Problemi difficili e ricerca esaustiva intelligente

25 maggio 2023

Dasgupta, Papadimitriou, Vazirani: Algorithms

Cap. 9 (inizio)

Introduzione

Finora ci siamo interessati alla progettazione di algoritmi efficienti

Efficienza = polinomiale

Tecniche: ricerca esaustiva, divide et impera, greedy, programmazione dinamica

Però….. esistono dei problemi

- Che non sappiamo risolvere efficientemente
- Né sappiamo dimostrare che non esistono soluzioni efficienti

Questione P e NP

P e NP

Informalmente:

P è la classe dei problemi risolvibili in tempo polinomiale

Es.: trovare cammino minimo fra due vertici in un grafo

NP è la classe dei problemi verificabili in tempo polinomiale

Es.: fattorizzare un intero

Dato intero $n = a \cdot b$, trovare a e b.

Se a e b sono primi molto grandi è difficile risolverlo, ma

se qualcuno mi suggerisce a e b, io posso facilmente verificare che $n = a \cdot b$.

P = NP?

Chiaramente $P \subseteq NP$.

Ma P = NP? Oppure $P \neq NP$?

La questione non è meramente «teorica»!

Il problema della fattorizzazione (vedi prima) è usato nel sistema crittografico RSA che è alla base del commercio elettronico!

Per simili motivazioni, per chi risolvesse il problema P=NP? c'è in palio…

Ricerca esaustiva

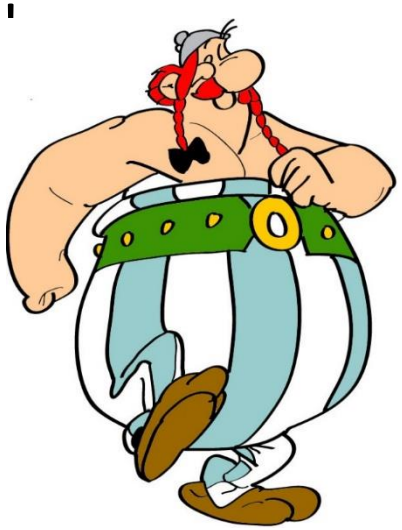
Supponiamo di dovere fornire un algoritmo per risolvere un problema.

Se non riusciamo a trovare un algoritmo efficiente con nessuna tecnica studiata…… magari…..

una soluzione efficiente non esiste'

Che fare?

Una possibilità: la forza bruta!



Ricerca esaustiva?

Ricerca esaustiva esamina tutto lo spazio delle soluzioni.

Ma in quanto tempo?

Per la fattorizzazione di n, se a e b avessero 2048 bit, non basterebbe una vita!

Tuttavia la tecnica può essere utilizzata per istanze piccole.

Vedremo adesso degli accorgimenti per migliorare l'efficienza di algoritmi di ricerca esaustiva

Ricerca esaustiva … intelligente

Backtracking: a volte si può rifiutare una possibile soluzione esaminando solo una sua piccola parte

Branch and bound: in un problema di ottimizzazione, a volte si può scartare una possibile soluzione senza averla esaminata completamente, perché il suo valore non può essere ottimale

Backtracking

[DPV] par. 9.1.1

Formule booleane e soddisfattibilità

Una formula (espressione) booleana ϕ è soddisfattibile se esiste un assegnamento delle sue variabili che la rende Vera/ 1

Es.: $\phi = (a + \neg b) \cdot (\neg a)$ è soddisfattibile perchè $\phi = 1$ assegnando a=0, b= 0

$\phi = (a) \cdot (\neg a)$ non è soddisfattibile

Stabilire se una formula booleana è soddisfattibile è «difficile» (NP-hard).

Ogni formula booleana può essere espressa in Forma Normale Congiuntiva, in breve CNF (espressione POS).

Il problema SAT

SAT (Satisfiability)
INPUT: una formula booleana
OUTPUT: SI, se la formula è soddisfattibile; NO, altrimenti

CNF - SAT

INPUT: una formula booleana in CNF

OUTPUT: SI, se la formula è soddisfattibile; NO, altrimenti

SAT e CNF-SAT sono due problemi decisionali (output binario) «difficili»

Risolvere SAT

Esempio: $\phi = (a + b + c + \neg d) \cdot (a + b) \cdot (a + \neg b) \cdot (\neg a + c) \cdot (\neg a + \neg c)$

Dato uno specifico assegnamento, è facile verificare se esso rende ϕ Vera/1 o Falsa/0.

Per esempio: per a=1, b=0, c=1, d=0 si ha

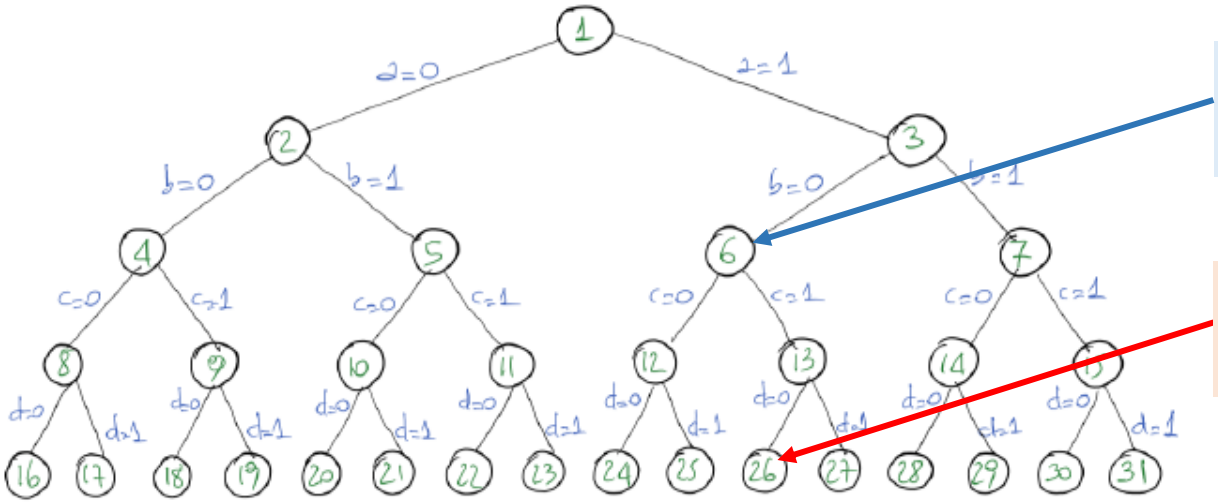
$\phi = (1 + 0 + 1 + \neg 0) \cdot (1+0) \cdot (1 + \neg 0) \cdot (\neg 1 + 1) \cdot (\neg 1 + \neg 1) =$

$= 1 \cdot 1 \cdot 1 \cdot 1 \cdot 0 = 0$

Per vedere se ϕ è soddisfattibile posso valutare
tutti i 16 assegnamenti possibili

Ricerca esaustiva e albero delle decisioni

Potremmo organizzare tale processo tramite un albero delle decisioni



Assegnamento parziale
a=1, b=0

Assegnamento
a=c=1, b=d=0

L'albero avrà 2^n foglie, se n è il numero delle variabili e 2^{n+1} -1 nodi in totale.
Dovremmo valutare un numero esponenziale di possibili assegnamenti con costo proporzionale al numero totale dei nodi.

Backtracking

Cominciamo dalla radice costruendo l'albero in maniera incrementale.

A volte potremo scartare alcuni assegnamenti, fermandoci ad un assegnamento parziale, dal quale è inutile proseguire.

Allora backtrack: torniamo indietro per provare altre strade.

Backtracking e SAT

Esempio:

$\phi(a,b,c,d) = (a + b + c + \neg d) \cdot (a + b) \cdot (a + \neg b) \cdot (\neg a + c) \cdot (\neg a + \neg c)$

Partiamo ponendo a=0

$\phi(0, b,c,d) = (b + c + \neg d) \cdot (b) \cdot (\neg b) \cdot (1) \cdot (1) =$

$= (b + c + \neg d) \cdot (b) \cdot (\neg b)$

Sottoproblema: $\phi(0, b,c,d) = (b + c + \neg d) \cdot (b) \cdot (\neg b)$ è soddisfattibile?

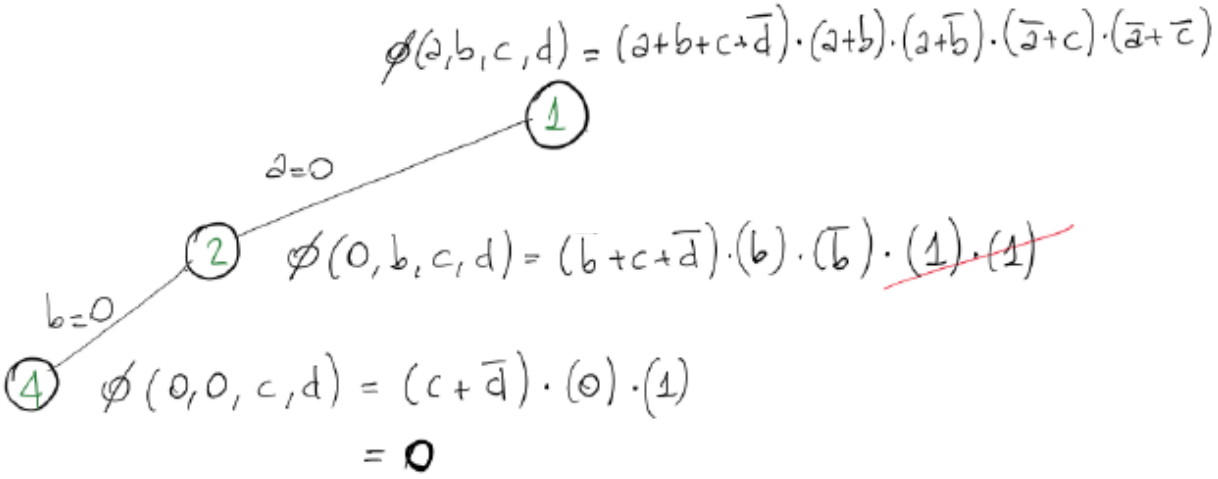
Poniamo b=0.

$\phi(0, 0, c,d) = (c + \neg d) \cdot (0) \cdot (1) = 0$

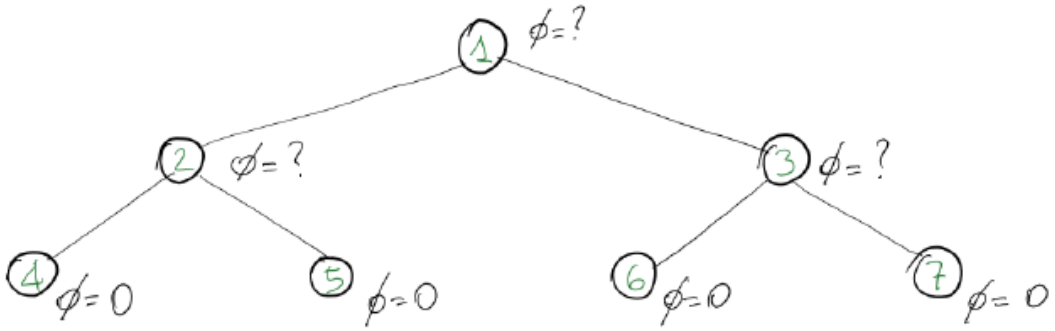
indipendentemente dagli assegnamenti per c e d: Backtrack!

Valutazione parziale

Nell'albero delle decisioni:

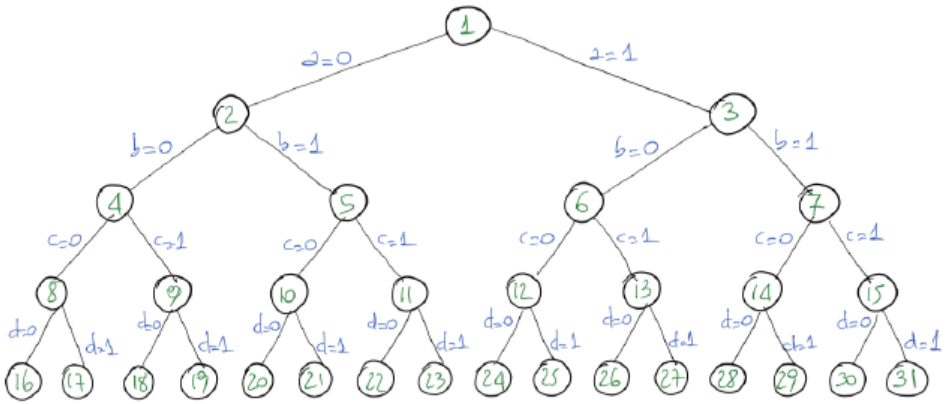


Alla fine….



Possiamo affermare che la formula non è soddisfattibile.

Avendo risparmiato un bel po’ di calcoli



Backtracking

Per ogni sottoproblema considerato esegue un test con 3 possibili risultati:

Failure: il sottoproblema non ha soluzioni
(es.: $\phi = 0$ in un nodo)

Success: trovo una soluzione al problema di partenza
(es.: $\phi = 1$ in una foglia)

Uncertainty: bisogna proseguire
(es.: $\phi = ?$)

Schema backtrack

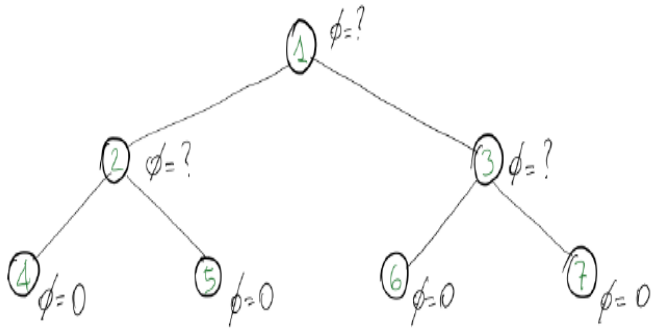
```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
Repeat while S is nonempty:
  choose a subproblem P ∈ S and remove it from S
  expand it into smaller subproblems P_1, P_2, ..., P_k
  For each P_i:
    If test(P_i) succeeds:  halt and announce this solution
    If test(P_i) fails:  discard P_i
    Otherwise:  add P_i to S
Announce that there is no solution
```

Nonostante la complessità di tempo nel caso peggiore resti esponenziale, il backtracking può essere molto efficiente nella pratica.

Schema backtrack: esempio

Esempio: $\phi(a,b,c,d) = (a + b + c + \neg d) \cdot (a + b) \cdot (a + \neg b) \cdot (\neg a + c) \cdot (\neg a + \neg c)$

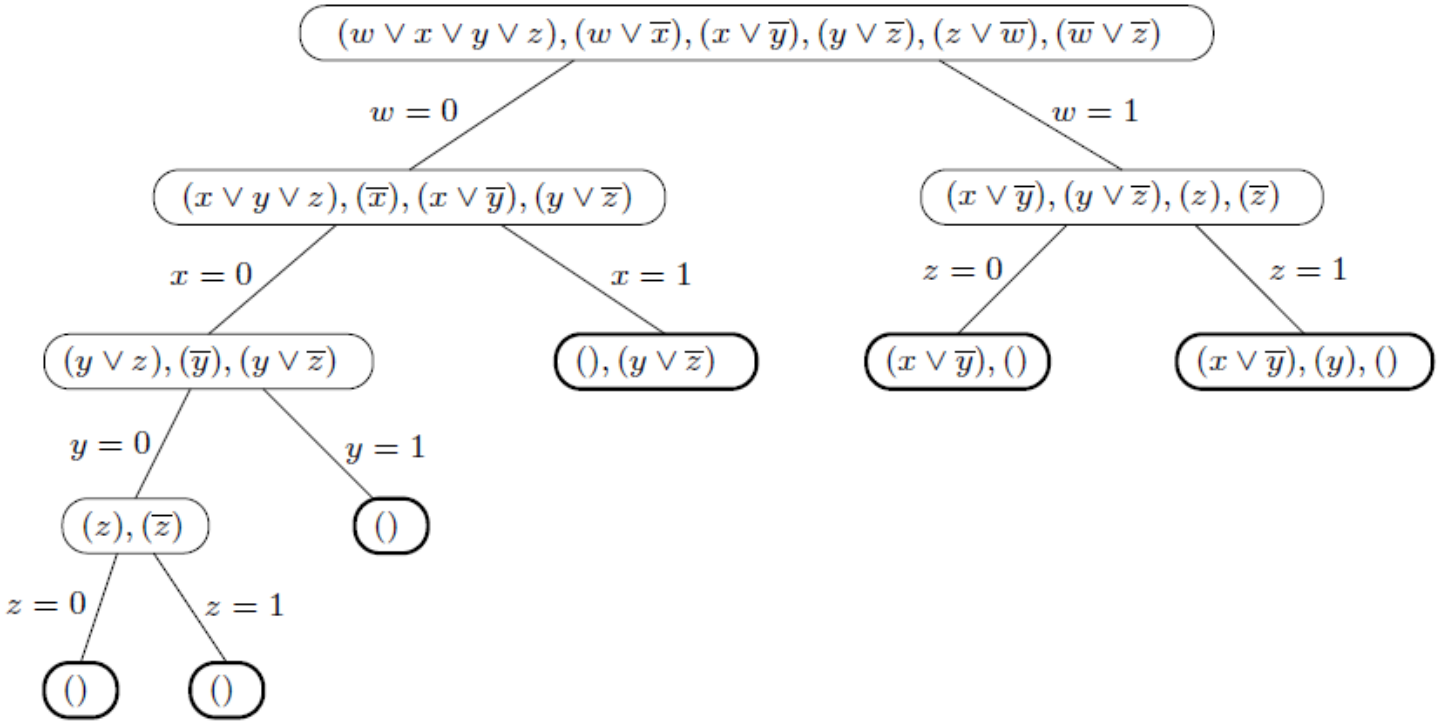
```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
Repeat while S is nonempty:
  choose a subproblem P ∈ S and remove it from S
  expand it into smaller subproblems P_1, P_2, ..., P_k
  For each P_i:
    If test(P_i) succeeds:  halt and announce this solution
    If test(P_i) fails:  discard P_i
    Otherwise:  add P_i to S
Announce that there is no solution
```



- P_0: (satisfiability) for $\phi(a,b,c,d)$; S = $\{P_0\}$
- Choose and remove P_0, S = {}, expand P_0 in P_1 and P_2 , where P_1 is for $\phi(0, b,c,d)$ and P_2 is for $\phi(1, b,c,d)$.
- Test(P_1): uncertainty S = $\{P_1\}$, Test(P_2): uncertainty S = $\{P_1 , P_2\}$, S is non-empty.
- Choose and remove P_1, S = $\{P_2\}$, expand P_1 in P_3 and P_4 , where P_3 is for $\phi(0, 0,c,d)$ and P_4 is for $\phi(0, 1,c,d)$.
- Test(P_3): fails, $\phi(0 ,0,c,d)=0$, discard P_3; Test(P_4): fails, $\phi(0 ,1,c,d)=0$, discard P_4; S = $\{P_2\}$, S is non-empty.
- Choose and remove P_2, S = {}, expand P_2 in P_5 and P_6 , where P_5 is for $\phi(1 ,0,c,d)$ and P_6 is for $\phi(1, 1,c,d)$.
- Test(P_5): fails, $\phi(1 ,0,c,d)=0$, discard P_5; Test(P_6): fails, $\phi(1 ,1,c,d)=0$, discard P_6; S = {} is empty.
- Announce NO SOLUTION.

Esempio del libro

Figure 9.1 Backtracking reveals that ϕ is not satisfiable.



Qui () indica una «clausola vuota» cioè insoddisfattibilità

Branch and bound

[DPV] par. 9.1.2

Branch and bound

La stessa idea del backtracking può essere estesa a problemi di ottimizzazione.

Ogni soluzione ha un valore e cerchiamo una soluzione ottimale (minimale o massimale). Nel seguito considereremo problemi di minimizzazione.

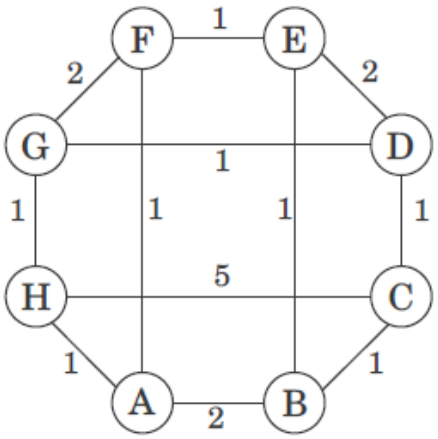
Anche stavolta considereremo soluzioni parziali a sottoproblemi.

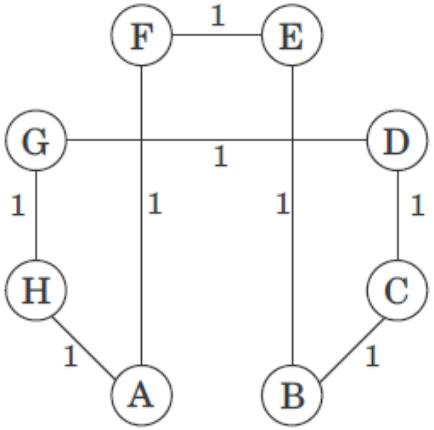
Per escludere una soluzione parziale dobbiamo essere certi che il costo di tutte le soluzioni sviluppate a partire da essa avranno un costo maggiore di quello di un'altra soluzione già calcolata. Cioè sia superiore ad un certo limite / bound.

Il commesso viaggiatore

Il problema del commesso viaggiatore, in breve TSP (Traveling Salesman Problem) è il seguente.

TSP
INPUT: Un grafo G=(V,E) completo con dei costi sugli archi
OUTPUT: Un ciclo di costo minimo che passa per ogni vertice una ed una sola volta





Si assume che gli archi non disegnati abbiano costi molto alti (per es. 100, o infinito)

Risolvere TSP

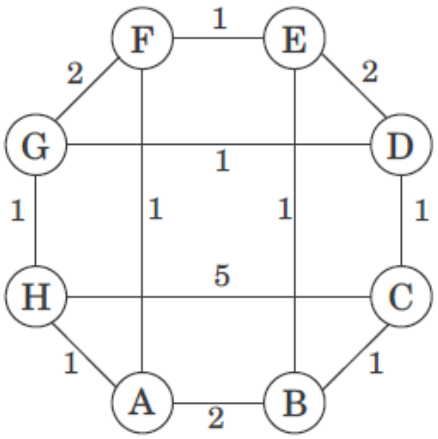
La ricerca esaustiva valuta ogni possibile «giro» del grafo.

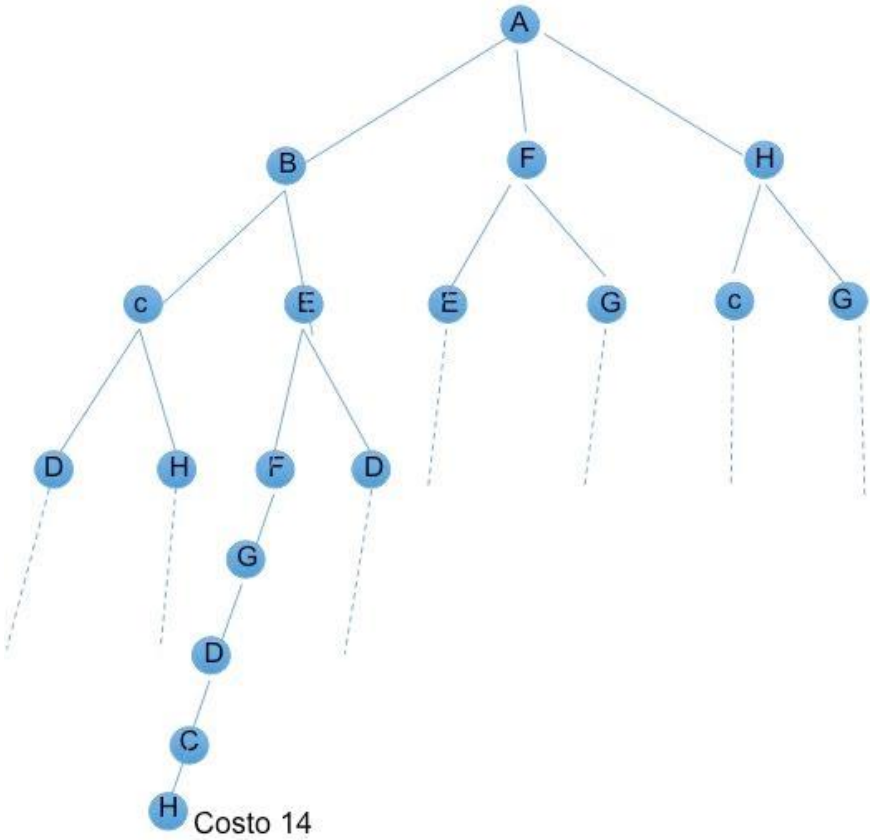
Partendo da un qualsiasi nodo, costruiamo incrementalmente l'albero delle decisioni.

Ogni foglia rappresenta un giro.

Ogni nodo interno un cammino parziale dal nodo di partenza.

Albero delle decisioni: Branch





Parto da A.

Il numero di foglie dell'albero è (n-1)!, dove n è il numero dei vertici del grafo.

Il numero di tutti i nodi ancora di più!!

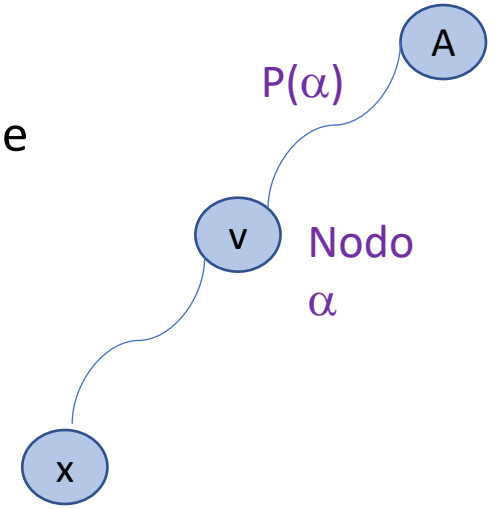
Limite inferiore: Bound

Per escludere una soluzione/cammino parziale bisogna essere sicuri che il costo di ogni suo completamento superi un certo limite inferiore.

Per ogni nodo interno α dell'albero indichiamo con

- $P(\alpha)$ il cammino rappresentato da α che va da A al vertice $v=v(\alpha)$ nel grafo
- $S(\alpha)$ l'insieme dei nodi attraversati da $P(\alpha)$ (inclusi A e v).

Vogliamo limitare il costo di qualsiasi giro completo da A in A che inizia con $P(\alpha)$ e prosegue con un certo cammino $Q(\alpha)$ che non visita più i nodi di $S(\alpha)$.
$Q(\alpha)$ è quindi un cammino da v a un certo nodo x, seguito dall'arco (x,A), che attraversa nodi di $V\backslash S(\alpha)$.



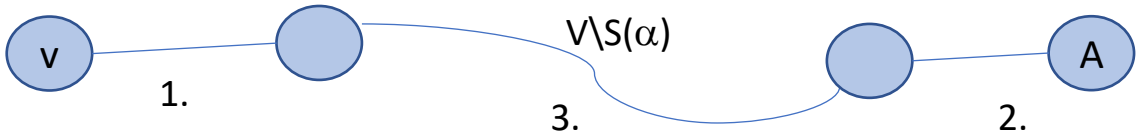
Se tale limite è ≥ del costo di un giro già noto, possiamo evitare di esplorare il sottoalbero di α.

Limite inferiore per TSP

Ogni nodo interno α dell'albero rappresenta un cammino $P(\alpha)$ da A ad un certo vertice $v=v(\alpha)$ del grafo che passa per l'insieme di nodi $S(\alpha)$ (inclusi A e v).

Vogliamo limitare il costo di qualsiasi giro completo che inizia con $P(\alpha)$ e prosegue con $Q(\alpha)$.

$Q(\alpha)$:



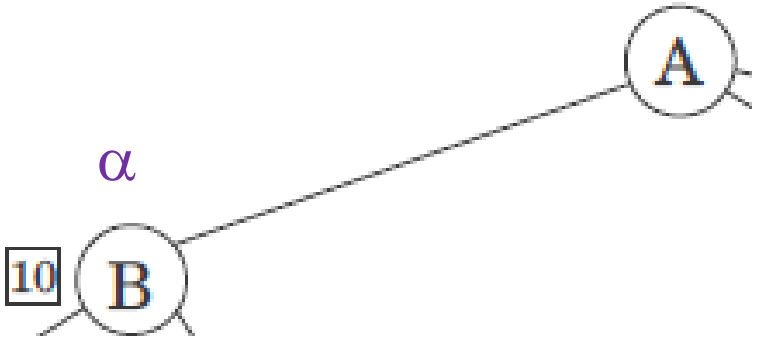
Il costo di $Q(\alpha)$ è $\geq$ della somma di:

1. costo minimo di un arco da v ad un vertice in $V\backslash S(\alpha)$
2. costo minimo di un arco da un vertice in $V\backslash S(\alpha)$ ad A
3. Il costo di un MST di $V\backslash S(\alpha)$ (un cammino semplice che tocca tutti i nodi di $V\backslash S(\alpha)$ è uno ST; il suo costo è $\geq$ costo MST)

Esempio

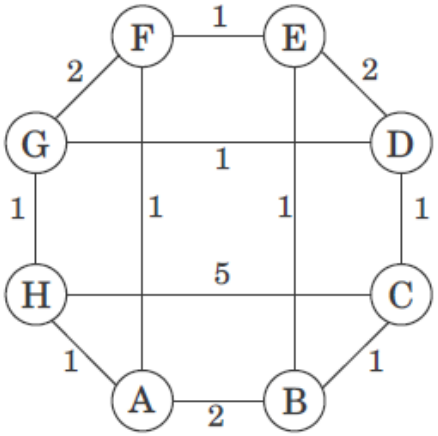
α nodo dell'albero

$v=v(\alpha)=B$ vertice del grafo



$P(\alpha)$ = (A, B) con costo $P(\alpha)=2$

$S(\alpha)$ = {A, B}

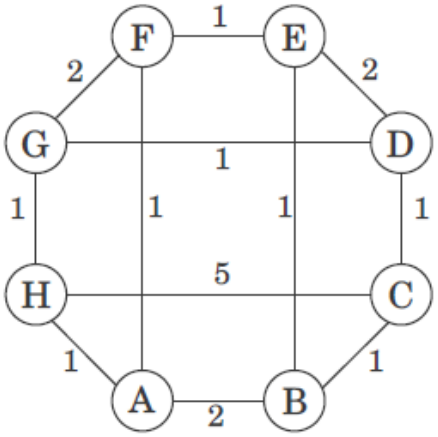


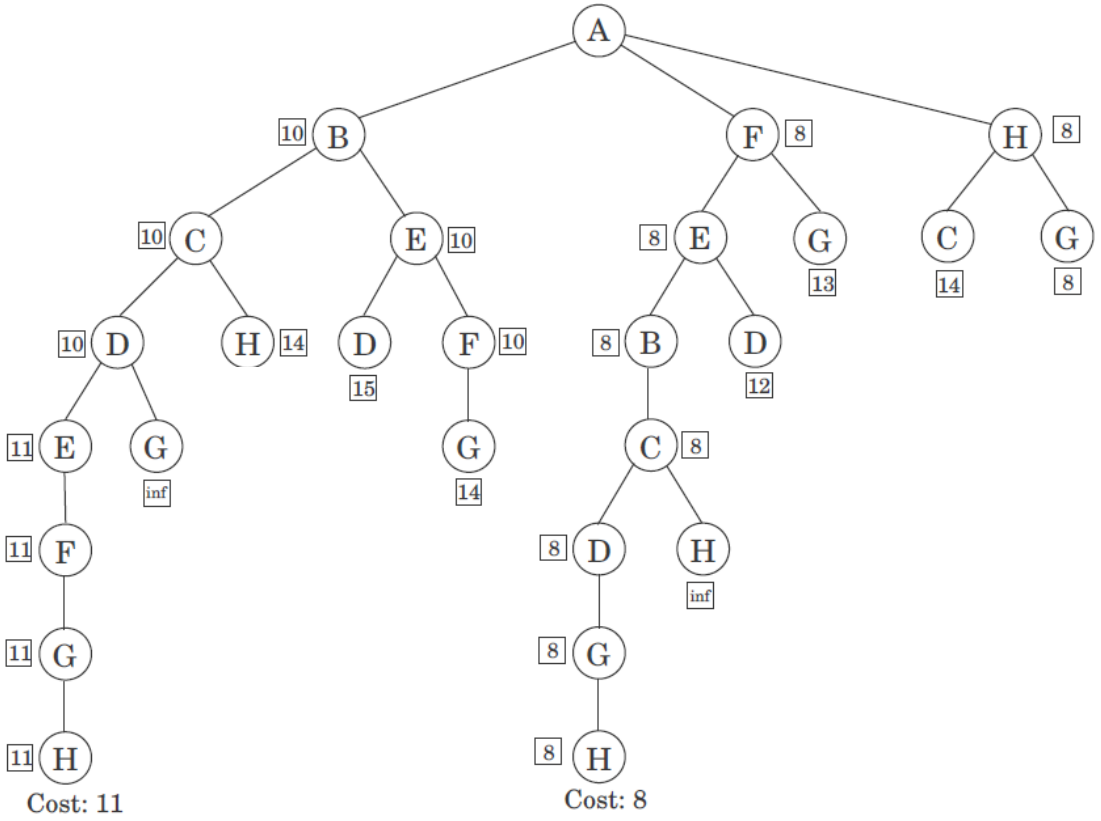
Il costo di $Q(\alpha)$ è $\geq 8 = 1+1+6$

1. costo arco BC = 1
2. costo arco AH o AF = 1
3. costo MST relativo a {C, D, E, F, G, H} = 6

Il costo di un giro completo che inizia con $P(\alpha)$ è $\geq 2+8=10$

Branch and bound: esempio

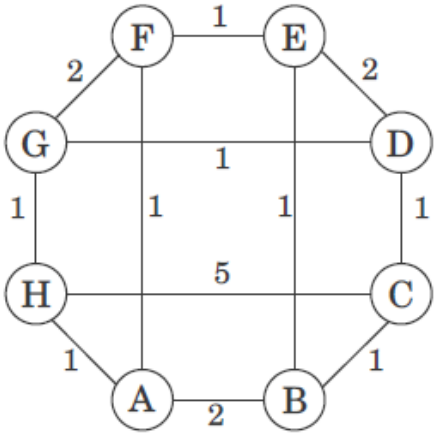




I numeri nei quadrati sono il limite ottenuto sul giro che inizia in quel modo.

Alla fine considereremo soltanto 28 nodi dell'albero invece di un numero >7!=5.040

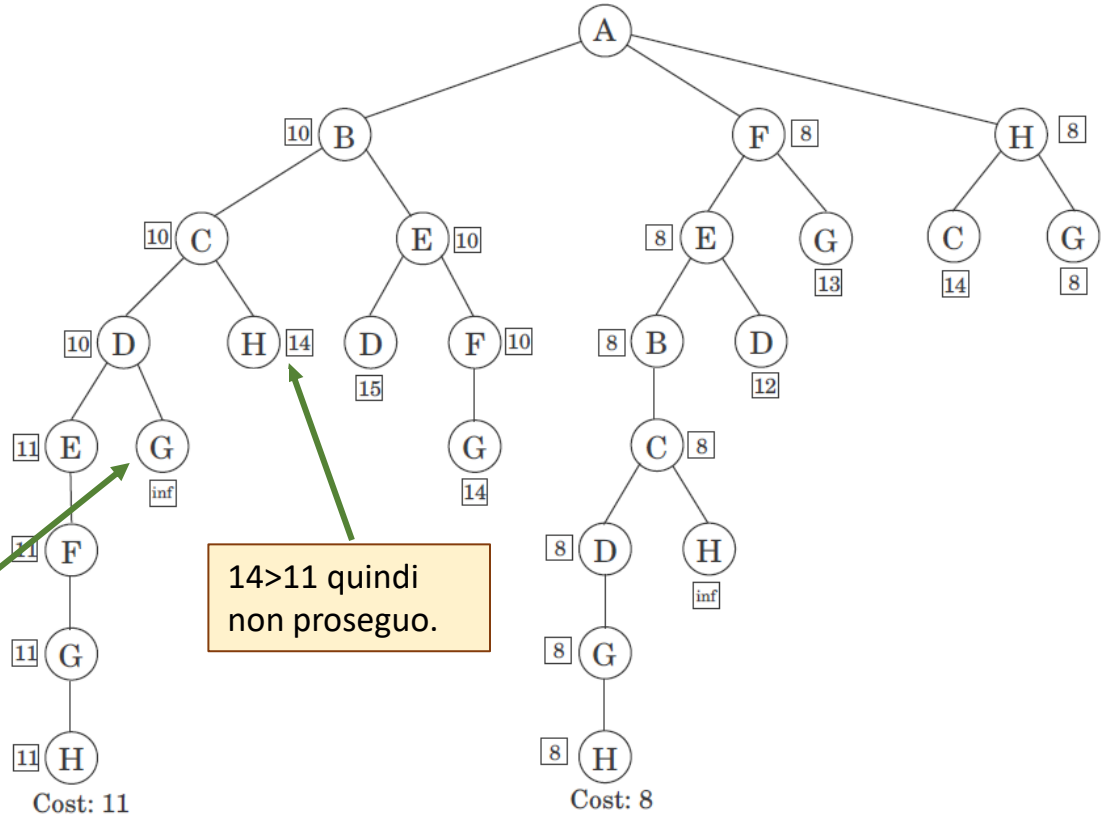
Branch and bound: esempio



Limite per G = inf(inito)

V\S(α)={E,F,H}

1. Costo GH=1
2. Costo AF=1
3. Costo MST per {E,F,H} = infinito (considerando infinito il costo degli archi non disegnati)



Schema algoritmo branch and bound (minimizzazione)

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
bestsofar = ∞
Repeat while S is nonempty:
  choose a subproblem (partial solution) P ∈ S and remove it from S
  expand it into smaller subproblems P_1, P_2, ..., P_k
  For each P_i:
    If P_i is a complete solution:  update bestsofar
    else if lowerbound(P_i) < bestsofar:  add P_i to S
return bestsofar
```

Fine programma!

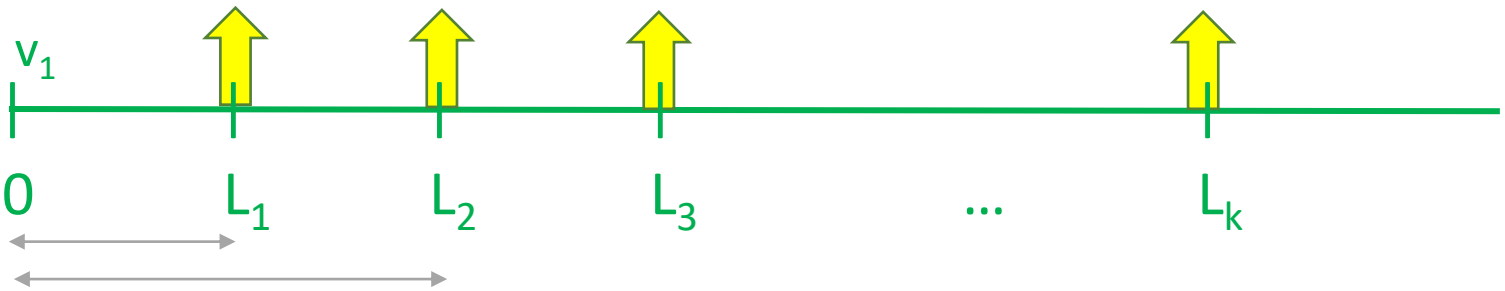
Lampioni (Greedy)

Una società elettrica vuole piazzare dei lampioni lungo un viale rettilineo dove sono situate delle villette, in modo da illuminare tutte le villette. Ogni lampione riesce ad illuminare tutta la zona compresa nel raggio di 500 metri. Percorrendo il viale si incontrano, in ordine, le villette $v_1, v_2, \dots, v_n$. Per ogni i =1, … , n−1, sia $d_{i,i+1}$ la distanza della villetta v_i da v_{i+1}. Immaginiamo che le villette siano «puntiformi».

a) **Descrivere** ed **analizzare** un algoritmo che presi in input i valori $d_{i,i+1}$ restituisca il numero minimo di lampioni necessari per illuminare tutte le villette.
b) **Argomentare** la correttezza dell'algoritmo.

Lampioni: correttezza

Sia $G = \{L_1, L_2, \dots, L_k\}$ la soluzione greedy, dove per ogni i=1, 2, … , k, L_i è la distanza dell'i-esimo lampione dalla prima villetta v_1.



Supponiamo per assurdo che G non sia ottimale e sia

$O = \{O_1, O_2, \dots, O_m\}$

una soluzione ottimale (dove per ogni i=1, 2, … , m, O_i è la distanza dell'i-esimo lampione dalla prima villetta v_1) scelta in modo tale che:

$O_1 = L_1, \dots, O_r = L_r$, ed r sia massimo.

Se r= k = m, allora O = G, ma G non è ottimale per ipotesi. Quindi $O_{r+1} \neq L_{r+1}$.
Sia v_h la prima villetta non illuminata da $O_1 = L_1, \dots, O_r = L_r$.
Osserviamo che L_{r+1}= dist(v_1, v_h) +500.
Quindi $O_{r+1} \leq L_{r+1}$, altrimenti v_h al buio, ma $O_{r+1} \neq L_{r+1}$ quindi $O_{r+1} < L_{r+1}$.
Ma allora, O' ottenuta da O, sostituendo O_{r+1} con L_{r+1} è soluzione ottimale contro massimalità di r.

Si consideri il problema dello *scheduling* che minimizza il ritardo. Se in uno *scheduling* esiste un'inversione fra due attività programmate l'una subito dopo l'altra, nello *scheduling* ottenuto invertendo tali attività

A. Il ritardo è strettamente minore del precedente e ho un'inversione in meno
B. Il ritardo è minore o uguale al precedente e ho un'inversione in meno
C. Il ritardo è minore o uguale al precedente e il numero di inversioni è lo stesso
D. Nessuna delle risposte precedenti

Sia H = [1, 6, 3, 12, 7, 9]. L'array H può rappresentare un *heap* binario (basato sulla priorità minima)?

A. No, perché gli elementi dovrebbero essere in ordine crescente

B. No, perché il massimo dovrebbe occupare l'ultima posizione

C. Sì

D. Nessuna delle risposte precedenti