Complessità computazionale

— ○ —

Complessità degli algoritmi e complessità dei problemi

Piero A. Bonatti

Università di Napoli Federico II

Laurea Magistrale in Informatica

Tema della lezione

- Nel corso di Algoritmi si insegna a valutare la complessità degli algoritmi

Tema della lezione

- Nel corso di Algoritmi si insegna a valutare la complessità degli algoritmi
- Nel corso di Ricerca Operativa si sfiora il problema di stimare quante risorse *come minimo* deve usare un qualunque algoritmo che risolve un dato problema
    - *esistono algoritmi che risolvono il problema del commesso viaggiatore e terminano entro un tempo che cresce come una funzione polinomiale nella dimensione dell'input?*

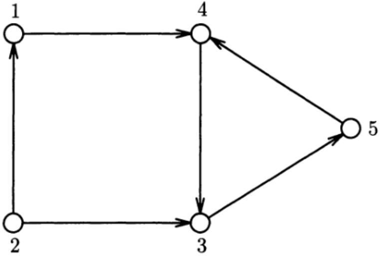
Tema della lezione

- Nel corso di Algoritmi si insegna a valutare la complessità degli algoritmi
- Nel corso di Ricerca Operativa si sfiora il problema di stimare quante risorse *come minimo* deve usare un qualunque algoritmo che risolve un dato problema
  - *esistono algoritmi che risolvono il problema del commesso viaggiatore e terminano entro un tempo che cresce come una funzione polinomiale nella dimensione dell'input?*
- Quindi iniziamo a distinguere la complessità degli algoritmi da quella dei problemi
  - approfittandone per rinfrescare alcune nozioni di base
  - e introdurre nuove problematiche

Graph Reachability

Cos'è un grafo

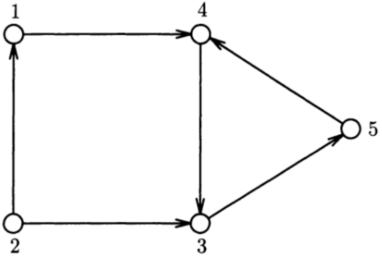
- Un grafo $G = (V, E)$ consiste di
  - un insieme V di nodi (o *vertici*)
  - un insieme E di archi (*edges*)
    - coppie ordinate di nodi (x, y)
    - quindi il grafo è diretto



Reachability

Definizione del problema

- Dati un grafo G e due nodi $x, y \in V$
- esiste un cammino da x a y?
- Esempi:
  - c'è un cammino da $x = 1$ a $y = 5$: $(1, 4, 3, 5)$
  - se invertissimo la direzione dell'arco $(4, 3)$ non vi sarebbero cammini



Reachability

Commenti

- Come ogni problema interessante REACHABILITY ha

1. un insieme infinito di istanze
   - ciascuna delle quali è un oggetto matematico
   - consiste di un grafo G e due dei suoi nodi
2. sulle quali formuliamo una domanda (e ci aspettiamo una risposta)

Reachability

Commenti

- Come ogni problema interessante REACHABILITY ha

1. un insieme infinito di istanze
   - ciascuna delle quali è un oggetto matematico
   - consiste di un grafo G e due dei suoi nodi
2. sulle quali formuliamo una domanda (e ci aspettiamo una risposta)

- Quando la risposta è "si" o "no" parliamo di problema di decisione (decision problem)
   - in teoria della complessità si riesce spesso a ridursi a problemi di decisione, e noi ci adegueremo

Reachability - Un (?) algoritmo che risolve il problema

Algorithm 1: Search algorithm

Input: $G = (V, E)$ e $x, y \in V$
Output: true o false ("yes" o "no")

for all $v \in V$ **do** marked$[v]$ = false
marked$[x]$ = true
$S := \{x\}$
while $S \neq \emptyset$ **do**
    estrai v da S
    forall $(v, w) \in E$ **do**
        if not marked$[w]$ **do** marked$[w]$ = true; inserisci w in S
    end
end
return marked$[y]$

Reachability

Correttezza

- Per la correttezza occorre dimostrare che

  L'algoritmo restituisce true sse (se e solo se) esiste un cammino da x a y

- La correttezza si dimostra con due semplici induzioni
  1. Se c'è un cammino tra x e y allora y sarà marcato
     - induzione su lunghezza cammino
  2. Se un nodo v è marcato allora c'è un cammino da x a v
     - induzione sul numero di iterazioni del "while"

Sviluppate i dettagli da soli per <u>esercizio</u>

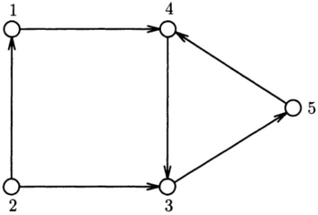
Reachability

Dettagli mancanti

- Com'è rappresentato esattamente il grafo? (codifica, encoding del problema)
  - può influire sulla stima di complessità?
  - una rappresentazione artificialmente grande può far sembrare l'algoritmo più efficiente
  - e viceversa
  - vedremo che se la codifica è "ragionevole" i dettagli della rappresentazione non contano
  - per adesso usiamo matrice di adiacenza

Reachability

Rappresentazione con matrice di adiacenza



$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Reachability

Dettagli mancanti (II)

- Com'è implementato l'insieme S?
  - in particolare inserzione ed estrazione (linee 5,7)
  - FIFO $\rightarrow$ visita breadth-first
  - LIFO $\rightarrow$ visita depth-first
  - (pseudo) casuale

Reachability

Dettagli mancanti (II)

- Com'è implementato l'insieme S?
  - in particolare inserzione ed estrazione (linee 5,7)
  - FIFO $\rightarrow$ visita breadth-first
  - LIFO $\rightarrow$ visita depth-first
  - (pseudo) casuale
  - non influisce sulla correttezza!
  - nè sulla performance (viene comunque visitato tutto il grafo)

Stima di efficienza (complessità dell'algoritmo)

- Ogni riga della matrice viene elaborata una sola volta
  - quando il suo indice v viene estratto da S (linea 5)
- Ogni elemento della riga viene elaborato una volta sola
- Quindi il numero di operazioni *elementari* è proporzionale a n^2 dove n è il numero di vertici

Stima di efficienza (complessità dell'algoritmo)

- Ogni riga della matrice viene elaborata una sola volta
  - quando il suo indice v viene estratto da S (linea 5)
- Ogni elemento della riga viene elaborato una volta sola
- Quindi il numero di operazioni *elementari* è proporzionale a n^2 dove n è il numero di vertici
- assumendo che le operazioni elementari (lettura e scrittura di singole variabili) richiedano tempo costante, possiamo identificare questo numero con il *tempo* di esecuzione dell'algoritmo
  - Al massimo il numero di vertici può essere proprio n^2
  - quindi, informalmente, siamo vicini all'ottimo (giusto il tempo di scandire l'input...)
- Diciamo che la complessità è $O(n^2)$

Digressione sulla notazione O & simili

- Date due funzioni f e g da $\mathbb{N}$ a $\mathbb{N}$
- scriviamo $f = O(g(n))$ se esistono $c, n_0 \in \mathbb{N}$ tali che
  - per ogni $n > n_0$, $f(n) \leq c \cdot g(n)$
  - (f cresce come g o più lentamente)

Digressione sulla notazione O & simili

- Date due funzioni f e g da $\mathbb{N}$ a $\mathbb{N}$
- scriviamo $f = O(g(n))$ se esistono $c, n_0 \in \mathbb{N}$ tali che
  - per ogni $n > n_0$, $f(n) \leq c \cdot g(n)$
  - (f cresce come g o più lentamente)
- scriviamo $f = \Omega(g(n))$ nel caso opposto
  - ovvero $g = O(f(n))$

Digressione sulla notazione O & simili

- Date due funzioni f e g da $\mathbb{N}$ a $\mathbb{N}$
- scriviamo $f = O(g(n))$ se esistono $c, n_0 \in \mathbb{N}$ tali che
  - per ogni $n > n_0$, $f(n) \leq c \cdot g(n)$
  - (f cresce come g o più lentamente)
- scriviamo $f = \Omega(g(n))$ nel caso opposto
  - ovvero $g = O(f(n))$
- scriviamo $f = \Theta(g(n))$ se $f = O(g(n))$ e $f = \Omega(g(n))$
  - (f e g hanno esattamente lo stesso *tasso di crescita*)

Digressione sulla notazione O & simili

Notazione delle funzioni f e g

- n è l'argomento della funzione, come in n^2, 2^n, $n^3 - 2n + 5$

Digressione sulla notazione O & simili

Notazione delle funzioni f e g

- n è l'argomento della funzione, come in n^2, 2^n, $n^3 - 2n + 5$
- essendo funzioni $\mathbb{N} \to \mathbb{N}$, anche quando il valore della funzione non sarebbe un intero noi lo intendiamo approssimato all'intero non-negativo superiore
  - ad esempio quando f è $\sqrt{n}, \log n$ ecc.
  - $f(n)$ in realtà significa $\max\{\lceil f(n) \rceil, 0\}$

Digressione sulla notazione O & simili

Proprietà interessanti

- Per ogni funzione f e per *ogni* costante $k > 0$,
  $f(n) = \Theta(k \cdot f(n))$ (*le costanti non contano*)
  - prendere $n_0 = 1$ e $c = 2/k$ in un verso e $c = k + 1$ nell'altro

Digressione sulla notazione O & simili

Proprietà interessanti

- Per ogni funzione f e per *ogni* costante $k > 0$,
  $f(n) = \Theta(k \cdot f(n))$ (*le costanti non contano*)
  - prendere $n_0 = 1$ e $c = 2/k$ in un verso e $c = k + 1$ nell'altro
- Se $p(n)$ è un polinomio di grado d, allora $p(n) = \Theta(n^d)$
  - cioè conta solo il termine principale

Digressione sulla notazione O & simili

Proprietà interessanti

- Per ogni funzione f e per *ogni* costante $k > 0$,
  $f(n) = \Theta(k \cdot f(n))$ (*le costanti non contano*)
  - prendere $n_0 = 1$ e $c = 2/k$ in un verso e $c = k + 1$ nell'altro
- Se $p(n)$ è un polinomio di grado d, allora $p(n) = \Theta(n^d)$
  - cioè conta solo il termine principale
- Se $c > 1$ e $p(n)$ è un polinomio, allora
  - $p(n) = O(c^n)$
  - ma *non* vale $c^n = O(p(n))$
  - (i polinomi crescono più lentamente di qualunque esponenziale)

Digressione sulla notazione O & simili

Proprietà interessanti

- Per ogni funzione f e per *ogni* costante $k > 0$,
  $f(n) = \Theta(k \cdot f(n))$ (*le costanti non contano*)
  - prendere $n_0 = 1$ e $c = 2/k$ in un verso e $c = k + 1$ nell'altro
- Se $p(n)$ è un polinomio di grado d, allora $p(n) = \Theta(n^d)$
  - cioè conta solo il termine principale
- Se $c > 1$ e $p(n)$ è un polinomio, allora
  - $p(n) = O(c^n)$
  - ma *non* vale $c^n = O(p(n))$
  - (i polinomi crescono più lentamente di qualunque esponenziale)
- Analogamente
  - $\log n = O(n)$
  - $\log^k n = O(n)$ per qualunque k

Digressione sugli algoritmi polinomiali

E problemi trattabili

- Per convenzione, si ritiene che i problemi *trattabili* in pratica siano quelli risolubili in tempo polinomiale
  - mentre quelli risolubili in tempo esponenziale, come 2^n, $n!$ o peggio ci preoccupano

Digressione sugli algoritmi polinomiali

E problemi trattabili

- Per convenzione, si ritiene che i problemi *trattabili* in pratica siano quelli risolubili in tempo polinomiale
  - mentre quelli risolubili in tempo esponenziale, come 2^n, $n!$ o peggio ci preoccupano
  - la maggior parte di questo corso riguarda i problemi per cui, nonostante i nostri sforzi, *non troviamo* un algoritmo polinomiale

Digressione sugli algoritmi polinomiali

E problemi trattabili

- Per convenzione, si ritiene che i problemi *trattabili* in pratica siano quelli risolubili in tempo polinomiale
  - mentre quelli risolubili in tempo esponenziale, come 2^n, $n!$ o peggio ci preoccupano
  - la maggior parte di questo corso riguarda i problemi per cui, nonostante i nostri sforzi, *non troviamo* un algoritmo polinomiale
  - esistono anche problemi che *sicuramente* non possono essere risolti in tempo polinomiale

Digressione sugli algoritmi polinomiali

E problemi trattabili

- Per convenzione, si ritiene che i problemi *trattabili* in pratica siano quelli risolubili in tempo polinomiale
  - mentre quelli risolubili in tempo esponenziale, come 2^n, $n!$ o peggio ci preoccupano
  - la maggior parte di questo corso riguarda i problemi per cui, nonostante i nostri sforzi, *non troviamo* un algoritmo polinomiale
  - esistono anche problemi che *sicuramente* non possono essere risolti in tempo polinomiale
  - (per non parlare di quelli che sicuramente non possono essere risolti, punto)

Digressione sugli algoritmi polinomiali (II)

E problemi trattabili

- La classe degli algoritmi polinomiali risulta particolarmente elegante e conveniente da un punto di vista matematico
  - I polinomi costituiscono una classe stabile, chiusa rispetto a somma, moltiplicazione e certe composizioni:
  - se $p(n)$ e $f(n)$ sono polinomi, sia $p(f(n))$ che $f(p(n))$ sono polinomi
  - Di conseguenza, varie forme di combinazione e composizione di algoritmi polinomiali continuano a produrre algoritmi polinomiali

Digressione sugli algoritmi polinomiali (III)

E problemi trattabili

- L'equazione "polinomiale = trattabile" è forse comoda ma controversa
  - algoritmi $O(n^2)$ non sono applicabili a grandi basi di dati o all'insieme dei linked open data pubblicati sul web

Digressione sugli algoritmi polinomiali (III)

E problemi trattabili

- L'equazione "polinomiale = trattabile" è forse comoda ma controversa
  - algoritmi $O(n^2)$ non sono applicabili a grandi basi di dati o all'insieme dei linked open data pubblicati sul web
  - il simplesso, anche se esponenziale nel caso peggiore, in pratica risolve i problemi di programmazione lineare più velocemente dell'*ellipsoid algorithm* che è polinomiale

Digressione sugli algoritmi polinomiali (III)

E problemi trattabili

- L'equazione "polinomiale = trattabile" è forse comoda ma controversa
  - algoritmi $O(n^2)$ non sono applicabili a grandi basi di dati o all'insieme dei linked open data pubblicati sul web
  - il simplesso, anche se esponenziale nel caso peggiore, in pratica risolve i problemi di programmazione lineare più velocemente dell'*ellipsoid algorithm* che è polinomiale
  - i ragionatori per il semantic web (standard OWL) arrivano fino a $O(2^{2^n})$ (double exponential time!) ma grazie a ottimizzazioni di varia natura sono utilizzabili in pratica

Digressione sugli algoritmi polinomiali (III)

E problemi trattabili

- L'equazione "polinomiale = trattabile" è forse comoda ma controversa
  - algoritmi $O(n^2)$ non sono applicabili a grandi basi di dati o all'insieme dei linked open data pubblicati sul web
  - il simplesso, anche se esponenziale nel caso peggiore, in pratica risolve i problemi di programmazione lineare più velocemente dell'*ellipsoid algorithm* che è polinomiale
  - i ragionatori per il semantic web (standard OWL) arrivano fino a $O(2^{2^n})$ (double exponential time!) ma grazie a ottimizzazioni di varia natura sono utilizzabili in pratica
  - la teoria studia *worst case complexity* e i casi in cui il caso peggiore si manifesta possono essere rari

Digressione sugli algoritmi polinomiali (III)

E problemi trattabili

- L'equazione "polinomiale = trattabile" è forse comoda ma controversa
  - algoritmi $O(n^2)$ non sono applicabili a grandi basi di dati o all'insieme dei linked open data pubblicati sul web
  - il simplesso, anche se esponenziale nel caso peggiore, in pratica risolve i problemi di programmazione lineare più velocemente dell'*ellipsoid algorithm* che è polinomiale
  - i ragionatori per il semantic web (standard OWL) arrivano fino a $O(2^{2^n})$ (double exponential time!) ma grazie a ottimizzazioni di varia natura sono utilizzabili in pratica
  - la teoria studia *worst case complexity* e i casi in cui il caso peggiore si manifesta possono essere rari
  - i problemi più interessanti sono quelli per cui non si conoscono soluzioni polinomiali – non si rinuncia a risolverli...

Reachability - Commenti

- Incidentalmente, tutti e tre gli algoritmi per REACHABILITY ottenuti in questo modo hanno la stessa complessità
- Questo farebbe pensare che la complessità degli algoritmi e della soluzione del problema siano la stessa cosa
- Ma voi sapete già che non è così

Sort

Ordinamento di un array con n elementi

algoritmo	tempo		spazio
	complessità media	worst case	
insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \cdot \log n)$	$O(n^2)$	$O(n)$ (naive)
merge sort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
heap sort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(1)$

Sort - Commenti

- Quindi uno stesso problema può essere risolto con algoritmi anche *radicalmente* diversi
- Alcuni sono "migliori" di altri
- E allora qual è la complessità *intrinseca* del problema?

Sort - Commenti

- Quindi uno stesso problema può essere risolto con algoritmi anche *radicalmente* diversi
- Alcuni sono “migliori” di altri
- E allora qual è la complessità *intrinseca* del problema?
- Proposta: le risorse minime che occorrono per risolvere il problema, ad es.
  - tempo di computazione del migliore algoritmo
  - quantità di memoria del migliore algoritmo

  (ma se ne potrebbero inventare altre...)
- Peccato che determinare le risorse minime non sia affatto semplice...

Risorse minime per Sort

- Spazio: meglio di $O(1)$ non si può!
- Tempo: $O(n \log n)$ vicino al minimo possibile ($O(n)$)
  - non si può fare a meno di esaminare almeno una volta ogni elemento dell'array
  - la sua posizione non dipende solo dai valori degli altri elementi
  - può finire in qualunque posizione

Risorse minime per Sort

- Spazio: meglio di $O(1)$ non si può!
- Tempo: $O(n \log n)$ vicino al minimo possibile ($O(n)$)
  - non si può fare a meno di esaminare almeno una volta ogni elemento dell'array
  - la sua posizione non dipende solo dai valori degli altri elementi
  - può finire in qualunque posizione
- Ma esistono problemi ben più complicati da analizzare
  - come vedremo in relazione al problema del commesso viaggiatore
  - e alla questione se $\mathrm{P} \neq \mathrm{NP}$

Esercitazione

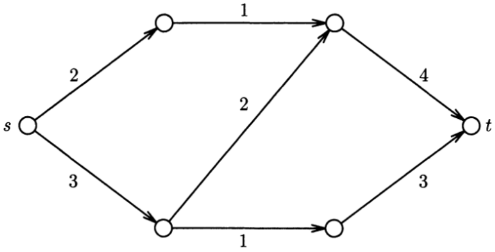
Esercizio 1

Dimostrare la correttezza dell'Algoritmo 1

Max Flow

Networks

- Una rete (network) è una n-upla $N = (V, E, s, t, c)$ dove
  - (V, E) è un grafo
  - s e t sono il *source node* e il *sink node*, rispettivamente
  - $c(i, j)$ è la capacità dell'arco i, j
- Può astrarre problemi di traffico, trasporti, reti idrauliche, ...



Max Flow

Flows

- Un flusso (flow) f in N
  - assegna un intero $f(i,j) \leq c(i,j)$ ad ogni arco $(i,j) \in E$
  - per ogni nodo, tranne s e t, la somma degli f entranti deve essere uguale alla somma degli f uscenti

$$\forall i \in V \setminus \{s,t\}, \quad \sum_{(j,i)\in E} f(j,i) = \sum_{(i,k)\in E} f(i,k)$$

- il valore di un flusso è la somma degli f uscenti da s
  - equivalentemente la somma degli f entranti in t

Max Flow

Definizione del problema

- Data una rete N
- trovare un flusso di valore massimo
- Nota:
  - non è un problema di decisione (la risposta non è semplicemente "si" o "no")
  - è un problema di *ottimizzazione*
  - ma i problemi di ottimizzazione hanno un problema di decisione corrispondente
  - se il problema di decisione si può risolvere in tempo polinomiale, anche il problema di ottimizzazione ha la stessa proprietà

Max Flow

Problema di decisione corrispondente

- Data una rete N e un intero K
- dire se esiste un flusso il cui valore è $\geq K$
- Con un numero polinomiale di chiamate a un algoritmo che risolve questo problema si può risolvere MAX FLOW originale
  - si comincia con una ricerca binaria del K massimo
  - trovatolo, si fa una ricerca binaria dei flussi riducendo progressivamente le capacità dei singoli archi (ancora ricerca binaria)
  - si cercano le capacità minime che non riducono il valore K ottimo del flusso
  - queste costituiscono un flusso massimo della rete originale
  - l'esempio 10.4 del libro mostra come fare per il *Traveling Salesman problem* [da studiare a casa]

Esercitazione 2

Esercitazione 2

Trovare un flusso massimo per l'esempio

- io faccio da oracolo (l'algoritmo che dice se esiste flusso $\geq K$)
- voi dovete farmi le domande "giuste"

Max Flow

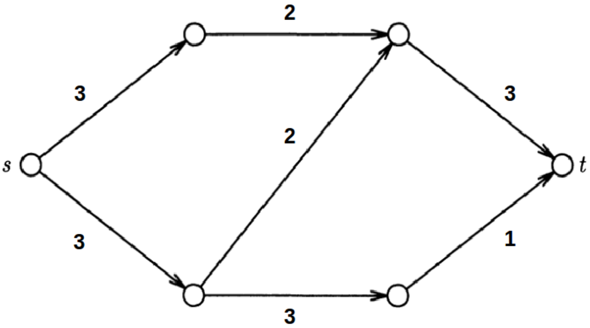
Soluzione del problema di decisione - V0.0

- Prendere un flusso f qualsiasi (ad es. tutto 0)
- Finchè non è ottimo, migliorarlo progressivamente
  - se f non ottimo allora esiste f' migliore
  - sia $\Delta f = f' - f$; il suo valore è > 0
  - però per qualche arco potrebbe essere $\Delta f(i,j) < 0$ es.
  - rappresentiamolo aggiungendo un arco in direzione inversa e settando $c(j,i) = f(i,j)$ (perchè $f'(i,j) < f(i,j)$)
  - quindi f è subottimo sse esiste un flusso positivo Δf in una rete N' ottenuta da N secondo queste linee guida es.
    - vedere i dettagli nel libro
  - ma esiste un flusso positivo in una rete con capacità positive sse t è raggiungibile da s con archi $\neq 0$ (REACHABILITY!)

Esercitazione 3

Esercitazione 3

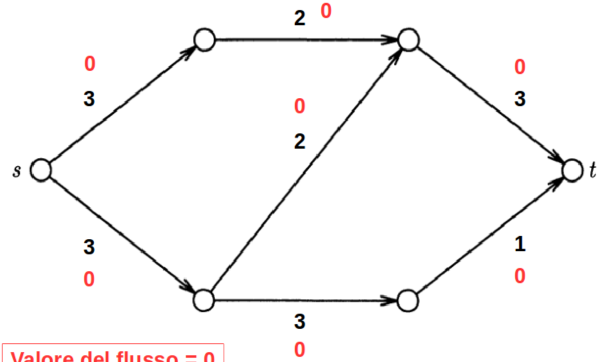
Applicare il procedimento 0.0 a questa rete:



Esercitazione 3

Esercitazione 3

Applicare il procedimento 0.0 a questa rete:

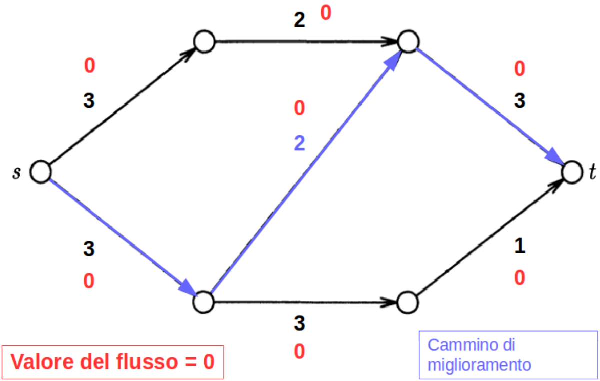


Valore del flusso = 0

Esercitazione 3

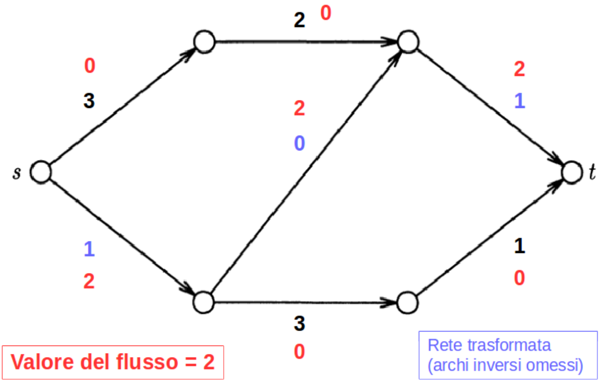
Esercitazione 3

Applicare il procedimento 0.0 a questa rete:



Esercitazione 3

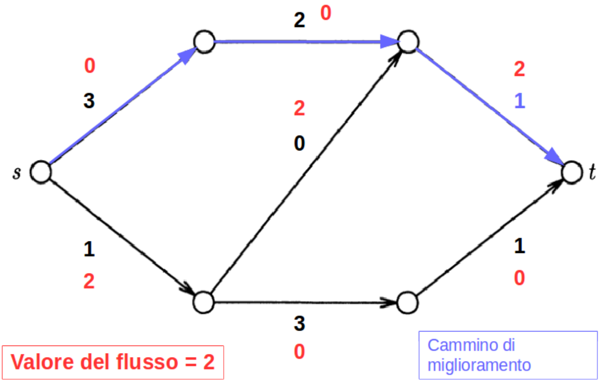
Applicare il procedimento 0.0 a questa rete:



Esercitazione 3

Esercitazione 3

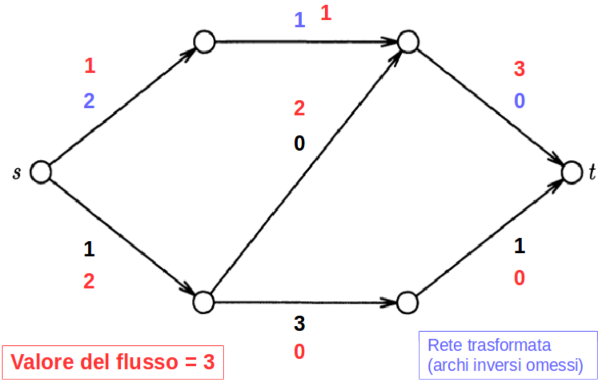
Applicare il procedimento 0.0 a questa rete:



Esercitazione 3

Esercitazione 3

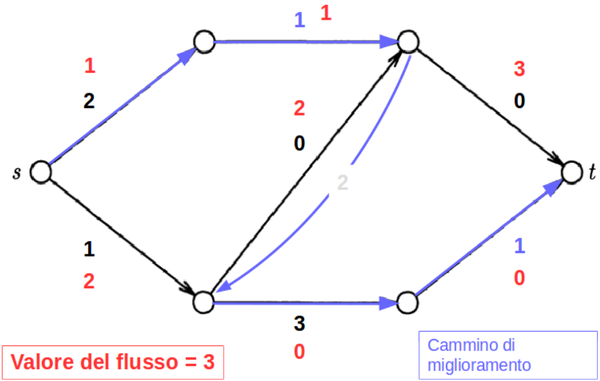
Applicare il procedimento 0.0 a questa rete:



Esercitazione 3

Esercitazione 3

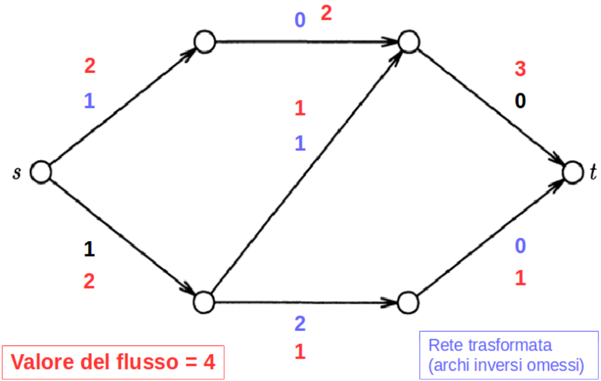
Applicare il procedimento 0.0 a questa rete:



Esercitazione 3

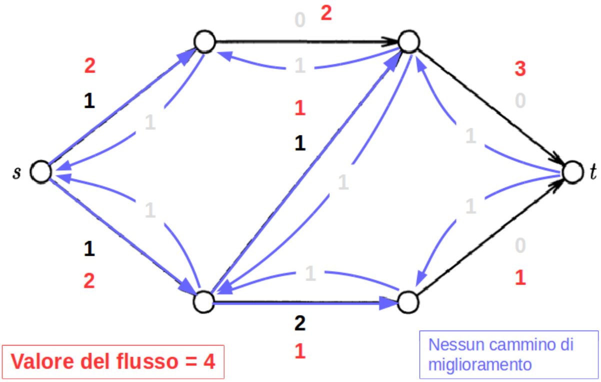
Esercitazione 3

Applicare il procedimento 0.0 a questa rete:



Esercitazione 3

Applicare il procedimento 0.0 a questa rete:



Max Flow

Complessità della versione 0.0

- ad ogni miglioramento ($f \rightsquigarrow f'$) il valore del flusso cresce almeno di 1
- il valore del flusso non può superare nC dove C è la massima capacità degli archi in E e $n = |V|$
  - perchè da s escono al massimo n archi di capacità $\leq C$
- considerando il costo di ogni miglioramento (modifica della rete + $\textsc{Reachability}$) si ottiene tempo $O(n^3 C)$
- vedete qualche problema?...

Max Flow

Attenzione alle misure e all'encoding

- le capacità sono rappresentate in notazione posizionale
  - user interface: in decimale
  - internamente: in binario
  - la rappresentazione di un intero C è lunga solo $\log_{base} C$
- il valore C è esponenzialmente più grande del suo encoding!
- quindi la versione 0.0 non è polinomiale
- esiste una versione più furba di costo $O(n^5)$
  - dettagli sul libro
- Nota: la versione 0.0 sarebbe polinomiale se la codifica dei numeri fosse *unaria*
  - *ma non sarebbe un encoding ragionevole*
  - *la codifica influisce sulla misura della complessità*

Max Flow

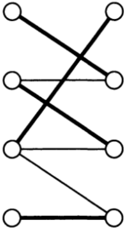
How about space?

- Lo spazio richiesto per memorizzare il flusso corrente è $O(n^2)$
  - perchè gli archi sono in numero $O(n^2)$ nel caso peggiore
- Tanto spazio, ma non sembra possibile evitarlo...
  - questione discussa nel Capitolo 16 del libro
- Questo costo domina quello del resto dell'algoritmo

Bipartite matching

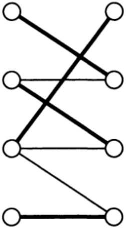
Grafi bipartiti

- Un grafo bipartito è una tripla $B = (U, V, E)$ dove
  - U, V sono i nodi
  - $|U| = |V| = n$
  - $E \subseteq U \times V$



- Un (perfect) matching è un insieme $M \subseteq E$
  - con esattamente n archi
  - se $(u, v) \in M$ e $(u', v') \in M$ allora $u \neq u'$ e $v \neq v'$

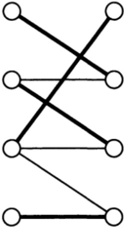
- Dato un grafo bipartito B
- B ha un matching?

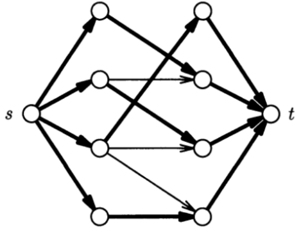


Bipartite matching

Soluzione del problema

- Soluzione per riduzione a MAX FLOW (versione decisionale)
  - costruire una rete N con nodi $U \cup V \cup \{s, t\}$
  - e archi $E \cup \{(s, u) \mid u \in U\} \cup \{(v, t) \mid v \in V\}$
  - tutte le capacità $= 1$





- Chiaramente B ha un matching sse N ha flusso di valore n

Riduzioni

Caratteristiche generali

- Risolvono un problema P concatenando un algoritmo A_1
  - che traduce le istanze di P in istanze di Q

  con un algoritmo A_2 che risolve Q
- Proprietà importanti
  - correttezza della traduzione (la risposta ad ogni istanza I di P deve essere uguale alla risposta dell'istanza $A_1(I)$ di Q)
  - la traduzione A_1 non deve aumentare troppo il costo di A_2: vogliamo che impieghi tempo polinomiale
  - così se Q è risolubile in tempo polinomiale allora anche la soluzione a P lo è (i polinomi sono chiusi rispetto alla somma)
- Vedremo che le riduzioni hanno un ruolo fondamentale nello studio delle classi di complessità
  - e della complessità intrinseca dei problemi

The Traveling Salesman Problem (TSP)

Definizione del problema

- Date n città $1, \ldots, n$
- e date le distanze $d_{ij} \geq 0$ tra ogni coppia di città i e j $(d_{ij} = d_{ji})$

- trovare lo shortest tour (giro più corto) delle città, ovvero
  - una permutazione π di $1, \ldots, n$
    - funzione bigettiva $\{1, \ldots, n\} \to \{1, \ldots, n\}$
  - che minimizza il costo qui sotto (dove $\pi(n+1) = \pi(1)$)

$$\sum_{i=1}^{n} d_{\pi(i),\pi(i+1)}$$

The TSP

Corrispondente problema di decisione

- Il problema di decisione associato a TSP è
- Date n città, le distanze $d_{i,j}$ e un intero B
  - (il *budget* del commesso viaggiatore)
- dire se esiste un tour di costo $\leq B$
  - ovvero una permutazione π di $1, \ldots, n$ tale che

$$\sum_{i=1}^{n} d_{\pi(i),\pi(i+1)} \leq B$$

The TSP

Soluzione naive

- Enumerare tutte le permutazioni e calcolarne il costo
- poi selezionare una delle migliori
- Lo spazio richiesto è solo $O(n)$
  - dobbiamo memorizzare la permutazione in esame + quella ottima trovata fino a quel punto
- Il tempo invece non è polinomiale: $O(n!)$
  - vi sono $n!$ permutazioni
  - considerando il punto di partenza fissato (diciamo 1) si riducono a $(n-1)!$
  - se evitiamo di considerare i tour inversi diventano $\frac{1}{2}(n-1)!$
  - ma sono sempre troppe: per $n > 3$

$$n! = 1 \cdot 2 \cdots n > \underbrace{2 \cdot 2 \cdots 2}_{n} = 2^n$$

The TSP

E la sua complessità intrinseca

- Nonostante anni di ricerche non sono mai stati trovati algoritmi polinomiali (nel tempo) per TSP
  - si può migliorare il tempo $O(n!)$ con tecniche di *dynamic programming*, ma lo spazio richiesto diventa esponenziale!
  - esistono algoritmi euristici che però non garantiscono l'ottimalità
- Dopo tanti fallimenti si è arrivati a congetturare che vi sia un ostacolo fondamentale alla soluzione polinomiale del TSP e problemi "simili"
  - la "similitudine" sarà basata sulle riduzioni
- Questa congettura, formalizzata come $\mathrm{P} \neq \mathrm{NP}$, è anche sfuggita ad anni di tentativi di dimostrare che non esistono algoritmi polinomiali per TSP e problemi "simili"

TSP e “simili”

Complessità intrinseca

- La questione si riduce a determinare se esista o meno
  - un modo “mirato” di esplorare uno spazio non strutturato “grande” (esponenziale), ad es. tutti i *tour*
  - cercando un elemento che abbia certe caratteristiche date
  - esprimibili con condizioni non banali (condizioni booleane, ottimalità di funzioni di costo,...)
  - “mirato” significa che basta esplorare un sottinsieme “piccolo” (polinomiale) di quello spazio

TSP e “simili”

Complessità intrinseca

- La questione si riduce a determinare se esista o meno
  - un modo “mirato” di esplorare uno spazio non strutturato “grande” (esponenziale), ad es. tutti i *tour*
  - cercando un elemento che abbia certe caratteristiche date
  - esprimibili con condizioni non banali (condizioni booleane, ottimalità di funzioni di costo,...)
  - “mirato” significa che basta esplorare un sottinsieme “piccolo” (polinomiale) di quello spazio
- La questione non riguarda solo P e NP: c’è una quantità infinita di classi di complessità che non sappiamo se siano uguali o diverse per lo stesso motivo
  - ad es. la gerarchia polinomiale

TSP e "simili"

Complessità intrinseca

- La questione si riduce a determinare se esista o meno
  - un modo "mirato" di esplorare uno spazio non strutturato "grande" (esponenziale), ad es. tutti i *tour*
  - cercando un elemento che abbia certe caratteristiche date
  - esprimibili con condizioni non banali (condizioni booleane, ottimalità di funzioni di costo,...)
  - "mirato" significa che basta esplorare un sottinsieme "piccolo" (polinomiale) di quello spazio
- La questione non riguarda solo P e NP: c'è una quantità infinita di classi di complessità che non sappiamo se siano uguali o diverse per lo stesso motivo
  - ad es. la gerarchia polinomiale
- Non bisogna confondere TSP e fratelli con i problemi intrinsecamente *esponenziali*: per quelli <u>sappiamo</u> che non esistono algoritmi polinomiali

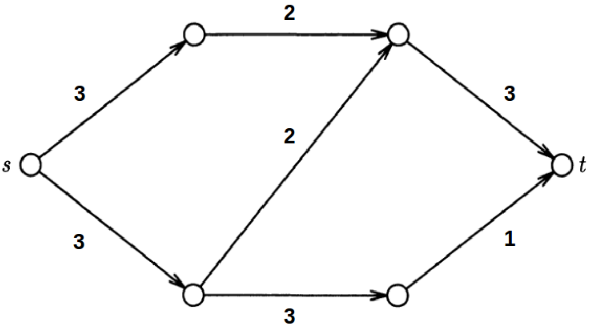
Capitolo di riferimento

Papadimitriou

- Parte I, Capitolo 1, tutti i paragrafi

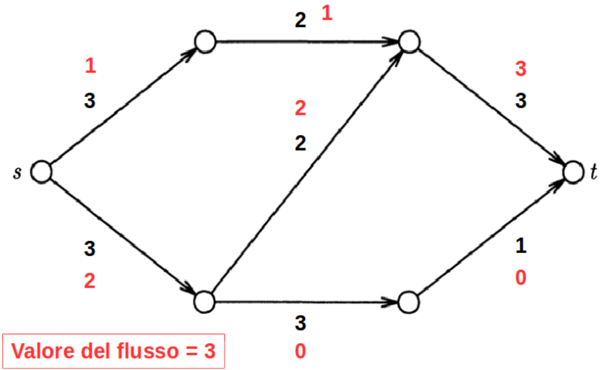
Esempio

Miglioramento di flusso che riduce il flusso di un singolo arco a vantaggio di un altro



Esempio

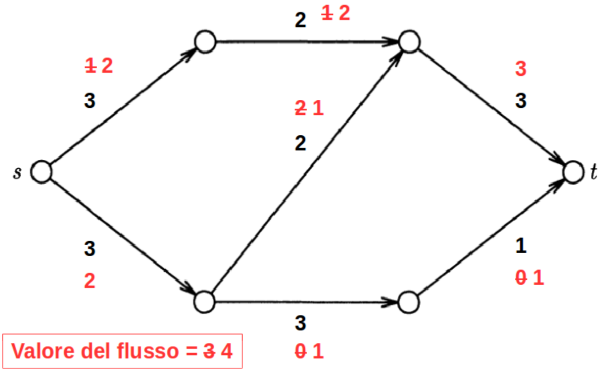
Miglioramento di flusso che riduce il flusso di un singolo arco a vantaggio di un altro



Valore del flusso = 3

Esempio

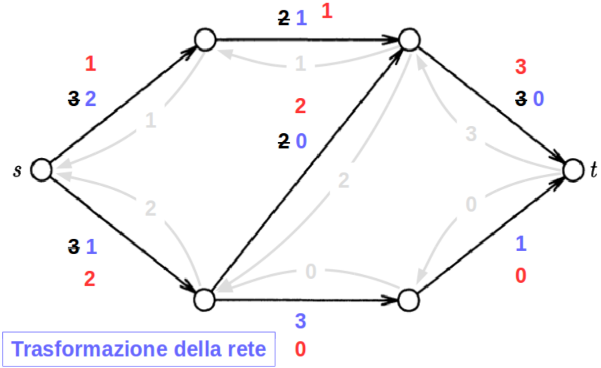
Miglioramento di flusso che riduce il flusso di un singolo arco a vantaggio di un altro



Valore del flusso = ~~3~~ 4

Esempio

Trasformazione della rete per risolvere il problema di decisione



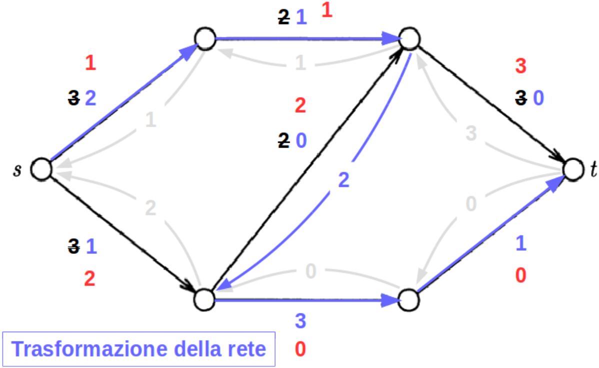
Trasformazione della rete

In rosso il flusso f dato, in grigio e blu i flussi rimanenti

back

Esempio

Trasformazione della rete per risolvere il problema di decisione



Trasformazione della rete

In rosso il flusso f dato, in grigio e blu i flussi rimanenti

back