Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Matematica

MODELLI ASTRATTI DI CALCOLO: MACCHINE A PUNTATORI

Tesi di Laurea in Informatica

Relatore:
Chiar.mo Prof.
Simone Martini

Presentata da:
Giacomo Borghi

III Sessione
Anno Accademico 2017/2018

Introduzione

Le macchine a puntatori sono modelli di computazione le cui origini risalgono alla nascita dell'informatica: il primo modello, la Macchina di Kolmogorov-Uspensky fu infatti presentato da Andrei N. Kolmogorov nel 1953. Nate come generalizzazione della Macchina di Turing, esse hanno però interessanti peculiarità che le distinguano da questa classica macchina astratta. La più importante riguarda la struttura della memoria: essa, come vedremo, consiste in un grafo che viene modificato, cambiando topologia durante la computazione.

Si tratta di una classe di modelli di computazione che fornisce due principali contributi allo sviluppo dell'informatica. Il primo riguarda il problema di natura logica (ancora non del tutto risolto) relativo alla definizione di algoritmo. Kolmogorov, infatti, in seguito ad una profonda riflessione sul significato di processo algoritmico, ne propone una definizione matematica e rigorosa proprio attraverso il suo modello computazionale [29].

Il secondo contributo riguarda un aspetto più informatico: le macchine a puntatori forniscono una misura della complessità diversa, considerata più realistica, rispetto ad altri modelli di computazione [15]. In questo ambito l'apporto più significativo è dato dalla Macchina di Schönhage, presentata per la prima volta nel 1970, ideata da Arnold Schönhage con il preciso intendo di fornire uno strumento per la teoria della complessità.

Nonostante la loro lunga storia, la letteratura dedicata a queste macchine risulta però poco approfondita e frammentaria. Una delle ragioni principali per questa scarsità di risultati è stata la mancanzadi un loro utilizzo preciso. Ad esempio, nella teoria della complessità viene usata principalmente la Macchina di Turing, mentre per l'analisi degli algoritmi le macchine ad accesso casuale (RAM) [10].

L'obbiettivo di questo elaborato è mettere in luce le caratteristiche e le peculiarità delle classe di macchine a puntatori, mettendole a confronto con questi classici modelli di computazione.

Nel primo capitolo è quindi fornita una descrizione dettagliata di una macchina astratta appartenente a questa classe. Viene presentata la Mac-

china di Schönhage in quanto considerata la più rappresentativa [9] e più studiata [7]. Viene fornita anche una definizione di *space complexity* e *time complexity*.

Nel secondo capitolo la Macchina di Schönhage viene messa a confronto con gli altri modelli. In particolare, ci chiediamo se essa appartenga alla classe di macchine descritta dalla Tesi di Invarianza. Questa tesi afferma, in estrema sintesi, che esiste un insieme di modelli di computazione (di cui fanno parte le Macchine di Turing e le RAM) simulabili a vicenda con un limitato (cioè: polinomiale) utilizzo di spazio e di tempo.

Per fare ciò introduciamo la nozione di Simulazione in Tempo Reale e dimostriamo nel dettaglio che la Macchina di Schönhage può simulare in tempo reale la Macchina di Turing. Vediamo poi alcuni risultati riguardanti l'utilizzo dello spazio nelle Macchine di Schonhage: essi, insieme alla simulazione presentata, suggeriscono che le macchine a puntatori appartengano ad una classe di modelli più potenti rispetto a quella descritta dalla Tesi di Invarianza.

Nel terzo capitolo, infine, presentiamo la Macchina di Kolmogorov-Uspensky ponendo un accento particolare alla riflessione che ne ha portato all'ideazione. Come abbiamo già in parte sottolineato, l'intento di Kolmgorov era quello di fornire una descrizione di modello computazionale il più vicino possibile all'intuizione di algoritmo.

Uspensky definisce questa intuizione *umana*, ovvero l'intuizione che questa macchina (pensata comunque come macchina unicamente astratta) processi le informazioni in uno spazio fisico e in un tempo fisico [29]. Presentiamo quindi un interessante esperimento di *computazione non convenzionale*, settore di ricerca fortemente multidisciplinare che unisce matematica, informatica, chimica, fisica e biologia. Nello specifico viene descritta una implementazione della Macchina di Kolmogorov-Uspensky tramite un particolare organismo, il *Physarum polycephalum*. Essa evidenzierà come, grazie alle loro caratteristiche, le macchine a puntatori si prestino anche a fornire modelli matematici per la comprensione di sistemi fisici.

Indice

Capitolo 1

Le Macchine di Schönhage

Le macchine a puntatori sono modelli computazionali ideati fin dalla nascita dell'informatica teorica. In origine, però, esse non furono concepite come uno strumento per la teoria della complessità, ma semplicemente come una generalizzazione delle Macchine di Turing (MdT in seguito) al fine di studiare la validità della tesi di Church-Turing e provare a fornire una definizione di algoritmo. Con questi obbiettivi i matematici russi Andrei N. Kolmogorov e Vladimir Uspensky presentarono nel 1958 un nuovo modello di computazione [20], ideando quella che fu denominata in seguito Macchina di Kolmogorov-Uspensky (KUM) dove la memoria è rappresentata da un grafo non diretto e finito. Essa si può considerare come il primo esempio di macchina a puntatori. Dimostrarono, inoltre, che questo modello di computazione è equivalente in termini di computabilità alla MdT.

Un altro modello di macchine a puntatori fu introdotto dopo 12 anni, questa volta, con l'intento specifico di fornire uno strumento per lo studio della complessità: nel 1970, Arnold Schönhage definì infatti un macchina a puntatori che chiamò Macchina a Modificazione della Memoria (SMM, da *Storage Modification Machine*) [24]. Schönhage introdusse il suo modello, come scrisse lui stesso, in quanto "possiede una estrema flessibilità e dovrebbe quindi servire come base per una nozione adeguata di complessità computazionale". Questa tesi ha ricevuto valutazioni concordi, come ad esempio quella di Stephen Cook che affermò in merito: "le macchina di Schönhage mi sembra che rappresenti la macchina più generale che possa essere costruita, tra quelle performanti una quantità di lavoro limitato ad ogni *step*" [11].

Sebbene Schönhage abbia introdotto il suo modello senza essere a conoscenza del lavoro di Kolmogorov, le SMM possono essere viste come una generalizzazione delle KUM dove il grafo è diretto e solo il numero di archi uscenti da un nodo è limitato. Schönhage stesso suggerisce di chiamare gli archi, in quanto diretti, *puntatori*, da cui il nome di macchine a puntatori.

Per questo motivo Andreas Blass e Yuri Gurevich affermano [9] che le SMM rappresentano meglio questa classe di macchine rispetto alle KUM.

Con il termine macchine a puntatori, quindi, si possono indicare diversi modelli di computazione; è possibile trovarne un elenco con la rispettiva bibliografia in [6, 7]. Oltre a quelli già citati il più conosciuto è il *Linking Automaton* (LA), ideato da Donald Knuth nel 1968 [18].

Procediamo ora con la definizione di SMM.

1.1 Definizione di SMM

Una macchina di Schönhage consiste in una struttura dinamica, chiamata Δ-*struttura* abbinata ad un programma di controllo che manipola la struttura mentre legge una stringa di input e scrive una stringa di output. Intuitivamente, questa macchina legge (in una sola direzione) una stringa di input e memorizza l'informazione in un multigrafo orientato. Con multigrafo si indica un grafo in cui due nodi possono essere collegati da più di un arco; il grafo è orientato in quanto ogni arco è dotato di una direzione. Gli archi nel grafo sono etichettati da simboli di un alfabeto Δ; ogni elemento nel grafo è indicato (non in modo necessariamente unico) dal percorso che lo connette ad un nodo centrale distinto dagli altri. Le macchine modificano la memoria aggiungendo nuovi elementi e reindirizzando gli archi. Alcuni elementi potrebbero, perciò, risultare irraggiungibili in seguito a delle modifiche.

Formalmente, Δ è un insieme finito, o alfabeto, di *direzioni*. La Δ-struttura è una tripla $S = (X, a, p)$, dove X è un insieme finito di nodi di un grafo; $a \in X$ è un nodo *centrale* distinto; e p è un insieme (con cardinalità $|\Delta|$) di funzioni da X a X indicizzate da elementi di Δ. Ogni $\delta \in \Delta$ definisce, quindi, una mappatura p_δ da X ad X; $p_\delta(b)$ è il nodo a cui punta l'arco che parte da b ed etichettato con δ. Per un esempio si veda Fig. 1.1.

La funzione p può essere estesa con la mappa $p^* : \Delta^* \to X$, dove $\Delta^* = \{W \mid \exists\, n \in \mathbb{N} : W \in \Delta^n\}$ indica l'inseme delle parole in Δ e pensiamo ad ogni parola $W \in \Delta^*$ come ad un percorso lungo la struttura che parte dal nodo centrale. Possiamo quindi definire $p^*(\epsilon) = a$ dove ϵ indica la parola vuota e a, ricordiamo, è il nodo centrale. Ricorsivamente, poi, definiamo $p^*(W\delta) = p_\delta(p^*(W))$. Ogni parola in Δ^* sarà perciò associata ad un elemento di X.

Una SMM è manipolata tramite un programma di controllo finito scritto in un linguaggio di programmazione semplice. In questo linguaggio ci sono due tipi di istruzioni: le istruzioni *comuni* che sono le stesse per tutte le SMM, e le istruzioni *interne* che dipendono dalla Δ-struttura. Le istruzioni comuni sono `input`, `output`, `goto` e `halt`, mentre le interne sono `new`, `set` e `if`. Ogni

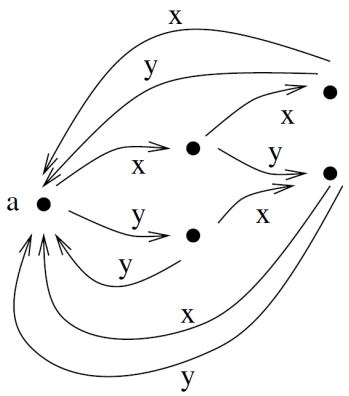


Figura 1.1: Una possibile Δ-struttura per $\Delta = \{x, y\}$

istruzione può avere associata un'etichetta in modo tale che altre istruzioni in un programma possano fare a riferimento a quella istruzione particolare. Se due istruzioni hanno la stessa etichetta la prima nel programma è trattata come l'unica istruzione avente quell'etichetta. Input e output consistono in singole stringhe binarie. Gli unici comandi che manipolano queste stringhe sono `input` e `output`, un simbolo alla volta.

Di seguito una descrizione dell'azione di ciascuna istruzione sullo stato della macchina.

- `input` $\lambda_0\lambda_1$; viene letto un simbolo $\beta \in \{0, 1\}$ dalla string in input. Se $\beta = 0$, il controllo viene trasferito sull'istruzione etichettata con λ_0, se $\beta = 1$ a quella etichettata con λ_1.

- `output` β; intuitivamente β è emesso nell'ambiente durante l'esecuzione di questa istruzione.

- `goto` λ; il controllo viene trasferito all'istruzione con etichetta λ.

- `halt`; questa istruzione induce l'arresto del programma. La macchina si ferma anche se il controllo arriva alla fine del programma.

- `new` W; questa istruzione crea un nuovo nodo y e lo aggiunge ad X. La sua posizione rispetto agli altri nodi e puntatori è determinata da W. Se W è la stringa vuota ϵ, l'istruzione ha l'effetto di creare un nuovo nodo centrale a, con tutti gli archi che puntano dal nuovo nodo al vecchio nodo centrale. Tutti gli altri puntatori rimangono immutati. Se invece W è della forma $U\delta$, con $U \in \Delta^*$, l'istruzione `new` $U\delta$ reindirizza il δ-puntatore dal nodo indicato con U al nuovo nodo y e tutti i puntatori

uscenti da y al nodo originariamente descritto da $U\delta$. Ancora una volta, tutti gli altri puntatori rimangono immutati. Ad esempio, in Fig. 1.2 vediamo come viene modificata la struttura in seguito all'istruzione `new` xy. Viene creato un nuovo nodo raggiungibile seguendo il puntatore y dal nodo indicato con x. Tutti i puntatori uscenti da questo nuovo nodo puntano al nodo precedentemente indicato da xy.

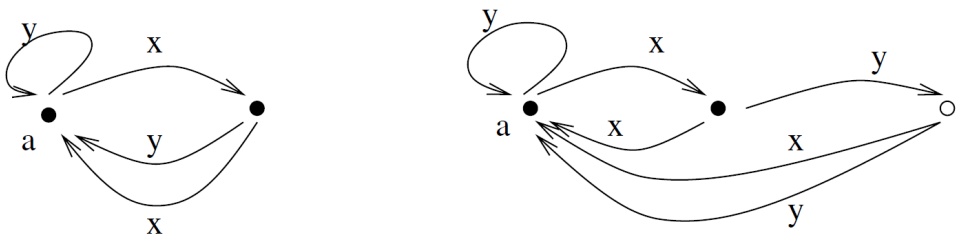


Figura 1.2: `new` xy

– `set` W `to` V; questa istruzione reindirizza un puntatore. Se W è la stringa vuota, il nodo centrale a risulta il nodo indicato da V. Se, invece, $W = U\delta$ il puntatore δ uscente da U viene diretto al nodo indicato da V e nessun altro puntatore è cambiato. In Fig. 1.3 viene mostrato un esempio dell'effetto dell'istruzione `set`.

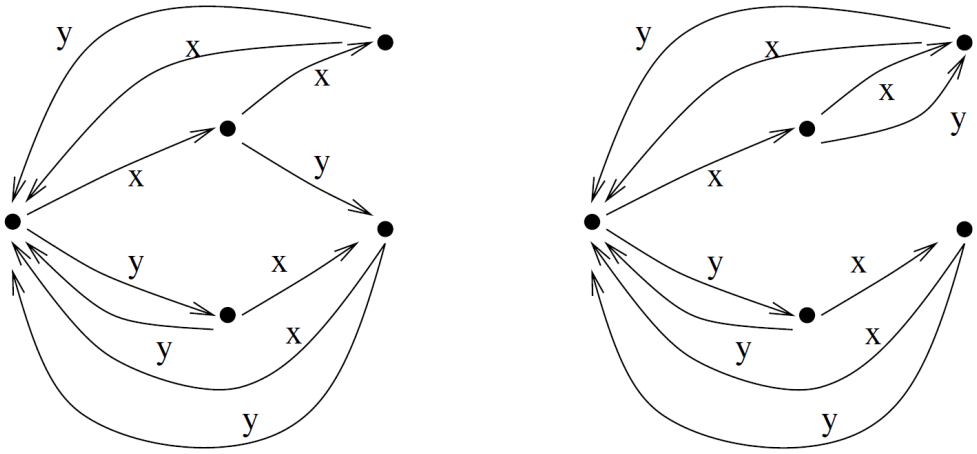


Figura 1.3: `set` xy `to` xx

– `if` $U = V$ `then` σ e `if` $U \neq V$ `then` σ; dove σ è una delle istruzioni precedenti (quindi non una istruzione `if`) che è eseguita se e solo se, rispettivamente, $p^*(U) = p^*(V)$ o $p^*(U) \neq p^*(V)$.

In ogni momento lo *stato* di un SMM è dato dall'input rimanente, dall'output accumulato, dall'istruzione corrente e dalla Δ-struttura. Allo stato iniziale la SMM, l'input rimanente è la stringa di input originale, l'output accumulato è la stringa vuota, l'istruzione corrente è la prima del programma e la Δ-struttura consiste in un solo nodo, quello centrale a con p tale che $p_\delta(a) = a$ per ogni $\delta \in \Delta$. Ovvero, tutti i puntatori del nodo centrale puntano al nodo stesso.

Una *run* di una SMM è una sequenza di stati tali che ogni stato è calcolato dallo stato precedente tramite l'esecuzione dell'istruzione dello stato precedente.

Al fine di investigare le relazioni tra le SMM e i classici modelli di computazione come le MdT e le RAM diamo una precisa definizione di *time* e *space complexity* per le SMM.

1.2 *Time* e *Space complexity* delle SMM

Il tempo usato dalla macchina è il numero di istruzioni eseguite prima dell'arresto della computazione. Lo spazio può invece essere definito in due modi differenti.

Si definisce *massa* il numero di istruzioni `new` eseguite prima dell'arresto, ovvero il numero di nodi creati duranti l'esecuzione. La massa fu introdotta come misura dello spazio da Joseph Y. Halpern in [17]. Si definisce *capacità* della computazione la quantità $dn \log n$, dove n è il numero di nodi creati, d è il numero di puntatori per nodo ($d = |\Delta|$), e il log è il logaritmo in base 2.

L'idea di considerare la capacità come misura dello spazio viene proposta da Allan Borodin in [8]. Con n nodi è possibile avere al più $Q = n^{dn}$ possibili configurazioni della Σ-struttura. Borodin definì lo spazio di controllo (la capacità) come $\log(Q)$, ovvero il logaritmo del numero di possibili configurazioni. È quindi ragionevole, nel caso nelle macchine a puntatori, definire la capacità come $dn \log n$. Dal momento che la definizione standard di spazio per la MdT non tiene conto della grandezza dell'alfabeto del nastro, possiamo, in modo alternativo, definire la capacità (indicata anche come misura logaritmica dello spazio) semplicemente come $n \log n$. Il modo in cui viene definito lo spazio per le SMM è molto importante in quanto modifica, come vedremo, il rapporto di equivalenza con gli altri modelli di computazione.

Diciamo quindi che una macchina ha *time complexity* $t(n)$ se per ogni input di lunghezza n il tempo massimo di esecuzione è $t(n)$. Allo stesso modo si definiscono i termini *mass complexity* e *capacity complexity*.

Prima di confrontare le SMM con gli altri modelli di computazione presentiamo un primo risultato riguardante le SMM: il teorema di compressione dello spazio.

Teorema 1.2.1 (Compressione dello stato)**.** *Per ogni costante $c > 0$ ogni SMM con mass complexity $s_m(n)$ può essere simulata da una qualche SMM con mass complexity $cs_m(n)$.*

È possibile trovare una dimostrazione del teorema in [22]. Presentiamo questo risultato in quanto molto simile ad una importante proprietà delle MdT, l'accelerazione lineare:

Teorema 1.2.2 (Accelerazione lineare)**.** *Per ogni costante $c > 0$ ogni MdT con time complexity t(n) può essere simulata da una qualche MdT con time complexity $ct(n) + n + 2$.*

Sebbene la compressione dello spazio sia possibile usando la nozione di massa come misura dello spazio, non è chiaro se le SMM godano della proprietà di accelerazione lineare delle MdT. Nel prossimo capitolo analizzeremo nel dettaglio la relazione che intercorre tra le SMM, la MdT e l'altro importante modello di computazione: le macchine ad accesso casuale.

Capitolo 2

Tesi di Invarianza

Nella teoria della complessità ci sono due modelli di macchine astratte, in particolare, che si sono affermate come modelli di computazione standard. Il primo, per ragioni storiche, è la Macchina di Turing (MdT), ideata nel 1936 da Alan M. Turing [28]; il secondo è l'insieme delle macchine ad accesso casuale (RAM, *Random Access Machine*) introdotte da Sthepen Cook e Robert A. Reckhow nel 1973 [12]. L'utilizzo principale delle RAM, in quanto molto simili ai computer reali, riguarda l'analisi degli algoritmi, mentre le MdT giocano un ruolo essenziale nella teoria della complessità.

Entrambi i modelli possono essere declinati in diverse varianti, ognuna delle quali con una propria misura del tempo e dello spazio. Nonostante ciò la teoria della complessità preserva la sua robustezza in quanto i vari modelli, sebbene differenti, non portano a nozioni di complessità troppo distanti tra loro. La connessione che tiene legati i vari modelli e che garantisce la stabilità delle classi di complessità fondamentali è nota come Tesi di Invarianza:

Tesi di Invarianza. *Esiste una classe standard di modelli di macchine, che include tutte le varianti delle MdT, tutte le varianti delle RAM e delle RASP con la misura logaritmica dello spazio e del tempo, e anche le RAM e le RASP con misura uniforme per il tempo e logaritmica nello spazio (purché vengano utilizzate solo istruzioni aritmetiche standard di tipo additivo).*

I modelli di macchina appartenenti a questa classe si simulano a vicenda con al più un limite superiore polinomiale per il tempo e uno costante per lo spazio.

La tesi, così come riportata, è stata formulata da Cees Slot e Peter van Emde Boas [27] e può essere vista come una variante della Tesi di Church-Turing applicata alla complessità computazionale. La tesi afferma quindi che i modelli principali di macchine astratte (Le RASP sono un variante delle

RAM) sono equivalenti in quanto è possibile utilizzarne uno per simulare l'altro senza un eccessivo utilizzo delle risorse, ovvero spazio e tempo.

La validità della tesi dipende ovviamente dal modo in cui viene definito l'utilizzo delle risorse. È stato provato [21] che, utilizzando un'adeguata nozione di spazio per le RAM (la *misura logaritmica* citata nella tesi), esse possono simulare le MdT, e viceversa, senza un utilizzo di tempo esponenziale ed utilizzando una quantità di spazio direttamente proporzionale, come indicato dalla tesi.

Sorge spontanea ora una domanda: le macchine a puntatori appartengono alla classe di macchine definita dalla Tesi dell'Invarianza? La risposta, come vedremo, dipende dal tipo di misura di spazio che si utilizza.

Occorre però fare un passo indietro e definire in modo rigoroso cosa si intenda per simulazione. Così facendo infatti possiamo fornire un'adeguata misura del tempo e dello spazio che tale simulazione richiede. Introduciamo quindi la nozione di simulazione in tempo reale, introdotta da Schönhage in [26].

2.1 Simulazione in Tempo Reale

Per comprendere meglio la nozione di simulazione in tempo reale è necessario sottolineare prima due caratteristiche comuni a tutti i modelli di computazione visti finora:

- tutti i passi della computazione vengono compiuti in passi discreti e ogni passo utilizza solo una parte limitata e finita del risultato delle precedenti operazioni;
- tutti i modelli ammettono una memoria potenzialmente illimitata, ma ad ogni istante solo una quantità finita di informazione è presente: si ipotizza l'esistenza di un numero illimitato di elementi, ma essi vengono costruiti basandosi su un insieme finito di tipi e le relazioni che intercorrono tra essi che hanno complessità limitata.

Tali proprietà definiscono la classe delle macchine *atomistiche*, così come individuate da Amir M. Ben-Amram [6]. Di seguito, quindi, la definizione di simulazione in tempo reale e le conseguenti nozioni di riducibilità ed equivalenza in tempo reale.

Definizione 2.1 (Simulazione in tempo reale)**.** Si dice che una macchina M' simula in tempo reale un'altra macchina M, e si denota con $M \xrightarrow{r} M'$, se esiste una costante c tale che per ogni sequenza in input x vale la seguente affermazione: se x causa a M la lettura di un simbolo in input, o la

stampa di un simbolo in output, o l'halt agli istanti $0 = t_0 < t_1 < \ldots < t_l$ rispettivamente, allora x causerà a M' di agire nella stessa maniera rispetto alle stesse istruzioni esterne negli istanti $0 = t'_0 < t'_1 < \ldots < t'_l$ dove $t'_j - t'_{j-1} \leq c(t_j - t_{j-1})$ con $1 \leq j \leq l$.

Definizione 2.2 (Riducibilità ed equivalenza in tempo reale)**.** Per ogni modello di macchine $\mathcal{M}$, $\mathcal{M}'$ la riducibilità in tempo reale $\mathcal{M} \xrightarrow{r} \mathcal{M}'$ è definita tramite la seguente condizione: $\forall\ M \in \mathcal{M}\ \exists$ una macchina $M' \in \mathcal{M}'$ t.c. $M \xrightarrow{r} M'$.

Se $\mathcal{M} \xrightarrow{r} \mathcal{M}'$ e $\mathcal{M}' \xrightarrow{r} \mathcal{M}$ si dice che le due classi di macchine sono equivalenti, indicandolo con $\mathcal{M} \xleftrightarrow{r} \mathcal{M}'$.

Schönhage introdusse la nozione di simulazione in tempo reale in quanto la sua tesi era che ogni modello di computazione atomistico $\mathcal{M}$ fosse riducibile in tempo reale alle SMM; $\forall\, \mathcal{M}$, $\mathcal{M} \xrightarrow{r}$ SMM. Egli dimostrò, a favore della sua tesi, che le SMM sono equivalenti in tempo reale ad un modello di RAM, le *successor* RAM, [26] e che ogni MdT è simulabile in tempo reale da una SMM [25, 26]. Presentiamo in seguito i dettagli dell'implementazione delle MdT tramite SMM.

Prima di procedere è necessario sottolineare che i questi risultati di simulazione in tempo reale rispondono parzialmente alla domanda iniziale, ovvero se le SMM facciano parte della classe di modelli definita dalla Tesi di Invarianza. Essi infatti provano che una simulazione che utilizzi un tempo al più polinomiale è possibile, ma non dicono nulla sul consumo di spazio che questa simulazione comporta. Analizzeremo quindi più avanti le relazioni che intercorrono tra RAM e MdT e SMM per quanto riguarda lo spazio.

2.2 Simulazione di una Macchina di Turing

Per prima cosa, descriviamo la classe di MdT che si vuole simulare. Ci riferiremo a macchine la cui memoria è composta da una o più componenti di dimensioni finite, ognuna delle quali è isomorfa a $\mathbb{Z}^k$ per un certo $k \geq 1$ e a cui possono accedere una o più testine ciascuna. Inizialmente tutta la memoria è vuota e ogni testina è posizionata all'inizio della propria componente. Le testine possono muoversi lungo una coordinata alla volta ed essa deve essere incrementata o decrementata di al più una unità per passo. Viene utilizzato, infine, $\{0, 1\}$ come alfabeto input/output.

Il controllo è dato da una sequenza finita di istruzioni ed etichette. In aggiunta ad input, output, goto, halt abbiamo le seguenti istruzioni:

- head v; con $v \in \mathbb{N}$ e $v \geq 1$. Le istruzioni successive si riferiscono alla testina contrassegnata dal numero v fino all'istruzione head successiva. Dopo l'esecuzione di questo codice, v è la testina attiva;
- write α; $\alpha \in \{0, 1, \beta\}$, dove β sta per *blank*, il carattere "vuoto". Nella cella sotto la testina attiva viene scritto il carattere α;
- read λ_0, λ_1; se la cella sotto la testina attiva contiene 0 il controllo è trasferito all'istruzione etichettata con λ_0. Se la cella sotto la testina contiene 1 il controllo è trasferito all'istruzione etichettata con λ_1. Se la cella è vuota il controllo passa all'istruzione successiva;
- move δ; la testina attiva viene spostata di una posizione in direzione δ, dove δ appartiene all'alfabeto delle direzioni.

Il teorema dimostrato da Schönhage [25, 26] è il seguente:

Teorema 2.2.1. *Ogni macchina di Turing appartenente alla classe appena descritta è simulabile da una SMM appropriata in tempo reale.*

Si osservi che una MdT con n testine e l componenti di memoria diversi di dimensione $k_1, \ldots, k_l$ può essere simulata in tempo reale da un'apposita MdT con un solo componente di memoria isomorfo a Z^d, dove $d = 1 + \max\{k_1, \ldots, k_l\}$. Per la costruzione della simulazione poniamo $d = 2$: una volta dimostrato il procedimento sarà infatti chiaro come generalizzare la tecnica per componenti di memoria di dimensione maggiori.

La MdT che verrà simulata quindi possiede r testine che si muovono su un componente di memoria isomorfo a $\mathbb{Z}^2$. Si stabilisce, perciò, che la SMM simulante userà l'alfabeto $\Delta = \{N, S, W, E, U, P\}$ dove le lettere N, S, W, E rappresentano i punti cardinali e U, P *up* e *down*. Durante la simulazione, i dati e la configurazione della macchina vengono memorizzati in una particolare struttura così composta:

- Struttura piramidale: memorizza il contenuto del piano della MdT e, tramite una sofisticata configurazione geometrica, indica le aree del piano che sono già state visitate da una testina.
- Contatori C_j: ve n'è presente uno per ogni testina ($j = 1, \ldots, r$). I contatori vengono utilizzati per accedere allla posizione della testina nella struttura piramidale;
- Struttura centrale H: contiene il centro A della struttura, i nodi base $B_1, B_2, \ldots, B_r$ e due nodi A_0 e A_1 usati per codificare l'informazione contenuta nel grafo. Serve per selezionare e accedere al contatore della testina attiva della MdT ogni volta che viene simulata una istruzione interna della stessa MdT.

2.2.1 Struttura Piramidale

In ogni momento della simulazione, la struttura piramidale P è una parte finita della struttura infinita $\mathcal{P}$ con nodi $M_k(x, y)$, $(x, y) \in \mathbb{Z}^2$, $k \in \mathbb{N}$. k indica il livello della struttura che si vuole considerare e assume per questa ragione il significato di coordinata verticale. La struttura P si estende quindi su più livelli:

$$\mathcal{L}_k = \{M_k(x, y) \mid (x, y) \in \mathbb{Z}^2\},$$

ognuno dei quali consiste in una griglia rettangolare di nodi connessi dai puntatori N, S, W, E. Da ogni nodo $M_k(x, y)$ partono i quattro puntatori, rispettivamente diretti verso $M_k(x, y+1)$, $M_k(x-1, y)$, $M_k(x, y-1)$, $M_k(x+1, y)$. Il sottoinsieme $\{M_k(\zeta, \eta) \mid |\zeta - x| \leq |\eta - y| \leq 1\}$ insieme ai 24 puntatori che connettono due a due questi nodi è detto *vicinanza* di $M_k(x, y)$. La Fig. 2.1 rappresenta la vicinanza di un nodo, essa è tratta (così come le seguenti) dall'articolo di Schönhage che descrive questa struttura [26].

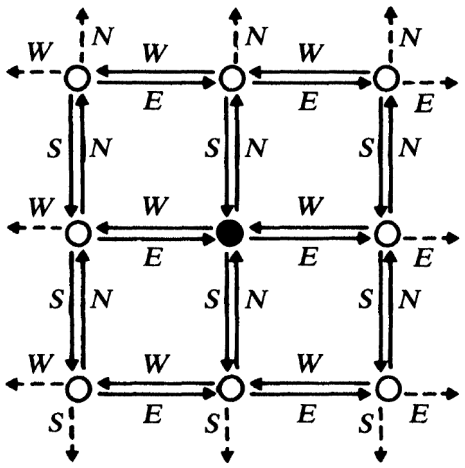


Figura 2.1: Vicinanza del nodo •.

$\mathcal{L}_0$ viene utilizzato per rappresentare il piano della MdT. Essendo questo piano potenzialmente infinito, la struttura piramidale contiene solo un sottoinsieme finito L_0 di $\mathcal{L}_0$ che rappresenta le aree che sono state visitate almeno una volta dalle testine e le rispettive vicinanze. I livelli superiori $\mathcal{L}_1, \mathcal{L}_2 \ldots$ vengono utilizzati per rappresentare il piano su scale ridotte con rapporto lineare $1 : 3 : 9 \ldots$. In particolare $M_{k+1}(x, y)$ corrisponde alla vicinanza del nodo $M_k(3x, 3y)$. La connessione tra i vari livelli viene fornita tramite i puntatoti U e D:

– puntatori U: vanno dal nodo $M_k(x, y)$ al nodo $M_{k+1}(\lfloor (x+1)/3 \rfloor, \lfloor (y+1)/3 \rfloor)$;

– puntatori D: vanno dal nodo $M_{k+1}(x, y)$ al nodo $M_k(3x, 3y)$, $k \geq 1$, mentre sul livello $\mathcal{L}_1$ vengono utilizzati per codificare l'informazione contenuta nelle celle della MdT.

A questo punto è possibile definire la struttura piramidale come

$$P = L_0 \cup L_1 \cup \ldots \cup L_m$$

dove $\forall\, k\ 0 \leq k \leq m$, L_k è un sottoinsieme finito di $\mathcal{L}_k$. Inoltre L_m contiene uno e un solo nodo $M_m(0, 0)$ detto *nodo vertice*. Le principali proprietà di P sono le seguenti:

(a) Ogni puntatore che parte da un nodo $M \in P$ è diretto, o esattamente come lo è in $\mathcal{P}$, o verso il centro A. Nel primo caso il puntatore si dice *correttamente installato* e anche il nodo a cui punta deve appartenere a P (questo non vale per i puntatori D del livello L_0). Altrimenti la destinazione A indica che il puntatore non è stato ancora installato correttamente (si noti che questo può accadere anche se la corretta destinazione appartiene già a P).

(b) Per ogni nodo $M \in P$ (ad eccezione del nodo vertice) sono correttamente installati rispettivamente il puntatore U, la sua destinazione MU, il puntatore D e tutta la vicinanza di MUD (ricordiamo che per $M\delta$ si intende il nodo raggiungibile da M seguendo il puntatore δ).

Dunque, P risulta essere un insieme di singoli elementi piramidali composti da 10 nodi tra loro connessi.

2.2.2 Contatori

I contatori vengono utilizzati per tenere traccia della posizione delle varie testine sul piano. Per tracciare una posizione di coordinate (x, y) utilizziamo un puntatore che indichi al nodo $H_0 = M_0(x, y)$. Poiché però si vogliono mantenere i requisiti necessari alla simulazione in tempo reale è anche necessario l'accesso immediato alle rappresentazioni di livello superiore di H_0 già appartenenti a P. Per questa ragione esistono dei puntatori ai nodi H_1, H_2 $\ldots$, H_t appartenenti ai rispettivi livelli L_1, L_2, …, L_t che rappresentano la posizione della testina in ogni momento in maniera approssimata, ovvero $H_{j+1} = H_j$U. La struttura procede quindi la seguente proprietà:

(c) se M appartiene alla vicinanza di H_j, allora MU appartiene alla vicinanza di $H_{j+1} (0 \leq j \leq l)$.

Oltre a questo avremo sempre $H_l = M_l(0,0)$, dove l è il nodo al vertice P.

I contatori si occupano dunque di aggiornare dinamicamente questi nodi basandosi sui movimenti delle testine. Il contatore corrispondente ad una testina controlla una sequenza di stadi numerati. I primi due stadi vengono eseguiti all'inizializzazione della simulazione. Quando viene eseguito il t-esimo spostamento sulla testina, si eseguono i rispettivi stadi $2t+1$ e $2t+2$. Più precisamente gli stadi dispari vengono usati per aggiornare H_0 e quelli dispari per i livelli superiori. Gli aggiornamenti sono pianificati seguendo lo stesso principio: se H_0 è aggiornato ogni 2 stadi, H_1 lo è ogni 4, H_2 ogni 8, ecc. . Se definiamo k il livello aggiornato allo stadio t, la programmazione può essere calcolata ricorsivamente in tempo reale nel seguente modo:

$$k(2t-1) = 0, \quad k(2t) = k(t) + 1 \text{ per } t \geq 1.$$

Quindi progettiamo il contatore C in modo tale che, dopo il il completamento dello stadio $2t$, aggiorni il grafico di questa funzione per tutti gli argomenti τ con $1 \leq \tau \leq 2t$. Per fare ciò contiene i nodi $F_1, F_2, \cdots F_{2t}$ e $G_0, G_1, \ldots, G_l$, dove

$$l = \max\{k(\tau) \mid \tau \leq 2t\} = \lfloor \log_2(2t) \rfloor.$$

Le connessioni tra questi nodi sono mostrate in Fig. 2.2 (dove tutti i puntatori senza un ruolo sono stati omessi). In particolare, il puntatore S da F_τ indica $G_{k(t)}$, e il puntatore E da G_k indica la posizione approssimata della testina H_k nel livello k della struttura piramidale. Allo stadio $2t$ si ha accesso al contatore stesso tramite il suo nodo *base* B_r (appartenente alla struttura centrale) attraverso i puntatori N, E, S indirizzati rispettivamente a F_{2t}, F_t e G_0. Quando la testina ν della MdT è attiva il puntatore S dal centro A porta a B_ν, selezionando quindi questo contatore particolare $C = C_\nu$.

2.2.3 Aggiornamento

Descriviamo quindi ora la simulazione della SMM costruita dell'istruzione move δ della MdT, dove $\delta \in \{\text{N,W,S,E}\}$. Assumiamo che tale istruzione corrisponda, lungo tutta la computazione della MdT, al t-esimo movimento della ν-esima testina. Come si può vedere dalla Fig. 2.2 il corretto spostamento di H_0 può essere attuato da una sola istruzione della SMM, `set` SSE `to` SSEδ, purché il puntatore δ sia già installato correttamente. Questo sarà sempre vero grazie alla corretta installazione eseguita al primo passo, ma anche per quelli successivi in quanto, grazie alla regola (c), la nostra simulazione preserverà anche la seguente condizione:

(d) dopo la simulazione di ogni passo, la struttura piramidale contiene le vicinanze di tutti i nodi H_j per tutte le testine.

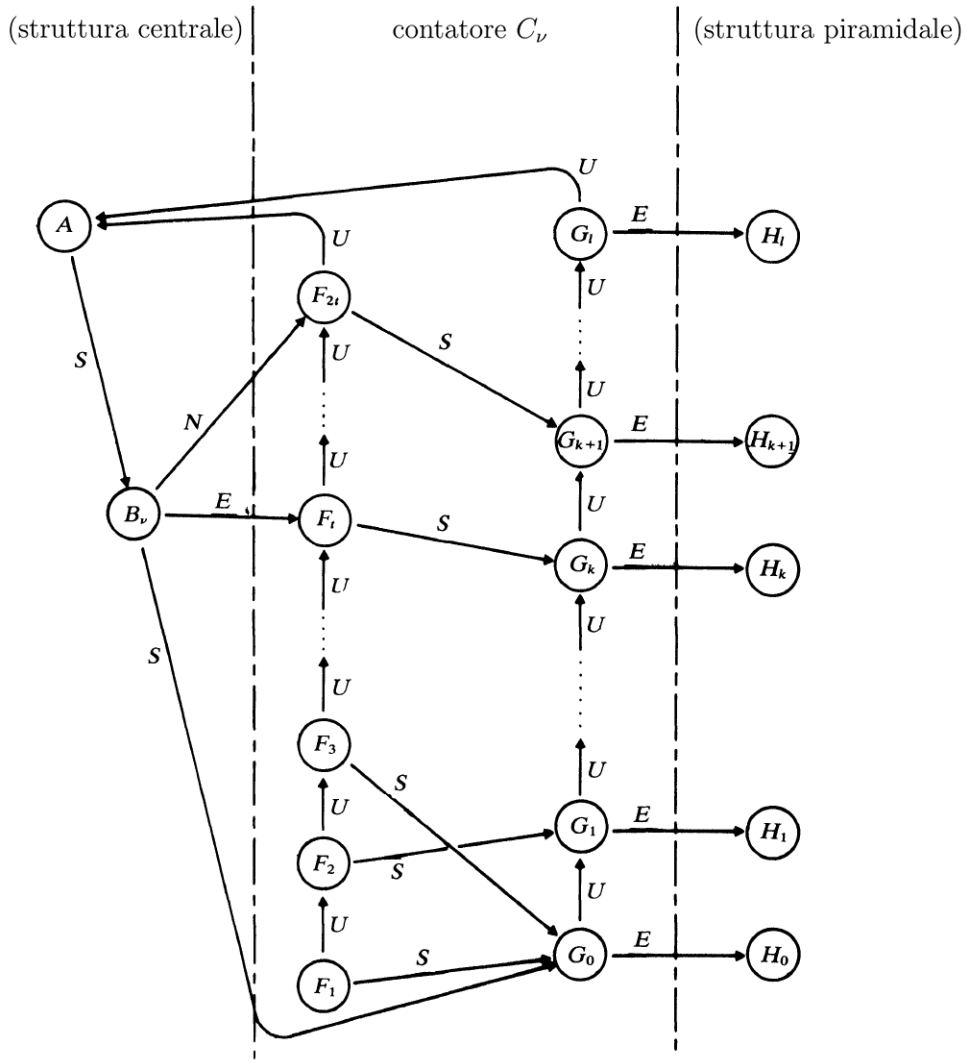


Figura 2.2: Il contatore C_ν dopo il completamento dello stadio $2t$ e le sue connessioni con la struttura centrale e la struttura piramidale.

La simulazione dell'istruzione move consiste quindi negli stadi $2t+1$ e $2t+2$ del contatore C_ν, incluso l'aggiornamento del livello 0 e del livello $k(2t+2) = k(t+1)+1$. In generale l'aggiornamento del livello j consiste in due fasi: la preparazione e adattamento.

La preparazione significa assicurarsi che la vicinanza del nodo H_j appartenga alla struttura piramidale P. Considerata la condizione (a) è facile controllare se manchino delle parti della vicinanza. In questo caso gli ele-

menti mancanti possono essere costruiti in un tempo limitato, mantenendo le condizioni (a) e (b). L'algoritmo corrispondente si implementa nel seguente modo:

- se $j = l$ allora $H_l = M_l(0,0)$ e per (b) esso ha la vicinanza in P, tranne nel caso in cui H_l sia il nodo vertice di P. Questo avviene se l è appena stato creato come nuovo livello, e cioè se i puntatore U da H_l conduce ad A. In questo caso viene creato $M_{l+1}(0,0)$ come nuovo nodo vertice insieme alla vicinanza di H_l;

- se $j < l$ allora (c) ci permette di sapere che tutti i nove nodi sono stati posizionati nella vicinanza di H_{j+1}, che per (d) è definita nella struttura. Da qui è possibile ottenere tutte le informazioni su come riempire i gli elementi mancanti. Se, ad esempio, il puntatore W da H_j non è installato correttamente, allora H_j deve appartenere per forza alla parte *occidentale* della vicinanza di H_jUD (ovvero H_j è uno dei seguenti nodi: H_jUDW, H_jUDWN, H_jUDWS). Ci sono quindi due possibilità: se H_jUWD$\neq A$, allora il puntatore W può essere diretto immediatamente alla sua destinazione H_jUWDES che, grazie a (a), (b), deve essere già presente in P. Altrimenti, per prima cosa devono essere costruiti tutti e nove i nodi alla base della piramide elementare con cima H_jUW, insieme alle rispettive connessioni interne. Da questo esempio si capisce come la preparazione sia fattibile in un numero limitato di passi.

La fase di adattamento del livello j consiste nello spostare H_j correttamente. Per $j = 0$ significa sostituire H_j con $H_j\delta$, per $j > 0$ sostituire H_j con $H_{j-1}U$.

All'interno di ogni stadio l'adattamento precede la preparazione nel modo seguente. Quando il livello j è aggiornato per la prima volta, cosa che avviene nel passo 2^j, allora l è appena stato aumentato e $j = l$. In questo caso l'adattamento significa che viene introdotto $H_j = M_j(0,0)$. Altrimenti la preparazione allo stadio $(2i-1)2^j$ prepara l'adattamento allo stadio $(2i+1)2^j$, quando il livello j verrà aggiornato nuovamente. Nel frattempo verranno eseguiti esattamente due aggiornamenti del livello $j-1$ (agli stadi $(4i \pm 1)2^{j-1}$). Otteniamo quindi per induzione su j che l'adattamento del livello j può spostare H_j di al più un nodo in una direzione data e che anche dopo due di tali spostamenti la condizione (c) rimane comunque soddisfatta.

Mantenendo sempre come riferimento la Fig. 2.2 possiamo ora presentare la sequenza di istruzioni della SMM che simulano move δ. Utilizzeremo una pseudo-istruzione `prepare` H_j per indicare la parte di programma che esegue la preparazione attorno al nodo H_j come descritto sopra.

`set` SEE `to` SSEδ; `prepare` SSE	aggiornamento del livello 0
`new` SNU; `set` SNUS `to` SS	$k(2t+1) = 0$
`new` SNUU	creazione di F_{2t+2}
`set` SE `to` SEU; `set` SN `to` SNUU;	
`if` SESU$= \epsilon$ `then new` SESU	aumenta l
`set` SNS `to` SESU	$k(2t+2) = k(t+1) + 1$
`set` SNSE `to` SESEU	aggiornamento del livello $k(2t+2)$
`prepare` SNSE	

2.2.4 Struttura centrale e simulazione

Per poter illustrare la simulazione delle altre istruzioni interne della MdT dobbiamo prima descrivere la struttura centrale. In aggiunta al centro A e ai nodi di base $B_1, B_2, \ldots, B_r$ per i contatori contiene due altri nodi A_0, A_1 (Fig. 2.3) usati per codificare l'informazione memorizzata: se la cella della memoria con coordinate (x, y) contiene $\alpha \in \{0, 1, b\}$ e se $M_0(x, y)$ esiste già in P, allora il puntatore D da $M_0(x, y)$ è diretto verso A_α nella struttura centrale (nel caso della cella vuota A_b sta per A). Di conseguenza quando nuovi nodi del livello 0 vengono creati nella struttura P essi devono essere diretti verso il centro A in quanto la memoria della MdT è inizialmente vuota per ipotesi.

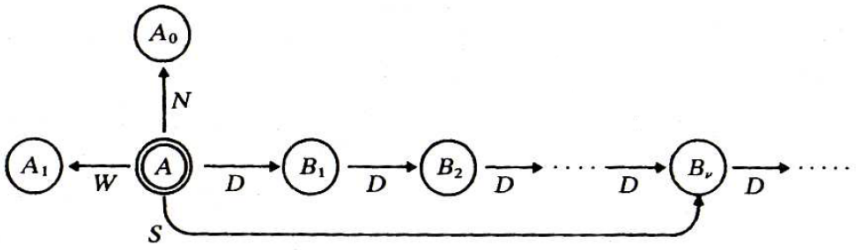


Figura 2.3: La strutttura centrale con centro A, quando la tesina ν è attiva.

Le altre istruzioni vengono quindi simulate nel seguente modo:

`set` S `to` D$^\nu$	head ν
`set` SSED `to` β (con $\beta =$ N,W,ϵ)	write α ($\alpha = 0, 1, b$ rispettivamente)
`if` SSED = N `then goto` λ_0;	
`if` SSED = W `then goto` λ_1	read λ_0, λ_1

L'intero processo è inizializzato creando tutti i nodi della struttura centrale, i nodi F_1, F_2, G_0, G_1 per ciascun contatore, e la struttura piramidale

iniziale consistente della vicinanza di $M_0(0,0), M_1(0,0)$, e il suo nodo in cima $M_2(0,0)$, insieme con tutte le connessioni installate correttamente. Per tutti i contatori i puntatori E da G_0, G_1 sono diretti rispettivamente verso $H_0 = M_0(0,0), H_1(0,0)$.

Tutto questo richiede a sua volta un tempo limitato. Abbiamo quindi costruito una SMM che simula in tempo reale la MdT data.

2.3 Lo spazio nelle SMM

I risultati che lo stesso Schönhage prova riguardo il suo modello di computazione mettono in luce, come abbiamo visto, che la Tesi di Invarianza è soddisfatta nella parte riguardante il tempo, ovvero le SMM sono equivalenti in tempo reale a TM e RAM. Rimane invece da chiarire quale sia il comportamento dello spazio nelle SMM.

Come già accennato, la situazione per lo spazio è più complessa in quanto non è stata scelta una precisa misura per le macchine a puntatori, ma ne sono state definite due, la *massa* e la *capacità*. La massa, ricordiamo brevemente, consiste nel numero n di nodi creati durante la computazione, mentre la capacità consiste nella quantità $n \log n$. Come fa notare van Emde Boas [13], questa ambiguità deriva dal fatto che nella teoria della complessità l'attenzione è solitamente concentrata sul tempo, dando per scontato che la condizione di utilizzare una adeguata quantità di spazio durante la simulazione sia soddisfatta.

Van Emde Boas analizza in modo approfondito, quindi, il rapporto tra SMM e MdT, suggerendo che le SMM siano macchine più potenti nel caso si utilizzi la nozione di massa. Fornendo infatti una propria simulazione della MdT tramite una SMM, egli prova il seguente teorema [13]:

Teorema 2.3.1. *Una computazione di una SMM che richieda una tempo $t(n)$ e una massa $s(n)$, può essere simulata da una MdT in un tempo $O(t(n) \cdot s(n))$ e spazio $O(s(n) \cdot \log s(n))$.*

Il teorema non esclude però che si possa avere un utilizzo dello spazio migliore, ovvero sostituire $O(s(n) \cdot \log s(n))$ con $O(s(n))$, come richiesto dalla Tesi di Invarianza. L'osservazione seguente però dimostra che ciò, in generale, non si può fare:

Proposizione 2.3.2. *Esiste un linguaggio che può essere riconosciuto in tempo reale da una SMM, ma che richiede uno spazio $\Omega(s(n) \cdot \log s(n))$ per essere riconosciuto una MdT, dove $s(n)$ è la massa utilizzata dalla SMM.*

È possibile trovare una dimostrazione in [13].

David R. Lungibuhl e Michael C. Loui [22] rafforzarono il risultato di van Emde Boas mostrando che è possibile simulare una MdT con una SMM in tempo reale utilizzando più efficientemente lo spazio:

Teorema 2.3.3. *Ogni MdT che utilizza uno spazio $s(n)$ può essere simulata in tempo reale da una SMM utilizzando una massa $O(s(n)/\log s(n))$.*

Qual è la ragione di tale asimmetria tra le SMM o, più in generale, tra le macchine a puntatori e gli altri modelli di computazione? Essa sta nella particolare struttura a grafo di tali macchine che rende possibile una rappresentazione dell'informazione più efficiente. Il seguente teorema, la cui dimostrazione si può trovare in [10], mette in luce infatti come il grafo sia uno strumento più potente della normale stringa utilizzata da RAM e MdT.

Teorema 2.3.4. *Per ogni stringa S di lunghezza n possiamo creare una struttura dati su una SMM usando solo $O(n \log n)$ nodi che ci permetta di recuperare o modificare un qualsiasi bit nella stringa S o aggiungerne uno.*

In conclusione, i risultati visti giustificano l'affermazione che le macchine a puntatori non appartengano alla classe di macchine descritta nella Tesi di Invarianza, ma che siano, invece, modelli di computazione più potenti se viene utilizzata la nozione di massa. Allo stesso tempo è immediato notare che invece, utilizzando come misura dello spazio la capacità, che aumenta di un fattore $\log n$ lo spazio utilizzato, le SMM sarebbero equivalenti sia col rispetto del tempo che dello spazio alle MdT e alle RAM. Con questa "correzione" infatti tutti modelli apparterebbero alla stessa classe avente una stessa potenza computazionale, permettendo così di estendere la Tesi di Invarianza a tutti i modelli di computazione atomistici visti.

Secondo van Emde Boas [13], però, il numero di nodi, ovvero la massa, è una misura più naturale dello spazio utilizzato e per questo motivo andrebbe prediletta, riconoscendo un potenziale maggiore alle macchine a puntatori.

Capitolo 3

Le Macchine di Kolmogorov-Uspensky

Oltre alle Macchine di Schönhage, esistono, come già accennato, altri due importanti modelli di macchine a puntatori: le Macchine di Kolmogorov-Uspensky (KUM) ideate da A. N. Kolmogorov e perfezionate in seguito con il suo studente V. A. Uspenski [19, 20] e i Link Automata ideati da D. E. Knuth [18].

La KUM è un modello di computazione di particolare importanza essendo la prima macchina dove la memoria può modificare la propria topologia durante la computazione. Essa consiste infatti, come nelle SMM, in un grafo che viene creato e modificato durante l'esecuzione del programma. Nella macchina di Turing, lo ricordiamo, la memoria è formata invece da uno o più nastri la cui topologia rimane fissa lungo tutta la computazione.

Questa importante novità mette in luce il punto di vista differente che ebbe Kolmogorov nell'ideare il suo modello di computazione. Come fa notare Yuri Gurevich [16], mentre Turing analizzò la computazione di un computer umano, Kolmogorov e Uspensky arrivarono al loro modello di macchina analizzando la computazione dal punto di vista della fisica. Nella KUM infatti ogni informazione è rappresentata in uno spazio fisico ed essa ha un numero limitato di elementi *vicini*.

Vedremo come questo diverso punto di vista arricchisca la nozione di computazione permettendo anche di comprendere aspetti dell'evoluzione di sistemi fisici, chimici e biologici. Viene portato come esempio la Macchina Physarum, implementazione tramite materiale biologico della KUM.

Presentiamo ora la macchina di Kolmogorov-Uspensky, mettendo in evidenza anche la riflessione che portò Kolmogorov a concepire questo nuovo paradigma di computazione.

3.1 Definizione di KUM

Se Schönhage introdusse il suo modello di computazione come strumento per la misura della complessità, l'intento di Kolmogorov era più teorico: fornire una precisa definizione di algoritmo. La sua indagine partì quindi col delineare le proprietà del processo algoritmico per poi fornirne successivamente una descrizione matematica attraverso il modello di computazione.

In seguito quindi le caratteristiche distintive che Kolmogorov individuò negli algoritmi [9]:

- in ogni istante è possibile individuare lo stato S in cui si trova il processo algoritmico;
- il processo algoritmico si divide in passi la cui complessità è limitata in partenza e ogni passo consiste in una trasformazione diretta e immediata dello stato corrente;
- questa trasformazione si applica solo alla parte attiva dello stato e non modifica la restante parte;
- la grandezza della zona attiva è limitata in partenza;
- il processo avanza fin quando il passo successivo è impossibile o un segnale dice che la soluzione è stata raggiunta.

Come scrive Uspensky in [29], per Kolmogorov ogni stato è un insieme di elementi e connessioni, il cui numero totale è finito. Ogni elemento è di una certa tipologia, così come ogni connessione, e per ogni algoritmo il numero totale di tipi di elementi e di connessioni è finito. Come vedremo, nel suo modello di computazione scelse comprensibilmente di rappresentare geometricamente ogni stato del processo attraverso un grafo.

Kolmogorov e Uspensky definirono quindi un modello di computazione matematico che tenesse conto di queste caratteristiche, la KUM. Forniamo la descrizione di KUM così come è stata data da Y. Gurevich in [15]. Essa si discosta parzialmente dalla descrizione originaria in quanto utilizza come punto di partenza le SMM di Schönhage. Sarà più facile, in questo modo, mettere a confronto i due modelli di macchine a puntatori.

Una KUM è formata, perciò, da un grafo diretto e simmetrico. Tale grafo è descrivibile tramite una Δ-struttura analoga a quella delle SMM, in cui ogni arco (o puntatore) è contrassegnato con una etichetta α appartenente all'alfabeto finito Δ. Essendo simmetrico, però, l'esistenza di un arco dal nodo u al nodo v contrassegnato con α, implica l'esistenza di un altro arco da v a u con la stessa etichetta. È importante notare che questo comporta una

significativa differenza con le SMM: nelle KUM il numero di archi entranti in un nodo è limitato. Grazie alla simmetria del grafo, infatti, il numero di archi entranti è lo stesso di quelli uscenti, il quale è limitato in quanto l'alfabeto Δ è finito. Nelle SMM, inoltre, la Δ-struttura è necessariamente completa (da ogni nodo partono esattamente un contatore per ogni etichetta in Δ) mentre questa restrizione non è presente nelle KUM.

Ancora, Kolmogorov utilizza la nozione di nodo *attivo* e di zona *attiva* che consiste nella zona raggiungibile con al più un numero k di nodi prefissato. La zona attiva è la parte di Δ-struttura che può essere modificata durante un particolare passo della computazione. Essa viene modificata, come nelle SMM, da un programma di controllo. Il programma consiste in una sequenza di istruzioni che possono eseguire le seguenti azioni:

- aggiungere un nuovo nodo e una coppia di archi con la stessa etichetta tra il nuovo nodo e il nodo centrale;
- rimuovere un nodo e gli archi entranti ed uscenti da esso;
- aggiungere una coppia di archi tra due archi esistenti;
- arrestare la computazione (halt).

La nozione di zona attiva, non presente nelle SMM, è stata introdotta probabilmente con l'intenzione di restituire un modello di computazione più realistico. È ragionevole infatti pensare che la quantità di informazione disponibile e manipolabile in un determinato momento della computazione sia limitata a priori. Non vi è alcuna istruzione che faccia riferimento ad un input o all'output. Il motivo è che l'input consiste nella Δ-struttura iniziale che quindi, a differenza delle SMM, è ammessa non essere vuota. L'output, a sua volta, consiste nella configurazione finale del grafo al momento dell'esecuzione dell'istruzione d'arresto.

Dal punto di vista della complessità computazionale, le KUM, in quanto macchine a puntatori, godono delle stesse proprietà che abbiamo visto nelle SMM. Risultano quindi essere macchine più potenti delle MdT, grazie alla struttura di memoria a grafo.

La relazione tra KUM e SMM in termini di potenza computazionale è invece più complessa. Dalla descrizione fatta delle KUM è semplice dimostrare che $KUM \xrightarrow{r}$ SMM, ovvero che ogni KUM può essere simulata in tempo reale da una apposita SMM. Il problema inverso, se $SMM \xrightarrow{r}$ KUM, rimane invece una questione aperta. Essa fu posta per la prima volta da Schönhage in [26]. Come afferma Cloteaux [10], sono stati pubblicati risultati parziali che sembrerebbero suggerire che le SMM non siano simulabili in tempo reale dalle KUM. La questione rimane però tuttora aperta.

Nella prossima sezione procediamo col descrivere un possibile implementazione della macchina di Kolmogorov-Uspensky, contestualizzando il significato di tale esperimento.

3.2 La Macchina Pysharum, implementazione della KUM

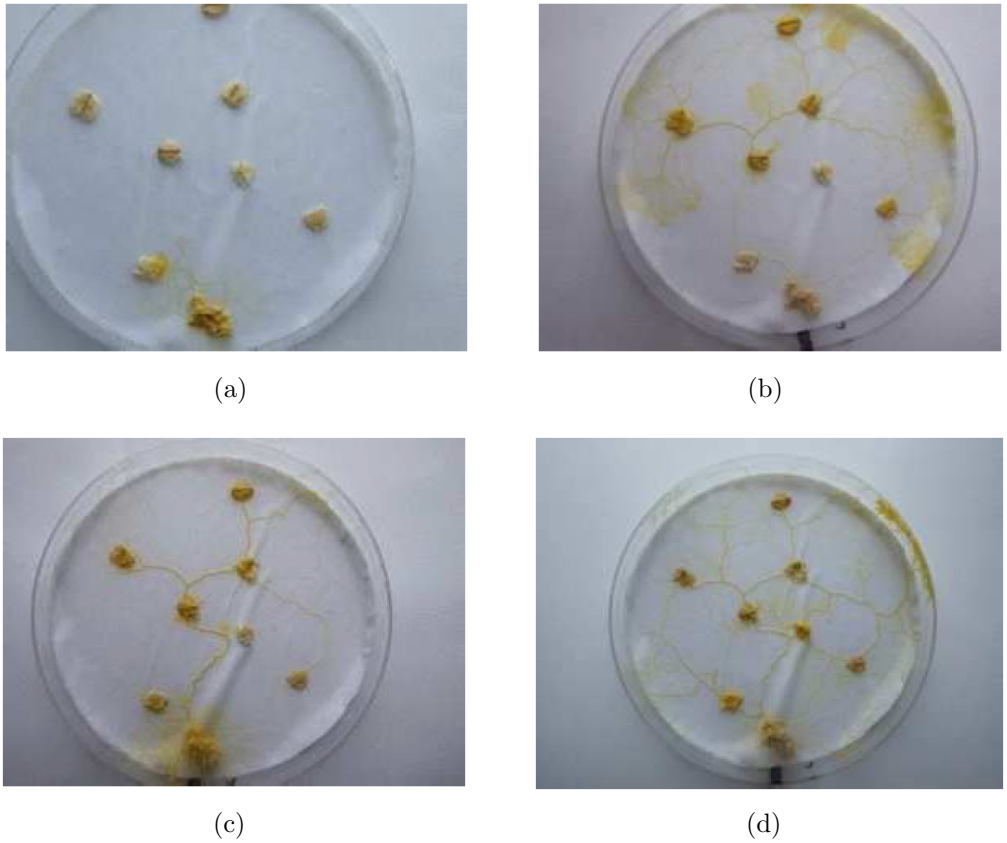
La computazione non convenzionale (da *unconventional computing*) è un campo interdisciplinare della scienza dove informatici, fisici e matematici applicano i principi di computazione nei sistemi naturali per progettare architetture e computer innovativi.

Descriviamo una di queste macchine, la Macchina Physarum (*Physarum Machine*), per mostrare come, grazie alla sua topologia dinamica, la macchina di Kolmogorov-Uspensky non solo si presti ad essere un valido strumento per l'interpretazione di sistemi biologici, ma possa essere anche implementata da essi.

Il *Physarum polycephalum* consiste in un una muffa composta da una singola cellula di grandi dimensioni, visibile a occhio nudo, e da diversi nuclei collegati da vene. Una volta posizionato in un ambiente adeguato, esso si propaga alla ricerca di nutrimento creando una fitta rete di vene e nuclei. È stato osservato che tale muffa possiede una capacità computazionale che utilizza per ottimizzare la ricerca delle sostanze nutritive ed il trasporto di esse tra i nuclei. Questa capacità gli permette, ad esempio, di risolvere anche articolati labirinti per arrivare alle fonti di nutrimento [23].

È possibile sfruttarne il comportamento per trovare soluzioni a classici problemi riguardanti i grafi, come il calcolo del cammino minimo, tassellazioni del piano e la creazione di grafici di prossimità [2]. L'insieme di nodi e vene, infatti, può essere facilmente interpretato come un grafo planare non diretto. In questo senso la macchina astratta di Kolmogorov può essere un utile strumento per rappresentare il modello di computazione del Physarum. Pur avendo la KUM un grafo diretto come memoria, essendo esso simmetrico è facilmente rappresentabile da un grafo non diretto: è sufficiente sostituire ogni coppia di puntatori con un solo arco senza direzione.

Andrew Adamatzky nella sua ricerca sulla computazione non convenzionale va però oltre proponendo una implementazione della KUM stessa tramite la muffa Physarum. Egli crea, sfruttando le abilità computazionali della muffa, una vera e propria macchina che chiama Macchina Physarum (MP) [1]. Adamtzky descrive la MP come "un *computer device* biologico e amorfo, che incorpora il *Physarum polycephalum*, programmabile attraverso la configura-



(a) (b)

(c) (d)

Figura 3.1: Computazione della Macchina Physarum. Le foto sono state scattate in un arco di tempo di 10 ore.

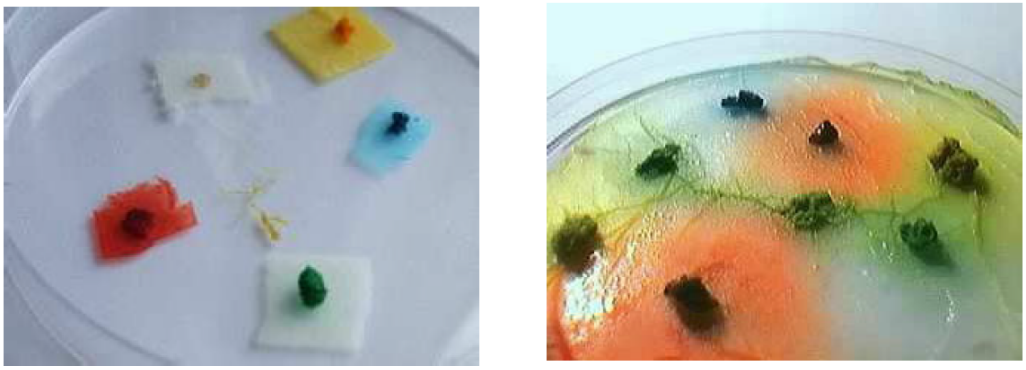
zione di gradienti di sostanze attrattive e repellenti" [14]. Procediamo ora schematicamente alla descrizione della MP mettendo in evidenza per ciascun elemento della macchina astratta il corrispettivo biologico. Per rendere la presentazione più comprensibile tralasciamo, ove possibile, i termini specifici e i dettagli più tecnici dell'esperimento.

Nodi. La MP ha due tipi di nodi: i nodi stazionari che consistono nelle fonti di nutrimento e i nodi dinamici, dove due o più vene hanno origine. All'inizio della computazione i nodi stazionari sono distribuiti nello spazio computazionale e il Physarum è posizionato a sua volta in punto dello spazio. Dalle condizioni iniziali la muffa si svilupperà alla ricerca di cibo occupando i nodi stazionari.

Archi. Gli archi della MP consistono nelle vene che connettono tra loro i nodi, sia stazionari che dinamici. Essi formano quindi un grafo non diretto, equivalente, però, al grafo diretto e simmetrico delle KUM.

Input, output e arresto. Il programma e l'input sono rappresentati dalla configurazione spaziale dei nodi stazionari e dalla posizione iniziale del Physarum. La memoria durante il processo algoritmico è rappresentata dalla configurazione del Physarum nello spazio. All'inizio essa consiste in un unico punto isolato. Esattamente come nella KUM, il risultato della computazione consiste nella configurazione finale dei nodi e degli archi. Il Physarum però procede nella computazione anche una volta che la soluzione è stata raggiunta e si ferma solamente all'esaurimento delle sostanze nutritive. Si considera quindi che l'arresto della MP avvenga una volta che tutti i nodi stazionari siano stati utilizzati.

Zona attiva. All'inizio della computazione esiste una sola zona attiva, dove il Physarum è stato posizionato. La muffa inizierà poi ad espandersi attraverso le vene. Esse vengono generate tramite un flusso periodico di sostanze chimiche, simile a delle pulsazioni, dalla parte centrale della muffa verso la periferia, fino ad arrivare alle estremità delle vene. Si potrebbe immaginare quindi che la zona attiva iniziale continui ad espandersi, violando il principio di limitatezza a priori della KUM. Questo però non accade. Quando infatti il Physarum viene posizionato in un ambiente scarso di fonti nutritive, la diffusione si concentra in poche direzioni specifiche. La ricerca di nutrimento, anche quando la muffa possiede un reticolo esteso, viene infatti localizzata in un numero limitato di zone. In ogni zona, poi, è localizzabile un nodo centrale da cui parte il flusso generante le vene. Le consideriamo essere quindi zone *attive* (e i rispettivi nodi centrali, nodi attivi) in quanto sono le uniche parti dell'organismo che, in quel determinato momento della computazione, modificano la propria configurazione.



(a) (b)

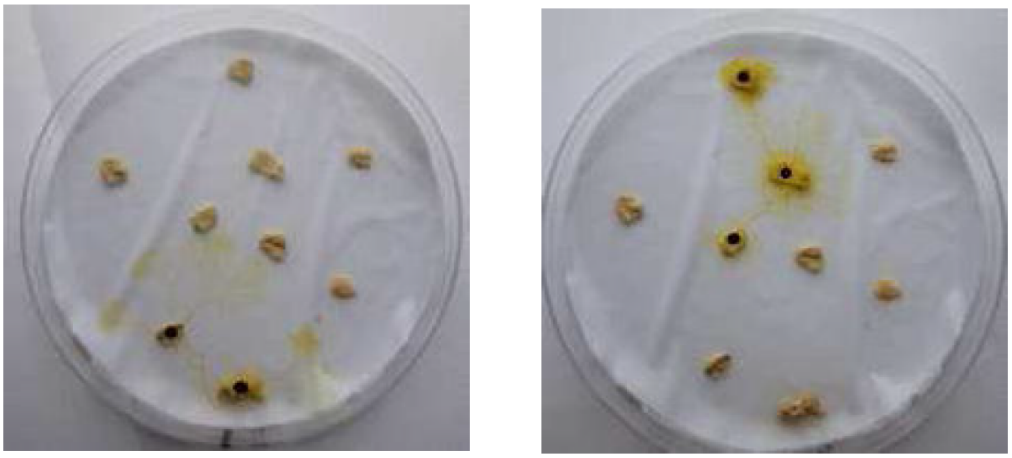
Figura 3.2: Etichettamento degli archi tramite colorazione dei nodi stazionari. In (a) la configurazione iniziale della MP.

Limitatezza delle connessioni. A differenza delle SMM, come abbiamo visto, le KUM hanno un numero limitato di archi entranti e uscenti da ogni nodo. È stato osservato in laboratorio che solitamente il numero di archi in un nodo creato dal Physarum non è superiore a 3.

Etichettamento degli archi. Non c'è una diretta implementazione di questa funzione nella MP. Per i nodi stazionari essa può essere implementata o colorando i nodi o modificando la composizione chimica dei nutrimenti. Il Physarum non tratta le colorazioni allo stesso modo ma si ramifica assegnando ai colori priorità diverse Questo comportamento quindi può essere sfruttato per programmare la MdP.

Istruzioni di base. Un possibile insieme di istruzioni della MP potrebbe essere il seguente: `input`, `output`, `go halt` le istruzioni comuni, `new`, `set`, `if` quelle interne. Assumiamo che l'istruzione `input` sia eseguita attraverso la distribuzione di risorse nutritive, e che `output` sia registrato visivamente. L'istruzione `halt` è già stata discussa precedentemente. L'istruzione `set` reindirizza gli archi e può essere implementata posizionando sostanze nutritive fresche nello spazio. Analizziamo ora l'implementazione delle istruzioni principali nella MP:

– AGGIUNGI UN NODO. Per aggiungere un nodo stazionario b alla vicinanza di a, il Physarum si deve propagare da a a b attraverso una vena,



(a) (b)

Figura 3.3: Implementazione dell'istruzione RIMUOVI NODO della MP. All'inizio il Physarum occupa due nodi stazionari a sud della piastra. Successivamente, rimuove questi nodi dalla sua memoria abbandonandoli e occupando tre nodi stazionari a nord della piastra (i nodi inclusi nella memoria sono marcati con punti neri).

che rappresenta l'arco *ab*. Per creare un nodo dinamico una o due vene devono convergere in un punto che rappresenterà il nuovo nodo.

- RIMUOVI NODO. Per rimuovere un nodo stazionario il Physarum abbandona la fonte di nutrimento corrispondente. Elimina i nodi dinamici eliminando le vene a cui sono connessi.

- AGGIUNGI ARCO. Per aggiungere un arco in una zona attiva, il nodo attivo corrispondente genera un flusso di materiale biologico verso la zona periferica che genererà una nuova vena.

- RIMUOVI ARCO. Quando una vena viene eliminata, ad esempio per esaurimento della fonte di nutrimento, l'arco rappresentato dalla vena viene eliminato.

La MP non consiste però in un generico computer. Essa deve essere considerata infatti come un esperimento il cui scopo è quello di approfondire il comportamento di questo organismo. Si prevede che il Physarum grazie alle sue caratteristiche e alla sua capacità di sopravvivere in ambienti non laboratoriali, possa avere un ruolo importante nell'implementazione di futuri computer chimici, molecolari e biologici [5]. È possibile trovare lo stato dell'arte della ricerca sulla Macchina Physarum in [3].

Bibliografia

[1] Andrew Adamatzky: Physarum Machine: Implementation of a Kolmogorov-Uspensky Machine on a Biological substrate. Parallel Processing Letters 17(4): 455-467 (2007)

[2] Andrew Adamatzky: Developing Proximity Graphs by Physarum polycephalum: Does the Plasmodium Follow the Toussaint Hierarchy? Parallel Processing Letters 19(1): 105-127, (2009)

[3] Andrew Adamtzky: Advances in Physarum Machine. Springer (2016)

[4] Andrew Adamatzky, Larry Bul, Benjamin De Lacy Costello, Susan Stepney, Christof Teuscher: Unconventional Computing 2007. Luniver Press (2007)

[5] Andrew Adamatzky, Victor Erokhin, Martin Grube, Theresa Schubert, Andrew Schumann: Physarum Chip Project: Growing Computers From Slime Mould. IJUC 8(4): 319-323, (2012)

[6] Amir M. Ben-Amram: What is a "pointer machine"? SIGACT News 26(2): 88-95 (1995)

[7] Amir M. Ben-Amram, Pointer Machines and Pointer Algorithms: an Annotated Bibliography. (1998).

[8] Allan Borodin, Michael J. Fischer, David G. Kirkpatrick, Nancy A. Lynch, Martin Tompa: A Time-Space Tradeoff for Sorting on Non-Oblivious Machines. J. Comput. Syst. Sci. 22(3): 351-364 (1981).

[9] Andreas Blass, Yuri Gurevich: Algorithms: A Quest for Absolute Definitions. Bulletin of the EATCS 81: 195-225 (2003)

[10] Brian D. Cloteaux, An Investigation of Pointer Machine. New Mexico State University Las Cruces, NM, USA (2007)

[11] Stephen A. Cook: An Overview of Computational Complexity. Commun. ACM 26(6): 400-408 (1983)

[12] Stephen A. Cook, Robert A. Reckhow: Time Bounded Random Access Machines. J. Comput. Syst. Sci. 7(4): 354-375 (1973)

[13] Peter van Emde Boas: Space Measures for Storage Modification Machines. Inf. Process. Lett. 30(2): 103-110 (1989)

[14] Jeremy Garwood , A conversation with Andrew Adamatzky. Lab Times, 1-2013: 24-27 (2013)

[15] Yuri Gurevich: On Kolmogorov Machines and Related Issues, the column on Logic in Computer Science, Bulletin of European Association for Theoretical Computer Science: 35, 71-82 (1988)

[16] Yuri Gurevich: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Log. 1(1): 77-111 (2000)

[17] Joseph Y. Halpern, Michael C. Loui, Albert R. Meyer, Daniel Weise: On Time versus Space III. Mathematical Systems Theory 19(1): 13-28 (1986)

[18] Donald E. Knuth: The Art of Computer Programming. Addison-Wesley, Reading, MA: vol. 1 (1968)

[19] Andrei N. Kolmogorov: On the notion of algorithm. Uspekhi Matematicheskikh Nauk: 8:4(56), 175-176 (1953)

[20] Andrei N. Kolmogovov, Vladimir A. Uspensky: On the definition of an algorithm. Uspekhi Mat. Nauk: 13:4(82), 3–28 (1958)

[21] David R. Lungibuhl: Computational Complexity of Random Access Models. Univeristy of Illinois at Urbana-Champaign (1990)

[22] David R. Luginbuhl, Michael C. Loui: Hierarchies and Space Measures for Pointer Machines. Inf. Comput. 104(2): 253-270 (1993)

[23] Toshiyuki Nakagaki: Smart behaviour of true slim mold in a labyrinth. Research in Microbiology 152(9): 767-70 (2001)

[24] Arnold Schönhage, Universelle Turing Speicherung, Automatentheorie und Formale Sprachen, Dörr, Hotz, eds. Bibliogr. Institut, Mannheim: 69-383 (1970)

[25] Arnlod Schönhage Real-time simulation of multidimensional Turing machines by storage modification machine. Technical Memorandum 37, M.I.T. Project MAC, Cambridge, MA (1973)

[26] Arnold Schönhage: Storage Modification Machines. SIAM J. Comput. 9(3): 490-508 (1980)

[27] Cees F. Slot, Peter van Emde Boas: The Problem of Space Invariance for Sequential Machines. Inf. Comput. 77(2): 93-122 (1988)

[28] Alan M. Turing: On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society Volume s2-42, Issue 1: 230-265 (1937)

[29] Vladimir A. Uspensky: Kolmogorov and Mathematical Logic. J. Symb. Log. 57(2): 385-412 (1992)