

3.24pt

# Complessità computazionale



## Problemi difficili e nondeterminismo

Piero A. Bonatti

Università di Napoli Federico II

Laurea Magistrale in Informatica

# Tema della lezione

- Parleremo spesso di problemi che sono “difficili” in due sensi:  
*computazionale*
  - tutti gli algoritmi noti che li risolvono richiedono tempo (almeno) esponenzialee *teorico*: l'analisi del problema è difficile; non si riesce a dimostrare
  - nè che esistono algoritmi polinomiali
  - nè che non ne esistono

# Tema della lezione

- Abbiamo già incontrato un problema simile: il TSP
- Ma non è il solo: abbiamo anche
  - La **fattorizzazione** degli interi
  - Il problema di decidere se una formula della logica proposizionale è **soddisfacibile**
  - In uno scenario dove alcune aziende posseggono quote di altre aziende: decidere se una **azienda è strategica**
    - appartiene a un sottinsieme minimo delle controllate sufficiente a produrre tutti i prodotti attualmente venduti

# Tema della lezione

- Alcune domande da porci per capire meglio questi problemi:
  - hanno tutti la stessa complessità?
  - hanno tutti la stessa struttura?
  - è possibile uno schema di soluzione comune?
- In realtà TSP e soddisfacibilità hanno la stessa complessità e struttura simile
- Mentre è ragionevole congetturare che la fattorizzazione sia leggermente più facile e il problema delle *strategic companies* strettamente più difficile
- Durante il corso arriveremo a tracciare una mappa delle inclusioni tra molte diverse classi di complessità
- e qualche risultato di separazione stretta tra queste classi
- visto che sono pochi, analizzeremo cosa cambierebbe se alcune di quelle classi coincidessero

# Tema della lezione

- Alcune domande da porci per capire meglio questi problemi:
  - hanno tutti la stessa complessità?
  - hanno tutti la stessa struttura?
  - è possibile uno schema di soluzione comune?
- In realtà TSP e soddisfacibilità hanno la stessa complessità e struttura simile
- Mentre è ragionevole congetturare che la fattorizzazione sia leggermente più facile e il problema delle *strategic companies* strettamente più difficile
- Durante il corso arriveremo a tracciare una mappa delle inclusioni tra molte diverse classi di complessità
- e qualche risultato di separazione stretta tra queste classi
- visto che sono pochi, analizzeremo cosa cambierebbe se alcune di quelle classi coincidessero

# Tema della lezione

- Alcune domande da porci per capire meglio questi problemi:
  - hanno tutti la stessa complessità?
  - hanno tutti la stessa struttura?
  - è possibile uno schema di soluzione comune?
- In realtà TSP e soddisfacibilità hanno la stessa complessità e struttura simile
- Mentre è ragionevole congetturare che la fattorizzazione sia leggermente più facile e il problema delle *strategic companies* strettamente più difficile
- Durante il corso arriveremo a tracciare una mappa delle inclusioni tra molte diverse classi di complessità
- e qualche risultato di separazione stretta tra queste classi
- visto che sono pochi, analizzeremo cosa cambierebbe se alcune di quelle classi coincidessero

# Tema della lezione

- Alcune domande da porci per capire meglio questi problemi:
  - hanno tutti la stessa complessità?
  - hanno tutti la stessa struttura?
  - è possibile uno schema di soluzione comune?
- In realtà TSP e soddisfacibilità hanno la stessa complessità e struttura simile
- Mentre è ragionevole congetturare che la fattorizzazione sia leggermente più facile e il problema delle *strategic companies* strettamente più difficile
- Durante il corso arriveremo a tracciare una mappa delle inclusioni tra molte diverse classi di complessità
- e qualche risultato di separazione stretta tra queste classi
- visto che sono pochi, analizzeremo cosa cambierebbe se alcune di quelle classi coincidessero



# Tema della lezione

- Tutti i problemi citati hanno una caratteristica in comune
  - devono cercare un elemento in uno spazio (almeno) esponenzialmente grande
    - tour, sequenze di interi, interpretazioni proposizionali, ...
  - controllare se un elemento ha le proprietà desiderate è “facile” (richiede tempo polinomiale)
- Il problema, a livello teorico, è che da quando esiste l'informatica non si riesce a dimostrare
  - nè che bisogna per forza esplorare una porzione “grande” (esponenziale) dello spazio di ricerca
  - nè che c'è un modo furbo per ignorare la maggior parte dello spazio di ricerca e limitarsi a considerare “pochi” candidati (polinomiali)
    - senza perdere soluzioni, naturalmente

# Tema della lezione

- Tutti i problemi citati hanno una caratteristica in comune
  - devono cercare un elemento in uno spazio (almeno) esponenzialmente grande
    - tour, sequenze di interi, interpretazioni proposizionali, ...
  - controllare se un elemento ha le proprietà desiderate è “facile” (richiede tempo polinomiale)
- Il problema, a livello teorico, è che da quando esiste l'informatica non si riesce a dimostrare
  - nè che bisogna per forza esplorare una porzione “grande” (esponenziale) dello spazio di ricerca
  - nè che c'è un modo furbo per ignorare la maggior parte dello spazio di ricerca e limitarsi a considerare “pochi” candidati (polinomiali)
    - senza perdere soluzioni, naturalmente

# Tema della lezione

- Come facciamo ad analizzare algoritmicamente quei problemi difficili
- senza sapere se e come si può esplorare lo spazio di ricerca “velocemente”?
- Bisognerebbe “scorporare” dall’analisi il problema della ricerca e analizzare il resto
- Trattando la ricerca come una “scatola nera” otteniamo una stima del consumo di risorse necessario *relativa* al costo della ricerca
- Con la stessa idea riusciremo anche a confrontare problemi diversi ed evidenziarne somiglianze e differenze strutturali

# Tema della lezione

- Come facciamo ad analizzare algoritmicamente quei problemi difficili
- senza sapere se e come si può esplorare lo spazio di ricerca “velocemente”?
- Bisognerebbe “scorporare” dall’analisi il problema della ricerca e analizzare il resto
- Trattando la ricerca come una “scatola nera” otteniamo una stima del consumo di risorse necessario *relativa* al costo della ricerca
- Con la stessa idea riusciremo anche a confrontare problemi diversi ed evidenziarne somiglianze e differenze strutturali

# Tema della lezione

- Come facciamo ad analizzare algoritmicamente quei problemi difficili
- senza sapere se e come si può esplorare lo spazio di ricerca “velocemente”?
- Bisognerebbe “scorporare” dall’analisi il problema della ricerca e analizzare il resto
- Trattando la ricerca come una “scatola nera” otteniamo una stima del consumo di risorse necessario *relativa* al costo della ricerca
- Con la stessa idea riusciremo anche a confrontare problemi diversi ed evidenziarne somiglianze e differenze strutturali

# Tema della lezione

- Formalmente, il costo della esplorazione dello spazio di ricerca verrà scorporato nel caso più semplice mediante
  - algoritmi *nondeterministici*
  - formalizzati con *MdT nondeterministiche*
- Questi due argomenti sono il tema centrale di questa lezione

# Pseudocodice

- Abbiamo già formulato algoritmi *parzialmente specificati*
- mediante pseudocodice
- alcune operazioni non elementari sono indicate ma non dettagliate
- come nell'algoritmo per REACHABILITY
  - la gestione dell'insieme di nodi da elaborare poteva essere FIFO, LIFO, random...
  - risultato: diverse modalità di visita del grafo

# 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")

```
1 for all  $v \in V$  do marked[ $v$ ] = false
2 marked[ $x$ ] = true
3  $S := \{x\}$ 
4 while  $S \neq \emptyset$  do
5     estrai  $v$  da  $S$ 
6     forall  $(v, w) \in E$  do
7         if not marked[ $w$ ] do marked[ $w$ ] = true; inserisci  $w$  in  $S$ 
8     end
9 end
10 return marked[ $y$ ]
```

---



# Algoritmi nondeterministici

- Negli algoritmi nondeterministici si fa qualcosa di simile
- alcune istruzioni operano delle scelte senza dire come

---

**Algorithm 2:** Nondeterministic algorithm for TSP (decision version)

---

**Input:**  $n$  (num. città),  $d_{ij}$  e budget  $B$

**Output:** *true* sse esiste un tour di costo  $\leq B$

```
1 i=1; tour[i++] = 1; cost = 0
2  $V = \{2, \dots, n\}$  // città da visitare
3 while  $V \neq \emptyset$  do
4     choose  $c \in V$ ;  $V = V \setminus \{c\}$ 
5     tour[i++] = c; cost +=  $d_{tour[i-1],c}$ 
6 end
7 cost +=  $d_{tour[n],tour[1]}$ 
8 return cost  $\leq B$ 
```

---

# Algoritmi nondeterministici

- Nell'algoritmo precedente **choose** opera scelte *giuste*
  - se esistono tour di costo  $\leq B$ , **choose** seleziona proprio i membri di uno di quei tour nell'ordine giusto
- L'analisi di complessità considera **choose** come una operazione elementare (singolo passo, tempo=1)
  - scorporando così il costo dell'esplorazione dello spazio di ricerca
- L'algoritmo precedente risulta *polinomiale a meno del costo della ricerca*
  - il tour “candidato” può essere generato in tempo polinomiale
    - il costo della ricerca è stato scorporato
  - verificare se il candidato ha la proprietà desiderata ( $\text{costo} \leq B$ ) ha costo polinomiale

# MdT nondeterministiche

## Definizione

- Formalizzano gli algoritmi nondeterministici
- Sono 4-uple  $M = (K, \Sigma, \Delta, s)$  dove  $\Delta$  non è più una funzione

$$K \times \Sigma \rightarrow (K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$$

bensì una *relazione*

$$\Delta \subset K \times \Sigma \times (K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$$

- Quindi per ogni coppia  $(q, \sigma)$  può esistere *più di una transizione*
  - è l'analogo di **choose**
- Versione a  $k$  nastri definita nel modo ovvio

# MdT nondeterministiche

## Definizione

- Formalizzano gli algoritmi nondeterministici
- Sono 4-uple  $M = (K, \Sigma, \Delta, s)$  dove  $\Delta$  non è più una funzione

$$K \times \Sigma \rightarrow (K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$$

bensì una *relazione*

$$\Delta \subset K \times \Sigma \times (K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$$

- Quindi per ogni coppia  $(q, \sigma)$  può esistere *più di una transizione*
  - è l'analogo di **choose**
- Versione a  $k$  nastri definita nel modo ovvio

# MdT nondeterministiche

## Configurazioni e transizioni

- Le configurazioni sono come quelle delle MdT:  $(q, w, u)$
- Le transizioni singole sono definite analogamente:
- Siano  $w = w_1 \cdots w_m$  e  $u = u_1 \cdots u_n$  stringhe in  $\Sigma^*$
- Scriviamo  $(q, w, u) \xrightarrow{M} (q', w', u')$  sse
  - $(q, w_m, r, \sigma, -) \in \Delta$  e  $(q', w', u') = (r, w_1 \cdots w_{m-1} \sigma, u)$
  - $(q, w_m, r, \sigma, \leftarrow) \in \Delta$  e  $(q', w', u') = (r, w_1 \cdots w_{m-1}, \sigma u)$
  - $(q, w_m, r, \sigma, \rightarrow) \in \Delta$  e  $(q', w', u') = (r, w_1 \cdots w_{m-1} \sigma u_1, u_2 \cdots u_n)$ 
    - se  $u \neq \epsilon$ , altrimenti
  - $(q, w_m, r, \sigma, \rightarrow) \in \Delta$  e  $(q', w', u') = (r, w_1 \cdots w_{m-1} \sigma \sqcup, \epsilon)$

# MdT nondeterministiche

## Transizioni in più passi

- Ovviamente, adesso che la MdT può scegliere tra diverse mosse in una data configurazione,  $\xrightarrow{M}$  *non è più una funzione*
  - possiamo avere  $(q, w, u) \xrightarrow{M} (q_i, w_i, u_i)$  per  $i = 1, \dots, c$   
( $c \leq |\Delta|$ )
- Le transizioni  $\xrightarrow{M^k}$  e  $\xrightarrow{M^*}$  si definiscono come prima a partire da  $\xrightarrow{M}$
- Neanche esse sono funzioni: ogni input  $x$  può corrispondere a diverse computazioni  $(s, \triangleright, x) \xrightarrow{M^*} (H_i, w_i, u_i)$ 
  - che possono terminare in modo diverso (alcune con "yes", altre con "no", altre con  $h$ )
  - alcune potrebbero addirittura non terminare
  - corrispondono a tutti i modi di eseguire i **choose**

# Esempio

- Una MdT che formalizzasse l'algoritmo nondeterministico per il TSP...
- ...avrebbe una computazione (o *run*) per ogni tour che parte da 1
- in generale, alcuni avranno costo  $\leq B$ , altri no.
- Se almeno un run trova un tour di costo  $\leq B$  dobbiamo rispondere "yes"
  - qualunque ricerca "corretta", che non perde soluzioni, dovrebbe trovare uno di quei tour
- Solo se tutti i run costruiscono tour di costo  $> B$  dobbiamo rispondere "no"

# MdT nondeterministiche

## Decisione di linguaggi

### Definizione (decisione)

Una MdT nondeterministica  $N$  *decide* un linguaggio  $L$  sse, per ogni input  $x$ :

$$x \in L \Leftrightarrow (s, \triangleright, x) \xrightarrow{M^*} ("yes", w, u)$$

(per qualche stringa  $wu$ )

In altre parole,  $x \in L$  sse almeno un run di  $N$  accetta  $x$



# MdT nondeterministiche

## Tempo impiegato

### Definizione

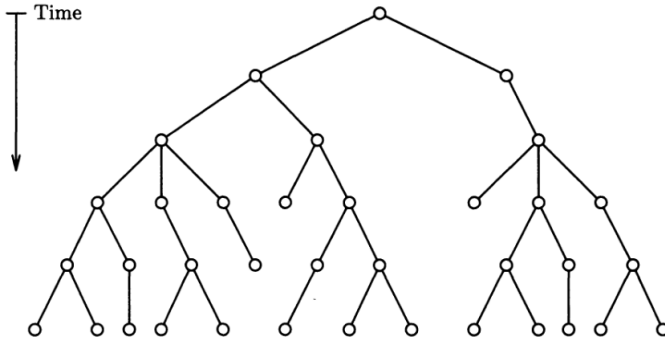
Una MdT nondeterministica  $N$  decide  $L$  in tempo  $f(n)$  sse

- $N$  decide  $L$ , e
- per ogni input  $x$ , se  $(s, \triangleright, x) \xrightarrow{M^k} (q, w, u)$ , allora  $k \leq f(|x|)$

Ovvero, nessun run è più lungo di  $f(|x|)$

# MdT nondeterministiche

Rappresentazione ad albero delle computazioni



# Esercitazione

Scrivere una MdT nondeterministica a 2 nastri che riconosce le stringhe non palindrome

Suggerimento:

- scrivere sul secondo nastro un numero  $c \leq |x|$  scelto nondeterministicamente (in unario, per fare le cose più semplici)
  - il modo più semplice è scandire l'input e ad ogni passo scegliere nondeterministicamente se aggiungere un '1' al secondo nastro oppure no
- spostare la testina sul carattere dell'input in posizione  $|x| - c$  e memorizzare il carattere nello stato corrente
  - per spostarsi di  $c$  caratteri, far scorrere la testina indietro sul secondo nastro fino all'inizio
- spostare la testina sul carattere dell'input in posizione  $c$  e confrontare il carattere in quella posizione con quello memorizzato nello stato

# Nondeterminismo $\neq$ parallelismo

- Il modello nondeterministico non è adatto al parallelismo
  - Il numero di processori necessario sarebbe illimitato
  - La misura del tempo dovrebbe essere la lunghezza della computazione *più breve* che accetta l'input
    - mentre abbiamo adottato essenzialmente la lunghezza *massima*
  - Manca la sincronizzazione tra processi

# Le classi di complessità **NTIME**, **P** ed **NP**

(Sono insiemi di linguaggi  $L \subseteq (\Sigma \setminus \{\sqcup\})^*$ )

## Definizione (nondeterministic time)

$L \in \text{NTIME}(f(n))$  sse  $L$  è deciso da una MdT nondeterministica in tempo  $f(n)$

- Le classi dei problemi risolvibili in tempo polinomiale deterministico e nondeterministico
  - $P = \bigcup_{k>0} \text{TIME}(n^k)$
  - $NP = \bigcup_{k>0} \text{NTIME}(n^k)$

# Prime relazioni tra **P** ed **NP**

## Proposizione

$$P \subseteq NP$$

- Prova: È sufficiente notare che le MdT deterministiche sono casi particolari di MdT nondeterministiche in cui  $\Delta$  è una funzione (associa ogni coppia in  $K \times \Sigma$  ad una unica tripla).

QED

Viceversa, nessuno è mai riuscito a dimostrare nè  $NP \subseteq P$ , (che implicherebbe  $P = NP$ ) nè  $NP \neq P$ .

# TSP è in NP

- Descriviamo (cenni) una MdT nondeterministica  $N$  a 2 nastri che approssimativamente corrisponde all'algoritmo nondeterministico per TSP visto prima
- Scrive sul 2° nastro una arbitraria stringa di simboli lunga al massimo quanto l'input
  - usando scelte nondeterministiche (analoghe a **choose**)
- Poi torna indietro e controlla se la stringa codifica un tour (potrebbe non avere senso)
  - in caso di errore si ferma con "no"
- Infine calcola il costo del tour. Se è  $\leq B$  termina in "yes", altrimenti in "no"
- Poichè i nastri sono ad accesso sequenziale, le fasi qui sopra richiedono  $O(n^2)$  passi

# TSP è in NP (II)

## Correttezza

- Sia  $x$  l'input di  $N$
  - Se  $N$  ha un run che termina con "yes", allora ha trovato un tour di costo  $\leq B$ , quindi  $x \in \text{TSP}$
  - Viceversa, se  $x \in \text{TSP}$ , allora esiste un tour di costo  $\leq B$ , ed esiste una sequenza di scelte che scrive proprio quel tour sul secondo nastro.
    - è importante che  $N$  sia programmata in modo da poter scrivere almeno *tutte* le stringhe che rappresentano un tour
- Siccome la lunghezza del tour è  $\leq B$ ,  $N$  termina con "yes" e accetta  $x$
- Siccome  $N$  accetta  $x$  sse  $x \in \text{TSP}$ ,  $N$  decide TSP



# TSP è in NP (III)

- Il teorema di speedup quadratico vale anche per le MdT nondeterministiche:
  - quindi esiste una MdT nondeterministica a 1 nastro che simula  $N$  con una perdita di efficienza quadratica
- Si conclude che TSP è deciso da una MdT nondeterministica in tempo  $O(n^4)$ , quindi  $\text{TSP} \in \text{NP}$

QED

# Da pseudocodice nondeterministico ad algoritmi deterministici

- Se lavorassimo con automi a stati finiti, potremmo trasformare ogni algoritmo nondeterministico in uno deterministico che risolve lo stesso problema nello stesso tempo
  - il costo è una crescita esponenziale dell'automa, ricordate?
- Invece con le MdT non sappiamo come fare (altrimenti avremmo risolto la questione  $P \stackrel{?}{=} NP$ )
  - il “trucco” usato per gli automi a stati finiti qui non funziona perchè l'insieme delle possibili configurazioni è infinito
  - l'ampiezza dell'albero delle computazioni può crescere senza limiti al crescere di  $|x|$
- Sicuramente c'è un modo per trasformare ogni algoritmo nondeterministico in uno deterministico che risolve lo stesso problema *con un aumento esponenziale del tempo*

# Da pseudocodice nondeterministico ad algoritmi deterministici

- Se lavorassimo con automi a stati finiti, potremmo trasformare ogni algoritmo nondeterministico in uno deterministico che risolve lo stesso problema nello stesso tempo
  - il costo è una crescita esponenziale dell'automa, ricordate?
- Invece con le MdT non sappiamo come fare (altrimenti avremmo risolto la questione  $P \stackrel{?}{=} NP$ )
  - il “trucco” usato per gli automi a stati finiti qui non funziona perchè l'insieme delle possibili configurazioni è infinito
  - l'ampiezza dell'albero delle computazioni può crescere senza limiti al crescere di  $|x|$
- Sicuramente c'è un modo per trasformare ogni algoritmo nondeterministico in uno deterministico che risolve lo stesso problema *con un aumento esponenziale del tempo*

# Da pseudocodice nondeterministico ad algoritmi deterministici

- Se lavorassimo con automi a stati finiti, potremmo trasformare ogni algoritmo nondeterministico in uno deterministico che risolve lo stesso problema nello stesso tempo
  - il costo è una crescita esponenziale dell'automa, ricordate?
- Invece con le MdT non sappiamo come fare (altrimenti avremmo risolto la questione  $P \stackrel{?}{=} NP$ )
  - il “trucco” usato per gli automi a stati finiti qui non funziona perchè l'insieme delle possibili configurazioni è infinito
  - l'ampiezza dell'albero delle computazioni può crescere senza limiti al crescere di  $|x|$
- Sicuramente c'è un modo per trasformare ogni algoritmo nondeterministico in uno deterministico che risolve lo stesso problema *con un aumento esponenziale del tempo*

# Da nondeterministico a deterministico

## Teorema

- Se  $L$  è deciso da una MdT nondeterministica  $N$  in tempo  $f(n)$
- Allora è deciso da una MdT deterministica  $M$  a 3 nastri
- in tempo  $O(c^{f(n)})$ , dove la costante  $c > 1$  dipende da  $N$

Equivalentemente:

$$\text{NTIME}(f(n)) \subseteq \bigcup_{c>1} \text{TIME}(c^{f(n)})$$

# Da nondeterministico a deterministico

## Prova

- Sia  $N = (K, \Sigma, \Delta, s)$  la MdT nondeterministica data
- per ogni  $(q, \sigma) \in K \times \Sigma$ , sia

$$C_{q,\sigma} = \{(q', \sigma', D) \mid (q, \sigma, q', \sigma', D) \in \Delta\}$$

l'insieme delle *scelte* (choices) di  $N$  nel contesto  $(q, \sigma)$

- Nota: ogni  $C_{q,\sigma}$  è finito (perchè?)
- Quindi definiamo il *grado* (degree) di nondeterminismo di  $N$

$$d = \max_{q,\sigma} |C_{q,\sigma}|$$

# Da nondeterministico a deterministico

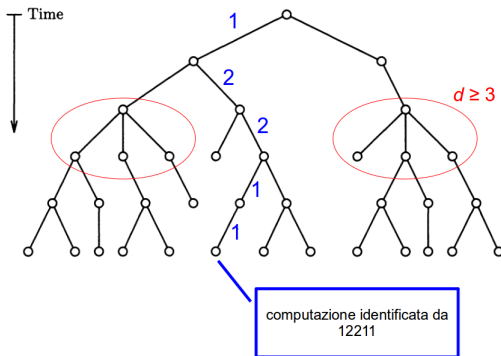
## Prova (II)

- Idea di base per la simulazione:
  - Ogni computazione di  $N$  è determinata da una sequenza di scelte nondeterministiche
  - che ad ogni passo possiamo numerare da 1 a  $d$
  - quindi ogni sequenza di  $t$  passi corrisponde a una sequenza di  $t$  interi compresi tra 1 e  $d$

# Da nondeterministico a deterministico

## Prova (IIb)

- Ogni sequenza di  $t$  passi corrisponde a una sequenza di  $t$  interi compresi tra 1 e  $d$





# Da nondeterministico a deterministico

## Prova (III)

- $M$  considera progressivamente tutte le sequenze di scelte in ordine *lessicografico* finchè scopre di poter terminare  
 $1, 2, \dots, d, 11, 12, \dots, 1d, 21, 22, \dots$
- le sequenze di scelte (*choices*)  $c_1, \dots, c_t$  sono memorizzate sul secondo nastro
- per ciascuna sequenza  $c_1, \dots, c_t$ ,  $M$  simula le transizioni che  $N$  avrebbe fatto con quelle scelte (solo  $t$  passi)
  - usa il 3° nastro per la simulazione; il 1° è di input
- se la simulazione termina con "yes", anche  $M$  accetta
- altrimenti si passa alla sequenza di scelte successiva
  - è come incrementare un intero espresso in base  $d$
  - analogo all'incremento di una cifra binaria (già visto)

# Da nondeterministico a deterministico

## Prova (IV)

- Condizione di terminazione con rigetto:
- Quando tutte le simulazioni di lunghezza  $t$  terminano in uno stato finale  $\neq$  "yes"
  - finchè esistono simulazioni che possono continuare oltre  $t$  passi dobbiamo andare avanti
- Analisi di complessità:
  - numero di simulazioni:  $\sum_{t=1}^{f(n)} d^t = O(d^{f(n)})$
  - costo di ogni simulazione: risulta  $O(n + f(n))$
  - il costo complessivo è il prodotto:  $O(d^{f(n)})$

QED

## Il consumo di memoria delle MdT nondeterministiche

- Le MdT nondeterministiche *con input e output* si definiscono in modo analogo a quelle deterministiche
  - primo nastro: read only; ultimo: write only

### Definizione (spazio utilizzato da MdT nondeterministiche con i/o)

Una MdT nondeterministica con input e output  $N$  decide  $L$  in spazio  $f(n)$  sse

- $N$  decide  $L$
- per ogni input  $x \in (\Sigma \setminus \{\sqcup\})^*$ , se

$$(s, \triangleright, x, \triangleright, \epsilon, \dots, \triangleright, \epsilon) \xrightarrow{N^*} (q, w_1, u_1, \dots, w_k, u_k)$$

$$\text{allora } \sum_{i=2}^{k-1} |w_i u_i| \leq f(|x|)$$

# Impatto del nondeterminismo sullo spazio

- Riprendiamo REACHABILITY
- L'algoritmo deterministico usava spazio lineare
  - le marcature booleane dei nodie si può scendere fino a  $O(\log^2 n)$  (vedere Cap. 7)
- Con un algoritmo nondeterministico si può arrivare a  $O(\log n)$

# REACHABILITY in spazio logaritmico

## L'algoritmo

- Usa 3 nastri
- il secondo contiene il nodo corrente  $i$  (inizialmente 1)
- il terzo il nodo successivo  $j$  del cammino, scelto nondeterministicamente in  $\{1, \dots, n\}$
- se il bit  $(i, j)$  nella matrice di adiacenza ( $1^{\circ}$  nastro) è 0, rigetta con "no"
- se è 1 e  $j = n$ , accetta ("yes")
- altrimenti copia  $j$  nel secondo nastro e ripeti

# REACHABILITY in spazio logaritmico

## Commenti

- Chiaramente, se 1 è connesso a  $n$ , c'è una sequenza di scelte che genera proprio un cammino tra i due nodi
- Alcune delle computazioni potrebbero non terminare
  - non si controlla se si sta percorrendo un ciclo
  - volendo si potrebbe evitare decrementando un contatore inizializzato a  $n$
- Ma la definizione di accettazione permette casi come questo
  - Nel Cap. 7 si dimostra che si possono normalizzare le MdT con spazio limitato per farle terminare sempre
- La cosa importante qui è che le variabili ausiliarie (i nastri 2 e 3) contengono sempre due numeri tra 1 e  $n$  quindi occupano spazio  $\log n$

# Nondeterminismo e spazio

Altri commenti

- Il guadagno in spazio permesso dal nondeterminismo è inferiore a quello in tempo
- Vedremo che non è un caso
- e che il guadagno che vale per REACHABILITY è anche il massimo possibile

# Nondeterminismo e problemi reali I

## Commenti

- Le MdT nondeterministiche non sono un modello di calcolo “realistico”
- Tuttavia permettono di esprimere in modo elegante e conciso (anche se parziale) la soluzione di problemi importanti in AI e ottimizzazione combinatoria
  - Trovare una dimostrazione di una formula logica
  - Trovare una soluzione a un problema di ottimizzazione

Perchè permettono di tralasciare la modalità di ricerca della prova/soluzione



# Nondeterminismo e problemi reali II

## Commenti

- Gli inference engines per il semantic web sono stati progettati essenzialmente iniziando con un algoritmo nondeterministico (*tableaux*)
  - sufficiente a valutare se tempo e spazio usati sono ottimali rispetto alla complessità intrinseca del problema
  - sufficiente a valutare la correttezza dell'algoritmo
- la concreta strategia di ricerca e le relative euristiche possono essere progettate indipendentemente
  - si possono anche realizzare diverse strategie ed euristiche per lo stesso algoritmo
  - confrontarle
  - eventualmente selezionare quella più appropriata in ogni contesto applicativo
  - mantenendo immutato il cuore dell'algoritmo

# Capitolo di riferimento

Papadimitriou

- Parte I, Capitolo 2, paragrafo 2.7