UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

TESI DI LAUREA MAGISTRALE IN INFORMATICA

Un simulatore di Macchine di Turing non deterministiche a scopo didattico

CANDIDATO
Francesco De Martino

RELATORE
Prof. Agostino Dovier

Anno accademico 2019-2020

Contatti dell'Istituto
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
https://www.dmif.uniud.it/

A tutti quelli che hanno creduto in me,
voi sapete chi siete.

Sommario

In questo elaborato verrà presentato con attenzione un simulatore di Macchine di Turing non deterministiche, esso verrà esposto sia dal punto di vista teorico che da quello pratico. Successivamente verrà presentato un 3SAT Solver creato appositamente per il simulatore.

All'inizio di questo elaborato verranno brevemente introdotti i principali passi che hanno portato alla definizione della Macchina di Turing e successivamente della Macchina di Turing non deterministica.

Indice

Elenco delle figure

1

Inquadramento storico

Nei primi anni del novecento il mondo matematico visse quella che fu chiamata "la crisi dei fondamenti della matematica", in questa fase storica furono messi in discussione quelli che fino ad allora erano sempre stati considerati i pilastri fondanti di questa disciplina, tra cui per esempio la geometria euclidea.

Una delle figure scientifiche di quel tempo che si interessò alla questione fu David Hilbert, matematico brillante che divenne particolarmente conosciuto in ambito accademico per la sua riduzione della geometria in 20 assiomi, tenenti conto delle nuove geometrie non euclidee e che non risentivano delle contraddizioni della geometria Euclidea.

David Hilbert propose un sfida che prese il nome di enteschеidungsproble [8] (problema di decisione). Tale problema è fondamentale nello sviluppo del presente elaborato.

1.1 Entscheidungsproblem

L'Entscheidungsproblem è una sfida che venne proposta da David Hilbert nel 1928. La sfida consisteva nella realizzazione, o perlomeno nella prova dell'esistenza anche solo a livello teorico, di un algoritmo in grado di stabilire se un enunciato nella logica del primo ordine fosse accettato o meno, restituendo ipoteticamente *true* o *false*, dagli assiomi di partenza.

Hilbert riteneva che un algoritmo del genere potesse realmente esistere, ciò però venne confutato, almeno in parte, da Kurt Gödel nel 1931 con i due teoremi di incompletezza, per maggiori informazioni al riguardo si veda per esempio [3] e [10].

1.2 Teoremi di incompletezza di Gödel

I due teoremi di incompletezza di Gödel confutavano in parte quella che era la richiesta dell'Entscheidungsproblem.

Teorema 1 (Primo Teorema di Incompletezza)**.** *In ogni teoria matematica $\boldsymbol{T}$, in grado di rappresentare tutte le funzioni ricorsive primitive, esiste una formula $\boldsymbol{\varphi}$, tale che, se $\boldsymbol{T}$ è coerente, allora né $\boldsymbol{\varphi}$ né $\neg\boldsymbol{\varphi}$ sono dimostrabili in $\boldsymbol{T}$.*

Osservazione 1. Immaginiamo di avere accesso ad una macchina capace di elaborare, almeno a livello teorico, qualsiasi formulazione logica e di rispondere in un tempo finito se essa è vera o falsa.

Immaginiamo ora di chiedere alla macchina di elaborare la seguente formulazione logica *"Questa macchina non può dirmi se tale asserzione sia vera"*. Malgrado sia stato assicurato che la macchina possa rispondere a qualsiasi formulazione logica, essa non sarà in grado di dare un responso a quest'ultima. Se la macchina rispondesse *vero*, ciò significherebbe che la macchina è appena arrivata a una conclusione errata, ma lo stesso varrebbe se rispondesse *falso*. A questo punto potremmo arrivare alla conclusione che la proposizione sia effettivamente vera, perché la macchina non è in grado di dare una risposta; ciò però non potrebbe essere asserito dalla macchina, altrimenti si andrebbe in contro ai problemi sopraccitati.

In questo esempio la macchina rappresenta una teoria matematica $\mathbf{T}$ in grado di rappresentare funzioni ricorsive primitive e la formulazione logica è un esempio di φ che non può essere dimostrata in $\mathbf{T}$.

Gödel riuscì a dimostrare che questa particolare condizione era presente in ogni teoria matematica in grado di rappresentare tutte le funzioni ricorsive primitive.

Presentiamo ora il secondo teorema di incompletezza.

Teorema 2 (Secondo Teorema di Incompletezza)**.** *Nessuna teoria matematica $\boldsymbol{T}$, che sia abbastanza coerente ed espressiva da contenere l'aritmetica, può essere utilizzata per dimostrare la sua stessa coerenza.*

Questo teorema mette in seria difficoltà quello che era l'intento di Hilbert, cioè riuscire a dimostrare la coerenza di sistemi formali complessi scomponendoli in sistemi più semplici. Infatti, secondo questo teorema, anche un sistema semplice come quello dell'aritmetica elementare non può essere usato per dimostrare se stesso.

Questo teorema può essere riformulato nella seguente maniera:

Teorema 3 (Secondo Teorema di Incompletezza riformulato)**.** *Se una teoria matematica $\boldsymbol{T}$ può dimostrare la sua stessa coerenza, allora $\boldsymbol{T}$ è sicuramente incoerente.*

Questa riscrittura rende ancora più evidente come l'intento di Hilbert sia probabilmente irraggiungibile.

Osservazione 2. Questi teoremi non cancellano del tutto la possibilità che un algoritmo come quello richiesto dall'Entscheidungsproblem possa esistere. Infatti, se in una qualche maniera, magari utilizzando un altro algoritmo, fosse possibile distinguere un enunciato decidibile da uno indecidibile, allora l'algoritmo richiesto dall'Entscheidungsproblem potrebbe ancora essere definito.

Bisognerà aspettare il 1936 perché Alonzo Church in [4] e contemporaneamente Alan Turing in [13] neghino definitivamente la possibilità dell'esistenza di un algoritmo del genere.

1.3 Tesi di Church-Turing

Alonzo Church e Alan Turing arrivarono entrambi alla conclusione che un algoritmo di decisione, come quello richesto da Hilbert, non potesse esistere. I due lavorarono alle loro rispettive soluzioni in modo del tutto indipendente. Successivamente il matematico Stephen Kleene dimostrò in [9] che le due tesi erano completamente equivalenti, fu lui infatti a parlare per la prima volta di tesi di Church-Turing.

In questo elaborato viene analizzato il lavoro di Alan Turing in quanto, oltre a dimostrare l'impossibilità dell'esistenza di un algoritmo di decisione come quello richiesto da Hilbert, introdusse nuovi concetti e strumenti di primario interesse per questo scritto.

1.4 On Computable Numbers, with an Application to the Entscheidungsproblem

On Computable Numbers, with an Application to the Entscheidungsproblem [13] è il titolo della pubblicazione nel quale Turing provò che un algoritmo come quello richiesto da Hilbert non potesse esistere. Per arrivare a questa conclusione formalizza, anzitutto, la nozione di computabilità.

1.4.1 Nozione di computabilità e macchina di Turing

Turing definisce un numero compatibile nel modo seguente:

Definizione 1. Un numero compatibile è un numero decimale che può essere calcolato con mezzi finiti.

Il matematico giustifica questa affermazione adducendo al fatto che anche la mente umana ha una memoria limitata.

Successivamente rende chiara la sua idea in merito al fatto che tutto ciò che può essere calcolato da un essere umano allora può essere calcolato anche da una macchina. Secondo la sua visione infatti qualsiasi procedura matematica poteva essere scomposta in una serie di passi elementari molto semplici. Questi passi oltre che da un essere umano potevano essere svolti anche da una macchina in modo automatico. Essa prese il nome di *automatic machine*, più semplicemente *a-machine* o, come sicuramente è più universalmente nota, *Macchina di Turing*.

Questa macchina, almeno per il momento, non venne concepita al fine di creare una vera e propria macchina fisica, essa infatti nacque al solo scopo di realizzare un'astrazione matematica dalla quale Turing riuscì a stabilire la seguente tesi:

Tesi 1. *Se una funzione sui numeri reali è umanamente calcolabile, allora può essere calcolata anche da una Macchina di Turing.*

Più formalmente possiamo dire che:

Tesi 2. *La classe delle funzioni calcolabili coincide con quella delle funzioni calcolabili da una macchina di Turing.*

È doveroso ricordare che si sta presentando una tesi e non è presente alcuna dimostrazione che ne garantisca l'assoluta veridicità, tuttavia essa è universalmente accettata.

1.4.2 Confutazione dell'Entscheidungsproblem

Una volta stabilito che la classe delle funzioni Turing calcolabili coincide con quella delle funzioni calcolabili, si hanno finalmente tutti gli strumenti per dimostrare che una funzione di decisione come quella richiesta da Hilbert non possa esistere. Viene dunque dimostrato che non può esistere un algoritmo in grado di stabilire se un enunciato nella logica del primo ordine sia accettato o meno dagli assiomi di partenza.

Dimostrazione. Supponiamo per assurdo che esista una macchina di Turing D a due parametri x e y tale per cui se x(y) termina restituisce 1, altrimenti restituisce 0.

Sia L una macchina di Turing che non termina mai. Definiamo ora una macchina di Turing H ad un parametro c, essa restituisce L se D(c,c) termina, altrimenti se non termina restituisce 1.

A questo punto possiamo porre come parametro di H se stesso. Così facendo però noteremmo una grossa contraddizione, H(H) termina soltanto se H(H) non termina, il che è assurdo e di conseguenza la macchina di Turing D non può esistere. □

Osservazione 3. Dato che non può esistere una macchina di Turing del genere e dato che l'insieme delle funzioni Turing calcolabili coincide con quello delle funzioni calcolabili, allora non può esistere un algoritmo come quello richiesto da Hilbert.

1.5 Lettera di Gödel a von Neumann

Il 20 marzo 1956 Kurt Gödel scrisse una lettera a John von Neumann [1] nella quale gli pose un'interessante quesito destinato a restare nella storia. Nella lettera Gödel dice di essere interessato all'opinione di von Neumann in merito a una domanda che si stava ponendo. Il quesito era il seguente:

Problema 1. *Data una formula F nella logica del primo ordine di lunghezza n (lunghezza = numero di simboli), quanti passi al massimo sono richiesti da una macchina per stabilire se, per F, esiste almeno una combinazione di simboli per il quale risulta vera?*

Gödel era ottimista sulla possibilità che potesse esistere un algoritmo in grado di dare una risposta in tempi lineari o quadratici. Purtroppo, quando Gödel inviò questa lettera, von Neumann morì di cancro poco dopo e non poté mai dare la sua opinione sul problema.

A più di sessant'anni di distanza non sappiano ancora se esista o meno un algoritmo in grado di dare una risposta in tempi lineari o quadratici, in ogni caso molti sono gli studi nati attorno a questo problema, che prese il nome di *SAT problem*.

1.6 SAT problem

Riuscire a trovare una risposta al *SAT problem* è un obiettivo di forte interesse da parte di molti: tante sono le competizioni nate a questo scopo. Una delle più note è la *SAT Competition*[1]. L'interesse per questo problema è dovuto al fatto che, oltre alla possibilità di guadagnare un milione di dollari, si arriverebbe a risolvere uno dei sette problemi del millennio e se si trovasse un algoritmo polinomiale in grado di risolvere il problema SAT, allora si dimostrerebbe che P è uguale a NP, provocando una serie di effetti a catena che porterebbero inevitabilmente alla fine del mondo informatico così come lo conosciamo e all'apertura di una nuova era dell'informatica.

Detto ciò si può pienamente comprendere perché trovare una soluzione a tale problema susciti molto interesse da parte di tutti. Prima di procedere è doveroso comprendere quali sono gli avvenimenti che hanno portato questo problema a ricoprire un ruolo così rilevante nel panorama informatico e per farlo è necessario essere a conoscenza di alcuni importanti avvenimenti storici.

[1] http://www.satcompetition.org/

Nel 1965 Juris Hartmanis e Richard Stearns pubblicarono *On the Computational Complexity of Algorithms* [7]. Con questa pubblicazione per la prima volta venne affrontato in modo rigoroso il tema della complessità computazionale degli algoritmi, loro definirono il concetto di complessità in termini di tempo e di spazio e dimostrarono il *Hierarchy Theorem*. A livello informale questo teorema afferma che maggiore è il tempo o maggiore è lo spazio che viene dato ad una macchina di Turing per essere computata, maggiori saranno i problemi che riuscirà a risolvere. Grazie a queste scoperte venne loro assegnato il premio Turing nel 1993.

Nello stesso anno Alan Cobham e Jack Edmonds definirono l'esistenza della classe dei cosiddetti algoritmi *realistically solvable* [5], cioè tutti quei problemi decisionali che possono essere computati in tempo polinomiale rispetto alla dimensione dei dati di input, cioè in un tempo $O(n^c)$ con n pari alla dimensione dei dati in input e c costante. Questa classe di algoritmi prese il nome di P, iniziale di *Polynomial*.

Una pubblicazione di fondamentale importanza fu *The Complexity of Theorem Proving Procedures* [6], pubblicata da Stephen Cook nel 1971 e grazie a cui ricevette nel 1982 il premio Turing.

In questa pubblicazione Cook definì per la prima volta in modo rigoroso la classe NP dando una definizione e facendo uso del concetto di macchina di Turing non deterministica. A livello informale la si può considerare come la classe contenente tutti quei problemi decisionali per il quale ogni istanza di soluzione può essere verificata tramite un algoritmo che opera in tempo polinomiale. Una definizione più formale e rigorosa di questa classe è fornita nei capitoli successivi di questo elaborato.

Cook inoltre definì il concetto di NP-completezza e dimostrò che il problema SAT è NP-completo. Il concetto di NP-completezza e la dimostrazione della NP-completezza di SAT verranno affrontati approfonditamente nei capitoli successivi. Quello che al momento è importante sapere è che grazie alla consapevolezza che SAT è NP-completo, se si dovesse riuscire a dimostrare che SAT appartiene alla classe P, allora automaticamente verrebbe dimostrato che $P = NP$. Al momento non esiste nessun algoritmo polinomiale per decidere un problema NP completo, nè una dimostrazione che tale algoritmo non possa esistere. A partire da ciò si può comprendere quanto la scoperta di Cook sia di fondamentale importanza nel panorama informatico.

A livello puramente storico è giusto ricordare che in realtà Stephen Cook non fu il primo a raggiungere questa conclusione, bensì il matematico russo Leonid Levin, tuttavia la sua pubblicazione fu tradotta dal russo all'inglese anni dopo rispetto a quando Cook presentò la sua pubblicazione. Per questo motivo, dato che entrambi arrivarono alla medesima conclusione in modo del tutto indipendente, il teorema che prova la NP-completezza di SAT viene tutt'ora chiamato *Teorema di Cook-Levin*.

2

Macchina di Turing deterministica e non deterministica

Come anticipato nel capitolo precedente, la Macchina di Turing fu ideata per scopi puramente teorici. Il suo creatore fu Alan Turing, egli la trattò in modo approfondito nella sua pubblicazione *On Computable Numbers, with an Application to the Entscheidungsproblem* del 1937.

2.1 Computing machines

La *Computing machine* viene comparata da Turing ad un essere umano nel processo di computazione di un numero reale, secondo lui infatti una macchina era in grado di risolvere un problema su numeri finiti al pari di un essere umano.

La *Computing machine* è composta da un numero finito di stati $q_0, q_1, q_2 \dots$ che sono chiamati *m-configurations*, durante l'esecuzione la computazione si potrà trovare solo in uno di questi stati.

Nella macchina è presente un nastro che è il corrispettivo di un foglio di carta per un essere umano. Esso è suddiviso in tante sezioni chiamate quadrati sui quali è possibile memorizzare uno è un solo simbolo, l'insieme di simboli è limitato è deve essere specificato nella *Computing machine*.

In ogni momento la macchina è in grado di leggere uno e un solo quadrato, di conseguenza è in grado di leggere solo un simbolo alla volta. Il quadrato viene chiamato *scanned square* e il simbolo *scanned symbol*.

Il comportamento della macchina è determinato dalla *m-configuration* corrente e dallo *scanned symbol*. Nel caso nel quale lo *scanned square* è vuoto e di conseguenza non contiene nessun simbolo, allora ne verrà scritto al suo interno uno predefinito.

Alla macchina è concesso di cambiare *scanned square*, ma soltanto spostandosi di un quadrato a destra o di uno a sinistra, quando viene effettuata questa operazioni la macchina può modificare lo *scanned symbol* con un altro, inoltre può modificare la *m-configurations* corrente.

Le sequenze di simboli sul nastro compongono dei numeri reali, alcuni di essi fanno parte del risultato, altri invece sono utilizzati solamente come bozza e possono essere liberamente cancellati.

In accordo al meccanismo di funzionamento appena descritto è possibile catalogare le diverse tipologie di macchina in base a queste categorie:

Automatic machines Se per ogni stadio il movimento della macchina è completamente determinato dalla configurazione, allora essa viene chiamata *a-machine*. Tuttavia possono essere definiti intenzionalmente dei movimenti che sono solo parzialmente determinati dalla configurazione. In tal caso quando la macchina raggiunge una situazione che non sa come gestire, essa entra in stallo e diventa necessario l'intervento di un operatore esterno.

Computing machines Se una *a-machine* stampa due tipi di simboli, nel quale il primo tipo chiamato *figures* consiste dei simboli 0 e 1, e tutti gli altri simboli appartengono al secondo gruppo e vengono chiamati appunto simboli del secondo tipo, allora la *a-machine* prende il nome di *c-machine*.

Circular and circle-free machines Se una *c-machine* scrive sempre sul nastro un numero finito di simboli del primo tipo, allora essa prende il nome di *circular*, altrimenti di *circle-free*. Una macchina si dice circolare se raggiunge una configurazione oltre il quale non si potrà più muovere o in caso continuerà a muoversi ma solo stampando simboli del secondo tipo.

2.2 Macchina di Turing deterministica

In questa sezione e in quella successiva sono date le principali definizioni in merito alla MdT deterministica e non deterministica. Tuttavia per una comprensione più completa dell'argomento si può fare riferimento al libro [11].

Ogni macchina di Turing deterministica è composta da un insieme finito di istruzioni, ognuna di esse è definita da una quintupla $\langle q, s, q', s', x\rangle$.

- **q**: stato confrontato con la *m-configuration* corrente;
- **s**: simbolo confrontato con quello letto dalla testina;
- **q'**: nuovo stato che andrà a sostituire quello precedente;
- **s'**: nuovo simbolo che andrà a sostituire quello precedente;
- **x**: direzione in cui si dovrà muovere la testina, può essere verso sinistra (L), destra (R) o restare fermo (S).

Definizione 2. Una *funzione di transizione* $\delta : Q \times \Sigma \to Q \times \Sigma \times \{R, L, S\}$ è una funzione che, dati uno stato e un simbolo di ingresso, restituisce il nuovo stato, il nuovo simbolo e la direzione.

Si tratta di una funzione parziale in quanto, in conformità alla definizione, non è garantito che per ogni corrispondenza stato-simbolo esista una corrispondenza.

Il comportamento di questa funzione può essere anche espresso tramite la *matrice funzionale*.

	q_0	...	q_i
s_0	$q_0\ s_k$ L		
...			
s_j	$q_k\ s_l$ R		$q_j\ s_r$ R

Le colonne rappresentano gli stati, mentre le righe i simboli. L'intersezione tra uno stato e un simbolo qualsiasi fornisce quale sarà il nuovo stato, il nuovo simbolo e la direzione.

L'esempio sovrastante è un esempio di matrice funzionale, si può notare come non sia garantito che per ogni combinazione stato simbolo sia presente una corrispondenza.

Si può comprendere che il comportamento della macchina dipende solamente dallo stato nel quale si trova la macchina e dal simbolo letto dalla testina in un dato istante, mentre tutto ciò che si trova alla destra e alla sinistra della testina non determinerà il comportamento della macchina in quel dato istante, tuttavia lo determinerà in quelli successivi.

Definizione 3. La *descrizione istantanea* (ID) fornisce la situazione della macchina in un dato istante, essa è definita da una quadrupla $\langle q, v, s, w \rangle$.

- **q**: rappresenta lo stato in cui si trova la macchina;
- **v**: $v \in \Sigma^\star$, rappresenta tutti i simboli alla sinistra della testina, escludendo però le sequenze infinite di \$;
- **s**: il simbolo letto dalla testina;
- **w**: $w \in \Sigma^\star$, rappresenta tutti i simboli alla destra della testina, escludendo però le sequenze infinite di \$.

Ogni ID rappresenta la situazione istantanea di una macchina in un dato istante, tuttavia una macchina è caratterizzata da un susseguirsi di descrizioni istantanee, a partire da quella di inizio computazione a quella di termine. Il simbolo *successore* $\vdash$, posto tra due descrizione istantanee, afferma che la descrizione istantanea a destra è immediatamente successiva a quella di sinistra.

$$\langle q, v, s, w \rangle \vdash \langle q', v', s', w' \rangle$$

Ciò significa che $\langle q', v', s', w' \rangle$ è immediatamente successiva a $\langle q, v, s, w \rangle$.

Per ogni configurazione istantanea, se la macchina di Turing è deterministica, esiste solo una configurazione istantanea successiva, ovvero:

Teorema 4. *$\alpha \vdash \beta$ e $\alpha \vdash \gamma$ con $\alpha, \beta, \gamma \in ID$, allora $\beta \equiv \gamma$.*

Ciò risulta scontato in quanto a partire da una coppia stato-simbolo può esistere una sola corrispondenza, di conseguenza la macchina potrà assumere un solo comportamento.

Definizione 4. Si dice che una MdT M *decide* un linguaggio $L \subseteq \Sigma^*$ se per ogni $x \in \Sigma^*$ vale che:

- se $x \in \Sigma^*$, allora M con input x termina nello stato *yes*
- se $x \notin \Sigma^*$, allora M con input x termina nello stato *no*

In entrambi i casi la macchina dunque deve terminare. Se una MdT che decide L termina sempre in un numero di passi limitato da $f(|x|)$ allora si dice che M opera in tempo $f(n)$.

2.3 Macchina di Turing non deterministica

Ogni macchina di Turing non deterministica è composta da un insieme finito di istruzioni, ognuna di esse è definita da una quintupla $\langle q, s, q', s', x\rangle$, dove q, s, q', s', x sono già stati definiti nel capitolo 2.2.

Seguendo la definizione una macchina di Turing non deterministica non è altro che una macchina di Turing deterministica nel quale per ogni coppia stato-simbolo possono esistere uno o più azioni applicabili simultaneamente.

Quindi, considerando questa definizione, non si può garantire che a partire da una ID ci sarà sempre la stessa ID successiva, ma bensì un insieme finito di ID successive.

Definizione 5. Si definisce la seguente relazione:

$$\delta : Q \times \Sigma \times Q \times \Sigma \times \{R, L, S\}$$

Che può essere espressa come funzione nel seguente modo:

$$\delta : Q \times \Sigma \to \wp(Q \times \Sigma \times \{R, L, S\}).$$

A differenza della MdT deterministica perciò il risultato non sarà solo uno, bensì un insieme finito. La grandezza di questo insieme determina il numero di azioni possibili che si possono eseguire a partire da una coppia stato-simbolo, di conseguenza il numero di strade non deterministiche che si possono eseguire simultaneamente.

Anche la *matrice funzionale* sarà necessariamente diversa.

	q_0	...	q_i
s_0	$\{q_0\ s_k\ \mathrm{L},\ q_j\ s_0\ \mathrm{S}\}$		
...			
s_j	$\{q_k\ s_l\ \mathrm{R}\}$		$\{q_j\ s_r\ \mathrm{R},\ q_k\ s_t\ \mathrm{L}\}$

Da questa tabella d'esempio è possibile notare che per ogni cella è presente o un insieme finito di azioni o nessun azione.

Per quanto riguarda il simbolo *successore* $\vdash$, a differenza del caso deterministico nel quale per ogni ID era presente una sola ID successiva, per ogni ID potranno essere presenti più ID simultaneamente:

$$\langle q, v, s, w\rangle \vdash_\star \langle q', v', s', w'\rangle.$$

A questo punto risulta chiaro che a differenza del caso precedente non è più vero il teorema 4.

Dato che ogni ID ha più di un successore contemporaneamente, ciò implica inevitabilmente che durante la computazione si genererà un albero delle computazioni non deterministiche.

La radice rappresenta la configurazione iniziale, mentre i figli di ogni nodo rappresentano i diversi flussi di computazione che si sono generati nel momento nel quale il flusso di computazione del nodo padre aveva più ID successivi contemporaneamente. Ogni percorso radice foglia rappresenta un flusso di computazione che può essere eseguito da una MdT deterministica.

Definizione 6. Si dice che una ND-MdT M *decide* un linguaggio $L \subseteq \Sigma^*$ se per ogni $x \in \Sigma^*$ vale che:

- se $x \in L$, allora esiste almeno una computazione non deterministica che termina nello stato *yes*
- se $x \notin L$, allora non esiste alcuna computazione non deterministica che termina nello stato *yes*

Se ogni computazione di una MdT non deterministica che decide L termina sempre in un numero di passi limitato da $f(|x|)$ allora si dice che M opera in tempo $f(n)$. Si osservi che in questo caso se $x \notin L$ allora tutte le computazioni non deterministiche terminano sempre (in al più $f(|x|)$ passi) ma mai con yes.

2.4 Cenni storici

Pochi anni dopo la sua pubblicazione, Turing ed altri matematici furono ingaggiati dal governo Britannico. Erano gli anni della seconda guerra mondiale e il principale scopo della Gran Bretagna era quello di sconfiggere la Germania nazista, a questo scopo Turing fu ingaggiato insieme ad altri matematici al fine di decifrare le comunicazioni naziste. Questa impresa però non era affatto semplice in quanto la Germania aveva costruito un potente cifrario che apparentemente sembrava inviolabile e era caratterizzato da circa 10^{21} combinazioni di cifratura possibili.

Turing tuttavia riusci a costruire un dispositivo elettromeccanico al quale diede il nome di *Bomba* in grado di decifrare i messaggi nazisti cifrati con *Enigma*. Il suo funzionamento era basato sull'approccio della *forza bruta*, cioè venivano testate tutte le possibili permutazioni fino a trovare quella corretta, approccio che per un essere umano risulta impensabile, ma non per una macchina.

Bomba non può essere propriamente considerato il primo computer della storia, esso infatti era un macchinario elettromeccanico di estremo ingegno, tuttavia non era una macchina programmabile. Esso però contribui alla nascita di *Colossus* la quale viene considerata la prima macchina programmabile della storia, nata grazie all'esperienza acquisita con il progetto *Bomba* e a partire dai concetti di *macchina di Turing universale* definiti da Turing. Questa macchina permetteva di decodificare i messaggi personali che venivano trasmessi tra Adolf Hitler e i suoi capi di stato maggiore.

Sfortunatamente col termine della seconda guerra mondiale tutti questi progetti vennero classificati da parte del governo britannico come segreto di stato. Inoltre Turing morì, suicidandosi, pochi anni dopo.

3

P vs NP e SAT

Si supponga di disporre di un semplice zaino e di un certo quantitativo di oggetti, ognuno dei quali caratterizzato da un peso espresso in chilogrammi e da un valore espresso in euro. Supponiamo ora che il nostro obiettivo sia quello di voler inserire all'interno dello zaino alcuni di questi oggetti al fine di ottenere il maggiore valore economico possibile, dato dalla somma del valore di tutti gli oggetti inseriti, senza però superare mai il peso massimo che lo zaino può sopportare.

Apparentemente questo problema, che a primo impatto può non risultare complesso, è di difficile risoluzione. Supponiamo di disporre di un numero di oggetti che non supera le cinque unità. Sapere quale combinazione di oggetti porta al maggiore valore economico possibile potrebbe non risultare difficile. Se però venisse posta la stessa domanda con un quantitativo di oggetti che supera le venti unità, riuscire a trovare una risposta potrebbe non essere banale.

Ogni qual volta vengano selezionati degli oggetti nel tentativo di trovare la migliore combinazione possibile, verificare se il peso complessivo di quest'ultimi non supera il peso massimo consentito è un operazione che può essere svolta molto velocemente. Stabilire invece se quella determinata combinazione è la migliore tra tutte quelle possibili è un'operazione più complessa ed attualmente non sembra esserci altra soluzione se non quella di confrontarla con tutte le altre combinazioni accettabili.

Questo problema è noto come *il problema dello zaino*, o anche chiamato *Knapsack problem*; riassume perfettamente quella che è la difficoltà principale di tutti quei problemi all'interno della classe NP, classe che nelle sezioni successive verrà trattata approfonditamente.

3.1 Classi P e NP

P, NP sono tra le più importanti classi di complessità conosciute, benché si sappia molto su di esse, molte domande riguardanti degli aspetti fondamentali sono ancora senza risposta.

La classe P contiene tutti i problemi decisionali che possono essere risolti con un algoritmo polinomiale. Una definizione più formale è la seguente:

Definizione 7. Un problema decisionale appartiene alla classe P se esiste una macchina di Turing deterministica che lo decide in tempo polinomiale.

P può essere considerata informalmente come la classe contenente tutti quei problemi che possono essere risolti in un tempo ragionevole, cioè tutti quei problemi che hanno un algoritmo di risoluzione efficiente rispetto alla dimensione dei dati.

La classe NP contiene tutti quei problemi decisionali le cui istanze di soluzioni possono essere verificate in tempo polinomiale. Più formalmente:

Definizione 8. Un problema decisionale appartiene alla classe NP se esiste una macchina di Turing non deterministica che lo decide in tempo polinomiale.

A partire da questa definizione si può ottenere il seguente risultato:

Teorema 5. $P \subseteq NP$.

Dimostrazione. Secondo la definizione, un problema decisionale appartiene a NP se esiste un algoritmo che ne verifica un'istanza di soluzione in tempo polinomiale. Dato che, però, tutti i problemi contenuti in P possono essere risolti in tempo polinomiale, definire un algoritmo che verifica se una istanza di soluzione è corretta a partire dal suo algoritmo di risoluzione è un operazione banale. Ne consegue che $P \subseteq NP$. □

Considerando l'esempio del *Knapsack problem*, risulta facile definire una macchina di Turing non deterministica che risolva il problema in tempo polinomiale, dal momento che questa ipotetica macchina di Turing esplorerebbe in modo non deterministico tutte le possibili istanze di soluzione verificando ognuna di esse in tempo polinomiale. In questo modo sarebbe possibile conoscere in tempo polinomiale quale combinazione di oggetti garantisce il maggiore guadagno possibile.

Definizione 9. Siano $A, B \subseteq \mathbb{N}$. A si dice riducibile a B, o anche $A \preceq B$, se esiste una funzione totale ricorsiva f tale che per ogni $x \in \mathbb{N}$:

$$x \in A \iff f(x) \in B$$

Definizione 10. Un problema decisionale si dice NP-Completo se esso appartiene a NP e se ogni problema in NP può essere ridotto a esso.

Osservazione 4. I problemi NP-Completo sono informalmente quelli che potrebbero essere definiti come i "più difficili" problemi in NP, ovvero tutti quei problemi che, se fosse dimostrato che $P \neq NP$, allora sicuramente nessun problema di P sarebbe contenuto nella classe $NP - completo$ e ne conseguirebbe che $P \neq NP - Completo$.

Se si riuscisse a dimostrare che un problema contenuto al suo interno è anche contenuto in P, allora ne deriverebbe che $P = NP$. Invece se si riuscisse a dimostrare che un problema NP-Completo sicuramente non è in P, allora verrebbe dimostrato che $P \neq NP$.

Fino ad oggi non è stato ancora dimostrato se le classi P e NP sono uguali o meno, tuttavia è stato individuato da Stephen Cook e Leonid Levin un problema NP-Completo [2]. Questo problema, come già preannunciato nei capitoli precedenti, è chiamato *SAT problem*.

3.2 SAT problem

Per dimostrare la NP-completezza di SAT bisogna procedere con una serie di passi: innanzittutto bisogna dimostrare la P-completezza di *CIRCUIT VALUE*; poi la NP-completezza di *CIRCUIT SAT* e infine si può terminare con la dimostrazione della NP-completezza di SAT. Per quanto atipico possa essere, in questa dimostrazione è necessario procedere dimostrando che il problema visto dal punto di vista circuitale è NP-completo, solo una volta dimostrato si potrà provare che anche il problema dal punto di vista puramente matematico è NP-completo.

CIRCUIT VALUE Riceve in input un circuito di porte *AND*, *OR* e *NOT* e una combinazione di valori in ingresso. L'obiettivo è stabilire se il circuito in ingresso per la combinazione di valori in ingresso restituisce *true*.

CIRCUIT SAT Riceve in input un circuito di porte *AND*, *OR* e *NOT*. L'obiettivo è stabilire se esiste una qualche combinazione di valori in ingresso tale per cui il circuito in input restituisce *true*.

Teorema 6. *CIRCUIT VALUE è P-completo.*

Dimostrazione. Per prima cosa bisogna dimostrare che *CIRCUIT VALUE* è in P. Dati i valori di input si percorre il circuito valutando l'output di ogni porta fino ad arrivare all'uscita del circuito che determina l'output. In questo modo è stato dimostrato che *CIRCUIT VALUE* è in P.

L'obiettivo ora è dimostrare che è P-completo. Ciò si ottiene dimostrando che ogni problema in P è riconducibile ad esso.

Sia $L \in P$, allora esiste una 1-MdT M che decide L in tempo polinomiale.

L'obiettivo è ottenere un circuito F tale che, con $x \in \Sigma^*$, $x \in L$ sse $F(x) = true$. Per ottenere il circuito dobbiamo studiare la computazione della macchina M. Per fare ciò deve essere realizzata una matrice T $k * k$, con k pari al numero massimo di passi che svolge M per una certa grandezza di x. Una volta realizzata deve essere simulata la configurazione all'i-esimo passo per ogni riga i.

A questo punto si è in grado di osservare che, per ogni cella $T_{i,j}$, essa può essere calcolata a partire dalle celle $T_{i-1,j-1}$, $T_{i-1,j}$, $T_{i-1,j+1}$ e a partire da queste informazioni si è in grado di scrivere delle funzioni booleane che calcolano la codifica di $T_{i,j}$. Infine il tutto può essere composto per realizzare un circuito che simula M. □

Teorema 7. *CIRCUIT SAT è NP-completo.*

Dimostrazione. Per prima cosa bisogna dimostrare che *CIRCUIT SAT* è in NP. Per raggiungere questo scopo bisogna definire una ND k-MdT che risolve SAT generando diverse strade non deterministiche allo scopo di simulare tutti i possibili input del circuito. Ognuna di queste strade è un'istanza di *CIRCUIT VALUE* e in questo modo è stato dimostrato che *CIRCUIT SAT* è in NP.

A questo punto bisogna dimostrare che *CIRCUIT SAT* è NP-completo. Questa dimostrazione ricalca quella inerente a *CIRCUIT VALUE*.

Sia $L \in NP$, assumiamo che L sia decisa da una ND k-MdT e che il suo grado di non determinismo sia pari a 2. A questo punto si può ripetere il ragionamento adottato nel *CIRCUIT VALUE* aggiungendo in ogni porta un input che definisce il grado di non determinismo. □

Teorema 8. *SAT è NP-completo.*

Dimostrazione. Per dimostrare la NP-completezza di SAT è necessario ridurre CIRCUIT-SAT a SAT. Per fare ciò è necessario trovare una formula logica in CNF equivalente al circuito dato. Per il teorema di De Morgan si può assumere che ogni circuito abbia solo porte *or* e *not*. Sono presenti solo tre tipi di nodi: i nodi *or* hanno due archi entranti, ogni nodo *not* uno e ogni nodo di input zero. A questo punto bisogna assegnare una variabile di output ad ogni nodo del circuito:

- **or:** $Y \iff A \vee B$ è una funzione dove Y rappresenta la variabile di output e A, B le variabili di output dei due nodi dal quale partono gli archi entranti;
- **not:** $Y \iff \neg A$ è una funzione dove Y rappresenta la variabile di output e A la variabile di output del nodo dal quale parte l'arco entrante;
- **input:** l'output corrisponde esattamente all'input.

La formula finale ottenuta è equi-soddisfacibile al circuito finale. □

4

Simulatore di Macchine di Turing non deterministiche

Il contenuto della tesi è stato quello di realizzare concretamente un simulatore grafico, multipiattaforma di MdT non deterministiche.

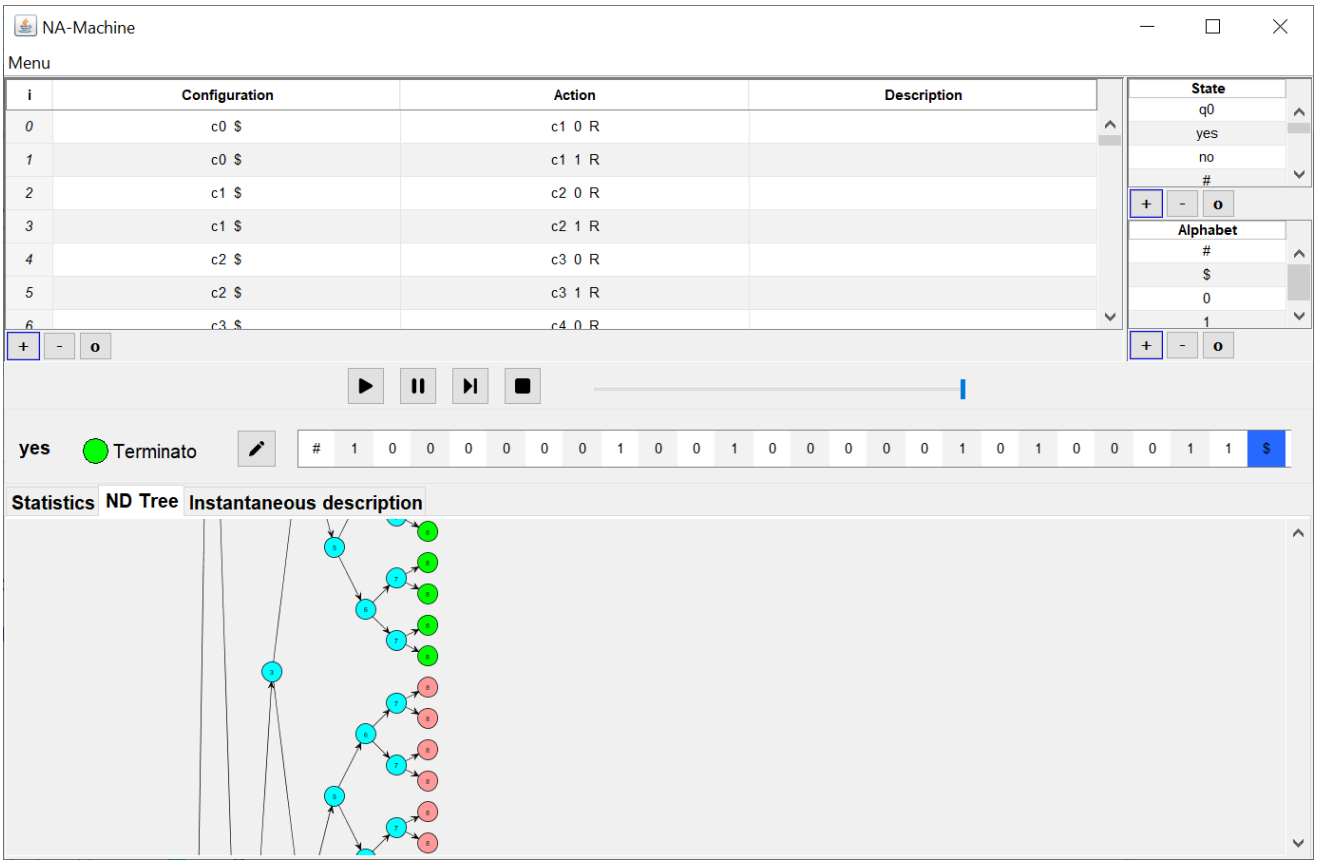

Figura 4.1: Interfaccia simulatore

È stato realizzando utilizzando la piattaforma Java e per la libreria per l'ambiente grafico Swing. Ciò rende il simulatore multi piattaforma. Questo simulatore prende il nome di *NA-Machine* ed è disponibile in `https://github.com/francescodemartino/NA-Machine`.

4.1 Suddivisione in sezioni

Il programma è strutturato in quattro diverse sezioni. Partendo dall'alto verso il basso è presente la sezione riguardante la gestione delle istruzioni, successivamente la sezione per il controllo dell'esecuzione del nastro. In seguito la sezione per visualizzare e modificare una singola configurazione del nastro e in conclusione la sezione atta a visualizzare l'andamento complessivo del nastro.

4.1.1 Gestione delle istruzioni

Quest'area permette di gestire in modo completo le istruzioni ed è suddivisa in tre sotto aree.

States Viene data la possibilità di aggiungere e rimuovere stati. All'avvio del programma vengono caricati automaticamente gli stati *q0*, *yes* e *no*. Essi possono essere liberamente rimossi, tuttavia questa operazione è sconsigliata in quanto *q0* è lo stato di partenza per qualsiasi nastro, mentre gli stati *yes* e *no* godono di una particolare attenzione a livello grafico. Questo punto verrà esposto più nel dettaglio nei paragrafi successivi.

Alphabets Come per gli stati, in questa sezione viene data la possibilità di rimuovere e aggiungere caratteri dell'alfabeto. Vengono precaricati i caratteri # e *$*. Come nel caso degli stati possono essere liberamente rimossi, tuttavia questa operazione è sconsigliata in quanto il carattere # è utilizzato come carattere di partenza per qualsiasi nastro, mentre il *$* è il carattere *blank*.

Instructions Viene data la possibilità di aggiungere e rimuovere istruzioni. Per ogni istruzione può essere aggiunto un commento che però non influisce in alcun modo con l'esecuzione della macchina di Turing. A differenza dei simulatori delle macchine di Turing deterministiche viene data la possibilità di aggiungere istruzioni che hanno lo stesso stato e simbolo dell'alfabeto di partenza.

4.1.2 Controllo esecuzione del nastro

In questa sezione viene data la possibilità di controllare l'esecuzione della macchina di Turing non deterministica. Essa è composta da quattro bottoni e uno slider. Per quanto riguarda i bottoni, partendo da sinistra verso destra, essi permettono di effettuare le seguenti azioni: avviare l'interpretazione; metterla in pausa; eseguire un singolo passo; terminare l'esecuzione.

Lo slider invece dà la possibilità di regolare la velocità d'esecuzione dell'interprete.

4.1.3 Visualizzazione e modifica del nastro

Questa sezione offre la possibilità di visualizzare una sola configurazione del nastro alla volta e in particolare la visualizzazione dello stato corrente, dello stato d'esecuzione corrente, della visualizzazione completa del nastro per quella configurazione e la relativa posizione della testina per esso. Lo stato d'esecuzione viene specificato sia tramite una codifica testuale, sia attraverso l'uso di una bolla colorata. La corrispondenza tra il testo e il relativo colore verrà spiegata più approfonditamente nel paragrafo successivo.

Oltre alla visualizzazione viene offerta la possibilità di modificare il nastro attraverso due metodi: il primo mediante il bottone con l'icona della matita, se premuto permette di modificare interamente i simboli presenti sul nastro; il secondo consiste nell'effettuare il doppio click su una qualsiasi cella del nastro, il che darà la possibilità di modificare il simbolo di quella cella.

Tutte le operazioni di modifica possono essere effettuate solamente quando l'interprete non è in esecuzione.

4.1.4 Visualizzazione andamento complessivo

Questa sezione permette di visualizzare il comportamento del nastro nel suo complesso. Essa è divisa in tre sottosezioni, ognuna delle quali mira ad analizzarne un aspetto diverso.

Prima di procedere con la presentazione delle diverse sottosezioni, è necessario definire in che modalità i colori vengono utilizzati al fine di agevolarne la comprensione da parte dell'utente. Come è noto, ogni configurazione del nastro può avere uno stato d'esecuzione differente. Dato che quest'ultimo ha un ruolo rilevante nella comprensione dell'andamento complessivo del nastro, è stata affiancata alla codifica testuale dello stato di esecuzione una codifica basata sui colori:

- **bianco:** la configurazione è attiva
- **azzurro:** la configurazione è stata clonata
- **verde:** la configurazione ha terminato la sua esecuzione e l'ha conclusa nello stato *yes*
- **rosso:** la configurazione ha terminato la sua esecuzione e l'ha conclusa nello stato *no*
- **giallo:** la configurazione ha terminato la sua esecuzione e l'ha conclusa in un qualsiasi stato diverso da *yes* e *no*
- **arancione:** la configurazione ha terminato la sua esecuzione sforando il nastro, ossia la testina si è spostata alla sinistra della cella di partenza del nastro

Le tre sottosezioni sono trattate nei paragrafi esposti qui di seguito.

Statistics In questa sezione vengono presentate le informazioni di carattere generale riguardanti l'andamento del nastro. Il primo valore che viene presentato riguarda il numero di step eseguiti dall'interprete. Successivamente vengono esposti i dati riguardanti al numero delle configurazioni che si sono generate.

ND Tree In questa sezione viene esposto l'albero delle computazioni non deterministiche. Ogni configurazione viene rappresentata da un nodo, dentro al quale è scritto il numero del livello che occupa nei confronti dell'albero. Per ogni nodo viene specificato lo stato d'esecuzione corrente della corrispettiva configurazione attraverso la codifica basata sui colori esposta precedentemente.

L'albero può essere ridimensionato tramite l'uso della rotella di scorrimento del mouse. Inoltre, è possibile trascinarlo mediante il mouse all'interno della sua interfaccia.

Cliccando su uno qualsiasi dei nodi dell'albero viene aperta la visualizzazione dettagliata della configurazione corrispondente.

Instantaneous description In questa sezione è presente la lista completa di tutte le configurazioni del nastro. È possibile filtrarla o in base allo stato corrente o in base allo stato d'esecuzione corrente. Possono essere effettuate alcune operazioni di modifica sulle configurazioni direttamente da questa interfaccia. Comunque, se si vogliono fare delle operazioni di modifica con maggiore precisione è consigliato aprire la visualizzazione dettagliata.

4.2 Limiti computazionali

Questo interprete, benché non imponga nessun limite al numero di configurazioni che possono essere gestite simultaneamente, non è progettato per elaborare carichi computazionali troppo elevati. Questo aspetto comporta che, sebbene non sia possibile definire un tetto massimo di configurazioni eseguibili oltre il quale il simulatore mostra chiare difficoltà nell'elaborazione, poiché il tetto massimo è fortemente correlato alla potenza di calcolo del computer sul quale viene eseguito il simulatore, è fortemente consigliato simulare macchine di Turing che non producano un numero di configurazioni esageratamente elevato.

5

3SAT Solver non deterministico

Il SAT problem, come già esposto nei capitoli precedenti, è un problema di altissima rilevanza nel mondo dell'informatica teorica. Tuttavia questo capitolo non intende concentrarsi ulteriormente su questo problema, bensì su un'implementazione di un 3SAT Solver non deterministico sviluppato appositamente per la *NA-Machine*.

Un SAT solver è un qualsiasi programma che ha come unico scopo stabilire se esista una combinazione di valori tale per cui una formula logica del primo ordine ha una soluzione o meno. Un 3SAT Solver ha lo stesso scopo di un SAT Solver, i due differiscono però sulla sintassi della formula logica che accettano in input. Un SAT Solver è in grado di interpretare qualsiasi formula logica del primo ordine, un 3SAT Solver invece accetta solo formule logiche che rispettano una particolare sintassi:

$$(\ell_1^1 \vee \ell_2^1 \vee \ell_3^1) \wedge \cdots \wedge (\ell_1^k \vee \ell_2^k \vee \ell_3^k)$$

ove ogni ℓ_j^i è una variabile o la negazione di una variabile.

La sintassi sovraesposta è l'unica che può essere accettata, è composta da una serie di microespressioni che possono essere separate solo dall'and. Ognuna di queste microespressioni è obbligatoriamente composta da tre letterali separate da due or, eventualmente ognuna di queste variabili può essere singolarmente negata. Qualsiasi altra formulazione logica non può essere accetta da un 3SAT solver.

Benché questa formulazione abbia una sintassi limitata rispetto a quella tradizionale, è stato dimostrato che SAT è riducibile a 3SAT e che 3SAT appartiene a NP, ciò lo rende di conseguenza *NP-completo* [12]. Questo risulta un grande vantaggio in quanto la formulazione 3SAT è molto più facile da interpretare rispetto a quella tradizionale.

Il 3SAT Solver esposto in questo capitolo è non deterministico, tuttavia sia le soluzioni deterministiche che le non deterministiche sono in grado di fornire gli stessi risultati, ciò che cambia è il tempo impiegato per farlo.

5.1 Sintassi

L'alfabeto della macchina di Turing è composto solamente dai caratteri *$*, *0* e *1*. Questo significa che l'input della macchina di Turing necessariamente non può coincidere con quello del 3SAT, ma dovrà essere adattato ad un alfabeto molto limitato.

Al fine di comprendere come trasformare una qualsiasi formula 3SAT nella corrispettiva formulazione accettata dalla macchia di Turing, si utilizzerà la seguente formula come esempio:

$$(x \vee \neg y \vee \neg z) \wedge (\neg x \vee z \vee t) \wedge (x \vee y \vee \neg t)$$

La formulazione 3SAT, dato che rispetta una sintassi molto rigida, può essere compresa anche omettendo parentesi, or e and:

$$x\ \neg y\ \neg z\ \neg x\ z\ t\ x\ y\ \neg t$$

Questa scrittura non perde alcun tipo di informazione in quanto si è in grado di comprendere dove poter inserire nuovamente parentesi, or e and.

Il passaggio successivo consiste nel convertire la formula utilizzando solamente i caratteri 0 e 1:

0000 1001 1010 1000 0010 0011 0000 0001 1011

Ogni variabile e il corrispettivo segno sono convertiti in un blocco composto sempre da quattro simboli.

Il primo simbolo definisce il segno: 1 significa che è presente il simbolo di negato, 0 invece che non è presente. I tre simboli successivi invece definiscono la variabile.

La possibilità di poter inserire solo tre simboli implica come conseguenza che è possibile definire al massimo otto variabili, da 000 a 111. Nell'esempio precedente x è stato trasformato in 000, y in 001, z in 010 e t in 011.

Nell'esempio precedente sono presenti degli spazi separatori tra ogni blocco al fine di comprendere meglio la modalità nel quale deve essere effettuata la conversione. Tuttavia nella stringa finale devono essere omessi:

0000100110101000001000110000000011011

La conversione deve essere effettuata sempre in questa modalità, se erroneamente non venissero inseriti il numero corretto di 0 e 1 l'esecuzione non segnalerebbe nessun errore, tuttavia nella maggior parte dei casi l'esecuzione terminerebbe in degli stati diversi da quelli attesi.

5.2 Esecuzione

Per eseguire il 3SAT Solver è necessario inserire la formula sul nastro alla destra del simbolo #, non deve essere presente nessun altro simbolo, altrimenti l'esecuzione sarebbe compromessa.

Poco dopo l'inizio dell'esecuzione della macchina di Turing viene generato l'albero non deterministico. I nodi all'ottavo livello dell'albero si riferiscono alle configurazioni che hanno l'obiettivo di testare tutte le possibili combinazioni delle variabili, successivamente ognuna di esse termina in momenti diversi o nello stato *yes* o in quello *no*. Se l'esecuzione di una configurazione termina nello stato *yes* ciò significa che la combinazione di simboli che è stata testata in quella configurazione rende vera la formula, altrimenti se termina nello stato *no* significa che quella determinata configurazione non rende vera la formula.

5.3 Funzionamento

Il funzionamento del 3SAT Solver è diviso in tre fasi principali. La prima fase consiste nella generazione di tutte le combinazioni possibili, nella seconda fase ogni configurazione viene impostata in modo tale da affrontare la fase successiva, nell'ultima fase viene testato se una certa combinazione rende vera o meno la formula.

Fase 1 Nella prima fase la macchina di Turing crea l'albero non deterministico con tutte le possibili combinazioni con il quale le otto variabili possono essere combinate, ciò significa che viene generato un albero con 256 foglie, una per ogni combinazione. La stringa contenente la combinazione viene posta alla destra della formula ed è separata dalla formula da un $, questa stringa è composta da una sequenza di caratteri che può essere o 0 o 1. 0 equivale a $false$ e 1 equivale a $true$.

Fase 2 Al termine della creazione dell'albero per ogni foglia, più precisamente per ogni configurazione ancora attiva, è assegnata una combinazione di variabili, ogni configurazione deve verificare se la formula è soddisfatta per quella data combinazione. Per raggiungere questo scopo però avviene un passaggio intermedio, esso caratterizza la seconda fase.

L'obiettivo è quello di ottenere la formula logica dove al posto delle variabili sono inseriti i corrispettivi valori booleani, tenendo anche in considerazione il fatto che le variabili possono essere negate. La formula logica così espressa viene codificata sostituendo al simbolo del segno della formula il corrispettivo valore booleano. A titolo d'esempio la seguente formula:

$$000010011010100000100011000000011011$$

con la seguente combinazione di caratteri:

$$01001000$$

viene trasformata in questa formula:

$$000000011010100000100011000010011011$$

Si può notare che per ogni blocco da quattro simboli, il primo simbolo corrisponde al valore booleano che assume la variabile di quel blocco per una determinata combinazione. 0 significa che ha valore $false$, 1 che ha valore $true$.

Fase 3 In questa fase viene verificato se ogni combinazione di ogni configurazione soddisfa o meno la formula.

La macchina di Turing in questa fase scandisce il nastro da sinistra verso destra a partire dal primo valore del nastro. Ogni tre blocchi identificano tre variabili separate da un or, perciò se anche solo uno di questi tre ha come primo simbolo del blocco un 1, cioè $true$, allora la macchina di Turing può procedere verificando che anche gli altri tre blocchi successivi hanno almeno un valore pari a 1. Se nessun valore è pari a 1, ciò significa che il valore restituito dalle tre variabili è un $false$, di conseguenza quella combinazione non rende la formula vera e l'esecuzione termina nello stato no. Se l'esecuzione

invece riesce a procedere fino a raggiungere un $, cioè il termine della formula, allora ciò significa che l'esecuzione può terminare nello stato *yes*, in quanto altrimenti si sarebbe dovuto concludere prima nello stato *no*.

5.4 Esercizi d'esempio

A titolo d'esempio in questa sezione verrà esposta la macchina di Turing computata con due formule CNF, una soddisfacibile e una insoddisfacibile.

5.4.1 Caso soddisfacibile

La formula soddisfacibile è la seguente:

$$(x \vee \neg y \vee \neg z) \wedge (\neg x \vee z \vee t) \wedge (x \vee y \vee \neg t)$$

questa formula è stata utilizzata a titolo d'esempio nel capitolo 5.1. Una volta convertita si ottiene la seguente riscrittura:

000010011010100000100011000000011011

Al termine della computazione della MdT si può comprendere dalla sezione *Statistics* che è soddisfacibile, in quanto esiste almeno una configurazione che termina nello stato *yes*.

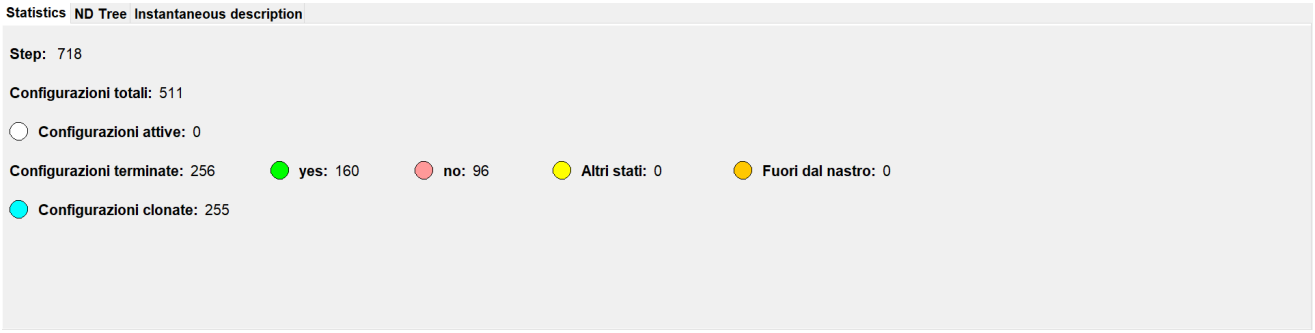

Figura 5.1: Sezione *Statistics* caso soddisfacibile

La figura 5.1 è lo screenshot della sezione *Statistics* al termine della computazione della MdT, mentre la figura 5.2 è lo screenshot di una parte dell'albero delle computazioni non deterministiche nel quale si può vedere che alcune configurazioni sono terminate nello stato *yes* e altre in quello *no*.

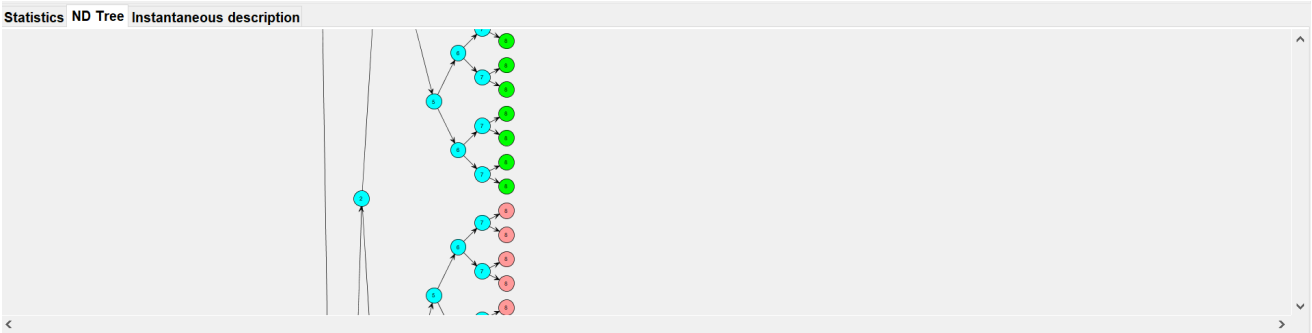

Figura 5.2: Sezione *ND Tree* caso soddisfacibile

5.4.2 Caso insoddisfacibile

La seguente formula invece è insoddisfacibile:

$$(x \vee y \vee z) \wedge (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee \neg y \vee \neg z)$$

$$\wedge(\neg x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

e la sua riscrittura è:

000000010010000000011010000010010010000010

011010100000010010100000011010100010010010100010011101

Al termine dell'esecuzione si può comprendere tramite la sezione *Statistics* che non è presente alcuna configurazione che termina nello stato *yes*, ciò significa che la formula è insoddisfacibile.

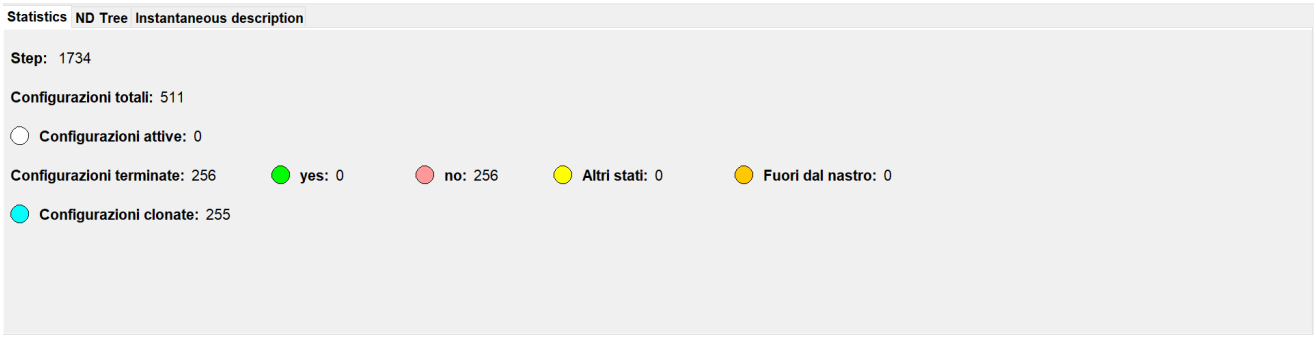

Figura 5.3: Sezione *Statistics* caso insoddisfacibile

Analizzando l'albero delle computazioni non deterministiche si può notare che non è presente alcun nodo verde. La figura 5.4 è lo screenshot di una parte dell'albero dal quale si può notare che i nodi sono solamente rossi. Benché l'albero sia di dimensioni troppo grandi per essere visualizzato per intero in uno screenshot, se si guardasse una qualsiasi altra area dell'albero si vedrebbe che tutti i nodi foglia sono rossi.

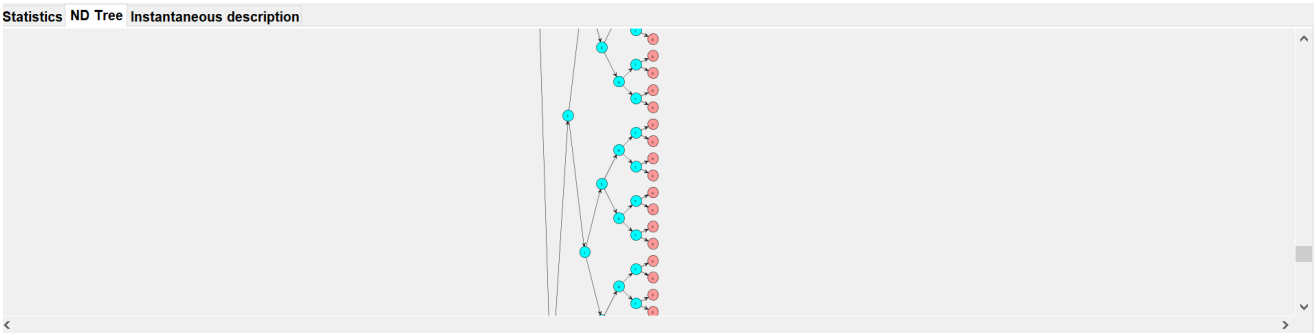

Figura 5.4: Sezione *ND Tree* caso insoddisfacibile

6

Conclusioni

Il tool presentato in questo elaborato lo ho sviluppato al fine di creare un simulatore di MdT non deterministiche di supporto ai corsi di Fondamenti dell'Informatica e in generale per i corsi nel quale viene trattata la computabilità e la complessità computazionale. L'eseguibile e il codice sorgente del programma sono pubblici e on-line. In caso di necessità sono disponibile per modificarlo e migliorarlo.

Bibliografia

[1] Lettera di Gödel a von Neumann. https://rjlipton.wordpress.com/the-gdel-letter/.

[2] P vs NP Problem. http://www.claymath.org/millennium-problems/p-vs-np-problem.

[3] Francesco Berto. *Tutti pazzi per Gödel*. 2008.

[4] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. 58(2):345–363, 1936.

[5] Alan Cobham. The intrinsic computational difficulty of functions. https://www.cs.toronto.edu/~sacook/homepage/cobham_intrinsic.pdf, 1965.

[6] Stephen Cook. The complexity of theorem-proving procedures. pp. 151–158, 1971.

[7] Juris Hartmanis e Richard Stearns. On the computational complexity of algorithms. 1965.

[8] David Hilbert e Wilhelm Ackermann. *Grundzüge der theoretischen Logik*. Springer-Verlag, 1928.

[9] Stephen Cole Kleene. Introduction to Metamathematics. p. 376, 1952.

[10] Stefano Nasini. Il primo teorema di incompletezza di Gödel. http://www-eio.upc.es/~nasini/Blog/PrimoTeoremaIncompletezza.pdf.

[11] Agostino Dovier Roberto Giacobazzi. *Fondamenti dell'informatica*. 2020.

[12] Thomas Jerome Schaefer. The complexity of satisfiability problems. http://www.ccs.neu.edu/home/lieber/courses/csg260/f06/materials/papers/max-sat/p216-schaefer.pdf, 1978.

[13] Alan Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. s2-42(1):230–265, 1937.