UNIVERSITÀ DEGLI STUDI DI MILANO
Dipartimento di Tecnologie dell'Informazione
Polo Didattico e di Ricerca di Crema

Progetto di algoritmi per induzione matematica

Dispensa del corso di
Progettazione e analisi di algoritmi

Federico Biancini

Sara Parsani

- Prof. Roberto Cordone -
– Anno Accademico 2007/08 –

Indice

Capitolo 1

Introduzione

Il progetto di algoritmi può essere affrontato attraverso differenti metodologie. Due approcci molto importanti e strettamente legati sono il *progetto per riduzione* e il *progetto per induzione*.
Il primo approccio opera come segue. Considerando un problema P da risolvere, esso è trasformabile nel problema Q, tramite la trasformazione polinomiale R, quando:

- da ogni istanza I_P del problema è possibile ricavare in tempo polinomiale rispetto a $|I_P|$ un'istanza I_Q del problema Q
- dalla soluzione S_Q di quest'ultimo, è possibile ricavare in tempo polinomiale rispetto a $|I_P|$ la soluzione S_P del problema di partenza.

$$P \preceq_R Q$$

In altre parole:

1. si passa dalle istanze di P alle istanze di Q,
$$I_P \longrightarrow I_Q \quad \forall I_P \in P$$
2. si risolve Q attraverso un opportuno algoritmo A_Q,
$$I_Q \longrightarrow_{A_Q} S_Q$$
3. si ottiene la soluzione di P dalla soluzione di Q.
$$S_Q \longrightarrow S_P$$

Se la trasformazione R e l'algoritmo di risoluzione A_Q sono entrambi polinomiali, l'insieme dei tre passaggi costituisce un *algoritmo A_P polinomiale per il problema P*, e questo algoritmo è stato *progettato per riduzione*. Il progetto per riduzione richiede, quindi, di saper risolvere il problema Q a cui

P viene ridotto.
È possibile che Q non sia un problema distinto da P, ma un sottoinsieme delle istanze del problema P iniziale. Ossia, la riduzione avviene dall'intero P (tutte le sue istanze) ad un suo sottoproblema. Per esempio, è possibile trasformare qualsiasi istanza di SAT in un'istanza di *3-SAT*. Purtroppo non si può sfruttare questa riduzione per progettare un algoritmo polinomiale perché non si conoscono algoritmi polinomiali per *3-SAT*.
Esistono, in ogni problema, delle istanze banali, cioè facili da risolvere. Se una trasformazione R fosse in grado di ricondurre qualsiasi altra istanza del problema alle sue istanze banali, questa consentirebbe di risolvere l'intero problema. Il progetto per induzione consiste nel ripetere questo processo per più trasformazioni successive. Si opera tramite i seguenti passi:

- si dividono le istanze di P in classi P_k associate a numeri naturali (l'indice di classe k può essere qualsiasi numero naturale caratteristico dell'istanza; in genere è legato alla dimensione dell'istanza, ma non coincide con essa).
- Si progetta un algoritmo risolutivo per le istanze di classe 0.
- Si costruisce una riduzione che riduce ogni istanza di classe $n > 0$ ad un'istanza di classe n' strettamente minore ($n' < n$).

A questo punto si ottiene automaticamente un algoritmo per l'intero problema. Infatti, data l'istanza di classe P_k:

1. se l'istanza è di classe $k = 0$, si applica l'algoritmo risolutivo sopra definito e si passa al punto 3.
2. Se l'istanza è di classe $k > 0$, si applica la riduzione e si torna al punto 1.
3. Si trasforma la soluzione del problema ridotto nella soluzione del problema iniziale.

Si porti come esempio di progetto per induzione l'ordinamento di un vettore $\boldsymbol{v}$. Ordinare un'istanza generica (cioè un generico vettore di interi) è un compito non elementare, ma se si considera il sottoinsieme costituito dai vettori di un solo elemento, esiste un algoritmo banale per risolverli, che consiste nel non fare nulla, dato che tali vettori sono già ordinati. Da cui:

1. *istanza di classe* 0: P_0 rappresenta il vettore contenente 1 elemento. Come detto, il suo ordinamento è banale.

2. *istanza di classe* $k > 0$:

   - P_k rappresenta il vettore di *k+1* elementi. È possibile eliminarne uno per ricondursi ad un'istanza di classe k.
   - ...
   - P_2 rappresenta il vettore contenente 3 elementi. Per ricondurre il problema all'istanza P_1 si elimina un elemento.
   - P_1 rappresenta il vettore contenete 2 elementi. Anche in questo caso si elimina un elemento per ricondursi all'istanza P_0.

3. *Recupero della soluzione.* Applicata la riduzione e ottenuta la soluzione dell'istanza di classe 0, si reinserisce in modo ordinato ciascun elemento eliminato. Si ottiene, quindi, l'ordinamento del vettore dato.

1.1 Progetto per induzione e per riduzione

Entrambi gli approcci si basano sull'uso di una trasformazione e di un algoritmo per il problema trasformato. L'induzione applica la trasformazione più volte (secondo la classe dell'istanza) producendo problemi che sono un sottoinsieme di quello di partenza, mentre la riduzione la applica un numero dato di volte (spesso una sola) producendo problemi distinti da quello di partenza.

1.2 La correttezza del progetto per induzione

Affinché questo procedimento funzioni, occorre che partendo da qualsiasi istanza si giunga sempre, dopo un opportuno numero di trasformazioni, a un'istanza di classe zero. Che così avvenga pare ovvio, ma è bene chiarire dove si fondi tale ovvietà. Alla base vi è la struttura stessa dei numeri naturali, in particolare uno dei cinque assiomi con cui il matematico Giuseppe Peano definì l'insieme dei numeri naturali $\mathbb{N}$.

1. L'insieme $\mathbb{N}$ dei numeri naturali non è vuoto: $0 \in \mathbb{N}$.
2. Esiste una funzione $s : \mathbb{N} \to \mathbb{N}$ che definisce per ogni $n \in \mathbb{N}$ uno e un solo successore:
$$n \in \mathbb{N} \Rightarrow \exists n' = s(n) \in \mathbb{N}.$$
3. Lo 0 non è successore di alcun numero naturale: $s(n) \neq 0$ per ogni $n \in \mathbb{N}$.

4. A numeri naturali diversi corrispondono successori diversi:
$$n \neq n' \Rightarrow s(n) \neq s(n').$$

5. *Se un sottoinsieme K di $\mathbb{N}$ contiene lo zero ed è tale che se un numero n appartiene a K anche il suo successore appartiene a K, allora $K = \mathbb{N}$ (Principio di induzione).*

$$\begin{cases} K \subseteq N \\ 0 \in N \Rightarrow K = N \\ n \in K \Rightarrow s(n) \in K \end{cases} \Rightarrow K = \mathbb{N}$$

Applicando tale approccio alla risoluzione di problemi, si consideri K come il sottoinsieme dei numeri naturali tale che l'algoritmo A risolve P_k, cioè tutte le istanze di classe k del problema P. Se A risolve tutte le istanze di P_0 e contiene una riduzione che consente, nell'ipotesi di saper risolvere tutte le istanze di P_k, di risolvere tutte le istanze di P_{k+1}, l'assioma garantisce che A risolve le istanze di P_k qualunque sia k, cioè risolve P.
È possibile anche sostituire il quinto assioma di Peano con una versione più forte: *se una proprietà vale per* 0 *e, quando vale per tutti i numeri naturali* $\leq n$*, vale anche per* $n+1$*, allora tale proprietà vale per tutti i numeri* $\forall n \in \mathbb{N}$. Si parla in tal caso di *induzione forte*.

Capitolo 2

Valutazione di polinomi

Dato un polinomio $p(x) = \sum_{i=0}^{n} a_i x^i = a_n x^n + a_{n-1} x^{n-1} + ... + a_0$, si vuole calcolarne il valore in x. Ogni istanza del problema è definita da una coppia $(p(\cdot), x)$. Volendo affrontare il problema con l'induzione matematica, si deve innanzitutto definire:

- come dividere l'insieme delle istanze in classi numerate;
- come risolvere le istanze di classe 0;
- come trasformare un'istanza di classe n in una o più di classe $n' < n$, in modo che la soluzione di quest'ultime consenta di ricavare quella della prima.

La classe n corrisponde all'insieme dei polinomi di grado n. Ad esempio:

$$p(x) = 2x^3 + 3x^2 + x + 2 \tag{2.1}$$

che è un polinomio di terzo grado, fa parte della classe P_3 del problema P. Un'istanza di classe n è rappresentata da un vettore di $n + 1$ coefficienti numerici a_i:

$$a_0 = 2, a_1 = 1, a_2 = 3, a_3 = 2$$

i	0	1	2	3
A	2	1	3	2

In tal modo, un algoritmo per il calcolo del valore di un polinomio in un punto ammette come parametri il vettore A, di estremi s e d, e il punto x in cui valutare il polinomio:

$$Valuta\_polinomio(A, s, d, x).$$

Si noti la flessibilità offerta dagli indici sinistro e destro del vettore, che consente di rappresentare con lo stesso vettore, combinando gli indici, diversi polinomi derivanti da quello dato, ossia le istanze dei sottoproblemi. Ad esempio, $s = 1$ e $d = 2$ corrispondono al polinomio $3x + 1$. La differenza degli indici definisce anche il grado del nuovo polinomio, che in questo caso vale: $d - s = 2 - 1 = 1$.
L'approccio induttivo suggerisce di ridurre l'istanza data ad una più piccola, la quale verrà poi ridotta ricorsivamente fino a giungere all'istanza di classe 0. Quest'ultima rappresenta il polinomio di grado zero, cioè una costante. Il problema così ridotto viene risolto restituendo il valore della costante, che è l'elemento di indice $s = d$ nel vettore A. Questo si può tradurre in molti algoritmi differenti, come si vedrà nelle sezioni seguenti.

2.1 Eliminazione del termine di grado massimo

Questo algoritmo riduce un polinomio di grado n in un polinomio di grado $n-1$ rimuovendo il termine di grado massimo $a_n x^n$. Il polinomio residuo ha grado $n-1$, e quindi corrisponde a un'istanza di classe $n-1$. L'algoritmo risulta così strutturato:

1. Le istanze di classe $k = 0$ sono risolte semplicemente restituendo il coefficiente a_0, cioè $A[s]$.

2. La riduzione avviene decrementando l'indice destro d del vettore A corrente. Ciò che si ottiene, applicando al polinomio d'esempio (2.1) è:

   $$p_{n-1}(x) = 3x^2 + x + 2$$

   La riduzione viene riapplicata fino ad ottenere la classe d'indice zero.

3. La soluzione del problema originario si ottiene ricostruendo in una variabile q il termine di grado massimo $q = a_n \cdot x^n$ prima assegnando il coefficiente a_n a q e poi moltiplicandolo per x tante volte quante il suo grado (n). Il polinomio rimanente, che ha grado $n-1$, è valutato tramite la stessa funzione *Valuta_polinomio1* e sommato alla variabile q.

   $$q(x) = a_n \cdot x^n + p_{n-1}(x) \rightarrow q(x) = p_n(x)$$

```
Valuta_polinomio1(A, s, d, x)
  if s = d                                  {polinomio di grado 0}
    then return A[s]
  else
    d' := d − 1                             {riduzione a polinomio
                                             di classe n − 1}
    p_1 := Valuta_polinomio1(A, s, d', x)   {calcolo di p_{n−1}(x)}
    q = A[d]
    for i = 1 to d − s do
      q := q · x                            {calcolo di a_n x^n}
    return p_1 + q                          {ricostruzione della
                                             soluzione}
```

Questa risoluzione non è molto efficiente, infatti, ha una complessità:

$$T(n) = \begin{cases} 1 & se \quad n = 0 \\ n + T(n-1) & se \quad n > 0 \end{cases} \Rightarrow T(n) \in \Theta(n^2)$$

Essa richiede, infatti, $\frac{n \cdot (n+1)}{2}$ moltiplicazioni ed n addizioni.

2.2 Rafforzamento dell'ipotesi induttiva

Il *paradosso di Polya* o *paradosso dell'inventore* afferma che risolvere un problema più complesso può essere più facile che risolvere un problema più semplice. In particolare, può dar luogo ad un algoritmo computazionalmente migliore. Applicato allo studio dei polinomi, tale approccio prevede non solo la valutazione del polinomio $p(\cdot)$ in x, ma anche il calcolo di x^n, dove n è il grado del polinomio $p(\cdot)$. Anche in questo caso le classi di indice k corrispondono al grado del polinomio.

1. Le istanze di classe $k = 0$ restituiscono insieme al valore del coefficiente a_0, cioè $A[s]$, il valore della potenza di grado 0 valutata in x, cioè x^0, che è sempre 1.

2. La riduzione, che funziona esattamente come prima rimuovendo il termine di grado massimo, computa l'istanza in una classe strettamente inferiore, facendo decrementare l'indice *d*.

3. La soluzione del problema originario si ottiene in modo simile al caso precedente: si ricompone cioè nella variabile q il termine di grado massimo e gli si somma il polinomio residuo p_1, che viene determinato dalla

funzione *Valuta_polinomio2* applicata al problema ridotto. Si noti che non è più necessario moltiplicare n volte q per x, dato che basta moltiplicarla una volta per x e una per x^{n-1}, il cui valore y viene calcolato e restituito da *Valuta_polinomio2*. Rimane da calcolare x^n che si ottiene moltiplicando $y = x^{n-1}$ per x.

Valuta_polinomio2(A, s, d, x)
  if $s = d$
    then return $(A[s], 1)$ $\{p_0(x) = a_0 \text{ e } x^0 = 1\}$
  else
    $d' = d - 1$
    $(p_1, y) := Valuta\_polinomio2(A, s, d', x)$
    $y = y \cdot x$
    $q = A[d] \cdot y$
    return (q, y)

Quindi nel caso generale si risolve un problema di dimensione $n - 1$:

$$T(n) = \begin{cases} 1 & se \quad n = 1 \\ T(n-1) + O(1) & se \quad n > 1 \end{cases}$$

Precisamente $T(n)$ contiene $2n$ moltiplicazioni ed n addizioni.

2.3 Regola di Horner

Un ulteriore approccio alla risoluzione dei polinomi è offerto dall'algoritmo di Horner. Dato il solito polinomio di esempio (2.1), si può pensare di ridurlo eliminando non il termine di grado massimo, ma quello di grado 0:

$$p_3(x) = (2x^3 + 3x^2 + x) + 2$$

per poi sviluppare il polinomio $(2x^3 + 3x^2 + x)$ dopo aver raccolto una x fra gli addendi, e quindi riducendo il grado del polinomio stesso.

$$p_n(x) = x \cdot p_{n-1}(x) + a_0$$

Si associa sempre all'indice delle classi k il grado del polinomio.

1. Per le istanze di classe 0 si definisce la soluzione restituendo il valore del termine noto $a_n = A[d]$.

2. La riduzione di un'istanza di classe $k > 0$ ad una classe strettamente inferiore avviene facendo incrementare l'indice s (si noti la differenza rispetto ai primi due casi in cui si decrementava d).

3. Si ricostruisce la soluzione del problema originario moltiplicando per x il polinomio residuo p_{n-1}, che è la soluzione del problema ridotto restituita da *Valuta_polinomio3* e sommando il termine noto.

Valuta_polinomio3(A, s, d, x)
  if $s = d$
    then return $A[s]$
  else
    $s' = s + 1$
    $p_1 := Valuta\_polinomio3(A, s', d, x)$
    return $p_1 \cdot x + A[s]$

La complessità asintotica coincide con quella dell'algoritmo 2; infatti:

$$T(n) = \begin{cases} 1 & se \quad n = 1 \\ T(n-1) + O(1) & se \quad n > 1 \end{cases} \Rightarrow T(n) \in \Theta(n)$$

Precisamente il metodo di Horner richiede n addizioni ed n moltiplicazioni.

2.4 Induzione forte

L'induzione forte consente di dividere il polinomio $p(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_0$ in polinomi di qualsiasi grado inferiore a quello di origine anziché di grado immediatamente inferiore. Il polinomio di esempio $p_3(x) = 2x^3 + 3x^2 + x + 2$ può essere scomposto nei binomi $2x^3 + 3x^2$ e $x + 2$, i quali possono essere trasformati in polinomi di grado inferiore rispetto a p_3 raccogliendo un x^2 nel primo binomio di scomposizione. In generale, un polinomio viene suddiviso nel seguente modo:

$$p_n(x) = x^{\lfloor \frac{n}{2} \rfloor + 1} \cdot p_{\lfloor \frac{n}{2} \rfloor}(x) + q_{\lfloor \frac{n}{2} \rfloor}(x)$$

L'indice di classe k è associato al grado di ciascun polinomio ridotto.

1. L'istanza di classe 0 restituisce il valore del termine noto.

2. La riduzione di un'istanza ad una classe strettamente inferiore avviene dividendo il vettore A, di estremi (s, d) in due sottovettori con estremi rispettivamente:

$$(s, \lfloor \tfrac{d+s}{2} \rfloor) \text{ e } (\lfloor \tfrac{d+s}{2} \rfloor + 1, d).$$

3. Si ricostruisce nella variabile q il valore della potenza di x di ordine $\lfloor \frac{n}{2} \rfloor + 1$. Il $+1$ si ottiene con l'assegnamento $q = x$ effettuato prima del ciclo **for**. Si moltiplica q per il valore in x del primo polinomio e vi si somma il valore in x del secondo per ottenere la soluzione del problema originario.

Valuta_polinomio4(A, s, d, x)
if $s = d$
then return $A[s]$
else
$p_1 := Valuta\_polinomio4(A, s, \lfloor \frac{d+s}{2} \rfloor, x)$
$p_2 := Valuta\_polinomio4(A, \lfloor \frac{d+s}{2} \rfloor + 1, d, x)$
$q = x$
for $i = 1$ **to** $\lfloor \frac{d-s}{2} \rfloor$ **do**
$q := q \cdot x$
return $q \cdot p_1 + p_2$

Nel caso generale si risolvono due problemi di dimensione $\frac{n}{2}$, il ciclo **for** è eseguito $\frac{n}{2}$ volte e, infine, vi sono le operazioni del **return** e l'assegnamento di q, il che porta ad una complessità:

$$T(n) = \begin{cases} 1 & se \quad n = 1 \\ 2T(\frac{n}{2}) + \frac{n}{2} + 3 & se \quad n > 1 \end{cases} \Rightarrow T(n) \in \Theta(n \log n)$$

La complessità può scendere a $\Theta(n)$ se si rafforza l'ipotesi induttiva in modo tale che il sottoproblema restituisca oltre a $p(x)$ anche $x^{\lfloor \frac{d-s}{2} \rfloor}$. Nel caso base si tratta banalmente di restituire il termine noto e $x^0 = 1$.

2.5 Separazione dei coefficienti

L' ultimo approccio al problema dei polinomi, presentato in questa dispensa, si basa sulla separazione dei termini di grado pari e dispari. Si consideri il seguente polinomio:

$$p(x) = 2 + x + 3x^2 + 2x^3 + x^4$$

e il corrispondente vettore:

posizione	0	1	2	3	4
A	2	1	3	2	1

Per sfruttare l'idea di separazione dei coefficienti, si devono creare due vettori che conterranno i coefficienti del polinomio rispettivamente con indice pari e dispari:

posizione	0 2 4	1 3
A	2 3 1	1 2

da cui:

posizione	0 1 2
B	2 3 1

posizione	0 1
C	1 2

Dal punto di vista matematico la separazione dei coefficienti consiste nel suddividere $p_A(x)$ come segue:

$$p_A(x) = (2 + 3x^2 + x^4) + (x + 2x^3)$$

per poi raccogliere x fra i coefficienti di grado dispari.

$$p_A(x) = (2 + 3x^2 + x^4) + x(1 + 2x^2)$$

e sostituire $t = x^2$ riducendo il grado dei polinomi, che risultano della forma:

$$(2 + 3t + t^2) + x(1 + 2t) = \pi_2^{'}(t) + x \cdot \pi_2^{''}(t)$$

cioè una combinazione dei polinomi $\pi^{'}$ e $\pi^{''}$ che corrispondono ai vettori B e C.
Come in tutti i casi precedenti, si associa all'indice k il grado di ciascun polinomio e:

1. la classe 0 restituisce il valore del termine noto.
2. Le operazioni di riduzione sopra descritte si sviluppano come segue. Il primo ciclo **while** con variabili i e *contb* servono per creare il vettore dei coefficienti di indice pari (che s sia pari lo si verifica attraverso s *mod* 2).

Nel ciclo **while**, l'indice i che scorre il vettore A, viene incrementato di due a partire dalla posizione definita per s per individuare tutti i coefficienti di indice pari, i quali vengono memorizzati in posizioni successive del vettore B. Allo stesso modo si opera per la creazione del vettore C, che contiene i coefficienti di indice dispari.

3. La soluzione del problema originario viene ricostruita sommando al risultato della chiamata *Valuta_polinomio5* sul vettore B valutato in x^2 il risultato della chiamata *Valuta_polinomio5* sul vettore C valutato in x^2 moltiplicato per x.

```
Valuta_polinomio5(A, s, d, x)
if s = d
  then return A[s]
else
  i := s; i_b := 0;                          { raccoglie i termini di grado pari }
  while i ≤ d do
     B[i_b] := A[i];
     i := i + 2;
     i_b := i_b + 1;
  end while
  i := s + 1; i_c := 0;                      { raccoglie i termini di grado dispari }
  while i ≤ d do
     C[i_c] := A[i]
     i = i + 2
     i_c := i_c + 1;
  end while
  t = x · x
  p_1 = Valuta_polinomio5(B, 0, i_b − 1, t)
  p_2 = Valuta_polinomio5(C, 1, i_c − 1, t)
  return p_1 + x · p_2
```

Capitolo 3

Ordinamento

Un altro classico problema che può essere risolto per induzione è il problema dell'ordinamento: data una sequenza di n numeri, si vuole ottenere una sequenza con gli stessi numeri ordinati per valori non decrescenti. Consideriamo quattro (in realtà cinque) algoritmi tutti basati sull'induzione.

L'insieme P delle istanze viene sempre partizionato in classi P_n in base al numero n di elementi dell'insieme da ordinare. In tutti gli algoritmi seguenti, inoltre, il *caso base* riguarda vettori che contengono un solo elemento o nessuno e l'algoritmo risolutivo consiste nel non far nulla, dato che tali vettori sono già ordinati.

L'idea induttiva è, secondo i casi, trasformare l'istanza di classe n in un'istanza di classe $n-1$ (colonna sinistra della tabella seguente) oppure ridurla in due istanze di classe n' e n'' con $n' < n$ e $n'' < n$ (colonna destra). In entrambi i casi, si possono costruire le istanze ridotte in modo molto semplice e con sforzo basso, pagando poi uno sforzo più alto al momento di ricostruire la soluzione del problema complessivo a partire da quelle dei problemi ridotti (prima riga della tabella), oppure dedicare uno sforzo più alto alla costruzione delle istanze ridotte e ricavare poi uno sforzo inferiore la soluzione del problema complessivo da quelle dei problemi ridotti (seconda riga).

	Riduzione	
	$I_n \preceq_T I_{n-1}$	$I_n \preceq_R I_{n'}, I_{n''}$ con $n' < n$ e $n'' < n$
Sforzo basso nel dividere Sforzo alto nel ricombinare	*Insertion Sort*	*Merge Sort*
Sforzo alto nel dividere Sforzo basso nel ricombinare	*Selection Sort*	*Quick Sort*

3.1 Insertion Sort

L'algoritmo di *Insertion Sort* trasforma l'insieme di numeri I_n dato in un sottoinsieme I_{n-1} di $n-1$ elementi, rimuovendone l'ultimo (oppure il primo, oppure uno qualsiasi). Applicheremo ancora la rappresentazione dell'insieme con un vettore A e due indici estremi s e d. Questo consente di eliminare l'ultimo elemento semplicemente decrementando d. Se si volesse rimuovere il primo, basterebbe incrementare s; per rimuoverne un altro, occorrerebbe scambiarne la posizione con l'ultimo o con il primo e procedere come sopra.

$$1 \quad 7 \quad 2 \quad 4 \quad 5 \quad | \quad 6$$

Si risolve poi l'istanza I_{n-1} attraverso lo stesso algoritmo, ottenendo:

$$1 \quad 2 \quad 4 \quad 5 \quad 7 \quad | \quad 6$$

Per ricavare la soluzione S_n da S_{n-1}, l'elemento 6 viene collocato nella posizione corretta nella sottosequenza ordinata, cioè fra il 5 e il 7.

$$1 \quad 2 \quad 4 \quad 5 \quad 6 \quad 7$$

InsertionSort(A, s, d)

if $s < d$ { 0 o 1 elemento: vettore già ordinato }

then

$d' := d - 1$; { si estrae l'ultimo elemento }

InsertionSort(A, s, d') {si risolve il problema ridotto }

AddSorted$(A[d], A, s, d')$ { si inserisce $A[d]$ nella posizione corretta tra s e d' }

Questo non è il classico pseudocodice dell'algoritmo *Insertion Sort*, ma gli è del tutto equivalente. Infatti, passando dalla forma ricorsiva a quella iterativa con l'uso di uno stack sul quale si pongono gli elementi via via rimossi dalle istanze:

InsertionSort(A, s, d)

stack = *s_create*()

for $d' = d - 1$ **downto** s **do**

stack = *s_push*(stack, $A[d'+1]$) {inserimento elementi nello stack}

for $d' = s$ **to** $d-1$ **do**

x = *s_pop*(stack) {recupero degli elementi}

AddSorted(x, A, s, d') { ricostruisce A estraendo x dallo stack e inserendolo nella posizione corretta fra s e d'}

s_destroy(stack)

Le successive iterazioni del primo ciclo producono la situazione seguente, che corrisponde alle successive chiamate della versione ricorsiva:

$\boxed{2\ \ 7\ \ 1\ \ 4\ \ 5\ \ 6}$	
Vettore	*Stack*
2 7 1 4 5	6
2 7 1 4	5 6
2 7 1	4 5 6
...	
2	7 1 4 5 6

Quando $d' = s$ si è nel caso base, che viene risolto non facendo nulla. Gli elementi vengono poi via via ricollocati nel vettore A in modo ordinato estraendoli dallo stack con la funzione *s_pop*.

Questa versione iterativa può essere semplificata utilizzando la parte finale dello stesso vettore A come stack. Infatti, ridurre l'indice d significa contemporaneamente estrarre l'elemento $A[d]$ dal vettore e porlo in cima allo stack (anche se, in realtà, ovviamente l'elemento rimane dov'è). Il risultato di questa scelta è che le procedure di creazione e distruzione dello stack sono inutili e quelle di inserimento ed estrazione dallo stack si riducono all'aggiornamento di d'.

Lo pseudocodice finale è quello classico:

InsertionSort(A, s, d)

for $d' = s$ **to** $d - 1$ **do**

$AddSorted(A[d'+1], A, s, d')$ { ricostruisce A inserendo $A[d'+1]$ nella posizione corretta fra s e d'}

3.2 Selection Sort

L'algoritmo di *Selection Sort* concentra lo sforzo computazionale sulla costruzione dell'istanza ridotta: invece di estrarre un elemento qualsiasi, estrae l'elemento massimo. Questo garantisce che l'istanza ridotta contenga i valori più piccoli dell'istanza originale e che l'elemento rimosso vada banalmente aggiunto in posizione finale, senza bisogno di cercare la posizione corretta.

SelectionSort(A, s, d)

if $s < d$ { 0 o 1 elemento: vettore già ordinato }

then

$j_{\max} := FindMax(A, s, d)$; { {scorre il vettore e trova l'elemento massimo }

$Exchange(A, j_{\max}, d$; { sposta l'elemento massimo in fondo al vettore }

$d' := d - 1$; { estrae l'ultimo elemento }

$SelectionSort(A, s, d')$ { risolve il problema ridotto }

$Append(A[d], A, s, d')$ { accoda $A[d]$ in fondo al vettore A }

Anche questa versione non è quella generalmente presentata nei libri di testo, ma le è equivalente. Infatti, passando alla forma iterativa con l'uso di uno stack:

SelectionSort(A, s, d)

stack = $s\_create()$;

for $d' = d - 1$ **downto** s **do**

$j_{\max} = FindMax(A, s, d'+1)$ { trova l'elemento massimo }

$Exchange(A, j_{\max}, d'+1$; { sposta l'elemento massimo

in fondo al vettore }

$s\_push(A[d'+1]$,stack);

for $d' = s$ **to** $d - 1$ **do**

$x = s\_pop$(stack) { estrae l'elemento in cima allo stack }

$Append(x, A, s, d')$ { accoda x in fondo al vettore A }

$s\_destroy$(stack)

Gli elementi nello stack risultano già ordinati quando la trasformazione giunge al caso base I_1.

2 7 1 4 5 6	
Vettore	*Stack*
2 6 1 4 5	7
2 5 1 4	6 7
2 4 1	5 6 7
...	
1	2 4 5 6 7

A questo punto è sufficiente estrarre dalla cima dello stack un elemento alla volta e riporlo alla fine del vettore.

Come per l'*Insertion Sort*, anche in questo caso è possibile utilizzare la parte finale del vettore A come stack. La rimozione dell'ultimo elemento (dopo aver spostato in tale posizione l'elemento massimo) avviene ancora decrementando d.

1 7 2 4 5 6
1 6 2 4 5 *7*
1 5 2 4 *6 7*
1 4 2 *5 6 7*
1 2 *4 5 6 7*
1 *2 4 5 6 7*

Ottenuta la soluzione per l'istanza ridotta, si riaggiunge l'elemento rimosso in coda alla soluzione semplicemente incrementando l'indice destro. Di conseguenza, le operazioni di costruzione e distruzione dello stack sono inutili, quelle di inserimento ed estrazione sono banali e lo pseudocodice iterativo si riduce a:

SelectionSort(A, s, d)

for $d' = d - 1$ **downto** s **do**

$j_{\max} = FindMax(A, s, d' + 1)$ { trova l'elemento massimo }

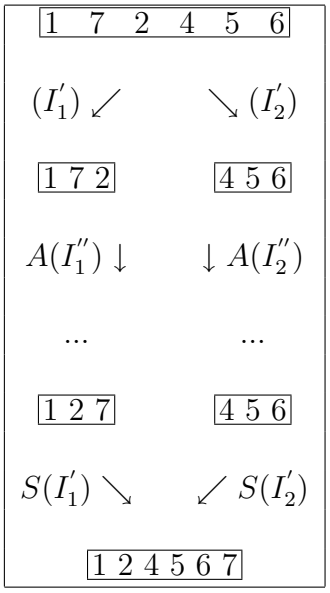
$Exchange(A, j_{\max}, d' + 1;$ { sposta l'elemento massimo in fondo al vettore }

Nota L'algoritmo *Heap Sort* coincide concettualmente con l'algoritmo *Selection Sort*, ma implementa la funzione *FindMax()* in tempo costante attraverso uno *heap*. Ciò significa che l'intero algoritmo va preceduto da una procedura che trasformi il vettore A in uno *heap* e che la *Exchange* e la rimozione dell'ultimo elemento devono essere seguite da una procedura che ristabilisca la proprietà *heap*.

3.3 Merge Sort

L'algoritmo di *Merge Sort* riduce l'istanza data a due istanze di classe inferiore determinando l'indice medio del vettore e considerando le due metà del vettore così individuate. Le due istanze hanno quindi classe $\lfloor n/2 \rfloor$ (o $\lfloor n/2 \rfloor + 1$).

Il fulcro dell'algoritmo *Merge Sort* è nel passo che ricava la soluzione complessiva a partire da quelle ridotte, attraverso la funzione *Merge* che fonde due sottosequenze ordinate in una sequenza complessiva ordinata.



MergeSort(A, s, d)

if $s < d$ { 0 o 1 elemento: vettore già ordinato }

then

$q = \lfloor \frac{s+d}{2} \rfloor$; {identificazione del punto medio}

$MergeSort(A, s, q)$;

$MergeSort(A, q+1, d)$;

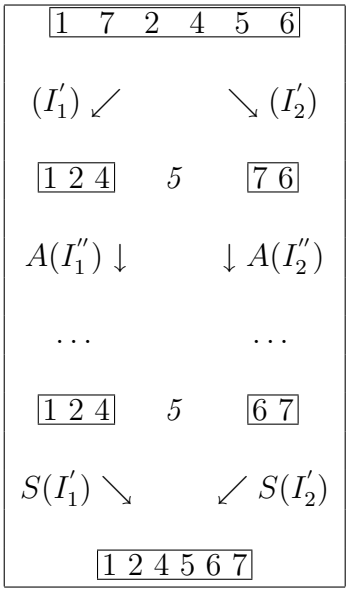
$Merge(A, s, q, d)$;

L'algoritmo *Merge Sort* si può implementare iterativamente, ma la realizzazione è molto più complessa e meno leggibile, per cui non verrà qui presentata.

3.4 Quick Sort

L'algoritmo *Quick Sort* riduce l'istanza del problema a due istanze di classe inferiore scegliendo un elemento come *pivot* e ponendo nella prima istanza gli elementi che minori del *pivot*, nella seconda gli elementi maggiori. La funzione *Partiziona* definito l'elemento di pivot in A, preleva gli altri e li ripone nelle posizioni comprese tra s e $q-1$ se minori del pivot, nelle posizioni

tra $q+1$ e d se maggiori e restituisce q, che non è definito a priori. Poi ordina ciascuna sottosequenza con lo stesso algoritmo. Infine, per ricostruire la soluzione, basta banalmente concatenare le soluzioni dei due sottoproblemi, con l'elemento *pivot* in mezzo.



QuickSort(A, s, d)

if $s < d$ {caso base}

then

$q = \lfloor \frac{s+d}{2} \rfloor$; {identificazione del punto medio}

$q :=$ *Partiziona*(A, s, d); {suddivide nelle due istanze}

QuickSort(A, s, q);

QuickSort$(A, q+1, d)$;

L'algoritmo di *Quick Sort* presenta uno sforzo alto nella fase di divisione, a causa della chiamata alla funzione *Partiziona*. Come per l'algoritmo di *Merge Sort* è possibile definire una versione iterativa dell'algoritmo *Quick Sort*, non presentata qui a causa della sua complessità e scarsa leggibilità.

Capitolo 4

Celebrity

Si suppone che in un insieme di n persone sia definita una relazione di *conoscenza* c che può anche essere asimmetrica (a conosce b, ma b non conosce a). Si definisce *celebrità* (rispetto all'insieme dato) una persona che è conosciuta da tutti e non conosce nessuno.

Se rappresentiamo la relazione di conoscenza con un grafo orientato i cui nodi rappresentano le persone e gli archi le relazioni di conoscenza, una celebrità corrisponde a un nodo con $n-1$ archi in ingresso, e 0 archi in uscita.

È facile dimostrare che un insieme ammette al massimo una celebrità: se ce ne fossero due, dovrebbero per definizione conoscersi (tutti conoscono le celebrità) e insieme ignorarsi a vicenda (la celebrità non conosce nessuno), e questo è assurdo. Nell'esempio di Figura 4.1, esiste una sola celebrità: il vertice E. In altri casi potrebbero non esistere celebrità (ad esempio se tutti si conoscono o se nessuno conosce nessuno).

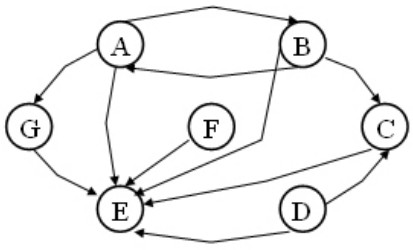


Figura 4.1: Grafo orientato

La stessa relazione può essere rappresentata attraverso una matrice di adiacenza le cui righe e colonne corrispondono agli individui. La matrice contiene un 1 nella cella (i, j) se l'individuo i conosce l'individuo j, uno 0 altrimenti. La Tabella 4.1 riporta la matrice corrispondente all'esempio

in Figura 4.1. Per ciascun nodo, in riga viene riportata anche la somma degli elementi della riga stessa (pari al numero di archi in uscita), così come in colonna la somma degli elementi della colonna stessa (pari al numero di archi in entrata). Ciascuna somma di riga rappresenta il numero di persone conosciute dall'individuo in esame, mentre ciascuna somma di colonna rappresenta il numero di persone da cui l'individuo è conosciuto. Quindi una persona è una celebrità, se e solo se la sua somma di riga è nulla e quella di colonna è pari a $n - 1$. La ricerca della celebrità sul grafo e sulla matrice di adiacenza comporta una complessità quadratica.

	A	B	C	D	E	F	G	
A	-	1	0	0	1	1	1	4
B	1	-	1	0	1	0	0	3
C	0	0	-	0	1	0	0	1
D	0	0	1	-	1	0	0	2
E	0	0	0	0	-	0	0	0
F	0	0	1	0	1	-	0	2
G	0	0	0	0	1	0	-	1
	1	1	3	0	6	1	1	

Tabella 4.1: Matrice di adiacenza

Affrontiamo il problema col metodo induttivo. Partizioniamo le istanze del problema in classi numerate secondo il numero di individui dell'insieme.

- il caso base corrisponde alle istanze di una sola persona: ovviamente, essa non conosce nessun altro ed è conosciuta da tutti gli altri, per cui è una celebrità;

- l'insieme può essere ridotto da n a $n - 1$ persone eliminandone una;

Per completare la trasformazione $I_n \preceq_T I_{n-1}$, però, bisogna ricostruire la soluzione del problema originario a partire da quella del problema ridotto. Ciò significa aggiungere un individuo precedentemente eliminato e ricostruire la soluzione a partire da quella del problema ridotto.

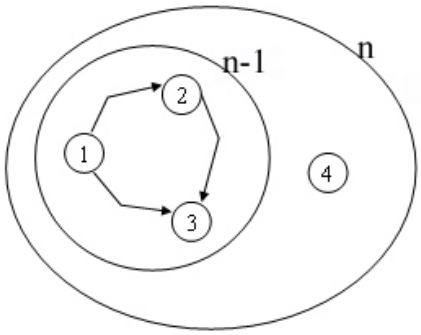


Figura 4.2: Aggiunta dell'$n-$esima persona.

La soluzione del problema ridotto può ricadere in due casi:

1. la celebrità del problema ridotto è l'individuo i^*_{n-1} (nell'esempio di Figura 4.2, $i^*_{n-1} = 3$)

2. il problema ridotto non ha celebrità (per convenzione, $i^*_{n-1} = 0$)

Partendo dal presupposto che il nodo eliminato ($n = 4$) sia del tutto generico, la sua aggiunta può comportare tre casi possibili:

1. I_n ammette come celebrità una delle prime $n - 1$ persone;

2. I_n ammette come celebrità l'individuo aggiunto ($i^*_n = n$);

3. I_n non ammette celebrità ($i^*_n = 0$).

Si noti anche che nel primo dei tre casi la celebrità deve essere tale anche per il problema ridotto, per cui $i^*_n = i^*_{n-1}$. Non è possibile che l'arrivo di un nuovo individuo renda celebre una persona che prima non lo era. Questo significa che il primo caso è impossibile se il problema ridotto non ammette celebrità.

Nel problema di Figura 4.2, l'aggiunta del nodo $n = 4$ comporta tre casi possibili:

1. se 4 conosce 3 e 3 non conosce 4, la vecchia celebrità rimane tale per il problema complessivo, cioè $i^*_n = i^*_{n-1} = 4$;

2. se 4 non conosce nessuno e tutti conoscono 4, la celebrità è il nuovo elemento, cioè $i^*_n = n = 4$;

3. altrimenti non c'è celebrità.

La Tabella 4.2 riassume i casi possibili e le condizioni che danno loro luogo. Verificando tali condizioni si può estendere la soluzione di I_{n-1} a quella di I_n.

	I_{n-1}	
I_n	$\exists i^*$	$\nexists i^*$
$\exists i_n^* = i_{n-1}^*$	$c_{ni_{n-1}^*}$ = TRUE AND $c_{i_{n-1}^* n}$ = FALSE	IMPOSSIBILE
$\exists i_n^* \neq i_{n-1}^*$	c_{in} = TRUE $\forall i$ AND c_{ni} = FALSE $\forall i$	c_{in} = TRUE $\forall i$ AND c_{ni} = FALSE $\forall i$
$\nexists i_n^*$	else ...	else ...

Tabella 4.2: Tabella dei casi possibili nella trasformazione $I_n \preceq_T I_{n-1}$

Sulla base della Tabella 4.2, l'estensione della soluzione da S_{n-1} a S_n costa $\Theta(n)$, perché la seconda riga della tabella comporta (in entrambe le colonne, cioè qualunque sia la soluzione del problema ridotto) di scorrere tutti gli individui. La complessità di questo algoritmo è $\Theta(n^2)$.

Tuttavia, non si è costretti a ridurre l'istanza del problema eliminando un nodo generico: è possibile sceglierne uno specifico. Siccome i casi fastidiosi sono quelli della seconda riga, in cui il nuovo individuo diventa celebrità, dobbiamo fare in modo che essi non si presentino, cioè dobbiamo eliminare ogni volta un individuo che sicuramente non è una celebrità. In tal modo è sicuro che l'elemento aggiunto nella fase di ricombinazione non può essere una celebrità, e quindi la seconda riga non si verifica mai e la relativa condizione non va testata. Si eliminano così tutti i casi della seconda riga, e quindi tutte le verifiche di complessità $\Theta(n)$.

Per individuare una persona che non sia una celebrità, basta estrarre una qualsiasi coppia di individui (i, j). Si danno infatti due casi:

1. i conosce j: allora i non può essere una celebrità (si noti che anche j potrebbe non esserlo, ma per ora ci basta un individuo solo che non sia una celebrità)

2. i non conosce j: allora j non può essere una celebrità (come sopra, anche i potrebbe non esserlo, ma questo non conta)

Grazie a questo approccio, ad ogni passo di ricombinazione:

- se I_{n-1} non ammette celebrità, anche I_n non ne ammette;

- se I_{n-1} ammette una celebrità i^*, si testa in tempo $O(1)$ la condizione della prima riga, per concludere o che I_n ammette anch'esso la celebrità i^*, o che non ammette celebrità.

La complessità dell'intero algoritmo si riduce a $O(n)$.

Si utilizzi, come esempio, il grafo in Figura 4.1 e la relativa matrice di adiacenza. L'algoritmo induttivo procede nel modo seguente:

(A	B)	C	D	E	F	G
A	(B	C)	D	E	F	G
A	B	(C	D)	E	F	G
A	B	D	(C	E)	F	G
A	B	D	C	(E	F)	G
A	B	D	C	F	(E	G)
A	B	D	C	F	G	E

L'algoritmo scarta tutti gli individui tranne E, che è celebrità rispetto a se stesso, e potenziale celebrità rispetto all'intero insieme. Gli altri individui sicuramente non sono celebrità. Quindi, si proceda a reimmettere uno alla volta gli individui eliminati e si verifichi se conoscono E e se E non li conosce. Ad esempio, reimmettendo il nodo G, si testa se:

$$c_{GE} = TRUE \text{ e } c_{EG} = FALSE$$

Poiché la condizione è verificata, la reimmissione del nodo G soddisfa il test e $i_n^* = i_{n-1}^*$. Ovviamente il test va ripetuto su tutti gli individui reimmessi.

Nell'esempio seguente, nonostante venga individuata una celebrità nell'istanza I_1, non vengono superati i test in fase di trasformazione all'istanza I_2.

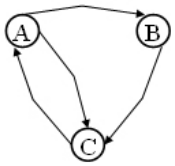


Figura 4.3: Istanza che non ammette celebrità

(A	B)	C
A	(B	C)
A	B	C

Si identifica come potenziale celebrità l'individuo C. La reimmissione dell'individuo B soddisfa il test, mentre quella del nodo A non lo soddisfa.

$$c_{AC} = TRUE \text{ ma } c_{CA} \neq FALSE$$

Quindi non vi è celebrità. Si noti che, non appena la celebrità potenziale smette di essere tale, si può interrompere l'esecuzione perché nessun altro può rivelarsi celebre.

Una possibile implementazione del problema della celebrità è illustrata dal seguente pseudocodice:

Celebrity(A, n, c)

if $n = 1$

then return $A[1]$;

else

if $c[a_{n-1}, a_n]$ **then** Exchange$(A, n-1, n)$;

{ Ora a_n non è una celebrità }

$n' := n$;

$a_{i^*} :=$ Celebrity(A, n', c);

if $c[i^*, a_n]$ **and not** $c[a_n, i^*]$

then return a_n

else return $\nexists i^*$

Capitolo 5

Sottomapping biunivoco massimo

Si consideri una funzione $f : A \to A$ che mappa un insieme finito A su se stesso, ossia ogni elemento di A è mappato su un elemento di A.

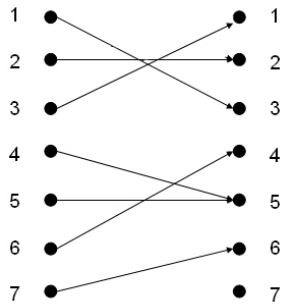


Figura 5.1: Rappresentazione del mapping fra elementi di un insieme

Si osservi la Figura 5.1: per semplicità, ad ogni elemento di A viene associato un numero naturale e il mapping viene rappresentato con un grafo bipartito le cui parti rappresentano copie dell'insieme A e la parte sinistra si mappa sulla destra. Il problema che si vuole risolvere, consiste nel trovare un sottoinsieme $S \subseteq A$ con il massimo numero di elementi tale che:

1. la funzione f mappi ogni elemento di S su un altro elemento di S;
2. ogni elemento di S ha esattamente un elemento in S, il quale è mappato in essa.

In altre parole, si vuole che $f(x)$ sia biunivoca su S, mentre in generale non lo è su A. Si osservi nuovamente la Figura 5.1, ed in particolare i nodi 1 - 3 - 5 a sinistra: essi sono mappabili sui nodi 1 - 3 - 5 a destra, rendendo la funzione $f(x)$ biunivoca in tale sottoinsieme.
Per progettare un algoritmo per induzione, si identificano come classi di istanze quelle con insieme A di una data cardinalità.

1. Le istanze di classe uno contengono un solo elemento, che necessariamente si mappa su se stesso. Quindi la soluzione S è l'intero insieme.

2. La riduzione a istanze di classe inferiore avviene eliminando un elemento.

3. A questo punto occorre definire l'estensione della soluzione dal problema ridotto S_{n-1} a quella del problema completo S_n.

Ricondursi alla soluzione S_n presenta però in generale grosse difficoltà. Dato il mapping massimo per l'istanza I_{n-1}, il mapping massimo per l'istanza I_n, in cui viene aggiunto un nuovo elemento, può essere molto diverso. Ad esempio, se si passa da $A = \{1, 2\}$, dove c'è mapping $S = \{2\}$, a $A = \{1, 2, 3\}$, dove il mapping è $S = \{1, 3\}$, la soluzione cambia completamente:

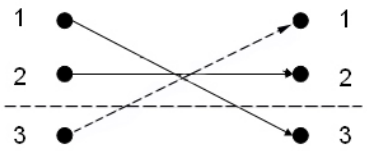


Figura 5.2: Esempio di passaggio di istanza

Il progetto per induzione permette però di scegliere il nodo con cui ridurre il problema. Una possibile soluzione può essere rimuovere, ad ogni passo, solo elementi che sicuramente non fanno parte del sottomapping biunivoco massimo. Elementi di questo tipo sono, ad esempio, quelli il cui nodo destro è privo di archi entranti, cioè quegli elementi tali che nessun elemento di A vi corrisponde. A questo punto è chiaro che la soluzione S_n deve coincidere con S_{n-1} perché nessuna soluzione può includere l'elemento n e d'altra parte ogni altra soluzione era già presente nel problema ridotto I_{n-1}. La soluzione diviene, allora, quella di eliminare i nodi di destra che non ricevono archi e contemporaneamente i corrispondenti nodi di sinistra. Ciò fatto, è chiaro che spariscono anche gli archi uscenti da questi ultimi, il che riduce il grado di altri nodi di destra, eventualmente portando a nuove eliminazioni. Nel caso

in esame:

- si elimina il nodo 7 di destra, con esso il nodo 7 di sinistra, e, poiché $f(7) = 6$, il nodo 6 vede diminuire il suo grado entrante.
- Poiché il nodo 6 a destra ha grado entrante nullo lo si elimina, e con esso si elimina il relativo 6 di sinistra. Dato che $f(6) = 4$, il nodo 4 vede diminuire il suo grado entrante.
- Il nodo 4 di destra viene eliminato poiché ha grado entrante nullo. Si elimina, quindi, il relativo 4 di sinistra e, dato $f(4) = 5$, il nodo 5 vede diminuire il suo grado entrante.
- Dato che il grado entrante di 5 è ancora positivo, essendo dato $f(5) = 5$, l'algoritmo si arresta.

Si è arrivati ad un grafo in cui nel lato destro non ci sono più nodi privi di archi entranti, questo significa che l'intero grafo è un mapping biunivoco, cioè che $S_k = A_k$. Questo è il caso base. Si noti la differenza rispetto agli algoritmi precedenti in cui il caso base corrispondeva a $n = 0$ o $n = 1$. Questo problema ammette casi base distribuiti su tutte le classi. L'importante, però, è che l'induzione porta sicuramente qualsiasi problema sino a un caso base, al limite di classe 0, dove $S_k = A_k$.
L'insieme S ricercato è costituito dai nodi $\{1, 2, 3, 5\}$.
Il relativo pseudocodice, dove E mappa l'insieme dei lati fra i nodi delle due sponde ed A è l'insieme base del mapping, è:

```
Mapping(A, E)
if |A| = 1
    then return A
else
   e* = CercaElementoSenzaArchiEntranti(S)
   if e* = ∅
     then return A
    else
     A' := A\{e*}
     E' := E\Δ_e*
     S' := Mapping(A', E')
     return S'
```

Per non dare una definizione ricorsiva dell'algoritmo, è possibile conservare in un vettore il grado entrante di ciascun elemento i dell'insieme A.

Inizialmente, $c[i]$ ha un valore pari al numero di elementi che sono mappati in i, ossia il numero di archi che i riceve; questa operazione, può essere realizzata in n passi, scorrendo l'array di elementi. Quindi si scorre il vettore: tutti gli elementi che presentano il contatore a zero, vengono posti in una coda. Quindi si scorre la coda: ad ogni passo, viene rimosso un elemento j dalla coda (e dall'insieme) e decrementato il contatore dell'elemento $c[f(j)]$. Se $c[f(j)]$ viene azzerato, $f(j)$ viene collocato in fondo alla coda. L'algoritmo termina quando la coda è vuota.

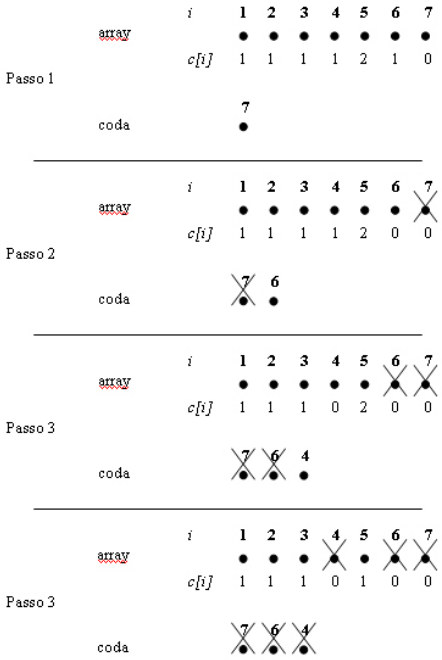


Figura 5.3: Funzionamento dell'algortimo di mapping

La Figura 5.3 consente di osservare l'effettivo funzionamento tramite l'appoggio di un contatore e di una coda. La complessità di tale algoritmo è legata alle parti di inizializzazione, che richiedono $O(n)$ operazioni. Ogni elemento può essere messo nella coda al massimo una sola volta e i passi richiesti per la rimozione di un elemento dalla coda richiedono tempo costante. Il numero totale di operazioni perciò è $O(n)$.

Capitolo 6

Ordinamento topologico

Si consideri un grafo orientato aciclico $G = (N, A)$ con n nodi, per comodità numerati da $1, ..., n$, come nell'esempio in Figura 6.1.

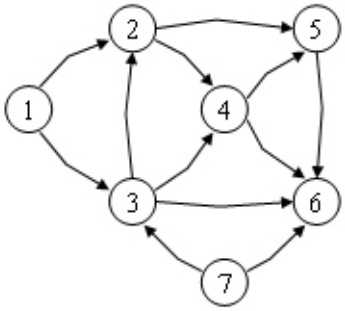


Figura 6.1: Grafo orientato aciclico

Si vuole determinare un ordinamento topologico per il grafo $G = (N, A)$. Per *ordinamento topologico* si intende una permutazione dei nodi del grafo, tale che non vi siano archi la cui testa preceda la coda nell'ordinamento. Graficamente significa passare dalla struttura di Figura 6.2:

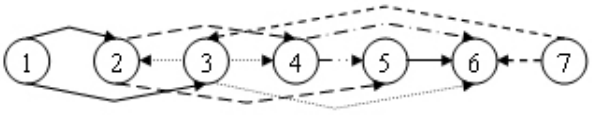


Figura 6.2: Struttura sequenziale rispetto i nodi

a quella di Figura 6.3 priva di archi all'indietro:

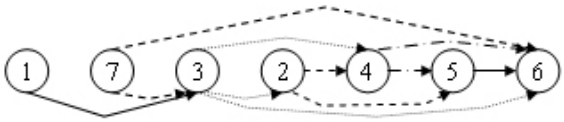


Figura 6.3: Ordinamento topologico del grafo in fig. 6.1

Ovviamente esiste un ordinamento topologico se e solo se il grafo è aciclico.
Le applicazioni di questo problema sono molteplici. Ad esempio, si tratta di descrivere la dipendenza fra mansioni: in un progetto complesso alcune operazioni dipendono dallo svolgimento di altre e, quindi, non possono essere avviate fin tanto che le mansioni precedenti non sono state portate a termine. Note le dipendenze fra le mansioni, l'obiettivo consiste nell'ordinarle in un modo ammissibile.
Per risolvere il problema per induzione, si suddivide il problema in classi numerate in base al numero dei nodi.

1. Il caso base (istanze di classe 0) corrisponde a grafi di un solo nodo, i quali, evidentemente, risultano già ordinati.

2. A questo punto si tratta di ridurre i grafi da n nodi a grafi da $n-1$ nodi. L'***ipotesi di induzione*** è quindi la seguente: *si conosce come risolvere il problema per qualsiasi grafo con* $n-1$ *nodi.*

3. Se $n > 1$, occorre un algoritmo che recuperi il nodo eliminato e lo inserisca correttamente entro la soluzione del problema ridotto, cioè entro una sequenza di $n-1$ nodi ordinati topologicamente.

Il problema che si riscontra nel ricostruire la soluzione è che non sempre è possibile aggiungere un nodo ad un ordinamento dato. Si può verificare che in qualsiasi posizione un nodo rimosso venga reinserito, il nodo abbia archi uscenti o entranti la cui testa precede la coda nell'ordinamento dato.

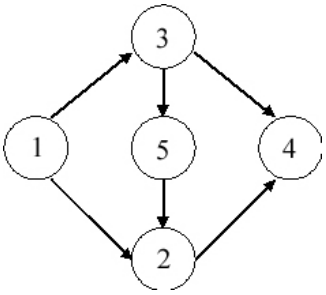


Figura 6.4: Esempio di rimozione casuale

Se dall'esempio di Figura 6.4 viene rimosso il nodo 5 e si ordinano i rimanenti nodi, per esempio nella sequenza $\{1, 2, 3, 4\}$, il nodo 5 non può essere posto in nessuna posizione, perché dovrebbe stare dopo il 3, ma prima del 2. Il problema è che l'ordinamento topologico ha tante soluzioni (nell'esempio in Figura 6.4 anche $\{1, 3, 2, 4\}$ rappresenta una soluzione) e non tutte sono estendibili correttamente (solo la seconda soluzione lo è). D'altra parte, è lecito eliminare un nodo specifico, anziché un qualsiasi nodo. Ciò consente di ridurre il problema, garantendo che l'ordinamento del problema ridotto sia estendibile a un ordinamento corretto per il problema complessivo. Si opera come segue:

1. si elimina un nodo privo di dipendenze (privo cioè di archi entranti) e i relativi archi uscenti,
2. si riordina topologicamente il sottografo rimanente,
3. si reimmette il nodo eliminato per ottenere la soluzione del problema totale: affinché si mantenga l'ordine topologico, il nodo reimmesso deve essere posto all'inizio della sequenza ordinata.

Non è l'unica strada: è possibile eliminare un nodo che possiede solo dipendenze, cioè con nessun arco uscente, ordinare il sottografo rimanente, e reimmettere il nodo, obbligatoriamente, in ultima posizione.

```
Ordinamento_topologico(N, A)
if |N| = 1
    then return i_1
else
    i* = nodo_grado_uscente_nullo(N, A)
    if ∄i*
      then return errore                    {il grafo è ciclico}
    else
      N' := N\{i*}
      A' := A\Δ⁻_i*                         {Δ⁻_i* è l'insieme degli
                                             archi entranti in i*}
      L = Ordinamento_topologico(N', A')    {restituisce un possibile
                                             ordinamento per il grafo
                                             ridotto }
      return L + (i*)
```

È facile dimostrare che, quando la funzione *nodo_grado_uscente_nullo* non trova nodi privi di dipendenze, significa che il grafo è ciclico. Di conseguenza, l'algoritmo termina con un messaggio di errore.

Capitolo 7

Bilanciamento di alberi binari

Un *albero binario* è un insieme vuoto (*albero vuoto*) oppure una terna (r, T_{sx}, T_{dx}) il cui primo elemento è detto *radice*, mentre gli altri due sono alberi binari (rispettivamente, *sottoalbero sinistro* e *sottoalbero destro*). Si considerino alberi i cui elementi sono associati ad etichette intere.

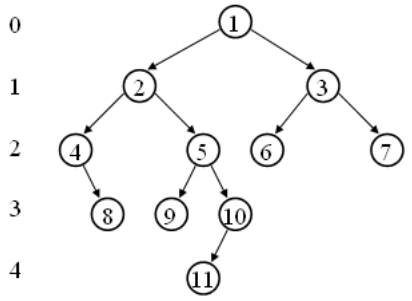
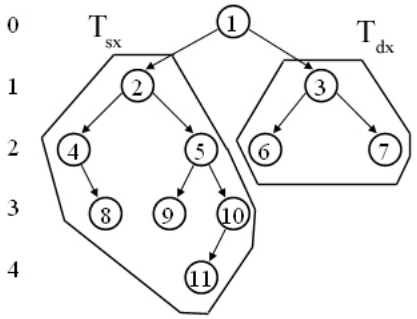


Figura 7.1: Albero binario di radice r

Dato un albero binario T di radice r come in Figura 7.1, si definiscono:

1. la *profondità* $p(n)$ di un elemento n, come il numero di archi da percorrere per raggiungere n a partire dalla radice. Ad esempio:
   (a) $P(6) = 2$
   (b) $P(11) = 4$
   (c) $P(1) = 0$
2. L'*altezza* $a(T)$, come la profondità massima degli elementi di T. Nella Figura 7.1, $a(T)$ è pari a 4. Nel caso di un albero vuoto, per convenzione $a(T) = -1$.

3. Il *bilanciamento* $b(T)$, come la differenza in valore assoluto tra l'altezza del sottoalbero sinistro e del sottoalbero destro di T. Nel caso di un albero vuoto, per convenzione $b(T) = 0$.



L'altezza del sottoalbero sinistro è 3, l'altezza del sottoalbero di destra è 1, quindi il bilanciamento $b(T)$ è: $|a(T_{sx}) - a(T_{dx})| = |3 - 1| = 2$.

Si vuole progettare per induzione un algoritmo che determini il bilanciamento dell'albero T. Nel caso di un albero vuoto, la sua altezza è definita per convenzione pari a -1 e il bilanciamento pari a zero. Nel caso in cui l'albero non sia vuoto, invece, si procede al partizionamento in classi numerate del problema, usando come indice l'altezza. Si osservi che:

$$\begin{cases} a(T_{dx}) \leq a(T) - 1 \\ a(T_{sx}) \leq a(T) - 1 \end{cases}$$

Quindi da un albero di altezza n, si ottengono due alberi di altezza minore o uguale ad $n - 1$. Questo suggerisce una semplice riduzione del problema a due sottoproblemi relativi ai sottoalberi. Tuttavia, non è banale ricostruire la soluzione del problema complessivo da quelle dei problemi ridotti, perché dal bilanciamento dei due sottoalberi non si può ottenere il bilanciamento dell'albero originario. Si deve rafforzare l'ipotesi induttiva, cioè risolvere un problema più complesso, in modo da sfruttare soluzioni più ricche di informazioni per ricostruire la soluzione del problema complessivo. Si affronta quindi il problema di ricavare sia il bilanciamento sia l'altezza. Nuovamente si suddivide il problema in classi numerate in base al numero dei nodi.

1. Il caso base (istanza di classe 0) presenta l'albero vuoto, quindi altezza pari a -1 e bilanciamento pari a zero.

2. Per passare dall'altezza n a $n - 1$ si definiscono i sottoalberi sinistro e destro, eliminando quindi il nodo radice.

3. La ricostruzione della soluzione avviene chiamando ricorsivamente la funzione *Bilanciamento*, che restituisce l'altezza e il bilanciamento dell'albero considerato ad ogni chiamata.

Si consideri, allora, il seguente codice:

```
Bilanciamento(T)
if vuoto(T)                          {se l'albero è vuoto
    then return (−1, 0)              a(T) = −1 e b(T) = 0}
else
    T' = sottoalberodestro(T)        {considera i
    T'' = sottoalberosinistro(T)     due sottoalberi}
    (b', a') = Bilanciamento(T')
    (b'', a'') = Bilanciamento(T'')
    a = max(a', a'') + 1
    b = |a' − a''|
    return (b, a)
```

Capitolo 8

Rimozione degli intervalli dominati

Definito un intervallo I, come una coppia di numeri (s, d) con $s \leq d$, si dice che un intervallo ne domina un altro quando lo contiene completamente: I_i domina I_j quando $s_i \leq s_j$ e $d_i \geq d_j$. Dato un insieme di intervalli $I_1, I_2, ..., I_n$ il problema consiste nell'identificare gli intervalli dominati, cioè completamente contenuti in altri.

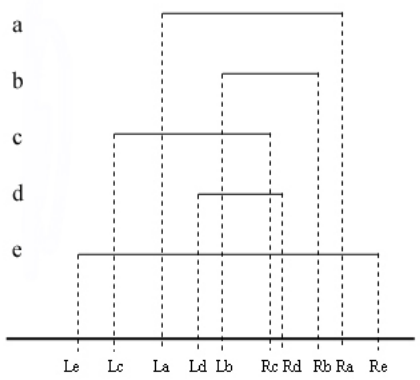


Figura 8.1: Esempio di intervalli

Come si osserva dalla Figura 8.1:
- l'intervallo a è contenuto solo in e,
- l'intervallo b è contenuto in a e in e,
- l'intervallo c è contenuto in e,
- l'intervallo d è contenuto in e e in a,

- l'intervallo e non è contenuto in nessun intervallo.
Quindi, tutti gli intervalli tranne e sono dominati.
Per partizionare il problema in classi numerate, si fa corrispondere ad ogni classe l'insieme delle istanze che contengono un dato numero di intervalli.

1. P_0 contiene le istanze con 1 intervallo: tale intervallo è ovviamente non dominato, perché non è incluso da altri.

2. Per ricondurre un'istanza I_n in P_n ad un'istanza in P_{n-1} si deve eliminare dall'insieme di partenza un intervallo e risolvere ricorsivamente il problema per gli $n-1$ intervalli rimanenti.

3. Per risolvere il problema completo, si aggiunge l'intervallo eliminato inizialmente, si estende la soluzione del problema ridotto verificando che l'intervallo aggiunto contenga gli intervalli precedenti o sia contenuto nei precedenti.

Per chiarezza si pensi alla soluzione del problema come ad un vettore in cui ogni posizione corrisponda ad un intervallo e contenga una marcatura che specifica se l'intervallo è dominato o no.

- Per ridurre il problema si elimina un intervallo qualsiasi, ad esempio l'ultimo, per semplicità.

- Per ricostruire la soluzione del problema completo si confronta l'intervallo reinserito rispetto agli $(n-1)$ altri intervalli per determinare se il nuovo intervallo è dominato da uno dei vecchi o ne domina uno. Questo comporta $(n-1)$ comparazioni ad ogni passo, cioè $(n-1)+(n-2)+...+1=\frac{n\cdot(n-1)}{2}$ comparazioni in complesso.

Algoritmo progettato per induzione Si pone la seguente ***ipotesi di induzione****: si conosce come risolvere il problema per gli intervalli* $I_1, I_2, ..., I_{n-1}$. Il caso base prevede di considerare un solo intervallo, esso non viene marcato, perché ovviamente non incluso da altri. Al passo induttivo l'ipotesi ci restituisce la soluzione per il problema P_{n-1}, definendo quali degli intervalli risultano marcati (e quindi contenuti). L'introduzione dell'intervallo I_n scelto a caso richiede di stabilire se il nuovo intervallo introdotto:
- contiene intervalli precedenti (non marcati)
- è contenuto dai precedenti.
Questo approccio è equivalente all'algoritmo esaustivo precedentemente riportato. Nonostante si possegga la soluzione del problema di dimensione $n-1$, l'introduzione dell' $n-$esimo intervallo può modificare la soluzione ottenuta per ipotesi.

Scelta della sequenza di induzione È possibile giungere ad un algoritmo più efficiente attraverso una scelta opportuna dell'intervallo da eliminare. Si scelga di rimuovere l'intervallo I con il più alto estremo di sinistra; se si ordinano gli intervalli in base a valori crescenti del primo estremo s, nel caso della Figura 8.1 la sequenza di intervalli ordinati è: E, C, A, D, B. Quindi si elimina B. Questa scelta risulta conveniente perché, nel momento in cui I verrà reintrodotto per la ricostruzione della soluzione S_n del problema P_n, garantisce che I_n non contiene alcuno dei precedenti intervalli, e non essendo dominante non modifica il vettore soluzione ottenuto dal problema ridotto. Rimane da stabilire se I è contenuto nei vecchi intervalli. Si deve verificare se esiste, fra gli $n-1$ intervalli, un intervallo con l'estremo sinistro più piccolo di quello di I (ed è ovvio dato che tutti lo hanno) e l'estremo destro più largo di quello di I. Nel caso considerato, sia l'intervallo A, sia E, hanno l'estremo destro maggiore di quello di B: B quindi è dominato. Nonostante l'eliminazione del controllo sul primo indice di ogni intervallo, la complessità dell'algoritmo rimane $\Theta(n^2)$.

Rafforzamento dell'ipotesi induttiva Si passa quindi ad un rafforzamento dell'ipotesi induttiva risolvendo un problema più complesso per ottenere informazioni dal problema ridotto. Si cercano non solo gli intervalli dominanti, ma anche il massimo valore dell'estremo destro d_i. Questo semplifica il secondo controllo. Infatti quando l'intervallo I_n viene reimmesso nel problema, esso non può dominare i precedenti perché ha indice sinistro s maggiore, ma rimane da confrontare il valore dell'indice destro di tale intervallo (d_n) con il massimo valore di d restituito dagli $n-1$ intervalli precedenti (d_{max}). Se si verifica $d_n > d_{max}$ l'intervallo non è ridondante (non domina, né è dominato), se $d_n \leq d_{max}$ è dominato.

Interpretazioni dell'algoritmo con le invarianti di ciclo Come ultima analisi, si osservi che l'algoritmo così progettato può essere facilmente implementato in modo iterativo, con la stessa tecnica usata nel Capitolo 3 per gli algoritmo di ordinamento. L'algoritmo iterativo risultante gode, come tutti gli algoritmi iterativi, di interessanti proprietà che sono invarianti di ciclo. Si vedranno quali sono in questo caso, per studiare la similitudine forte che esiste fra l'approccio induttivo e l'approccio iterativo. Quest'ultimo si basa su un'invariante di ciclo che non subisca modifiche nelle iterazioni necessarie a giungere alla soluzione del problema. Analogamente si opera con l'induzione matematica: si ottiene la soluzione del problema P_n a partire dalla soluzione del problema ridotto P_{n-1}, conservando la proprietà che il risultato è la soluzione del problema corrente. Di seguito la definizione dei

punti trattati per via induttiva ponendo come invariante di ciclo la soluzione per $n-1$ intervalli.

- Scelta casuale dell'intervallo.
  - Passo iniziale: *i primi $f = 1$ intervalli sono marcati correttamente gli uni rispetto agli altri*;
  - passo iterativo: *conserva l'invariante di ciclo, confrontando un frammento non ancora esaminato con quelli già esaminati e viceversa.*

  Il passo iniziale corrisponde al caso base: l'invariante di ciclo si può tradurre dicendo che l'algoritmo risolve correttamente problemi di un solo intervallo. Il passo iterativo corrisponde all'induzione: se il risultato dell'iterazione precedente è la soluzione di I_{n-1}, il risultato dell'iterazione corrente deve essere la soluzione di I_n. Si veda l'invariante e il modo in cui viene conservata nelle due versioni più sofisticate dell'algoritmo.

```
IntRidondanti(NumInt, I, Ridondante)
  Ridondante[1] = falso
  for f = 2 to NumInt do
    Ridondante[f] = falso
    for g = 1 to f do
     if ((s_f ≤ s_g) and
         (d_f ≥ d_g) then
     Ridondante[g] = vero
     else if ((s_f ≥ s_g) and
              (d_f ≤ d_g)) then
     Ridondante[f] = vero
```

- Scelta della sequenza di induzione.
  - Passo iniziale: *gli $f = 1$ intervalli con s_i minimo sono marcati correttamente*;
  - passo iterativo: *conserva l'invariante di ciclo, cercando il frammento non ancora esaminato con s_i massimo e marcandolo correttamente.*

```
IntRidondanti2(NumInt, I, Ridondante)
  gMin = 1
  for g = 2 to NumInt do
    if ((s_g < s_gMin) or
        (s_g = s_gMin) and
        (d_g > d_gMin)) then          {instaura l'invariante di
                                       ciclo}
      gMin = g
      Temp = I[1]
      I[1] = I[gMin]
      I[gMin] = Temp
      Ridondante[1] = falso
      for g = 1 to f do
        if (d_g ≥ d_f) then
          Ridondante[f] = vero
```

- Rafforzamento dell'ipotesi induttiva.
  - Passo iniziale: *gli $f = 1$ intervalli con s_i minimo sono marcati correttamente e d_{max} è il valore massimo del loro indice destro*;
  - passo iterativo: *conserva l'invariante di ciclo, cercando il frammento non ancora esaminato e marcandolo correttamente.*

  Si riporta di seguito un frammento di codice.

```
IntRidondanti3(NumInt, I, Ridondante)
  gMin = 1
  for g = 2 to NumInt do
    if ((s_g < s_gMin) or
        (s_g = s_gMin) and
        (d_g > d_gMin)) then          {instaura l'invariante di
    gMin = g                          ciclo: gli intervalli
    Temp = I[1]                       con s minimo
    I[1] = I[gMin]                    sono marcati correttamente}
    I[gMin] = Temp
    Ridondante[1] = falso
    d_max = d_1                       {d_max è il valore
                                      massimo di d_i}
  for f = 2 to NumInt do
    gMin = f
    for g = f + 1 to NumInt do
      if ((s_g < s_gMin) or
          (s_g = s_gMin) and
          (d_g > d_gMin)) then        {conserva l'invariante di ciclo,
        gMin = g                      cerca il frammento non ancora
    Temp = I[f]                       esaminato con IntIn minimo
    I[f] = I[gMin]                    e marcandolo correttamente}
    I[gMin] = Temp

    Ridondante[f] = falso
    if (MaxFin ≥ d_f) then
      Ridondante[f] = vero
    else
      d_max = d_f
```

Capitolo 9

Coppia di punti a minima distanza

Dati n punti in un piano si vuole determinare, fra questi, la coppia di punti che presenta la minima distanza reciproca.

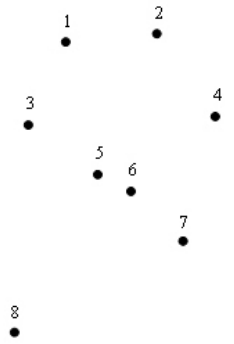


Figura 9.1: Nell'insieme di punti nel piano, la coppia di punti a minima distanza è (5,6)

L'insieme delle istanze che costituiscono il problema viene diviso in classi numerate in relazione al numero di punti nel piano.

Algoritmo esaustivo Un approccio banale al problema consiste nell'implementare un algoritmo esaustivo. Si determina la distanza che ciascun punto possiede rispetto agli $n-1$ punti rimanenti e si definisce fra le distanze calcolate quella minima. Questo tipo di approccio però porta a $n-1+n-2+....+1 = \frac{n\cdot(n-1)}{2}$ confronti. Si cerca un algoritmo più efficiente.

Algoritmo progettato per induzione Si affronta il problema della minima distanza secondo l'approccio induttivo, riconducendo un problema di classe n a un problema di classe $n-1$ con l'eliminazione di un punto. Si osservano due casi:

- caso base: se l'insieme dei punti è pari a 0 o 1, si restituisce una distanza infinita, mentre se i punti sono 2, banalmente, la minima distanza è la distanza fra i punti stessi.
- Passo induttivo: per ipotesi si suppone di avere la soluzione per $n-1$ punti, ossia si possiede il valore δ_{n-1} che è la minima distanza fra i primi $n-1$ punti. Per ottenere la soluzione del problema completo I_n si reimmette l'ultimo punto: questo comporta comunque di confrontare quest'ultimo con tutti i precedenti e valutare se tale punto possiede una distanza $\delta_{in} < \delta_{n-1}$ da qualche punto i fra i primi $n-1$.

Tale approccio corrisponde però all'approccio esaustivo precedentemente illustrato.

Algoritmo progettato per induzione forte La riduzione operata eliminando un elemento ad ogni passo (come avveniva per gli esempi precedenti) non offre una via risolutiva conveniente per il problema della minima distanza. E' possibile intraprendere differenti strade di riduzione. Ad esempio si può giungere alle classi banali del problema riducendo il problema stesso in due sottoproblemi di dimensione $\frac{n}{2}$. In questo caso si può dividere il problema in due sottoproblemi a caso. Per induzione si otterrebbe: δ_1 che è la minima distanza del primo sottoproblema e δ_2 che è la minima distanza del secondo sottoproblema. Tale suddivisione però comporta di dover valutare tutte le distanze fra le coppie di punti a cavallo della divisione e di confrontarle con la minima fra δ_1 e δ_2, incorrendo in $\frac{n}{2}\cdot\frac{n}{2} = \frac{n^2}{4}$ confronti. Questo approccio corrisponde, anch'esso, all'algoritmo esaustivo.

Costruzione sofisticata dei due sottoproblemi Risulta conveniente ordinare i punti secondo il valore di una coordinata (ad esempio secondo le ascisse) e dividerli in modo tale che ogni sottoproblema sia di dimensione $\frac{n}{2}$. Questo comporta che ogni sottoproblema corrisponda ad un semipiano, come in Figura 9.2, e i punti che ricadono sulla retta di sezione, dovranno essere assegnati arbitrariamente ad uno o all'altro semipiano. Si ipotizzi, per comodità, che il numero di punti n sia una potenza di 2, questo affinché sia sempre possibile ottenere una divisione semplice e utile per risolvere efficientemente il problema.

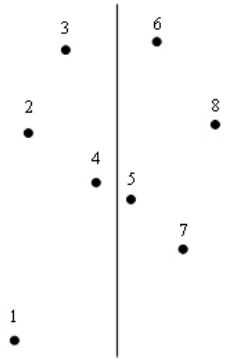


Figura 9.2: Realizzazione dei due sottoinsiemi

Il caso base riguarda sempre insiemi di 0, 1 o 2 punti. Applicando questa riduzione, si ottengono per via induttiva i valori δ_1 e δ_2, che sono le minime distanze dei due sottoproblemi. Si definisca δ_{min} la distanza minima fra δ_1 e δ_2. Resta ora da analizzare la distanza fra i punti a cavallo della linea di separazione dei due sottoproblemi.

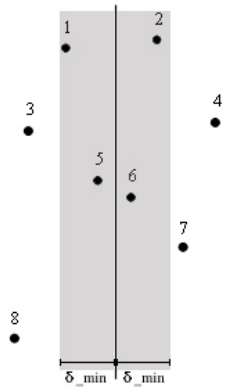


Figura 9.3: Definizione dell'area a cavallo della linea di separazione

Come si osserva dalla Figura 9.3, la definizione della linea di separazione permette di escludere dalla valutazione i punti che distano più di δ_{min} dalla linea, perché sicuramente distano più di δ_{min} dai punti dell'altro sottoproblema. Purtroppo questa selezione potrebbe non essere efficace nel momento in cui tutti i punti (o gran parte di essi) sono ammassati intorno alla linea di separazione. Si vede necessario operare un'ulteriore selezione. Si procede quindi ad ordinare i punti candidati per ordinata y crescente. Per ogni punto p_i si calcola quindi la distanza non con tutti gli altri punti nell'area, bensì solo con i punti p_j la cui ordinata y_j differisce da y_i per un valore $|y_j - y_i| < \delta_{min}$.

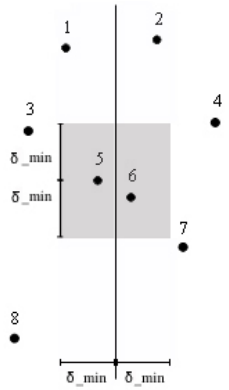


Figura 9.4: Selezione dell'area di valutazione per il punto 5

Dato che nel caso limite, le coppie di punti che interessano distano fra loro meno di δ_{min}, il punto p_i può trovarsi sulla linea di divisione e i

punti con cui effettuare il confronto sono a distanza δ_{min}. Si può dimostrare che i candidati alla valutazione della distanza sono al massimo 5, disposti come in Figura 9.5. I candidati si possono ridurre da 5 punti con ordinata vicina a y_i a al massimo 3 punti con ordinata precedente a y_i e al massimo 3 punti con ordinata successiva a y_i e solo questi ultimi vanno considerati, poiché la distanza è simmetrica rispetto al punto stesso.

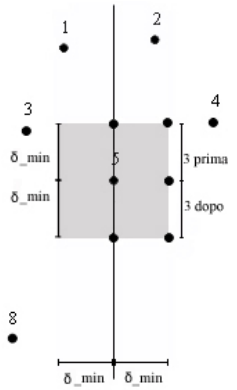


Figura 9.5: Caso limite di studio

In tal modo la complessità dell'algoritmo è $T(n) = 2T(\frac{n}{2}) + \Theta(n \log n)$ e quindi $T(n) = \Theta(n\log^2 n)$.
L'operazione di ordinamento rispetto i valori di y non interferisce con la soluzione dei due sottoproblemi P_1 e P_2, poiché le soluzioni dei problemi ridotti sono ottenute precedentemente all'ordinamento.

Rafforzamento dell'ipotesi induttiva È possibile rafforzare l'ipotesi precedente in modo tale che dalla divisione del problema non solo si ottenga la distanza minima di ogni sottoproblema, ma anche l'ordinamento dei punti di ciascun sottoproblema per valori di y crescenti. Il vantaggio è che, dopo aver ottenuto la soluzione dei due sottoproblemi non si ha più un'operazione di *sort* sui punti, ma direttamente una *merge* per fondere le due aree a cavallo della linea di divisione. Manca, quindi, il solo confronto fra i punti candidati. Questo approccio abbassa la complessità a: $T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n)$. Questo rafforzamento richiede però di complicare la soluzione delle classi banali del problema: per 0 o 1 punto non si fa nulla, per due punti, dovendo

eventualmente fornire i punti ordinati per y crescente, si richiede uno scambio.

Di seguito un estratto dal codice.

FindClosestPair (pi, pf, *Aux, *Point)
if $pf - pi < 1$ **then**
$P1Min->id = P1Min->x = P1Min->y = 0$
$P2Min->id = P2Min->x = P2Min->y = 0$
$\delta_{min} = +\infty$
return $(P1Min, P2Min, \delta_{min})$
else if $pf - pi = 1$ **then**
$\delta_{min} = Distanza(Point[pi], Point[pf])$
$P1Min = Point[pi]$
$P2Min = Point[pf]$
if $Point[pi].y > Point[pf].y$ **then**
$Temp = Point[pi]$ {ordina i due
$Point[pi] = Point[pf]$ punti per
$Point[pf] = Temp$ y crescente}
return $(P1Min, P2Min, \delta_{min})$
else

$m = \frac{(pi+pf)}{2}$ {determina i due
$x = Point[m].x$ sottovettori e la
cordinata che divide
in due il piano}

$(P1Min1, P2Min1, \delta_1) =$
$FindClosestPair(pi, m, Aux, Point)$ {vengono risolti i
$(P1Min2, P2Min2, \delta_2) =$ due sottoproblemi}
$FindClosestPair(m + 1, pf, Aux, Point)$
$MergePointsByY(pi, m, pf, Aux, Point)$ {fonde le due liste
in una ordinata, secondo
valori di y crescenti}

{Determina la coppia di
punti vicina al taglio}

$P1Min = P1Min1$
$P2Min = P2Min1$
$\delta_{min} = \delta_1$
if $\delta_2 < \delta_{min}$ **then**
$P1Min = P1Min2$
$P2Min = P2Min2$
$\delta_{min} = \delta_2$

{Trova la coppia di punti più vicini a cavallo del taglio
e salta i punti la cui ascissa differisce da x più di δ_{min}}

$pp = pi - 1$
for $p = pi$ **to** pf **do**
if $(|Point[p].x - x| < \delta_{min})$ **then**
$Aux[++pp] = Point[p]$

{per ognuno, valuta la distanza dai 5 punti piu' vicini
nella sequenza e dato che le distanze sono simmetriche,
si valutano solo rispetto ai 3 successivi}

for $p = pi$ **to** pp **do**
if $((p + 1 \leq pp)$ **and** $(Distanza(Aux[p], Aux[p + 1]) < \delta_{min}))$ **then**
$\delta_{min} = Distanza(Aux[p], Aux[p + 1])$
$P1Min = Aux[p]$
$P2Min = Aux[pp]$
if $((p + 2 \leq pp)$ **and** $(Distanza(Aux[p], Aux[p + 2]) < \delta_{min}))$ **then**
$\delta_{min} = Distanza(Aux[p], Aux[p + 2])$
$P1Min = Aux[p]$
$P2Min = Aux[pp]$
if $((p + 3 \leq pp)$ **and** $(Distanza(Aux[p], Aux[p + 3]) < \delta_{min}))$ **then**
$\delta_{min} = Distanza(Aux[p], Aux[p + 3])$
$P1Min = Aux[p]$
$P2Min = Aux[pp]$
return $(P1Min, P2Min, \delta_{min})$