Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Triennale in Matematica

Il teorema di Cook-Levin e i SAT-Solver

Tesi di Laurea in Informatica

Relatore:
Prof
Simone Martini

Presentata da:
Fabio Le Piane

I Sessione
Anno Accademico 2014/15

Indice

Introduzione

In questa tesi tratteremo il tema della soddisfacibilità booleana o proposizionale, detta anche **SAT**, ovvero il problema di determinare se una formula booleana è soddisfacibile o meno. Soddisfacibile significa che è possibile assegnare le variabili in modo che la formula assuma il valore di verità *vero*; viceversa si dice insoddisfacibile se tale assegnamento non esiste e se quindi la formula esprime una funzione identicamente falsa.
A tal fine introdurremo degli strumenti preliminari che ci permetteranno di affrontare più approfonditamente la questione: partiremo dalla definizione basilare di macchina di Turing, affronteremo poi le classi di complessità e la riduzione. Vedremo poi la nozione di **NP**-completezza e dimostreremo quindi che **SAT** è un problema **NP**-completo. Infine daremo una definizione generale di **SAT**-solver e discuteremo due dei principali algoritmi utilizzati a tale scopo.

Capitolo 1

La teoria della calcolabilità e la Macchina di Turing

1.1 L'idea intuitiva di algoritmo

Il concetto di *algoritmo* è l'elemento centrale della teoria della calcolabilità. Il punto di partenza di questa teoria è infatti l'esigenza di formalizzare l'idea intuitiva di funzione *calcolabile* tramite un algoritmo, ovvero di *funzione algoritmica*.
Informalmente, possiamo dire che un algoritmo è un procedimento di calcolo che consente di pervenire alla soluzione di un problema, numerico o simbolico, mediante una sequenza finita di operazioni, completamente e univocamente determinate. In altri termini, un algoritmo consiste in una serie di istruzioni (operazioni) la cui esecuzione consente di "trasformare" l'insieme finito di dati simbolici che descrivono un problema nella soluzione del problema stesso.
Elenchiamo di seguito le caratteristiche essenziali della nozione informale di algoritmo:

- l'insieme di istruzioni che definisce l'algoritmo è finito;
- l'insieme delle informazioni che rappresentano il problema e i requisiti richiesti alla sua soluzione hanno una descrizione finita;
- esiste un agente di calcolo in grado di eseguire le istruzioni;
- il procedimento di calcolo, o *computazione*, è suddiviso in passi discreti e non fa uso di dispositivi analogici;
- la computazione è *deterministica*, ossia la sequenza dei passi computazionali è determinata senza alcuna ambiguità.

In conclusione, possiamo afermare che una procedura algoritmica riceve in ingresso una descrizione finita dei dati del problema e restituisce, dopo un tempo finito, una descrizione finita del risultato. La natura deterministica della procedura fa sì che l'algoritmo fornisca sempre lo stesso risultato ogni volta che riceve in ingresso gli stessi dati. In questo modo, un algoritmo stabilisce una reazione di tipo funzionale tra l'insieme dei dati e quello dei risultati.
Nell'analisi che segue ci si limiterà, senza perdita di generalità, a considerare algoritmi

che ricevono in ingresso k valori interi e restituiscono una soluzione data da j numeri interi, con $k, j \geq 1$. Ai fini di tale analisi, un algoritmo può quindi essere visto come una procedura per il *calcolo* della funzione $f : \mathbb{N}^k \to \mathbb{N}^j$ che associa ad ogni insieme di dati il risultato ad essi corrispondente.
È molto importante, oltre che naturale, operare una distinzione tra il concetto di algoritmo e quello di funzione calcolabile da un algoritmo. Per il calcolo di una funzione (o la risoluzione di un problema) si possono infatti definire infiniti algoritmi, diversi tra loro e più o meno efficienti.
La nozione di *funzione calcolabile da un algoritmo*, o *funzione algoritmica*, è formalizzata dal concetto (definito più avanti) di *funzione ricorsiva*.

1.2 Le funzioni primitive ricorsive

Nella prima metà del ventesimo secolo, vennero avanzate alcune proposte per identificare il concetto intuitivo di funzione algoritmica con nozioni formali più precise. Uno dei primi tentativi di formalizzazione portò alla definizione della classe delle *funzioni primitive ricorsive*. L'idea alla base di questa formulazione è di individuare alcune funzioni algoritmiche estremamente semplici e di chiudere la classe rispetto ad opportune operazioni, la *composizione* e la *ricorsione primitiva*, nella speranza di ottenere così una controparte formale delle funzioni algoritmiche. Come vedremo, l'operazione di ricorsione primitiva definisce una funzione f in termini di sé stessa. Formalmente:

Definizione 1.1. La classe delle funzioni primitive ricorsive, che denotiamo con C, è la più piccola classe di funzioni da $\mathbb{N}^k$ a $\mathbb{N}^j$ tale che:

1. tutte le funzioni costanti appartengono a C;

2. la funzione successore, ossia $S(x) = x + 1$, appartiene a C;

3. le funzioni identità, ossia $I_j(x_1, \ldots, x_n) = x_j$, $j = 1, \ldots, n$, appartengono a C;

4. se f è una funzione di k variabili in C e $g_1, \ldots, g_k$ sono funzioni di m variabili in C, allora la funzione

$$h(x_1, \ldots, x_m) = f(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))$$

   appartiene a C (operazione di composizione);

5. se g e h sono due funzioni in C rispettivamente di $k-1$ e $k+1$ variabili, allora l'unica funzione f di k variabili per cui

$$f(0, x_2, \ldots, x_k) = g(x_2, \ldots, x_k)$$
$$f(y+1, x_2, \ldots, x_k) = h(y, f(y, x_2, \ldots, x_k), x_2, \ldots, x_k)$$

   appartiene a C (operazione di ricorsione primitiva)

Questo modo di procedere consente di definire funzioni sempre più complesse a partire da funzioni più semplici (le funzioni di base) già disponibili.
Dato che sia le funzioni di base sia gli operatori sono definiti in modo costruttivo, la definizione della classe delle funzioni primitive ricorsive indica anche come queste funzioni possono essere calcolate. In particolare, possiamo affermare che una funzione f è primitiva ricorsiva se e solo se esiste una sequenza finita di funzioni $f_1, f_2, \ldots, f_n = f$ e per ogni $j \leq n$

- $f_j \in C$ (per una tra le proprietà 1,2,3), oppure
- f_j si può ricavare direttamente dalle funzioni f_i, $i \leq j$, per mezzo delle operazioni di composizione o di ricorsione primitiva

Una descrizione di questa sequenza, insieme alla specifica di come sia possibile ottenere ciascuna $f_i, i \leq n$, è detta *derivazione* di f.
$\grave{E}$ inoltre importante notare che

Proposizione 1.2. *Ogni funione primitiva ricorsiva è totale (cioè è definita su tutti gli argomenti).*

A differenza di quanto possa sembrare a prima vista, la classe delle funzioni primitive ricorsive è tutt'altro che limitata, e in realtà tutte le funzioni di interesse sono primitive ricorsive.
Viene dunque immediato chiedersi se le funzioni primitive ricorsive esauriscano la classe delle funzioni calcolabili da un algoritmo, cioè se esse possano costituire la controparte formale della nozione intuitiva di funzione algoritmica. A questa domanda si può rispondere immediatamente in senso negativo, esibendo funzioni chiaramente calcolabili ma non primitive ricorsive. Un esempio celebre è la funzione di Ackermann:

Definizione 1.3. La funzione di Ackermann è una funzione A di tre variabili tale che:

$$A\,(0, x, y) = x + 1,$$

$$A\,(1, x, y) = x + y,$$

$$A\,(2, x, y) = x \cdot y,$$

e per $n \geq 2$

$$A(n+1, x, y) = \begin{cases} 1 & \text{se } y = 0 \\ A(n, x, A(n+1, x, y-1)) & \text{se } y > 0 \end{cases}$$

Introduciamo ora il metodo di *diagonalizzazione*, che è un metodo generale che può essere utilizzato, ad esempio, per dimostrare l'esistenza di funzioni che non sono primitive ricorsive, o di funzioni non calcolabili tramite algoritmi. Data una classe di funzioni C, questa tecnica consiste infatti nel costruire una funzione f che ha la proprietà di differire, almeno su un argomento, da ciascuna funzione della classe C. La funzione f così costruita non può quindi appartenere a C.
Descriveremo la tecnica di diagonalizzazione tramite un esempio basato sulla classe

delle funzioni primitive ricorsive.
Consideriamo tutte le possibili derivazioni di funzioni primitive ricorsive e supponiamo di avere a disposizione un alfabeto che ci permette di *codificare* tutte le derivazioni, ossia di associare ad ogni derivazione una stringa finita di *simboli base*. Questi simboli base dovrebbero includere un simbolo di funzione, diversi simboli per le variabili, le cifre per gli indici delle variabili, le cifre per gli ordinari numeri naturali, le parentesi, la virgola, i simboli "+" e "=", una serie di simboli speciali per indicare le costanti, le funzioni *successore* ed *identità* e la fine di linea.
Questo modo di procedere rende immediatamente disponibile una tecnica per determinare se una data stringa di simboli base costituisca o meno una derivazione primitiva ricorsiva legittima. Possiamo infatti elencare, in sequenza, tutte le posibili derivazioni primitive ricorsive, esaminando prima tutte le stringhe di lunghezza 1, poi tutte quelle di lunghezza 2 e così via. Questa lista è naturalmente infinita, ma al tempo stesso ciascuna derivazione può essere caratterizzata come un preciso elemento della lista.
Supponiamo ora di aver generato la lista che contiene tutte le derivazioni primitive ricorsive per le funzioni di una sola variabile. Indichiamo con Q_x la $(x+1)$-esima derivazione e con g_x la funzione determinata dalla derivazione Q_x. Consideriamo la funzione h definita nel seguente modo:

$$h(x) = g_x(x) + 1$$

La funzione h è chiaramente algoritmica: dato x, il valore $h(x)$ può essere ottenuto generando la lista delle derivazioni fino a Q_x, utilizzando Q_x per calcolare $g_x(x)$, e quindi aggiungendo 1.
D'altra parte, h non è una funzione primitiva ricorsiva. Infatti, se h fosse primitiva ricorsiva, allora dovrebbe esistere un indice x_0 tale che la derivazione Q_{x_0} calcoli h, ossia $h = g_{x_0}$, per un certo x_0. Ma allora avremmo anche:

$$g_{x_0}(x_0) = h(x_0) = g_{x_0} + 1,$$

che è una evidente contraddizione. Abbiamo così ottenuto un altro esempio di funzione algoritmica ma non primitiva ricorsiva.
È difficile pensare ad una caratterizzazione formale delle funzioni algoritmiche che sia in grado di sfuggire all'applicazione della tecnica di diagonalizzazione. Questa tecnica sembra infatti suggerire la possibilità che non esista una classe massimale di funzioni, formalmente caratterizzabile, che da sola riesca a catturare ed eprimere in modo soddisfacente la nozione informale di funzione algoritmica. Nelle fasi successive di questa tesi vedremo come questa intuizione sia sbagliata.
Un primo modo per cercare di risolvere il problema è quello di utilizzare le *funzioni parziali*, ossia le funzioni il cui dominio può non essere tutto $\mathbb{N}^k$. Richiamiamo qui di seguito le definizioni di funzione parziale e di funzione totale.

Definizione 1.4. • Una funzione parziale è una funzione che può non essere definita su tutti i suoi argomenti. Più precisamente, una funzione parziale $\varphi : \mathbb{N}^k \to \mathbb{N}^j$ assegna un valore $\varphi(x)$ solo agli elementi di un particolare sottoinsieme $DOM(\varphi) \subseteq \mathbb{N}^k$, detto *dominio di definizione*. Se $x \notin DOM(\varphi)$, diciamo che φ è *indefinita (o divergente)* su quell'argomento.

- Una funzione totale $f : \mathbb{N}^k \to \mathbb{N}^j$ assegna un valore $f(x) \in \mathbb{N}^j$ ad ogni $x \in \mathbb{N}^k$, ossia $DOM(f) = \mathbb{N}^k$.

Alla nozione di funzione parziale si associa quella di *funzione algoritmica parziale*.

Definizione 1.5. Una *funzione algoritmica parziale* è una funzione per il cui calcolo esiste un algoritmo che termina se l'input corrisponde ad un elemento del dominio di definizione della funzione, ma che altrimenti procederebbe indefinitamente nella computazione.

In altre parole, l'algoritmo fornisce un risultato se e quando questo può essere ottenuto; diversamente, l'algoritmo impiega un tempo infinito nella ricerca di esso. Come abbiamo già osservato, la classe delle funzioni parziali non è necessariamente soggetta alla contraddizione indotta dall'applicazione del metodo di diagonalizzazione. Vediamone la ragione.
Ossserviamo innanzitutto che ogni funzione parziale può essere specificata tramite una procedura di calcolo descritta da un insieme di istruzioni. Poiché non necessariamente tale procedura termina sempre fornendo un risultato, si usa il termine "funzione parziale" per indicare la corrispondenza tra input e output fornita dalla procedura.
Supponiamo ora di avere a disposizione un alfabeto di simboli base che ci consenta di codificare gli insieme di istruzioni che definiscono la procedura di calcolo associata ad una funzione parziale. Indichiamo con ψ_x la funzione parziale determinata dall'$(x+1)$-esimo insieme di istruzioni Q_x. Scegliamo x_0 in modo tale che ψ_{x_0} coincida con la funzione parziale φ definita dalle seguenti istruzioni:

- si determina Q_x;
- si calcola $\psi_x(x)$;
- se e quando si ottiene un valore per $\psi_x(x)$, allora si pone $\varphi(x) = \psi_x(x) + 1$

In questo caso, l'equazione

$$\psi_{x_0}(x_0) = \varphi(x_0) = \psi_{x_0}(x_0) + 1$$

non costituisce necessariamente una contraddizione, poiché φ potrebbe non essere definita in corrispondenza di x_0.
Abbiamo così evidenziato che la natura stessa delle funzioni parziali pone ostacoli all'applicazione del metodo di diagonalizzazione. Tali ostacoli potrebbero essere superati qualora fosse possibile definire una procedura algoritmica in grado di selezionare gli insiemi di istruzioni che definiscono esclusivamente funzioni totali. Tuttavia l'esistenza di una procedura algoritmica siffatta sembra estremamente improbabile.

1.3 La Macchina di Turing

Negli anni '30 Kleene, Church e Turing avanzarono, in maniera indipendente l'uno dall'altro, tre diverse caratterizzazioni di un'ampia classe di funzioni parziali e quindi, mediante la nozione di *modello di calcolo*, giunsero a tre definizioni formali del

concetto di *algoritmo* e dunque ad una formulazione precisa della nozione di *funzione parziale calcolabile da un algoritmo*. In seguito fu dimostrato che in realtà queste tre formalizzazioni coincidono. In virtù di ciò si potè scegliere come formalizzazione standard quella di Turing, poiché più semplice a livello concettuale e di più pratico utilizzo. L'idea del matematico inglese è basata su un modello di calcolo denominato appunto *Macchina di Turing* (d'ora in poi MdT). Si tratta di un modello concettualmente elementare che punta ad individuare le componenti elementari, dette *passi* di una computazione.
Informalmente una MdT è composta da un'unità di controllo e di un insieme di k nastri su cui agiscono altrettanti dispositivi mobili di lettura e scrittura denominati *testine*. I nastri sono finiti a sinistra e infiniti a destra ed ognuno di essi è suddiviso in *celle* che a loro volta possono contenere ognuna un unico carattere fra quelli appartenenti all'*alfabeto* della macchina; fra questi simboli ve ne è uno speciale detto *spazio* che indica l'assenza di informazione nella cella. Ad ogni istante solo una finita porzione dei nastri può essere composta da caselle contenenti un carattere diverso dallo spazio e le testine possono leggere e/o scrivere un solo carattere alla volta. L'organo di controllo può trovarsi in uno solo di un insieme finito di stati e la transizione tra un istante e l'altro avviene a istanti fissati (es: t_0, t_1, t_2 ecc). Inoltre, per distinguere fra celle utilizzate per la memorizzazione di informazioni di ingresso e quelle utilizzate per la memorizzazione di informazioni addizionali si suppone che un intero nastro sia addetto alla memorizzazione delle prime che sarà detto *nastro di ingresso* (solitamente tale nastro è il primo); su tale nastro sono possibili solo operazioni di lettura. In maniera analoga si definisce il *nastro di uscita*, sul quale potranno essere svolte solo operazioni di scrittura (di solito tale nastro è identificato con l'ultimo). Tutti gli altri nastri saranno chiamati *nastri di lavoro* (ovviamente su di essi si potranno svolgere sia operazioni di scrittura che di lettura).
Un passo della computazione della MdT consiste dunque della seguente sequenza di operazioni:

- L'unità di controllo modifica il proprio stato in funzione dello stato attuale e dei simboli letti dalle testine;
- Ogni testina, salvo quella del nastro d'ingresso, può scrivere un nuovo simbolo nella cella su cui è correntemente posizionata;
- Ogni testina si sposta di una posizione a destra o a sinistra oppure resta ferma.

A questo punto possiamo dare la seguente definizione formale:

Definizione 1.6. Una MdT a k nastri è una 7-pla $(Q, T, I, q_0, q_f, b, \delta)$, in cui

1. Q è l'insieme degli stati in cui può trovarsi l'unità di controllo;
2. T è l'alfabeto della macchina, ovvero l'insieme dei simboli che possono comparire nelle celle dei nastri, eccetto il nastro d'ingresso;
3. I è l'alfabeto di ingresso, ovvero l'insieme dei simboli che possono trovarsi sul nastro d'ingresso ed è un sottoinsieme proprio di T;

4. q_0 è lo stato iniziale, ovvero lo stato nel quale si trova l'unità di controllo all'istante t_0;

5. q_f è lo stato finale;

6. $b \in T/I$ è il carattere spazio;

7. δ è la funzione di transizione, ovvero la funzione che determina i cambiamenti di stato della macchina. Più precisamente:

$$\delta : Q \times T^{k-1} \longrightarrow Q \times \{D, S, F\} \times (T \times \{D, S, F\})^{k-1},$$

dove i simboli D ed S indicano rispettivamente lo spostamento a destra e a sinistra, mentre il simbolo F indica che la testina resta ferma.

Osservazione 1.7. La funzione di transizione di una MdT è a tutti gli effetti un algoritmo.

Lo stato della macchina ad ogni istante è determinato dallo stato del controllo, dal contenuto dei nastri e dalla posizione delle testine sui nastri.

Notazione 1.8. per evitare confusione fra stato della macchina nel suo complesso e stato del controllo il primo viene chiamato *configurazione* La descrizione formale della configurazione di una MdT ad un determinato istante è chiamata *descrizione istantanea* (in breve DI)

Formalmente:

Definizione 1.9. Una descrizione istantanea è una k-pla $(\alpha_1, \alpha_2, \ldots, \alpha_k)$, in cui ogni α_i è una stringa che rappresenta la situazione sull'i-esimo nastro e lo stato del controllo. Più precisamente, ogni α_i è della forma β_i q z_i λ_i , dove:

- $\beta_i \in T^*$ è la sequenza di simboli sull'i-esimo nastro a partire dall'estremità a sinistra fino alla posizione della i-esima testina;
- q è lo stato corrente del controllo;
- $z_i \in T$ $(z_1 \in I)$ è il simbolo correntemente letto dall'i-esima testina;
- $\lambda_i \in T^*$ $(\lambda_1 \in I^*)$ è la sequenza dei simboli sull'i-esimo nastro compresa fra il simbolo immediatamente alla destra della testina e l'ultimo simbolo diverso dallo spazio, a destra della testina.

Otteniamo poi la seguente definizione formale:

Definizione 1.10. Una computazione di una MdT è costituita da una sequenza di descrizioni istantanee $DI_0, DI_1, DI_2, \ldots$, tale che DI_0 è una descrizione iniziale e DI_{i+1} è il risultato dell'applicazione della funzione δ alla configurazione descritta da DI_i. Quest'ultimo fatto si indica usando la notazione $DI_i \vdash DI_{i+1}$. Se esiste $n \geq 1$ tale che

$$DI_0 \vdash DI_1 \vdash \cdots \vdash DI_n,$$

dove DI_n è una descrizione finale, si dice che la computazione converge, in caso contrario che diverge

Alle computazioni di una MdT può essere associato il calcolo di una funzione da (I^*) a $(T \setminus \{b\})^*$. A questo scopo consideriamo come risultato di una computazione convergente la stringa prodotta sul nastro di uscita a partire dalla prima cella a sinistra (inclusa) fino alla prima cella contenente il carattere spazio. Formalmente:

Definizione 1.11. Diciamo che una MdT M calcola la funzione $f : I^* \longrightarrow (T \setminus \{b\})^*$ se, in corrispondenza di un ingresso $z \in I^*$, si verifica quanto segue:

- se f è definita su z, allora esiste n per cui $DI_1 \vdash \cdots \vdash DI_n$, dove DI_n è una descrizione istantanea finale; inoltre la stringa $\omega \in (T/b)^*$ prodotta da M sul nastro di uscita soddisfa $f(z) = \omega$;
- se $f(z)$ non è definita, allora la computazione diverge.

In tal modo possiamo dire che le MdT si caratterizzano come dispositivi per la manipolazione di stringhe. A patto di considerare opportune modifiche possiamo associare alle computazioni di una MdT il calcolo di funzioni su domini diversi, ad esempio una MdT può operare su $\mathbb{N}$ semplicemente rappresentando i numeri naturali tramite stringhe di simboli dell'alfabeto di input. Ad esempio se l'alfabeto di input contiene i simboli 0 e 1 possiamo utilizzare la notazione binaria. Se indichiamo con $i(z)$ il numero naturale la cui rappresentazione binaria è data dalla stringa z, possiamo rappresentare la n-pla $(i(z_1), i(z_2), \ldots, i(z_n))$ mediante la stringa $z_1 \sharp z_2 \sharp \ldots \sharp z_n$, dove $\sharp$ è un simbolo dell'alfabeto di input utilizzato per separare gli elementi della n-pla. Si ottiene dunque la:

Definizione 1.12. Una MdT calcola la funzione parziale $\psi : \mathbb{N}^k \longrightarrow \mathbb{N}$ se, su input $z_1 \sharp z_2 \sharp \ldots \sharp z_n$ si verifica quanto segue:

- se $(i(z_1), i(z_2), \ldots, i(z_n)) \in \mathbb{N}^k$ appartiene al dominio di definizione di ψ, allora esiste n per cui
$$DI_0 \vdash DI_1 \vdash \cdots \vdash DI_n,$$
dove DI_n è una descrizione finale; inoltre la stringa ω prodotta da M sul nastro di uscita soddisfa $i(\omega) = \psi(i(z_1), i(z_2), \ldots, i(z_n))$;
- se $(i(z_1), i(z_2), \ldots, i(z_n)) \notin DOM(\psi)$, allora la computazione diverge.

La definizione precedente è particolarmente significativa perché individua chiaramente e con precisione la classe delle *funzioni algoritmiche parziali secondo Turing*, ovvero delle funzioni parziali calcolabili da una MdT (si vedrà in seguito che questo permette di definire in termini generali l'intera classe delle *funzioni parziali calcolabili da un algoritmo*.

1.4 La tesi di Church-Turing

Dopo aver parlato delle MdT introduciamo una importante tesi che ci permetterà di non dover trattare direttamente gli altri modelli di calcolo ideati successivamente: la tesi di Church-Turing.
Infatti apparentemente le varie classi di funzioni calcolabili definibili (secondo Turing,

Kleene, Church, Post, Markov) sembrano differenti, ma è in realtà possibile dimostrare che i diversi formalismi sono tra loro equivalenti, ovvero che ogni funzione calcolabile in un formalismo è calcolabile anche in ognuno degli altri purché i dati e i risultati siano adeguatamente codificati nelle strutture di dati su cui operano i diversi formalismi. In altri termini, tutti i modelli formali di calcolo proposti fino ad oggi descrivono algoritmi per il calcolo dello stesso inseme di funzioni: le funzioni parziali ricorsive.
Anziché soffermarsi sui dettagli della dimostrazione dell'equivalenza tra i diversi formalismi (che sfrutta tecniche di codifica piuttosto complesse) ci si concentrerà sull'approfondimento delle implicazioni di tale risultato, che costituisce uno dei fondamenti di tutta la teoria della calcolabilità.
Innanzitutto l'implicazione fondamentale dell'identificazione di modelli di calcolo formali apparentemente molto lontani è sicuramente quella di rendere molto plausibile la loro identificazione con il concetto stesso di *cacolabilità effettiva*. Andando ancora oltre: poiché l'evidenza indica che non esiste alcuna possibilità di definire un concetto di calcolabilità più ampio di quello di Turing o di tutti quelli ad esso equivalenti, sembra emergere il fatto che ogni funzione calcolabile sia calcolabile secondo Turing; quest'ultima asserzione è nota come *Tesi di Church-Turing*.
Precisamente, scriviamo:

Proposizione 1.13 (Tesi di Church-Turing)**.** *Una funzione calcolabile in un qualunque ragionevole modello di calcolo è calcolabile mediante una MdT*

L'importanza della tesi di Church-Turing sta nel fatto che essa permette di trasferire la proprietà di *calcolabilità* dai modelli alle funzioni, ossia di classificare le funzioni in calcolabili e non calcolabili, indipendentemente dal modello adottato per descrivere gli algoritmi per il calcolo di tali funzioni. Ciò conduce alla seguente definizione.

Definizione 1.14. Una funzione è calcolabile (o ricorsiva) se è calcolabile mediante una *MdT*.

Naturalmente la tesi di Church-Turing non è dimostrabile. Essa può essere solo accettata o rifiutata, poiché non si può escludere a priori che in futuro venga individuato uno strumento che permetta di calcolare una funzione al momento ritenuta non ricorsiva. In ogni modo vi sono ragioni abbastanza convincenti per credere che ciò non potrà accadere; ad esempio è possibile dimostrare direttamente per molti strumenti o modelli apparentemente più potenti della *MdT* che qualsiasi funzione da essi calcolabile è in realtà calcolabile anche da una *MdT*.

1.5 Enumerazione delle Macchine di Turing

Nelle sezioni precedenti abbiamo accennato alla coincidenza tra l'insieme delle funzioni parziali ricorsive e l'insieme delle funzioni calcolabili tramite macchine di Turing. Questo fatto dà una certa rilevanza alla testi di Church-Turing.
Diventa allora evidente come l'esistenza di funzioni non calcolabili segue da una semplice considerazione di cardinalità: le funzioni parziali ricorsive possono essere esplicitamente messe in corrispondenza con gli interi, ossia sono un insieme numerabile; le funzioni parziali hanno invece la potenza del continuo. Vediamo allora, per sommi

capi, come costruire la corrispondenza tra funzioni parziali ricorsive e numeri interi. Consideriamo una computazione su una *MdT* che calcola una funzione parziale ricorsiva. A partire da un opportuno alfabeto di simboli, possiamo definire un procedimento per la codifica di stati, caratteri, configurazioni della macchina e per la funzione di transizione. Tramite questo procedimento possiamo associare una stringa finita di simboli ad una qualsiasi computazione e generare una lista contenente tutte le computazioni eseguibili sulle *MdT*. Questo modo di procedere dà luogo ad una procedura algoritmica che associa ad ogni numero naturale x la computazione che occupa l'$(x+1)$-esimo posto nella lista.
Grazie a questa procedura possiamo introdurre il concetto di *numero di Gödel* di una computazione.

Definizione 1.15. Sia P_x la computazione associata al numero naturale x nella lista di tutte le computazioni. x è chiamato *numero di Gödel* di P_x. Sia inoltre $\varphi_x^{(k)}$ la funzione parziale ricorsiva di k variabili calcolata tramite la computazione P_x. x è anche detto *numero di Gödel* per $\varphi_x^{(k)}$.

La procedura di enumerazione delle computazioni fornisce sia un algoritmo per passare da un qualsiasi numero naturale x alla computazione P_x ad esso corrispondente, sia un algoritmo per identificare il numero di Gödel di x di una generica computazione P, vale a dire il numero naturale x tale che $P \equiv P_x$: è sufficiente generare la lista delle computazioni fino a quando si trova la computazione che corrisponde a P; la posizione di tale computazione nella lista definisce il numero di Gödel di P.
Si osservi però che la corrispondenza così costruita è di molti a uno: ad ogni funzione corrispondono infiniti numeri di Gödel poiché vi sono infiniti modi di alterare una computazione senza in realtà alterare la funzione calcolata (si supponda ad esempio di modificare la computazione in modo che essa esegua un numero arbitrario di volte la sequenza di operazioni $x = x + 1$; $x = x - 1$).
Come già anticipato, la corrispondenza con gli interi fornita dall'enumerazione di Gödel consente di dimostrare facilmente l'esistenza di funzioni non calcolabili.

Teorema 1.16. *1. Esiste un'infinità numerabile di funzioni ricorsive.*

2. Esiste un'infinità numerabile di funzioni parziali ricorsive.

Dimostrazione. 1. Dalla tesi di Church-Turing, segue che tutte le funzioni costanti sono ricorsive. Dunque le funzioni ricorsive sono almeno un'infinità numerabile. D'altro canto, il processo di enumerazione di Gödel garantisce che esiste al più un'infinità numerabile di funzioni ricorsive.

2. Per dimostrare che le funzioni parziali ricorsive sono almeno un'infinità numerabile è sufficiente osservare che le funzioni parziali ricorsive sono almeno tante quante le funzioni ricorsive. Infatti, una funzione ricorsiva può essere vista come una funzione parziale ricorsiva il cui dominio di definizione coincide con tutto $\mathbb{N}^k$ $(k \geq 1)$. Anche in questo caso, il processo di enumerazione di Gödel garantisce che esiste al più un'infinità numerabile di funzioni parziali ricorsive.

□

Dal teorema precedente discende poi il:

Teorema 1.17. *Esistono funzioni non ricorsive.*

La procedura di enumerazione delle *MdT* ci permette inoltre di introdurre una nozione fondamentale: quella di *funzione universale parziale*. L'idea è quella di definire una funzione che per ogni coppia (x, y) in ingresso restituisca il risultato ottenuto calcolando la funzione parziale (di un argomento) con numero di Gödel x,φ_x, sul numero naturale y, ovvero una funzione in grado di simulare il comportamento di qualsiasi funzione parziale di un argomento. Il teorema seguente dice che esiste una funzione parziale ricorsiva che presenta questa proprietà.

Teorema 1.18 (Teorema di Enumerazione)**.** *Esiste un numero naturale z tale che la funzione parziale ricorsiva di due variabili calcolata dalla computazione avente z come numero di Gödel, $\varphi_z^{(2)}$, è tale che, per ogni coppia (x, y), vale:*

$$\varphi_z^{(2)}(x,y) = \begin{cases} \varphi_x(y) & \text{se } \varphi_x(y) \text{ è definita} \\ \text{indefinita} & \text{se } \varphi_x(y) \text{ è indefinita} \end{cases}$$

La funzione $\varphi_z^{(2)}$ è chiamata *funzione universale parziale* per le funzioni parziali ricorsive di una sola variabile. L'aggettivo *universale* discende dalla proprietà fondamentale della funzione $\varphi_z^{(2)}$, e cioè quella di essere in grado di simulare qualsiasi funzione parziale di un argomento.
Si può dimostrare poi che il Teorema di Enumerazione può essere generalizzato in modo da ottenere, per ogni numero naturale $k \geq 1$, la *funzione universale parziale* $\varphi_z^{(k+1)}$ per le funzioni parziali ricorsive di k variabili.
La *MdT* che esegue la computazione P_z è in grado di simulare la computazione di una qualsiasi funzione parziale su un qualsiasi input, e per questo motivo è detta *Macchina di Turing Universale*.
Vediamo ora un teorema dovuto a Kleene, che costituisce uno strumento molto potente e di grande applicabilità. Prima però vediamo da un punto di vista intuitivo in cosa consiste questo risultato: consideriamo la famiglia di tutte le funzione parziali di $m+n$ variabili, esprimibile come $\varphi_x^{(m+n)}(y_1, \ldots, y_m, z_1, \ldots, z_n)$ (al variare del numero di Gödel x otteniamo infatti le diverse funzioni di $m+n$ variabili). Assegnando un valore alle variabili $(y_1, \ldots, y_m)$ e al parametro x che indica il numero di Gödel, la funzione $\varphi_x^{(m+n)}(y_1, \ldots, y_m, z_1, \ldots, z_n)$ si riduce ad una specifica funzione parziale che dipende solo dalle n variabili $z_1, \ldots, z_n$. Indichiamo con $f(z_1, z_2, \ldots, z_n)$ tale funzione. Il risultato di Kleene consiste nella dimostrazione dell'esistenza di una funzione ricorsiva, che dipende dalle variabili $(y_1, \ldots, y_m)$ e dal parametro x, che permette di calcolare il numero di Gödel di $f(z_1, z_2, \ldots, z_n)$.
Questo risultato permette dunque di determinare la relazione tra le famiglie $\varphi_x^{(n)}(y_1, \ldots, y_n)$ di funzioni parziali ricorsive per valori diversi del numero n di argomenti.

Teorema 1.19 (Teorema s-m-n)**.** *Per ogni $m, n \geq 1$, esiste una funzione ricorsiva s_n^m di $m+1$ variabili tale che, per ogni $(x, y_1, \ldots, y_m)$,*

$$\varphi_x^{(m+n)}(y_1, \ldots, y_m, z_1, \ldots, z_n) = \varphi_{s_n^m(x, y_1, \ldots, y_m)}^{(n)}(z_1, \ldots, z_n)$$

Osservazione 1.20. La dimostrazione più comune di questo teorema ha carattere informale, poiché fa uso della tesi di Church-Turing. Utilizzando argomenti più formali è possibile dimostare un risultato ancora più forte, ossia che la funzione s_n^m è ricorsiva e bigettiva.

Si vede infine un'applicazione del Teorema s-m-n:

Teorema 1.21. *Esiste una funzione ricorsiva g di due variabili tale che, per ogni coppia (x, y),*

$$\varphi_{g(x,y)} = \varphi_x \circ \varphi_y.$$

Capitolo 2

Introduzione alla complessità computazionale e alle classi di complessità

2.1 La complessità computazionale e la tesi dell'efficienza polinomiale

Dopo gli accenni dati in precedenza di teoria della calcolabilità è ora il momento di trattare la teoria della complessità computazionale, ovvero dopo aver risposto a domande riguardo l'esistenza di algoritmi per risolvere i problemi bisogna ora capire se tali algoritmi sono *efficienti*, ovvero se una soluzione del problema può essere trovata con un utilizzo di una quantità ragionevole di risorse. Innanzitutto dovremo quindi passare dalla nozione di problema risolvibile tramite un algoritmo a quella di problema risolvibile tramite un algoritmo in modo efficiente e parallelamente da quella di problema non risolvibile tramite un algoritmo a quella di problema non risolvibile tramite un algoritmo in modo efficiente (o intrattabile).
Le valutazioni di complessità dei problemi dipendono sostanzialmente da tre fattori:

- il modello di calcolo adottato,
- la risorsa il cui uso si intende analizzare,
- il criterio con cui misurare il costo associato all'uso della risorsa scelta.

Dati un problema computazionale e una risorsa di riferimento, si cerca da un lato di trovare un limite superiore all'utilizzo fatto della risorsa per risolvere il problema (quantificare l'utilizzo della risorsa fatto da un determinato algoritmo) e dall'altro di trovare un limite inferiore (cioè la quantità di risorsa necessaria ad **ogni** algoritmo). La complessità computazionale di un problema è determinata precisamente quando le due limitazioni coincidono.
Le risorse che prenderemo in esame sono lo spazio (di memoria occupato) e il tempo (necessario al calcolatore per completare la computazione). La qualità di un algoritmo si valuta rispetto al suo comportamento asintotico, ovvero quando la dimensione del

problema tende all'infinito. Questa scelta è motivata da ragionamenti sulla generalità; infatti un buon comportamento asintotico garantisce che ad una crescita della dimensione corrisponde una crescita ragionevole della funzione che misura il costo dell'algoritmo, mentre al contrario un cattivo comportamento asintotico rende applicabile l'algoritmo solo per dimensioni ridotte. A tal proposito, per convenzione un algoritmo si classifica come efficiente se il suo tempo di esecuzione è polinomiale (cioè se il suo tempo di esecuzione su qualsiasi input è limitato da una funzione polinomiale) nella dimensione del problema, inefficiente se è esponenziale (cioè se non esiste una funzione polinomiale come sopra) o più correttamente *superpolinomiale*.
La distinzione tra algoritmi di costo polinomiale ed algoritmi di costo esponenziale determina i concetti di *trattabilità* ed *intrattabilità*, che definiamo ora in maniera formale:

Definizione 2.1. Un problema è detto intrattabile se può essere dimostrato che non esistono algoritmi di costo in tempo polinomiale per risolverlo. Un problema è detto presumibilmente intrattabile se da un lato non si conoscono algoritmi di costo polinomiale che lo risolvono, ma dall'altro lato non ne è stata dimostrata formalmente l'intrattabilità. Infine, un problema è detto trattabile se esiste un algoritmo per la sua risoluzione di costo in tempo polinomiale

Viene dunque spontaneo fondare la classificazione degli algoritmi sul criterio secondo il quale *gli algoritmi efficienti sono quelli il cui costo ha una crescita polinomiale nella dimensione di input*. Senza dimenticare che nella pratica algoritmi con costi del tipo n^{1000} difficilmente saranno utilizzabili concretamente, possiamo portare diverse argomentazioni a sostegno di questo criterio, detto *efficienza polinomiale*. Una prima motivazione piuttosto forte è rappresentata dal fatto che una teoria basata sul criterio di efficienza polinomiale è indipendente dal modello. Per cominciare diamo la

Definizione 2.2. Diciamo che due modelli di calcolo A e B sono polinomialmente correlati se è possibile simulare l'esecuzione di un passo del modello A tramite un numero di passi nel modello B che è polinomiale nella dimensione del problema in esame, e viceversa.

Si vede facilmente che il criterio di efficienza polinomiale è indipendente da quale modello venga scelto all'interno di un insieme di modelli polinomialmente correlati, ma è più importante notare che tutti i *ragionevoli* modelli di calcolo sono polinomialmente equivalenti alla MdT rispetto a tale criterio; tale proprietà è chiamata *tesi del calcolo sequenziale*. Ne consegue che si può parlare di algoritmi efficienti o inefficienti in generale, cioè senza fare riferimento al modello di calcolo utilizzato. In generale, le caratteristiche fondamentali di una teoria della complessità basata sul criterio di efficienza polinomiale sono:

- la risorsa di riferimento è il tempo, ossia il costo computazionale corrispondente al tempo di calcolo;
- il modello di riferimento è la MdT;
- il costo computazionale viene misurato nel caso peggiore;

- l'obbiettivo è l'analisi del comportamento asintotico della funzione che esprime tale costo, ignorando le costanti moltiplicative;
- l'efficienza è sinonimo di un costo computazionale polinomiale.

La tesi dell'efficienza polinomiale non è direttamente dimostrabile, ma vi sono varie prove a sostegno della sua validità. Inoltre accettarla permette di sviluppare una teoria elegante e in grado di fornire utili indicazioni anche riguardo all'efficienza che si può ottenere in pratica.

2.2 Le classi di complessità e la classe P

Un altro concetto chiave nella teoria della complessità è quello di *classe di complessità*, ossia di insieme di tutti i problemi risolvibili su un dato modello di calcolo con una comune limitazione nell'uso di una o più risorse. Più precisamente, per definire una classe di complessità dobbiamo specificare un certo numero di parametri:

- il modello di calcolo di riferimento;
- una modalità con cui si eseguono le computazioni (nel nostro caso: deterministico);
- una risorsa *costosa* disponibile entro un certo limite;
- una funzione $f : \mathbb{N} \to \mathbb{N}$ che esprime tale limite.

Abbiamo allora la seguente definizione:

Definizione 2.3. Una classe di complessità è un insieme di linguaggi decisi da una certa Macchina di Turing che opera secondo la modalità appropriata e che consuma, per ogni input x, al più $f(|x|)$ unità della risorsa specificata

Le funzioni f che esprimono limiti all'uso di risorse devono soddisfare certi requisiti circa la loro stessa calcolabilità efficiente, onde evitare che l'uso di funzioni *strane* generi problemi tecnici. A tal proposito si definisce il concetto di funzione di costo *propria*. Una funzione f è propria se esiste una MdT in grado, per ogni input x, di 'marcare' esattamente $f(|x|)$ celle prima di fermarsi entro un tempo proporzionale a $|x| + f(|x|)$. A questo punto possiamo definire alcune classi di complessità di base:

Definizione 2.4. Sia f una funzione propria. La classe di complessità $TEMPO(f)$ è l'insieme dei linguaggi decisi da una MdT che consuma, per ogni input x, al più $f(|x|)$ unità di tempo.

Definizione 2.5. Sia f una funzione propria. La classe di complessità $SPAZIO(f)$ è l'insieme dei linguaggi decisi da una MdT che consuma, per ogni input x, al più $f(|x|)$ unità di nastro (celle), dove $|x|$ denota la lunghezza della stringa x.

Le osservazioni precedenti sul criterio di efficienza polinomiale motivano l'importanza di caratterizzare i problemi risolvibili in tempo polinomiale, ossia la classe dei probemi trattabili. La nozione di trattabilità vista come sinonimo di efficienza polinomiale è catturata dalla prima classe di complessità rilevante che tratteremo: **P**.

Definizione 2.6. La classe **P** è la classe dei linguaggi L per i quali esiste una *MdT* M tale che:

- $L = L_M$;
- per ogni input x, M si arresta in uno stato finale dopo un numero di passi polinomiale in $|x|$.

Equivalentemente, la classe **P** può essere definita come

$$\bigcup_{k=0}^{\infty} TEMPO(n^k)$$

Notiamo che dunque la classe **P** è definita in termini di una famiglia di funzioni costo piuttosto che da una singola funzione.
La macchina M si arresta, su ogni stringa x di input, dopo un numero di passi al più polinomiale in $|x|$ (naturalmente se x appartiene al linguaggio L la macchina termina in uno stato di accettazione, mentre se non appartiene al linguaggio termina in uno stato di rifiuto). Inoltre grazie alla corrispondenza tra linguaggi e problemi decisionali la classe **P** può essere caratterizzata, in modo equivalente, come la classe dei problemi decisionali che possono essere risolti in tempo polinomiale su una *MdT*.
Per ora abbiamo definito la classe **P** sulla *MdT*. Tuttavia possiamo invocare la tesi del calcolo sequenziale e mostrare che la classe **P** è *robusta*, ossia che, al variare del modello, purché questo sia ragionevole, non cambia l'insieme dei linguaggi che vi appartengono (proprietà non banale).
La classe di complessità più importante ai fini di comprendere i fondamenti dell'intrattabilità è invece la classe **NP**, che tratteremo nel capitolo successivo. A tal fine si discuterà brevemente anche della nozione di non determinismo, cioè un *modo* diverso di calcolare.

Capitolo 3

La classe NP e la NP-completezza

3.1 Il non determinismo

Il problema generale che ci si pone, centrale in complessità computazionale ma che travalica i confini di questa disciplina, è:

. *È* più 'difficile' calcolare la soluzione di un problema oppure verificare la correttezza di un'ipotetica soluzione?

Tale problema infatti non è estraneo neanche all'esperienza quotidiana: spesso infatti si cerca inutilmente di risolvere un problema e ci si stupisce poi della facilità della soluzione quando questa viene fornita da un agente esterno.
Questa osservazione ci induce a pensare che la verifica sia più 'facile' del calcolo, e si può dimostrare che questo equivarrebbe, in un ben preciso ambito formale, al risolvere il problema aperto del confronto tra le classi **P** e **NP**.
Una grande varietà di problemi computazionali presentano la seguente caratteristica: pur essendo presumibilmente intrattabili, ammettono una procedura di verifica efficiente per una soluzione "candidata". Questa proprietà è espressa dalla classe di complessità **NP**, che può infatti esser definita informalmente come la classe dei problemi decisionali per i quali ipotetiche soluzioni possono essere verificate in tempo polinomiale.
Per definire però formalmente la classe **NP** è necessario introdurre il concetto di *non determinismo*.

Computazioni non deterministiche. La transizione dal determinismo al non determinismo è un passaggio da modelli di calcolo *ragionevoli*, come la *MdT*, a modi di calcolare che perdono il contatto con la pratica dell'esecuzione di algoritmi. Si vedrà infatti che il non determinismo è una sorta di di artificio che si rivela estremamente utile nello studio della complessità di alcuni problemi presumibilmente intrattabili; tale artificio consiste nella possibilità di eseguire il calcolo con l'aiuto di un agente esterno in grado di dare 'suggerimenti' sulle scelte da compiere.
Un'idea del funzionamento di una macchina non deterministica M è data da:

- M è in grado di generare efficientemente la 'configurazione giusta' (se ne esiste una) e di verificare che essa descrive effettivamente una soluzione;

- dovendo decidere il procedimento da seguire, M compie sempre le scelte 'opportune';
- M è in grado di procedere considerando simultaneamente tutte le possibili alternative, tra cui quindi, se esiste, anche una di quelle che portano alla soluzione;
- M è guidata da *suggerimenti* corretti rispetto alle scelte da compiere.

Possiamo ora dare due definizioni formali (alternative ma equivalenti) di non determinismo. Nella prima definizione faremo riferimento al concetto di *stato* di una computazione, intendendo una configurazione che descrive la computazione in modo non ambiguo ad un certo istante di tempo (ad esempio, nel caso della *MdT*, lo stato è descritto dalla *descrizione istantanea*, vista in una capitolo precedente).

Definizione 3.1 (Prima definizione di non determinismo)**.** Una computazione si dice non deterministica quando consiste in una successione di passi discreti, ciascuno dei quali è una transizione da uno stato ad un insieme di stati (piuttosto che a un singolo stato).

Definizione 3.2 (Seconda definizione di non determinismo)**.** Siano $f_A : \Sigma^* \to \{0,1\}$ la funzione caratteristica di un insieme A, Y un insieme detto insieme delle variabili non deterministiche e g una funzione ($g : \Sigma^* \times Y \to \{0,1\}$) tale che, per ogni x, esiste una variabile non deterministica $y \in Y$ tale che $f_A(x) = g(x,y)$. In altri termini, alla funzione da calcolare,

$$f_A(x) = \begin{cases} 1 & \text{se } x \in \text{A} \\ 0 & \text{altrimenti} \end{cases}$$

viene associata la funzione

$$g(x,y) = \begin{cases} 1 & \text{se } R(x,y) \text{ è vera} \\ 0 & \text{altrimenti} \end{cases}$$

dove $R(x,y)$ è un opportuno predicato. Si dice che una computazione non deterministica è efficiente rispetto al tempo o allo spazio quando il costo computazionale (tempo o spazio deterministico) del calcolo di $g(x,y)$, espresso solamente in funzione di $|x|$, è inferiore rispetto al costo computazione (deterministico) di $f(x)$.

Notiamo ora che è possibile introdurre l'elemento non deterministico nella definizione di macchina di Turing: una macchina di Turing non deterministica (in breve *MdTN*) è definita esattamente come una *MdT*, con un'unica modifica che riguarda la funzione di transizione che non restituisce una singola configurazione successiva della macchina ma un insieme di possibili configurazione successive. Possiamo perciò vedere la *MdTN* come una variante della *MdT* in cui ad ogni passo la macchina può passare dallo stato attuale ad un insieme di stati successivi piuttosto che ad un singolo stato. Alternativamente, la *MdTN* può essere vista come una *MdT* arricchita dalla presenza di una nuova componente, detta *modulo di ipotesi*, dotata di una propria testina di sola scrittura. Il compito svolto da questo modulo è di scrivere una stringa sul nastro all'inizio della computazione (*fase di ipotesi*). Questa stringa (che chiameremo *certificato*) corrisponde al *suggerimento* di cui abbiamo parlato informalmente in precedenza; infatti può essere utilizzata dalla *MdTN* come se fosse un'informazione

aggiuntiva all'input. Dopo aver letto l'input e la stringa non deterministica prodotta dal modulo di ipotesi, la *MdTN* procede in modo deterministico, verificando se la stringa conduce effettivamente ad una soluzione (*fase di verifica*). Si osservi che una *MdTN* prevede una computazione diversa per ogni possibile stringa fornita dal modulo di ipotesi.

Definizione 3.3. Una MdTN M accetta una stringa di input x se e solo se almeno una delle computazioni originate relativamente a x termina nello stato di accettazione q_S. Il linguaggio L_M riconosciuto da M è dato da

$$L_M = \{x \in \Sigma^* | M \text{ accetta } x\}$$

In modo del tutto analogo, diremo che una *MdTN* M risolve un problema decisionale Π se, per ogni istanza $I \in D_\Pi$ si ha che

1. Se $I \in S_\Pi$, allora esiste un'*ipotesi* che, qualora prodotta in corrispondenza di I, fa sì che la fase di *verifica* termini in uno stato di accettazione;

2. Se $I \notin S_\Pi$, allora non esiste alcuna *ipotesi* che, qualora prodotta in corrispondenza di I, faccia sì che la fase di *verifica* termini in uno stato di accettazione.

Si noti che non viene specificato esattamente il comportamente della *MdTN* nel caso in cui $I \notin S_\Pi$: la macchina può fermarsi in uno stato di rifiuto oppure addirittura non terminare.
Il tempo impiegato da una *MdTN* M ad accettare una stringa $x \in L_M$ è pari al minimo numero di passi, preso su tutte le computazioni accettanti originate da x, impiegati per raggiungere lo stato finale q_s (ovvero il minimo numero di passi sufficienti a leggere sia la stinga di input sia la stringa prodotta dal modulo di ipotesi e ad eseguire la verifica dell'ipotesi). Si ha dunque la seguente definizione:

Definizione 3.4. Una MdTN M ha un costo in tempo espresso da una funzione $T : \mathbb{N} \to \mathbb{N}$ se, su input $x \in L_M$, M entra in uno stato finale di accettazione dopo al più $T(|x|)$ passi.

Si noti che il tempo viene misurato esclusivamente sulle computazioni accettanti; più precisamente il numero di passi viene fatto corrispondere alla lunghezza del cammino più breve tra tutti quelli che terminano in uno stato di accetazione.
Associati al calcolo deterministico, vi sono criteri di valutazione del costo computazionale, meccanismi di accettazione/rifiuto e condizioni di terminazione che trovano confronto immediato nell'intuizione. Viceversa, analizzando computazioni non deterministiche, ci scontriamo con criteri a prima vista controintuitivi: alcune delle computazioni possono procedere anche dopo che un ramo non deterministico ha raggiunto uno stato finale; inoltre, ai fini del costo computazionale, viene valutata la lunghezza del minimo cammino accettante, mentre la lunghezza (e addirittura la finitezza) dei cammini non accettanti non riveste interesse.

3.2 La classe NP

La *MdTN* consente di definire formalmente classi di complessità non deterministiche e in particolare la classe **NP**.

Definizione 3.5 (Classe di Complessità $NTEMPO\,(f)$)**.** La classe di complessità $NTEMPO\,(f)$ è l'insieme di quei linguaggi decisi da una *MdTN* che consuma , su ogni input $x \in L$, al più $f(|x|)$ unità di tempo.

Definizione 3.6. La classe **NP** è la classe dei linguaggi L per i quali esiste una *MdTN* M tale che

- $L = L_M$;
- per ogni input $x \in L$, M si arresta in uno stato finale di accettazione dopo un numero di passi polinomiale in x.

Equivalentemente, la classe **NP** può essere definita come

$$\bigcup_{k=0}^{\infty} NTEMPO(n^k).$$

Si osservi che imporre un limite polinomiale al costo in tempo implica che la lunghezza della stringa prodotta nella fase di ipotesi debba anch'essa essere limitata polinomialmente, poiché per esaminarla può essere speso solo un ammontare di tempo al più polinomiale.
Un'importante osservazione sulla classe **NP** riguarda la differenza tra i cammini non deterministici che corrispondono alla risposta SÌ e quelli che corrispondono alla risposta NO. Nel primo caso l'appartenenza a **NP** garantisce che esiste un cammino la cui lunghezza è limitata superiormente da un polinomio, mentre nel secondo non vengono fatte ipotesi di alcun genere. Tale *mancanza di simmetria* dipende dal nuovo criterio di accettazione e non si verifica per la classe **P** e in generale per tutte le classi definite su macchine deterministiche. Infatti se un problema decisionale Π può essere risolto in tempo polinomiale su una *MdT*, lo stesso accade anche per il problema ad esso complementare. Ciò è conseguenza del fatto che le computazioni sulla *MdT* limitata polinomialmente in tempo terminano, indipendentemente dall'input specifico, entro un numero polinomiale di passi, e quindi è sufficiente scambiare tra di loro i due stati finali q_S e q_N per ottenere, da una computazione per un linguaggio, la corrispondente computazione per il suo complemento. Che la stessa cosa accada per tutti iproblemi risolvibili in tempo polinomiale su una *MdTN*, cioè per tutti i problemi in **NP**, non è ovvio ed è anzi altamente improbabile.
La mancanza di simmetria della classe **NP** suggerisce che, al contrario di quanto accade per la classe **P**, l'appartenenza di un problema alla classe **NP** non sembra implicare l'appartenenza a **NP** anche del problema complementare. Questo suggerisce l'opportunità di introdurre la classe dei problemi complementari a quelli in **NP**:

$$\mathbf{coNP} = \{\Sigma^* - L \mid L \in \mathbf{NP}\}\,.$$

Per quanto detto sopra si è portati a credere che valga $\mathbf{NP} \neq \mathbf{coNP}$, ossia che esistano problemi in **coNP** per cui non esistono algoritmi polinomiali non deterministici. Un risultato importante (dimostrato dal fatto che $\mathbf{P} = \mathbf{coP}$) è:

Proposizione 3.7. *Se* $\boldsymbol{NP \neq coNP}$*, allora* $\boldsymbol{P \neq NP}$

Al contrario, l'eventuale dimostrazione della improbabile uguaglianza $\mathbf{NP} = \mathbf{coNP}$ non escluderebbe la possibilità che $\mathbf{P} \neq \mathbf{NP}$.

3.3 Riduzioni calcolabili in tempo polinomiale

Definizione 3.8. Si dice che l'insieme A è funzionalmente (o molti a uno) riducibile in tempo polinomiale all'insieme B, e si scrive $A \preceq_m^{\mathbf{P}} B$, se esiste una funzione f calcolabile in tempo polinomiale tale che $x \in A$ se e solo se $f(x) \in B$.

Se $A \preceq_m^{\mathbf{P}} B$, si dice anche che A è Karp-riducibile a B. La funzione f è chiamata *trasformazione polinomiale*. Si vedrà in seguito che le riduzioni di Karp permettono di classificare i problemi in **NP** in base alla loro difficoltà e di identificare una classe di equivalenza che contiene i problemi più difficili di **NP** (detta la classe dei problemi **NP**-completi).
Per le riduzioni limitate in tempo polinomiale, è immediato dimostrare le proprietà espresse dai seguenti teoremi.

Teorema 3.9 (Proprietà delle riduzioni di Karp)**.** *Si considerino gli insiemi A, B e C. Valgono le seguenti relazioni:*

$A \preceq_m^{\boldsymbol{P}} A$*;*

$A \preceq_m^{\boldsymbol{P}} B \wedge B \preceq_m^{\boldsymbol{P}} C \rightarrow A \preceq_m^{\boldsymbol{P}} C$*;*

$A \preceq_m^{\boldsymbol{P}} B \wedge B \in \boldsymbol{P} \rightarrow A \in \boldsymbol{P}$*;*

$A \preceq_m^{\boldsymbol{P}} B \wedge B \in \boldsymbol{NP} \rightarrow A \in \boldsymbol{NP}$*.*

Infine si ha la:

Definizione 3.10 (Equivalenza e gradi)**.** Se $A \preceq_m^{\mathbf{P}} B$ e $B \preceq_m^{\mathbf{P}} A$ si dice che A e B sono equivalenti rispetto a $\preceq_m^{\mathbf{P}}$, e si scrive $A \equiv_m^{\mathbf{P}} B$. La relazione $\equiv_m^{\mathbf{P}}$ è una relazione di equivalenza. Le classi di equivalenza sono dette m-gradi polinomiali, e l'insieme quoziente è un insieme parzialmente ordinato che chiameremo K-ordinamento.

Se escludiamo gli insieme 'patologici' $\emptyset$ e Σ^*, esiste un gado minimo nel K-ordinameno, costituito da tutti gli insiemi in $\mathbf{P}$-$\{\emptyset, \Sigma^*\}$.
Allo stato attuale delle conoscenze, l'esistenza in **NP** di gradi distinti da **P** (o da $\mathbf{P}$-$\{\emptyset, \Sigma^*\}$ nel caso del K-ordinamento) può solo essere predicata sulla base dell'assunzione che $\mathbf{P} \neq \mathbf{NP}$. Se accettiamo tale 'ipotesi di lavoro', almeno un grado distinto è quello costituito dalla classe degli insiemi completi per **NP**, che verranno trattati più approfonditamente nella sezione seguente.

3.4 La classe dei problemi NP-completi

Utilizzando lo strumento delle riduzioni limitate in tempo polinomiale, è possibile orientare lo studio delle relazioni tra le classi **P** e **NP** alla ricerca dei problemi più 'difficili' all'interno della classe **NP**, e dunque problemi che sembrano proporsi come ragioneovli candidati a non appartenere alla classe **P**. In particolare, nella classe **NP** è possibile individuare una classe di problemi, i problemi completi per **NP**, la cui appartenenza a **P** implicherebbe l'uguaglianza **P** = **NP**. Si danno dunque le:

Definizione 3.11. Un insieme A si dice completo per **NP** rispetto a $\preceq_m^{\mathbf{P}}$ (o, più brevemente, **NP**-completo) se:

1. $A \in \mathbf{NP}$;
2. $\forall B \in \mathbf{NP}$ si ha che $B \preceq_m^{\mathbf{P}} A$.

Definizione 3.12. Un insieme A si dice completo per **NP** rispetto a $\preceq_T^{\mathbf{P}}$ se :

- $A \in \mathbf{NP}$;
- $\forall B \in \mathbf{NP}$ si ha che $B \preceq_T^{\mathbf{P}} A$.

(Si noti che il termine **NP**-completo viene riservato alla classe dei problemi completi per **NP** rispetto alle riduzioni di Karp).
L'esistenza di problemi completi per **NP** è molto importante, in quanto consente di importare il confronto tra **P** e **NP** sulla base dell'analisi di singoli problemi. Sia infatti A un problema completo per **NP**; allora dimostrare l'uguaglianza **P** = **NP** coincide con esibire un algoritmo deterministico di costo polinomiale per A dal quale (tramite le riduzioni) si avrebbero automaticamente algoritmi deterministici di costo polinomiale per tutti i problemi in **NP**. Viceversa dimostrare che **P** $\neq$ **NP** è equivalente a dimostrare che A non appartiene alla classe **P**.
L'esistenza di problemi completi per **NP** costituisce così un indizio a favore della congettura **P** $\neq$ **NP**: falsificare la congettura vorrebbe dire dimostrare che esistono algoritmi efficienti per una ampia classe di problemi per cui, finora, non sembra possibile sfuggire alla ricerca esaustiva.
La definizione di linguaggio **NP**-completo sembra a prima vista difficile da utilizzare direttamente per dimostrare la **NP**-completezza di uno specifico linguaggio: dimostrare che un linguaggio è **NP**-completo significa stabilire che *tutti* i linguaggi in **NP** gli si riducono in tempo polinomiale (ovvero bisogna verificare una proprietà che richiede il confronto di un linguaggio con un'intera classe).

Notazione 3.13. Da ora in poi indicheremo la classe dei problemi **NP**-completi con **NPC**

Siamo ora pronti per enunciare il teorema di Cook, ma prima diamo alcuni elementi per capirne l'importanza nello sviluppo della teoria della complessità e del suo ruolo nello studio della **NP**-completezza. L'importanza del risultato di Cook è dovuta sia al fatto che in questo modo si è dimostrato che la classe dei problemi **NPC** contiene problemi naturali (e questo giustifica ulteriormente l'importanza della teoria della **NP**-completezza), sia perché fornisce una metodologia per dimostrare agilmente la **NP**-completezza di molti problemi. Si può infatti applicare il seguente lemma:

Lemma 3.14. *Siano A e B due insiemi che appartengono alla classe **NP**. Se $A \in$ **NPC** e $A \preceq_m^{\mathbf{P}} B$, allora $B \in$ **NPC**.*

Utilizzando il teorema di Cook ed il lemma precedente, è stato possibile determinare la **NP**-completezza di molti altri problemi in **NP**, semplicemente dimostrando che un problema **NP**-completo vi si riduce. Esistono anche problemi a cui tutti i problemi in **NP** si riducono in tempo polinomiale, ma per i quali non sono note dimostrazioni di appartenenza a **NP**. Problemi con queste caratteristiche sono detti **NP**-hard, ad indicare che sono 'difficili almeno' quanto i problemi **NP**-completi.

3.5 Il teorema di Cook

Dopo aver mostrato l'utilità del concetto di **NP**-completezza nello sviluppo della teoria della complessità computazionale e dunque della teoria della calcolabilità tutta, definiremo ora il problema **SAT** e quindi enunceremo e dimostreremo il teorema di Cook a chiusura del capitolo.
Il problema **SAT** è definito come segue: data una formula booleana, espressa in forma normale congiuntiva e con un numero qualunque di variabili proposizionali e di clausole, dire se esiste un assegnamento di verità alle variabili che rende vera la formula.

Teorema 3.15. *Il problema **SAT** è **NP**-completo*

Cenni. La prima cosa da far vedere è che **SAT** $\in$ **NP**. Questo è facile, in quanto ogni istanza positiva di **SAT** è certificata da un assegnamento di verità, che consiste in una stringa lunga quanto il numero di variabili della formula. La verifica può essere agevolmente eseguita in tempo lineare nella lunghezza della formula, semplicemente sostituendo all'i-esima variabile della formula x_i l'i-esimo bit del certificato e alla k-esima variabile complementata $\neg x_j$ il complemento del j-esimo bit del certificato. In questo modo la formula viene trasformata in un'espressione che coinvolge solo le due costanti e le operazioni logiche $\wedge$ e $\vee$ e che può quindi essere valutata banalmente.
La parte difficile della dimostrazione consiste nel far vedere che qualsiasi linguaggio $L \in$ **NP** si riduce a **SAT**. L'idea cruciale è di associare ad una generica computazione non deterministica una formula la cui soddisfattibilità corrisponde all'esistenza di un cammino accettante. Ogni linguaggio in **NP** può essere descritto in modo standard fornendo una *MdTN* che lo riconosce in tempo polinomiale. Questo consente di lavorare con una generica macchina M e di derivare una trasformazione polinomiale 'generica' dal linguaggio L_M a **SAT**. □

Il teorema di Cook consente di dare un'altra definizione di **NP**-completezza in termini di insiemi in **NP** che si riducono a **SAT**; questa definizione mette in rilievo come si possa esprimere la difficoltà dell'intera classe **NP** tramite un singolo problema. Formalmente:

Definizione 3.16. Un insieme $A \in$ **NP** è **NP**-completo se **SAT** $\preceq_m^{\mathbf{P}} A$.

Capitolo 4

I SAT-solver

4.1 Definizione generale

Un **SAT**-solver, in generale, è un algoritmo che, data una formula booleana F, determina se è possibile trovare un assegnamento di verità delle variabili che rende vera F (diremo: modello di F) e in tal caso lo fornisca in output, mentre restituisce una prova dell'assenza di tale modello in caso contrario.
In generale, un metodo di risoluzione della logica proposizionale si articola così:

- Il problema da risolvere P
- Una formula F che rappresenti il problema P (ovvero tale una soluzione di P sia un modello di F)
- La conversione di F in Forma Normale Congiuntiva (CNF)
- La verifica della soddisfacibilità di F tramite un **SAT**-solver

Il procedimento che "trasforma" P in F (detto *encoding*) è ovviamente specifico di ogni problema, mentre la conversione di F in CNF è un procedimento generale applicabile ad ogni formula specifica.
È dunque facile capire l'importanza di **SAT** e dei **SAT**-solver in logica e in generale in tutti i casi in cui sia possibile ridurre un problema ad una formula che lo rappresenti. Diventa dunque fondamentale capire come si strutturano in generale e ideare degli algoritmi che implementino un **SAT**-solver. Uno degli algoritmi più famosi ed utilizzati è il DPLL (Davis-Putnam-Logemann-Loveland, dai nomi degli ideatori). Prima di dicutere direttamente la procedura DPLL introduciamo brevemente la tenica di *backtracking*, che ci servirà per definire il DPLL stesso.
Il backtracking (traducibile in italiano come "monitoraggio a ritroso") è una tecnica utilizzata per trovare soluzioni a problemi in cui devono essere soddisfatti dei vincoli. Questa tecnica, infatti, enumera tutte le possibili soluzioni e scarta quelle che non soddisfano i vincoli.
Una tecnica classica consiste nell'esplorazione di strutture ad albero e tenere traccia di tutti i nodi e i rami visitati in precedenza, in modo da poter tornare indietro al più vicino nodo che conteneva un cammino ancora inesplorato nel caso che la ricerca nel ramo attuale non abbia successo. I nodi a profondità uguale rappresentano i possibili

valori di una variabile.
Il backtracking nel caso peggiore analizza tutti nodi dellalbero ed ha dunque complessità pari al numero di nodi. In situazioni in cui non si riesce a potare lalbero in modo significativo, come in tutti i problemi NP-completi, ha dunque una complessità esponenziale, ed quindi è poco efficiente nell'affrontare problemi che non siano NP-completi. In generale, comunque, l'algoritmo integra euristiche che permettono di diminuirne la complessità.

4.2 Il DPLL

La procedura di Davis-Putnam-Logemann-Loveland (DPLL) rappresenta probabilmente il metodo più efficace per affrontare il problema (computazionalmente intrattabile allo stato attuale delle conoscenze) della soddisfacibilità di una formula della logica proposizionale classica.
Descritta in modo schematico e sommario, la procedura tenta di costruire (mediante un meccanismo di backtracking) un assegnamento V che soddisfi una data formula A; la formula A viene usualmente preprocessata in modo da ridurla ad un insieme di clausole. Ad ogni nodo di scelta, la procedura assegna un valore di verità ad una lettera proposizionale; prima, però, esegue tutte le operazioni deterministiche utili a propagare le informazioni già acquisite.
La procedura in quanto tale è molto flessibile, ma solo se completata con opportune tecniche euristiche complementari risulta davvero efficace nelle applicazioni. Tali tecniche consistono ad esempio in adeguate selezioni della lettera proposizionale su cui operare la scelta non deterministica di un assegnamento, ma soprattutto in meccanismi di backtracking non cronologico e in appropriate strategie di apprendimento dai tentativi di ricerca falliti. Si esporrà prima uno schema-base della procedura DPLL, che prescinde dalle tecniche sopra accennate, che verranno poi discusse a parte singolarmente. Si osservi innanzitutto che la DPLL in quanto tale (a meno di farne estensioni ad hoc) non opera su tipi di dati che siano formule, ma solo su insiemi di clausole. Il suo utilizzo necessita quindi di una fase di preprocessamento.
Vediamo ora l'algoritmo di DPLL nel dettaglio: la formula F (che come visto precedentemente è in CNF) sarà del tipo $C_1 \wedge \cdots \wedge C_k$, dove le $C_1, \ldots, C_k$ sono clausole. Identifichiamo F con linsieme di clausole $C_0 = \{C_1, \ldots, C_k\}$.
La procedura DPLL manipola coppie (V, C) dove C è un insieme di clausole e V è un assegnamento parziale (cioè V assegna un valore di verità solo ad alcune delle lettere proposizionali della formula F). Scriviamo la coppia V, C con

$$V \vdash C$$

. La procedura si inizializza a

$$\emptyset \vdash C_0,$$

e sviluppa un albero etichettato con coppie $V \vdash C$; al termine dellesecuzione della procedura, le foglie dellalbero finale avranno etichetta $V \vdash C$, dove C è vuoto oppure contiene la clausola vuota.
Linsieme di clausole originario C_0 è soddisfacibile se e solo se lalbero generato da una (qualsiasi) esecuzione della procedura contiene una foglia etichettata con $V \vdash \emptyset$ (nel

qual caso V o una sua qualsiasi estensione totale sono un assegnamento che soddisfa C_0).
Lalbero generato dalla procedura deve essere esplorato (preferibilmente in profondità) secondo le regole che verranno esposte in seguito (Sussunzione, Risoluzione Unitaria, Asserzione, Letterale Puro, Spezzamento), dando precedenza alle regole che non introducono sdoppiamenti (in altri termini: la regola di spezzamento si applica solo se le altre non sono applicabili).
Le regole vanno applicate in modo esaustivo; per applicare una regola, si sceglie una foglia dellalbero corrente la cui etichetta sia del tipo indicato dalla premessa della regola stessa e si generano uno (o due nel caso della regola di spezzamento) nodi figli etichettandoli come previsto dal conseguente della regola.
Enunciamo ora le cinque regole di base:

Definizione 4.1 (Sussunzione)**.**

$$\frac{V \vdash C \cup \{p \vee C\}}{V \vdash C} \text{ se } V(p) = 1$$

$$\frac{V \vdash C \cup \{\neg p \vee C\}}{V \vdash C} \text{ se } V(p) = 0$$

Definizione 4.2 (Risoluzione unitaria)**.**

$$\frac{V \vdash C \cup \{p \vee C\}}{V \vdash C \cup \{C\}} \text{ se } V(p) = 0$$

$$\frac{V \vdash C \cup \{\neg p \vee C\}}{V \vdash C \cup \{C\}} \text{ se } V(p) = 1$$

Definizione 4.3 (Asserzione)**.**

$$\frac{V \vdash C \cup \{p\}}{V \cup \{V(p) = 1\} \vdash C}$$

$$\frac{V \vdash C \cup \{\neg p\}}{V \cup \{V(p) = 0\} \vdash C}$$

Definizione 4.4 (Letterale puro)**.**

$$\frac{V \vdash C}{V \cup \{V(p) = 1\} \vdash C} \text{se } p \text{ occorre in } C \text{ e } \neg p \text{ non occorre in } C$$

$$\frac{V \vdash C}{V \cup \{V(p) = 0\} \vdash C} \text{se } \neg p \text{ occorre in } C \text{ e } p \text{ non occorre in } C$$

Definizione 4.5 (Spezzamento)**.**

$$\frac{V \vdash C}{V \cup \{V(p) = 1\} \vdash C \| V \cup \{V(p) = 0\} \vdash C}$$

4.3 Le euristiche del DPLL

L'efficienza dell'algoritmo DPLL può essere ulteriormente migliorata tramite l'utilizzo delle euristiche. Le euristiche più note sono le seguenti:

- *Euristica del simbolo puro*
- *Euristica della clausola unitaria*
- *Euristica della scomposizione*
- *Euristiche dinamiche*

Vediamole ora singolarmente nel dettaglio:

Euristica del simbolo puro. Un simbolo puro è un letterale che compare nelle clausole sempre nella stessa forma, positiva o negativa. Ad esempio, nella formula seguente il letterale A è un simbolo puro poiché compare sempre nella stessa forma (positiva):

$$(A \vee B) \wedge (\neg B \vee \neg C)$$

In questo caso è consigliabile elaborare prima i simboli puri per ottenere una potatura logica più efficace e per concentrare il resto dell'elaborazione alle clausole restanti. Assegnando il valore *vero* al letterale A la formula si riduce alla sola clausola residuale $(\neg B \vee \neg C)$. È quindi più semplice valutare la soddisfacibilitÃ della formula e la ricerca delle sue soluzioni.

Euristica della clausola unitaria. Una clausola unitaria è una clausola composta da un unico letterale. Le clausole unitaria devono essere analizzate prima delle altre, in quanto ciò consente di effettuare una potatura logica più marcata dell'albero logico. Ad esempio, nella seguente formula è presente una clausola unitaria (C):

$$(A \vee B) \wedge (C) \wedge (\neg B \vee \neg C)$$

Per risolvere la formula in modo positivo è necessario assegnare al letterale C il valore di verità *vero*. Sarebbe inutile analizzare i casi con valori *falso* del letterale C. L'euristica della clausola unitaria consente di semplificare la formula residuale nella seguente forma: $(A \vee B) \wedge (\neg B)$.

Euristica della scomposizione. Quando una clausola è composta da un letterale non in comune con le altre clausole, è sufficiente cercare il valore booleano di verità del letterale specifico per rendere vera la clausola ed eliminarla dalla formula. Ad esempio, nella seguente formula logica la terza clausola è caratterizzata da un letterale specifico (D).

$$(A \vee B) \wedge (\neg B \vee C) \wedge (\neg A \vee \neg C \vee D)$$

Scomponendo la clausola di due sottoinsiemi disgiunti, è possibile mettere in evidenza il letterale non in comune, al fine di poter trattare la clausola come una clausola unitaria. Una volta assegnato il valore *vero* al letterale D, l'intera clausola può essere eliminata dalla formula.

Euristica di grado. L'euristica di grado determina l'ordine di assegnazione delle variabili e dei valori. Secondo l'euristica le variabili con maggiori vincoli con le altre devono essere elaborate prima. In una formula logica ciò equivale a dire che il letterale più comune tra le clausole deve essere elaborato prima nelle operazioni di assegnazioni. Ad esempio, la seguente formula è composta da tre clausole e da quattro variabili.

$$(A \vee B) \wedge (\neg B \vee C) \wedge (\neg B \vee \neg C \vee D)$$

La variabile B è presente tre volte, di cui due in forma negata ($\neg B$), la variabile C due volte e le restanti variabili A e $\neg D$ una volta. L'ordine di assegnazione migliore è, pertanto, il seguente: $\neg B$, C, A, $\neg D$. È molto importante considerare la forma booleana più frequente per ciascuna variabile nell'ordine di assegnamento.

Notiamo infine che tali euristiche, per essere effettivamente efficaci, devono essere elaborate ricorsivamente, cioè ogni qualvolta che la formula logica viene modificata. Durante l'elaborazione dei vari modelli (*model checking*) e l'assegnamento dei valori alle variabili, le clausole della formula logica originaria sono modificate o annullate. È quindi possibile che sorgano situazioni intermedie in cui le euristiche possono essere efficacemente applicate. Ad esempio, nella seguente formula non è applicabile nessuna euristica.

$$(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee \neg C)$$

Tuttavia, dopo aver assegnato il valore *vero* al letterale A, è possibile eliminare la prima clausola e il primo letterale ($\neg A$) della seconda clausola. La formula logica intermedia è $(C) \wedge (\neg B \vee \neg C)$ dove emerge chiaramente una clausola unitaria (C). graveE quindi consigliabile non limitare l'applicazione delle euristiche soltanto nella fase preliminare dell'elaborazione, cioè quella che precede l'esecuzione vera e propria dell'algoritmo, ma eseguirle ricorsivamente ogni qualvolta la formula logica viene modificata durante l'elaborazione.
Diciamo quindi che le euristiche del DPLL sono *euristiche dinamiche*.

4.4 L'algoritmo GSAT

Esaminiamo ora un altro possibile algoritmo utilizzabile per risolvere **SAT**, decisamente più efficiente del DPLL senza euristiche, denominato **GSAT**. **GSAT** è un algoritmo di tipo "greedy" (dall'inglese: avido o anche aggressivo) cioè che cerca di ottenere una soluzione del problema ottima da un punto di vista globale attraverso la scelta della soluzione migliore nellimmediato piuttosto che adottare una strategia a lungo termine.
I risultati sperimentali dimostrano che **GSAT** è particolarmente indicato nella risoluzione di problemi *hard*, generati randomicamente e con dimensioni maggior di quelli risolvibili tramite il DPLL standard. Innanzitutto, **GSAT** è una procedura *model-finding* invece che *theorem-proving*, cioè, se esiste un modello per la nostra formula booleana, allora **GSAT** ha una certa probabilità di trovarla ma se **GSAT** ritorna una risposta negativa allora *non è detto* che non esista un modello che soddisfi la nostra formula. Più formalmente:

model-finding: cerca un'interpretazione delle variabili che renda vera la formula oppure restituisce un messaggio di esito negativo. Se tale interpretazione esiste allora ovviamente si ha la certezza che la formula sia soddisfacibile.

theorem-proving: trova una prova formale del fatto che la *negazione* della formula sia soddisfacibile oppure restituisce un messaggio comunicando l'assenza di tale dimostrazione. Se tale dimostrazione esiste, allora ovviamente la formula originale non è soddisfacibile.

Passiamo ora a descrivere direttamente l'algoritmo **GSAT**. **GSAT** compie una ricerca locale "greedy" di un assegnamento delle clausole proposizionali della formula di partenza.
La procedura inizia generando un assegnamento di verità casuale, quindi cambia (*flips*) l'assegnamento delle variabili in modo da portare al maggior incremento possibile nel numero totale di clausole soddisfatte. Questi *flips* procedono fino a quando non viene trovato un assegnamento che soddisfi la formula o fino a quando non si raggiunge un numero massimo prefissato di *flips* (detto MAX−FLIPS). Se dopo MAX−FLIPS cambiamenti di assegnamento non si è ancora trovato un assegnamento positivo, allora **GSAT** genera un nuovo assegnamento randomico e ricomincia; tale processo è ripetuto al più un numero di volte prefissato (MAX−TRIES). Per dare un riferimento generale, MAX−FLIPS può essere settato ad un numero pari a qualche volta il numero delle variabili, mentre MAX−TRIES è scelto arbitrariamente in base a quanto si vuole rendere approfondita la ricerca.
Un'altra caratteristica di **GSAT** è che le variabili il cui assegnamento viene cambiato da un passo all'altro sono a loro volta scelte randomicamente fra quelle che porano ad un miglioramento dell'assegnamento generale ugualmente buono; questa componente di non determinismo nel cambio degli assegnamenti ci rende abbastanza sicuri del fatto che il procedimento non passerà due volte dallo stesso assegnamento. Inoltre, è importante notare che **GSAT**, a differenza degli algoritmi standard di ricerca locale, continua a compiere i *flips* anche quando questi non incrementano il numero totale di clausole soddisfatte ma lo lasciano invariato (tali passi sono chiamate *sideways moves*) Dopo averlo descritto, vediamo ora, tramite un esempio, il limite principale di **GSAT**. Prendiamo ad esempio la seguente congiunzione di clausole (dove i numeri rappresentano le variabili):

$$(1 \vee \neg 2 \vee 3) \wedge (1 \vee \neg 3 \vee 4) \wedge$$
$$(1 \vee \neg 4 \vee \neg 2) \wedge (1 \vee 5 \vee 2) \wedge$$
$$(1 \vee \neg 5 \vee 2) \wedge (\neg 1 \vee \neg 6 \vee 7) \wedge$$
$$(\neg 1 \vee \neg 7 \vee 8) \wedge \cdots \wedge$$
$$(\neg 1 \vee \neg 98 \vee 99) \wedge (\neg 1 \vee \neg 99 \vee 6)$$

Si nota che, sebbene la maggior parte delle clausole contengano un'occorrenza negativa della variabile 1, la formula può essere risolta solo se tale variabile ha assegnamento positivo. Il problema nasce dal fatto che l'approccio greedy indirizza ripetitivamente la ricerca verso un assegnamento negativo di 1, perché questo rende vere un maggior numero di clausole.

L'unico modo in cui **GSAT** può risolvere questo problema è che la ricerca inizi da un assegnamento molto vicino ad uno che soddisfi la formula.

Bibliografia

[1] A. Bernasconi, B. Codenotti, *Introduzione alla Complessità Computazionale*, capitoli 1 e 2, edito da Springer, 1998.

[2] C.H. Papadimitriou, *Computational Complexity*, parte 1 capitoli 2 e 3, parte 3 capitoli 8, 9, 10, edito da Addison-Wesley Publishing Company, 1995

[3] R. Nieuwenhuis, A., E. Rodriguez-Carbonell, *Introduction to SAT*, SAT and SMT for Solving CSPs - Session 1, Seminar on Constraint Programming, University of Bergen, 29 March 2011.

[4] S. Ghilardi, E. Nicolini, D. Zucchelli, *Introduzione alla logica proposizionale e alla procedura DPLL*, capitolo 2, Università degli studi di Milano, 2007

[5] B. Selman, H. Levesque, D. Mitchell, *A New Method for Solving Hard Satisfability Problems*, Proceedings of the Tenth National Conference on Artificial Inteligence (AAAI-92), San Jose, CA, july 1992, 440-446.