

Algoritmi e Strutture Dati

Backtracking

Alberto Montresor

Università di Trento

2018/11/07

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Sommario

- 1 Introduzione
- 2 Enumerazione
 - Sottoinsiemi
 - Permutazioni
 - Giochi
- 3 Backtracking iterativo
 - Involuppo convesso

Introduzione

Classi di problemi (decisionali, di ricerca, di ottimizzazione)

- Definizioni basate sul concetto di **soluzione ammissibile**: una soluzione che soddisfa un certo insieme di criteri

Problemi tipici

- Costruire almeno una o tutte le soluzioni ammissibili
- Contare le soluzioni ammissibili
- Trovare le soluzioni ammissibili "più grandi", "più piccole" o in generale "ottimali"

Problemi tipici

Enumerazione

- Elencare alitmicamente tutte le soluzioni ammissibili (spazio di ricerca)
- Esempio: elencare tutte le permutazioni di un insieme

Costruire almeno una soluzione

- Si utilizza l'algoritmo per l'enumerazione, fermandosi alla prima soluzione disponibile
- Esempio: identificare una sequenza di mosse nel gioco del 15

Problemi tipici

Contare le soluzioni

- In alcuni casi, è possibile contare in modo analitico
- Esempio: contare il numero di sottoinsiemi di k elementi presi da un insieme di n elementi
- In altri casi, si costruiscono le soluzioni e si contano
- Esempio: numero di sottoinsiemi di un insieme di interi S la cui somma è un numero primo.

$$\frac{n!}{k!(n-k)!}$$

Problemi tipici

Trovare le soluzioni ottimali

- Si enumerano tutte le soluzioni, che vengono valutate tramite una funzione di costo
- Si possono utilizzare altre tecniche:
 - Programmazione dinamica, greedy
 - Tecniche risolutive per problemi intrattabili
- Esempio: Circuito hamiltoniano (commesso viaggiatore)

Costruire tutte le soluzioni

Per costruire tutte le soluzioni, si utilizza un approccio "brute-force"

- Si esamina interamente lo spazio delle possibili soluzioni
- A volte è l'unica strada possibile
- La potenza dei computer moderni rende "affrontabili" problemi di dimensioni medio-piccole
 - $10! = 3.63 \cdot 10^6$ (permutazione di 10 elementi)
 - $2^{20} = 1.05 \cdot 10^6$ (sottoinsieme di 20 elementi)
- Inoltre, a volte lo spazio delle soluzioni non deve essere analizzato interamente

Backtracking

Filosofia

- *"Prova a fare qualcosa, e se non va bene, disfalo e prova qualcos'altro"*
- *"Ritenta, e sarai più fortunato"*

Come funziona?

- Un metodo sistematico per iterare su tutte le possibili istanze di uno spazio di ricerca
- E' una tecnica algoritmica che, come altre, deve essere personalizzata per ogni applicazione individuale

Organizzazione generale

Organizzazione generale

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

Esempi

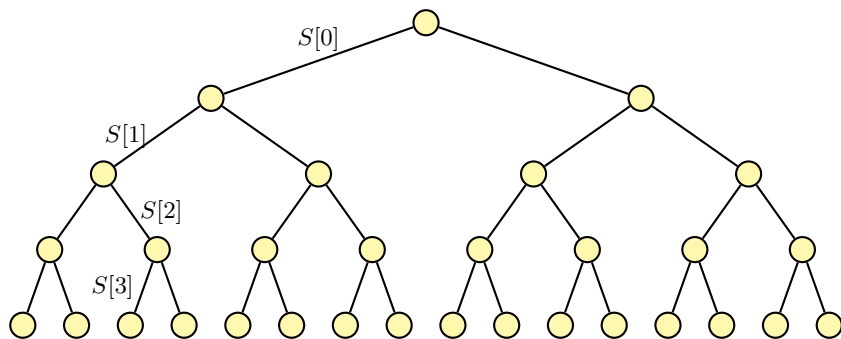
- C insieme generico, possibili soluzioni **permutazioni** di C
- C insieme generico, possibili soluzioni **sottoinsiemi** di C
- C mosse di gioco, possibili soluzioni **sequenze di mosse**
- C archi di un grafo, possibili soluzioni **percorsi sul grafo**

Soluzioni parziali

- Ad ogni passo, partiamo da una soluzione parziale $S[1 \dots k]$ in cui $k \geq 0$ scelte sono state prese
- Se $S[1 \dots k]$ è una soluzione ammissibile, la "processiamo"
 - Viene stampata, contata, valutata
 - Si può decidere di terminare o continuare elencando tutte le soluzioni
- Se $S[1 \dots k]$ non è una soluzione completa
 - Se è possibile, estendiamo $S[1 \dots k]$ con una delle possibili scelte in una soluzione $S[1 \dots k + 1]$
 - Altrimenti, "cancelliamo" l'elemento $S[k]$ (backtrack) e ripartiamo dalla soluzione $S[1 \dots k - 1]$

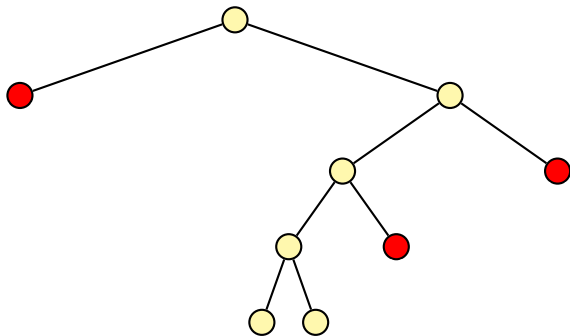
Albero delle decisioni

- **Albero di decisione** \equiv Spazio di ricerca
- **Radice** \equiv Soluzione parziale vuota
- **Nodi interni dell'albero di decisione** \equiv Soluzioni parziali
- **Foglie in un albero di decisione** \equiv Soluzioni ammissibili



Pruning

- “Rami” dell’albero che sicuramente non portano a soluzioni ammissibili possono essere “potati” (**pruned**)
- La valutazione viene fatta nelle soluzioni parziali radici del sottoalbero da potare



Backtracking - Due possibili approcci

Ricorsivo

- Lavora tramite una visita in profondità nell'albero delle scelte, basata su un approccio ricorsivo

Iterativo

- Utilizza un approccio greedy, eventualmente tornando sui propri passi
- Esempio: **Inviluppo convesso**
- Esempio: **String matching**

Enumerazione

```
boolean enumeration(ITEM[]  $S$ , int  $n$ , int  $i$ , ...)
```

```
SET  $C$  = choices( $S$ ,  $n$ ,  $i$ , ...)           % Determina  $C$  in funzione di  

 $S[1 \dots i - 1]$ 
```

```
foreach  $c \in C$  do
```

```

|    $S[i] = c$ 
|   if isAdmissible( $S$ ,  $n$ ,  $i$ ) then
|       |   if processSolution( $S$ ,  $n$ ,  $i$ , ...) then
|           |   return true
|       if enumeration( $S$ ,  $n$ ,  $i + 1$ , ...) then
|           return true
|
```

```
return false
```

Enumerazione

- S : vettore contenente le soluzioni parziali $S[1 \dots i]$
- i : indice corrente
- \dots : informazioni aggiuntive
- C è l'insieme dei possibili candidati per estendere la soluzione
- $\text{isAdmissible}()$: restituisce **true** se $S[1 \dots i]$ è una soluzione ammissibile
- $\text{processSolution}()$: restituisce
 - **true** per bloccare l'esecuzione alla prima soluzione ammissibile,
 - **false** per esplorare tutto l'albero

Esempio 1

Elencare tutti i sottoinsiemi dell'insieme $\{1, \dots, n\}$

```
subsets(int[] S, int n, int i)
```

```
SET C =  $\text{iif}(i \leq n, \{0, 1\}, \emptyset)$ 
```

```
foreach c  $\in C$  do
```

```
    S[i] = c
```

```
    if i = n then
```

```
        processSolution(S, n)
```

```
    subsets(S, n, i + 1)
```

Esempio 1 (Versione più "pulita")

Elencare tutti i sottoinsiemi dell'insieme $\{1, \dots, n\}$

```
subsets(int[] S, int n, int i)
```

```
  foreach  $c \in \{0, 1\}$  do
```

```
     $S[i] = c$ 
```

```
    if  $i = n$  then
```

```
      | processSolution( $S, n$ )
```

```
    else
```

```
      | subsets( $S, n, i + 1$ )
```

Esempio 1

- Non c'è pruning. Tutto lo spazio possibile viene esplorato. Ma questo avviene per definizione
- Complessità $O(n2^n)$
- In che ordine vengono stampati gli insiemi?
- E' possibile pensare ad una soluzione iterativa, ad-hoc? (non-backtracking)

Esempio 1

```
subsets(int n)
for  $j = 0$  to  $2^n - 1$  do
    print '{'
    for  $i = 0$  to  $n - 1$  do
        if ( $j$  and  $2^i$ )  $\neq 0$  then
            print  $i$ 
    print '}'
```

Esempio 2

Elencare tutti i sottoinsiemi di dimensione k di un insieme $\{1, \dots, n\}$

- Versione iterativa
 - Qual è il costo?
- Soluzione basata su backtracking
 - Possiamo potare?

Esempio 2 - Tentativo 1

```
subsets(int[] S, int n, int k, int i)
```

```
SET  $C = \text{iif}(i \leq n, \{0, 1\}, \emptyset)$ 
```

```
foreach  $c \in C$  do
```

```
     $S[i] = c$ 
```

```
    if  $i = n$  then
```

```
        int  $count = 0$ 
```

```
        for  $j = 1$  to  $n$  do
```

```
             $count = count + S[j]$ 
```

```
        if  $count = k$  then
```

```
            processSolution( $S, n$ )
```

```
    subsets( $S, n, k, i + 1$ )
```

Esempio 2 - Tentativo 2

```
subsets(int[] S, int n, int k, int i, int count)
```

```
SET  $C = \text{iif}(i \leq n, \{0, 1\}, \emptyset)$ 
```

```
foreach  $c \in C$  do
```

```
     $S[i] = c$ 
```

```
     $count = count + S[i]$ 
```

```
    if  $i = n$  and  $count = k$  then
```

```
        processSolution( $S, n$ )
```

```
    subsets( $S, n, k, i + 1, count$ )
```

```
     $count = count - S[i]$ 
```

Esempio 2 - Tentativo 3 (Corretto)

```
subsets(int[] S, int n, int k, int i, int count)
```

```
SET  $C = \text{iif}(count < k \text{ and } count + (n - i + 1) \geq k, \{0, 1\}, \emptyset)$ 
```

```
foreach  $c \in C$  do
```

```
     $S[i] = c$ 
```

```
     $count = count + S[i]$ 
```

```
    if  $count = k$  then
```

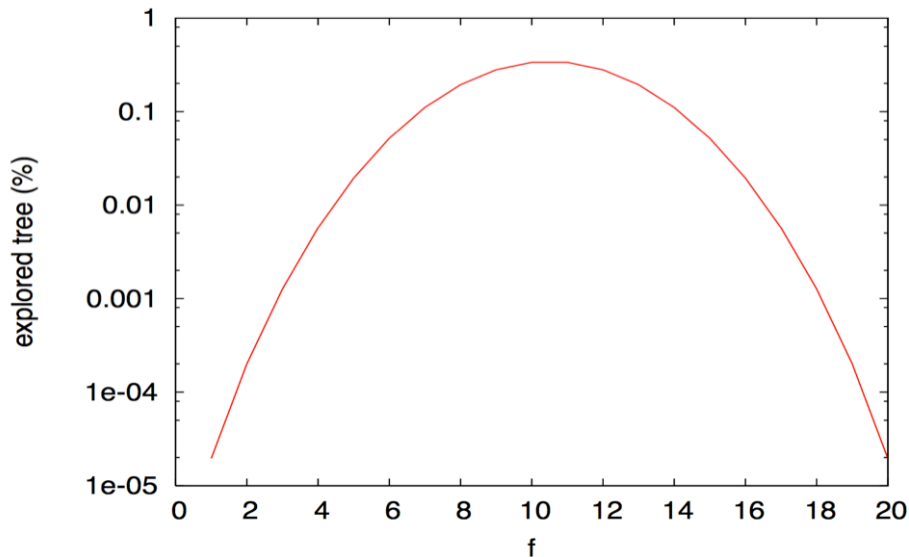
```
        processSolution( $S, i$ )
```

```
    else
```

```
        subsets( $S, n, k, i + 1, count$ )
```

```
     $count = count - S[i]$ 
```

Esempio 2: vantaggi



Esempio 2 - Sommario

Cosa abbiamo imparato?

- “Specializzando” l’algoritmo generico, possiamo ottenere una versione più efficiente
- Versione efficiente per
 - valori di k “piccoli” (vicini a 1)
 - valori di k “grandi” (vicini a n)
- Miglioramento solo parziale verso $n/2$
- E’ difficile ottenere la stessa efficienza con un algoritmo iterativo

Esempio 3

Stampa di tutte le permutazioni di un insieme A

- L'insieme dei candidati dipende dalla soluzione parziale corrente

```
permutations(SET  $A$ , int  $n$ , ITEM[]  $S$ , int  $i$ )
```

```
foreach  $c \in A$  do
```

```
     $S[i] = c$ 
```

```
     $A.remove(c)$ 
```

```
    if  $A.isEmpty()$  then
```

```
        | processSolution( $S, n$ )
```

```
    else
```

```
        | permutations( $A, n, S, i + 1$ )
```

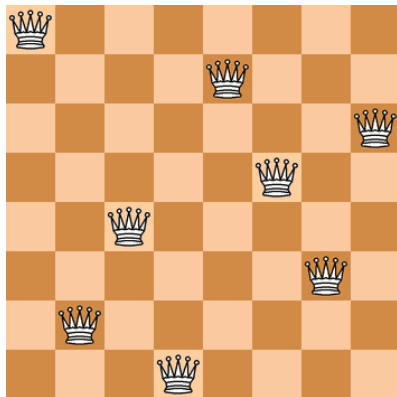
```
     $A.insert(c)$ 
```

Problema delle otto regine

Problema

Posizionare n regine in una scacchiera $n \times n$, in modo tale che nessuna regina ne "minacci" un'altra.

- Un po' di storia:
 - Introdotto da Max Bezzel (1848)
 - Gauss trovò 72 delle 92 soluzioni
- Partiamo dall'approccio più stupido, e mano a mano raffiniamo la soluzione.



Problema delle otto regine

Idea: Ci sono n^2 caselle dove piazzare una regina

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

Commenti

- Forse abbiamo un problema di rappresentazione?
- Matrice binaria molto sparsa

Problema delle otto regine

Idea: Dobbiamo piazzare n regine, ci sono n^2 caselle

$S[1 \dots n]$ coordinate in $\{1 \dots n^2\}$	$S[i]$ coordinata della regina i
controllo soluzione	se $i = n$
$\text{choices}(S, n, i)$	$\{1 \dots n^2\}$
pruning	restituisce il sottoinsieme di mosse legali
# soluzioni per $n = 8$	$(n^2)^n = 64^8 = 2^{48} \approx 2.81 \cdot 10^{14}$

Commenti

- C'è un miglioramento, ma lo spazio è ancora grande ...
- Problema: come si distingue una soluzione "1-7-..." da "7-1-..." ?

Problema delle otto regine

Idea: non mettere regine in caselle precedenti a quelle già scelte

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

Commenti

- Ottimo, abbiamo ridotto molto, ma si può ancora fare qualcosa

Problema delle otto regine

Idea: ogni riga della scacchiera deve contenere esattamente una regina

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

Commenti

- Quasi alla fine

Problema delle otto regine

Idea: anche ogni colonna deve contenere esattamente una regina

$S[1 \dots n]$ coordinate in $\{1 \dots n\}$	permutazione di $\{1 \dots n\}$
controllo soluzione	se $i = n$
$\text{choices}(S, n, i)$	$\{1 \dots n\}$
pruning	elimina le diagonali
# soluzioni per $n = 8$	$n! = 8! = 40320$

Commenti

- Soluzioni effettivamente visitate = 15720

Problema delle otto regine

Minimum-conflicts heuristic

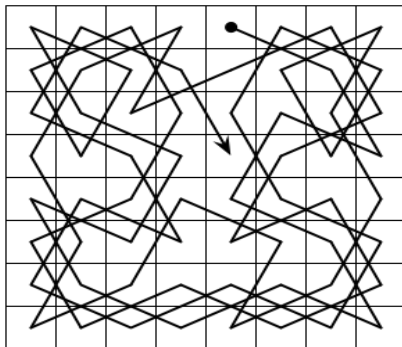
Si parte da una soluzione iniziale “ragionevolmente buona”, e si muove il pezzo con il più grande numero di conflitti nella casella della stessa colonna che genera il numero minimo di conflitti. Si ripete fino a quando non ci sono più pezzi da muovere.

- Algoritmo in tempo lineare
- Ad esempio, con $n = 1,000,000$, richiede 50 passi in media
- Questo algoritmo non garantisce che la terminazione sia sempre corretta

Giro di cavallo

Problema

Si consideri ancora una scacchiera $n \times n$; 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



Giro di cavallo

Soluzione

- Matrice $n \times n$ le cui celle contengono:
 - 0 se la cella non è mai stata visitata
 - i se la cella è stata visitata al passo i -esimo
- # soluzioni: $64! \approx 10^{89}$
- Ma: ad ogni passo ho al massimo 8 caselle possibili, quindi ne visito al più $8^{64} \approx 10^{57}$
- In realtà, grazie al pruning ne visito molto meno

Giro di cavallo

```
boolean cavallo(int[][]  $S$ , int  $i$ , int  $x$ , int  $y$ )
```

```
SET  $C = \text{mosse}(S, x, y)$ 
```

```
foreach  $c \in C$  do
```

```
     $S[x, y] = i$ 
```

```
    if  $i = 64$  then
```

```
        processSolution( $S$ )
```

```
        return true
```

```
    else if cavallo( $S, i + 1, x + m_x[c], y + m_y[c]$ ) then
```

```
        return true;
```

```
     $S[x, y] = 0$ 
```

```
return false
```

Giro di cavallo

```
SET mosse(int[][] S, int x, int y)
```

```
SET C = Set()
```

```
for int i = 1 to 8 do
```

```
     $n_x = x + m_x[i]$ 
```

```
     $n_y = y + m_y[i]$ 
```

```
    if  $1 \leq n_x \leq 8$  and  $1 \leq n_y \leq 8$  and  $S[n_x, n_y] = 0$  then
```

```
        C.insert(i)
```

```
return C
```

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

Sudoku - "Suuji wa dokushin ni kagiru"

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

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

Sudoku

```
boolean sudoku(int[][] S, int i)
```

```
int x = i mod 9
```

```
int y =  $\lfloor i/9 \rfloor$ 
```

```
SET C = Set()
```

```
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
```

Sudoku

```
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  $b_x = \lfloor x/3 \rfloor$ 
```

```
int  $b_y = \lfloor y/3 \rfloor$ 
```

```
for  $i_x = 0$  to 2 do
```

```
    for int  $i_y = 0$  to 2 do
```

% Controllo sulla sottotabella

```
        if  $S[b_x \cdot 3 + i_x, b_y \cdot 3 + i_y] = c$  then
```

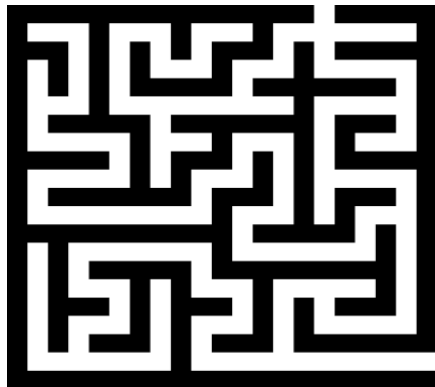
```
            return false
```

```
return true
```

Generazione labirinti

Problemi

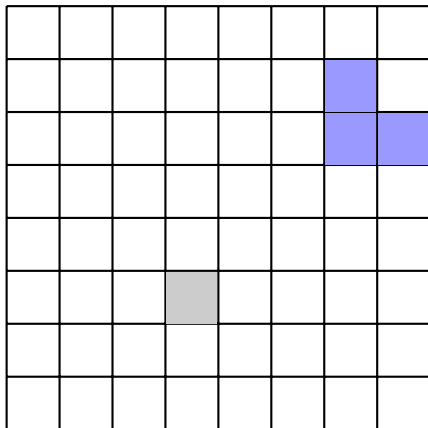
- Come generare un labirinto in una griglia $n \times n$?
- Come uscire da un labirinto?



Un ultimo puzzle

Problema

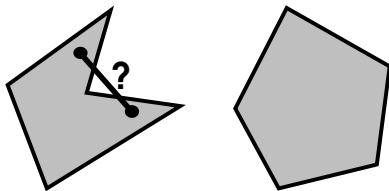
- Si consideri una scacchiera $n \times n$, con $n = 2^k$
- Qualsiasi scacchiera di questo tipo con una cella rimossa può essere ricoperta da triomini a forma di L
- Trovare un algoritmo che trovi una possibile ricopertura della scacchiera



Inviluppo convesso

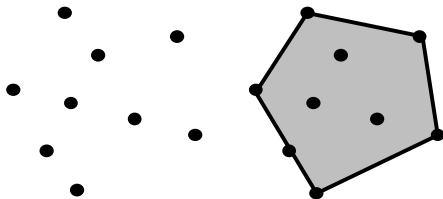
Poligono convesso

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.



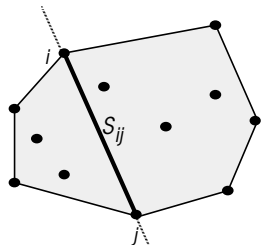
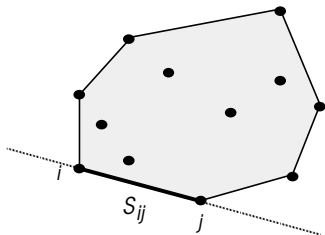
Inviluppo convesso

Dati n punti p_1, \dots, p_n nel piano, con $n \geq 3$, l'**inviluppo convesso** (**convex hull**) è il più piccolo poligono convesso che li contiene tutti



Algoritmo inefficiente – $O(n^3)$

- 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'involuppo convesso



Stessa parte

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  $dx = p_2.x - p_1.x$ 
```

```
float  $dy = p_2.y - p_1.y$ 
```

```
float  $dx_1 = p.x - p_1.x$ 
```

```
float  $dy_1 = p.y - p_1.y$ 
```

```
float  $dx_2 = q.x - p_2.x$ 
```

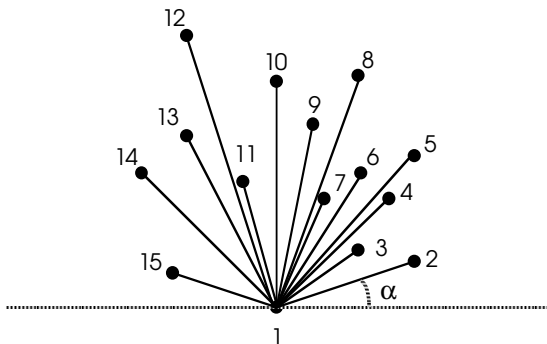
```
float  $dy_2 = q.y - p_2.y$ 
```

```
return  $((dx \cdot dy_1 - dy \cdot dx_1) \cdot (dx \cdot dy_2 - dy \cdot dx_2) \geq 0)$ 
```

Algoritmo di Graham

Fase 1

- 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



Algoritmo di Graham

```
STACK graham(POINT[]  $p$ , int  $n$ )
```

```
int  $min = 1$ 
```

```
for  $i = 2$  to  $n$  do
```

```
    if  $p[i].y < p[min].y$  then  
         $min = i$ 
```

```
 $p[1] \leftrightarrow p[min]$ 
```

```
{ riordina  $p[2, \dots, n]$  in base all'angolo formato rispetto all'asse  
  orizzontale quando sono connessi con  $p[1]$  }
```

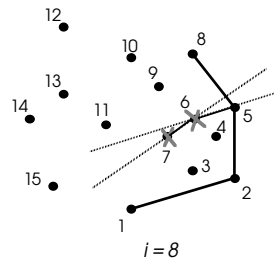
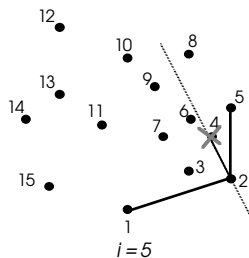
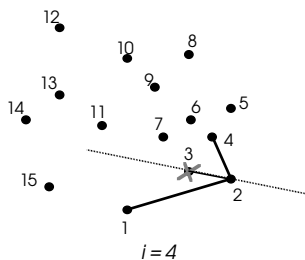
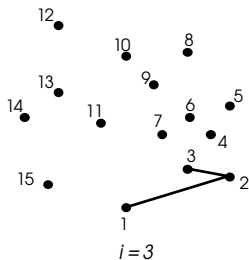
```
{ elimina eventuali punti "allineati" tranne i più lontani da  $p_1$ ,  
  aggiornando  $n$  }
```

Algoritmo di Graham

Fase 2

- Inserisci p_1, p_2 nell'inviluppo corrente
- Per tutti i punti $p_i = 3, \dots, n$:
 - siano p_h e p_j , con $h < j = i - 1$, gli ultimi due vertici dell'inviluppo corrente
 - scandisci “a ritroso” i punti nell'inviluppo “corrente” ed elimina p_j se $\text{stessaparte}(p_j, p_h, p_1, p_i) = \mathbf{false}$;
 - termina tale scansione se p_j non deve essere eliminato;
 - aggiungi p_i all'inviluppo “corrente”

Algoritmo di Graham



Algoritmo di Graham

```
STACK graham(POINT[]  $p$ , int  $n$ ) (continua)
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  $S$ 
```

Algoritmo di Graham

Complessità

- $O(n \log n)$, dominato dall'ordinamento