Capitolo 16

Backtrack

Problemi tipici Problemi tipici sono:

- *elencare tutte le possibili soluzioni* (enumerazione), come ad esempio elencare tutte le possibili permutazioni di un insieme;
- *costruire almeno una soluzione del problema*, in questo caso si utilizza l'algoritmo di enumerazione fermandosi alla prima soluzione dispoonibile;
- *contare le soluzioni*, in alcuni casi è possibile contare in modo analitico, in altri casi si costruiscoo le soluzioni e si contano;
- *trovare le soluzioni ottimali*, si enumerano tutte le soluzioni che vengono valutate tramite una funzione di costo. In questo caso si utilizzano anche altre tecniche di programmazione (come la programmazione dinamica, o la tecnica greedy).

16.1 Eumerazione

Per costruire tutte le soluzioni, si utilizza un approccio di "forza bruta" (*brute force*). In alcuni casi è l'unica strada percorribile. Fortunatamente i processori moderni rendono affrontabili problemi considerati di dimensioni medio-piccole. Inoltre, a volte lo spazio delle soluzioni non deve essere analizzato interamente.

Idea (backtracking). "Prova a fare qualcosa, e se non va bene, disfalo e prova qualcos'altro"

Il backtracking è una tecnica algoritmica che, come altre, deve essere personalizzata per ogni applicazione individuale. Dobbiamo quindi trovare un metodo sistematico per iterare su tutte le possibili istanze di uno spazio di ricerca.

Organizzazione del problema Una soluzione viene rappresentata come il vettore $S[1][n]$; il contenuto degli elementi $S[i]$ è una *sequenza di scelte* (possibili) C dipendenti dal problema.

Ad esempio preso un insieme C che rappresenta un insieme generico possiamo avere possibili *permutazioni* o *sottoinsiemi*. Se C è un insieme di mosse ottenerremo una *sequenza di mosse*; se nell'insieme C sono contenuti archi di un grafo, allora otterremo possibili percorsi del grafo; e così via.

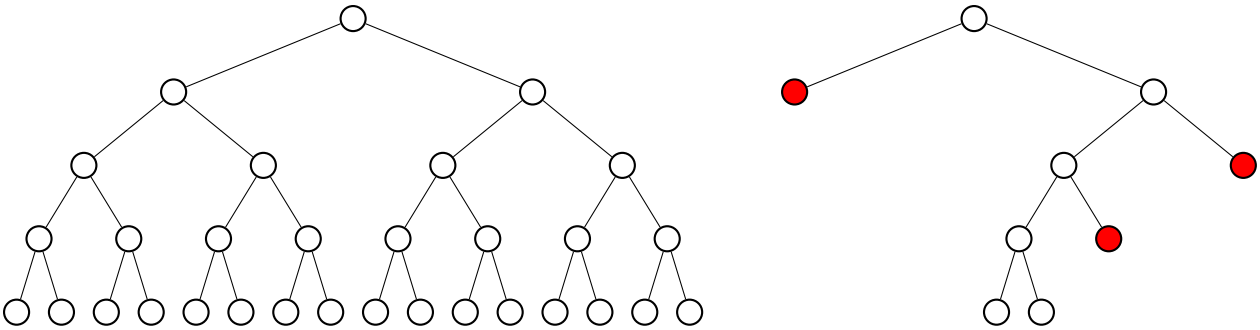
Scherma di risoluzione generale Ad ogni passo, partiamo da una soluzione generale $S[1][k]$ in cui $k \geqslant 0$ scelte sono state prese (ovvero abbiamo effettuato k scelte, dove k può essere anche nullo).

Se la sequenza di scelte che abbiamo effettuato ($S[1][k]$) costituiscono una soluzione ammissibile allora la "processiamo". Può essere quindi stampata, contata, valutata, oppure si può decidere di terminare elencando tutte le possibili soluzioni.

Indipendentemente dalla precedente se $S[1][k]$ non rappresenta una soluzione completa, allora proviamo, se è possibile, ad *estendere* $S[1][k]$ con una delle possibili scelte in una soluzione $S[1][k+1]$; altrimenti "cancelliamo" l'ultima scelta effettuata $S[k]$ tornando sui nostri passi (facendo quindi *backtracking*) e ripartendo dalla soluzione precedente $S[1][k-1]$.

Rappresentazione del problema Il nostro problema viene rappresentato da un *albero di decisione* nel quale:

- lo *spazio di ricerca* viene rappresentato dall'albero stesso;
- la *soluzione parziale vuota* (ossia quella dove non abbiamo preso nessuna decisione) è la radice;



(a) Lo spazio di ricerca di un determinato problema è rappresentato da un albero delle decisioni

(b) *pruning* dei rami

Figura 16.1: Un albero di decisione ed il suo corrispondente albero potato dei rami che non portano a soluzioni ammissibili.

- le *soluzioni parziali* sono rappresentate dai nodi interni;
- le *soluzioni ammissibili* vengono rappresentate dalle foglie.

Pruning I "rami" dell'albero che sicuramente non portano a soluzioni ammissibili possono essere "potati" (*pruned*). La valutazione della "potatura" viene effettuata nelle soluzioni parziali radici del sottoalbero da potare.

Nota. Il pruning riduce drasticamente lo spazio delle soluzioni del problema.

Approcci Ci sono due possibili approcci alla tecnica del backtracking:

1. la prima consiste nello sviluppo di un algoritmo *ricorsivo* che lavora tramite una visita in profondità nell'albero delle scelte;
2. la seconda consiste nello sviluppo di un algoritmo *iterativo* ed utilizza un approccio ingordo (*greedy*), eventualmente tornando sui propri passi (effetuando quindi *backtracking*).

Sia il problema dell'inviluppo convesso, che quello della corrispondenza fra stringhe (*string matching*) utilizzano questo approccio.

In base alle i scelte già effettuate decido quali sono le mie future scelte.

Algoritmo 1: Schema generale per il problema dell'enumerazione

```
boolean enumerazione(ITEM[] S, int n, int i, ...)
    // S:   vettore contenente le soluzioni parziali S[1][i]
    // n:   il numero massimo di scelte possibili
    // i:   indice corrente
    // ...: parametri addizionali dipendenti dal problema

    SET C = scelte(S, n, i, ...) // Determina C in funzione di S[1][i-1]
    // C:   l'insieme dei possibili candidati per estendere la soluzione
    foreach c ∈ C do // per ogni possibile scelta fra quelle possibili
        S[i] ← c  // registro la scelta

        if isAdmissible(S, n, i) then
            // S[1][i] è una soluzione ammissibile
            if processSolution(S, n, i, ...) then
                // vogliamo bloccare l'esecuzione alla prima soluzione possibile
                return true

        // non decido di fermarmi alla prima soluzione ammissibile
        if enumerazione(S, n, i+1, ...) then
            // effettuo la i+1-esima scelta
            return true
    // non ho trovato la soluzione cercata
    return false
```

16.1.1 Problemi che trattano di sottoinsiemi

Definizione del problema Elencare tutti i sottoinsiemi dell'insieme $\{1, \ldots, n\}$

Algoritmo 2: Primo tentativo

```
subSets(int[] S, int n, int i)
    SET C ← iif(i ⩽ n, {0,1}, ∅)
    foreach c ∈ C do
        S[i] ← c
        if i == n then
            processSolution(S, n)
        subSets(S, n, i + 1)
```

Algoritmo 3: Versione "più pulita"

```
subSets(int[] S, int n, int i)
    foreach c ∈ {0,1} do
        S[i] ← c
        if i == n then
            processSolution(S, n)
        else
            subSets(S, n, i + 1)
```

Fare riferimento alla spiegazione grafica qui non riportata.

Considerazioni Non c'è pruning. Tutto lo spazio possibile viene esplorato. Ma questo avviene per definizione. Questo porta ad una complessità di $\mathcal{O}(n \cdot 2^n)$. È possbile pensare ad una soluzione iterativa, ad-hoc? (che non utilizza la tecnica del backtracking)

Algoritmo 4: Versione "più pulita"

```
subSets(int[] S, int n, int i)
    from j = 0 until 2^n − 1 do // O(n^2)
        stampa "{"
        from i = 0 until n − 1 do // O(n)
            if (j and 2^i) ≠ 0 then
                stampa i
            stampa "}"
```

Definizione del problema Elencare tutti i sottoinsiemi di dimensione k di un insieme $\{1, \ldots, n\}$

Algoritmo 5: Tentativo 1

```
subSets(int[] S, int n, int k, int i)
    SET C ← iif(i ⩽ n, {0,1}, ∅)
    foreach c ∈ C do
        S[i] ← c
        if i == n then
            int count ← 0
            from j ← 1 until n do
                count += S[j]
            if count == k then
                // ho finito
                processSolution(S, n)
        subSets(S, n, k, i+1)
```

Algoritmo 6: Tentativo 2

```
subSets([...], int count)
    SET C ← iif(i ⩽ n, {0,1}, ∅)
    foreach c ∈ C do
        S[i] ← c
        count += S[i]
        // logica dell'algoritmo
        if i == n and count == k then
            processSolution(S, n)
        subSets(S, n, k, i+1, count)
        count -= S[i]
```

~~Tengo conto del contatore nelle chiamate successive~~ e gli sottraggo i valori aggiunti nelle chiamate di *backtracking*.

Algoritmo 7: Elencare tutti i sottoinsiemi di dimensione k di un insieme $\{1, \ldots, n\}$

```
subSets(int[] S, int n, int k, int i, int count)
    // n:  numeri di elementi
    // k:  numeri di elementi considerati
    // i:  la scelta che ho già fatto
    // se ho già raggiunto k o non ci sono abbastanza bit per arrivare a k non ho più bisogno di
       visitare il sotto-albero delle scelte relativo)
    SET C ← iif(count < k and count + (n − i + 1) ⩾ k, {0,1}, ∅)
    // count < k:  ho ancora la possibilità di accendere un bit
    // count + (n − i + 1) ⩾ k:  ho ancora abbastanza bit da accendere per raggiungere k
    foreach c ∈ C do
        S[i] ← c
        // i bit scelti non cambiano nelle chiamate successive
        count += S[i]  // sommo i bit a 1
        if count == k then
            processSolution(S, i)
        else
            subSets(S, n, k, i+1, count)
        // quando effettuo il backtrack devo tornare allo stato precedente
        count -= S[i]  // sottraggo i bit a 1
```

Considerazioni Abbiamo imparato che "specializzando" l'algoritmo generico, possiamo ottenere una versione più efficiente. Tuttavia abbiamo ottenuto solo un miglioramento parziale (verso $n/2$).

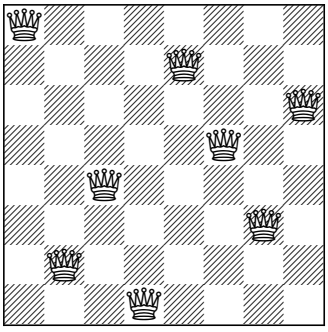
Nota. È difficile ottenere la stessa efficienza con un algoritmo iterativo.

16.2 Permutazioni

Definizione del problema Stampa tutte le permutazioni di un insieme A. L'insieme dei candidati dipende dalla soluzione parziale *corrente*.

```
permutazioni(SET A, int n, ITEM // S, int i)
    // A: insieme dalla quale prendo gli oggetti
    // n: numero di permutazioni
    foreach c ∈ A do
        S[i] ← c  // scelgo l'oggetto
        A.remove(c)  // lo tolgo dall'insieme
        if A.isEmpty then
            processSolution(S, n)
        else
            // l'insieme A ha un elemento in meno
            permutazioni(A, n, S, i + 1)
        A.insert(c)
```

16.3 Problema delle otto regine

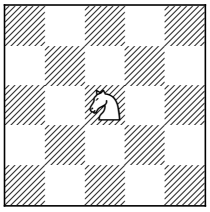


Definizione del problema Il problema consiste nello posizionare n regine in una scacchiera $n \times n$, in modo tale che nessuna regina ne "minacci" un'altra.
Idea. Ogni riga ed ogni colonna deve contenere esattamente una regina. Rappresentiamo la scacchiera con un vettore $S[1][n]$, effettuiamo *pruning* eliminando le diagonali.
Il numero di soluzioni ammonta a $n! = 8! = 40\,320$ della quale solo $15\,720$ vengono esplorate.

Esiste un algoritmo *probabilistico* che risolve il problema in tempo lineare, tuttavia non garantisce che la terminazione sia sempre corretta. Quindi viene fatto eseguire più volte fintanto che restituisce una soluzione corretta. Il meccanismo consiste nel partire da una soluzione "ragionevolmente buona" e di muovere il pezzo con il più grande numero di conflitti nella posizione, all'interno della stessa colonna, in cui ne genera il numero minore. La soluzione iniziale è scelta in modo "casuale" (ne parleremo più approfonditamente nel prossimo capitolo). L'algoritmo si ferma quando non ci sono più pezzi da muovere.

16.4 Giro di cavallo

Definizione del problema Lo scopo è trovare un "giro di cavallo", ovvero un percorso di mosse valide del cavallo in modo che ogni casella venga visitata al più una volta.



16.4.1 Algoritmo risolutivo

Problema del giro di cavallo

```
cavallo(int[][] S, int i, int x, int y)
    SET C ← mosse(S,x,y)
    foreach c ∈ C do
        S[x,y] ← i  // ho raggiunto la posizione (x,y) all'i-esimo passo
        if i == 64 then
            processSolution(S)
            return true
        else if cavallo(S, i+1, x+m_x[c], y+m_y[c]) then
            // effettuo l'i-esima mossa
            return true
        S[x,y] ← 0
    return false
// trova le possibili mosse
mosse(int[][] S, int x, int y)
    SET C ← Set
    // fra tutte le possibili mosse
    from i ← 1 until 8 do
        // calcola la nuova posizione
        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
            // la posizione è libera
            C.insert(i)
    return C
```

Le prime due condizioni ($1 \leqslant n_x \leqslant 8$ **and** $1 \leqslant n_y \leqslant 8$ controllano se rientro nei limiti della scacchiera, mentre la terza ($S[n_x, n_y] \leftarrow 0$) controlla se la posizione non è stata toccata.

16.5 Sudoku

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

4	2	*6*	5	*7*	1	*3*	9	*8*
8	*5*	*7*	2	*9*	3	*1*	*4*	6
1	3	*9*	*4*	6	*8*	*2*	7	*5*
9	*7*	1	*3*	*8*	*5*	6	*2*	*4*
5	4	*3*	*7*	*2*	*6*	*8*	1	9
6	*8*	2	*1*	*4*	*9*	7	*5*	*3*
7	9	*4*	*6*	3	*2*	*5*	8	*1*
2	*6*	*5*	8	*1*	4	*9*	*3*	7
3	1	*8*	9	*5*	7	*4*	6	*2*

Figura 16.2: Il gioco del sudoku

16.5.1 Algoritmo risolutivo

Sudoku

```
boolean sudoku(int[][] S, int i)
    int x ← i mod 9
    int y ← ⌊i/9⌋
    SET C ← Set

    // il numero è già stato scelto
    if S[x,y] ≠ 0 then
        S.insert(S[x,y])
    else
        // inseriamo un numero
        from c ← 1 until 9 do
            if verifica(S, x, y, c) then
                C.insert(c)

    int old ← S[x,y]
    foreach c ∈ C do
        S[x,y] ← c

        // arriviamo all'ultima casella
        if i == 80 then
            processSolution(S,n)
            return true

        if sudoku(S, i + 1) then
            return true

    S[x,y] ← old
    return false
```

```
// se posso inserire un numero in quella cella
verifica(int[][] S, int x, int y, int c)
    from j ← 0 until 8 do
        // controllo sulla colonna
        if S[x,j] == c then
            return false

        // controllo sulla riga
        if S[j,y] == c then
            return false

    int b_x ← ⌊x/3⌋
    int b_y ← ⌊y/3⌋
    from i_x ← 0 until 2 do
        from i_y ← 0 until 2 do
            // controllo sottotabella
            if S[b_x · 3 + i_x, b_y · 3 + i_y] = c then
                return false
```

16.6 Inviluppo convesso

Definizione 16.6.1 (poligono convesso)**.** Un poligono nel piano è convesso se ogni segmento di retta che congiunge due punti del poligono sta interamente nel poligono stesso, incluso il bordo.

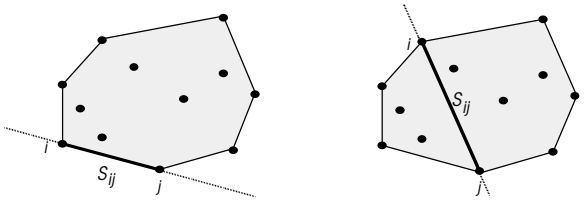
Definizione del problema Dati n punti $p_1, \ldots, p_n$ nel piano, con $n \geqslant 3$, l'inviluppo convesso (*convex null*) è il più piccolo poligono convesso che li contiene tutti.

16.6.1 Algoritmo inefficiente

Un poligono può essere rappresentato per mezzo dei suoi spigoli. Si consideri la retta che passa per una coppia di punti (p_i, p_j), che divide il piano in due semipiani chiusi. Se tutti i rimanenti $n-2$ punti stanno dalla *stessa parte*, allora lo spigolo S_{ij} fa parte dell'inviluppo convesso.

inserire didascalia

Data una retta definita dai punti p_1 e p_2, determinare se due punti p e q stanno nello stesso semipiano definito dalla retta.



boolean stessaparte(POINT p_1, POINT p_2, POINT p, POINT q)

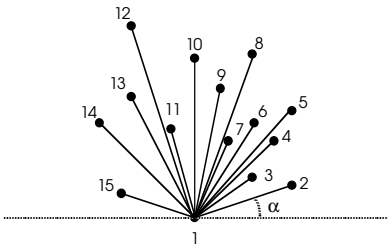
```
float dx ← p2.x − p1.x
float dy ← p2.y − p1.y
float dx1 ← p.x − p1.x
float dy1 ← p.y − p1.y
float dx2 ← q.x − p2.x
float dy2 ← q.y − p2.y
// se ⩾ 0 allora sono dalla stessa parte
return ((dx · dy1 − dy · dx1) · (dx · dy2 − dy · dx2) ⩾ 0)
```

Complessità Prendiamo tutte le coppie di punti (n^2) e controlliamo se tutti gli altri punti $(n-2)$ stanno "dall'altra parte". La complessità ammonta a $\mathcal{O}(n^2 \cdot (n-2) = \mathcal{O}(n^3)$

16.6.2 Algoritmo di Graham

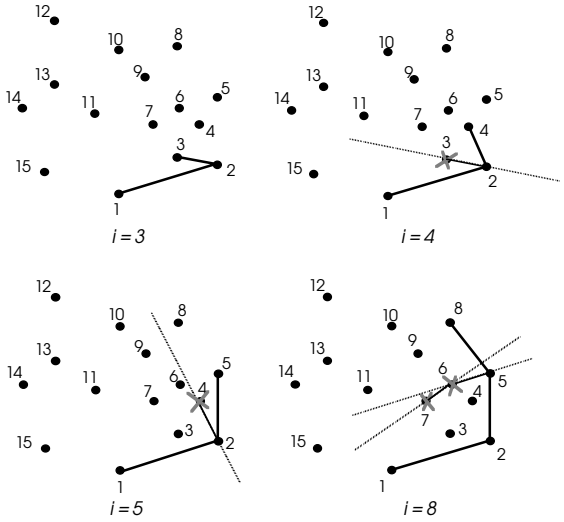
Prima fase

- il punto con ordinata minima fa parte dell'inviluppo convesso;
- si ordinano i punti in base all'angolo formato dalla retta passante per il punto con ordinata minima e la retta orizzontale.



Seconda fase

- inserisci p_1, p_2 nell'inviluppo corrente;
- per tutti i punti $p_i = 3, \ldots, n$:
  1. siano p_h e p_j, con $h < j = i - 1$, gli ultimi due vertici dell'inviluppo corrente;
  2. scandisci a "ritroso" i punti nell'inviluppo "corrente" ed elimina p_j se stessaparte(p_j,p_h,p_1,p_i) == **false**;
  3. termina tale scansione se p_j non deve essere eliminato;
  4. aggiungi p_i all'inviluppo "corrente".



Algoritmo di Graham

```
STACK graham(POINT p, int n)
    int min = 1
    // trovo il minimo
    from i ← 2 until n do
        if p[i].y < p[min].y then
            min ← i
    p[1] ↔ p[min]
    { riordina p[2,...,n] in base all'angolo formato rispetto all'asse orizzontale quando sono
      connessi con p[1]}
    { elimina eventuali punti "allineati" tranne i più lontani da p1, aggiornando n }
    STACK S ← Stack
    // inserisci p1, p2 nell'inviluppo corrente
    S.push(p1)
    S.push(p2)
    // per tutti gli altri punti
    from i ← 3 until n do
        // escludo tutti i punti all'interno dell'inviluppo
        while not stessaparte(S.top, S.top2, p1, pi) do // top2 restituisce il secondo elemento
            S.pop
        S.push(pi)
```

Complessità L'algoritmo di Graham ha complessità $\mathcal{O}(n \log n)$ in quanto è dominato dall'ordinamento dei punti.