti foglie (leaves).
- I nodi nè foglie nè radice sono detti interni (internal nodes).
- La lunghezza del cammino semplice dalla radice ad un nodo, misurato nel numero di archi è detto profondità (depth) del nodo.
- I nodi alla stessa profondità formano un insieme chiamato livello (level).
- La profondità massima dell'albero si dice altezza (height).

4.2 Visite di alberi

La visita di un albero o ricerca è una strategia per visitare tutti i nodi di un albero. Il costo computazionale di una visita su un albero su n nodi è $\Theta(n)$ in quanto ogni nodo viene visitato un'unica volta.

4.2.1 Visita in profondità

La visita in profondità o depth-first-search (dfs) si compie visitando ricorsivamente tutti i sottoalberi dell'albero, richiede uno stack ed esiste in tre varianti: pre, in e post order.

4.2.2 Visita in ampiezza

La visita in ampiezza o breadth-fisrt-search (bfs) visita completamente ogni livello prima di passare al successivo partendo dalla radice. Richiede una queue.

4.3 Albero binario

Un albero binario è un albero radicato in cui ogni nodo ha al massimo due figli, indicati con figlio destro e sinistro. Due alberi binari differiscono anche se uno stesso nodo è designato come figlio sinistro invece che destro o viceversa.

4.3.1 Specifica

: Tree

```
%Costruisce un nuovo nodo contenente v senza figli o genitori
Tree(Item v)
%Legge il valore memorizzato nel nodo
Item read()
%Modifica il valore memorizzato nel nodo
write(Item v)
%Restituisce il padre o nil se è il nodo radice
Tree Parent()
%Restituisce il figlio sinistro (destro) di questo nodo o nil se assente
Tree left()
Tree right()
%Inserisce il sottoalbero radicato in t come figlio sinistro (destro) di questo nodo
insertLeft(Tree t)
insertRight(Tree t)
%Distrugge ricorsivamente il figlio sinistro (destro) di questo nodo
deleteLeft()
deleteRight()
```

4.3.2 Memorizzazione e implementazione

Ogni nodo deve memorizzare oltre al proprio valore la reference al nodo padre (parent), la reference ai figli sinistro (left) e destro (right).

Implementazione

: Binary tree

```
Tree(Item v)
    Tree t = new Tree
    t.parent = nil
    t.left = t.right = nil
    t.value = v
    return t
insertLeft(Tree T)
    if left == nil then
        T.parent = this
        left = T
insertRight(Tree T)
    if right == nil then
        T.parent = this
        right = T
deleteLeft()
    if left ≠ nil then
        left.deleteLeft()
        left.deleteRight()
        left = nil
deleteRight()
    if right ≠ nil then
        right.deleteLeft()
        right.deleteRight()
        right = nil
```

4.3.3 Visita in profondità

```
: dfs(Tree t)
  if t ≠ nil then
      %visita in pre-ordine
      print t
      dfs(t.left)
      %visita in in-ordine
      print t
      dfs(t.right)
      %visita in post-ordine
      print t
```

4.4 Alberi generici

4.4.1 Specifica

```
: Tree
  %Costruisce un nuovo nodo contenente v senza figli o genitori
  Tree(Item v)
  %Legge il valore memorizzato nel nodo
  Item read()
  %Modifica il valore memorizzato nel nodo
  write(Item v)
  %Restituisce il padre o nil se è il nodo radice
  Tree Parent()
  %Restituisce il primo figlio o nil se è un nodo foglia
  Tree leftmostchild()
  %Restituisce il prossimo fratello o nil se assente
  Tree rightSibling()
  %Inserisce il sottoalbero t come primo figlio di questo nodo
  insertChild(Tree t)
  %Inserisce il sottoalbero t come prossimo fratello di questo nodo
  insertSibling(Tree t)
  %Distrugge l'albero radicato nel primo figlio
  deleteChild()
  %Distrugge l'albero radicato nel prossimo fratello
  deleteSibling()
```

Depth-first search

```
: dfs(Tree t)
if t ≠ nil then
    %visita in pre-ordine
    print t
    Tree u = t.leftMostChild()
    while u ≠ nil do
        dfs(u)
        u = u.rightSibling()
    %visita in post-ordine
    print t
```

Breadth-first search

```
: bfs (Tree t)
Queue Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    Tree u = Q.dequeue()
    %Visita per livelli nodo u
    print u
    u = u.leftMostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

4.4.2 Memorizzazione

Esistono vari modi per salvare un albero, scelti in base al numero massimo e medio dei figli presenti.

Vettore dei figli

Nel nodo viene memorizzato il genitore e un vettore contenente i figli, che a seconda del loro numero può portare a uno spreco dello spazio.

Primo figlio, prossimo fratello

```
: Tree
```

```
Tree parent %Reference al padre
Tree child %Reference al primo figlio
Tree sibling %Reference al prossimo fratello
Item value %Valore memorizzato nel nodo
Tree(Item v)
    Tree t = new Tree
    t.value = v
    t.parent = t.child = t.sibling = nil
    return t
insertChild(Tree t)
    t.parent = self
    t.sibling = child
    child = t
insertSibling(Tree t)
    t.parent = parent
    t.sibling = sibling
    sibling = t
deleteChild()
    Tree newChild = child.rightSibling()
     delete(child)
    child = newChild
deleteSibling()
    Tree newBrother = sibling.rightSibling()
    delete(sibling)
    sibling = newBrother
delete(Tree t)
    Tree u = t.leftMostChild()
    while u ≠ nil do
        Tree next = u.rightSibling()
        delete(u)
        u = next
```

Vettore dei padri

L'albero è rappresentato da un vettore i cui elementi contengono il valore associato al nodo e l'indice della posizione del padre nel vettore.

Capitolo 5

Alberi di ricerca

Si vuole portare la ricerca binaria negli alberi in modo da implementare un dizionario.

- Le associazioni chiave-valore vengono memorizzate in un albero binario.
- Ogni nodo u contiene la coppia $u.key$ e $u.value$.
- Le chiavi devono appartenere ad un insieme totalmente ordinato.

Le chiavi contenute nel sottoalbero sinistro di u sono minori di $u.key$ e quelle contenute nel sottoalbero destro maggiori. In questo modo è possibile realizzare un algoritmo di ricerca dicotomica.

5.1 Specifica

: Search Binary Tree	
Tree parent	`insert`(Item k, Item v)
Tree left	`remove`(Item k)
Tree right	%Ordinamento
Item value	Tree `successorNode`(Tree t)
Item key	Tree `predecessorNode`(Tree t)
%Getters	Tree `min`()
Item `key`()	Tree `max`()
Item `value`()	%Funzioni interne
Tree `parent`()	**private**: Tree `lookupNode`(Tree t, Item k)
Tree `right`()	**private**: Tree `insertNode`(Tree t, Item k, Item v)
Tree `left`()	**private**: Tree `removeNode`(Tree t, Item k)
%Dizionario	
Item `lookup`(Item k)	

```
: Dictionary
Tree tree
Dictionary()
    tree = nil
```

5.1.1 Ricerca

La funzione *lookUpNode(Tree t, Item k)* restituisce il nodo dell'albero t che contiene la chiave k o **nil** se non presente.

Implementazione nel dizionario

```
: Item lookup(Item k)
Tree t = lookupNode(tree, k)
if t ≠ nil then
    return t.value
else
    return nil
```

Implementazione

Iterativa

```
: Tree lookupNode (Tree t, Item k)
Tree u = t
while u ≠ nil and u.key ≠ k do
    if k < u.key then
        u = u.left
    else
        u = u.right
return u
```

Ricorsiva

```
: Tree lookupNode (Tree t, Item k)
if t == nil or t.key == k then
    return t
else if k < t.key then
    return lookupNode(t.left, k)
else
    return lookupNode(t.right, k)
```

5.1.2 Minimo e massimo

Implementazione

```
: Tree min (Tree t)
Tree u = t
while u.left() ≠ nil do
    u = u.left
return u
```

```
: Tree max (Tree t)
Tree u = t
while u.right ≠ nil do
    u = u.right
return u
```

5.1.3 Successore e predecessore

Si definisce successore di un nodo u il più piccolo nodo maggiore di u.

Implementazione

```
: successorNode(Tree t)
if t = nil then
    return t
if t.left ≠ nil then
    return max(t.left)
else
    Tree p = t.parent()
    while p ≠ nil and t = p.left do
        t = p
        p = p.parent
    return p
```

```
: predecessorNode(Tree t)
if t = nil then
    return t
if t.right() ≠ nil then
    return min(t.right)
else
    Tree p = t.Parent
    while p ≠ nil and t = p.right do
        t = p
        p = p.Parent
    return p
```

5.1.4 Inserimento

L'operazione *insertNode(Tree t, Item k, Item v)* inserisce un'associazione chiave-valore *(k, v)* nell'albero *t*. Se la chiave è già presente sostiutuisce il valore associato, altrimenti viene inserita una nuova associazione. Se $T =$ ***nil*** restituisce il primo nodo dell'albero, altrimenti restituisce t inalterato.

Implementazione dizionario

```
: insert(Item k, Item v)
tree = insertNode(tree, k, v)
```

Implementazione

```
: Tree insertNode(Tree t, Item k, Item v)
Tree p = nil
Tree u = t
while u ≠ nil and u.key ≠ k do
    p = u
    if k < u.key then
        u = u.left
    else
        u = u.right
if u ≠ nil and u.key == k then
    u.value = v
else
    Tree new = Tree(k, v)
    link(p, new, k)
    if p = nil then
        t = new
    return t
```

```
: link(Tree p, Tree u, Item k)
if u ≠ nil then
    u.parent = p
if p ≠ nil then
    if k < p.key then
        p.left = u
    else
        p.right = u
```

5.1.5 Cancellazione

L'operazione *Tree removeNode(Tree t, Item k)* rimuove il nodo contenente la chiave k dall'albero t e restituisce la radice dell'albero possibilmente cambiata.

Implementazione dizionario

```
: remove(Item k)
tree = removeNode(tree, k)
```

Implementazione

```
: Tree removeNode(Tree T, Item k)
Tree t
Tree u = lookupNode(T, k)
if u ≠ nil then
    if u.left == nil and u.right == nil then
        link(u.parent, nil, k)
        delete u
    else if u.left ≠ nil and u.right ≠ nil then
        Tree s = successorNode()
        link(s.parent, s.right, s.key)
        u.key() = s.key()
        u.value() = s.value()
        delete s
    else if u.left ≠ nil and u.right == nil then
        link(u.parent, s.left, k)
        if u.parent = nil then
            T = u.left
    else
        link(u.parent, s.right, k)
        if u.parent = nil then
            T = u.right
return T
```

Dimostrazione correttezza

- Caso 1: Eliminare foglie non cambia l'ordine dei nodi rimanenti.
- Caso 2: Se u è il figlio destro (sinistro) di p, tutti i valori nel sottoalbero di f sono maggiori (minori) di p, pertanto f può essere attaccato come figlio destro (sinistro) di p al posto di u.
- Caso 3: il successore s è sicuramente $\geq$ dei nodi del sottoalbero sinistro di u e sicuramente $\leq$ dei nodi del sottoalbero destro di u, pertanto può essere sostituito a u e a quel punto si ricade nel caso 2.

5.2 Costo computazionale

Tutte le operazioni sono confinate ai nodi posizionati lungo un cammino semplice dalla radice ad una foglia, pertanto detta h altezza di un albero, avranno complessità $O(h)$. Il caso pessimo si ha nel caso in cui l'altezza h sia uguale a n ($O(n)$), il caso ottimo quando $h = \log n$ ($O(\log n)$).

5.3 Alberi di ricerca bilanciati

Per mantenere un grado di complessità il più vicino possibile a $O(\log n)$ si utilizzano tecniche di bilanciamento per tenere sotto controllo l'altezza di un albero. Si introduce un fattore di bilanciamento $\beta(v)$, ovvero la massima differenza di altezza fra i sottoalberi di v.

- Alberi AVL: $\beta(v) \leq 1$ per ogni nodo v, bilanciamento ottenuto tramite rotazioni.
- B-Alberi: $\beta(v) = 0$ per ogni nodo v, specializzati per strutture in memoria secondaria.
- Alberi 2-3: $\beta(v) = 0$ per ogni nodo v, bilanciamento ottenuto tramite merge/split, grado variabile.

Rotazioni

Rotazione sinistra Si prende un nodo di un albero, si fa diventare suo figlio destro il figlio sinistro del figlio destro, successivamente si pone il nodo iniziale come figlio sinistro del vecchio nodo destro e il vecchio genitore del nodo iniziale diventa il genitore del vecchio nodo destro.

Rotazione destra Si prende un nodo di un albero, si fa diventare suo figlio sinistro il figlio destro del suo figlio sinistro, successivamente si pone il nodo iniziale come figlio destro del vecchio nodo sinistro e il vecchio genitore del nodo iniziale diventa il genitore del vecchio nodo sinistro.

```
: Tree rotateLeft(Tree x)
    Tree y ← x.right
    Tree p ← x.parent
    x.right ← y.left
    if y.left ≠ nil then
        y.left.parent ← x
    y.left ← x
    x.parent ← y
    y.parent ← p
    if p ≠ nil then
        if p.left == x then
            p.left ← y
        else
            p.right ← y
    return y
```

```
: Tree rotateRight(Tree x)
    Tree y ← x.left
    Tree p ← x.parent
    x.right ← y.right
    if y.right ≠ nil then
        y.right.parent ← x
    y.right ← x
    x.parent ← y
    y.parent ← p
    if p ≠ nil then
        if p.left == x then
            p.left ← y
        else
            p.right ← y
    return y
```

5.3.1 Alberi red-black

Un albero red-black è un albero binario di ricerca in cui ogni nodo è colorato di rosso o nero, le chiavi vengono salvate solo nei nodi interni all'albero e le foglie sono costituiti da nodi speciali **Nil**. Un albero red-black è costruito in modo che rispetti questi vincoli:

- La radice è nera.
- Tutte le foglie sono nere.

- Entrambi i figli di un nodo rosso sono neri.
- Tutti i cammini semplici da un nodo u a una delle foglie contenute nel sottoalbero radicato in u hanno lo stesso numero di nodi neri.

Memorizzazione

: Tree
Tree parent;
Tree left;
Tree right;
int color;
Item key;
Item value;

I nodi **Nil** sono nodi sentinella il cui scopo è evitare di trattare diversamente i puntatori ai nodi dai puntatori **nil**. Al posto di un puntatore **nil** si utilizza un puntatore ad un nodo speciale **Nil**. Ne esiste solo uno per risparmiare memoria e un nodo con figli **Nil** corrisponde ad una foglia nell'albero binario di ricerca.

Altezza nera

L'altezza nera $b(v)$ di un nodo v è il numero di nodi neri lungo ogni percorso da v escluso ad ogni foglia inclusa del sottoalbero. L'altezza nera di un albero red-black è pari all'altezza nera della sua radice. Quando esistono più colorazioni che rispettano i limiti possono esistere diverse altezze nere per lo stesso albero.

Inserimento

Quando si vuole inserire un nodo in un albero red-black si ricerca la posizione usando la stessa procedura per gli alberi di ricerca e si colora il nuovo nodo di rosso. Si possono pertanto violare dei vincoli in questo modo e si rende necessario aggiungere nella funzione *insertNode(Tree T, Item k, Item v)* un processo di bilanciamento (eseguito successivamente alla chiamata della funzione *link*). Il principio generale per il bilanciamento consiste nel spostarsi verso l'alto lungo il percorso di inserimento, ripristinare il vincolo dei figli neri di un nodo rosso spostando le violazioni verso l'alto mantenendo l'altezza nera dell'albero e colorando la radice di nero alla fine. Queste operazioni sono necessarie unicamente quando due nodi consecutivi sono rossi.

balanceInsert(Tree t) I nodi coinvolti sono il nodo inserito t, suo padre p, suo nonno n e suo zio z. E si possono verificare sette casi diversi in cui avviene una violazione.

- Caso 1: il nuovo nodo t non ha padre: è il primo nodo ad essere inserito o si è risaliti fino alla radice, si colora t di nero.
- Caso 2: il padre p di t è nero: non si viola nessun vincolo.
- Caso 3: t rosso, p rosso, z rosso: se z è rosso si possono colorare di nero p, z e di rosso n. Poichè tutti i cammini che passano per z e p passano per n l'altezza nera non è cambiata. Il problema ora potrebbe sussistere sul nonno, si pone pertanto $t = n$ e il ciclo continua.
- Caso 4a (4b): t rosso, p rosso, z nero: si assuma che t sia figlio destro (sinistro) di p e p figlio sinistro (destro) di n. Una rotazione a sinistra (destra) a partire dal nodo p scambia i ruoli di t e p ottenendo il caso 5a (5b) essendo entrambi i nodi coinvolti nel cambiamento rossi, l'altezza nera non cambia.
- Caso 5a (5b): t rosso, p rosso, z nero: si as-

suma che t sia figlio sinistro (destro) di p e p figlio sinistro (destro) di n. Una rotazione a destra su n porta ad una situazione in cui t e n sono figli di p. Colorando n di rosso e p di nero ci si ritrova in una situazione in cui tutti i vincoli sono rispettati.

```
: balancedInsert(Tree t)
    t.color() ← RED
    while t ≠ nil do
        Tree p ← t.parent %padre
        Tree n %nonno
        Tree z %zio
        if p ≠ nil then
            n ← p.parent
        else
            n ← nil
        if n == nil then
            z ← nil
        else
            if n.left == p then
                z ← p.right
            else
                z ← p.left
        if p == nil then
            %Caso 1
            t.color ← BLACK
            t ← nil
        else if p.color == BLACK then
            %Caso 2
            t ← nil
        else if z.color == RED then
            %Caso 3
            p.color ← z.color ← BLACK
            n.color ← RED
            t ← n
        else
            if t == p.right and p == n.left then
                %Caso 4a
                rotateLeft(p)
                t ← p
            else if t == p.left and p == n.right then
                %Caso 4b
                rotateRight(p)
                t ← p
            else
                if t == p.left and p == n.left then
                    %Caso 5a
                    rotateRight(n)
                else if t == p.right and p == n.right then
                    %Caso 5b
                    rotateLeft(n)
                p.color ← BLACK
                n.color ← RED
                t ← nil
```

Complessità La complessità totale per un inserimento è $O(\log n)$, con tre passaggi: $O(\log n)$ per scendere fino al punto di inserimento, $O(1)$ per effettuare l'inserimento e $O(\log n)$ per risalire e sistemare le violazioni. È possibile implementare un inserimento top-down che aggiusta l'albero mano a mano che scende fino al punto di inserimento.

Altezza albero red-black

Teorema In un albero red-black un sottoalbero di radice u contiene almeno $n \geq 2^{bh(u)} - 1$ nodi interni.

Dimostrazione Si dimostra per induzione sull'altezza (non sull'altezza nera).

- Caso base $h = 0$: u è una foglia **nil** e il sottoalbero con radice in u contiene $n \geq 2^{bh(u)} - 1 = 2^0 - 1 = 0$ nodi interni.
- Passo induttivo $h > 1$: allora u è un nodo interno con due figli tali che ogni figlio v ha un'altezza nera $bh(v)$ pari a $bh(u)$ se rosso o a $bh(u) - 1$ se nero. Per ipotesi induttiva ogni figlio ha almeno $2^{bh(u)} - 1$ nodi interni. Pertanto il sottoalbero con radice in u ha almeno $n \geq 2^{bh(u)-1} - 1 + 2^{bh(u)-1} - 1 + 1 = 2^{bh(u)-1} - 1$ nodi.

Teorema In un albero red-black almeno la metà dei nodi dalla radice ad una foglia deve essere nera.

Dimostrazione Per il secondo vincolo se un nodo è rosso, entrambi i suoi figli devono essere neri, pertanto la situazione in cui sono presenti il maggior numero di nodi rossi è il caso in cui rossi e neri sono alternati, dimostrando il teorema.

Teorema In un albero red-black nessun percorso da un nodo v ad una foglia è lungo più del doppio del percorso da v ad un'altra foglia.

Dimostrazione Per definizione ogni percorso da un nodo ad una qualsiasi foglia contiene lo stesso numero di nodi neri. Dal lemma precedente almeno la metà di questi nodi sono neri, pertanto al limite uno dei due percorsi è costituito da soli nodi neri mentre l'altro è costituito da nodi neri e rossi alternati.

Teorema L'altezza massima di un albero red-black contenente n nodi interni è al più $2\log(n+1)$.

Dimostrazione

$$\begin{aligned} n \geq 2^{bh(r)} - 1 &\Leftrightarrow n \geq 2^{\frac{h}{2}} - 1 \\ &\Leftrightarrow n + 1 \geq 2^{\frac{h}{2}} \\ &\Leftrightarrow \log n + 1 \geq \frac{h}{2} \\ &\Leftrightarrow h \leq 2\log(n+1) \end{aligned}$$

Cancellazione

L'algoritmo di cancellazione per gli alberi red-black è costruito su quello di cancellazione per gli alberi generici. Dopo la cancellazione è necessario decidere se ribalanciare o meno. Le operazioni di ripristino sono necessarie solo dopo la cancellazione di un nodo nero in quando se il nodo cancellato è rosso l'altezza nera rimane invariata, non sono creati nodi rossi consecutivi e la radice resta nera. Se il nodo cancellato è nero si possono rompere i vincoli uno, tre e quattro. L'algoritmo seguente ripristina i vincoli con rotazioni e cambiamenti di colore. Ci sono quattro casi possibili con i corrispettivi simmetrici.

```
: balancedDelete(Tree T, Tree t)
while T ≠ t and t.color == BLACK do
    Tree p = t.parent %Padre
    if t == p.left then
        Tree f = p.right %Fratello
        Tree ns = f.left %Nipote sinistro
        Tree nd = f.right %Nipote destro
        if f.color == RED then
            %Caso 1a
            p.color = RED
            f.color = BLACK
            rotateLeft(p)
            %t viene lasciato inalterato, pertanto si
              ricade nei casi 2, 3 o 4
        else
            if ns.color == nd.color == BLACK
             then
                %Caso 2a
                f.color = RED
                t = p
            else if ns.color == RED and nd.color
             == BLACK then
                %Caso 3a
                ns.color = BLACK
                f.color = RED
                rotateRight(f)
                %t viene lasciato inalterato, pertanto
                  si ricade nel caso 4
            else if nd.color == RED then
                %Caso 4a
                f.color = p.color()
                p.color = BLACK
                nd.color = BLACK
                rotateLeft(p)
                t = T
    else
        Tree f = p.left %Fratello
        Tree ns = f.left %Nipote sinistro
        Tree nd = f.right %Nipote destro
        if f.color == RED then
            %Caso 1b
            p.color = RED
            f.color = BLACK
            rotateRight(p)
            %t viene lasciato inalterato, pertanto si
              ricade nei casi 2, 3 o 4
        else
            if ns.color == nd.color == BLACK
             then
                %Caso 2b
                f.color = RED
                t = p
            else if nd.color == RED and ns.color
             == BLACK then
                %Caso 3b
                nd.color = BLACK
                f.color = RED
                rotateLeft(f)
                %t viene lasciato inalterato, pertanto
                  si ricade nel caso 4
            else if ns.color == RED then
                %Caso 4a
                f.color = p.color()
                p.color = BLACK
                ns.color = BLACK
                rotateRight(p)
                t = T
```

Complessità La cancellazione è concettualmente complicata ma efficiente in quanto dal caso 1 si passa al 2, 3 o 4, dal caso 2 si passa agli altri risalendo l'albero, dal caso 3 si passa al caso 4 e al caso 4 termina. È possibile visitare un massimo di $O(\log n)$ di casi, ognuno dei quali risolti in $O(1)$.

Capitolo 6

Hashing

Le tabelle hash sono l'implementazione ideale per dizionari, insiemi dinamici di coppie chiave-valore indicizzati sulla chiave. Si sceglie una funzione hash H che mappa ogni chiave $k \in \mathcal{U}$ in un intero $H(k)$. La coppia chiave-valore viene memorizzata in un vettore nella posizione $H(k)$ che viene detto tabella hash.

6.0.1 Definizioni

- L'insieme delle possibili chiavi è rapprentato dall'insieme universo $\mathcal{U}$ di dimensione u.
- Il vettore $T[0 \dots m-1]$ ha dimensione m.
- Una funzione hash è definita: $H : \mathcal{U} \to \{0, \dots, m-1\}$.
- Quando due o più chiavi nel dizionario hanno lo stesso valore di hash avviene una collisione, idealmente non dovrebbero avvenire.

Tabelle ad accesso diretto

Si utilizzano le tabelle ad accesso diretto nel caso particolare in cui $\mathcal{U} \subset \mathbb{Z}^+$. Si utilizza come funzione hash l'identità $H(k) = k$ e $m = |\mathcal{U}|$. Presenta dei problemi quando u è molto grande e uno spreco di memoria quando u non è grande ma il numero di chiavi effettivamente registrate in memoria è molto minore di m.

Funzioni hash perfette

Una funzione hash si dice perfetta se è iniettiva, ovvero $\forall k_1, k_2 \in \mathcal{U}, k_1 \neq k_2 \Rightarrow H(k_1) \neq H(k_2)$. Ci sono dei problemi in quanto lo spazio delle chiavi è spesso grande, sparso e non conosciuto ed è pertanto spesso impraticabile ottenere una funzione di hash perfetta.

6.1 Funzioni hash

Se non è possibile eliminare le collisioni si cerca per lo meno di minimizzare il loro numero, cercando funzioni che distribuiscano le chiavi uniformemente negli indici $[0, \dots, m-1]$ della tabella hash.

Uniformità semplice

Sia $P(k)$ la probabilità che una chiave sia inserita nella tabella. Sia $Q(i)$ la probabilità che una chiave finisca nella cella i: $Q(i) = \sum_{k \in \mathcal{U}: h(k)=i} P(k)$. Una funzione hash gode dell'uniformità semplice se: $\forall i \in [0, \ldots, m-1] : Q(i) = \frac{1}{m}$.

6.1.1 Come realizzare una funzione hash

Per realizzare una funzione hash con uniformità semplice è necessario che la distribuzione P sia nota, cosa non completamente possibile nella realtà. Si utilizzano pertanto tecniche euristiche. Pertanto si assuma che ogni chiave può essere tradotta in numeri interi non negativi anche interpretando la loro rappresentazione in memoria come un numero. Si intenda pertanto con *bin(k)* la rappresentazione binaria della chiave.

Estrazione

Si consideri $m = 2^p$ e $H(k) = int(b)$, dove b è un sottoinsieme di p bit presi da *bin(k)*. Presenta dei problemi in quanto selezionare bit presi dal suffisso della chiave può generare collisioni con alta probabilità (o anche in generale).

XOR

Si consideri $m = 2^p$ e $H(k) = int(b)$ dove b è dato dalla somma modulo 2 effettuata bit a bit di sottoinsiemi di p bit di *bin(k)*. Presenta dei problemi in quanto le permutazioni possono generare lo stesso valore di hash.

Metodo della divizione

Si consideri m numero dispari, meglio se primo. $H(k) = int(k) \mod m$. Non vanno bene $m = 2^p$ in quanto considera unicamente i p bit meno significativi e $m = 2^p - 1$ in quanto permutazioni con set di elementi con dimensione 2^p hanno lo stesso valore di hash. Si devono pertanto scegliere numeri primi distanti da potenza di 2 e 10.

Metodo della moltiplicazione

Si consideri un m qualsiasi, meglio se potenza di 2, $C \in]0;1[$ una costante reale e $i = int(k)$. $H(k) = \lfloor m(C \cdot i - \lfloor C \cdot i \rfloor) \rfloor$.

Implementazione Si scelga un valore $m = 2^p$, sia w la dimensione della parola in memoria: $i, m \leq 2^w$. Sia $s = \lfloor C \cdot 2^w \rfloor$. $i \cdot s$ può essere riscritto come $r_1 \cdot 2^w + r_0$, dove r_1 contiene la parte intera di iC e r_0 la parte frazionaria. Si restituiscando i p bit più significativi di r_0.

Reality check

Il metodo della moltiplicazione non fornisce hashing uniforme. Esistono test per valutare la bontà di una funzione hash: Avalanche effect che se si cambia un bit nella chiave, deve cambiare almeno la metà dei bit del valore hash e test statistici con il chi-quadro. Si devono implementare inoltre funzioni hash crittografiche.

6.2 Le collisioni

Per gestire le collisioni si rende necessario trovare posizioni alternative per le chiavi e se la chiave non si trova nella posizione attesa la si deve cercare nelle posizioni alternative. Questa ricerca deve essere $O(1)$ nel caso medio e $O(n)$ nel caso pessimo. Presentano problemi in quanto sono strutture dati complesse vista la presenza di liste e puntatori.

6.2.1 Liste o vettori di trabocco (Concatenamento o chaining)

Le chiavi con lo stesso valore di hash vengono memorizzate in una lista monodirezionale o vettore dinamico. Si memorizza un puntatore alla testa della lista nello slot $H(k)$-esimo della tabella hash. Le operazioni sono di inserimento in testa *insert*, di ricerca *lookup* o rimozione *remove* che implicano una scansione della tabella per cercare la chiave.

Analisi complessità

Si considerino i seguenti valori:

n	Numero di chiavi memorizzate nella tabella hash
m	Capacità della tabella hash
$\alpha = \frac{n}{m}$	fattore di carico
$I(\alpha)$	Ricerca con insuccesso o numero medio di accessi alla tabella per una chiave non presente
$S(\alpha)$	Ricerca con successo o numero medio di accessi alla tabella per una chiave presente

Caso pessimo Tutte le chiavi sono collocate in un'unica lista: *insert* $\Theta(1)$, *lookup(), remove()* $\Theta(n)$.

Analisi del caso medio Il caso medio dipende dalla distribuzione delle chiavi, si assuma pertanto hashing uniforme semplice e costo di calcolo della funzione di hashing $\Theta(1)$. Il valore atteso della lunghezza della lista è di $\alpha = \frac{n}{m}$. Una ricerca senza successo visita tutte le chiavi nella lista corrispondente, pertanto ha costo atteso $\Theta(1) + \alpha$, mentre una ricerca con successo visita in media la metà delle chiavi nella lista corrispondente, pertanto $\Theta(1) + \frac{\alpha}{2}$. Il fattore di carico influenza il costo computazionale delle operazioni sulla tabella hash e se $n = O(m)$, $\alpha = O(1)$, pertanto tutte le operazioni sono $O(1)$.

6.2.2 Indirizzamento aperto

Nell'indirizzamento aperto tutte le chiavi vengono memorizzate nella tabella stessa e ogni slot contiene una chiave oppure un valore **nil**. Nell'inserimento se uno slot è utilizzato se ne cerca uno alternativo. Nella ricerca si cerca nello slot prescelto e poi negli slot alternativi fino a che si trova la chiave oppure un nodo **nil**.

Definizioni

- Un'ispezione è l'esame di uno slot durante la ricerca.
- La funzione di hash viene estesa: $H : \mathcal{U} \times [0, \ldots, m-1] \to [0, \ldots, m-1]$, dove il secondo insieme del dominio rappresenta il numero di ispezione metre il codominio l'indice del vettore.

- Sequenza di ispezione: una sequenza di ispezione $[H(k,0), H(k,1), \ldots, H(k,m-1)]$ è una permutazione degli indici corrisondente all'ordine in cui vengono visitati gli slot. Non si vogliono visitare gli slot più di una volta e potrebbe rendersi necessario visitare tutti gli slot della tabella.

Fattore di carico Il fattore di carico è compreso tra 0 e 1 e la tabella può andare in overflow.

Tecniche di ispezione

Hashing uniforme La situazione ideale prende il nome di hashing uniforme, in cui ogni chiave ha la stessa probabilità di avere come sequenza di ispezione una delle qualisasi $m!$ permutazioni degli indici.

Ispezione lineare La funzione di $H(k,i) = (H_1(k) + h \cdot i) \mod n$. La sequenza di ispezione è determinata dal primo elemento. Al massimo m sequenze di ispezione diverse sono possibili.

Agglomerazione primaria Causa lunghe sotto-sequenze occupate che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $\frac{i+1}{m}$. I tempi medi di inserimento e cancellazione crescono.

Ispezione quadratica La funzione di $H(k,i) = (H_1(k) + h \cdot i^2) \mod n$. Dopo il primo elemento $H_1(k,0)$, le ispezioni successive hanno un offset che dipende da una funzione quadratica nel numero di ispezione i. La sequenza risultante non è una permutazione, al massimo m sequenze di ispezioni distinte sono possibili.

Agglomerazione secondaria Se due chiavi hanno la stessa ispezione iniziale le loro sequenze sono identiche.

Doppio hashing Funzione: $H(k,i) = (H_1(k) + iH_2(k)) \mod n$. Esistono pertanto due funzioni ausiliare, H_1 fornisce la prima ispezione, H_2 fornisce l'offset rispetto alla prima ispezione. Sono possibili al massimo m^2 sequenze di ispezione. Per garantire una permutazione completa $H_2(k)$ e m devono essere primi tra loro. Pertanto se si sceglie $m = 2^p$ $H_2(k)$ deve restituire numeri dispari. Se si sceglie m primo, $H_2(k)$ deve restituire numeri minori di m.

Cancellazione

Non si possono sostituire le chiavi da eliminare con **nil** in quanto potrebbe introdurre errori nella ricerca che terminerebbe troppo presto. Si utilizza pertanto un valore speciale **deleted** che vengono trattati come slot vuoti dal'inserimento e come slot pieni dalla ricerca. Il tempo di ricerca non dipende più da α ma il concatenamento diventa più comune.

Implementazione dell'hashing doppio

```
: Hash
    Item [] K %Tabella delle chiavi
    Item [] V %Tabella dei valori
    int m %Dimensione della tabella
    Hash Hash(int dim)
        Hash t = new Hash
        t.m = dim
        t.K = new Item [0, ..., dim − 1] t.V = new Item [0, ..., dim − 1] for i = 0 to dim-1 do
            t.K[i] = nil
        return t
    int scan(Item k, boolean insert)
        int c = m
        int i = 0
        int j = H(k)
        while K[j] ≠ k and K[j] ≠ nil and i < m do
            if K[j] == deleted and c == m then
                c = j
            j = (j + H′(k)) mod n
            i += 1
        if insert and K[j] ≠ k and c < m then
            j = c
        return j
    Item lookup(Item k)
        int i = scan(k, false)
        if K[i] == k then
            return V[i]
        else
            return nil
    insert(Item k, Item v)
        int i = scan(k, true)
        if K[i] == nil or K[i] == deleted or K[i] == k then
            K[i] = k
            V[i] = v
        else
            %Errore, tabella hash piena
    remove(Item k)
        int i = scan(k, false)
        if k[i] = k then
            K[i] = deleted
```

6.3 Complessità

Metodo	α	$I(\alpha)$	$S(\alpha)$
Lineare	$0 \leq \alpha < 1$	$\frac{(1-\alpha)^2+1}{2(1-\alpha)^2}$	$\frac{1-\frac{\alpha}{2}}{1-\alpha}$
Hashing doppio	$0 \leq \alpha < 1$	$\frac{1}{1-\alpha}$	$-\frac{1}{\alpha}\ln(1-\alpha)$
Liste di trabocco	$\alpha \geq 0$	$1+\alpha$	$1+\frac{\alpha}{2}$

Si noti come la complessità aumenti con l'aumentare di α. Pertanto sopra una soglia prefissata t_α, solitamente tra 0.5 e 0.75 si alloca una nuova dimensione $2m$ e si reinseriscono tutte le chiavi presenti nella tabella. Si dimezza pertanto il fattore di carico e si eliminano tutti gli elementi **deleted**. Nel caso pessimo c'è un costo di $O(m)$ per la ristrutturazione nel caso pessimo con costo ammortizzato costante.

Capitolo 7

Insiemi e dizionari

7.1 Insiemi

7.1.1 Insiemi realizzati con vettori booleani

Implementazione

: Set (vettore booleano)

```
boolean [] V
int size
int dim
Set Set(int m)
    Set t = new Set
    t.size = 0
    t.dim = =m
    t.V = [false ] * m
    return t
boolean contains(int x)
    if 1 ≤ x ≤ dim then
        return V[x]
    else
        return false
int size()
    return size
insert(int x)
    if 1 ≤ x ≤ dim then
        if not V[x] then
            size += 1
            V[x] = true
remove(int x)
    if 1 ≤ x ≤ dim then
        if V[x] then
            size == 1
            V[x] = false
```

```
Set union(Set A, Set B)
    Set C= Set(max(A.dim, B.dim))
    for i = 1 to A.dim do
        if A.contains(i) then
            C.insert(i)
    for i = 1 to B.dim do
        if B.contains(i) then
            C.insert(i)
    return C
Set intersection(Set A, Set B)
    Set C= Set(min(A.dim, B.dim))
    for i = 1 to min(A.dim, B.dim) do
        if A.contains(i) and B.contains(i)
         then
            C.insert(i)
    return C
Set difference(Set A, Set B)
    Set C= Set(A.dim)
    for i = 1 to A.dim do
        if A.contains(i) and not
         B.contains(i) then
            C.insert(i)
    return C
```

Caratteristiche

Gli insiemi possono essere memorizzati come un vettore di m elementi, se si vogliono salvare gli interi da 1 a m. Questo tipo di implementazione è molto semplice ed è efficiente verificare se un elemento appartiene ad un insieme. Come svantaggi presenta lo spreco di memoria, in quanto la memoria occupata è sempre $O(m)$ indipendentemente dagli elementi salvati. E alcune operazioni sono inefficienti in $O(m)$.

7.1.2 Insiemi realizzati con liste

Liste non ordinate

Operazioni di ricerca, inserimento e cancellazione $O(n)$, operazioni di inserimento assumendo assenza $O(1)$. Operazioni di unione intersezione, differenza $O(nm)$.

```
: Set difference(Set A, Set B)
    Set C = Set()
    foreach s ∈ A do
        if not B.contains(s) then
            C.insert(s)
    return C
```

Liste ordinate

Ricerca $O(n)$ per liste e $O(\log n)$ per i vettori, inserimento e cancellazione $O(n)$, unione, intersezione e differenza $O(n)$.

```
: List intersection(List A, List B)
    List C = List()
    Pos p = A.head()
    Pos q = B.head()
    while not A.finished(p) and not B.finished(q) do
        if A.read(p) = B.read(q) then
            C.insert(C.tail(), A.read(p))
            p = A.next(p)
            q = B.next(q)
        else if A.read(p) ¡ B.read(q) then
            p = A.next(p)
        else
            q = B.next(q)
    return C
```

7.1.3 Strutture dati complesse

Alberi bilanciati

Si ottengono insiemi con ordinamento, con ricerca, inserimento, cancellazione $O(\log n)$, iterazione $O(n)$.

Tabelle hash

Si ottengono insiemi senza ordinamento, con ricerca, inserimento, cancellazione $O(1)$, iterazione $O(m)$.

7.2 Bloom filters

I bloom filters sono una via di mezzo tra gli insiemi realizzati con i vettori booleani e le tabelle hash, sono una struttura dati dinamica con bassa occupazione di memoria che non offre la possibilità di cancellazioni, nessuna memorizzazione e dà risposte probabilistiche.

7.2.1 Specifica

L'operazione di *insert(x)* inserisce l'elemento x nel bloom filter. ***booleans*** *contains(x)* se restituisce **false** l'elemento non sicuramente è presente, ma è possibile che ci siano dei falsi positivi. Si deve fare un trade-off tra occupazione di memoria e probabilità di un falso positivo. Indicata con ε la probabilità di un falso positivo, i bloom filters richiedono $1.44 \log_2(\frac{1}{\varepsilon})$ bit per elemento inserito.

7.2.2 Applicazioni

In chrome vengono utilizzati per indicare i siti con possibile malware, o in generale vengono utilizzati quando una verifica locale permette di evitare operazioni di I/O più costose. I falsi positivi inoltre possono essere utilizzati per mascherare un messaggio e garantire così un certo livello di privacy.

7.2.3 Implementazione

Vengono implementati attraverso un vettore booleando A di m bit inizializzato a **false** e k funzioni di hash $H_1, \ldots, H_k : U \to [0, \ldots, m-1]$.

```
: insert(x)
  for i = 1 to k do
    A[H_i(x)] = true
```

```
: boolean contains(x)
  for i = 1 to k do
    if A[H_i(x)] == false then
      return
    false
  return true
```

7.2.4 Caratterizzazione matematica

Dati n oggetti, m bit e k funzioni hash, la probabilità di un falso positivo è:

$$\varepsilon = (1 - e^{-k\frac{n}{m}})^k$$

Dati n oggetti e m bit il valore ottimale per k è:

$$k = \frac{m}{n} \ln 2$$

Dati n oggetti e una probabilità di falsi positivi ε il numero di bit m richiesti è pari a:

$$m = \frac{n \ln \varepsilon}{(\ln 2)^2}$$

Capitolo 8

Grafi

8.0.1 Definizioni

Grafo orientato Si dice grafo orientato la coppia $G = (V, E)$, dove V è l'insieme di nodi o vertici ed E è l'insieme di coppie ordinate (u, v) di nodi dette anche archi.

Grafo non orientato Si dice grafo non orientato la coppia $G = (V, E)$, dove V è l'insieme di nodi o vertici ed E è l'insieme di coppie non ordinate (u, v) di nodi dette anche archi.

- Un vertice v è detto adiacente a u se esiste un arco (v, u).
- Un arco (v, u) è detto incidente a v e u.
- In un grafo non orientato la relazione di adiacenza è simmetrica.
- $n = |V|$ numero di nodi.
- $m = |E|$ numero di archi.
- In un grafo non orientato $m \leq \frac{n(n-1)}{2} = O(n^2)$.
- In un grafo orientato $m \leq n^2 - n = O(n^2)$.
- La complessità di algoritmi sui grafi deve essere espressa sia un termini di n che di m.
- Un grafo con un arco tra tutte le coppie di nodi è detto completo.
- Un grafo si dice sparso se ha "pochi archi", ovvero con $m = O(n)$ o $m = O(n \log n)$.
- Un grafo si dice dento se ha "molti archi", ovvero con $m = \Omega(n^2)$.
- Un albero libero è un grafo connesso con $m = n - 1$.
- Un albero radicato è un albero libero sul quale è stata scelta una radice.
- Un insieme di alberi è un grafo detto foresta.
- Il grado di un nodo è il numero di archi incidenti su di esso.

- Nei grafi orientati si divide il grado in grado entrante per gli archi che incidono su di esso e in grado uscente per gli archi che incidono da esso.
- Si dice cammino C di lunghezza k in un grafo $G = (V, E)$ una sequenza di nodi $u_0, \ldots, u_k$ tali che $(u_i, u_{i+1}) \in E$ per $0 \leq i \leq k-1$.
- Si dice semplice un cammino i cui nodi sono tutti distinti.
- Si dice peso di un arco un valore associato all'arco dato da una funzione di peso $w : V \times V \to \mathbb{R}$.
- Un nodo v si dice raggiungibile da un nodo u se esiste almeno un cammino da u a v.
- Un grafo non orientato $G = (V, E)$ è connesso se e solo se ogni suo nodo è raggiungibile da ogni altro nodo.
- G' è detto sottografo di G $G' \subseteq G$ se e solo se $V' \subseteq V$ e $E' \subseteq E$.
- G' è sottografo massimale di G se e solo se non esiste un altro sottografo G'' tale che G'' è connesso e più grande di G'.
- Un grafo $G' = (V', E')$ è una componente connessa di G se e solo se è un sottografo connesso e massimale di G.
- In un grafo non orientato $G = (V, E)$ un ciclo C di lunghezza $k > 2$ è una sequenza di nodi $u_0, \ldots, u_k$ tale che $(u_i, i_{i+1}) \in E$ per ogni $0 \leq i \leq k-1$ e $u_0 = u_k$. Un ciclo è detto semplice se tutti i suoi nodi sono distinti.
- Un grafo è detto aciclico se non contiene cicli. Se orientato è detto DAG (directed acyclic graph).
- Dato un DAG, un suo ordinamento topologico è un ordinamento lineare dei suoi nodi tale che se $(u, v) \in E$, allora u appare prima di v nell'odinamento.
- Un grafo orientato $G = (V, E)$ si dice fortemente connesso se e solo se ogni suo nodo è raggiungibile da ogni altro suo nodo.
- Un grafo $G' = (V', E')$ è una componente fortemente connessa di G se e solo se è un sottografo connesso e massimale di G.

8.0.2 Specifica

: Graph

```
Graph() %Crea un nuovo grafo
Set V() %Restituisce l'insieme di tutti i nodi
int size() %Restituisce il numero di nodi
Set adj(Node u) %Restituisce i nodi adiacenti al nodo u
insertNode(Node u) %Inserisce il nodo u nel grafo
insertEdge(Node u, Node v) %Inserisce l'arco tra u e v nel grafo
deleteNode(Node u) %Elimina il nodo u dal grafo
deleteEdge(Node u, Node v) %Elimina l'arco tra i nodi u e v dal grafo
```

In alcuni casi il grafo è dinamico ma sono possibili solo inserimenti e non è necessaria la parte di eliminazione (grafo caricato all'inizio).

8.1 Memorizzare grafi

8.1.1 Matrice di adiacenza

Le matrici di adiacenza sono ideali per i grafi densi. Un grafo orientato viene salvato in una matrice di dimensione $n \times n$ bit tale che:

$$m_{uv} = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases}$$

Per il grafo non orientato basta salvare la matrice sopra la diagonale principale esclusa in modo da occupare solamente $\frac{n(n-1)}{2}$ bit.

Grafi pesati Per i grafi pesati la matrice di booleani diventa una matrice di reali tale che:

$$m_{uv} = \begin{cases} peso & (u,v) \in E \\ +\infty & (u,v) \notin E \end{cases}$$

Analisi complessità

- Spazio richiesto $O(n^2)$.
- Verificare se u è adiacente a v richiede $O(1)$.
- Iterare su tutti gli archi richiede $O(n^2)$

8.1.2 Liste di adiacenza

Le liste di adiacenza sono ideali per i grafi sparsi. Viene creato un vettore di dimensione n tale che in posizione u si trovi un insieme tale che contenga gli adiacenti. Occupa $an + bm$ bit. Dove a è la dimensione del puntatore al nodo adiacente e b è la dimensione dell'adiacente. Per il grafo non orientato viene salvato anche l'arco inverso e occupa $an + 2bm$.

Grafi pesati Gli elementi della lista puntata sono coppie nodo di arrivo-peso.

Analisi complessità

- Spazio richiesto $O(n + m)$.
- Verificare se u è adiacente a v richiede $O(n)$.
- Iterare su tutti gli archi richiede $O(n + m)$

8.1.3 Iterazioni su nodi e archi

: Iterazione su tutti i nodi del grafo

foreach $u \in G$.`V()` **do**
  Esegui operazioni sul nodo u

: Iterazione su tutti i nodi e archi del grafo

```
foreach u ∈ G.V() do
    Esegui operazioni sul nodo u;
    foreach v ∈ G.adj(u) do
        Esegui operazioni sull'arco (u, v)
```

8.2 Visite dei grafi

Dato un grafo $G = (V, E)$ e $r \in V$ radice o sorgente si intende per visita visitare una e una sola volta tutti i nodi che possono essere raggiunti da r.

: `graphTraversal(Graph` G`, Node` r`)`

```
Set S = Set() %Insieme generico
S.insert(r) %Da specificare
{Marca il nodo r}
while S.size() > 0 do
    Node u = S.remove() %Da specificare
    {Visita il nodo u}
    foreach v ∈ G.adj(u) do
        {Visita l'arco (u,v)}
        if v non è stato marcato then
            Marca il nodo v
            S.insert(v) %Da specificare
```

8.2.1 Visita in ampiezza o breadth-first search

In questo tipo di visita si visitano i nodi per livelli crescenti a partire dalla sorgente. Gli obiettivi di questa visita sono di visitare i nodi a distanza k prima dei nodi a distanza $k + 1$, calcolare il cammino più breve da r a tutti gli altri nodi (distanze misurate come numero di archi attraversati). Generare un albero breadth-first contenente tutti i nodi raggiungibili da r tale per cui il cammino dalla radice r al nodo u nell'albero corrisponde al cammino più breve nel grafo.

Calcolo dei cammini più brevi o del numero di Erdös

L'algoritmo di erdos oltre a determinare la lunghezza del cammino più breve genera l'arco di copertura bfs che contiene tali cammini.

```
: bfs(Graph G, Node r)
Queue Q = Queue()
Q.enqueue(r)
boolean [] visited = new boolean [G.size()]
foreach u ∈ G.V() - {r} do
    visited[u] = false
visited[r] = true
while not Q.empty() do
    Node u = Q.dequeue()
    {Visita il nodo u}
    foreach v ∈ G.adj(u) do
        {Visita l'arco (u,v)}
        if not visited[v] then
            visited[v] = true
            Q.enqueue(v)
```

```
: erdos(Graph G, Node r, int [] erdos, Node [] parents)
Queue Q = Queue()
Q.enqueue(r)
foreach u ∈ G.V() - {r} do
    erdos[u] = ∞
erdos[r] = 0
parents[r] = nil
while not Q.empty() do
    Node u = Q.dequeue()
    {Visita il nodo u}
    foreach v ∈ G.adj(u) do
        if erdos[v] = ∞ then
            erdos[v] = erdos[u] + 1
            parents[v] = u
            Q.enqueue(v)
```

```
: printPath(Node r, Node s, Node [] parents)
if r = s then
    print s
else if parents[s] == nil then
    print "error"
else
    printPath(r, parents[s], parents)
    print s
```

Complessità

La complessità di una bfs è $O(m+n)$ in quanto ognuno degli n nodi viene salvato nella coda al più una volta. Ogni volta che un nodo viene estratto tutti i suoi archi vengono analizzati una sola volta. Il numero di archi analizzati è pertanto $m = \sum_{u \in V} out_d(u)$, dove $out_d(u)$ è l'out-degree del nodo u.

8.2.2 Visita in profondità o depth-first search

La visita in profondità è una visita ricorsiva: per ogni nodo adiacente si visita ricorsivamente tale nodo, visitando ricorsivamente i suoi adiacenti e così via. Questo tipo di ricerca è spesso una subroutine in altri problemi, visita tutto il grafo, nono solo i nodi raggiungibili dalla radice e genera pertanto una foresta formata da una collezione di alberi depth-first. Si realizza attraverso uno stack che viene reso implicito attraverso la ricorsione.

Implementazione ricorsiva

```
: dfs(Graph G, Node u, boolean [] visited)
    visited[u] = true
    {visita il nodo u in pre-order}
    foreach v ∈ G.adj(u) do
        if not visited[v] then
            {Visita l'arco (u,v)}
            dfs(G, v, visited)
    {visita il nodo u in post-order}
```

Implementazione iterativa

Una dfs ricorsiva in casi di grafi molto profondi può superare la dimensione dello stack del linguaggio, si rende pertanto esplicito lo stack utilizzando una versione iterativa. Un nodo può essere inserito nella pila più volte in quanto il controllo viene fatto all'estrazione. Ha complessità $O(m+n)$ con $O(m)$ visite degli archi, $O(m)$ estrazioni e inserimenti e $O(n)$ visite dei nodi. Per la visita in post order quando un nodo viene scoperto viene inserito nello stack con il tag *discovery*, quando viene estratto un nodo con il tag *discovery* viene re-inserito con il tag *finish* e vengono inseriti tutti i suoi vicini. Quando viene estratto un nodo con il tag *finish* viene fatta la post-visita.

```
: dfs(Graph G, Node r)
Stack S = Stack()
S.push(r)
boolean [] visited = new boolean [G.size()]
foreach u ∈ G.V() do
    visited[u] = false
while not S.empty() do
    Node u = S.pop()
    if not visited[u] then
        {Visita il nodo u in pre-order}
        visited[u] = true
        foreach v ∈ G.adj(u) do
            {Visita l'arco (u,v)}
            S.push(v)
```

Componenti connesse

Ci si pone il problema di verificare se il grafo è connesso e se non lo è di identificare le sue componenti connesse. Un grafo risulta connesso se al termine della dfs tutti i suoi nodi sono marcati, altrimenti la visita deve ricominciare da capo da un nodo non marcato identificando una nuova componente del grafo. L'appartenenza ad una componente connessa viene identificata attraverso un vettore *id[]* tale per cui *id[u]* è l'identificatore della componente connessa di appartenenza.

```
: int [] cc(Graph G)
int [] id = new int [G.size()]
foreach u ∈ G.V() do
    id[u] = 0
int counter = 0
foreach u ∈ G.V() do
    if id[u] == 0 then
        counter += 1
        ccdfs(G, counter, u, id)
return id
```

```
: int [] ccdfs(Graph G, int counter,
Node u, int [] id)
id[u] = counter
foreach v ∈ G.adj(u) do
    if id[v] == 0 then
        ccdfs(G, counter, v, id)
```

```
: boolean [] hasCycleRec(Graph G, Node u, Node p, boolean [] visited)
visited[u] = true
foreach v ∈ G.adj(u) - {p} do
    if visited[u] then
        return true
    else if hasCycleRec(G, v, u, visited) then
        return true
return false
```

Grafo non orientato aciclico

```
: boolean [] hasCycle(Graph G)
boolean [] visited = new boolean [G.size()]
foreach u ∈ G.V() do
    visited[u] = false
foreach u ∈ G.V() do
    if not visited[u] then
        if hasCycleRec(G, u, nil, visited) then
            return true
return false
```

Grafo orientato aciclico

Classificazione degli archi Ogni qual volta si esamina un arco da un nodo marcato ad un nodo non marcato tale arco diventa un arco dell'albero di copertura dfs T. Gli archi non inclusi nell'albero possono essere divisi in tre categorie:

- Se u è un antenato di v in T (u, v) è detto arco in avanti.
- Se u è un discendente di v in T (u, v) è detto arco all'indietro.
- Se i primi due non sussistono viene detto arco di attraversamento.

Indicato con time un contatore, il vettore dt il discovery time, o tempo di scoperta di un nodo e ft il finish time o tempo di fine di un nodo.

```
: dfs-schema(Graph G, Node u, int & time, int [] dt, int [] ft)
{Visita il nodo u in pre-order}
time += 1
dt[u] = time
foreach v ∈ G.adj(u) do
    {Visita l'arco (u, v) qualsiasi}
    if dt[v] = 0 then
        {Visita l'arco (u, v) albero}
        dfs-schema(G, v, time, dt, ft)
    else if dt[u] > dt[v] and ft[v] = 0 then
        Visita l'arco (u, v) indietro
    else if dt[u] < dt[v] and ft[v] ≠ 0 then
        Visita l'arco (u, v) avanti
    else
        Visita l'arco (u, v) attraversamento
{Visita il nodo u in post-order}
time +=1
ft[u] = time
```

Classificare gli archi permette di dimostrare proprietà rispetto a questa classificazione e pertanto costruire degli algoritmi migliori. In particolare data una visita dfs di un grafo G per ogni coppia di nodi $u, v \in V$ solo una delle seguenti condizioni è vera:

- Gli intervalli $[dt[u], ft[u]]$ e $[dt[v], ft[v]]$ sono non-sovrapposti: allora u e v non sono discendenti l'uno dell'altro nella foresta DF.
- L'intervallo $[dt[u], ft[u]]$ è contenuto in $[dt[v], ft[v]]$, allora u è un discendente di v nella foresta DF.
- L'intervallo $[dt[v], ft[v]]$ è contenuto in $[dt[u], ft[u]]$, allora v è un discendente di u nella foresta DF.

Teorema un grafo è aciclico se e solo se non esistono archi all'indietro nel grafo.

Dimostrazione Se esiste un ciclo, sia u il primo nodo del ciclo che viene visitato e $(v, u$ un arco del ciclo. Il cammino che connette v a u verrà prima o poi visitato e da v verrà scoperto l'arco all'indietro (v, u). Se esiste un arco all'indietro (u, v), dove v è un antenato di u allora esiste un cammino da v a u e un arco da u a v, ovvero un ciclo.

Ordinamento topologico

L'algoritmo per un ordinamento topologico consiste in una dfs in cui l'operazione di visita consiste nell'aggiungere il nodo in testa ad una lista a tempo di fine (ovvero post-ordine). L'output sarà pertanto una sequenza di nodi ordinata per tempo decrescente di fine. In questo modo quando un nodo è finito tutti i suoi discendenti sono stati aggiunti alla lista e aggiungendolo in testa è nell'ordine corretto.

```
: hasCycle(Graph G, Node u, int & time, int [] dt, int [] ft)
time += 1
dt[u] = time
foreach v ∈ G.adj(u) do
    if dt[v] == 0 then
        if hasCycle(G, v, time, dt, ft) then
            return true
    else if dt[u] > dt[v] and ft[v] == 0 then
        return true
time +=1
ft[u] = time
return false
```

```
: Stack topSort(Graph G)
Stack S = Stack()
boolean [] visited = boolean [G.size()]
foreach u ∈ G.V() do
    visited[u] = false
foreach u ∈ G.V() do
    if not visited[u] then
        ts-dfs(G, visited, S)
return S
```

```
: Stack ts-dfs(Graph G, Node u, boolean [] visited, Stack S)
visited[u] = true
foreach v ∈ G.adj(u) do
    if not visited[v] then
        ts-dfs(G, v, visited, S)
S.push(u)
```

Algoritmo di Kosaraju

Utilizzato per trovare le componenti fortemente connesse di un grafo orientato. Per farlo effettua una visita dfs al grafo, calcola il grafo trasposto G_t ed esegue una visita dfs sul grafo G_t utilizzando *cc* esaminando i nodi in ordine inverso di tempo di fine della prima visita. Le componenti connesse e i relativi alberi DF rappresentano le componenti fortemente connesse di G. (Sostituito dall'algoritmo di Tarjan che richiede una sola visita). Applicando l'algoritmo topologico si è sicuri che se un arco

```
: int [] scc(Graph G)
Stack S = topSort(G) %First visit
Graph G^T = transpose(G) %Graph transposal
return ccG^T, S %Second visit
```

(u, v) non appartiene ad un ciclo, u viene listato prima di v nella sequenza ordinata e gli archi di un ciclo vengono listati in un qualche ordine che è ininfluente. Si utilizza *topSort()* per ottenere i nodi in ordine decrescente di tempo di fine. Essendo che ogni passo di questo algoritmo richiede $O(n + m)$, la complessità totale è $O(m + n)$.

Calcolo del grafo trasposto Dato un grafo orientato $G = (V, E)$ il corrispettivo grafo trasposto $G_t = (V, E_t)$ è un grafo con gli stessi nodi ma con gli archi orientati nel senso opposto: $E_t = \{(u, v)|(v, u) \in E\}$. L'algoritmo per il suo calcolo ha costo $O(m + n)$.

```
: int [] transpose(Graph G)
Graph G^T = Graph(G)
foreach u ∈ G.V() do
    G^T.insertNode(u)
foreach u ∈ G.V() do
    foreach v ∈ G.adj(u) do
        G^T.insertEdge(v, u)
return G^T
```

Calcolo delle componenti connesse Invece di esaminare i nodi in ordine arbitrario questa versione li esamina nell'ordine LIFO memorizzato nello stack.

```
: int [] cc(Graph G, Stack S)
int [] id = new int [G.size()]
foreach u ∈ G.V() do
    id[u] = 0
int counter = 0
while not S.empty() do
    u = S.pop()
    if id[u] == 0 then
        counter += 1
        ccdfs(G, counter, u, id)
return id
```

```
: ccdfs(Graph G, int counter, Node u,
int [] id)
id[u] = counter
foreach v ∈ G.adj(u) do
    if id[u] == 0 then
        ccdfs(G, counter, v, id)
```

Dimostrazione di correttezza Si costruisca il grafo delle componenti $C(G) = (V_C, E_C)$, dove $V_C = \{C_1, \ldots, C_k\}$, dove C_i è la i-esima SCC di G e $E_C = \{(C_i, C_j) | \exists (u_i, u_j) \in E : u_i \in C_i \wedge u) j \in C_j\}$. Il grafo delle componenti è aciclico e inoltre $C(G^T) = [C(G)]^T$. I discovery e finish time del grafo delle componenti corrispondono ai corrispettivi del primo nodo visitato in C: $dt(C) = \min\{dt(u) | u \in C\}$ e $ft(C) = \max\{ft(u) | u \in C\}$.

Teorema Siano C e C' due distinte SCCs nel grafo orientato $G = (V, E)$. Se c'è un arco $(C, C') \in E_C$, allora $ft(C) > ft(C')$.

Corollario Siano C_u e C_v due SCCs distinte nel grafo orientato $G = (V, E)$, se cè un arco $(u, v) \in E_T$ tale che $u \in C_u$ e $v \in C_v$, allora $ft(C_u) < ft(C_v)$.

$$
\begin{aligned}
(u, v) \in E_T \Rightarrow \\
(v, u) \in E \Rightarrow \\
(C_v, C_u) \in E_C \Rightarrow \\
ft(C_v) > ft(C_u) \Rightarrow \\
ft(C_u) < ft(C_v)
\end{aligned}
$$

Conclusione Se le componenti C_u e C_v sono connesse da un arco $(u, v) \in E_T$ allora dal corollario $ft(C_u) < ft(C_v)$ e dall'algoritmo la visita di C_v inizierà prima della visita di C_u. Non esistono cammini tra C_u e C_v in E_T altrimenti il grafo sarebbe ciclico, pertando dall'algoritmo la visita di C_v non raggiungerà mai C_u, pertanto *cc()* assegnerà correttamente gli identificatori delle componenti ai nodi.

Capitolo 9

Strutture dati speciali

9.1 Code con priorità

Una coda con priorità è una struttura dati astratta simile ad una coda in cui ogni elemnto inserito possiede una priorità. Si dividono in min-priority queue in cui l'estrazione avviene per valore crescente di priorità e max-priority queue, in cui l'estrazione avviene per valore decrescente di priorità. Le operazioni permesse sono di inserimento in coda, estrazione dell'elemento con priorità di valore min/max e di modifica di priorità di un elemento inserito.

9.1.1 Specifica

: MinPriorityQueue
%Crea una coda con priorità vuota `MinPriorityQueue()`
%Restituisce **true** se la coda con priorità è vuota **boolean** `isEmpty()`
%Restituisce l'elemento minimo di una coda con priorità non vuota Item `min()`
%Rimuove e restituisce l'elemento minimo di una coda con priorità non vuota Item `DeleteMin()`
%Inerisce l'elemento x con priorità p all'interno della coda con priorità, restituendo un oggetto PriorityItem che identifica x all'interno della coda PriorityItem `insert`(Item *x,* **int** *p*)
%Diminuisce la priorità dell'oggetto identificato da y portandola a p `decrease`(PriorityItem *x,* **int** *p*)

9.1.2 Implementazioni

Metodo	Lista o vettore non ordinato	Lista ordinata	Vettore ordinato	Albero RB
min()	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$
deleteMin()	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$
insert()	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
decrease()	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$

L'implementazione più comune è attraverso gli heap, strutture speciali che associano i vantaggi di un albero (esecuzioni in tempo $O(\log n)$) e vantaggi di un vettore (memorizzazione efficiente).

9.1.3 Heap

Definizioni di alberi

Albero binario perfetto In un albero binario perfetto tutte le foglie hanno la stessa altezza h, tutti i nodi interni hanno grado 2 e dato n il numero di nodi ha altezza $h = \lfloor \log n \rfloor$. Data l'altezza h ha numero di nodi $n = 2^{h+1} - 1$.

Albero binario completo In un albero binario completo tutte le foglie hanno altezza h o $h - 1$, tutti i nodi a livello h sono accatastati a sinistra e tutti i nodi hanno grado 2 tranne al più uno. Dato il numero di nodi n ha altezza $h = \lfloor \log n \rfloor$.

Alberi binari heap

Un albero binario max-heap (min-heap) è un albero completo tale che il valore memorizzato nel nodo è maggiore (minore) dei valori memorizzati nei suoi figli. Un albero heap non impone una relazione di ordinamento totale tra i figli di un nodo, pertanto è un ordinamento parziale.

Memorizzazione

Gli alberi binari heap sono memorizzati tramite un vettore heap $A = [0, \ldots, n-1]$ tale per cui:

- La radice si trova all'indice $i = 0$, ovvero *root()=0*.
- Il padre del nodo salvato all'indice i si trova in $p(i) = \lfloor \frac{(i-1)}{2} \rfloor$.
- Il figlio sinistro del nodo salvato all'indice i si trova in $l(i) = 2i + 1$.
- Il figlio destro del nodo salvato all'indice i si trova in $r(i) = 2i + 2$.

Proprietà max-heap sul vettore: $A[i] \geq A[l(i)], [i] \geq A[r(i)]$. Proprietà min-heap sul vettore: $A[i] \leq A[l(i)], [i] \leq A[r(i)]$.

heapsort()

L'*heapsort()* ordina un max-heap "in-place" prima costruendo un max-heap nel vettore e poi spostando l'elemento max in ultima posizione, ripristinando la proprietà max-heap. Utilizza pertanto le funzioni *heapBuild()* che costruisce un max-heap a partire dal vettore non ordinato e *maxHeapRestore()* che ripristina le proprietà max-heap.

maxHeapRestore() Riceve come input un vettore A e un indice i, tale per cui gli alberi binari con radici $l(i)$ e $r(i)$ sono max-heap e il suo obiettivo è modificare "in-place" il vettore A in modo tale che l'albero binario con radice i sia un max-heap. Essendo che ad ogni chiamata vengono eseguiti $O(1)$ confronti, e se il nodo i non è massimo si chiama ricorsivamente sui figli e che l'esecuzione termina quando si raggiunge una foglia e l'altezza dell'albero è pari a $\lfloor \log n \rfloor$ $T(n) = O(\log n)$.

```
: maxHeapRestore(Item [] A, int i, int i)
int max = i
if l(i) ≤ dim and A[l(i)] > A[max] then
    max = l(i)
if r(i) ≤ dim and A[r(i)] > A[max] then
    max = r(i)
if i ≠ max then
    A[i] ↔ A[max] maxHeapRestore(A, max, dim)
```

Dimostrazione della correttezza per induzione sull'altezza Al termine dell'esecuzione l'albero radicato in $A[i]$ rispetta la proprietà max-heap.

- Caso base $h = 0$: se l'altezza è zero esiste un unico nodo che rispetta la proprietà.
- Ipotesi induttiva: l'algoritmo funziona correttamente su tutti gli alberi di altezza inferiore ad h.
- Passo induttivo:
  - Caso 1: $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$, allora l'albero radicato in $A[i]$ rispetta la proprietà di max-heap e l'esecuzione termina.
  - Caso 2: $A[l(i)] > A[i], A[r(i)] > A[i]$, viene fatto uno scambio $A[i] \leftrightarrow A[l(i)]$, dopo lo scambio $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$, il sottoalbero radicato in $r(i)$ rimasto inalterato, mentre il sottoalbero radicato in $l(i)$ può aver perso la proprietà, pertanto si svolge la chiamata ricorsiva su di esso che ha altezza minore di h, pertanto è corretto.
  - Caso 3: simmetrico rispetto al caso 2.

heapBuild() Sia $A[1, \ldots, n]$ un vettore da ordinare, tutti i nodi $A[\lfloor \frac{n}{2} \rfloor + 1, \ldots, n]$ sono foglie dell'albero e pertanto heap contenenti un elemento. La procedura *heapBuild()* attraversa i restanti nodi dell'albero a partire da $\lfloor \frac{n}{2} \rfloor$ fino ad 1 ed esegue *maxHeapRestore()* su ognuno di essi. L'algoritmo ha complessità $\Theta(\log n)$. Aggiungere dimostrazione, non la capisco.

```
: heapBuild(Item [] A, int n)
for i = ⌊n/2⌋ down to 1 do
    maxHeapRestore(A, i, n)
```

Correttezza Si dimostra attraverso l'invariante di ciclo: all'inizio di ogni iterazione del ciclo **for** i nodi $[i+1, \ldots, n]$ sono radice di uno heap.

- Inizializzazione: all'inizio $i = \lfloor \frac{n}{2} \rfloor$. Si supponga che $\lfloor \frac{n}{2} \rfloor + 1$ non sia una foglia, pertanto esiste almeno il figlio sinistro $2\lfloor \frac{n}{2} \rfloor + 2$, con indice superiore ad n, pertanto assurdo. La dimostrazione vale per tutti gli indici successivi.
- Conservazione: È possibile applicare *maxHeapRestore()* al nodo i in quanto $2i \leq 2i+1 \leq n$ sono entrambi radici di heap. Al termine dell'iterazione tutti i nodi $[1, \ldots, n]$ sono radici di heap.
- Conclusione: al termine $i = 0$, pertanto il nodo 1 è radice di uno heap.

Implementazione *heapsort* Il primo elemento contiene il massimo, viene collocato in fondo, l'elemento in fondo viene collocato in testa, si chiama *maxHeapRestore()* per ripristinare la situazione e la dimensione dello heap viene progressivamente ridotta. L'algoritmo ha complessità $\Theta(n \log n)$ in quanto *heapBuild()* ha costo $\Theta(n)$ e la chiamata di *maxHeapRestore()* costa $\Theta(\log i)$ in un heap con i elementi. Pertanto $T(n) = \sum_{i=2}^{n} \log i + \Theta(n) = \Theta(n \log n)$.

```
: heapsort(Item [] A, int n)
heapBuild(A, n)
for i = n down to 2 do
    A[1] ↔ A[i]
    maxHeapRestore(A, 1, i-1)
```

Correttezza Si dimostra la correttezza per invariante di ciclo. Al passo i il sottovettore $A[i+1, \ldots, n]$ è ordinato, $A[1, \ldots, i] \leq A[i+1, \ldots, n]$ e $A[1]$ è la radice di un vettore heap di dimensione i. COMPLETARE DIMOSTRAZIONE

9.1.4 Implementazione di code con priorità

Si vedrà l'implementazione di una min-priority queue, visto che la max-priority è speculare.

Memorizzazione

```
: PriorityItem
int priority
Item value
int pos;
```

```
: swap(PriorityItem [] H, int i, int j)
    H[i] ↔ H[j]
    H[i].pos = i
    H[j].pos = j
```

```
: PriorityQueue
    int capacity
    int dim
    PriorityItem [] H
    PriorityQueuePriorityQueue(int n)
        PriorityQueue t = new PriorityQueue
        t.capacity = n
        t.dim = 0
        t.H = new PriorityItem [1, . . . , n]
        return t
```

Inizializzazione

Inserimento

```
: PriorityItem insert(Item x, int p)
    precondition: dim < capacity
    dim += 1
    H[dim] = new PriorityItem()
    H[dim].value = x
    H[dim].priority = p
    H[dim].pos = dim
    int i = dim
    while i > 1 and H[i].priority < H[p(i)].priority do
        swap(H, i, p(i))
        i = p(i)
    return H[i]
```

minHeapRestore()

```
: minHeapRestore(PriorityItem [] A, int i, int dim)
int min = i
if l(i) ≤ dim and A[l(i)].priority < A[min].priority then
    min = l(i)
if r(i) ≤ dim and A[r(i)].priority < A[min].priority then
    min = r(i)
if i ≠ min then
    swap(A, i, min)
    minHeapRestore(A, min, dim)
```

Cancellazione e lettura minimo

```
: Item DeleteMin()
precondition: dim > 0
swap(H, 1, dim)
dim -= 1
minHeapRestore(H, 1, dim)
return H[dim+1].value
```

```
: Item min()
precondition: dim > 0
return H[1].value
```

Decremento priorità

```
: decrease(PriorityItem x, int p)
precondition: p < x.priority
x.priority = p
int i = x.pos
while i > 1 and H[i].priority < H[p(i)].priority do
    swap(H, i, p(i))
    i = p(i)
```

Complessità

Tutte le operazioni che modificano gli heap devono sistemare la proprietà heap lungo un cammino radice-foglia (*deleteMin()*) oppure lungo un cammino nodo-radice (*insert()*, *decrease()*). Essendo l'altezza $\lfloor \log n \rfloor$ il costo di tali operazioni è $O(\log n)$.

Operazione	Costo
insert()	$O(\log n)$
deleteMin()	$O(\log n)$
min()	$\Theta(1)$
decrase()	$O(\log n)$

9.2 Insiemi disgiunti - Merge-find set

In alcune applicazioni si è interessati a gestire una collezione $S = \{S_1, \ldots, S_k\}$ di insiemi dinamici disgiunti tali che:

- $\forall i, j : i \neq j \Rightarrow S_i \cap S_j = \emptyset$.
- $\bigcup_{i=1}^{k} S_i = S$.

Un esempio di questi insiemi sono le componenti di un grafo. Le operazioni fondamentali sono:

- Creare n insiemi disgiunti, ognuno composto da un unico elemento.
- *merge()*: unire più insiemi.
- *find()*: identificare l'insieme a cui appartiene un elemento.

Rappresentante

Ogni insieme è identificato da un rappresentante univoco, un qualunque membro dell'insieme. Le operazioni di ricerca del rappresentante su uno stesso insieme devono restituire sempre lo stesso oggetto. Il rappresentante può cambiare solo in caso di unione.

9.2.1 Specifica

```
: Mfset
  %Crea n componenti {1},...,{n}
  Mfset Mfset(int n)
  %Restituisce il rappresentante della componente x
  int find(int x)
  %Unisce le componenti che contengono x e y
  merge(int x, int y)
```

9.2.2 Componenti connesse dinamiche

Per trovare le componenti connesse di un grafo non orientato dinamico si inizia con componenti connesse costituite da un unico vertice, per ogni $(u, v) \in E$ si esegue *merge(u, v)* e alla fine ogni insieme disgiunto rappresenta una componente connessa.

```
: Mfset cc(Graph G)
  Mfset M = Mfset(G.n)
  foreach u ∈ G.V() do
      foreach v ∈ G.adj(u) do
          M.merge(u, v)
  return M
```

Questo algoritmo ha complessità $O(n) + m$ operazioni *merge()*, interessante per la capacità di gestire grafi dinamici.

9.2.3 Implementazione con insieme di liste

Ogni insieme viene rappresentato da una lista concatenata: il primo oggetto di una lista è il rappresentante dell'insieme e ogni elemento della lista contiene un oggetto, un puntatore all'elemento successivo e un puntatore al rappresentante.

Operazione *find(x)*

Si restituisce il rappresentante di x, l'operazione *find(x)* richiede tempo $O(1)$.

Operazione *merge(x, y)*

Si appende la lista che contiene y alla lista che contiene x modificando i puntatori ai rappresentanti nella lista appesa. Nel caso pessimo per n operazioni il costo è $O(n^2)$, il costo ammortizzato è $O(n)$.

9.2.4 Implementazione con insieme di alberi

Ogni insieme viene rappresentato da un albero, ogni nodo dell'albero contiene un oggetto e un puntatore al padre, la radice è il rappresentante dell'insieme e ha un puntatore a sè stessa.

Operazione *find(x)*

Risale la lista dei padri di x fino a trovare la radice e restituisce la radice come rappresentante. Ha costo $O(n)$ nel caso pessimo

Operazione *merge(x, y)*

Si aggancia l'albero radicato in y a y modificando il puntatore al padre di y, con costo $O(1)$.

9.2.5 Tecniche euristiche

Si dice algoritmo euristico un particolare tipo di algoritmo progettato per risolvere un problema più velocemente se i metodi classici siano troppo lenti, trovare una soluzione approssimata qualora i metodi classici falliscano nel trovare una soluzione esatta.

Liste: euristica sul peso

Si utilizza per diminuire il costo dell'operazione *merge()* si memorizza nella lista l'informazione sulla loro lunghezza e si aggancia la lista più corta a quella più lunga. La lunghezza della lista può essere mantenuta in tempo $O(1)$. Tramite analisi ammortizzata si nota come il costo di $n-1$ operazioni merge è $O(n \log n)$, pertanto il costo ammortizzato delle singole operazioni è $O(\log n)$.

Alberi: euristica sul rango

Si utilizza per diminuire il costo dell'operazione *find(x)*: ogni nodo mantiene informazioni sul proprio rango, *rank[x]* di un nodo x è il numero di archi del cammino più lungo fra x e una foglia sia discendente, ovvero l'altezza del sottoalbero associato al nodo. L'obiettivo è mantenere bassa l'altezza degli alberi. Un albero *Mfset* con radice r ottenuto tramite euristica sul rango ha almeno $2^{ranl[r]}$ nodi. Inoltre lo stesso albero con radice r e n nodi ha altezza inferiore a $\log n$.

```
: Mfset
int [] parent
int [] rank[]
Mfset Mfset(int n)
    Mfset t = new Mfset()
    t.parent = int [1...n]
    t.rank = int [1...n]
    for int i = 1 to n do
        t.parent [i] = i
        t.rank [i] = 0
    return t
```

```
: merge(int x, int y)
Item r_x = find(x)
Item r_y = find(y)
if r_x ≠ r_y then
    if rank[r_x] > rank[r_y] then
        parent[r_y] = r_x
    if rank[r_y] > rank[r_x] then
        parent[r_x] = r_y
    else
        parent[r_x] = r_y
        rank[r_y] += 1
```

Alberi: euristica di compressione dei cammini

Nell'operazione *find(x)* l'albero viene appiattito in modo che ricerche successive di x siano svolte in $O(1)$.

```
: int find(int x)
if parent[x] ≠ x then
    parent[x] = find(parent[x])
return parent[x]
```

Alberi: euristica sul rango e compressione dei cammini

Applicando entrambe le euristiche il rango non è più l'altezza del nodo ma il limite superiore alla sua altezza. Non viene calcolato il rango corretto. Il costo ammortizzato di m operazioni merge-find in un insieme di n elementi è $O(m\alpha(n))$, dove $\alpha(n)$ o funzione inversa di Ackermann cresce lentamente.

Capitolo 10

Scelta

10.1 Il problema dei cammini minimi

Definizione Dato un cammino $p =< v_1, v_2, \ldots, v_k$ con $k > 1$ il costo del cammino è dato da

$$w(p) = \sum_{i=2}^{k} w(v_{i-1}, v_i)$$

Input Un grafo orientato $G = (V, E)$, un nodo sorgente s e una funzione di peso $w : E \to \mathbb{R}$.

Output Trovare un cammino da s a u per ogni nodo $u \in V$, il cui costo sia minimo.

10.1.1 Varianti del problema

Cammini minimi da sorgente unica

Input: un grafo pesato con nodo radice s.

Output: i cammini minimi che vanno da s a tutti gli altri nodi.

Cammino minimo tra una coppia di vertici

Input: un grafo pesato e una coppia di vertici s, d.

Output: un cammino minimo fra s e d, per risolverlo si risolve il probelma dei cammini minimi da sorgente unica e si estrae il cammino richiesto in quanto non si conoscono algoritmi con tempo di esecuzione migliore.

Cammini minimi tra tutte le coppie di vertici

Input: un grafo pesato.

Output: i cammini minimi fra tutte le coppie di vertici, la soluzione si basa su programmazione dinamica.

Pesi

Alcuni algoritmi possonofunzionare unicamente con alcune categorie speciali di pesi: positivi o sia positivi che negativi e reali o interi.

10.1.2 Sottostruttura ottima

Si noti come due cammini possono avere un tratto in comune $A \rightsquigarrow B$ ma non possono convergere in un nodo comune B dopo aver percorso un tratto iniziale distinto.

Albero dei cammini minimi

L'albero dei cammini minimi è un albero di copertura radicato in s avente un cammino da s a tutti i nodi raggiungibili da s. Viene detto anche spanning tree. Dato un grafo $G = (V, E)$ non orientato e connesso, un albero di copertura di G è un sottografo $T = (V, E_T)$ tale che T è un albeero, $E_T \subseteq E$ e T contiene tutti i vertici di G.

Rappresentazione albero Per rappresentare l'albero si utilizza la rappresentazione basata sul vettore dei padri.

Soluzione ammissibile

Una soluzione ammissibile può essere descritta da un albero di copertura T radicato in s e da un vettore di distanza d, i cui valori $d[u]$ rappresentano il costo del cammino da s a u in T.

```
: void printPath(Node s, Node d, Node [] T)
if s == d then
    print s
else if T[d] == nil then
    print "error"
else
    printPath(s, T[d], T)
    print d
```

10.2 Teorema di Bellman

Enunciato Una soluzione ammissibile T è ottima se e solo se:

$$d[v] = d[u] + w(u, v) \qquad \text{per ogni arco } (u, v) \in T$$

$$d[v] \leq d[u] + w(u, v) \qquad \text{per ogni arco } (u, v) \in E$$

Dimostrazione

Se T è una soluzione ottima, allora valgono le condizioni di Bellman Sia T una soluzione ottima e $(u, v) \in E$. Se $(u, v) \in T$ allora $d[v] = d[u] + w(u, v)$. Se $(u, v) \notin T$ allora $d[v] \leq d[u] + w(u, v)$ perchè altrimenti esisterebbe nel grafo G un cammino da s a v più corto di quello in T, che è un assurdo.

Se valogono le condizioni di Bellman allora T è una soluzione ottima Si supponga per assurdo che il cammino da s a u non sia ottimo, allora esiste un albero T' in cui il cammino da s a u ha distanza $d'[u] < d[u]$ con d' il vettore delle distanze associato a T'. Essendo $d'[s] = d[s] = 0$¡ ma $d'[u] < d[u]$ esiste un arco (h, k) per cui $d'[h] \geq d[h]$ e $d'[k] < d[k]$. Si noti ora come per costruzione $d'[h] \geq d[h]$ e $d'[k] = d'[h] + w(h, k)$, e per ipotesi $d[k] \leq d[h] + w(h, k)$. Combinando queste due relazioni si ottiene

$$d'[k] = d'[h] + w(h, k) \geq d[h] + w(h, k) \geq d[k]$$

Pertanto $d'[k] \geq d[k]$, che contraddice $d'[k] < d[k]$.

10.2.1 Implementazioni

Algoritmo prototipo

: (**int** [], **int** []) `prototipoCamminiMinimi(Graph` G`, Node` s`)`

```
%Inizializza T ad una foresta di copertura composta da nodi isolati
%Inizializza d con sovrastima della distanza:  d[s] = 0,  d[x] = +∞
while ∃(u, v) : d[u] + G.w(u, v) < d[v] do
    d[v] = d[u] + w(u, v)
    %Sostituisci il padre di v in T con u
return (T, d)
```

Se al termine dell'esecuzione qualche nodo mantiene una distanza infinita non è raggiungibile.

Algoritmo generico

```
: (int [], int []) shortestPath(Graph G, Node s)
int [] d = new int [1...G.n]%d[u] è la distanza da s a u
int [] T = new int [1...G.n]%T[u] è il padre di u nell'albero T
boolean [] b = new boolean [1...G.n]%b[u] è true se u ∈ S
foreach u ∈ G.V() − {s} do
    T[u] = nil
    d[u] = +∞
    b[u] = false
T[s] = nil
d[s] = 0
b[s] = true
[...]
```

```
: (int [], int []) shortestPath(Graph G, Node s)
DataStructure S = Datastructure()
S.add(s)
while not S.isEmpty() do
    int u = S.extract()
    b[u] = false
    foreach v ∈ G.adj(u) do
        if d[u]+G.w(u, v) < d[v] then
            if not b[v] then
                S.add(v)
                b[v] = true
            else
                %Azione da svolgere nel caso v sia già presente in S
            T[v] = u
            d[v] = d[u] + G.w(u, v)
return (T, d)
```

10.3 Algoritmo di Dijkstra

Nella versione originale viene utilizzatro per trovare la distanza minima fra due nodi utilizzando il concetto di code di priorità. Funziona unicamente con pesi positivi e viene utilizzato in protocolli di rete.

10.3.1 Funzionamento

Inizializzazione

Viene creato un vettore di dimensione n in cui l'indice u rappresenta il nodo u-esimo. Le priorità vengono inizializzate a $+\infty$ e quella di s uguale a 0. Questa operazione ha costo computazionale $O(n)$.

Estrazione minimo

Si ricerca il minimo all'interno del vettore, una volta trovato si cancella la sua priorità, con costo computazionale $O(n)$.

Inserimento in coda

Si registra la priorità nella posizione v-esima con costo $O(1)$.

Aggiornamento priorità

Si aggiorna la priorità nbella posizione v-esima con costo computazionale $O(1)$.

10.3.2 Correttezza per pesi positivi

Ogni nodo viene estratto una e una sola volta e al momento dell'estrazione la sua distanza è minima. La dimostrazione si fa per induzione sul numero k di nodi estratti.

Caso base È vero in quanto $d[s] = 0$ e non ci sono lunghezze negativa.

Passo induttivo Si consideri per ipotesi induttiva che sia vero per i primi $k-1$ nodi. Quando viene estratto il k-esimo nodo u la sua distanza $d[u]$ dipende dai $k-1$ nodi già estratti e non può dipendere dai nodi ancora da estrarre in quanto hanno distanza $\geq d[u]$, pertanto $d[u]$ è minimo e u non verrà più re-inserito in quanto non ci sono distanze negative.

Implementazione

```
: (int [], int []) shortestPath(Graph G, Node s)
    PriorityQueue Q = PriorityQueue()
    Q.insert(s, 0)
    while not Q.isEmpty() do
        int u = Q.deleteMin()
        b[u] = false
        foreach v ∈ G.adj(u) do
            if d[u]+G.w(u, v) < d[v] then
                if not b[v] then
                    Q.insert(v, d[u]+G.w(u, v))
                    b[v] = true
                else
                    Q.decrease(v, D[u]+G.w(u, v))
                T[v] = u
                d[v] = d[u] + G.w(u, v)
    return (T, d)
```

Costo computazionale con coda con priorità basata su vettore

- Inizializzazione: costo $O(n)$ con una ripetizione.
- Estrazione minimo: costo $O(n)$ con $O(n)$ ripetizioni.
- Inserimento in coda: costo $O(1)$ con $O(n)$ ripetizioni.
- Aggiornamento priorità: costo $O(1)$ con $O(m)$ ripetizioni.

Per un costo totale di $O(n^2)$.

Costo computazionale con coda con priorità basata su heat binario

- Inizializzazione: costo $O(n)$ con una ripetizione.
- Estrazione minimo: costo $O(\log n)$ con $O(n)$ ripetizioni.
- Inserimento in coda: costo $O(\log n)$ con $O(n)$ ripetizioni.
- Aggiornamento priorità: costo $O(\log n)$ con $O(m)$ ripetizioni.

Per un costo totale di $O(m \log n)$.

Costo computazionale con coda con priorità basata su heap binario

- Inizializzazione: costo $O(n)$ con una ripetizione.
- Estrazione minimo: costo $O(\log n)$ con $O(n)$ ripetizioni.
- Inserimento in coda: costo $O(\log n)$ con $O(n)$ ripetizioni.
- Aggiornamento priorità: costo ammortizzato $O(1)$ con $O(m)$ ripetizioni.

Per un costo totale di $O(m + n \log n)$.

10.4 Coda Bellman-Ford, Moore

Questo algoritmo è computazionalmente più pesante di Dijkstra, ma funziona anche con archi di peso negativo.

Inizializzazione

Viene creata una coda di dimensione n con costo computazionale $O(n)$.

Estrazione

Viene estratto il prossimo elemento della coda con costo computazionale $O(1)$.

Inserimento in coda

Si inserice l'indice v in coda con costo computazionale $O(1)$. Non si deve considerare il caso in cui v sia già presente in S.

10.4.1 Correttezza

Si definisce una passata:

- Per $k = 0$ la zeresima passata consiste nell'estrazione del nodo s dalla coda S.
- Per $k > 0$ la k-esima passata consiste nell'estrazione di tutti i nodi presenti in S al termine della passata $k-1$-esima.

Al termine della passata k i vettori T e d descrivono i cammini minimi di lunghezza al più k, al termine della passata $n-1$ descrivono i cammini minimi di lunghezza al più $n-1$.

Implementazione

```
: (int [], int []) shortestPath(Graph G, Node s)
Queue Q = Queue()
Q.enqueue(s)
while not Q.isEmpty() do
    int u = Q.dequeue()
    b[u] = false
    foreach v ∈ G.adj(u) do
        if d[u]+G.w(u, v) < d[v] then
            if not b[v] then
                Q.enqueue(v)
                b[v] = true
            else
                Q.decrease(v, D[u]+G.w(u, v))
            T[v] = u
            d[v] = d[u] + G.w(u, v)
return (T, d)
```

Complessità

- Inizializzazione: costo $O(1)$ ripetuto una volta.
- Estrazione: costo $O(1)$ ripetuto $O(n^2)$ volte.
- Inserimento in coda: costo $O(1)$ ripetuto $O(nm)$ volte.

Il costo totale è pertanto $O(mn)$. Ogni nodo può essere inserito ed estratto al massimo $n-1$ volte.

10.5 Cammini minimi su DAG

I cammini minimi in un DAG sono sempre ben definiti, anche in presenza di pesi negativi in quanto non esistono cicli negativi. È possibile rilassare gli archi in ordine topologico una volta sola e non essendoci cicli non è possibile tornare su un nodo già visitato e abbassare il valore del suo campo d.

L'algoritmo utlizza pertanto l'ordinamento topologico.

```
: (int [], int []) shortestPath(Graph G, Node s)
int [] d = new int [1...G.n] %d[u] è la distanza da s a u
int [] T = new int [1...G.n] %T[u] è il padre di u nell'albero T
foreach u ∈ G.V() - {s} do
    T[u] = nil
    d[u] = +∞
T[s] = nil
d[s] = 0
Stack S = topSort(G)
while not S.isEmpty() do
    u = S.pop()
    foreach v ∈ G.adj(u) do
        if d[u] + G.w(u, v) < d[v] then
            T[v] = u
            d[v] = d[u] + G.w(u, v)
return (T, d)
```

10.6 Conclusioni

Dijkstra	$O(n^2)$	Pesi positivi, grafi densi
Johnson	$O(m \log n)$	Pesi positivi, grafi sparsi
Fredman-Tarjan	$O(m + n \log n)$	Pesi positivi, grafi densi dimensioni molto grandi
Bellman-Ford	$O(mn)$	Pesi negativi
	$O(m + n)$	DAG
DFS	$O(m + n)$	Senza pes

10.7 Cammini minimi, sorgente multipla

Input	**Complessità**	**Approccio**
Pesi positivi, grafo denso	$O(n \cdot n^2)$	Applicazione ripetuta dell'algoritmo di Dijkstra
Pesi positivi, grafo sparso	$O(n \cdot (m \log n))$	Applicazione ripetuta dell'algoritmo di Johnson
Pesi negativi	$O(n \cdot mn)$	Applicazione ripetuta di Bellman-Ford, sconsigliata
Pesi negativi, grafo denso	$O(n^3)$	Algoritmo di Floyd e Warshall
Pesi negativi, grafo sparso	$O(nm \log n)$	Algoritmo di Johnson per sorgente multipla

10.8 Floyd-Warshall

10.8.1 Cammini minimi k-vincolati

Sia k un valore in $\{0, \ldots, n\}$. Si dice che un cammino p^k_{xy} è un cammino k-vincolato fra x e y se esso ha il costo minimo fra tutti i cammini fra x e y che non passano per nessun vertice in $v_{k+1}, \ldots, v_n$ (x e y sono esclusi dal vincolo). Si assuma che esista un ordinamento tra i nodi del grafo.

10.8.2 Distanza k-vincolata

Si denota con $d^k[x][y]$ il costo totale del cammino minimo k-vincolato fra x e y se esiste.

$$d^k[x][y] = \begin{cases} w(p^k_{xy} & \textit{se esiste } p^k_{xy} \\ +\infty & \textit{altrimenti} \end{cases}$$

Si può definire ricorsivamente come:

$$d^k[x][y] = \begin{cases} w(x,y) & k = 0 \\ \min(d^{k-1}[x][y], d^{k-1}[x][k] + d^{k-1}[k][y]) & k > 0 \end{cases}$$

Matrice dei padri

Oltre a definire d si definisce la matrice T, dove $T[x][y]$ rappresenta il predecessore di y nel cammino più breve da x a y.

```
: (int [], int []) floydWarshall(Graph G)
int [] d = new int [1...G.n]
int [] T = new int [1...G.n]
foreach u, v ∈ G.V() - {s} do
    T[u][v] = nil
    d[u][v] = +∞
T[s] = nil
d[s] = 0
foreach u ∈ G.V() do
    foreach v ∈ G.adj(u) do
        d[u][v] = G.w(u, v)
        T[u][v] = u
for int k = 1 to G.n do
    foreach u ∈ G.V() do
        foreach v ∈ G.V() do
            if d[u][k] + d[k][v] < d[u][v] then
                d[u][v] = d[u][k] + d[k][u]
                T[u][v] = T[k][v]
return (T, d)
```

10.9 Chiusura transitiva (algoritmo di Warshall)

La chiusura transitiva $G* = (V, E*)$ di un grafo $G = (V, E)$ è il grafo orientato tale che $(u, v) \in E*$ se e solo se esiste un cammino da u a v in G. Supponendo di avere il grafo G rappresentato da una matrice di adiacenza M, la matrice M^n rappresenta la matrice di adiacenza di $G*$.

Formulazione ricorsiva

$$M^k[x][y] = \begin{cases} M[x][y] & k = 0 \\ M^{k-1}[x][y] \vee (M^{k-1}[x][k] \wedge M^{k-1}[k][y] & k > 0 \end{cases}$$

Capitolo 11

Risoluzione problemi

Dato un problema è possibile evidenziare quattro fasi, non necessariamente sequenziali in modo trovare una soluzione efficiente:

- Classificazione del problema.
- Caratterizzazione della soluzione.
- Tecnica di progetto.
- Utilizzo di strutture dati.

11.1 Classificazione dei problemi

Problemi decisionali

Determinare se un dato di ingresso soddisfa una certa proprietà.

Problemi di ricerca

È presente uno spazio di ricerca, un insieme di soluzioni possibili, una soluzione ammissibile, che rispetta certi vincoli.

Problemi di ottimizzazione

Ogni soluzione viene associata ad una funzione di costo, si vuole trovare la soluzione di costo minimo.

Problemi di approssimazione

Questo tipo di problemi nasce quando trovare la soluzione ottima è computazionalmente impossibile e pertanto ci si accontenta di una soluzione approssimata con costo basso, ma l'ottimalità è sconosciuta.

11.2 Definizione matematica del problema

È fondamentale definire il problema dal punto id vista matematico, spesso la formulazione è banale ma può suggerire una prima soluzione.

11.3 Tecniche di soluzione problemi

Divide-et-impera

Un problema viene suddiviso in sotto-problemi indipendenti che vengono risolti ricorsivamente (top-down) nell'ambito di problemi di decisione e ricerca.

Programmazione dinamica

La soluzione viene costruita bottom-up a partire da un insieme di sotto-problemi potenzialmente ripetuti nell'ambito dei problemi di ottimizzazione.

Memoization

Una versione top-down della programmazione dinamica.

Tecnica greedy

Approccio ingordo: si fa sempre la scelta localmente ottima.

Backtrack

Si procede per tentativi tornando quando necessario all'indietro.

Ricerca locale

La soluzione ottima viene trovata migliorando via via soluzioni esistenti.

Algoritmi probabilistici

Algoritmi in cui certe scelte avvengono in maniera casuale.

Capitolo 12

Divide-et-impera

In questo metodo di risoluzione problemi ci sono tre fasi:

- Divide: si divide il problema in sotto-problemi più piccoli e indipendenti.
- Impera: si risolvono i sotto-problemi ricorsivamente.
- Combina: si uniscono le soluzioni dei sottoproblemi.

12.1 Le torri di Hanoi

Le torri di Hanoi è un gioco matematico costituito da tre pioli con n dischi di dimensioni diverse. Inizialmente i dischi sono impilati in ordine decrescente nel piolo di sinistra. Lo scopo del gioco è impilare in ordine decrescente i dischi sul piolo di destra; senza mai impilare un disco più grande su uno più piccolo, muovendo al massimo un disco alla volta e utilizzando il piolo centrale come appoggio.

```
: hanoi(int n, int src, int middle)
if n = 1 then
    print srt → dest
else
    hanoi(n -1, src, middle, dest)
    print src → dest
    hanoi(n -1, middle, dest, src)
```

Questo algoritmo trova la soluzione ottima, la ricorrenza è

$$T(n) = 2T(n-1) + 1$$

con costo computazionale $O(2^n)$.

12.2 Quicksort

Quicksort è un algoritmo di ordinamento basato su divide-et-impera, nel caso medio è $O(n \log n)$, nel caso pessimo $O(n^2)$. Il fattore costante è migliore di quello di Mergesort, non utilizza memoria addizionale e presenta tecniche euristiche per evitare il caso pessimo.

12.2.1 Caratterizzazione

Input

Un vettore $A[1 \dots n]$, indici *start*, *end* tali che $1 \leq start \leq end \leq n$.

Divide

Si sceglie un valore $p \in A[start \dots end]$ detto perno o pivot. Si spostano gli elementi del vettore $A[start \dots end]$ in modo tale che:

- $\forall i \in [start \dots j-1] : A[i] \leq p$.
- $\forall i \in [j+1 \dots end] : A[i] \geq p$.

L'indice j viene calcolato in modo tale da rispettare tale condizione, il perno viene messo in posizione $A[j]$.

Impera

Ordina i due sottovettori $A[start \dots j-1]$ e $A[j+1 \dots end]$ richiamando ricorsivamente Quicksort.

Combina

Non fa nulla: il primo vettore, $A[j]$ e il secondo vettore formano già un vettore ordinato.

```
: int pivot(Item [] A, int start, int end)
    Item p = A[start]
    int j = start  for int i = start +1 to end
     do
        if A[i] < p then
            j += 1
            A[i] ↔ A[j]
    A[start] = A[j]
    A[j] = p
    return j
```

```
: Quicksort(Item [] A, int start, int end)
    if start < end then
        int j = pivot(A, start, end)
        Quicksort(A, start, j -1)
        Quicksort(A, j+1, end)
```

12.2.2 Complessità computazionale

Il costo di *pivot()* è $\Theta(n)$, mentre il costo di *Quicksort()* dipende dal partizionamento:

- Partizionamento peggiore: dato un vettore di dimensione n viene diviso i due sotto-problemi di dimensione 0 e $n-1$.

$$T(n) = T(n-1) + T(0) + \Theta(n) = \Theta(n^2)$$

- Partizionamento migliore: dato un vettore di dimensione n viene sempre diviso in due sotto-problemi di dimensione $\frac{n}{2}$.

$$T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n)$$

Nel caso medio si alternano partizionamenti peggiori e migliori con il caso medio nel numero maggiore, pertanto nel caso medio ha complessità $O(n \log n)$.

12.3 Conclusioni

Si deve applicare divide-et-impera quando i passi "divide" e "combina" sono semplici e i costi sono migliori del corrispondente algoritmo iterativo. Presenta inoltre una più facile parallelizzazione e un utilizzo ottimale della cache ("cache oblivious").

Capitolo 13

Programmazione dinamica

Si dice programmazione dinamica un metodo per dividere un problema ricorsivamente in sottoproblemi in modo che ogni sottoproblema venga rivolto una volta sola e la sua soluzione venga memorizzata in una tabella. Se un sottoproblema deve essere risolto nuovamente si ottiene la sua soluzione dalla tabella (lookup in $O(1)$).

13.1 Approccio generale

Se si parte da un problema di ottimizzazione si definisce la soluzione in maniera ricorsiva, successivamente si definisce il valore della soluzione in maniera ricorsiva (primo passo per problemi di conteggio). Se da qui non sono presenti sottoproblemi ripetuti si utilizza divide et impera; se sono presenti sottoproblemi ripetuti e bisogna risolverli tutti si utilizza la programmazione dinamica, se non devono essere risolti tutti si usa memoization. Dalla tabella delle soluzioni si ottiene l'output numerico o la ricostruzione della soluzione che permette di arrivare alla soluzione ottima.

13.1.1 Evitare di risolvere i problemi più di una volta

Quando si risolve un problema si memorizza il risultato ottenuto in una tabella DP, che può essere un vettore, una matrice o un dizionario. La tabella deve contenere un elemento per ogni sottoproblema che deve essere risolto.

- Casi base: si memorizzano i casi base direttamente nella tabella.
- Iterazione bottom-up: si parte dai sottoproblemi che possono essere risolti direttamente a partire dai casi base, si sale verso problemi sempre più grandi fino a raggiungere il problema originale.

13.1.2 Ricostruire la soluzione

Per ricostruire la soluzione si parte dalla definizione ricorsiva del problema, la dimensione di n indica l'indice sulla tabella che si deve controllare e si definisce un controllo su quale delle chiamate ricorsive viene realmente effettuata.

13.2 Memoization

Questa tecnica fonde l'approccio di memorizzazione della programmazione dinamica con quello top-down di divide-et-impera: si crea una tabella DP inizializzata con un valore speciale ad indicare che un certo sottoproblema non è ancora stato risolto. Ogni volta che si deve risolvere un sottoproblema si controlla nella tabella se è già stato risolto precedentemente:

- già risolto: si utilizza il risultato della tabella.
- non risolto: si calcola il risultato e lo si memorizza

In questo modo ogni sottoproblema viene calcolato una sola volta e memorizzato come nella versione bottom-up. Questa tecnica presenta vantaggi quando invece di utilizzare una tabella si utilizza un dizionario in quanto non è più necessario fare inizializzazione e il costo di esecuzione passa da $O(nC)$ a $O(\min(2^n, nC)$.

Implementazione 13.1: Memoization automatica in Python

```
1  from functools import wraps
2
3  def memo(func):
4      cache = {}
5      @wraps(func)
6      def wrap(*args):
7          if args not in cache:
8              cache[args] = func(*args)
9          return cache[args]
10     return wrap
```

13.3 Problemi

13.3.1 Domino lineare

Il gioco del domino è basato su tessere di dimensione 2×1. Si scriva un algoritmo efficiente che prenda in input un intero n e restituisca il numero di possibili disposizioni n di tessere in un rettangolo $2 \times n$.

Definizione ricorrenza

Si definisca una formula ricorsiva $DP[n]$ che permetta di calcolare il numero di disposizioni possibili quando si hanno n tessere. Con $n = 0$ esiste una sola disposizione possibile, con $n = 1$ esiste ancora una sola disposizione possibile. Mettendo una tessera in verticale si risolve il problema di dimensione $n-1$, con una in orizzontale ne devo mettere due e risolvo il problema di dimensione $n-2$. Essendo un problema di conteggio queste due possibilità si sommano insieme:

$$DP[n] = \begin{cases} 1 & n \leq 1 \\ DP[n-2] + DP[n-1] & n > 1 \end{cases}$$

Si noti come $DP[n]$ è pari al $n+1$-esimo numero della serie di Fibonacci.

Algoritmo ricorsivo

```
: int domino1(int n)
  if n ≤ 1 then
  |  return 1
  else
  |  return domino1(n - 1) + domino1(n - 2)
```

Complessità L'equazione di ricorrenza associata è:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n-2) + T(n-1) + 1 & n > 1 \end{cases}$$

Che utilizzando il master theorem per la ricorrenza lineare di ordine costante si ottiene una complessità pari a $T(n) = \Theta(2^n)$.

Algoritmo iterativo

```
: int domino2(int n)
  int [] DP = new int [0...n]
  DP[0] = DP[1] = 1
  for int i = 2 to n do
  |  DP[i] = DP[i - 1] + DP[i-2]
  return DP[n]
```

Complessità Ha complessità temporale $T(n) = \Theta(n)$ e spaziale $S(n) = \Theta(n)$.

Miglioramento Si può rendere con complessità spaziale costante $S(n) = \Theta(n)$.

```
: int domino3(int n)
  int DP_0 = 1
  int DP_1 = 1
  int DP_2 = 1
  for int i = 2 to n do
  |  DP_0 = DP_1
  |  DP_1 = DP_2
  |  DP_2 = DP_0 + DP_1
  return DP[n]
```

13.3.2 Hateville

Hateville è un villaggio composto da n case numerate da 1 a n lungo una singola strada. Ad Hateville ognuno odia i propri vicini da entrambi i lati. Si vuole organizzare una sagra e si rende necessario raccogliere i fondi. Ogni abitante ha intenzione di donare una quantità $D[i]$, ma non

intende partecipare ad una raccolta fondi a cui partecipano uno o entrambi i propri vicini. Si scriva un algoritmo che restituisca la quantità massima di fondi che può essere raccolta e un algoritmo che restituisca il sottoinsieme di indici $S \subseteq \{1, \ldots, n\}$ tale per cui la donazione totale $T = \sum_{i \in S} D[i]$ è massimale.

Definizione ricorsiva

Sia $HV(i)$ uno di possibili insiemi di indici da selezionare per ottenere una donazione ottimale delle prime i case di Hateville numerate $1 \ldots n$. $HV(n)$ è la soluzione del problema originale. Considerando vicino i-esimo si deve considerare se non accettare la sua donazione ($HV(i) = HV(i-1)$) o accettarla: $HV(i) = \{i\} \cup HV(i-2)$ e prendere il risultato massimo.

Sottostruttura ottima

Sia $HV_p(i)$ il problema dato dalle prime i case e $HV_s(i)$ una soluzione ottima per il problema $HV_p(i)$ e $|HV_s(i)|$ il totale di donazioni di $HV_s(i)$ ne consegue:

- Se $i \notin HV_s(i)$ allora $HV_s(i) = HV_s(i-1)$.
- Se $i \in HV_s(i)$ allora $HV_s(i) = HV_s(i-2) \cup \{i\}$.

Dimostrazione

$\mathbf{i \notin HV_s(i)}$ $HV_s(i)$ è una soluzione ottima anche per $HV_p(i-1)$, se così non fosse esisterebbe una soluzione $HV'_s(i-1)$ per il problema $HV_p(i)$ tale che $|HV'_s(i-1)| > |HV_s(i)|$, ma allora $HV'_S(i-1)$ sarebbe una soluzione per $HV_p(i)$ tale che $HV'_s(i-1)| > |HV_s(i)|$, assurdo.

$\mathbf{i \in HV_s(i)}$ $i-1 \notin HV_s(i)$, altrimenti non sarebbe una soluzione ammissibile, pertanto $HV_s(i) - \{i\}$ è una soluzione ottima per $HV_p(i-2)$ per il problema $HV_p(i-2)$ tale che $|HV'_s(i-2)| > |HV_s(i) - \{i\}|$, allora $HV'_s(i-2) \cap \{i\}$ sarebbe una soluzione per $HV_p(i)$ tale che $|HV'_s(i-2) \cup \{i\}| > |HV_s(i)|$, assurdo.

Determinare la ricorsione

$$HV(i) = \begin{cases} \emptyset & i = 0 \\ \{1\} & i = 1 \\ highest(HV(i-1), HV(i-2) \cup \{i\}) & i \geq 2 \end{cases}$$

Tabella DP Sia $DP[i]$ il valore della massima quantià di donazioni che si possono ottenere dalle prime i case di Hateville. $DP[n]$ è il valore della soluzione ottima.

$$DP[i] = \begin{cases} 0 & i = 0 \\ D[1] & i = 1 \\ \max(DP[i-1], DP[i-2] + D[i]) & i \geq 2 \end{cases}$$

Algoritmo iterativo

```
: int hateville(int [] D, int n)
int [] DP = new int [0 . . . n]
DP[0] = 0
DP[1] = D[1]
for int i = 2 to n do
    DP[i] = max(DP[i-1], DP[i-2]+D[i])
return DP[n]
```

Ricostruire la soluzione originale

Considerando l'elemento $DP[i]$ il suo valore può essere derivato considerando:

- Se $DP[i] = DP[i-1]$ la casa i non è stata selezionata.
- Se $DP[i] = DP[i-2] + D[i]$ la casa i è stata selezionata.

Si utilizza questa informazione per ricostruire la soluzione in modo ricorsivo: per costruire la soluzione fino a i si ricostruisce la soluzione fino a $i-2$ e si aggiunge i o si ricostruisce la soluzione fino a $i-1$ senza aggiungere nulla.

```
: Set solution(int [] DP, int [] D, int i)
if i == 0 then
    return ∅
else if i == 1 then
    return {1}
else if DP[i] == DP[i-1] then
    return solution(DP, D, i - 1)
else
    Set sol = solution (DP, D, i-2)
    sol.insert(i)
    return sol
```

Complessità computazionale

La complessità computazionale di *solution()* è $T(n) = \Theta(n)$, quella computazionale e spaziale di *hateville()* è $T(n) = \Theta(n)$ e $S(n) = \Theta(n)$. Questa soluzione non si può migliorare se si vuole ricostruire la soluzione.

13.3.3 Knapsack

Dato un insieme di oggetti, ognuno caratterizzato da un peso e da un profitto e uno zaino con un limite di capacità trovare un sottoinsieme di oggetti il cui peso sia inferiore alla capacità dello zaino e il profitto sia massimale, ovvero maggiore o uguale di qualunque altro sottoinsieme di oggetti.

Caratterizzazione

Input

- Un vettore w, dove $w[i]$ è il peso dell'oggetto i-esimo.
- Un vettore p, dove $p[i]$ è il profitto dell'oggetto i-esimo.
- La capacità C dello zaino.

Output Un insieme $S \subseteq \{1, \ldots, n\}$ tale che il volume totale deve essere minore uguale alla capacità: $w(S) \sum_{i \in S} w[i] < C$ e il profitto totale sia massimizzato: $p(S) = \sum_{i \in S} p[i]$.

Definizione matematica del valore della soluzione

Dato uno zaino di capacità C e n oggetti caratterizzati da peso w e profitto p si definisce $DP[i][c]$ il massimo profitto che può essere ottenuta dai primi $i \leq n$ oggetti contenuti in uno zaino di capacità $c \leq C$. Il massimo profitto ottenibile dal problema originale è rappresentato da $DP[n][C]$.

Parte ricorsiva Considerando l'ultimo oggetto lo si può non prendere: $DP[i][c] = DP[i-1][c]$, ovvero la capacità non cambia e non c'è profitto; se invece lo prendo $DP[i][c] = DP[i-1][c-w[i]] + p[i]$, ovvero si sottrae il peso alla capacità e si aggiunge il profitto relativo. La scelta della soluzione migliore è:

$$DP[i][c] = \max(DP[i-1][c], DP[i-1][c-w[i]] + p[i])$$

Casi base Il profitto massimo in caso di assenza di oggetti o di capacità è 0, se invece si ha capacità negativa vale $-\infty$.

Formula completa

$$DP[i][c] = \begin{cases} 0 & i = 0 \vee c = 0 \\ -\infty & c < 0 \\ \max(DP[i-1][c], DP[i-1][c-w[i]] + p[i]) & altrimenti \end{cases}$$

Pseudocodice

```
: int knapsack(int [] w, int [] p, int n, int C)
int [][] DP = new int [0...n][0...C]
for int i = 0 to n do
    DP[i][0] = 0
for int c = 0 to C do
    DP[0][c] = 0
for int i = 1 to n do
    for int c = 1 to C do
        if w[i] ¡ C then
            DP[i][c] = max(DP[i - 1][c - w[i]] + p[i], DP[i - 1][c])
        else
            DP[i][c] = DP[i - 1][c]
return DP[n][C]
```

Complessità computazionale

La complessità della funzione *knapsack()* è $T(n) = O(nC)$, una complessità pseudo-polinomiale in quanto sono necessari $k = \log C$ bit per rappresentare C e pertanto la complessità è $T(n) = O(n2^k)$.

Implementazione con memoization

Si nota come non tutti gli elementi della matrice sono necessari alla soluzione del problema. Risulta vantaggioso pertanto utilizzare la tecnica di memoization, in cui la tabella viene inizializzata esternamente in una funzione wrapper e il valore -1 viene utilizzato per identificare il valore di una cella non calcolata.

: Knapsack con memoization

```
int knapsack(int [] w, int [] p, int n, int C)
    int [][] DP = new int [1...n][1...C]
    for int i = 1 to n do
        for int c = 1 to C do
            DP[i][c] = -1
    return knapsackRec(w, p, n, C, DP)
int knapsackRec(int [] w, int [] p, int i, int c, int [][] DP)
    if c < 0 then
        return −∞
    else if i == 0 or c == 0 then
        return 0
    else
        if DP[i][c] < 0 then
            int nottaken = knapsackRec(w, p, i - 1, c, DP)
            int taken = knapsackRec(w, p, i - 1, c - w[i], DP) + p[i]
            DP[i][c] = max(nottaken, taken)
        return DP[i][c]
```

13.3.4 Knapsack senza limiti

Dato uno zaino di capacità C e n oggetti caratterizzati da peso w e profitto p si definisce $DP[i][c]$ come il massimo profitto che può essere ottenuto dai primi $i \leq n$ oggetti contenuti in uno zaino di capacità $c \leq C$ senza porre limiti al numero di volte che un oggetto può essere selezionato. Si modifica pertanto la funzione ricorsiva come:

$$DP[i][c] = \begin{cases} 0 & i = 0 \vee c = 0 \\ -\infty & c < 0 \\ \max(DP[i][c], DP[i-1][c - w[i]] + p[i]) & altrimenti \end{cases}$$

In questo possibile è possibile semplificare la soluzione riducendo lo spazio occupato: si definisce $DP[c]$ il massimo profitto che può essere ottenuto da tali oggetti in uno zaino di capacità $c \leq C$.

$$DP[c] = \begin{cases} 0 & c = 0 \\ \max\limits_{w[i]<c} \{DP[c - w[i]] + p[i]\} & c > 0 \end{cases}$$

Implementazione tramite memoization

: Knapsack senza limiti con memoization

```
int knapsack(int [] w, int [] p, int n, int C)
    int [] DP = new int [0...C]
    for int c = 0 to C do
        DP[i] = -1
    knapsackRec(w, p, n, C, DP)
    return DP[C]
int knapsackRec(int [] w, int [] p, int n, int c, int [][] DP)
    if c == 0 then
        return 0
    else if DP[c] < 0 then
        DP[c] = 0
        for int i = 1 to n do
            if w[i] ≤ c then
                int val = knapsackRec(w, p, n, c - w[i], DP) + p[i]
                DP[c] = max(DP[c], val)
    return DP[c]
```

Attraverso questo algoritmo non è detto che tutti gli elementi debbano essere riempiti e la complessità in spazio è $\Theta(C)$.

Ricostruire la soluzione

Questo approccio rende più difficile ricostruire la soluzione: si possono ispezionare tutti gli elementi per capire da dove deriva il massimo, ma risulta più semplice memorizzare l'indice da cui deriva il massimo.

Implementazione tramite memoization

: Knapsack senza limiti con memoization

```
int knapsack(int [] w, int [] p, int n, int C)
    int [] DP = new int [0...C]
    int [] pos = new int [0...C]
    for int c = 0 to C do
        DP[i] = -1
        pos[i] = -1
    knapsackRec(w, p, n, C, DP, pos)
    return solution(w, C, pos)
int knapsackRec(int [] w, int [] p, int n, int c, int [][] DP)
    if c == 0 then
        return 0
    else if DP[c] < 0 then
        DP[c] = 0
        for int i = 1 to n do
            if w[i] ≤ c then
                int val = knapsackRec(w, p, n, c - w[i], DP) + p[i]
                if val ≥ DP[c] then
                    DP[c] = val
                    pos[c] = i
    return DP[c]
List solution(int [] w, int c, int [] pos)
    if c == 0 or pos[c] > 0 then
        return List()
    else
        List L = solution(w, c - w[pos[c]], pos)
        L.insert(L.head(), pos[c])
        return L
```

Restituisce una lista di indici selezionati, se $c = 0$ lo zaino è stato riempito perfettamente, se $pos[c] < 0$ lo zaino non può essere riempito interamente.

13.3.5 Sottosequenza comune massimale

Sottosequenza Si definisce una sequenza P come sottosequenza di T se P è ottenuto da T rimuovendo uno o più dei suoi elementi o come il sottoinsieme di indici $\{1, \ldots, n\}$ degli elementi di T che compaiono anche in P. I rimanenti elementi sono indicati in ordine senza essere necessariamente contigui. La sottosequenza vuota $\emptyset$ è sottosequenza di qualsiasi altra sottosequenza.

Sottosequenza comune Una sottosequenza X è detta sottosequenza comune di due sequenze T e U se è sottosequenza sia di T che di U: $X \in \mathcal{CS}(T, U)$.

Sottosequenza comune massimale Una sequenza $X \in \mathcal{CS}(T,U)$ è una sottosequenza comune massimale di due sequenze T e U se non esiste altra sottosequenza comune $Y \in \mathcal{CS}(T,U)$ tale che Y sia più lunga di X, si scrive: $X \in \mathcal{LCS}(T,U)$.

Definizione del problema

Date due sequenze T e U si trovi la più lunga sottosequenza comune di T e U.

Soluzione di forza bruta

```
: int LCS(Item [] T, Item [] U)
Item [] maxsofar = nil
foreach subsequence X of T do
    if X is subsequence of U then
        if len(X) > len(maxsofar) then
            maxsofar = X
return maxsofar
```

Si noti come data una sottosequenza lunga n, le sue sottosequenze sono 2^n e verificare che una sequenza è sottosequenza di un'altra sottosequenza è $O(m+n)$, la complessità computazionale dell'algoritmo sopra è pertanto $\Theta(2^n(m+n))$.

Descrizione matematica della soluzione ottima

Data una sequenza T composta da caratteri $t_1 \dots t_n$, $T(i)$ denota il prefisso di T dato dai primi caratteri. L'obiettivo è, date due sequenze T e U di lunghezza n e m, di scrivere una formula ricorsiva $LCS(T(i),U(i))$ che restituisa la LCS dei prefissi $T(i)$ e $U(i)$.

Casi ricorsivi

Primo caso Si considerino due prefissi $T(i)$ e $U(j)$ tali per cui l'ultimo carattere coincide, $t_i = u_j$ allora:

$$LCS(T(i),U(j)) = LCS(T(i-1),U(j-1)) \oplus t_i$$

Secondo caso Si considerino due prefissi $T(i)$ e $U(j)$ tali per cui l'ultimo carattere non coincide, $t_i \neq u_j$ allora:

$$LCS(T(i),U(j)) = longest(LCS(T(i-1),U(j)), LCS(T(i),U(j-1)))$$

Casi base La sottosequenza più lunga quando una delle due sequenze è vuota, ovvero $i = 0$ o $j = 0$ è la sequenza vuota.

Soluzione completa

$$LCS(T(i),U(j)) = \begin{cases} \emptyset & i = 0 \vee j = 0 \\ LCS(T(i-1),U(j-1)) \oplus t_i & i > 0 \wedge j > 0 \wedge t_i = u_j \\ longest(LCS(T(i-1),U(j)), LCS(T(i),U(j-1))) & i > 0 \wedge j > 0 \wedge t_i \neq u_j \end{cases}$$

Sottostruttura ottima

Date le due sequenze $T = (t_1, \dots t_n)$ e $U = (u_1, \dots, u_m)$ e $X = (x_1, \dots x_k)$ una LCS di T e U. Sono dati tre casi:

1. $t_n = u_m \quad \Rightarrow \quad x_k = t_n = u_m \wedge X(k-1) \in \mathcal{LCS}(T(n-1), U(m-1))$
2. $t_n \neq u_m \wedge x_k \neq t_n \quad \Rightarrow \quad X \in \mathcal{LCS}(T(n-1), U(m))$
3. $t_n \neq u_m \wedge x_k \neq u_m \quad \Rightarrow \quad X \in \mathcal{LCS}(T(n), U(m-1))$

Dimostrazione primo caso

Prima parte

$$t_n = u_m \Rightarrow x_k = t_n = u_m$$

Si consideri per assurdo $x_k \neq t_n = u_m$. Si consideri ora $Y = X \oplus t_n$, allora $Y \in \mathcal{CS}(T, U)$ e $|Y| > |X|$, che è una contraddizione.

Seconda parte

$$t_n = u_m \Rightarrow X(k-1) \in \mathcal{LCS}(T(n-1), U(m-1))$$

Si consideri per assurdo $X(k-1) \notin \mathcal{LCS}(T(n-1), U(m-1))$, allora $\exists Y \in \mathcal{LCS}(T(n-1), U(m-1))$ tale che $|Y| > |X(k-1)|$, pertanto $Y \oplus t_n \in \mathcal{CS}(T, U)$ e $|Y \oplus t_n| > |X(k-1) \oplus t_n| = X$, che è una contraddizione.

Seconda parte (terza simmetrica)

$$t_n \neq u_m \wedge x_k \neq t_n \Rightarrow X \in \mathcal{LCS}(T(n-1), U(m))$$

Si consideri per assurdo $X \notin \mathcal{LCS}(T(n-1), U(m))$, allora $\exists Y \in \mathcal{LCS}(T(n-1), U(m))$ tale che $|Y| > |X|$, pertanto è anche vero che $Y \in \mathcal{LCS}(T, U)$, pertanto X non è $\mathcal{LCS}(T, U)$, che è un assurdo.

Ricorrenza per il valore della soluzione ottimale

Date due sequenze T e U di lunghezza n e m si scriva una formula ricorsiva $DP[i][j]$ che restituisca la lunghezza della $\mathcal{LCS}$ dei prefissi $T(i)$ e $U(j)$.

$$DP[i][j] = \begin{cases} 0 & i = 0 \vee j = 0 \\ DP[i-1][j-1] + 1 & i > 0 \wedge j > 0 land t_i = u_j \\ \max(DP[i-1][j], DP[i][j-1]) & i > 0 \wedge j > 0 land t_i \neq u_j \end{cases}$$

$DP[n][m]$ contiene la lunghezza della $\mathcal{LCS}$ del probema originale.

```
: int LCS(Item [] T, Item [] U, int n, int m)
    int [][] DP = new int [0...n][0...m]
    for int i = 0 to n do
        DP[0][n] = DP[n][0] = 0
    for int i = 1 to n do
        for int j = 1 to m do
            if T[i] == U[j] then
                DP[i][j] = DP[i - 1][j - 1] +1
            else
                DP[i][j] = max(DP[i - 1][j], DP[i][j - 1])
    return DP[n][m]
```

Ricostrurie la sottosequenza comune

```
:
    Item []int LCS(Item [] T, Item [] U, int n, int m)
        ...
        return subsequence(DP, T, U, n, m)
    Item []subsequence(int [][] DP, Item [] T, Item [] U, int i, int j)
        if i == 0 or j == 0 then
            return List()
        else if T[i] == U[j] then
            List S = subsequence(DP, T, U, i - 1, j - 1)
            S.insert(S.head(), T[i])
            return S
        else
            if DP[i - 1][j] ¿ DP[i][j - 1] then
                return subsequence(DP, T, U, i - 1, j)
            else
                return subsequence(DP, T, U, i, j - 1)
```

La complessità computazionale di *subsequence()* è $O(m+n)$, mentre quella di LCS è $O(mn)$.

13.3.6 String matching approssimato

Definizione

Si dice un'occorrenza k-approssimata di P in T, dove $P = p_1 \dots p_m$ è una stringa detta pattern, $T = t_1 \dots t_n$ è una stringa detta testo con $m \leq n$, una copia di P in T in cui sono ammessi k differenze tra caratteri di P e caratteri di T del tipo:

- I corrispondenti caratteri in P, T sono diversi (sostituzione).
- Un carattere in P non è incluso in T (inserimento).
- Un carattere in T non è incluso in P (cancellazione).

Problema

Il problema è trovare un'occorrenza k-approssimata di P in T con k minimo ($0 \leq k \leq m$).

Sottostruttura ottima

Sia $DP[0 \dots m][0 \dots n]$ una tabella di programmazione dinamica tale che $DP[i][j]$ sia il minimo valore k per cui esiste un'occorrenza k-approssimata di $P(i)$ in $T(j)$ che termina nella posizione j. Si hanno quattro possibilità:

- $DP[i-1][j-1]$ se $P[i] = T[j]$.
- $DP[i-1][j-1]+1$ se $P[i] \neq T[j]$ per la sostituzione.
- $DP[i-1][j]+1$ per l'inserimento.
- $DP[i][j-1]+1$ per la cancellazione.

$$DP[i][j] = \begin{cases} 0 & i = 0 \\ i & j = 0 \\ \min\{\ DP[i-1][j-1] + \delta, \quad \delta = iif(P[i] = T[j], 0, 1) & \\ \qquad DP[i-1][j] + 1, & altrimenti \\ \qquad DP[i][j-1] + 1\} & \end{cases}$$

Ricostruzione della soluzione finale

$DP[m][j] = k$ se e solo se esiste un'occorrenza k-approssimata di P in $T(j)$ che termina nella posizione j. La soluzione del problema è data dal più piccolo falore $DP[m][j]$ per $0 \leq j \leq n$. Si noti pertanto come la soluzione del problema non si trova nella casella "in basso a destra" ma vada essa stessa ricercata nella tabella DP.

Algoritmo

```
: int stringMatching(Item [] P, Item [] T, int m, int n)
int [][] DP = new int [0...m][0...n]
for int j = 0 to n do
    DP[0][j] = 0
for int i = 1 to m do
    DP[i][0] = i
for int i = 1 to m do
    for int j = 1 to n do
        DP[i][j] = min(DP[i-1][j-1] + iif(P[i] == T[j], 0, 1), DP[i-1][j]+1, DP[i][j-1]+1)
int pos = 0; for int j = 1 to n do
    if DP[m][j] < DP[m][pos] then
        pos = j
return pos
```

13.3.7 Prodotto di catena di matrici

Problema

Data una sequenza n di matrici $A_1, \ldots, A_n$ compatibili due a due al prodotto, si vuole calcolare il loro prodotto. SI noti come il prodotto di matrici non sia commutativo ma associativo. Notando come il prodotto di matrici si basa sulla moltiplicazione scalare come operazione elementare si vuole calcolare il prodotto delle n matrici impegando il minor numero possibile di moltiplicazioni scalari.

Parentesizzazione

La parentesizzazione $P_{i,j}$ del prodotto $A_i \dot{A}_{i+1} \cdots A_j$ è la matrice A_i se $i = j$, nel prodotto di due parentesizzazioni $(P_{i,k} \cdot P_{k+1,j})$ altrimenti.

Parentesizzazione ottima La parentesizzazione che richiede il minor numero di moltiplicazioni scalari per essere completata fra tutte le parentesizzazioni possibili si dice ottima. Vale la pena preprocessare i dati per cercare la parentesizzazione migliore per risparmiare tempo nel calcolo vero e proprio. Detto $P(n)$ il numero di parentesizzazioni per n matrici $A_1 \cdot \cdots \cdot A_n$ l'ultimo prodotto può occorrere in $n-1$ posizioni diverse. Pertanto fissato l'indice k dell'ultimo prodotto si hanno: $P(k)$ parentesizzazioni per $A_1 \cdot \cdots \cdot A_k$ e $P(n-k)$ parentesizzazioni per $A_{k+1} \cdot \cdots \cdot A_n$. Pertanto.

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{i=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases}$$

Numero di Catalan

$$P(n+1) = C(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} = \Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$$

In matematica $C(n)$ indica il numero di modi in cui un poligono convesso con $n+2$ lati può essre suddiviso in triangoli. Si può mostrare facilmente come $P(n) = \Omega(2^n)$.

Lista di definizioni matematiche

- $A_1 \cdot \cdots \cdot A_n$ è il prodotto di n matrici da ottimizzare.
- c_{i-1} è il numero di righe della matrice A_i.
- c_i è il numero di colonne della matrice A_i.
- $A[i \ldots j]$ è il sottoprodotto $A_i \cdot A_{i+1} \cdot \cdots \cdot A_j$.
- $P[i \ldots j]$ è una parentesizzazione per $A[i \ldots j]$, non necessariamente ottima.

Struttura di una parentesizzazione ottima

Sia $A[i \ldots j]$ una sottosequenza del prodotto di matrici e si consideri una parentesizzazione ottima $P[i \ldots j]$ di $A[i \ldots j]$. Esiste un ultimo prodotto: un indice k tale che $P[i \ldots j] = P[i \ldots k] \cdot P[k+1 \ldots j]$.

Teorema della sottostruttura ottima

Enunciato Se $P[i \dots j] = P[i \dots k] \cdot P[k+1 \dots j]$ è una parntesizzazione ottima del prodotto $A[i \dots j]$ allora $P[i \dots k]$ è una parentesizzazione ottima del prodotto $A[i \dots k]$ e $P[k+1 \dots j]$ è una parentesizzazione ottima del prodotto $A[k+1 \dots j]$.

Dimostrazione Si supponga che esista una parentesizzazione ottima $P'[i \dots k]$ di $A[i \dots k]$ con costo inferiore a $P[i \dots k]$. Allora $P'[i \dots k] \cdot P[k+1 \dots j]$ sarebbe una parentesizzazione di $A[i \dots j]$ con costo inferiore a $P[i \dots j]$, che è un assurdo.

Valore della soluzione ottima

Sia $DP[i][j]$ il minimo numero di moltiplicazioni scalari necessarie per calcolare il prodotto $A[i \dots j]$.

- Caso base $i = j$. Allora $DP[i][j] = 0$.
- Passo ricorsivo: $i < j$ esiste una parentesizzazione ottima

  $$P[i \dots j] = P[i \dots k] \cdot P[k+1 \dots j]$$

  Sfruttando la ricorsione

  $$DP[i][j] = DP[i][k] + DP[k+1][j] + c_{i-1} \cdot c_k \cdot c_j$$

Considerando $c_{i-1} \cdot c_k \cdot c_j$ il costo per moltiplicare la matrice $A_i \cdot \dots \cdot A_k$: c_{i-1} righe, c_k colonne e la matrice $A_{k+1} \cdot \dots \cdot A_j$: c_k righe, c_j colonne. Non si conosce il valore di k si devono provare tutti i valori che può assumere, tra i e $j-1$:

$$DP[i][j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \{DP[i][k] + DP[k+1][j] + c_{i-1} \cdot c_k \cdot c_j\} & i < j \end{cases}$$

Implementazione

Input L'algoritmo riceve in input un vettore $c[0 \dots n]$ contenente le dimensioni delle matrici, dove $c[0]$ è il numero di righe della prima matrice. $c[i-1]$ è il numero di righe della matrice A_i e $c[i]$ è il numero di colonne della matrice A_i e due indici i e j che rappresentano l'intervallo di matrici da moltiplicare.

Output Il numero di moltiplicazioni scalari per calcolare il prodotto delle matrici comprese tra gli indici i e j.

Approccio ricorsivo

```
: int recPar(Item [] c, int i, int j)
if i == j then
    return 0
else
    int min = +∞
    for int k = i to j - 1 do
        int q = recPar (c, i, k) + recPar (c, k + 1, j) +c[i-1]·c[k]·c[j]
        if q < min then
            min = q
    return min
```

Valutazione La soluzione ricorsiva top-down è $\Omega(2^n)$. Il problema è che i sottoproblemi che sono $\frac{n(n+1)}{2}$ vengono risolti più volte.

Versione bottom-up Si creino due tabelle di programmazione dinamica, due matrici DP e $last$ di dimensione $n \times n$ tali che $DP[i][j]$ contiene il numero di moltiplicazioni scalari necessarie per moltiplicare le matrici $A[i \dots j]$ e $last[i][j]$ contiene il valore k dell'ultimo prodotto che minimizza il costo del sottoproblema.

```
: int computePar(Item // c, int n)
    int [][] DP = new int [1...n][1...n]
    int [][] last = new int [1...n][1...n]
    for int i = 1 to n do
        DP[i][i] = 0
    for int h = 2 to n do
        for int i = 1 to n - h + 1 do
            int j = i + h - 1
            DP[i][j] = +∞
            for int j = i to j - 1 do
                int temp = DP[i][k] + DP[k+1][j] + c[i - 1]·c[k]·c[j]
                if temp < DP[i][j] then
                    DP[i][j] = temp
                    last[i][j] = k
    return DP[1][n]
```

Valutazioni Il costo computazionale è $O(n^3)$, in quanto ogni cella richiede tempo $O(n)$ per essere riempita. Il costo della funzione si trova nella posizione $DP[1][n]$. È anche necessario motrare la

soluzione trovata, motivo per cui si salvano le informazioni nella matrice last.

: Ricostruzione della soluzione - Stampa

```
void computePar(int [] c, int n)
    [...]
    printPar(last, 1, n)
void printPar(int [][] last, int i, int
    if i == j then
        print "A["
        print i
        print "]"
    else
        print "("
        printPar(last, i, last[i][j])
        print "·"
        printPar(last, last[i][j]+1, j)
        print ")"
```

: Ricostruzione della soluzione - Calcolo Effettivo

```
int [][]Multiply(matrix [] A, int [][] S, int i, int j)
    if i == j then
        return A[i]
    else
        int [][] X = Multiply(S, i, S[i][j])
        int [][] Y = Multiply(S, S[i][j]+1, j)
        return matrix-multiplication(X, Y)
```

13.3.8 Insieme indipendente di intervalli pesati

Problema

So deve trovare un insieme indipendente di peso massimo, un sottoinsieme di intervalli disgiunti tra loro ($b_j \leq a_i$ o $b_i \leq a_j$ $\forall i \neq j$) tale che la somma dei loro profitti sia la più grande possibile.

Input Siano dati n intervalli distinti $[a_1, b_i[, \ldots, [a_n, b_n[$ della retta reale aperti a destra, dove all'intervallo i è associato un profitto w_i con $1 \leq i \leq n$.

Pre-elaborazione

Per usare la programmazione dinamica è necessario ordinare gli intervalli per estremi finali crescenti. In una prima versione $DP[i]$ contiene il profitto massimo ottenivile con i primi i intervalli:

$$DP[i] = \begin{cases} 0 & i = 0 \\ \max(DP[i-1], max\{DP[j] + w_i : j < i \wedge b_j \leq a_i\}) & i > 0 \end{cases}$$

Il costo computazione dell'algoritmo associato è $O(n^2)$.
Una possibile seconda pre-elaborazione consiste nel pre-calcolare il predecessore $pred_i = j$ di i, dove

$j < i$ è il massimo indice tale che $b_j \leq a_i$ e se non esite tale indice $pred_i = 0$.

$$DP[i] = \begin{cases} 0 & i = 0 \\ \max(DP[i-1], DP[pred_i] + w_i) & i > 0 \end{cases}$$

```
: int [] computePredecessor(int [] a, int [] b, int n)
int [] pred = new int [0...n]
pred[0] = 0
for int i = 1 to n do
    int j = i - 1
    while j > 0 and b[j] > a[i] do
        j = j - 1
    pred[i] = j
return pred
```

Si noti come calcolare i predecessori costi $O(n^2$, ma può essere migliorato in $O(n \log n)$.

Implementazione

```
: Set maxInterval(int [] a, int [] b, int [] w, int n)
{ordina gli intervalli per estremi di fine crescenti}
int [] pred = computePredecessor(a, b, n)
int [] DP = new int [0...n]
DP[0] = 0
for int i = 1 to n do
    DP[i] = max(DP[i - 1], w[i] + DP[pred[i]])
int i = n
Set S = Set()
while i > 0 do
    if DP[i - 1] > w[i] + DP[pred[i]] then
        i = i - 1
    else
        S.insert(i)
        i = pred[i]
return S
```

Costo computazionale

- Ordinamento intervalli: $O(n \log n)$.
- Calcolo predecessori: $O(n \log n)$.
- Riempimento tabella DP: $O(n)$.
- Ricostruzione soluzione: $O(n)$.
- Algoritmo completo: $O(n \log n)$.

Capitolo 14

Algoritmi greedy

Gli algoritmi gridi vengono utilizzati per la risoluzione di problemi di ottimizzazione che eseguono una sequenza di decisioni. Se la programmazione dinamica valuta in maniera bottom-up valutando tutte le decisioni possibili evitando di ripetere sotto-problemi o decisioni già percorse gli algoritmi greedy selezionano una sola delle possibili decisioni che sembra ottima, ovvero è localmente ottima. In questo caso si rende necessario dimostrare che si ottiene un ottimo globale. La tecnica greedy va applicata se è possibile dimostrare che esiste una scelta ingorda, ovvero fra le molte scelte possibili ne può essere facilmente individuata una che porta sicuramente alla soluzione ottima e se il problema ha sottostruttura ottima, ovvero se fatta tale scelta resta un sottoproblema con la stessa struttura del problema principale. Non tutti i problemi hanno una scelta greedy e in alcuni casi soluzioni ottime possono comunque essere interessanti.

14.1 Insieme indipendente massimale di intervalli

Input Sia $S = \{1, \ldots, n\}$ un insieme di intervalli della retta reale. Ogni intervallo $[a_i, b_i[$, con $i \in S$ è chiuso a sinistra e aperto a destra. Si indica con a_i il tempo di inizio e con b_i iltempo di fine.

Output Si vuole trovare un insieem dipendente massimale, ovvero un sottoinsieme di massima cardinalità formato da intervalli tutti disgiunti tra di loro (naturale confronto con l'insieme indipendente di intervalli pesati).

14.1.1 Affrontare il problema

Programmazione dinamica

Si individui una sottostruttura ottima e si scriva una definizione ricorsiva per la dimensione della soluzione ottima con la corrispettiva versione iterativa bottom-up.

Tecnica greedy

Si cerchi una possibile scelta ingorda, si dimostri che porta alla soluzione ottima e si scriva un algoritmo ricorsivo o iterativo che effettua sempre la scelta ingorda.

14.1.2 Sottostruttura ottima

Si assuma che gli intervalli siano ordinati per tempo di fine: $b_1 \leq \cdots \leq b_n$ e si definisca il sottoproblema $S[i \dots j]$ come l'insieme di intervalli che iniziano dopo la fine di i e finiscono prima dell'inizio di j: $S[i \dots j] = \{k | b_j \leq a_k < b_k \leq a_j\}$. Si aggiungano due intervalli fittizi: lo zeresimo $b_0 = -\infty$ e l'$n+1$-esimo $a_{n+1} = +\infty$, Il problema iniziale corrisponde a $S[0, n+1]$.

Teorema

Enunciato Si supponga che $A[i \dots j]$ sia una soluzione ottimale di $S[i \dots j]$ e sia k un intervallo che appartiene a $A[i \dots j]$, allora:

- Il problema $S[i \dots j]$ viene diviso in due sottoproblemi:
  - $S[i \dots k]$, gli intervalli che finiscono prima di k.
  - $S[k \dots j$, gli intervalli che iniziano dopo k.
- $A[i \dots j]$ contiene le soluzioni ottimali di $S[i \dots k]$ e $S[k \dots j]$.
  - $A[i \dots j] \cap S[i \dots k]$ è la soluzione ottimale di $S[i \dots k]$.
  - $A[i \dots j] \cap S[k \dots j]$ è la soluzione ottimale di $S[k \dots j]$.

Dimostrazione Si dimostra utilizzando il metodo cut-and-paste.

14.1.3 Definizione ricorsiva del costo della soluzione

Definizione ricorsiva della soluzione

$$A[i \dots j] = A[i \dots k] \cup \{k\} \cup A[k \dots j]$$

Definizione ricorsiva del costo

k viene determinato analizzando tutte le possibilità. Sia pertanto $DP[i][j]$ la dimensione del più grande sottoinsieme $A[i \dots j] \subseteq S[i \dots j]$ di intervalli regolari:

$$DP[i][j] = \begin{cases} 0 & S[i \dots j] = \emptyset \\ \max\limits_{k \in S[i \dots j]} \{DP[i][k] + DP[k][j] + 1\} & altrimenti \end{cases}$$

Questa soluzione permette di scrivere un algoritmo basato su programmazione dinamica con complessità $O(n^3)$: bisogna risolvere tutti i problemi con $i < j$, con costo $O(n)$ per sottoproblema nel caso peggiore.

14.1.4 Scelta ingorda

L'algoritmo risulta migliorabile in quanto non è necessario analizzare tutti valori possibili di k.

Teorema

Enunciato Sia $S[i \dots j]$ un sottoproblema non vuoto e m l'intervallo di $S[i \dots j]$ con il minor tempo di fine, allora il sottoproblema $S[i \dots m]$ è vuoto e m è compreso in qualche soluzione ottima di $S[i \dots j]$.

Dimostrazione Si sa per definizione di intervallo che $a_m < b_m$ e che essendo che m ha minor tempo di fine $\forall k \in S[i \dots j] : b_m \leq b_k$. Ne consegue che $\forall k \in S[i \dots k] : a_m < b_k$ per la proprietà transitiva. Se nessun intervallo in $S[i \dots j]$ termina prima di a_m allora $S[i \dots m] = \emptyset$. Sia ora $A'[i \dots j]$ una soluzione ottima di $S[i \dots j]$ e m' l'intervallo con minor tempo di fine in $A'[i \dots j]$. Sia ora $A[i \dots j] = A - [j \dots j] - \{m'\} \cup \{m\}$ una nuova soluzione ottenuta togliendo m' e aggiungendo m a $A'[i \dots j]$. $A[i \dots j]$ è una soluzione ottima che contiene m in quanto ha la stessa dimensione di $A'[i \dots j]$ e gli intervalli sono indipendenti.

Conseguenze

Con questo teorema non è più necessario analizzare tutti i valori possibili di k: faccio una scelta ingormda ma sicura selezionando l'attività m con il minor tempo di fine. In questo modo non è più necessario analizzare due sottoproblemi eliminando tutte le attività non compatibili con la scelta ingorda, rimane da risolvere solo $S[m \dots j]$.

Implementazione

```
: Set indipendentSet(int [] a, int [] b)
{ordina a e b in modo che b[1] ≤ b[2] ≤ ··· ≤ b[n] }
Set S = Set()
S.insert(1)
int last = 1
for int i = 2 to n do
    if a[i] ≥ b[last] then
        S.insert(i)
        last = i
return S
```

La complessità è $O(n \log n)$ se l'input non è ordinato, $O(n)$ se ordinato.

14.2 Approccio a partire da programmazione dinamica

Si cerca di risolvere il problema della selezione delle attività tramite programmazione dinamica individuando una sottostruttura ottima e scrivendo una definizione ricorsiva per la dimensione della soluzione ottima. Si dimostra la proprietà della scelta greedy: per ogni sottoproblema esiste una soluzione ottima che contiene la scelta greedy e si scrive un algoritmo iterativo che effettua sempre la scelta ingorda.

14.3 Problema del resto

Input Un insieme di tagli di monete memorizzati in un vettore di interi positivi $t[1 \dots n[$ e un intero R rappresentante il resto che si deve restituire.

Definizione del problema Si deve trovare il più piccolo numero intero di pezzi necessari per dare un resto di R centesimi utilizzando i tagli dati assumendo di avere un numero illimitato di

monete per ogni taglio, ovvero trovare un vettore x di interi non negativi tale che $R = \sum_{i=1}^{n} x[i] \cdot t[i]$ e $m = \sum_{i=1}^{n} x[i]$ ha un valore minimo.

14.3.1 Soluzione basata su programmazione dinamica

Sottostruttura ottima

Sia $S(i)$ il problema per dare un resto pari a i e $A(i)$ la soluzione ottima del problema $S(i)$ rappresentata da un multi insieme e $j \in A(1)$, allora $S(i - t[j])$ è un sottoproblema di $S(i)$ la cui solulzione ottima è data da $A(i) - \{j\}$.

Definizione ricorsiva Sia la tabella di programmazione dinamica $DP[0 \dots R]$ e $DP[i]$ il minimo numero di monete per risolvere il problema $S(i)$.

$$DP[i] = \begin{cases} 0 & i = 0 \\ \min\limits_{1 \le j \le n} \{DP[i - t[j] | t[j] \le i\} + 1 & i > 0 \end{cases}$$

14.3.2 Implementazione

```
: int [] resto(int [] t, int n, int R)
int [] DP = new int [0 ... R]
int [] S = new int [0 ... R]
DP[0] = 0
for int i = 1 to R do
    DP[i] = +∞
    for int j = 1 to n do
        if i > t[j] and DP[i - t[j]] + 1 < DP[i] then
            DP[i] = DP[i - t[j]] + 1
            S[i] = j
int [] x = new int [1 ... n]
for int i = 1 to n do
    x[i] = 0
while R > 0 do
    x[S[r]] = x[S[r]] + 1
    R -= t[S[R]]
return x
```

Questo algoritmo ha complessità $O(nR)$.

14.3.3 Scelta greedy

È possibile pensare ad una soluzione greedy selezionando la moneta j più grande tale per cui $t[j] \le R$ e risolvere il problema $S(R - t[j])$.

Implementazione

: int [] `resto`(int // *t*, int *n*, int *R*)

```
int [] x = new int [1 ... n]
{ordina le monete in modo decrescente}
for int i = 1 to n do
    x[i] = ⌊R / t[i]⌋
    R -= x[i] · t[i]
return x
```

Questo algoritmo cha complessità $O(n \log n)$ se l'input non è ordinato e $O(n)$ se è ordinato.

14.4 Approccio greedy senza programmazione dinamica

Si evidenzino i passi di decisione trasformando il problema di ottimizzazione in un problema di scelte successive, scegliendo una possibile scelta ingorda dimostrando che ripetta il "principio della scelta ingorda". Si evidenzia la sottostruttura ottima dimostrando che la soluzione ottima del problema rimanente dopo la scelta ingorda può essere unita a tale scelta. La struttura del codice è top-down, anche iterativamente. Può essere necessario pre-processare l'input.

14.5 Scheduling

Input Si supponga di avere un processore e n job da eseguire su di esso, ognuno caratterizzato da un tempo di esecuzione $t[i]$ eseguito a priori.

Problema Si dee trovare una sequenza di esecuzione che minimizzi il tempo di completamento medio. Dato un vettore $A[1 \dots n]$ contenente una permutazione di $\{1 \dots n\}$ il tempo di completamento dell'h-esimo job nella permutazione è $T_A(h) = \sum_{i=1}^{h} t[A[i]$.

14.5.1 Dimostrazione di correttezza

Teorema di scelta greedy

Enunciato Esiste una soluzioen ottima A in cui il job con minore tempo di fine m si trova in proma posizione: $A[1] = m$.

Dimostrazione Si consideri la permutazione ottima $B = [B[1], \dots, B[m-1], B[m], B[m+1], \dots, B[n]]$ e sia m la posizione in cui si trova in B il job con il minor tempo di fine. Si consideri una permutazione in cui i job in posiizone 1 vengono scambiatiL $A = [B[m], B[2] \dots, B[m-1], B[1], B[m+1], \dots, B[n]]$. Il tempo di completamento medio di A è minore o uguale al tempo di completamento medio di B. In quanto i job in posizione $1, \dots, m-1$ in A hanno tempo di completamento $\leq$ dei job in posizione $1, \dots, m-1$ in B e i job in posizione $m, \dots, n$ in A hanno tempo di completamento uguale ai job in posizione $m, \dots, n$ in B. Essendo B ottima, A non può avere tempo di completamento medio minore e pertanto anche A è ottima.

Teorema della sottostruttura ottima

Enunciato Sia A una soluzione ottima di un problema con n job in cui il job con minor tempo di fine m si trova in prima posizione. La permutazione dei seguenti $n-1$ job in A è una soluzione ottima al sottoproblema in cui il job m non viene considerato.

14.6 Problema dello zaino reale

Input Un intero positivo C, la capacità dello zaino e n oggetti, tali che l'oggetto i-esimo è caratterizzato da un profitto $p_i \in \mathbb{Z}^+$ e un peso $w_i \in \mathbb{Z}^+$.

Problema Trovare un sottoinsieme S di $\{1, \dots n\}$ di oggetti tale che il loro peso totale non superi la capacità massima e il loro profitto totale sia massimo. Nella variante dello zaino reale è possibile prendere frazioni di oggetti.

14.6.1 Approcci greedy possibili

Gli oggetti si possono ordinare per profitto decrescente, per peso crescente o per profitto specifico ($\frac{p_i}{w_i}$) decrescente. Il terzo approccio funziona unicamente nella variante dello zaino reale.

Implementazione

: float [] `zaino(`**float** *[]* p, **float** *[]* v, **float** C,**float** n`)`

float [] x = **new float** $[1 \dots n]$
{ordina p e v in modo che $\frac{p[1]}{w[1]} \geq \cdots \geq \frac{p[n]}{w[n]}$ }
for int i = *1* **to** n **do**
  x[i] = `min(`$\frac{C}{w[i]}$, *1*`)`
  C -= x[i] · w[i]
return x

L'algoritmo ha complessità $O(n \log n)$ se l'input non è ordinato, $O(n)$ se è ordinato. $x[i] \in [0, 1]$ rappresenta la proporzione dell'oggetto i-esimo che deve essere prelevata.

14.6.2 Correttezza

Si assuma che gli oggetti siano ordinati per profitto specifico decrescente e x una soluzione ottima. Si supponga che $x[1] < \min(\frac{C}{w[i]}, 1)$ e la proporzione di uno o più oggetti è ridotta di conseguenza. Si ottiene così una soluzione di profitto uguale o superiore visto che il profitto specifico dell'oggetto 1 è massimo.

14.7 Problema della compressione

La compressione si occupa di rappresentare i dati in modo efficiente, impiegando il minore numero di bit per la rappresentazione risparmiando spazio su disco e tempo di trasferimento. Una possibile tecnica di compressione è la codifica di caratteri tramite una funzione di codifica $f : f(c) = x$, dove c è un carattere preso da un alfabeto Σ e x è una rappresentazione binaria.

14.7.1 Codifica a prefissi

In un codice a prefisso o senza prefissi nessun codice è prefisso di un altro codice, la condizione necessaria per la codifica.

Rappresentazione ad albero per la decodifica

L'alfabeto viene rappresentato come un albero binario le cui foglie sono i caratteri dell'alfabeto e tutti gli altri nodi hanno due figli. In base al bit letto si sceglie dove spostarsi nell'albero fino a trovare il carattere necessario.

: Algoritmo di decodifica

```
si parte dalla radice
while file non finito do
    si legge un bit
    if bit = zero then
        vai nel figlio sinistro
    else
        vai a destra
    if nodo foglia then
        stampa il carattere
        torna alla radice
```

14.7.2 Definizione formale del problema

Input Un file F composto da caratteri nell'alfabeto Σ. Sia T un albero che rappresenta la codifica, per ogni $c \in \Sigma$, sia $d_T(c)$ la profondità della foglia che rappresenta c, il codice richiederà $d_T(c)$ bit per c. Se $f[c]$ è il numero di occorrenze di c in F, allora la dimensione della codifica è $C[F,T] = \sum_{c \in \Sigma} f[c] \cdot d_T(c)$.

14.7.3 Algoritmo di Huffman

Il principio del codice di Huffman consiste del minimizzare la lunghezza dei caratteri che compaiono più frequentemente assegnando ai caratteri con la frequenza minore i codici corrispondenti ai percorsi più lunghi all'interno dell'albero. Ogni codice è progettato per un file specifico: si ottiene la frequenza di tutti i caratteri, si costruisce il codice, si rappresenta il file tramite il codice e si aggiunge al file una rappresentazione del codice per la decodifica.

Funzionamento algoritmo

- Si costruisce una lista ordinata di nodi foglia per ogni carattere etichettato con la propria frequenza.
- Si rimuovono i due nodi con frequenze minori f_x e f_y creando un nodo padre con etichetta "-" e frequenza $f_x + f_y$ collegando i due nodi rimossi con il nuovo nodo e aggiungendolo alla lista mantenendo l'ordine.
- Iterare il secondo punto fino a che rimane un unico nodo nella lista e al termine si etichettano gli archi all'interno dell'albero con bit 0, 1.

```
: Tree huffman(int [] c, int [] f, int n)
  PriorityQueue Q = MinPriorityQueue()
  for int i = 1 to n do
      Q.insert(f[i], Tree(F[i], c[i]))
  for int i = 1 to n - 1 do
      Tree z1 = Q.deleteMin()
      Tree z2 = Q.deleteMin()
      Tree z = Tree(z1.f + z2.f, nil)
      z.left = z1
      z.right = z2
      Q.insert(z.f, z)
  return Q.deleteMin()
```

Questo algoritmo ha complessità $O(n \log n)$.

Correttezza

L'output dell'algoritmo di Huffman per un dato file è un codice a prefisso ottimo. La proprietà della scelta greedy è che scegliere i due elementi con la frequenza più bassa conduce sempre ad una soluzione ottimale. La sottostruttura ottima è che dato un problema sull'alfabeto Σ è possibile costruire un sottoproblema con alfabeto più piccolo.

Scelta greedy

Enunciato Siano Σ un alfabeto, f un vettore di frequenze e x e y i due caratteri con frequenza più bassa. Esiste un codice prefisso ottimo per Σ in cui x e y hanno la stessa profondità massima e i loro codici differiscono solo per l'ultimo bit.

Dimostrazione Si basa sulla trasformazione di una soluzione ottima. Si supponga che esista un codice ottimo T in cui due catatteri a e b con profondità massima siano diversi da x e y. Si può assumere senza perdere di generalità che $f[x] \leq f[y] \wedge f[a] \leq f[b]$. Essendo le frequenze di x e y minime $f[x] \leq f[a] \wedge f[y] \leq f[b]$. Scambiando x con a si ottiene T' e scambiando y con b si ottiene T''. Si dimostri ora che $C(f, T'') \leq C(f, T') \leq C(f, T)$.

$$\begin{aligned} C(T) - C(T') &= \sum_{c \in \Sigma} f[c] d_T(c) - \sum_{c \in \Sigma} f[c] d_{T'}(c) \\ &= (f[x] d_T(x) + f[a] d_T(a)) - (f[x] d_{T'}(x) + f[a] d_{T'}(a)) \\ &= (f[x] d_T(x) + f[a] d_T(a)) - (f[x] d_T(x) + f[a] d_T(a)) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \\ &\geq 0 \end{aligned}$$

$C(T') - C(T'') \geq 0$ Come sopra

Essendo T ottimo allora $C(f, T) \leq C(f, T'')$, pertanto anche T'' è ottimo.

14.8 Albero di copertura di peso minimo

Dato un grafo pesato determinare come interconnettere tutti i suoi nodi minimizzando il costo del peso associato ai suoi archi, ovvero albero di copertura, di connessione di peso minimo o minimum spanning tree.

14.8.1 Definizione del problema

Sia $G = (V, E)$ un grafo non orientato e connesso e $w : V \times V \to \mathbb{R}$ una funzione di peso costo di connessione tale che se $[u, v] \in E$ allora $w(u, v)$ è il peso dell'arco $[u, v]$, se $[u, v] \notin E$, allora $w(u, v) = +\infty$. Essendo G non orientato $w(u, v) = w(v, u)$.

Albero di copertura

Dato un grafo non orientato e connesso un albero di copertura G è un sottografo $T = (V, E_T)$ tale che T è un albero, $E_T \subseteq E$ e T contiene tutti i vertici di G.

Output

Trovare l'albero di copertura il cui peso totale sia minimo rispetto a ogni altro albero di copertura, dove $w(T) = \sum_{[u,v] \in E_T} w(u, v)$. Non è detto che sia univoco.

14.8.2 Algoritmo generico

Ci approccia al problema tentando di accrescere un sottoinsieme A di archi in modo tale che venga sempre rispettato il fatto è A è un sottoinsieme di qualche albero di connessione minimo.

Arco sicuro

Un arco $[u, v]$ si dice sicuro per A se $A \cup \{[u, v]\}$ è ancora un sottoinsieme di qualche albero di connessione minimo.

```
: Set mst-generico(Graph G, int // w)
Set A = ∅
while A not albero di copertura do
    trova un arco sicuro [u, v]
    A = A ∪ {[u, v]}
return A
```

Definizioni

- Un taglio $(S, V - S)$ di un grafo non orientato è una partizione di V in due sottoinsiemi disgiunti.
- Un arco $[u, v]$ attraversa il taglio se $u \in S$ e $v \in V - S$.
- Un taglio rispetta un insieme di archi A se nessun arco di A attraversa il taglio.
- Un arco che attraversa un taglio è leggero nel taglio se il suo peso è minimo fra i pesi degli archi che attraversano un taglio.

Enunciato Sia $G = (V, E)$ un grafo non orientato e connesso, $w : V \times V \to \mathbb{R}$, $A \subseteq E$ un sottoinsieme contenuto in un qualche albero di copertura minimo per G, $(S, V - S)$ un qualunque taglio che rispetta A, sia $[u, v]$ un arco leggero che attraversa il taglio, allora l'arco $[u, v]$ è sicuro per A.

Dimostrazione Sia T un albero di copertura minimo che contiene A, se $(u, v) \in T$ allora (u, v) è sicuro per A, se $(u, v) \notin T$ si trasforma T in T' che contiene (u, v) e si dimostra che T' è un albero di copertura minimo. Per definizione di albero u e v sono connessi da un cammino $C \subseteq T$ e stanno in lati opposti del taglio e $\exists (x, y) \in C$ che attraversa il taglio. $T' = T - \{(x, y)\} \cup \{(u, v)\}$, T' è un albero di copertura, $w(T') \leq w(T)$ perchè $w(u, v) \leq w(x, y)$, $w(T) \leq w(T')$ in quanto T è minimo.

Corollario Sia $G = (V, E)$ un grafo non orientato e connesso , $w : V \times V \to \mathbb{R}$, $A \subseteq E$ un sottoinsieme contenuto in un qualche albero di copertura minimo per G, C una componente connessa nella foresta $G_A = (V, A)$ e $[u, v]$ un arco leggero che connettte C a qualche altra componente in G_A, allora l'arco $[u, v]$ è sicuro per A.

14.8.3 Algoritmo di Kruskal

Si ingrandiscono sottoinsiemi disgiunti di un albero di copertura minimo connettendoli fra di loro fino ad avere l'albero complessivo. Si individua un arco sicuro scegliendo un arco $[u, v]$ di peso minimo tra tutti gli archi che connettono due distinti alberi della foresta. L'algoritmo è greedy perchè ad ogni passo si aggiunge alla foresta un arco con il peso minore. Per implementarlo si utilizza un struttura dati Merge-Find set.

```
: Set kruskal(Edge [] A, int n, int m)
Set T = Set()
Mfset M = Mfset(n)
{ordina A[1.dots, m] in modo che A[1].weight ≤ ··· ≤ A[m].weight}
int count = 0; int i = 1; %Termina quando l'albero ha n − 1 archi o non ci sono più archi
while count < n - 1 and i ≤ m do
    if M.find(A[i].u) ≠ M.find(A[i].v) then
        M.merge(A[i].u, A[i].v)
        T.insert(A[i])
        count += 1
    i += 1
return T
```

Analisi della complessità

Il tempo di esecuzione per l'algoritmo di Kruskal dipende dalla realizzazione della struttura dati per Merge-Find Set, utilizzando la versione con euristica sul rango e compressione le cui operazioni hanno costo ammortizzato costante si ha:

Fase	Volte	Costo
Inizializzaizione	1	$O(n)$
Ordinamento	1	$O(m \log m)$
Operazioni *find()*, *merge()*	$O(m)$	$O(1)$

Il totale è $O(n + m \log m + m) = O(m \log m) = O(m \log n^2) = O(m \log n)$.

14.8.4 Algoritmo di Prim

Si mantiene in A un singolo albero che parte da un vertice arbitrario r e cresce fino a che non ricopre tutti i vertici. Ad oggni passo viene aggiunto un arco leggero che collega un vertice in V_A conun vertice in $V - V_A$.

Correttezza Per definizione $(V_A, V - V_A)$ è un taglio che rispetta A e per il corollario gli archi leggeri che attraversano il taglio sono sicuri.

Implementazione

Durante l'esecuzione i vertici non ancora nell'albero si trovano in una coda con min-priorità Q ordinata secondo la priorità: la priorità del nodo v è il peso minimo di un arco che collega v ad un vertice nell'albero o $+\infty$ se tale arco non esite. L'albero viene mantenuto come vettore dei padri. A è mantenuto implicitamente: $A = \{[v, p[v]] | v \in V - Q - \{r\}\}$.

```
: prim(Graph G, int [] p)
PriorityQueue Q = MinPriorityQueue()
PriorityItem [] pos = new PriorityItem [1...G.n]
foreach u ∈ G.V() - {r} do
    pos[u] = Q.insert(u, +∞)
pos[r] = Q.insert(r, 0)
p[r] = 0
while not Q.isEmpty() do
    Node u = Q.deleteMin()
    pos[u] = nil
    foreach v ∈ G.adj(u) do
        if pos[v] ≠ nil and w(u, v) < pos[v].priority then
            Q.decrease(pos[v], w(u, v))
            p[v] = u
```

Analisi complessità

L'efficienza dell'algorimto di Prim dipende dall'implementazione della coda di priorità. Se si utilizza uno heap binario si ha:

Fase	Volte	Costo
Inizializzaizione	1	$O(n \log n)$
deleteMin()	$O(n)$	$O(\log m)$
decreasePriority()	$O(m)$	$O(\log n)$

Per un tempo totale $O(n + n \log n + m \log n) = O(m \log n)$, asintoticamente uguale a quello di Kruskal. Le cose cambiano se si utilizza un vettore non ordinato.

Capitolo 15

Ricerca locale

Se si conosce una soluzione ammissibile e non necessariamente ottima ad un problema di ottimizzazione si può cercare di trovare una soluzione migliore nelle vicinanze di quella precedente, continuando in questo modo fino a che non si è più in grado di continuare.

```
: ricercaLocale()
```

Sol = una soluzione ammissibile del problema
while $\exists S \in I(sol)$ *migliore di Sol* **do**
  Sol = S
return Sol

15.1 Rete di flusso

Una rete di flusso $G = (V, E, s, t, c)$ è data da un grafo orientato $G = (V, E)$, un nodo $s \in V$ detto sorgente, un nodo $t \in V$ detto pozzo e una funzione di capacità $c : V \times V \to \mathbb{R}^{\geq 0}$. Si assume che per ogni nodo $v \in V$ esiste un cammino $s \rightsquigarrow v \rightsquigarrow t$ da s a t che passa per v e si possono ignorare i nodi che non godono di questa priorità.

15.1.1 Flusso

Un flusso in G è una funzione $f : V \times V \to \mathbb{R}$ che soddisfa le priorità:

- Vincolo sulla capacità: $\forall u, v \in V, f(u, v) \leq c(u, u)$, ovvero il flusso non deve eccedere la capacità sull'arco.
- Antisimmetria: $\forall u, v \in V, f(u, v) = -f(v, u)$, in modo da semplificare la proprietà successiva e altre regole.
- Conservazione del flusso: $\forall u \in V - \{s, t\}, \sum_{v \in V} f(u, v) = 0$, ovvero per ogni nodo la somma dei flussi entranti deve essere uguale alla somma dei flussi uscenti.

15.1.2 Definizioni

- Valore di flusso f: $|f| = \sum_{(s,v) \in E} f(s, v)$, la quantità di flusso uscente da s.

- Flusso massimo: data una rete $G = (V, E, s, t, c)$ trovare un flussoo che abbia valore massimo fra tutti i flussi associabili alla rete $f* = \max\{|f|\}$.
- Flusso nuillo: la funzione $f_0 : V \times V \to \mathbb{R}^{\geq 0}$ tale che $f(u, v) = 0 \forall u, v \in V$.
- Somma di flussi: per ogni coppia di flussi f_1 e f_2 in G si definisce il flusso somma $g = f_1 + f_2$ come un flusso tale per cui $g(u, v) = f_1(u, v) + f_2(u, v)$.
- Capacità residua di un flusso f in una rete $G = (V, E, s, t, c)$ una funzione $c_f : V \times V \to \mathbb{R}^{\geq 0}$ tale che $c_f(u, v) = c(u, v) - f(u, v)$.
- Reti residue: data una rete di flusso $G = (V, E, s, t, c)$ e un flusso f su G si può costruire una rete residua $G_f = (V, E_f, s, t, c_f)$ tale per cui $(u, v) \in E_f$ se e solo se $c_f(u, v) > 0$.

15.1.3 metodo delle reti residue

Questo algoritmo informalmente memorizza un flusso corrente di f inizialmente nullo e in ripetizione si sottrae il flusso attuale dalla rete iniziale ottenendo una rete residua, si cerca un flusso g all'interno della rete residua e si somma g ad f fino a quando non è più possibile trovare un flusso positivo g. Questo approccio restituisce il flusso massimo.

```
: int [][] maxFlow(Graph G, Node s, Node t, int [][] c)
f = f_0 %Inizializza un flusso nullo
r = c %Capacità iniziale
do
    g = trova un flusso in r tale che |g| > 0, altrimenti f_0
    f = f + g
    r = rete di flusso residua del flusso di f in G
while g = f_0
return f
```

Correttezza

Lemma Se f è un flusso in G e g è un flusso in G_f allora $f + g$ è un flusso in G.

Dimostrazione

Vincolo sulla capacità

$$
\begin{aligned}
g(u, v) &\leq c_f(u, v) && g \text{ è un flusso in } G_f \\
f(u, v) + g(u, v) &\leq c_f(u, v) + f(u, v) && \text{Aggiungo un termine uguale} \\
(f + g)(u, v) &\leq c(u, v) - f(u, v) + f(u, v) && \text{Sostituzione} \\
(f + g)(u, v) &\leq c(u, v) && \text{Semplificazione}
\end{aligned}
$$

Antisimmetria

$$f(u,v) + g(u,v) = -f(v,u) - g(v,u) \qquad \text{Antisimmetria di } f \text{ e } g$$
$$f(u,v) + g(u,v) = -(f(v,u) + g(v,u)) \qquad \text{Raccolta segno } -$$
$$(f+g)(u,v) = -(f+g)(v,u) \text{Sostituzione}$$

Conservazione

$$\begin{aligned}\sum_{v \in V}(f+g)(u,v) &= \sum_{v \in V}(f(u,v) + g(u,v)) \\ &= \sum_{v \in V} f(u,v) + \sum_{v \in V} g(u,v) \\ &= 0\end{aligned}$$

15.1.4 Metodo dei cammini aumentanti

Il punto principale del metodo precedente è come trovare un flusso aggiuntivo.

Ford-Fulkerson

Si trova un cammino $C = v_0, \ldots, v_n$ con $s = v_0$ e $t = v_n$ nella rete residua G_f. Si identifica la capacità del cammino, ovvero la minore capacità degli archi incontrati $c_f = \min_{i=2\ldots n} c_f(v_{i-1}, v_i)$, si crea un flusso addizionale g tale che: $g(v_{i-1}, v_i) = c_f(C)$, $g(v_i, v_{i-1}) = -c_f(C)$ e $g(x,y) = 0$ per tutte le altre coppie.

```
: int [][] maxFlow(Graph G, Node s, Node t, int [][][] c)
    int [][] f = new int [][] %Flusso parziale
    int [][] g = new int [][] %Flusso da cammino aumentante
    int [][] r = new int [][] %Rete residua
    foreach u, v ∈ G.V() do
        f[u][v] = 0 %inizializza un flusso nullo
        r[u][v] = c[u][v] %Copia c in r
    do
        g = flusso associato ad un cammino aumentante in r o f_0
        foreach u, v ∈ G.V() do
            f[u][v] = f[u][v] + g[u][v] %f = f + g
            r[u][v] = c[u][v] - f[u][v] %Calcola c_f
    while g = f_0
    return f
```

15.1.5 Ricerca del cammino

Ford e Fulkerson propongono una visita del grafo semplice in profondità o in ampiezza, mentre Eddomnds e Karp una visita in ampiezza. Il costo della visita è $O(V + E)$.

15.1.6 Complessità

Se le capacità sono intere l'algoritmo Ford-Fulkerson ha complessità $O((V+E)|f^*|)$ per le liste o $O(V^2|f^*|)$ per la matrice. L'algoritmo parte dal flusso nullo e termina quando il valore totale del flusso raggiunge $|f^*|$ e ogni incremento del flusso lo aumenta di almeno un'unità. Ogni ricerca di un cammino richiede una visita del grafo con costo $O(V+E)$ o $O(V^2)$. La somma dei flussi e il calcolo della rete residua può essere effettuato in tempo $O(V+E)$ o $O(V^2)$. Se le capacità della rete sono intere l'algoritmo di Edmonds e Karp ha complessità $O(VE^2)$ nel caso pessimo.

Dimostrazione

La complesità dell'algoritmo di Edmonds-Karp è $O(VE^2)$. Vegono esesguiti $O(VE)$ aumenti di flusso, ognuno dei quali richiede una visita in ampiezza $O(V+E)$: $O(VE(V+E)) = O(VE^2)$.

Lemma della monotonia Sia $\delta_f(s,v)$ la distanza minima da s a v in una rete residua G_f e $f' = f + g$ un flusso nella rete iniziale con g flusso non nullo derivante da un cammino aumentante, allora $\delta_{f'}(s,v) \geq \delta_f(s,v)$.

Dimostrazione Quando viene aumentato il flusso alcuni archi si spengono utilizzati nei cammini minimi e i cammini minimi non possono diventare più corti.

Lemma degli aumenti di flusso Il numero totale di aumenti di flusso eseguiti dall'algoritmo di Edmonds-Karp è $O(VE)$.

Dimostrazione Sia G_f una rete residua e C un cammino aumentante di G_f. (u,v) è un arco critico in C se $c_f(u,v) = \min_{x,y \in C}\{c_f(x,y)\}$. In ogni cammino esiste almeno un arco critico e una volta aggiunto il flusso associato a C l'arco critico scompare dalla rete residua.

Conclusione Poichè i cammini aumentanti sono cammini minimi si ha che $\delta_f(s,v) = \delta_f(s,u) + 1$. L'ascro (u,v) può ricomparire se e solo se il flusso lungo l'arco diminuisce, ovvero se (v,u) appare in un cammino aumentante. Sia g il flusso quando questo accade e come sopra si ha $\delta_g(s,u) = \delta_g(s,v) + 1$. Per il lemma della monotonia si ha che $\delta_g(s,v) \geq \delta_f(s,v)$, perciò:

$$\begin{aligned}\delta_g(s,u) &= \delta_g(s,v) + 1 \\ &\geq \delta_f(s,v) + 1 \\ &= \delta_f(s,u) + 2\end{aligned}$$

Dal momento in cui un nodo è critico al momento in cui può tornare ad essere critico il cammino si è allungato di almeno due passi e la lunghezza massima del cammino fino a u è $V-2$, pertanto un arco può diventare critico al massimo $\frac{(V-2)}{2} = \frac{V}{2} - 1$ volte. Essendoci $O(E)$ archi che possono diventare critici $O(V)$ volte si ha che il numero massimo di flussi aumentanti è $O(VE)$.

15.1.7 Dimostrazione Correttezza

Definizioni

- Un taglio (S,T) della rete di flusso $G = (V,E,s,t,c)$ è una partizione di V in S e $T = V - S$ tale che $s \in S$ e $t \in T$.

- Si dice capacità del taglio $C(S,T)$ attraverso (S,T) $C(S,T) = \sum_{u \in S, v \in T} c(u,v)$.
- Se f è flusso in G il flusso netto $F(S,T)$ attraverso (S,T) è $F(S,T) = \sum_{u \in S, v \in T} f(u,v)$.

Lemma del valore del flusso di un taglio

Dato un flusso f e un taglio (S,T) la quantità di flusso $F(S,T)$ che attraversa il taglio è uguale a $|f|$.

Dimostrazione

$$\begin{aligned}
F(S,T) &= \sum_{u \in S, v \in T} f(u,v) \\
&= \sum_{u \in S, v \in V} f(u,v) - \sum_{u \in S, v \in S} f(u,v) && T = V - S \\
&= \sum_{u \in S, v \in V} f(u,v) && \text{Antisimmetria} \\
&= \sum_{u \in S - \{s\}, v \in V} f(u,v) + \sum_{v \in V} f(s,v) && s \in S \\
&= \sum_{v \in V} f(s,v) && \text{Conservazione flusso} \\
&= |f| && \text{Definizione valore flusso}
\end{aligned}$$

Lemma della capacità di taglio

Il flusso massimo è limitato superiormente dalla capacità del taglio minimo, ovvero il taglio la cui capacità è minore fra tutti i tagli.

Dimostrazione Nessun flusso attraverso un taglio supera la capacità di taglio: $F(S,T) \leq C(S,T) \forall S \subseteq V, T = V - S$:

$$F(S,T) = \sum_{u \in S, v \in T} f(u,v) \leq \sum_{u \in S, v \in T} c(u,v) = C(S,T)$$

Il flusso che attraversa un taglio è uguale al valore di flusso: $|f| = F(S,T) \forall S \subseteq V, T = V - S$, pertanto il valore del flusso massimo è limitato superiormente dalla capacità di tutti i possibili tagli: $|f| \leq C(S,T) \forall S \subseteq V, T = V - S$.

Teorema del taglio minimo o del flusso massimo

Le seguenti tre affermazioni sono equivalenti:

1. f è un flusso massimo.
2. Non esiste nessun cammino aumentante per G.
3. Esiste un taglio minimo (S,T) tale che $C(S,T) = |f|$.

(**1**) $\Rightarrow$ (**2**) f è un flusso massimo implica che non esiste un cammino aumentante per G: se esistesse un cammino aumentante il flusso potrebbe essere aumentato e quindi non sarebbe massimo, che è un assurdo.

(**2**) $\Rightarrow$ (**3**) Non esiste nessun cammino aumentante per G implica che esiste un taglio minimo (S, T) tale che $C(S, T) = |f|$. Poichè non esiste nessun cammino aumentante per f, non esiste nessun cammino da s a t nella rete residua G_f. Sia S l'insieme dei vertici raggiungibili da s, $T = V - S$. Ovviamente $s \in S$ e $t \in T$, pertanto (S, T) è un taglio. Poichè t non è raggiungibile da s in G_f tutti gli archi (u, v) con $u \in S$ e $v \in T$ sono saturati e $f(u, v) = c(u, v)$. Per il lemma del valore del flusso di taglio $|f| = \sum_{u \in S, v \in T} f(u, v)$ e pertanto $|f| = \sum_{u \in S, v \in T} f(u, v) = \sum_{u \in S, v \in T} c(u, v) = C(S, T)$ e (S, T) è minimo in quanto $|f| = C(S, T)$ e per ogni taglio (S', T') si ha che $|f| \geq C(S', T')$.

(**3**) $\Rightarrow$ (**1**) Esiste un taglio (S, T) tale che $C(S, T) = |f|$ implica che f è un flusso massimo. In quanto per un qualsiasi flusso f e un qualsiasi taglio (S, T) vale la relazione $|f| \leq C(S, T)$ il flusso che soddisfa $|f| = C(S, T)$ deve essere massimo.

15.2 Abbinamento massimo nei grafi bipartiti o problema del job assignment

Input Un insieme J contenente n job, un insieme W contenente m worker. Una relazione $R \subseteq J \times W$ tale per cui $(j, w) \in R$ se e solo se il job può essere eseguito dal worker s.

Output Il più grande sottoinsieme $O \subseteq R$ tale per cui ogni job venga assengato al più ad un worker e ad ogni worker venga assegnato al più un job.

15.2.1 Metodo di risoluzione

Si crea una rete di flusso in cui $c(w, j)$ vale 1 se $(j, w) \in R$, altrimenti 0. Si aggiungono una super fonte che si collega ai lavoratori con archi di peso 1 e un superpozzo che si collega ai job con archi di peso 1 e si usa l'algoritmo di ricerca locale visto sopra per trovare una soluzione.

Capitolo 16

Backtracking

Dato un problema si dice soluzione ammissibile una soluzione che soddifa un insieme di criteri: nel caso dello zaino un sottoinsieme di oggetti di peso inferiore alla capacità, mentre nella sottosequenza comune una stringa che è sottosequenza di entrambe le stringhe in input. Nei problemi di ottimizazione viene definita una funzione di costo o guadagno definita sull'insieme delle soluzioni ammissibili: nello zaino è la somma dei guadagni degli oggetti selezionati, mentre nella sottosequenza comune massimale è la lunghezza della stringa.

16.1 Brute force

In alcuni problemi è richiesto o necessario esplorare l'intero spazio delle soluzioni ammissibili. Nell'enumerazione si richiede di elencare tutte le soluzioni possibili, nella ricerca di trovare una soluzione ammissibile in uno spazio delle soluzioni molto grande, nel conteggio contare tutte le soluzioni ammissibili (quando non è possibile contarle in modo analitico), mentre nell'ottimizzazione si trovare una delle soluzioni ammissibili migliori rispetto ad un criterio di valutazione.

16.1.1 Costruire lo spazio delle soluzioni è costoso

Lo spazio delle possibili soluzioni può essere superpolinomiale e a volta è l'unica strada possibile. A volte è possibile analizzare solo una parte dello spazio.

16.2 Backtracking

Il backtracking è una tecnica di programmazione che prova a fare qualcosa e se non va bene lo disfa e prova qualcos altro fino a che non arriva a una soluzione. Può essere implementata in maniera ricorsiva (un metodo sistematico per esplorare uno spazio di ricerca utilizzando la ricorsione per memorizzare le scelte fatte fin'ora) o iterativa (utilizzando un approccio greedy, tornando eventualmente sui propri passi). Questa tecnica algoritmica deve essere naturalmente personalizzata per ogni applicazione individuale.

16.2.1 Organizzazione generale

Una soluzione viene rappresentata come un vettore $S[1 \dots n]$ il contenuto degli elementi $S[i]$ è preso da un insieme di scelte C dipendente dal problema.

16.2.2 Soluzioni parziali

Ad ogni passo si parte da una soluzione parziale $S[1 \dots k]$ in cui $k \geq 0$ scelte sono state prese. Se $S[1 \dots k]$ è una soluzione ammissibile viene processata, se non è una soluzione completa se possibile la si estende con una delle possibili scelte in $S[1 \dots k+1]$, altrimenti si cancella l'elemento $S[k]$ (backtrack) e si riparte dalla soluzione $S[1 \dots k-1]$.

16.2.3 Albero delle decisioni

Durante l'algoritmo si genera un albero delle decisioni equivalente allo spazio di ricerca la cui radice è la soluzione parziale vuota, i nodi interni le soluzioni parziali e le foglie le soluzioni ammissibili.

16.2.4 Pruning

I rami dell'albero che sicuramente non portano a soluzioni ammissibili possono essere potati o "pruned" e la valutazione viene fatta nelle soluzioni parziali radici nel sottoalbero da potare.

16.3 Enumerazione

```
: enumeration(<dati problema<, Item [] S, int i, <dati parziali>)
    %Verifica se S[1...i-1] contiene una soluzione ammissibile
    if accept(<dati problema>, S, i <dati parziali>) then
        %Processa la soluzione
        processSolution(<dati problema>, S, i, <dati parziali>)
    else
        %Calcola l'insieme delle scelte in funzione di S[1...i-1]
        Set C = choices(<dati problema>, S, i, <dati parziali>)
        %Itera sull'insieme delle scelte
        foreach c ∈ C do
            S[i] = c
            %Chiamata ricorsiva
            enumeration(<dati problema>, S, n, i + 1, <dati parziali>)
    return false
```

16.3.1 Sottoinsiemi

Il problema consiste nell'elencare tutti i sottoinsiemi dell'insieme $\{1, \dots, n\}$.

```
: subsetsRec(int n, Item [] S, int i)
    %S ammissibile dopo n scelte
    if i > n then
        processSolution(S, n)
    else
        %Non presente o presente
        Set C = {0, 1}
        foreach c ∈ C do
            S[i] = c
            subsetsRec(n, S, n, i + 1)
    return false
```

```
: subsets(int n)
    %Vettore delle scelte
    int [] S = new int [1 . . . n]
    subsetsRec(N, S, 1)
```

```
: processSolution(int [] S, int n)
    print "{"
    for int i = 1 to n do
        if S[i] then
            print i, " "
    println "}"
```

Come richiesto dal problema tutto lo spazio possibile viene esplorato con complessità $\Theta(n \cdot 2^n)$.

Versione iterativa

```
: subsets(int n)
    for int j = 0 to 2^n − 1 do
        print "{"
        for int i = 1 to n - 1 do
            if (j && 2^i) ≠ 0 then
                %Bitwise and
                print i, " "
        println "}"
```

Complessità $\Theta(n \cdot 2^n)$.

16.3.2 Permutazioni

Stampa tutte le permutazioni di un insieme A.

```
: permRec(Item [] S, int i)
    %Caso base con un carattere
    if i == 1 then
        println S
    else
        for int j = 1 to i do
            swap(S, i, j)
            permRec(S, i-1)
            swap(S, i, j)
```

```
: permutations(Item [] S, int n)
    permRec(S, n)
```

Complessità:

- $n!$ permutazioni.
- $\Theta(n)$ per stamparle tutte.
- $2n$ swap per ogni permutazione.
- Costo totale: $\Theta(n \cdot n!)$.

16.3.3 Enumerazione schema completo

```
: enumeration(<dati problema<, Item [] S, int i,<dati parziali>)
if accept(<dati problema>, S, i <dati parziali>) then
    %Processa la soluzione S in quanto ammissibile
    processSolution(<dati problema>, S, i, <dati parziali>)
else if reject(<dati problema>, S, i, <dati parziali>) then
    return
else
    %Ricorsione
    Set C = choices(<dati problema>, S, i, <dati parziali>)
    foreach c ∈ C do
        S[i] = c
        enumeration(<dati problema>, S, n, i + 1, <dati parziali>)
```

Ci si può interrompere alla prima soluzione introcendo un *return* dopo *processSolution(<dati problema, S, i, <dati parziali>)*.

16.3.4 K-sottoinsiemi

Il problema consiste nell'elencare tutti i sottoinsiemi di k elementi di un insieme $\{1, \ldots, n\}$. Si può in questo caso specializzare l'algoritmo generico introducendo del pruning migliorandone l'efficienza. Questo avviene introducendo la variabile *missing* che permette di evitare di ricontare tutte le volte i bit a 1 ed evitando di proseguire in rami che non possono dare origine alla soluzione (non ci sono altri elementi da scegliere o quelli rimanenti sono troppi pochi).

```
: kssRec(int n, int missing, Item [] S, int i)
if missing == 0 then
    processSolution(S, i - 1)
else if i ≤ n and 0 < missing ≤ n - (i - 1) then
    foreach c ∈ {0, 1} do
        S[i] = c
        kssRec(n, missing - c, S, i + 1)
```

16.3.5 Somma di sottoinsiemi

Dati un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi e un intero positivo k, si trovi un sottoinsieme S di indici in $\{1, \ldots, n\}$ tale che $\sum_{i \in S} a_i = k$. Lo si risolve in backtracking in tempo $O(2^n)$ e può essere risolto tramite programmazione dinamica in tempo $O(kn)$, pseudopolinomiale. Si interrompe l'esecuzione alla prima soluzione trovata.

: Suset sum

```
subsetSum(int [] A, int n, int k)
    int [] S = new int [1...n]
    ssRec(A, n, k, S, 1)
boolean ssRec(int [] A, int n, int missing, Item [] S, int i)
    if missing == 0 then
        processSolution(S, i - 1)%Stampa gli indici della soluzione
        return true
    else if i > n or missing < 0 then
        return false %Terminati i valori o somma eccessiva
    else
        foreach c ∈ {0,1} do
            S[i] = c
            if ssRec(A, n, missing - A[i] · c, S, i + 1) then
                return true
        return false
```

16.4 Problemi

16.4.1 Problema delle otto regine

Il problema consiste di posizionare n nergine in una scacchiera $n \times n$ in modo tale che nessuna regini ne minacci un'altra.

Ci sono n^2 caselle dove piazzare una regina

$S[1 \dots n^2]$ array binario	$S[i]$ =**true** $\Rightarrow$ regina in $S[i]$
Controllo soluzione	se $i = n^2$
choices(S, n, i)	{**true**, **false**}
pruning	Se la nuova regina minaccia una delle regine esistenti restituisce $\emptyset$
# soluzioni per $n = 8$	$2^{64} \approx 1.84 \cdot 10^{19}$

La matrice binaria è molto sparsa.

Si devono piazzare n regine in n^2 caselle

$S[1 \dots n]$ coordinate in $\{1 \dots n^2\}$	$S[i]$ coordinata della regina i
Controllo soluzione	se $i = n$
choices(S, n, i)	$\{1 \dots n^2\}$
pruning	Restituisce il sottoinsieme delle mosse legali
# soluzioni per $n = 8$	$(n^2)^n = 64^8 \approx 2.81 \cdot 10^{14}$

Si nota un miglioramento ma lo spazio è ancora grande, inoltre è difficile distinguere le permutazioni di una soluzione.

Non mettere regine in caselle precedenti a quelle già scelte

$S[1 \dots n]$ coordinate in $\{1 \dots n^2\}$	$S[i]$ coordinata della regina i
Controllo soluzione	se $i = n$
choices(S, n, i)	$\{1 \dots n^2\}$
pruning	Restituisce il sottoinsieme delle mosse legali, $S[i] > S[i-1]$
# soluzioni per $n = 8$	$\frac{(n^2)^n}{n!} = \frac{2^{48}}{40320} \approx 6.98 \cdot 10^9$

Ogni riga della scacchiera deve contenere esattamente una regina

$S[1 \dots n]$ coordinate in $\{1 \dots n\}$	$S[i]$ colonna della regina i, dove riga $= i$
Controllo soluzione	se $i = n$
choices(S, n, i)	$\{1 \dots n\}$
pruning	Restituisce le colonne legali
# soluzioni per $n = 8$	$n^n = 8^8 \approx 1.67 \cdot 10^7$

Anche ogni colonna deve contenere esattamente una regina

$S[1 \dots n]$ coordinate in $\{1 \dots n\}$	Permutazione di $\{1 \dots n\}$
Controllo soluzione	se $i = n$
choices(S, n, i)	$\{1 \dots n\}$
pruning	Elimina le diagonali
# soluzioni per $n = 8$	$n! = 8! = 4.320$

Le soluzioni effettivamente visitate sono 15720.

Minimum-conflicts heuristic

Si parte da una soluzione iniziale ragionevolmente buona e si muove il pezzo con il più grande numero di conflitti nella casella della stessa colonna che genera il numero minimo di conflitti e si ripete fino a quando non ci sono più pezzi da muovere. È in tempo lineare ma non garantisce che la terminazione sia sempre corretta.

```
: queens(int n, int [] S, int i)
if i > n then
    print S
else
    for int j = 1 to n do
        %Prova a piazzare la regina nella colonna j
        boolean legal = true
        for int k = 1 to i - 1 do
            %Verifica le regine precedenti
            if S[k] == j or S[k] == j + 1 - k or S[k] == j - i + k then
                legal = false
        if legal then
            S[i] = j
            queens(n, S, i + 1)
```

16.4.2 Giro di cavallo

Si consideri una scacchera $n \times n$, lo scopo è trovare un giro di cavallo, un percorso di mosse valide del cavallo in modo che ogni casella venga visitata al più una volta.

Soluzione Matrice $n \times n$ le cui celle contengono 0 se la cella non è mai stata visitata e i se la cella è stata visitata al passo i-esimo. Il numero di soluzioni è $64! \approx 10^{89}$ ma ad ogni passo si hanno al massimo 8 caselle possibili e pertanto ne visito al più $8^{64} \approx 10^{57}$.

```
: boolean KnightTour(int [][] S, int i, int x, int y)
    %Se i = 64 ho fatto 63 mosse e ho completato un tour aperto
    if i == 64 then
        processSolution(S)
        return true
    Set C = moves(S, x, y)
    foreach c ∈ C do
        S[x][y] = i
        if CavalloS, i + 1, x + m_x[c], y + m_y[c] then
            return true
        S[x][y] = 0
    return false
```

```
: Set moves(int [][] S, int x, int y)
    Set C = Set()
    for int i = 1 to 8 do
        n_x = x + m_x[i]
        n_y = y + m_y[i]
        if 1 ≤ n_x ≤ 8 and 1 ≤ n_y ≤ 8 and S[n_x][n_y] == 0 then
            C.insert(i)
    return C
```

$m_x = \{-1, +1, +2, +2, +1, -1, -2, -2\}$
$m_y = \{-2, -2, -1, +1, +2, +2, +1, -1\}$

16.4.3 Sudoku

La soluzione proposta è molto veloce per $n = 9$, ma è possibile generalizzare per $n = k^2$ ed esistono tecniche euristiche per fissare altri numeri che possono risolvere completamente il problema o essere usate come pre-processamento.

```
: boolean sudoku(int [][] S, int i)
    if i == 81 then
        processSolution(S, n)
        return true
    int x = i  mod 9
    int y = ⌊i/9⌋
    Set C = moves(S, x, y)
    int old = S[x][y]
    foreach c ∈ C do
        S[x][y] = c
        if sudoku(S, i + 1) then
            return true
    S[x][y] = old
    return false
```

```
: Set moves(int [][] S, int x, int y)
    Set C = Set()
    if S[x][y] ≠ 0 then
        %numero pre-inserito
        C.insert(S[x][y])
    else
        %Verifica conflitti
        for int c = 1 to 9 do
            if check(S, x, y, c) then
                C.insert(c)
    return C
```

```
: boolean check(int [][] S, int x, int y, int c)
    for int j = 0 to 8 do
        if S[x][j] == c then
            return false %Controllo sulla colonna
        if S[j][y] == c then
            return false %Controllo sulla riga
    int b_x = ⌊x/3⌋
    int b_y = ⌊y/3⌋
    for int i_x = 0 to 2 do
        for int i_y = 0 to 2 do
            if S[b_x · 3 + i_x][b_y · 3 + i_y] == c then
                return false %Controllo sulla sottotabella
    return true
```