

Corso di Calcolabilità e Complessità
Corso di Laurea in Informatica 2024/2024
Esercizi risolti durante il corso

Stefano Berardi

Includiamo qui le soluzioni di alcuni degli esercizi svolti durante il corso. Una parte di questi esercizi sono presi dal nostro libro di testo: Sipser, Introduction to the Theory of Computation, Second Edition. Per questi esercizi il Sipser non include una soluzione, oppure include solo parte di una soluzione.

Le soluzioni che trovate qui non fanno parte del programma di esame, ma aiutano a capire i concetti fondamentali del corso. Ogni anno svolgiamo **solo una selezione degli esercizi qui inclusi**, mai tutti.

Calcolabilità

1. Divisibilità per 3 di numeri binari
2. Una Macchina di Turing per test di equaglianza (Es. 3.9 Sipser)
3. MT per successore lessicografico
4. MT per il predecessore
5. Enumeratore a partire da una MT
6. Soluzione Problema 3.8.a,b del Sipser
7. Decidibile, positivamente decidibile e negativamente decidibile
8. Macchina di Turing universale
9. Qualche osservazione sulla prova di Diagonalizzazione (Teor. 4.11 del Sipser)
10. Decidibilità dei linguaggi context-free
11. Indecidibilità delle relazioni di equaglianza e ordine tra numeri reali
12. Esercizio 5.3: Esempio del problema della corrispondenza di Post
13. Un algoritmo pcg di ricerca cieca per le soluzioni di PCP
14. Indecidibilità del problema di tassellazione di Wang
15. Sipser: elenco risultati di calcolabilità
16. Problema 5.28: il Teorema di Rice
17. Problemi 5.14,5.15: indecidibilità del problema dello spostamento fallito.

Complessità

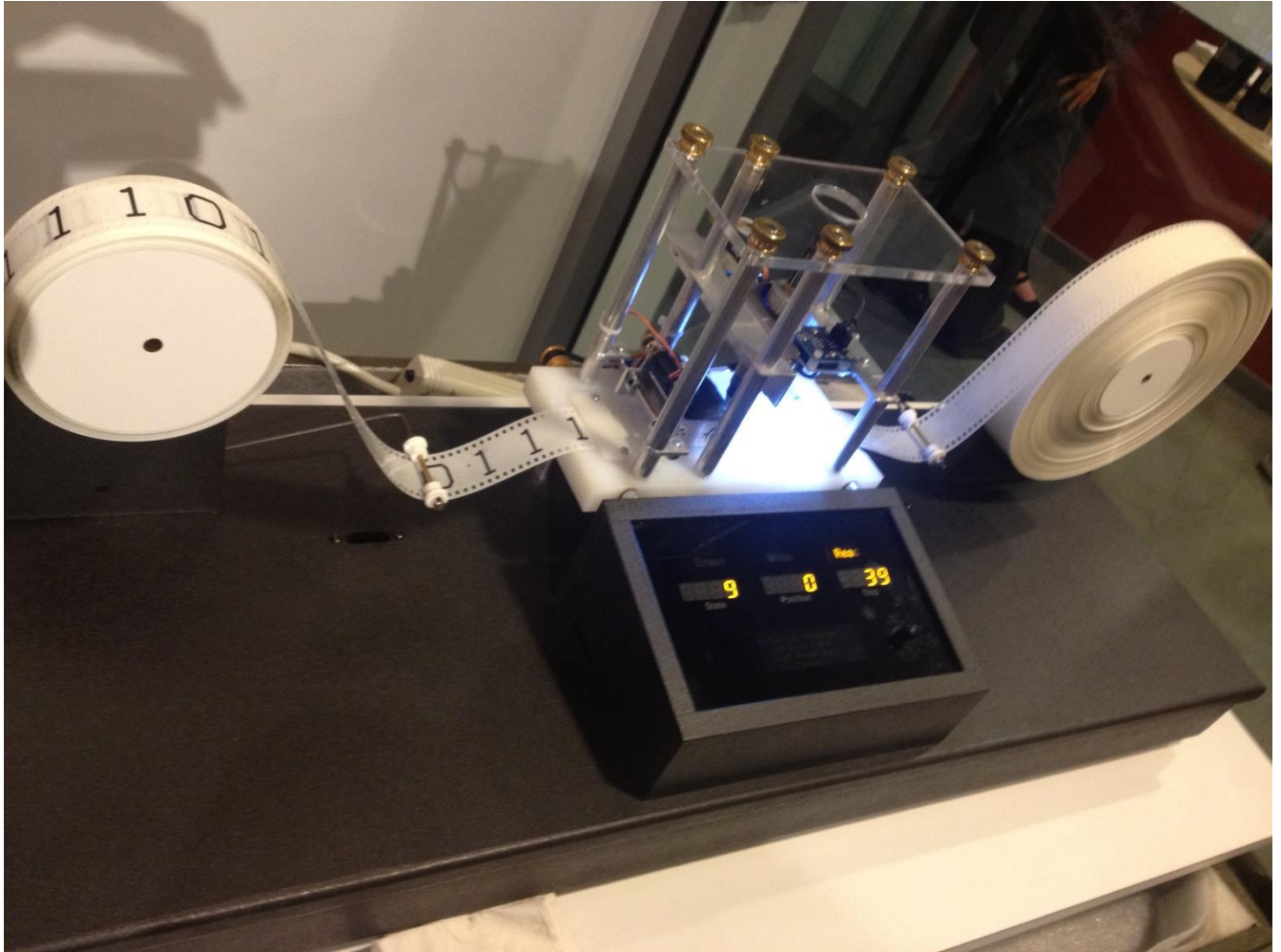
- 18. Forme normali congiuntive: un metodo per ottenerle e il tempo di calcolo richiesto
- 19. Riduzione polinomiale: da SAT a CNF-SAT
- 20. Riduzione polinomiale: da CNF-SAT a 3-SAT
- 21. Breve nota su 3-SAT e VERTEX-COVER
- 22. Problema 7.31: SOLITAIRE
- 23. Problema 7.26: PUZZLE
- 24. Problema 8.15: Un algoritmo polinomiale per trovare una strategia vincente nel gioco HAPPY-CAT
- 25. 3SAT si reduce polinomialmente a 3coloring
- 26. 3coloring si reduce polinomialmente al basic scheduling problem
- 27. Un esempio di Vertex-Covering approssimato

Complessità in spazio (solo cenni nel 2022)

- 28. Raggiungibilità in spazio $O(\log(n)^2)$
- 29. GG, il gioco generalizzato della Geografia, ha una strategia ottimale in PSPACE
- 30. Problema 8.19: il gioco del Nim è LogSpace

Suggerimenti per l' Esame di Calcolabilità e Complessità

Calcolabilità



Ricostruzione di una Macchina di Turing - Wikipedia

Divisibilità per 3 di numeri binari

Questa macchina di Turing M è una versione semplificata di un singolo circuito di una CPU dei computer di oggi. Controlla la divisibilità per 3 di un numero binario, ma non è in grado di controllare la divisibilità per un k qualsiasi.

Descriviamo la macchina M a parole. M passa attraverso le cifre binarie calcolandosi il valore modulo 3 del numero binario appena letto, aggiornandolo non appena legge una nuova cifra 0 oppure 1, e memorizzandolo nello stato della macchina. Lo stato è quindi q_0, q_1, q_2 per indicare che il modulo del numero letto finora vale 0,1,2. All'inizio lo stato vale q_0 , perché abbiamo letto zero cifre, che rappresentano il numero 0, e 0 modulo 3 vale 0. Supponiamo di aver letto 1111 (quindici). 15 modulo 3 vale 0, quindi lo stato è q_0 . Se la prossima cifra è 0, allora il nuovo numero è 11110, ovvero 1111 per 2 più 0, ovvero $15*2+0=30$. Per calcolare 30 modulo 3, osservo che per regole dell'aritmetica modulo 3 abbiamo che: $(15*2+0) \bmod 3 = ((15 \bmod 3)*2 + 0) \bmod 3 = (0*2+0) \bmod 3 = 0 \bmod 3 = 0$. Dunque il prossimo stato è ancora q_0 . Allo stesso modo, dato ogni stato q_0, q_1, q_2 e una cifra 0 oppure 1 posso calcolare il nuovo stato, sempre uno tra q_0, q_1, q_2 : ho bisogno di $3*2 = 6$ regole di transizione. Quando arrivo a un blank, che indica la fine dell'input, se sono nello stato q_0 accetto (il numero mod3 vale 0), altrimenti rifiuto.

La macchina M non torna mai indietro e non sovrascrive mai una cifra letta. Per chi di voi già conosce l'argomento, accenniamo che M potrebbe essere rimpiazzata da un *automa a stati finiti*, ma per la maggior parte delle macchine di Turing questo non è possibile.

Definiamo M attraverso la sua funzione δ . Forniamo una codice sorgente eseguibile. Potete compilare e eseguire la macchina M nel sito

<http://turingmachinesimulator.com>

Inoltre per qualche tempo troverete una copia del sorgente della macchina all'indirizzo:

<https://turingmachinesimulator.com/shared/ggiylsxqke>

Qualche osservazione sulle notazioni: quando $\delta(q,a)=(q',a',R)$ nel sorgente della macchina M usiamo il simbolo > per indicare R (lo spostamento a destra):

q,a
q',a',>

e analogamente per $\delta(q,a)=(q',a',L)$. In questo caso usiamo il simbolo < per indicare L (lo spostamento a sinistra). Indichiamo il blank con _ (underscore), l'assenza di movimento alla fine con "-". Ecco la funzione δ di M, descritta con le convenzioni usate nel sito qui sopra, ed eseguibile nel sito. Tutte le transizioni non elencate portano in uno stato di rifiuto, che non viene mai menzionato.

```
// Alphabet: 0,1
// Tape: _
// Language: { w | w mod3 == 0}

// Example: accepts 110 (=6), refuses 111 (=7)
//
// Divisible by 3 Algorithm
// for Turing Machine Simulator
// turingmachinesimulator.com
//
// ----- States -----|
// q0 - mod3 == 0          |
// q1 - mod3 == 1          |
// q2 - mod3 == 2          |
// qaccept - accepting state |
//-----|
//
//----- Algorithm -----|
// At each step we double the|
// current value of mod3 then|
// we add the current digit  |
// We accept if we read blank|
// and mod3 is currently 0    |
//-----|

name: Binary numbers divisible by 3
init: q0
accept: qAccept
```

```

//0 * 2 + 0 =mod3 0: we move to q0
q0,0
q0,0,>

//0 * 2 + 1 =mod3 1: we move to q1
q0,1
q1,1,>

//1 * 2 + 0 =mod3 2: we move to q2
q1,0
q2,0,>

//1 * 2 + 1 = 3 =mod3 0: we move to q0
q1,1
q0,1,>

//2 * 2 + 0 = 4 =mod3 1: we move to q1
q2,0
q1,0,>

//2 * 2 + 1 = 5 =mod3 2: we move to q2
q2,1
q2,1,>

//we read "_", which is "blank", and the current mod3 is 0: accept
q0,_
qAccept,_,-
// The symbol "-" denotes "no movement"

//any other case: refuse. these cases are not explicitly shown

```

Una Macchina di Turing per test di equaglianza (Es. 3.9 Sipser)

Nel libro di testo Sipser, esempio 3.9, viene descritta attraverso un grafo una macchina di Turin M che esegue il test di equaglianza tra stringhe binarie. Questa macchina corrisponde a un singolo circuito di una CPU dei computer di oggi. Trovate sul Sipser la descrizione della macchina sia a parole che attraverso il grafo degli stati. Come esercizio, qui definiamo la stessa macchina M in modo diverso, attraverso la sua funzione δ . Forniamo una codice sorgente eseguibile. Potete compilare e eseguire la macchina M nel sito

<http://turingmachinesimulator.com>

Inoltre per qualche tempo troverete una copia del sorgente della macchina all'indirizzo:

<http://turingmachinesimulator.com/shared/uhicjiqfis>

Qualche osservazione sulle notazioni: quando $\delta(q,a)=(q',a',R)$ nel sorgente della macchina M usiamo il simbolo $>$ per indicare R (lo spostamento a destra):

q,a
 $q',a',>$

e analogamente per $\delta(q,a)=(q',a',L)$. In questo caso usiamo il simbolo $<$ per indicare L (lo spostamento a sinistra). Ecco la funzione δ di M, descritta con le convenzioni usate nel sito qui sopra, ed eseguibile nel sito. Tutte le transizioni non elencate portano in uno stato di rifiuto, che non viene mai menzionato.

```
// Alphabet: 0,1,#
// Tape: _ (blank), x
// Language: { w#w | w in {0,1}* } (pairs of equal strings on 0/1)

// Examples: 011000#011000    ---> accept
//              011000#0110000  ---> reject
//              011000#011001   ---> reject

// Algorithm. We check whether in  $x..xa\#x..xb$  the strings  $a,b$ 
// are equal if we ignore the prefixes made of  $x$ .
```



```

// We start from the first 0/1 in the first string,
// we cross it, we move to the second string and we cross the
// first symbol if equal, otherwise we reject.
// If both strings are made of x we accept.

name: String equality tester
init: q1Start
accept: qAccept

//If there are no 0/1 in the first string
//we check whether the second string is only made of x
q1Start,#
q8SkipAllX,#,>

q8SkipAllX,x
q8SkipAllX,x,>

q8SkipAllX,_
qAccept,_,-

//We can skip all transitions returning a reject:
//they are implicitly given in our simulator
//
//q8SkipAllX,0
//qReject,_,-
//
//q8SkipAllX,1
//qReject,_,-

//If the first digit is 0 we cross it, we look for the first digit
//in the second string, if it is 0 we cross it
//then we move backwards
q1Start,0
q2LookFor#ThenQ4,x,>

q2LookFor#ThenQ4,0
q2LookFor#ThenQ4,0,>

q2LookFor#ThenQ4,1
q2LookFor#ThenQ4,1,>

```

q2LookFor#ThenQ4,#
q4CrossFirst0,#,>

q4CrossFirst0,x
q4CrossFirst0,x,>

q4CrossFirst0,0
q6BackTo#,x,<

*//If the first digit is 1 we cross it, we look for the first digit
//in the second string, if it is 1 we cross it
//then we move backwards*

q1Start,1
q3LookFor#ThenQ5,x,>

q3LookFor#ThenQ5,0
q3LookFor#ThenQ5,0,>

q3LookFor#ThenQ5,1
q3LookFor#ThenQ5,1,>

q3LookFor#ThenQ5,#
q5CrossFirst1,#,>

q5CrossFirst1,x
q5CrossFirst1,x,>

q5CrossFirst1,1
q6BackTo#,x,<

*//After we cross two symbols we come back to the first x
//in the first string, then we start again
//from the initial state q1Start*
q6BackTo#,x
q6BackTo#,x,<

q6BackTo#,#
q7BackToX,#,<

```
q7BackToX,0
q7BackToX,0,<
```

```
q7BackToX,1
q7BackToX,1,<
```

```
q7BackToX,x
q1Start,x,>
```

Una versione a 3 registri della Macchina di Turing per il test di equaglianza

Questa macchina usa 3 registri, prende un input nella forma $\#n\#m$ e decide se le due stringhe n, m sono uguali. Richiede un numero di passi nettamente inferiore alla soluzione precedente, che usa un registro solo. Come esempio: se n, m hanno 100 cifre ciascuna, dunque ci sono 201 caratteri di input, la soluzione con un registro richiede circa $100 \times 100 = 10$ mila passi, perché per accoppiare due simboli si sposta da un simbolo in n al corrispondente simbolo in m , e questa traversata richiede circa 100 passi. Invece la soluzione con 3 registri che diamo adesso richiede soltanto 3×100 passi.

Usiamo il simbolo $\#$ per indicare l'inizio di una stringa. Usiamo l'algoritmo seguente: ricopiamo $\#n, \#m$ dal registro 1 nei registri 2 e 3, poi percorriamo all'indietro i registri 2 e 3. Se troviamo due simboli diversi allora n, m sono diverse, se arriviamo al simbolo di inizio stringa $\#$ senza trovare simboli diversi allora $n=m$.

Potete compilare ed eseguire su esempi il codice sorgente che trovate qui sotto. Usate il sito: <http://turingmachinesimulator.com>. Per un certo tempo, troverete una copia del sorgente, sempre da compilare, su

<http://turingmachinesimulator.com/shared/dbmotktqis>

```
// Alphabet: 0,1,#
// Tape: _
// Language: { #n#n | n in {0,1}* } (pairs of equal strings on 0/1)
// Example: accepts #101#101, refuses #111#0111
//
// Equality test Algorithm, 3 tapes
```

```

// for Turing Machine Simulator
// turingmachinesimulator.com
//
// ----- States -----|
// qInit    - initial state      |
// qCopy2    - copies #n in tape 2 |
// qCopy3    - copies #m in tape 3 |
// qCheck    - checks tapes 2,3 for equality |
// qAccept   - accepting state    |
// -----|

name: Equality test with 3 tapes
init: qInit
accept: qAccept

// State qCopyInTape2 copies in tape 2 the first group #n
// with n binary
qInit,#,_,_
qCopyInTape2,#,#,_,>,>,-

qCopyInTape2,0,_,_
qCopyInTape2,0,0,_,>,>,-

qCopyInTape2,1,_,_
qCopyInTape2,1,1,_,>,>,-

// State qCopyInTape3 copies in tape 3 the second group #m
// with m binary
qCopyInTape2,#,_,_
qCopyInTape3,#,_,#,>,-,>

qCopyInTape3,0,_,_
qCopyInTape3,0,_,0,>,-,>

qCopyInTape3,1,_,_
qCopyInTape3,1,_,1,>,-,>

// If the end of input is found,
// state qCopyInTape3 moves pointers 2,3 back to the last symbols
// of n,m. Then State qCheck checks whether n and m are equal strings

qCopyInTape3,_,_,_

```

qCheck,_,_,_,-,<,<

qCheck,_,0,0

qCheck,_,0,0,-,<,<

qCheck,_,1,1

qCheck,_,1,1,-,<,<

qCheck,_,#,#

qAccept,_,#,#,-,-,-

MT per il successore lessicografico

Nella prova del Teorema 3.13, il Sipser utilizza il fatto che sia una MT può scorrere tutte le stringhe di un dato alfabeto, per esempio 0,1, svolgendo una operazione per ogni stringa. L'ordine scelto è per lunghezza (stringhe di lunghezza 0,1,2,...) e per stringhe della stessa lunghezza, l'ordine lessicografico. Una stringa è minore di un'altra della stessa lunghezza nell'ordine lessicografico se nel primo carattere (leggendo da sinistra) in cui le due stringhe sono diverse, la prima ha 0 e la seconda ha 1. In modo equivalente: una stringa è minore di un'altra della stessa lunghezza nell'ordine lessicografico se lette come numeri binari la prima stringa è più piccola. La prima stringa che viene dopo 0111 è dunque 1000. Dunque dopo la stringa vuota viene 0, poi viene 1, poi nell'ordine: 00, 01, 10, 11, e poi:

000, 001, 010, 011, 100, 101, 110, 111, 0000, ...

Come esercizio, proviamo qui a definire una macchina di Turing M che data una stringa produce la stringa successiva nell'ordine lessicografico. Per facilitare il calcolo aggiungiamo un simbolo # di inizio stringa e quindi scriviamo le stringhe nella forma: #, #0, #1, #00, #01, ...

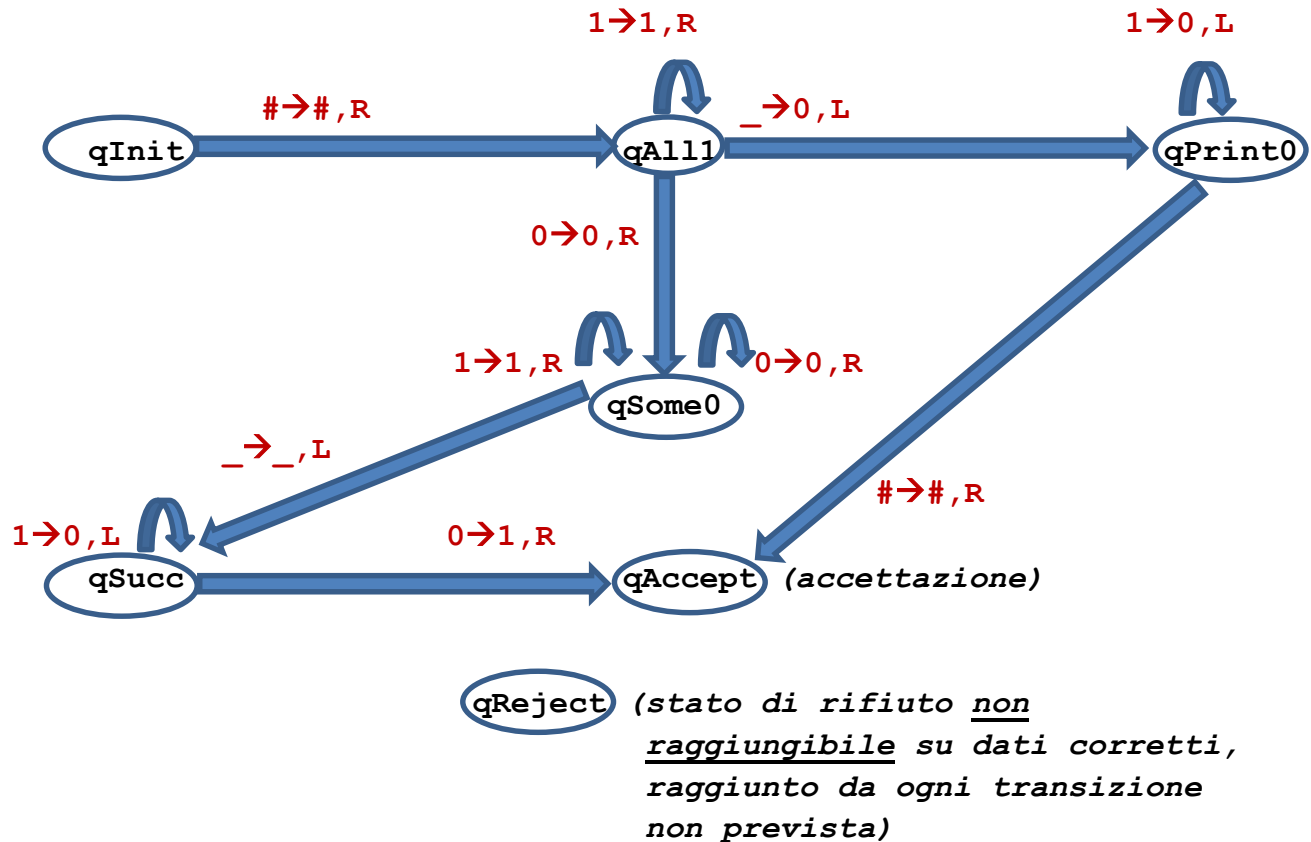
Definiamo la nostra macchina di Turing M. M prende una stringa #s in input, termina accettando sempre, e termina con sul nastro la stringa #t, con t la stringa immediatamente dopo s nell'ordine lessicografico. M sceglie tra due operazioni. (1) Se M trova tutti 1, allora M fa in fondo, aggiunge uno 0 e torna indietro trasformando ogni 1 in 0. Il risultato è #0...0. (2) Se M trova qualche 0, allora va in fondo, e trasforma il suffisso ...01...1 in ...10...0.

M ha come stati:

1. qInit (iniziale)
2. qAll1 (finché tutte le cifre lette sono 1)
3. qSome0 (quando qualche cifra letta è 0)
4. qPrint0 (per aggiungere uno 0 e trasformare tutti gli 1 in 0)
5. qSucc (trasforma il suffisso ...01...1 in ...10...0)
6. qAccept (di accettazione)

7. qRefuse (di rifiuto)

Ecco il grafo di M: ogni transizione non prevista porta a uno stato di rifiuto, ma su dati corretti la macchina M accetta sempre.



Nella prossima pagina definiamo la funzione delta di M, dove > indica R, il movimento a destra. Di nuovo: tutte le transizioni non in elenco portano a uno stato di rifiuto. Potete compilare la macchina su <http://turingmachinesimulator.com>. Inoltre per un certo tempo troverete una copia della macchina su:

<http://turingmachinesimulator.com/shared/mrbdrwkpax>

```
// NEXT BINARY STRING IN LEXICOGRAPHIC ORDER
// Input: #n with n some binary number
// Output: #m with m the next string in lexicographic ordering
```

```

// Example: #11 --> #000, #101 --> #110
//
// Lexicographic successor algorithm
// for Turing Machine Simulator
// turingmachinesimulator.com
//
// ----- States -----|
// qInit    - initial state      |
// qAll1    - all digits 1       |
// qSome0   - some digit 0       |
// qPrint0  - turns all 1 into 0 |
// qSucc    - binary successor   |
// qAccept  - accepting state    |
//-----|
//
// TM returns the next string in the following enumeration:
// #, #0,#1, #00,#01,#10,#11, #000,#001, #010, #011, #100, #101, ...
// string after an all-1 string = the all-0 string with one more digit
// otherwise: TM returns the binary successor
// if we repeatedly apply TM:
// all binary strings are eventually generated from the empty string

//-----CONFIGURATION
Name: lexicographic successor
init: qInit
accept: qAccept

//-----DELTA FUNCTION: first line input, second line output.

// qAll1
// checks whether all digits are 1
// if they all are prints 0 then calls qPrint0

qInit,#
qAll1,#,>

qAll1,0
qSome0,0,>

qAll1,1

```



```
qAll1,1,>
```

```
qAll1,_  
qPrint0,0,<
```

```
// qPrint0  
// when all digits are 1, replaces all 1 with 0  
// then accepts
```

```
qPrint0,1  
qPrint0,0,<
```

```
qPrint0,#  
qAccept,#,>
```

```
// qSome0  
// keeps track of the fact that some digit is 0  
// when the string is over calls qSucc
```

```
qSome0,0  
qSome0,0,>
```

```
qSome0,1  
qSome0,1,>
```

```
qSome0,_  
qSucc,_,<
```

```
// QSucc  
// when some digit is 0, replaces all consecutive 1 with 0  
// then the first 0 with 1  
// then accepts
```

```
qSucc,1  
qSucc,0,<
```

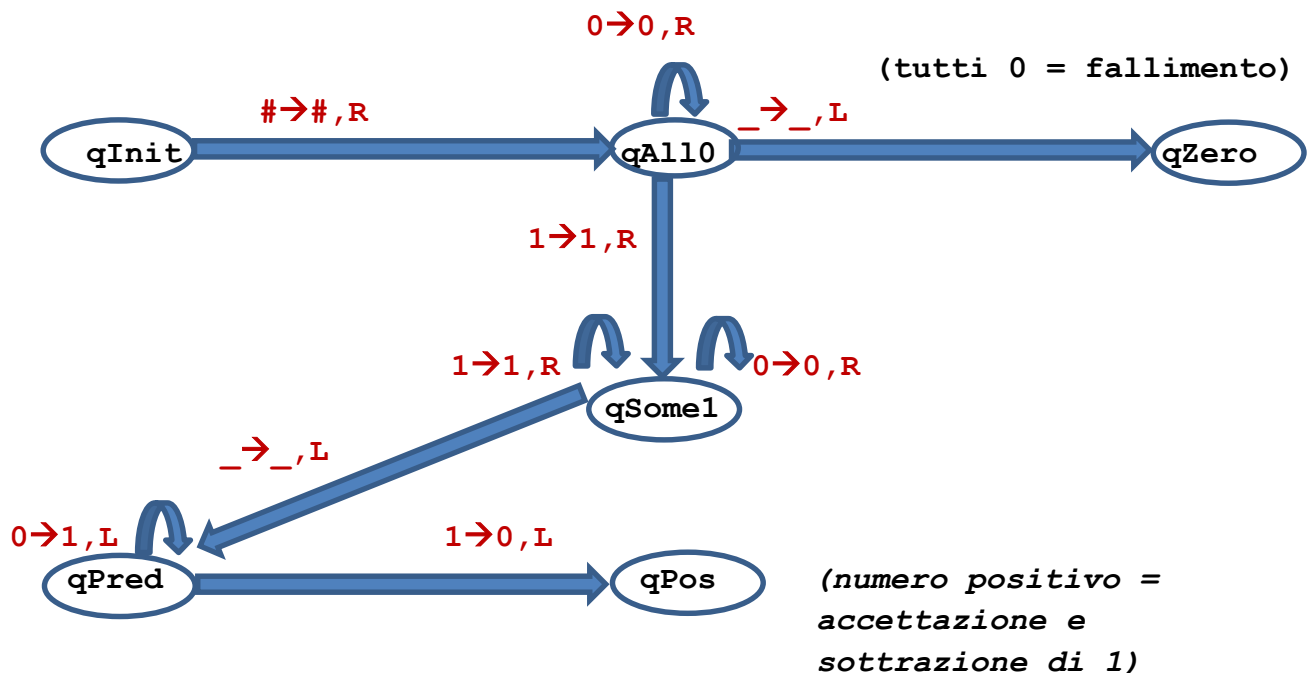
```
qSucc,0  
qAccept,1,>
```

MT per il predecessore (notazione binaria)

Nei capitoli 4,5, spesso il Sipser utilizza il fatto che sia possibile far ripetere a un MT una operazione un numero fisso n di volte. Un modo per ottenere questo effetto è definire una MT M che calcola il predecessore per un numero n in forma binaria.

M parte da uno stato iniziale q_{Init} . Se n è positivo (se contiene almeno una cifra 1), M decrementa n di 1 e entra in uno stato q_{Pos} , se n è zero (contiene solo la cifra 0) allora M entra in uno stato q_{Zero} . Nel caso il compito di M sia solo di calcolare il predecessore se esiste, e altrimenti di fallire, allora q_{Pos} è uno stato di accettazione e q_{Zero} di rifiuto. In generale q_{Pos} , q_{Zero} saranno due stati intermedi di M , se M arriva a q_{Pos} allora M ripete una certa azione e torna a q_{Init} e a inizio stringa, se M arriva a q_{Zero} allora M termina il gruppo di n azioni. Scriviamo le stringhe nella forma $\#n$ per rendere facile il ritorno a inizio stringa.

Esempi. Se M riceve in input $\#0100$ lo trasforma in $\#0011$ e passa nello stato q_{Pos} (ha successo). Se M riceve in input $\#0000$ non modifica la stringa e passa nello stato q_{Zero} (fallisce).



Nella prossima pagina definiamo la funzione delta di M, dove > indica R, il movimento a destra. Tutte le transizioni non in elenco portano a uno stato di rifiuto. Trovate una copia della macchina, da compilare, su:

<http://turingmachinesimulator.com/shared/rmzerttkfj>

```
//Example.  
//#0100 --> #0011 success  
//#0000 --> #0000 failure
```

```
//-----CONFIGURATION  
name: predecessor  
init: qInit  
accept: qPos
```

```
//-----DELTA FUNCTION:  
qInit,#  
qAll0,#,>
```

```
qAll0,0  
qAll0,0,>
```

```
qAll0,1  
qSome1,1,>
```

```
qAll0,_  
qZero,_,<
```

```
qSome1,0  
qSome1,0,>
```

```
qSome1,1  
qSome1,1,>
```

```
qSome1,_  
qPred,_,<
```

```
qPred,0  
qPred,1,<
```

qPred,1
qPos,0,<

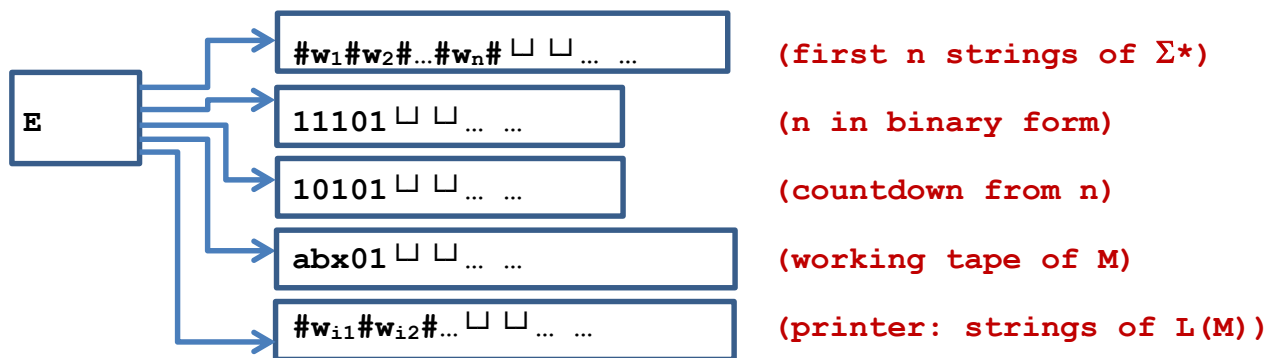
//if we reach _ (blank), then qZero (failure): the argument was zero.
//We turn any string 0...0 into 1...1 we reach qZero and we fail.

Enumeratore E a partire da una MT M

Supponiamo che M sia un MT che decide positivamente (cioè riconosce) un linguaggio $L = L(M)$ su un alfabeto Σ . Supponiamo di aver enumerato le stringhe $w \in \Sigma^*$ per lunghezza e a parità per l'ordine lessicografico. **Vogliamo definire un enumeratore E per L** (Teorema 3.21 del Sipser).

Non è sufficiente scorrere la lista delle stringhe $w \in \Sigma^*$ e chiedere a M ogni volta se $w \in L$. Infatti M è solo un decisore positivo di M e la risposta di M potrebbe non arrivare mai, impedendoci di esaminare le stringhe successive a w per stabilire se sono in L. Dobbiamo quindi simulare l'esecuzione in parallelo di M su tutte le stringhe $w \in \Sigma^*$, ripartendo il tempo di E tra un numero di esecuzioni sempre crescente.

Per farlo definiamo un enumeratore con 5 registri, riutilizzando le operazioni di successore lessicografico, di successore e di predecessore viste nei precedenti esercizi. Il primo registro contiene l'enumerazione delle prime n stringhe $w \in \Sigma^*$: qui periodicamente aggiungiamo una nuova stringa. Il secondo contiene il numero n, il terzo un contatore all'indietro che parte da n, il quarto il registro di M. Il quinto registro è il printer e contiene la lista delle stringhe di L anche con ripetizioni.



Sia $\#$ un simbolo non in Σ . E inizia ponendo la prima stringa $w_1 \in \Sigma^*$ nel primo registro nella forma $\#w_1\#$, e ponendo 1 sia nel registro di n sia nel registro del countdown, ponendo w nel registro di lavoro di M e ponendo $\#$ nel registro del printer (per la lista delle stringhe accettate da M). La prima stringa è w_1 =la stringa di 0 elementi, dunque nel primo registro abbiamo $\#\#$: il doppio simbolo $\#$ ci consente di

riconoscere l'inizio della lista di stringhe. Per ogni valore di n , l'enumeratore E legge una stringa w_i tra le stringhe w_1, \dots, w_n del primo registro, la pone nel registro di lavoro di M ed esegue le transizioni di M per n passi. Per garantire n passi di calcolo su ogni w_i , ad ogni passo viene decrementato di uno il registro di countdown, e il passo viene eseguito solo se il registro non contiene il valore zero. Se durante la simulazione M accetta la stringa w_i , allora E copia $w_i\#$ nel printer. Poi E passa alla stringa precedente w_{i-1} nel primo registro, e continua simulando le computazioni a partire da tutte le stringhe w_1, \dots, w_n , assegnando ad ogni simulazione n passi. Quando E ha eseguito la simulazione su $w_1 =$ stringa vuota, allora E incrementa n di uno, aggiunge nel primo registro il successore lessicografico w_{n+1} dell'ultima stringa w_n , e ricomincia da capo. L'enumeratore E non termina mai.

E enumera L . Infatti, ad ogni passo, E simula i primi n passi di computazione di M sulle prime n stringhe w_1, w_2, \dots, w_n di Σ^* . Dunque la computazione su ogni stringa w_i viene ripetuta infinite volte, ogni volta consentendo più passi di calcolo. Quindi se $w_i \in L$, cioè se M accetta w_i , allora prima o poi avviene una simulazione abbastanza lunga perché w_i sia accettata e aggiunta al printer. Viceversa se una stringa viene aggiunta al printer, allora è stata accettata da M e quindi appartiene a L . Dunque E stampa tutte e sole le stringhe di L , quindi E enumera $L=L(M)$, lo stesso linguaggio accettato da M .

NOTA. Non abbiamo impedito che la stessa stringa venga accettata più volte. Infatti non eliminiamo le stringhe dal primo registro $w_1\#w_2\#\dots\#w_n$. Quindi quando una stringa w_i viene accettata, ricontrolliamo ancora w_i nel prossimo ciclo e la accettiamo di nuovo. Volendo, potremmo evitarlo. Potremmo aggiungere un nuovo simbolo $*$ (asterisco), diverso da $\#$ e non in Σ , per indicare la cancellazione. Quando una stringa w_i in $w_1\#w_2\#\dots\#w_n$ viene accettata ne rimpiazziamo tutti i caratteri con $*$ e passiamo alla stringa precedente. Se istruiamo l'enumeratore E ad ignorare tutti gli asterischi in $w_1\#w_2\#\dots\#w_n$, non riconsideriamo mai più w_i .

Soluzione Problema 3.8.a del Sipser

Consideriamo il linguaggio L fatto di tutte e sole le stringhe in $\{0,1\}^*$ con il numero di zeri uguale al numero degli uno. Definiamo:

$$L = \{w \in \Sigma^* \mid (\text{numero degli } 0 \text{ in } w) = (\text{numero degli } 1 \text{ in } w)\}$$

dove: $\Sigma = \{0,1\}$. Esempi: stringa vuota, $01 \in L$, mentre $010 \notin L$. Il problema 3.8.a del Sipser ci chiede di definire una TM che decide L .

Soluzione: <http://turingmachinesimulator.com/shared/gijsitagkg>

Algoritmo. Quando troviamo 0 e poi 1, oppure 1 e poi 0, rimpiazziamo il primo simbolo con F (first) e il secondo con S (secondo). Torniamo indietro alla F più a destra e ripetiamo, finchè falliamo un accoppiamento e rispondiamo no, oppure accoppiamo tutti i simboli e rispondiamo sì.

Tempo di calcolo: per una stringa di n simboli, nel caso peggiore circa $(1/2)n^2$ passi. Esempio: $n=1000$, abbiamo 500 zeri seguiti da 500 uno. Ogni accoppiamento costa (circa) 500 passi in avanti e 500 indietro, dunque 1000 passi. Occorrono 500 accoppiamenti, dunque 500 mila passi = $(1/2)n^2$ passi.

Spazio di calcolo: "in place", ovvero entro la dimensione n dell'input.

Nota. Vedremo come aggiungere 2 tape diminuisce notevolmente il tempo di calcolo, a prezzo di un modesto incremento dello spazio di calcolo.

Codice sorgente:

```
// Initial state: qInit
// Accepting state: qAccept
// Whole alphabet:
// 0, 1, _ (blank), F (first symbol paired), S (second symbol paired)

name: zeros equals ones
init: qInit
accept: qAccept

//We pair one 0 and one 1 in any order.
```

```
//We replace the first symbol with F and the second symbol with S.  
//We come back to the last F then we repeat, until no 0,1 is left  
//or we fail to pair 0 and 1
```

```
//qInit. If there are no 0's and no 1's, accept  
qInit,_  
qAccept,_,-
```

```
//We replace the first 0 or 1 with F  
qInit,1  
q1,F,>
```

```
qInit,0  
q0,F,>
```

```
//We skip all S's  
//(the first time there is no S, later on there are plenty)  
qInit,F  
qInit,F,>
```

```
qInit,S  
qInit,S,>
```

```
//q0 is looking for some 1 to cross out. It skips the rest  
//If q0 finds _, fails (no transition in this case)  
q0,1  
qBack,S,<
```

```
q0,0  
q0,0,>
```

```
q0,S  
q0,S,>
```

```
//q0 is looking for some 0 to cross out. It skips the rest  
//If q0 finds _, fails (no transition in this case)  
q1,0  
qBack,S,<
```

```
q1,1  
q1,1,>
```


q1,S
q1,S,>

//qBack. If we crossed out 0,1 in any order then we come back
// to the rightmost F. Eventually we repeat.

qBack,S
qBack,S,<

qBack,0
qBack,0,<

qBack,1
qBack,1,<

//We repeat from the rightmost F (first symbol paired)
qBack,F
qInit,F,>

2° Soluzione Problema 3.8.a del Sipser

Con una macchina a 3 registri

Consideriamo di nuovo il linguaggio L fatto di tutte e sole le stringhe in $\{0,1\}^*$ con il numero di zeri uguale al numero degli uno, e con simbolo di inizio stringa $\#$. Definiamo:

$$L = \{\#w \in \Sigma^* \mid (\text{numero degli } 0 \text{ in } w) = (\text{numero degli } 1 \text{ in } w)\}$$

dove: $\Sigma = \{0,1\}$. Esempi: $\#$, $\#01 \in L$, mentre $\#010 \notin L$. Definiamo una TM a 3 registri che decide L . Come tempo di calcolo, la soluzione è nettamente meglio della soluzione con 1 tape.

Algoritmo. Copiamo $\#$ e gli zeri nella tape 2, copiamo $\#$ e gli uno nella tape 3, controlliamo che le tape 2 e 3 abbiano lo stesso numero di simboli. Usiamo gli stati q_{Init} , q_{Copy} , q_{Check} , e naturalmente q_{Accept} e q_{Reject} . Quest'ultimo non viene menzionato nella descrizione della macchina, ogni transizione non prevista porta a q_{Reject} .

Tempo di calcolo: per una stringa di n simboli, nel caso peggiore circa $(3/2)n$ passi (nettamente meglio della soluzione precedente con una tape). Esempio: $n=1000$, facciamo 1000 passi per copiare nelle tape 2 e 3, la tape con meno simboli ora contiene al più 500 simboli, dopo altri 500 passi finiamo. Dunque $1500 = (3/2)n$ passi.

Spazio di calcolo: $2n$ (poco più della soluzione precedente con una tape).

Codice sorgente:

```
// Initial state: qInit
// Accepting state: qAccept
// Whole alphabet:
// 0,1,_ (blank), #(start of a string)
// Examples. #0101 ---> accept, #01010 ---> refuse

name: zero equals one 1 tape
init: qInit
```

```

accept: qAccept

// 3 tapes. We copy # in tapes 2 and 3.
// We copy any 0 in tape 2 and any 1 in tape 3.
// Then we check whether tape 2 and 3 have same number of symbols

//qInit. We copy # in tape 2 and
qInit, #,_,_
qCopy, #,#,#, >,>,>

//We copy all 0's and all 1's in tapes 2 and 3
qCopy, 0,_,_
qCopy, 0,0,_, >,>,-

qCopy, 1,_,_
qCopy, 1,_,1, >,-,>

qCopy, _,'_,'_
qCheck, _,'_,'_, -, <,<

//We check whether tapes 2 and 3 have the same number of symbols
qCheck, _,0,1
qCheck, _,0,1, -, <,<

qCheck, _,#,#
qAccept, _,#,#, -, -, -

```

3° Soluzione Problema 3.8.a del Sipser

Trovata nel 2021, macchina a un registro

In questa soluzione proposta dagli studenti nel 2021 accoppiamo uno 0 e un 1 o viceversa trasformando entrambi in x. Formata una coppia ritorniamo a inizio stringa e ne cerchiamo un'altra. Se riusciamo ad accoppiare tutti gli 0,1 accettiamo, se non riusciamo a formare una coppia rifiutiamo. La prima coppia anziché x,x viene scritta #,x. Il simbolo # viene usato per consentirci di riconoscere l'inizio della stringa.

Ricordiamo che tutte le transizioni non indicate portano al rifiuto.

```
// <           = left
// >           = right
// _ (underscore) = blank cell
```

```
//-----CONFIGURATION
```

```
name: zeros=ones
```

```
init: qInit
```

```
accept: qAccept
```

```
//-----DELTA FUNCTION:
```

```
//Se non ci sono 0,1 accettiamo
```

```
qInit,_
```

```
qAccept,_,>
```

```
//Se ci sono cifre 0,1 la prima cifra diventa #
```

```
//poi cerchiamo la cifra opposta con LookForOne, LookForZero
```

```
qInit,0
```

```
qLookForOne,#,>
```

```
qInit,1
```

```
qLookForZero,#,>
```

```
//LookForOne salta ogni x,0, se trova un 1 lo trasforma in x
```

```
//dopo chiediamo a qBackToStart di tornare a #
```

```
qLookForOne,x
qLookForOne,x,>
```

```
qLookForOne,0
qLookForOne,0,>
```

```
qLookForOne,1
qBackToStart,x,<
```

```
// qBackToStart torna a # saltando 0,1,x
qBackToStart,0
qBackToStart,0,<
```

```
qBackToStart,1
qBackToStart,1,<
```

```
qBackToStart,x
qBackToStart,x,<
```

```
// Quando qBackToStart torna a #
// chiediamo a qSearchForZeroOrOne di cercare 0,1 saltando le x
qBackToStart,#
qSearchForZeroOrOne,#,>
```

```
qSearchForZeroOrOne,x
qSearchForZeroOrOne,x,>
```

```
//Se non ci sono 0,1, abbiamo accoppiato tutti gli 0,1 e accettiamo
qSearchForZeroOrOne,_
qAccept,_,>
```

```
//Se qSearchForZeroOrOne trova una cifra 0,1 la sostituiamo con x
//poi usiamo qLookForOne, qLookForZero per cercare la cifra opposta
qSearchForZeroOrOne,_
qAccept,_,>
```

```
qSearchForZeroOrOne,0
qLookForOne,x,>
```

```
qSearchForZeroOrOne,1
qLookForZero,x,>
```

Soluzione Problema 3.8.b del Sipser

Consideriamo il linguaggio L fatto di tutte e sole le stringhe in $\{0,1\}^*$ con il numero di zeri doppio del numero degli uno. Definiamo:

$$L = \{w \in \Sigma^* \mid (\text{numero degli } 0 \text{ in } w) = 2 * (\text{numero degli } 1 \text{ in } w)\}$$

dove: $\Sigma = \{0, 1\}$. Esempi: $001, 010, 100 \in L$, mentre $01 \notin L$. Il problema 3.8.b del Sipser ci chiede di definire una TM che decide L .

Definizione di una macchina di Turing M per decidere L . Sia n la lunghezza di w . Definiamo una TM di nome M , con tape alphabet $\Gamma = \{0, 1, \sqcup, F \text{ (first), } N \text{ (next)}\}$, e con stati $Q = \{q, q_0, q_{00}, q_1, q_{01}, q_{001}, q_{\text{accept}}, q_{\text{reject}}\}$.

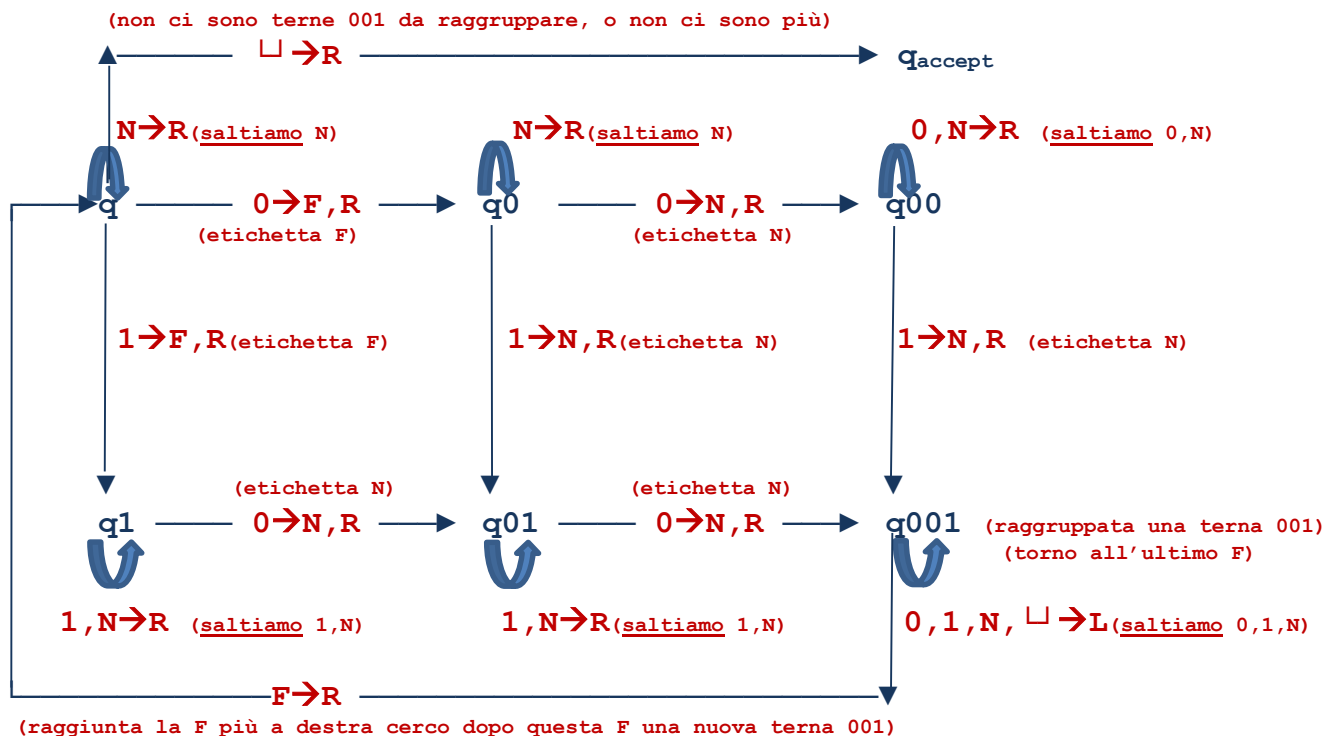
M traversa $n/3$ volte al massimo la stringa w avanti e indietro. Ogni volta M individua la terna $0,0,1$ (in qualsiasi ordine) il più possibile a sinistra tra quelle rimaste, e la rimpiazza con F ("first", al posto del primo simbolo della terna) e con N, N ("next", "next", al posto del secondo e il terzo). Dato che il simbolo F all'interno di w viene usato per indicare l'inizio di una nuova terna, allora a sinistra di F in w ci sono solo tutti simboli F (i primi simboli di tutte le terne precedenti) e alcune tra le N . M rimpiazza con F, N, N ogni gruppo di 3 simboli $0,0,1$ in w e raggiunge \sqcup (blank). Quindi arretra, saltando \sqcup ed ogni $N,0,1$ che trova in w , fino al simbolo F più a destra tra tutti i simboli F in w . Poi M muove un passo a destra e ricomincia a crocettare una nuova terna in quello che resta di w . Se M raggiunge un simbolo \sqcup (il primo a destra di w) senza aver crocettato almeno un simbolo di una nuova terna, allora M ha completamente crocettato w e M accetta. Se M raggiunge un simbolo \sqcup dopo aver crocettato uno oppure due simboli di una terna (che quindi resta incompleta) allora M rifiuta.

Nel prossimo diagramma tutte le transizioni non indicate portano allo stato q_{reject} di rifiuto (a sua volta non indicato). Notiamo che q_{01} , per esempio, indica lo stato di M dopo aver letto $0,1$ in qualsiasi ordine. Notiamo anche che quando M ha già trovato due 0 allora M salta ogni nuovo 0 , cercando solo più 1 . Quando M ha già trovato un 1 , allora M salta ogni nuovo 1 , cercando solo più 0 .

Sulle notazioni usate. $0 \rightarrow R$ sta per $0 \rightarrow 0, R$, spostamento a destra senza riscrivere 0: lo 0 viene saltato. $0, N \rightarrow R$ sta per $0 \rightarrow 0, R$ e $N \rightarrow N, R$: queste transizioni saltano 0 e N.

Transizioni di una macchina M che risolve il Problema 3.8.b

q cerca una terna da raggruppare. $q0, q00, q1, q01, q001$ indicano quanti zeri e uno sono stati trovati, in qualsiasi ordine. Gli zeri e uno in soprannumero rispetto alla richiesta di avere due 0 e un 1 vengono saltati, e così tutti i simboli N ("next"). Il primo 0,1 trovato viene etichettato F ("first"), gli altri due diventano N, N ("next", "next"). Raggruppata una terna M torna all'ultima F ("first") e ricomincia.



Qui sotto una descrizione alternativa della stessa macchina M, usando la funzione delta di M, sempre omettendo le transizioni che portano a un rifiuto. Per qualche tempo, trovate una descrizione compilabile di M su:

<http://turingmachinesimulator.com/shared/keljelmbuf>

```
// Initial state: q
// Accepting state: qAccept
// Whole alphabet: 0,1,_,N,F
```

```
name: zero doubles one
init: q
accept: qAccept
```

```
//Accept
q,_
qAccept,_,-
```

```
//q
q,0
q0,F,>
```

```
q,1
q1,F,>
```

```
q,N
q,N,>
```

```
//q1
q1,0
q01,N,>
```

```
q1,1
q1,1,>
```

```
q1,N
q1,N,>
```

```
//q0
q0,0
q00,N,>
```

```
q0,1
q01,N,>
```


q0,N
q0,N,>

//q00
q00,0
q00,0,>

q00,1
q001,N,>

q00,N
q00,N,>

//q01
q01,0
q001,N,>

q01,1
q01,1,>

q01,N
q01,N,>

//q001
q001,0
q001,0,<

q001,1
q001,1,<

q001,N
q001,N,<

//Repeat
q001,_
q001,_,<

q001,F
q,F,>

2° Soluzione Problema 3.8.b del Sipser

3 Tapes

Consideriamo di nuovo il linguaggio L fatto di tutte e sole le stringhe in $\{0,1\}^*$ con il numero di zeri doppio del numero degli uno, e con simbolo di inizio stringa $\#$. Definiamo:

$$L = \{\#w \in \Sigma^* \mid (\text{numero degli } 0 \text{ in } w) = 2 * (\text{numero degli } 1 \text{ in } w)\}$$

dove: $\Sigma = \{0,1\}$. Esempi: $\# \in L$, $\#010 \in L$, mentre $\#01 \notin L$. Definiamo una TM a 3 registri che decide L . Come tempo di calcolo, la soluzione è nettamente meglio della soluzione precedente con 1 tape.

Algoritmo. Copiamo $\#$ e gli zeri nella tape 2, copiamo $\#$ e gli uno nella tape 3, controlliamo che la tape 2 abbia un numero di 0 doppio del numero di 1 nella tape 3. Usiamo gli stati q_{Init} , q_{Copy} , q_{Check} , q_{Check2} , e naturalmente q_{Accept} e q_{Reject} .

- Quando q_{Check} trova uno zero nella tape 2 e un "uno" nella tape 3, si sposta alla casella dopo nella tape 2 ma non muove nella tape 3: q_{Check} mantiene la testina di lettura sullo stesso "uno" di prima.
- Quindi q_{Check} chiede a q_{Check2} di controllare che nella nuova casella sulla tape 2 ci sia un altro zero da accoppiare con lo stesso uno di prima.
- Se q_{Check2} lo trova, allora sposta di un passo le testine sulle tape 2 e 3, passando a un nuovo "uno", e chiede a q_{Check} di ricominciare ad accoppiare due "zeri" e un "uno".

q_{Reject} non viene menzionato nella descrizione della macchina, ogni transizione non prevista porta a q_{Reject} .

Tempo di calcolo: per una stringa di n simboli, nel caso peggiore circa $(3/2)n$ passi (in realtà meno, comunque nettamente meglio della soluzione precedente con una tape). Esempio: $n=1000$, facciamo 1000 passi per copiare nelle tape 2 e 3, la tape con meno simboli ora contiene al più 500 simboli, dopo altri 500 passi finiamo. Dunque $1500 = (3/2)n$ passi.

Spazio di calcolo: $2n$ (poco più della soluzione precedente con una tape).

```

// We start adding # at the beginning of tape 2 and of tape 3

qInit, #, _, _
qCopy, #, #, #, >, >, >

//qCopy copies all symbols 0 in tape 1 into tape 2
//and all symbols 1 in tape 1 into tape 3
qCopy, 0, _, _
qCopy, 0, 0, _, >, >, -

qCopy, 1, _, _
qCopy, 1, _, 1, >, -, >

qCopy, _, _ _
qCheck, _, _ _ -, <, <

// qCheck pairs two symbols 0 in tape 2
// with a single symbol 1 in tape 3
qCheck, _, 0, 1
qCheck2, _, 0, 1, -, <, -
// We do not move from the symbol 1 we found in tape 3

// Instead, qCheck2 looks for a second symbol 0
// to pair with 1 in tape 2
qCheck2, _, 0, 1
qCheck, _, 0, 1, -, <, <
// If we found a second symbol 0 in tape 2 to pair with a symbol 1
// in tape 3, then we call qCheck again. qCheck restarts pairing
// two symbols 0 in tape 2 with a single symbol 1 in tape 3

// If we paired all symbols 1 in tape 3 with two symbols 0 in tape 2
// then we accept.
qCheck, _, #, #
qAccept, _, #, #, -, -, -

```

**Decidibile, decidibile solo positivamente
e decidibile solo negativamente**

Riprendiamo le definizioni di decidibile, positivamente decidibile ("Turing recognizable" sul Sipser) e negativamente decidibile ("Turing co-recognizable" sul Sipser) viste nel corso. Per aiutare la comprensione, e per esercitarci sulle definizioni, ne forniamo una versione schematica. Sia L un linguaggio su Σ . In altre parole, L è un sotto-insieme dell'insieme Σ^* delle stringhe sull'alfabeto Σ .

L decidibile: esiste un M che decide L .

Per ogni $w \in \Sigma^*$ chiediamo:

se $w \in L$	M accetta w
se $w \notin L$	M rifiuta w

L positivamente decidibile (oppure: Turing-riconoscibile): esiste un M che decide positivamente L . Per ogni $w \in \Sigma^*$ chiediamo:

se $w \in L$	M accetta w
se $w \notin L$	M rifiuta w oppure M non termina su w

L negativamente decidibile: esiste un M che decide negativamente L .

Per ogni $w \in \Sigma^*$ chiediamo:

se $w \in L$	M accetta w oppure M non termina su w
Se $w \notin L$	M rifiuta w

Sia $\underline{L} = \{w \in \Sigma^* \mid w \notin L\}$ il linguaggio complemento di L . Se M è una macchina di Turing, sia \underline{M} la macchina complementare, ottenuta scambiando accettazione e rifiuto in M . In base alle definizioni precedenti otteniamo che il passaggio al complementare scambia tra loro la decidibilità positiva e negativa di L . Inoltre passando alla macchina complementare scambiamo i decisori di L con i decisori di \underline{L} , i decisori positivi di L con i decisori negativi di \underline{L} , e viceversa:

L decidibile $\Leftrightarrow \underline{L}$ decidibile

M decide L $\Leftrightarrow \underline{M}$ decide \underline{L}

L positivamente decidibile $\Leftrightarrow \underline{L}$ negativamente decidibile

M decide positivamente L $\Leftrightarrow \underline{M}$ decide negativamente \underline{L}

L negativamente decidibile $\Leftrightarrow \underline{L}$ positivamente decidibile

M decide negativamente L $\Leftrightarrow \underline{M}$ decide positivamente \underline{L}

Macchina di Turing universale

Riprendiamo ed ampliamo la definizione di macchina universale accennata nel nostro libro di testo, il Sipser, nella prova del Teor. 4.11. Definiamo una M.T. universale U con 5 tape che prenda un input $\langle M, w \rangle$, contenente la descrizione tramite stringhe di una macchina di Turing M con 1 tape, e un input w per M . U simula passo passo il calcolo di M su w , accetta se M accetta w , rifiuta se M rifiuta w , e continua per sempre se M continua per sempre su w .

Alfabeto di U . Supponiamo che M abbia insieme di stati Q , alfabeto Σ , alfabeto della tape Γ , funzione di transizione δ e stati: iniziale/accettazione/ rifiuto, uguali a $q_0, q_{\text{accept}}, q_{\text{reject}} \in Q$. Supponiamo $Q \cup \Gamma = \{s_1, \dots, s_n\}$ e poniamo s_i^* = la forma binaria di i . Usiamo la stringa binaria s_i^* come rappresentazione dello stato o simbolo s_i . Come alfabeto di U prendiamo $\Sigma_U = \{0, 1, \#\}$ e $\Gamma_U = \Sigma_U \cup \{\sqcup\}$: allora per ogni stato o simbolo $s \in Q \cup \Gamma$ abbiamo $s^* \in \Sigma_U^*$, cioè la stringa che rappresenta s è una stringa dell'alfabeto di U . Usiamo $\#$ per separare le stringhe binarie che rappresentano gli elementi di $Q \cup \Gamma$.

Le 5 tapes di U . La macchina universale U ha una tape 1, read-only (cioè che non modifichiamo mai), per la regola di transizione δ . Descriviamo δ descritta come una sequenza di quintuple $q_1, a_1, q'_1 a'_1 L$, poi $q_2, a_2, q'_2 a'_2 R$, eccetera, tali che $\delta(q_1, a_1) = q'_1 a'_1 L$, $\delta(q_2, a_2) = q'_2 a'_2 R$, Rappresentiamo i simboli di $Q \cup \Gamma$ con le loro stringa binaria, separate da $\#$, in modo da usare simboli di U e non simboli di M :

$\# \# q_1^* \# a_1^* \# q_1'^* \# a_1'^* \# L^* \# q_2^* \# a_2^* \# q_2'^* \# a_2'^* \# R^* \# \dots \# \#$ (stringa di $0, 1, \#$)

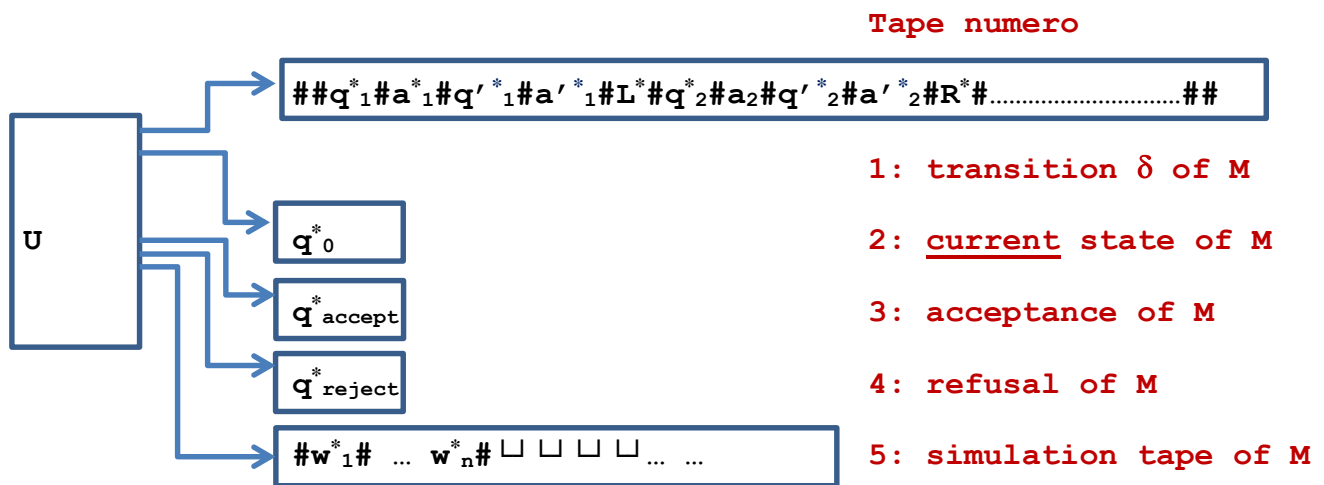
$\# \#$ indica il margine sinistro e quello destro di δ .

Una tape 2 contiene dapprima lo stato iniziale q_0^* di M , come stringa binaria, poi lo stato corrente di M . Una tape 3 e una tape 4, read-

only, contengono gli stati di accettazione e di rifiuto: q^*_{accept} e q^*_{reject} , sempre come stringhe binarie.

Una tape 5 simula la tape di M e contiene inizialmente l'input $w = w_1 \dots w_n$, scritto come stringhe binarie con separatore #: $\#w_1\# \dots w_n\#$. Ecco lo stato iniziale della macchina U . Sottolineiamo ancora: le tape 1,3,4 descrivono M e non vengono modificate durante il calcolo.

La macchina U in un disegno (stato iniziale)



Ad ogni passo U legge lo stato corrente q^* nella tape e la rappresentazione $a^*\#$ del simbolo sotto la testina di lettura di M nella tape 5 (se invece trova \square allora intende $\square^*\#$, cioè intende che $a=\square$). Inizialmente abbiamo $q^*=q^*_0$. Se $q^*=q^*_{\text{accept}}$ oppure $q^*=q^*_{\text{reject}}$, allora U accetta/rifiuta. Altrimenti U simula un passo di M . U cerca in δ la prima quintupla $\#q_i\#a_i\#q'_i\#a'_i\#L^*$ (oppure $\#q_i\#a_i\#q'_i\#a'_i\#R^*$) tale che $q^*=q_i$ e $a^*=a_i$. U inizia con $\#q_1\#a_1\#$, e se U fallisce con la coppia $q_i\#a_i$ allora U prova con la prossima coppia $\#q_{i+1}\#a_{i+1}$, posizionandosi dopo cinque separatori $\#$. C'è esattamente una coppia del genere se δ prevede una transizione corretta dallo stato q e dal simbolo a . Se invece U raggiunge la coppia $##$ senza trovare la coppia $q^*\#a^*$ richiesta allora U rifiuta. Se $q^*=q_i$ e $a^*=a_i$, cioè se U trova q_a come parte iniziale di una quintupla $\#q_i\#a_i\#q'_i\#a'_i\#L^*$ oppure $\#q_i\#a_i\#q'_i\#a'_i\#R^*$,

allora U legge q'^*_i e lo sovrascrive allo stato q^* della macchina simulata M nella tape 3. Poi U legge la rappresentazione del nuovo simbolo a'^*_i e lo sovrascrive ad a^* nella simulazione dell'input tape di M (tape 5), e questa volta aggiunge $\#$. Infine U muove la testina, sempre sulla tape 5, verso il simbolo $\#$ precedente o successivo, a seconda se ha trovato L^* oppure R^* in fondo alla quintupla.

In questo modo U ha simulato un passo del calcolo di M . Ora U riporta all'inizio la testina di lettura della tape 1, quella che contiene la transizione δ di M . La testina della tape 1 si posiziona subito a destra della coppia iniziale $\#\#$. Quindi U ricomincia a simulare un nuovo passo di M .

Qualche osservazione

sulla prova di Diagonalizzazione (Teor. 4.11 del Sipser)

Per ragioni logiche la maggior parte dei problemi matematici e informatici di tipo molto generale non è decidibile: vale per l'accettazione, la fermata, il linguaggio vuoto, l'eguaglianza di linguaggi, per quasi tutti i problemi di tipo molto generale. Non abbiamo una vera e propria spiegazione di questo fatto, ciò che sappiamo per certo è che la prova di diagonalizzazione ci dice che è vero.

Tuttavia, possiamo fare una osservazione di carattere generale. *L'indecidibilità avviene quando la macchina H che fa da "decisore" può venire simulata nella classe dei problemi che decide.* Il preteso decisore H non è affidabile quando viene applicato ad un dato D capace di simulare il calcolo eseguito dal "decisore" H su un dato qualsiasi, incluso il dato D stesso. In questo caso il dato D non si comporta come una informazione inerte, ma come un vero e proprio ostacolo al programma H . D può prevedere che risposta darà il programma H che lo analizza, e quindi può comportarsi in modo da smentire la risposta di H . Il fatto stesso che il programma H dia una risposta quando applicato a D significa che questa risposta è errata.

Proviamo a rileggere in questo modo la sezione 4.2 del Sipser, "The Halting Problem" (pag. 179), sezione che riguarda in realtà il problema dell'Accettazione. Supponiamo di avere un decisore per il problema dell'Accettazione, una macchina H che prende una coppia $\langle M, w \rangle$ di una macchina M e un input w , e dice se M accetta w . Definiamo una macchina D capace di simulare le risposte date da H . D che prende una macchina M , usa H per stabilire se M accetta il proprio codice sorgente $\langle M \rangle$, e sapendo prevedere fornisce la risposta opposta a quella di H . Quando applico D a $\langle D \rangle$, per effetto della definizione di D applico H alla coppia $\langle D, \langle D \rangle \rangle$, con D definita a partire da H e in modo contraddittorio con H .

Non stiamo fornendo qui spiegazioni scientifiche e rigorose, ma solo osservazioni sulla prova di indecidibilità. Per esempio, non sapremmo trasformare queste osservazioni in un criterio per stabilire quando un problema non è decidibile: dunque non si tratta di osservazioni così precise.

Decidibilità dei linguaggi context-free

Una grammatica G in forma normale di Chomsky è un insieme finito di variabili $\{S, A, B, C, \dots\}$, che include una variabile S detta start, e di simboli $\{a, b, \dots\}$ di un alfabeto Σ detti terminali della grammatica. La grammatica ha un insieme finito di regole della forma: $S \rightarrow \varepsilon$ (ε =stringa vuota), oppure $A \rightarrow BC$ con A, B, C variabili e B, C diverse da S , oppure $A \rightarrow a$ con A variabile e $a \in \Sigma$ (a terminale). Il linguaggio $L \subseteq \Sigma^*$ definito da G è l'insieme $L(G)$ delle stringhe $w \in \Sigma^*$ che possono essere ottenute in un numero finito di passi a partire da S con le regole date.

Come esempio, sia G con variabili $\{S, A, B\}$, terminali $\Sigma = \{a, b\}$, e regole: $\{S \rightarrow AB, A \rightarrow AA, A \rightarrow a, B \rightarrow b\}$. Allora il linguaggio $L(G)$ definito da G consiste in tutte e sole le stringhe fatte da una o più "a" seguite da una sola "b", dunque $L(G) = \{ab, aab, aaab, \dots\}$. Una successione di regole che ottiene una stringa w di L a partire da una variabile A viene detta derivazione e indicata con $A \Rightarrow w$. $L(G)$ è l'insieme delle stringhe $w \in \Sigma^*$ tali che $S \Rightarrow w$. Come esempio, usando una regola per volta, possiamo derivare la stringa $w=aaab$ di lunghezza 4 in 7 passi, come segue:

$$S \Rightarrow AB \Rightarrow AAB \Rightarrow AAAB \Rightarrow aAAB \Rightarrow aaAB \Rightarrow aaaB \Rightarrow aaab$$

Un linguaggio definibile da una grammatica in forma normale di Chomsky è detto **context-free**: esistono diverse altre definizioni equivalenti per context-free. I linguaggi context-free sono Turing-riconoscibili (ovvero positivamente decidibili), ma la maggior parte dei linguaggi Turing-riconoscibili non sono context-free. Vedremo ora che linguaggi context-free sono decidibili: sappiamo invece che molti linguaggi Turing-riconoscibili non lo sono. Proveremo il seguente

Teorema. I linguaggi context-free sono decidibili.

Ci sono due modi per provare il teorema: il primo è provare la decidibilità senza preoccuparsi di fornire un algoritmo efficiente, il secondo è fornire un algoritmo efficiente.

La prima prova di decidibilità. Utilizzeremo le seguenti proprietà delle grammatiche: (i) ogni derivazione $A \Rightarrow w$ contiene S al massimo

all'inizio, e solo se $A=S$; (ii) l'unica derivazione della stringa vuota è $S \rightarrow \varepsilon$, e solo se $S \rightarrow \varepsilon$ sta in G ; (iii) Ogni derivazione $(A \Rightarrow w)$ di $w \in \Sigma^*$, $w \neq \varepsilon$ richiede $2n-1$ regole, dove n è la lunghezza di w .

(Come esempio, vediamo una prova del solo punto (iii), per induzione su n . Se $w=a$ ha lunghezza $n=1$ allora w viene dedotta da una regola $A \rightarrow a$, e in effetti abbiamo $2n-1=2*1-1=1$. Altrimenti la prima regola di $(A \Rightarrow w)$ è $(A \rightarrow BC)$ con $B, C \neq S$, e abbiamo $w=w_1w_2$, con w, w_1, w_2 di lunghezza n, n_1, n_2 , e con $n=n_1+n_2$ e $(B \Rightarrow w_1), (C \Rightarrow w_2)$. Abbiamo $n_1, n_2 > 0$ dato che $B, C \neq S$, quindi $w_1 \neq \varepsilon$ e $w_2 \neq \varepsilon$. Per ipotesi le derivazioni di w_1, w_2 da B, C richiedono $2n_1-1, 2n_2-1$ passi. Dunque la derivazione di w ne richiede $1+(2n_1-1)+(2n_2-1) = 2(n_1+n_2)-1 = 2n-1$.)

Esistono un numero finito di deduzioni di lunghezza $2n-1$, dunque basta controllarle tutte per sapere se posso dedurre w .

Questo metodo richiede una tape 1 che contenga una descrizione di G fatta come lista di transizioni, qualcosa di simile alla descrizione di una macchina di Turing attraverso la lista delle sue transizioni.

In una tape 2 scriviamo una generica possibile deduzione di lunghezza $2n-1$ di w nella forma di una lista di $2n-1$ coppie i, j di interi (scritti come stringhe binarie):

i_1 # j_1 #... # i_{2n-1} # j_{2n-1} ## (stringa su $0, 1, \#$)

(Vediamo brevemente perché usare una lista di coppie di interi. Nel nostro esempio $G=\{S \rightarrow AB, A \rightarrow AA, A \rightarrow a, B \rightarrow b\}$ numeriamo le 4 regole come segue: 0 per $S \rightarrow AB$, 1 per $A \rightarrow AA$, 2 per $A \rightarrow a$, 3 per $B \rightarrow b$. Scriviamo un passo di deduzione come $i\#j\#$, dove i è il numero della regola ($i=0,1,2,3$) e j è la posizione del simbolo a cui viene applicata. Abbiamo $j=0,1, \dots$, al massimo j vale $n-1=4-1=3$, dato che w ha n simboli e in una deduzione abbiamo stringhe al massimo stringhe di lunghezza la lunghezza n di w . Infatti nel caso di w non vuota, una regola non decresce mai il numero dei simboli e l'ultima regola genera w che ha n simboli)

Possiamo scrivere automaticamente nella tape 2 tutte deduzioni di lunghezza $2n-1$, e per ciascuna di esse controllarne sempre automaticamente sia la correttezza, sia quale stringa deduce. Se la deduzione corrente è corretta e deduce la stringa $w = w_1 \dots w_n$ allora accettiamo, altrimenti passiamo alla successiva. Se anche l'ultima deduzione non va bene rifiutiamo.

Questa prima soluzione ha il difetto che le deduzioni di lunghezza $2n-1$ che usano r regole su stringhe con un numero di simboli da 1 a n sono una quantità esponenziale in n . (Vediamo perché. Usiamo le tecniche di matematica discreta per stimare il numero delle liste della forma $\#i_1\#j_1\#\dots\#i_{2n-1}\#j_{2n-1}\#\#$. L'indice i , il numero della regola, varia tra 1 e r , l'indice j , la variabile a cui applichiamo la regola, varia tra 1 e n , dunque le coppie $i\#j\#$ sono al massimo $r*n$ e le successioni di $2n-1$ coppie di questo tipo sono al più $(r*n)^{(2n-1)}$. Invece, dato che j ha sempre almeno un valore, una stima inferiore è $(r*1)^{(2n-1)}$. Nel nostro esempio, una stima superiore è $(r*n)^{(2n-1)} = (4*4)^7 = 16^7 = 268.435.456$, mentre una stima inferiore è $(r*1)^{(2n-1)} = (4*1)^7 = 16.384$. Quale che sia il valore esatto, questo metodo richiede un numero di passi esponenziale in n , quindi è molto lento per valori grandi di n .)

La seconda prova di decidibilità. In questo caso costruiamo una tabella con tutte le stringhe z incluse consecutivamente in w e per ognuna di esse ci chiediamo per quali variabili A abbiamo che $(A \Rightarrow z)$, ovvero che " A deduce z ". Partiamo dalle sottostringhe di lunghezza 1 (i caratteri di w), poi consideriamo quelle di lunghezza 2, eccetera, fino ad arrivare a w come unica sottostringa di w di lunghezza n . Per il programmatore, questa soluzione è decisamente più laboriosa della precedente, ma è anche molto più veloce, perché memorizziamo e riutilizziamo tutta l'informazione sulla appartenenza ad L di sottostringhe di w che riusciamo a raccogliere. In questo modo evitiamo di dover ricalcolare la stessa informazione un numero esponenziale di volte. Inoltre ci concentriamo sulle sottostringhe consecutive di w , e non perdiamo tempo a dedurre stringhe z che non hanno nulla a che fare con la deduzione della stringa w che ci interessa.

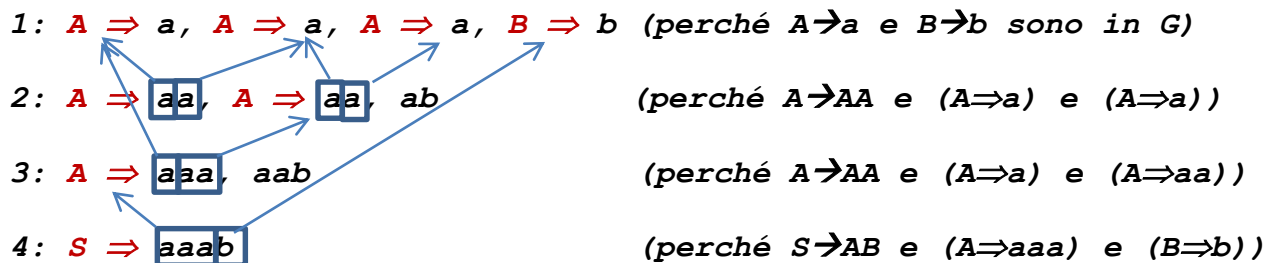
Mostriamo come fare per G come sopra e $w=aaab$ di lunghezza $n=4$. Le sottostringhe di w , ordinate per lunghezza crescente, sono:

- 1: a, a, a, b
- 2: aa, aa, ab
- 3: aaa, aab
- 4: aaab

Per ogni sottostringa z elenchiamo le variabili A tali che A derivi z . Se $z=c$ ha lunghezza $m=1$ (z ha un solo carattere), basta controllare se esiste la regola $A \rightarrow c$. Supponiamo che z abbia lunghezza m tale che $n \geq m > 1$, dove $n=4$ è la lunghezza di w . Prendiamo tutte le regole $A \rightarrow BC$

in G , con $A \neq S$ se $l < 4$, dato che sappiamo che non ci sono deduzioni che usano S in passi intermedi. Controlliamo se esiste una derivazione di $A \Rightarrow z$ che inizia per $A \Rightarrow BC$. Questo è possibile se e solo se z si può scomporre in $z = z_1 z_2$ con $B \Rightarrow z_1$ e $C \Rightarrow z_2$. Ci sono $(m-1)$ scomposizioni di z , con z_1 di lunghezza $1, \dots, m-1$. Per ogni scomposizione $z = z_1 z_2$ troviamo nelle righe precedenti le risposte alle domande $B \Rightarrow z_1$ e $C \Rightarrow z_2$: se le risposte sono positive annotiamo z con $A \Rightarrow z$. Nel solo caso che sia $m = n = \text{lunghezza di } w$, controlliamo se $S \Rightarrow w$.

Come esempio, prendiamo $w = aaab$ e $n = 4$. Per ogni riga inseriamo delle frecce tra una sottostringa e la riga che contiene l'informazione relativa a quella certa sottostringa. Otteniamo:



Nella riga $n = 4$ concludiamo che $w \in L(G)$. Controlliamo ora che, con questo metodo, il tempo di calcolo per decidere $w \in L(G)$ è polinomiale nella lunghezza n di w , quindi nettamente più veloce del precedente.

Prova che l'algoritmo qui sopra richiede tempo $O(n^3)$, supponendo la grammatica G fissa (considerando anche la dimensione di G arriviamo a $O(n^4)$). La struttura dati qui sopra ha n righe, e ogni riga ha tra 1 e n stringhe: dunque ci sono $O(n \cdot n)$ stringhe, tutte di lunghezza tra 1 e n . Ci sono n stringhe a di lunghezza $m = 1$ (di un carattere), e per ognuna di esse controlliamo se esiste una regola $A \Rightarrow a$ in G . Questo richiede tempo $O(n)$. Ci sono $O(n \cdot n) - O(n) = O(n \cdot n)$ stringhe s di lunghezza $m > 1$, per ognuna di esse proviamo $m-1$ scomposizioni $s = s' s''$ in due stringhe: s' e s'' hanno lunghezza 1 e $m-1$, 2 e $m-2$, ..., $m-1$ e 1. Dunque abbiamo $O(n)$ scomposizioni per ogni stringa s . A questo punto controlliamo tutte le regole $A \Rightarrow BC$ in G . Se abbiamo già scritto che $B \Rightarrow s'$ e che $C \Rightarrow s''$ nella struttura dati, allora aggiungiamo anche $A \Rightarrow s$ alla struttura dati. Dato che G ha dimensione costante, ogni controllo costa tempo $O(1)$, e controlliamo $O(n)$ scomposizioni per ognuna delle $O(n \cdot n)$ stringhe. Il costo totale è quindi $O(n \cdot n \cdot n)$ passi.

Indecidibilità delle relazioni di eguaglianza e ordine tra numeri reali

In programmazione utilizziamo delle operazioni su numeri reali: `==`, `!=`, `<`, `>`, `<=`, `>=` che ci dicono se due reali sono uguali, diversi, uno minore dell'altro eccetera. Tuttavia, è facile notare che le relazioni `==` e `!=` sono affidabili con probabilità vicina a piacere a 1, ma non con certezza. Proviamo a confrontare tra loro lo stesso numero reale ma calcolato in due modi diversi, per esempio $x=(1./3.)$ e $y=(x*100)/100$. Notiamo che di solito x e y vengono considerati diversi, a causa di leggere differenze di arrotondamento nei due calcoli. La ragione è che il linguaggio di programmazione chiama "numero reale" un numero binario con un numero finito di cifre dopo la virgola. La frazione $1/3$ non può essere rappresentata con esattezza con un numero finito di cifre binarie e deve venir troncata: infatti $1/3 = 0.01\ 01\ 01\ \dots$ in notazione binaria è un numero periodico. Quasi ogni calcolo comporta nuovi troncamenti e rende il risultato meno preciso. Non sempre $(x*100)/100$ vale esattamente x . Di conseguenza i test `==` e `!=`, quando vengono applicati a numeri reali, sono praticamente inutili: il primo risponde falso con probabilità vicina a 1, quale che sia la vera risposta, e il secondo ha il difetto opposto.

È possibile una rappresentazione esatta di un numero reale in programmazione? Sì, lo è, ma è molto costosa in termini di tempo di calcolo e spazio di memoria. Nel linguaggio Mathematica della Wolfram, nel caso più generale un numero reale è un programma che fornisce un numero di cifre decimali a piacere per il valore dato (oppure: un numero di cifre binarie a piacere eccetera). Per esempio, per il numero π possiamo chiedere quante cifre vogliamo, anche migliaia o milioni, e ci vengono rapidamente fornite. Solo l'ultima cifra di una rappresentazione è incerta e può cambiare aumentando il numero di cifre della rappresentazione (in un numero della forma 0.0799999 , può cambiare l'ultima cifra 7 prima del gruppo di tutti 9).

A essere precisi dovremmo dire: un programma rappresenta un numero reale r è un programma che prende in input un numero naturale n e restituisce due numeri razionali r_n e $r_n+(1/2^n)$ (approssimazioni di r

fino alla cifra binaria numero n), tali che il vero valore di r si trovi in mezzo ad esse:

$$r \in [r_n, r_n + (1/2^n)]$$

Se rappresentiamo un numero reale in questo modo (molto costoso e usato solo per scopi speciali), possiamo decidere l'eguaglianza tra reali usando un qualche programma? **No, perché vorrebbe dire decidere il problema della fermata.** Consideriamo una qualunque coppia $\langle M, w \rangle$, con M macchina di Turing e w input per M qualunque: M e w possono avere nessun rapporto con i numeri reali, ma ci consentono di definire un numero reale. **Definiamo un numero reale $X(M, w)$ (dunque un programma) nell'intervallo $[0, 1]$ come segue. $X(M, w)$ ha cifra binaria di posto n uguale a 0 se M applicata a w calcola per almeno n passi, ed ha cifra binaria di posto n uguale a 1 altrimenti, cioè se M applicata a w si ferma prima di n passi.** Possiamo calcolare la cifra n di $X(M, w)$ usando una macchina universale U che simula M applicata a w , e un contatore per il numero dei passi. Ci sono due possibilità per il numero reale $X(M, w)$. Possiamo avere

$$X(M, w) = 0.0000 \dots \text{(tutti 0)}$$

se M applicata a w non si ferma mai, e

$$X(M, w) = 0.0000 \dots 1111 \dots \text{(n volte 0, poi dopo tutti 1)}$$

se M applicata a w si ferma dopo n passi. Si tratta di un numero reale semplicissimo, eppure decidere se $X(M, w) == 0$ equivale a decidere se M si ferma su w . Dunque decidere la relazione $==$ per numeri reali qualunque, compresi i numeri reali $X(M, w)$ che abbiamo appena definito, implica la possibilità di decidere il problema della fermata, e quindi non è possibile.

Allo stesso modo possiamo provare che tutte le relazioni $!=, <, >, <=, >=$ non sono decidibili. Per esempio: per decidere se $x == y$ mi basta decidere se $x < y$ e $y < x$ sono entrambe false. Quindi, dato che $x == y$ non è decidibile, non lo è neppure $x < y$.

Se rappresentiamo i numeri reali con un numero finito di cifre binarie, possiamo osservare che le operazioni $x == y$, $x != y$, $x < y$, $x > y$, $x <= y$, $x >= y$ sono inaffidabili quando x , y sono molto vicini, perché in questo caso gli arrotondamenti influenzano la risposta. Quindi $x < y$, $x > y$, $x <= y$, $x >= y$ sono quasi sempre esatti, e in effetti ci sembrano esatti. Il risultato di non decidibilità ci dice che non sono esatti al 100%, ma questo non ci impedisce di avere quasi sempre la risposta

giusta. Invece, come abbiamo detto, l'operazione $x==y$ risponde quasi sempre "falso", anche quando x e y sono uguali, e $x!=y$ ha il difetto opposto.

Conclusione. Quando programiamo con i numeri reali, a rigore, non stiamo fornendo risposte certe, ma solo probabili, con *probabilità che possiamo rendere a piacere vicina a 1*, rendendo sempre più precisa la rappresentazione dei reali usati. Dobbiamo esserne consapevoli e fare attenzione a fornire risposte con probabilità di correttezza accettabile nel caso particolare considerato.

Es.5.3: Esempio del problema della corrispondenza di Post

Supponiamo di avere infinite copie di ciascuno dei seguenti 4 domini:

1	2	3	4
ab abab	b a	aba b	aa a

Trovare una successione anche ripetuta dei domini elencati qui sopra, in modo da avere la stessa parola sopra e sotto.

Soluzione: usare le copie dei domini 1-1-3-2-2-4-4 in quest'ordine. Notate come i domini 2 e 4 sono stati usati ciascuno due volte: abbiamo il diritto di farlo a norma delle regole del problema. Possiamo anche usare un domino più di due volte, oppure non usarlo affatto.

Il risultato è descritto qui sotto, prima con i domini nel formato originario, poi con gli stessi domini, ma deformati in modo da allineare le lettere della parola di sopra e della parola di sotto.

1	1	3	2	2	4	4
ab abab	ab abab	aba b	b a	b a	aa a	aa a

1	1	3	2	2	4	4					
a	b	a	b	a	b	b	a	a	a	a	= ababababbaaaa
a	b	a	b	a	b	b	a	a	a	a	= ababababbaaaa

Stabilire se esiste una soluzione oppure no per un dato insieme di domini è noto come *il problema della corrispondenza di Post*. Non esiste un programma in grado di risolvere il problema della corrispondenza di Post, in modo corretto e in tutti i casi. Questo non impedisce di trovare una soluzione in un caso particolare, come abbiamo appena fatto.

Una correzione per la prova sul Sipser dell'ind decidibilità del Problema di Post

Il libro di testo del corso, il Sipser, prova l'ind decidibilità del Problema della corrispondenza di Post provando prima l'ind decidibilità di MPCP, il problema della corrispondenza di Post modificato (vedi Teorema 5.15). Questa prova contiene un errore, l'unico che abbia trovato nel libro.

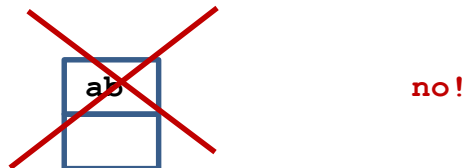
MPCP viene definito come segue:

MPCP = {(P) | P is an instance of the Post correspondence problem with a match that starts with the first domino}.

Tuttavia, nella prova viene in realtà utilizzata la seguente definizione, più lunga:

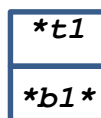
MPCP = {(P) | P is an instance of the Post correspondence problem with a match that starts with the first domino and with all words in the bottom of a domino non-empty}.

In altre parole nella prova del Sipser non si devono accettare domino della forma:



Infatti nella prova del Sipser che MPCP si riduce a PCP si utilizza il fatto che le parti inferiori dei domini non sono vuote. Succede quando si dice, alla fine del Teorema 5.15:

"the only domino that could possibly start a match is the first one,



because it is the only one where both the top and the bottom start with the same."

Tuttavia questo è vero solo se, per esempio, nel secondo domino

$*t2$
$b2*$

abbiamo $b2$ non vuota. Se $b2$ fosse vuota, il secondo domino sarebbe

$*t2$
$*$

e quindi potrebbe fare da domino iniziale avendo un simbolo $*$ sopra e sotto.

Nella prova del Sipser che A_{TM} si riduce a MPCP sono utilizzati solo dei domino con la parte inferiore non vuota, quindi si può cambiare la definizione di MPCP senza dover cambiare altro nella prova di 5.15.

La prova del Sipser è già implicitamente scritta secondo la nuova definizione di MPCP.

Un algoritmo pcp di ricerca cieca per le soluzioni di PCP

Definiamo un algoritmo pcp di ricerca cieca per le soluzioni a PCP. L'algoritmo pcp è un decisore positivo. pcp esplora l'albero di tutte le liste non vuote di domino a partire dalla lista P di domino dati, troncando le liste in cui le parole ottenute sopra e sotto hanno un carattere diverso, e le liste che superano un certo limite prefissato. Nel caso tutti i rami vengano troncati prima di raggiungere il limite di lunghezza pcp segnala che non esiste soluzione, ma questo avviene raramente. pcp invece è in grado di trovare qualunque soluzione per la lista P, purché la soluzione si trovi entro il limite di lunghezza scelto.

Nota. Per brevità, pcp non restituisce tutte le soluzioni entro il limite di lunghezza, ma solo quelle "minimali", quelle cioè che non hanno una soluzione più corta come prefisso. Ogni soluzione di PCP per un dato P si ottiene concatenando una o più soluzioni minimali per lo stesso P, quindi è sufficiente fornire le sole soluzioni minimali.

Questo algoritmo ha consentito di migliorare la soluzione **1-1-3-2-2-4-4** di lunghezza 7 trovata a mano per la seguente lista P di domino:

1	2	3	4								
<table><tr><td>ab</td></tr><tr><td>abab</td></tr></table>	ab	abab	<table><tr><td>b</td></tr><tr><td>a</td></tr></table>	b	a	<table><tr><td>aba</td></tr><tr><td>b</td></tr></table>	aba	b	<table><tr><td>aa</td></tr><tr><td>a</td></tr></table>	aa	a
ab											
abab											
b											
a											
aba											
b											
aa											
a											

La nuova soluzione trovata per gli stessi domino è **4-4-2-1**. Eccola:

4	4	2	1								
<table><tr><td>aa</td></tr><tr><td>a</td></tr></table>	aa	a	<table><tr><td>aa</td></tr><tr><td>a</td></tr></table>	aa	a	<table><tr><td>b</td></tr><tr><td>a</td></tr></table>	b	a	<table><tr><td>ab</td></tr><tr><td>abab</td></tr></table>	ab	abab
aa											
a											
aa											
a											
b											
a											
ab											
abab											

La nuova soluzione corrisponde alla parola "aaaabab", ha lunghezza 4 e non utilizza il domino numero 3. *Nota. Nella versione corrente pcp numera i domino da 0, dunque la soluzione trovata viene chiamata **3-3-1-0**.*

```

//SAVE AS PCP.java
/* Post Correspondence Problem (PCP)
We define a semi-algorithm with blind search for all possible matches
of a matrix "dominos" representing a list of dominos. Attempts above
a given bound are terminated, if required they are included in the
result as overflows */

import java.util.*; //For the Scanner class

public class PCP
{
//matrix nx2 of strings, each column is a domino
    private static String[][] dominos;
/* Example: dominos = { {"aa","a"}, {"b","ab"}, {"b","bb"} }
It represents:
                |aa| |b | |b |
                |a | |ab| |bb|
*/

//by default we do not include in the result the list of domino's
//which are past a given bound
    private static boolean skipOverflow=true;

//The map "blank" returning n times blank
public static String blank(int n)
{
    if (n<=0)
        return "";
    else
        return " "+blank(n-1);
}

/* The map dominos() returning a string with a drawing of the initial
set of dominos. If dominos = { {"aa","a"}, {"b","ab"}, {"b","bb"} }
we return the string:
|aa| |b | |b |
|a | |ab| |bb|
*/
public static String dominos()
{String result="\n";

```

```

for(int i=0;i<dominos.length;++i)
    result = result + " |" + dominos[i][0]
    + blank(dominos[i][1].length() - dominos[i][0].length()) + "|";
result=result+"\n";
for(int i=0;i<dominos.length;++i)
    result = result + " |" + dominos[i][1]
    + blank(dominos[i][0].length() - dominos[i][1].length()) + "|";
return result;
}

```

```

//The map match returning a commented result obtained from the
//algorithm pcp
    public static String match(int bound)
/*input:  bound = a maximum accepted length for a list of dominos
output:  list of matches within bound, plus all lists terminated
        because above bound, plus comments */
    {
        String msg = pcp("", "", "", bound);
        String overflow_msg;
        if (skipOverflow==false)
            overflow_msg = "\n Overflow messages INCLUDED";
        else
            overflow_msg = "\n Overflow messages NOT INCLUDED";
/* If there are no match and no overflow in m and overflows are
not skipped then REFUSE
If there is an accepting match in m then ACCEPT
Otherwise DONT KNOW */
        if (msg.equals("") && (skipOverflow==false))
            msg="\nAnswer: REFUSE";
        else if (msg.indexOf("ACCEPT")>=0)
            msg=msg+"\nAnswer: ACCEPT";
        else
            msg=msg+"\nAnswer: DONT KNOW";

        return "=====\n DOMINO TYPES"
        + dominos() + "\n Non-composed Matches with length<= "
        + bound + overflow_msg
        + "\n-----" + msg
        + "\n=====";
    }

```

```

//The map concatenateDominos(...) returning a drawing of the solution
    public static String concatenateDominos(String dominoNames)
/*Input:  a string with domino's names in the format:
        name1-blank-name2-blank-...
Output: a drawing of all dominos in string format.
Example. Input = "0 1 " (with initial dominos as before)
        Ouput = the string:
|aa|b|
|a|ab|
*/
{
//vector {name1,name2,...} of all domino's names
    String[] dominoVett = dominoNames.split(" ");
    int n = dominoVett.length;
//vector {n1,n2,...} of all domino's numbers
    int[] dominoNum = new int[n];
    for(int i=0;i<n;++i)
        dominoNum[i] = Integer.parseInt(dominoVett[i]);

//Top and bottom of all dominos in dominoNames, separated by "|"
    String line1="|", line2="|";
    for(int i=0;i<n;++i)
    {
        line1 = line1 + dominos[dominoNum[i]][0] + "|";
        line2 = line2 + dominos[dominoNum[i]][1] + "|";
    }
    return line1 + "\n" + line2;
}

//The algorithm pcg providing the result
    public static String pcg
        (String dominoNames, String topWord, String bottomWord, int bound)
/* input:
1. string dominoNames with domino's names in the format:
    name1-blank-name2-blank-...
2. topWord    =concatenation of all domino's top    in dominoNames
3. bottomWord=concatenation of all domino's bottom in dominoNames
output:
    a string including all minimal lists of dominos with topWord =
bottomWord, if requires all lists past the bound */

```



```

{
//if topWord and bottomWord have no common extension: empty string
    if (!topWord.startsWith(bottomWord) &&
        !bottomWord.startsWith(topWord) )
        return "";
//we skip or we return all domino's lists which are past the bound
    if (bound < 0)
        if (skipOverflow==true)
            return "";
        else
            return "\nOverflow with:" + dominoNames
                + " top:" + topWord + " bottom:" + bottomWord
                + "\n-----";

//if topWord and bottomWord have the same length then
// - if they are equal we return the dominos and more information
// - otherwise we return the empty list
    int t = topWord.length(), b = bottomWord.length();
    if ((t==b) && (t>0))
        if (topWord.equals(bottomWord))
            return
                "\nACCEPT MATCH:" + dominoNames + " WORD:" + topWord
                + "\n" + concatenateDominos(dominoNames)
                + "\n-----";
        else
            return "";
//Otherwise we compute pcg on all extensions of the domino's list
//with one more domino, then we concatenate the results
    String result="";
    for(int i=0;i<dominos.length;++i)
        result = result +
            pcg(dominoNames + i + " ",
                topWord + dominos[i][0],
                bottomWord + dominos[i][1],
                bound-1);

    return result;
}

```

```

public static void main(String[] args)
{

```

```

//A scanner pausing the execution until we press enter
Scanner pauseUntilEnter = new Scanner(System.in);

    System.out.println("EXAMPLE 1: no answer or one match found\n");
    String[][] Ex1 = { {"aa","a"}, {"b","ab"}, {"b","bb"} };
    dominos=Ex1;
    skipOverflow=false;
    System.out.println(match(1));
//    *** Now we skip all overflow messages ***
    System.out.println(match(4));

//We pause execution until we press enter
    pauseUntilEnter.nextLine();

    System.out.println("EXAMPLE 2: many matches found\n");
    skipOverflow=true;
    String[][] Ex2 =
    { {"ab","abab"}, {"b","a"}, {"aba","b"}, {"aa","a"} };
    dominos=Ex2;
    System.out.println(match(7));
    System.out.println(match(11));

//We pause execution until we press enter
    pauseUntilEnter.nextLine();

    System.out.println("EXAMPLE 3: refuse \n");
//    *** We include overflow messages again ***
    skipOverflow=false;
    String[][] Ex3 = { {"abc","ab"}, {"ca","a"}, {"acc","ba"} };
    dominos=Ex3;
    System.out.println(match(2));
    System.out.println("Remark. If the overflows are included, but
no match and no overflow are found then we refuse");
}
}

```

Indecidibilità del problema di tassellazione di Wang

Il problema di tassellazione di Wang consiste nel prendere un insieme finito di quadrati con lati colorati, per esempio:

$$W = \{ \begin{array}{|c|} \hline 1 \\ \hline \end{array} , \begin{array}{|c|} \hline 2 \\ \hline \end{array} , \begin{array}{|c|} \hline 3 \\ \hline \end{array} \}$$

Si richiede di formare con i quadrati in W una copertura di ogni quadrato del piano, in modo che due lati adiacenti abbiano colori uguali. I quadrati si possono duplicare a piacere oppure non usare affatto, ma non si possono ruotare né ribaltare.

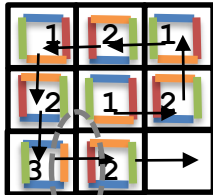
Questo problema è negativamente decidibile. Esiste una macchina di Turing M non deterministica che rifiuta in ogni ramo di calcolo se e solo se la copertura non esiste.

1. M usa una tape 1 con la lista delle descrizione dei quadrati e dei loro colori. Per ogni quadrato e ogni colore viene fornito un nome diverso, nella forma di un numero scritto in binario, per esempio 1,2,3 per i tre quadrati di W e 4,5,6,7 per i colori "rosso", "arancio", "verde", "blu". La descrizione di un quadrato consiste nel nome seguito dal colore della base e dagli altri colori in senso orario. Per esempio il primo quadrato di W diventa: 1, "arancio", "verde", "blu", "rosso", dove i nomi dei colori sono scritti come numeri in forma binaria.
2. M usa una tape 2 per scegliere in modo non deterministico una serie di nomi di quadrati, e due tape 3,4 per contenere rispettivamente le loro coordinate x,y . I quadrati vengono disposti a spirale a partire dal punto $(0,0)$, vedi dopo.
3. Dopo aver scelto un quadrato M lo confronta con tutti i quadrati scelti in precedenza, per esempio usando 3 diverse tape per immagazzinare nome, coordinata x , y del secondo quadrato. M controlla se il secondo quadrato ha coordinate adiacenti al primo: quando è così, M controlla che i due quadrati abbiano i due lati adiacenti dello stesso colore. M rifiuta se non lo sono, altrimenti M continua. Non è prevista accettazione.

Per esempio M sceglie i quadrati: 1,2,1,1,2,3,2. Per ognuno calcola le coordinate, e controlla se, nella figura ottenuta disponendo i quadrati a spirale, tutti quadrati adiacenti a un quadrato dato hanno i lati in comune dello stesso colore.

Nel caso della spirale di quadrati 1,2,1,1,2,3,2, l'ultimo quadrato aggiunto, di tipo 2, e il quadrato precedente ad esso, di tipo 3, hanno i due lati adiacenti di colore diverso:

percorso a spirale nel piano immaginato da M per i quadrati 1,2,1,1,2,3,2



questi due lati hanno colore diverso: arancio e rosso

Arrivata alla coppia di quadrati indicati, M rifiuta: due lati adiacenti hanno colore diverso. Per definizione di macchina non deterministica, M rifiuta se e solo se tutti i rami delle computazioni rifiutano. Come accennato nel corso (vedi esercizio 3.3 del Sisper), questo equivale a: esiste un livello dell'albero delle computazioni, per esempio il livello di numero $n=10$, in cui tutti i rami delle computazioni hanno rifiutato. Questo significa che ogni spirale dopo n passi contiene due lati adiacenti di colore diverso, e quindi una copertura di tutto il piano non può esistere.

Invece se esiste una copertura del piano con tutti i lati adiacenti con lo stesso colore, allora esiste una computazione di M che per caso sceglie i quadrati proprio come nella copertura, e dunque non trova mai una condizione di rifiuto e non termina mai. In questo caso esiste un ramo delle computazioni possibili di M che non rifiuta. Per definizione di macchina non deterministica, M non rifiuta.

Quindi M rifiuta se e solo se non esistono coperture del piano con tutti i lati adiacenti con lo stesso colore. Dunque M è un decisore negativo del problema di Wang.

Si prova (ma non lo vedremo nel corso) che il problema di Wang non è decidibile, perché il problema della non-fermata si riduce al problema di Wang, e il problema della non-fermata non è decidibile.

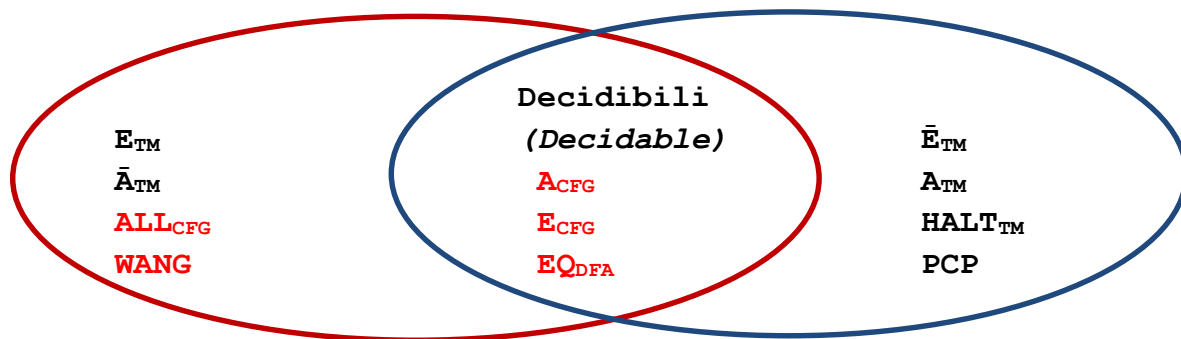
Sipser: elenco risultati di calcolabilità

In **rosso** i problemi che sono solo accennati, e non fanno parte del programma di esame. Per D.F.A intendiamo Deterministic Finite Automa, cioè una macchina di Turing che non sovrascrive, si sposta solo a destra e si ferma arrivata al primo blank.

E_{TM}	= Emptyness	for the language of a T.M.
ALL_{CFG}	= triviality for a Chomsky grammar (= <i>deriva ogni stringa</i>)	
\bar{A}_{TM}	= <u>non</u> -acceptance of an input	
WANG	= Wang tiling problem	
A_{CFG}	= acceptance	for a Chomsky grammar
E_{CFG}	= emptyness	for a Chomsky grammar
EQ_{DFA}	= Equality	for the languages of two D.F.A
\bar{E}_{TM}	= <u>non</u> -emptyness	for the language of a T.M.
A_{TM}	= acceptance	for the language of a T.M.
$HALT_{TM}$	= Halting Problem	for a T.M.
PCP	= Post Correspondence Problem	
$REGULAR_{TM}$	= Regularity	for the language of a T.M.
EQ_{TM}	= Equality	for the languages of two T.M.
$CONTEXT-FREE_{TM}$	= being equivalent to a Chomsky grammar for a T.M.	
ALL_{TM}	= triviality	for a T.M. (= <i>accetta ogni stringa</i>)

Negativamente decidibili
(Turing co-recognizable)

Positivamente decidibili
(Turing recognizable)



Né negativamente né positivamente decidibili

$REGULAR_{TM}$

EQ_{TM}

$CONTEXT-FREE_{TM}$

ALL_{TM}

(Le prove che $REGULAR_{TM}$, $CONTEXT-FREE_{TM}$ e ALL_{TM} non sono né positivamente né negativamente decidibili non fanno parte del corso. Questi casi sono aggiunti solo come esempi).

Problema 5.28: il Teorema di Rice

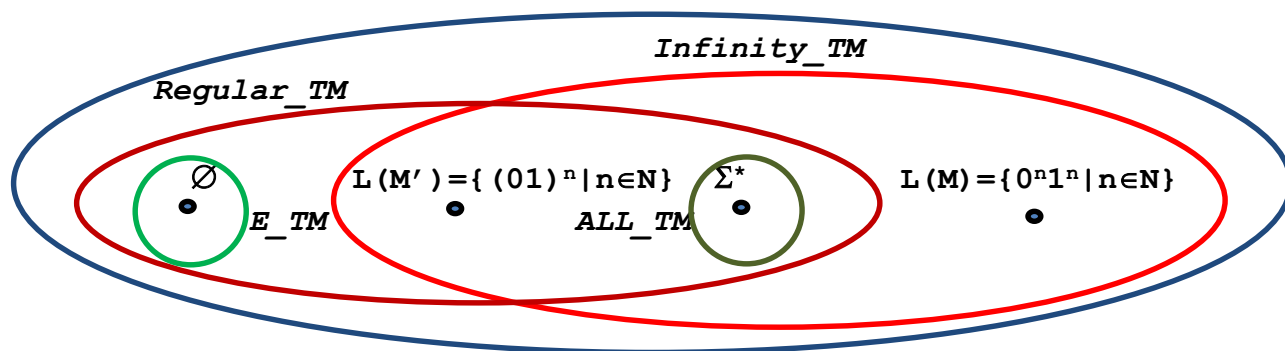
Problema 5.28. Il Teorema di Rice considera i linguaggi descritti da una qualche macchina di Turing e dice: **"solo le proprietà banali di questi linguaggi sono decidibili"**. Proviamo il Teorema di Rice.

Il primo passo è formulare il Teorema di Rice in modo esatto. Scegliamo un alfabeto Σ e consideriamo tutte le macchine di Turing M con alfabeto Σ . L'insieme delle stringhe di Σ accettate da M è un linguaggio, ovvero qualche sottoinsieme $L(M) \subseteq \Sigma^*$. Vogliamo considerare le proprietà di questi linguaggi e chiederci se possono essere decidibili. Tra queste proprietà abbiamo, per esempio: (i) $INFINITE_TM$: "il linguaggio $L(M)$ contiene infinite stringhe"; (ii) E_TM : "il linguaggio M è vuoto"; (iii) ALL_TM : "il linguaggio M contiene tutte le stringhe"; (iv) $REGULAR_TM$: "il linguaggio M è regolare".

Rappresentiamo un linguaggio L con una qualunque macchina M che lo definisce, tale che $L = L(M)$. Non tutte le proprietà delle macchine di Turing rappresentano proprietà di linguaggi. Non sono proprietà di linguaggi, per esempio: (a) il fatto che M abbia almeno 100 transizioni; (b) il fatto che M abbia al massimo 50 stati. Il motivo è che due macchine, una con più di 100 transizioni e una con meno di 100 possono definire lo stesso linguaggio, ma una soddisfa la proprietà e l'altra no. Allo stesso modo due macchine, una con al massimo 50 stati e una con più di 50 stati, possono definire lo stesso linguaggio.

Qui sotto rappresentiamo i linguaggi come punti, e le proprietà di linguaggi come diagrammi di Eulero-Venn.

Esempi di linguaggi e di proprietà di linguaggi
per l'alfabeto $\Sigma = \{0,1\}$



Per enunciare il Teorema di Rice, consideriamo un linguaggio $\Sigma' \supseteq \Sigma$, che consenta di descrivere le macchine di Turing di alfabeto Σ . Dunque Σ' contiene dei simboli per descrivere gli stati di una macchina M di alfabeto Σ , e il "tape alphabet" di M (i simboli che M aggiunge a Σ durante il calcolo). Indichiamo con $\langle M \rangle \in \Sigma'$ la stringa che descrive la macchina M in Σ' , e con $PROP \subseteq \Sigma'^*$ un insieme di stringhe che descrivono un insieme di macchine, dunque una proprietà delle macchine. Diciamo che $PROP$ è una proprietà di linguaggi se ha lo stesso valore di verità per due macchine M, M' che definiscono lo stesso linguaggio, cioè se:

(1) ogni volta che $\langle M \rangle \in PROP$ e $L(M) = L(M')$ allora $\langle M' \rangle \in PROP$

Per esempio, se $PROP = INFINITY_TM, E_TM, ALL_TM, REGULAR_TM$ allora $PROP$ è una proprietà di linguaggi in base alla nostra definizione. Invece se $PROP$ è l'insieme delle macchine di Turing M con almeno 100 transizioni, allora $PROP$ non è una proprietà di linguaggi. Se $PROP$ è una proprietà di linguaggi, allora il fatto che una macchina M soddisfi $PROP$ dipende dal linguaggio $L(M)$ definito da M , e non da M stessa.

Ci chiediamo quando $PROP$ possa essere decidibile: la risposta del Teorema di Rice è quasi mai.

Teorema di Rice. Se $PROP$ è una proprietà di linguaggi non banale, cioè se $PROP$ contiene qualche macchina di Turing ma non tutte, allora $PROP$ non è decidibile.

Alcune conseguenze del Teorema di Rice. $REGULAR_{TM}$, l'insieme delle $\langle M \rangle$ tali che M è una macchina di Turing e $L(M)$ è un linguaggio regolare, non è decidibile. Questo perchè esistono linguaggi regolari e linguaggi non regolari. Lo stesso vale per $INFINITE_{TM}$, l'insieme dei linguaggi con infinite stringhe, dato che esistono linguaggi infiniti e linguaggi non infiniti. E così per E_TM e per ALL_TM .

Prova del Teorema di Rice. Fissiamo una qualunque macchina M_0 che contenga tutte le stringhe sull'alfabeto Σ : dunque $L(M_0) = \Sigma^*$. Per ora proviamo il Teorema sotto l'ipotesi supplementare ($M_0 \in PROP$). Più tardi mostreremo come togliere questa ipotesi supplementare.

Sia A_{TM} il problema dell'accettazione per l'alfabeto Σ , dunque l'insieme delle coppie $\langle M, w \rangle$ con M macchina di Turing per Σ e w stringa di Σ tali che M accetta w . Dimostriamo che $A_{TM} \leq_m PROP$, cioè che esiste

una funzione calcolabile $f:A_{TM} \rightarrow PROP$ che preserva accettazione e non accettazione per A_{TM} . Dunque decidere PROP consentirebbe di decidere A_{TM} , e quindi non è possibile decidere PROP.

Per ipotesi esiste una macchina $M_1 \notin PROP$. Ora definiamo $f(\langle M, w \rangle) = M_2$, la macchina di Turing che prende un input $x \in \Sigma^*$ e simula in parallelo: $(M_1 \text{ su } x)$ e $(M \text{ su } w)$, e accetta non appena una delle due simulazioni accetta. Ci chiediamo quando abbiamo $\langle M_2 \rangle \in PROP$.

1. Supponiamo che M accetti w . Allora M_2 accetta ogni x : anche se M_1 non accettasse x , prima o poi M accetta w e M_2 accetta x . Dunque $L(M_2) = \Sigma^*$, e quindi $\langle M_2 \rangle \in PROP$ dato che PROP contiene $\langle M_0 \rangle$ di linguaggio Σ^* , e dunque tutte le macchine di linguaggio Σ^* .
2. Supponiamo che M non accetti w . Allora M_2 accetta esattamente le stringhe accettate da M_1 , dato che la simulazione di M su w non dà mai accettazione. Dunque $L(M_2) = L(M_1)$, e quindi $\langle M_2 \rangle \notin PROP$ dato che PROP non contiene M_1 , e dunque nessuna delle macchine che definiscono un linguaggio uguale a $L(M_1)$, e tra queste macchine c'è M_2 .

Abbiamo quindi provato che PROP non è decidibile. Tuttavia, abbiamo avuto bisogno di una ipotesi supplementare: $\langle M_0 \rangle \in PROP$. Supponiamo ora che $\langle M_0 \rangle \notin PROP$. Indichiamo con CPROP il complemento di PROP e proviamo che CPROP non è decidibile, provando che soddisfa tutte le ipotesi del Teorema, più $(\langle M_0 \rangle \in CPROP)$.

1. Dato che $M_0 \notin PROP$ abbiamo $\langle M_0 \rangle \in CPROP$.
2. Se $M \in CPROP$ e $L(M) = L(M')$ allora $\langle M' \rangle \in CPROP$, altrimenti avremmo $\langle M \rangle \notin PROP$ e $\langle M' \rangle \in PROP$, quindi PROP non sarebbe una proprietà di linguaggi.
3. PROP è una proprietà di linguaggi non banale, dunque esistono $\langle M \rangle \in PROP$ e $\langle M' \rangle \notin PROP$, quindi esistono $\langle M \rangle \notin CPROP$ e $\langle M' \rangle \in CPROP$.
Dunque CPROP è una proprietà di linguaggi non banale.

Per la prima parte della dimostrazione, CPROP non è decidibile, dunque anche PROP non lo è, dato che non lo è il suo complemento. Quindi il Teorema di Rice vale anche se $M_0 \notin PROP$. Questo completa la dimostrazione del Teorema di Rice.

Problemi 5.14 e 5.15: indecidibilità del problema dello spostamento fallito

Problema 5.14. Chiamiamo *spostamento fallito* in un passo di calcolo il tentativo di spostare la testina di lettura a sinistra della prima cella di memoria. Questo tentativo fallisce e la testina non si sposta. Invece lo stato del controllo e il simbolo letto vengono modificati come richiesto dalla transizione, e in seguito il calcolo continua, non si tratta di un errore fatale. Sia $FALLITO_{TM}$ l'insieme delle coppie $\langle M', w' \rangle$ di una macchina di Turing M' e un input w' tali che il calcolo di $\langle M', w' \rangle$ comporta almeno uno spostamento fallito. Notiamo che ci poniamo il problema dell'esistenza di uno spostamento fallito su un singolo input w' dato. Mostriamo che il problema dell'accettazione di M verso w si riduce, tramite una funzione calcolabile, al problema di stabilire l'esistenza di uno spostamento fallito per qualche M', w' : proveremo cioè che $A_{TM} \leq_m FALLITO_{TM}$. Dunque $FALLITO_{TM}$ non è decidibile dato che A_{TM} non lo è.

Definiamo quindi una funzione $f: A_{TM} \rightarrow FALLITO_{TM}$ calcolabile che riduce A_{TM} a $FALLITO_{TM}$. Definiamo f tale che: se M accetta w e se $f(\langle M, w \rangle) = \langle M', w' \rangle$, allora M' applicata a w' comporta uno spostamento fallito; viceversa, se M non accetta w allora M' applicata a w' non comporta uno spostamento fallito.

Descrizione di $\langle M', w' \rangle = f(\langle M, w \rangle)$. Sia M una macchina di Turing definita da:

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}} \rangle$$

Se $q_0 = q_{\text{accept}}$, M accetta subito. Allora definiamo M' come una macchina che fa uno spostamento fallito e insieme accetta.

Supponiamo ora che $q_0 \neq q_{\text{accept}}$: M non accetta subito. Definiamo M' e w' ponendo $w' = \#w$, dove $\#$ è un nuovo simbolo con $\# \notin \Gamma$. M' usa $\#$ (che non viene mai modificato né inserito altrove) per riconoscere l'inizio della tape, e per evitare così uno spostamento fallito. Quando avviene

una accettazione di M rispetto a w , la macchina M' entra in un nuovo stato $q_{\text{fallito}} \notin Q$, uno stato di attesa prima della accettazione. q_{fallito} sposta ripetutamente la testina di lettura a sinistra, fino all'inizio della tape, quindi esegue volutamente uno spostamento fallito prima di accettare.

La soluzione del Problema 5.14 potrebbe finire qui: tuttavia continuiamo e descriviamo M' in dettaglio, per convincerci che la funzione f che trasforma M in M' è calcolabile. Il linguaggio di M' contiene un simbolo $\#$ in più rispetto a quello di M . Le transizioni di q_{fallito} che producono uno spostamento fallito sono:

$\delta'(q_{\text{fallito}}, a) = (q_{\text{fallito}}, a, L)$ per ogni $a \neq \#$ mi sposto a sin.
finché non raggiungo $\#$

$\delta'(q_{\text{fallito}}, \#) = (q_{\text{accept}}, \#, L)$ spostamento fallito a sin. ($\#$ è
il primo simbolo della tape) e
accettazione

Definiamo una operazione g che pospone lo stato di accettazione q_{accept} in M' dopo q_{fallito} , come segue:

$$g = q \text{ se } q \neq q_{\text{accept}} \quad \text{e} \quad g = q_{\text{fallito}} \text{ se } q = q_{\text{accept}}$$

Definiamo M' come:

$$M' = \langle Q \cup \{q_{\text{fallito}}\}, \Sigma \cup \{\#\}, \Gamma \cup \{\#\}, \delta', q_0, q_{\text{accept}}, q_{\text{reject}} \rangle$$

Le transizioni δ' di M' sono quelle di M per $q \in Q$, con la variante che M' entra nello stato q_{fallito} prima di raggiungere q_{accept} . Siano $a \neq \#$, $q \in Q$, e $\delta(q, a) = (r, b, L)$ oppure $\delta(q, a) = (r, b, R)$. Allora M' si sposta nello stato \underline{r} anziché spostarsi nello stato r :

$$\delta'(q, a) = (\underline{r}, b, L) \text{ opp. } \delta'(q, a) = (\underline{r}, b, R) \quad (\underline{r} \text{ è definito sopra})$$

Inoltre M' "rimbalza" a destra senza cambiare stato quando trova il simbolo $\#$. In questo caso M fa uno spostamento fallito, invece M' no: M' simula il ritorno nella prima casella di w usando il simbolo $\#$ di fronte a w . Infine, all'inizio M' "supera" $\#$ restando nello stato q_0 . Per ottenere questi effetti poniamo

$$\delta'(q, \#) = (q, \#, R) \text{ per ogni } q \in Q \text{ ("rimbalzo" su } \# \text{ o "superamento" di } \#)$$

Il calcolo di M' applicata a w' avviene come il calcolo di M applicata a w , tranne che: (i) all'inizio M' supera $\#$, e (ii) in ogni spostamento fallito di M la macchina M' si sposta su $\#$ e torna indietro senza cambiare stato, simulando così lo spostamento fallito, ma senza eseguirne uno. Se e quando M accetta, M' entra nello stato q_{fallito} che continua a spostarsi a sinistra, poi esegue uno spostamento fallito. M' accetta in contemporanea al primo spostamento fallito.

Dunque $A_{\text{TM}} \leq_m \text{FALLITO}_{\text{TM}}$, quindi $\text{FALLITO}_{\text{TM}}$ non è decidibile.

Problema 5.15. Con una modifica (solo apparentemente) da poco, il problema dello spostamento fallito diventa *decidibile*. Sia $\text{SINISTRA}_{\text{TM}}$ l'insieme delle coppie $\langle M, w \rangle$ di una macchina di Turing M e un input w tali che il calcolo di $\langle M, w \rangle$ comporta almeno uno spostamento verso sinistra in una posizione qualunque della tape (dunque non necessariamente uno spostamento fallito, questa è la differenza rispetto a prima). Notiamo che ci poniamo il problema dell'esistenza di uno spostamento verso sinistra per un singolo input w dato. Mostriamo che $\text{SINISTRA}_{\text{TM}}$ è decidibile.

Sia n la lunghezza di w e sia m il numero di stati di M . Allora dopo $n+m+1$ spostamenti a destra il calcolo di M su w entra in ciclo e compie solo più spostamenti a destra. Infatti dopo n spostamenti a destra M legge solo più dei simboli \sqcup , e esegue solo transizioni

$$q \xrightarrow{\quad} \sqcup \xrightarrow{a, R} q'$$

Dopo m di queste transizioni sono comparsi $m+1$ stati, quindi c'è almeno una ripetizione di stato, dato che gli stati sono solo m . Quindi esiste uno stato q e una catena di transizioni da q a se stesso con sole transizioni a destra:

$$q \xrightarrow{\quad} \sqcup \xrightarrow{a, R} q' \xrightarrow{\quad} \sqcup \xrightarrow{a', R} q'' \xrightarrow{\quad} \dots \xrightarrow{\quad} q$$

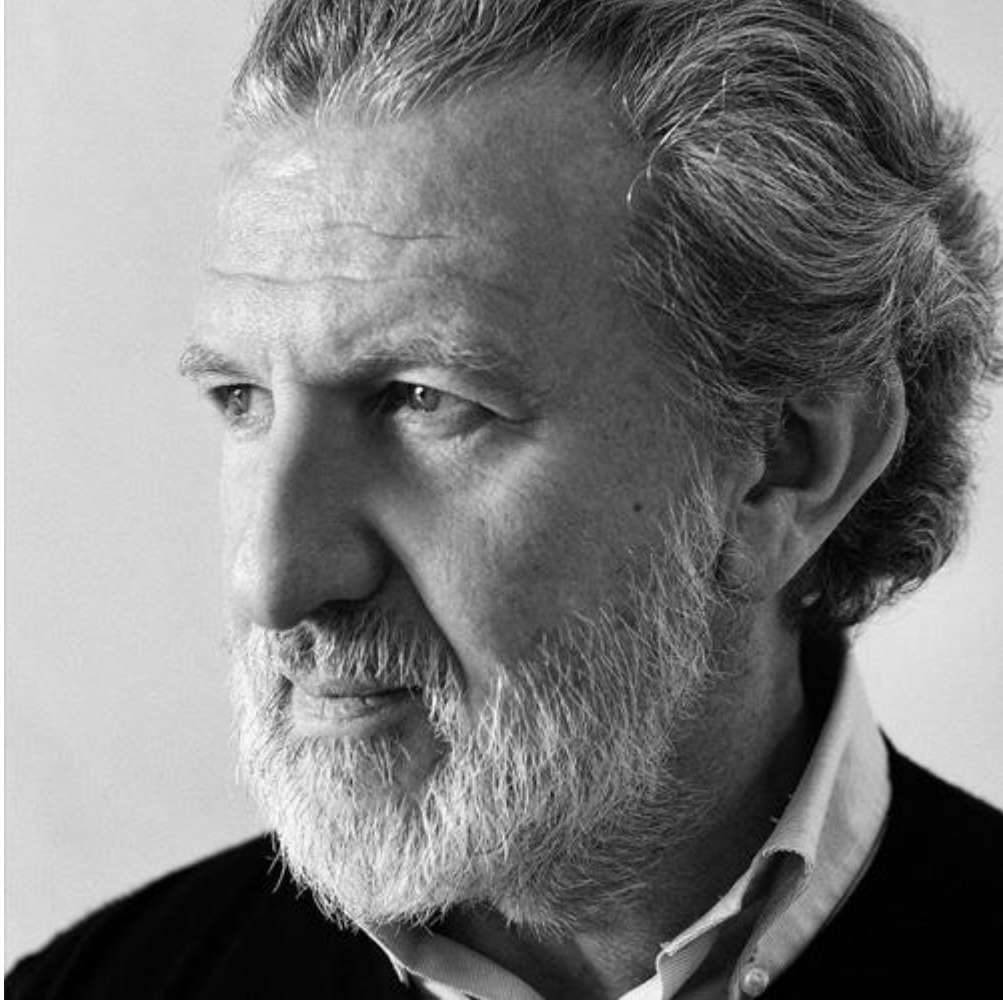
Da questo punto in poi questa catena di transizioni sarà ripetuta ciclicamente, per sempre, e M si sposterà per sempre a destra.

Dunque per decidere se esiste una transizione a sinistra nel calcolo di M applicata a w basta fornire $\langle M, w \rangle$ a una macchina universale con una tape in più che fa da contatore. Nel contatore contiamo gli m stati di M , poi gli n caratteri di w , dunque inseriamo $n+m+1$. Usiamo il contatore a ritroso per eseguire $n+m+1$ passi di calcolo (tutti i passi del calcolo, se il calcolo finisce prima).

- Se compare una transizione a sinistra allora ne esiste una.
- Se non compare, e il calcolo finisce prima di $n+m+1$ passi (accettando o rifiutando), allora non esistono transizioni a sinistra nel calcolo di M su w .
- Se eseguiamo $n+m+1$ transizioni tutte a destra, e troviamo 0 nel contatore, allora M entra in ciclo su input w , si sposta per sempre a destra, e non esistono transizioni a sinistra.

Dunque $\text{SINISTRA}_{\text{TM}}$ è decidibile.

Complessità



Il Prof. Piergiorgio Odifreddi, autore del trattato di Calcolabilità e Complessità "Classical Recursion Theory"

Forme normali congiuntive: un metodo per ottenerle e il tempo di calcolo richiesto

Qualche premessa. Una variabile booleana è una variabile x a valori 0,1. Una formula proposizionale F è ottenuta dalle variabili usando \neg (negazione), \vee (disgiunzione), \wedge (congiunzione). Indichiamo l'insieme delle formule proposizionali con PROP.

Un letterale è x oppure $\neg x$, dunque una variabile x booleana affermata oppure negata. Indichiamo i letterali con a, b, c, \dots . Una clausola è una disgiunzione C di letterali: $C = (a \vee b \vee c \vee \dots)$. Una formula F è una "una CNF-formula" (una formula in "conjunctive-normal form"), se F è una congiunzione di clausole: $F = C \wedge C' \wedge C'' \wedge \dots$. Indichiamo con CNF l'insieme delle CNF-formule. Una formula è in forma 3-CNF se è una congiunzione di clausole tutte di lunghezza 3. Indichiamo con 3-CNF l'insieme delle 3-CNF-formule.

La risoluzione automatica di problemi sulle formule proposizionali spesso richiede come primo passo la trasformazione di una formula in una CNF-formula (e a volte in una 3-CNF formula) ad essa equivalente. Vediamo in dettaglio il metodo per farlo descritto nel nostro libro di testo, e proviamo a stimare quanto tempo richiede. A volte, come vedremo adesso, questo metodo può rivelarsi eccessivamente costoso.

Trasformazione in forma normale congiuntiva. Vediamo un metodo per trasformare una formula proposizionale $F \in \text{PROP}$ in una formula equivalente $f(F) \in \text{CNF}$ normale congiuntiva. Innanzitutto applichiamo ad F le leggi di de Morgan $\neg(A \vee B) \leftrightarrow \neg A \wedge \neg B$ e $\neg(A \wedge B) \leftrightarrow \neg A \vee \neg B$ e la legge di doppia negazione $\neg \neg A \leftrightarrow A$ per trasportare tutte le negazioni accanto alle variabili: per esempio $F = \neg(\neg x \wedge (y \vee \neg z))$ diventa (1) $\neg \neg x \vee \neg(y \vee \neg z)$, poi (2) $x \vee (\neg y \wedge \neg \neg z)$ e infine (3) $x \vee (\neg y \wedge z)$. Ora F è fatta di variabili affermate o negate (dunque di letterali), di \wedge e \vee , la negazione non compare se non sulle variabili. A questo punto trasportiamo tutte le disgiunzioni dentro le congiunzioni usando la legge di distribuitività:

$$(A_1 \wedge \dots \wedge A_n) \vee (B_1 \wedge \dots \wedge B_m) \leftrightarrow \bigwedge_{1 \leq i \leq n, 1 \leq j \leq m} (A_i \vee B_j)$$

(porto la disgiunzione all'interno e la congiunzione all'esterno)

Il risultato è una formula $f(F)$ che contiene letterali e con disgiunzioni all'interno delle congiunzioni: dunque $f(F) \in \text{CNF}$. Per esempio, se $F = x \vee (\neg y \wedge z)$ allora $f(F) = (x \vee \neg y) \wedge (x \vee z) \in \text{CNF}$ (abbiamo portato l'unica disgiunzione all'interno). **La definizione ricorsiva di f è:**

- $f(a) = a$ se a è un letterale
- se $f(A) = C_1 \wedge \dots \wedge C_n$ e $f(B) = D_1 \wedge \dots \wedge D_m$ (congiunzioni di clausole) allora:
 1. $f(A \wedge B) = C_1 \wedge \dots \wedge C_n \wedge D_1 \wedge \dots \wedge D_m$
 2. $f(A \vee B) = \bigwedge_{1 \leq i \leq n, 1 \leq j \leq m} (C_i \vee D_j)$

Vediamo qualche altro esempio di calcolo di f :

1. $f(a_1 \wedge b_1) = a_1 \wedge b_1$
2. $f((a_1 \wedge a_2) \vee (b_1 \wedge b_2)) = \bigwedge_{1 \leq i \leq 2, 1 \leq j \leq 2} (a_i \vee b_j) = (a_1 \vee b_1) \wedge (a_1 \vee b_2) \wedge (a_2 \vee b_1) \wedge (a_2 \vee b_2)$
3. $f((a_1 \wedge a_2) \vee (b_1 \wedge b_2) \vee (c_1 \wedge c_2)) = f(\bigwedge_{1 \leq i \leq 2, 1 \leq j \leq 2} (a_i \vee b_j) \vee (c_1 \wedge c_2)) =$
 $(a_1 \vee b_1 \vee c_1) \wedge (a_1 \vee b_2 \vee c_1) \wedge (a_2 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_1) \wedge$
 $(a_1 \vee b_1 \vee c_2) \wedge (a_1 \vee b_2 \vee c_2) \wedge (a_2 \vee b_1 \vee c_2) \wedge (a_2 \vee b_2 \vee c_2)$

Questa trasformazione funziona bene su piccola scala, ma, come già si intravede nell'esempio numero 3, produce un risultato $f(F)$ di dimensione esponenziale nell'input F e non può essere usata su grandi formule. Infatti vediamo che $(A_1 \wedge \dots \wedge A_n) \vee (B_1 \wedge \dots \wedge B_m)$ è fatta di $n+m$ formule, ma viene trasformata dalla legge di distributività in $\bigwedge_{1 \leq i \leq n, 1 \leq j \leq m} A_i \vee B_j$ che è fatta di $n \times m$ formule. Ripetere questo passaggio produce un risultato esponenziale nella dimensione di F .

Un esempio di crescita esponenziale per $f(F)$. Partendo da questa osservazione possiamo trovare esempi di crescita esponenziale per $f(F)$. Per esempio sia $A_k = (a_{1,1} \wedge a_{1,2}) \vee \dots \vee (a_{k,1} \wedge a_{k,2})$ (con i letterali $a_{i,j}$ scelti a caso). A_k ha k congiunzioni e $2k$ letterali, invece in $f(A_k)$ abbiamo 2^k clausole, e i letterali sono ancora di più: $2^k \times k$.

Infatti, con un ragionamento per induzione possiamo far vedere quanto segue: le clausole in $f(A_k)$ sono $c = 2^k$, tutte e sole quelle della forma $C = (a_{1,i_1} \vee a_{2,i_2} \vee \dots \vee a_{k,i_k})$, con $1 = k$ letterali e per ogni possibile scelta di $i_1, \dots, i_k \in \{1, 2\}$. Dunque $f(A_k)$ ha $c \times 1 = 2^k \times k$ letterali in tutto. Ecco i dettagli:

- $f(A_1) = a_{1,1} \wedge b_{1,1}$ ha 2 clausole $a_{1,1}$ e $b_{1,1}$ con 1 letterale ciascuna.
- Se $f(A_k)$ ha c clausole $C_1 \wedge \dots \wedge C_c$ ciascuna con 1 letterali, allora $f(A_{k+1})$ ha 2 clausole $C_i \vee a_{k+1,1}$ e $C_i \vee a_{k+1,2}$ per ogni valore $1 \leq i \leq c$, dunque ha $2c$ clausole, ciascuna di lunghezza $l+1$.

Dunque a partire da formule F con qualche centinaio di letterali si possono ottenere formule $f(F)$ in CNF di dimensioni "impossibili". Per esempio, se A_k ha $k = 50, 100, 250, 500$ congiunzioni, e dunque ha $2k = 100, 200, 500, 1000$ letterali, allora $f(A_k)$ ha $2^k \times k = 2^{50} \times 50, 2^{100} \times 100, 2^{250} \times 250, 2^{500} \times 500$ letterali. Sono valori in crescita vertiginosa:

$$2^{50} \times 50 \approx 5,6 \times 10^{16}, \quad 2^{100} \times 100 \approx 1,2 \times 10^{32}, \quad 2^{250} \times 250 \approx 4,5 \times 10^{77}, \quad 2^{500} \times 500 \approx 1,6 \times 10^{153}$$

Per confronto, il numero di particelle nell'universo osservabile è oggi stimato da Google tra 10^{72} e 10^{87} . Scrivere la CNF di A_{250} , con $2^{250} \times 250 \approx 4,5 \times 10^{77}$ letterali, riempirebbe l'universo conosciuto, anche se riuscissimo a rappresentare i letterali con particelle elementari. La CNF di A_{500} ha $2^{500} \times 500 \approx 1,6 \times 10^{153}$ letterali e dunque, almeno in base a quanto ne sappiamo oggi sull'universo, non esisterà mai.

Riduzione polinomiale: da SAT a CNF-SAT

Soddisfacibilità. Una formula è soddisfacibile se esiste una sostituzione σ di valori 0,1 alle sue variabili che rende vera la formula. Possiamo immaginare σ come una funzione definita su tutte le variabili x : $\sigma(x) = 0$ oppure 1. σ viene estesa a tutte le formule booleane F usando le tavole di verità, quindi F è soddisfacibile se $\sigma(F)=1$ per qualche sostituzione σ . Un po' di terminologia:

- SAT è l'insieme delle $P \in \text{PROP}$ soddisfacibili. Dunque stabilire se $F \in \text{PROP}$ è il problema di stabilire se una formula booleana F è soddisfacibile.
- CNF-SAT è l'insieme delle $P \in \text{CNF}$ soddisfacibili. Dunque stabilire se $F \in \text{PROP}$ è il problema di stabilire se una CNF-formula F è soddisfacibile.
- 3-SAT è l'insieme delle $P \in \text{3-CNF}$ soddisfacibili. Dunque stabilire se $F \in \text{PROP}$ è il problema di stabilire se una 3-CNF-formula F è soddisfacibile.

Facciamo ora vedere che SAT si riduce polinomialmente a CNF-SAT che si riduce polinomialmente a 3-SAT.

SAT \leq_P CNF-SAT. Il libro di testo Sipser, che seguiamo, nel corollario 7.42 prova che CNF-SAT è NP-completo, e che dunque SAT, essendo in NP, si riduce polinomialmente a CNF. Qui forniamo una prova diretta di SAT \leq_P CNF-SAT presa da Wikipedia, con qualche modifica.

La trasformazione $f: \text{SAT} \rightarrow \text{CNF}$ che abbiamo visto nell'esempio precedente preserva l'equivalenza e dunque la soddisfacibilità, ma produce un output di dimensioni esponenziali nell'input, e a maggior ragione richiede un tempo di calcolo esponenziale. Definiamo un'altra trasformazione $g: \text{SAT} \rightarrow \text{CNF}$ che preserva la soddisfacibilità, non l'equivalenza questa volta: F e $g(F)$ non hanno gli stessi valori di verità per gli stessi valori delle variabili. Il calcolo di g richiede un tempo polinomiale e avviene come segue. Usando de Morgan, trasformiamo F in una formula fatta di di letterali a_1, \dots, a_L , di \wedge e \vee . Per ogni \vee in F introduciamo una nuova variabile t_1, \dots, t_v . Il significato intuitivo di t_j è: il lato destro della disgiunzione numero

j è vero. Definiamo $g(F) = C_1 \wedge \dots \wedge C_L$, dove $C_i = a_i \vee b_1 \vee \dots \vee b_v$, e $b_j = t_j$ se a_i si trova nel lato sinistro della disgiunzione numero j , e $b_j = \neg t_j$ altrimenti. Per esempio se $F = x \vee (\neg y \wedge z)$, allora $L=3$ (ci sono 3 letterali $x, \neg y, z$ in F) e $v=1$ (c'è una sola disgiunzione in F). Introduciamo una variabile t per l'unica disgiunzione in F . Il significato intuitivo di t è: il lato destro $(\neg y \wedge z)$ dell'unica disgiunzione è vero. Dato che x si trova nel lato sinistro della disgiunzione, e $\neg y, z$ nel lato destro, otteniamo

$$g(F) = (x \vee t) \wedge (\neg y \vee \neg t) \wedge (z \vee \neg t)$$

Se F ha lunghezza n allora $g(F)$ ha lunghezza $L \times v = O(n) \times O(n) = O(n^2)$: aggiungiamo tanti letterali quanto (il numero dei letterali in F) \times (il numero delle disgiunzioni in F). Il tempo di calcolo di g è polinomiale. Infine se σ soddisfa F allora σ si estende a una assegnazione che soddisfa $g(F)$, e viceversa se σ soddisfa $g(F)$ allora σ soddisfa F .

Per esempio siano $F = x \vee (\neg y \wedge z)$ e $g(F) = (x \vee t) \wedge (\neg y \vee \neg t) \wedge (z \vee \neg t)$. Analizzando F e $g(F)$ ci rendiamo conto che se F è soddisfacibile allora lo è anche $g(F)$, e viceversa, anche se le formule non sono equivalenti.

- Supponiamo che σ renda vera F . Se σ rende vero il lato sinistro di F , allora poniamo $t = \text{false}$ e rendiamo vere anche le clausole $(\neg y \vee \neg t)$ e $(z \vee \neg t)$ per i letterali del lato destro. Viceversa se σ rende vero il lato destro di F poniamo $t = \text{true}$ e rendiamo vera la clausola $(x \vee t)$ nel lato sinistro di F .
- Supponiamo invece che σ renda vera $g(F)$. Allora se $t = \text{false}$ allora σ rende vera x nel lato sinistro di F , e se $t = \text{true}$ allora σ rende vere $\neg y$ e z nel lato destro di F . In entrambi i casi σ rende vera F .

Partendo da questo esempio possiamo scrivere un ragionamento per induzione sulla lunghezza di F , e provare che:

- se σ rende vera F allora σ si estende a una assegnazione che rende vera $g(F)$,
- se σ rende vera $g(F)$ allora σ rende vera F .

Nota. Se vogliamo ottenere una formula più breve, usando la distribuitività del \vee possiamo alla fine raccogliere nella stessa clausola di $g(F)$ tutte le variabili a_i associate allo stesso gruppo di variabili $b_1 \vee \dots \vee b_v$. Questo però non è essenziale.

Riduzione polinomiale: da CNF-SAT a 3-SAT

CNF-SAT \leq_p 3-SAT. Il libro di testo Sipser nel corollario 7.42 afferma che **CNF-SAT è polinomialmente riducibile a 3-SAT**. Sipser include una parte della prova. Qui includiamo la prova completa.

Per definizione di riducibilità, per provare che $\text{CNF-SAT} \leq_p \text{3-SAT}$ dobbiamo definire una trasformazione $h : \{\text{CNF-formule}\} \rightarrow \{\text{3-CNF formule}\}$, tale che:

1. h si calcola in tempo polinomiale nella dimensione di F
2. se F è soddisfacibile allora $h(F)$ è soddisfacibile,
3. se F non è soddisfacibile allora $h(F)$ non è soddisfacibile.

Sia $F = C \wedge C' \wedge C'' \wedge \dots$: definiamo $h(C) \in \text{3-CNF}$ per ogni clausola C , quindi porremo $h(F) = h(C) \wedge h(C') \wedge h(C'') \wedge \dots \in \text{3-CNF}$. Sia $C = (a_1 \vee \dots \vee a_n)$. **Se $n=1,2$ definiamo $h(C)$ aggiungendo una o due volte " $\vee a_1$ ":** se $C = (a_1 \vee a_2)$ allora $h(C) = (a_1 \vee a_2 \vee a_1)$. Se $n=3$ poniamo $h(C) = C$. In questi casi $h(C)$ è equivalente a C : per qualunque sostituzione σ abbiamo $\sigma(C) = \sigma(h(C))$. Dunque C è soddisfacibile se e solo se lo è $h(C)$. Ora supponiamo $n \geq 4$. Come primo passo del calcolo di h , rimpiazziamo C con $F' = C' \wedge C'' \in \text{CNF}$, dove per qualche nuova variabile z abbiamo:

- $C' = (a_1 \vee \dots \vee a_{n-2} \vee z)$ (n-1 letterali)
- $C'' = (\neg z \vee a_{n-1} \vee a_n)$ (3 letterali)

Il significato intuitivo di z è $a_{n-1} \vee a_n$. Allora ogni sostituzione σ per le variabili di C che renda vera C si può estendere alla nuova variabile z in modo da rendere vera $F' = C' \wedge C''$, e viceversa ogni sostituzione che rende vera F' rende vera C . Infatti:

1. se $\sigma(C)=1$ allora $\sigma(a_i)=1$ per qualche i . Possiamo avere $i = 1, \dots, n-2$ oppure $i = n-1, n$. Nel primo caso $\sigma(C')=1$, e possiamo ottenere $\sigma(C'')=1$ scegliendo $\sigma(z)=0$, dunque $\sigma(\neg z)=1$. Nel secondo caso $\sigma(C'')=1$, e possiamo ottenere $\sigma(C')=1$ scegliendo $\sigma(z)=1$. In entrambi i casi possiamo estendere σ a z in modo tale che sia $\sigma(F')=1$.

2. Viceversa, se $\sigma(F')=1$, allora se $\sigma(z)=0$ dato che $\sigma(C')=1$ e $C' = (a_1 \vee \dots \vee a_{n-2} \vee z)$ abbiamo $\sigma(a_i)=1$ per qualche $i=1, \dots, n-2$. Se $\sigma(z)=1$ allora $\sigma(\neg z)=0$, e dato che $\sigma(C'')=1$ e $C''=(\neg z \vee a_{n-1} \vee a_n)$ abbiamo $\sigma(a_i)=1$ per qualche $i=n-1, n$. In entrambi i casi $\sigma(a_i)=1$ per qualche $i=1, \dots, n$, dunque $\sigma(C)=1$.

Definiamo $h(C)$ ripetendo la trasformazione da C a $F' = C' \wedge C''$ sulla prima clausola C' di lunghezza $n-1$ (la clausola C'' ha già lunghezza 3), e poi ripetendola ancora, finché non otteniamo solo clausole di 3 letterali. Dato che a ogni passo la prima clausola di F' passa da n a $n-1$ letterali, dobbiamo fare $(n-3)$ passi. Dato che ogni passo di h ha evidentemente un costo polinomiale, h è calcolabile in tempo polinomiale.

A ogni passo del calcolo di h , otteniamo una CNF-formula G soddisfacibile da una estensione della sostituzione σ che soddisfa C , e viceversa ogni sostituzione che soddisfa questa G soddisfa C . Alla fine otteniamo una 3-CNF formula $G=h(C)$, e poi una 3-CNF formula $h(F)$. Per costruzione, $h(F)$ è soddisfacibile da una estensione di σ se e solo σ soddisfa F . Dunque h riduce CNF a 3-CNF in tempo polinomiale.

Breve nota su 3-SAT e VERTEX-COVER

Forniamo una descrizione alternativa di 3-SAT, l'insieme delle formule soddisfacibili e in forma normale 3-congiuntiva, dunque congiunzioni di clausole tutte di lunghezza 3. Ricordiamo che una clausola è una disgiunzione di letterali, ovvero di variabili affermate o negate. Come esempio scegliamo $\phi = C_1 \wedge C_2 \wedge C_3$, dove

$$C_1 = (X_1 \vee X_2 \vee X_2), C_2 = (\neg X_1 \vee \neg X_2 \vee \neg X_2), C_3 = (\neg X_1 \vee X_2 \vee X_2)$$

(Fig.1)

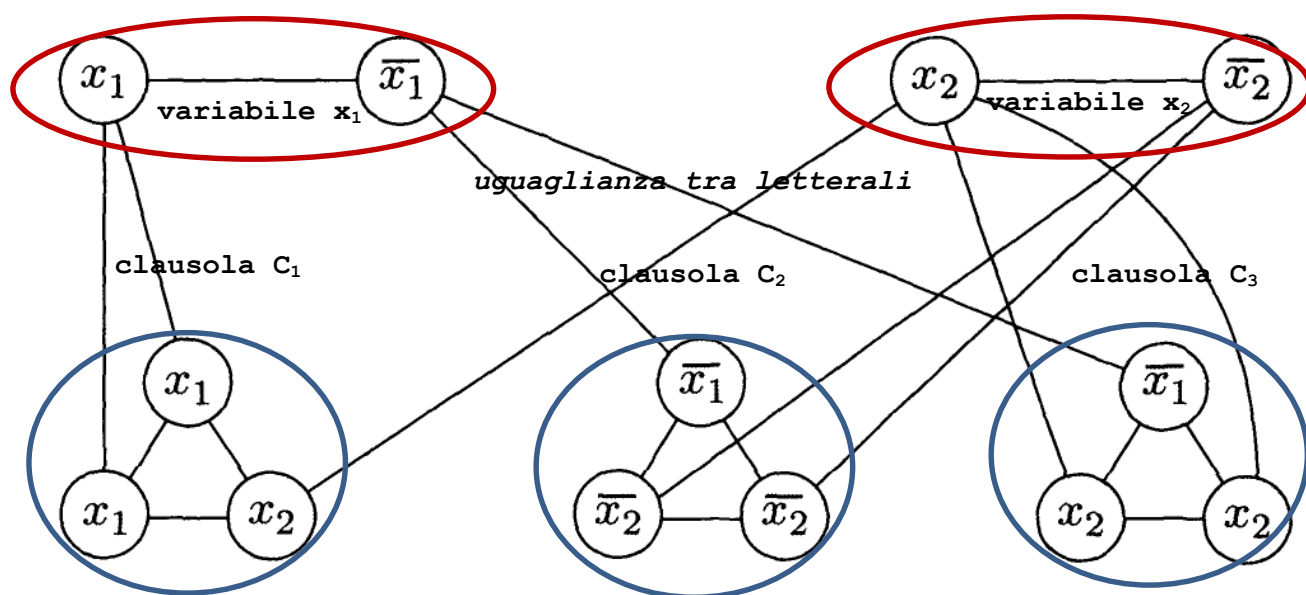
Le clausole C_1, C_2, C_3 hanno tutte 3 letterali. Ricordiamo che ϕ è detta soddisfacibile se esiste una funzione σ che assegna a ogni variabile x in ϕ un valore di verità $\sigma(x)$ ="vero" oppure $\sigma(x)$ ="falso" in modo che il valore di verità di ϕ (calcolato a partire dal valore di verità delle variabili usando le tavole di verità) sia "vero".

Dato che ϕ è una congiunzione di clausole, dobbiamo rendere vera ogni clausola; e dato che ogni clausola è una disgiunzione, dobbiamo rendere vero un letterale per clausola. Se $\phi = C_1 \wedge \dots \wedge C_h$ dobbiamo quindi poter scegliere dei letterali a_1 in C_1, \dots, a_h in C_h tali che se $a_i = X$ allora $\sigma(X)$ ="vero" e se $a_i = \neg X$ allora $\sigma(X)$ ="falso". Dunque non possiamo scegliere due letterali a_i, a_j l'uno la negazione dell'altro, dato che questi due letterali non potrebbero essere entrambi veri. Viceversa, se riusciamo a scegliere dei letterali a_1, \dots, a_k , uno per clausola in modo tale che non ci siano due letterali uno la negazione dell'altro, allora possiamo trovare una funzione σ che li renda tutti veri, ponendo: se $a_i = X$ allora $\sigma(X)$ ="vero" e se $a_i = \neg X$ allora $\sigma(X)$ ="falso". σ rende vero un letterale per clausola e dunque rende vera ϕ . Definiamo $\sigma(X)$ in modo arbitrario se non ci sono a_i tali che $a_i = X, \neg X$. Abbiamo quindi provato una caratterizzazione di 3-SAT:


"una formula 3-SAT è soddisfacibile se è possibile scegliere un letterale x_j oppure $\neg x_j$ per clausola in modo tale che non ci siano due letterali uno la negazione dell'altro (in modo tale che si possano rendere tutti veri)."

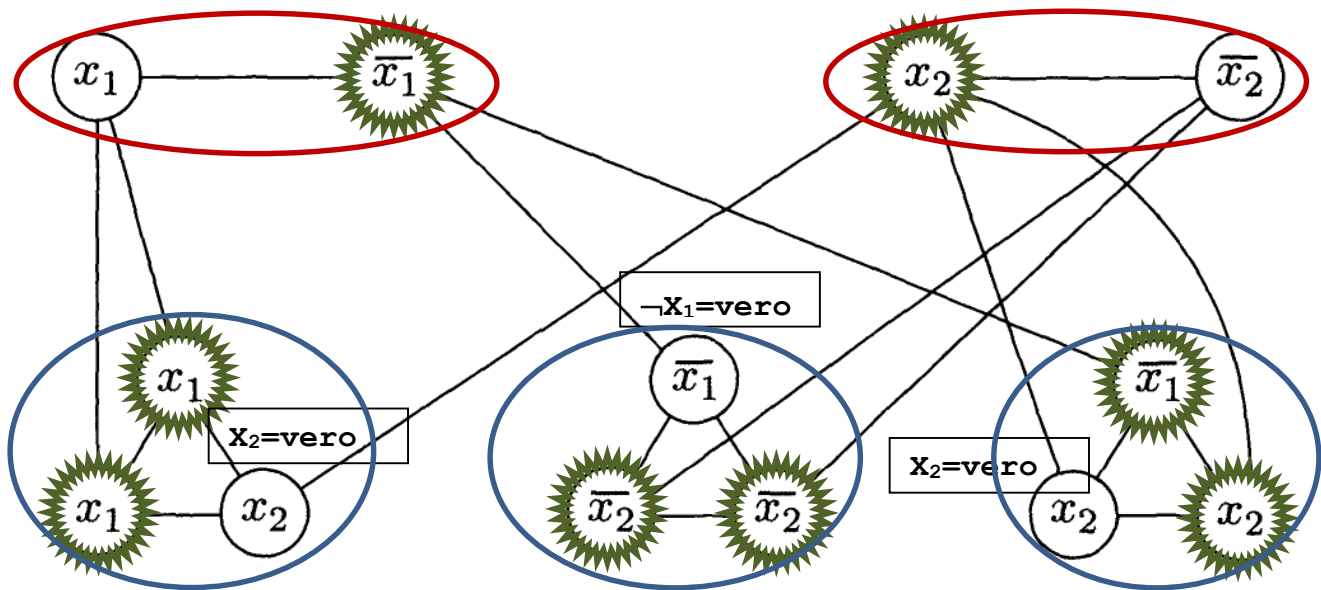
Usiamo questa caratterizzazione di 3-SAT per descrivere la prova del Sipser che $3SAT \leq_P VERTEX-COVER$. Dato un grafo G e un intero k , il problema VERTEX-COVER ci chiede se esiste un insieme di k vertici $C = \{v_1, \dots, v_k\}$ in G , detto copertura (degli archi di G), cioè tale che ogni arco di G inizi o finisca in qualche $v \in C$. Per esempi di coperture, guardate su Wikipedia oppure più sotto.

Ora riprendiamo la formula $\phi = C_1 \wedge C_2 \wedge C_3$ della figura 1, e consideriamo la traduzione del problema della soddisfacibilità per ϕ in un problema $\langle G, k \rangle$ di vertex-cover fornita dal testo Sipser. Questa traduzione trasforma ogni clausola di 3 letterali di ϕ in 3 punti connessi a due a due, e ogni variabile x di ϕ in una coppia $x, \neg x$ connessa da un arco. Il punto non coperto nella tripla rappresenta il letterale che vogliamo rendere vero nella clausola, il punto coperto nella coppia rappresenta il letterale vero tra x e $\neg x$. Alla fine aggiungiamo un arco tra ogni letterale l di ogni clausola e il letterale uguale nella coppia $x, \neg x$. Ecco il grafo G ottenuto a partire da ϕ (con in rosso le variabili e in blu le clausole):



Come k scegliamo ($2 \cdot \text{numero clausole} + \text{numero delle variabili}$), in questo caso $k = 2 \cdot 3 + 2 = 6 + 2 = 8$. Sia C una copertura di G di k elementi. Per coprire tre archi posti a triangolo dobbiamo avere almeno due vertici in C , dato che un solo vertice non copre l'arco a lui opposto. Per coprire un arco tra $x, \neg x$ dobbiamo avere almeno uno tra $x, \neg x$ in C . Dunque abbiamo bisogno di almeno 2 vertici per ogni triangolo che traduce una clausola, e di almeno un vertice per ogni arco tra $x, \neg x$. Dato che $k = (2 \cdot \text{clausole} + \text{variabili})$, dobbiamo scegliere esattamente 2 vertici per triangolo e esattamente un nodo per ogni arco tra $x, \neg x$, se ne mettiamo di più da una parte non ci bastano da un'altra parte. Dunque ci manca esattamente un nodo v_i per triangolo, dunque esattamente un letterale per clausola è scoperto. v_i è connesso a un nodo $v = x, \neg x$, e questo nodo deve essere coperto, altrimenti l'arco che connette v_i, v non lo è. Inoltre non è possibile scegliere due letterali opposti nelle clausole, altrimenti dovremmo coprire sia x che $\neg x$ nella parte sopra di G , mentre in ogni coppia $x, \neg x$ possiamo coprire esattamente un vertice. I nodi di C sono: i letterali in ciascuna clausola C_i che sono diversi da v_i , più i letterali uguali a qualche letterale v_i nella riga delle variabili, più un letterale a caso per ogni coppia $x, \neg x$ in cui non abbiamo fatto una scelta (per esempio, sempre x).

Dunque scegliere una copertura $C = \{v_1, \dots, v_k\}$ di k elementi in G corrisponde esattamente a scegliere un letterale v_i per clausola, letterale che non inseriamo in C , e senza mai scegliere due letterali opposti in due clausole diverse. Dunque scegliere una copertura corrisponde a scegliere k letterali in ϕ , uno per clausola e senza letterali opposti, in modo da poter rendere vera ϕ . Come esempio, qui sotto vediamo la copertura che corrisponde a lasciare scoperto: X_2 in C_1 , $\neg X_1$ in C_2 , X_2 in C_3 , e a coprire esattamente tutti i nodi adiacenti ai 3 letterali scelti. Ogni nodo coperto è cerchiato con una stella, come segue: 











Questa copertura corrisponde all'assegnazione: $\neg x_1 = \text{vero}$, $x_2 = \text{vero}$, che rende vera ϕ .

Quindi le coperture C di k elementi in G definiscono una scelta di un letterale per clausola che non scelgono letterali opposti in clausole diverse. Per la caratterizzazione fornita per 3-SAT, le coperture di k elementi definiscono delle assegnazioni che rendono vera ϕ , e viceversa. Dunque abbiamo ridotto (e polinomialmente) 3-SAT a VERTEX-COVER.

Problema 7.31: SOLITAIRE

Problema 7.31. "In the following solitaire game, you are given an $n \times m$ board. On each of its $n \times m$ positions lies either a blue stone, a red stone, or nothing at all. You play by removing stones from the board so that each column contains only stones of a single color (and possibly none) and each row contains at least one stone. You win if you achieve this objective. Winning may or may not be possible, depending upon the initial configuration. Let $SOLITAIRE = \{(G) \mid G \text{ is a winnable game configuration}\}$. Prove that $SOLITAIRE$ is NP-complete."

Un esempio di SOLITAIRE e di partita: ✕ indica una pietra tolta.

Abbiamo lasciato le sole pietre sulla diagonale. Restano righe tutte con una pietra, e colonne tutte dello stesso colore (blu, rosso e blu). Dunque abbiamo vinto la partita.

Proveremo che $3\text{-SAT} \leq_P SOLITAIRE$: dato che 3-SAT è NP-completo ne seguirà che SOLITAIRE è NP-completo, come richiesto dall'esercizio.

Sia $\phi = (X_1 \vee X_2 \vee X_3) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3) \wedge (\neg X_1 \vee X_2 \vee X_3)$ una forma normale congiuntiva con tutte clausole di lunghezza 3. Trasformiamo ϕ in $G=f(\phi)$ in modo tale che ϕ è soddisfacibile se e solo se G ha una strategia vincente.

Come primo passo, eliminiamo ogni clausola che contenga sia X che $\neg X$. Questo passo richiede un tempo polinomiale e le clausole eliminate sono sempre vere, dunque se le eliminiamo otteniamo una formula equivalente.

Resta una formula ϕ di n clausole C_1, \dots, C_n e m variabili X_1, \dots, X_m distinte. Per la modifica fatta, nessuna C_i può contenere sia X_j che $\neg X_j$. Ora definiamo una tabella $G=f(\phi)$ di $n \times m$ caselle. Poniamo righe=clausole, colonne=variabili, blu= X_j , rosso= $\neg X_j$. Nella riga i , colonna j , poniamo una pietra blu se la clausola C_i contiene X_j , una pietra rossa se C_i contiene $\neg X_j$. La trasformazione è polinomiale: proviamo che preserva accettazione e rifiuto.

Ricordiamo prima che: "una formula 3-SAT è soddisfacibile se è possibile scegliere un letterale X_j oppure $\neg X_j$ per clausola in modo tale che non ci siano due letterali uno la negazione dell'altro (in modo tale che si possano rendere tutti veri)."

Se ϕ è soddisfacibile allora G ha una strategia vincente. Scegliamo una assegnazione che rende vera ϕ . Scegliamo un letterale vero per clausola evitando di scegliere sia un letterale che la sua negazione. Se scegliamo X_j eliminiamo tutte le pietre rosse dalla colonna j , se scegliamo $\neg X_j$ eliminiamo tutte le pietre blu dalla colonna j . Adesso ogni colonna j contiene solo pietre dello stesso colore (corrisponde al valore di verità di X_j), e può essere vuota. Ogni riga i contiene almeno una pietra (blu corrisponde al letterale X_j in C_i , oppure rossa corrispondente al letterale $\neg X_j$ in C_i). Dunque G ha una strategia vincente.






Se G ha una strategia vincente, allora ϕ è soddisfacibile. Se la strategia rende la colonna j tutta blu poniamo $X_j = \text{vero}$, se restano pietre rosse poniamo $\neg X_j = \text{vero}$. Se la colonna è vuota scegliamo a caso il valore per X_j . Dato che la strategia è vincente, ogni riga (corrispondente a una clausola C_i) contiene almeno una pietra, che corrisponde a un $X_j = \text{vero}$ in C_i se la pietra è blu, e a un $\neg X_j = \text{vero}$ in C_i se la pietra è rossa. Dunque con la scelta fatta ogni clausola contiene un letterale vero.

Come esempio vediamo il caso $\phi = (X_1 \vee X_2 \vee X_2) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3) \wedge (\neg X_1 \vee X_2 \vee X_3)$. Non ci sono clausole con X , $\neg X$, dunque applichiamo subito la trasformazione di ϕ in una scacchiera $G = f(\phi)$ con pietre rosse e blu. Otteniamo la seguente scacchiera:

ϕ	X_1	X_2	X_3
$C_1 = X_1 \vee X_2 \vee X_2$	X_1	X_2	
$C_2 = \neg X_1 \vee \neg X_2 \vee \neg X_3$	$\neg X_1$	$\neg X_2$	$\neg X_3$
$C_3 = \neg X_1 \vee X_2 \vee X_3$	$\neg X_1$	X_2	X_3

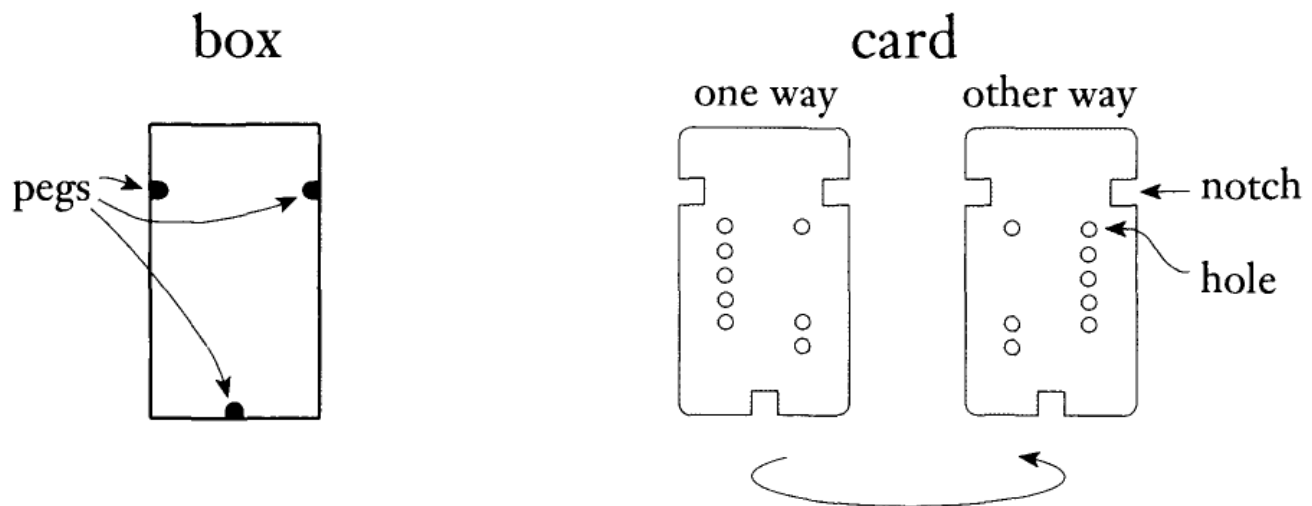
Scacchiera G

La sostituzione $X_1 = \text{vero}$, $\neg X_2 = \text{vero}$, $X_3 = \text{vero}$ rende vera ϕ e corrisponde alla strategia vincente su G , che toglie alcune pietre e lascia colonne dello stesso colore e righe con almeno una pietra:

ϕ	X_1	X_2	X_3
$C_1 = X_1 \vee X_2 \vee X_2$	X_1		
$C_2 = \neg X_1 \vee \neg X_2 \vee \neg X_3$		$\neg X_2$	
$C_3 = \neg X_1 \vee X_2 \vee X_3$			X_3

Problema 7.26: PUZZLE

Problema 7.26 Sipser. "You are given a box and a collection of cards as indicated in the following figure. Because of the pegs in the box and the notches in the cards, each card will fit in the box in either of two ways. Each card contains two columns of holes, some of which may not be punched out. The puzzle is solved by placing all the cards in the box so as to completely cover the bottom of the box, (i.e., every hole position is blocked by at least one card that has no hole there.) Let $PUZZLE = \{(C_1, \dots, C_k) \mid \text{each } C_i \text{ represents a card and this collection of cards has a solution}\}$. Show that $PUZZLE$ is NP-complete."



Chiamiamo "pozzo" la colonna verticale che si trova all'incrocio tra la riga j e la colonna sinistra o destra della prima carta. Un pozzo può raggiungere il fondo scatola oppure essere bloccato da una carta non forata in quel punto: in questo caso diciamo che il pozzo è "turato". Risolvere un PUZZLE significa trovare una successione di valori 0, 1 per C_1, \dots, C_k : quando $C_i=1$ poniamo C_i diritta e quando $C_i=0$ poniamo C_i rovesciata. Possiamo scegliere non-deterministicamente i valori di C_1, \dots, C_k e accettare se la scelta fatta "tura" tutti i "pozzi", un controllo che possiamo fare in tempo lineare. Dunque PUZZLE è NP, con la scelta non deterministica fatta in tempo lineare.

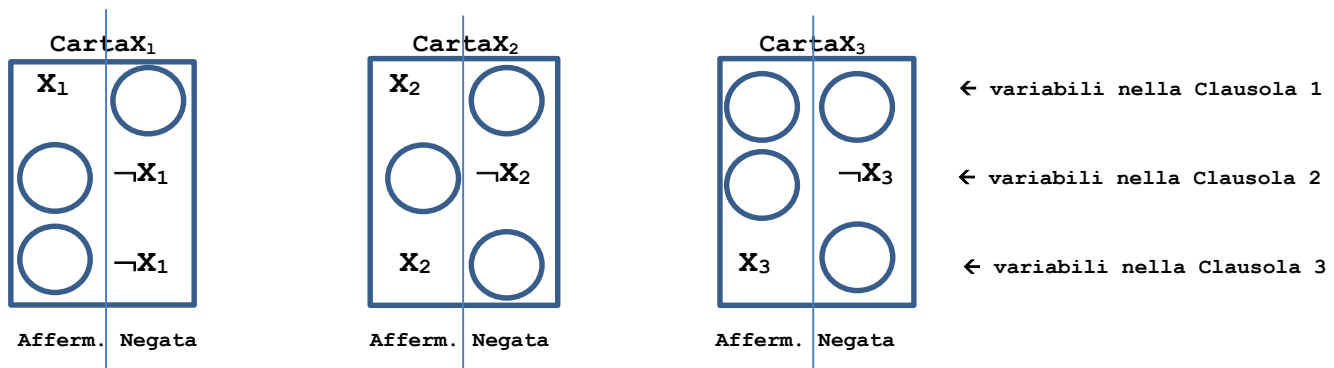
Proviamo che $3\text{-SAT} \leq_P \text{PUZZLE}$, cioè che 3-SAT, e dunque ogni problema NP, si riduce polinomialmente PUZZLE, o in altre parole, che PUZZLE è

NP-completo. A questo fine introduciamo un problema intermedio, LEFT-PUZZLE, la versione di PUZZLE in cui si chiede di turare i soli pozzi della colonna sinistra. Quindi proviamo nell'ordine che **3-SAT \leq_P LEFT-PUZZLE** e che **LEFT-PUZZLE \leq_P PUZZLE**: ne seguirà che ogni problema NP si riduce polinomialmente a 3-SAT, di qui a LEFT-PUZZLE, e infine a PUZZLE, come richiesto.

Sia ϕ una forma normale congiuntiva con tutte clausole di lunghezza 3 con k variabili e h clausole: per esempio

$$\phi = (X_1 \vee X_2 \vee X_2) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3) \wedge (\neg X_1 \vee X_2 \vee X_3)$$

dove $k=3$ e $h=3$. Trasformiamo ϕ in un puzzle $P=f(\phi)$ in modo tale che ϕ è soddisfacibile se e solo se P ha una soluzione in LEFT-PUZZLE. Per ogni $i=1, \dots, k$ definiamo una carta $\text{Carta}X_i$ di h righe e 2 colonne, che rappresenta la variabile X_i . Ogni riga rappresenta una clausola di ϕ e ogni colonna sinistra ci dice se la variabile in una data clausola è affermata, ogni colonna destra se è negata. Più esattamente, per ogni riga $j=1, \dots, h$: nella colonna sinistra di $\text{Carta}X_i$ poniamo uno spazio pieno con la scritta " X_i " se la clausola numero j contiene X_i , un buco se non la contiene; nella colonna destra di $\text{Carta}X_i$ poniamo uno spazio pieno con la scritta " $\neg X_i$ " se la clausola numero j contiene $\neg X_i$, un buco se non la contiene. Se ϕ è come sopra otteniamo le seguenti 3 carte:



Abbiamo già visto che ogni assegnazione σ che rende vera ϕ sceglie un letterale per clausola che viene reso vero, e non sceglie due letterali uno la negazione dell'altro in due clausole diverse.

1. Se σ sceglie di rendere vero qualche letterale X_i in almeno una clausola j , disponiamo $\text{Carta}X_i$ diritta: corrisponde a porre $X_i = \text{vero}$. Nella riga j di $\text{Carta}X_i$, lato sinistro, non c'è un foro, dato che la clausola j contiene X_i .

2. Se σ sceglie di rendere vero qualche letterale $\neg X_i$ in almeno una clausola, disponiamo la carta numero i rovesciata. Dunque la parte senza foro viene portata dal lato destro, riga j , dove si trova $\neg X_i$, nel lato sinistro, sempre riga j . Corrisponde a porre $X_i = \text{falso}$.
3. Se σ non sceglie mai X_i , $\neg X_i$ in nessuna clausola, scegliamo una posizione a caso per la carta $\text{Carta}X_i$. Questa carta è irrilevante.

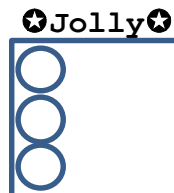
Data σ che rende vera ϕ , una scelta per $\text{Carta}X_i$ è sempre possibile, dato che non capita mai che σ scelga X_i in una clausola e $\neg X_i$ in un'altra, e dunque non facciamo mai due richieste contraddittorie tra loro. Dato che in ogni clausola j scegliamo un letterale X_i oppure $\neg X_i$, a ogni riga j corrisponde almeno una carta priva foro nella riga j . La carta viene voltata in modo da mettere la mancanza di foro a sinistra. Dunque il pozzo di ogni riga j , colonna sinistra è sbarrato da almeno una carta $\text{Carta}X_i$. Quindi abbiamo risolto il puzzle P in LEFT-PUZZLE.

Viceversa, se abbiamo risolto il puzzle P in LEFT-PUZZLE, per ogni clausola j abbiamo una carta i che sbarrava il pozzo verticale sinistro della riga j . Se la carta è dritta abbiamo X_i nella clausola j , se è rovesciata abbiamo $\neg X_i$ nella clausola j . A seconda dei casi scegliamo il letterale X_i oppure $\neg X_i$ nella clausola j . Non scegliamo mai letterali uno negazione dell'altro in due clausole diverse, perché se la carta numero i è dritta scegliamo X_i affermata in ogni clausola con la riga j sbarrata da $\text{Carta}X_i$, mentre se la carta numero i è rovesciata scegliamo X_i negata in ogni clausola con la riga j sbarrata da $\text{Carta}X_i$.

Abbiamo dunque provato che $3\text{-SAT} \leq_P \text{LEFT-PUZZLE}$.

Proviamo ora che $\text{LEFT-PUZZLE} \leq_P \text{PUZZLE}$. Definiamo una funzione $Q=f(P)$ che prende un puzzle e ne costruisce un altro, in modo che P ha una soluzione che tura tutti i pozzi sinistri se e solo se Q ha una soluzione che tura tutti i pozzi.

Sia "Jolly" la carta con tutti fori nella colonna sinistra e senza fori nella colonna destra. Il Jolly consente di turare una colonna a scelta, la sinistra se il Jolly è dritto e la destra se è rovesciato:



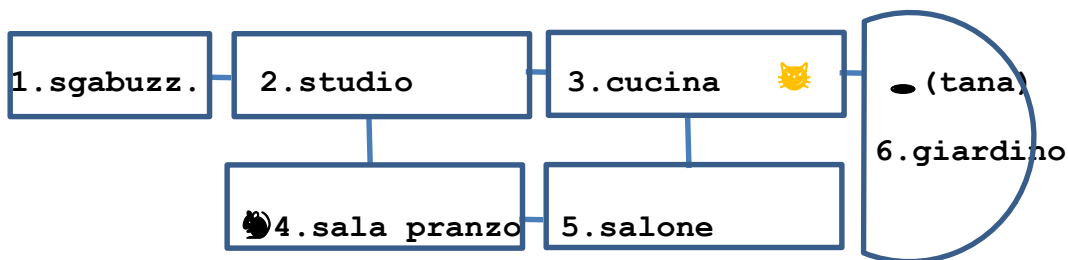
Sia $P = C_1, \dots, C_k$. Definiamo $Q = f(P) = C_1, \dots, C_k, \text{Jolly}$. f richiede tempo polinomiale (anzi, costante). Se P ha una soluzione che tura ogni pozzo sinistro, aggiungendo il Jolly turiamo tutti i pozzi e otteniamo una soluzione di PUZZLE. Viceversa, supponiamo di avere una soluzione che tura tutti i pozzi con le carte $C_1, \dots, C_k, \text{Jolly}$. Allora se il Jolly è disposto diritto, tura tutti i pozzi destri, e la disposizione di C_1, \dots, C_k tura tutti i pozzi sinistri e quindi risolve LEFT-PUZZLE. Se invece il Jolly è disposto rovesciato, tura tutti i pozzi sinistri, quindi la disposizione di C_1, \dots, C_k tura tutti i pozzi destri. Se adesso rovesciamo ogni carta in C_1, \dots, C_k otteniamo una soluzione che tura tutti i pozzi sinistri e quindi risolve LEFT-PUZZLE.

Abbiamo provato che P ha una soluzione in LEFT-PUZZLE se e solo se $Q = f(P)$ ha una soluzione in PUZZLE. Concludiamo che $\text{LEFT-PUZZLE} \leq_p \text{PUZZLE}$.

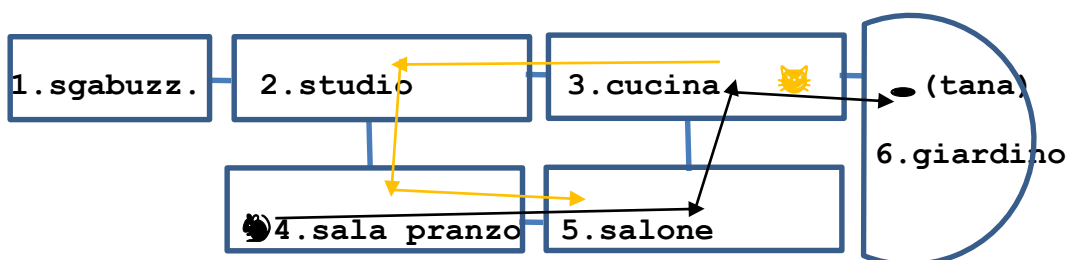
Problema 8.15: Un algoritmo polinomiale per trovare una
strategia vincente nel gioco HAPPY-CAT

"The cat-and-mouse game is played by two players, "Cat" and "Mouse," on an arbitrary undirected graph. At a given point each player occupies a node of the graph. The players take turns moving to a node adjacent to the one that they currently occupy. A special node of the graph is called "Hole." Cat wins if the two players ever occupy the same node. Mouse wins if it reaches the Hole before the preceding happens. The game is a draw if a situation repeats (i.e., the two players simultaneously occupy positions that they simultaneously occupied previously and it is the same player's turn to move)."

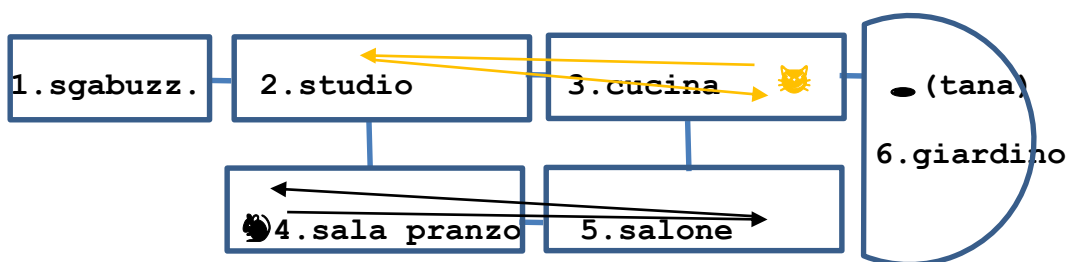
Il gioco viene chiamato HAPPY-CAT sul Sipser. Vengono forniti il grafo G che fa da base del gioco, i nodi c, m inizialmente occupati dal gatto e dal topo, e il nodo h in cui si trova la tana del topo. Si chiede di decidere se esiste una strategia vincente per il gatto. Come esempio, consideriamo il grafo G, con i simboli 🐱, 🐭, 🕳️ per le posizioni iniziali di gatto e topo, e per la posizione della tana.



Un esempio di strategia perdente. Se il gatto muove nello studio, il topo fugge in salone, e se il gatto lo insegue in sala da pranzo, il gatto fugge in cucina e a questo punto il topo ha vinto. Il gatto è distante, intanto che raggiunge il salone il topo raggiunge la tana.



Una strategia per la parità. Non ci sono in questo caso strategie vincenti per il gatto, né per il topo. Una strategia che garantisce il pareggio al gatto è muovere nello studio, e se il topo muove al salone il gatto torna in cucina e gli sbarra la strada. La mossa migliore del topo è tornare in sala da pranzo, la posizione è uguale a quella iniziale e la partita è pari.



Altre considerazioni. Se il gatto muove in salone, il topo deve muovere nello studio, e se il gatto gli taglia la strada verso la tana tornando in cucina, allora il topo deve evitare di ritirarsi nello sgabuzzino. Nello sgabuzzino infatti il topo è in trappola, il gatto deve solo andare a prenderlo muovendo verso lo studio. In questo caso, la mossa giusta per il topo è tornare in sala da pranzo, la posizione è uguale a quella iniziale e la partita è pari.

Von Neumann. Chiediamoci ora come scrivere un algoritmo che trovi in tutte le situazioni possibili in HAPPY-CAT una strategia vincente per il gatto, oppure vincente per il topo, oppure due strategie, una per giocatore, che garantiscono a chi la segue almeno la parità.

L'algoritmo più usato per trovare una strategia vincente un gioco sequenziale tra due giocatori e con partite tutte finite è detto **algoritmo di Von Neumann**, che esplora l'albero di tutte le partite possibili. Nel caso di HAPPY-CAT, però, le partite possibili sono in numero esponenziale. Infatti il grafo G ha $n=6$ nodi, dunque le posizioni possibili del gatto e del topo sono $6*6=36$. Se teniamo conto che in ogni posizione può essere di turno il gatto oppure il topo, le configurazioni del gioco sono $6*6*2=72$. Dunque dopo al più 72 mosse almeno una posizione si ripete, e se nessun giocatore ha ancora vinto la partita finisce pari per ripetizione della posizione. Quante mosse sono possibili a ogni passo? La risposta è al più k , dove k è il massimo numero di archi che esce da un nodo x di G e che rappresentano le mosse possibili di un giocatore posto in x . Nel nostro esempio $k=3$, perché i nodi studio e cucina sono connessi con 3 nodi, e tutte gli altri

nodi con 1 nodo oppure 2. Dunque le partite possibili sono al più tutte le successioni con ripetizione di 3 interi di lunghezza 72, quindi $\leq k^{72} = 3^{72} \approx 2,25 \cdot 10^{34}$. È un numero senza senso: ventidue milioni di miliardi di miliardi di miliardi. Si tratta di una **sovrastima**, ma comunque l'algoritmo di Von Neumann nella sua versione originaria è esponenziale per il problema HAPPY-CAT.

Un algoritmo polinomiale. Per HAPPY-CAT, tuttavia, esiste un algoritmo polinomiale per trovare una strategia vincente. Il motivo è che una strategia vincente dipende solo dalla configurazione di gioco, e le configurazioni di gioco sono appena $2n^2$, come vedremo, ed è sufficiente esplorare queste ultime. Vediamo come.

Una strategia è una matrice M di dimensioni $n \times n \times 2$. La prima coordinata è il numero i del nodo di G che contiene il gatto, la seconda è il numero j del nodo di G che contiene il topo, e la terza un messaggio $\text{turno} = \text{gatto, topo}$, che indica se muove il gatto oppure il topo. Il valore $M[i, j, \text{turno}] = \langle v, s \rangle$ è una coppia. v è il vincitore previsto: $v = \text{gatto, topo, ?}$ ($? = \text{non lo so}$). s è la mossa migliore suggerita da M al giocatore di turno, rappresentata dal numero s del nodo in cui muovere, con $0 = \text{nessun suggerimento}$.

All'inizio l'algoritmo pone $M[i, j, \text{turno}] = \langle ?, 0 \rangle$ sempre, vincitore e mossa migliore sono ignoti. In seguito un ciclo migliora la matrice M . In base alle regole del gioco, le posizioni (i, i, turno) sono tutte vinte dal gatto, le posizioni (i, h, turno) , con il topo nel nodo h con la tana e il gatto in una posizione $i \neq h$, sono tutte vinte dal topo. Ridefiniamo $M[i, i, \text{turno}] = \langle \text{gatto}, 0 \rangle$ e $M[i, h, \text{turno}] = \langle \text{topo}, 0 \rangle$ quando $i \neq h$. Un ciclo passa in rassegna tutte le posizioni (i, j, turno) con $M[i, j, \text{turno}] = \langle ?, 0 \rangle$, e cerca di migliorare M . Ci sono 3 casi.

1. Supponiamo che $\text{turno} = \text{gatto}$ (il gatto è di turno su (i, j)) e che dalla posizione (i, j, gatto) il gatto che è di turno può raggiungere da i in una mossa una posizione (i', j, topo) , con i' adiacente ad i e $M[i', j, \text{topo}] = \langle \text{gatto}, j' \rangle$, in cui il gatto è dichiarato vincente. Allora poniamo $M[i, j, \text{gatto}] = \langle \text{gatto}, i' \rangle$. Il significato del nuovo valore di M è: il gatto vince se muove dal nodo i al nodo i' .
2. Supponiamo che tutte le posizioni (i', j, topo) raggiungibili in una mossa dalla posizione (i', j, gatto) con i' adiacente ad i e il gatto di turno hanno $M[i', j, \text{topo}] = \langle \text{topo}, j' \rangle$ (vince il topo). Allora ogni mossa del gatto fa vincere il topo. Classifichiamo la posizione (i, j, gatto) come vincente per il topo, ponendo

$M[i,j,\text{gatto}] = \langle \text{topo}, 0 \rangle$ (vince il topo, nessuna mossa suggerita per il gatto).

3. Infine, se ci sono posizioni raggiungibili in una mossa con $M[i',j,\text{topo}] = \langle ?, 0 \rangle$ (non si sa), e tutte le altre hanno $M[i',j,\text{topo}] = \langle \text{topo}, h \rangle$ (sono vincenti per il topo), per ora non modifichiamo il valore di $M[i,j,\text{gatto}]$.

Modifichiamo M in modo analogo per le posizioni in cui è di turno il topo. Dobbiamo esaminare tutte le posizioni della matrice, che sono $2n^2$. Se ridefiniamo M in almeno un caso una ripetiamo il ciclo, altrimenti terminiamo. Occorrono al massimo $2n^2 * 2n^2$ passi, una quantità polinomiale.

Nota. L'algoritmo si può migliorare in modo che impieghi al massimo un numero di passi pari a $2n^2 * (\text{massimo numero di archi da un nodo di } G)$. Basta esaminare le sole posizioni del gioco adiacenti a una posizione appena marcata. Nel nostro esempio gli archi da un nodo sono al massimo 3, e il numero di passi richiesti è al massimo $3 * 72 = 216$, mentre la stima precedente era $3^{72} = 2,25 * 10^{34}$.

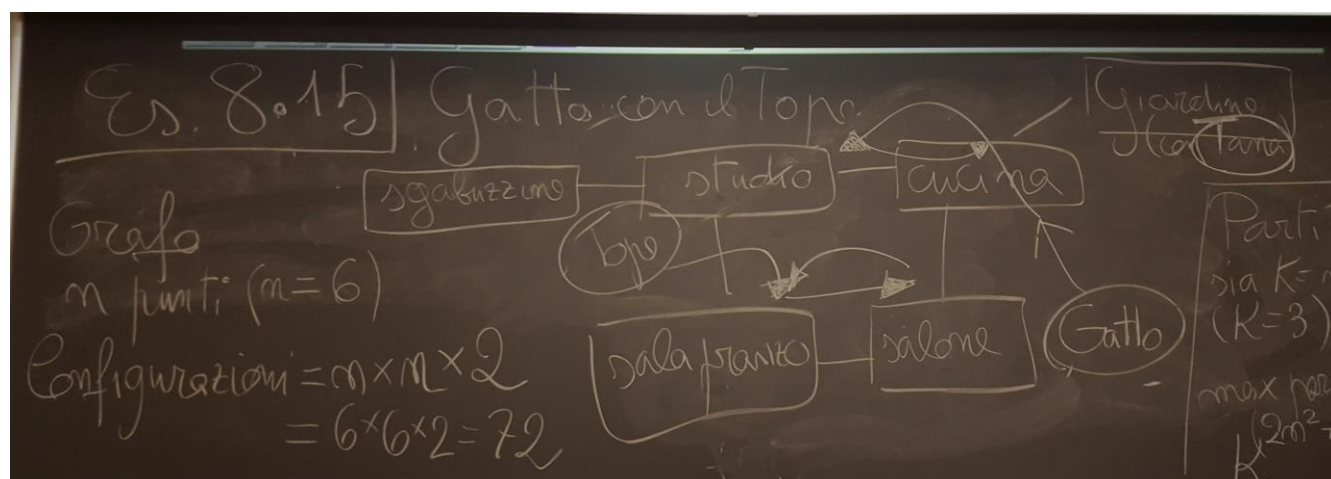
Correttezza. Resta da controllare che alla fine M definisce una strategia vincente. Quando abbiamo posto $M[i,j,\text{gatto}] = \langle \text{gatto}, i' \rangle$ oppure $M[i,j,\text{topo}] = \langle \text{topo}, j' \rangle$ abbiamo fatto attenzione di scegliere una mossa vincente per il gatto oppure per il topo.

Completezza. Resta da stabilire se abbiamo trovato una strategia vincente per un giocatore tutte le volte che ne esiste una. Per farlo, facciamo vedere che nei casi $M[i,j,\text{gatto}] = \langle ?, 0 \rangle$ rimasti, quelli che cadono sempre nel caso 3 dell'algoritmo, entrambi i giocatori possono forzare il pareggio.

In questi casi, infatti, ogni posizione (i',j,topo) raggiungibile da (i,j,gatto) abbiamo $M[i',j,\text{topo}] = \langle \text{topo}, j' \rangle$ (vince il topo) oppure $M[i',j,\text{topo}] = \langle ?, 0 \rangle$ (non si sa). Inoltre, c'è almeno un caso in cui abbiamo $M[i'_0,j_0,\text{topo}] = \langle ?, 0 \rangle$. Dunque la mossa migliore per il gatto è muovere a una di queste posizioni "non si sa", quindi poniamo $M[i',j,\text{gatto}] = \langle ?, i'_0 \rangle$. Facciamo lo stesso per il topo.

Finite le nuove modifiche, se $M[i',j,\text{gatto}] = \langle ?, i'_0 \rangle$ il gatto muove a una posizione (i'_0,j,topo) con $M[i'_0,j,\text{topo}] = \langle ?, j'_0 \rangle$, e lo stesso fa il topo. La partita continua attraverso posizioni tutte marcate "?" senza vincitori, perché tutte le posizioni vinte da uno dei due giocatori non sono più marcate "?" alla fine dell'algoritmo. Dopo al massimo $2n^2$ mosse una posizione si ripete e la partita è pari.

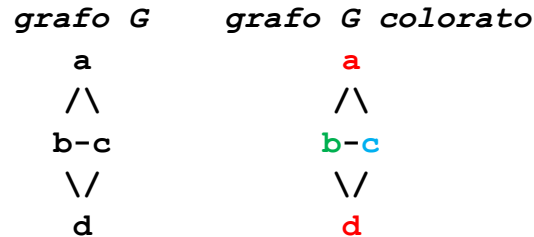
Nel caso di una posizione ancora marcata "?" alla fine dell'algoritmo, quindi, M fornisce a entrambi i giocatori una strategia in grado almeno di pareggiare. Queste posizioni non sono vincenti, dunque le posizioni vincenti per un giocatore sono soltanto quelle già trovate.



Lavagna al termine della lezione del 20 Gennaio 2020

3-SAT si reduce polinomialmente a 3-coloring

k-coloring è il problema di colorare i vertici di un grafo non orientato con k colori, in modo tale che due vertici adiacenti hanno colori diversi. Qui sotto un esempio di 3-coloring di un grafo G, con rosso, verde, blu, con ogni vertice colorato in modo diverso dai vertici adiacenti.



A prima vista la k-colorazione sembra un gioco a incastro, come il sudoku. Invece si tratta di un problema con applicazioni più ampie. Per esempio i punti del grafo possono essere un insieme di risorse, i colori possono essere i compiti da assegnare a queste risorse, e gli archi indicare delle incompatibilità, sotto forma di coppie di risorse che non possono essere dedicate allo stesso compito. Allora la k-colorazione diventa il problema di distribuire i compiti tra le risorse rispettando le incompatibilità.

Qualche esempio. In informatica, il problema dell'allocazione di registri può venire espresso come k-colorazione. Più avanti vedremo come il problema della generazione automatica di un orario per esami possa venire espressa come k-colorazione.

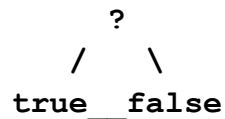
NP-completezza. Il problema di k-Coloring è NP, con certificato la colorazione stessa, che è polinomialmente verificabile. Proviamo che 3SAT si riduce polinomialmente a 3-coloring, e che dunque 3-coloring (e a maggior ragione k-coloring per $k \geq 3$) è NP-completo.

Sia data una formula booleana ϕ che sia 3-congiuntiva, ovvero una congiunzione di un numero qualsiasi di clausole di 3 letterali, per esempio

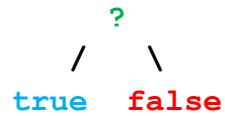
$$\phi = (X_1 \vee X_2 \vee X_3) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3) \wedge (\neg X_1 \vee X_2 \vee X_3)$$

Vogliamo definire una trasformazione polinomiale f , che prenda una formula ϕ in forma 3-congiuntiva e restituisca un grafo $G=f(\phi)$, in modo che ϕ sia soddisfacibile se e solo se G sia 3-colorabile.

Innanzitutto inseriamo in G un triangolo con vertici $true$, $false$, $?$ ($=$ indefinito).



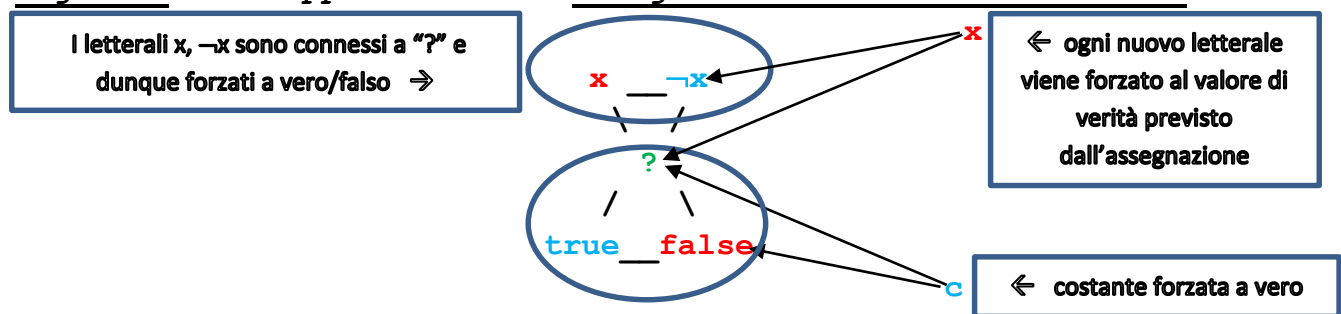
I 3 colori **rosso**, **verde**, **blu** del triangolo devono essere distinti:



Consideriamo il colore del vertice " $true$ " (**blu** in questo caso) come " $vero$ ", il colore del vertice " $false$ " (**rosso** in questo caso) come " $falso$ ", il colore del vertice " $?$ " (**verde** in questo caso) come " $indefinito$ ".

Per esprimere in G il fatto che x è una variabile aggiungiamo i nodi x , $\neg x$ collegati tra loro e al nodo " $?$ ". In questo modo x , $\neg x$ non devono essere colorati " $indefinito$ ", ma devono avere " $vero$ " o " $falso$ ", e dato che x , $\neg x$ sono connessi tra loro, se x è colorato " $vero$ " allora $\neg x$ è falso e viceversa. Nell'esempio sotto abbiamo scelto " $falso$ " per x , ma potevamo anche scegliere " $vero$ ". Se invece vogliamo forzare un valore di verità per un nodo c , per esempio " $vero$ ", connettiamo c con " $?$ ", " $false$ ", come qui sotto.

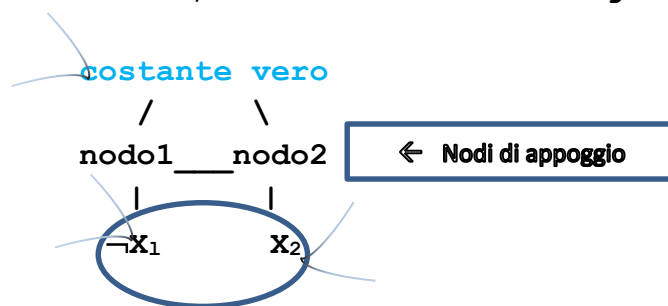
Segmento che rappresenta una assegnazione di valori di verità ad x



Triangolo che rappresenta i valori di verità

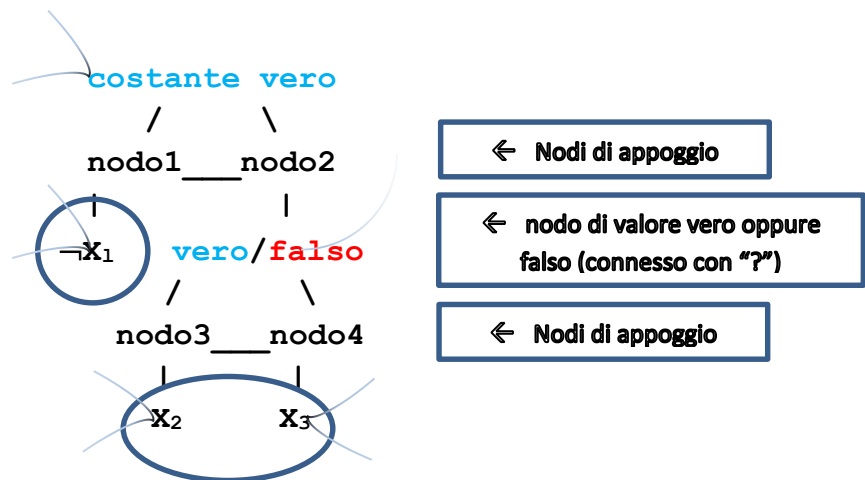
Ci resta da tradurre i letterali delle clausole della formula ϕ , per esempio della clausola, $C = \neg X_1 \vee X_2 \vee X_3$. Introduciamo 3 nodi etichettati $\neg X_1, X_2, X_3$. Per far sì che $\neg X_1, X_2, X_3$ abbiano il valore di verità ad essi assegnato, connettiamo ogni nodo X_i con (i) il nodo "?", forzando X_i al colore vero/falso; (ii) con il nodo $\neg X_i$ posto nel segmento $X_i \neg X_i$ che rappresenta la variabile X_i . In questo modo il valore di verità di X_i sarà diverso da quello di $\neg X_i$, e dunque sarà uguale al valore di verità corretto per X_i . Facciamo lo stesso per ogni nodo $\neg X_i$.

Ci resta da far sì che la colorazione di G sia corretta se e solo se almeno uno dei valori di verità assegnati ai letterali di una stessa clausola C , per esempio $C = \neg X_1 \vee X_2 \vee X_3$, vale vero. A questo scopo aggiungiamo nuovi archi, e combiniamo i letterali $\neg X_1, X_2$, già forzati al valore di verità richiesto dall'assegnazione, in un grafo che rappresenta l'operazione OR su $\neg X_1, X_2$. Facciamo come segue:



Forziamo il vertice ad essere colorato vero, connettendolo con "false", "?". Allora nodo1, nodo2 sono colorati falso, indefinito oppure indefinito, falso. Nel primo caso $\neg X_1$ deve essere colorato vero e X_2 deve essere colorato vero oppure falso. Nel secondo caso $\neg X_1$ deve essere colorato vero oppure falso e X_2 deve essere colorato vero. In altre parole l'assegnazione scelta per le variabili nei segmenti relativi alle variabili è corretta se e solo se rende vero almeno uno tra i letterali $\neg X_1, X_2$.

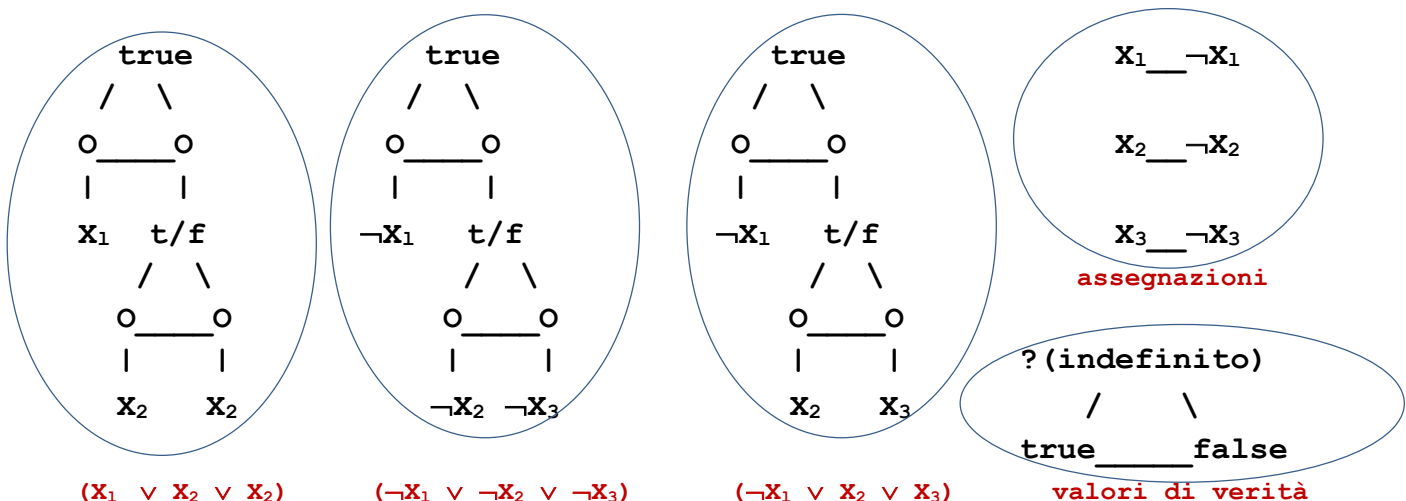
Ci resta da definire una disgiunzione ternaria di $\neg X_1, X_2, X_3$. Scambiando il ruolo di vero e di falso, proviamo che se il vertice del triangolo è falso allora almeno uno tra i letterali alla base è falso. Quindi se il vertice dell'OR è vero oppure falso, allora tutte e sole le assegnazioni di verità per i letterali alla base sono colorazioni accettabili. Questa considerazione ci consente di definire un OR ternario, aggiungendo un nodo di colore vero/falso (connesso con "?").



Per il ragionamento già fatto ci sono due possibilità. Se $\neg X$ vale vero, il nodo a lato può essere vero o falso e dunque X_2 , X_3 possono avere un valore qualsiasi. Se invece $\neg X$ vale falso, il nodo a lato può essere solo vero, e dunque uno almeno tra i letterali X_2 , X_3 deve avere il valore vero. Dunque il valori di verità dei $\neg X_1$, X_2 , X_3 corrispondenti a colorazioni accettabili sono tutti e soli quelli che includono almeno un valore vero. Quindi abbiamo tradotto fedelmente la condizione di verità di una clausola con 3 letterali.

Pertanto data una formula 3-congiuntiva ϕ possiamo definire un grafo G in modo tale che G abbia una 3-colorazione con colori distinti per punti adiacenti se e solo se ϕ ha una assegnazione che la rende vera.

Qui sotto una versione incompleta del grafo G ottenuto nell'esempio a partire da ϕ . Mancano i seguenti archi: (i) ogni letterale deve essere ancora connesso con "?" (indefinito) e con il letterale opposto, (ii) ogni costante vera deve essere connessa con indefinito e con false, e (iii) ogni nodo t/f (vero o falso) con il solo indefinito.



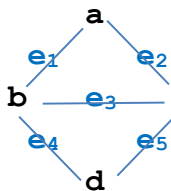
**3coloring si reduce polinomialmente
al basic scheduling problem**
(detto anche *problema dell'orario*)

Consideriamo il basic scheduling decision problem, "dato un insieme di studenti S , un insieme di esami E , e k orari possibili, c'è una assegnazione degli esami agli orari tale che nessuno studente ha un conflitto di orario?"

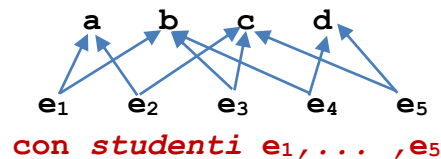
Il basic scheduling decision problem è verificabile polinomialmente, con certificato l'assegnazione di orari ai corsi. Pertanto è un problema NP. Il problema è NP-completo for $k \geq 3$, la prova è per riduzione dal problema di k -coloring che abbiamo già provato essere NP-completo.

Dato un problema di k -colorazione di un grafo G , trasformiamo ogni colore in un orario, ogni vertice in un esame a , infine trasformiamo ogni arco tra due esami a, b in uno studente s che si è iscritto a entrambi gli esami a, b . La presenza di s crea un conflitto di orario. Dunque vieta che gli esami a, b si tengano nello stesso orario, o in modo equivalente, che come nodi abbiamo lo stesso colore.

Grafo G , con archi e_1, \dots, e_5



Problema orario O



- Questa riduzione è fedele: un grafo è k -colorabile se e solo se il corrispondente problema dell'orario si risolve usando k orari diversi.
- Questa riduzione è polinomiale.

Dunque il problema dell'orario è k -completo dato che il k -coloring è NP-completo.

Un esempio di approssimazione per il problema di Vertex-Covering

Forniamo un esempio di uso dell'algoritmo di vertex-covering approssimato, definito dal Sipser a pagina 365, Teorema 10.1.

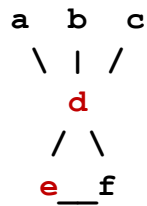
Sia dato un grafo non diretto G . Un vertex covering C per G è un insieme di vertici di G tale che ogni arco di G inizia oppure finisce in qualche vertice di C (vedi sotto per un esempio). Il problema del vertex covering minimo è trovare un covering C con un minimo numero di vertici. Il problema del k -vertex covering è trovare un covering C con k vertici.

Il k -vertex covering si riduce polinomialmente al problema del vertex covering minimo: se siamo in grado di trovare un covering minimo C di G , per sapere se esiste un covering di G con k vertici basta controllare se i vertici di C sono $\leq k$. Abbiamo provato che il k -vertex covering è NP-completo. Dunque anche il vertex covering minimo è NP-completo. Quindi ogni algoritmo che trova il vertex covering minimo richiede tempo esponenziale nel caso pessimo, a meno che sia $P=NP$.

Il Sipser presenta un algoritmo di vertex covering approssimato, che trova in tempo polinomiale un covering di numero di elementi al più doppio del covering minimo. L'algoritmo sceglie un arco, marca come già scelti l'arco dato e tutti gli archi con un vertice in comune con esso. Quindi sceglie un secondo arco e ripete il procedimento fino a che tutti gli archi sono marcati. Sia H l'insieme degli archi scelti e C l'insieme dei vertici iniziali e finali degli archi di H : gli archi di H sono scelti privi di vertici in comune, dunque C ha un numero di elementi esattamente doppio degli elementi di H , quindi ha un numero pari di elementi.

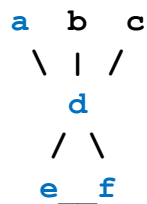
Si prova C è una copertura, con numero di elementi al più il doppio del numero minimo. Rimandiamo al Sipser per la (semplice) dimostrazione.

Vediamo invece in un esempio cosa cambia passando da vertex-covering alla sua versione approssimata. Consideriamo il seguente grafo G , con 6 punti e 5 archi:



Una vertex-covering è data dai vertici $\{d, e\}$: tutti e 5 gli archi di G hanno come uno dei due estremi d oppure e . Invece non esiste una covering $C=\{x\}$ con un vertice solo, perché almeno due dei tre vertici d, e, f manca in C , quindi almeno uno degli archi (d, e) , (e, f) (f, d) non viene coperto. Dunque la covering minima ha due elementi, e il vertex-covering approssimato fornisce un covering C con un numero di vertici compreso tra 2 (il minimo richiesto) e 4 (il doppio del minimo). Inoltre C deve avere un numero pari di elementi, dunque le possibilità rimaste sono C di due 2 elementi oppure C di 4 elementi. Vediamo come entrambi i casi siano possibili.

1. Supponiamo di partire con l'arco (d, e) . Quest'arco tocca ogni altro arco in G , dunque ci fermiamo subito e otteniamo come archi $H = \{(d, e)\}$, e come copertura gli estremi di (d, e) . Dunque otteniamo $C=\{d, e\}$, una covering minima.
2. Supponiamo di partire con l'arco (a, d) . Questo arco tocca ogni altro arco di G tranne (e, f) . Aggiungiamo (e, f) e ci fermiamo. Otteniamo come archi $H=\{(a, d), (e, f)\}$, e come copertura gli estremi di $(a, d), (e, f)$, dunque $C=\{a, d, e, f\}$, una covering di dimensione doppia della covering minima. Eccola:



Vediamo quindi come il passaggio da un algoritmo esponenziale a una sua approssimazione polinomiale fornisce un risultato di minor valore, ma comunque utile e molto meno dispendioso da ottenere.

Complessità in spazio (solo cenni nel 2022)

PATH, il problema della raggiungibilità in un grafo, è risolubile in spazio L (spazio logaritmico)?

Probabilmente no. Vediamo perché e quando vicino possiamo andare oggi a una soluzione logaritmica in spazio di PATH.

Sia G un grafo diretto di p punti, codificato con una matrice $p \times p$ di booleani, dove $G[i][j] = \text{vero}$ se e solo se c'è un arco da i a j in G . Sia $n = p^2$ il numero di bit necessari a rappresentare G come matrice. Supponiamo che R sia una macchina che decide PATH, il problema della raggiungibilità in G : $R(a,b) = \text{vero}$ se $a,b \in G$ sono connessi da un cammino in G , altrimenti $R(a,b) = \text{falso}$. Confrontiamo due algoritmi per realizzare $R(a,b)$. Il primo algoritmo per PATH è il solito, con tempo e spazio limitato dal numero degli archi di G , e tempo e spazio almeno $O(n)$. Il secondo algoritmo per PATH richiede spazio $O(\log(n)^2)$ ma tempo non polinomiale. Sappiamo che un algoritmo in spazio $O(\log(n)^2)$ esiste: infatti abbiamo $\text{PATH} \in \text{NL}$, ovvero c'è un algoritmo non-deterministico per PATH in spazio logaritmico (esempio 8.19 del Sipser). Dunque per il per il Teorema di Savitch $\text{PATH} \in O(\log(n)^2)$ (Teor. 8.5).

Al momento, spazio $O(\log(n)^2)$ è il meglio che sappiamo fare per PATH, e, come vedremo, solo al prezzo di un aumento proibitivo del tempo di calcolo. Non sappiamo se sia possibile risolvere il problema della raggiungibilità con un algoritmo in L , cioè in spazio $O(\log(n))$ (logaritmico) e dato che $L \subseteq P$, sempre in tempo polinomiale. Come spiegato nel Teorema 8.25 del Sipser, PATH è NL-completo. Dunque $\text{PATH} \in L$ se e solo se $L = \text{NL}$, ma non sappiamo se $L = \text{NL}$ è vero. Anzi, tendiamo a credere che non lo sia.

1. L'algoritmo canonico per $R(a,b)$. Prende una lista L di punti, con all'inizio $L = \text{il solo punto } a$. Ha l'obiettivo di etichettare tutti e soli i punti raggiungibili dai punti di L . Prende il primo punto x sulla lista, se x è etichettato lo cancella, altrimenti etichetta x e poi rimpiazza x in L con tutti i punti connessi a x da un arco di G .

Quando L è vuota allora $R(a,b)=\text{vero}$ se e solo se b è etichettato. Quando consideriamo gli archi che escono da x etichettiamo x , e quando x è etichettato lo cancelliamo, quindi non possiamo considerare gli archi da x due volte. Dunque il massimo numero di passi è il numero degli archi di G . Al massimo, quando da ognuno dei p punti di G partono p archi, il numero degli archi vale $n=p^2$. Se ogni accesso alla matrice richiede tempo 1, allora R richiede al massimo tempo $O(n)$ e quindi al massimo spazio $O(n)$. Il minimo tempo e la minima memoria usata, nel caso ogni punto di G tranne b sia connesso al punto iniziale a , è uguale il numero dei punti. Infatti dobbiamo etichettare ogni punto diverso da b per finire, dunque abbiamo bisogno di almeno un passo per punto, quindi tempo $p=\sqrt{n}$. Abbiamo bisogno di un vettore di $p=\sqrt{n}$ etichette, una per punto. Quindi il minimo tempo e spazio richiesti in questo caso vale almeno $p=\sqrt{n}$. Conclusione: *la complessità in tempo e spazio per l'algoritmo R nel caso peggiore è almeno $O(\sqrt{n})$, e in ogni caso è al massimo $O(n)$.*

Chiediamoci ora come sia possibile risparmiare spazio nel calcolo di $R(a,b)$. In programmazione dinamica noi immagazziniamo il risultato di alcuni calcoli parziali in memoria per evitare di ripeterli, consumando più spazio per risparmiare tempo. Facciamo ora il ragionamento opposto: immaginiamo di non avere più spazio (qualche volta capita, per fortuna non è frequente) e chiediamoci se sia possibile ripetere molte volte gli stessi calcoli per risparmiare spazio ma impiegando più tempo. Nel caso di $R(a,b)$, e in molti altri casi, la risposta è: sì, è possibile, ma il consumo di tempo può diventare proibitivo, rendendo praticamente impossibile raggiungere la soluzione quando G ha grandi dimensioni.

2. Un algoritmo per $R(a,b)$ in spazio ridotto: $O(\log(n)^2)$ Vediamo come sia possibile ridurre e di molto lo spazio richiesto R , con un algoritmo che richiede solo spazio $O(\log(n)^2)$, e decide la connessione tra a e b , ma **a prezzo di un tempo di calcolo non polinomiale**. Usiamo il fatto che le definizioni ricorsive esplorano l'albero delle chiamate ricorsive **tenendo in memoria un ramo per volta**. Definiamo ricorsivamente $R(a,b,h)$, un algoritmo che decide la raggiungibilità in al più h passi. Cominciamo con la raggiungibilità in un passo: questa si ha quando $a=b$ oppure a,b sono connessi in G , quindi:

$$R(a,b,1) = (a=b) \vee (G[a,b]=\text{vero})$$

La raggiungibilità da a fino a b in 2^{k+1} passi si ha quando esiste un punto c raggiungibile in 2^k passi, da cui raggiungiamo b in 2^k passi, quindi:

$$R(a,b,2^{k+1}) = \bigvee_{c \in G} R(a,c,2^k) \wedge R(c,b,2^k)$$

Infine, possiamo raggiungere b da a quando è possibile raggiungere b da a in al più tanti passi quanti i punti p di G , dunque:

$$R(a,b) = R(a,b,2^k) \text{ per il primo } 2^k \geq p = \text{numero punti } G.$$

Il calcolo ricorsivo di $R(a,b)$ richiede al più $k = \log(p) + 1 = O(\log(p))$ chiamate ricorsive in sospeso (ogni chiamata ricorsiva riduce k di 1). Ogni chiamata ricorsiva richiede di scrivere un punto c (un intero da 1 a p) e 2^k (un intero da 1 a p circa). Questi due interi richiedono spazio $O(\log(p))$, lo spazio totale richiesto da una singola chiamata ricorsiva. Il totale di tutte le chiamate ricorsive in sospeso è quindi $O(\log(p)^2)$. Dato che $p = \log(\sqrt{n}) = \log(n)/2$, abbiamo $O(\log(p)) = O(\log(\sqrt{n})) = O(\log(n))$. Concludiamo che il totale di spazio richiesto è $O(\log(n)^2)$.

Il tempo richiesto, in base alla relazione generale tra spazio e tempo (esempio 8.19 del Savitch), vale al più

$$2^{O(\log(n)^2)} = 2^{\log(n) * O(\log(n))} = (2^{\log(n)})^{O(\log(n))} = n^{O(\log(n))}$$

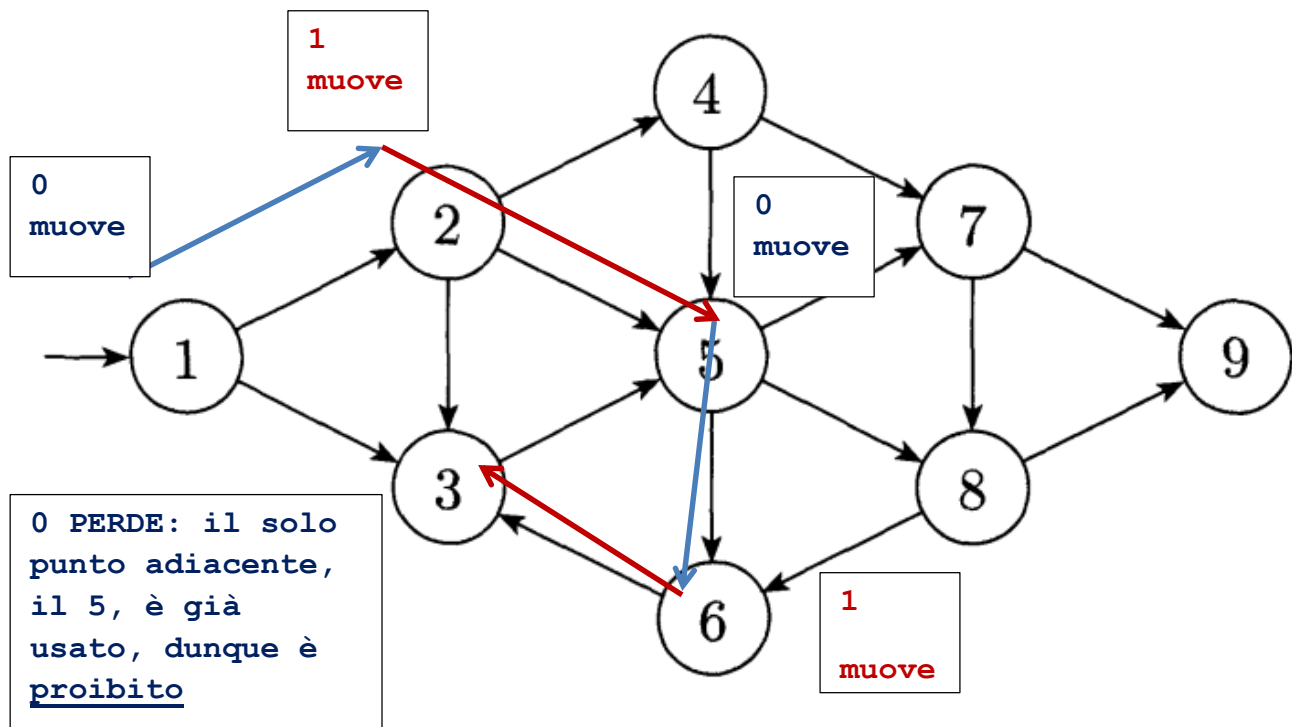
Purtroppo, $n^{O(\log(n))}$ non è un tempo polinomiale: l'esponente di n vale $O(\log(n))$ e può crescere all'infinito. Come abbiamo detto, non sappiamo se si può fare di meglio e scendere a uno spazio $O(\log(n))$ e tempo polinomiale. Questo richiede decidere se $L=NL$, un problema per ora aperto.

GG, il gioco generalizzato della Geografia,
ha un strategia ottimale in PSPACE

Nel "gioco della geografia" si considerano "adiacenti" due città se il nome della prima termina con la lettera iniziale della seconda. Scelta una città di partenza, due giocatori si alternano muovendosi da una città a una "adiacente", ma senza mai tornare per una città già vista. Il primo giocatore che non può indicare una città perde. Vediamo ora una versione più generale di questo gioco che chiamiamo GG.

La descrizione di GG (gioco "generalizzato" della geografia). Viene dato un grafo G con p punti e una matrice (che indichiamo ancora con G) con $p \times p$ booleani. Indichiamo con $n=p \times p$ la dimensione in memoria di G . Con $G[i,j] = \text{vero}$ indichiamo che il punto i e il punto j sono connessi da un arco in G . Ci sono due giocatori, che indichiamo con 0 e 1 . La posizione iniziale del gioco consiste in G , un insieme X di punti proibiti di G , un punto iniziale $a \in X$ (a cui è "proibito" tornare), e un giocatore $g=0,1$ a cui tocca muovere. Chiamiamo $\neg g=1,0$ l'altro giocatore. Il giocatore g sceglie un punto $b \in G$ tale che $G[a,b]=\text{vero}$ (c'è un arco da a verso b) e b non è proibito ($b \notin X$), se ce ne sono, altrimenti g perde. Il punto a viene aggiunto all'insieme X dei punti proibiti. Ora tocca a muovere all'altro giocatore, $\neg g$, e così via. Dato che ogni posizione raggiunta viene in seguito vietata, il massimo numero di mosse è il numero dei punti di G . Dunque ci sono p mosse al massimo. Non esiste una posizione di parità. Nel gioco originario G è una lista di città, e abbiamo un arco tra due città qualunque che siano "adiacenti" rispetto alle regole del gioco.

Ecco un grafo G e una partita per GG. All'inizio $a=1$ e $X=\{1\}$ (il solo punto "proibito" è 1).



Il problema GG. Sia $\langle a, X, g \rangle$ una posizione di GG per un grafo G . Ci chiediamo se esiste una strategia per il giocatore g che consente a g di vincere sempre a partire da $\langle a, X, g \rangle$, oppure no.

Il problema GG è decidibile. Definiamo un algoritmo $\text{VonN}(a, X, g)$ (detto algoritmo di Von Neumann) per decidere se esiste una strategia vincente per il giocatore g su $\langle a, X, g \rangle$, una volta fissato G . Definiamo:

$$\text{VonN}(a, X, g) = \bigvee_{b \in G, G[a, b] = \text{vero}, b \notin X} \neg \text{VonN}(b, X \cup \{a\}, \neg g)$$

$\text{VonN}(a, X, g)$ vale vero se esiste una mossa b che conduca a una posizione $\langle b, X \cup \{a\}, \neg g \rangle$ che non è vincente per il giocatore opposto $\neg g$. Notiamo che se non ci sono mosse lecite per g a partire da a e che evitino X , allora $\text{VonN}(a, X, g)$ è la disgiunzione di un insieme vuoto, dunque vale falso, in coerenza con le regole del gioco.

L'algoritmo di VonNeumann è esponenziale per GG. L'algoritmo esplora tutte un sotto-insieme delle partite possibili a partire da una configurazione iniziale $\langle a, X, g \rangle$. Il numero di mosse di una singola partita è limitato da p , il numero di punti di G , e il numero di mosse a ogni passo è limitato da k , il massimo numero di archi che escono da un nodo di G . Dunque il numero di partite è $\leq k^p$. Si possono trovare

esempi in cui il numero di partite esplorate dall'algoritmo è effettivamente esponenziale.

Il problema GG è PSPACE. Si può migliorare la soluzione di GG a una soluzione polinomiale? Sembra di no, ora vediamo perché. Se implementiamo la ricorsione con un algoritmo a pila, lo spazio di cui abbiamo bisogno è la massima dimensione della pila di chiamate ricorsive. La massima dimensione della pila nel caso del calcolo di $\text{VonN}(a, X, g)$ è p , il numero dei punti di G , dato che ogni chiamata aggiunge un punto all'insieme dei punti vietati. Ogni chiamata ricorsiva viene descritta da un giocatore $\neg g = 0, 1$, e da un punto b , il primo punto adiacente ad a e non ancora considerato nel calcolo di $\text{VonN}(a, X, g)$. La descrizione di un qualunque punto b di G richiede un numero tra 1 e p , dunque $\log(p)$ cifre binarie. La chiamata ricorsiva al punto b avviene se per ogni $c = 1, \dots, b-1$ il valore di $\neg \text{VonN}(c, X \cup \{a\}, \neg g)$ vale false, e dunque è necessario calcolare $\neg \text{VonN}(b, X \cup \{a\}, \neg g)$ per il prossimo valore b . Non è invece necessario memorizzare l'insieme X dei punti proibiti, perché X si può ricostruire ripercorrendo all'indietro le chiamate ricorsive in sospeso (e aggiungendo il valore iniziale di X). Dunque lo spazio richiesto è $O(p \log(n))$. $p \cdot p = n$ è la dimensione della matrice G , dunque lo spazio richiesto è $O(\sqrt{n} \log(\sqrt{n}))$. Quindi GG è PSPACE nella dimensione n del grafo G .

Il Sipser include una prova che GG è PSPACE-completo, quindi tendiamo a escludere che sia in P. Infatti se GG fosse decidibile in tempo (e non solo in spazio) polinomiale, allora ogni problema in PSPACE sarebbe decidibile in tempo polinomiale. In questo caso avremmo $\text{PSPACE} = \text{P}$, e con maggior ragione $\text{NP} = \text{P}$, dato che NP è incluso in PSPACE per un corollario del Teorema di Savitch. La possibilità di avere $\text{PSPACE} = \text{P} = \text{NP}$ non si può ancora escludere, ma è considerata molto improbabile.





Il gioco del Nim è LogSpace (Problema 8.19 del Sipser)

Chiamiamo complessità di un gioco la complessità del problema di trovare una strategia vincente per un giocatore dato e una posizione data. Molti giochi noti, per esempio il gioco del GO con una scacchiera di dimensioni arbitrarie, sono PSPACE-completi. Forniamo un esempio di gioco con una strategia particolarmente semplice: il "NIM", che ha una strategia vincente in LogSpace: in dettaglio, spazio $\leq 2\log(n)$ e tempo all'incirca $3n$. Usando un algoritmo diverso: sia spazio che tempo $\leq n$.

Il NIM. La posizione iniziale del gioco del NIM consiste di un numero fisso di righe ciascuna con un numero fisso di sbarre. Per esempio, 3 righe rispettivamente con 1,2,3 sbarre:

I
II
III

Due giocatori, A e B, si alternano togliendo una o più sbarre ma da una sola riga. Il primo giocatore che non trova più sbarre perde. Ecco un esempio di partita, a partire dalla posizione I, II, III:

A  II III	B II II 	A  II	B 	A perde
---	---	--	--	---------

Per spiegare come calcolare una strategia vincente per il giocatore di turno, rappresentiamo ogni posizione con la lista dei numeri delle sbarre di ogni riga, scritto in forma binaria. Per esempio, la posizione I, II, III diventa 01,10,11. Indichiamo con "+" la **somma binaria senza riporto** (se vi interessa, questa operazione viene chiamata "*bitwise exclusive or*", abbreviata "*xor*"). Per esempio, $01+10+11 = 00$, dato che $0+1+1=0$ (senza riporto) e $1+0+1=0$ (senza riporto):

01
10
11
--
00

Una cifra binaria di una somma senza riporto vale 0 se un numero pari di addendi ha quella cifra uguale a 1, e vale 1 se un numero dispari di addendi ha quella cifra uguale a 1. D'ora in poi abbrevieremo "somma senza riporto" con "somma". Questa "somma" è associativa, commutativa, e soddisfa $S+S=0$ (per esempio: $3+3 = 11+11 = 00$ senza riporto). Proveremo ora che le posizioni la cui somma vale zero sono perdenti, mentre le posizioni la cui somma non vale zero sono vincenti, e la mossa vincente riduce a zero la somma delle righe. Prendiamo una somma $S = a+b+c > 0$, partiamo dalla numero 1 più a sinistra e risaliamo la somma finchè troviamo un numero 1:

```

a=00001
b=11010
c=11101
---|---
S=00110
    ↑
    00
    →

```

Sopra una cifra 1 c'è almeno un 1, se fossero solo zeri la cifra sarebbe 0. Dunque troviamo sempre un 1, in questo caso nella riga c. Definiamo ora una mossa che riduce a 0 la somma delle righe. La strategia vincente è rimpiazzare il numero c con $c+S$. Osserviamo innanzitutto che $d = c+S < c$, per esempio nel caso sopra abbiamo:

```

S=00110
c=11101
---|---
d=11011
    00
    ↓

```

Non è un caso che sia $c+S < c$: nella prima colonna a sinistra in cui S somma un 1 a una cifra di c abbiamo la cifra 1 in c, dunque otteniamo $1+1=0 < 1$, così otteniamo un risultato $< c$. Abbiamo così definito una mossa: ridurre il numero delle sbarre nella riga c lasciandone solo una quantità $S+c < c$. Questa strategia si può applicare tutte le volte che troviamo una posizione a somma $S > 0$. Controlliamo che la mossa produca una posizione a somma 0. Per esempio, se partiamo da a,b,c e muoviamo a,b,c+S, con $S=a+b+c$, la somma delle righe diventa: $a+b+(c+S) = (a+b+c)+S = S+S = 0$. Invece se partiamo da una posizione a somma zero e cambiamo una riga sola, otteniamo sempre una posizione a somma non zero (in almeno una colonna uno 0 diventa un 1 o viceversa, mentre le altre cifre della stessa colonna non cambiano, dunque la somma della colonna diventa 1).

La strategia che abbiamo descritto è vincente: noi riduciamo la somma delle righe a zero, l'avversario è costretto a riportare la somma a non-zero, noi la riduciamo di nuovo a zero e così via. Quindi noi abbiamo sempre una mossa da fare, quella definita dalla strategia. Dato che il gioco finisce e perde il primo giocatore che non ha mosse da fare, allora perde l'avversario.

Esempio. Partiamo da I, II, II, dunque 01,10,10. La somma vale $S = 01 + 10 + 10 = 01$, quindi chi inizia ha una strategia vincente. Il calcolo ci mostra che l'1 più a sinistra in S proviene dalla riga numero $i_0=1$ (la riga 01), e si trova nella colonna numero $j_0=2$:

colonna $j_0=2$

01	riga $i_0=1$
10	
10	
--	
01	

L'algoritmo ci richiede di trasformare riga numero $i_0=1$, che vale 01, in: $01 + S = 01 + 01 = 00$. Dunque la mossa vincente cancella la prima riga e ci porta a II,II. Se continuiamo a giocare, vediamo che l'algoritmo ci chiede di cancellare su una riga lo stesso numero di sbarre cancellato dal nostro avversario sull'altra riga. Alla fine l'avversario resterà senza mosse.

Il NIM è LOGSPACE. Rappresentiamo una posizione del NIM con cifre binarie separate da virgole: per esempio I, II, II diventa 01,10,10. Supponiamo di avere p righe ciascuna con q cifre binarie. La dimensione dell'input è $n=pq$. Ora calcoliamo lo spazio necessario per produrre la mossa vincente.

1. La somma senza riporto di una colonna di n cifre binarie si può fare in spazio $\log(q)$, dato che la rappresentazione binaria dell'indice j della colonna (un intero da 1 a q) richiede $\log(q)$ cifre, e che il risultato parziale vale 0 oppure 1.
2. Trovata la prima colonna j_0 di somma 1 (se esiste), la ricerca della riga i_0 da ridurre richiede spazio $\log(n)$, dato che la rappresentazione binaria dell'indice i della riga (un intero da 1 a p) richiede $\log(p)$ cifre.
3. La mossa vincente è ricopiare nell'output la posizione del nim ricevuta in input, ma con la riga i_0 rimpiazzata dalla somma della

riga i_0 e di tutte le altre. La somma d di tutte le righe di indice $i=1, \dots, q$, con la riga i_0 ripetuta due volte, si può fare in spazio $\log(q)$. Basta fare la somma della colonna di indice $j=1$, scrivere il risultato nell'output, e passare alla somma della colonna di indice $j=2$, eccetera. Tutto quello di cui abbiamo bisogno di ricordare sono i_0 , i e j , che richiedono uno spazio $\log(p) + \log(p) + \log(q) = \log(p) + \log(pq) = \log(p) + \log(n) \leq 2\log(n)$.

Tempo di Calcolo per il NIM. Infine, analizziamo il numero di passi di calcolo fatti. Per calcolare i_0 siamo passati da ogni cifra binaria una volta, tranne la colonna j_0 che abbiamo percorso due volte e ha lunghezza p . Dunque per calcolare i_0 abbiamo bisogno di $pq+p$ passi, all'incirca $n=pq$ passi. Con lo stesso ragionamento, per ricopiare l'input nell'output abbiamo bisogno di altri n passi, tranne che per la riga i_0 da modificare. Nella riga i_0 rimpiazziamo ogni cifra nella colonna j con la somma di tutta la colonna j più la cifra stessa, e questo richiede altri n passi. Infatti non abbiamo memorizzato la somma S per risparmiare spazio (S richiederebbe spazio q , dunque in media spazio \sqrt{n} , e al massimo n). Questo ci costringe a ricalcolare S . Totale: **$3n$ passi circa per eseguire il calcolo in spazio $\leq 2\log(n)$.**

Come ridurre il tempo di calcolo per il NIM aumentando lo spazio di calcolo. È interessante notare che le implementazioni correnti del gioco del Nim utilizzano una soluzione leggermente diversa: viene riscritto l'input anziché ricopiarlo (risparmiando n passi), dunque si usa spazio $n=pq$, e si usa uno spazio addizionale pari a q , la lunghezza di una riga, dove immagazzinare una nuova riga con la somma S di tutte le righe. Totale spazio: $pq+p = n+p$, circa n . Una volta trovata la riga i_0 da modificare, la rimpiazziamo aggiungendo la riga con la somma S di tutte le righe. Memorizzando S non dobbiamo ricalcolarlo, quindi percorriamo ogni cifra binaria che descrive il gioco una volta sola (risparmiando altri n passi), tranne che per la colonna j_0 e per la riga i_0 da modificare. Totale dei passi: $pq+p+q$, dunque circa $n=pq$. **Accettando di usare spazio n al posto di spazio $\leq 2\log(n)$, passiamo da un tempo di calcolo $3n$ a un tempo n .** In generale, accettando di usare più spazio risparmiamo tempo, perché se memorizziamo un valore evitiamo di ricalcolarlo.

Alcuni algoritmi, come quello per il NIM, hanno versioni che richiedono meno spazio, ma in genere preferiamo usare le versioni che richiedono meno tempo. In genere ci procuriamo più facilmente un maggiore spazio di memoria che un maggiore tempo di calcolo.

Suggerimenti per l' Esame di Calcolabilità e Complessità

1. Negli esami, l'errore più comune (tutti gli anni) è confondere il fatto che non vediamo nessuna soluzione per un problema con il fatto che la soluzione non ci sia. Per esempio:

- Non riusciamo ad immaginare nessun programma per stabilire se un dato enumeratore stamperà almeno una stringa oppure no. Ma per essere certi che non sia possibile farlo abbiamo bisogno di una prova. In questo caso possiamo provare che non esiste il programma richiesto, mostrando che altrimenti potremmo decidere il problema dell'accettazione.
- Un altro esempio. Non riusciamo ad immaginare nessun programma per stabilire in tempo polinomiale se esiste un cammino di Hamilton tra due punti di un grafo, perchè in ogni modo che immaginiamo dobbiamo controllare tutti i cammini o quasi, e questo richiede tempo esponenziale. Ma anche in questo caso avremmo bisogno di una prova per essere sicuri: e in questo caso una prova ancora non c'è.

Tenete presente che può capitare di credere a lungo che un problema richieda tempo esponenziale, e poi di cambiare idea. Questo non comporta una contraddizione, se non avevamo una prova che non esistevano soluzioni polinomiali. Per esempio il problema COMPOSITE, stabilire se un intero positivo è composto, è stato creduto esponenziale a lungo. Ma da vent'anni sappiamo che è polinomiale.

2. Siate precisi quando formulate una definizione. Questa forse è la richiesta più difficile del corso.

- Essere NP (non deterministico polinomiale) ed essere NP-completo (di massima difficoltà entro NP) sono due cose ben diverse.
- Un linguaggio è una cosa ben diversa dalla macchina che lo definisce.
- Un linguaggio vuoto è ben diverso da una stringa vuota. Un linguaggio vuoto non contiene nessuna stringa, neppure la stringa vuota.
- In una riduzione di un problema A ad un problema B risolviamo A con B, non B con A. Abbiamo provato che 3-SAT si riduce a CLIQUE (viene risolto da CLIQUE), e non il viceversa.

- Un teorema, per esempio il teorema di Cook (SAT NP-completo) è diverso dalla definizione di NP-completezza, è una proprietà della NP-completezza.
 - Un cammino di Hamilton (almeno quello che abbiamo visto noi) riguarda un grafo orientato, e ha un punto di inizio e uno di fine. Questi due punti fanno parte della formulazione del problema e sono richiesti come input da una macchina che lo risolve. Ogni problema ha bisogno di una definizione precisa.
 - State attenti ai dettagli, possono cambiare tutto. Un linguaggio L è indecidibile se ogni macchina M che definisce L diverge per almeno una stringa. Invece non è corretto dire che: "un linguaggio L è indecidibile se ogni macchina M che definisce L diverge per tutte le stringhe".
3. Giustificate e spiegate una risposta per quanto vi è possibile.
- Una riduzione di un problema A ad un problema B parte da una macchina R che risolve B e definisce una macchina M che risolve A . Ma non basta ricopiare la definizione di M vista sul libro, bisogna spiegare perché, ammesso che R risolva B , la macchina M risolve A .
 - Se usate termini impegnativi come riduzione, riduzione polinomiale, NP, NP-completo, O-grande, o-piccolo, spiegate cosa vogliono dire. Dovete farlo una volta sola, se ne avete bisogno una seconda volta vi basta dire che l'avete già fatto.
4. Cominciate sempre a rispondere partendo dai punti più importanti. Alcuni esempi.
- Per i problemi di HAMPATH, CLIQUE, SUBSET-SUM, 3-SAT, la cosa più importante da dire è che sono NP-completi, non che sono NP.
 - A volte una soluzione di un problema richiede tecniche particolari. Capita nella prova del Lemma 4.22 del Sipser, detto anche Lemma di Post, o nella prova del fatto che E_{TM} è negativamente decidibile. Queste prove richiedono di eseguire in parallelo due o più macchine. Si tratta di una richiesta inconsueta, che quindi va sottolineata e giustificata: perché in questo caso il parallelismo è necessario?

Messaggio del 7 Dicembre 2021 a proposito di una simulazione di esame

1. Fate attenzione ai dettagli delle definizioni. Dire " M è un enumeratore di un linguaggio L se E stampa tutte le stringhe di L " non basta, bisogna dire "tutte e sole". Allo stesso modo c'è differenza tra dire "almeno una volta, al più una volta,

esattamente una volta" dentro una definizione, di solito queste tre frasi portano a definizioni non equivalenti tra loro.

2. Quando dite "eseguimo M su w", con la descrizione della macchina M parte dell'input, spiegate che è possibile eseguire M su w perché esistono macchine universali, che prendono in input $\langle \langle M \rangle, w \rangle$, con $\langle M \rangle$ stringa che descrive M, ed eseguono una computazione di M su input w, quale che sia M.
3. "Definite un linguaggio che esprima il problema". Qui ricordatevi che per il Sipser un "linguaggio" è un insieme qualsiasi di stringhe. Attraverso l'appartenenza di una stringa a un insieme di stringhe possiamo esprimere qualsiasi domanda che sia stata resa precisa.
4. Attenti alle spiegazioni che fornite. "il problema del linguaggio vuoto è indecidibile perché altrimenti sarebbe possibile decidere il problema della fermata." Questo non è una vera spiegazione, se non dite come mai dall'ind decidibilità del problema della fermata segue l'ind decidibilità del problema del linguaggio vuoto. Se decidete di omettere una dimostrazione dovete dire: "dall'ind decidibilità del problema della fermata è possibile dedurre l'ind decidibilità del problema del linguaggio vuoto", così chiarite che l'affermazione non è ovvia ma non volete o non avete tempo di provarla. Se avete tempo, però, vi consiglio di inserire le prove che ricordate, specie se sono corte.
5. Ricordatevi che nel libro "M riconosce un linguaggio L" significa che "M decide positivamente L". In questo caso "M non accetta w" equivale a "M non termina oppure M rifiuta w", un decisore positivo di L può non terminare su un input w se w non sta in L.
6. "Provate l'equivalenza tra esistenza di un enumeratore per L e esistenza di un decisore per L". Una prova di equivalenza richiede un metodo per trasformare un enumeratore per il linguaggio L in un decisore positivo per lo stesso linguaggio L, e un decisore positivo per L in un enumeratore per lo stesso linguaggio L.
7. Attenti alle affermazioni "ovvie". "Ovviamente non è possibile decidere la fermata perché se la macchina non si ferma potrebbe essere in una normale fase di calcolo oppure essere in una situazione di non convergenza." Qui giustamente sottolineate come sia difficile prevedere se un calcolo si fermerà. Ma questa non basta a dimostrare che la fermata non è decidibile: basta solo a dire che appare poco probabile che lo sia, ma talvolta l'improbabile avviene. Per dire che la fermata non è decidibile occorre una prova, e come ho detto quando è possibile inserite le prove, specie se brevi. In questo caso, per esempio, potete assumere l'ind decidibilità dell'accettazione e per riduzione provare l'ind decidibilità della fermata.