1. Backtrack

Una tecnica non molto astuta, ma che talvolta può risultare utile, si basa sulla ovvia considerazione seguente:

"Prova a fare qualcosa e, se non funziona, disfala e riprova qualcos'altro".

Questa tecnica, detta *backtrack*, è alla base di molti algoritmi di visita di alberi e grafi, ed è usata per generare sistematicamente tutte le soluzioni ammissibili di un problema (*to backtrack* significa "tornare sui propri passi").

Esempio (Visita di alberi e grafi) Le procedure pre-visit(), post-visit() e in-visit() negli alberi [cfr. Capitolo 5] sono basate sul *backtrack*, dove l'elaborazione su un nodo è sospesa temporaneamente per permettere la visita di alcuni suoi sottoalberi. Anche la dfs() nei grafi [cfr. Capitolo 9], che è una diretta estensione della pre-visit(), è basata sul *backtrack*. ■

Per illustrare questa tecnica, vedremo innanzitutto un problema particolare, l'INVILUPPO CONVESSO; vedremo poi uno schema generale che può essere applicato ad una vasta classe di problemi.

1.1 Inviluppo convesso

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

INVILUPPO CONVESSO (CONVEX HULL). *Dati n punti $p_1, \ldots, p_n$ nel piano, con $n \geq 3$, trovare il più piccolo poligono convesso che li contiene tutti.*

Intuitivamente, i punti possono essere pensati come chiodi conficcati parzialmente in un'asse di legno ed il bordo del loro inviluppo convesso come la sagoma formata da un elastico che li stringe tutti.

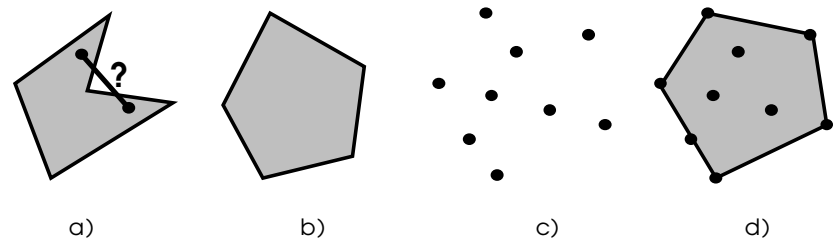

Figura 1.1: a) Un poligono non convesso; b) un poligono convesso; c) un insieme di punti; d) il loro inviluppo convesso.

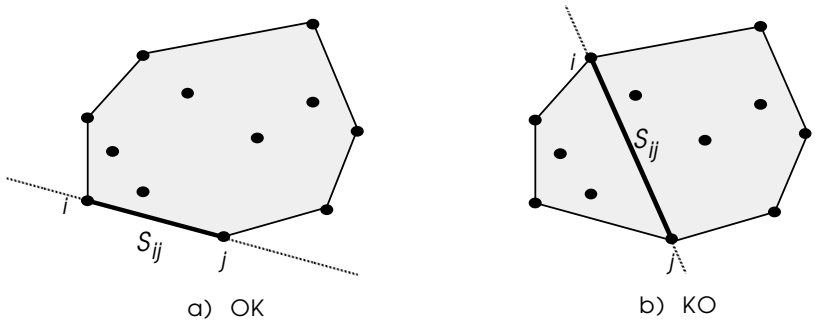

Figura 1.2: a) Il segmento S_{ij} è uno spigolo dell'inviluppo convesso; b) il segmento S_{ij} non è uno spigolo dell'inviluppo convesso.

Esempio (Inviluppo convesso) La Figura 1.1 mostra un poligono non convesso, un poligono convesso, un insieme di punti e il loro inviluppo convesso. ■

Un algoritmo banale per trovare l'inviluppo convesso si basa sulla seguente caratterizzazione della soluzione. Un poligono può essere rappresentato per mezzo dei suoi spigoli, cioè dei segmenti di retta che giacciono sul suo bordo. Si consideri allora un generico segmento S_{ij} che unisce due punti distinti p_i e p_j, $1 \leq i < j \leq n$, e si consideri la retta che include S_{ij}. Tale retta divide il piano in due parti (dette *semipiani chiusi*, ognuno dei quali comprende anche la retta). Se tutti i rimanenti $n-2$ punti stanno dalla stessa parte rispetto alla retta che passa per S_{ij}, cioè stanno tutti nello stesso semipiano, inclusa la retta, allora S_{ij} è uno spigolo dell'inviluppo convesso e p_i e p_j ne sono due vertici (si veda la Figura 1.2). Ripetendo per tutte le $n(n-1)/2$ coppie distinte di punti, si possono trovare tutti gli spigoli dell'inviluppo convesso. Individuati gli spigoli, il poligono può essere facilmente rappresentato tramite la sequenza dei suoi vertici, elencati secondo il verso antiorario (oppure orario).

Per realizzare tale algoritmo, vediamo innanzitutto come sia possibile verificare in tempo $O(1)$ se due punti stanno dalla stessa parte rispetto ad una retta oppure no. Ricordiamo che l'equazione di una retta è $y = ax + b$, dove $a = \Delta_y/\Delta_x$ ne è il coefficiente angolare. La retta divide il piano nei due "semipiani" dati dalle disequazioni $y \leq ax + b$ e $y \geq ax + b$.

Poiché la retta potrebbe essere verticale (cioè Δ_x potrebbe valere 0), si moltiplica per Δ_x ottenendo $y\Delta_x - x\Delta_y - b\Delta_x = 0$, dalla quale è possibile ricavare $b\Delta_x$ se sono date le coordinate di un punto $p = (x_p, y_p)$ che appartiene alla retta: $b\Delta_x = y_p\Delta_x - x_p\Delta_y$. Consideriamo uno dei due semipiani, per esempio quello dato dalla disequazione

$$y\Delta_x - x\Delta_y - y_p\Delta_x + x_p\Delta_y \leq 0.$$

Se un punto $q = (x_q, y_q)$ appartiene a tale semipiano, allora deve valere

$$y_q\Delta_x - x_q\Delta_y - y_p\Delta_x + x_p\Delta_y \leq 0,$$

dalla quale segue che

$$(y_q - y_p)\Delta_x - (x_q - x_p)\Delta_y \leq 0.$$

Pertanto, per ogni punto q che appartiene a tale semipiano, la precedente quantità $(y_q - y_p)\Delta_x - (x_q - x_p)\Delta_y$ deve essere negativa o nulla, mentre per ogni punto q appartenente all'altro semipiano la stessa quantità deve essere positiva o nulla. Ne consegue che due punti q e q' stanno dalla stessa parte se le rispettive quantità hanno lo stesso segno, ovvero se il loro prodotto è positivo o nullo.

Consideriamo adesso l'operazione stessaparte(), che prende in input una retta (rappresentata da due punti p_1 e p_2) e due punti p e q, e restituisce **true** se essi stanno dalla stessa parte rispetto alla retta (incluso il caso in cui almeno uno dei punti giaccia su di essa), e **false** se stanno da parti opposte rispetto ad essa. I punti sono realizzati come record contenenti due campi reali x e y. Dati una retta individuata dai due punti p_1 ed p_2, e due punti p e q, la funzione stessaparte(p_1, p_2, p, q) verifica in tempo $O(1)$ se le quantità discusse precedentemente hanno lo stesso segno, cioè se il loro prodotto è non negativo.

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

real $dx = p_2.x - p_1.x$
real $dy = p_2.y - p_1.y$
real $dx_1 = p.x - p_1.x$
real $dy_1 = p.y - p_1.y$
real $dx_2 = q.x - p_2.x$
real $dy_2 = q.y - p_2.y$
return $((dx \cdot dy_1 - dy \cdot dx_1) \cdot (dx \cdot dy_2 - dy \cdot dx_2) \geq 0)$

Utilizzando questa funzione, è un facile esercizio scrivere una procedura per trovare l'inviluppo convesso. Infatti, fissati due punti qualsiasi p_i e p_j, cioè un segmento S_{ij}, per vedere se i rimanenti $n-2$ punti stanno dalla stessa parte rispetto alla retta che passa per S_{ij} basta tenere fisso uno di tali punti, per esempio p, ed eseguire stessaparte(p_i, p_j, p, q) per $n-3$ volte al variare di q per ciascuno dei rimanenti $n-3$ punti. Pertanto verificare se un segmento S_{ij} è uno spigolo dell'inviluppo convesso richiede $O(n)$ tempo. Poiché ci sono in totale $n(n-1)/2$ segmenti S_{ij} distinti, si possono individuare tutti gli spigoli dell'inviluppo convesso con complessità $O(n^3)$.

1.1.1 Algoritmo di Graham

L'algoritmo appena illustrato non è molto efficiente, perché esamina gli n punti in modo troppo caotico. Vediamo come un esame sistematico dei punti, basato su un ordinamento ed un *backtrack*, permetta di progettare un algoritmo più efficiente, proposto da Graham (1972).

Innanzitutto, si osservi che in un insieme di punti, quello con ordinata minima è un vertice dell'inviluppo convesso. Inoltre, tracciando una retta orizzontale in tale punto e facendola ruotare

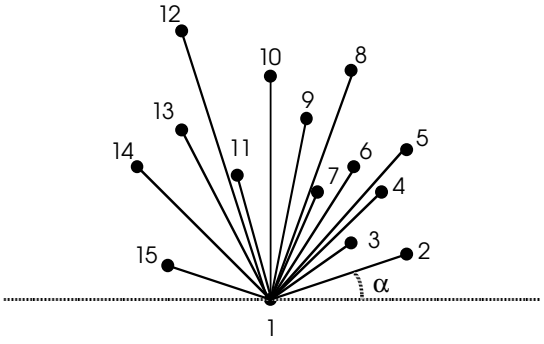

Figura 1.3: Ordinamento dei punti in base all'angolo α formato con l'asse orizzontale quando sono connessi al punto p_1.

intorno al punto stesso in senso antiorario si incontrano in sequenza tutti i rimanenti punti (uno per volta, se non ci sono due punti "allineati"). Come illustrato nella Figura 1.3, i punti si possono rinumerare in base all'angolo α che la retta passante per ciascun punto p_i e quello di ordinata minima (il punto p_1) forma con l'asse orizzontale che passa per p_1. Se ci sono più punti con ordinata minima, allora quelli con ascissa minima e massima, sono vertici dell'inviluppo convesso, mentre gli altri giacciono sullo spigolo che unisce i due punti con ascissa minima e massima e possono essere eliminati, e può essere scelto come punto p_1 quello di ascissa massima. Allo stesso modo, se ci sono più punti allineati, cioè con lo stesso angolo α, allora possono essere eliminati tutti tranne il più distante da p_1, cioè quello di ordinata massima, perché tali punti non possono essere vertici dell'inviluppo convesso. Una volta che i punti sono ordinati per angolo α crescente, l'eliminazione dei punti allineati può essere eseguita in tempo $O(n)$ con una scansione della sequenza ordinata dei punti. Pertanto, senza perdere in generalità, si può assumere che non ci siano punti allineati, tranne eventualmente l'ultimo punto, che può avere la stessa ordinata del primo, ma ascissa minore.

Si osservi che, una volta ordinati, il secondo e l'ultimo punto dell'ordinamento fanno parte dell'inviluppo convesso, oltre ovviamente a p_1. È possibile considerare i punti $p_1, \dots p_n$ per indici (cioè angoli α) crescenti e costruire in modo incrementale l'inviluppo convesso "corrente" dei primi i punti $p_1, \dots, p_i$, per $i = 3, 4, \dots, n$. Ovviamente, se $i = 3$, allora i punti p_1, p_2 e p_3 sono i vertici dell'inviluppo convesso di p_1, p_2 e p_3. Si assuma di aver costruito l'inviluppo convesso dei primi $i-1$ punti, dato dalla sequenza dei suoi vertici in senso antiorario, e si provi ad aggiungere il nuovo punto p_i. Chiaramente, sia p_i che p_1 sono vertici dell'inviluppo convesso di $p_1, \dots, p_i$. Posto $j = i-1$, si verifica se i punti p_i e p_1 stanno dalla stessa parte rispetto alla retta che passa per gli ultimi due vertici p_{j-1} e p_j dell'inviluppo di $p_1, \dots, p_{i-1}$; se questo non avviene, il punto p_j non può essere un vertice dell'inviluppo convesso di $p_1, \dots, p_i$ e va eliminato. Il procedimento è ripetuto aggiornando j a ritroso e considerando la retta che passa per gli ultimi due vertici p_h e p_j con $h < j$, dell'inviluppo gli ultimi due vertici p_h e p_j dell'inviluppo di $p_1, \dots, p_{i-1}$, continuando ad eliminare l'ultimo vertice p_j se i punti p_i e p_1 non stanno dalla stessa parte rispetto alla retta. Tale procedimento di eliminazione ha sicuramente termine, al più, quando si arriva a considerare la retta che passa per p_1 e p_2, poiché tali punti sono vertici dell'inviluppo convesso di $p_1, \dots, p_i$ per ogni i, $2 \le i \le n$. Queste osservazioni suggeriscono il seguente algoritmo informale:

La dimostrazione di correttezza dell'algoritmo segue facilmente per induzione su i in accordo alle considerazioni fatte precedentemente. La fase di *backtrack* avviene all'interno del ciclo **for**, dove si eliminano i punti che sono vertici dell'inviluppo convesso di $p_1, \dots, p_{i-1}$, ma che non lo sono più per $p_1, \dots, p_i$.

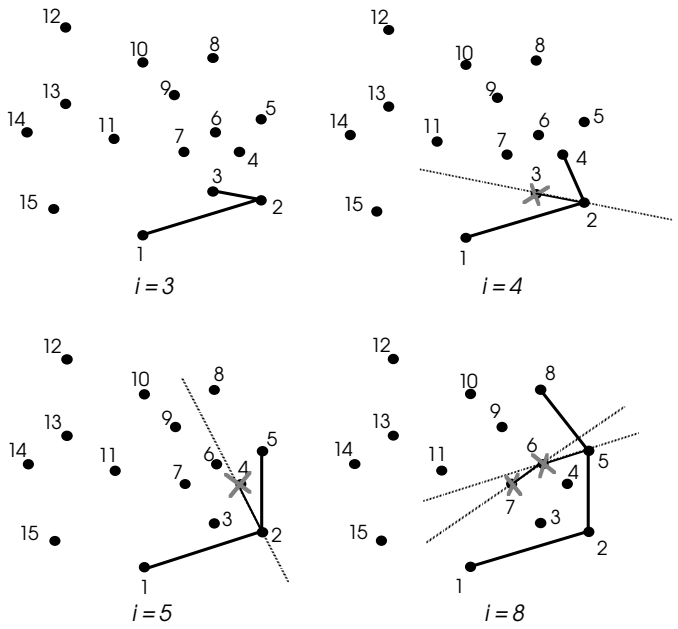

Figura 1.4: Esecuzione dell'algoritmo di Graham fino ad $i = 8$.

Esempio (Algoritmo di Graham) L'esecuzione dell'algoritmo di Graham è illustrata nella Figura 1.4, dove sono evidenziati gli spigoli dell'inviluppo "corrente" (in nero), le rette (tratteggiate), ed i punti che vengono eliminati ad ogni iterazione (con la croce), fino ad $i = 8$. Si noti che l'aggiunta del punto p_8 provoca l'eliminazione di due punti (p_7 e p_6), ma che ogni punto è inserito solo una volta ed è eliminato al più una volta. ■

Il modo più intuitivo di memorizzare l'inviluppo "corrente" consiste nell'utilizzare una pila S nella quale sono accatastati i vertici dell'inviluppo. In questo modo, il nodo p_j eventualmente da eliminare è sempre quello in testa alla pila, mentre il nuovo nodo p_i è anch'esso sempre inserito in testa alla pila. Per usare più semplicemente la pila, è conveniente introdurre la nuova operazione top2(), che restituisce il valore del secondo elemento a partire dalla testa della pila, senza modificare la pila.

Analizziamo la complessità del ciclo **for**. Tale ciclo è eseguito $n-2$ volte ed al suo interno tutte le istruzioni, inclusa la condizione del ciclo **while**, richiedono tempo $O(1)$. Pertanto la complessità del **for** dipende esclusivamente dal numero di ripetizioni del **while**, che a sua volta dipende dal numero di punti che sono cancellati dalla pila. Ogni punto è inserito nell'inviluppo "corrente" una ed una sola volta ed è eliminato al più una volta. Quando un punto è eliminato, esso non viene più riconsiderato. Inoltre, ogni volta che è eseguito il corpo del **while** è eliminato un punto. Pertanto, la complessità del ciclo **for** è $\Theta(n)$. La complessità di tutto l'algoritmo è invece $O(n \log n)$, perché dominata dalla fase di ordinamento dei punti.

1.2 Enumerazione tramite backtrack

Introducendo le classi di problemi (decisionali, di ricerca, di ottimizzazione) [cfr. Capitolo 11], abbiamo definito il concetto di *soluzione ammissibile*, che soddisfa un certo insieme di criteri. In

```
STACK graham(POINT[] p, int n)
    int min = 1
    for i = 2 to 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 gli eventuali punti "allineati" tranne i più lontani da p_1, aggiornando n }
    STACK S = Stack()
    S.push(p_1); S.push(p_2)
    for i = 3 to n do
        while not stessaparte(S.top(), S.top2(), p_1, p_i) do
            S.pop()
        S.push(p_i)
    return S
```

alcuni casi, può essere necessario non solo cercare una soluzione ammissibile, ma anche essere in grado di elencarle tutte, dalla prima all'ultima; questo è il problema dell'*enumerazione*.

Esempio (Permutazioni) Dato l'insieme $A = \{1,\ldots,n\}$, si consideri il problema di elencare tutte le permutazioni di A, ovvero tutte le sequenze ordinate di n elementi in cui gli elementi di A compaiono una e una sola volta. ■

Esempio (Sottoinsiemi) Dato l'insieme $A = \{1,\ldots,n\}$, si consideri il problema di elencare tutti i sottoinsiemi di A contenenti esattamente k elementi. ■

In questi esempi, sottoinsiemi di k elementi e permutazioni rappresentano le soluzioni ammissibili.

Fra le motivazioni per elencare tutte le soluzioni ammissibili, possiamo ricordare le seguenti:

- Contare il numero di soluzioni ammissibili: tutte le volte che si ottiene una soluzione ammissibile, un contatore viene incrementato. Si tenga tuttavia conto che in molti casi il calcolo combinatorio può fornire risposte analitiche a tale problema. Ad esempio, il numero di permutazioni di un insieme di n elementi è $n!$ e il numero di sottoinsiemi di dimensione k di un insieme di n elementi è $n!/(n-k)!$
- Identificare la soluzione ottima, in base ad una certa funzione di costo. Anche in questo caso, è possibile sfruttare l'enumerazione elencando tutte le possibili soluzioni ammissibili, valutando la funzione di costo per ognuna di esse, e memorizzando quella (o quelle) con costo ottimale. Sebbene sia preferibile adottare tecniche più efficienti quali programmazione dinamica e *greedy*, vi sono innumerevoli problemi [cfr. Capitolo 18] in cui questo non è possibile, e l'enumerazione è una delle possibili tecniche risolutive .
- Produrre (se esiste) almeno una soluzione ammissibile. Questo è il caso di molti giochi matematici, come il Sudoku. Questo problema può essere risolto semplicemente terminando l'enumerazione alla prima occorrenza di una soluzione.

Per enumerare tutte le soluzioni ammissibili, si utilizza un approccio a "forza bruta", basato su *backtrack*, che esplora lo spazio delle soluzioni in modo sistematico. Una soluzione viene

rappresentata come un vettore $S[1 \dots n]$. Il contenuto degli elementi $S[i]$ è preso da un insieme di scelte dipendente dal particolare problema. Ad ogni passo, si parte da una *soluzione parziale* $S[1 \dots i-1]$; se è possibile, si estende tale soluzione con una delle possibili scelte in una soluzione parziale $S[1 \dots i]$ e si valuta la nuova soluzione parziale ricorsivamente; altrimenti, si opera un passo di *backtrack* (ritornando indietro nella ricorsione) e si effettua una nuova scelta fra quelle possibili a partire da $S[1 \dots i-1]$.

La procedura enumerazione() realizza questo schema. Ad ogni chiamata della procedura, l'insieme C delle possibili opzioni per la scelta i-esima viene calcolato a partire dalle scelte precedenti $S[1 \dots i-1]$. Tutte le possibili strade vengono quindi intraprese: uno dopo l'altro, tutti i valori di C vengono scelti e memorizzati in $S[i]$; se $S[1, \dots, i]$ è una soluzione ammissibile, tale soluzione viene elaborata in qualche modo (stampata, contata, valutata in base ad una funzione di costo, etc.) dalla procedura processSolution(). Infine, la procedura enumerazione() chiama ricorsivamente se stessa, dando per assodate le prime i scelte e spostandosi sull'$(i+1)$-esima. La ricorsione termina quando l'albero è stato visitato completamente.

Per supportare il caso in cui è sufficiente generare una e una sola soluzione ammissibile, le procedure enumerazione() e processSolution() restituiscono un valore booleano, vero se una soluzione ammissibile è stata trovata, falso altrimenti.

```
boolean enumerazione(item[] S, int n, int i, ...)
    SET C = choices(S, n, i, ...)        % Determina C in funzione di S[1...i-1]
    foreach c ∈ C do
        S[i] = c
        if S[1...i] è una soluzione ammissibile then
            if processSolution(S, n, i, ...) then return true
        if enumerazione(S, n, i+1, ...) then return true
    return false
```

Applichiamo lo schema precedente all'Esempio 1.2. In questo caso, la scelta possibile è se inserire o meno un elemento; l'insieme delle scelte è quindi dato da {**true**,**false**} e il vettore S costituisce una rappresentazione booleana dell'insieme generato. L'insieme delle scelte è tuttavia vuoto quando tutti gli elementi sono già stati esaminati, ovvero quando $i > n$.

Dato un insieme di n elementi, l'elemento i-esimo può essere presente oppure no. In altre parole, l'insieme delle scelte è dato da $\{0,1\}$ che rappresentano i valori vero e falso e il vettore S è una rappresentazione booleana dell'insieme generato. Una prima versione dell'algoritmo potrebbe essere la seguente:

```
subsets(int[] S, int n, int k, int i, int count)
    SET C = iif(i ≤ n, {0,1}, ∅)
    foreach c ∈ C do
        S[i] = c
        count = count + S[i]
        if i = n and count = k then
            processSolution(S, n)
        subsets(S, n, k, i+1, count)
        count = count - S[i]
```

Questo algoritmo, invocato da subsets$(S,n,k,1,0)$, genera tutti i sottoinsiemi di un insieme di n elementi, e invoca processSolution() sui soli sottoinsiemi di k elementi. A tal scopo, la variabile *count* conta quanti elementi sono presenti nel sottoinsieme corrente. Il costo della procedura è ovviamente $O(2^n)$.

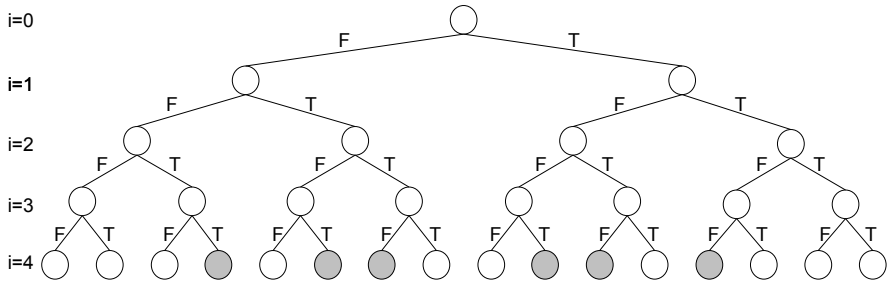

Figura 1.5: L'albero delle scelte per la procedura subsets() con $n = 4$, $k = 2$.

Esempio (Albero delle scelte per sottoinsieme) L'enumerazione può essere visualizzata come la visita in profondità di un albero di scelte, dove ad ogni nodo corrisponde una soluzione (parziale oppure no). Nel caso di subsets(), l'albero è binario e completo, con tutte le sue foglie al livello n. La figura Figura 1.5 mostra l'albero con $n = 4$, $k = 2$. I sottoinsiemi con esattamente $k = 2$ nodi sono evidenziati in grigio.

In realtà, non è necessario analizzare completamente l'albero. Ci sono tanti casi in cui una soluzione parziale non potrà mai generare una soluzione ammissibile: ad esempio, se un insieme ha già k elementi, oppure se le scelte ancora da prendere non sono sufficienti a ottenere un insieme di k elementi. In questo caso, si dice che l'albero di decisione viene "*potato*" (in inglese *pruned*): i rami che non possono generare (ulteriori) soluzioni possibili vengono ignorati.

Il nostro algoritmo può essere semplicemente modificato cambiando l'insieme delle scelte, e modificando la condizione di ammissibilità, ma il tempo resta comunque $O(2^n)$.

```
subsets(int[] S, int n, int k, int i, int count)
    SET C = iif(count < k and count + (n − i + 1) ≥ k, {0,1}, ∅)
    foreach c ∈ C do
        S[i] = c
        count = count + S[i]
        if count = k then
            processSolution(S, i)
        else
            subsets(S, n, k, i + 1, count)
        count = count − S[i]
```

1.3 Reality check

Per concludere questo capitolo, vediamo un gioco matematico che può essere risolto agilmente tramite *backtrack*. Nel Sudoku è data una tabella formata da 9 righe e 9 colonne, suddivisa in 9 sottotabelle di 3 righe e 3 colonne ciascuna, dove alcune caselle sono riempite con numeri compresi tra 1 e 9. Si devono riempire le caselle vuote con numeri compresi tra 1 e 9 in modo che ciascuna riga, ciascuna colonna e ciascuna sottotabella contenga tutti i numeri tra 1 e 9. Per esempio, per la tabella in ingresso

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

la soluzione è:

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

L'algoritmo risolutivo è mostrato con la procedura sudoku(). La tabella del Sudoku (parzialmente inizializzata) viene memorizzata in una matrice $S[0\ldots8][0\ldots8]$ di interi, presa in input. Assumiamo che alcune caselle siano già riempite da numeri in $\{1,\ldots,9\}$, mentre quelle vuote abbiano valore 0. Utilizziamo un indice i compreso fra 0 e 80 per indicare la casella che stiamo analizzando; questo indice viene tramutato in una coppia di coordinate (x,y) nella matrice. Se la casella è già piena, l'unica scelta possibile è data dal valore già presente. Altrimenti, un valore $c \in \{1,\ldots,9\}$ è aggiunto all'insieme delle scelte C se e solo se scrivere c in $S[x,y]$ è una mossa ammissibile, condizione verificata dalla chiamata check(S,x,y,c). L'enumerazione termina quando viene trovata una soluzione completa (ovvero, è stata riempita l'ultima casella) e la procedura restituisce **true**. Se non esiste una soluzione ammissibile (l'input non è corretto), la procedura restituisce **false**. Il costo della procedura è... costante!, in quanto il Soduku ha dimensione fissa (ma la costante è enorme). Ma è possibile generalizzare il problema a tabelle di dimensione $n^2 \times n^2$ da riempire con numeri nell'insieme $\{1,2,\ldots,n^2\}$, dove il Sudoku classico corrisponde a $n = 3$. In questo caso il costo della procedura è superpolinomiale.

```
boolean sudoku(int[][] S, int i)
    SET C = Set()
    int x = i mod 9
    int y = ⌊i/9⌋
    if i ≤ 80 then
        if S[x,y] ≠ 0 then
            C.insert(S[x,y])
        else
            for c = 1 to 9 do
                if check(S,x,y,c) then C.insert(c)
    int old = S[x,y]
    foreach c ∈ C do
        S[x,y] = c
        if i = 80 then
            processSolution(S,n)
            return true
        if sudoku(S,i+1) then return true
    S[x,y] = old
    return false
```

1.4 Esercizi

Esercizio 1.1 (Angoli α)**.** Se dx e dy sono le differenze tra le ascisse e le ordinate dei punti p_i e p_1, calcolare l'angolo $\alpha = \tan^{-1} dy/dx$ con la funzione di libreria per l'arcotangente richiede parecchio tempo. Dato che α serve solo nella fase di ordinamento dei punti dell'algoritmo di Graham, si può utilizzare una quantità diversa dall'angolo α, più veloce da calcolare e che produca lo stesso ordinamento. Si dimostri che $dy/(dy+dx)$ è adatta allo scopo e si scriva una opportuna funzione per calcolarla in tempo $O(1)$, tenendo conto dei casi in cui dx e dy possono essere positive, negative, o nulle.

Esercizio 1.2 (Tutte le permutazioni)**.** Si scriva una procedura basata sulla tecnica *backtrack* per stampare tutte le permutazioni dei valori contenuti in un insieme A.

Esercizio 1.3 (Tutte le dismutazioni)**.** Una *dismutazione* π dell'insieme $\{1,\ldots,n\}$ è una permutazione in cui nessun elemento è nella posizione "corretta", ovvero $\pi_i \neq i$, per $1 \leq i \leq n$. Si scriva una procedura basata sulla tecnica *backtrack* per stampare tutte le dismutazioni dei valori contenuti in un insieme A.

Esercizio 1.4 (Tutti i sottoinsiemi con somma k)**.** Dato un vettore di n interi positivi $A[1\ldots n]$, si scriva una procedura basata sulla tecnica *backtrack* per stampare tutti i sottoinsiemi la cui sommatoria sia pari a k.

Esercizio 1.5 (n regine)**.** Il problema scacchistico delle n regine richiede di collocare n regine in un scacchiera $n \times n$, in modo che nessuna delle regine ne minacci un'altra - ovvero due regine non possono stare sulla stessa riga, colonna o diagonale. Scrivere un algoritmo che ritorni una disposizione delle regine seguendo queste regole, progettando anche un metodo per codificare la soluzione.

Esercizio 1.6 (Giro di cavallo)**.** Il problema del "giro di cavallo" richiede di posizionare un cavallo su una casella di una scacchiera $n \times n$, e identificare un percorso che tocchi tutte le caselle una e una sola volta, seguendo le regole di movimento dei cavalli negli scacchi.

Esercizio 1.7 (Sudoku)**.** Si completi la procedura sudoku() scrivendo la procedura check().

Esercizio 1.8 (Anagrammi)**.** Si supponga di avere una funzione $d(S,i,j)$ che, dato un vettore S di caratteri, restituisce vero o falso a seconda che $S[i \dots j]$ sia o non sia una parola della lingua italiana. Scrivere un programma per generare anagrammi, ovvero un programma che prende in input una stringa di caratteri (senza spazi) e genera tutte le possibili stringhe formate solo da parole italiane separate da spazi.

1.5 Soluzioni

Soluzione Esercizio 1.2

Seguendo lo schema discusso sopra, ad ogni passo è possibile scegliere uno degli elementi che non è già stato scelto in precedenza. Per fare questo, rimuoviamo da A l'elemento scelto, e lo reinseriamo al momento di fare un passo di *backtrack*. La chiamata iniziale è permutations$(A,n,S,1)$.

```
permutations(SET A, int n, item[] S, int i)
    foreach c ∈ A do
        S[i] = c
        A.remove(c)
        if A.isEmpty() then processSolution(S,n)
        permutations(A,n,S,i+1)
        A.insert(c)
```

Soluzione Esercizio 1.3

La soluzione è simile a quella vista per le permutazioni, con l'unico accorgimento che l'elemento c scelto nel ciclo **foreach** deve essere diverso da i.

Soluzione Esercizio 1.5

Una regina ne può minacciare un'altra se si trovano sulla stessa riga, colonna o diagonale. Data una scacchiera $n \times n$, è possibile rappresentare la soluzione in vari modi, molti dei quali poco efficienti. Ad esempio,

- se l'insieme delle possibili scelte è metto / non metto una regina in una particolare casella, la soluzione è rappresentata da una matrice $n \times n$ di booleani e la dimensione dello spazio da analizzare (salvo potature) è pari a 2^{n^2};
- se l'insieme delle possibili scelte contiene le n^2 posizioni in cui una delle n regine può essere collocata, la soluzione è rappresentata da un vettore di n interi in $\{1,\dots,n^2\}$ e la dimensione dello spazio da analizzare (salvo potature) è pari a $(n^2)^n$.

Queste dimensioni sono molto grandi; ma è possibile semplificare le cose notando che ogni riga deve contenere esattamente una regina, e lo stesso vale per le colonne. Quindi, la posizione di ognuna delle n regine può essere rappresentata da un numero in $\{1,\dots,n\}$, e l'insieme delle posizioni deve essere una permutazione di tale insieme. Lo spazio da analizzare è quindi grande $n!$. Per completare la soluzione, lo spazio deve essere ulteriormente potato tenendo conto delle diagonali.

Soluzione Esercizio 1.6

Rappresentiamo una soluzione con una matrice A di dimensione $n \times n$, dove $n = 8$ per la scacchiera classica. Poniamo $A[x,y] = k$ se il cavallo ha visitato la casella x,y al passo k-esimo. La funzione cavallo() prende in input la scacchiera e la sua dimensione n, nonché il contatore di passi k e la posizione attuale x,y. Al passo k inserisce k nella cella $A[x,y]$, e se abbiamo riempito totalmente la scacchiera restituisce **true**. Altrimenti, prova a fare un'ulteriore mossa delle 8 possibili mosse del cavallo, utilizzando due vettori $X = [-1,+1,+2,+2,+1,-1,-2,-2]$ e

$Y = [-2,-2,-1,+1,+2,+2,+1,-1]$ per calcolare la prossima posizione nx, ny; una mossa è ammissibile se il cavallo non esce dalla scacchiera e se la casella non è già occupata. La chiamata iniziale è cavallo$(A,n,1,x,y)$, dove A è inizializzato a tutti zero e x,y è una posizione qualsiasi. Al termine, se la risposta è **true**, la matrice A contiene il percorso del cavallo.

```
boolean cavallo(int[][] A, int n, int k, int x, int y)
A[x,y] = k
if k = n^2 then return true
for i = 1 to 8 do
    nx = x + X[i]
    ny = y + Y[i]
    if nx ≥ 1 and nx ≤ n and ny ≥ 1 and ny ≤ n and A[nx,ny] = 0 then
        if cavallo(A,n,k+1,nx,ny) then return true
A[x,y] = 0
return false
```

Soluzione Esercizio 1.7

La procedura check(S,x,y,c) controlla che l'aggiunta del numero c nella casella $S[x,y]$ non sia in contrasto con i valori già presenti.

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