Università degli Studi di Salerno
Dipartimento di Informatica

Tesi di Laurea di I livello in
Informatica

Analisi e implementazione in Java di un algoritmo di pattern matching basato su automi

Relatore
Prof. Rosalba Zizza

Candidato
Nunzia Esposito
Matricola
0512102328

Anno Accademico 2016-2017

A mio cugino Gennaro,
con te è andata via una parte di me,
ma i ricordi di un'infanzia insieme restano.
Grazie per quei momenti.
Mi manchi.

Indice

Introduzione

Gli algoritmi di pattern matching sono usati nelle implementazioni di software pratici presenti nella maggior parte dei sistemi operativi. Inoltre, sono alla base di tanti strumenti di ricerca di editor di testo.
Questo è particolarmente evidente in letteratura o linguistica dove i dati sono composti da vaste raccolte (corpus-a) e dizionari. Ciò si applica anche all'informatica dove un'ampia quantità di dati viene archiviata in file sequenziali. Naturale e recente campo di applicazione è la bioinformatica, in cui le sequenze molecolari possono spesso essere approssimate in catene di nucleotidi o amminoacidi. In problemi reali, il testo ha quindi dimensioni notevoli. Questa è la ragione per cui gli algoritmi dovrebbero essere efficienti anche se la velocità e capacità di archiviazione dei computer incrementa regolarmente.

Il matching di stringhe consiste nel trovare una, o più in generale, tutte le occorrenze di una stringa (più comunemente chiamata *pattern*) in un testo. Gli algoritmi analizzati da [1, 4] danno in output tutte le occorrenze del pattern nel testo. Il pattern è denotato con $x = x[0, \ldots, m-1]$, la sua lunghezza è uguale a m. Il testo è denotato con $y = y[0, ..., n-1]$ e la sua lunghezza è pari a n. Entrambe le stringhe sono costruite su un set di caratteri finito chiamato $alfabeto$ e denotato con Σ di taglia σ.
Le applicazioni hanno bisogno di due tipi di soluzioni a seconda di quale stringa, il pattern o il testo, è fornita per prima in input. Algoritmi basati sull'uso di automi o proprietà combinatorie di stringhe sono comunemente implementati per pre-processare il pattern e risolvere il primo tipo di problema. La nozione di *indici* realizzata dagli alberi o automi è usata nel secondo tipo di soluzioni. In [4] vengono trattati solo algoritmi del primo tipo e il presente lavoro di tesi analizza proprio algoritmi che lavorano su strutture dati dette *indici*, le quali rappresentano tutti i fattori di un dato testo. In [1, 2] viene mostrata la costruzione di tali strutture, quali: *suffix trie, suffix automaton, grafi diretti aciclici della parola (DAWG).*

Gli algoritmi di matching di stringhe presentati in [1, 4] lavorano in questo modo. Analizzano il testo con l'aiuto di una finestra la cui taglia è generalmente pari a m. Per prima cosa allineano l'estremità sinistra della finestra e del testo, poi confrontano i caratteri della finestra (che contiene i caratteri del pattern) con quelli del testo - quest'attività specifica è chiamata *attempt* (ten-

tativo) - e dopo un intero abbinamento del pattern o dopo un' incompatibilità essi spostano la finestra a destra. Gli algoritmi ripetono la stessa procedura di nuovo finché l'estremità destra della finestra va oltre l'estremità destra del testo. Questo metodo è solitamente chiamato *meccanismo della finestra scorrevole*. Associamo ogni tentativo con la posizione j nel testo quando la finestra è posizionata su $y[j, \ldots, j + m - 1]$. L'algoritmo di $Forza\ Bruta$ localizza tutte le occorrenze di x in y in tempo $O(mn)$. Mostreremo, qui di seguito, che i molti miglioramenti apportati al metodo di Forza Bruta possono essere classificati a seconda dell'ordine con cui sono compiuti i confronti tra i caratteri del pattern e quelli del testo durante ogni prova.
Il modo più naturale di rappresentare i confronti è da sinistra a destra, seguendo la direzione di lettura occidentale; compiere i confronti da destra a sinistra generalmente porta ad algoritmi più efficienti nella pratica, invece i migliori limiti teorici sono raggiunti quando i confronti sono fatti in un ordine specifico. Infine esistono alcuni algoritmi per cui l'ordine nel quale i confronti sono fatti è irrilevante (come ad esempio per l'algoritmo di Forza Bruta) per trasformare una stringa in un'altra, come specificheremo meglio dopo.

Come abbiamo già accennato, gli algoritmi di testo sono applicati anche alla biologia, nel campo della genetica. Le molecole degli acidi nucleici portano con sè una grande quantità di informazioni sui fondamentali fattori che determinano la vita, in particolare delle cellule riproduttive. Ci sono due tipi di acidi nucleici conosciuti come acido desossiribonucleico (DNA) e ribonucleico (RNA). Il DNA è una molecola a doppia elica; il suo scheletro è una sequenza di quattro lettere di nucleotidi: adenina, guanina, citosina e timina (rispettivamente A, G, C, T). L'RNA invece presenta un solo filamento composto da ribonucleotidi: A, G, C e uracile (U). I processi di *trascrizione* e *traslazione* portano alla produzione delle proteine, che hanno la struttura di una stringa composta da 20 amminoacidi. Tutte queste molecole possono essere viste come testi.
La scoperta di tecniche potenti di sequenziamento, quindici anni fa, ha portato ad un rapido accumulo di sequenze di dati (più di 10 milioni di sequenze di nucleotidi). Molti algoritmi sono impiegati per la loro raccolta e conseguente analisi.
Le sequenze vengono collezionate attraverso gel audioradiografici. La trascrizione automatica di questi gel in sequenze è un tipico ***problema del pattern matching bidimensionale***.
Una volta che si è ottenuta una nuova sequenza di dati, la prima domanda importante che ci si pone è se essa assomigli a una qualsiasi altra sequenza già memorizzata nei databank. Prima di inserire una sequenza si controlla che essa non sia già presente nei database. Il confronto di diverse sequenze è realizzato scrivendo una sull'altra. Il risultato è conosciuto come l'*allineamento dell'insieme delle sequenze*. Questo è una dimostrazione comune nella biologia molecolare che può dare un'idea sull'evoluzione delle sequenze attraverso mutazioni, inserimenti e cancellazioni di nucleotidi. L'allineamento di due sequenze è anche conosciuto come ***problema della distanza di Levenshtein***, ossia calcolare il numero minimo di operazioni di modifica fatte per trasformare una stringa in

un'altra.

Gli allineamenti sono usati per confrontare stringhe. Sono ampiamente usati nel calcolo della biologia molecolare. Costituiscono un mezzo per visualizzare somiglianze tra stringhe. Sono basati sulle nozioni di distanza e similarità.
Tre differenti tipi di allineamento di due stringhe x e y sono considerate: *allineamento globale* (che considera le intere stringhe x e y), *allineamento locale* (che abilita il ritrovamento del segmento di x che è più vicino ad un segmento di y) e la più lunga sottosequenza comune di x e y.

Ulteriori domande sulle sequenze sono legate, invece, alla loro analisi. Lo scopo è quello di scoprire le funzioni di tutte le parti della sequenza. Ad esempio, le sequenze di DNA contengono importanti regioni (sequenze di codifica) per la produzine di proteine. Ma non è stata trovata nessuna risposta adatta per trovare tutte le sequenze di codifica di una sequenza di DNA.

Il problema del matching di stringhe ha molte varianti, che riguardano problemi legati alla struttura dei segmenti di testo, compressione di dati, problemi di string matching approssimato, estensioni di immagini a due dimensioni e allineamento di stringhe. Qualche volta desideriamo trovare una stringa, ma non la ricordiamo completamente. La ricerca deve essere fatta con un pattern non interamente specificato: questa è un'istanza del ***pattern matching approssimato***.
Lo string matching approssimato consiste nel trovare un'occorrenza approssimata di un pattern in un testo, ovvero, una posizione i tale che $dist(pat, text[i+1 \ldots i+m])$ è minima, dove $dist$ è una funzione che definisce una distanza tra due parole. Un'altra istanza del problema è trovare tutte le posizioni i tali che $dist(pat, text[i+1 \ldots i+m])$ è minore di un intero k dato.
Ci sono due distanze standard: quella di $Hamming$ (numero di posizioni in cui due stringhe differiscono) e la distanza di *Levenshtein* (numero minimo di operazioni di modifica necessarie per trasformare una stringa in un'altra, dove le operazioni di modifica sono: inserimento di un carattere, cancellazione e cambiamento di esso).
Con la distanza di Hamming, il problema è anche conosciuto come matching di stringhe approssimato con k mismatch. Con quella di Levenshtein (o *distanza di modifica*), il problema è conosciuto come matching di stringhe approssimato con k differenze.
Il problema della distanza di modifica è strettamente legato a quello dello string matching approssimato e simile alla nozione di allineamento di sequenze di DNA nella Biologia Molecolare.

Uno scanner di immagini è un tipo di fotocopiatrice. Viene usato per dare una versione generalizzata di un'immagine. Quando l'immagine è una pagina di testo, l'output naturale dello scanner deve essere una forma digitalizzata che può essere utilizzata da un editor di testo. L'elaborazione delle immagini è collegato al problema del ***pattern matching bidimensionale***.

Gli array possono essere visti come rappresentazioni di una mappa di bit delle immagini, dove ogni cella dei vettori contiene il nome in codice di un pixel.
Per questi tipi di problemi, il pattern e il testo sono array bidimensionali (o matrici) i cui elementi sono simboli. Dobbiamo capire se il pattern occorre nel testo come un sottoarray, o trovare tutte le posizioni di esso nel testo. Quindi il problema è quello di localizzare tutte le occorrenze di un pattern bidimensionale $x = x[0 \dots m_1 - 1, 0 \dots m_2 - 1]$ di taglia $m_1 \times m_2$ al'interno di un testo a due dimensioni $y = y[0 \dots n_1 - 1, 0 \dots n_2 - 1]$ di dimensioni $n_1 \times n_2$.

Il seguente lavoro di tesi ha riguardato l'analisi e l'implementazione di un algoritmo di pattern matching esatto basato su un automa, il $DAWG$. Il documento è strutturato in questo modo.
Nel Capitolo 1 vengono dati gli strumenti teorici necessari per comprendere gli argomenti trattati, si spiegano alcuni concetti legati ai linguaggi formali [2].
Nel Capitolo 2 sono mostrati alcuni esempi di algoritmi per il pattern matching [1, 4], partendo dal più naive (Forza Bruta) fino a mostrare algoritmi più complessi basati su strutture ad indici come suffix tree e DAWG.
Nel Capitolo 3 viene illustrato in dettaglio l'algoritmo sul quale si basa il presente lavoro, non limitandosi solo a spiegare gli algoritmi, ma mostrando la costruzione di un suffix automaton e l'esecuzione dell'algoritmo impiegando tale struttura.
Nel Capitolo 4 ci concentriamo sull'implementazione dell'algoritmo, mettendo in risalto le aggiunte fatte ad esso.
Infine nel Capitolo 5 vengono mostrate alcune classi di test, fornendo l'output dell'esecuzione dell'algoritmo su tali esempi.

Capitolo 1

Notazioni e definizioni formali

Questo capitolo fornisce i fondamenti teorici necessari per la comprensione del presente lavoro di tesi, in particolare concetti e notazioni usate per lavorare sulle stringhe, linguaggi e automi. Il resto del capitolo è rivolto all'introduzione di strutture dati scelte per implementare automi e per la progettazione di tecniche di pattern matching [1, 2].

1.1 Linguaggi e automi

Sia Σ un *alfabeto* finito di input. Gli elementi di Σ sono chiamati lettere, caratteri o simboli. Esempi tipici di alfabeti sono: l'insieme di tutte le lettere alfabetiche o l'insieme delle cifre binarie.

Una ***stringa*** su un alfabeto Σ è una sequenza finita di elementi di Σ. La sequenza di zero lettere è chiamata ***stringa vuota*** ed è denotata con ϵ. Per semplicità, delimitatori e separatori usualmente usati nella notazione di sequenze sono rimossi e la stringa viene scritta come la semplice giustapposizione delle lettere che la compongono. Ad esempio ϵ, a, b e $baba$ sono stringhe su un qualsiasi alfabeto che contiene le due lettere a e b. L'insieme di tutte le stringhe sull'alfabeto Σ è denotato con Σ^* e l'insieme di tutte le stringhe sull'alfabeto Σ eccetto la stringa ϵ è denotato con Σ^+.
La ***lunghezza*** di una stringa x è definita come la lunghezza della sequenza associata con la stringa x, cioè il numero dei simboli (con ripetizione) che la compongono, ed è denotata con $|x|$. Indichiamo con $x[i]$, per $i = 0, 1, ..., |x| - 1$, la lettera all'indice i di x con la convenzione che gli indici iniziano con 0.

Una stringa x è un ***fattore*** di una stringa y se esistono due stringhe u e v tali che $y = uxv$. Quando $u = \epsilon$, x è un ***prefisso*** di y; quando $v = \epsilon$, x è un ***suffisso*** di y. La stringa x è una ***sottosequenza*** di y se esistono $|x|+1$ stringhe

$w_0, w_1, ..., w_{|x|}$ tali che $y = w_0x[0]w_1x[1]...x[|x|-1]w_{|x|}$; in modo meno formale, x è una stringa ottenuta da y eliminando $|y| - |x|$ lettere.
Sia x una stringa non vuota e y una stringa. Diciamo che c'è un'***occorrenza*** di x in y, o semplicemente che x ***occorre in*** y, quando x è un fattore di y.

Un automa finito deterministico (DFA) $\mathscr{A}$ è una quadrupla (Q, q_0, T, F) dove:

- Q è un insieme finito di stati;
- q_0, appartenente a Q, è lo stato iniziale;
- T, sottoinsieme di Q, è l'insieme degli stati finali;
- F $\subseteq$ Q$\times\Sigma\times$Q è l'insieme delle transizioni.

Diciamo di un arco (p, a, q) che lascia lo stato p ed entra nello stato q che: p è la ***sorgente*** dell'arco, la lettera a è la sua ***etichetta***, lo stato q il suo ***target***. Un ***cammino*** di lunghezza n nell'automa $\mathscr{A}$ =(Q, q_0, T, F) è una sequenza di n archi consecutivi

$$\left\langle (p_0, a_0, p'_0), (p_1, a_1, p'_1), ..., (p_{n-1}, a_{n-1}, p'_{n-1}) \right\rangle$$

che soddisfa

$$p'_k = p_{k+1}$$

per k = 0,...,n-2. L'etichetta del cammino è la stringa $a_0, a_1, ..., a_{n-1}$, la sua ***origine*** è lo stato p_0, la sua ***fine*** lo stato p'_{n-1}.
Una stringa è ***riconosciuta*** o ***accettata*** dall'automa se è l'etichetta di un cammino di successo, cioè tale che $p'_{n-1} \in T$.

In altre parole, il linguaggio $L(\mathscr{A})$ definito da $\mathscr{A}$ è il seguente insieme $L(\mathscr{A}) = \{w \in \Sigma^* : \exists\ q_0, \ldots, q_n,\ n{=}|w|,\ q_n \in T\ t.c.\ \forall\ 0 \leq i < n,\ (q_i, w[i], q_{i+1}) \in F\}$. Informalmente, il linguaggio definito o accettato da $\mathscr{A}$ è l'insieme delle parole le cui lettere sono, sequenzialmente, etichette di archi consecutivi a partire da quello iniziale fino ad uno stato finale.

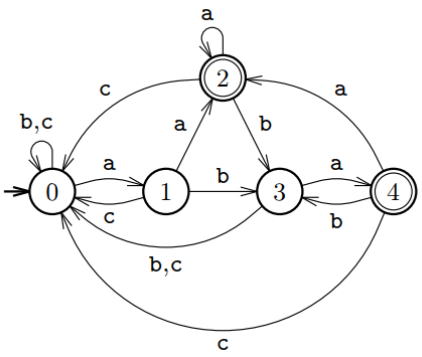
Un automa $\mathscr{A}$ è ***deterministico*** se per ogni coppia $(p, a) \in Q \times \Sigma$ esiste al più uno stato $q \in Q$ tale che $(p, a, q) \in F$. In questo caso, è naturale considerare la ***funzione di transizione***

$$\delta : Q \times \Sigma \to Q$$

dell'automa definito per ogni arco appartenente a F da

$$\delta(p, a) = q.$$

La funzione δ è facilmente estesa alle stringhe. La sua estensione da considerare è $\bar{\delta} : Q \times \Sigma^* \to Q$ ricorsivamente definita da $\bar{\delta}(p, \epsilon) = p$ e $\bar{\delta}(p, wa) = \delta(\bar{\delta}(p, w), a)$ per $p \in Q$, $w \in \Sigma^*$ e $a \in \Sigma$. La stringa w è riconosciuta dall'automa $\mathscr{A}$ se e solo se $\bar{\delta}(q_0, w) \in T$.



Rappresentazione di un automa sull'alfabeto Σ = {a, b, c}

1.2 Matching di stringhe

Un classico problema è, dati i testi *pat* e *text*, verificare se *pat* occorre in *text*. Questo è un problema di decisione: l'output è un valore booleano. Si assume di solito che $m \leq n$, dove m è la lunghezza del pattern ed n è la lunghezza del testo. In questo modo, la taglia del problema è n.

Una versione avanzata, leggermente migliorata, riguarda la ricerca di tutte le occorrenze di *pat* in *text*, cioè, calcolare l'insieme di posizioni di *pat* in *text*. Si denoti con *MATCH(pat, text)* quest' insieme. Nella maggior parte dei casi un algoritmo che calcola *MATCH(pat, text)* è una modifica superflua dell'algoritmo di decisione, quindi spesso presentiamo solo l'algoritmo di decisione per il matching di stringhe.

1.2.1 Costruzione dell' automa di string-matching

Per un dato pattern *pat*, si costruisce il minimo automa finito deterministico che accetta tutte le parole contenenti *pat* come suffisso. Si denota tale automa con *SMA(pat)*.
Successivamente, definiremo il suffix automaton; mentre nel terzo capitolo daremo l'algoritmo per la costruzione del suffix automaton, più precisamente del grafo diretto aciclico della parola.

1.3 Strutture dati per indici

In questa sezione presentiamo alcune strutture dati per memorizzare i suffissi di un testo [2]. Queste sono concepite per fornire un accesso diretto e veloce ai fattori del testo. Le strutture possono essere usate per il pattern matching e viste come macchine di ricerca.

Vengono qui considerati gli alberi e automi. Gli alberi hanno per effetto di fattorizzare i prefissi delle stringhe nell'insieme. Gli automi in aggiunta fattorizzano i loro suffissi comuni.

La rappresentazione dei suffissi di una stringa da un trie ha il vantaggio di essere semplice ma può portare ad uno spazio di memoria quadratico a seconda della lunghezza della stringa considerata. Il (compatto) suffix tree evita quest'inconveniente e ammette un'implementazione lineare nello spazio. La minimizzazione dei suffix trie ci fornisce il minimo suffix automaton.

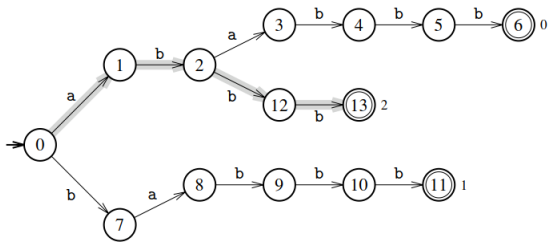
1.3.1 Suffix trie

Il ***suffix trie*** di una stringa è un automa deterministico che riconosce l'insieme dei suffissi della stringa e in cui due differenti cammini della stessa radice terminano in nodi diversi. La struttura del grafo sottostante dell'automa è un albero i cui archi sono etichettati da lettere.
Il suffix trie di una stringa y è l'albero $T(Suff(y))$ i cui nodi sono i fattori di y, ϵ è lo stato iniziale e i suffissi di y sono gli stati terminali. La funzione di transizione δ di $T(Suff(y))$ è definita da $\delta(u, a) = ua$ se ua è un fattore di y e $a \in \Sigma$. L'output di uno stato terminale, che è un suffisso, è la posizione di questo suffisso in y.

Diamo la definizione di *grado* di un vertice: in un grafo non orientato è il numero di archi incidenti sul vertice; in un grafo orientato il *grado uscente* di un vertice è il numero di archi uscenti, mentre quello *entrante* riguarda il numero di archi che entrano nel vertice.

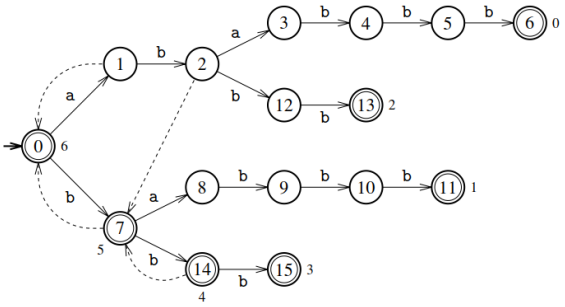
Chiamiamo *testa* (*head*) del suffisso corrente il suo più lungo prefisso comune ad un suffisso che occorre in una posizione più piccola. Se $y[i...k-1]$ è la testa del suffisso alla posizione i, la stringa $y[k...n-1]$ è chiamata *coda* del suffisso. Chiamiamo *bivio* dell'automa ogni stato che ha grado uscente di almeno 2, o che sia uno stato terminale di grado 1.

Mostriamo il trie $T(Suff(ababbb))$ durante la costruzione, subito dopo l'inserimento del suffisso *abbb*. Il *bivio*, stato 2, corrisponde alla *testa ab* di questo suffisso. *È* il più lungo prefisso di *abbb* che occorre prima della posizione del suffisso corrente. La *coda* del suffisso è *bb*, etichetta del cammino innestato dal *bivio* a questo passo della costruzione.



1.3.2 Suffix link

Sia av un suffisso di y che ha una *testa* az non vuota con $a \in \Sigma$. Il prefisso z di v occorre quindi in y prima dell'occorrenza considerata. Questo implica che z è un prefisso della *testa* del suffisso v. La ricerca per la *testa*, e per il *bivio* corrispondente, può essere in questo modo fatto dallo stato z invece di ricominciare sistematicamente dallo stato iniziale. Ciò assume che, dato lo stato az, esiste un accesso veloce allo stato z. Per questo, introduciamo una funzione sugli stati dell'automa, chiamata ***suffix link***. Denotata con *suff* e definita da *suff(az)*$= z$ per ogni stato az $(a \in \Sigma, z \in \Sigma^*)$. Lo stato *suff(az)* è chiamato ***suffix target*** di az.



L'automa $T(Suff(ababbb))$ con i suffix link delle biforcazioni e dei loro antenati indicati con frecce tratteggiate.

1.3.3 Suffix tree

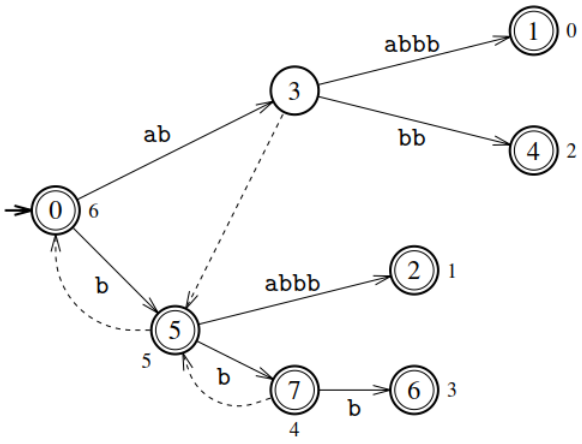
Il ***suffix tree*** per una stringa y di lunghezza n è un albero radicato tale che:

- vi sono esattamente n foglie numerate da 1 a n;
- ogni nodo interno esclusa al più la radice, ha almeno due figli;
- ogni arco è etichettato con una sottostrings di y;
- due archi uscenti dallo stesso nodo non possono avere etichette che iniziano con lo stesso carattere;
- la concatenazione delle etichette lungo il cammino dalla radice alla foglia numerata i è esattamente il suffisso $y[i, n]$ di y di lunghezza $n - i + 1$.

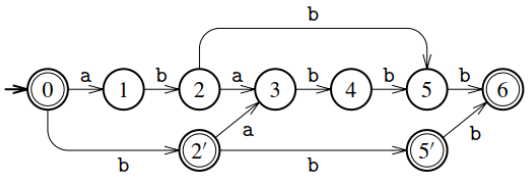
Il suffix tree di y, denotato con $T_c(y)$, dove c sta per *compresso*, (in quanto il suffix tree è una versione compressa del suffix trie) è ottenuto eliminando i nodi di grado 1 che non sono terminali nei suoi suffix trie $T(Suff(y))$. L'albero conserva le biforcazioni e i nodi terminali del suffix trie. Le etichette degli archi diventano quindi stringhe di lunghezza variabile. Notiamo che se due archi uscenti da uno stesso nodo sono etichettati da stringhe u e v, allora le loro prime lettere sono distinte. Questo accade perché il suffix trie è un automa deterministico.
Definiamo il *cammino* in un albero come una sequenza di vertici, senza ripetizioni, dove vertici successivi nella sequenza sono adiacenti. Un cammino è *massimale*, se non si può trovare un cammino con una lunghezza maggiore; oppure possiamo dire che, se un cammino è almeno lungo tanto quanto ogni altro cammino nell'albero, allora è *massimale*. È possibile che ci siano più cammini massimali.
Una *catena* in un albero è un cammino massimale, i cui nodi hanno grado uscente 1 ad eccezione dell'ultimo.

1.3.4 Suffix automaton

Il ***suffix automaton*** di una stringa y, denotato con $S(y)$, è un automa minimale che accetta l'insieme dei suffissi di y. La struttura è programmata per essere usata come un indice sulla stringa, ma costituisce anche una macchina per la ricerca di fattori di y all'interno di un altro testo. La proprietà più sorprendente di questo automa è che la sua taglia è lineare nella lunghezza di y, sebbene il numero di fattori di y può essere quadratico. Anche la costruzione dell'automa prende un tempo lineare su un alfabeto fissato. La taglia è espressa sia dal numero dei suoi stati e sia dal numero dei suoi archi. Il suffix automaton di y possiede meno di $2|y|$ stati e meno di $3|y|$ archi, per una taglia totale di $O(|y|)$.



Il suffix tree T_c(ababbb) con i suoi suffix link



Il suffix automaton S(ababbb), automa minimo che accetta i suffissi della stringa ababbb

Funzione di fallimento

Definiamo il ***suffix target*** di p, denotato con $f(p)$, e la funzione f chiamata ***suffix link*** dell'automa. Il suffix link è una funzione di fallimento e $f(p)$ è lo stato di fallimento di p.
Diamo la definizione formale della ***funzione di fallimento***. L'idea è di ridurre lo spazio necessario per l'implementazione dell'automa, reindirizzando la computazione della transizione dallo stato corrente a quello da un altro stato, ma della stessa lettera. Formalmente, siano

$$\gamma : Q \times \Sigma \to Q$$

e

$$f : Q \to Q$$

due funzioni. Diciamo che la coppia (γ, f) rappresenta la nuova funzione di transizione δ di un automa, dove γ risulta essere una sottofunzione di δ, f definisce un ordine sugli stati di Q, e per ogni coppia $(p, a) \in Q \times \Sigma$

$$\delta(\text{p,a}) = \begin{cases} \gamma(p,a) & \text{se } \gamma(p,a) \text{ è definita,} \\ \delta(f(p),a) & \text{altrimenti} \end{cases}$$

D'ora in poi indichiamo il collegamento della funzione di transizione con una freccia tratteggiata.

Per uno stato p di $S(y)$, denotiamo con $length(p)$ la lunghezza del più lungo cammino dallo stato iniziale a p, cammino etichettato dalla stringa u. I più lunghi cammini dallo stato iniziale formano uno spanning tree di $S(y)$. Gli archi che appartengono a questo albero sono qualificati come ***solidi***. L'arco (p, a, q) è solido se e solo se

$$length(q) = length(p) + 1.$$

Questa nozione di solidità degli archi è approfondita e usata nella costruzione del suffix DAWG nel Capitolo 3.

1.3.5 I grafi diretti aciclici della parola

L'automa $S(y)$ è essenzialmente la stessa struttura dati del così detto grafo diretto aciclico della parola, ***DAWG***. I DAWG sono grafi diretti i cui archi sono etichettati da simboli, e tale grafo rappresenta l'insieme di tutte le parole prendendo di cui viene fatto lo spelling dai suoi percorsi che partono dalla radice al fondo. I DAWG sono più convenienti dei suffix tree perché ogni arco del DAWG è etichettato con un singolo simbolo, mentre gli archi degli alberi dei fattori sono etichettati con parole di lunghezza variabile.
Oggetto del Capitolo 3 sarà proprio una costruzione del DAWG.

Capitolo 2

Una panoramica sul pattern matching

Abbiamo già detto che il problema del ***pattern matching*** è quello di cercare occorrenze di stringhe (detti pattern) in altre stringhe (detti ***testi***), e può anche essere esteso a linguaggi [2].
Gli output possono avere diverse forme:

- Valori booleani: l'output è semplicemente un valore booleano TRUE o FALSE che testimonia la presenza o meno di un pattern nel testo, senza specificare le posizioni delle possibili occorrenze.
- Una stringa: durante una ricerca sequenziale di y, è appropriato produrre una stringa $\bar{y}$ sull'alfabeto $\{0, 1\}$ che, per ogni posizione, indica con 0 o 1, rispettivamente, l'esistenza o assenza di un occorrenza del pattern in y. La stringa $\bar{y}$ è tale che $|\bar{y}| = |y|$ e $\bar{y}[i] = 1$ se e solo se i è la giusta posizione di un'occorrenza del pattern in y.
- Un insieme di posizioni: l'output può anche prendere la forma di un insieme P di posizioni sinistre di occorrenze del pattern su y.

2.1 Algoritmi di base

Il problema del matching di stringhe è quello maggiormente studiato nell'algoritmica delle parole e ci sono molti algoritmi per risolvere questo problema in modo efficiente [1]. Ricordiamo che il pattern *pat* è di lunghezza m e il testo *text* di lunghezza n, $m \leq n$.
Se l'unico accesso al testo *text* e al pattern *pat* è attraverso il confronto di simboli, allora è noto che il limite inferiore al massimo numero di confronti richiesti dall'algoritmo di string-matching è $n - m$. Il miglior algoritmo farà non più di $2 \times (n - m)$ confronti nel caso peggiore [1].

2.1.1 L'algoritmo Brute-Force

L'algoritmo di forza bruta controlla, in tutte le posizioni nel testo tra 0 ed $n-m$, se un'occorrenza del pattern inizia lì o meno. In seguito, dopo ogni tentativo, sposta il pattern di esattamente una posizione a destra.
L'algoritmo non ha bisogno di alcuna fase di preprocessing, ma necessita di uno spazio extra costante in aggiunta al pattern e al testo. Ovviamente il tempo di esecuzione è quadratico.

Un algoritmo così naive è all'origine di una serie di sempre più complicati ed efficienti algoritmi. Lo schema informale di tale algoritmo è:

`for` i `:=` 0 `to` n-m `do`

check if ($pat == text[i+1 \ldots m+1]$).

L'implementazione differisce a seconda di come implementiamo l'operazione di verifica, cioè se analizziamo il pattern da sinistra o da destra. In questo modo ne ricaviamo due algoritmi di forza bruta. Entrambi hanno una complessità di tempo quadratica nel caso peggiore. Mostreremo qui l'algoritmo che analizza il pattern da sinistra a destra.
Per abbreviare la presentazione di alcuni algoritmi, assumiamo che *pat* e *text*, siano variabili globali.

```
function brute_force1: boolean;
  var i, j : integer;
begin
  i := 0;
  while i≤n-m do begin
    { left-to-right scan of pat }
    j := 0;
    while j<m and pat[j+1]=text[i+j+1] do j := j+1;
    if j=m then return(true);
    { inv1(i, j) }
    i := i+1; { length of shift = 1 }
  end;
  return(false);
end;
```

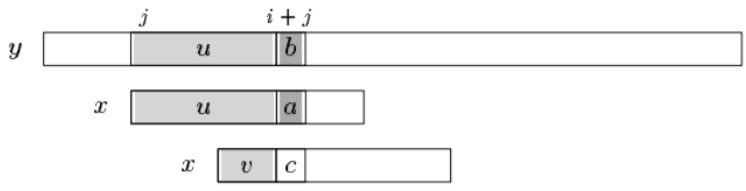
Guardando lo pseudocodice notiamo l'invariante principale dell'algoritmo, $inv1$, che definiamo nella seguente maniera.

$$inv1(i,j)\text{: } pat[1...j] = text[i+1...i+j] \text{ e } pat[j+1] \neq text[i+j+1].$$

Se $pat = a^{n/2}b$ e $text = a^{n-1}b$ allora abbiamo un numero quadratico di confronti dei simboli. Assumiamo che l'alfabeto abbia due simboli e che ognuno appaia con la stessa probabilità. Allora la probabilità che il test "`pat[j+1]==text[i+j+1]` " abbia successo è al massimo 1/2. Si può provare che il numero medio di tali test di successo per un fissato i non supera 1 e il numero di test senza successo non supera n.

2.1.2 L'algoritmo di Morris e Pratt

La progettazione dell'algoritmo di Morris-Pratt segue una stretta analisi dell'algoritmo di Forza Bruta, specialmente evitando di sprecare le informazioni guadagnate durante la scansione del testo.
Vediamo, quindi, come viene modificato l'algoritmo naive analizzato nel paragrafo precedente. È possibile migliorare la lunghezza degli spostamenti e contemporaneamente ricordare alcune porzioni del testo che si abbinano col pattern. Questo salva confronti tra i caratteri del pattern e caratteri del testo e di conseguenza incrementa la velocità di ricerca.
Consideriamo un tentativo alla posizione sinistra j su y, ovvero quando la finestra è posizionata sul fattore del testo $y[j \dots j+m-1]$. Assumiamo che la prima incompatibilità occorra tra $x[i]$ e $y[i+j]$ con $0 < i < m$. Allora, $x[0 \dots i-1] = y[j \dots i+j-1] = u$ ed $a = x[i] \neq y[i+j] = b$.
Quando si sta compiendo l'operazione di shift, è ragionevole aspettarsi che un prefisso v del pattern sia compatibile con qualche suffisso della porzione u del testo. Tale prefisso v, il più lungo, è chiamato ***bordo*** di u (occorre ad entrambe le estremità di u). Questo introduce la notazione: sia $Bord[i]$ la lunghezza del più lungo bordo di $x[0 \dots i\text{-}1]$ per $0 < i \leq m$. Allora, dopo uno shift, i confronti si possono riassumere tra i caratteri $c = x[Bord[i]]$ e $b = y[i+j]$ senza dimenticare alcuna occorrenza di x in y, ed evitando di tornare indietro nel testo. Il valore di $Bord[0]$ è settato a -1.



Shift nell'algoritmo di Morris-Pratt: v è il bordo di u

La tabella $Bord$ può essere calcolata con complessità di spazio e tempo dell'ordine di $O(m)$, prima della fase di ricerca, applicando lo stesso algoritmo di ricerca al pattern stesso, come se $x = y$.
In seguito la fase di ricerca può essere fatta in un tempo $O(m+n)$. L'algoritmo di Morris-Pratt compie al più $2n-1$ confronti dei caratteri del testo durante la fase di ricerca. Il **ritardo** (numero massimo di confronti per un singolo carattere di testo) è limitato da m.

```
function MP : boolean;  { algorithm of Morris and Pratt }
  var i, j : integer;
begin
  i := 0; j := 0;
  while i≤n-m do begin
    while j<m and pat[j+1]=text[i+j+1] do j := j+1;
    if j=m then return(true);
    i := i+j-Bord[j]; j := max(0, Bord[j]);
  end;
  return(false);
end;
```

L'algoritmo di Knuth-Morris-Pratt

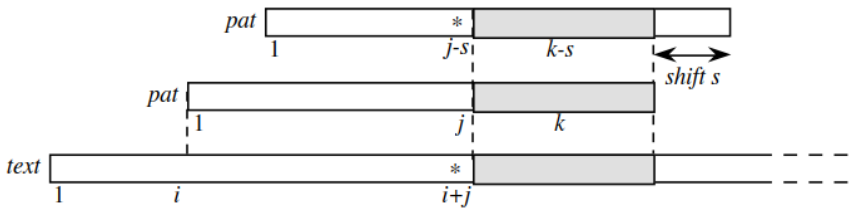
L'algoritmo di Knuth-Morris-Pratt (KMP) rappresenta un miglioramento dell'algoritmo precedente.
Questo nuovo algoritmo, che incorpora la proprietà di incompatibilità, migliora il numero di confronti compiuti su una data lettera nel testo.
La chiave per il miglioramento è la seguente: si assuma che un'incompatibilità occorra alla lettera $pat[j+1]$ del pattern. Il confronto successivo è tra la stessa lettera del testo e $pat[k+1]$ se $k = Bord[j]$. Ma se $pat[k+1]$=$pat[j+1]$ la stessa incompatibilità ricorre. Per questo dobbiamo evitarlo considerando il bordo di $pat[1...j]$ di lunghezza k.
Nell'algoritmo di Morris e Pratt veniva calcolata per una data posizione j, $Bord[j]$; mentre qui, per migliorare l'efficienza di KMP, abbiamo la tabella $s\_Border[j]$, definita come k, quando k è il più piccolo intero che soddisfa la seguente condizione: $pat[1...k]$ è un suffisso proprio di $pat[1...j]$ e $pat[k+1] \neq pat[j+1]$. Altrimenti $s\_Bord[j]$ vale -1.

2.2 L'algoritmo di Boyer-Moore

L'algoritmo di Boyer-Moore è considerato il più efficiente algoritmo di matching di stringhe nelle tipiche applicazioni. Una versione semplificata o l'intera procedura è spesso implementata negli editor di testo per i comandi "ricerca" e "sostituisci".
Il presente algoritmo analizza attentamente l'informazione sprecata nel precedente algoritmo di forza bruta. Quest'informazione è correlata all'invariante

inv2: pat[j+1...m]=text[i+j+1...i+m] and pat[j]≠ text[i+j].

L'informazione raccolta dall'algoritmo è memorizzata nel valore di j. Tuttavia, all'iterazione successiva, quest'informazione è cancellata dato che a j viene assegnato un valore fissato. Supponiamo di voler fare uno spostamento maggiore.



Caso in cui s<j

Lo spostamento s è chiamato *sicuro* se siamo certi che tra i e $i+s$ non c'è alcuna posizione di inizio per il pattern nel testo. Supponiamo che il pattern appaia alla posizione $i+s$. Abbiamo così la seguente condizione:

cond1(j,s) : per ogni k tale che $j < k \leq m$, *s≥k o pat[k-s]= pat[k],*
cond2(j,s) : se s<j allora pat[j-s]≠pat[j].

Definiamo due tipi di shift, ognuno associato ad un suffisso del pattern rappresentato dalla posizione j ($< m$) e definito dalla sua lunghezza:

D1[j] : min{s>0 : cond1(j,s) ha luogo},
D[j] : min{s>0 : cond1(j,s) e cond2(j,s) accadono}.

Definiamo anche $D1[m] = D[m] = m - Bord[m]$. L'algoritmo di Boyer-Moore, quando accade un'incompatibilità, esegue uno shift di lunghezza $D[j]$ invece di uno spostamento di una sola posizione.
Mostriamo, di seguito, lo pseudocodice del presente algoritmo.

2.3 Matching di stringhe basato sui suffix tree

Le strutture dati di base presenti rappresentano tutti i fattori di una data parola ovvero l'insieme $Fac(text)$ di tutti i fattori del testo $text$ di lunghezza n. La

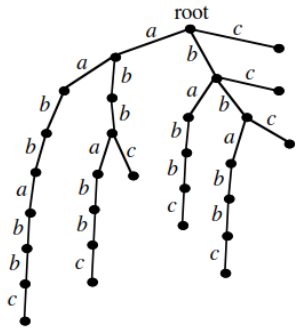
```
function BM : boolean;
{ improved version of brute_force2 }
begin
  i := 0;
  while i≤n-m do begin
    j := m;
    while j>0 and pat[j]=text[i+j] do j := j-1;
    if j=0 then return(true);
    { inv2(i, j) }
    i := i+D[j];
  end;
  return(false);
end;
```

sua taglia è solitamente quadratica, ma esistono rappresentazioni succinte che sono lineari nella loro taglia. Vedremo anche i *subword graph* di un testo, poiché hanno la stessa origine dei loro alberi dei suffissi.
Il problema più tipico è quello di verificare se una data parola x è in $Fac(text)$. Si denoti il corrispondente predicato con $FACTORIN(x, text)$. L'utilità della struttura dati per questo problema significa che $FACTORIN(x, text)$ può essere calcolato in tempo $O(|x|)$, anche se x è molto più corto di $text$. La complessità qui è calcolata a seconda dell'operazione base della struttura, *branching*, che richiede un'unità di tempo.
La struttura dati D che rappresenta l'insieme $Fac(text)$ è buona se:

- D ha taglia lineare;
- D può essere costruita in tempo lineare;
- D permette di calcolare $FACTORIN(x, text)$ in tempo $O(|x|)$.

L'albero in figura è un tipo di struttura chiamata *trie*, usata nell'algoritmo di McCreight [1].
In questi alberi, i collegamenti da un nodo a i suoi figli sono etichettati da lettere. Nell'albero associato a $text$, un cammino verso il basso dell'albero fa lo spelling di un fattore di $text$. Tutti i cammini dalla radice alle foglie fanno lo spelling di suffissi di $text$. E tutti i suffissi di $text$ sono etichette dei cammini dalla radice.

La tecnica principale usata per la costruzione dell'albero è chiamata ***UP-LINK-DOWN***. Questa è la chiave per il miglioramento della costruzione. Viene usata per trovare, partendo dall'ultima foglia creata, il nodo (*head*) dove un nuovo cammino deve essere innestato. La struttura dati incorpora collegamenti



Trie dei fattori di *aabbabbc*, con otto foglie che corrispondono agli otto, non vuoti, suffissi.

che rappresentano delle scorciatoie per velocizzare la ricerca di *head*. La strategia *Up_link_down* è la seguente: risale l'albero partendo dall'ultima foglia finché non si trova una scorciatoia attraverso un link esistente; va poi attraverso il collegamento e inizia a scendere nell'albero per trovare la nuova *head*.
La funzione *Up_link_down* è lo strumento di base usato negli algoritmi per la costruzione di suffix tree e DAWG. Il tempo di una singola chiamata della funzione può essere proporzionale all'intera lunghezza del pattern, ma la somma di tutti i costi è lineare.

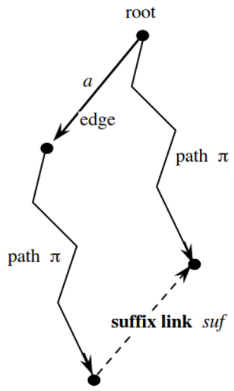
Per il pattern $text = a_1 a_2 \cdots a_n$, definiamo p_i con il suffisso $a_i a_{i+1} ... a_n$. $Trie(p_1, p_2, \cdots, p_n)$ è l'albero i cui rami sono etichettati con i suffissi $p_1, p_2, ..., p_i$. In questo albero, $leaf_i$ è la foglia corrispondente al suffisso p_i e $head_i$ è il primo antenato di $leaf_i$ avente almeno due figli; è la radice se non esiste tale nodo.

Per un nodo corrispondente ad un fattore non vuoto aw, definiamo il ***suffix link*** da aw a w, con $suf[aw] = w$.

La tabella suf servirà per creare scorciatoie durante la costruzione del trie. Sono i link menzionati nella funzione Up_link_down. Nel trie può accadere che $suf[v]$ è indefinito, per qualche nodo v. Questa situazione non si presenta per i suffix tree.

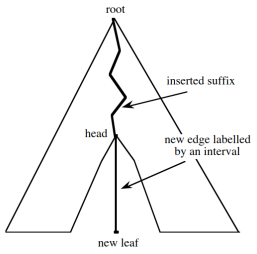
2.3.1 L'algoritmo di McCreight

Un approccio diretto per la costruzione di $T(text)$ potrebbe essere il seguente: prima costruire $Trie(text)$, successivamente comprimere tutte le sue catene. Il principale svantaggio di questo schema è che le dimensioni di $Trie(text)$ possono essere quadratiche, risultando l'algoritmo di tempo e spazio quadratici.
L'algoritmo è di tipo incrementale. L'albero è calcolato per un sottoinsieme di suffissi consecutivi. In seguito il suffisso successivo è inserito nell'albero, questo continua finché tutti i suffissi (non vuoti) sono inclusi nell'albero.
Si consideri la struttura del cammino corrispondente al nuovo suffisso p inse-



Un suffix link suf

rito nell'albero T. Tale cammino è indicato nella figura dalla linea centrale in grassetto.



Inserimento di un suffisso nell'albero

Si denoti con $insert(p, T)$ l'albero ottenuto da T dopo l'inserimento della stringa p. Il cammino corrispondente a p in $insert(p, T)$ termina nell'ultima foglia creata dell'albero. Si denoti con *head* il padre di questa foglia. Potrebbe accadere che il nodo *head* non sia già presente nell'albero iniziale T (è solo un nodo implicito al momento della costruzione), e deve essere creato durante l'operazione di inserimento. Ad esempio, ciò succede se proviamo a inserire il cammino $p = abcdeababba$ che inizia al nodo v (si guardi la figura alla pagina successiva). In questo caso, un arco di T (etichettato con $abadc$) deve essere diviso. Questo perché consideriamo l'operazione di *break* definita in seguito.

Diamo qui di seguito la nozione di nodo *implicito*. L'obiettivo è quello di ripristinare i nodi di $Trie(text)$ all'interno del suffix tree $T(text)$. Diciamo che una coppia (w, α) è un nodo *implicito* in T se w è un nodo di T e α è un prefisso proprio dell'etichetta di un arco da w a un figlio di w. Se $val(w) = x$, allora $val((w, \alpha)) = x\alpha$. Un nodo *implicito* corrisponde ad un posto dove in seguito un nuovo nodo verrà creato. Il nodo *implicito* (w, α) è *reale* se α è la parola

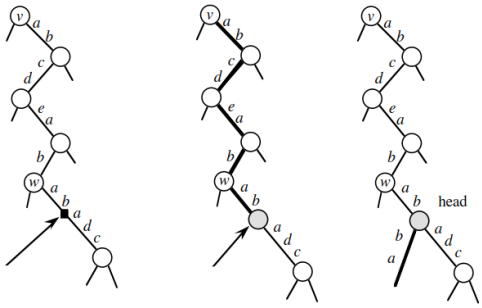
vuota. In questo caso (w, ϵ) è identificato con il nodo w stesso.

Sia (w,α) un nodo *implicito* dell'albero T (w è un nodo di T, α è una parola). L'operazione di $break(w, \alpha)$ sull'albero è definita solo se esiste un arco uscente dal nodo w la cui etichetta δ ha α come prefisso. Sia β tale che $\delta = \alpha\beta$. L'effetto (collaterale) dell'operazione di $break(w, \alpha)$ è la rottura dell'arco corrispondente: un nuovo nodo è inserito al punto di rottura, l'arco è diviso in due archi con rispettive etichette α e β. Il valore di $break(w, \alpha)$ è il nodo creato al punto di rottura.

Sia v un nodo di T, e sia p una sottoparola della parola di input $text$ rappresentata da una coppia di interi l, r, dove $p = text[l...r]$. La funzione di base usata nella costruzione dell'albero dei suffissi è la funzione $find$. Il valore di $find(v, p)$ è l'ultimo nodo *implicito* lungo il cammino che inizia in v ed etichettato da p. Se questo nodo *implicito* non è reale, è (w, α) per quache α non vuota, e la funzione $find$ lo converte nel nodo "reale" $break(w, \alpha)$.

Nell'algoritmo vengono usate due differenti implementazioni della funzione $find$.
La prima, chiamata $fastfind$, impiegata quando conosciamo in anticipo che il cammino di ricerca etichettato da p è pienamente contenuto in qualche cammino che comincia in v. Questa conoscenza ci permette di trovare il nodo desiderato molto più velocemente usando gli archi compressi dell'albero come scorciatoie.
La seconda implementazione di $find$ è la funzione $slowfind$ che segue il suo cammino lettera per lettera. L'applicazione di $fastfind$ è una caratteristica fondamentale dell'algoritmo di McCreight, e gioca una parte centrale nella sua performance (insieme ai collegamenti).

Per esempio, consideriamo il suffix tree a sinistra. Proviamo a inserire la stringa $p = abcdeababba$, il risultato di $fastfind$ e $slowfind$ è il nuovo nodo creato $find(v, p)$. $fastfind$ compie 5 passi, mentre $slowfind$ ne compie $O(|p|)$. Nella prima figura, la freccia indica l'ultimo nodo *implicito* sul cammino di ricerca. La seconda immagine invece punta al nodo creato $break(w, ab) = find(v, p)$



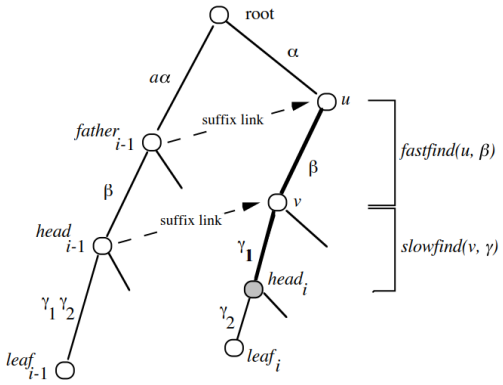
L'algoritmo di McCreight costruisce una sequenza di alberi compatti T_i nell'ordine i = 1,2,...,n. L'albero T_i contiene i più lunghi i-esimi suffissi di $text$. Si noti che T_n è l'albero dei suffissi $T(text)$, ma gli alberi intermedi non sono strettamente suffix tree.

Ad un dato stadio dell'algoritmo di McCreight, abbiamo $T = T_{k-1}$, e tentiamo la costruzione di T_k. La tavola dei suffix link, chiamata suf, gioca un ruolo cruciale nella riduzione della complessità.
Se il cammino dalla radice al nodo v ha una divisione in lettere pari a $a\pi$ allora suf[v] è definito come il nodo corrispondente al cammino π. Nell'algoritmo, la tavola suf è calcolata ad ogni passo per tutti i nodi, ad eccezione delle foglie e (forse) dell'$head$ corrente.

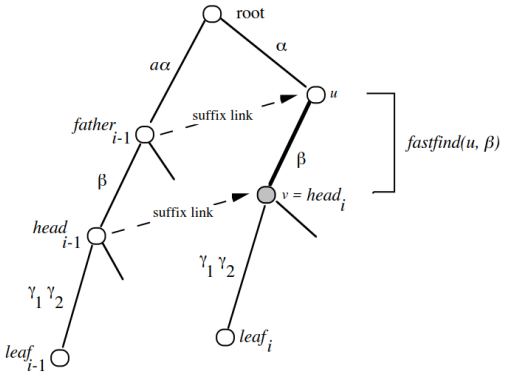
L'algoritmo è basato sulle due seguenti ovvie proprietà:

- $head_i$ è un discendente del nodo $suf[head_{i-1}]$,
- $suf[v]$ è un discendente di suf[father(v)] per ogni v, dove $father$ risulta essere il *padre* del nodo v.

Il lavoro basilare fatto dall'algoritmo di McCreight è speso nella localizzazione degli *head*. Se è fatto in modo naive (ricerca dalla cima al fondo partendo dalla radice) allora il tempo è quadratico. La chiave per il miglioramento è la relazione tra $head_i$ e $head_{i-1}$. Dunque, la ricerca per il prossimo *head* può iniziare dai nodi foglia dell'albero invece che partire dalla radice. Questo ci permette di risparmiare del lavoro e la complessità ammortizzata è lineare.



Algoritmo di McCreight: caso in cui v è un nodo esistente.



Algoritmo di McCreight: caso in cui $v = head_i$ è un nodo appena creato.

```
Algorithm McCreight;
begin
  T := two-node tree with one edge labelled by p1=text;
  for i:=2 to n do begin
    { insert next suffix p_i = text[i..n] }
    let β be the label of the edge (father[head_{i-1}], head_{i-1});
    let γ be the label of the edge (head_{i-1}, leaf_{i-1});
    u := suf[father[head_{i-1}]];
    v := fastfind(u, β);
    if v has only one son then
      { v is a newly inserted node } head_i := v
    else head_i := slowfind(v, γ);
    suf[head_{i-1}] := v;
    create a new leaf leaf_i; make leaf_i a son of head_i;
    label the edge (head_i, leaf_i) accordingly;
  end
end.
```

Capitolo 3

Forward DAWG Matching algorithm

3.1 Descrizione

L'algoritmo *Forward Dawg Matching* calcola il più lungo fattore del pattern che termina in ogni posizione nel testo.
Questo è reso possibile dall'impiego del più piccolo suffix automaton del pattern, chiamato anche ***(suffix) DAWG***, ossia Directed Acyclic Word Graph, ovvero grafo diretto aciclico della parola. Il più piccolo suffix automaton di una parola w è un automa finito deterministico *S(w) = (Q, q_0, T, Σ)*. Il linguaggio accettato da $S(w)$ è $L(S(w))$={ $\forall$ u in Σ^*: $\exists$ v in Σ^* tale che $w = vu$}.
La fase di preprocessing del *Forward dawg matching algorithm* consiste nel calcolo del più piccolo suffix automaton per il pattern x. È lineare in tempo e spazio nella lunghezza del pattern.

Durante la fase di ricerca l'algoritmo analizza i caratteri del testo da sinistra a destra con l'automa *S(x)* iniziando con lo stato q_0.

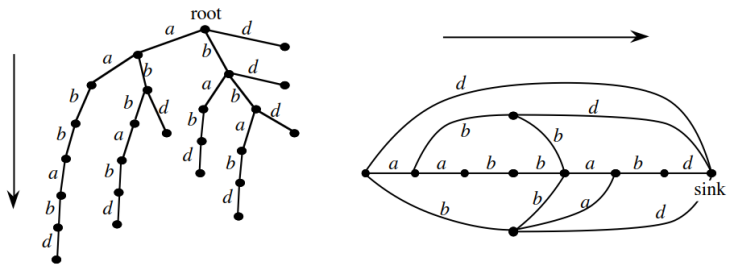
Per ogni stato q in *S(x)* il più lungo cammino da q_0 a p è denotato con *length(q)*. Questa struttura fa estensivamente uso della nozione di *suffix link*. Per ogni stato p il *suffix link* di p è denotato con *suff[p]*. Per ogni stato p, sia $Path(p) = (p_0, p_1, ..., p)$ il suffix path di p tale che $p_0 = p$, per $1 \leq i \leq l$, $p_i = suff[p_{i-1}]$ e $p_l = q_0$. Per ogni carattere del testo $y[j]$, sia p lo stato corrente. L'algoritmo prende una transizione definita su $y[j]$, partendo dal primo stato di $Path(p)$, per il quale una tale transizione è definita (cioè esiste un arco uscente da tale stato con etichetta $y[j]$). Lo stato corrente p è aggiornato con lo stato target di questa transizione o con lo stato iniziale q_0, se non esistono transizioni etichettate con $y[j]$ che partono da uno stato di $Path(p)$. Un'occorrenza di x è trovata quando $length(p) = m$. Il *Forward Dawg Matching algorithm* fa esattamente n visite dei caratteri del testo.

3.2 Il DAWG

Il ***grafo diretto aciclico della parola*** è una buona struttura dati che rappresenta l'insieme $Fac(text)$ di tutti i fattori della parola $text$ di lunghezza n. Il $DAWG$ è un'alternativa al suffix tree. Il grafo $DAWG(text)$, chiamato il $DAWG$ dei suffissi di $text$, è ottenuto dall'identificazione di sottoalberi isomorfi dell'albero non compatto $Trie(text)$ che rappresenta $Fac(text)$.

Un vantaggio dei DAWG è che ogni arco è etichettato da un singolo simbolo. In qualche modo è più conveniente da usare quando le informazioni devono essere associate agli archi invece che ai nodi.
Analizzeremo due sorprendenti impieghi di questo tipo di grafo: nel primo, vedremo come il suffix DAWG del pattern serve per cercare quest'ultimo all'interno di un testo, invece il secondo metodo ci porta ad uno dei più efficienti algoritmi per cercare un pattern.
Un nodo nel grafo $DAWG(text)$ naturalmente corrisponde ad un insieme di fattori del testo.
Tutti questi fattori hanno la seguente proprietà: le loro prime occorrenze terminano nella stessa posizione in $text$.
Sia x un fattore di $text$, denotiamo con *end-pos(x)* l'insieme di tutte le posizioni in $text$ in cui termina un'occorrenza di x. Sia y un altro fattore di $text$. Allora i sottoalberi di $Trie(text)$ che hanno radici rispettivamente x e y sono isomorfi se *end-pos(x) = end-pos(y)*.
Nel grafo $DAWG(text)$, cammini x aventi lo stesso insieme *end-pos(x)* portano allo stesso nodo. I nodi di G corrispondono ai insiemi non vuoti della forma $end - pos(x)$. La radice del DAWG corrisponde all'intero insieme di posizioni $\{0, 1, 2, ..., n\}$ sul testo.

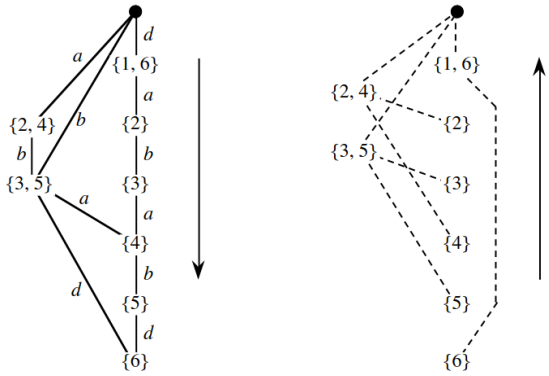
La figura sottostante presenta sia il trie del testo *aabbabd* sia il DAWG per la stessa parola. Il grafo ha 10 nodi e 15 archi.



A sinistra il *trie* e a destra il *DAWG* per la stessa parola *"aabbabd"*

Le piccole dimensioni del DAWG sono dovute alla struttura speciale della famiglia Φ degli insiemi *end-pos*. Associamo ad ogni nodo v del DAWG il suo valore $val(v)$ equivalente alla più lunga parola che porta ad esso partendo dalla radice.

La nozione di ***funzione di fallimento*** è usata anche nei DAWG ed è legata al concetto di suffix link.
Sia v un nodo di $DAWG(text)$ distinto dalla radice. Definiamo $suf[v]$ come il nodo w tale che $val(w)$ è il più lungo suffisso di $val(v)$ non equivalente a esso. In altri termini, $val(w)$ è il più lungo suffisso di $val(v)$ corrispondente ad un nodo diverso da v. Definiamo $suf[root]$ uguale alla radice $root$. Nell'implementazione suf è rappresentato come una tabella, chiamata tabella dei *suffix link* (gli archi $(v, suf[v])$ sono *suffix link*).



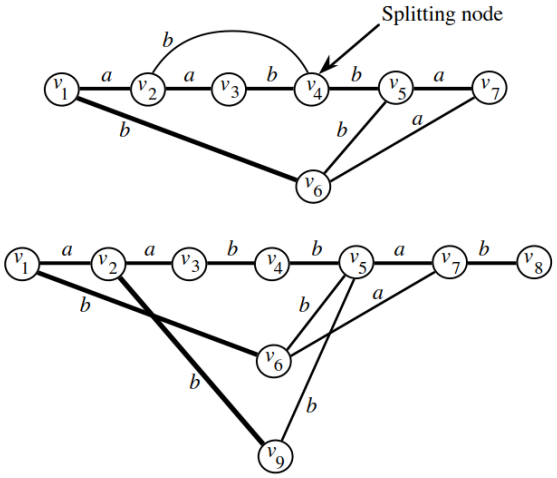
A sinistra il $DAWG(dababd)$ e a destra i suoi suffix link.

Dato che $suf[v]$ è una parola strettamente più corta di $val(v)$, suf induce una struttura ad albero sull'insieme di nodi. Il nodo $suf[v]$ è interpretato come il padre di v nell'albero.
La taglia di $DAWG(text)$ è lineare. Più precisamente, se N è il numero di nodi del DAWG e $n = |text| > 0$, allora $N < 2n$. Oltretutto, $DAWG(text)$ ha meno di $N + n - 1$ archi. Questo è indipendente dalla taglia dell'alfabeto.

3.2.1 Costruzione del DAWG

Descriviamo l'algoritmo lineare per il calcolo del suffix DAWG. L'algoritmo processa il testo da sinistra a destra. Ad ogni passo, legge la successiva lettera del testo e aggiorna il DAWG costruito fino a quel momento. Nel corso dell'algoritmo, due tipi di archi sono considerati: archi *solidi* e *non solidi*. Gli archi solidi sono quelli contenuti nei più lunghi cammini dalla radice. In altri termini, gli archi non solidi sono quelli che creano scorciatoie nel DAWG. L'aggettivo

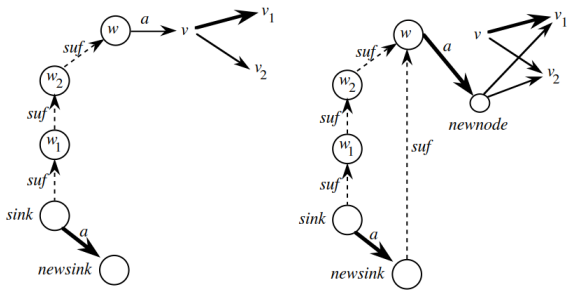
solido si riferisce al fatto che una volta che questi archi vengono creati, non sono modificati durante la costruzione. Al contrario, l'obiettivo degli archi non solidi può cambiare durante l'esecuzione dell'algoritmo.
Mostriamo uno stadio dell'algoritmo dove è rappresentata la trasformazione di $DAWG(aabba)$ in $DAWG(aabbab)$, che mette in luce un'operazione cruciale dell'algoritmo.



Trasformazione del $DAWG(aabba)$ nel $DAWG(aabbab)$. Gli archi solidi sono in grassetto. Il nodo v_9 è un clone del nodo v_4

L'inserimento della lettera b in $aabba$ aggiunge nuovi fattori all'insieme $Fac(aabba)$. Sono tutti suffissi di $aabbab$. Nel $DAWG(aabba)$, i nodi corrispondenti ai suffissi di $aabba$, v_1, v_2, v_7, avranno ora un arco uscente con etichetta b. Consideriamo il nodo v_2; ha un arco uscente non solido con etichetta b. L'arco è una scorciatoia tra v_2 e v_4, in confronto al più lungo cammino da v_2 a v_4 che è etichettato con ab. I due fattori ab e aab sono associati al nodo v_4. Ma in $aabbab$, solo ab diventa un suffisso e non aab. Ecco spiegato il motivo per cui v_4 è diviso in v_4 e v_9 nel $DAWG(aabbab)$. Facendo in questo modo non si presenteranno problemi nel caso in cui i due nodi avranno comportamenti diversi. Nell'algoritmo, denotiamo con $son(w, a)$ il figlio v del nodo w tale che $label(w, v) = a$.
Sui nodi del DAWG è definito un suffix link chiamato suf.

Il working path è w_1, $w_2 = suf[w_1]$, $w_3 = suf[w_2]$, $w = suf[w_3]$. Generalmente, il nodo w è il primo nodo sul cammino che ha un arco uscente (w, v) etichettato dalla lettera a. Se questo arco è non solido, allora v è diviso in due nodi: v stesso e *newnode*, il quale è un clone di v, nel senso che gli archi uscenti e il suffix link di *newnnode* sono gli stessi di v. Il suffix link di *newsink* è assegnato a *newnode*. Altrimenti se l'arco (w, v) è solido, l'unica azione fatta a questo passo è di porre il suffix link di *newsink* a v.



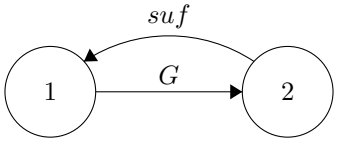
Stadio dell'algoritmo con la lettera a. L'arco non solido a uscente da w è trasformato in un arco solido entrante in un clone di v.

Classificare gli archi del DAWG in solidi e non solidi è una caratteristica importante dell'algoritmo, usata per conoscere quando un nodo deve essere diviso. Tuttavia possiamo evitare di ricordare se un nodo è solido o meno. La categoria di un nodo può essere testata dal suo valore $length(v)$ del più lungo cammino dalla radice a v. Infatti, si ha che l'arco (v, w) è solido se $|length(v)| + 1 = |length(w)|$.

3.2.2 Esempio di costruzione del DAWG

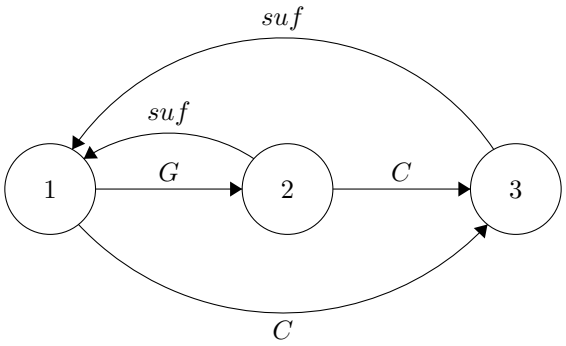
Illustriamo i passi della costruzione del DAWG sul pattern "$GCAGAGAG$" di lunghezza 8.

Nei primi passi dell'algoritmo, abbiamo la creazione del nodo radice del grafo, il cui suffix link è settato ad un valore nullo. Per la prima iterazione del ciclo `for`, istanziamo un nuovo nodo; dato che dalla radice non parte nessun arco con etichetta uguale al carattere del pattern nella posizione i, allora esso viene creato. Abbiamo ora un arco *solido* etichettato dal carattere i-esimo, che collega il nodo radice ed il nuovo nodo. Viene, alla fine imposto che il suffix link del nuovo vertice sia la radice.

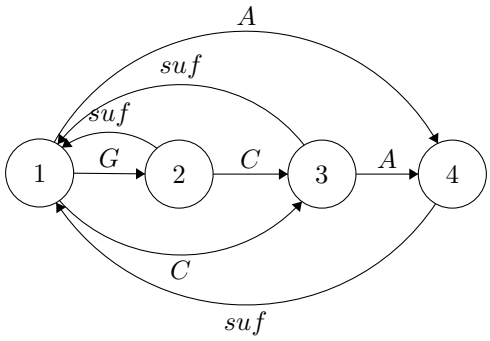


Nella seconda iterazione dell'algoritmo creiamo un nuovo collegamento per il successivo carattere del pattern, uscente dal secondo nodo ed entrante in uno appena creato. Si torna indietro nel grafo attraverso i suffix link e si arriva alla radice, dal quale non parte alcun arco etichettato dalla lettera i-esima. Allora creiamo un arco *non solido*, collegando così la radice e il nuovo nodo e ponendo di nuovo il vertice iniziale, come il suffix link del nuovo nodo.

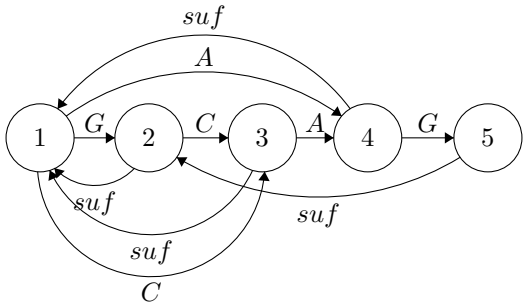
```
Algorithm suffix-dawg;
begin
  create the one-node graph G = DAWG(ε);
  root := sink := the node of G; suf[root] := nil;
  for i := 1 to n do begin
    a := text[i];
    create a new node newsink;
    make a solid edge (sink, newsink) labelled by a;
    w := suf[sink];
    while (w≠nil) and (son(w, a)=nil) do begin
      make a non-solid a-edge (w, newsink); w := suf[w];
    end;
    v := son(w, a);
    if w=nil then suf[newsink] := root
    else if (w, v) is a solid edge then suf[newsink] := v
    else begin { split the node v }
      create a node newnode;
      newnode has the same outgoing edges as v
      except that they are all non-solid;
      change (w, v) into a solid edge (w, newnode);
      suf[newsink] := newnode;
      suf[newnode] := suf[v]; suf[v]:=newnode;
      w := suf[w];
      while (w≠nil) and ((w, v) is a non-solid a-edge) do begin
      {*} redirect this edge to newnode; w :=suf[w];
      end;
    end;
    sink:=newsink;
  end;
end.
```



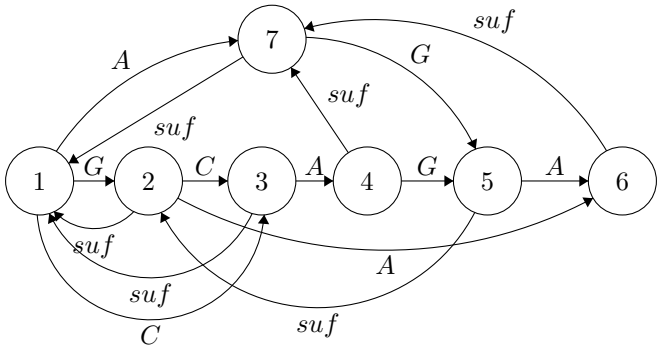
Questo terzo passo non è molto diverso dal precedente: dal nodo 3 viene creato un arco *solido* che entra nel nuovo nodo 4, lo stesso viene fatto per la radice, aggiungendo però un arco *non solido* con tale etichetta; infine si pone la radice come il suffix link del nuovo vertice.



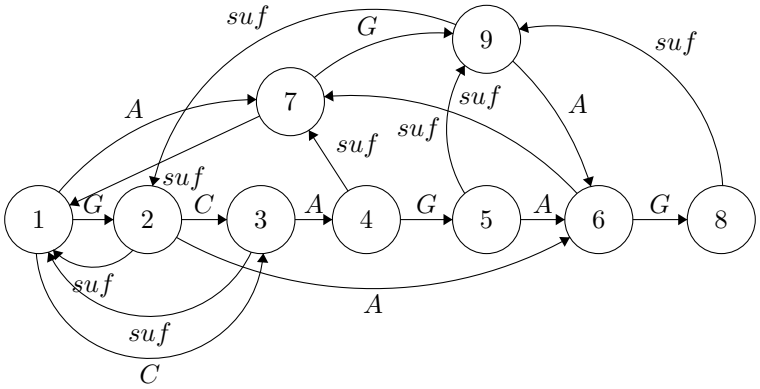
Nella quarta fase creiamo un arco *solido* dal nodo 4 al 5 con etichetta G, poi attraverso il suffix link di 4 arriviamo al nodo 1. Da esso parte già un arco con tale etichetta, che porta nel nodo 2, il quale diventa il suffix link del nuovo nodo 5.



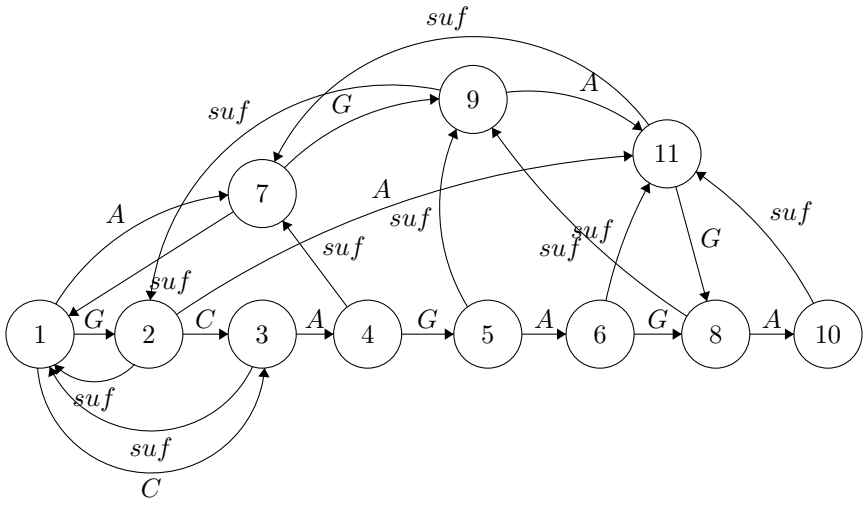
In questo passo mostriamo un'operazione interessante, ovvero quella della *divisione* di un nodo. Innanzitutto creiamo un arco *solido* uscente da 5 ed entrante nel nuovo nodo 6 e attraverso i suffix link raggiungiamo il nodo 2, dal quale non parte alcun arco con etichetta A, per questo ne creiamo uno *non solido* verso 6. Il suffix link di 2 è il nodo 1 e notiamo che esiste un arco *non solido* uscente da esso con etichetta A ed entrante in 4.
Inizia l'operazione di *divisione* del nodo 4. Creiamo un nuovo nodo con gli stessi archi uscenti di 4, cambiamo l'obiettivo della radice sull'arco A, reindirizzandolo al nuovo nodo 7; infine impostiamo i vari suffix link, il nodo 7 diventa il suffix link di 4 e 6, mentre quello del nodo 7 è il nodo iniziale.



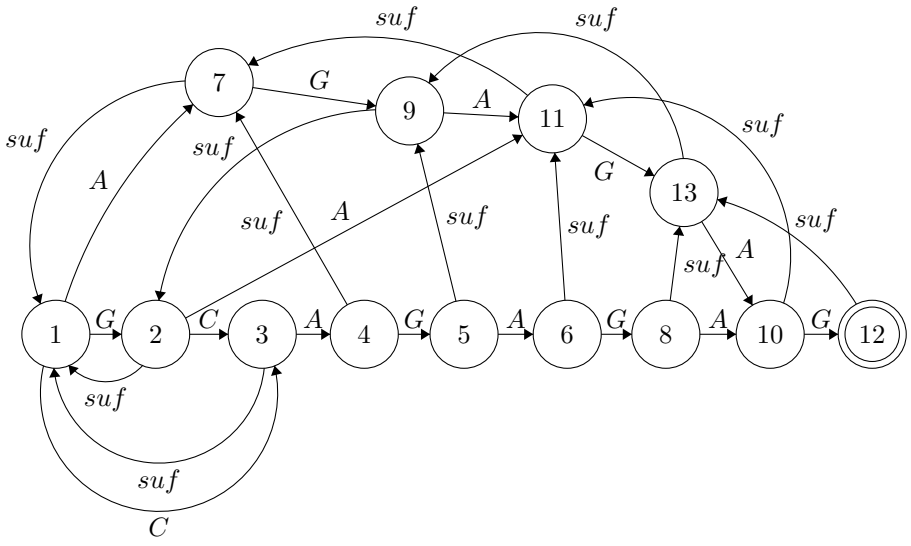
Anche in questa iterazione, possiamo notare l'operazione di *divisione* compiuta sul nodo 5. Dato che è simile a quella della fase precedente, tralasciamo la sua spiegazione e mostriamo soltanto l'automa dopo tale iterazione.



Questa penultima fase, simile alle precedenti, prevede la *divisione* del nodo 6 con la conseguente creazione del nodo 11. Interessante notare come, indicato dall'ultimo ciclo `while` dell'algoritmo, dopo la creazione del nodo 11e i vari reindirizzamenti degli archi, arriviamo al nodo 2 (attraverso il suffix link del nodo 9) e qui cambiamo l'obiettivo dell'arco *non solido* con etichetta A di tale nodo, indirizzandolo verso il nodo 11. Si continua il ciclo tornando al nodo 1, il quale però ha già un arco *solido* con etichetta A e per questo non c'è bisogno di cambiarlo; si esce, in questo modo, dal ciclo `while`.



Mostriamo il DAWG completo del pattern, contenente anche tutti i suffix link dell'automa.



3.2.3 Relazioni tra suffix tree e suffix DAWG

Suffix tree e suffix DAWG rappresentano entrambi l'insieme di fattori $Fac(text)$. La relazione tra le due strutture è ovvia poiché sono versioni compatte dello stesso oggetto: l'albero $Trie(text)$. Comprimere le catene (cammini che consistono di nodi di grado uscente uno), che produce la struttura dati *suffix tree* del testo, e l'unione di sottoalberi isomorfi, che porta alla creazione del DAWG del testo, sono due tipi di compressione dell'albero $Trie(text)$.
Ognuno di questi metodi ha i suo vantaggi e svantaggi.

Il primo metodo produce un albero; questo è un vantaggio dovuto al fatto che è più facile avere a che fare con gli alberi rispetto ad altri tipi di grafi. Per di più, nell'albero alle foglie possono essere assegnati valori particolari. Lo svantaggio invece riguarda la lunghezza variabile delle etichette degli archi.
Un vantaggio del secondo metodo è che ogni arco è etichettato da un solo simbolo. Questo ci permette di attaccare informazioni ai simboli degli archi. E lo svantaggio principale dei DAWG è proprio quello di non essere alberi.
La taglia lineare dei suffix tree è di poco conto, ma quella dei DAWG è ancora più banale. Probabilmente è uno dei più sorprendenti fatti legati alle rappresentazioni di $Fac(text)$.

3.3 Matching di stringhe usando i DAWG: Forward Dawg Matching algorithm

L'algoritmo che analizziamo è basato sulla costruzione di un suffix DAWG sul pattern. Una semplice applicazione dei DAWG fornisce un algoritmo efficiente per calcolare il più lungo fattore comune di due stringhe. Abbiamo già discusso un algoritmo per questo scopo nel capitolo precedente che era basato sui suffix tree.
La presente soluzione è abbastanza diversa. Prima di tutto, calcola il più lungo fattore comune mentre viene analizzata una delle due parole di input (la più lunga). Secondo aspetto a variare riguarda l'uso del DAWG di una sola parola (la più corta); ciò rende l'intero processo più economico rispetto allo spazio impiegato, poiché in memoria conserviamo il DAWG del pattern (che in genere ha dimensioni ridotte rispetto al testo e, non dimentichiamo che tale grafo potrebbe anche essere riutillizzato in altre esecuzioni dell'algortimo, usando stringhe di testo differenti per la ricerca).

L'algoritmo ***Forward dawg matching*** cerca nel testo i fattori del pattern. Calcola il più lungo fattore del pattern che termina ad ogni posizione nel testo. Siano pat e $text$ due parole. Sia $DAWG(pat)$ il DAWG del pattern. Il testo viene analizzato da sinistra a destra. Ad ogni passo, sia p il prefisso di $text$ che è stato appena processato. Allora conosciamo il più lungo suffisso s di p che è un fattore di pat, e dobbiamo calcolare lo stesso valore associato al prossimo prefisso pa di $text$. *È* chiaro che il DAWG può aiutare nel calcolarlo efficientemente se sa è un fattore di pat. Ma questo è ancora vero se sa non è un fattore di pat perché in questa situazione possiamo usare il ***suffix link*** del DAWG in modo simile alla funzione di fallimento usata nell'algoritmo di Morris e Pratt. In una situazione corrente dell'algoritmo, memorizziamo uno stato di $DAWG(pat)$. Ma la localizzazione di un'occorrenza di pat in $text$ non può essere fatta solo basandosi sullo stato del DAWG, perché gli stati sono di solito raggiungibili attraverso diversi cammini. Perciò viene calcolata anche la lunghezza len del più lungo fattore di pat che termina alla presente posizione nel testo. Quando i suffix link sono usati per calcolare il prossimo fattore, l'algoritmo resetta il valore di len con

l'aiuto della funzione *length* definita sugli stati del DAWG. La funzione, ricordiamo, che rappresenta la lunghezza del più lungo cammino dalla radice al nodo.

Presentiamo, di seguito, lo pseudocodice dell'algoritmo.

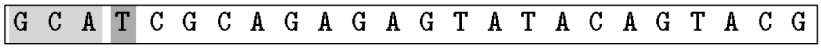
```
function FDM : boolean; { forward-dawg-matching algorithm }
{ it computes longest factors of pat occurring in text }
begin
  D := DAWG(pat);
  len := 0; w := root of D; i := 0;
  while not i≤|text| do begin
    if there is an edge (w, v) labelled by text[i] then begin
      len := len+1; w := v;
    end else begin
      repeat w := suf[w];
      until (w undefined) or
            (there is (w, v) labelled by text[i]);
      if w undefined then begin
        len := 0; w := root of D;
      end else begin
        let (w, v) be the edge labelled by text[i] from w;
        len := length(w)+1; w := v;
      end;
    { len = larger length of factor of pat ending at i in text }
      if len=|pat| then return(true);
      i := i+1;
  end;
  return(false);
end;
```

3.3.1 Esempio di esecuzione dell'algoritmo

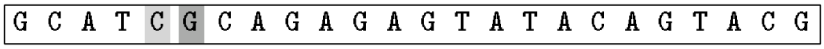
Mostriamo l'esecuzione del nostro algoritmo utilizzando il grafo costruito in precedenza sul pattern "$GCAGAGAG$", cercando sue occorrenze nella stringa di testo "$GCATCGCAGAGAGTATACAGTACG$" di lunghezza 24.

Una volta costruito il DAWG, si parte con la ricerca nel testo di un'occorrenza del pattern.
Partendo dallo stato iniziale dell'automa, vediamo se esiste un arco con etichetta uguale al primo carattere della stringa di testo; nel nostro caso dal nodo 1 parte un arco G. Quindi ci spostiamo allo stato 2, incrementiamo il valore della

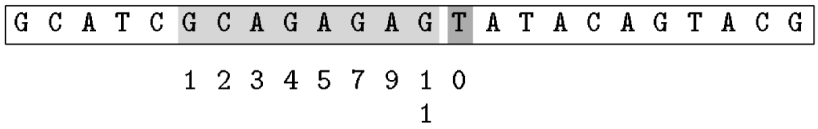
variabile *len* che memorizza la lunghezza del fattore fin'ora trovato del pattern, e continuiamo alla ricerca del prossimo matching.
Arrivati alla quarta iterazione siamo nello stato 4 del DAWG e il carattere i-esimo del testo è T. Da questo stato però non parte alcun arco con tale etichetta, per questo ci risalire nell'automa, attraverso i suffix link, alla ricerca di uno stato dal quale parta un arco etichettato da T. Tale collegamento non esiste nell'automa, allora viene azzerato il valore di *len* e si ricomincia la ricerca del pattern nel testo dalla radice del DAWG.



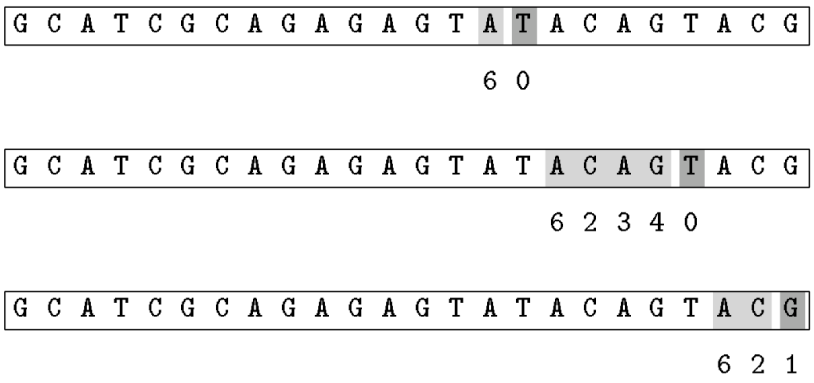
Nella successiva iterazione abbiamo un match per il carattere C, dato che dallo stato iniziale tale arco esiste. Vediamo nel passo seguente che dal vertice 2 non parte alcun arco etichettato da G; allora si torna indietro nell'automa alla ricerca di un vertice dal quale parta quell'arco. Troviamo una corrispondenza nel vertice 1, quindi possiamo continuare senza dover resettare il valore di *len* e continuiamo la ricerca dallo stato 2 dell'automa.



Vediamo che nelle successive iterazioni si susseguono continui successi e si avanza nel DAWG fino ad arrivare allo stato finale. Il valore di *len* risulta essere uguale alla lunghezza del pattern, allora possiamo dire di aver trovato almeno un'occorrenza di esso in corrispondenza della posizione 5 del testo. Facciamo notare che nello pseudocodice l'algoritmo ritorna semplicemente `true` quando tale condizione è soddisfatta, in quanto viene indicato, da tale implementazione, solo quando è trovato l'intero pattern nel testo.



Mostriamo alcune delle successive fasi, nelle quali si continua a scorrere la stringa ma senza riuscire a trovare un'altra occorrenza del pattern.



L'algoritmo termina dopo aver compiuto 24 ispezioni dei caratteri del testo.

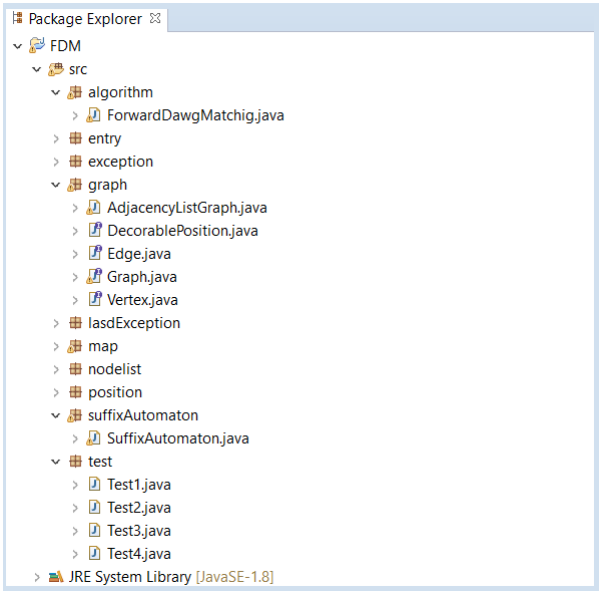
Capitolo 4

Implementazione

Nel seguente capitolo viene mostrato come si è deciso di implementare l'algoritmo per la costruzione del suffix automaton di un dato pattern e l'algoritmo *Forward Dawg Matching*. Un'implementazione degli algoritmi era presente nel liguaggio di programmazione C [4].
Il linguaggio di programmazione usato è *Java*, l'ambiente di sviluppo è *Eclipse*. Le strutture dati impiegate sono state fornite durante il corso di *Strutture dati*, le cui informazioni sono riprese dal libro di testo [3]. Le aggiunte che abbiamo fatto nell'implementazizone sono state evidenziate e messe in corsivo nel codice. Queste hanno consentito di ottenere anche le occorrenze dei fattori del pattern nel testo, output non possibile con il codice originale in *C*.

4.1 Panoramica sulle operazioni di base

Abbiamo integrato e modificato le strutture a nostra disposizione, per poterci allineare con quelle presentate da [4].
L'astrazione principale utilizzata è quella del grafo orientato, che da noi è stata implementata con liste di adiacenza.
Riportiamo il workspace del nostro progetto di tesi FDM, mostrando i pacchetti e le classi da cui è composto:



Notiamo la presenza dei package usati durante il corso di *Strutture dati,* quali *entry*, *exception*, *graph*, *lasdException*, *map*, *nodelist*, *position*, che contengono le implementazioni necessarie per la creazione dell'automa e dell'esecuzione dell'algoritmo.
Nel pacchetto *suffixAutomaton* la funzione principale è *buildSuffixAutomaton*, che da un grafo orientato, costruisce l'automa che riconosce i fattori di una data stringa.
Nel pacchetto *algorithm* abbiamo il codice del nostro algoritmo di pattern matching.
Infine il pacchetto *test*, che come ci suggerisce il nome, contiene le classi di *testing* del'algoritmo.

4.1.1 Strutture dati

Lista di posizioni

La *lista di posizioni* è un contenitore di oggetti, il quale memorizza ciascun elemento in una ***posizione*** e mantiene le posizioni degli oggetti in un ordinamento lineare. Il concetto di posizione formalizza l'idea di ***posto*** di un elemento. Possimo dire, semplicemente, che la posizione è il nome che permette di riferirisi all'elemento ad esso associato. La posizione è associata sempre allo stesso elemento nella lista e non cambia, neanche in seguito ad operazioni quali inserimenti o cancellazioni.

La posizione definisce essa stessa un tipo astratto di dati che supporta l'unico metodo `element()`, il quale restituisce l'elemento in *quella* posizione.

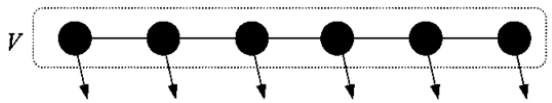
Iteratore

Un *iteratore* permette la scansione di un qualsiasi tipo di dato astratto che rappresenta una collezione ordinata di elementi.
L'iteratore è molto utile in quanto un'operazione molto frequente sulle strutture è quella di scandire la collezione di elementi, un elemento alla volta.
Un iteratore può essere visto come un'estensione del concetto di posizione, poiché, come detto in precedenza, la posizione incapsula il concetto di *posto*, mentre l'iteratore quello di ***posto*** e ***posto successivo***.
Un iteratore supporta i metodi `hasNext()`, il quale determina se ci sono altri elementi nell'iteratore, `next()`, che restituisce l'elemento successivo in un iteratore.
Dato che l'iteratore deve essere indipendente dal modo con cui è organizzata la collezione di oggetti a cui è associato, abbiamo bisogno di un meccanismo uniforme per la scansione di una struttura dati ad accesso sequenziale; per questo motivo, tutti i tipi dati di questo tipo devono supportare il metodo `iterator()`, il quale restituisce un iteratore degli elementi della collezione.

Grafo

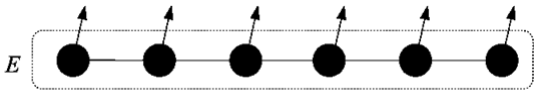
Un grafo G è costituito da due insiemi (V, E) dove V è l'insieme dei ***vertici***, anche detti nodi, ed E è l'insieme delle coppie (u, v), con u e v in V, dette ***archi***.
Nel grafo G, si dice che l'arco diretto (u, v) è ***incidente*** da u a v, se l'arco esce da u ed entra in v; allora i due vertici sono ***adiacenti*** tra di loro.
Il ***grado*** di un nodo v corrisponde al numero di archi incidenti di v.

Il tipo di dati astratto grafo è un contenitore di elementi memorizzati nelle posizioni dei vertici e degli archi che li collegano. Abbiamo utilizzato l'implementazione dei grafi mediante ***liste di adiacenza***. Ciascun vertice del grafo è rappresentato da un oggetto di tipo *vertex* e tutti questi oggetti sono memorizzati in un contenitore V.

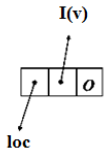


La stessa cosa per gli archi, dove ognuno è rappresentato da un oggetto di tipo *edge* e tutti gli oggetti *edge* sono memorizzati in un contenitore E.

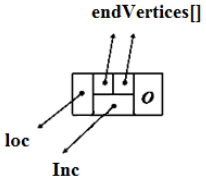
L'oggetto *vertex* per un vertice v che memorizza l'elemento o ha un riferimento a quest'ultimo e un riferimento alla posizione dell'oggetto stesso nel



contenitore V dei vertici del grafo, un riferimento ad un contenitore $I(v)$ (contenitore di incidenza di v) i cui elementi hanno i riferimenti agli archi incidenti su v.



L'oggetto *edge* per un arco *e=(u,v)* che memorizza l'elemento o ha un riferimento a quest'ultimo, un riferimento agli oggetti *vertex* per u e v, un riferimento alla posizione dell'oggetto *edge* nel contenitore E degli archi del grafo, un riferimento alle posizioni associate all'arco e nei contenitori di incidenza $I(u)$ e $I(v)$.



La rappresentazione di G consiste di un array Adj di $|V|$ liste (una per vertice). Per ogni u in V, la lista $Adj[u]$ contiene tutti i vertici v adiacenti ad u in G.
Per i grafi orientati lo spazio di memoria richiesto è uguale alla somma delle lunghezze delle liste, $|E|$, che risulta essere $\Theta(|V| + |E|)$.
Le operazioni `insertVertex(o)`, `insertEdge(v,w,o)`, `removeEdge(e)` richiedono tempo lineare, invece i metodi `incidentEdges(v)`, `removeVertex(v)` hanno complessità $O(deg(v))$ (dove *deg* sta per *degree* ovvero il grado di un vertice), infine `areAdjacent(v,w)` è $O(min(deg(v), deg(w)))$.

La classe *AdjacencyListGraph* fornisce gli strumenti necessari per la creazione e manipolazione di un grafo. Di seguito mostriamo le classi *MyVertex* e

MyEdge estrapolate da essa e spieghiamo alcune delle operazioni fondamentali per la creazione di un automa, tralasciando quei metodi banali del tipo *get* e *set*, che erano già presenti nella struttura dati fornita. Si darà, quindi, importanza ai metodi aggiunti nelle implementazioni esistenti.

L'integrazione necessaria per l'utilizzo della classe *MyVertex* ha previsto l'aggiunta di alcuni campi :

- `initial`, intero che, se uguale a 1, indica che il vertice corrente è quello iniziale;
- `terminal`, intero che, se uguale a 1, indica che il vertice è quello finale dell'automa;
- `length`, intero che specifica la lunghezza del vertice nel grafo, ovvero il numero di nodi che intercorrono nel cammino dalla radice al suo raggiungimento;
- `sufEdges`, una lista di archi, che conterrà un solo elemento per ogni vertice, ovvero l'arco del suffix link.

```
1  public class MyVertex<V> extends MyPosition<V> implements Vertex<V> {
2
3        protected PositionList<Edge<E>> incEdges;
4
5        protected PositionList<Edge<E>> sufEdges;
6
7        protected Position<Vertex<V>> loc;
8
9        protected int length, terminal, initial;
10
11       MyVertex(V o) {
12         elem = o;
13         terminal = 0;
14         initial= 0;
15         length=0;
16         incEdges = new NodePositionList<Edge<E>>();
17         sufEdges = new NodePositionList<Edge<E>>();
18       }
19
20       public int degree() {
21         return incEdges.size();
22       }
23
24       public Iterable<Edge<E>> sufEdges() {
25         return sufEdges;
26       }
```

```
27
28        public Position<Edge<E>> insertSuf(Edge<E> e) {
29          sufEdges.addLast(e);
30          return sufEdges.last();
31        }
32
33
34        public void removeSuf(Position<Edge<E>> p) {
35          sufEdges.remove(p);
36        }
37
38        public Iterable<Edge<E>> incidentEdges() {
39          return incEdges;
40        }
41
42        public Position<Edge<E>> insertIncidence(Edge<E> e) {
43          incEdges.addLast(e);
44          return incEdges.last();
45        }
46
47        public void removeIncidence(Position<Edge<E>> p) {
48          incEdges.remove(p);
49        }
50
51        public Position<Vertex<V>> location() {
52          return loc;
53        }
54
55        public void setLocation(Position<Vertex<V>> p) {
56          loc = p;
57        }
58
59        public String toString() {
60          return elem.toString();
61        }
62      }
```

Invece la classe *MyEdge* ha soltanto un campo in più, ovvero il *boolean* `type`, che di default è settato a $false$: indica se l'arco è associato ad un suffix link, ovvero è l'arco della transizione di fallimento f, oppure è un arco della funzione di transizione δ dell'automa.

```
1   public class MyEdge<E> extends MyPosition<E> implements Edge<E> {
2
3         protected MyVertex<V>[] endVertices;
4
5         protected Position<Edge<E>>[] Inc;
```

```
6
7         protected Position<Edge<E>> loc;
8
9         protected boolean type;
10
11        MyEdge (Vertex<V> v, Vertex<V> w, E o) {
12          elem = o;
13          type = false;
14          endVertices = (MyVertex<V>[]) new MyVertex[2];
15          endVertices[0] = (MyVertex<V>)v;
16          endVertices[1] = (MyVertex<V>)w;
17          Inc = (Position<Edge<E>>[]) new Position[1];
18        }
19
20        public boolean isSuffix() {
21          if(type)
22             return true;
23          else
24             return false;
25        }
26
27        public E getElement(){
28          return elem;
29        }
30
31        public void setSuffix(boolean isSuffix) {
32          type= true;
33        }
34
35        public MyVertex<V>[] endVertices() {
36          return endVertices;
37        }
38
39        public Position<Edge<E>>[] incidences() {
40          return Inc;
41        }
42
43        public void setIncidences(Position<Edge<E>> pv) {
44          Inc[0] = pv;
45        }
46
47        public Position<Edge<E>> location() {
48          return loc;
49        }
50
51        public void setLocation(Position<Edge<E>> p) {
52          loc = p;
53        }
54
```

```
55        public String toString() {
56          return "(" + endVertices[0].toString() + ")";
57        }
58      }
```

Il metodo `getTarget(Vertex<V> v, Character c)`, chiamato su un dato grafo, ha come parametri un vertice v e un carattere c. Il metodo restituisce un vertice u, tale che esiste un arco uscente da v etichettato con c verso u. Il corrispondente metodo set, `setTarget(Vertex<V> v, Character c, Vertex<V> u)`, crea un arco uscente da v con etichetta c ed entrante in u.

Il metodo `getSuffixLink(Vertex<V> v)`, per un dato vertice v, restituisce il vertice u che rappresenta il suffix link di v. Il metodo `insertSuff(Vertex<V> v, Vertex<V> u, Character c)` crea un arco da v a u, in modo tale che u diventi il suffix link di v. Dato che non è di alcuna rilevanza per il funzionamento dell'algoritmo quale sia l'etichetta dell'arco del suffix link, questo viene semplicemente etichettato con la lettera s.

Il metodo `copyVertex(Vertex<V> target, Vertex<V> source)` copia in target tutti gli archi incidenti uscenti da source.

```
1     public Vertex<V> getTarget(Vertex<V> v, Character c){
2           MyVertex<V> vv = checkVertex(v);
3           MyEdge<E> ee = new MyEdge<E>(null, null, null);
4           Iterable<Edge<E>> list = vv.incidentEdges();
5           Iterator<Edge<E>> iterator = list.iterator();
6           while(iterator.hasNext()){
7              MyEdge<E> e = (MyEdge<E>)iterator.next();
8                 if(e.getElement() == c)
9                    ee = e;
10          }
11          if(ee != null)
12             return opposite(vv, ee);
13          else return null;
14
15    }
16
17    public Edge<E> setTarget(Vertex<V> v, Character c, Vertex<V> u)
          throws InvalidPositionException{
18          MyVertex<V> vv = checkVertex(v);
19          MyVertex<V> uu = checkVertex(u);
20          return insertEdge(vv, uu, (E) c);
21    }
22
23    public Vertex<V> getSuffixLink(Vertex<V> v){
24          MyVertex<V> vv = checkVertex(v);
25          MyVertex<V> uu= null;
26          Iterable<Edge<E>> list = vv.sufEdges();
```

```
27          Iterator<Edge<E>> iterator = list.iterator();
28          while(iterator.hasNext()){
29               MyEdge<E> e = (MyEdge<E>)iterator.next();
30               if(opposite(vv, e) != null)
31                  uu = (MyVertex<V>) opposite(vv, e);
32             }
33              if(uu != null)
34                return uu;
35               else return null;
36     }
37
38     public Edge<E> insertSuff(Vertex<V> u, Vertex<V> v, Character c)
           throws InvalidPositionException {
39             MyVertex<V> uu = checkVertex(u);
40
41             MyVertex<V> vv = checkVertex(v);
42             MyEdge<E> ee = new MyEdge<E>(u, v, (E)c);
43             ee.setSuffix(true);
44             Position<Edge<E>> pu = uu.insertSuf(ee);
45             ee.setIncidences(pu);
46             EList.addLast(ee);
47             Position<Edge<E>> pe = EList.last();
48             ee.setLocation(pe);
49             return ee;
50     }
51
52     public void copyVertex(Vertex<V> target, Vertex<V> source){
53          MyVertex ss = checkVertex(source);
54          MyVertex tt = checkVertex(target);
55          tt.setLocation(ss.location());
56          Iterable<Edge<E>> list = ss.incidentEdges();
57          Iterator<Edge<E>> iterator = list.iterator();
58          while(iterator.hasNext()){
59             MyEdge<E> e = (MyEdge<E>)iterator.next();
60             MyVertex<V> vv = (MyVertex<V>) opposite(ss, e);
61             setTarget(tt, (Character) e.getElement(), vv);
62          }
63          Iterable<Edge<E>> sufList = ss.sufEdges();
64          Iterator<Edge<E>> sufIterator = sufList.iterator();
65          while(sufIterator.hasNext()){
66             MyEdge<E> e = (MyEdge<E>)sufIterator.next();
67             MyVertex<V> vv = (MyVertex<V>) opposite(ss, e);
68             insertSuff(tt, vv, 's');
69           }
70     }
```

4.2 Il metodo *buildSuffixAutomaton*

La classe *SuffixAutomaton* contiene semplicemente il metodo statico `builSuffixAutomaton.`

Quest'ultimo ha come parametri:

- la stringa x, che rappresenta il pattern per cui deve essere costruito l'automa;
- la lunghezza m del pattern x;
- un *grafo* con il quale verrà creato il suffix automaton.

Innanzitutto vengono dichiarate alcune variabili, come il carattere `c` che sarà etichetta di un arco tra due vertici, l'intero `vertexC` contatore del numero di nodi nel grafo, usato per il loro inserimento durante la costruzione del suffix automaton; infine, il vertice `init`. La prima operazione che viene fatta è quella di rendere `init`, il nodo iniziale del nostro automa. Il valore del contatore dei vertici viene incrementato, azione che viene compiuta ogni volta dopo l'aggiunta di un nodo nel grafo; si continua ponendo la lunghezza di `init` a `0`. Alla riga 10 istanziamo un nuovo nodo, `art`, che diventerà il suffix link del nodo iniziale; poco dopo, un nuovo vertice, `last`, assumerà il valore di `init`.
A questo punto inizia la vera e propria creazione dell'automa, che copre la maggior parte del metodo (linee 18-62). Il ciclo `for`, viene iterato tante volte quanto è il valore della lunghezza del pattern.
Al carattere `c` viene assegnato quello del pattern nella posizione `i`, dove `i` rappresenta la corrente iterazione del ciclo.

Due nuovi nodi sono creati: `p` e `q`, che rispettivamente, assumeranno lo stesso valore di last, mentre l'altro verrà inserito nel grafo.

Il ciclo `while` (righe 25-28) viene eseguito finché `p` non assume il valore di `init` e finché è vera la condizione, per cui non esiste un vertice con arco entrante con etichetta uguale a `c` che esce dal nodo `p`. Se queste condizioni sono entrambe vere, viene creato un arco orientato da `p` a `q` etichettato da `c`; `p` diventa, in seguito, il nodo che è suffix link di `p` stesso.
Se una delle due condizioni non è soddisfatta si controlla perché si è usciti dal ciclo `while`. L'`if` alle righe 29-32 controlla se è accaduto perché siamo finiti nello stato iniziale (`p` è uguale a `init`). Allora crea un arco da `init` a `q` con etichetta `c` e fa diventare `init` il suffix link di `q`.

Dopo la condizione di `if` appena illustrata, abbiamo l'`else` (righe 33-60). Subito abbiamo un'interrogazione: l'algoritmo controlla se il nodo con arco entrante, etichettato con `c`, che parte da p, dista una sola posizione dal nodo p. Questo controllo, come abbiamo già spiegato nei capitoli precedenti, permette di capire se un arco è di tipo *solido* o *non solido*. Se la lunghezza coincide, allora l'arco è *solido* e il suo *target* non viene più cambiato durante la costruzione dell'automa; viene inserito un suffix link da `q` al nodo *target* di `p` sul carattere `c`.
Se invece non è così, l'arco è non solido e le operazioni compiute sono le seguenti:

- Viene creato un nuovo vertice `r`.
- Il vertice target di `p` su `c` viene copiato in `r`; questo significa che `r` avrà gli stessi archi del nodo sorgente del metodo `copyVertex` (stessi target e stesso suffix link).
- Viene imposta la lunghezza di `r` uguale a quella di `p` +1.
- Viene cambiato il suffix link del target di `p` su `c`, prima lo si rimuove e poi lo si inserisce nuovamente usando le due funzioni alle righe 43-45, facendo diventare `r` il nuovo suffix link.
- Si controlla con un `if` se il nodo `q` non ha un suffix link, se ciò è vero, poniamo, anche in questo caso, `r` come il suffix link di `q`.
- Vi è un ciclo `while` che opera fin quando lo stato `p` non è uguale al vertice `art` e finchè la lunghezza del target di `p` con il carattere `c` è maggiore o uguale alla lunghezza di `r`. Se le due condizioni sono soddisfatte si elimina l'arco uscente da `p` etichettato con `c` verso quel vecchio nodo target, e si pone `r` come nuovo obiettivo di `p` per il carattere `c`.
- Infine `p` assume il valore del nodo che è il suffix link di `p`.

Fuori dal nostro `else`, quello che resta da fare è assegnare a `last` il valore di `q` e ricominciare con la prossima iterazione (linea 61).
Una volta fuori dal ciclo, non ci resta che settare il valore *terminal* del nodo `last` a 1 (riga 63), poiché esso è lo stato finale del nostro suffix automaton.

```
1  public static void builSuffixAutomaton(String x, int m,
       AdjacencyListGraph graph){
2        char c;
3        int vertexC= 0;
4        MyVertex init = (MyVertex) graph.insertVertex(0);
5        graph.setInitial(init);
6        vertexC++;
7
8        graph.setLength(init, 0);
9
10       MyVertex art = (MyVertex) graph.insertVertex(vertexC);
11
12       vertexC++;
13       graph.setLength(art, 0);
14
15       MyEdge suf1 = (MyEdge) graph.insertSuff(init, art, 's');
16
17       MyVertex last = init;
18       for(int i = 0; i < m; ++i){
19          c = x.charAt(i);
20          MyVertex p = last;
21          MyVertex q = (MyVertex) graph.insertVertex(vertexC);
```

```
22          vertexC++;
23          graph.setLength(q, graph.getLength(p)+1);
24
25          while(p!= init && (graph.getTarget(p, c)== null)){
26             graph.setTarget(p, c, q);
27             p = (MyVertex) graph.getSuffixLink(p);
28          }
29          if(graph.getTarget(p, c) == null){
30             graph.setTarget(init, c, q);
31             graph.insertSuff(q, init, 's');
32          }
33          else{
34             if(((graph.getLength(p))+1) ==
                    (graph.getLength(graph.getTarget(p, c)))){
35                graph.insertSuff(q, graph.getTarget(p, c), 's');
36             }
37             else{
38                MyVertex r = (MyVertex) graph.insertVertex(vertexC);
39                vertexC++;
40                graph.copyVertex(r, graph.getTarget(p, c));
41                graph.setLength(r, graph.getLength(p)+1);
42
43                graph.removeSuff(graph.getSufEdge(graph.getTarget(p, c)));
44
45                graph.insertSuff(graph.getTarget(p, c), r, 's');
46                if(graph.getSufEdge(q).element() != null){
47                   graph.removeSuff(graph.getSufEdge(q));
48                }
49                graph.insertSuff(q, r, 's');
50
51                while(p!= art && graph.getLength(graph.getTarget(p, c)) >=
                       graph.getLength(r)){
52                   graph.removeEdge(graph.getTargetEdge(p, c));
53                   graph.setTarget(p, c, r);
54                   p = (MyVertex) graph.getSuffixLink(p);
55                }
56
57             }
58
59
60          }
61          last = q;
62       }
63       graph.setTerminal(last);
64    }
```

4.3 L'algoritmo *Forward Dawg Matching*

Il cuore dell'algoritmo è contenuto nella classe *ForwardDawgMatching*, in cui troviamo il metodo statico FDM. I parametri sono la stringa `x` del pattern, la sua lunghezza `m`, la stringa di testo `y` e la lunghezza `n`.

Le prime operazioni compiute riguardano l'inizializzazione delle variabili, tra cui gli interi `j`, utilizzato per le iterazioni del ciclo `for`, `ell`, che tiene traccia della lunghezza del fattore del pattern che viene trovato nel testo e fact, campo ausiliario per la stampa del messaggio di output (viene incrementato solo se almeno un campo di `temp` è diverso da zero, ciò implica che si è trovato almeno un fattore nel testo). Subito dopo vi è un vettore di interi `temp`, che associato ad ogni iterazione, ci permette di memorizzare, la lunghezza del fattore trovato e in seguito di calcolare l'indice del testo in cui inizia l'occorrenza della sottoparola di `x`.
Vengono creati i nodi `init`, che come suggerisce il nome, è un riferimento al primo vertice del suffix automaton, e `state`.
Istanziamo il grafo (riga 7), con il quale verrà richiamato il metodo `buildSuffixAutomaton` (riga 9). Inizia l'algoritmo vero e proprio con il ciclo `for` che si estende dalle righe 14-46, e scorre tutta la stringa del testo. Nel primo `if`, si vuole controllare se esiste un nodo con arco entrante etichettato dal carattere `j`-esimo della stringa `y`, che parte da `state` (il quale è stato settato a `init` prima dell'inizio del ciclo). Se tale nodo è presente nel grafo, la variabile `ell` è incrementata e `state` diventa tale vertice. Il successivo `if` (righe 18-20) controlla se l'iterazione in cui ci troviamo è l'ultima, ovvero siamo all'ultimo carattere di `y`; se è così memorizziamo il fattore trovato nel vettore `temp`.

Alle righe 22-43 inizia l'`else`, in cui troviamo un `while`, nel quale abbiamo due condizioni che devono essere entrambe vere; la prima chiede che `state` sia diverso da `init`, la seconda invece si assicura la mancanza di un target per `state` sul carattere `y[j]`. Nel ciclo viene semplicemente preso il vertice che è il suffix link del corrente nodo `state`.
Fuori dal ciclo controlliamo perché siamo usciti; se il nodo target esiste, allora memorizziamo in `temp[j]` il valore assunto da `ell`, poichè abbiamo trovato una sottosequenza del pattern, aggiorniamo `ell` in modo tale che assuma il valore della lunghezza del nodo `state` incrementato di 1; infine assegniamo a `state` il nodo target. Se siamo nello stato iniziale e non abbiamo trovato un vertice collegato a `state` da un arco orientato che parte da quest'ultimo e con etichetta `c`, allora ci ricordiamo di memorizzare la lunghezza del fattore trovato in `temp[j]`, poniamo `ell` a zero e `state` viene aggiornato allo stato iniziale.
Al termine di ogni iterazione viene controllato il valore di `ell`, se esso è pari alla lunghezza del pattern, possiamo affermare di aver trovato un'intera occorrenza di `x` nel testo e stampiamo l'indice in cui inizia tale fattore.

L'ultimo ciclo `for` (linee 47-53) ci permette di scorrere il vettore `temp` e di stampare l'indice in cui i fattori sono comparsi con le rispettive lunghezze, se

essi esistono. Se gli elementi di tale vettore sono tutti uguale 0 non abbiamo incrementato il valore di `fact`, il quale risulterà essere ancora `0`. Quindi, se è questo il caso, viene stampato a video che non sono stati trovati fattori del pattern nel testo.

```
1  public static void FDM(String x, int m, String y, int n){
2
3        int j, ell=0, fact=0;
4        int[] temp = new int[n+1];
5        Vertex init, state;
6
7        AdjacencyListGraph grafo = new AdjacencyListGraph();
8
9        SuffixAutomaton.builSuffixAutomaton(x, m, grafo);
10
11       init = grafo.getInitial();
12       state = init;
13
14       for(j = 0; j < n; ++j){
15          if(grafo.getTarget(state, y.charAt(j)) != null){
16             ++ell;
17             state = grafo.getTarget(state, y.charAt(j));
18             if(j == n-1) {
19                temp[n]=ell;
20             }
21          }
22          else{
23             while((state != init) && (grafo.getTarget(state, y.charAt(j))
                    == null)){
24                state = grafo.getSuffixLink(state);
25                }
26                if(grafo.getTarget(state, y.charAt(j))!= null){
27
28                   temp[j] = ell;
29                   ell = grafo.getLength(state)+1;
30                   if(j == n-1) {
31                      temp[n]=ell;
32                   }
33                   state = grafo.getTarget(state, y.charAt(j));
34                }
35                else{
36                   temp[j] = ell;
37                   if(j == n-1) {
38                      temp[n]=ell;
39                   }
40                   ell = 0;
41                   state = init;
```

```
42              }
43           }
44         if(ell == m)
45            System.out.println("Posizione in cui compare un'occorrenza
                  intera del pattern: "+ (j-m+1)+ "\n");
46         }
47         for(int i = 0; i < n+1; ++i){
48            if(temp[i]!=0){
49               fact++;
50            System.out.println("Posizione del fattore del pattern nel
                  testo: "+(i-temp[i]));
51               System.out.println("Lunghezza del fattore: "+
                     temp[i]+"\n");
52            }
53         }
54         if(fact == 0 )
55            System.out.println("Non ci sono fattori del pattern nel
                  testo.");
56
57      }
```

Capitolo 5

Testing

Nel seguente capitolo mostreremo l'esecuzione dell'algoritmo attraverso una serie di test. Per ogni stringa del pattern e del testo abbiamo costruito una classe *Test*, nella quale illustriamo l'output del nostro algoritmo e il suffix automaton, del quale presentiamo anche una versione grafica.
Per prima cosa verranno stampati gli stati del suffix automaton; successivamente per ogni stato elenchiamo gli archi uscenti con la rispettiva etichetta e stato entrante. Infine diamo una lista dei suffix link per ogni vertice. Per rendere più chiara la rappresentazione dell'automa, i suffix link degli stati dell'automa non verranno inseriti in essa ma in una tabella.
L'output dell'algoritmo implementato, risulta diverso da quello dell'algoritmo originale, il quale restituisce soltanto l'indice del testo in cui compare l'occorrenza intera del pattern. Abbiamo modificato l'algoritmo in modo tale che vengano trovati tutti i fattori del pattern nel testo e ne stampiamo la posizione in cui esso inizia nel testo e la sua lunghezza.

5.1 Test 1

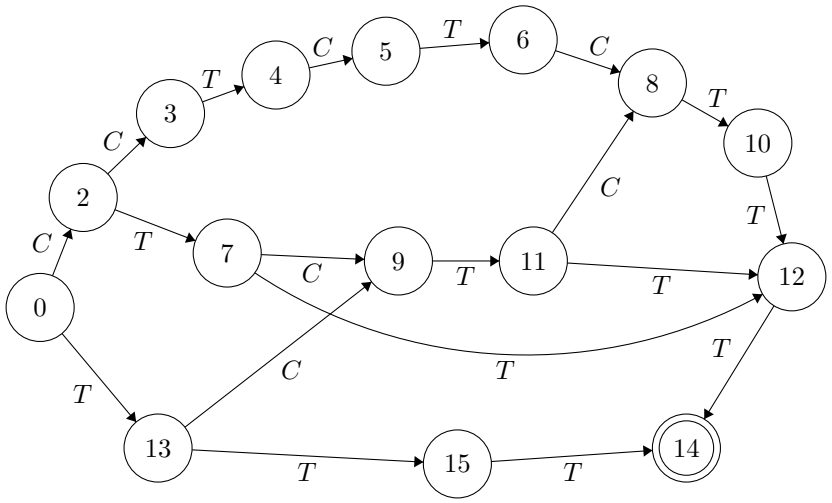
Mostriamo il primo esempio sul pattern *"CCTCTCTTT"*, del quale ricerchiamo i fattori nel testo *"AAAAAGGGGGGGGGGGGGGGAAAAA"*.

Vediamo che l'output dell'algoritmo è una stringa, la quale ci informa dell'assenza di fattori del pattern nel testo. Successivamente viene stampato l'automa, che disegniamo qui di seguito.

```
Stampa del testo: AAAAAGGGGGGGGGGGGGGGAAAAA
Lunghezza del testo: 25

Stampa del pattern: CCTCTCTTT
Lunghezza del pattern: 9

Non ci sono fattori del pattern nel testo.
```

```
Gli stati dell'automa sono:
[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ]

Stato 0, numero di archi uscenti = 2
Etichetta dell'arco = C, verso lo stato --> 2
Etichetta dell'arco = T, verso lo stato --> 13

Stato 1, numero di archi uscenti = 0

Stato 2, numero di archi uscenti = 2
Etichetta dell'arco = C, verso lo stato --> 3
Etichetta dell'arco = T, verso lo stato --> 7

Stato 3, numero di archi uscenti = 1
Etichetta dell'arco = T, verso lo stato --> 4

Stato 4, numero di archi uscenti = 1
Etichetta dell'arco = C, verso lo stato --> 5

Stato 5, numero di archi uscenti = 1
Etichetta dell'arco = T, verso lo stato --> 6

Stato 6, numero di archi uscenti = 1
Etichetta dell'arco = C, verso lo stato --> 8

Stato 7, numero di archi uscenti = 2
Etichetta dell'arco = C, verso lo stato --> 9
Etichetta dell'arco = T, verso lo stato --> 12

Stato 8, numero di archi uscenti = 1
Etichetta dell'arco = T, verso lo stato --> 10

Stato 9, numero di archi uscenti = 1
Etichetta dell'arco = T, verso lo stato --> 11

Stato 10, numero di archi uscenti = 1
Etichetta dell'arco = T, verso lo stato --> 12

Stato 11, numero di archi uscenti = 2
Etichetta dell'arco = C, verso lo stato --> 8
Etichetta dell'arco = T, verso lo stato --> 12

Stato 12, numero di archi uscenti = 1
Etichetta dell'arco = T, verso lo stato --> 14
```

```
Stato 13, numero di archi uscenti = 2
Etichetta dell'arco = C, verso lo stato --> 9
Etichetta dell'arco = T, verso lo stato --> 15

Stato 14, numero di archi uscenti = 0

Stato 15, numero di archi uscenti = 1
Etichetta dell'arco = T, verso lo stato --> 14

Il suffix link dello stato 0 è lo stato 1

Il suffix link dello stato 2 è lo stato 0

Il suffix link dello stato 3 è lo stato 2

Il suffix link dello stato 4 è lo stato 7

Il suffix link dello stato 5 è lo stato 9

Il suffix link dello stato 6 è lo stato 11

Il suffix link dello stato 7 è lo stato 13

Il suffix link dello stato 8 è lo stato 9

Il suffix link dello stato 9 è lo stato 2

Il suffix link dello stato 10 è lo stato 11

Il suffix link dello stato 11 è lo stato 7

Il suffix link dello stato 12 è lo stato 15

Il suffix link dello stato 13 è lo stato 0

Il suffix link dello stato 14 è lo stato 15

Il suffix link dello stato 15 è lo stato 13
```



state	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15
suffix link	1	0	2	7	9	11	13	9	2	11	7	15	0	15	13

5.2 Test 2

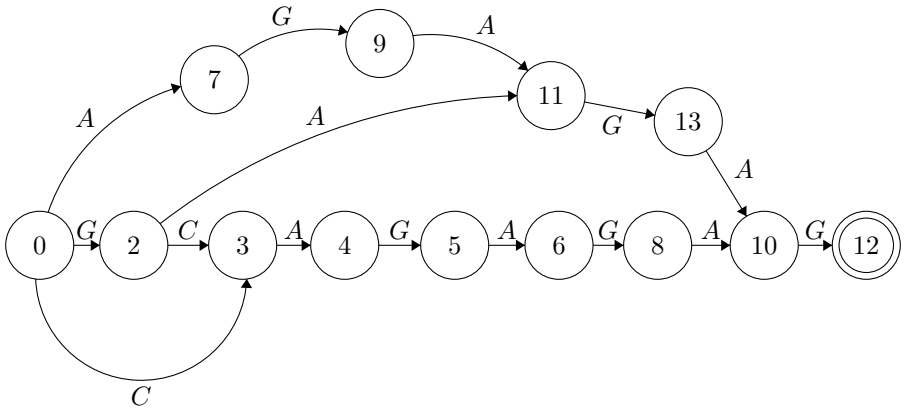
In questa classe presentiamo un esempio preso da [4]. Il pattern in esame è *"GCAGAGAG"* e la stringa di testo *"GCATCGCAGAGAGTATACAGTACG"*. Notiamo che nel testo vengono trovati tutti i fattori del pattern, anche quello che coincide con la lunghezza m del pattern, equivalente ad una sua intera occorrenza.

```
Stampa del testo: GCATCGCAGAGAGTATACAGTACG
Lunghezza del testo: 24

Stampa del pattern: GCAGAGAG
Lunghezza del pattern: 8

Posizione in cui compare un'occorrenza intera del pattern: 5

Posizione del fattore del pattern nel testo: 0
Lunghezza del fattore: 3

Posizione del fattore del pattern nel testo: 4
```

```
Lunghezza del fattore: 1

Posizione del fattore del pattern nel testo: 5
Lunghezza del fattore: 8

Posizione del fattore del pattern nel testo: 14
Lunghezza del fattore: 1

Posizione del fattore del pattern nel testo: 16
Lunghezza del fattore: 1

Posizione del fattore del pattern nel testo: 17
Lunghezza del fattore: 3

Posizione del fattore del pattern nel testo: 21
Lunghezza del fattore: 1

Posizione del fattore del pattern nel testo: 22
Lunghezza del fattore: 1

Posizione del fattore del pattern nel testo: 23
Lunghezza del fattore: 1


Gli stati dell'automa sono:
[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 ]

Stato 0, numero di archi uscenti = 3
Etichetta dell'arco = G, verso lo stato --> 2
Etichetta dell'arco = C, verso lo stato --> 3
Etichetta dell'arco = A, verso lo stato --> 7

Stato 1, numero di archi uscenti = 0

Stato 2, numero di archi uscenti = 2
Etichetta dell'arco = C, verso lo stato --> 3
Etichetta dell'arco = A, verso lo stato --> 11

Stato 3, numero di archi uscenti = 1
Etichetta dell'arco = A, verso lo stato --> 4

Stato 4, numero di archi uscenti = 1
Etichetta dell'arco = G, verso lo stato --> 5

Stato 5, numero di archi uscenti = 1
Etichetta dell'arco = A, verso lo stato --> 6
```

```
Stato 6, numero di archi uscenti = 1
Etichetta dell'arco = G, verso lo stato --> 8

Stato 7, numero di archi uscenti = 1
Etichetta dell'arco = G, verso lo stato --> 9

Stato 8, numero di archi uscenti = 1
Etichetta dell'arco = A, verso lo stato --> 10

Stato 9, numero di archi uscenti = 1
Etichetta dell'arco = A, verso lo stato --> 11

Stato 10, numero di archi uscenti = 1
Etichetta dell'arco = G, verso lo stato --> 12

Stato 11, numero di archi uscenti = 1
Etichetta dell'arco = G, verso lo stato --> 13

Stato 12, numero di archi uscenti = 0

Stato 13, numero di archi uscenti = 1
Etichetta dell'arco = A, verso lo stato --> 10

Il suffix link dello stato 0 è lo stato 1

Il suffix link dello stato 2 è lo stato 0

Il suffix link dello stato 3 è lo stato 0

Il suffix link dello stato 4 è lo stato 7

Il suffix link dello stato 5 è lo stato 9

Il suffix link dello stato 6 è lo stato 11

Il suffix link dello stato 7 è lo stato 0

Il suffix link dello stato 8 è lo stato 13

Il suffix link dello stato 9 è lo stato 2

Il suffix link dello stato 10 è lo stato 11

Il suffix link dello stato 11 è lo stato 7
```

```
Il suffix link dello stato 12 è lo stato 13

Il suffix link dello stato 13 è lo stato 9
```



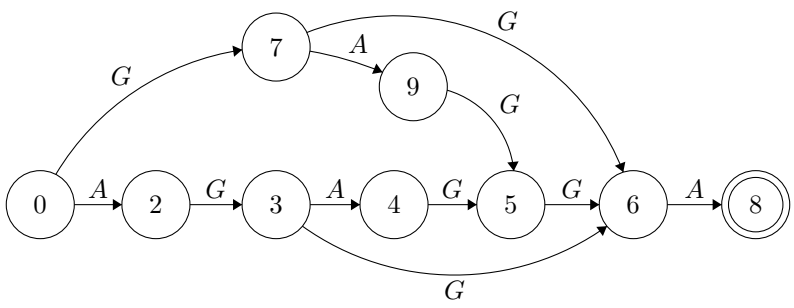
state	0	2	3	4	5	6	7	8	9	10	11	12	13
suffix link	1	0	0	7	9	11	0	13	2	11	7	13	9

5.3 Test 3

Nel terzo esempio abbiamo il testo t = *"TCCCAGAGGATCCT"* e il pattern p = *"AGAGGA"*. Nel testo abbiamo un solo match col pattern p, poiché non vi sono altri fattori di p che compaiono in t in posizioni esterne all'occorrenza intera di p.

```
Stampa del testo: TCCCAGAGGATCCT
Lunghezza del testo: 14

Stampa del pattern: AGAGGA
Lunghezza del pattern: 6

Posizione in cui compare un'occorrenza intera del pattern: 4

Posizione del fattore del pattern nel testo: 4
Lunghezza del fattore: 6


Gli stati dell'automa sono:
[ 0 1 2 3 4 5 6 7 8 9 ]

Stato 0, numero di archi uscenti = 2
Etichetta dell'arco = A, verso lo stato --> 2
Etichetta dell'arco = G, verso lo stato --> 7

Stato 1, numero di archi uscenti = 0

Stato 2, numero di archi uscenti = 1
Etichetta dell'arco = G, verso lo stato --> 3

Stato 3, numero di archi uscenti = 2
Etichetta dell'arco = A, verso lo stato --> 4
Etichetta dell'arco = G, verso lo stato --> 6

Stato 4, numero di archi uscenti = 1
Etichetta dell'arco = G, verso lo stato --> 5

Stato 5, numero di archi uscenti = 1
Etichetta dell'arco = G, verso lo stato --> 6

Stato 6, numero di archi uscenti = 1
Etichetta dell'arco = A, verso lo stato --> 8

Stato 7, numero di archi uscenti = 2
Etichetta dell'arco = G, verso lo stato --> 6
```

```
Etichetta dell'arco = A, verso lo stato --> 9

Stato 8, numero di archi uscenti = 0

Stato 9, numero di archi uscenti = 1
Etichetta dell'arco = G, verso lo stato --> 5

Il suffix link dello stato 0 è lo stato 1

Il suffix link dello stato 2 è lo stato 0

Il suffix link dello stato 3 è lo stato 7

Il suffix link dello stato 4 è lo stato 9

Il suffix link dello stato 5 è lo stato 3

Il suffix link dello stato 6 è lo stato 7

Il suffix link dello stato 7 è lo stato 0

Il suffix link dello stato 8 è lo stato 9

Il suffix link dello stato 9 è lo stato 2
```



state	0	2	3	4	5	6	7	8	9
suffix link	1	0	7	9	3	7	0	9	2

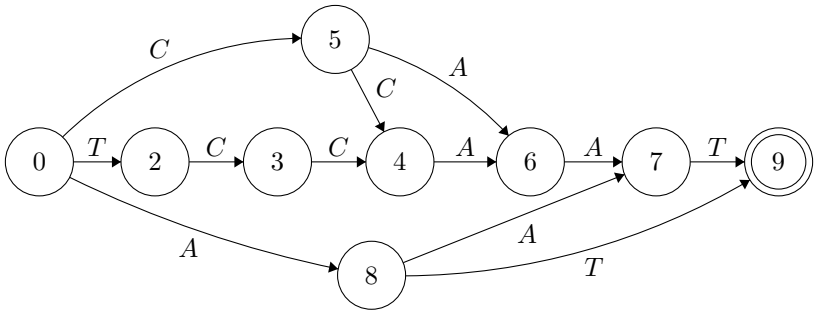
5.4 Test 4

L'ultima classe di test mostra l'esecuzione dell'algoritmo con il pattern *"TCCAAT"* e il testo *"CCTAGGTTGTAGGGCCAAC"*. Nel testo vengono trovati tutti i fattori possibili del pattern ma è assente quello con lunghezza uguale a quella del pattern.

```
Stampa del testo: CCTAGGTTGTAGGGCCAAC
Lunghezza del testo: 19

Stampa del pattern: TCCAAT
Lunghezza del patern: 6

Posizione del fattore del pattern nel testo: 0
Lunghezza del fattore: 2

Posizione del fattore del pattern nel testo: 2
Lunghezza del fattore: 1

Posizione del fattore del pattern nel testo: 3
Lunghezza del fattore: 1

Posizione del fattore del pattern nel testo: 6
Lunghezza del fattore: 1

Posizione del fattore del pattern nel testo: 7
Lunghezza del fattore: 1

Posizione del fattore del pattern nel testo: 9
Lunghezza del fattore: 1

Posizione del fattore del pattern nel testo: 10
Lunghezza del fattore: 1

Posizione del fattore del pattern nel testo: 14
Lunghezza del fattore: 4

Posizione del fattore del pattern nel testo: 18
Lunghezza del fattore: 1


Gli stati dell'automa sono:
[ 0 1 2 3 4 5 6 7 8 9 ]

Stato 0, numero di archi uscenti = 3
Etichetta dell'arco = T, verso lo stato --> 2
```

```
Etichetta dell'arco = C, verso lo stato --> 5
Etichetta dell'arco = A, verso lo stato --> 8

Stato 1, numero di archi uscenti = 0

Stato 2, numero di archi uscenti = 1
Etichetta dell'arco = C, verso lo stato --> 3

Stato 3, numero di archi uscenti = 1
Etichetta dell'arco = C, verso lo stato --> 4

Stato 4, numero di archi uscenti = 1
Etichetta dell'arco = A, verso lo stato --> 6

Stato 5, numero di archi uscenti = 2
Etichetta dell'arco = C, verso lo stato --> 4
Etichetta dell'arco = A, verso lo stato --> 6

Stato 6, numero di archi uscenti = 1
Etichetta dell'arco = A, verso lo stato --> 7

Stato 7, numero di archi uscenti = 1
Etichetta dell'arco = T, verso lo stato --> 9

Stato 8, numero di archi uscenti = 2
Etichetta dell'arco = A, verso lo stato --> 7
Etichetta dell'arco = T, verso lo stato --> 9

Stato 9, numero di archi uscenti = 0

Il suffix link dello stato 0 è lo stato 1

Il suffix link dello stato 2 è lo stato 0

Il suffix link dello stato 3 è lo stato 5

Il suffix link dello stato 4 è lo stato 5

Il suffix link dello stato 5 è lo stato 0

Il suffix link dello stato 6 è lo stato 8

Il suffix link dello stato 7 è lo stato 8

Il suffix link dello stato 8 è lo stato 0
```

`Il suffix link dello stato 9 è lo stato 2`



state	0	2	3	4	5	6	7	8	9
suffix link	1	0	5	5	0	8	8	0	2

Conclusioni

In questo lavoro di tesi si è affrontata, nell'ambito del pattern matching esatto tra stringhe, la ricerca di un pattern e dei suoi fattori in un testo, utilizzando i suffix automaton. Queste strutture, costrutite a partire dal pattern, ne determinano tutti i suoi suffissi.

Un impiego di tale algoritmo potrebbe essere quello di cercare le occorrenze di varie parole in un testo fissato. Il vantaggio di tale approccio è dovuto al fatto che, effettuare più ricerche di pattern differenti in uno stesso testo, può essere fatto in modo efficiente impiegando il DAWG costruito su ognuno dei pattern. Tale struttura risulta essere succinta rispetto al testo, le cui dimensioni potrebbero essere anche molto grandi. Perciò questo approccio è efficace, poiché la struttura che bisogna manipolare, il DAWG, ha una taglia lineare rispetto al pattern; pertanto le operazioni di costruzione e memorizzazione, hanno una complessità di tempo e spazio lineari.

L'algoritmo *Forward Dawg Matching* è stato implementato in *Java* a partire da codice *C* già sviluppato [4]. Sono state aggiunte nuove funzionalità, la più importante riguarda la restituzione delle posizioni nel testo, di tutti i fattori di un pattern con le rispettive lunghezze.
Sono stati effettuati vari test che hanno mostrato la costruzione del DAWG per ogni pattern e il funzionamento dell'algoritmo, mettendo in risalto le funzionalità che sono state inserite in questo lavoro di tesi.
Gli obiettivi posti all'inizio sono stati raggiunti e l'implementazione fornita ha lasciato inalterate le complessità dell'algoritmo.

Bibliografia

[1] Maxime Crochemore, Wojciech Rytter. *Text Algorithms.* Oxford University Press, Oxford, 1994.

[2] Maxime Crochemore, Christophe Hancart, Thierry Lecroq. *Algorithms on strings.* Cambridge University Press, 2007

[3] Michael T. Goodrich, Roberto Tamassia. *Data Structures and Algorithms in Java.* Wiley, 2007.

[4] Christian Charras, Thierry Lecroq. *Exact string matching algorithms.* Université de Rouen, Faculté des Sciences et des Techniques.
`http://www-igm.univ-mlv.fr/ lecroq/string/`

[5] $Java^{TM}$ Platform, Standard Edition 8 API Specification.

[6] Eclipse versione: Neon.2, release: 4.6.2. 2017.

Ringraziamenti

In primis desidero ringraziare la prof.ssa Rosalba Zizza, relatore di questa tesi, per essersi dimostrata sempre disponibile, gentile e per il costante supporto fornitomi durante la stesura della tesi.

Ringrazio i miei genitori, che mi hanno dato la possibilità di intraprendere questo percorso di studi e, in questo modo, di costruirmi un futuro; ma in particolare devo ringraziarli per avermi fatto il regalo più grande della mia vita: mio fratello. Se un giorno, quando sarai cresciuto, ti capiterà di leggere queste parole, sappi che ti ringrazio per il solo fatto di essere venuto al mondo, per avermi fatto capire cosa significa guardare una persona e rendersi conto di amarla in una maniera davvero inquantificabile. Sei tutto per me.

Si dice che gli amici siano la famiglia che si sceglie: io ho voi, i migliori, che mi supportano (e sopportano) da anni ormai. Amedeo, sei il mio migliore amico e un punto di riferimento, grazie per avermi fornito aiuto e consigli per qualsiasi cosa quando ne avevo bisogno. Carmela, sei dolcissima quando mi dimostri il tuo affetto con quei piccoli gesti improvvisi, mi hai abituato a farlo e lo adoro. Grazie perché sei presente, mi supporti, mi inciti, mi dai forza, ti voglio tantissimo bene! Federica, sei la piccolina del gruppo, la mascotte insomma. Spero di essere in grado di accompagnarti in questo tuo periodo di crescita dove sarò al tuo fianco. Antonio, la nostra amicizia si basa sul sarcasmo e su continui punzecchiamenti. Spero che tu sappia che ti voglio bene, in fondo. Sono felice di averti conosciuto. Veronica e Simona, non saprei da dove iniziare. Siete la cosa più sicura che ho nella mia vita. Avete fatto davvero tanto per me: mi avete aperto il cuore, mi avete insegnato a ricevere e dimostrare amore, a non aver paura di provare sentimenti ed esprimerli, come con un abbraccio. Un "vi voglio bene" sarebbe troppo poco, lo sapete.

Ringrazio i miei colleghi universitari, in particolare i miei compagni di corso: Mauro, Emanuele, Davide e Johnny, avete reso questi anni meno pesanti e divertenti, non cambiate mai ragazzi, vi adoro.
Sara, in questi ultimi anni sei stata la mia compagna di avventure (e disgrazie); sei riuscita anche tu a sciogliermi pian piano, con il tuo essere affettuosa e più espansiva. Grazie per i momenti di pazzia condivisi, di ansia, paure, per le risate, per tutto insomma. Ti voglio bene topina.

Ci auguro il meglio per il prossimo viaggio che stiamo per intraprendere, e spero vivamente insieme e anche con i ragazzi, così da rendere i prossimi anni altrettanto indimenticabili.

Infine merito qualche ringraziamento anche io. Ce l'hai fatta, sei riuscita a non mollare nell'anno più distruttivo della tua vita. Sei l'unica a sapere cosa hai attraversato in questo periodo, ti auguro di riuscire a trovare un po' di pace e sopratutto di riuscire a esternare di più ciò che senti, ciò che hai dentro. Intanto, goditi questo momento!