Università di Pisa

Facoltà di Scienze
Matematiche, Fisiche e Naturali

Corso di Laurea in Matematica
Anno Accademico 2002/2003

Tesi di Laurea
30 marzo 2004

Complessità computazionale di un gioco combinatorio sui grafi.

Candidato
Marcello Mamino

Relatore
Prof. Alessandro Berarducci

Tenue rey, sesgo alfil, encarnizada
reina, torre directa y peón ladino
sobre lo negro y blanco del camino
buscan y libran su batalla armada.

No saben que la mano señalada
del jugador gobierna su destino,
no saben que un rigor adamantino
sujeta su albedrío y su jornada.

También el jugador es prisionero
(la sentencia es de Omar) de otro tablero
de negras noches y blancos días.

Dios mueve al jugador, y éste, la pieza.
¿Qué dios detrás de Dios la trama empieza
de polvo y tiempo y sueño y agonías?

JLB

Indice

Introduzione

La *teoria dei giochi combinatori* è nata allo scopo di indagare la struttura dei diversi *giochi*, o classi di giochi, e, dove possibile, *risolverli*.

I *giochi* cui si riferisce la nostra teoria, sono, innanzitutto, *games of no chance*: giochi nei quali il caso non gioca. Per questo si chiede che solo due giocatori vi prendano parte (onde evitare i complessi meccanismi delle possibili alleanze fra tre o più giocatori, che esulano dal determinismo), alternandosi successivamente nell'esecuzione di *mosse* su una *scacchiera* il cui stato è sempre noto ad entrambi. Una *partita* ad uno dei nostri giochi, se si conclude, risulta necessariamente nella vittoria di uno dei giocatori e nella sconfitta dell'altro: altri risultati (*puntate*, *scommesse*, *guadagni*, etc.) non ci interessano.

Il menzionato concetto di *soluzione* di un gioco, che è lo scopo ed il cardine della teoria, non è ben definito. A prima vista, potremmo ritenere risolto un gioco quando sappiamo quale dei due giocatori, se segue la strategia opportuna, può ottenere la vittoria. È famoso, però, il caso di Hex, un gioco inventato da John Nash nel 1948, del quale si sa che il primo giocatore possiede una strategia vincente; tuttavia *quale* sia questa strategia è tuttora ignoto.

Alla luce di questo parrà ragionevole ritenere risolto un gioco quando conosciamo *l'intera strategia* che assicura la vittoria ad un determinato giocatore, o la patta ad entrambi. Quest'approccio pone, tuttavia, un problema: *cosa significa conoscere una strategia?*

Chiaramente, quando abbiamo tabulato tutte le possibili posizioni di un gioco, e per ciascuna la corrispondente *mossa vincente*, allora possiamo dire di conoscerne la strategia. Il guaio è che una simile tabella rischia di essere infinitamente estesa, e, nei casi *interessanti* lo è; infatti, pare che la comune sensibilità dei matematici attribuisca alla *bellezza* di un gioco proprietà quali la *genericità* ed *estensibilità* delle regole di questo. Ed è inevitabile che un gioco arbitrariamente estensibile preveda una quantità arbitrariamente (infinitamente) grande di posizioni possibili.

In generale, l'esame di tutte le possibili configurazioni, poniamo scacchistiche, su una scacchiera $n \times n$ richiede un tempo esponenziale in n. Quando, come nel caso sicuramente noto del Nim, riusciamo ad accorciare questo tempo di calcolo per una strategia vincente, ad una durata polino-

miale nella *dimensione della scacchiera*, allora riteniamo di aver *risolto* il nostro gioco.

L'esempio dei *giochi ottali* mostra, però, che questa sicurezza è ingannevole. Noi abbiamo, infatti, un algoritmo polinomiale per determinare il risultato (il valore) di una data posizione in ciascuno di questi giochi; tuttavia congetturiamo che non sia *il più semplice algoritmo possibile*. I valori delle posizioni in questi giochi mostrano, infatti, un andamento apparentemente periodico che li renderebbe assai fcilemente calcolabili (più facilmente di quanto non siano coll'algoritmo polinomiale di prima), se solo si riuscisse a dimostrare che essi sono *necessariamente* periodici.

Tutte queste considerazioni mostrano come la *teoria della complessità computazionale* sia implicata e necessaria nella definizione stessa di *soluzione* di un gioco; quindi, in ultima analisi, nella teoria dei giochi.

Per entrare nell'argomento della tesi, notiamo che, dal punto di vista che andiamo delineando, la soluzione di sempre nuovi giochi (o l'elaborazione di sempre nuove tecniche di soluzione) è solo uno dei due fronti della battaglia. L'altro è la ricerca di quei giochi (o categorie di giochi) che non possono essere risolti: di quei giochi la cui complessità è, in qualche modo, *inerente* e *incomprimibile*.

Per quanto riguarda i giochi più noti e nobiltati dalla tradizione (essenzialmente gli scacchi, la dama, ed il go), si sa che questi appartengono tutti alla classe di quelli inerentemente complessi. Questa classe è popolata da giochi completi ora per **PSPACE**, ed ora per **EXPTIME**. A decidere quale di queste sarà la destinazione di un dato gioco è, usualmente, la *lunghezza massima delle partite* che questo ammette. Se sono permesse partite infinitamente (esponenzialmente) lunghe, allora il gioco cade in **EXPTIME**; se è posto un limite polinomiale alla lunghezza delle partite (come dalla *regola delle 50 mosse* negli scacchi), allora cade in **PSPACE** (osservazione, questa, in stretto accordo di analogia con i teoremi 2.1.7 e 2.1.8 presentati nel secondo capitolo).

La tesi verte sull'esame di un gioco proposto da Nowakowski e Wintler, che risulta appartenere al numero di quelli che non ammettono una soluzione (strategia) *semplice*. Si tratta di una sorta di *guardie & ladri* che chiameremo $C\&R$, giocato sui nodi di un grafo. Le *pedine* sono una n-tupla di poliziotti ed un ladro. I due giocatori si alternano, l'uno muovendo il ladro e l'altro i poliziotti (tutti contemporaneamente); il gioco ha termine quando il ladro viene catturato, ed in questo caso il giocatore che muove i poliziotti ha vinto.

La strategia di questo gioco è stata studiata da Neufeld e Nowakowski, che forniscono [13] alcune stime sul comportamento del numero dei poliziotti necessari per catturare un ladro in relazione a diverse operazioni sui grafi.

I grafi sui quali *un* poliziotto è sufficiente a catturare il ladro sono stati identificati da Nowakowski e Wintler [14] come tutti e soli i grafi *smantellabili*. Mente Aigner e Fromme dimostrano [1] che sui grafi planari 3 poliziotti sono sempre sufficienti per portare a termine la cattura.

È possibile dimostrare (lo fanno Berarducci ed Intrigila) che fissato k, esiste un algoritmo in grado di determinare in tempo polinomiale se k poliziotti bastano per catturare il ladro su un dato grafo. Tuttavia, il problema di determinare il numero minimo di poliziotti necessari a catturare il ladro su un grafo dato, il *copnumber* del grafo, è decisamente più complesso, e probabilmente non può essere risolto facendo affidamento su quelle caratteristiche del grafo che si possono calcolare in tempo polinomiale.

In questa tesi presento essenzialmente due risultati:

- la **EXPTIME**-completezza del gioco *C&R con data posizione iniziale*;
- e la **PSPACE**-hardness del gioco *C&R quando il ladro possa scegliere la sua posizione iniziale*, ossia la **PSPACE**-hardness di *copnumber*.

ove si è identificato un gioco col problema di determinare quel giocatore che dispone di una strategia vincente (o, se vogliamo, con il linguaggio costituito dalle posizioni vincenti di quel gioco per il primo giocatore).

Il primo di questi risultati, cui è dedicato il terzo capitolo della tesi, è già stato ottenuto da Goldstein [11], tuttavia la mia dimostrazione è differente ed indipendente dalla sua. Il secondo, che costituisce il quarto capitolo, è, a mio avviso, nuovo.

Rimane aperto il problema della classificazione precisa della complessità di *copnumber*, che come congettura Goldstein, congetturo anch'io essere **EXPTIME**-completo. È molto probabile, infatti, che questo possa essere dimostrato affinando ulteriormente la tecnica esposta in questa mia tesi. Infatti, il risultato di **PSPACE**-hardness esposto nella tesi suggerisce fortemente che il problema sia, in realtà, **EXPTIME**-completo, (poiché, come ho detto, i giochi *loopy*, ossia quelli che ammettono partite di lunghezza infinita, tendono alla completezza per **EXPTIME**).

Capitolo 1

Preliminari

Esporrò ora alcuni brevi richiami di teoria della complessità, per chiarire il quadro nel quale si collocheranno i nostri discorsi.

1.1 Macchine di Turing

Definizione 1.1.1 (TM). *Una* macchina di Turing *è una* 7*-tupla* $M = (Q, \Gamma, \Sigma, n, \delta, q_0, q_1)$ *ove:*

- $Q \supset \{q_0, q_1\}$ *è detto* insieme degli stati *di* M*,*
- $\Gamma = \Gamma' \sqcup \{\#\}$ *è detto alfabeto di* M*,*
- $\Sigma \subset \Gamma'$ *è detto* alfabeto di input *di* M*,*
- n *è detto* numero dei nastri *di* M*, e*
- $\delta : Q \times \Gamma^n \mapsto Q \times \Gamma'^n \times \{L, S, R\}^n$ *è una funzione parziale detta* funzione di transizione *di* M*.*

In aggiunta chiediamo che $\{q_1\} \times \Gamma^n \cap \mathcal{D}(\delta) = \varnothing$

Come il lettore già sa, queste macchine lavorano spostando le loro n *testine* su altrettanti *nastri*. Ciascun nastro è *infinito verso destra* e diviso il *celle* ciascuna delle quali ospita un simbolo di Γ...

Definizione 1.1.2. *Le configurazioni di* $M = (Q, \Gamma, \Sigma, n, \delta, q_0)$ *sono* $(2n + 1)$*-tuple* $(N_1, p_1, N_2, p_2, \ldots, N_n, p_n, q)$ *ove:*

- *per ogni* $1 \leq i \leq n$*,* $N_i \in \Gamma^*$ *rappresenta la porzione del* i*-esimo nastro già esaminata da* M*;*
- *per ogni* $1 \leq i \leq n$*,* $p_i \in \mathbb{N}^+$ *rappresenta la posizione dell'*i*-esima testina su* N_i *(*1 *indica che si trova sul primo carattere,* 2 *sul secondo, etc.);*

- $q \in Q$ *è lo stato nel quale si trova la* M*.*

Indichiamo con $\mathcal{C}_M$ *l'insieme delle configurazioni di* M*.*

...ed una macchina di Turing calcola (eh sì, è per questo che esiste) successivamente ripetendo le operazioni seguenti: legge gli n simboli che trova sotto le sue testine, quindi, in accordo con la funzione di transizione, scrive n nuovi simboli ed eventualmente sposta le testine a destra o a sinistra di una cella.

Definizione 1.1.3. *Date due configurazioni*

$$C = (N_1, p_1, N_2, p_2, \ldots, N_n, p_n, q)$$

e

$$C' = (N'_1, p'_1, N'_2, p'_2, \ldots, N'_n, p'_n, q)$$

di una medesima TM $M = (Q, \Gamma, \Sigma, n, \delta, q_0)$*, si dice che* C' *è immediatamente successiva a* C *(*$C \vdash_M C'$*) quando*

$$\delta(q, N_1[p_1], N_2[p_2], \ldots, N_n[p_n]) = (q', \alpha_1, \alpha_2, \ldots, \alpha_n, m_1, m_2, \ldots, m_n)$$

e, per ogni $1 \leq i \leq n$*, vale:*

$$p'_i - p_i = \begin{cases} 1 & \text{se } m_i = R \\ 0 & \text{se } m_i = S \\ -1 & \text{se } m_i = L \end{cases}$$

$$|N'_i| = \begin{cases} |N_i| + 1 & \text{se } p'_i = |N_i| + 1 \\ |N_i| & \text{altrimenti} \end{cases}$$

$$N'_i[j] = \begin{cases} N_i[j] & \text{se } j \neq p_i \text{ e } j \leq |N_i| \\ \# & \text{se } j = |N'_i| = |N_i| + 1 \\ \alpha_i & \text{se } j = p_i \end{cases}$$

Definizione 1.1.4. *Una* computazione *per* M *è una successione (possibilmente infinita) di configurazioni* $C_1, C_2, \ldots$*, tali che, per ogni coppia* C_i, C_{i+1} *di configurazioni successive nella computazione, valga* $C_i \vdash_M C_{i+1}$*. La computazione di* M *su input* $s \in \Sigma^*$ *è* la più lunga *computazione per* M *con* $C_1 = (s, 1, \langle\#\rangle, 1, \langle\#\rangle, 1, \ldots, q_0)$*.*

Diciamo che la computazione di M *su input* s*:*

- termina*, se è di lunghezza finita;*
- accetta*, se termina e la sua ultima configurazione è nello stato* q_1*.*

Questa definizione è giustificata in quanto a noi interessa essenzialmente classificare la complessità di problemi *di riconoscimento di linguaggi*. In pratica dato un linguaggio $L \subset \Sigma^*$ (nel nostro caso la codifica delle posizioni vincenti di un gioco) ci chiederemo quante *risorse di calcolo* una macchina di Turing deve impegnare per riconoscerlo (ove con *riconoscere* un linguaggio L intendo accettare tutte e sole le stringhe di L).

Osservazione 1.1.5. *Quanto alla definizione precedente, si osservi che continuare la computazione all'infinito non è l'unico modo che ha M per rifiutare (non accettare) una stringa dell'input. M può terminare la computazione senza entrare nello stato q_1 semplicemente* tentando di superare il limite sinistro del nastro.

1.2 Complessità

Ora darò una definizione precisa del concetto di *risorse di calcolo* impiegate da una macchina di Turing.

Definizione 1.2.1. *Sia $t : \mathbb{N} \mapsto \mathbb{N}$; diciamo che un linguaggio $L \subset \Sigma^*$ è in* **TIME**(t)*, quando esiste una macchina di Turing M che accetta in input l'alfabeto Σ tale che:*

- *per ogni $\sigma \in \Sigma^*$, $\sigma \in L$ se e solo se M accetta σ;*
- *per ogni $\sigma \in \Sigma^*$, la computazione di M su σ ha meno di $t(|\sigma|)$ elementi (M impiega meno di $t(|\sigma|)$ transizioni per capire se σ appartiene o no ad L).*

Definizione 1.2.2. *Sia $s : \mathbb{N} \mapsto \mathbb{N}$; diciamo che un linguaggio $L \subset \Sigma^*$ è in* **SPACE**(s)*, quando esiste una macchina di Turing M che accetta in input l'alfabeto Σ tale che:*

- *per ogni $\sigma \in \Sigma^*$, $\sigma \in L$ se e solo se M accetta σ;*
- *per ogni $\sigma \in \Sigma^*$, essendo $(N_1, p_1, N_2, p_2, \ldots, N_n, p_n, q)$ la configurazione finale della computazione di M su σ, si ha che per ogni $1 \leq i \leq n$ $|N_i| \leq s(|\sigma|)$.*

Osservazione 1.2.3. *Al lettore che già conosca l'argomento, la nostra definizione di* **SPACE**(s) *parrà poco parsimoniosa; per esempio, secondo tale definizione, la classe* **SPACE**$(\log)$ *è vuota. Tuttavia, questa è stata scelta per semplicità, ed una discussione più accurata dell'esempio precedente si troverà nella sezione dedicata a* **LOGSPACE**.

Osservazione 1.2.4. *Per pulizia ed eleganza, non abbiamo dotato le nostre macchine di uno* stato rifiutante*, la qual cosa ci costa osservare che* l'essenza *delle definizioni precedenti non cambierebbe, se M fosse obbligata a*

rifiutare esplicitamente le stringhe che non sono in L nei limiti di tempo (o spazio) imposti. Questa nostra osservazione apparirà più evidente alla luce dei teoremi che mi accingo ad enunciare.

Teorema 1.2.5 (Speedup theorem). *Per ogni* $\epsilon > 0$

$$\mathbf{TIME}(f) \subset \mathbf{TIME}(f')$$

con $f'(n) = \lceil \epsilon f(n) \rceil + n + 1$.

Teorema 1.2.6. *Per ogni* $\epsilon > 0$

$$\mathbf{SPACE}(f) \subset \mathbf{SPACE}(f')$$

con $f'(n) = \lceil \epsilon f(n) \rceil + n + 1$.

I teoremi precedenti, fin tanto che consideriamo limiti di spazio e tempo più che lineari, ci giustificano nell'uso che faremo delle notazioni $\mathbf{TIME}(O(f))$ e $\mathbf{SPACE}(O(f))$; infatti, in forza di detti teoremi, possiamo osservare che:

Osservazione 1.2.7. *Se per ogni* n, $f(n) > n + 1$, *e* $lim_{n\to\infty} \frac{n}{f(n)} = 0$, *allora per ogni* $\epsilon > 0$

$$\mathbf{TIME}(f) = \mathbf{TIME}(\epsilon f)$$

e

$$\mathbf{SPACE}(f) = \mathbf{SPACE}(\epsilon f)$$

Il lettore potrebbe chiedersi se, come è possibile risparmiare un fattore lineare sulla quantià di risorse impiegate per risolvere un determinato problema, sia possibile anche risparmiare sul numero dei nastri dei quali sono dotate le macchine di Turing. Quello che si può vedere è che, finché si considerano classi di complessità spaziali, le macchine con *un solo* nastro sono precisamente potenti quanto quelle con un numero arbitrario di nastri; mentre le classi temporali sembrano non tollerare l'eccessiva penuria di nastri (si vede facilmente, però, che riducendo ad uno il numero dei nastri, il tempo di computazione aumenta al più di un fattore quadratico).

1.3 Alcune classi

In futuro avremo principalmente a che fare con le classi **EXPTIME**, **PSPACE** e **LOGSPACE**. A **LOGSPACE** deve essere dedicato un paragrafo a parte, un po' per le sue caratteristiche peculiari ed un po' perché quello che a noi interessa dello spazio logaritmico sono piuttosto le funzioni che si possono calcolare entro questo limite, che non i linguaggi che vi si possono riconoscere.

Le classi **P**, **EXPTIME** e **PSPACE** sono definite come segue:

$$\mathbf{P} = \bigcup_{c,k\in\mathbb{N}} \mathbf{TIME}(c^{\bullet^k})$$

$$\mathbf{EXPTIME} = \bigcup_{c,k\in\mathbb{N}} \mathbf{TIME}(c^{\bullet^k})$$

$$\mathbf{PSPACE} = \bigcup_{c,k\in\mathbb{N}} \mathbf{SPACE}(c^{\bullet^k})$$

(dove con $\phi(\bullet)$ intendiamo la funzione che mappa n in $\phi(n)$)

L'uso di simili oggetti, dai contorni apparentemente sfumati, è, in realtà, non soltanto giustificato, ma quasi necessario. Infatti, per meglio capire *il significato* di queste definizioni dobbiamo osservare che la macchina di Turing è soltanto uno dei tanti *modelli di computazione* possibili. Per di più le noste macchine prendono in input delle stringhe, ma *non* perché queste siano materia computabile comoda *per noi*, bensì perché le stringhe sono la cosa più naturale per questo preciso modello di computazione. Potremmo, ad esempio, fissare come modello di computazione la *macchina a registri*, ed in questo caso l'input sarebbe verosimilmente un numero intero; potremmo considerare come modello i *circuiti booleani*, ed in questo caso eseguiremmo i nostri calcoli su stringhe di zeri ed uni di lunghezza fissata; etc. In ogni caso è chiaro che i tipi di dati che vorremmo fare oggetto di computazione sono tali e tanto variegati (formule, insiemi, grafi, circuiti, macchine di Turing, posizioni scacchistiche, etc.) che qualunque modello scegliamo per la computazione, prima di questa sarà sempre necessaria una fase di *codifica* dell'input, per portare questo in quella forma che alla macchina calcolatrice che abbiamo fissato è più congegnale.

Quest'osservazione, però, impone due considerazioni. In prima istanza la *complessità di un problema* non è concetto ben definito, né ben definibile in modo sufficientemente generale: anche fissato il modello di computazione al quale ci riferiamo, rimane sempre la possibilità che esistano codifiche per gli elementi del problema più convenienti di altre (e sappiamo che la codifica più conveniente di tutte, ossia quella che contiene sia i dati del problema che la soluzione, esiste sempre!). Tuttavia è spesso *ragionevole* enunciare risultati di complessità prescindendo dalle codifiche precise, nella misura in cui le codifiche *ragionevoli* dell'input non alterano la complessità del problema. In seconda istanza, dovremo assicurarci che le nostre classi di complessità siano abbastanza *robuste* contro i cambiamenti di codifica *ragionevoli* dell'input.

Consideriamo per esempio le possibili codifiche per i grafi. Si può descrivere un grafo indicando la lista dei suoi nodi e, successivamente, la lista degli archi scritti come coppie di nodi. Altresì lo stesso grafo si può descrivere come matrice $n \times n$ di uni e zeri, ove n è il numero dei nodi. Il lettore vedrà che la prima codifica richiede $O\big((n+a)\log(n)\big)$ simboli dove n è il numero dei nodi ed a il numero degli archi, mentre la seconda n^2 sim-

boli. Qual'è la più conveniente? Intuitivamente un grafo con molti archi è più convenientemente scritto nella seconda, mentre un grafo con pochi archi nella prima; tuttavia quest'esempio illustra bene come la lunghezza delle codifiche ragionevoli sia definita solo *a meno di fattori polinomiali*. Pertanto è ugualmente necessario che le classi di complessita nelle quali classifichiamo i problemi siano stabili *per riscalamenti polinomiali dell'input* (e tali sono **P**, **EXPTIME** e **PSPACE**).

Per di più queste classi hanno la proprietà di essere fra quelle che si definiscono **classi sintattiche**. Ossia, per ciascuna di esse, è possibile fissare una forma standard (una codifica!) nella quale descrivere le macchine di Turing in modo tale che:

- ogni macchina scritta in una forma standard *definisce* un linguaggio della corrispondente classe di complessità;
- ogni linguaggio in una classe di complessità sintattica è deciso da una macchina di Turing scritta nella corrispondente forma standard.

Questo ci dà una definizione quasi costruttiva dei linguaggi in una classe di complessità sintattica, la quale è ciò che permette di trovare problemi **completi** per queste classi (ad oggi non si conoscono problemi completi per una sola classe che non sia sintattica)... Tuttavia, prima di parlare della completezza, dovremo fare la conoscenza di **LOGSPACE**.

1.4 LOGSPACE

In questa sezione intendiamo dare qualche ragguaglio sulla zoologia di **LOGSPACE**. In breve ci interessa dare un'esposizione chiara di ciò che può essere calcolato in spazio logaritmico, cosicché il lettore che ancora non conosca questa classe di complessità non senta troppo angusti i limiti di spazio che essa impone.

Precisamente, a noi interessano macchine di Turing che, lavorando in spazio logaritmico, prendono in input una stringa e restituiscono una stringa in output di lunghezza polinomialmente legata all'input. Dal momento che il limite di spazio imposto è così stretto, dovremo prestare attenzione affinché le nostre macchine non *riusino* la porzione di nastro destinata all'input oppure all'output (altrimenti disporrebbero di una quantità di spazio polinomiale!).

Il modo più semplice è considerare le **LOGSPACE**-TM definite qui di seguito:

Definizione 1.4.1 (LOGSPACE-TM). *Una* **LOGSPACE***-TM è una macchina di Turing a* 3 *nastri: un nastro* di input *(diciamo il primo) dal quale la macchina può solo leggere (e mai scrivere), un nastro* di lavoro *(il secondo) che la macchina può usare tanto in lettura quanto in scrittura ma*

soltanto per una porzione di lunghezza logaritmica rispetto alla lunghezza dell'input, ed un nastro di output *(il terzo) sul quale la macchina può solo scrivere (e mai leggere). Supponiamo inoltre che la nostra macchina debba eseguire soltanto una quantità polinomiale di passi su qualunque input.*

Definizione 1.4.2. *Diciamo che una funzione è calcolabile in* **LOGSPACE***, quando è la funzione calcolata da una* **LOGSPACE***-TM.*

Osservazione 1.4.3. *Questa definizione necessita di alcune osservazioni.*

- *Una macchina di Turing che lavori in spazio logaritmico e che non* vada in loop *deve eseguire soltanto una quantità polinomiale di passi (poiché dispone di una quantità soltanto polinomiale di configurazioni possibili); questo ci assicura che la condizione sul numero polinomiale dei passi non restringa il campo d'azione delle nostre* **LOGSPACE***-TM.*

- *È possibile definire una* forma standard *in cui presentare le* **LOGSPACE***-TM, in modo tale che detta condizione sul numero dei passi sia automaticamente verificata; questo prova che non è necessario affrontare il problema della fermata per riconoscere una* **LOGSPACE***-TM valida. (Su quale sia questa forma standard, forse, dirò in seguito.)*

- *La nostra definizione non impone una limitazione indebita dotando le macchine di un solo nastro di lavoro; infatti notoriamente le classi di complessità spaziali sono indipendenti dal numero di nastri concessi alle macchine di Turing.*

- *Tuttavia, un osservatore acuto può intuire come una* **LOGSPACE***-TM costruita in modo opportuno sia in grado di* sfruttare *una quantità più che logaritmica di spazio. Il trucco sottile, che la nostra macchina può mettere in atto, consiste nel riscrivere più volte le medesime caselle del nastro di output. Intuitivamente, infatti, una macchina in grado di scrivere l'output soltanto un carattere per volta ed in ordine a partire dal primo, appare meno potente di una* **LOGSPACE***-TM che può, per esempio, iniziare l'output dal fondo e* cambiare idea *più volte su un determinato carattere dell'output mano a mano che procede nella computazione. Dimostreremo che il nostro osservatore non è stato* abbastanza *acuto.*

Lemma 1.4.4. *Ogni* **LOGSPACE***-TM M è equivalente a (calcola la medesima funzione di) una* **LOGSPACE***-TM M' che verifica le seguenti condizioni*

- *M' non può spostare verso sinistra (cioè indietro) la testina del nastro di input;*

- *M' non può scrivere sul nastro di output un carattere diverso da __ senza spostare la testina di quel nastro (verso destra) durante la medesima transizione;*
- *M' non può scrivere sul nastro di output __ e spostare la testina di quel nastro durante la medesima transizione.*

Su M facciamo l'ipotesi non restrittiva che non abbia __ fra i simboli del suo alfabeto.

Osservazione 1.4.5. *È chiaro che le condizioni precedenti ad altro non servono, se non ad assicurarci che M' scriva l'output* da sinistra a destra *e senza la possibilità di* cambiare idea *su un carattere dopo averlo scritto.*

Dimostrazione. L'idea è di far simulare ad M' la macchina M senza scriverne l'output ma *ricordandosi* del valore della prima cella del nastro di output. Terminata la simulazione M' scrive il valore *definitivo* che ha ottenuto per la prima cella e ripete il medesimo procedimento per la seconda; poi per la terza, e così via, simulando *un'intera computazione* di M per ogni caratte di output, finché ce ne sono. Ora procediamo alla descrizione di M'.

Il nastro di lavoro di M' è diviso in tre tracce. Sulla prima M' simulerà il fuzionamento di M, mentre sulle altre due troveranno spazio due numeri interi scritti in notazione binaria. M', inoltre, ha, nel suo controllo finito, una *cella di memoria* che può contenere un simbolo dell'alfabeto di output di M oppure __. M' inizia il suo lavoro scrivendo il simbolo 0 sulle prime celle della seconda e della terza traccia del nastro, e memorizzando il simbolo __ nella sua cella; quindi avvia la simulazione di M. Va da sé che M' non scrive l'output di M, bensì reagisce alle azioni di M sul nastro di output nel modo seguente:

- se M sposta la testina del nastro di output verso destra, M' incrementa il valore del numero scritto sulla seconda traccia di 1;
- se M sposta la testina del nastro di output verso sinistra, M' decrementa il valore del numero scritto sulla seconda traccia di 1;
- se M scrive un simbolo sul nastro di output, M' confronta i due numeri scritti sulla prima e sulla seconda traccia: se essi coincidono, allora memorizza il simbolo nella sua cella di memoria, altrimenti lo ignora.

Giunta alla fin della simulazione, M' controlla il valore della cella di memoria: se questo è __ termina. Altrimenti scrive sul nastro di output il simbolo che si trova nella cella di memoria, incrementa di 1 il numero scritto sulla prima traccia, inizializza nuovamente la cella a __, quindi riprende daccapo la simulazione di M.

Non vi sono dubbi sul fatto che M' possa eseguire le operazioni richieste sui numeri interi, e che questi non superino una lunghezza logaritmica segue

dal fatto che non superano mai *in valore* la lunghezza della stringa di output di M. □

Nonostante, però, si possa scrivere ordinatamente, non è possibile imporre una simile restrizione anche sulla lettura senza che la potenza delle nostre macchine ne venga intaccata. Ne segue che la composizione di due funzioni in **LOGSPACE** non può essere calcolata in **LOGSPACE** scrivendo l'output della prima (la cui lunghezza è polinomiale) sul nastro di lavoro, quindi calcolando la seconda; né si può calcolare passando l'output della prima *un carattere alla volta* alla seconda, poiché questa necessita, per essere calcolata, di poter leggere l'input *disordinatamente*. Tuttavia, il lemma precedente avrà dato al lettore un'idea su come risolvere la questione...

Lemma 1.4.6. *La composizione di due funzioni calcolabili in* **LOGSPACE** *è calcolabile in* **LOGSPACE**.

Dimostrazione. Supponiamo che le nostre due funzioni componende siano calcolate dalle **LOGSPACE**-TM M ed M'; quindi costruiamo una **LOGSPACE**-TM M'' che ne calcoli la composizione. Per economia di spaziotempo non darò una descrizione esplicita di M'', in quanto la costruzione è precisamente analoga a quella del lemma precedente; tuttavia spiegherò come questa debba essere portata a termine.

In breve, M'' simula M' tenendo memoria di dove si trova la sua testina di input. Ad ogni transizione di M', M'' determina qual è il carattere di input che M' sta leggendo simulando un'intera computazione di M nella maniera delineata durante la dimostrazione del lemma precedente. Come prima, M'' non necessita di una quantità più che logaritmica di memoria per rappresentare la posizione delle testine di M ed M' sotto forma di numeri interi, e tanto basta per concludere la dimostrazione. □

Osservazione 1.4.7. *In realtà tutto quello che abbiamo fatto nel lemma 1.4.4 è stato comporre M con la funzione identità nel modo ora descritto.*

Ora, un'osservazione più attenta della dimostrazione precedente renderà palese come lo stesso argomento basti per ottenere di più di quanto il lemma sostenga: precisamente che è possibile *da una* **LOGSPACE**-*TM* richiamare un arbitrario numero di volte un'altra **LOGSPACE**-TM. In altre parole, seguendo la notazione di [16]

Lemma 1.4.8.

$$\mathbf{LOGSPACE}^{\mathbf{LOGSPACE}} = \mathbf{LOGSPACE}$$

la qual cosa ha un'aspetto assai scontato e del tutto innocuo.

Come si è visto le nostre **LOGSPACE**-TM possono facilmente *tenere in memoria* una quantià fissata (indipendentemente dall'input) di numeri

interi, a patto che questi siano limitati da una funzione polinomiale della lunghezza dell'input. In più possono con semplicità incrementare e decrementare il valore di uno di questi; ne segue che dati due interi ne possono calcolare la somma iterativamente incrementando uno di questi e decrementando l'altro. Iterativamente sommando o sottraendo, allora, possono anche calcolare il prodotto ed il quoziente fra due numeri interi (dette iterazioni sono possibili, in generale, in forza dell'osservazione che abbiamo appena svolto sulla possibilità di *richiamare* una **LOGSPACE**-TM da un'altra). Abbiamo quindi il seguente

Lemma 1.4.9. *Le operazioni aritmetiche (su numeri scritti in* unario*) sono calcolabili in* **LOGSPACE**.

1.5 Space- & time- constructibility

Ora che abbiamo definito le nostre classi di complessità, ci sembrerebbe naturale che date due funzioni da $\mathbb{N}$ in $\mathbb{N}$, f e g, la prima delle quali cresce *abbastanza più rapidamente* della seconda, la classe $\mathbf{TIME}(f)$ ($\mathbf{SPACE}(f)$) contenesse strettamente $\mathbf{TIME}(g)$ ($\mathbf{SPACE}(g)$). Tuttavia, purtroppo, così non è, come dimostra il seguente

Teorema 1.5.1 (Gap Theorem). *Per ogni funzione calcolabile totale f, tale che per ogni $n \in \mathbb{N}$ valga $f(n) \geq n$, esiste una funzione calcolabile totale strettamente crescente g, tale che* $\mathbf{SPACE}(g) = \mathbf{SPACE}(f \circ g)$.

dal quale seguono, come il lettore avrà intuito, risultati assai poco intuitivi, ad esempio

Corollario 1.5.2. *Esiste una funzione ricorsiva totale strettamente crescente f, tale che* $\mathbf{TIME}(f) = \mathbf{SPACE}(f) = \mathbf{TIME}(f^f)$.

Onde ridurre alla ragione la tendenza del nostro modello di calcolo a simili anomalie, è necessario porre qualche limitazione sulle funzioni che usiamo per definire le classi di complessità (il lettore che consideri strano esplicitare i limiti che si pongono su certe definizioni *dopo* che queste sono state date sarà soddisfatto prima del termine di questo paragrafo).

Definizione 1.5.3 (time constructibility). *Una funzione $f : \mathbb{N} \mapsto \mathbb{N}$ si dice* time constructible *quando esiste una macchina di Turing M che prende in input l'alfabeto $\{1\}$ e su input $\langle \overbrace{11\ldots1}^{n \text{ volte}} \rangle$ accetta dopo esattamente $f(n)$ transizioni.*

Definizione 1.5.4 (space constructibility). *Una funzione $f : \mathbb{N} \mapsto \mathbb{N}$ si dice* space constructible *quando esiste una macchina di turing M, avente un unico nastro, che prende in input l'alfabeto $\{1\}$ e su input $\langle \overbrace{11\ldots1}^{n \text{ volte}} \rangle$*

accetta restituendo sul nastro la stringa $\langle \overbrace{11\ldots1}^{f(n)\ volte} \rangle$ *(se questo avviene, automaticamente, M lavora in spazio* $f(n)$*).*

Per quanto queste definizioni sembrino richiedere notevole finezza al fine di costrure macchine che usino *precisamente* il numero giusto di caselle, o peggio che compiano *esattamente* un certo numero di transizioni, in realtà i seguenti teoremi semplificano parecchio le cose.

Teorema 1.5.5. *Se la funzione* $f : \mathbb{N} \mapsto \mathbb{N}$ *è tale che, per un opportuno* $\epsilon > 0$ *e per ogni* $n \in \mathbb{N}$*, vale* $f(n) \geq n(1+\epsilon)$*; allora* f *è time constructible se e solo se è calcolabile in tempo* $O(f)$ *(esiste una macchina di Turing che legge* $\langle \overbrace{11\ldots1}^{n\ volte} \rangle$ *ed accetta dopo* $O(f)$ *transizioni restituendo sul primo nastro* $\langle \overbrace{11\ldots1}^{f(n)\ volte} \rangle$*).*

Teorema 1.5.6. *La funzione* $f : \mathbb{N} \mapsto \mathbb{N}$ *è space constructible se e solo se è calcolabile in spazio* $O(f)$*.*

Per quanto, in questa sede, sia fuori luogo dimostrare le due proposizioni precedenti, si osservi che non si tratta di nulla di particolarmente ostico: questi due teoremi, come si sarà intuito, sono anzi parenti molto stretti dei teoremi 1.2.5 e 1.2.6 già enunciati.

Alla luce di questi e del lemma 1.4.9 apparirà chiaro i seguente

Lemma 1.5.7. *Le funzioni*

$$n \mapsto p(n)$$
$$n \mapsto p(n)^{q(n)}$$

ove p *e* q *sono polinomi a coefficienti interi positivi sono sia space constructible che time constructible.*

che ci assicura della *ragionevolezza* delle classi che stiamo considerando.

L'utilità di detta ragionevolezza contro l'anomalia espressa dal teorema 1.5.1 è resa esplicita dai seguenti

Teorema 1.5.8 (time hierarchy theorem). *Se* f *è una funzione time constructible tale che* $f(n) \geq n$*, allora:*

$$\mathbf{TIME}(f) \subsetneq \mathbf{TIME}(f \log^2(f))$$

Teorema 1.5.9 (space hierarchy theorem). *Se* f *è una funzione space constructible, allora:*

$$\mathbf{SPACE}(f) \subsetneq \mathbf{SPACE}(f \log(f))$$

da cui segue, per altro, che

Lemma 1.5.10.

$$\mathbf{P} \subsetneq \mathbf{EXPTIME}$$

1.6 Riduzioni & completezza

In questo paragrafo verrà introdotto il concetto di completezza di un linguaggio per una determinata classe di complessità. Intuitivamente un linguaggio è completo per una certa classe di complessità quando ad esso si può *ridurre* ogni altro linguaggio di quella classe. L'idea è che l'operazione di riduzione debba essere di complessità molto inferiore alla classe che si sta esaminando, tanto che sia giustificato considerare quella classe *a meno di riduzioni*. Tuttavia, questo non è un concetto *univocamente definito* nella letteratura, in quanto si considerano comunemente svariati generi di riduzione; quello a cui noi faremo riferimento è la **LOGSPACE**-riduzione che ora definisco.

Definizione 1.6.1. *Diciamo che un linguaggio L è* **LOGSPACE***-riducibile ad un linguaggio L' ($L <_{\mathbf{LOGSPACE}} L'$) quando esiste una funzione F calcolabile in* **LOGSPACE** *che mappa stringhe nell'alfabeto di L in stringhe dell'alfabeto di L' tale che per ogni stringa s nell'alfabeto di L, $s \in L$ se e solo se $F(s) \in L'$.*

Osserviamo che se L' è in **EXPTIME** (o **PSPACE**, ma anche **P** o **NP**, o **PP**, o, pleonasticamente, **LOGSPACE**) ed $L <_{\mathbf{LOGSPACE}} L'$, allora anche L è nella medesima classe di complessità. Questo è vero poiché una **LOGSPACE**-riduzione non può aumentare che di un fattore polinomiale la lunghezza dell'input e le classi che abbiamo elencato sono tutte *tolleranti* rispetto a questo effetto.

Definizione 1.6.2 (Hardness). *Un linguaggio L è* hard *per la classe (generica)* **C** *quando ogni linguaggio in* **C** *si riduce in* **LOGSPACE** *ad L.*

Definizione 1.6.3 (Completezza). *Un linguaggio L è* completo *per la classe (generica)* **C** *quando*

- *L appartiene alla classe* **C** *e*
- *L è hard per la classe* **C**.

L'importanza del concetto di completezza risiede nel fatto che, in un certo senso, un linguaggio completo per una determinata classe *cattura* la complessità di quella classe; un risultato di completezza ci assicura, infatti, che risolvere un certo problema completo per **C** non è più facile di risolvere *ogni* altro problema della classe **C**. Questo non fornisce *sempre* una classificazione precisa della complessità di quel problema, tuttavia, grazie alla completezza, sappiamo che l'unico elemento di variabilità è la collocazione nella gerarchia delle complessità *dell'intera classe*.

I linguaggi completi per una certa classe ci danno, insomma, una *immagine concreta* del *concetto astratto* di classe di complesstà: ci assicurano di non aver radunato una collezione di linguaggi *a caso* sotto l'etichetta di

una medesima classe. Infatti, una classe che abbia *un* linguaggio completo rappresenta un vincolo assai stretto fra i suoi elementi: rappresenta *quel* linguaggio.

A questo punto nulla ci assicura che *esistano* linguaggi completi per una classe; tant'è che per parecchie classi di complessità non si danno di tali linguaggi. Però, come abbiamo accennato in precedenza, questi esistono per quelle classi che si definiscono *sintattiche*. Nel seguito della tesi incontreremo svariati linguaggi completi per **EXPTIME** e **PSPACE**.

1.7 Giochi

Nel seguito ci interesseremo della complessità di alcuni *giochi combinatori* (o semplicemente *giochi*). I giochi che studieremo sono giocati da due giocatori, che chiameremo I e II, i quali si alternano nell'eseguire le proprie mosse su una scacchiera della quale conoscono *completamente* la configurazione. Non sono ammessi, nei nostri giochi, elementi di casualità o *mosse segrete* che un giocatore esegue all'oscuro dell'altro.

Tecnicamente diamo le seguenti definizioni.

Definizione 1.7.1 (Gioco). *Un* gioco $G = (\mathcal{C}(G), \vdash_G^I, \vdash_G^{II})$ *è il fato di un insieme* $\mathcal{C}(G)$ *(l'insieme delle configurazioni) e due relazioni binarie* $\vdash_G^I$ *e* $\vdash_G^{II}$ *definite fra elementi di* $\mathcal{C}$*.*

Definizione 1.7.2 (Posizioni vincenti). *Dato un gioco come nella definizione precedente, sono definiti gli insiemi* $\mathcal{W}^I(G)$ *e* $\mathcal{W}^{II}(G)$ *(delle posizioni vincenti, rispettivamente, per* I *e per* II*) come i più piccoli sottoinsiemi di* $\mathcal{C}(G)$ *tali che:*

$$p \in \mathcal{W}^I(G) \longrightarrow \exists q : p \vdash_G^I q \wedge \big(\forall r : q \vdash_G^{II} r \rightarrow r \in \mathcal{W}^I(G)\big)$$
$$p \in \mathcal{W}^{II}(G) \longrightarrow \exists q : p \vdash_G^{II} q \wedge \big(\forall r : q \vdash_G^{I} r \rightarrow r \in \mathcal{W}^{II}(G)\big)$$

Nelle definizioni precedenti non si è fatto riferimento alla *attività* dei giocatori nell'alternanza delle mosse; né ai giocatori stessi, se non fosse per gli apici di qualche simbolo... Tuttavia è chiaro che la relazione $\vdash_G^I$ indica quali sono le mosse possibili per il giocatore I in una determinata posizione, e simmetricamente $\vdash_G^{II}$ quelle possibili per II. Nonostante la staticità imposta dalla *matematica* nelle sue definizioni, noi faremo spesso (e volentier assai) uso di un linguaggio più ludico, dicendo che il tale giocatore *esegue* la tale mossa, *è forzato* a entrare nella tale posizione, *vuole* questo o quell'altro, etc. Verificherà il lettore che le nostre abitudini semantiche non sono poi del tutto ingiustificate se il fine è la chiarezza, e che i concetti espressi possono essere formalizzati e ricondotti induttivamente alla definizione 1.7.2, a patto di renderli incomprensibili.

Secondo la definizione precedente un giocatore *perde* quando si trova in una posizione nella quale non può eseguire alcuna mossa (la qual cosa è

assai ragionevole: come osserva Conway *dal momento che siamo convinti di perdere quando non riusciamo a trovare nessuna buona mossa, ovviamente dovremmo perdere quando non possiamo trovare nessuna mossa in generale*). Quando daremo condizioni per la vittoria (risp. la sconfitta) di un giocatore in certe posizioni, intenderemo, quindi, che quel giocatore può muovere da quelle posizioni in una posizione nella quale l'avversario non ha mosse possibili (risp. non ha mosse possibili in quelle posizioni).

1.8 Codifiche

Nel seguito esamineremo la completezza (hardness) di svariati giochi rispetto a meno svariate classi di complessità facendo riferimanto a riduzioni in **LOGSPACE**. Per questo ci troveremo spesso di fronte alla necessità di eseguire computazioni su oggetti come formule proposizionali o grafi; è, quindi, buona cosa esplicitare una volta per tutte la codifica che adottiamo per simili oggetti sotto forma di stringhe di simboli, ed indagare le operazioni che più elementarmente possono essere eseguite su queste in **LOGSPACE**.

Innanzitutto adottiamo come codifica per una lista di codifiche di oggetti omologhi (una lista di sole formule, una lista di soli numeri, etc...)

$$\langle \alpha_1 \rangle, \langle \alpha_2 \rangle, \ldots, \langle \alpha_n \rangle$$

la stringa

$$\langle (\alpha_1 \alpha_2 \ldots \alpha_n) \rangle$$

la quale cosa non pone problemi di ambiguità a patto di usare per i diversi oggetti *codifiche auto-delimitanti* (intendendo con tale locuzione che una codifica valida di un oggetto non deve mai essere un prefisso di una codifica valida di un altro oggetto della medesima specie) che non iniziano mai per il carattere). Ovviamente la verifica di questa proprietà, per altro ovvia, delle nostre codifiche è lasciata al lettore.

Fatto questo nulla è più facile che codificare:

- **i numeri interi positivi**, in notazione binaria per mezzo delle stringhe $(1\beta_1\beta_2 \ldots \beta_n)$ dove i β_i sono elementi di $\{0, 1\}$;
- **i nodi dei grafi**, che *sono* numeri interi positivi;
- **gli archi (orientati) dei grafi** (eventualmente orientati), che sono liste di due nodi, ossia stringhe nella forma $(\alpha\beta)$ dove α e β sono numeri interi;
- **i grafi**, che sono liste di archi, ossia stringhe nella forma

$$\langle ()^\frown \langle (\alpha_1\beta_1) \rangle^\frown \langle (\alpha_2\beta_2) \rangle^\frown \cdots^\frown \langle (\alpha_k\beta_k) \rangle^\frown \langle () \rangle$$

dove le sottostringhe $\langle(\alpha_i\beta_i)\rangle$ rappresentano i singoli archi (qui stiamo supponendo di escludere dal nostro interesse quei grafi che presentano nodi isolati);

- **le variabili proposizionali** con $\langle X\alpha\rangle$ dove α è un numero;
- **le formule proposizionali**, come di stringhe nella forma $\langle(\phi\xi\psi)\rangle$ o $\langle\neg(\phi)\rangle$, ove $\langle\phi\rangle$ e $\langle\psi\rangle$ sono formule o variabili e ξ è un simbolo appartenente all'insieme $\{\vee, \wedge\}$;
- **le formule proposizionali con quantificatori**, similmente alle formule, come di stringhe nella forma $\langle(\phi\xi\psi)\rangle$, $\langle\neg(\phi\xi\psi)\rangle$ o $\langle\chi\nu\phi\rangle$, ove $\langle\phi\rangle$ e $\langle\psi\rangle$ sono formule con quantificatori o variabili, ξ è un simbolo appartenente all'insieme $\{\vee, \wedge\}$, ν è una variabile, e χ è un simbolo appartenente all'insieme $\{\forall, \exists\}$.

Già sappiamo cosa si può fare sui numeri, e sui grafi, per ora, c'è poco da fare. Quanto alle formule seguono alcuni lemmi di qualche utilità; essi, com'è ovvio, valgono altresì nel caso di formule con quantificatori.

Lemma 1.8.1 (Costruzione di formule). *È possibile in* **LOGSPACE** *scrivere la disgiunzione $\langle\phi \vee \psi\rangle$ e la congiunzione $\langle\phi \wedge \psi\rangle$ di due formule $\langle\phi\rangle$ e $\langle\psi\rangle$. È, poi, possibile scrivere la negazione di una formula $\langle\phi\rangle$, definita come $\neg\langle\phi\rangle = \langle\neg\phi\rangle$ se $\langle\phi\rangle$ non è della forma $\langle\neg\psi\rangle$, altrimenti $\neg\langle\phi\rangle = \langle\psi\rangle$.*

Dimostrazione. Ovvia. □

Lemma 1.8.2 (Sostituzione delle variabili). *Data una funzione F calcolabile in* **LOGSPACE** *che mappa $\{0,1\}^*$ in formule ed una formula $\langle\phi\rangle$, si può scrivere in* **LOGSPACE** *la formula $\langle\psi\rangle$ che presenta per ogni occorrenza della variabile $\langle X(\sigma)\rangle$ la formula $F(\langle\sigma\rangle)$ (eliminando eventualmente le doppie negazioni).*

Dimostrazione. Infatti, come abbiamo dimostrato, per una macchina che lavora in **LOGSPACE** è lecito richiamare una procedura anch'essa in **LOGSPACE** (sebbene questa possa richiedere un input di dimensioni polinomiali) senza causare un salto di complessità. Allora possiamo facilmente eseguire l'intera sostituzione semplicemente leggendo e *copiando* pedestremente la formula finché non incontriamo la sottostringa $\langle X(\rangle$ che non copiamo; a questo punto passiamo alla macchina che calcola F qualunque cosa venga fino alla successiva parentesi chiusa, e ne copiamo l'output. Poi saltiamo la $\langle)\rangle$ che termina il simbolo di variabile che abbiamo appena sostituito e riprendiamo a copiare la formula... □

Ci servirà ancora qualche operazione, la cui complessità sarà però discussa quando verrà utile. Per ora anticipiamo che la maggior parte delle nostre riduzioni consisteranno in costruzioni *modulari* di oggetti come formule o di

grafi; in altre parole noi definiremo alcune sottoformule o sottografi (*gadget*) che poi collegheremo fra loro dipendentemente dall'input della riduzione. Generalmente i singoli gadget saranno distinti da uno o più numeri interi (indici di variabili proposizionali, numeri che rappresentano nodi di un grafo, etc.) e saranno costruibili in **LOGSPACE**. Il modo nel quale uniremo fra loro i diversi gadget farà, quindi, con somma frequenza, appello, implicitamente, al lemma 1.4.8 ed al precedente.

Capitolo 2

Alternanza & giochi

2.1 Macchine di Turing alternanti

Strumento fondamentale per indagare la complessità computazionale dei giochi matematici è la *macchina di Turing alternante (ATM)* che qui sotto definisco. Essenzialmente una ATM lavora in modo simile ad una macchina di Turing non deterministica *eseguendo* ad ogni transizione più copie di se stessa *in parallelo*; tuttavia, a differenza di una macchina non deterministica, una ATM ha *stati universali* e *stati esistenziali*. Com'è chiaro dalla nomenclatura, diremo che un nodo dell'albero di computazione di una ATM è accettante quando **o** la macchina vi si trova in uno stato esistenziale ed il nostro nodo ha almeno un *figlio* accettante, **oppure** la macchina vi si trova in uno stato universale ed il nodo ha una prole interamente accettante.

Formalmente diamo le seguenti definizioni.

Definizione 2.1.1 (ATM). *Una ATM è definita similmente ad una TM come una* 6*-tupla* $(Q, \Gamma, \Sigma, \delta, U, q_0)$*, ove:*

- $Q \ni q_0$ *è l'insieme degli stati,*
- $\Gamma = \Gamma' \sqcup \{\#\}$ *è l'alfabeto,*
- $\Sigma \subset \Gamma'$ *è l'alfabeto di input,*
- $\delta \subset Q \times \Gamma \times Q \times \Gamma' \times \{L, S, R\}$ *è la* relazione di transizione*, e*
- $U \subset Q$ *è l'insieme degli* stati universali.

Ovviamente, come nel caso di una TM, Q *e* Γ *devono essere insiemi finiti. Nel seguito useremo la locuzione* stato esistenziale *come equivalente di* stato non universale*.*

Definizione 2.1.2. *Le configurazioni di* $M = (Q, \Gamma, \Sigma, \delta, U, q_0)$ *sono terne* (N, p, q) *ove:*

- $N \in \Gamma^*$ *rappresenta la porzione del nastro già esaminata dalla macchina,*
- $p \in \mathbb{N}^+$ *indica la posizione della macchina su* N *(quando la macchina è sul primo carattere di* N*,* $p = 1$*; quando è sull'ultimo* $p = |N|$*),*
- $q \in Q$ *è lo stato nel quale si trova la macchina.*

Indichiamo con $\mathcal{C}_M$ *l'insieme delle configurazioni di* M*.*

Definizione 2.1.3. *Date due configurazioni* $C = (N, p, q)$ *e* $C' = (N', p', q')$ *si dice che* C' *è immediatamente successiva a* C *(*$C \vdash_M C'$*) quando esiste* $(\alpha, q, \alpha', q', m) \in \delta$ *tale che:*

$$N[p] = \alpha$$

$$p' - p = \begin{cases} 1 & \text{se } m = R \\ 0 & \text{se } m = S \\ -1 & \text{se } m = L \end{cases}$$

$$|N'| = \begin{cases} |N| + 1 & \text{se } p' = |N| + 1 \\ |N| & \text{altrimenti} \end{cases}$$

$$N'[n] = \begin{cases} N[n] & \text{se } n \neq p \text{ ed } n \leq |N| \\ \# & \text{se } n = |N'| = |N| + 1 \\ \alpha' & \text{se } n = p \end{cases}$$

Diciamo quindi che C' *è raggiungibile da* C *se vale* $C \vdash_M^* C'$*, ove* $\vdash_M^*$ *indica la chiusura riflessiva e transitiva di* $\vdash_M$*.*

Definizione 2.1.4. *Un* albero accettante *per* $s \in \Sigma^*$ *di* $M = (Q, \Gamma, \Sigma, \delta, U, q_0)$ *è un insieme* $T \subset \mathcal{C}_M \times \mathbb{N}$ *che verifica le seguenti proprietà:*

- *Esiste* $k \in \mathbb{N}$ *tale che* $((s, 1, q_0), k) \in T$*;*
- *se* $(c, n) \in T$ *e* c *è in uno stato universale allora per ogni* c' *tale che* $c \vdash_M c'$*,* $(c', n - 1) \in T$*;*
- *se* $(c, n) \in T$ *e* c *è in uno stato esistenziale allora esiste un* c' *tale che* $c \vdash_M c'$ *e* $(c', n - 1) \in T$*.*

Quando esiste un albero accettante per s *diciamo che la macchina* M *accetta* s*.*

Si noti che non è necessario presupporre l'esistenza di uno *stato accettante* in quanto questo è semplicemente uno stato universale che non ha mai *stati successivi*; simmetricamente non ci serve alcuno *stato rifiutante* preconfezionato.

Come per le macchine di Turing ordinarie, si danno classi di complessità spaziale e temporale anche per le macchine di Turing alternanti. Diciamo quindi che:

Definizione 2.1.5. *Sia $s : \mathbb{N} \mapsto \mathbb{N}$; un linguaggio $L \in \Sigma^*$ è in* **ASPACE**(s) *se e solo se esiste una ATM M che accetta in input l'alfabeto Σ e:*

- *per ogni $\sigma \in \Sigma^*$ M accetta σ se e solo se $\sigma \in L$;*
- *per ogni $\sigma \in L$ esiste un albero T accettante per σ tale che per ogni $\big((N, p, q), s\big) \in T$ la lunghezza di N sia minore od uguale a $s(|\sigma|)$.*

Definizione 2.1.6. *Sia $s : \mathbb{N} \mapsto \mathbb{N}$; un linguaggio $L \in \Sigma^*$ è in* **ATIME**(s) *se e solo se esiste una ATM M che accetta in input l'alfabeto Σ e:*

- *per ogni $\sigma \in \Sigma^*$ M accetta σ se e solo se $\sigma \in L$;*
- *per ogni $\sigma \in L$ esiste un albero T accettante per σ tale che $\big((s, 1, q_0), s(|\sigma|)\big) \in T$.*

Insomma chiediamo, affinché M lavori in spazio (tempo) s, che l'appartenenza di ogni input σ ad L possa essere decisa veridicamente anche limitando il nostro *campo visivo* soltanto a quelle configurazioni che occupano uno spazio (che possono essere raggiunte in un tempo) minore od uguale ad $s(|\sigma|)$.

Dalle definizioni precedenti emerge che ci stiamo limitando a considerare ATM dotate di un unico nastro (che è contemporaneamente *nastro di input* e *di lavoro*). Sebbene questo possa sembrare una limitazione, in realtà, contrariamente a quanto accade nel caso delle macchine di Turing ordinarie, non lo è: per ogni ATM dotata di un qualsivoglia numero di nastri è sempre possibile trovare una ATM equivalente con un unico nastro ugualmente veloce o ugualmente economa.

Chiaramente valgono per le ATM versioni perfettamente simmetriche dei classici *speed-up theorem* e *hierarchy theorem*.

A questo punto, mantenendo il parallelo col caso ordinario, poniamo:

$$\mathbf{APTIME} = \bigcup_{c,k \in \mathbb{N}} \mathbf{ATIME}(c\bullet^k)$$

$$\mathbf{APSPACE} = \bigcup_{c,k \in \mathbb{N}} \mathbf{ASPACE}(c\bullet^k)$$

ed enunciamo i seguenti

Teorema 2.1.7.

$$\mathbf{APTIME} = \mathbf{PSPACE}$$

Teorema 2.1.8.

$$\mathbf{APSPACE} = \mathbf{EXPTIME}$$

I quali rappresentano ovvie conseguenze dei lemmi che ora dimostreremo.

Lemma 2.1.9. *Se s è una funzione tale che $s(n) \geq n$ per ogni $n \in \mathbb{N}$, allora*

$$\mathbf{SPACE}(s) \subset \mathbf{ATIME}(s^2)$$

Dimostrazione. Supponiamo data una macchina di Turing M operante in spazio s; descriveremo una ATM M' equivalente che opera in tempo s^2. Osserviamo innanzitutto che la nostra macchina di Turing deterministica può raggiungere, su un input di lunghezza l, non più di $c(k+1)^{s(l)}$ diverse configurazioni (ove k e c rappresentano la cardinalità dell'alfabeto e dell'insieme degli stati di M); quindi M accetta se e solo se raggiunge uno stato accettante entro $c(k+1)^{s(l)}$ passi.

Ora l'anatomia di M' sarà a grandi linee la seguente (le parole *a grandi linee* non dovrebbero mai comparire in una dimostrazione salvo quando questo è inevitabile...). M' lavora essenzialmente sui primi $s(l)$ simboli di un unico nastro diviso in tre tracce. Inizialmente M' predispone sulla prima traccia la configurazione iniziale di M, sulla seconda traccia *indovina esistenzialmente* la (eventuale) configurazione finale, e sulla terza *indovina* un numero maggiore od uguale a $c(k+1)^{s(l)}$, scritto in una base decisa a priori purché abbastanza alta per farcelo stare (ad esempio basta $c(k+1)^{s(l)}$ in base $c(k+1)$). Quindi esegue ricorsivamente la procedura seguente, il cui scopo è di verificare se la configurazione sulla prima traccia porta alla configurazione sulla seconda nel numero di passi indicato sulla terza (questo basta se supponiamo che le configurazioni terminali siano stazionarie). Tale obbiettivo è raggiunto ripetutamente indovinando una configurazione intermedia e richiamando *due copie* della procedura sulle *due coppie* di configurazioni in cui questa divide la coppia di partenza.

```
procedura "raggiungibile"

siano C1 la configurazione sulla prima traccia
      C2 la configurazione sulla seconda traccia
      N  il numero sulla terza traccia

se N = 0
    se C1 = C2 o C1 |- C2
        accetta
    altrimenti
        rifiuta
altrimenti
    poni M=[N/2]
    indovina esistenzialmente una configurazione C3
    verifica universalmente le seguenti alternative
        1 - scrivi C1 sulla prima traccia
            scrivi C3 sulla seconda traccia
            scrivi M sulla terza traccia
```

```
        esegui la procedura "raggiungibile"
    2 - scrivi C3 sulla prima traccia
        scrivi C2 sulla seconda traccia
        scrivi N-M sulla terza traccia
        esegui la procedura "raggiungibile"
```

Chiaramente ciascuna esecuzione di questa procedura richiederà un tempo lineare in $s(l)$, e ogni ramo dell'albero di computazione di M' avrà tempo di annoverare soltanto una quantità lineare di chiamate alla procedura prima che la spada di Damocle `N` si annulli forzando l'uscita (cosa che avviene prima di $1 + \log(\mathtt{N}) = O\big(s(l)\big)$ dimezzamenti); quindi la condizione per M' di lavorare in tempo $O\big(s^2(l)\big)$ è rispettata. □

Lemma 2.1.10. *Se t è una funzione* space constructible*, allora*

$$\mathbf{ATIME}(t) \subset \mathbf{SPACE}(t^2)$$

Dimostrazione. Supponiamo data la ATM M' che opera in tempo t; quindi costruiamo una macchina di Turing M equivalente che lavora in spazio t^2. Il metodo è banale, basta osservare, in maniera del tutto simmetrica a quanto fatto prima, che su input di lunghezza l M' non ha tempo di raggiungere configurazioni più grandi di $t(l)$ (quando scritte nell'alfabeto opportuno), e che, per di più, l'albero della computazione di M' non ha tempo di estendersi oltre la profondità $t(l)$. Ne segue che per esaminare semplicemente l'intero albero basta costruire uno *stack* di profondità $t(l)$ composto di elementi ciascuno di $t(l)$ simboli (ed è soltanto in questo che necessita la *space constructibility* di t).

Un po' più precisamente questo è quanto la macchina M deve fare, e può fare in spazio $O(t^2(l))$

```
calcola T=t(l)
predisponi sul nastro T blocchi di T simboli
scrivi sul primo blocco la configurazione iniziale di M'
se la procedura "simula" accetta
    accetta
altrimenti
    rifiuta

procedura "simula"

sia C la configurazione sul nastro
avanza di un blocco
se C è una configurazione universale
    al variare di C' fra le configurazioni successive di C
        scrivi C' sul nastro
```

```
        se la procedura "simula" rifiuta
            retrocedi di un blocco
            rifiuta
    retrocedi di un blocco
    accetta
altrimenti
    al variare di C' fra le configurazioni successive di C
        scrivi C' sul nastro
        se la procedura "simula" accetta
            retrocedi di un blocco
            accetta
    retrocedi di un blocco
    rifiuta
```

□

Lemma 2.1.11. *Se t è una funzione tale che $t(n) \geq 2^n$ per ogni $n \in \mathbb{N}$, allora*

$$\mathbf{TIME}(t) \subset \mathbf{ASPACE}(\log \circ t)$$

Osservazione 2.1.12. *La condizione $t(n) \geq 2^n$ potrebbe essere notevolmente addolcita, se la nostra definizione di ATM fosse maggiormente tollerante nei confronti di quelle macchine che sfruttano una quantità sublineare di spazio. Per ottenere un simile risultato, avremmo potuto adottare una definizione simile a quella data per le* **LOGSPACE***-TM; tuttavia tali complicazioni sono inutili ai nostri scopi.*

Dimostrazione. Come al solito prendiamo in considerazione una macchina di Turing M che lavora in tempo t e costruiamo l'*equivalente* ATM M'. Per semplicità d'esposizione possiamo considerare M dotata di un unico nastro. È facile indagare il caso generale come ovvia generalizzazione dell'argomento, oppure ricordando che si può sempre ridurre il numero dei nastri usati da una macchina di Turing ad uno, a patto di ammettere una perdita al più quadratica nella *velocità* di questa.

Fissiamo, al solito, una stringa di input di lunghezza l così da poter osservare che M non uscirà mai dalle prime $t(l)$ caselle del suo unico nastro; quindi procediamo a codificare le configurazioni di M tramite stringhe nell'alfabeto $L = (\Gamma \sqcup \{\#\}) \times (Q \sqcup \{\#\})$ (essendo Q e Γ l'insieme degli stati e l'alfabeto di M). Precisamente associamo alla configurazione $(\gamma_1 \dots \gamma_n, i, q)$ la stringa $(\gamma'_1, q'_1) \dots (\gamma'_{t(l)}, q'_t(l))$ ove

$$\begin{array}{ll} \gamma'_j = \gamma_j & \text{per } j \leq i; \\ \gamma'_j = \# & \text{per } j > i; \\ q'_i = q; & \\ q'_j = \# & \text{per } j \neq i. \end{array}$$

È chiaro che se $\sigma_1 \dots \sigma_{t(l)} \vdash_M \tau_1 \dots \tau_{t(l)}$, allora ciascun simbolo τ_j dipende unicamente dai tre simboli *precedenti* $\sigma_{j-1}\sigma_j\sigma_{j+1}$, secondo una relazione che chiameremo $R(\tau_j, \sigma_{j-1}, \sigma_j, \sigma_{j+1})$.

L'espediente con cui possiamo permettere ad M' di esaminare queste dilatate configurazioni, è fargliele *indovinare* e *verificare* un simbolo alla volta, tenendo in memoria soltanto l'indirizzo del simbolo sotto esame. In pratica M' è

```
indovina i numeri interi i e j
indovina un simbolo accettante q di L
scrivi i$j$q sul nastro
richiama la procedura "verifica"
```

Ora M' ha indovianto la *configurazione finale* della computazione di M, o meglio tutto quello che a noi può importare di questa, cioè l'ultimo simbolo letto da M e l'ultimo stato (accettante) di M; poi ha indovinati i due numeri `i` e `j` che rappresentano rispettivamente la posizione di M (a partire da 1, all'estremo sinistro del nastro) e *l'istante* nel quale la computazione è terminata (a partire da 0, prima della prima transizione). Quindi M' ha richiamato la procedura "verifica" che riporto qui sotto, il cui scopo è di assicurare la correttezza degli elementi appena indovinati; scopo che viene raggiunto indovinando una terna di possibili *padri* per `q` (che si trovano in posizioni adiacenti e nell'istante immediatamente precedente) e richiamando su ciascuno di essi una nuova istanza della medesima procedura.

```
procedura "verifica"

leggi i,j e q dal nastro
se i = 0
    se q = #
        accetta
    altrimenti
        rifiuta
se j = 0
    se l'i-esimo simbolo dell'input è q
        accetta
    altrimenti
        rifiuta
indovina r,s e t tali che R(q,r,s,t)
verifica universalmente le seguenti alternative
    1 - scrivi (i-1)$(j-1)$r sul nastro
        richiama la procedura "verifica"
    2 - scrivi i$(j-1)$s sul nastro
        richiama la procedura "verifica"
    3 - scrivi (i+1)$(j-1)$t sul nastro
```

```
richiama la procedura "verifica"
```

L'idea soggiacente è che i numeri `i` e `j` possono essere indovinati *senza perdere la generalità* di dimensione $O\big(\log(t(l))\big)$. La procedura `verifica` controlla che un simbolo di una configurazione sia valido riducendo il problema all'identico controllo operato sui suoi tre *simboli padri*, che può facilmente indovinare in quanto la relazione R è finita e, come tale, può essere inclusa nel *controllo finito* della macchina M'. Quanto al test `i = 0`, serve soltanto per evitare che la macchina simulata riesca a superare furtivamente il bordo sinistro del nastro. □

Lemma 2.1.13. *Se s è una funzione* space constructible*, allora*

$$\mathbf{ASPACE}(s) \subset \bigcup_{c\in\mathbb{N}} \mathbf{DTIME}(c^s)$$

Dimostrazione. Ovviamente dobbiamo prendere la solita ATM M' e cercare di costruire una TM M in grado di simularla. La dimensione dell'albero di computazione di una ATM limitata a lavorare in spazio s, tuttavia, può raggiungere a priori c^{c^s} che è troppo per essere esaminato in tempo c^s. Il modo corretto di costruire M è osservare che quest'albero gigantesco sarebbe in realtà molto ridondante, in quanto M' ammette soltanto c^s (per un opportuno valore di c) configurazioni diverse. Pertanto M deve dapprima scrivere le c^s configurazioni di M'; cosa che può fare partendo dalla configurazione iniziale, quindi scrivendo ciascuna delle sue *figlie*, poi le *figlie delle figlie*, e così via, ogni volta controllando di non scrivere la medesima configurazione due volte. Terminata questa operazione M deve esaminare ricorsivamente l'albero delle configurazioni a partire da quella iniziale marcando ciascun nodo con 0 o 1 secondoché sia rifiutante od accettante, la quale cosa si può portare a termine *rapidamente* poiché ogni nodo necessita di essere esaminato una sola volta, così eliminando la ridondanza.

Compendiosamente la scrittura di ogni configurazione impiega tempo $O(c^s)$ (per controllare che non sia uguale ad una configurazione già scritta), quindi la scrittura di tutte le configurazioni richiede $O(c^s c^s) = O(c^s)$ (a patto di sostituire furbescamente c con c^2 durante l'=). L'esame ricorsivo di ogni configurazione richiede, a sua volta, $O(c^s)$ passi per trovare fra quelle scritte le sue configurazioni figlie, più il tempo dedicato all'esame di queste ultime che però ha luogo una volta soltanto; quindi la valutazione della configurazione iniziale non può richiedere più del solito $O(c^s c^s) = O(c^s)$. □

2.2 Macchine di Turing Alternanti & Giochi

In questa sezione intendo descrivere formalmente la connessione fra le ATM ed i giochi matematici.

Innanzitutto una *ATM è già un gioco.* Infatti possiamo immaginare due giocatori (Esiste e Perogni) che simulano il comportamneto di una ATM nel modo seguente: la loro scacchiera è una configurazione della ATM (definita dal nastro più lo stato e la posizione) e la *disposizione iniziale* delle loro pedine è determinata dall'input della macchina (dal momento che le macchine più potenti che ci possano interessare lavorano in spazio lineare, e catturano precisamente la complessità di **EXPTIME**, non dobbiamo farci spaventare troppo dal fatto che il nastro-scacchiera è potenzialmente infinito). I giocatori simulano una transizione della ATM per ogni semimossa: quando la ATM (simulata) si trova in uno stato esistenziale l'iniziativa è di Esiste, che può scegliere lo stato successivo della macchina fra quelli *permessi* dalla relazione di transizione; parimenti quando lo stato è universale tocca a Perogni eseguire la medesima mossa (in questo modo le mosse di Esiste e Perogni potrebbero non *alternarsi* in senso proprio, tuttavia non è difficile interporre delle *mosse di attesa* per un giocatore ogniqualvolta si presenti una coppia di mosse successive dell'altro). Come di consueto perde il primo giocatore che non può eseguire la propria mossa. Se la macchina accetta l'input allora un suo albero accettante fornisce chiaramente ad Esiste una strategia vincente, e simmetricamente una strategia per Esiste può essere vista come un albero accettante della macchina.

In questo modo, data una ATM, abbiamo ottenuto un gioco tale che l'insieme delle posizioni vincenti per un (volendo: *il primo*) giocatore *rappresenta* il linguaggio accettato dalla macchina. D'ora in poi studieremo, più formalmente, la complessità computazionale dell'insieme $\mathcal{W}^I(G)$ delle posizioni vincenti per il giocatore I, per diversi giochi G, essendo questo un sottolinguaggio dell'insieme di tutte le posizioni *una volta codificate in modo opportuno.* Diremo quindi che un certo gioco G è completo per una certa classe di complessità (tipicamente **EXPTIME**) quando $\mathcal{W}^I(G)$ è completo per quella classe.

Quanto alla codifica delle posizioni, in questa sezione non avremo bisogno di trattare elementi diversi da formule proposizionali e valutazioni di variabili proposizionali. Ci si convince facilmente del fatto che la codifica delineata nell'osservazione 2.2.8 assolve pienamente ai nostri scopi.

Ora, però, ci servono giochi più *ragionevoli* cui associare il fuzionamento di una ATM, che possano essere usati come passo intermedio nella catena di riduzioni fra i giochi *veri* ed il gioco descritto sopra. Questi sono i seguenti *giochi proposizionali.* Ciascuno di essi si gioca sul valore di verità delle variabili di una o più formule proposizionali: essenzialmente i due giocatori si alternano nel modificare il valore di verità di queste finché non si verifichi un'opportuna condizione sul valore di verità delle formule. I nomi G_1, G_2 e G_5 sono mantenuti per coerenza con quelli assegnati in [18].

Definizione 2.2.1 (G_1). *Le posizioni di G_1 sono valutazioni delle variabili di una formula proposizionale F_1. Tali variabili sono ripartite nei tre*

sottoinsiemi V_1^I, V_1^{II} e $\{t\}$. Ad ogni turno il primo giocatore, I, muove decidendo a piacer suo il valore delle variabili in V^I e ponendo a 1 la valutazione di t; II decide la valutazione delle variabili di V^{II} e pone t a 0. Ciascun giocatore perde se F_1 è falsa dopo la sua mossa.

Definizione 2.2.2 (G_2). *Le posizioni di G_2 sono valutazioni delle variabili di due formule: W_2^I e W_2^{II} (che possono avere variabili* in comune*). Queste sono ripartite nei due insiemi disgiunti V_2^I e V_2^{II}. Il primo (risp. secondo) giocatore muove cambiando il valore assegnato ad al più una variabile di V_2^I (V_2^{II}). Vince il primo (risp. secondo) giocatore se la formula W_2^I (risp. W_2^{II}) è* vera *al termine della sua mossa.*

Definizione 2.2.3 (G_5). *Le posizioni di G_5 sono valutazioni delle variabili di una formula proposizionale F_5. Tali variabili sono ripartite nei due sottoinsiemi V_5^I e V_5^{II}. Il primo giocatore, I, muove cambiando il valore di verità di al più una variabile di V_5^I; mentre il secondo cambiando il valore di verità di al più una delle variabili di V_5^{II}. I vince quando la valutazione rende vera F_5 (che ciò avvenga dopo una mossa sua o dell'avversario, oppure anche all'inizio del gioco).*

Osservazione 2.2.4. *Le condizioni per la vittoria dell'uno o dell'altro giocatore che abbiamo dato nel corso delle definizioni precedenti non sono per nulla esaustive di tutte le possibili partite (nel caso di G_5, addirittura, non è definita alcuna condizione per la vittoria del secondo giocatore!). Però, benché vi siano partite infinitamente lunghe, il lettore si convincerà del fatto che l'insieme $W_{G_i}(I)$ è ben definito: semplicemente esistono posizioni dei nostri giochi G_i che non appartengono né a $W_{G_i}(I)$ né a $W_{G_i}(II)$. Infatti, se entrambi i giocatori seguono la* strategia ottimale*, ciascuna di queste posizioni porta ad un'altra della medesima specie, e mai ad una terminale, cosicché ha luogo una partita infinita, infinitamente ripetitiva.*

Se, tuttavia, lo scetticismo del lettore nel riguardo di giochi tanto noiosi (sembra proprio che certe partite non debbano mai finire!) dovesse permanere, egli osservi che si può facilmente introdurre una sorta di regola delle cinquanta mosse *dichiarando vincitore del gioco II dopo che sono state giocate $2^n + 1$ mosse, dove n è il numero delle variabili che intervengono nella istanza di G_i sotto esame. Questo lascia inalterato l'insieme $W_{G_i}(I)$, in quanto, se prima che il termine $2^n + 1$ scada I non ha ancora vinto, allora una certa posizione deve esserglisi presentata almeno due volte: ciò significa, ammesso che entrambi i giocatori seguano la migliore strategia possibile, che la successione di mosse compresa fra la prima e la seconda occorrenza di quella posizione si ripeterebbe periodicamente all'infinito nella* versione infinitaria *del gioco. Altresì l'aggiunta di questa regola non richiede che un aumento lineare della dimensione delle posizioni, per includere in queste il* contatore *delle $2^n + 1$, quindi non altera la validità dei discorsi che seguono. Tuttavia, si osservi, $W_{G_i}(I)$ e $W_{G_i}(II)$ sono ben definiti* a prescindere *da questo argomento!*

Di ciascuno di questi proveremo la completezza per **EXPTIME**; all'uopo utilizzando l'identità **EXPTIME** = **APSPACE** e sfruttando il fatto che ogni linguaggio in **APSPACE** si riduce banalmente in **LOGSPACE** ad uno in **ALINEARSPACE** (basta aggiungere al fondo dell'input una quantità polinomiale di *spazi bianchi*, la qual cosa richiede soltanto di determinare la lunghezza dell'input e calcolarne una funzione polinomiale...). Per questo è utile premettere qualche osservazione sul modo di codificare le configurazioni di una ATM per mezzo di variabili proposizionali.

Innanzitutto fissiamo una ATM $M = (Q, \Gamma, \Sigma, \delta, U, q_0)$ e codifiche dell'alfabeto Γ e dell'insieme degli stati Q di M in stringhe binarie (nell'alfabeto $\{0,1\}$). Tutto quello che ci interessa è che le codifiche dei simboli di Γ abbiano tutte la medesima lunghezza, che lo stesso valga per le codifiche degli stati e che per di più nulla sia codificato da una stringa di soli zeri.

Siano quindi

$$\alpha : \Gamma \mapsto \{0,1\}^*$$

e

$$\kappa : Q \sqcup \{\#\} \mapsto \{0,1\}^*$$

le nostre codifiche, dove $\kappa(\#)$ è la stringa di zeri che ci siamo risparmiati. Per mezzo di queste costruiamo la codifica delle configurazioni di M:

$$\digamma : \mathcal{C}_M \mapsto \{0,1\}^*$$

$$\digamma\big((\langle c_1 c_2 \dots c_n\rangle, p, q)\big) = \langle \alpha(c_1)\kappa(q_1)\alpha(c_2)\kappa(q_2)\dots\alpha(c_n)\kappa(q_n)\rangle$$

ove i q_i sono $\#$ per $i \neq p$ e $q_p = q$. Questa codifica è chiaramente iniettiva e, una volta fissata M, *lineare*, ossia la lunghezza $\big|\digamma\big((N,p,q)\big)\big|$ della codifica è direttamente proporzionale alla lunghezza $|N|$ della porzione di nastro codificata.

Consideriamo, ora, due configurazioni C e C' tali che $C \vdash_M C'$ e le loro codifiche

$$\digamma(C) = \langle \alpha(c_0)\kappa(q_0)\rangle^\frown\langle \alpha(c_1)\kappa(q_1)\rangle^\frown\langle \alpha(c_2)\kappa(q_2)\rangle\dots$$

e

$$\digamma(C') = \langle \alpha(c'_0)\kappa(q'_0)\rangle^\frown\langle \alpha(c'_1)\kappa(q'_1)\rangle^\frown\langle \alpha(c'_2)\kappa(q'_2)\rangle\dots$$

Si vede subito che ciascuna delle sottostringhe nelle quali ho diviso C' dipende soltanto dalle tre sottostringhe di C *soprastanti*, nel senso che esiste un insieme $S \in \{0,1\}^*$ tale che $C \vdash_M C'$ se e solo se per ogni valore di i si ha:

$$\langle \alpha(c_{i-1})\kappa(q_{i-1})\rangle^\frown\langle \alpha(c_i)\kappa(q_i)\rangle^\frown\langle \alpha(c_{i+1})\kappa(q_{i+1})\rangle^\frown\langle \alpha(c_i)\kappa(q_i)\rangle \in S$$

(nella quale formula si è commesso per semplicità un abuso di notazione: il lettore supponga che $\alpha(c_{-1})$ e $\kappa(q_{-1})$ siano stringhe di zeri *della lunghezza*

giusta, idem per l'altro capo della stringa.) Sia l la lunghezza delle stringhe di S; siccome, una volta fissata M, l è fissata, allora esiste una formula proposizionale $\phi_M(x_1, x_2, \ldots, x_l)$ tale che:

$$\langle b_1 b_2 \ldots b_l \rangle \in S \leftrightarrow \phi_M(b_1, b_2, \ldots, b_l)$$

Se la nostra M esegue i suoi calcoli *in place* (ossia senza uscire dalla porzione di nastro occupata dall'input) allora le stringhe che codificano ogni configurazione del suo albero di computazione su un determinato input hanno la medesima lunghezza, L, quindi possono essere rappresentate tutte come valori di verità di L variabili proposizionali. In questa codifica è palese che la relazione $\vdash_M$ è rappresentata da una formula, $\psi_M(x_1, x_2, \ldots, x_L, y_1, y_2, \ldots, y_L)$, la quale, però, può essere costruita in spazio logaritmico in L seguendo una tecnica standard, in quanto non è altro che la congiunzione di un numero sufficiente di copie di ϕ_M nelle quali si siano sostituite le variabili opportune.

Da ultimo ipotizziamo che q_0 sia uno stato esistenziale e che ogniqualvolta $Q \vdash_M Q'$ e Q è una configurazione esistenziale (risp. universale), Q' sia universale (risp. esistenziale); la qual cosa può essere fatta senza perdita di generalità aumentando il numero degli stati della macchina M.

Ora siamo pronti per dimostrare che

Teorema 2.2.5. *G_1 è* **EXPTIME***-completo.*

Dimostrazione. Mantenendo le medesime notazioni usate nel discorso precedente ci basta porre:

$$\begin{aligned} F_1 =& \big(t \vee \psi_M(x_1, x_2, \ldots, x_L, y_1, y_2, \ldots, y_L)\big) \wedge \\ & \big(\neg t \vee \psi_M(y_1, y_2, \ldots, y_L, x_1, x_2, \ldots, x_L)\big) \\ V_1^I =& \{x_1, x_2, \ldots, x_L\} \\ V_1^{II} =& \{y_1, y_2, \ldots, y_L\} \end{aligned}$$

Chiediamo che inizialmente valga $t = 1$ e le variabili y_i codifichino l'input della macchina M.

Non è difficile capire che con questo si ottiene lo scopo voluto; infatti la formula F forza alla prima mossa il giocatore I a scegliere un assegnamento dei valori di verità per $x_1, x_2, \ldots, x_L$ che verifichi $\psi_M(y_1, y_2, \ldots, y_L, x_1, x_2, \ldots, x_L)$, il quale quindi deve rappresentare una configurazione *legale* della macchina successiva alla configurazione iniziale. Fatto questo II agisce simmetricamente cifrando nelle variabili $y_1, y_2, \ldots, y_L$ uno stato successivo a quello giocato da I, e così via. Per l'ipotesi fatta su M, I si trova a giocare sempre su stati esistenziali e II su stati universali; quindi, come nel caso del gioco accennato all'inizio di questa sezione, una strategia per I è precisamente un albero accettante per M e la posizione iniziale è una vittora per I se e solo se M accetta l'input. □

Il precedente teorema non ci dice nulla di nuovo, dal momento che riguarda semplicemente una versione un po' più *formale* del gioco di già esposto in apertura. Nel seguente farà la sua comparsa invece un utile elemento: la possibilità di limitare il campo d'azione dei giocatori ad una sola variabile per mossa.

Teorema 2.2.6. *G_2 è* **EXPTIME***-completo.*

Dimostrazione. In questa dimostrazione descriveremo dapprima il *campo di gioco* (ossia i due insiemi di variabili con la loro interpretazione nell'ottica della simulazione di una ATM), quindi daremo la definizione di un modo *corretto* di giocare; a questo punto costruiremo le due formule W_2^I e W_2^{II} in modo tale da raggiungere un duplice scopo: assicurarci che le mosse dei due giocatori effettivamente simulino la ATM (un po' come si è fatto per G_1) e punire quel giocatore che per primo deviasse dal *gioco corretto*.

I nostri insiemi V_I e V_{II} sono costituiti da $4L+4$ variabili ciascuno:

$$\begin{aligned} V_2^I &= \{x_1, \tilde{x}_1, x_2, \tilde{x}_2, \ldots, x_{2L+2}, \tilde{x}_{2L+2}\} \\ V_2^{II} &= \{y_1, \tilde{y}_1, y_2, \tilde{y}_2, \ldots, y_{2L+2}, \tilde{y}_{2L+2}\} \end{aligned}$$

(d'ora in poi il valore dei pedici sarà sempre considerato modulo $2L+2$) ove $\{x_1, x_2, \ldots, x_L\}$ e $\{y_{L+2}, y_{L+3}, \ldots, y_{2L+1}\}$ giocano i ruoli di rispettivamente V_1^I e V_1^{II} nel gioco G_1, mentre le altre variabili servono per permettere ai giocatori di *perdere dei tempi* senza rompere il gioco corretto. x_{L+1}, $\tilde{x}_{L+1}$, y_{2L+2} e $\tilde{y}_{2L+2}$, che non sembrano avere un posto nel nostro piano, ci serviranno come *indicatori di fine-corsa* per capire quando la simulazione di una mossa di G_1 è terminata e quindi ne possiamo verificare la correttezza tramite la formula ψ_M.

L'idea è di costringere il primo giocatore a muovere in una delle variabili x_1 e $\tilde{x}_1$, quindi il secondo in una fra y_1 e $\tilde{y}_1$, poi nuovamente il primo in x_2 o $\tilde{x}_2$, il secondo in y_2 o $\tilde{y}_2$, e così via fino ad arrivare a $2L+2$; dopo di che il ciclo si ripete. Ad ogni iterazione viene simulata una mossa completa del gioco G_1, o se vogliamo due transizioni di M. Onde forzare il *gioco corretto* così definito saranno comode le seguenti sottoformule:

$$\begin{aligned} x_i^{\oplus} &= x_i \oplus \tilde{x}_i \\ y_i^{\oplus} &= y_i \oplus \tilde{y}_i \\ \hat{x}_i &= \begin{cases} x_{i-1}^{\oplus} \oplus x_i^{\oplus} & \text{per } i \neq 1 \\ \neg(x_{2L+2}^{\oplus} \oplus x_1^{\oplus}) & \text{per } i = 1 \end{cases} \\ \hat{y}_i &= \begin{cases} y_{i-1}^{\oplus} \oplus y_i^{\oplus} & \text{per } i \neq 1 \\ \neg(y_{2L+2}^{\oplus} \oplus y_1^{\oplus}) & \text{per} i = 1 \end{cases} \end{aligned}$$

Innanzitutto noi richiederemo che le stringhe

$$s_x^{\oplus} = \langle x_1^{\oplus} x_2^{\oplus} \ldots x_{2L+2}^{\oplus} \rangle$$

e

$$s_y^{\oplus} = \langle y_1^{\oplus} y_2^{\oplus} \dots y_{2L+2}^{\oplus} \rangle$$

(inizialmente poste a $00\dots0$) stiano in $0^*1^* \cup 1^*0^*$; la quale cosa si vede subito che equivale a chiedere che le stringhe

$$\hat{s}_x = \langle \hat{x}_1 \hat{x}_2 \dots \hat{x}_{2L+2} \rangle$$

e

$$\hat{s}_y = \langle \hat{y}_1 \hat{y}_2 \dots \hat{y}_{2L+2} \rangle$$

(le quali valgono inzialmente $100\dots0$) siano elementi di 0^*10^*. Questo basta per forzare i giocatori ad eseguire le loro mosse *in ordine*, in quanto così è possibile giocare solo sulla *zona di confine* fra gli 1 e gli 0 di $s_x^{\oplus}$ e $s_y^{\oplus}$. Onde forzarli anche a giocare *all'unisono* (ossia onde evitare che un giocatore si attardi a giocare più volte sulla stessa variabile, *passi* una mossa, o cominci a *procedere a ritroso*) richiederemo che i due 1 di $\hat{s}_x$ ed $\hat{s}_y$ procedano *di pari passo*.

A dar forza alle nostre richieste saranno le formule seguenti:

$$\alpha_I = \bigvee_i (\hat{x}_i \wedge \hat{x}_{i+1}) \vee (\hat{x}_i \wedge \hat{y}_{i+1} \wedge \neg \hat{y}_{i-1})$$
$$\alpha_{II} = \bigvee_i (\hat{y}_i \wedge \hat{y}_{i+1}) \vee (\hat{y}_i \wedge \hat{x}_{i+2} \wedge \neg \hat{x}_i)$$

il cui valore durante una partita di gioco corretto dovrebbe rimanere sempre 0. Dimostriamo ora che se un giocatore devia per primo dal gioco corretto allora o la *sua* formula assume il valore 1, quella *dell'avversario* rimanendo a 0, oppure l'avversario ha la possibilità di portare la formula *del primo* ad 1 alla mossa successiva (chiaramente le formule W_2^I e W_2^{II} diranno che un giocatore vince qualora la formula α dell'avversario sia vera).

Supponiamo (tanto l'argomentazione è simmetrica) che sia il turno di I. La tipica posizione di gioco corretto in questa situazione è la seguente:

	$\dots$	$j-3$	$j-2$	$j-1$	j	$j+1$	$j+2$	$j+3$	$\dots$
$s_x^{\oplus}$	$\dots$	1	1	1	0	0	0	0	$\dots$
$s_y^{\oplus}$	$\dots$	1	1	1	0	0	0	0	$\dots$
$\hat{s}_x$	$\dots$	0	0	0	1	0	0	0	$\dots$
$\hat{s}_y$	$\dots$	0	0	0	1	0	0	0	$\dots$

Osserviamo che I non può giocare in una posizione diversa da j o $j-1$ senza con questo rendere vera la sottoformula $\hat{x}_i \wedge \hat{x}_{i+1}$ di uno dei disgiunti di α_I. Secondo il gioco corretto egli a questo punto dovrebbe giocare in i, in questo modo ci si verrebbe a trovare nella seguente posizione:

	$\ldots$	$j-3$	$j-2$	$j-1$	j	$j+1$	$j+2$	$j+3$	$\ldots$
$s_x^{\oplus}$	$\ldots$	1	1	1	1	0	0	0	$\ldots$
$s_y^{\oplus}$	$\ldots$	1	1	1	0	0	0	0	$\ldots$
$\hat{s}_x$	$\ldots$	0	0	0	0	1	0	0	$\ldots$
$\hat{s}_y$	$\ldots$	0	0	0	1	0	0	0	$\ldots$

e si vede che entrambe le formule α_I e α_{II} rimangono false. Se supponiamo allora che I giochi in $i-1$, poi si troverà in questa posizione:

	$\ldots$	$j-3$	$j-2$	$j-1$	j	$j+1$	$j+2$	$j+3$	$\ldots$
$s_x^{\oplus}$	$\ldots$	1	1	0	0	0	0	0	$\ldots$
$s_y^{\oplus}$	$\ldots$	1	1	1	0	0	0	0	$\ldots$
$\hat{s}_x$	$\ldots$	0	0	1	0	0	0	0	$\ldots$
$\hat{s}_y$	$\ldots$	0	0	0	1	0	0	0	$\ldots$

con $\alpha_I = 1$ (poiché $\hat{x}_{i-1} \wedge \hat{y}_i \wedge \neg\hat{y}_{i-2}$ è vera) e $\alpha_{II} = 0$. L'unico altro modo in cui I potrebbe *barare* è non giocando affatto, ma in questo caso II giocherà in i ottenendo la situazione seguente (che è identica alla precedente, salvo per una traslazione):

	$\ldots$	$j-3$	$j-2$	$j-1$	j	$j+1$	$j+2$	$j+3$	$\ldots$
$s_x^{\oplus}$	$\ldots$	1	1	1	0	0	0	0	$\ldots$
$s_y^{\oplus}$	$\ldots$	1	1	1	1	0	0	0	$\ldots$
$\hat{s}_x$	$\ldots$	0	0	0	1	0	0	0	$\ldots$
$\hat{s}_y$	$\ldots$	0	0	0	0	1	0	0	$\ldots$

Adesso siamo pronti per descrivere le formule:

$$W_2^I = \alpha_{II} \vee \big(\hat{y}_{2L+2} \wedge \hat{x}_{2L+1} \wedge \neg\psi_M(x_1, x_2, \ldots, x_L, y_{L+2}, y_{L+3}, \ldots, y_{2L+1})\big)$$
$$W_2^{II} = \alpha_I \vee \big(\hat{x}_{L+1} \wedge \hat{y}_{L+1} \wedge \neg\psi_M(y_{L+2}, y_{L+3}, \ldots, y_{2L+1}, x_1, x_2, \ldots, x_L)\big)$$

Per quanto detto prima è chiaro che queste bastano a forzare il gioco corretto; in più al termine di ogni *mezza iterazione*, cioè di ogni simulazione di una semimossa di G_1, forzano il *controllo di correttezza* di quest'ultima attraverso la formula ψ_M. Assumiamo, infatti, il punto di vista del giocatore I (come al solito il simmetrico è simmetrico): terminata la fase nella quale egli sceglie il valore delle proprie variabili *importanti* $x_1, x_2, \ldots, x_L$ (eventualmente giocando sulle corrispondenti $\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_L$ quando non vuole cambiare il valore di queste), l'1 che percorre la stringa $\hat{s}_x$ raggiungerà la posizione $L+1$ (se I non agisse in modo tale che questo avvenga, II vincerebbe a causa del suo *gioco scorretto*) in questo modo attivando il controllo di correttezza. Ora la posizione tipica è la seguente:

	...	$L-2$	$L-1$	L	$L+1$	$L+2$	$L+3$	$L+4$	...
$s_x^{\oplus}$	...	1	1	1	0	0	0	0	...
$s_y^{\oplus}$	...	1	1	0	0	0	0	0	...
$\hat{s}_x$	...	0	0	0	1	0	0	0	...
$\hat{s}_y$	...	0	0	1	0	0	0	0	...

Affinché anche $\hat{y}_{L+1}$ sia 1 e il controllo abbia veramente luogo è necessario che II giochi in L (dove dovrebbe) o $L+1$, e questo garantisce che egli non possa modificare le *sue* variabili *importanti* (che sono dalla $L+2$ in poi) appena prima del momento decisivo; in questo modo l'input della formula ψ quando entrambe $\hat{x}_{L+1}$ ed $\hat{y}_{L+1}$ sono vere è assicurato essere quello richiesto dalla simulazione.

Durante tutti questi discorsi non abbiamo considerato cosa capita quando la successione dei turni arriva alla *frontiera* fra $2L+2$ e 1; il lettore si convincerà tuttavia che la validità di quanto detto rimane inalterata anche in quelle circostanze. La dimostrazione è così conclusa. □

Teorema 2.2.7. *G_5 è* **EXPTIME***-completo.*

Dimostrazione. Supponiamo di aver costruito un'istanza di G_2 come nel teorema precedente (e non un'istanza *qualunque* di G_2: a noi interessa che un giocatore non possa *passare* un turno senza per questo perdere alla mossa successiva). Iniziamo col descrivere gli insiemi

$$V_5^I = V_2^I \sqcup \{x\}$$
$$V_5^{II} = V_2^{II} \sqcup V_2'^{II} \sqcup \{y, y'\}$$

ove $V_2'^{II}$ è un insieme in corrispondenza biunivoca con V_2^{II}.

Ora costruiamo le formule W_5^I e W_5^{II}. Queste sono semplicemente W_2^I e W_2^{II} nelle quali si sia sostituita ogni istanza di ogni variabile $z \in V_2^{II}$ con la disgiunzione esclusiva $z \oplus z'$ di lei e la sua corrispondente in $V_2'^{II}$. Infine sia la formula

$$T = \bigoplus \Big(V_2'^{II} \cup \{y'\} \Big)$$

L'idea è che in questo modo I può giocare sulle variabili di V_2^I oppure su x, mentre II può giocare sulle variabili di V_2^{II} e contemporaneamente su T, oppure y o y'.

Adesso poniamo:

$$F_5 = \big((T \wedge W_5^I) \vee (\neg T \wedge x) \big) \wedge \neg W_5^{II} \wedge (T \vee y)$$

Inizialmente T ed y sono vere, mentre x è falsa; e, a gioco corretto, questa situazione rimane inalterata fino alla fine della istanza (*partita*) di G_2 che simuliamo. Supponiamo innanzitutto che il gioco proceda correttamente, e che, ad un certo punto, uno dei due giocatori vinca la partita simulata, allora una delle formule W_5^I e W_5^{II} deve avere assunto il valore 1.

- Se questa è W_5^I, allora F_5 è anch'essa vera e I ha vinto.
- Se, invece, vince II, allora egli ha la possibilità di porre il valore di T a 0 durante la sua ultima mossa (quella che rende vera W_5^{II}). Alla semimossa successiva a questa operazione, I non può rendere vera F_5, perché per farlo dovrebbe *sia* rendere nuovamente falsa W_5^{II}, giocando in qualche modo in V_2^I, *sia* rendere vera $(\neg T \wedge x)$ giocando su x. II è, così, certo di arrivare incolume alla *sua* semimossa successiva, quando porrà a 0 anche il valore di y, in questo modo impedendo che in qualunque momento futuro F_5 divenga vera; d'ora in poi gli basterà giocare senza che T né y assumano nuovamente il valore 1 perché la clausola $(T \vee y)$, e con lei tutta F_5, rimangano false.

Adesso esaminiamo i diversi modi nei quali un giocatore potrebbe deviare dal gioco corretto.

- I **potrebbe porre** x **a** 1, ma in questo caso II reagirebbe rendendo falsa y, e, alla mossa successiva, falsa anche T, quindi, *di fatto, vincendo*. Durante la mossa intermedia I non può vincere se non ottenendo che W_5^I divenga vera, ma in questo caso avrebbe potuto vincere anche una mossa prima, *senza barare*.
- II **potrebbe porre** T **a** 0, ma in questo caso I vincerà immediatamente ponendo x a 1; sempre che II non abbia *anche* reso vera W_5^{II}, nel quale caso avrebbe semplicemente seguito la strategia *corretta* che abbiamo delineato in precedenza.
- II **potrebbe porre** y **a** 0, ma questo apparirebbe come un *turno saltato* nel gioco simulato, ed I reclamerebbe la propria vittoria rendendo vera W_5^I alla semimossa successiva. (A questo punto il lettore accorto osserverà che non tutte le mosse giocate da II nell'istanza simulata di G_2 sono in realtà *turni* di gioco corretto che lo sottopongono al pericolo di una *punizione* qualora vengano saltati: esistono infatti le mosse giocate per punire I a causa di una *sua* scorrettezza. Tuttavia a II, come al solito, *non conviene* saltare una delle mosse di questo tipo.)

□

Osservazione 2.2.8. *In realtà nelle dimostrazioni precedenti abbiamo semplicemente costruito istanze di diversi giochi proposizionali; mentre quello che avremmo dovuto fare sarebbe stato mostrare che queste possono essere prodotte in* **LOGSPACE** *a partire dall'input della nostra ATM; quindi mostrare che l'insieme delle posizioni vincenti per I di ciascuno dei nostri giochi può essere riconosciuto in* **EXPTIME**. *Onde saldare un vecchio debito espliciteremo di seguito una codifica per le formule e ne esamineremo sommariamente le (per altro poco interessanti) proprietà.*

2.2.1 Le riduzioni formalizzate

Innanzitutto definiamo le codifiche...

- **Una valutazione delle variabili** è una stringa nella forma

$$\langle(\nu_1\alpha_1 \dots \nu_n\alpha_n)\rangle$$

  ove le $\langle\nu_i\rangle$ sono variabili ed i simboli α_i appartengono a $\{0,1\}$. Alle variabili ν_i chiediamo inoltre di essere ordinate per *indici* strettamente crescenti se letti come rappresentazioni binarie di numeri naturali (questo per assicurarci che la medesima variabile non compaia due volte).

- Codifichiamo **le posizioni** dei giochi G_i sotto forma di liste delle codifiche dei diversi oggetti (formule o valutazioni delle variabili) che costituiscono le posizioni del gioco che ci interessa.

Osservazione 2.2.9. *Ora che ci siamo posti il problema della codifica dei giochi emerge una questione inaspettata, sebbene banale. Chi ci impedisce di scegliere fra le diverse codifiche possibili una che già in sé contenga la risposta alla domanda:"Chi vince?"? Se io, ad esempio, codificassi i nostri G_i come descritto sopra, tuttavia preponendo a quelle stringhe un simbolo che indica a quale giocatore la posizione è favorevole, allora ciascuno di quelli sarebbe ben lungi dalla completezza per* **EXPTIME** *(sarebbero linguaggi regolari!). Tuttavia è evidente come codifiche tanto sfacciate male riflettano la natura dei giochi: benché data una posizione sia facilissimo capire chi vince, eseguire una sola mossa legale è incredibilmente (***EXPTIME***escamente) complicato. Intuitivamente un gioco (un gioco giocato intendo, non l'astrazione matematica per un gioco) dovrebbe avere regole semplici dalle quali scaturiscono posizioni complesse; mentre una codifica del genere sortisce esattamente l'effetto opposto.*

Queste considerazioni ci spingono a dare la definizione seguente.

Definizione 2.2.10 (Gioco ragionevole). *Un gioco G è* ragionevole *quando:*

- *le posizioni sono stringhe in un qualche alfabeto finito,*
- *se p e q sono posizioni tali che $p \vdash_G^I q$ o $p \vdash_G^{II} q$, allora $|p| = |q|$,*
- *è possibile riconoscere in $\mathbf{P}$ le mosse legali (esistono macchine di Turing M^I ed M^{II} in $\mathbf{P}$ che riconoscono il linguaggio $p^\frown\langle\vdash\rangle^\frown q$ al variare di p e q fra le posizioni tali che rispettivamente $p \vdash_G^I q$ o $p \vdash_G^{II} q$, dove abbiamo fatto l'ipotesi che il simbolo $\vdash$ non appartenga all'alfabeto nel quale sono codificate le posizioni).*

L'unica di queste condizioni a necessitare di una giustificazione è la seconda; a noi, infatti, interessa che la dimensione delle posizioni non possa *scoppiare* durante la partita: se questo potesse avvenire non avremmo alcuna assicurazione sulla posibilità stessa di calcolare l'esito del gioco in un qualunque tempo finito!

La nostra definizione è giustificata dal

Lemma 2.2.11. *Dato un gioco ragionevole G esiste una macchina in* **EXPTIME** *che prende in input le posizioni di G ed accetta quelle fra queste che sono in $\mathcal{W}^I(G)$.*

Dimostrazione. Per questo, sfruttando il teorema 2.1.8, ci basta far vedere che l'insieme $\mathcal{W}^I(G)$ si può riconoscere in **APSPACE**. Una ATM che faccia questo deve innanzitutto predisporre sul proprio nastro tre tracce: sulle prime due scriverà due posizioni di G, mentre userà la terza per verificare la legalità delle mosse tramite l'algoritmo in **P** che esiste per ipotesi.

Quello che deve fare la nostra macchina è ripetere ciclicamente i due passi seguenti:

- indovinare esistenzialmente la mossa di I, rifiutando le mosse scorrette;
- controllare universalmete ogni possibile contromossa di II, accettando le mosse scorrette.

È fondamentale che le operazioni di successiva ipotesi e verifica delle mosse vengano eseguite senza mantenere traccia di tutte le mosse giocate, bensì scrivendo ogni volta la nuova posizione *sotto* quella vecchia, controllando la correttezza della mossa, quindi sostituendo la posizione appena costruita a quella precedente (se così non facesse, la macchina non potrebbe esaminare rami dell'albero delle mosse la cui lunghezza sia più che polinomiale). □

Ora il primo passo è mostrare che le codifiche fissate rendono i giochi G_1, G_2 e G_5 ragionevoli, e così ottenere che questi sono in **EXPTIME**.

Lemma 2.2.12. *I giochi G_1, G_2 e G_5 sono ragionevoli.*

Dimostrazione. La prima e la seconda condizione sono verificate banalmente. Quanto alla terza, basta osservare che la verifica di correttezza di una mossa non richiede che di calcolare, al più, il valore di verità di qualche formula proposizionale; ed è questo un compito che può essere eseguito facilmente in **P** per ricorsione. □

Ci rimane, infine, da osservare che

Lemma 2.2.13. *la traduzione dell'input della macchina di Turing alternante che abbiamo fissato all'inizio dei nostri discorsi, nelle posizioni dei giochi G_1, G_2 e G_5 può essere operata in* **LOGSPACE**.

Dimostrazione. Infatti questo richiede di:

- ottenere la nostra *codifica binaria* delle stringhe di input in assegnamenti di variabili proposizionali, la quale cosa si può fare molto facilmente e non necessita di più di un *contatore* per il valore degli indici delle variabili;
- scrivere le formule che determinano lo svolgimento dei giochi, la quale cosa non richiede più di qualche sostituzione di variabili.

□

2.3 Macchine di Turing alternanti & PSPACE

Dal momento che nella sezione precedente abbiamo avuto svariati esempi di giochi completi per la classe **ESPTIME**, nella presente (e più breve), intendo dare un esempio di linguaggio (che è quasi un gioco, e ci servirà per calcolare la complessità di un gioco) **PSPACE**-completo.

Definizione 2.3.1 (QBF). *Il linguaggio QBF (per* Quantified Boolean Formula*) è l'insieme delle formule proposizionali*

$$\Phi = \forall v_1 \exists v_2 \forall v_3 \dots \exists v_n \phi$$

con ϕ in forma normale congiuntiva, prive di variabili libere, vere.

Sarà chiaro per il lettore che la stretta alternanza di quantificatori universali ed esistenziali nella formula Φ sopra esposta è fatto del tutto inessenziale: se non fosse verificata, infatti, basterebbe aggiungere fra le coppie di quantificatori omologhi successivi delle *variabili cuscinetto* quantificate, che non abbiano altra funzione nella formula. È essenziale, tuttavia, verificare che la formula ϕ possa essere scritta in forma normale congiuntiva; infatti, benché ogni formula abbia sempre un equivalente in forma normale congiuntiva, questo può essere esponenzialmente più grande della formula di partenza.

Passiamo ora alla dimostrazione del seguente

Teorema 2.3.2. *QBF è* **PSPACE***-completo.*

Dimostrazione. Innazitutto è chiaro (dal titolo del capitolo) che sfrutteremo l'uguaglianza **PSPACE** = **APTIME**.

Per prima cosa dobbiamo, quindi, verificare che QBF sia in **APTIME**, la qual cosa non pone difficoltà. Basterà, infatti, costruire una ATM che legga la stringa iniziale di quantificatori esplorando entrambi i possibili valori di ciascuna variabile. La nostra macchina userà uno stato universale se la variabile che sta esaminando è quantificata universalmente, ed uno

stato esistenziale se la variabile è quantificata esistenzialmente; terminati i quantificatori iniziali, valuterà la verità di ϕ secondo il classico algoritmo deterministico in **P**.

Ora, ci rimane da ridurre ogni possibile linguaggio in **APTIME** a QBF. Come nella sezione precedente, fissiamo una ATM M che lavora in place **ed** in tempo inferiore al doppio della lunghezza dell'input (la quale cosa non è restrittiva per via del solito argomento di *padding*), e costruiamo una riduzione dal linguaggio deciso da M a QBF. Come nella sezione precedente, possiamo immaginare che M alterni stati universali a stati esistenziali (a partire da uno stato universale), e che le sue transizioni siano codificate dalla formula ψ_M allora costruita.

Trascuriamo, inizialmente, la restrizione che abbiamo imposto a Φ di essere in forma prenessa, ed a ϕ in forma normale congiuntiva. Consideriamo, quindi, un ramo dell'albero di computazione di M: l'input è codificato dalle variabili proposizionali $x_1^1, x_2^1, \ldots, x_L^1$, la configurazione di M dopo la prima transizione (universale) dalle variabili $x_1^2, x_2^2, \ldots, x_L^2$, e così via fino ad $x_1^{2L}, x_2^{2L}, \ldots, x_L^{2L}$. Costruiremo $2L$ formule α_i, la cui interpretazione è: *il sottoalbero della computazione di M che origina dalla configurazione codificata dalle variabili $x_1^i, x_2^i, \ldots, x_L^i$ è accettante.*

$$\alpha_i = \begin{cases} \bot & \text{per } i = 2L \\ \forall x_1^{i+1} \forall x_2^{i+1} \ldots \forall x_L^{i+1} (\neg \psi_M(x_1^i, x_2^i, \ldots, x_L^i) \vee \alpha_{i+1}) & \text{per } i \text{ dispari} \\ \exists x_1^{i+1} \exists x_2^{i+1} \ldots \exists x_L^{i+1} (\psi_M(x_1^i, x_2^i, \ldots, x_L^i) \wedge \alpha_{i+1}) & \text{per } i \text{ pari} \end{cases}$$

La coerenza di questa definizione con l'interpretazione data prima è facile conseguenza della definizione induttiva di *configurazione accettante*; quindi la formula α_1 è vera se e solo se la stringa codificata dalle variabili $x_1^1, x_2^1, \ldots, x_L^1$ appartiene al linguaggio deciso da M. Questa nostra α_1 è, altresì, di lunghezza quadratica in L; quindi, dal momento che la definizione delle α_i *è* un algoritmo in **LOGSPACE** per calcolarle, abbiamo un metodo per ridurre il linguaggio accettato dalla nostra generica macchina M a formule proposizionali nella forma

$$i\Phi = \exists x_1^1 \exists x_2^1 \ldots \exists x_L^1 (w_1 \wedge w_2 \wedge \cdots \wedge w_L \wedge \alpha_1)$$

ove col simbolo w_i si è indicato x_i^1 se la variabile x_i^1 è vera nella codifica dell'input di M, e $\neg x_i^1$ se x_i^1 è falsa.

Tutto quello che ci rimane da fare è portare queste formule nella nostra *forma normale* in **LOGSPACE**. Innanzitutto dobbiamo portare tutte le negazioni della formula Φ a ridosso dei simboli di variabile. Ciò significa leggere nell'ordine i simboli di Φ e per ogni simbolo letto eseguire le operazioni specificate di seguito:

- se il simbolo non appartiene a $\{X, \wedge, \vee, \exists, \forall, \neg\}$ scriverlo tale e quale nell'output;

- se il simbolo è $\neg$ non eseguire alcuna operazione;
- se il simbolo è X (il marcatore delle variabili), allora scrivere X se il numero delle negazioni *che agiscono* sul nostro simbolo è pari, altrimenti $\langle\neg X\rangle$;
- se il simbolo è $\vee$ ed il numero delle negazioni *che vi agiscono* è pari scrivere $\vee$, altrimenti $\wedge$;
- se il simbolo è $\wedge$ viceversa;
- se il simbolo è $\exists$ ed il numero delle negazioni *che vi agiscono* è pari scrivere $\exists$, altrimenti $\forall$;
- se il simbolo è $\forall$ viceversa.

Per far questo, ci rimane da contare il numero delle negazioni che agiscono, poniamo, su una determinata variabile (altrimenti su un determinato operatore o su un determinato quantificatore: questi altri due casi sono analoghi al primo). All'uopo sono sufficienti due *contatori*, inizialmente entrambi azzerati. Il primo, n, conta il numero delle negazioni, il secondo, p, ci serve per riconoscere le coppie di parentesi corrispondenti; la macchina leggerà la formula a ritroso dalla nostra variabile fino all'inizio agendo in questo modo:

- se legge $\neg$ e $p = 0$ incrementa n;
- se legge) incrementa p;
- se legge (e $p \neq 0$ decrementa p.

Questa procedura termina col contatore n che indica il numero cercato.

Adesso che Φ non ha più sottoformule negate, siccome, per costruzione, non ci sono variabili ripetute, possiamo portare tutti i quantificatori all'inizio. Basterà leggere la formula due volte, la prima scrivendo *solo* i quantificatori e la seconda scrivendo *solo* il resto.

Ora dimentichiamoci della stringa di quantificatori iniziali e concentriamoci sulla formula ϕ, che dobbiamo portare in forma normale congiuntiva. Come abbiamo osservato il *metodo classico* non funziona, poiché aumenta esponenzialmete la dimensione di ϕ. Quello che noi faremo è, invece, aggiungere una nuova variabile per ogni disgiunzione di ϕ che quantificheremo esistenzialmente.

Se supponiamo che ϕ abbia k disgiunzioni, applicheremo l'algoritmo seguente: scriviamo la stringa di quantificatori

$$\exists y_1 \exists y_2 \ldots \exists y_k$$

quindi la congiunzione di tante clausole quante sono le occorrenze di ciascuna variabile in ϕ. Ad ogni occorrenza di una variabile, diciamo v_i (eventualmente negata), corrisponde la clausola

$$v_i \vee z_1 \vee z_2 \vee \cdots \vee z_k$$

ove ciascuna z_i vale y_i se la nostra occorrenza di v_i ricade nel disgiunto di sinistra della i-esima disgiunzione, vale $\neg y_i$ se ricade nel disgiunto di destra, e vale $\bot$ altrimenti.

L'algoritmo appena delineato può essere implementato in **LOGSPACE**, in maniera per nulla dissimile da come abbiamo implementato l'algoritmo di *spostamento delle negazioni*, e sarebbe ozioso descriverlo più dettagliatamente. Convincersi della *correttezza* del nostro algoritmo è, poi, un semplice esercizio dell'induzione sulla complessità di ϕ. La dimostrazione è, quindi, conclusa. □

Capitolo 3

$C\&R$ è EXPTIME-completo

3.1 Definizioni

In questo capitolo ci occuperemo della complessità computazionale del gioco seguente: su un grafo G sono fissati n nodi, *i poliziotti del giocatore* I, ed un nodo, *il ladro del giocatore* II. I ed II si alternano spostando *i loro uomini* da un nodo ad uno adiacente del grafo (ciascuno ad ogni mossa, se desidera, può restare fermo, ed un qualunque numero di poliziotti si può muovere contemporaneamente ad ogni turno). Se in qualche momento (presumibilmente dopo una mossa di I) un poliziotto viene a trovarsi sullo stesso nodo del grafo sul quale si trova il ladro, allora I vince (lo scopo di II, chiaramente, è di sfuggire in perpetuo alla cattura).

Più precisamente:

Definizione 3.1.1 ($C\&R$). *$C\&R$ è il gioco così definito:*

- *Le posizioni sono rappresentate da triple (G, l, p) ove G è un grafo (non orientato), $l \in G$ è un nodo di G (che rappresenta la* posizione del ladro*), $p \in G^n$ è una n-tupla di nodi di G (*i poliziotti*);*
- *II, quando detiene l'iniziativa, può passare il turno o sostituire il nodo l con un altro nodo l' tale che $\{l, l'\}$ sia un arco di G;*
- *I, quando detiene l'iniziativa, può sostituire ciascun $p' = p_i$ con un p'' tale che $\{p', p''\}$ sia un arco di G, o lasciarlo qual'è;*
- *I vince se al termine di una sua mossa esiste i tale che $l = p_i$.*

Nel seguito, proseguendo la catena di riduzioni iniziata nel capitolo precedente, dimostreremo la **EXPTIME**-completezza di $C\&R$. Per farlo avremo bisogno del gioco seguente:

Definizione 3.1.2 ($\widetilde{C\&R}$). *Le posizioni di $\widetilde{C\&R}$ sono 4-tuple (K, G, l, p), ove*

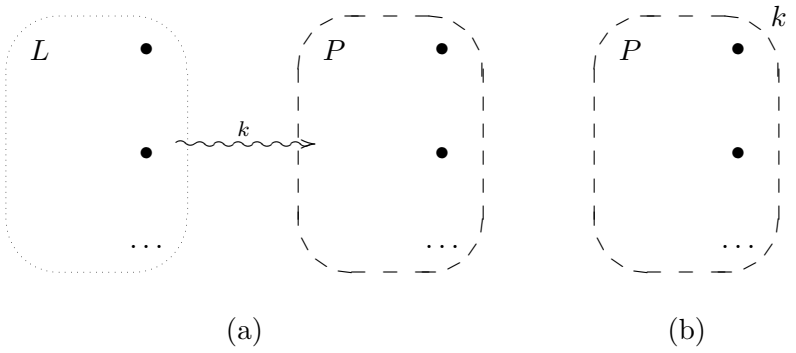

Figura 3.1: Alcuni esempi

- *l e p sono come in C&R;*
- *G è un grafo orientato;*
- *$K \in \mathcal{P}(G) \times \mathcal{P}(G) \times \mathbb{N}$ è una relazione fra coppie di insiemi di nodi di G e numeri naturali.*

Quanto alle mosse, sono come in C&R, salvo il fatto che il ladro può muoversi solamente nel verso dell'orientamento degli archi di G (mentre i poliziotti in entrambi i versi); questo implica che il ladro non può passare il turno *sostando sul medesimo nodo su cui si trova.*

Infine la condizione per la vittoria di I è la medesima di C&R, alla quale però si aggiunge la seguente:

- *II vince se al termine di un turno di I* il ladro non è stato preso *ed esiste una terna $(L, P, k) \in K$ tale che o $l \in L$ e $\nexists^{!k} i\ p_i \in P$ (i nodi di P non contengono esattamente k poliziotti) oppure $l \notin L$ e $\exists i\ p_i \in P$.*

Dapprima ridurremo G_5 a $\widetilde{C\&R}$, poi $\widetilde{C\&R}$ a $C\&R$, la qual cosa conclude la dimostrazione poiché come abbiamo dimostrato G_5 è **EXPTIME**-completo e, come dimostreremo, $C\&R$ è in **EXPTIME**. Dal momento che si parla di Ladri e Poliziotti, d'ora in poi P ed L verranno assunti come sinonimi per I e II.

3.2 Riduzione di G_5 a $\widetilde{C\&R}$

In questo paragrafo intendiamo mostrare un algoritmo in **LOGSPACE** che trasforma un'istanza di G_5 in un'opportuna istanza di $\widetilde{C\&R}$ tale che P disponga di una strategia vincente nella prima se e solo se dispone di una strategia vincente nella seconda.

Nei diagrammi, per maggior chiarezza, non sarà indicata l'orientazione degli archi del grafo, quando questa è ininfluente. Le relazioni in

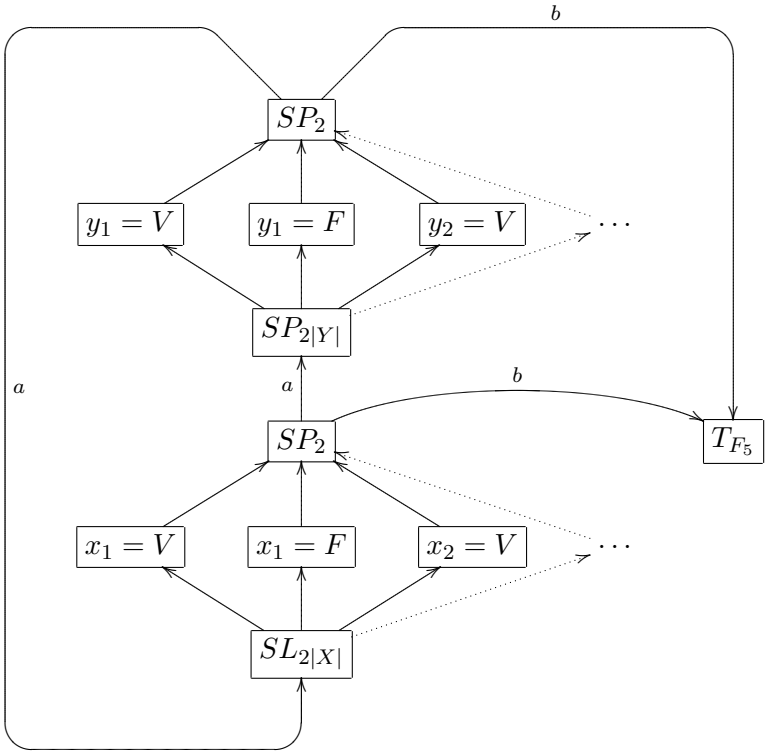

Figura 3.2: Piano della riduzione di G_5 a $\widetilde{C\&R}$

$(L, P, k) \in K$ verranno rappresentate come in figura 3.1(a). Spesso saranno utili relazioni del tipo $(G, P, k) \in K$ (che semplicemente forzano esattamente k poliziotti a restare sempre nei nodi di P dovunque si trovi il ladro in G); dette relazioni verrano indicate come in figura 3.1(b).

L'idea generale della costruzione è quella che appare dal diagramma in figura 3.2. Come di consueto definiamo una nozione di *gioco corretto*, se si deviasse dal quale, il giocatore colpevole dell'infrazione sarebbe destinato alla sconfitta. A gioco corretto, il ladro percorrerà gli archi del diagramma spostandosi fra i diversi blocchi (*gadget*) che vi si trovano, le peculiarità del cui funzionamento ora descriviamo.

Il blocco indicato con T_{F_5} rappresenta un meccanismo di verifica della formula $F_5(x_1, \ldots, x_{|V_5^I|}, y_1, \ldots, y_{|V_5^{II}|})$, essendo $|V_5^I = \{x_1, \ldots, x_{||V_5^I|}\}$ ed $V_5^{II} = \{y_1, \ldots, y_{|V_5^{II}|}\}$. Dopo che il ladro vi sia entrato, ammesso che egli giochi correttamente, può essere catturato dai poliziotti se e solo se la formula F_5 è vera; se questa condizione non fosse verificata, una volta entrato *correttamente* nel gadget, il ladro può raggiungere un rifugio sicuro che gli permette di evitare indefinitamente l'assalto della polizia.

Ovviamente, per poter decidere della verità di F_5, sono necessari mecca-

nismi che permettano di memorizzare, salvaguardare, e recuperare a tempo debito i valori delle variabili di V_5^I e V_5^{II}. A questo provvedono i blocchi indicati con $v_i =?$ nel diagramma: ogniqualvolta il ladro attraversa uno di questi (operazione che può sempre compiere senza correre rischi) la variabile v_i assume il valore ?.

Infine i gadget SP ed SL permettono rispettivamente a P e ad L (beh, in questo caso la cosa è un po' banale...) di scegliere la via di uscita del ladro dopo che vi sia entrato, pena per questo l'essere catturato se non rispetta la scelta di P una volta entrato in un gadget SP.

Inizialmente le variabili di F_5 hanno i valori che gli sono assegnati dalla posizione iniziale della istanza di G_5 che stiamo riducendo, ed il ladro si trova all'ingresso del meccanismo $SP_{2|V_5^I|}$. Questo meccanismo dà la possibilità a P, che detiene l'iniziativa nella suddetta istanza di G_5, di determinare il valore di una delle sue variabili indirizzando il ladro verso il meccanismo $x_i =?$ opportuno. Attraversato il gadget $x_i =?$ scelto da P, il ladro sarà forzato (dall'orientazione degli archi del grafo, che gli impedisce di sostare) ad entrare nel successivo gadget SP_2 che permetterà a P di inviarlo lungo uno dei due archi contrassegnati con a o con b. Chiaramente, se in quel momento F_5 sarà vera, P dirigerà il ladro lungo b verso il blocco T_{F_5} *reclamando* così la sua vittoria; mossa questa che risulterebbe fatale ai suoi propositi se invece F_5 fosse falsa, nel qual caso P opterebbe per la via a (a qualunque conseguenza porti questa mossa, la sua situazione può solo migliorare...). In questo caso il ladro si troverà all'ingresso di $SL_{2|V_5^{II}|}$, e la successione delle mosse che abbiamo descritto si ripetera similmente, salvo il fatto che ora è L ad avere l'opportunità di decidere il valore di una delle proprie variabili. Al termine di questo secondo ciclo il ladro giungerà nuovamente nel suo nodo iniziale (oppure verrà intrappolato per mezzo del gadget SP_2, sebbene la presenza di questo sia chiaramente ridondante e giustificata soltanto dalla simmetria) completando così la simulazione di una mossa completa di G_5.

Passiamo ora alla descrizione dettagliata dei singoli gadget.

• SL_n **ed** SP_n Questi sono i due meccanismi più semplici. SL_n consiste unicamente in n archi in uscita dal *nodo di ingresso* del meccanismo.

SP_n è, invece, rappresentato in figura 3.3. Il ladro vi entra dal nodo α ed alla mossa successiva è forzato a spostarsi verso uno degli n nodi indicati con a_i, da dove non potrà che essere preso da un poliziotto o seguire l'unica via d'uscita disponibile. In questo modo la scelta della via d'uscita dal gadget ricade solo sulla scelta di uno fra i nodi suddetti che sia *sicuro*, cioè non direttamente minacciato dalla presenza di un poliziotto nel corrispondente nodo b_i.

Inizialmente $n-1$ poliziotti si trovano nel nodo β ed a gioco corretto questi poliziotti rimarranno costantemente nel sottografo rappresentato a destra nella figura, se non eventualmente per eseguire una *mossa di cattura*

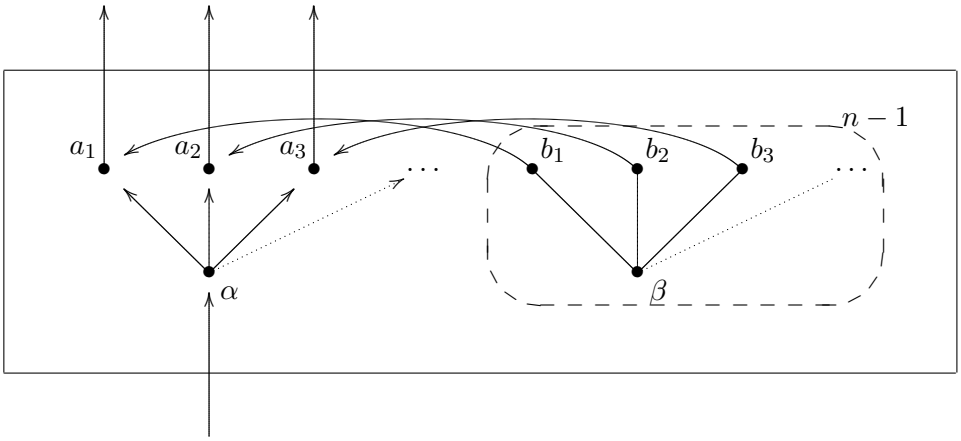

Figura 3.3: SP_n

lungo uno degli archi (b_i, a_i). L'unico momento nel quale questi poliziotti possono effettivamente *interagire* col gioco è, quindi, quando il ladro si trova in α, poiché la loro posizione al termine di questa mossa determina quali vie saranno interdette al ladro nella successiva; e proprio durante questa mossa essi possono disporsi (P li può disporre) in modo tale da minacciare esattamente $n-1$ qualunque degli a_i, forzando effettivamente la mossa del ladro verso l'unica via non controllata ed ottenendo lo scopo del gadget. Va da sé che minacciare più di $n-1$ degli a_i è impossibile, minacciarne meno inutile o dannoso.

• $v =?$ Per ogni variabile v di F_5 compariranno nel nostro G tre sottografi, due dei quali sono quelli rappresentati nel diagramma di figura 3.2 dai blocchi $v = F$ e $v = T$ forzando il ladro attraverso i quali, i giocatori possono alterare il valore della variabile v; il terzo, che nel diagramma non compare, è il meccanismo di cui ci serviremo per *memorizzare* il valore della variabile. Quest'ultimo è il sottografo rappresentato al centro di figura 3.4 (i nodi v e $\neg v$ saranno poi opportunamente collegati al blocco T_{F_5} del diagramma; della qual cosa, però, dirò di più in seguito: il lettore mi conceda che il modo nel quale detto collegamento ha luogo non altera il funzionamento di questo gadget).

Inizialmente un poliziotto è posto in v o $\neg v$ secondoché il valore di partenza della variabile v sia V o F; questo poliziotto, chiaramente, non si può spostare da dove si trova finché il ladro non transita per uno dei gadget $v = V$ o $v = F$; infatti i nodi β_V, β_F ed α gli sono preclusi, e non si può transitare fra l'uno e l'altro dei nodi v e $\neg v$ senza transitare per uno di

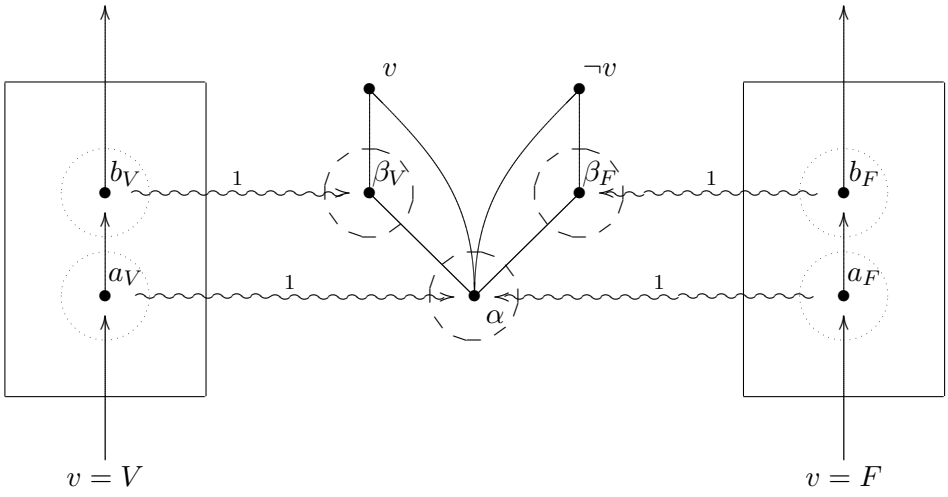

Figura 3.4: v =?

questi. In questo modo la memoria del valore di v si conserva affidabilmente.

All'ingresso del ladro nel sottografo $v = X$ attraverso il nodo a_X il poliziotto è forzato a spostarsi in α, dopo di che il ladro deve muoversi in b_X spingendo il poliziotto in β_X ed infine deve uscire dal gadget. Alla mossa successiva, essendo il ladro nuovamente fuori dal gadget, il poliziotto dovrà tornare verso uno dei due nodi v e $\neg v$; precisamente sarà forzato a spostarsi in v se $X = V$, poiché questo è l'unico raggiungibile da β_V, e viceversa in $\neg v$ se $X = F$.

In questo modo lo scopo del nostro gadget è stato ottenuto.

• T_ϕ Per la costruzione di quest'ultimo gadget procederemo ricorsivamente sulle sottoformule di ϕ. Per questo si supporrà che in ϕ compaiano solo gli operatori booleani $\wedge$, $\vee$ e $\neg$, quest'ultimo esclusivamente applicato a simboli di variabile; chiaramente data una qualunque formula è possibile produrne una siffatta in **LOGSPACE**.

In figura 3.5(a) è rappresentato il caso in cui ϕ sia atomica (precisamente $\phi = v$; se $\phi = \neg v$ si procede allo stesso modo sostituendo al nodo v il nodo $\neg v$). Il nodo d'ingresso a costituisce chiaramente una barriera invalicabile ai poliziotti, se quindi il ladro potrà raggiungere i due nodi b e b' gli sarà sufficiente muoversi fra l'uno e l'altro di questi due per sfuggire perpetuamente alla cattura. Ne segue che, una volta entrato nel gadget, il ladro può essere catturato se e solo se un poliziotto si trova in v (se e solo se $v = V$), in caso contrario, infatti, il ciclo b-b' garantirà la futura incolumità del ladro. Il nodo v è ovviamente il medesimo che abbiamo descritto nella costruzione dei gadget v =?; come abbiamo già notato la presenza dell'arco (v, a) non

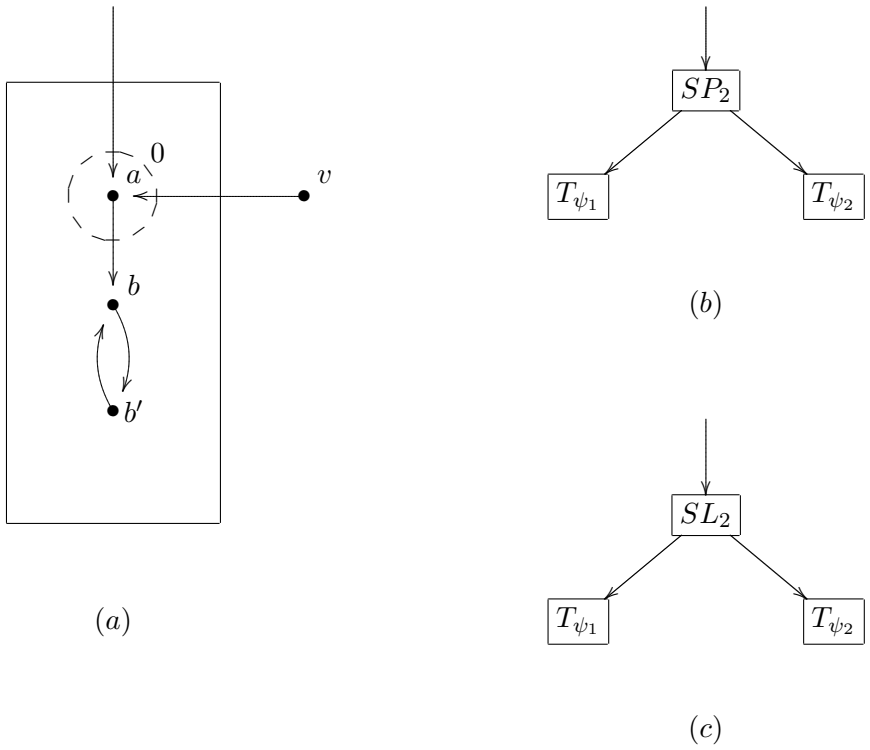

Figura 3.5:

altera il funzionamento di detto gadget (e di ogni altro gadget del tipo T_ϕ collegato al nodo v) poiché l'eventuale poliziotto in v, a gioco corretto, non può percorrere questo arco se non per prendere il ladro. Con questo abbiamo mostrato che il nostro meccanismo verifica le proprietà per T_ϕ nel caso di ϕ atomica.

Supponiamo ora $\phi = \psi_1 \vee \psi_2$; allora T_ϕ sarà costruito come in figura 3.5(b). Se almeno una delle formule ψ_1 e ψ_2 è vera P può indirizzare il ladro per mezzo del meccanismo SP_2 verso il gadget corrispondente ottenendo la vittoria, altrimenti qualunque sia la scelta di P, il ladro sfuggirà alla cattura (per ipotesi induttiva).

Simmetricamente, se $\phi = \psi_1 \wedge \psi_2$, si vede che il sottografo di figura 3.5(c) ottiene lo scopo.

A questo punto la riduzione di G_5 a $\widetilde{C\&R}$ è completa, a patto di verificare che la costruzione appena esposta possa essere eseguita in **LOGSPACE**.

3.2.1 Complessità della riduzione di G_5 a $\widetilde{C\&R}$

Alla nostra lista di codifiche è semplice aggiungere:

- **gli insiemi di nodi**, come *liste* di nodi;
- **le relazioni fra insiemi di nodi**, come liste di coppie (liste di due) insiemi di nodi;
- ed infine **le posizioni di** $\widetilde{C\&R}$ che non sono altro che liste finite degli elementi sopra descritti.

Ora descriviamo, gadget per gadget, una macchina in **LOGSPACE** che esegue la riduzione tratteggiata nei paragrafi precedenti. Osserviamo che la macchina che ci accingiamo a costruire restituisce in output (la posizione iniziale del ladro e una lista di) tre liste: il grafo, la relazione K e la posizione iniziale dei poliziotti p. Noi la tratteremo esattamente come se avesse tre nastri di output e potesse scrivere, *a sua scelta*, in ogni momento, su ciascuno di questi tre nastri. Questa semplificazione è giustificata; infatti otteniamo la triplice lista di liste che ci interessa eseguendo il medesimo algoritmo per tre volte consecutive, la prima scrivendo *solo* l'output destinato al primo nastro, la seconda *solo* quello del secondo e la terza *solo* quello del terzo.

• $SL_{|V_5^{II}|}$ **ed** $SP_{|V_5^{I}|}$ **rivisitati** Come al solito questi sono semplici. Associamo al nodo radice del primo il numero $\langle(1)\rangle$, ed ai nodi che se ne diramano i numeri $\langle(n_i1001)\rangle$ dove gli $\langle(n_i)\rangle$ sono gli indici delle variabili di V_5^{II}. La nostra **LOGSPACE**-TM non deve far altro che leggere nell'ordine gli n_i dall'istanza data di G_5 e scrivere le coppie $\langle((1)(n_i1001))\rangle$ nella lista degli archi del grafo.

Per $SP_{|V_5^{I}|}$ agiamo in modo simile ed associamo ai suoi nodi α e β i numeri $\langle(101)\rangle$ ed $\langle(1101)\rangle$, quindi alle rispettive diramazioni $\langle(n_i10101)\rangle$ ed $\langle(n_i11101)\rangle$, etc.

• v =? **rivisitati** Precisamente nella stessa maniera di prima associamo ai nove vertici che *gestiscono* una certa variabile altrettanti numeri univoci ($\langle(n_i1000110)\rangle$,$\langle(n_i1001010)\rangle$,...,$\langle(n_i1100110)\rangle$), quindi leggiamo una per una le variabili e per ciascuna scriviamo le componenti del gadget corrispondenti.

• T_ϕ **rivisitato** Questa costruzione è vagamente più noiosa delle precedenti. Innanzitutto portiamo le negazioni di ϕ a ridosso dei simboli di variabile, seguendo la tecnica esposta nella dimostrazione del teorema 2.3.2.

Ora supponiamo di avere una formula le cui negazioni sono applicate unicamente ai simboli di variabile. Per generare il grafo che ci interessa, innanzitutto associamo ad ogni operatore ($\vee$ o $\wedge$) un numero univoco dato dalla

sua posizione nella formula (1 per il primo operatore, 2 per il secondo, etc.). Quindi osserviamo che per ogni operatore (o numero fra 1 ed il numero degli operatori della formula) possiamo calcolare gli (indirizzi degli) operatori che a lui sono *collegati direttamente* (infatti basta partire da questo operatore, saltare la prima parentesi andando, poniamo, verso destra e proseguire fino alla parentesi che corrisponde alla seconda: il carattere successivo sarà uno dei nostri due operatori; l'altro si ottiene simmetricamente).

Il lettore accorto avrà anche osservato che nel numerare i nodi dei gadget precedenti abbiamo usato un suffisso diverso per ogni gadget, onde assicurarci di non usare due volte lo stesso nodo. Ora ci è rimasto il suffisso $\langle 11\rangle$, che useremo per i nodi che stiamo costruendo. Diciamo che ogni operatore, il cui numero sia, scritto in notazione binaria, $\langle\alpha\rangle$, ha un nodo di ingresso, $\langle(\alpha 100011)\rangle$, e due nodi di uscita, $\langle(\alpha 101011)\rangle$ e $\langle(\alpha 110011)\rangle$. Questo ci lascia liberi di usare i nodi $\langle(\alpha 10011011)\rangle$, $\langle(\alpha 10111011)\rangle$ ed $\langle(\alpha 11011011)\rangle$ per i meccanismi interni degli SP_2, quando capitano.

Già abbiamo descritto come produrre questi gadget, sappiamo, inoltre, che dato ogni operatore possiamo calcolare a quali nodi di ingresso (se esistono) collegare i suoi nodi di uscita. Ci resta da produrre un grafo simile al sottografo della figura 3.5(a) per ogni variabile e da collegare questi ai nodi opportuni dell'albero che andiamo costruendo. Tuttavia il lettore (ormai anche quello poco accorto, se ve ne sono) avrà osservato che ci siamo riservati l'utile suffisso $\langle 111\rangle$ e, con questo, saprà risolvere il problema da sé.

3.3 Riduzione di $\widetilde{C\&R}$ a $C\&R$

Per prima cosa dobbiamo risolvere la questione degli archi orientati. Il modo più semplice è costruire un gioco intermedio fra $\widetilde{C\&R}$ e $C\&R$, che non perderemo tempo a definire in modo formale, e verboso. In pratica si tratta di una versione semplificata di $\widetilde{C\&R}$ del quale mantiene tutte le regole, fatta eccezione per l'orientazione degli archi: nel nostro gioco intermedio, che chiameremo $\widehat{C\&R}$, gli archi *non* sono orientati ed il ladro può sostare per più di un turno sul medesimo nodo. In altre parole $\widehat{C\&R}$ è $C\&R$ al quale è stata aggiunta la condizione per la vittoria di L basata sulla relazione K.

3.3.1 $\widehat{C\&R}$ si riduce a $\widetilde{C\&R}$

Onde simulare il sottografo in figura 3.6(a), o più precisamente l'arco orientato fra b e c di questo, usiamo il gadget che il lettore può vedere in figura 3.6(b). Un poliziotto si trova, inizialmente, in f' e, finché il ladro non passa per b' o c', vi resta. Quando il ladro arriva in b' il nostro poliziotto è forzato a spostarsi in d', da dove minaccia i nodi a' e b'. In questo modo il ladro deve *sfuggire alla presa* scappando in c', costringendo il poliziotto a *rientrare* tramite il nodo e'. Se il ladro dovesse passare per c' senza essere,

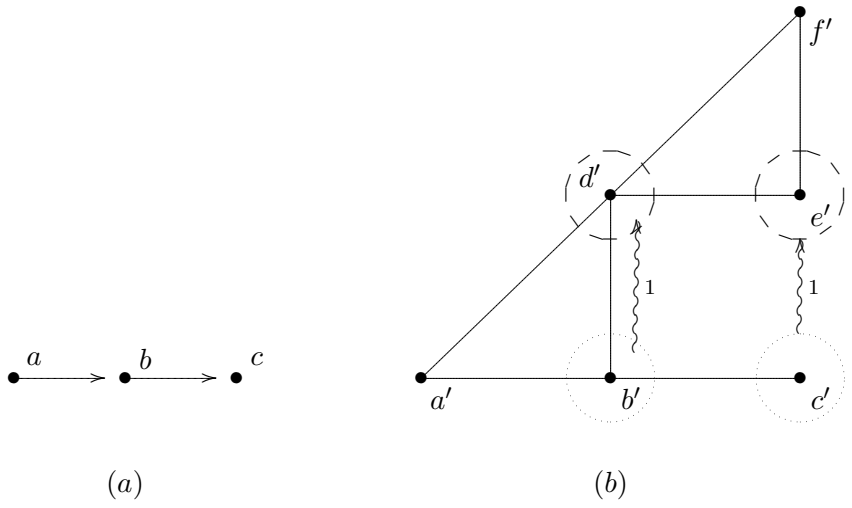

Figura 3.6: Simulazione di un arco orientato

prima, passato per b', il poliziotto si sposta semplicemente da f' in e', per poi tornarvi quando il ladro si sia allontanato nuovamente dal nodo c'. Con questo abbiamo provato che, se il ladro ad una certa mossa si trova in b', alla mossa successiva è obbligato a spostarsi in c', pena la sconfitta.

Chiaramente, il gadget di figura 3.6 non basta *da solo* a simulare ogni arco orientato: esso è utile, *così com'è*, soltanto quando da un certo nodo (b) parte un solo arco orientato ed uno solo ne arriva. Tuttavia un maggior numero di archi *in ingresso* non è un problema: basterà collegare ciascuno dei possibili *nodi di provenienza* con d'. Nel caso di un maggior numero di archi *in uscita* la soluzione è analoga: si dovrà costruire un nodo che fa le veci di e' per ciascuno di essi.

3.3.2 Simulazione della relazione K

Infine ci rimane la relazione K; tuttavia il lettore non si illuda che la riduzione arrivi ad un termine con *questo* capitolo: non è così! Per ora, accetteremo l'assunto seguente, che troverà la sua dimostrazione, per motivi di convenienza, nel prossimo capitolo.

- Il grafo che costruiremo ha un *nodo speciale*, che indicheremo con $\star$: se il ladro riesce a raggiungere il nodo $\star$, allora ha vinto.

Otterremo il nostro scopo per gradi. Supponiamo, innanzitutto, di avere una istanza di $C\&R$, e di voler imporre fra due nodi del grafo, a e b, la relazione seguente:

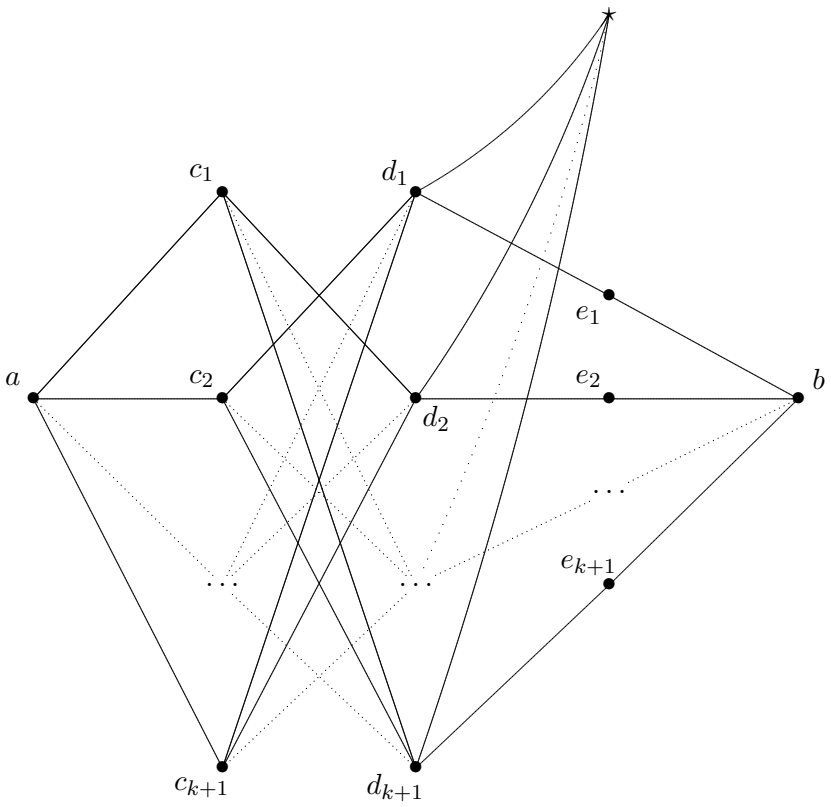

Figura 3.7: $F_{\{a\},\{b\},k}$

> quando il ladro si trova nel nodo a, almeno k poliziotti devono trovarsi nel nodo b.

Questo è uno scopo che può essere raggiunto per mezzo del sottografo in figura 3.7.

Supponiamo che i nodi c_i, d_i ed e_i siano sgombri da poliziotti, che il ladro si trovi in a, e che k poliziotti si trovino in b. Allora, se il ladro tenta di avviarsi verso $\star$ muovendo in c_j, i k poliziotti si possono spostare verso i nodi $e_1, e_2, \ldots, e_{j-1}, ej+1, \ldots, e_{k+1}$ impedendogli la fuga (il fatto essenziale è che ognuno dei nodi c_i è connesso con esattamente k dei $k+1$ nodi d_i, i quali si trovano ad un passo dalla salvezza). Se, tuttavia, i poliziotti in b sono meno di k, essi non potranno minacciare tutte le k possibili vie di fuga del ladro, il quale riuscirà a scappare verso $\star$. Supponiamo infine che k (o meno) poliziotti, inizialmente in b, tentino di *barare* muovendosi verso i nodi e_i senza che il ladro lasci il nodo a; allora il ladro potrà raggiungere il nodo $\star$ scegliendo un j tale che il nodo e_j sia occupato da un poliziotto e spostandosi in c_j (infatti da questo nodo sono raggiungibili *tutti* i d_i *eccetto* d_j, che possono essere *controllati* solo dal numero insufficiente di $k-1$ poliziotti,

poiché uno dei k poliziotti si trova in e_j). Riassumendo abbiamo dimostrato che

> **se** i nodi b_i, c_i e d_i sono inizialmente sgombri, al più k poliziotti possono accedere al nodo b, ed il ladro si trova in a, **allora** esattamente k poliziotti sono costretti a portarsi in b, pena la sconfitta.

Ora il lettore vedrà che è possibile estendere la costruzione del gadget precedente in modo da ottenere, dati due insiemi di nodi A e B, un gadget $F_{A,B,k}$ che verifichi la proprietà seguente:

> **se** i nodi di $F_{A,B,k}$ sono inizialmente sgombri, al più k poliziotti possono accedere ai nodi di B, ed il ladro si trova in uno dei nodi di A, **allora** esattamente k poliziotti sono costretti a portarsi all'interno dell'insieme di nodi B.

L'espediente è immediato: se $A = a$ è costituito da un solo nodo costruiamo i c_i ed i d_i come in figura 3.7, poi colleghiamo ciascun d_i con ciascun nodo di B con $k+1$ diversi nodi e_i (in pratica, alla fine, avremo $(k+1)|B|$ nodi diversi al posto dei $k+1$ e_i); se $|A| > 1$ ci basta costruire un diverso gadget per ogni nodo di A.

Adesso siamo pronti per l'ultimo gradino. Supponiamo che (A, B, k) appartenga a K e *attacchiamo* al nostro grafo G i seguenti gadget, onde essere certi che la condizione rappresentata da quella terna abbia forza. I gadget che ci servono sono i seguenti:

- $F_{A,B,k}$,
- $F_{A,\mathcal{P}(G)\setminus B,T-k}$,
- $F_{\mathcal{P}(G)\setminus A,\mathcal{P}(G)\setminus B,T}$;

dove con T si è indicato il *numero totale* dei poliziotti nel grafo G. La concomitanza dei primi due gadget ci assicura che, quando il ladro si trova in A, *esattamente* k poliziotti si trovino in B; il terzo garantisce che il sottografo B sia vuoto quando il ladro non si trova in A. Dal momento che *dovunque sia il ladro* tutti e T i poliziotti sono *costretti* in determinati sottografi, nessuno di essi può tentare di barare *sovraccaricando* uno dei gadget F_{qualcosa}, o uscendo dal grafo G per avventurarsi nei nodi *interni* di uno dei grafi F_{qualcosa}.

Con questo anche il penultimo passo della nostra riduzione si può dire concluso, **LOGSPACE**-fattibilità permettendo...

3.3.3 Complessità della riduzione di $\widetilde{C\&R}$ a $C\&R$

Per le stesse considerazioni che abbiamo fatto nel caso della riduzione di G_5 a $\widetilde{C\&R}$, l'unico punto che merita qualche attenzione è la costruzione

dei grafi F_{qualcosa}. Questa stessa costruzione non presenta problemi; l'unico fatto apparentemente imbarazzante è che, se gli elementi di K sono scritti nella forma $\langle\alpha\beta\gamma\rangle$, con $\langle\alpha\rangle$ e $\langle\beta\rangle$ liste di nodi e $\langle\gamma\rangle$ un numero intero, allora per produrre uno dei corrispondenti F_{qualcosa} potrebbe servire spazio $O\big(\log(\gamma)\big) = O\big(|\langle\gamma\rangle|\big) \overset{\text{possibilmente}}{=} O\big(|\langle\alpha\beta\gamma\rangle|\big)$. Questo fatto tuttavia è solo apparentemente indice di un'eccessiva complessità, infatti ogniqualvolta si usa un gaget $F_{A,B,k}$, allora k è sempre minore di $|\mathcal{P}(G)|$ (per costruzione).

Capitolo 4

$C\&R^*$ è PSPACE-hard

In questo capitolo mostreremo la **PSPACE**-hardness del gioco $C\&R^*$ qua sotto definito.

Definizione 4.0.1 ($C\&R^*$). *La posizione iniziale di $C\&R^*$ è il dato di un grafo G e di un numero intero positivo n. Inizialmente l'iniziativa è di P, il quale sceglie una n-tupla p di nodi di G (non necessariamente distinti). Quindi l'iniziativa passa ad L che sceglie un nodo l di G. Il gioco prosegue, quindi, come in $C\&R$ a partire dalla posizione (G, l, p).*

In pratica si tratta del nostro vecchio $C\&R$, al quale è stata preposta una fase di due mosse durante le quali i giocatori *scelgono* le proprie posizioni iniziali. Non diamo (per non essere riusciti ad ottenerla) una classificazione completa della complessità di questo gioco: quello che possiamo dire è che esso è (ovviamene) in **EXPTIME** ed è **PSPACE**-hard. Goldstein congettura che sia **EXPTIME**-completo, cosa, per altro, probabile; tuttavia, il problema rimane aperto.

La dimostrazione si ottiene per riduzione dal linguaggio QBF, che si è dimostrato **PSPACE**-completo. La tecnica sarà, tuttavia, più chiara, se preponiamo alla nostra dimostrazione un'altra, altrimenti inutile, dimostrazione di **PSPACE**-hardness del gioco $C\&R$.

In questo capitolo salderemo, altresì, il debito contratto con la fiducia del lettore nel precedente, e costruiremo il nodo $\star$ che in quello abbiamo utilizzato.

4.1 Il gioco $C\&R^{++}$, e la sua riduzione...

Ora descriveremo il gioco $C\&R^{++}$ e la relativa riduzione a $C\&R$ (lettore resisti: ogni nuovo gioco è un gioco in meno prima dell'ultimo).

Le regole di $C\&R^{++}$ sono le seguenti:

Definizione 4.1.1 ($C\&R^{++}$). *$C\&R^{++}$ è il gioco così definito:*

- *Le posizioni sono* 5*-tuple* (G, l, p, ν, α) *ove* G *è un grafo (non orientato),* $l \in G$ *è un nodo di* G *(il solito ladro),* $p \in G^n$ *è una* n*-tupla di nodi di* G *(i soliti poliziotti),* ν *è una funzione che mappa i nodi di* G *in* $\{\bullet, \circ\}$*, e* α *è una funzione che mappa gli archi di* G *in* $\{$ $,$ $\}$*;*
- *L, quando detiene l'iniziativa, può passare il turno o sostituire il nodo* l *con un altro nodo* l' *tale che* $\{l, l'\}$ *sia un arco di* G*;*
- *P, quando detiene l'iniziativa, può sostituire ciascun* $p' = p_i$ *con un* p'' *tale che* $\{p', p''\}$ *sia un arco di* G*, o lasciarlo qual'è;*
- *P vince se al termine di una mossa di L* **o** *esiste un* i *tale che* $l = p_i$ *e* $\nu(l) = \bullet$*,* **o** *esiste un arco* g *di* G *che connette il nodo* l *ed uno dei nodi* p_i*, tale che* $\alpha(g) =$ *.*

In pratica, i grafi su cui si gioca $C\&R^{++}$ hanno nodi di due tipi, $\bullet$ e $\circ$, ed archi di due tipi, e . Che $C\&R^{++}$ sia un'estensione di $C\&R$ è abbastanza chiaro: infatti una posizione di $C\&R$ non è altro che una posizione di $C\&R^{++}$ in cui tutti i nodi sono di tipo $\bullet$ e tutti gli archi di tipo . L'idea è che gli elementi aggiuntivi (i nodi di tipo $\circ$ e gli archi di tipo) rappresentano delle *zone franche*, sulle quali i poliziotti possono transitare, ma muovendosi sulle quali i poliziotti non possono prendere il ladro.

4.1.1 Riduzione di $C\&R^{++}$ a $C\&R$

Supponiamo di avere fissata una istanza (G, l, p, ν, α) di $C\&R^{++}$ con k poliziotti.

Innanzitutto consideriamo il grafo H così definito:

- l'insieme dei nodi di H è $\mathbb{Z}_p \times \mathbb{Z}_p$, con p un numero primo;
- due nodi (a, b) e (a', b') sono collegati da un arco di H se e solo se $aa' \equiv b + b' \pmod{p}$.

Consideriamo, quindi, la seguente ripartizione dei nodi di H:

$$\mathcal{P}(H) = \bigsqcup_{0 \le i < \lfloor \sqrt{p} \rfloor} H_i$$

con

$$H_i = \left\{ (a, b) \in \mathcal{P}(H) : i\frac{p}{\lfloor \sqrt{p} \rfloor} \le a < (i+1)\frac{p}{\lfloor \sqrt{p} \rfloor} \right\}$$

Si può vedere facilmente che ciascun nodo di $\mathcal{P}(H)$ ha almeno $\lfloor \sqrt{p} \rfloor$ nodi adiacenti in ognuno degli H_i, e che due nodi diversi hanno sempre al più un nodo adiacente in comune. Ne segue che **se** una posizione dei $C\&R$ è costituita dal grafo H, sul quale si trovano k poliziotti con $2k < \lfloor \sqrt{p} \rfloor$, il ladro non si trova sul medesimo nodo di un poliziotto, e l'iniziativa è del ladro; **allora**, ad ogni turno

scelto un qualunque i minore di $\lfloor\sqrt{p}\rfloor$, egli potrà muoversi in un nodo dell'insieme H_i, per di più evitando la cattura in perpetuo.

La strategia del ladro per ottenre questo risultato è semplice: ogni poliziotto minaccia al più due dei nodi che il ladro può raggiungere, ed egli può raggiungere almeno $\lfloor\sqrt{p}\rfloor > 2k$ nodi in ciascun H_i.

Ora fissiamo un valore di p tale che $2k < \lfloor\sqrt{p}\rfloor$ e $|\mathcal{P}(G)| < \lfloor\sqrt{p}\rfloor$; quindi *modifichiamo* il grafo H nel modo seguente, cosicché *simuli* l'istanza di $C\&R^{++}$ che avevamo fissata.

- Sia $\mathcal{P}(G) = \{n_0, n_1, \ldots, n_m\}$.
- Per ogni i tale che $m < i < \lfloor\sqrt{p}\rfloor$, eliminiamo tutti i nodi di H_i, insieme con tutti gli archi cui questi nodi appartengono.
- Per ogni $i \leq m$, se $\nu(n_i) = \bullet$ aggiungiamo ad H tutti gli archi che collegano fra loro due nodi di H_i.
- Per ogni $i, j \leq m$, se $\{i, j\}$ *non* è un arco di G, eliminiamo tutti gli archi che collegano un nodo di H_i con un nodo di H_j.
- Per ogni $i, j \leq m$, se $\{i, j\}$ è un arco di G di tipo $\diagup$, aggiungiamo ad H tutti gli archi che collegano un nodo di H_i con un nodo di H_j.

Chiamiamo H' il grafo H modificato nella maniera descritta sopra. Il grafo H' simula la nostra istanza di $C\&R^{++}$ in questo senso:

- i nodi $n_0, n_1, \ldots, n_m$ di G *corrispondono* agli insiemi di nodi $H_0, H_1, \ldots, H_m$ di H': ogniqualvolta un nodo, n_j, è occupato dal ladro (o un poliziotto) nella *partita simulata*, uno dei nodi del corrispondente H_j è occupato dal ladro (o da un poliziotto) nella simulazione;
- in questo modo, chiaramente, se L perde nel gioco simulato, allora il ladro può essere catturato anche in H';
- L, in più, può giocare in modo tale da sfuggire alla presa finché P non vince nell'istanza di $C\&R^{++}$ simulata, seguendo la seguente strategia: *ad ogni mossa il ladro si sposta in uno dei nodi del sottoinsieme H_i corrispondente alla sua destinazione nel gioco simulato, in modo tale da non poter essere preso dai poliziotti se questi si muovessero sugli archi del grafo H (prima della modifica).*

La correttezza della strategia delinata per il giocatore L è evidente, infatti un poliziotto che si muovesse su un arco di tipo $\smile$ o che *presidiasse* un nodo di tipo $\circ$ non potrebbe che spostarsi sugli archi di H, senza, quindi, rappresentare un pericolo per il ladro. Che questa strategia possa essere applicata (cioè che esista sempre un nodo raggiungibile al ladro ed H-sicuro) può essere dedotto dalla considerazione seguente. Supponiamo che il *ladro*

simulato si sia mosso da n_i ad n_j, allora il *ladro simulante* sarà in H_i e si dovrà dirigere verso H_j. l'arco $\{n_i, n_j\}$ è presente in G (poiché il ladro simulato l'ha percorso), quindi nessuno degli archi fra i nodi di H_i e quelli di H_j è stato rimosso; allora, siccome la strategia suddetta poteva essere seguita in H, deve poter essere seguita anche in H'.

4.1.2 Complessità della riduzione da $C\&R^{++}$ a $C\&R$

Come al solito dobbiamo verificare la **LOGSPACE**-fattibilità.

Innanzitutto dobbiamo trovare il numero primo p. Siccome, per un noto teorema, esiste sempre un numero primo fra n e $2n$; si può trovare un valore di p che verifica le ipotesi inferiore a $2\big(max(|\mathcal{P}(G)|, 2k)+1\big)^2$. Questo numero è $O(|\mathcal{P}(G)|^2)$, quindi polinomialmente limitato nella lunghezza dell'input; ne segue che le operazioni richieste per trovare p possono essere eseguite in **LOGSPACE**.

Ora si tratta di costruire il grafo H'. Chiaramente non possiamo permetterci di costrure H per poi ottenerne H' tramite le operazioni di taglio e cucito descritte nella sezione precedente. L'algoritmo da seguire è il seguente:

- si enumerano tutte le coppie di nodi di H, la qual cosa richiede quattro contatori, per ciascuna eseguendo le operazioni seguenti;
- si determinano i e j tali che i due nodi appartengono ad H_i ed H_j, la qual cosa richiede altri due contatori;
- se $i > m$ o $j > m$ si passa alla coppia successiva;
- se $\{n_i, n_j\}$ non è un arco di G e $i \neq j$, si passa alla coppia successiva;
- se $\{n_i, n_j\}$ è un arco di tipo $\diagup$ si scrive la coppia nell'output e si passa alla coppia successiva;
- se $i = j$ ed n_i è un nodo di tipo $\bullet$ si scrive la coppia e si passa alla successiva;
- se la coppia di nodi è un arco di H la si scrive;
- si passa alla coppia successiva;

Questo algoritmo può essere implementato in **LOGSPACE** dal momento che, come è già stato osservato, le operazioni aritmetiche sono calcolabili in **LOGSPACE**.

4.2 $C\&R$ è EXPTIME-completo

Ora, finalmente, sappiamo come terminare la costruzione del capitolo precedente. Tutto quello che ci mancava era la simulazione di un nodo *magico*, $\star$, che dà la vittoria al ladro, qualora questo lo raggiunga. Il nostro nodo $\star$ è semplicemente un nodo di tipo $\circ$ collegato al resto del grafo da soli archi di tipo $\frown$. Possiamo quindi enunciare il seguente:

Lemma 4.2.1. *G_5 si riduce in* **LOGSPACE** *a $C\&R$.*

Per di più si ha che:

Lemma 4.2.2. *$C\&R$ è un gioco ragionevole.*

Dimostrazione. Con le codifiche che abbiamo descritto, in realtà, *$C\&R$ non è* ragiovevole. Tuttavia la correttezza delle mosse può essere ovviamente *controllata* in **P**: tutto quello che si deve fare è assicurarsi che ciascun poliziotto si sia spostato su un arco del grafo (qui ci viene in aiuto il fatto che l'insieme dei poliziotti è una n-tupla *ordinata*, ma anche se non lo fosse la complessità non aumenterebbe di troppo: sostanzialmente si tratterebbe di fare un matching bipartito...) e che il ladro non sia stato preso. Ciò che inficia la ragionevolezza della nostra codifica è la possibilità di trovare due posizioni *consecutive* rappresentate da stringhe di diversa lunghezza; tuttavia il lettore che sia in grado di rendersi conto di questa oziosa difficoltà è anche in grado di trovarne l'oziosa soluzione. □

Quindi:

Teorema 4.2.3. *Il gioco $C\&R$ è* **EXPTIME**-*completo.*

Il lettore può osservare che l'intera riduzione da G_5 a $C\&R$ aumenta la dimensione dell'istanza di G_5 che riceve in input di un fattore pari alla settima potenza. Senza difficoltà, questo può essere abbassato alla quarta potenza; infatti non è necessario vedere tutto il grafo G, nel quale compare il nodo $\star$, come un'istanza di $C\&R^{++}$ col nodo $\star$ sostituito da un nodo $\circ$: basta collegare (nella maniera opportuna) al grafo G *così com'è*, in luogo del nodo $\star$, una istanza di $C\&R^{++}$ costituita da un solo nodo $\circ$. Un'osservazione attenta dell'intera costruzione dovrebbe, poi, permettere di guadagnare un ulteriore ordine di grandezza.

Tuttavia, il problema di abbassare ulteriormente questo limite è, per me, aperto. Infatti, il nocciolo della questione è legato al problema di costruire un grafo che fornisca al ladro un rifugio sicuro contro un numero dato, k, di poliziotti. Seguendo la tecnica che si è vista nella riduzione di $C\&R^{++}$ a $C\&R$, è facile costruire un simile grafo di dimensione $O(k^3)$ (con $O(k^2)$ nodi), però non so se questo valore sia il minimo possibile (né, ai fini della nostra dimostrazione, ciò ha qualche rilevanza).

4.3 $C\&R$ è PSPACE-hard

Quello che vogliamo mostrare in questa sezione non è una banale conseguenza del teorema 4.2.3, bensì una costruzione fondamentale nella dimostrazione di **PSPACE**-hardness di $C\&R^*$.

L'idea è di ottenere una riduzione da QBF a $C\&R$. Data quindi una formula

$$\Phi = \forall v_1 \exists v_2 \forall v_3 \ldots \exists v_n \phi$$

con ϕ in forma normale congiuntiva, costruiremo una posizione di $C\&R$ nella quale P dispone di una strategia vincente se e solo se Φ è vera.

Il grafo, G_Φ, che si assume il compito di simulare la formula Φ sarà costituito da un insieme di *piste* tutte della medesima lunghezza, ciascuna delle quali termina col nodo $\star$. All'inizio di una di queste piste si troverà il ladro, all'inizio delle altre dei poliziotti. Se la formula è falsa, vedremo che il ladro potrà seguire la propria pista fino al nodo $\star$ incolumemente; tuttavia, se Φ è vera, i poliziotti riusciranno a catturare il ladro *in extremis* appena prima che guadagni il nodo $\star$, e con esso l'impunità. Il trucco soggiacente si basa sul fatto che nel nostro grafo ha luogo una specie di *corsa* verso $\star$: se uno dei due giocatori *si attarda* retrocedendo per una sola mossa, allora viene inevitabilmente sconfitto.

G_Φ è diviso in due parti corrispondenti alla stringa di quantificatori $\forall v_1 \exists v_2 \forall v_3 \ldots \exists v_n$ la prima, ed alla formula ϕ la seconda. Dal momento che, come abbiamo preannunciato, i nostri uomini si muoveranno su piste tutte della medesima lunghezza, daremo al nostro grafo una struttura parzialmente ordinata in modo lineare. Per questo diremo che al livello 1 il ladro sceglie il valore di v_1, al livello 2 P sceglie il valore di v_2, etc. Al livello $n+1$ inizia la seconda parte del grafo (che non dura poi troppo). Al livello $n+3$ si trova il nodo $\star$. Il fatto importante è che i nodi di ciascun livello sono collegati solo fra loro, con quelli del livello precedente e con quelli del successivo; ciò ci garantisce che non possano esserci *scorciatoie*.

In figura 4.1 vediamo una sezione del grafo G_Φ, precisamente dal livello $j-2$ fino al livello $j+4$ (con j un numero intero dispari); vi è rappresentata la pista del ladro (quella centrale) e le due piste del j-esimo (a destra) e del $(j+1)$-esimo (a sinistra) poliziotto. I nodi a_i^j e b_i^j, se le cose si svolgono correttamente, non dovrebbero mai essere occupati né dai poliziotti, né dal ladro; al termine della sequenza di questi nodi si trova il nodo $\star$.

Supponiamo che il ladro muova per primo e che entrambi i giocatori *si comportino correttamente*, entrambi precipitando i loro uomini giù per le loro piste senza farli fermare o arretrare. Useremo le sequenze di nodi V_i^k e F_i^k per memorizzare il valore di v_k: se il k-esimo poliziotto (un poliziotto) si trova in uno dei nodi V_i^k, allora $v_k = 1$, se si trova in uno degli F_i^k, allora $v_k = 0$.

Alla $(j-1)$-esima mossa (come sempre a gioco corretto) il ladro arriverà

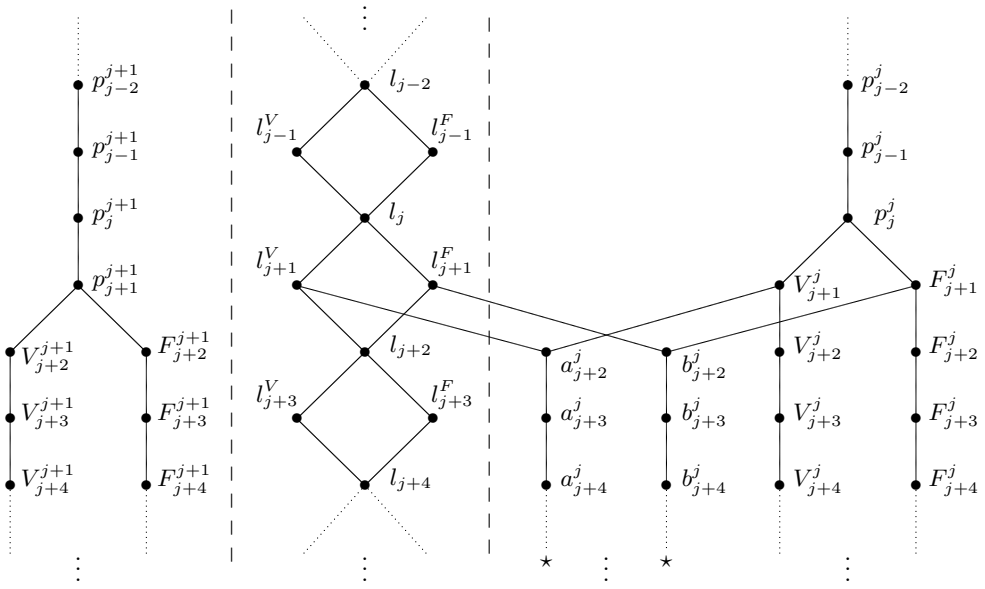

Figura 4.1: Una sezione di G_Φ

nel nodo l_j ed il j-esimo poliziotto nel nodo p^j_j. L'iniziativa è al ladro che può decidere di muoversi in l^V_{j+1} o l^F_{j+1}. Così facendo, però, L minaccia di guadagnare il nodo $\star$ tramite uno dei nodi a^j_{j+2} e b^j_{j+2}; quindi P deve prendere le contromisure spostando il j-esimo poliziotto da p^j_j a V^j_{j+1} o F^j_{j+1} secondoché il ladro si sia mosso in l^V_{j+1} oppure l^F_{j+1}. Alla mossa successiva il ladro si sposterà in l_{j+2} ed il j-esimo poliziotto in V^j_{j+2} o F^j_{j+2} (infatti sarebbe inutile muoversi verso i nodi a^j_{j+2} e b^j_{j+2} che il ladro non raggiungerà più). In questo modo L ha *deciso* il valore della j-esima variabile.

Intanto alla j-esima mossa il $(j+1)$-esimo poliziotto ha raggiunto il nodo p^{j+1}_{j+1}, ed alla $(j+1)$-esima si è spostato in V^{j+1}_{j+2} o F^{j+1}_{j+2} così decidendo il valore della $(j+1)$-esima variabile.

Ora passiamo alla descrizione del fatidico livello $n+2$, che decide la sorte del ladro. Siamo alla fine della mossa n, il ladro si trova in l_{n+1} e ciascun poliziotto nel suo V^i_{n+1} o F^i_{n+1}. La formula ϕ è composta da m clausole, ciascuna delle quali è la disgiunzione di un certo sottoinsieme delle variabili v_i e delle loro negazioni $\neg v_i$. L'$(n+2)$-esimo livello è costituito da m nodi (uno per ogni clausola), tutti collegati al nodo l_{n+1} da un arco. Per ogni clausola, il nodo corrispondente è collegato a V^i_{n+1} quando vi compare v_i, ed a F^i_{n+1} quando vi compare $\neg v_i$. Tutti questi m nodi sono infine collegati al nodo $\star$ che costituisce il livello $n+3$.

Se alla $(n+1)$-esima mossa almeno una delle clausole di ϕ è falsa (ϕ è

falsa), allora il corrispondente nodo del livello $n+2$ non sarà minacciato da alcun poliziotto ed il ladro vi potrà entrare con tranquillità, per poi raggiungere $\star$ alla mossa successiva. Se, al contrario, ϕ è vera, allora tutti gli m nodi del livello $n+2$ saranno minacciati ed il ladro verrà catturato alla mossa successiva, qualunque via egli scelga (a patto che non decida di contravvenire al gioco corretto restando fermo o arretrando).

Con questo la simulazione è conclusa, a patto che il nostro assunto sul *gioco corretto* sia verificato. Per assicurarci che l'assunto sia verificato diamo un'occhiata ai *blocchi di partenza* della nostra gara verso $\star$. A questo punto della costruzione abbiamo, al livello 1 (il primo livello di G_Φ) gli $n+1$ nodi $l_1, p_1^1, p_1^2, \ldots, p_1^n$. Nel nodo l_1 si trova inizialmente il ladro e nei nodi $p_1^1, p_1^2, \ldots, p_1^n$ i poliziotti. La prima mossa è al ladro (ma se volessimo darla ai poliziotti ci basterebbe farli partire *con un livello di ritardo* da $p_0^1, p_0^2, \ldots, p_0^n$). Onde assicurarci che il ladro giochi correttamnete ci basta aggiungere una nodo l_0 collegato ad l_1 da un arco ed inizialmente abitato da due *poliziotti di Damocle*.

Dimostriamo che il nostro espediente ha effetto.

- *Se il ladro dispone di una strategia vincente a gioco corretto*, allora non deve fare altro che precipitarsi giù per la sua pista seguendo la strategia vincente. I poliziotti di Damocle non rappresentano per lui una minaccia, in quanto non potranno mai raggiungerlo; e se P devia dal gioco corretto, L continua a seguire la strategia immaginando che i poliziotti ritardatari non esistono, o che siano in una posizione *corretta* qualunque, semplicemente questi poliziotti non rappresentano più una minaccia.

- *Se il ladro non dispone di una strategia vincente a gioco corretto*, allora P deve seguire la *sua* strategia vincente. I due poliziotti di Damocle saranno sufficienti ad impedire che il ladro retroceda o si fermi, così obbligandolo al gioco corretto.

Dal momento che questa osservazione ci sarà utile in seguito, notiamo che non è necessario che i due poliziotti di Damocle tallonino il ladro così da vicino: svolgerebbero in modo soddisfacente il loro compito anche seguendo il ladro con 2 livelli di distacco (partendo da l_{-1}). Le strategie, in questo caso sono le seguenti.

- *Se il ladro dispone di una strategia vincente a gioco corretto*: idem.

- *Se il ladro non dispone di una strategia vincente a gioco corretto*, allora P deve seguire la sua strategia vincente, ad ogni mossa abbassando di un livello i poliziotti di Damocle. Se il ladro contravviene al gioco corretto passando un turno, allora P porta i due poliziotti di Damocle ad un livello dal ladro lasciando tutti gli altri poliziotti fermi; in

questo modo P si riconduce alla situazione del caso precedente. Se il ladro contravviene al gioco corretto arretrando, viene preso alla mossa successiva.

4.3.1 Complessità della riduzione di QBF a $C\&R$

Come al solito si tratta di verificare che le nostre riduzioni si possano eseguire in **LOGSPACE**. Questa è tuttavia una questione di routine che non presenta difficoltà: il lettore codifichi pedestremente la numerazione che abbiamo dato ai nodi di G_Φ durante la sezione precedente e, con questo, otterrà un algoritmo in **LOGSPACE** per la riduzione. Il lettore interessato può osservare che la nostra riduzione provoca un aumento *quadratico* della dimensione (lunghezza della codifica) dell'istanza del problema che riceve in input.

4.4 $C\&R^*$ è PSPACE-hard

In questa sezione forniremo una prova della **PSPACE**-hardness di $C\&R^*$ servendoci della riduzione esposta nella sezione precedente. In realtà dimostreremo la **PSPACE**-hardness del gioco $C\&R^{++*}$, definito come il gioco $C\&R^{++}$ a cui è preposta una fase di due mosse nella quale i due giocatori decidono le loro posizioni iniziali (come in $C\&R^*$). Le considerazioni fatte per la riduzione da $C\&R^{++}$ a $C\&R$ valgono infatti per stabilire il seguente lemma:

Lemma 4.4.1. *$C\&R^{++*}$ si riduce in* **LOGSPACE** *a $C\&R^*$.*

Data una formula

$$\Phi = \forall v_1 \exists v_2 \forall v_3 \ldots \exists v_n \phi$$

con ϕ in forma normale congiuntiva, come nella sezione precedente, costruiremo un grafo G^*_Φ (con archi dei due tipi e nodi dei due tipi) costiuito da $2n + 4$ copie di G_Φ (tutte con archi di tipo $\diagup$ e nodi di tipo $\bullet$, cosicché si comportino come in $C\&R$) opportunamente collegate a due *meccanismi di fuga*. Quindi dimostreremo che se Φ è vera, P, con $n + 2$ poliziotti, riuscirà a riprodurre le *condizioni iniziali* di $C\&R$ in una delle $2n + 4$ copie di G_Φ ed a prendere il ladro; mentre, se Φ è falsa, il ladro potrà scappare in eterno, perpetuamente spostandosi dall'uno all'altro meccanismo di fuga. Ci sarà utile supporre, senza perdita della generalità, che Φ abbia almeno sette variabili (che ci siano almeno nove poliziotti in gioco), e che ϕ non sia la *formula vuota*.

4.4.1 Un complicato meccanismo di fuga...

Ciascuno dei due meccanismi di fuga avrà *nodi di ingresso* e *nodi di uscita*, questi ultimi ripartiti in *uscite per i poliziotti* ed *uscite per il ladro*. $n + 2$

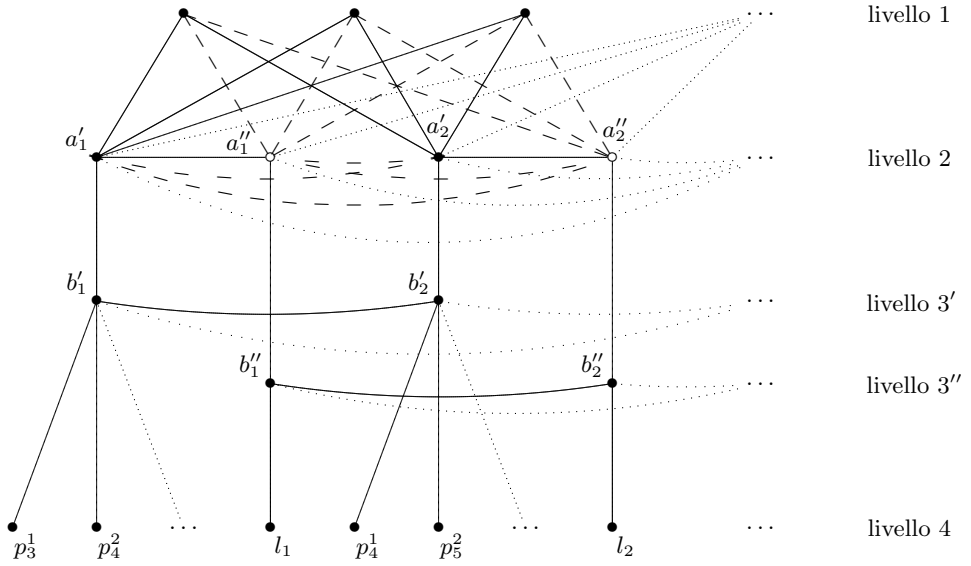

Figura 4.2: Un meccanismo di fuga

delle nostre $2n + 4$ copie di G_Φ avranno come nodi di partenza (quelli di primo livello) i nodi di uscita della prima via di fuga, e come nodi di arrivo (il livello $n + 2$) i nodi di ingresso della seconda via di fuga. Le altre $n + 2$ saranno disposte simmetricamente dalla seconda via di fuga alla prima.

In figura 4.2 è rappresentato un meccanismo di fuga. I nodi di ingresso costituiscono il livello 1, i nodi di uscita il livello 4. I nodi del livello 2 sono tutti collegati fra loro con archi di tipo ⌣, mentre i nodi del livello $3'$ ed i nodi del livello $3''$ sono collegati fra loro con archi di tipo ⟋.

Chiamiamo l_1 il nodo di partenza per il ladro della prima copia di G_Φ, e $p^1_1, \ldots, p^n_1$ i nodi di partenza dei poliziotti (senza contare quello destinato ai poliziotti di Damocle); per la seconda copia usiamo i nomi l_2 e $p^1_2, \ldots, p^n_2$; etc. Allora, ciascuno dei nodi del livello $3'$, b'_i, è collegato ai nodi $p_j + i + 1^j$ (dove apici e pedici sono valutati modulo $n + 2$); mentre i nodi b''_i del livello $3''$ sono collegati ai corrispondenti l_i.

4.4.2 ...istruzioni per l'uso

Consideriamo, ora, l'intero grafo G^*_Φ costituito dai due meccanismi di fuga e dalle $2n + 4$ copie di G_Φ, e dimostriamo che $n + 2$ possono catturarvi un ladro se e solo se Φ è vera.

• **se Φ è falsa** Questo è il caso più semplice. Osserviamo che ciascun poliziotto può minacciare al più uno dei nodi a''_i di ciascun meccanismo di

fuga, e siccome questi sono in totale $2n + 4$ ($n + 2$ per ciascun meccanismo di fuga) ce ne sarà almeno uno inizialmente sicuro. La strategia del ladro è, quindi, di posizionarsi inizialmente in questo nodo.

Ad ogni mossa, se uno dei nodi a_i'' del meccanismo di fuga prescelto (che d'ora in poi chiameremo *il primo*) è ancora sicuro, il ladro vi si sposta (o vi resta). Affinché tutti gli $n + 2$ nodi a_i'' siano minacciati, è necessario che tutti gli $n + 2$ poliziotti si trovino nei nodi a_i' e b_i''. Se si verifica questa circostanza ed i poliziotti non si trovano tutti nei nodi a_i' (ossia se qualcuno dei poliziotti è in uno dei b_i''), allora almeno uno dei nodi a_i' non è minacciato ed il ladro vi si sposta; in questo caso egli farà ritorno in un nodo a_i'' libero appena possibile.

La suddetta strategia permette al ladro di non lasciare il meccanismo di fuga nel quale si trova finché gli a_i' non siano occupati ciascuno da un poliziotto. Quando ha luogo questa circostanza, il ladro deve trovarsi in uno dei nodi a_i'' (infatti il ladro coscenzioso, che abbia seguito la nostra strategia, si troverà in uno dei nodi a_i' solo quando almeno un poliziotto è in uno dei nodi b_i'', dal quale non si può spostare direttamente in uno degli a_i'), dal quale è ora costretto a fuggire in b_i''.

Ora *l'idea del ladro* è di correre a rotta di collo verso l_i, entrare nella i-esima copia di G_Φ e, tramite questa, raggiungere il secondo meccanismo di fuga (che, ora come ora, è deserto). È chiaro che i poliziotti non possono contrastare questa strategia uscendo dal meccanismo di fuga tramite i nodi del livello 1, percorrendo a ritroso una delle copie di G_Φ, e raggiungendo il ladro nel secondo meccanismo di fuga; poiché quelli che scegliessero questa via raggiungerebbero il livello 3 del secondo meccanismo di fuga dopo che il ladro ha raggiunto il livello 1, dal quale può mettere in pratica la strategia descritta nei capoversi precedenti.

Per altro, quei poliziotti che si attardassero, rimanendo indietro rispetto al ladro nella gerarchia dei livelli, sarebbero inutili per contrastare questa strategia (precisamente come si è visto che avviene all'interno di G_Φ). Ne segue che, per quanto riguarda la nostra strategia di fuga, è come se i collegamenti *orizzontali* fra i nodi dello stesso livello non esistessero (poiché percorrerli significa, per i poliziotti, perdere un tempo).

Quindi al più un poliziotto può raggiungere ciascun nodo di partenza della i-esima copia di G_Φ (quella nella quale il ladro si è gettato). Come prima, entrare in un'altra copia di G_Φ sarebbe inutile, poiché i poliziotti che tentassero questa via arriverebbero, è vero, contemporaneamente al ladro al livello 1 del secondo meccanismo di fuga, ma in nodi di ingresso distinti.

Ne segue che, se i poliziotti seguono la migliore strategia possibile per contrastare quella del ladro, allora si generano nei nodi di partenza della i-esima copia di G_Φ le condizioni iniziali perché il ladro raggiunga uno dei nodi di arrivo della medesima (cioè uno dei nodi di ingresso del secondo meccanismo di fuga) incolumemente; mentre tutto ciò che avviene al di

fuori di questa copia di G_Φ è ininfluente finché il ladro non avrà raggiunto uno dei nodi del livello 1 del secondo meccanismo di fuga.

A questo punto nessuno dei poliziotti avrà avuto tempo di occupare i nodi del livello 2, né avrà preso il ladro. Ne segue che il suddetto potrà riprendere questa nostra strategia dall'inizio e raggiungere uno dei nodi a''_i non minacciato, oppure, se questi sono tutti minacciati, uno degli a'_i.

• **se Φ è vera** In questo caso la strategia di P si divide in due fasi: dapprima egli riesce a forzare la *posizione iniziale* in cui ciascuno dei nodi a'_i di un certo meccanismo di fuga è occupato da un poliziotto ed il ladro si trova in uno degli a''_i, quindi a partire da questa obbliga il ladro a giocare una partita persa su una delle copie di G_Φ.

Descriviamo innanzitutto la prima fase. Da principio, P dispone tre poliziotti nei nodi b'_1, b''_2 ed a'_3 di ciascun meccanismo di fuga. Questi bastano a controllare ciascuno dei nodi di ingresso e dei nodi di uscita di ciascun meccanismo di fuga; quindi, finché mantengono le loro posizioni, dal punto di vista del ladro, dividono il grafo in $2n + 6$ sottografi separati. Infatti ciascuna delle $2n+4$ copie di G_Φ e i due livelli 2 dei due meccanismi di fuga non sono collegati se non dai nodi di ingresso e di uscita minacciati.

A questo punto, se il ladro si trova in una delle copie di G_Φ può essere catturato da tre poliziotti seguendo la strategia seguente. Due dei tre poliziotti percorrono la pista del ladro finché non si trovano allo stesso livello del ladro; dal momento in cui questa condizione è verificata, essi continuano a seguire il ladro rimanendo sempre al suo medesimo livello. Essi sono chiaramente sufficienti a garantire che il ladro non possa entrare nella pista del ladro (egli dovrà, allora, rimanere in una delle piste dei poliziotti). Tolta la pista del ladro, però, le piste dei poliziotti sono tutte disconnesse fra di loro, ed hanno tutte la forma di grafi ad albero; quindi un poliziotto sarà più che sufficiente a catturare il ladro su una di esse. (È in questo ragionamento che abbiamo sfruttato il fatto di avere a disposizione almeno nove poliziotti.)

Quindi il ladro, se non vuole essere catturato, deve trovarsi nel livello 2 di uno dei meccanismi di fuga. Ora, perché il ladro non possa lasciare detto livello, sono sufficienti i poliziotti in b'_1, b''_2 ed a'_3 di quel meccanismo di fuga. Supponiamo, quindi, che tutti gli altri poliziotti si spostino nei nodi $a'_4, a'_5, \ldots, a'_{n+2}$ del nostro meccanismo di fuga (quello nel quale c'è il ladro, d'ora in poi l'altro non ci interesserà più).

Ci troviamo quindi in questa posizione: i poliziotti sono in $b'_1, b''_2, a'_3, a'_4, \ldots, a'_{n+2}$ ed il ladro in a''_1 oppure a'_2, poiché questi sono gli unici due nodi non minacciati del livello 2. Le successioni di mosse, forzate per il ladro, che portano alla suddetta *posizione iniziale* sono le seguenti:

- **se il ladro si trova in** a''_1, il poliziotto in b'_1 si sposta in a'_1 **-** il ladro si sposta in a'_2 **-** il poliziotto in b''_2 si sposta in a''_2 e il poliziotto in a'_1 torna in b'_1 **-** il ladro si sposta in a''_1 o a''_2 **-** il poliziotto in b'_1 si sposta

in a'_1 e il poliziotto in a''_2 si sposta in a'_2, in questo modo la *posizione iniziale* è stata ottenuta;

- **se il ladro si trova in** a'_2, il poliziotto in b''_2 si sposta in a''_2 **-** il ladro si sposta in a''_1 o a''_2 **-** il poliziotto in b'_1 si sposta in a'_1 e il poliziotto in a''_2 si sposta in a'_2, in questo modo la *posizione iniziale* è stata ottenuta.

Ora inizia la seconda fase. Tutti gli a'_i sono occupati da un poliziotto ed il ladro si trova, poniamo, in a''_2; la mossa è al ladro, che deve portarsi in b''_2. I poliziotti in $a'_3, a'_4, \ldots, a'_{n+2}$, allora, si spostano in $b'_3, b'_4, \ldots, b'_{n+2}$, il poliziotto in a'_2 rimane fermo, e quello in a'_1 si porta in a''_1.

Se, a questo punto, il ladro si muove verso l_2, allora i poliziotti in $b'_3, b'_4, \ldots, b'_{n+2}$ si spostano in $p_2^n, p_2^{n-1}, \ldots, p_2^1$, il poliziotto in a'_2 si sposta in a''_2, e quello in a''_1 va in b''_1. Ora il ladro è nel nodo di partenza della seconda copia di G_Φ, mentre tutti gli altri nodi di partenza di questa stessa copia sono occupati dai poliziotti in $p_2^n, p_2^{n-1}, \ldots, p_2^1$; contemporaneamente i due poliziotti in a''_2 e b''_1 fungono da poliziotti di Damocle. Ne segue che, a partire dalla posizione nella quale ora ci troviamo, P può forzare la cattura del ladro prima che esca dalla seconda copia di G_Φ (siccome Φ è vera).

Quindi il ladro, da b''_2, non si può dirigere verso l_2. Il nodo a''_2 è ancora controllato dal poliziotto in a'_2; pertanto il ladro non può far altro che spostarsi in un altro dei nodi b''_i o stare fermo. Supponiamo che, dopo la sua mossa, il ladro si trovi in b''_j; allora i poliziotti in $b'_3, b'_4, \ldots, b'_{n+2}$ si spostano in $b'_{j+2}, b'_{j+3}, \ldots, b'_{j+n+1}$ (dove i pedici sono, come al solito, valutati modulo $n+2$), il poliziotto in a'_2 si sposta verso a'_j, ed il poliziotto in a''_2 raggiunge b''_2. A questo punto il poliziotto in b''_2 minaccia tutti i nodi b''_i, ed il ladro non può arretrare poiché il nodo a''_j è controllato dal poliziotto in a'_j; egli deve quindi spostarsi in l_j.

Ora, finalmente, i poliziotti in $b'_{j+2}, b'_{j+3}, \ldots, b'_{j+n+1}$ si spostano verso $p_j^n, p_j^{n-1}, \ldots, p_j^1$, il poliziotto in a'_j va in a''_j, ed il poliziotto in b''_2 resta fermo. In questo modo ci troviamo nuovamente nella situazione già esaminata nella quale L è costretto a giocare una partita persa all'interno della j-esima (era la seconda) copia di G_Φ.

4.4.3 Complessità della riduzione di QBF a $C\&R^*$

Ci rimane il triste compito di verificare la **LOGSPACE**-fattibilità (che insisto ad elencare per pura velleità di completezza). Compito triste in quanto reso sempre più banale dalla semplicità del grafo in figura 4.2 e dall'abitudine dal lettore a svolgerlo.

Terminata questa verifica (che sia portata a termine dal lettore, da me, o da *Nessuno* poco importa), possiamo enunciare quest'ultimo

Teorema 4.4.2. *Il gioco* $C\&R^*$ *è* **PSPACE**-*hard.*

Volendo valutare, poi, la perdita implicata nella nostra catena di riduzioni da una istanza di QBF fino alla corrispondente di $C\&R^*$, otterremmo che si tratta di una ventunesima potenza. Ovviamente, facendo le cose con attenzione, questa può essere notevolmente abbassata. Tuttavia, io non credo che la tecnica sopra esposta possa portare al di sotto della undicesima potenza; né credo che *questo* possa avere *qualche* interesse per *qualcuno*.

Bibliografia

[1] M. Aigner and M. Fromme. A game of cops and robbers. *Discrete Applied Mathematics*, 8:1–12, 1984.

[2] T. Andreae. Note on a pursuit game played on graphs. *Discrete Mathematics*, 9:111–115, 1984.

[3] T. Andreae. On a pursuit game played on graphs for wich a minor is excluded. *Journal of Combinatorial Theory (Series B)*, 41:37–41, 1986.

[4] J. L. Balcázar, J. Díaz, and G. J. *Structural complexity (2 vol.)*. Springer Verlag, Berlin, 1988.

[5] A. Berarducci and B. Intrigila. On the cop number of a graph. *Advances in Applied Mathematics*, 14:389–403, 1993.

[6] E. R. Berlekamp, J. H. Conway, and G. R. K. *Winning ways, for your mathematical plays (2 vol.)*. Academic Press, London, 1982.

[7] G. Brightwell and P. Winkler. Gibbs measures and dismantlable graphs. *Journal of Combinatorial Theory (Series B)*, 78, 2000.

[8] J. H. Conway. *On Numbers and Games*. London mathematical society monographs. Academic Press, London, 1976.

[9] S. Fitzpatrik and R. Nowakowski. Copnumber of graphs with strong isometric dimension two. *ARS COMBINATORIA*, 59:65–73, 2001.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the Theory of* **NP***-Completeness*. Freeman & Co., New York, 1979.

[11] A. Goldstein. Unbounded unimodal search and pursuit problems. Master's thesis, University of Illinois, 1993.

[12] J. E. Hopcroft and U. J. D. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

[13] S. Neufeld and N. R. A game of cops and robbers played on products of graphs. *Discrete Mathematics*, 186:253–268, 1998.

[14] R. Nowakowski and W. P. Vertex-to-vertex pursuit in a graph. *Discrete Mathematics*, 43:235–239, 1983.

[15] R. J. Nowakowski. *Games of no chance: combinatorial games at MSRI, 1994*. Cambridge University Press, Cambridge, 1996.

[16] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[17] B. Poizat. *Les petits cailloux, une approche modèle-théorique de l'algorithmie*. Nul al-Mantiq wal-Ma'rifah. Aleas, Lyon, 1995.

[18] L. J. Stockmayer and C. A. K. Provably difficult combinatorial games. *SIAM, Journal on computing*, 8:151–174, 1979.