Strutture dati per Electronic Design Automation al livello logico

M. Favalli

Engineering Department in Ferrara

Sommario

Motivazioni

Fornire alcuni strumenti rilevanti per la comprensione e la realizzazione di algoritmi di EDA

Alcuni di questi strumenti sono utilizzati anche in altri ambiti

Sommario

- Cenni sulla complessitá computazionale degli algoritmi
- Strumenti per la rappresentazione e la manipolazione di funzioni Booleane
  - forme normali
  - diagrammi di decisione binaria (BDD)
  - grafi di tipo and-or-invert (AIG)
  - reti di equazioni Booleane

Complessitá computazionale degli algoritmi

- Efficienza di un algoritmo (tempo di calcolo e spazio di memoria)
- La legge di Moore rende particolarmente rilevante l'efficienza degli algoritmi utilizzati nella sintesi dei sistemi digitali
- Vedremo alcuni concetti base

Introduzione

- Algoritmo
- Dimensione del problema n
  - numero di elementi di un vettore da riordinare
  - numero di porte logiche in un circuito da simulare
- Dipendenza della complessitá di un algoritmo dall'insieme di dati in ingresso
  - complessitá media
  - complessitá di caso peggiore

Esempio

- Ricerca di un elemento in un vettore ordinato di n elementi
- Ricerca lineare
  - complessitá di caso peggiore proporzionale a n
  - complessitá media proporzionale a n
- Ricerca binaria
  - complessitá di caso peggiore proporzionale a $\log_2 n$
  - complessitá media prorzionale a $\log_2 n$

Notazione asintotica

- Interesse per il rateo di crescita dei tempi di calcolo
- La legge di Moore porta alla crescita delle dimensioni dei dati
- Interessa maggiormente il termine dominante dell'espressione del tempo di calcolo
- Tre possibili notazioni asintotiche O, Ω, Θ
- Considereremo solo quella O

Notazione O

- **Limite superiore asintotico**
- Data una funzione $g(n)$, $O(g(n))$ denota un insieme di funzioni

$$O(g(n)) = \{f(n) : \exists\, c > 0,\ n_0 > 0 :$$
$$\forall n > n_0,\ 0 \leq f(n) \leq cg(n)\ \}$$

- Esempio: $f(n) = 4n^2 + 2n + 9 \in O(n^2)$ perché per $c = 5$ e $n_0 = 5$ $f(n) \leq cg(n)$

Esempi di complessitá computazionale

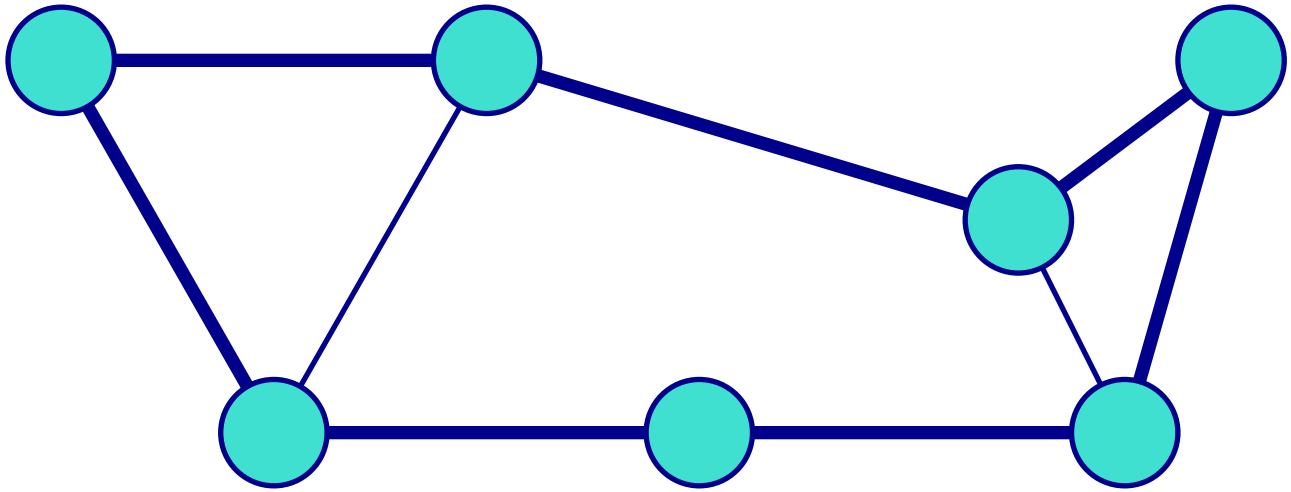
O	denominazione
1	costante
$log\ n$	logaritmico
n	lineare
n^2	quadratico
n^3	cubico
2^n	esponenziale
$n!$	fattoriale

Classi di complessitá

- Classificazione dei problemi
- **La complessitá di un problema é equivalente alla complessitá computazionale del miglior algoritmo in grado di risolverlo**
- Dal punto di vista dell'EDA ci interessiamo in particolare a problemi di decisione e ottimizzazione

Problema di decisione

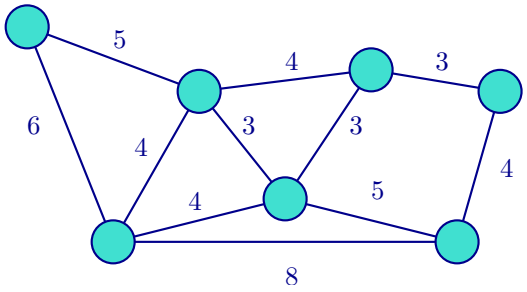
Esistenza di un percorso che connetta tutti i nodi di un grafo senza passare due volte dallo stesso nodo (ciclo hamiltoniano)



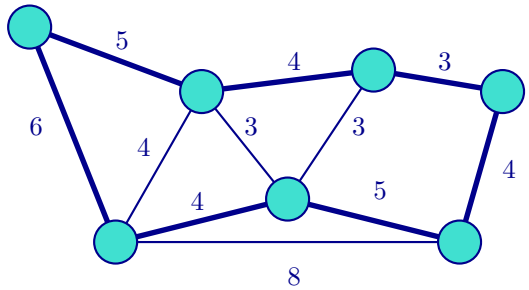
Problema di ottimizzazione

Dato un grafo in cui a ciascun arco é associata una distanza fra i due nodi collegati si vuole trovare un percorso ciclico di lunghezza minima (problema del commesso viaggiatore)

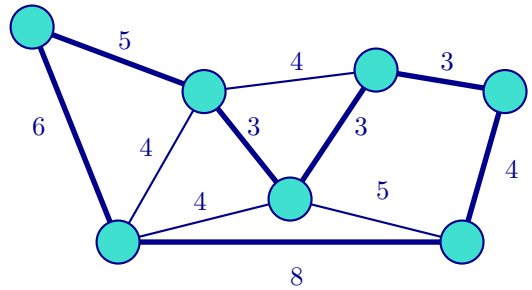
Grafo



Soluzione ottima ($d = 31$)



Soluzione non ottima ($d = 32$)



Relazione fra problemi di decisione e di ottimizzazione

- Il problema di decisione corrispondente al problema del commesso viaggiatore consiste nel determinare se esiste un percorso di lunghezza $\leq k$
- Un problema di ottimizzazione é in generale piú complesso del corrispondente problema di decisione
- Se ho risolto il problema di ottimizzazione ho risolto i problemi di decisione per ogni k

Classe di complessitá P

- Un problema si dice di classe P (polinomiale) se puó essere risolto con un algoritmo di complessitá polinomiale ($O(n^k)$)
- Static Timing Analysis
- Simulazione logica (per reti combinatorie) ?
  - 0-delay é sicuramente polinomiale nel numero di gate e nel numero di vettori logici applicati
  - event-driven ? hazard e riconvergenza di path

Classe di complessitá *NP*

- Si tratta di problemi di decisione
- Un problema si dice *NP* (polinomiale non determininistico) se puó essere risolto in un tempo polinomiale da una macchina non deterministica
- Calcolatore non deterministico: a ogni passo di decisione sceglie la migliore fra le possibili alternative
- **Proprietá**: la correttezza del risultato puó essere verificata in un tempo polinomiale da una calcolatore deterministico
- $P \subseteq NP$, ma non si sa se $P = NP$

Esempi

- Esempi di problemi con classe di complessitá *NP*:
  - ricerca di un ciclo Hamiltoniano
  - problema di decisione associato al problema del commesso viaggiatore
  - ne vedremo diversi altri nell'ambito dell'EDA

Classe di complessitá NPC

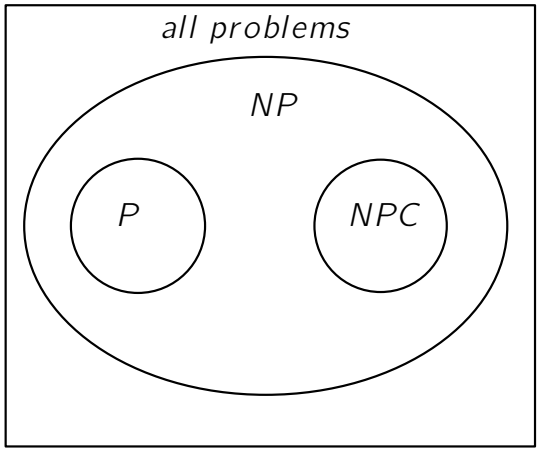
- É la classe dei problemi piú difficili da risolvere all'interno della classe *NP* (*NP*-completi)
- Sono ottenuti tramite una trasformazione di costo polinomiale dai problemi *NP*
- Ad esempio il problema del ciclo hamiltoniano puó essere trasformato in un tempo lineare in quello del commesso viaggiatore (decisione con $k = n$) attribuendo una distanza 1 a tutti gli archi del grafo
- Tutti i problemi di tipo *NPC* sono riducibili polinomialmente fra loro
- Quindi basta avere un singolo problema che si sa essere *NPC*: il problema della Boolean Satisfiability

Boolean Statisfiability (SAT)

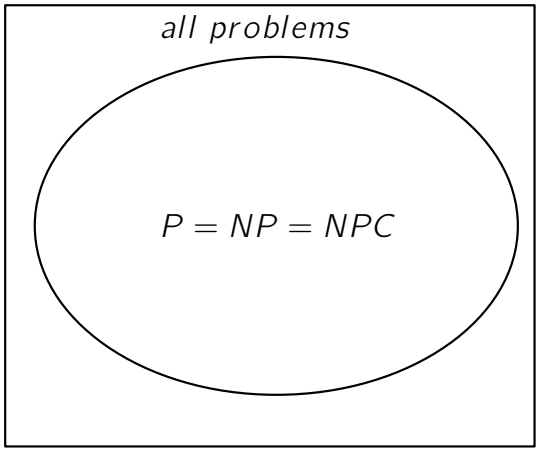
- Il problema della soddisfacibilitá Booleana consiste nel determinare un assegnamento a un gruppo di variabili Booleane che renda vera un espressione PS
- In tale caso l'espressione si dice soddisfacibile
- $\phi = (a + b)(a' + b' + c)(b' + d)$ é soddisfatta da $a = 1,\ b = 0$
- Si tratta chiaramente di un problema di decisione che é stato provato essere di tipo *NPC*

Problema

- Esistono problemi di tipo *NPC* risolubili in tempo polinomiale?
- Non si sa, ma se ne esistesse uno, poiché tutti i problemi *NP* sono riducibili in un tempo polinomiale a *NPC*, allora si avrebbe $P = NP = NPC$



Ipotesi: $NP \neq P$



Ipotesi: $NP = P$

Problemi NP-hard

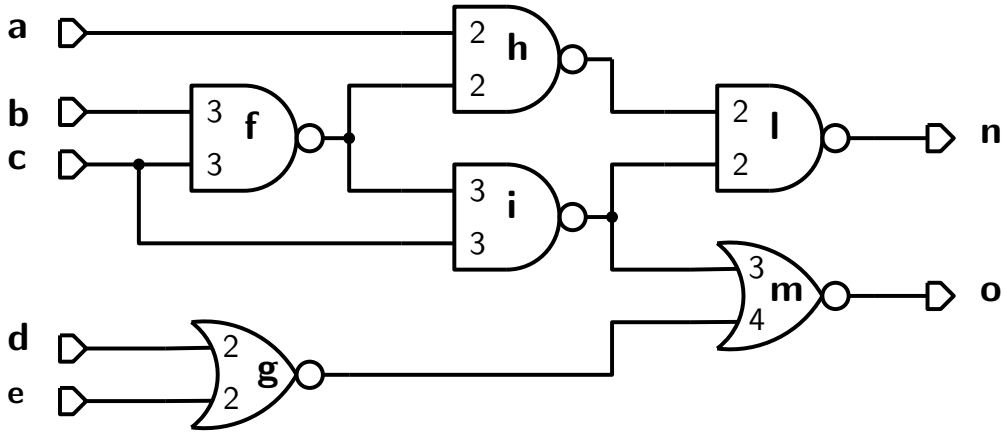
- Sono problemi di una classe di complessitá superiore a quella *NPC*
- Si tratta di problemi per i quali non esiste un algoritmo di verifica della soluzione di complessitá polinomiale

Esempio di analisi di un algoritmo

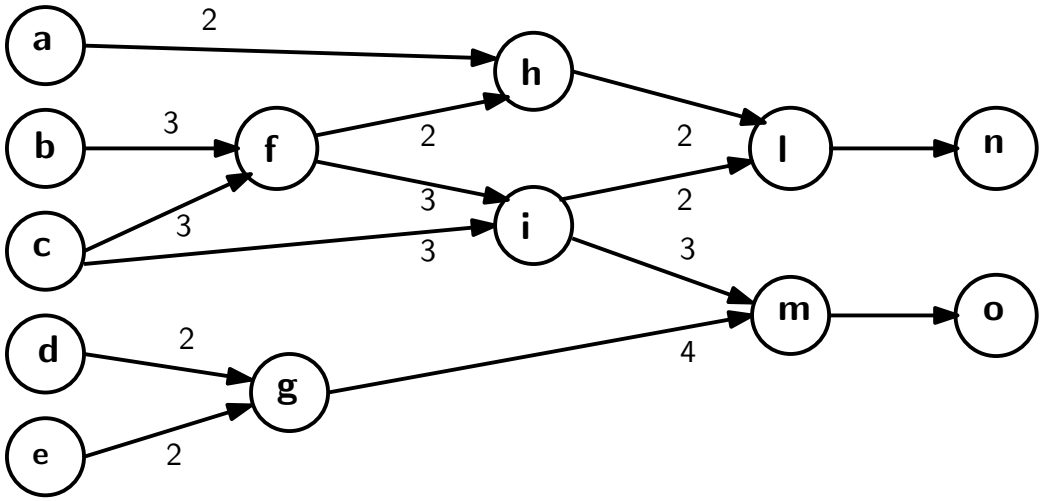
- Calcolo del cammino piú lungo in un grafo diretto aciclico $G = (V, E)$
  - insieme di vertici V
  - insieme di archi E
  - funzione che associa a ogni arco una lunghezza $L : E \rightarrow N$
- Se si interpretano:
  - V come l'insieme dei gate e pseudo-gate (PI e PO)
  - E come l'insieme di connessioni fra l'uscita di un gate e l'ingresso di un altro (non sono i segnali)
  - L come il ritardo associato a ciascuna di queste connessioni
- Il problema si riconduce al calcolo del cammino critico **statico** in una rete combinatoria

Reti combinatorie vs. DAG

- Rete combinatoria descritta con il modello di ritardo timing arc



- Grafo corrispondente



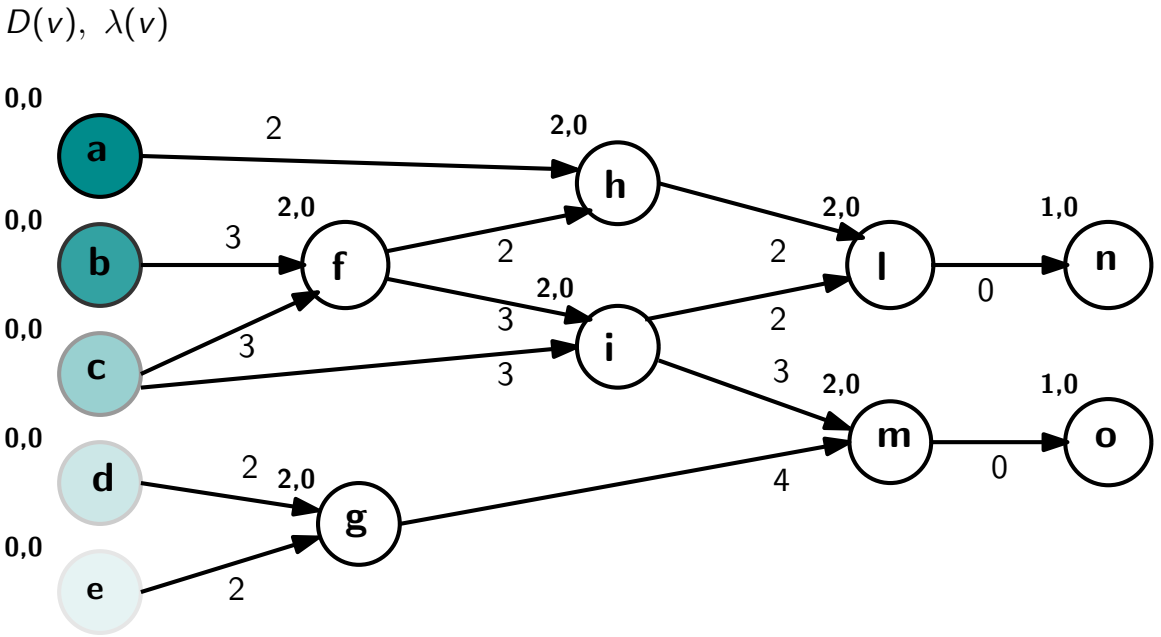
Algoritmo di Static Timing Analysis

- L'algoritmo attraversa in maniera iterattiva il grafo a partire dai nodi con grado di ingresso 0 (PIs) aggiornando a ogni passo un estremo inferiore ($\lambda(v)$) per il cammino piú lungo da ogni nodo $u \in I$ a ogni vertice v
- Alla fine dell'attraversamento, tale vettore contiene la lunghezza esatta di tale cammino piú lungo che rappresenta il Latest Stabilization Time LST del gate corrispondente
- Strutture dati ausiliarie:
  - $D(v)$ che viene inizializzata al grado di ingresso (fan-in della rete combinatoria) di ogni vertice
  - coda Q di tipo FIFO

Algoritmo STA

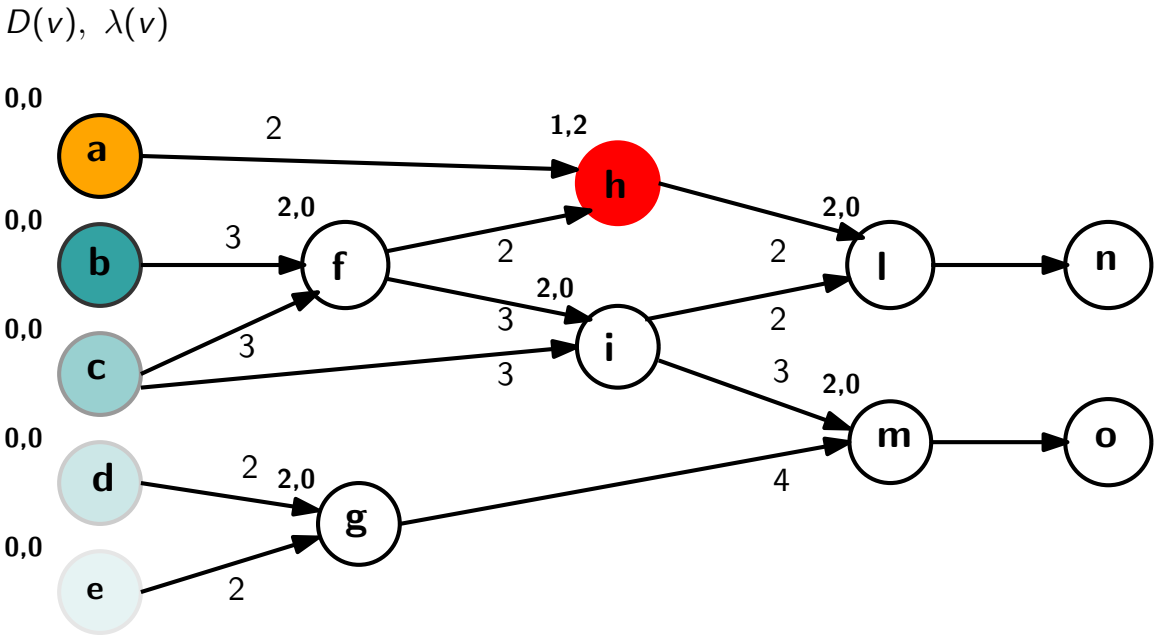
```
1  procedure Longest_Path(V, E, L, l)
2    {
3      n=|V|; m=|E|; q=|l|;
4      for (v ∈ V) {
5        λ(v) = 0;
6        D(v)=|pred(v)|; }
7      Q=Queue(l);
8      while (Q ≠ ∅) {
9        v=Dequeue(Q);
10       for (a ∈succ(v)) {
11         λ(a)=max(λ(a),(λ(v) + L(v, a)));
12         D(a)=D(a) − 1;
13         if (D(a) = 0) Queue(Q, a); }
14       }
15     λ_max = max_∀v(λ(v));
16     v_max=Select(V, λ_max);
17     π_crit=Backtrace(V, E, L, λ, v_max);
18     return (π_crit, λ);
19   }
```

Esempio



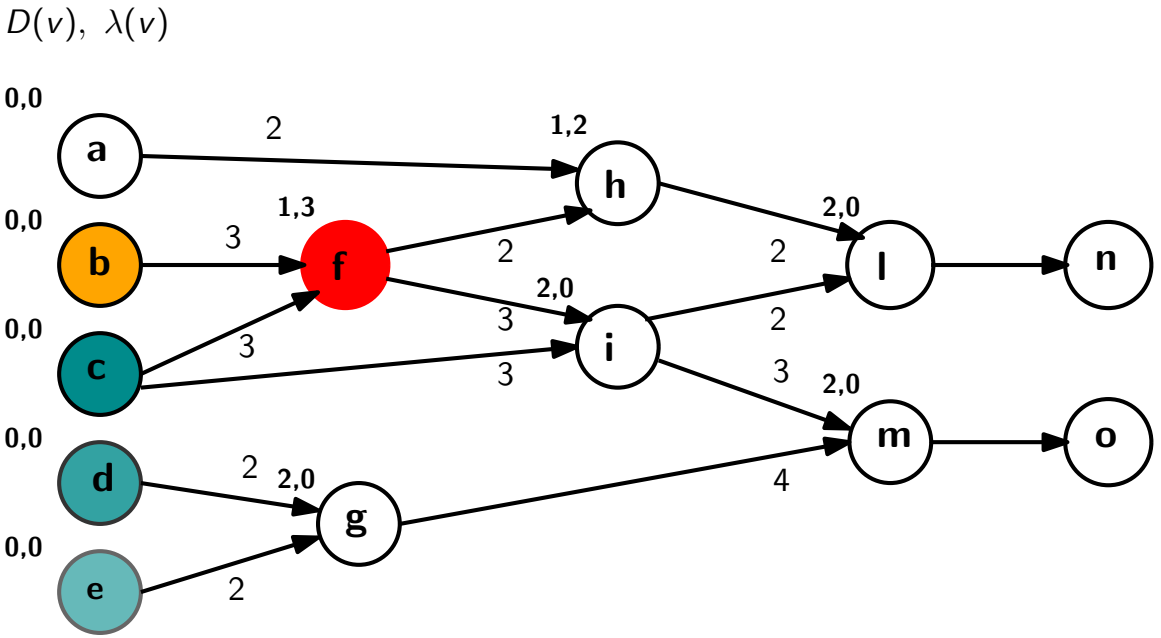
Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio



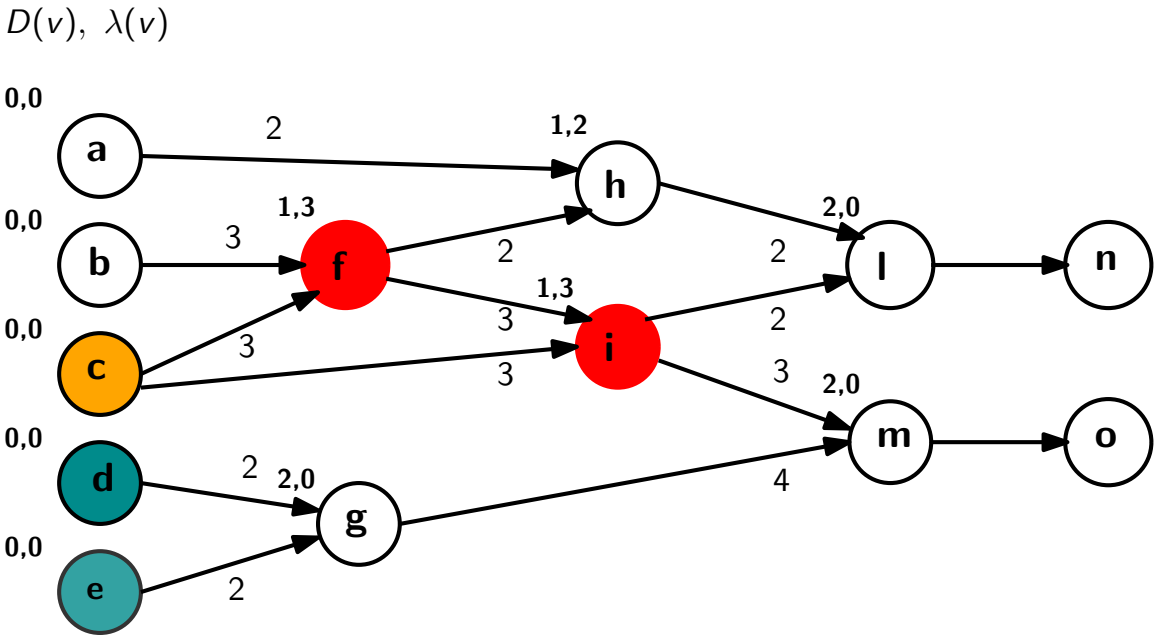
Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio



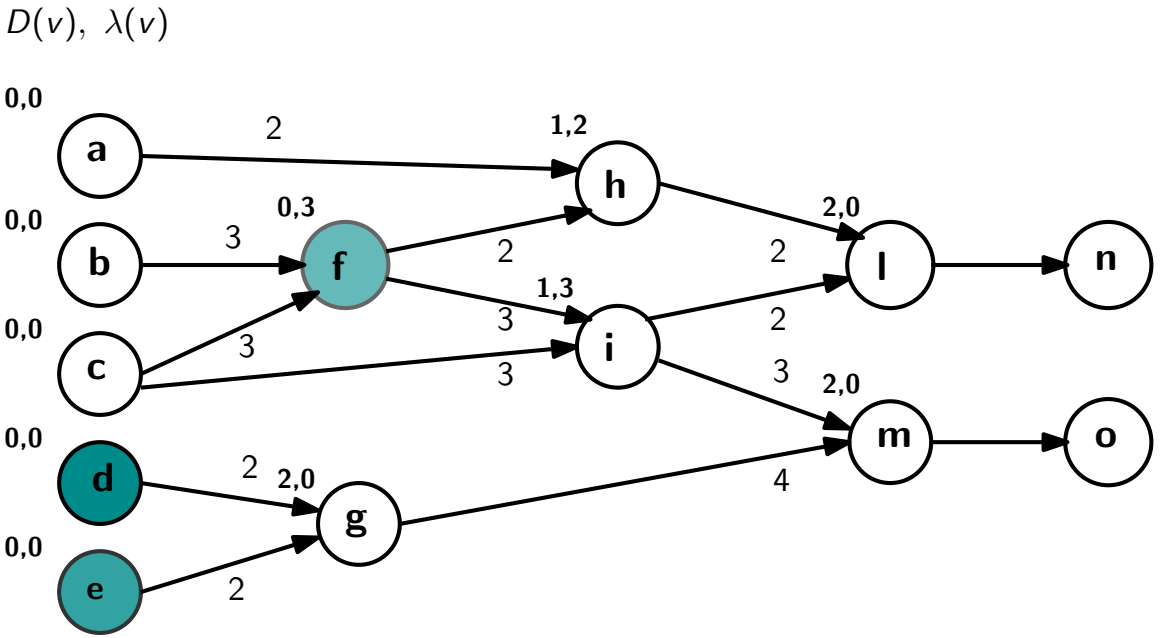
Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio



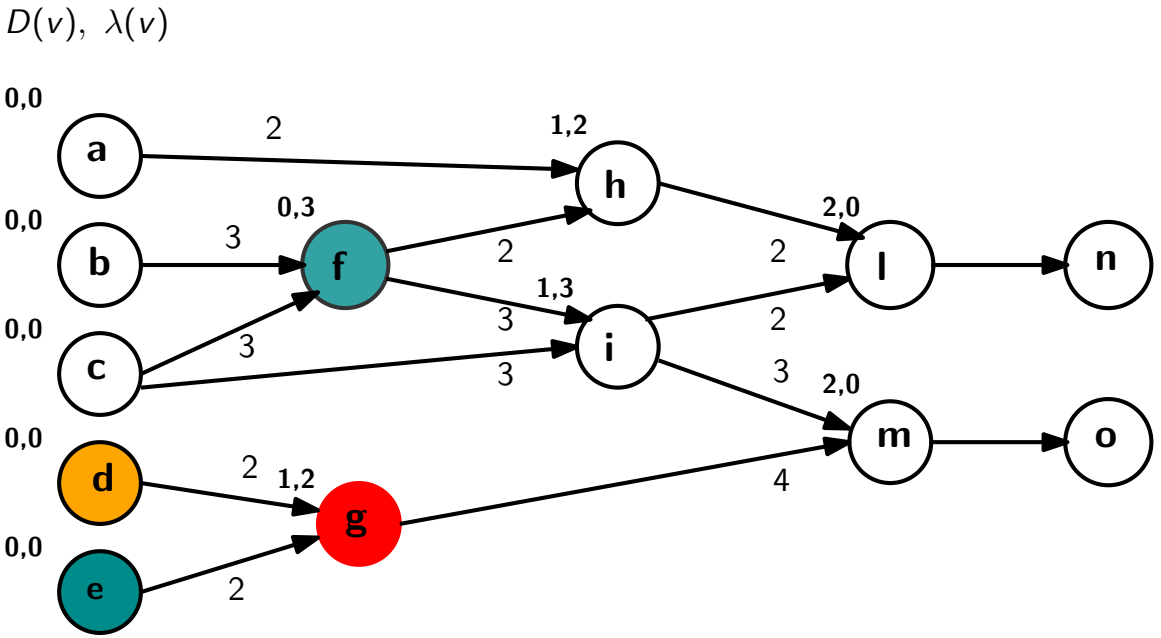
Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio



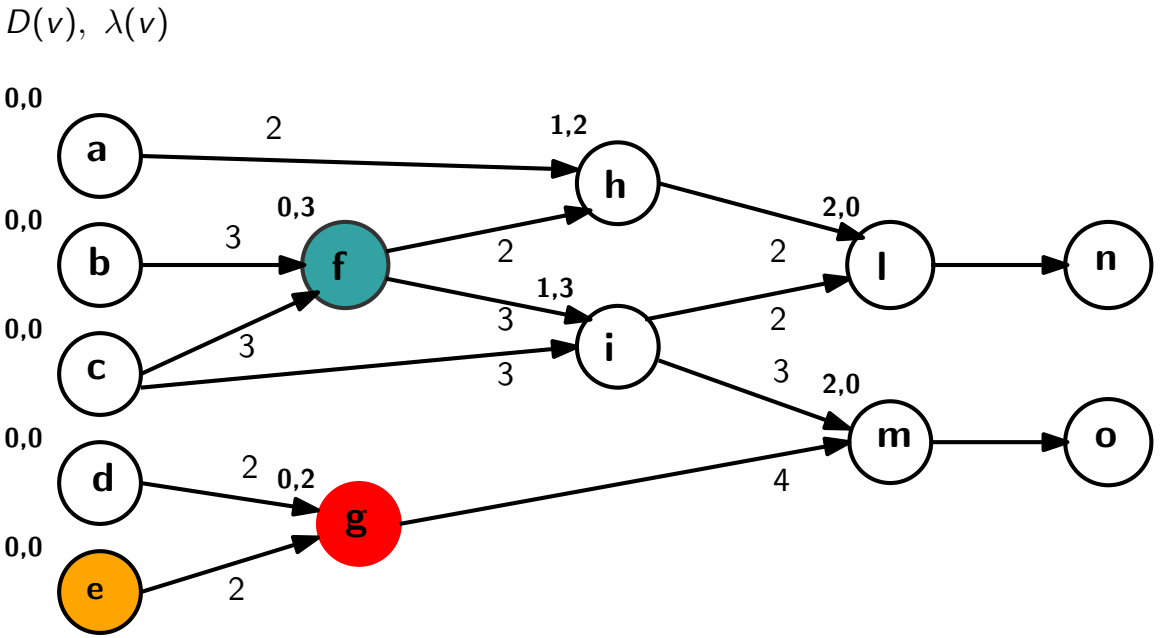
Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

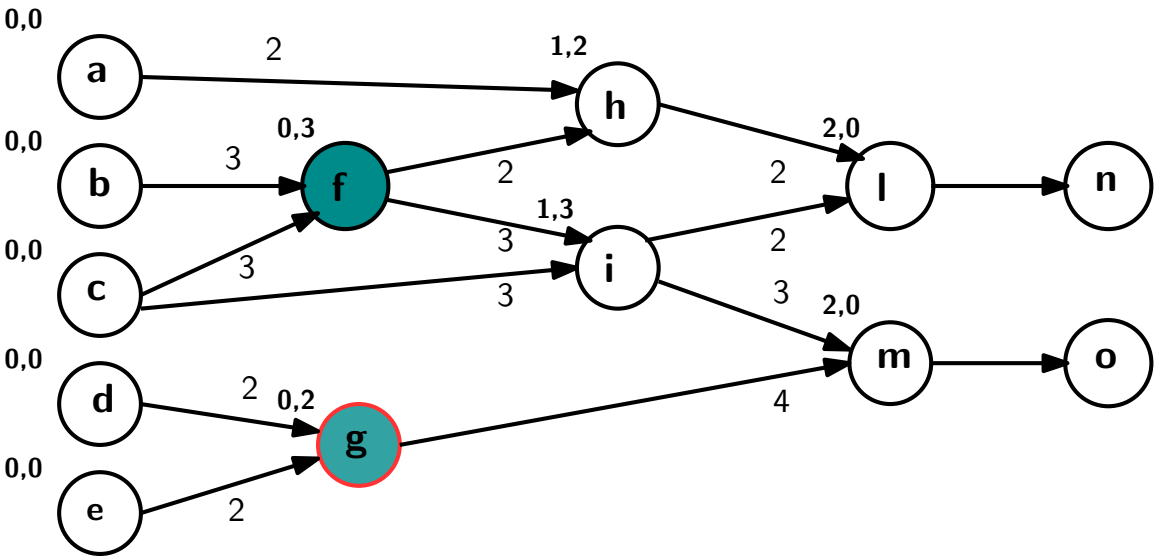
Esempio



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio

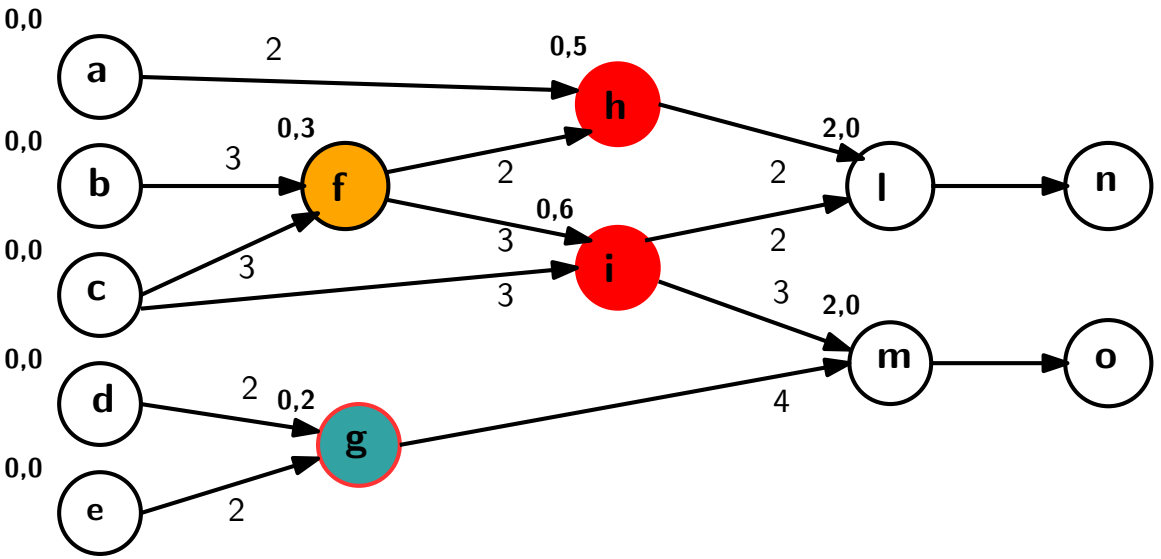
$D(v),\ \lambda(v)$



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio

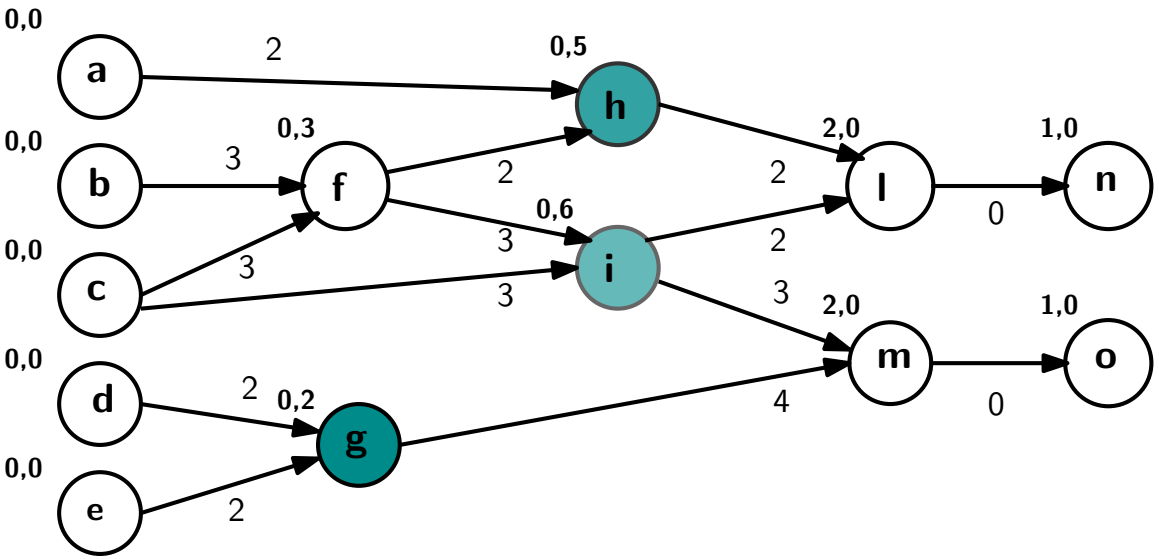
$D(v),\ \lambda(v)$



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio

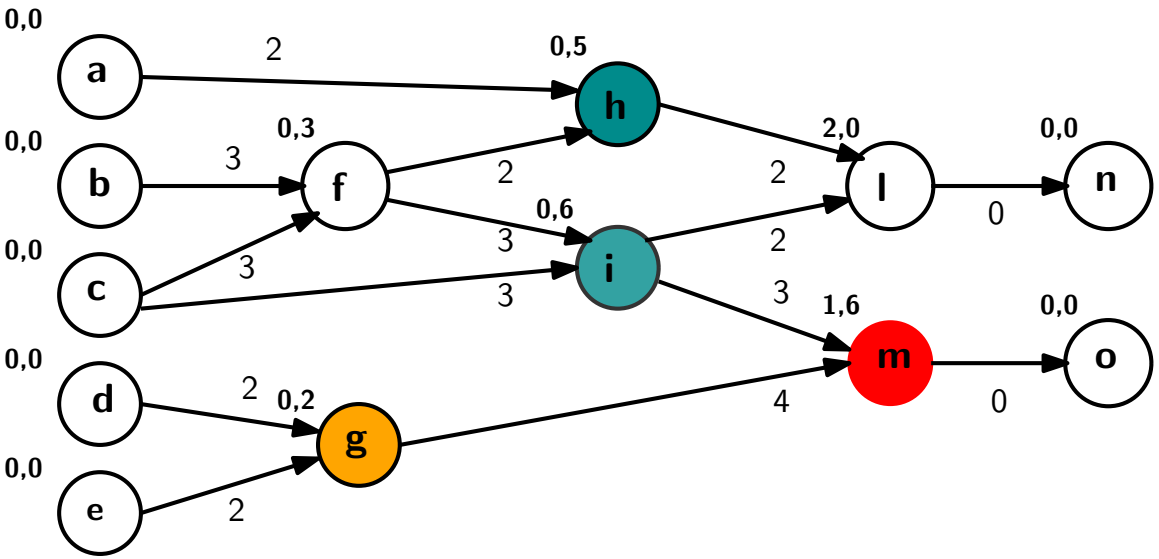
$D(v),\ \lambda(v)$



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

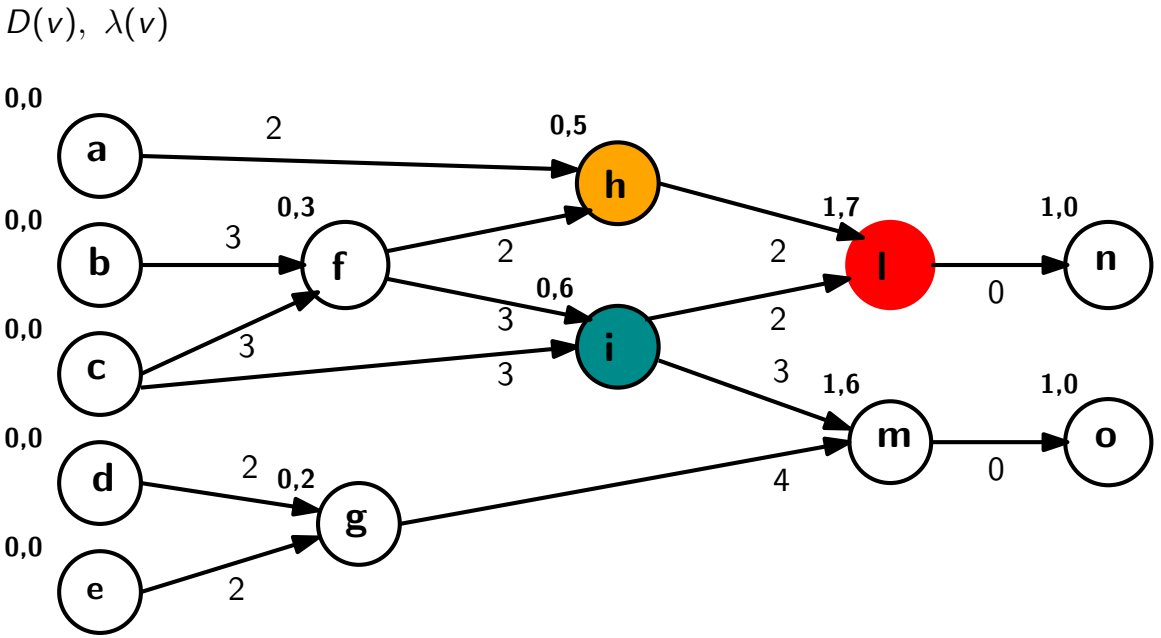
Esempio

$D(v),\ \lambda(v)$



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

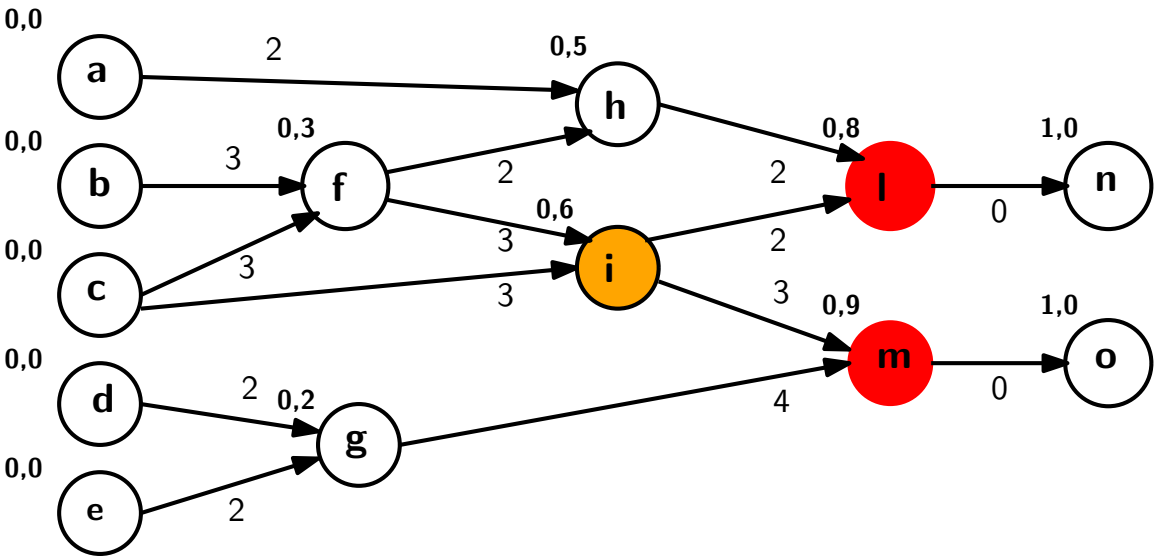
Esempio



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio

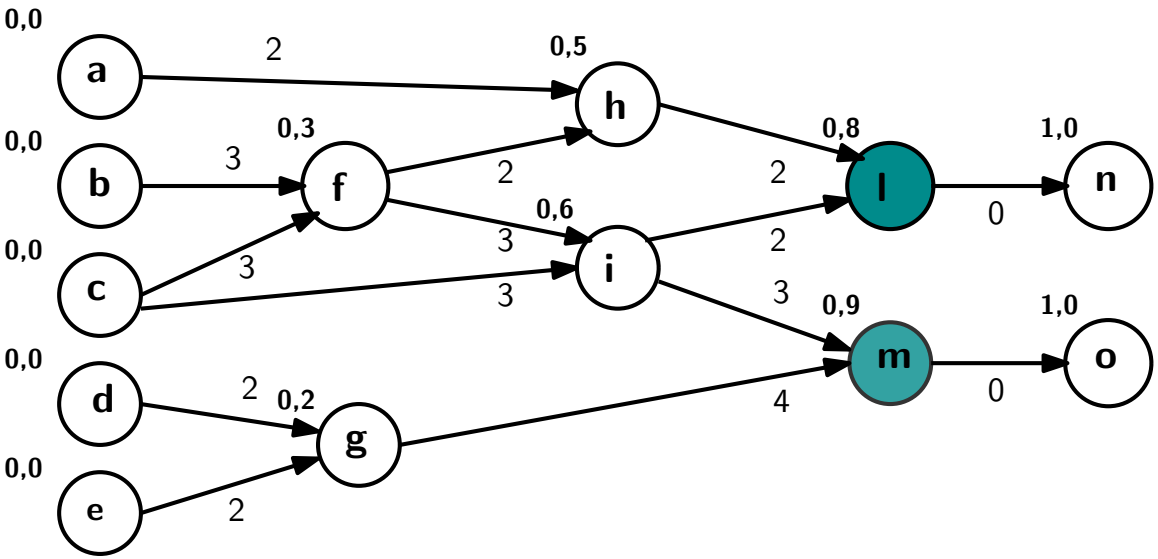
$D(v),\ \lambda(v)$



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio

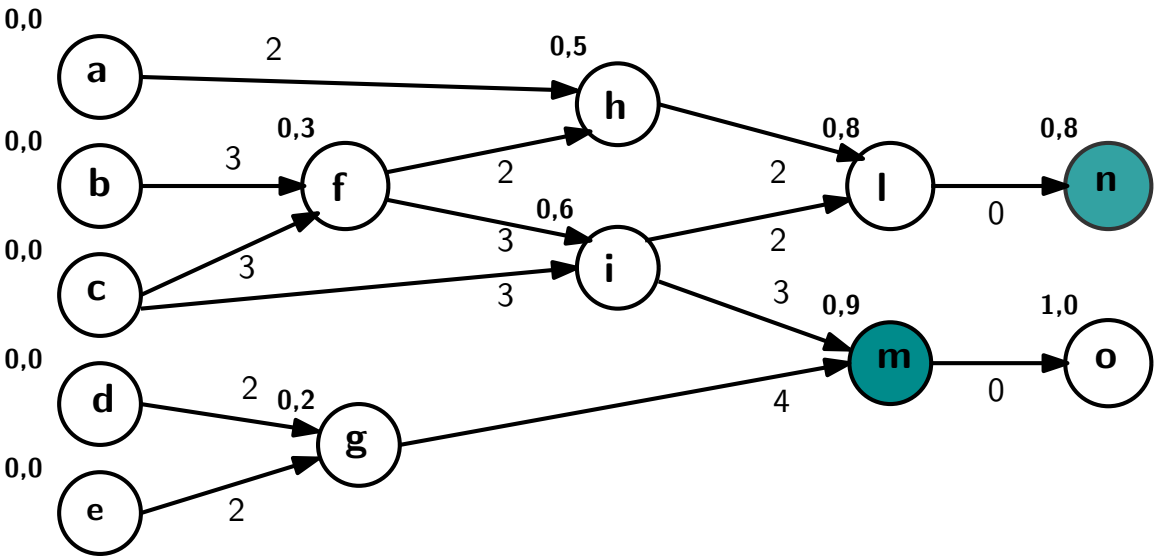
$D(v),\ \lambda(v)$



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio

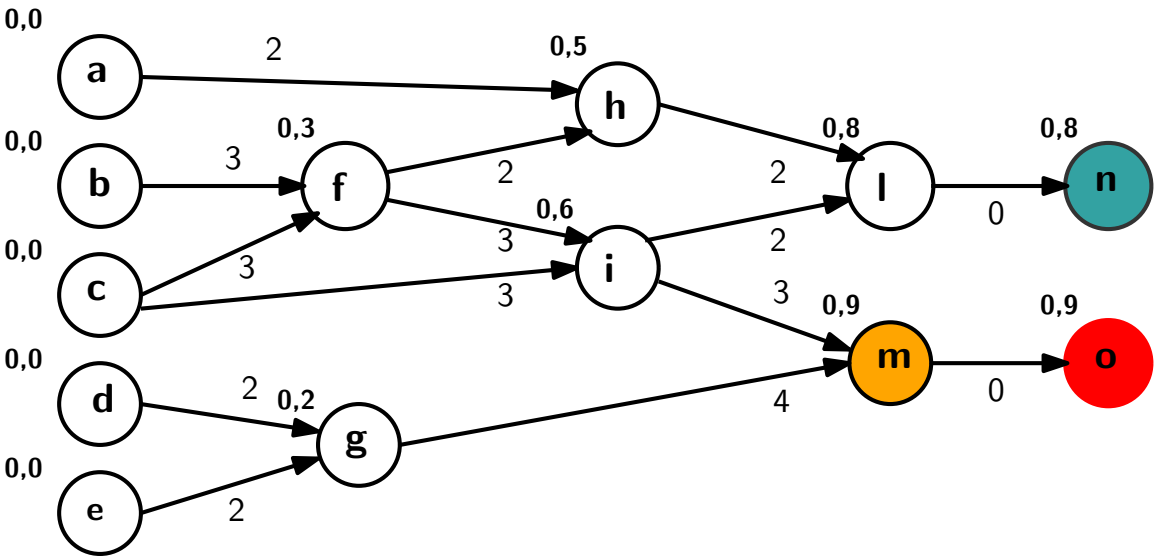
$D(v),\ \lambda(v)$



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio

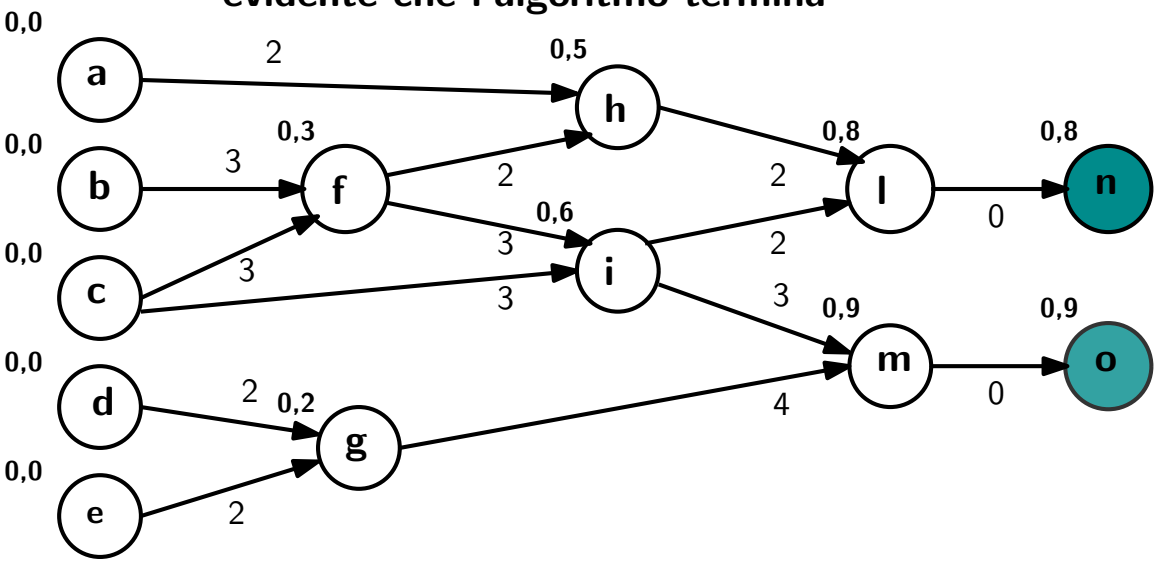
$D(v),\ \lambda(v)$



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Esempio

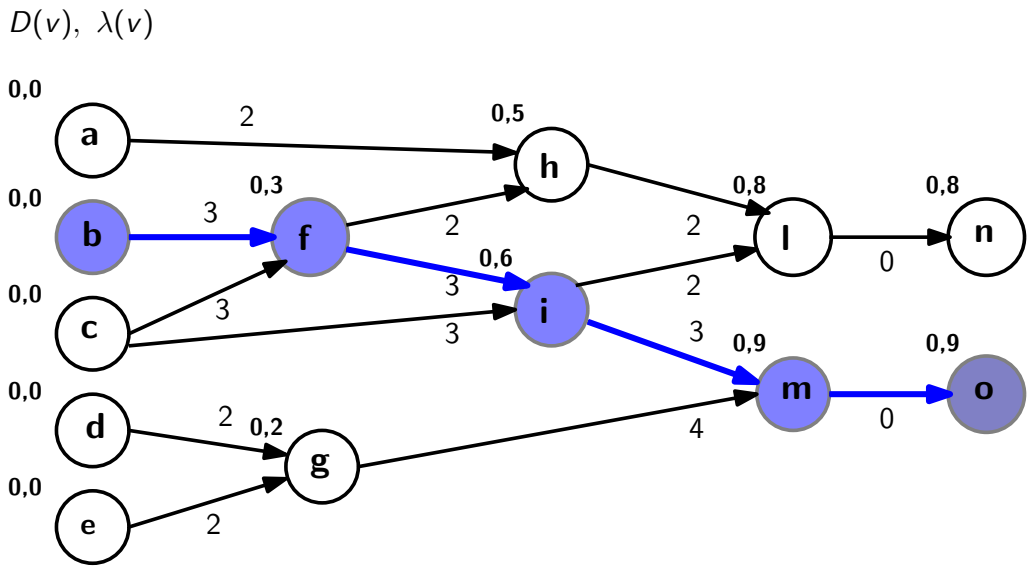
$D(v)$, $\lambda(v)$ **I passi successivi non sono illustrati in quanto é evidente che l'algoritmo termina**



Colori: FIFO Q azzurro (scuro per il primo elemento da estrarre), v arancione, i vari a in rosso

Backtracing

- Prima dell'esecuzione di questa procedura si é determinato il ritardo massimo a ciascun nodo
- La procedura `Backtrace` determina il cammino con il maggiore ritardo a partire dal vertice $v_{max} : \lambda(v) = \lambda_{max}$



Analisi della complessitá

- Utilizzeremo un approccio informale
- É importante scegliere la dimensione del problema
  - numero di nodi n: ogni nodo viene visitato piú volte
  - numero di archi m: ogni arco viene visitato una singola volta
- Quindi l'algoritmo é $O(m)$
- Si noti che si é supposto che le procedure di gestione siano eseguite in un tempo costante

Problemi dell'algoritmo di STA

- Si noti che l'algoritmo puó trovare un ritardo massimo e un cammino critico che esistono nel grafo ma non sono sensibilizzabili da alcuna configurazione di ingresso
- É interessante notare che se aggiungiamo questa condizione, il problema passa da $O(m)$ a NP-completo
- Infatti puó essere risolto applicando metodi di Boolean Satisfiability

Aspetti pratici

- Il comportamento asintotico di un algoritmo e la classe di complessitá di sono un punto di partenza
- Sono comunque possibili passi avanti notevoli nell'ambito delle tecniche di EDA ($0.1n^2$ é comunque meglio di $n^2 + n$ anche se entrambi gli algoritmi sono $O(n^2)$)
- Ad esempio nella Test Generation (problema *NP*-completo) sono stati fatti passi avanti corrispondenti a diversi ordini di grandezza di CPU time
- Analisi degli algoritmi (sviluppatori)
- Benchmarking (sviluppatori e utilizzatori)
- **Se il problema risulta intrattabile sia da un punto di vista teorico che pratico bisogna rinunciare alla soluzione esatta**

Euristici

- La maggior parte dei problemi di EDA sono *NPC* o *NP*-hard
- Gli algoritmi esatti risultano computazionalmente non fattibili in circuiti di dimensioni realistiche
- Per questo motivo si ricorre spesso a tecniche di tipo euristico che trovano una buona soluzione in alternativa a quella ottima in tempi accettabili
- Si tratta tipicamente di tecniche iterattive in cui la qualitá dell'ottimo locale raggiunto é tipicamente dipendente da
  - i tempi di calcolo concessi all'euristico
  - la soluzione iniziale
  - le scelte fatte durante l'esecuzione dell'algoritmo

Algoritmo di tipo Greedy

- A ogni iterazione viene fatta la scelta piú conveniente in quel momento
- In semplici casi questa tecnica porta a un ottimo assoluto:
  - proprietá *greedy-choice*: l'ottimo assoluto puó essere raggiunto tramite una sequenza di scelte localmente ottime
  - proprietá *optimal-substructure*: se la rcierca dell'ottimo puó essere ricondotta alla ricerca dell'ottimo in una serie di sotto-problemi

Esempio

- Ottimizzazione del costo di una espressione SP per una funzione Booleana
- Procedura Greedy: a ogni passo si utilizza l'implicante primo che copre il maggior numero di uni non ancora coperti

ab \ cd	00	01	11	10
00	0	1	0	0
01	0	1	1	0
11	1	1	1	0
10	0	0	1	0

$e = 0$

ab \ cd	00	01	11	10
00	0	0	1	1
01	0	0	1	0
11	0	0	0	0
10	0	0	0	0

$e = 1$

Branch and bound

- Si consideri un problema di ottimizzazione definito su insieme di variabili che appartengono a domini finiti e quindi con un insieme finito di possibili soluzioni
- L'esplorazione esaustiva delle possibili soluzioni (esatta) puó essere descritta tramite un albero di decisioni
- Senza perdere in generalitá consideriamo un albero di decisioni binario
- Evidentemente il costo computazionale della ricerca in tale albero (*branching*) é esponenziale

Branch and bound

- É evidente che nella maggior parte dei casi lo spazio delle possibili soluzioni é troppo vasto per poter essere esplorato
- Gli algoritmi di branch and bound cercano di ridurlo
  - mettendo limiti sulla qualitá della soluzione: bounding
  - eliminando le parti di albero che si stimano non essere interessanti: pruning
- Si noti che questa tecnica puó essere di tipo esatto o euristico dipendentemente dal fatto che il bounding sia esatto o approssimato

Esempio: tabelle di copertura

- Una tabella di copertura utilizzata per la minimizzazione di un espressione SP ha tante righe quanti sono gli implicanti e tante colonne quanti sono i mintermini
- La sua dimensione é $O(\frac{3^n}{n}2^n) \Rightarrow$ problema NP-completo
- In un primo passo possono essere applicate le condizioni di essenzialitá e dominanza
- Quello che rimane é dato da una situazione di tipo ciclico

Branching

- Il problema puó essere risolto con tecniche di branching (Quine - Mc Cluskey)
- L'algoritmo prova recursivamente tutte le possibili soluzioni determinandone il costo e trattenendo la migliore
- Attraversamento di un albero binario
- A ogni passo si possono applicare essenzialitá e dominanza, cosa che comunque rappresenta un costo (anche dal punto di vista del ripristino della tabella quando viene cambiata una decisione)

Branch and bound

- Tecnica implementata in Espresso (exact)
- Algoritmo recursivo che parte da una tabella ridotta con essenzialitá e dominanza
- A ogni passo viene calcolato un estremo inferiore per il costo previsto del sottoalbero corrispondente alla tabella ridotta
- Questo lower bound é dato dalla cardinalitá (utilizziamo il numero di implicanti come funzione di costo) del massimo insieme di colonne che sono fra loro disgiunte a coppie
- Trovare questo insieme é anch'esso un problema *NP*-completo, si possono utilizzare degli euristici (che possono rendere non esatto l'algoritmo)

Esempio di lower-bound per matrice di copertura X

	m	n	o	p	q
P_q	x				
P_r	x				
P_t		x		x	
P_u		x			x
P_v			x		
P_x			x		
P_y				x	x

Insiemi di colonne indipendenti: $X_i = \{m, n, o\}$, $X_j = \{m, o, p\}$, $X_k = \{m, o, q\}$ $\Rightarrow$ lower bound = 3

Si puó osservare che la copertura minima $C = \{P_q, P_t, P_x, P_y\}$ ha costo = 4

Esempio di lower-bound per matrice di copertura X

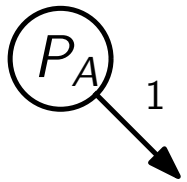
	m	n	o	p	q
P_q	x				
P_r	x			x	
P_t		x			
P_u		x			x
P_v			x		
P_x			x		
P_y				x	x

Insiemi di colonne indipendenti: $X_i = \{m, n, o\}$, $X_j = \{n, o, p\}$, $X_k = \{m, o, q\}$ $\Rightarrow$ lower bound = 3

Si puó osservare che la copertura minima $C = \{P_r, P_u, P_y\}$ (ad esempio) ha costo = 3

Applicazione

	7	10	14	15
P_A	x			
P_B	x			x
P_C			x	x
P_D		x	x	
P_E		x		

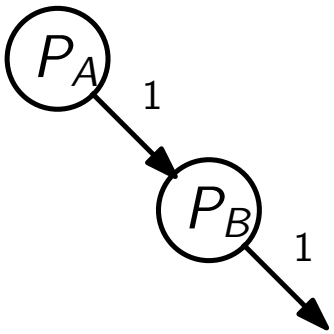


Lower bound=2 ($\{10, 15\}$)

Costo $\geq 1 + 2$

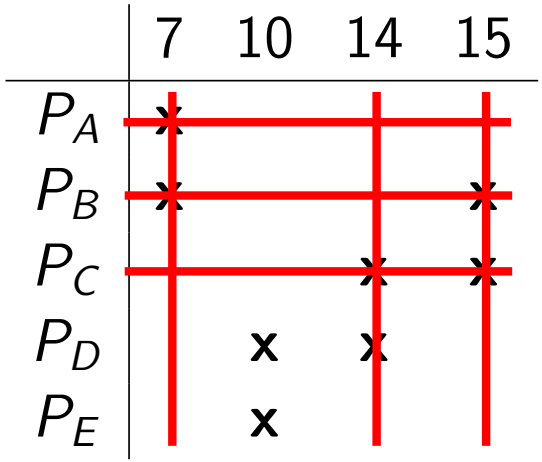
Applicazione

	7	10	14	15
A	*			*
B	*			*
C			x	x
D		x	x	
E		x		

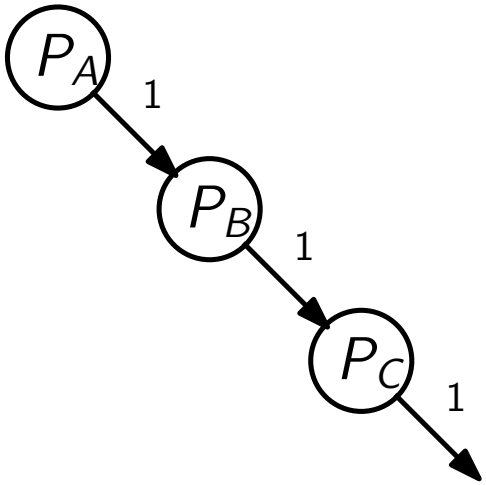


Lower bound=1 ($\{10\}$)
Costo $\geq 2 + 1$

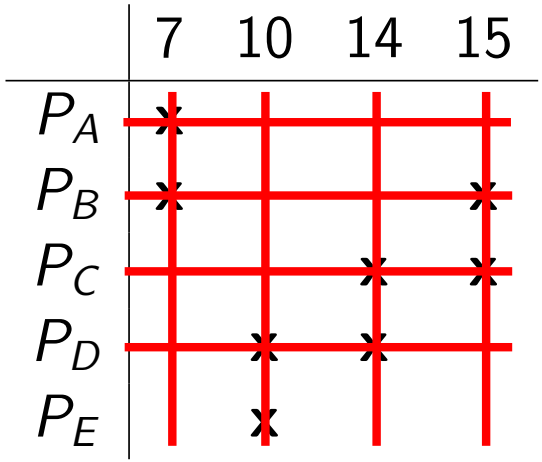
Applicazione



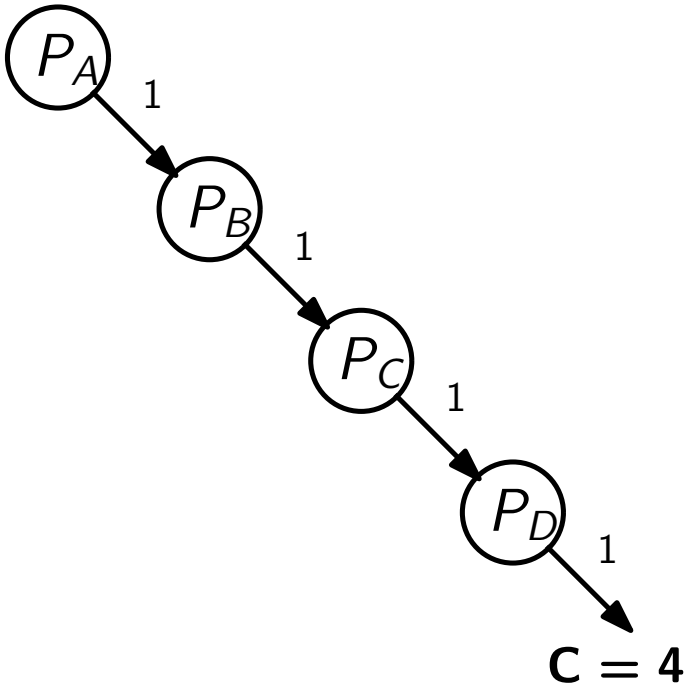
Lower bound=1 ($\{10\}$)
Costo $\geq 3 + 1$



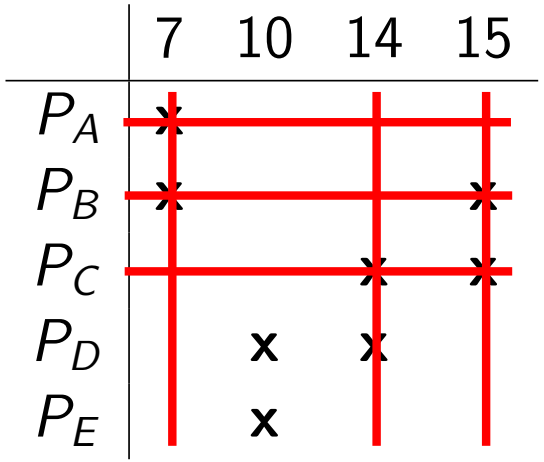
Applicazione



Lower bound=0
Costo = 4

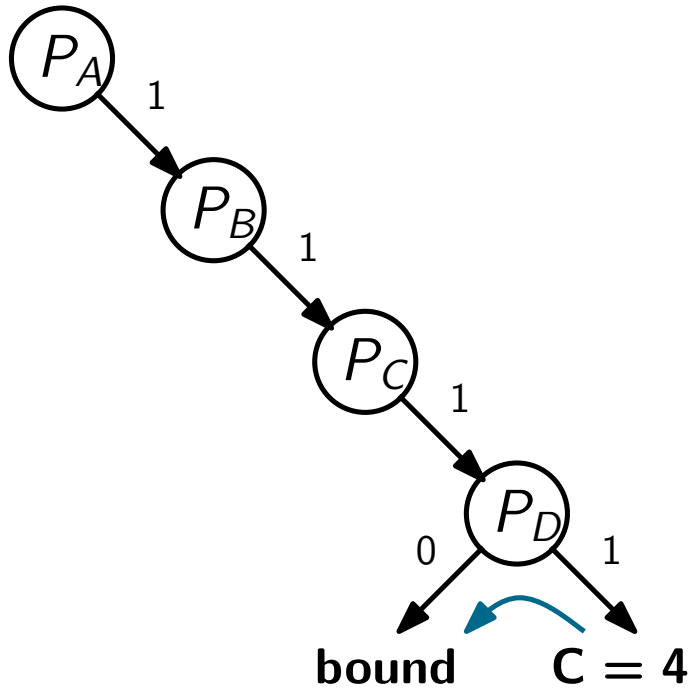


Applicazione



Lower bound=1 ($\{10\}$)

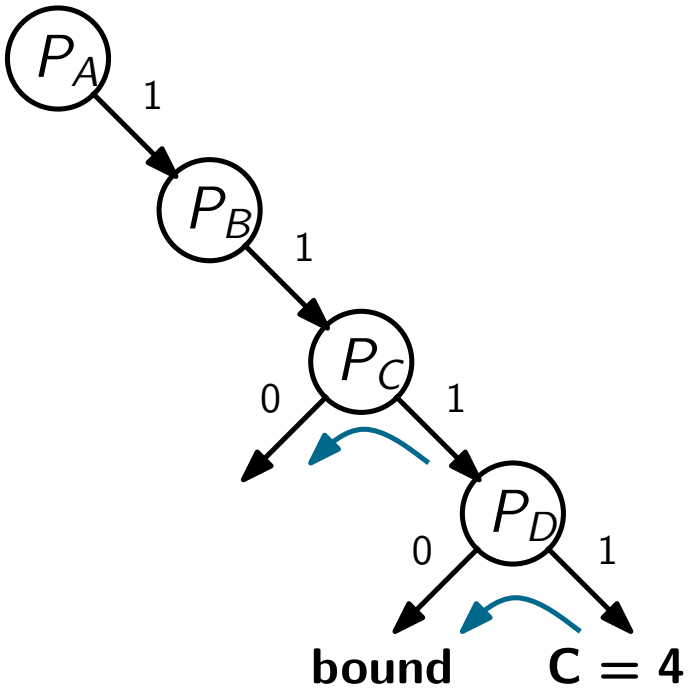
Costo $\geq 3 + 1 \Rightarrow$ bound



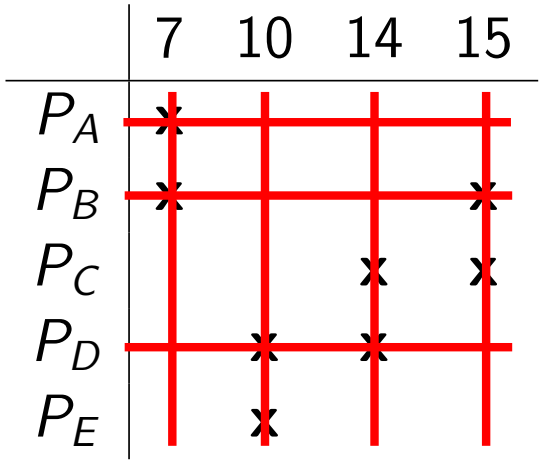
Applicazione

	7	10	14	15
P_A	x			x
P_B	x			x
P_C			x	x
P_D		x	x	
P_E		x		

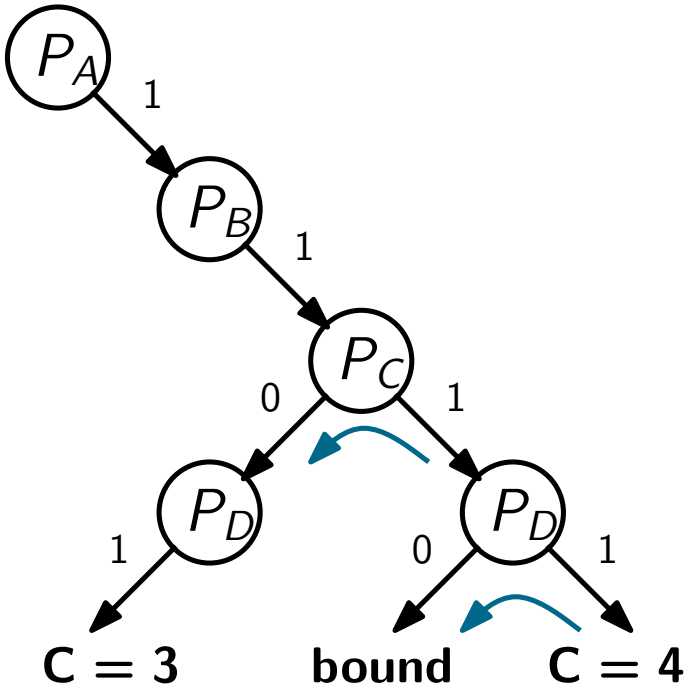
Lower bound=1 ({10})
Costo $\geq 2 + 1$



Applicazione



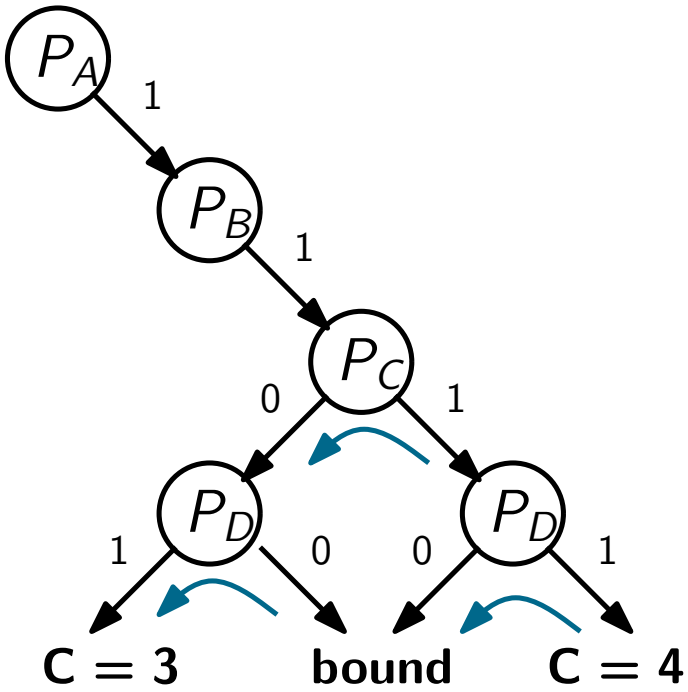
Lower bound=0
Costo = 3



Applicazione

	7	10	14	15
P_A	x			x
P_B	x			x
P_C			x	x
P_D		x	x	
P_E		x		

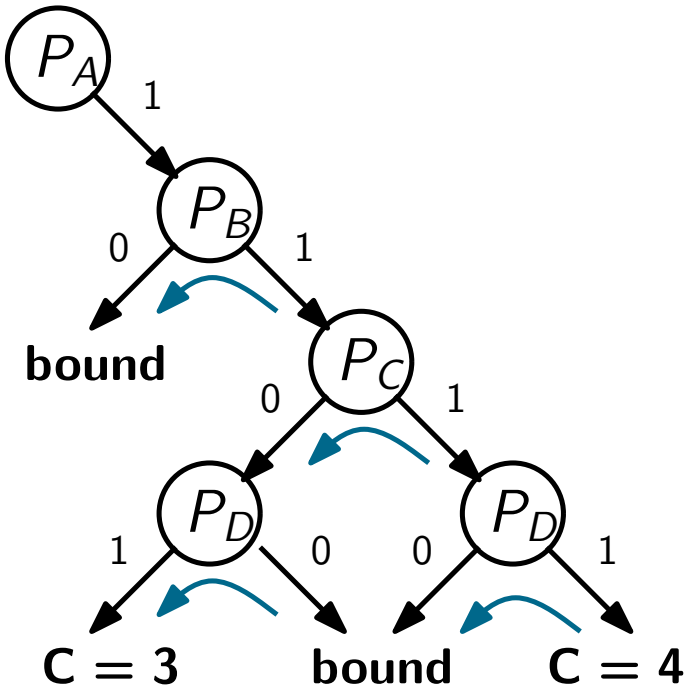
Lower bound=1 ({10})
Costo $\geq 2 + 1$



Applicazione

	7	10	14	15
P_A	x			
P_B	x			x
P_C			x	x
P_D		x	x	
P_E		x		

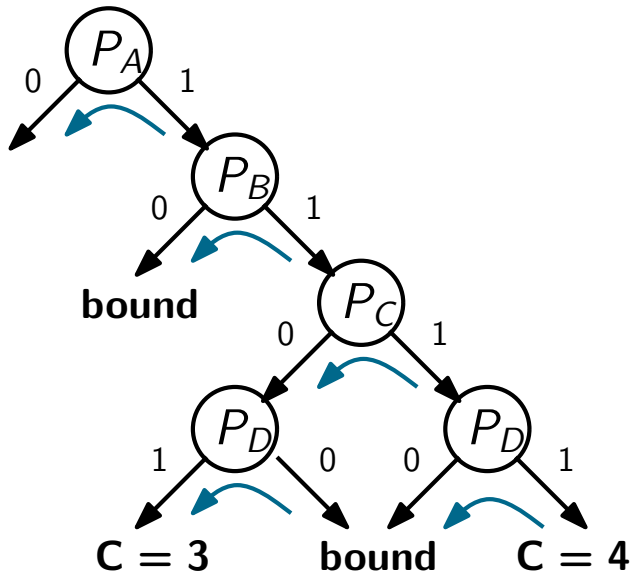
Lower bound=2 ($\{10, 15\}$)
Costo $\geq 1 + 2$



Applicazione

	7	10	14	15
P_A	x			
P_B	x			x
P_C			x	x
P_D		x	x	
P_E		x		

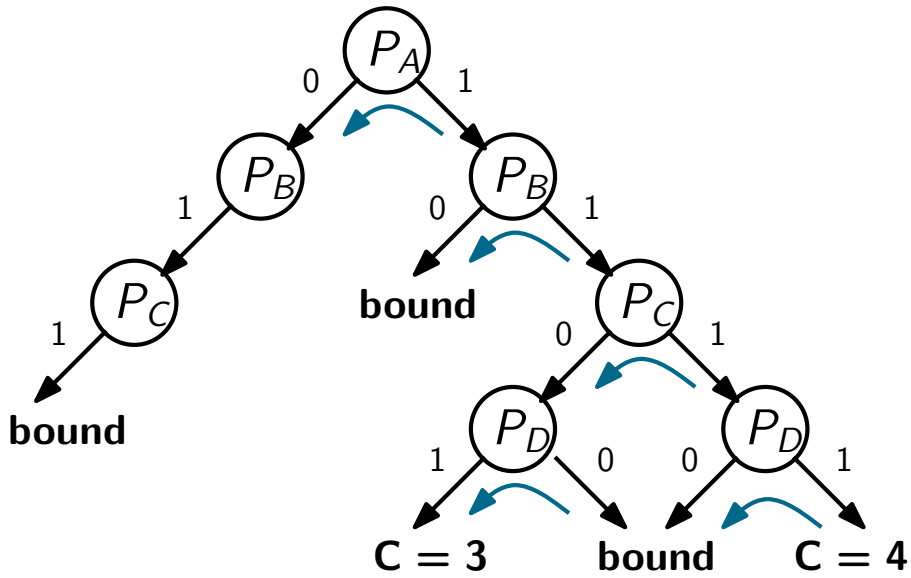
Lower bound=2 ($\{10, 15\}$)
Costo $\geq$ 2



Applicazione

	7	10	14	15
P_A	x			
P_B	x			x
P_C			x	x
P_D		x	x	
P_E		x		

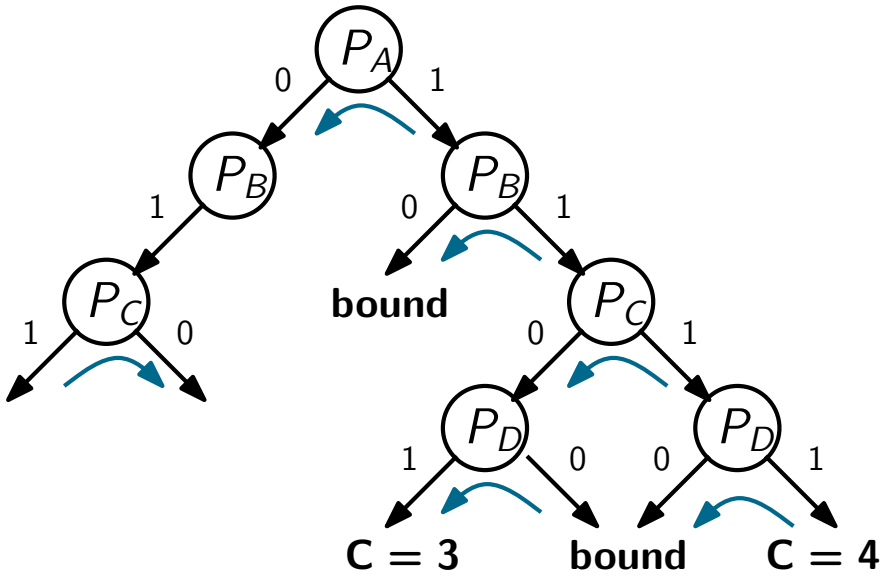
Lower bound=1 ($\{10\}$)
Costo $\geq 2+1$



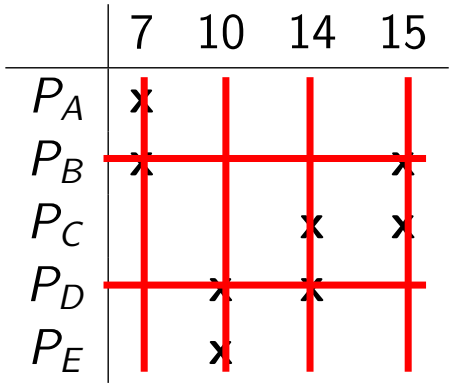
Applicazione

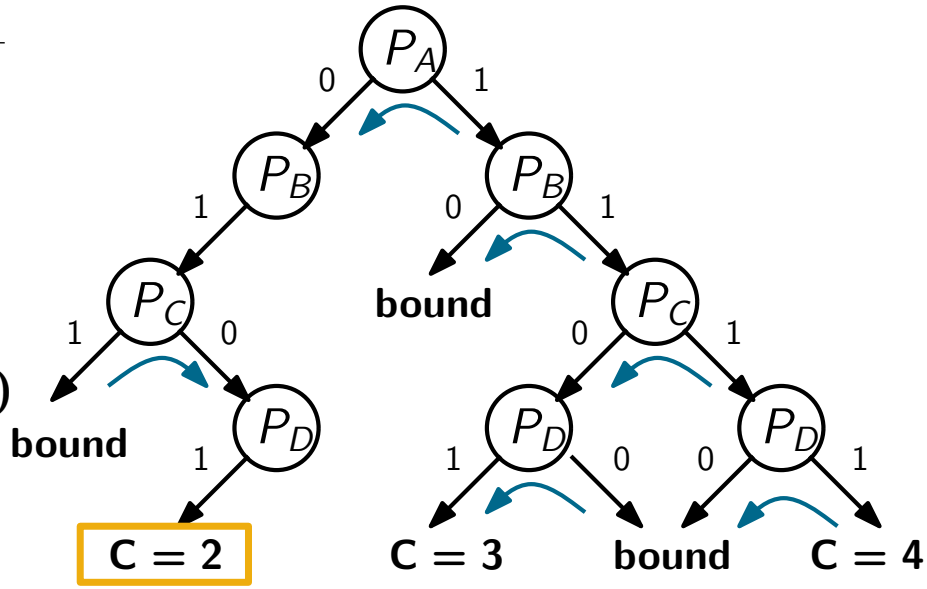
	7	10	14	15
P_A	x			
P_B	x			x
P_C			x	x
P_D		x	x	
P_E		x		

Lower bound=1 ({10})
Costo $\geq 1 + 1$



Applicazione





Euristici di ispirazione fisica e biologica

- Simulated annealing
- Algoritmi evolutivi
  - algoritmi genetici

Simulated annealing

- Storicamente ha trovato le maggiori applicazioni nell'ambito dell'ottimizzazione (place e route)
- L'annealing consiste nell'ottenere un cristallo (minimo di energia) da un materiale policristallino
- Si hanno diverse iterazioni con cicli di riscaldamento e raffreddamento
- Dopo ogni raffreddamento si raggiunge un minimo locale di energia, con il successivo riscaldamento si fornisce al sistema energia sufficiente per uscire da tale minimo
- Analogie:
  - energia - funzione di costo
  - movimenti molecolari - perturbazioni casuali nella soluzione
  - temperatura - soglia di accettazione di una nuova soluzione che viene affinata rispetto a quelle vecchie

Algoritmi genetici

- Problemi di ottimizzazione (sviluppati inizialmente per il problema del collaudo di reti sincrone)
- Radicalmente diversi da qualsiasi altro euristico: a ogni passo non si ha un valore di soluzione, ma una popolazione di soluzioni
- Ogni soluzione é rappresentata da un cromosoma, ovvero una codifica matematica che nel caso piú semplice é un vettore di bit di lunghezza fissa ciascuno dei quali viene chiamato gene

Semplice algoritmo genetico

- **Inizializzazione**: generazione della popolazione iniziale
- Fino a quando non si raggiunge una soluzione soddisfacente:
  - **Valutazione**: ogni individuo viene valutato sulla base di una funzione di fitness
  - **Selezione**: vengono selezionati i possibili genitori per una nuova generazione di soluzione
  - **Crossover**: viene generata una nuova generazione in cui due figli ereditano il patrimonio genetico di due genitori secondo diversi schemi di tipo casuale
  - **Mutazione**: i cromosoni di alcuni di questi individui vengono mutati in maniera casuale (si evita di rimanere bloccati in minimi locali)
  - **Sostituzione**: la nuova generazione sostituisce in tutto o in parte quella vecchia

Note sugli algoritmi evolutivi

- Vantaggi:
  - ottimizzazione multi-obbiettivo
  - esplorazione di soluzioni originali
- Svantaggi:
  - tempo di convergenza
  - ....

Conclusioni

- Decisioni per lo sviluppatore di tool EDA:
  - studio di soluzioni dedicate che sfruttino tutte le caratteristiche del problema in esame
  - utilizzo di risolutori general purpose
- Utilizzo di accelleratori hardware

Algoritmi dedicati vs general purpose solver

- il problema del calcolo dell'automa minimo puó essere facilmente descritto utilizzando il linguaggio **minizinc**

```
constraint forall (x in 1..ni)
 (forall (i in 1..states-1)
   (forall (j in i..states)
     (((outp[i,x]!=outp[j,x]) \/ (p[ns[i,x]]!=p[ns[j,x]])) -> (p[i]!=p[j])))));

constraint sums = sum (i in 1..states) (p[i]);
solve minimize sums;
```

  - Queste due linee di codice fanno il lavoro di circa 4,000 linee di codice C dedicato (commenti, i/o ....) nel programma **stamina**
  - Circa 54s di CPU per una FSM con 15 stati per un solver **minizinc** contro un tempo trascurabile per **stamina**