Dispense per

Algoritmi per Internet e Web: Ricerca e Indicizzazione dei Testi

Corsi di Studio in Informatica

Università degli Studi di Pisa

Docente: *Roberto Grossi*

Dipartimento di Informatica

grossi@di.unipi.it

Anno accademico 2007-2008

versione 2.2.bis

Indice

CAPITOLO 1

Introduzione

Le stringhe, le sequenze e i testi elettronici rappresentano il formato più diffuso di informazione digitale insieme ai dati multimediali (immagini e suoni). Siamo letteralmente sommersi da questi dati testuali. Prima, l'obiettivo dei sistemi di recupero delle informazioni era quello di organizzare i dati in modo da convogliare *tutti* i possibili documenti utili verso l'utente. Adesso, occorre invece limitare, filtrare e classificare tali informazioni per fornirne solo una *parte significativa* all'utente. La limitazione di banda al flusso di informazioni è infatti dovuta alla velocità di lettura dell'utente stesso: non sorprende affatto sapere che non basta la propria vita per leggere tutte le informazioni, di proprio interesse, che sono reperibili attraverso Internet e il World Wide Web.

Immaginiamo un frequente atto quotidiano come quello, per esempio, di scrivere una porzione di testo al calcolatore mediante un "editore" di testi, integrato o meno nell'applicazione che stiamo usando. A un certo punto, vogliamo cercare un paragrafo che contiene una frase o una parola chiave: l'editore di testi offre delle operazioni di ricerca più o meno raffinate che ci consentono di individuare velocemente il paragrafo a cui siamo interessati. Queste operazioni sono spesso disponibili in librerie standard di software; per esempio, la funzione `strstr()` di ricerca di una stringa è contenuta nella libreria standard `<string.h>` per il linguaggio di programmazione C. In mancanza di tali librerie, non è diffile progettare un algoritmo immediato che sia in grado di cercare una sequenza di caratteri all'interno di un testo.

Questo scenario intuitivo ha motivato una serie di approfonditi studi che hanno prodotto tecniche algoritmiche e strutture di dati sofisticate, in parte descritte nei capitoli seguenti. Il lettore può giustamente chiedersi perché semplici algoritmi di ricerca non possano essere efficaci, per esempio, con i testi disponibili in Internet e nel Web. Il problema nasce dalla *grande quantità* di informazioni testuali distribuite geograficamente e disponibili in formato elettronico. Esistono delle statistiche che affermano che questa quantità raddoppia regolarmente dopo alcuni mesi. Svariate decine di migliaia di calcolatori, destinati all'elaborazione di milioni di ricerche su enormi collezioni di testi, trattano ordini di grandezza decisamente superiori a quelli trattati dalla ricerca all'interno di un editore di testi. Rendere le operazioni di ricerca dieci volte più veloci, per esempio, permette di gestire decine di milioni di richieste (assumendo una completa scalabilità delle prestazioni). Probabilmente un tale miglioramento non produce una sensibile differenza nell'editore di testi. Invece, le tecniche impiegate per gestire grandi quantità di testi devono essere necessariamente più evolute.

A rendere caotica la situazione contribuisce il fatto che spesso i dati testuali sono eterogenei, dovuti ai diversi modelli di riferimento. Per esempio, lo *spelling* di alcune parole inglesi e americane è diverso; certi nomi stranieri vengono spesso traslitterati in maniera differente. I testi datati sono spesso riportati in versione elettronica usando termini ormai decaduti. Possono esistere più versioni modificate dello stesso testo. In generale il livello di omogeneità dei dati è basso in quanto non è sempre possibile controllare la qualità editoriale dei documenti resi pubblici in formato elettronico.

In sostanza, esistono documenti interessanti che rischiano di non poter essere trovati altrimenti. Cercare tramite una semplice parola chiave non è più sufficiente come nel caso del nostro editore di testi. In questo scenario, abbiamo bisogno di agire a tre livelli.

(1) Potenziare gli strumenti di ricerca ammettendo metacaratteri, espressioni regolari e un certo grado di imprecisione, così da renderli più selettivi e mirati (cfr. Parte I: Algoritmi di ricerca).
(2) Usare e progettare indici testuali più potenti che supportino ricerche veloci in grandi collezioni di testi, senza il bisogno di esaminare interamente i testi a ogni richiesta di ricerca (cfr. Parte II: Strutture di dati per testi), presentando i testi individuati secondo un qualche ordine di rilevanza che sia efficacemente realizzabile tramite il calcolatore
(3) Sfruttare le caratteristiche proprie delle applicazioni per specializzare le tecniche a particolari problemi su certi insiemi di testi (cfr. Parte III: Applicazioni).

Tali possibilità vengono esplorate e descritte rispettivamente nelle tre parti che compongono queste dispense, con enfasi sulle strutture di dati e gli algoritmi adottati e sulle loro connessioni con le attuali tecnologie, contenendo al minimo i tecnicismi tipici dell'area. Vengono descritti gli argomenti più utili didatticamente, discutendo i principi che sono dietro ad alcuni sistemi software come *Glimpse* e *Google*.

Gli argomenti trattati non sono esaustivi; per esempio, non vengono discussi l'estrazione di semantica dai testi, la loro catalogazione automatica o la loro compressione. Certamente la ricerca *sintattica* (specificando le parole che si intendono cercare) va trasformandosi in ricerca *per contenuti* (specificando le frasi caratterizzanti) con lo scopo di arrivare in un prossimo futuro alla ricerca *semantica* (specificando i concetti).

I metodi descritti per la ricerca sintattica sono comunque il punto di partenza per strumenti più sofisticati: i campi di applicazione delle nozioni introdotte sono vari, come brevemente esemplificato di seguito.

- Bioinformatica: l'informazione sottostante la biochimica, la biologia molecolare e lo sviluppo degli organismi può essere rappresentata da semplici sequenze di lettere {A, C, G, T} (le basi azotate) chiamate sequenze di DNA. Rappresentano la struttura informativa alla base della biologia degli organismi, per cui la biologia computazionale cerca di ridurre fenomeni biologici complessi a interazioni tra sequenze di DNA.
- Compressione dei dati: un testo può essere rappresentato e memorizzato in un formato compatto che può richiedere un numero di bit inferiore a quello richiesto dal testo stesso. In diversi casi, è possibile avvicinarsi al minimo teorico di bit di rappresentazione, denominato entropia del testo (cfr. anche la complessità di Kolmogorov).
- Estrazione di contenuti (*pattern discovery* e *data mining*): le sequenze contengono spesso delle dipendenze funzionali o regole di associazione tra i loro simboli che vengono esplicitate attravero la presenza di ripetizioni o motivi ricorrenti (*pattern* o *motif*). La scoperta di tali pattern nei testi permette di mettere in luce potenziali dipendenze finora ignote.

- Recupero veloce di testi digitali (*information retrieval*): i documenti vengono propriamente elaborati e archiviati per permetterne la ricerca e l'identificazione attraverso l'uso di parole chiave.

Altre applicazioni che si avvalgono delle tecniche descritte in queste dispense riguardano programmi di utilità come `grep` nei sistemi operativi Linux e Unix; banche dati come BLAST, Medline, LexisNexis; cataloghi librari elettronici; motori di ricerca in Internet; agenzie stampa; newsgroup di Internet; riviste e libri pubblicati in forma elettronica; collezioni di lavori scientifici e database bibliografici; elenchi telefonici disponibili in rete; enciclopedie elettroniche e altre applicazioni didattiche; vocabolari elettronici con riferimenti incrociati; riconoscimento di sequenze speciali nelle telecomunicazioni; riconoscimento di siti marcatori nelle sequenze biologiche; plagio di documenti.

1.1. Un semplice caso di studio

Ipotizziamo di voler vedere se `ananas` appare nel nome esotico `bananananassata` di una bibita immaginaria. Nel gergo tecnico degli informatici, `ananas` e `bananananassata` sono due sequenze di caratteri o *stringhe* chiamate rispettivamente *pattern* e *testo*. Vogliamo stabilire se il pattern occorre nel testo, un problema classico del cosiddetto *string matching*. Possiamo immaginare il pattern e il testo come due sequenze di caselle, in cui ogni casella può contenere un possibile *carattere*, inclusi spazi, segni di punteggiatura, ecc. Questi caratteri sono scelti da un insieme prefissato di simboli, l'*alfabeto* Σ.

Possiamo riferirci al carattere contenuto nella prima casella delle stringhe indicandolo come primo carattere o carattere in posizione 1. Enumerando le caselle del pattern e del testo a partire da 1, il carattere i-esimo delle stringhe è il carattere contenuto nella casella i-esima. Nel nostro esempio, il terzo carattere del pattern è `a` mentre quello del testo è `n`. Il numero di caselle, quindi di caratteri, è chiamato *lunghezza* della stringa e viene indicato aggiungendo due barre verticali: $|\texttt{ananas}| = 6$ caratteri e $|\texttt{bananananassata}| = 13$ caratteri. Una identica notazione vale per $|\Sigma|$ così da indicare il numero di simboli presenti nell'alfabeto delle stringhe. Nel resto delle dispense, indicheremo con σ la dimensione $|\Sigma|$, per cui $\Sigma = \{1, 2, \ldots, \sigma\}$.

Esempi tipici di alfabeti Σ sono quelli binari, quelli composti dalle maiuscole, oppure realisticamente quelli derivanti dagli standard adottati per codificare i file di testo nei calcolatori come ASCII (8 bit per carattere) e Unicode (16 o più bit per carattere codificati a lunghezza variabile con UTF8). La tabella riassume le loro caratteristiche.

Σ	σ	uso
$\{0, 1\}$	2	sequenze binarie
$\{A, \ldots, Z\}$	26	parole maiuscole
ASCII	256	sequenze alfanumeriche
Unicode	$\geqslant 65536$	sequenze alfanumeriche

Ipotizziamo l'esistenza di un simbolo speciale $\sqcup$ non appartenente all'alfabeto: $\sqcup \notin \Sigma$. Tale simbolo è sempre minore di qualunque simbolo in Σ e viene usato come terminatore di stringa (si pensi a '\0' nel linguaggio C). Ricordiamo che, senza perdita di generalità, i simboli dell'alfabeto Σ vengano codificati all'interno del calcolatore come numeri da 1 a σ. Dati due caratteri $a, b \in \Sigma$, se $a = b$ diciamo che abbiamo un *match*; se $a \neq b$, abbiamo un *mismatch*.

Il problema di cercare una parola P in un testo elettronico T usa la loro *rappresentazione* come stringhe su un alfabeto Σ, per cui denotiamo con $P[i]$ il carattere i-esimo del pattern P, per $1 \leqslant i \leqslant |P|$, e con $T[i]$ il carattere i-esimo del testo T, per $1 \leqslant i \leqslant |T|$. Continuando il nostro esempio, $P[3] = \texttt{a}$ e $T[3] = \texttt{n}$. Ma non vogliamo fermarci ai singoli caratteri. Per indicare

segmenti delle stringhe, chiamate *sottostringhe*, introduciamo la notazione $\mathsf{T}[\mathtt{i}..\mathtt{j}]$ per denotare la sequenza di caratteri $\mathsf{T}[\mathtt{i}]\mathsf{T}[\mathtt{i}+1]\cdots\mathsf{T}[\mathtt{j}]$ dove $\mathtt{i} \leqslant \mathtt{j}$. Per esempio, $\mathsf{T}[7..9] = \mathtt{nas}$. Usiamo la coppia di interi $(\mathtt{i},\mathtt{j})$ per rappresentare la sottostringa $\mathsf{T}[\mathtt{i}..\mathtt{j}]$ con un numero costante di celle di memoria, indipendentemente dalla sua lunghezza $\mathtt{j}-\mathtt{i}+1 = \big|\mathsf{T}[\mathtt{i}..\mathtt{j}]\big|$. (La lunghezza di una sottostringa è chiaramente il numero di caratteri in essa contenuta.) Se $\mathtt{j} < \mathtt{i}$, allora otteniamo una stringa speciale, la stringa *vuota*, denotata da ϵ per indicare che non contiene alcun carattere. Una notazione analoga vale per il pattern e per ogni altra stringa che tratteremo.

Le sottostringhe hanno un nome speciale quando includono la prima o l'ultima posizione della stringa. Per $1 \leqslant \mathtt{i} \leqslant |\mathsf{T}|$, la sottostringa $\mathsf{T}[1..\mathtt{i}]$ si chiama *prefisso* $\mathtt{i}$-esimo; la sottostringa $\mathsf{T}\big[\mathtt{i}..|\mathsf{T}|\big]$ si chiama *suffisso* $\mathtt{i}$-esimo. Per esempio, il terzo prefisso di T è $\mathtt{ban}$, mentre il suo terzo suffisso è $\mathtt{nananassata}$.

Ritornando al problema di cercare il pattern $\mathsf{P} = \mathtt{ananas}$ nel testo $\mathsf{T} = \mathtt{bananananassata}$, possiamo formularlo con la nostra terminologia. Vogliamo identificare le sottostringhe del testo che sono uguali al pattern. In altre parole, vogliamo individuare le posizioni $\mathtt{j}$ nel testo per cui $\mathsf{T}\big[\mathtt{j}..\mathtt{j}+|\mathsf{P}|-1\big] = \mathsf{P}$. Nel nostro caso, poiché $\mathsf{T}[4..9] = \mathtt{ananas}$, abbiamo una occorrenza in posizione $\mathtt{j} = 4$ del testo. La lunghezza nota del pattern permette di identificare il resto dei caratteri dell'occorrenza nel testo. Notiamo dall'esempio che ogni occorrenza del pattern è una sottostringa del testo. In generale, siamo interessati alla risposta di tre tipi di richieste:

- Decisionale — Il pattern P occorre nel testo T?
- Quantitativa — Quante volte P occorre in T?
- Enumerativa — In quali posizioni di T occorre P?

Dopo aver eseguito la ricerca quantitativa, siamo in grado di rispondere anche alla richiesta decisionale; basta controllare se il numero di occorrenze è maggiore di zero o no. Mentre risolviamo la richiesta enumerativa possiamo contare le occorrenze, per cui siamo in grado di rispondere alla richiesta quantitativa. In tali richieste, il pattern P viene fornito in anticipo o simultaneamente al testo T. Studieremo anche il caso in cui il testo è fornito in anticipo per essere elaborato. Rimandiamo il lettore alla Parte 2 per la discussione di questo caso.

Per rispondere alle richieste decisionali, quantitative ed enumerative possiamo usare la seguente idea generale. Immaginiamo di mantenere una *finestra* di $|\mathsf{P}|$ caselle che facciamo scorrere sul testo. Allineiamo la finestra su tutte le posizioni $\mathtt{j}$ possibili del testo ($1 \leqslant \mathtt{j} \leqslant |\mathsf{T}|$). Ad ogni istante, la finestra mostra esattamente $|\mathsf{P}|$ caratteri contigui del testo, ossia quelli in $\mathsf{T}\big[\mathtt{j}..\mathtt{j}+|\mathsf{P}|-1\big]$, su cui procedere mediante confronti. Esistono $|\mathsf{T}|-|\mathsf{P}|+1$ allineamenti di finestre possibili.

Per ciascun allineamento, confrontiamo i caratteri della finestra con gli omologhi caratteri del pattern: il primo carattere visibile nella finestra confrontato con il primo carattere del pattern e così via.

- Se troviamo un mismatch tra una coppia di caratteri a un certo punto, il pattern non può apparire in posizione $\mathtt{j}$.
- Se invece i confronti danno tutti esito positivo (tutti match), possiamo dichiarare di aver trovato un'occorrenza in posizione $\mathtt{j}$. In tal caso rispondiamo vero se la richiesta è di tipo decisionale; incrementiamo un contatore di occorrenze nel caso di richiesta quantitativa; stampiamo la posizione $\mathtt{j}$ quando la richiesta è enumerativa.

Mostreremo tecniche più sofisticate per rendere questo approccio efficiente e per ammettere pattern flessibili e con metacaratteri nella Parte 1. Applicheremo le nozioni acquisite all'indicizzazione di testi. Concludiamo questo capitolo dando i prerequisiti e le nozioni preliminari necessarie alla comprensione del materiale che discuteremo nei prossimi capitoli.

1.2. Prerequisiti e nozioni preliminari

Al fine di facilitare la comprensione delle dispense, il lettore dovrebbe avere conoscenze di base nella programmazione dei calcolatori e nell'analisi e nel progetto degli algoritmi e delle strutture di dati elementari.

In particolare, adottiamo lo pseudocodice standard per descrivere i nostri algoritmi, preferendo la chiarezza didattica alla ingegnerizzazione del codice. I valori primitivi usati come costanti nei nostri programmi sono i caratteri e le stringhe (ovvio!), i valori booleani TRUE e FALSE e gli interi. Possiamo usare le variabili, le operazioni aritmetiche standard e i connettivi logici AND e OR *condizionali*. Ovvero, se la condizione non è soddisfatta dal primo termine, non valutiamo il secondo termine. Per esempio, (FALSE AND $a = c$) fa sì che non valutiamo mai $a = c$ in quanto il risultato è comunque falso. Permettiamo l'uso di procedure e di funzioni realizzate mediante i seguenti costrutti di programmazione mescolati a istruzioni in lingua italiana:

- assegnamento: $a \leftarrow b$;
- condizionale: IF $a \leqslant b$ THEN $a \leftarrow b$ ELSE $b \leftarrow c$;
- ciclo while: WHILE $a \leqslant b$ AND $a > c$ DO $a \leftarrow a + 1$;
- ciclo for: FOR $i \leftarrow 1$ TO n DO $a \leftarrow a * i$;
- ciclo foreach sugli elementi x di un insieme S: FOREACH $x \in S$ DO stampa x;
- interruzione del codice e restituzione di un valore: RETURN a.

Il ciclo for prevede l'incremento automatico della variabile i alla fine di ogni iterazione; il ciclo foreach è più generale e prevede che la variabile x assuma tutti i valori in S secondo un ordine irrilevante.

Per quanto riguarda le strutture di dati, usiamo quelle elementari: vettori, array e tabelle (multidimensionali o meno); code e pile; liste con puntatori o riferimenti; alberi radicati e ordinati; grafi con visite in profondità (*DFS*) e ampiezza (*BFS*); tabelle hash. Per gli algoritmi assumiamo la conoscenza della ricerca binaria e degli algoritmi di ordinamento di base come mergesort, quicksort e radix sort. Per valutare la complessità in spazio e in tempo, adottiamo il modello classico RAM a costo uniforme, ispirato al calcolatore di von Neumann, e le nozioni di complessità al caso medio e al caso pessimo e di limite inferiore. Usiamo la notazione asintotica $O(), \Omega(), \Theta()$ per esprimere la complessità degli algoritmi.

Infine, un motivo dominante della Parte 1 delle dispense è quello degli automi, di cui ricordiamo brevemente la definizione. Denotiamo con ϵ sia la stringa vuota che il carattere vuoto (non crea ambiguità nel nostro caso; altrimenti usiamo λ per il carattere vuoto). Un automa è una macchina con un nastro di sola lettura che può trovarsi in un numero finito di stati. È guidato nella transizione da uno stato all'altro dalla lettura di caratteri consecutivi del nastro e da un controllo finito. Formalmente, un automa finito è definito da una quintupla $\langle Q, \Sigma, \delta, q_0, F \rangle$, dove

- Q è l'insieme finito di stati,
- Σ è l'alfabeto dei simboli del nastro,
- $q_0 \in Q$ è lo stato iniziale,
- $F \subseteq Q$ è l'insieme degli stati finali (che fanno accettare l'automa),
- δ è la funzione parziale delle transizioni da stato in stato.

In particolare, δ è definita sulle coppie stato-carattere $Q \times (\Sigma \cup \{\epsilon\})$ e restituisce sottoinsiemi (possibilmente vuoti) di Q. In altre parole, $\delta(q, c)$ descrive il prossimo stato (o i prossimi stati) raggiungibili a partire dallo stato q leggendo il carattere c dal nastro. Se nessuno stato esiste a partire da q dopo aver letto c, indichiamo tale fallimento con $\delta(q, c) = \text{NIL}$. Quando

$\delta(q, c) \neq$ NIL è sempre composto da un singolo stato, allora l'automa viene indicato come *deterministico*. Altrimenti, esistono transizioni multiple per q e c, e l'automa viene chiamato *non deterministico*.

Un automa inizia la computazione nella stato q_0 con la testina posizionata sul simbolo non vuoto più a sinistra nel nastro. Al passo generico, l'automa si trova in uno stato q, legge un simbolo c dal nastro e sposta la testina in avanti di una posizione, andando nello stato o negli stati indicati da $\delta(q, c)$. Se uno di questi stati è in F, allora l'automa accetta la sequenza di caratteri letta fino a quel punto. Altrimenti, rifiuta se $\delta(q, c) =$ NIL oppure se si trova in uno stato non appartenente a F e non ci sono più simboli da leggere sul nastro. Secondo un teorema di Kleene, il linguaggio riconosciuto dagli automi (deterministici o non), ovvero l'insieme di tutte le stringhe accettate, coincide con le espressioni regolari che studieremo nel Capitolo 3. Nel seguito, identificheremo un automa con la sua funzione di transizione δ quando ciò non genererà confusione, per cui parleremo di automa δ.

Parte 1

ALGORITMI DI RICERCA
(ossia con scansione del testo)

Questa parte descrive gli algoritmi di ricerca testuali, in cui i testi vengono *interamente* esaminati a ogni richiesta. Seppure il loro utilizzo immediato su grandi collezioni di testi non è auspicabile perché la scansione di tali testi richiede un tempo di esecuzione elevato, le tecniche algoritmiche ivi descritte sono comunque utili per gli indici testuali: questi ultimi evitano la scansione completa del testo a ogni ricerca ed esistono indici che richiedono soltanto la scansione di piccole porzioni del testo anche se le collezioni sono vaste. Il valore didattico di tali algoritmi è pertanto elevato. Nel Capitolo 2, vengono descritti alcuni algoritmi di ricerca esatta di stringhe e sequenze, introducendo nozioni e strumenti che saranno utili negli altri capitoli. Nel Capitolo 3, vengono descritte le espressioni regolari e la loro potenza descrittiva nelle ricerche testuali. Il Capitolo 4 descrive la nozione di errore e presenta alcuni algoritmi di ricerca in cui la sequenza di caratteri specificata e le sue occorrenze possono essere soggette a errori e imprecisioni. Infine, nel Capitolo 5, viene descritta una modalità flessibile di ricerca che permette di combinare le tecniche di ricerca esatta con le espressioni regolari, specificando le porzioni della sequenza di ricerca in cui ammettere gli errori.

CAPITOLO 2

Ricerca Esatta di Stringhe e Sequenze

In questo capitolo prendiamo in esame vari metodi per la risoluzione efficiente del problema della ricerca esatta, in cui data una stringa *pattern* P di lunghezza m e una stringa *testo* T di lunghezza n, vogliamo stabilire se P occorre in T. Entrambe le stringhe sono definite sull'alfabeto Σ. Consideriamo la versione decisionale del problema in cui vogliamo sapere se esiste una posizione j del testo, $1 \leqslant j \leqslant n-m+1$, tale che la sottostringa $\mathsf{T}[j..j+m-1]$ è uguale al pattern P. Non è difficile estendere le tecniche di risoluzione alla versione quantitativa e alla versione enumerativa del problema in cui vogliamo conoscere, rispettivamente, quante e quali sono le posizioni j dove il pattern P occorre. L'Introduzione riporta la terminologia e la notazione necessaria alla comprensione di questo capitolo e dei successivi, e fornisce delle motivazioni per lo studio dei problemi presentati.

Partendo da un algoritmo immediato per il problema della ricerca esatta, mostriamo come ottenere un algoritmo veloce.

- *Preprocessing*(P): Prima eseguiamo un'elaborazione preliminare sul pattern usando tabelle opportune e automi deterministici.
- *Scanning*(T): Poi sfruttiamo l'elaborazione preliminare per rendere più veloce la scansione del testo alla ricerca di occorrenze del pattern.

Durante la scansione, manteniamo *implicitamente* una finestra di m caselle da far scorrere sul testo. Allineiamo l'inizio della finestra alle posizioni $1 \leqslant j \leqslant n - m + 1$ del testo. A ogni istante, la finestra mostra gli m caratteri nel sottostante segmento $\mathsf{T}[j..j + m - 1]$ del testo. La finestra è solo concettuale e serve a delimitare la porzione di testo da confrontare con il pattern. Spostare la finestra richiede perciò tempo costante: basta semplicemente fissare il nuovo valore di j.

Fissata una posizione j, confrontiamo dei caratteri della finestra con gli omologhi caratteri del pattern, ovvero il k-esimo carattere della finestra viene confrontato con il k-esimo carattere del pattern. Se troviamo un *mismatch* tra una coppia di caratteri a un certo punto, il pattern non può apparire in posizione j. In tal caso, dobbiamo stabilire la prossima posizione da esaminare ($j + 1$ o una successiva). Se invece i confronti danno tutti esito positivo (*match*), possiamo dichiarare di aver trovato un'occorrenza in posizione j. In tal caso rispondiamo vero se la ricerca è di tipo decisionale; incrementiano un contatore di occorrenze nel caso di ricerca di tipo quantitativo; infine, stampiamo la posizione j quando la ricerca è di tipo enumerativo. Alla fine del capitolo, mostreremo come estendere uno degli algoritmi efficienti al caso della ricerca simultanea di più pattern su uno stesso testo.

2.1. Algoritmo immediato

Nel caso dell'algoritmo immediato, denominato RICERCA◇INGENUA, esaminiamo tutte le posizioni $\mathtt{j}$ della finestra scorrevole: per ciascun posizionamento della finestra, confrontiamo tutti i caratteri a partire da quello più a sinistra, fino a raggiungere la fine della finestra con esito positivo oppure a interrompere i confronti perché abbiamo riscontrato un mismatch.

Esempio. Vediamo il comportamento dell'algoritmo con $\mathtt{P} = \mathtt{ananas}$ e $\mathtt{T} = \mathtt{bananananassata}$, dove $\mathtt{m} = 6$ e $\mathtt{n} = 13$. Iniziando con $\mathtt{j} = 1$, confrontiamo solo il primo carattere $\mathtt{b}$ della finestra che dà luogo a un mismatch. Proseguiamo con $\mathtt{j} = 2$, confrontando tutti i caratteri $\mathtt{ananan}$ della finestra corrente e scoprendo solo alla fine un mismatch. Per $\mathtt{j} = 3$, abbiamo un mismatch con il primo carattere $\mathtt{n}$ della finestra corrente. Infine, con $\mathtt{j} = 4$, abbiamo tutti match, per cui abbiamo trovato un'occorrenza del pattern $\mathtt{P}$.

Pseudocodice. Usiamo una variabile $\mathtt{j}$ che assume i valori $1, 2, \ldots, \mathtt{n} - \mathtt{m} + 1$ in ordine crescente per indicare tutte le posizione possibili con cui allineare la finestra scorrevole. Introduciamo una variabile $\mathtt{k}$ di appoggio che assume valori crescenti $0, 1, \ldots, \mathtt{m} - 1$ per scorrere gli $\mathtt{m}$ caratteri all'interno della finestra e del pattern, fintanto che troviamo dei match. La computazione è regolata da due cicli annidati. Il ciclo for esterno si occupa di far scorrere la finestra incrementando il valore di $\mathtt{j}$. Il ciclo while interno ha il compito di confrontare i caratteri della finestra $\mathtt{T}[\mathtt{j}..\mathtt{j} + \mathtt{m} - 1]$ con quelli del pattern $\mathtt{P} \equiv \mathtt{P}[1..\mathtt{m}]$; al passo $\mathtt{k}$ confronta $\mathtt{P}[1 + \mathtt{k}]$ con $\mathtt{T}[\mathtt{j} + \mathtt{k}]$ e incrementa $\mathtt{k}$ in caso di successo. Se $\mathtt{k}$ diventa uguale a $\mathtt{m}$, vuol dire che abbiamo confrontato tutti i caratteri con esito positivo. Altrimenti, c'è un mismatch che interrompe il ciclo while.

RICERCA◇INGENUA($\mathtt{P}, \mathtt{T}$):

```
1: m ← |P|; n ← |T|;
2: FOR j ← 1 TO n − m + 1 DO
3:    k ← 0;
4:    WHILE k < m AND P[1 + k] = T[j + k] DO
5:       k ← k + 1;
6:    IF k = m THEN
7:       RETURN TRUE;    {occorrenza T[j..j + m − 1] = P}
8: RETURN FALSE;
```

Per restituire tutte le occorrenze, o contarle, basta sostituire la linea 7 con la stampa del valore di $\mathtt{j}$ oppure con l'incremento di un contatore inizializzato a zero.

Complessità. L'algoritmo occupa spazio costante in aggiunta a quello comunque richiesto dal pattern e dal testo. Tuttavia, può richiedere $O(\mathtt{mn})$ passi al caso pessimo. Una tale situazione si configura scegliendo, in maniera artificiale, il testo $\mathtt{T} = \mathtt{a}\cdots\mathtt{aaa}$ ottenuto come sequenza ripetuta della stessa lettera $\mathtt{a}$ e il pattern $\mathtt{P} = \mathtt{a}\cdots\mathtt{aab}$ ottenuto come sequenza ripetuta della stessa lettera $\mathtt{a}$ tranne che nell'ultima posizione, dove troviamo una lettera diversa $\mathtt{b}$. In tal caso, il corpo del ciclo for (linee 3–7) viene eseguito tutte le $\mathtt{n} - \mathtt{m} + 1$ volte e, ogni volta, il ciclo while interno (linee 4–5) richiede $O(\mathtt{m})$ tempo perché si accorge del mismatch $\mathtt{a} \neq \mathtt{b}$ solo quando $\mathtt{k} = \mathtt{m} - 1$. Fissando un ordine diverso per confrontare i caratteri nella finestra, invece che da sinistra a destra, non migliora la situazione del caso pessimo. Basta scegliere il pattern $\mathtt{P}$ in modo che l'ultimo carattere da confrontare nell'ordine stabilito sia comunque $\mathtt{b}$. Ne deriva che il numero totale di confronti di caratteri è $O(\mathtt{mn})$. Nelle sequenze di DNA, i valori tipici di $\mathtt{m}$ sono dell'ordine delle centinaia di caratteri, per cui il costo di RICERCA◇INGENUA risulta eccessivo.

Eseguendo RICERCA◇INGENUA su testi reali, tuttavia, il comportamento medio si discosta da quanto previsto sopra per il caso pessimo. In pratica, l'algoritmo è una soluzione ragionevole

quando la prima occorrenza del pattern ha una buona probabilità di trovarsi all'inizio del testo oppure quando si cercano pattern di piccola dimensione (meno di una decina di caratteri). Eseguendo tale algoritmo su testi generati casualmente, il comportamento al caso pessimo di $O(mn)$ confronti è decisamente raro ed emerge con stringhe veramente particolari come quelle appena discusse sopra.

Un'analisi più approfondita su testi generati casualmente permette di chiarire tale comportamento di RICERCA⋄INGENUA. Poiché il costo dell'algoritmo è dominato dal numero dei confronti nel ciclo while interno (linee 4–5), mostriamo che RICERCA⋄INGENUA esegue un totale di circa

$$\frac{\sigma}{\sigma-1}\left(1-\frac{1}{\sigma^m}\right)\cdot(n-m+1)\leqslant 2(n-m+1)=O(n)$$

confronti nel caso medio, dove $\sigma \geqslant 2$ è la cardinalità dell'alfabeto Σ. Tale risultato vale nel modello di distribuzione uniforme dei caratteri, dove il testo e il pattern sono ottenuti scegliendo i caratteri di Σ con uguale probabilità in maniera indipendente e uniforme. In altre parole, la probabilità che un carattere c compaia in una data posizione è $1/\sigma$. Secondo tale modello di distribuzione, $1/\sigma$ è anche la probabilità di match tra un carattere del pattern P e uno del testo T, in quanto c'è una possibilità su σ che entrambi siano uguali allo stesso simbolo $c \in \Sigma$ (infatti, $\sum_{c\in\Sigma} 1/\sigma^2 = 1/\sigma$). Altrimenti le rimanenti possibilità danno due simboli diversi, per cui la probabilità di mismatch è $1-(1/\sigma)$.

Proviamo a estendere queste osservazioni all'intero pattern P con m caratteri scelti casualmente da Σ. Calcoliamo il numero medio di confronti necessari a stabilire se P è uguale o meno a una stringa qualunque di m caratteri scelta casualmente (quest'ultima stringa rappresenta il contenuto di una finestra di un testo casuale sufficientemente lungo). Per procedere nel calcolo, possiamo immaginare il seguente esperimento mentale. Elenchiamo tutte le possibili stringhe di m caratteri che possono essere costruite con Σ. Tali stringhe sono in numero di σ^m. Per esempio, se Σ è composto dai quattro simboli `A`, `C`, `G` e `T` del DNA e scegliamo $m = 3$, otteniamo $4^3 = 64$ stringhe: `AAA`, `AAC`, `AAG`, `AAT`, `ACA`, `ACC`, `ACG`, `ACT`, ..., `TTA`, `TTC`, `TTG`, `TTT`.

Prendendo ciascuna stringa dall'elenco, la confrontiamo carattere per carattere con il pattern P in maniera analoga al ciclo while (linee 4–5), fermandoci non appena troviamo un mismatch. Nel contempo, contiamo quanti confronti effettuiamo. Per calcolare la media facciamo la somma dei valori e la dividiamo per il loro numero: sommando il numero di confronti effettuati in ciascuna delle σ^m stringhe, otteniamo il numero totale di confronti che va diviso per σ^m. Il risultato ci fornisce il numero di confronti medio $\frac{\sigma}{\sigma-1}\left(1-\frac{1}{\sigma^m}\right)$ per P (nel ciclo while alle linee 4-5) e va moltiplicato per il numero di possibili posizioni $n-m+1$ della finestra scorrevole (nel ciclo for alle linee 3–7).

Guardiamo cosa succede in dettaglio procedendo a ventaglio, cioè considerando il primo carattere del pattern e quello di ogni stringa nell'elenco, poi il secondo carattere del pattern e quello di ogni stringa nell'elenco, quindi il terzo carattere, e così via. Per il primo carattere $P[1]$, abbiamo σ^m stringhe il cui primo carattere carattere viene confrontato con $P[1]$. Ne deriva un parziale di σ^m confronti per $P[1]$. Solo una frazione delle stringhe superano il confronto con esito positivo, precisamente una frazione $1/\sigma$ delle stringhe (quelle che appunto iniziano con il carattere $P[1]$). Nel nostro esempio, se $P[1] =$ `C`, abbiamo che $16 = (1/4)\cdot 4^3$ stringhe nell'elenco superano il confronto con esito positivo: `CAA`, `CAC`, `CAG`, `CAT`, ..., `CTA`, `CTC`, `CTG`, `CTT`.

Quindi, dobbiamo eseguire il resto dei confronti con solo $\sigma^{m-1} = (1/\sigma)\cdot\sigma^m$ stringhe. Le altre stringhe non hanno infatti bisogno di ulteriori confronti in quanto abbiamo trovato subito un mismatch nel primo carattere. Passiamo quindi a confrontare $P[2]$ con il secondo carattere delle σ^{m-1} stringhe che ancora necessitano di essere confrontate. Aggiungiamo un

parziale di σ^{m-1} confronti al totale. In maniera analoga a prima, possiamo concludere che solo un'ulteriore frazione $(1/\sigma)$ sopravvive, ovvero $\sigma^{m-2} = (1/\sigma) \cdot \sigma^{m-1}$ stringhe. Il loro terzo carattere va confrontato con $P[3]$.

A questo punto, dovrebbe essere chiara la legge che regola il nostro esperimento: quando dobbiamo confrontare il carattere $P[i]$ con l'i-esimo carattere delle stringhe ancora in ballo, per $1 \leqslant i \leqslant m$, esaminiamo σ^{m-i+1} stringhe dando luogo a un parziale di σ^{m-i+1} confronti; dopo tali confronti, sopravvivono σ^{m-i} stringhe. Dalla somma dei parziali otteniamo il numero totale di confronti

$$\sum_{i=1}^{m} \sigma^{m-i+1} = \sum_{k=1}^{m} \sigma^k = \frac{\sigma^{m+1}-1}{\sigma-1} - 1 = \frac{\sigma^{m+1}-\sigma}{\sigma-1} = \sigma^m \frac{\sigma}{\sigma-1}\left(1 - \frac{1}{\sigma^m}\right).$$

Notare che abbiamo usato la nota identità $\sum_{k=0}^{m} c^k = \frac{c^{m+1}-1}{c-1}$ per $c \neq 1$. Dividendo il totale così ottenuto per il numero σ^m di stringhe nell'elenco, troviamo il numero medio di confronti del pattern P, ovvero $\frac{\sigma}{\sigma-1}(1 - \frac{1}{\sigma^m})$.

I testi elettronici non hanno però necessariamente una distribuzione casuale uniforme: certi simboli, come le vocali, occorrono più frequentemente di altri. In effetti, esistono delle statistiche su certe classi di testi (per esempio, quelli in lingua inglese dell'epoca vittoriana) in cui i linguisti associano a ogni simbolo $c \in \Sigma$ la frequenza, o meglio la probabilità f_c di apparire in un testo di quella classe, dove $\sum_{c\in\Sigma} f_c = 1$. Tali distribuzioni sono di tipo non uniforme e, nella probabilità di match tra un carattere del pattern P e uno del testo T, il termine $1/\sigma$ va sostituito con $p = \sum_{c\in\Sigma} f_c^2$ (si noti che $p = 1/\sigma$ se $f_c = 1/\sigma$). Di conseguenza, la probabilità di mismatch diventa $1 - p$. Il resto dell'argomentazione appena descritta vale, per cui lasciamo al lettore l'esercizio di completare i dettagli.

2.2. Scansione veloce del testo tramite l'automa

L'inefficienza di RICERCA◇INGENUA al caso pessimo, durante la ricerca del pattern P nel testo T, deriva dall'eventualità che molti caratteri del testo vengano confrontati $O(m)$ volte, ciascuno con corrispondenti caratteri del pattern. Quando cerchiamo P = `ananas` in T = `bananananassata`, per esempio, confrontiamo tutti i caratteri del pattern con quelli del testo che si trovano a partire dalla posizione $j_0 = 2$ (cioè `bananananassata`). Alcuni di questi caratteri (`ana` in `bananananassata`), pur essendo dei match, vengono nuovamente confrontati con i caratteri del pattern quando in seguito consideriamo la posizione $j_0 = 4$, contenente un'occorrenza. Vorremmo invece confrontare solo i rimanenti caratteri (cioè `nas` in `bananananassata`) di cui non conosciamo l'esito come match o mismatch.

Questo comporta lo spostamento della finestra scorrevole permettendole di saltare le posizioni che sicuramente non sono favorevoli ai match, senza però tralasciare alcun'occorrenza. Le posizioni da saltare vanno inferite dai simboli $P[1..i]$ del pattern finora esaminati e dal simbolo corrente $T[j]$ nel testo. Nel nostro esempio $j = 7$ e $T[j] =$ `n`: per $j_0 = 2$, abbiamo $P[1..i] =$ `anana` (con $i = 5$) mentre, per $j_0 = 4$, abbiamo $P[1..i] =$ `ana` (con $i = 3$).

L'idea di base per avere un algoritmo di ricerca più veloce consiste nel separare il procedimento di ricerca in due fasi distinte. Nella prima fase, l'elaborazione preliminare (preprocessing) del pattern produce una o più tabelle di informazioni da utilizzare successivamente nella seconda fase, che consiste in una scansione veloce del testo (scanning) utilizzando le tabelle del pattern. Si noti che l'elaborazione preliminare del pattern avviene *senza* bisogno di accedere ai caratteri del testo. Discutiamo in questa sezione l'utilizzo della tabella delle transizioni δ di un automa deterministico a stati finiti.

	$\delta[i, c]$	a	...	n	...	s	...
------	0	**1**					
a-----	1	*1*		**2**			
an----	2	**3**					
ana---	3	*1*		**4**			
anan--	4	**5**					
anana-	5	*1*		*4*		**6**	
ananas	6	*1*					

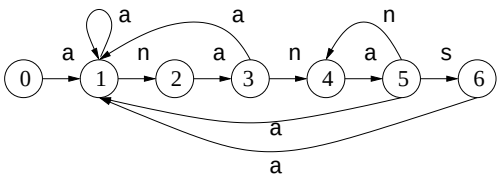


FIGURA 2.1. Tabella δ per il pattern $P = \texttt{ananas}$ e sua rappresentazione grafica. I simboli '-' denotano i caratteri del pattern che non sono stati ancora confrontati con successo all'interno della finestra scorrevole. Le caselle vuote contengono il valore 0, omesso dalla figura. Sono riportate le colonne corrispondenti ai caratteri a, n e s contenuti nel pattern; le altre colonne contengono tutti 0. Nella rappresentazione grafica, un arco dal vertice i al vertice j con etichetta c indica che $\delta[i, c] = j$. Gli archi con $j = 0$ non sono rappresentati.

La tabella δ consiste di $m + 1$ righe e σ colonne. Le righe corrispondono agli stati dell'automa e sono numerate da 0 a m. La riga i corrisponde al prefisso $P[1..i]$ del pattern (quindi, la riga 0 corrisponde alla stringa vuota ϵ). In relazione alla finestra di scorrimento, la riga i indica che abbiamo confrontato con successo i primi i caratteri del pattern e il prossimo carattere c all'interno della finestra va confrontato con il carattere $P[i+1]$ (e imporremo $c = T[j]$ durante la scansione del testo). Le colonne corrispondono ai simboli di Σ e prefigurano quale può essere il carattere c (senza accedere al testo però!). Per esempio, la tabella δ per il pattern $P = \texttt{ananas}$ è riportata in Figura 2.1. L'elemento $\delta[i, c]$ della tabella, dove $0 \leqslant i \leqslant m$ e $c \in \Sigma$, rappresenta il fatto che abbiamo confrontato con successo i primi i caratteri del pattern con i primi caratteri della finestra scorrevole; dobbiamo quindi confrontare il carattere $P[i + 1]$ con il successivo carattere c nella finestra. Inizialmente, per $i = 0$, non abbiamo ancora dei match nel pattern e quindi iniziamo a esaminare $P[1]$. Per $i = m$, abbiamo trovato un'occorrenza e quindi siamo nello stato finale. In generale, abbiamo due possibilità:

- $c = P[i+1]$ per $i < m$ (cfr. i numeri in **neretto** nella tabella in Figura 2.1). Poniamo $\delta[i, c] = i + 1$ per passare allo stato successivo in quanto abbiamo confrontato con successo i primi $i + 1$ caratteri del pattern.
- $c \neq P[i+1]$ oppure $i = m$ (cfr. i numeri in *corsivo* nella tabella in Figura 2.1). Sia $k \leqslant i$ il massimo intero per cui $P[1..k]$ è suffisso della stringa $P[1..i]\,c$, ottenuta concatenando $P[1..i]$ e c. La stringa $P[1..k]$ viene chiamata il *bordo* di $P[1..i]\,c$ a indicare il fatto che occorre sia come suo prefisso che come suo suffisso, ed è la più lunga in tal senso. Poniamo $\delta[i, c] = k$ in questo caso.

La riga $i = 5$ della tabella in Figura 2.1 esemplifica le due possibilità. Tale riga corrisponde ad aver confrontato con successo i primi 5 caratteri anana. Consideriamo quindi la colonna per $c = \texttt{s}$. Poiché $c = P[i + 1]$, possiamo porre $\delta[5, \texttt{s}] = \mathbf{6}$, che rappresenta lo stato finale. Prendendo la colonna per $c = \texttt{a}$, abbiamo $P[1..i]\,c = \texttt{ananaa}$ e quindi $P[1..1] = \texttt{a}$ è il bordo ($k = 1$), per cui $\delta[5, \texttt{a}] = \mathit{1}$. La colonna per $c = \texttt{n}$ è analoga in quanto $P[1..i]\,c = \texttt{ananan}$ e $P[1..4] = \texttt{anan}$ è il bordo ($k = 4$), per cui $\delta[5, \texttt{n}] = \mathit{4}$. Le altre colonne contengono caratteri che non appaiono in P, per cui otteniamo il bordo vuoto ($k = 0$). È chiaro lo scopo della tabella: dopo aver letto il carattere c vogliamo riutilizzare al massimo i caratteri del prefisso del pattern, confrontati con successo fino a quel momento. Questi caratteri costituiscono il bordo che non viene più riesaminato nei confronti successivi.

Esempio. La fase di elaborazione preliminare si rivela pienamente efficace durante la fase di scansione del testo. Infatti, ciascun carattere del testo viene confrontato una *sola* volta. Prendiamo il testo $\mathtt{T} = \mathtt{bananananassata}$. Partendo dallo stato $\mathtt{i} = 0$ e leggendo i caratteri del testo da sinistra a destra otteniamo la sequenza di transizioni: $0 \xrightarrow{\mathtt{b}} 0 \xrightarrow{\mathtt{a}} 1 \xrightarrow{\mathtt{n}} 2 \xrightarrow{\mathtt{a}} 3 \xrightarrow{\mathtt{n}} 4 \xrightarrow{\mathtt{a}} 5 \xrightarrow{\mathtt{n}} 4 \xrightarrow{\mathtt{a}} 5 \xrightarrow{\mathtt{s}} 6 \xrightarrow{\mathtt{s}} 0 \xrightarrow{\mathtt{a}} 1 \xrightarrow{\mathtt{t}} 0 \xrightarrow{\mathtt{a}} 1$. Abbiamo un'occorrenza quando lo stato raggiunto è $\mathtt{i} = 6$. Dopo aver letto `banana` nel testo, siamo nello stato 5. Il carattere successivo del testo è $\mathtt{n} \neq \mathtt{s}$, per cui non troviamo un'occorrenza. Tuttavia, i caratteri `anana` confrontati con successo non sono persi. Avendo letto `n`, andiamo nello stato 4 che rappresenta il bordo di `ananan`. Infatti, è come se ripartissimo avendo già confrontato con successo `anan`.

Pseudocodice. La scansione parte dallo stato $\mathtt{i} = 0$ e applica ripetutamente la transizione definita nella tabella δ alla riga $\mathtt{i}$ e alla colonna $\mathtt{c} = \mathtt{T}[\mathtt{j}]$ che corrisponde al prossimo carattere da leggere nel testo $\mathtt{T}$. Se $\mathtt{i} = \mathtt{m}$, allora siamo nello stato finale e c'è un'occorrenza.

RICERCA◇AUTOMA($\mathtt{P}, \mathtt{T}$):

```
1: m ← |P|; n ← |T|;
2: costruisci l'automa mediante la tabella δ;
3: i ← 0;
4: FOR j ← 1 TO n DO
5:     i ← δ[i, T[j]]
6:     IF i = m THEN
7:         RETURN TRUE   {occorrenza T[j − m + 1..j] = P}
8: RETURN FALSE;
```

Complessità. Assumendo di aver già costruito la tabella δ il cui accesso all'elemento $\delta[\mathtt{i}, \mathtt{c}]$ richiede tempo costante, l'algoritmo RICERCA◇AUTOMA richiede $\mathtt{O}(\mathtt{n})$ tempo per le ricerche di tipo decisionale. Non è difficile modificare lo pseudocodice per effettuare ricerche di tipo quantitativo ed enumerativo. Si noti che l'algoritmo lavora in *tempo reale*, cioè è adatto a elaborare flussi di caratteri richiedendo tempo costante per ciascun carattere letto, senza aver bisogno di mantenere i caratteri di $\mathtt{T}[1..\mathtt{j}]$ in memoria dopo averli utilizzati per le transizioni dell'automa.

Rimane da chiarire come costruire la tabella δ dell'automa. Possiamo procedere per induzione. Partiamo dalla tabella dell'automa che riconosce la stringa vuota. È composta dalla sola riga 0, e ha tutti gli elementi a 0. Aggiungiamo quindi il carattere $\mathtt{i}$-esimo del pattern, per $\mathtt{i} = 1, 2, \ldots, \mathtt{m}$, in modo da ottenere l'automa che trova le occorrenze di $\mathtt{P}[1..\mathtt{i}]$. Quando $\mathtt{i} = \mathtt{m}$, la tabella risultante è δ.

Esempio. Per renderci conto di come realizzare il passo $\mathtt{i}$ nella costruzione della tabella in Figura 2.1 per il pattern $\mathtt{P} = \mathtt{ananas}$, possiamo fissare il valore di $\mathtt{i} = 5$ e vedere come ottenere la tabella che riconosce le occorrenze del pattern $\mathtt{P}_5 = \mathtt{P}[1..5] = \mathtt{anana}$ partendo da quella per $\mathtt{P}_4 = \mathtt{P}[1..4] = \mathtt{anan}$:

	$\delta[\mathtt{i},\mathtt{c}]$	a	...	n	...	s	...
----	0	**1**					
a---	1	*1*		**2**			
an--	2	**3**					
ana-	3	*1*		**4**			
anan	4	*3*					

Tabella per $\mathtt{P}_4 = \mathtt{anan}$.

	$\delta[\mathtt{i},\mathtt{c}]$	a	...	n	...	s	...
-----	0	**1**					
a----	1	*1*		**2**			
an---	2	**3**					
ana--	3	*1*		**4**			
anan-	4	**5**					
anana	5	*1*		*4*			

Tabella per $\mathtt{P}_5 = \mathtt{anana}$

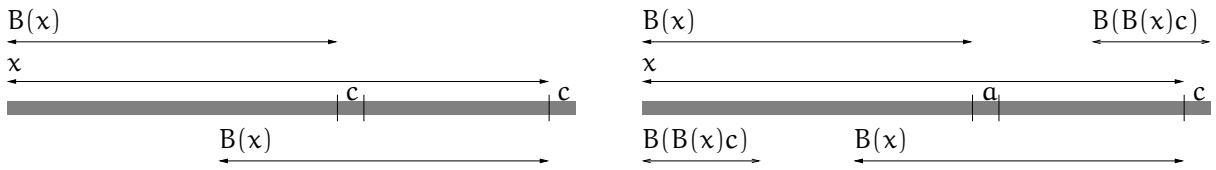


FIGURA 2.2. Definizione ricorsiva di bordo $\mathsf{B}(\mathtt{x}\mathtt{c})$, dove $\mathtt{x}$ è una stringa (vuota o meno) e $\mathtt{a}, \mathtt{c} \in \Sigma$ soddisfano $\mathtt{a} \neq \mathtt{c}$. Il caso 2 è mostrato a sinistra mentre il caso 3 è mostrato a destra.

Osserviamo che lo stato finale per P_4 corrisponde alla riga $\mathtt{i}-1$ e che per ottenere la tabella per P_5 dobbiamo inserire la riga $\mathtt{i} = 5$. Partendo dallo stato $\mathtt{i}-1$, eseguiamo una transizione con il carattere $\mathtt{i}$-esimo $\mathsf{P}[5] = \mathtt{a}$, ottenendo lo stato $\mathtt{s} = \delta\big[\mathtt{i}-1, \mathtt{a}\big] = 3$ nella tabella a sinistra. A questo punto, siccome vogliamo riconoscere P_5, dobbiamo porre $\delta\big[\mathtt{i}-1, \mathtt{a}\big] = \mathbf{5}$ nella tabella a destra per definire la transizione al nuovo stato finale $\mathtt{i}$. Poiché lo stato $\mathtt{i}$, dopo aver letto $\mathtt{a}$, si comporta come lo stato $\mathtt{s}$ (che non è finale!), è sufficiente copiare gli elementi dalla riga $\mathtt{s}$ alla riga $\mathtt{i}$, ottenendo la tabella per P_5 a destra. Come ulteriore esempio, il lettore può eseguire il passo successivo $\mathtt{i} = 6$, per ottenere la tabella in Figura 2.1 a partire dalla tabella di P_5 a destra.

Pseudocodice. La costruzione dell'automa segue lo schema illustrato nell'esempio:

COSTRUZIONE⋄AUTOMA(P):

```
1: m ← |P|;
2: FOREACH c ∈ Σ DO    {Stato iniziale 0}
3:    δ[0, c] ← 0;
4: FOR i ← 1 TO m DO    {Crea lo stato i}
5:    s ← δ[i − 1, P[i]];
6:    δ[i − 1, P[i]] ← i;
7:    FOREACH c ∈ Σ DO    {Copia la riga s nella riga i}
8:       δ[i, c] ← δ[s, c];
9: RETURN tabella δ;
```

La correttezza dell'algoritmo COSTRUZIONE⋄AUTOMA segue dalla definizione ricorsiva di bordo $\mathsf{B}(\mathtt{x})$ di una stringa $\mathtt{x}$ in quanto, al passo $\mathtt{i} \geqslant 1$, l'algoritmo calcola $\mathsf{B}(\mathtt{x})$ per la stringa $\mathtt{x} = \mathsf{P}[1..\mathtt{i}]$ creando un nuovo stato uguale allo stato $\mathtt{s} = |\mathsf{B}(\mathtt{x})|$. Dello stato $\mathtt{s}$, eredita tutti i suoi bordi $\mathsf{B}(\mathtt{x}\mathtt{c}) = \mathsf{B}(\mathtt{y}\mathtt{c})$ per $\mathtt{c} \in \Sigma$, dove $\mathtt{y} = \mathsf{P}[1..s]$.

In generale, possiamo imporre che il bordo della stringa vuota ϵ sia ϵ stessa e definire ricorsivamente il bordo $\mathsf{B}(\mathtt{x}\mathtt{c})$, dove $\mathtt{x}$ è una stringa (vuota o meno) e $\mathtt{c} \in \Sigma$, come segue, in accordo alla Figura 2.2:

(1) $\mathsf{B}(\epsilon) = \epsilon$, e $\mathsf{B}(\mathtt{x}\mathtt{c}) = \epsilon$ se $\mathtt{x} = \epsilon$;
(2) $\mathsf{B}(\mathtt{x}\mathtt{c}) = \mathsf{B}(\mathtt{x})\mathtt{c}$ se $\mathtt{x} \neq \epsilon$ e $\mathtt{c}$ è contenuto nella posizione $|\mathsf{B}(\mathtt{x})| + 1$ di $\mathtt{x}$;
(3) $\mathsf{B}(\mathtt{x}\mathtt{c}) = \mathsf{B}(\mathsf{B}(\mathtt{x})\mathtt{c})$ altrimenti.

Notare che, per induzione, i bordi $\mathsf{B}(\mathtt{x})$ and $\mathsf{B}(\mathsf{B}(\mathtt{x})\mathtt{c})$ necessari al calcolo di $\mathsf{B}(\mathtt{x}\mathtt{c})$ sono stati già trovati nei passi precedenti dell'algoritmo.

Complessità. L'algoritmo COSTRUZIONE⋄AUTOMA esegue $\mathtt{m}+1$ passi, uno per riga, e riempie ciascuna riga in tempo $O(\sigma)$. Il costo totale è quindi $O(\mathtt{m}\sigma)$ tempo e $O(\mathtt{m}\sigma)$ spazio. È possibile migliorare la complessità osservando che i caratteri che non occorrono nel pattern possono condividere la stessa colonna. Sia Σ_{P} l'insieme dei caratteri che compaiono nel pattern P, dove $|\Sigma_{\mathsf{P}}| \leqslant \mathtt{m}$ (nel nostro esempio, $\Sigma_{\mathsf{P}} = \{\mathtt{a}, \mathtt{n}, \mathtt{s}\}$). Usando un vettore di appoggio C di lunghezza σ, possiamo ridurre la tabella δ a sole $|\Sigma_{\mathsf{P}}| + 1$ colonne, denotata δ_{P}, in cui

l'ultima colonna è composta da tutti 0 e le altre colonne sono quelle corrispondenti ai caratteri in Σ_P. Inizializziamo il vettore C ponendo $C[c]$ uguale alla colonna di c in δ_P se $c \in \Sigma_P$; altrimenti, poniamo $C[c]$ uguale all'ultima colonna se $c \notin \Sigma_P$. Non è difficile convincersi che $\delta[i, c] = \delta_P[i, C[c]]$ può essere calcolata in tempo costante. Inoltre, la costruzione ora richiede $O(m|\Sigma_P| + \sigma)$ tempo e spazio, che risulta in un netto miglioramento quando Σ_P è sensibilmente più piccolo di Σ.

In conclusione, la ricerca con automa deterministico, ottenuta eseguendo COSTRUZIONE◇AUTOMA come elaborazione preliminare del pattern e RICERCA◇AUTOMA come scansione del testo, richiede in totale $O(m|\Sigma_P| + n)$ tempo e $O(m|\Sigma_P| + \sigma)$ spazio.

2.3. Un automa compatto: algoritmo di Knuth, Morris e Pratt

L'automa precedente presenta dei problema con pattern lunghi scelti da un alfabeto di grandi dimensioni. Dal punto di vista teorico, ci sono casi in cui l'alfabeto è della stessa dimensione asintotica del testo, per cui la complessità della ricerca con l'automa diventa $O(mn)$ come l'algoritmo RICERCA◇INGENUA. Anche se quest'ultimo caso è estremo, possiamo però notare che il numero di colonne può aumentare al crescere della dimensione dell'alfabeto. Vediamo pertanto come memorizzare una tabella simile a quella dell'automa in modo da avere due sole colonne *indipendentemente* dalla dimensione dell'alfabeto. L'algoritmo che ci accingiamo a studiare è di natura teorica ed è una versione semplificata di quello originariamente proposto da Knuth, Morris e Pratt. Esso ammette un'interessante estensione a più stringhe che studieremo nella Sezione 2.6.

Il fulcro dell'algoritmo è dato da un automa di $m + 1$ righe che ha solo due colonne rappresentate come vettori. Il primo vettore rappresenta, in forma compatta, le transizioni di δ in corrispondenza di un match; lo chiameremo quindi ancora δ ma sarà distinguibile dalla tabella bidimensionale δ in quanto vettore con un solo indice. Il secondo vettore φ rappresenta la tabulazione di una funzione da usare in corrispondenza di un mismatch, la quale è anche chiamata funzione di fallimento: l'elemento $\varphi[i]$ memorizza la lunghezza del bordo di $P[1..i]$. Seppure sembri una lieve differenza rispetto alla definizione adottata nella Sezione 2.2, in cui si cerca il bordo di $P[1..i]\,c$, la definizione di φ permette di rimuovere la dipendenza dall'alfabeto perché non deve tenere conto dei caratteri c che possono seguire $P[1..i]$ durante l'elaborazione preliminare del pattern.

Illustriamo i concetti fin qui svolti partendo dalla tabella in Figura 2.1 per il pattern $P = \texttt{ananas}$. Mostriamo prima il vettore δ che raccoglie in modo compatto i numeri in neretto della tabella in Figura 2.1:

i	0	1	2	3	4	5	6
$\delta[i]$	**1**	**2**	**3**	**4**	**5**	**6**	–

Come si può notare, possiamo evitare di rappresentare esplicitamente il vettore δ in quanto la transizione dallo stato $i \geqslant 0$ allo stato $i + 1$ avviene se e solo se viene letto dal testo un simbolo uguale a $P[i+1]$. Il vettore φ per $P = \texttt{ananas}$ ha tre elementi maggiori di 0, in quanto solo tre prefissi di P hanno un bordo che non è vuoto. Il prefisso $P[1..3] = \texttt{ana}$ ha bordo $\texttt{a}$ di lunghezza 1, il prefisso $P[1..4] = \texttt{anan}$ ha bordo $\texttt{an}$ di lunghezza 2, e il prefisso $P[1..5] = \texttt{anana}$ ha bordo $\texttt{ana}$ di lunghezza 3. Gli altri prefissi hanno bordo vuoto di lunghezza 0. Si usa inoltre il valore al contorno $\varphi[0] = -1$ per garantire la correttezza dello pseudocodice che andremo a esaminre tra poco. I valori della funzione di fallimento sono riassunti nella tabella seguente:

i	0	1	2	3	4	5	6
$\varphi[i]$	-1	0	0	1	2	3	0

Esempio. Proviamo a usare i vettori δ e φ per cercare il pattern $P = \texttt{ananas}$ nel testo $T = \texttt{bananananassata}$. Osserviamo che δ va usato in caso di match per avanzare di stato, mentre φ va usato in caso di mismatch per riposizionare la finestra scorrevole sul testo, andando nello stato opportuno: solo φ va effettivamente memorizzato.

La posizione $j = 1$ dà subito un mismatch per cui passiamo alla posizione $j = 2$. Qui riusciamo ad avere dei match nei primi $i = 5$ caratteri $P[1..i] = \texttt{anana}$ del pattern in corrispondenza delle posizioni 2, 3, ..., 6, del testo (cioè in $\texttt{b}\underline{\texttt{anana}}\texttt{nassata}$). Durante questi match abbiamo implicitamente usato il vettore δ. Il carattere successivo $T[j] = \texttt{n}$ del testo (ora $j = 7$) è diverso da quello $P[i + 1] = \texttt{s}$ nel pattern. Essendo un mismatch, usiamo il vettore φ che sfrutta le proprietà dei bordi. In particolare per $i = 5$, indica che il bordo di $P[1..i]$ è dato dai suoi primi $\varphi[i] = 3$ caratteri $\texttt{ana}$, per cui possiamo tranquillamente spostare la finestra scorrevole ponendo $i = \varphi[i] = 3$ senza perdere alcuna occorrenza. Adesso $P[1..i] = \texttt{ana}$ è allineato a sinistra di $T[j] = \texttt{n}$ (cioè $\texttt{ban}\underline{\texttt{ana}}\texttt{nassata}$) per cui possiamo ripetere il confronto con $P[i + 1] = \texttt{n}$ derivante dal nuovo valore di i.

Questa situazione evidenzia il prezzo da pagare per avere usato due vettori al posto della tabella completa per l'automa: pur evitando di dover esaminare più volte i caratteri di un bordo che hanno dato luogo a un match, uno stesso carattere del testo (quello successivo a un bordo) può essere confrontato più di una volta con i caratteri del pattern. Nel nostro esempio, dove $j = 7$, il primo confronto di $T[j]$ ha dato esito negativo mentre il secondo confronto lo ha dato positivo e quindi possiamo avanzare con i confronti fino a trovare l'occorrenza del pattern nel testo (cioè in $\texttt{ban}\underline{\texttt{anana}}\texttt{s}\underline{\texttt{s}}\texttt{ata}$).

Pseudocodice. Assumento di aver calcolato il vettore φ, manteniano la seguente invariante sulla finestra del testo prima di confrontare il carattere corrente del testo con un opportuno carattere del pattern. Per $i \geqslant 0$, i caratteri di $P[1..i]$ appaiono all'inizio della finestra e il carattere $P[i+1]$ deve essere confrontato con il carattere $T[j]$ nella posizione corrente del testo.

L'invariante è chiaramente valida all'inizio della scansione e in generale quando $i = 0$ poiché $P[1..i]$ è la stringa vuota ϵ. Nel passo generico, confrontiamo $P[i + 1]$ con $T[j]$, dove $0 \leqslant i \leqslant m$ e $1 \leqslant j \leqslant n$. Abbiamo due casi possibili. Match: incrementiamo le variabili i e j, avanzando così allo stato $i + 1$. Mismatch: non potendo avanzare allo stato successivo, spostiamo logicamente la finestra in avanti andando nello stato $i = \varphi[i]$ per mantenere l'invariante. Quindi ritentiamo il confronto con il carattere $P[i + 1]$ derivante dal nuovo valore di i. Iterando questo caso, terminiamo quando troviamo un match oppure $i = \varphi[0] = -1$. In entrambi i casi, possiamo successivamente incrementare l'indice i per P e l'indice j per T, in quanto manteniamo l'invariante.

Vediamo come realizzare lo schema suddetto impiegando solo φ. L'incremento della variabile i è esplicito (e denota l'avanzamento di stato nel vettore implicito δ), mentre quello della variabile j è a carico del ciclo for.

RicercaSemplice$\diamond$KnuthMorrisPratt(P, T):

```
1: m ← |P|; n ← |T|;
2: costruisci la tabella φ;
3: i ← 0;
4: FOR j ← 1 TO n DO
5:    WHILE i ⩾ 0 AND P[i + 1] ≠ T[j] DO
6:       i ← φ[i];
7:    i ← i + 1;
8:    IF i = m THEN
9:       RETURN TRUE;    {occorrenza T[j − m + 1..j] = P}
10: RETURN FALSE;
```

Non è difficile modificare il codice per eseguire ricerche di tipo quantitativo ed enumerativo. Basta sostituire la linea 9 con l'incremento di una variabile contatore oppure con la stampa della posizione $\mathtt{j}-\mathtt{m}+1$. Va inoltre trovato il bordo dell'intero pattern, eseguendo l'istruzione $\mathtt{i} \leftarrow \varphi(\mathtt{i})$ nella linea 9 (si noti che $\mathtt{i} = \mathtt{m}$ in questo caso).

Complessità. L'algoritmo RicercaSemplice$\diamond$KnuthMorrisPratt richiede $O(n)$ tempo più il costo della costruzione del vettore φ. Per valutare il numero di passi durante la scansione del testo è sufficiente mostrare che il numero totale di confronti eseguito dal ciclo while (linee 5–6) è limitato superiormente da $2n$. Infatti il resto del corpo del ciclo for richiede tempo costante e quindi contribuisce $O(n)$ al costo totale.

Notiamo che ogni volta che il carattere $T[j]$ è coinvolto in un match alla linea 5, la variabile j è incrementata insieme alla variabile i. Quindi ci sono esattamente n match nelle n iterazioni del ciclo for. Purtroppo se il carattere $T[j]$ è coinvolto in un mismatch nella linea 5, la variabile i viene decrementata nella linea 6 e quindi $T[j]$ viene confrontato di nuovo.

Fortunatamente, è possibile fissare un tetto al numero di volte in cui questa situazione può accadere: ogni esecuzione della linea 6 corrisponde a uno spostamento in avanti della finestra sul testo. In altre parole, la finestra può solo avanzare, per cui il numero totale di esecuzioni della linea 6 (considerando tutte le iterazioni del ciclo for!) è limitato superiormente dal numero di spostamenti della finestra. Avendo già visto che ci possono essere al più n spostamenti della finestra, possiamo concludere che il numero di esecuzioni (leggi mismatch) della linea 6 è al più n. Ne consegue che il numero totale di confronti è al più $2n$.

Rimane da discutere come costruire il vettore φ. Abbiamo già visto che il valore per $j = 0$ è uguale a -1 come valore al contorno. Per $j > 0$, calcoliamo $\varphi[j]$ a partire da $\varphi[j-1] = i$. In accordo alla definizione di φ, il bordo di $P[1..j-1]$ è di lunghezza i. Ricordiamo che il bordo occorre sia come prefisso che come suffisso e possiamo usare la sua definizione ricorsiva data nella Sezione 2.2 e illustrata nella Figura 2.2. Vediamo come mettere in relazione questo con il bordo di $P[1..j] = P[1..j-1]P[j]$.

Esempio. Prendiamo il pattern $P = \mathtt{ananas}$ e fissamo $j = 5$ supponendo di avere calcolato il bordo $\mathtt{an}$ di $P[1..j-1] = \mathtt{anan}$ per cui $\varphi[j-1] = i = 2$. Nel considerare il bordo di $P[1..j] = \mathtt{anana}$, proviamo a estendere il bordo di $P[1..j-1]$ con $P[j]$. In effetti, se $P[i+1] = P[j]$, abbiamo trovato che il bordo è di lunghezza $i+1$, per cui poniamo $\varphi[j] = i+1$ (siamo nel caso 2 della Figura 2.2). In questo caso siamo fortunati, perché il bordo è $\mathtt{ana}$ in quanto $P[3] = P[5] = \mathtt{a}$.

Cosa succede invece se $P[i+1] \neq P[j]$? Questo è il caso che incontriamo quando $j = 6$ (ora $\varphi[j-1] = i = 3$). Vale $P[4] = \mathtt{n} \neq \mathtt{s} = P[5]$, per cui il bordo di $P[1..6] = \mathtt{ananas}$ non può essere $i+1$ ma un più *piccolo* prefisso e suffisso che termina con $\mathtt{s}$ (siamo nel caso 3 della Figura 2.2). La proprietà interessante è che se rimuoviamo la $\mathtt{s}$ otteniamo un bordo del bordo! In altre parole trasformiamo il problema di cercare il bordo di $P[1..j] = P[1..j-1]P[j]$ nel problema di trovare il bordo di $P[1..i]P[j]$. Iterando con $i = \varphi[i]$ scaliamo ogni volta il bordo fino a che non troviamo il bordo adatto, che è il bordo vuoto quando $j = 6$.

Pseudocodice. L'algoritmo per calcolare il vettore φ si basa quindi sui bordi dei prefissi del pattern. Per brevità di esposizione, identifichiamo ciascun prefisso $P[1..i]$ con il proprio indice i. Inizialmente, poniamo $\varphi[0] = -1$ e poi, per $j = 1, 2, \ldots, m$, calcoliamo $\varphi[j]$ usando il valore $i = \varphi[j-1]$ trovato al passo precedente. Estendere il bordo precedente significa verificare che $P[i+1] = P[j]$. In caso di successo, poniamo $\varphi[j] = i+1$; altrimenti proviamo con il bordo del bordo i, definito da $\varphi[i]$, e iteriamo ponendo $i = \varphi[i]$. L'iterazione ha termine quando troviamo il bordo per j, oppure concludiamo che il suo bordo è vuoto ($i = -1$). In entrambi i casi, il valore $i+1$ correttamente fornisce il valore del bordo per $P[1..j]$, per cui

possiamo porre $\varphi[j] = i + 1$.

CostruzioneSemplice$\diamond$KnuthMorrisPratt(P):

```
1: m ← |P|;
2: φ[0] ← −1;    {condizione al contorno}
3: FOR j ← 1 TO m DO
4:    i ← φ[j − 1];
5:    WHILE i ⩾ 0 AND P[i + 1] ≠ P[j] DO
6:       i ← φ[i];
7:    φ[j] ← i + 1;
8: RETURN tabella φ;
```

Complessità. L'analogia tra gli algoritmi CostruzioneSemplice$\diamond$KnuthMorrisPratt e RicercaSemplice$\diamond$KnuthMorrisPratt non è casuale. Infatti, il calcolo di φ può essere visto come il problema di cercare il pattern in se stesso, per cui la complessità è lineare anche in questo caso. In caso di match, avanzano entrambi gli indici i e j. In caso di mismatch, ogni esecuzione della linea 6 decrementa i di un certo valore. Tuttavia, abbiamo già visto che fa avanzare la finestra scorrevole, per cui la linea 6 non può essere eseguita più di m volte. Il costo totale è quindi $O(m)$ tempo e spazio.

In conclusione, la ricerca del pattern con gli algoritmi appena esposti richiede tempo $O(m+n)$ indipendentemente dalla dimensione dell'alfabeto. La riduzione a $O(m)$ dello spazio e del tempo di elaborazione preliminare del pattern ha come controparte che l'algoritmo non lavora più in tempo reale come l'algoritmo Ricerca$\diamond$Automa.

L'analogia tra lo pseudocodice di CostruzioneSemplice$\diamond$KnuthMorrisPratt e di RicercaSemplice$\diamond$KnuthMorrisPratt suggerisce che, volendo, si può utilizzare solo l'algoritmo di CostruzioneSemplice$\diamond$KnuthMorrisPratt per risolvere il problema dello string matching esatto. Scegliamo un simbolo # non appartenente all'alfabeto Σ e concateniamo il pattern e il testo separandoli con #. Applichiamo alla stringa risultante $S = P\#T$, l'algoritmo CostruzioneSemplice$\diamond$KnuthMorrisPratt(S) per trovare i bordi di S. Lasciamo come esercizio al lettore la dimostrazione che P occorre in $T[j - m + 1..j]$ se e solo se P è un bordo di $P\#T[1..j]$.

Lo svantaggio di tale metodo è l'occupazione di spazio, visto che abbiamo bisogno di $O(m + n)$ celle di memoria. Tuttavia è possibile ridurre lo spazio a $O(m)$ osservando che i bordi in S sono lunghi al massimo quanto P perché abbiamo inserito il separatore #. Non c'è bisogno di mantenere φ per valori maggiori di m in quanto $\varphi[\,] \leqslant m$ nelle linee 4, 6 e 7 di CostruzioneSemplice$\diamond$KnuthMorrisPratt.

Un'ultima considerazione riguarda l'algoritmo originale di Knuth, Morris e Pratt. La loro definizione di φ è data da una funzione di fallimento f. Vale $f(m) = \varphi[m]$. Tuttavia, per $j < m$, la funzione f effettua spostamenti maggiori della finestra scorrevole in quanto il bordo i di $P[1..j-1]$ associato a $f(j)$ ha l'ulteriore proprietà che $P[i + 1] \neq P[j]$.

Nel caso del pattern $P =$ `ananas`, fissando $j = 5$ abbiamo che il bordo calcolato da $\varphi[j - 1] = 2$ è `an` mentre il bordo associato a $f(j)$ è vuoto in quanto entrambe le occorrenze di `an` sono seguite da `a`, per cui se c'è un mismatch su tale `a` dopo aver letto `anan` è inutile considerare il bordo `an` in quanto anch'esso è seguito da `a`: quest'ultimo bordo produrrà quindi un mismatch come `anan`. Il codice risultante differisce in parte da quello di RicercaSemplice$\diamond$KnuthMorrisPratt e CostruzioneSemplice$\diamond$KnuthMorrisPratt, anche se l'idea algoritmica rimane identica nella sostanza.

2.4. Praticamente veloce: algoritmo di Boyer, Moore, Gosper e Horspool

Gli algoritmi finora discussi hanno la caratteristica comune di esaminare, come minimo, *tutti* i caratteri del testo al caso pessimo. A tale proposito, un risultato di Rivest afferma che gli algoritmi di string matching che usano confronti devono esaminare almeno $n - m + 1$ caratteri al caso pessimo: questo risultato non implica però che si possano esaminare meno caratteri al caso medio, ma purtroppo gli algoritmi delle sezioni precedenti non hanno questa proprietà.

Presentiamo quindi un algoritmo veloce, inizialmente proposto da Boyer e Moore (e indipendentemente da Gosper) in contemporanea a quello di Knuth, Morris e Pratt, che ha il vantaggio di richiedere $O\big(m+(n/m)\log m\big)$ tempo in media. In altre parole, per ogni porzione di m caratteri del testo, questo algoritmo ne confronta *solo* $O(\log n)$ in media, per cui il tempo atteso di ricerca è sublineare per pattern lunghi, quando $m > \log n$.

Descriviamo qui solo le idee principali dell'algoritmo, per poi fornire i dettagli di una sua popolare variante pratica. Partiamo innanzi tutto da un'osservazione sulla finestra di scorrimento sul testo. Pur essendo più intuitivo confrontare i caratteri nella finestra da sinistra a destra, non esiste una particolare motivazione algoritmica per impedire che i confronti vengano effettuati da destra a sinistra, ossia partendo dalla fine della finestra. Un potenziale vantaggio di procedere in questa maniera è la possibilità, in caso di mismatch, di spostare la finestra più in avanti degli altri algoritmi. Per esempio, se l'ultimo carattere della finestra non appare affatto nel pattern, possiamo spostare la finestra di m posizioni, pari alla lunghezza del pattern.

L'euristica sopra non basta a garantire una buona complessità nel caso pessimo. Per questo motivo, l'algoritmo di Boyer e Moore trae vantaggio sia dai confronti con esito positivo (match) che da quelli con esito negativo (mismatch), contrariamente a quanto accade nell'algoritmo di Knuth, Morries e Pratt che trae vantaggio dai soli match. Il cardine della computazione è ancora una volta il calcolo veloce dello spostamento della finestra, che avviene prendendo il massimo tra gli spostamenti suggeriti da due tabelle descritte di seguito. Ipotizziamo di aver confrontato con successo zero o più caratteri della finestra a partire da destra, denotati da u, e di essere incappati in un mismatch tra il carattere a nel pattern e l'omologo carattere $b \neq a$ nella finestra. La Figura 2.3.a illustra una tale situazione in cui au è suffisso del pattern e bu è suffisso della finestra.

La prima tabella trae vantaggio dal fatto che i caratteri in u sono dei match. È simile per certi versi a quella di Knuth, Morris e Pratt, solo che lavora sul suffisso u. Partendo dalla destra del pattern, cerchiamo la prima occorrenza di u verso sinistra purché sia preceduta da un carattere diverso. La tabella permette di recuperare, in tempo costante, la prima occorrenza di cu a sinistra di au nel pattern, dove $c \neq a$. La motivazione di tale tabella risiede nel fatto che l'allineamento della sottostringa cu del pattern con quella bu del testo provoca uno spostamento della finestra verso sinistra (cfr. Figura 2.3.b). Non è detto che cu esista nel pattern. In tal caso, lo spostamento della finestra è ancora possibile, in quanto la tabella indica quale è il più lungo prefisso del pattern che appare come suffisso di u (il lettore avrà collegato tale prefisso alla nozione di bordo). La Figura 2.3.c mostra una tale situazione. In ogni caso abbiamo uno scorrimento della finestra a sinistra.

La seconda tabella trae vantaggio dal mismatch di a e b. La tabella memorizza la posizione dell'occorrenza più a destra di b nel pattern. Se b non vi occorre, tale posizione viene posta uguale a 0. Usando la tabella, facciamo slittare la finestra in modo da allineare il carattere b nel pattern con quello nel testo che precede u (cfr. Figura 2.3.d). Chiaramente, se b non appare nel pattern, spostiamo la finestra completamente a destra di b (cfr. Figura 2.3.e).

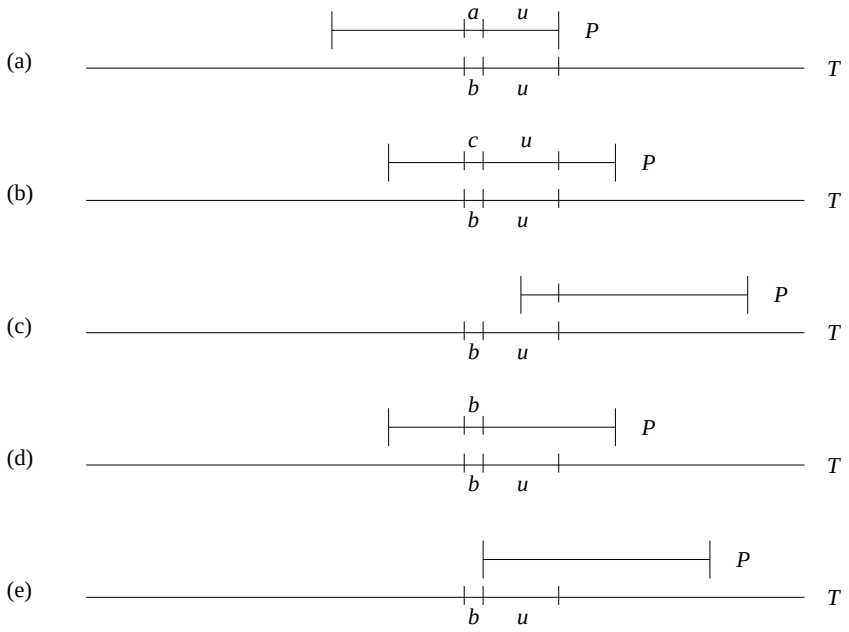


FIGURA 2.3. Situazione di mismatch nella finestra con l'algoritmo di Boyer e Moore. (a) Dopo aver confrontato con successo una sottostringa $\mathtt{u}$ (possibilmente vuota) a partire da destra, troviamo due caratteri $\mathtt{a} \neq \mathtt{b}$. (b)–(c) Spostamento della finestra con la prima tabella. (d)–(e) Spostamento della finestra con la seconda tabella.

Osserviamo che non sempre si ottiene uno spostamento in avanti con la seconda tabella, per cui l'algoritmo di Boyer e Moore sceglie il massimo spostamento tra quelli indicati dalle due tabelle. Dopo ogni spostamento, l'algoritmo parte sempre dalla fine della finestra per confrontare i caratteri verso sinistra. In tal modo può confrontare di nuovo porzioni di testo che avevano dato luogo a dei match. Questo causa un comportamento al caso pessimo che richiede $O(mn)$ tempo, quando il pattern è periodico, ovvero è formato da una stringa ripetuta più volte seguita da un suo prefisso. Sono state proposte delle varianti che riducono tale costo a $O(m+n)$ nel caso pessimo ed è stato dimostrato che il costo medio è $O\big(m + (n/m)\log n\big)$, motivando così la sua velocità in pratica.

Diverse varianti dell'algoritmo di Boyer e Moore sono state successivamente trovate: tra queste, discutiamo quella di Gosper e Horspool, in cui si usa solamente un vettore λ che è una semplice variante della seconda tabella descritta sopra. In particolare, l'elemento $\lambda[\mathtt{c}]$ permettte di allineare il carattere $\mathtt{c}$ in *fondo* alla finestra del testo con la sua occorrenza più a destra nel pattern. A tal fine vengono considerati in λ tutti i caratteri del pattern tranne l'ultimo (altrimenti la finestra non scorrerebbe se $\mathtt{c}$ occorresse anche come ultimo carattere del pattern). Come si può notare, indipendentemente dal carattere $\mathtt{b}$ di mismatch, usiamo l'ultimo carattere $\mathtt{c}$ della finestra per riallineare il pattern. In questo modo la finestra scorre sempre di almeno una posizione in avanti.

Esempio. Proviamo a calcolare il vettore λ per il pattern $P = \texttt{ananas}$, escludendo l'ultimo carattere $\mathtt{s}$ dalla computazione. Vogliamo usare λ per far spostare la finestra, il cui ultimo carattere è $\mathtt{c}$. Di conseguenza, se $\mathtt{c}$ non appare nelle prime $m-1$ posizioni del pattern dobbiamo porre $\lambda[\mathtt{c}] = m = 6$ a indicare lo spostamento completo della finestra a destra di $\mathtt{c}$. Se invece $\mathtt{c}$ appare in $\mathtt{i}$ come ultima posizione nel pattern, dobbiamo spostare la finestra di $m-\mathtt{i}$ posizioni per allineare $\mathtt{c}$.

Vediamo alcuni esempi concreti. Il carattere `s` non appare nelle prime $m-1$ posizioni di P, per cui il suo valore di λ è $m = 6$. Il carattere `a` appare per l'ultima volta in posizione $i = 5$ del pattern: poniamo quindi il valore di λ pari a $m - i = 1$. Infatti, se la finestra contiene `a` come ultimo carattere, lo spostamento della finestra di una posizione in avanti fa arretrare `a` di una posizione all'interno della finestra, per cui si allinea con il corrispettivo `a` in posizione 5 del pattern.

Analogamente, il carattere `n` appare per l'ultima volta in posizione $i = 4$ del pattern: poniamo quindi il valore di λ pari a $m - i = 2$. Alla fine, otteniamo il seguente vettore, dove le colonne non specificate contengono il valore $m = 6$:

c	`a`	...	`n`	...	`s`	...
$\lambda[c]$	1		2		6	

Ora vediamo come utilizzare il vettore λ per la scansione del testo: a tal fine, prendiamo il testo $T =$ `bananananassata` e la finestra `banana` in prima posizione. Iniziamo a confrontare dall'ultimo carattere $c =$ `a`, ma scopriamo subito un mismatch con l'ultimo carattere `s` del pattern. Ci spostiamo di $\lambda[c] = 1$ posizioni, ottenendo la finestra `ananan` (ora i caratteri `a` in penultima posizione del pattern e della finestra sono allineati).

Qui osserviamo che l'ultimo carattere $c =$ `n` della finestra dà luogo a un mismatch, per cui facciamo scorrere la finestra di altre $\lambda[c] = 2$ posizioni, ottenendo `ananas` e quindi trovando l'occorrenza. Proseguendo la computazione, abbiamo $c =$ `s` nell'ultima posizione della finestra, per cui spostiamo la finestra di $\lambda[c] = 6$ posizioni terminando la ricerca del pattern.

Come si può notare, abbiamo evitato di esaminare tutti i caratteri del testo: precisamente, solo i caratteri sottolineati in `bananananassata` sono stati confrontati.

Pseudocodice. L'algoritmo che presentiamo è molto veloce in pratica poiché mantiene le prestazioni sperimentali dell'algoritmo originale di Boyer e Moore senza la complicazione delle due tabelle: il prezzo da pagare è che il caso pessimo diventa $O(mn)$ tempo. La sua sorprendente semplicità è illustrata nel codice descritto di seguito che prende il nome anche dei suo proponenti Gosper e Horspool.

La costruzione del vettore λ prevede una semplice elaborazione preliminare del pattern per cercare l'ultima occorrenza di ogni carattere nelle prime $m - 1$ posizioni. Se il carattere occupa la posizione i, allora lo spostamento della finestra è di $m - i$ posizioni in avanti. I caratteri che non compaiono nel pattern danno luogo a spostamenti di m posizioni.

Notare che l'ultimo carattere $P[m]$ del pattern non va considerato per garantire uno spostamento di almeno una posizione.

COSTRUZIONE ◇BOYERMOOREGOSPERHORSPOOL(P):

1: $m \leftarrow |P|$;
2: FOREACH $c \in \Sigma$ DO
3: $\quad \lambda[c] \leftarrow m$;
4: FOR $i \leftarrow 1$ TO $m - 1$ DO
5: $\quad \lambda\big[P[i]\big] \leftarrow m - i$;
6: RETURN tabella λ;

Il vettore λ viene poi utilizzato nella scansione del testo, sempre in corrispondenza dell'ultimo carattere della finestra corrente, indicato da $T[j]$, indipendentemente dall'esito del confronto con il pattern.

Tale confronto è realizzato nelle linee 7–8 e procede a partire dall'ultimo carattere. Il passo i confronta i caratteri i-esimi da destra, ossia $P[m - i]$ e $T[j - i]$. Lo spostamento della finestra di almeno una posizione in avanti avviene nella linea 11.

RICERCA⋄BOYERMOOREGOSPERHORSPOOL(P, T):

```
1: m ← |P|; n ← |T|;
2: costruisci la tabella λ;
3: j ← m;
4: WHILE j ⩽ n DO
5:    IF T[j] = P[m] THEN
6:       i = 1;
7:       WHILE i < m AND P[m − i] = T[j − i] DO
8:          i ← i + 1;
9:       IF i = m THEN
10:         RETURN TRUE;    {occorrenza T[j − m + 1..j] = P}
11:   j ← j + λ[T[j]];
12: RETURN FALSE;
```

L'algoritmo RICERCA⋄BOYERMOOREGOSPERHORSPOOL può effettuare uno spostamento maggiore in alcuni casi con una semplice modifica. Anziché usare l'ultimo carattere $T[j]$ della finestra, possiamo usare il successivo carattere $T[j + 1]$ senza perdere eventuali occrrenze. Infatti, lo spostamento della finestra deve comunque coinvolgere $T[j + 1]$. La linea 11 usa perciò $\lambda[T[j+1]]$. Tuttavia la costruzione di λ va modificata in quanto dobbiamo tenere conto che anche l'ultimo carattere del pattern contibuisce e che tutti gli spostamenti devono essere aumentati di una posizione. È quindi possibile modificare COSTRUZIONE⋄BOYERMOORE-GOSPERHORSPOOL come segue: l'assegnamento alla linea 3 impiega il valore $m + 1$, il ciclo for alla linea 4 arriva fino a m e l'assegnamento alla linea 5 impiega il valore $m-i+1$. Lasciamo al lettore come modificare di conseguenza RICERCA⋄BOYERMOOREGOSPERHORSPOOL.

Complessità. La complessità di RICERCA⋄BOYERMOOREGOSPERHORSPOOL è $O(mn)$ tempo al caso pessimo e quindi peggiore delle varianti di Boyer e Moore che richiedono tempo lineare. Tuttavia, il suo ottimo comportamento in pratica ci spinge a studiare la sua complessità al caso medio utilizzando lo stesso modello di distribuzione probabilistica adottato per l'analisi dell'algoritmo RICERCA⋄INGENUA descritto nella Sezione 2.1, in cui un match ha probalilità $1/\sigma$ e un mismatch ha probalilità $1 - (1/\sigma)$, dove σ è la dimensione dell'alfabeto Σ.

Mostriamo infatti che RICERCA⋄BOYERMOOREGOSPERHORSPOOL ha un costo medio di $O(m+\sigma+n/\sigma)$, ovvero esamina mediamente uno ogni σ caratteri del testo, per cui con alfabeti sufficientemente grandi diventa sublineare. Per un confronto, tale complessità è migliore di quella media $O(n)$ dell'algoritmo RICERCA⋄INGENUA e asintoticamente peggiore di quella media $O(m + (n/m) \log m)$ dell'algoritmo di Boyer e Moore, in quanto σ è fissato.

Il costo di COSTRUZIONE⋄BOYERMOOREGOSPERHORSPOOL è comunque $O(m + \sigma)$, per cui concentriamo l'analisi sul resto della computazione in RICERCA⋄BOYERMOOREGOSPER-HORSPOOL per mostrare il suo costo medio $O(m/\sigma)$. Il costo medio è dato dal prodotto di due componenti, il numero medio di spostamenti della finestra e il numero medio di confronti all'interno della finestra. Abbiamo già valutato nella Sezione 2.1 il numero medio di confronti all'interno di una finestra:

$$\frac{\sigma}{\sigma - 1}\left(1 - \frac{1}{\sigma^m}\right) \leqslant 2.$$

Rimane da valutare il numero medio di spostamenti della finestra. Poiché ci sono $n-m+1$ possibili posizioni per la finestra, se riusciamo a valutare il numero medio $\bar{s}$ di posizioni saltate con uno spostamento di finestra (quando eseguiamo la linea 11), possiamo concludere che il

numero medio di finestre esaminate è $(n - m + 1)/\bar{s}$. Nel resto della sezione mostriamo che

$$\bar{s} = \sigma \left[1 - \left(1 - \frac{1}{\sigma} \right)^m \right],$$

dove tale valore medio aumenta al crescere della dimensione dell'alfabeto o della lunghezza del pattern. Possiamo concludere che con RicercaBoyerMooreGosperHorspool dobbiamo considerare solo $(n - m + 1)/\bar{s} = O(n/\sigma)$ finestre in media, esaminando un numero costante di caratteri in media. Il costo medio totale è quindi $O(n/\sigma)$ tempo.

Il resto di questa sezione è rivolto al lettore interessato all'analisi degli algoritmi. Proviamo a dimostrare analiticamente il valore di $\bar{s}$ considerando l'ultimo carattere $c = T[j]$ della finestra e il suo valore nel vettore λ. Abbiamo due casi:

- $\lambda[c] = m$: vuol dire che c non appare nelle prime $m - 1$ posizioni del pattern (ricordiamo che l'ultima posizione m non viene considerata). In altre parole, c è diverso da tali $m - 1$ caratteri e ciò accade con probabilitá pari a $[1 - (1/\sigma)]^{m-1}$.
- $\lambda[c] = m - i$, per $1 \leqslant i \leqslant m - 1$: in altre parole, c è uguale al carattere $P[i]$, mentre è diverso dagli altri caratteri in $P[i + 1, m - 1]$. Questo accade con probabilità pari a $(1/\sigma)[1 - (1/\sigma)]^{m-i-1}$.

Riassumendo i due casi, abbiamo uno spostamento di m posizioni con probabilità $[1 - (1/\sigma)]^{m-1}$ e, per $1 \leqslant i \leqslant m - 1$, abbiamo uno spostamento di $m - i$ posizioni con probabilità $(1/\sigma)[1 - (1/\sigma)]^{m-i-1}$. Ne segue che lo spostamento medio $\bar{s}$ è dato dalla media pesata di questi valori,

$$\bar{s} = m[1 - (1/\sigma)]^{m-1} + \sum_{i=1}^{m-1} (m - i)(1/\sigma)[1 - (1/\sigma)]^{m-i-1} \tag{1}$$

Con un po' di manipolazioni algebriche, mostriamo come semplificare l'espressione in (1). Usiamo la nota formula

$$\sum_{j=1}^{m-1} j c^j = \frac{c + (m - 1)c^{m+1} - m c^m}{(c - 1)^2}, \tag{2}$$

dimostrabile per induzione su m. Partendo dalla formula (1), mettiamo in evidenza $\frac{1/\sigma}{1-(1/\sigma)}$ nella sommatoria e sostituiamo $m - i$ con j, per cui la formula diventa

$$m[1 - (1/\sigma)]^{m-1} + \frac{1/\sigma}{1 - (1/\sigma)} \sum_{j=1}^{m-1} j[1 - (1/\sigma)]^j. \tag{3}$$

Applicando la formula nota (2) con $c = 1 - (1/\sigma)$, otteniamo

$$m[1 - (1/\sigma)]^{m-1} + \frac{1/\sigma}{1 - (1/\sigma)} \frac{[1 - (1/\sigma)] + (m - 1)[1 - (1/\sigma)]^{m+1} - m[1 - (1/\sigma)]^m}{1/\sigma^2}. \tag{4}$$

Semplificando le due frazioni otteniamo

$$m[1 - (1/\sigma)]^{m-1} + \sigma\big\{1 + (m - 1)[1 - (1/\sigma)]^m - m[1 - (1/\sigma)]^{m-1}\big\}. \tag{5}$$

Possiamo portare il primo termine all'interno delle parentesi graffe, dove poi mettiamo in evidenza il termine $-[1 - (1/\sigma)]^m$, ottenendo

$$\sigma\big\{1 - [1 - (1/\sigma)]^m\big(-(m/\sigma)/[1 - (1/\sigma)] - (m - 1) + m/[1 - (1/\sigma)]\big)\big\}. \tag{6}$$

A questo punto, non è difficile semplificare $-(m/\sigma)/[1-(1/\sigma)]-(m-1)+m/[1-(1/\sigma)] = 1$, ottenendo che la (1) si riduce a

$$\bar{s} = \sigma \left[1 - \left(1 - \frac{1}{\sigma} \right)^m \right]. \tag{7}$$

2.5. Comportamento pratico degli algoritmi di ricerca esatta

Abbiamo discusso degli algoritmi di ricerca esatta nelle sezioni precedenti mettendo in rilievo la loro complessità computazionale. Discutiamo brevemente del loro comportamento pratico, che dipende dalla cardinalità σ dell'alfabeto e dalla lunghezza m del pattern.

Da osservazioni sperimentali risulta che RICERCA◇INGENUA risulta meno efficiente degli altri algoritmi nella maggior parte dei casi. In particolare, RICERCA◇AUTOMA e RICERCASEMPLICE◇KNUTHMORRISPRATT risultano più veloci con alfabeto binario o pattern corti, in quanto aumenta la probabilità che un prefisso del pattern abbia un match. In tutti gli altri casi (in pratica i più frequenti), RICERCA◇BOYERMOOREGOSPERHORSPOOL è sorprendentemente veloce per la sua semplicità. L'estensione dell'algoritmo a trattare coppie di caratteri consecutivi invece che singoli caratteri dovrebbe non peggiorare lo spostamento ottenuto; pur migliorando le prestazioni in diversi casi, va precisato che il sovraccarico computazionale necessario a gestire coppie di caratteri anziché singoli caratteri non sempre ripaga. Per completezza va detto che RICERCA◇INGENUA è una soluzione competitiva quando si vuole trovare la prima occorrenza di un pattern e questa occorrenza ha una buona probabilità di trovarsi all'inizio del testo, oppure si cercano pattern di piccola dimensione (con meno di una decina di caratteri). In tal caso, il costo addizionale di COSTRUZIONE◇AUTOMA, COSTRUZIONESEMPLICE◇KNUTHMORRISPRATT e COSTRUZIONE◇BOYERMOOREGOSPERHORSPOOL potrebbe non ripagare il vantaggio della conseguente scansione veloce del testo in quanto quest'ultima si ferma nelle posizioni iniziali.

Ci sono varianti che impongono un ordine sui caratteri del pattern da confrontare in base alle frequenze, ma ciò non sembra aiutare molto, a meno che uno non ricalcoli di volta in volta tali frequenze in base al testo da cercare (il che richiede ulteriore tempo di elaborazione). Inoltre usare due euristiche di spostamento per la finestra (come l'algoritmo originale di Boyer e Moore) al posto di una (come gli altri algoritmi) non aiuta in maniera significativa dal punto di vista pratico. Infine, c'è poca correlazione tra la velocità effettiva dell'algoritmo e il numero di confronti effettuati al caso pessimo, anche se un basso numero di confronti garantisce comunque un algoritmo efficiente.

2.6. Ricerca simultanea di più stringhe: algoritmo di Aho e Corasick

Diverse applicazioni richiedono la ricerca simultanea di più pattern $P_1, \ldots, P_k$ durante la scansione del testo T. Per esempio, tale ricerca è utile per espandere gli acronimi e per verificare la correttezza delle citazioni bibliografiche: ogni pattern rappresenta un acronimo o una macro da espandare e ogni sua occorrenza viene sostituita dalla sua macroespansione.

Un altro esempio è l'individuazione di file contenenti virus informatici. Molti virus lasciano una "impronta digitale" della loro infezione riconoscibile come occorrenza di una certa sequenza di byte all'interno del codice eseguibile. La presenza dell'impronta non implica necessariamente quella di un virus, ma viene usata come filtro veloce per selezionare i file su cui eseguire dei controlli più approfonditi e costosi computazionalmente. Il riconoscimento di impronte equivale al problema dello string matching di più stringhe e deve essere efficiente per non rallentare eccessivamente l'uso del calcolatore a causa dei controlli: se una certa sequenza occorre in un file allora viene lanciato un programma più sofisticato (e lento) di individuazione

dei virus; se invece la sequenza non appare, il file non è stato infettato da quel tipo di virus. Ora il punto è che esistono migliaia di virus, per cui è impensabile scandire migliaia di volte lo stesso file. Piuttosto, in una singola scansione, bisogna verificare velocemente se almeno una delle migliaia di impronte digitali occorre nel file.

Nella nostra terminologia, indichiamo con $M = |P_1| + \cdots + |P_k|$ la somma delle lunghezze dei pattern e con n la lunghezza del testo T. Usando le tecniche viste finora possiamo eseguire k ricerche indipendenti, una per ciascun pattern, in tempo totale pari a $O(M + kn)$. Aho e Corasick hanno proposto un metodo, coevo a quello di Knuth, Morris e Pratt, che richiede solo $O(M + n \log \sigma)$ tempo a cui va aggiunto un costo lineare con il numero di occorrenze riportate. In altre parole, la fase di scansione del testo è k volte più veloce se $\sigma = O(1)$, per cui tale metodo è preferibile a quello che ripete k ricerche indipendenti. Si ha un guadagno notevole in termini di velocità quando vi è un grande numero k di pattern da cercare. Per esempio, questo metodo di ricerca multipla è realizzato nel programma di utilità `fgrep` presente nei sistemi operativi Linux e Unix.

In questa sezione, vediamo una forma semplificata del metodo di Aho e Corasick. L'idea principale si può descrivere partendo dai singoli pattern. Immaginiamo di voler cercare indipendentemente ciascun pattern P_j, dove $1 \leqslant j \leqslant k$. Costruiamo i vettori δ_j e φ_j del rispettivo automa mediante l'algoritmo CostruzioneSemplice$\diamond$KnuthMorrisPratt, applicato singolarmente a ciascun P_j. Volendo realizzare la ricerca su tutti questi automi in una singola passata, possiamo realizzare un qualche meccanismo di fusione dei vettori δ_j e φ_j ottenendo due sole tabelle δ e φ, condivise da tutti i pattern, che ne permettono la ricerca simultanea. Il processo di fusione avviene in due parti.

- Nella prima parte, costruiamo uno *scheletro* dell'automa in cui fondiamo i soli vettori δ_j, in modo da condividere le porzioni iniziali dei vettori che corrispondono ai prefissi uguali dei pattern.
- Nella seconda parte, fondiamo i vettori φ_j estendendo la nozione di bordo, introdotta nella Sezione 2.2, a un insieme di stringhe piuttosto che a una singola stringa: informalmente, il bordo all'interno di un pattern può anche essere prefisso di almeno uno degli altri pattern.

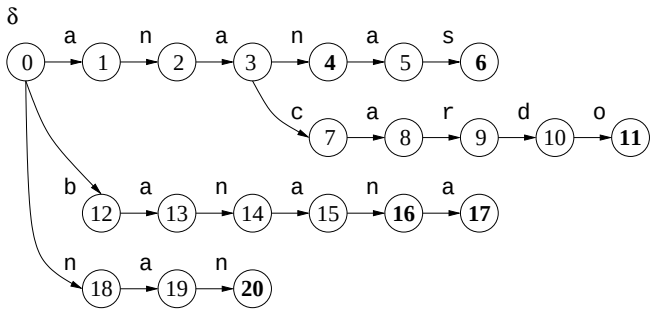
Esempio. Illustriamo i concetti appena esposti con un esempio, in cui vogliamo cercare simultaneamente i pattern $P_1 =$ `ananas`, $P_2 =$ `anacardo`, $P_3 =$ `banana` e $P_4 =$ `nan` (quest'ultimo viene introdotto perché occorre all'interno di P_1 e P_3). Mostriamo adesso la prima parte, in cui costruiamo lo scheletro ottenendo una sola tabella δ per tutti i pattern da cercare.

Con riferimento alla Figura 2.4, la tabella δ viene inizializzata con il pattern P_1: viene creato uno stato iniziale 0 e viene allocata, per ciascun carattere $P_1[i]$, la riga per lo stato i, analogamente a quanto fatto con l'algoritmo RicercaSemplice$\diamond$KnuthMorrisPratt. In altre parole, guardando ai soli stati numerati da 0 fino a 6, otteniamo la tabella δ_1 di P_1.

Quando invece trattiamo P_2, non creiamo più la tabella δ_2: cerchiamo il più lungo prefisso di P_2, ossia `ana`, in comune con i pattern fino al quel momento esaminati, ovvero P_1, per riutilizzare al massimo la sequenza iniziale di stati già esistenti in quel momento. Nell'esempio, tale prefisso corrisponde ad avere attraversato gli stati 0, 1, 2, e 3, che quindi vengono condivisi con gli altri pattern. È sufficiente pertanto allocare le righe per gli stati 7–11 in modo da sistemare il rimanente suffisso di P_2, ossia `cardo`.

I rimanenti due pattern P_3 e P_4 non condividono prefissi comuni (a parte quello vuoto che corrisponde allo stato 0), per cui allochiamo un nuovo stato per ciascun carattere.

Dalla costruzione segue che ogni stato creato nello scheletro corrisponde a un prefisso di uno dei pattern. Gli stati finali corrispondenti agli stati terminali (ossia gli interi pattern nelle



i	0	1	2	3	**4**	5	**6**	7	8	9	10	**11**	12	13	14	15	**16**	**17**	18	19	**20**
φ[i]	-1	0	18	19	20	3	0	0	1	0	0	0	0	1	2	3	4	5	0	1	2

FIGURA 2.4. In alto, rappresentazione grafica della tabella δ per lo scheletro dell'automa multiplo per i pattern $P_1 =$ `ananas`, $P_2 =$ `anacardo`, $P_3 =$ `banana` e $P_4 =$ `nan`. L'arco dal vertice i al vertice j con etichetta c indica che $\delta[i, c] = j$; gli archi con $\delta[i, c] =$ NIL non sono rappresentati. Gli stati finali 4, 6, 11, 16, 17 e 20 sono mostrati in neretto. In basso, la tabella φ per l'automa così ottenuto.

foglie 6, 11, 17 e 20) sono quindi marcati in neretto. Inoltre, poiché alcuni pattern possono occorrere all'interno di altri pattern, bisogna marcare come finali anche altri stati, come gli stati 4 e 16: l'uno corrisponde al prefisso `anan` e l'altro al prefisso `banan`, entrambi aventi P_4 come suffisso. A tal fine, manteniamo un vettore *finale* di booleani, tale che *finale*[i] = TRUE se e solo se lo stato i è finale (nel nostro caso, abbiamo 4, 6, 11, 16, 17 e 20 come stati finali).

Pseudocodice. L'algoritmo di costruzione dello scheletro, nelle linee 1–5 inizializza a NIL gli elementi della tabella δ e a FALSE gli elementi in *finale* (di fatto, crea lo stato 0). Prende quindi un pattern P_j alla volta (ciclo for alle linee 6–15) e cerca il suo più lungo prefisso in δ (linee 7–10), identificabile partendo dallo stato $i = 0$ e seguendo di volta in volta lo stato indicato da δ alla riga i e alla colonna $P_j[p]$. Non appena un valore NIL viene trovato e il pattern P_j ha ancora caratteri non esaminati, il codice crea gli stati necessari per memorizzare il resto del pattern (linee 11–14). Infine, marca lo stato finale così ottenuto (linea 15). Si noti che per adesso l'algoritmo non marca gli stati interni 4 e 16 come finali, in quanto tale compito sarà svolto successivamente dalla costruzione dell'altra tabella φ.

COSTRUZIONE ◇ SCHELETRO ◇ AHOCORASICK($P_1, \ldots, P_k$):

```
 1: ultimo_stato ← 0;
 2: FOREACH i ⩾ 0 DO
 3:    finale[i] ← FALSE;
 4: FOREACH i ⩾ 0 AND c ∈ Σ DO
 5:    δ[i, c] ← NIL;
 6: FOR j ← 1 TO k DO
 7:    i ← 0; p ← 1;
 8:    WHILE p ⩽ |P_j| AND δ[i, P_j[p]] ≠ NIL DO
 9:       i ← δ[i, P_j[p]];
10:       p ← p + 1;
11:    WHILE p ⩽ |P_j| DO
12:       ultimo_stato ← ultimo_stato + 1;
13:       δ[i, P_j[p]] ← ultimo_stato;
14:       i ← ultimo_stato; p ← p + 1;
15:    finale[i] ← TRUE;
```

16: RETURN tabelle δ, *finale*;

Dal nostro esempio, possiamo già evincere alcune proprietà dello scheletro (che poi incontremo nuovamente nei capitoli successivi con il nome di *trie*):

- lo stato iniziale corrisponde sempre alla riga 0;
- ciascun arco è etichettato con un carattere;
- se due archi partono dallo stesso vertice, allora sono etichettati con caratteri diversi;
- ogni stato ν corrisponde al prefisso comune di uno o più pattern: concatenando i caratteri lungo il cammino dallo stato 0 a ν otteniamo tale prefisso;
- se un pattern non appare all'interno degli altri pattern, corrisponde a un unico cammino dallo stato iniziale a uno terminale;
- se un pattern occorre in altri pattern, allora corrisponde anche a uno o più stati interni.

Illustriamo ora la seconda parte dell'algoritmo di Aho-Corasick, in cui calcoliamo una tabella φ analoga a quella calcolata con COSTRUZIONESEMPLICE$\diamond$KNUTHMORRISPRATT, per cui fissiamo il valore al contorno di $\varphi[0] = -1$.

Ricordiamo che lo scopo di φ è quello di memorizzare i bordi dei prefissi dei pattern. Per lo stato $\mathtt{s} > 0$ nello scheletro, sia $\mathtt{P_j}[1..\mathtt{p}]$ il prefisso associato a $\mathtt{s}$. Definiamo il *bordo* di $\mathtt{P_j}[1..\mathtt{p}]$ come il suo più lungo suffisso che è prefisso di almeno uno dei pattern (incluse $\mathtt{P_j}$ stesso). Lo stato che corrisponde a tale bordo esiste, in quanto ogni bordo è prefisso di uno o più pattern, per cui esso viene memorizzato in $\varphi[\mathtt{s}]$. Quindi il bordo è ben definito in questa estensione a più pattern perché per ogni prefisso esiste sempre un solo stato che lo rappresenta.

Esempio. Nel nostro esempio di Figura 2.4, con $\mathtt{s} = 7$, abbiamo un bordo vuoto per $\mathtt{P_j}[1..\mathtt{p}] = \mathtt{anac}$ e poniamo $\varphi[7] = 0$. Con $\mathtt{s} = 4$, il bordo di $\mathtt{P_j}[1..\mathtt{p}] = \mathtt{anan}$ è `nan` (non più `an`!) a cui corrisponde lo stato 20. Poniamo quindi $\varphi[4] = 20$.

Intendiamo usare la stessa tecnica di COSTRUZIONESEMPLICE$\diamond$KNUTHMORRISPRATT per trovare i valori di φ: osserviamo che, se il prefisso $\mathtt{P_j}[1..\mathtt{p}]$ è memorizzato nello stato $\mathtt{s}$, allora $\mathtt{P_j}[1..\mathtt{p}-1]$ è memorizzato nello stato $\mathtt{r}$ tale che $\delta\big[\mathtt{r}, \mathtt{P_j}[\mathtt{p}]\big] = \mathtt{s}$. Non è però detto che valga sempre la relazione $\mathtt{r} = \mathtt{s} - 1$ tra questi stati: ad esempio, per lo stato $\mathtt{s} = 4$ abbiamo $\mathtt{r} = 3$ mentre per $\mathtt{s} = 7$ abbiamo nuovamente $\mathtt{r} = 3$.

Mostriamo ora come calcolare $\varphi[4] = 20$ e $\varphi[7] = 0$. Per $\mathtt{s} = 4$, abbiamo $\mathtt{r} = 3$. Prendiamo il suo bordo `na` poichè $\varphi[3] = 19$ e vediamo che si estende con il carattere $\mathtt{P_j}[\mathtt{p}] = \mathtt{n}$ ottenendo `nan` che ha sede nello stato 20. Per $\mathtt{s} = 7$, abbiamo nuovamente $\mathtt{r} = 3$. Tuttavia, il bordo `na` rappresentato da $\varphi[3] = 19$ non si estende con $\mathtt{P_j}[\mathtt{p}] = \mathtt{c}$. Passiamo perciò a considerare il bordo `a`, rappresentato da $\varphi[19] = 1$, e anch'esso non si estende con $\mathtt{P_j}[\mathtt{p}] = \mathtt{c}$. Rimane da iterare con il bordo vuoto rappresentato da $\varphi[1] = 0$, ma anch'esso non si estende. Possiamo perciò concludere che il bordo di `anac` è vuoto per cui $\varphi[7] = 0$.

Pseudocodice. Da quanto discusso sopra, segue che il calcolo di $\varphi[\mathtt{s}]$ richiede di aver già calcolato i valori di φ per tutti gli stati la cui distanza dallo stato 0 è inferiore a quella dello stato $\mathtt{s}$. A tal fine l'algoritmo di costruzione di φ pone $\varphi[0] = -1$ e costruisce lo scheletro (linee 1–2). Poi esamina gli stati $\mathtt{r}$ per livelli (linee 3–11), dove lo stato 0 ha livello 0. In generale, lo stato $\mathtt{s} > 0$ ha livello $\mathtt{l} + 1$ se il suo predecessore $\mathtt{r}$ ha livello $\mathtt{l}$. I valori di φ vengono quindi fissati per livelli, partendo dal livello 1. In tal modo gli stati dei livelli precedenti hanno il valore φ corretto e possono contribuire ai valori per il livello corrente. A parte questa differenza concettuale, il calcolo di φ (linee 6–11) procede come in COSTRUZIONESEMPLICE$\diamond$KNUTHMORRISPRATT.

Le linee 12–14 si occupano invece di identificare ulteriori stati finali, quando alcuni dei pattern occorrono in altri pattern. Come si può notare in Figura 2.4, questi possono dare

luogo a più di uno stato finale. Fortunatamente, tali stati sono facilmente individuabili per un tale pattern P_j. Partendo dallo stato v che corrisponde a P_j, tutti gli stati finali s generati da P_j sono quelli da cui è possibile raggiungere v attraverso l'applicazione ripetuta di $e \geqslant 1$ volte di φ. In altri termini, denotando con $\varphi^e[s]$ lo stato raggiungibile a partire da s mediante e esecuzioni dell'istruzione $s \leftarrow \varphi[s]$, abbiamo che $\varphi^e[s] = v$ e *finale*$[v]$ = TRUE. Se infatti v corrisponde a P_j allora uno stato s_1, tale che $\varphi[s_1] = v$, corrisponde a una stringa che ha P_j come suffisso per definizione di bordo: quindi, s_1 è finale. Prendendo uno stato s_2 tale che $\varphi[s_2] = s_1$, possiamo osservare che anche la sua corrispondente stringa ha P_j come suffisso (in quanto s_1 è suffisso di s_2), per cui s_2 è finale, e inoltre vale $\varphi^2[s_2] = \varphi[s_1] = v$. L'argomentazione può essere estesa a s_3, s_4, ..., s_e: per induzione su e, possiamo quindi derivare che s è finale perché $\varphi^e[s] = v$.

COSTRUZIONESEMPLICE⋄AHOCORASICK(P_1, ..., P_k):

```
 1: φ[0] ← −1;
 2: costruisci lo scheletro per ottenere le tabelle δ e finale;
 3: FOREACH livello l >= 0 AND stato r a livello l AND carattere c tale che δ[r, c] ≠ NIL DO
 4:    s ← δ[r, c];
 5:    i ← φ[r];
 6:    WHILE i ⩾ 0 AND δ[i, c] = NIL DO
 7:       i ← φ[i];
 8:    IF i ⩾ 0 THEN
 9:       φ[s] ← δ[i, c];
10:    ELSE
11:       φ[s] ← 0;
12: FOREACH s ⩾ 0 DO
13:    IF esiste e ⩾ 1 tale che finale[φ^e[s]] = TRUE THEN
14:       finale[s] = TRUE
15: RETURN tabelle δ, φ, finale;
```

Complessità. Il costo totale della costruzione dell'automa è $O(M)$ se i pattern sono forniti in ordine lessicografica, altrimenti occorre aggiungere il costo del loro ordinamento mediante radix sort o altri algoritmi. Per la costruzione di φ, realizziamo COSTRUZIONESEMPLICE⋄AHOCORASICK mediante un'opportuna visita in ampiezza dello scheletro. La scoperta di ulteriori stati finali (linee 12–14) sfrutta implicitamente la struttura ad albero indotta da φ: infatti, $\varphi[s] = t$ equivale a dire che il padre dello stato s è lo stato t in questo albero implicito. La condizione se esiste $e \geqslant 1$ tale che *finale*$[\varphi^e[s]]$ = TRUE equivale a verificare se esiste un antenato v di s tale che *finale*$[v]$ = TRUE: tale calcolo può essere effettuato in tempo $O(M)$ per individuare tutti gli stati finali attraverso una sola visita di tale albero (infatti, se uno stato v è finale, lo diventano tutti i suoi discendenti). Infine, lo spazio richiesto è $O(M)$ perché in realtà è sufficiente mantenere solo i valori $\delta[r, c] \neq$ NIL, per cui δ forma un albero che ha radice nello stato 0.

Pseudocodice. A questo punto possiamo proporre l'algoritmo di ricerca, che è un'immediata estensione di RICERCASEMPLICE⋄KNUTHMORRISPRATT.

RICERCASEMPLICE⋄AHOCORASICK(P_1, ..., P_k, T):

```
1: M ← |P1| + ··· + |Pk|; n ← |T|;
2: costruisci le tabelle δ, φ e finale;
3: i ← 0;
4: FOR j ← 1 TO n DO
5:    WHILE i ⩾ 0 AND δ[i, T[j]] = NIL DO
6:       i ← φ[i];
```

```
 7:    IF i ⩾ 0 THEN
 8:       i ← δ[i, T[j]];
 9:    ELSE
10:       i ← 0;
11:    IF finale[i] THEN
12:       RETURN TRUE;    {occorrenza T[j − m + 1..j] del pattern P nello stato i, dove m = |P|}
13: RETURN FALSE;
```

Per restituire tutte le occorrenze a partire da un certo stato $\mathtt{i}$ finale, basta sostituire la riga 12 con un ciclo che procede a elencare tutti gli stati finali raggiungibili da $\mathtt{i}$ attraverso una o più applicazioni di φ:

```
f ← i;
WHILE f ⩾ 0 AND finale[f] DO    {occorrenza T[j − m + 1..j] del pattern P nello stato f}
  f ← φ[f];
```

Complessità. Durante la scansione, abbiamo necessità di calcolare il prossimo stato $\delta[\mathtt{i}, \mathtt{T}[\mathtt{j}]]$: tuttavia, avendo memorizzato δ come un albero i cui nodi possono avere fino a σ figli, il costo di $\delta[\mathtt{i}, \mathtt{T}[\mathtt{j}]]$ equivale a scegliere uno dei figli dello stato $\mathtt{i}$ in temp $O(\log \sigma)$ perché occorre memorizzarli in un array ordinato. Di conseguenza, ciascuno degli $O(n)$ passi della scansione richiede $O(\log \sigma)$ tempo più il numero di occorrenze riportate, a cui va aggiunto il costo $O(M)$ per la costruzione delle tabelle δ, φ e *finale*. L'argomento è analogo a RicercaSemplice$\diamond$KnuthMorrisPratt: durante la scansione, abbiamo una finestra che scorre su una posizione diversa a ogni iterazione di uno dei due cicli for e while. Segue dal fatto che ci sono al più n posizioni. Pertanto, l'algoritmo di Aho-Corasick richiede $O(M + n \log \sigma)$ tempo in totale più l'eventuale costo di ordinamento lessicografico dei pattern. Osserviamo infine che avere più testi da scandire non cambia la complessità del problema in quanto, una volta costruito l'automa di Aho-Corasick, esso può essere usato su molteplici testi, analogamente agli altri algoritmi discussi in questo capitolo.

CAPITOLO 3

Ricerca con Espressioni Regolari e Caratteri Speciali

In questo capitolo trattiamo un tipo di ricerca molto flessibile basata sulle espressioni regolari, introdotte da Kleene negli anni '50 nell'ambito della teoria degli automi e successivamente divenute l'asse portante di programmi di utilità come `grep` per Unix/Linux, di linguaggi come Perl e PHP, di server come Apache HTTP.

Tali espressioni sono usate per descrivere la sintassi di *Document Type Definition* nei formati HTML, SGML e XML molto diffusi per lo scambio di dati in Internet; per sfruttare certe regolarità in alcune porzioni di testo durante le ricerche: per esempio, una targa ha un formato `LL CCC LL` dove `L` rappresenta una delle lettere `A...Z` e `C` rappresenta una delle cifre `0...9` (quindi potenzialmente abbiamo $26^4 \times 10^3 = 456976000$ numeri di targa a disposizione).

Le espressioni regolari ci permettono di definire delle ricerche in cui è possibile formulare dei pattern più generali delle ricerche esatte. Tali pattern sono ottenuti tramite semplici regole di composizione per specificare sequenze di caratteri, concatenandole zero, una o più volte, o ponendole in alternativa. La composizione di tali regole permette di poter cercare, per esempio, le stringhe con alcune posizioni non specificate oppure le varianti ortografiche di una stessa parola.

In generale, un'espressione regolare serve a descrivere in maniera concisa un *insieme di stringhe* da cercare, eventualmente infinito e chiamato linguaggio definito dall'espressione regolare: in tale contesto, il termine pattern indica un insieme implicito di stringhe da cercare.

Anche se le espressioni regolari sono note da decenni e metodi di ricerca basati su di esse sono stati sviluppati, diversi utenti non ne sfruttano appieno le possibilità in quanto la loro formulazione appare ostica a prima vista. Il presente capitolo è un invito ad approfondire meglio questo tipo di ricerche che, se opportunamente utilizzate, permettono di ottenere risultati più selettivi rispetto alla ricerca esatta descritta nel Capitolo 2. In altre parole, le espressioni regolari non solo permettono di cercare pattern più generali, ma consentono di raffinare la ricerca di un pattern selezionando le occorrenze in base a delle regole che i simboli precedenti e successivi devono rispettare.

3.1. Definizione delle espressioni regolari

Introduciamo una notazione nata nel contesto della teoria dei linguaggi formali e adottata in seguito in diversi sistemi software di ricerca con estensioni e variazioni. Le espressioni regolari sono costruite mediante costanti e operatori e, come detto in precedenza, denotano insiemi (finiti e infiniti) di stringhe.

La definizione per un alfabeto Σ è ricorsiva, ovvero vengono definite prima le espressioni regolari più semplici e poi vengono date delle regole di composizione che permettono di costruire delle espressioni regolari più complesse a partire da quelle semplici, un po' come succede nelle espressioni aritmetiche. Nel seguito, continuiamo il nostro abuso di notazione per ϵ, la quale indica sia la stringa vuota che il carattere vuoto. Inoltre, assumiamo che i simboli speciali $|\ (\ )\ *\ \epsilon$, usati per definire le espressioni regolari, non fanno parte di Σ.

(1) Costanti: le espressioni regolari semplici sono i simboli dell'alfabeto Σ oppure ϵ. Ogni espressione regolare $c \in \Sigma$ rappresenta il simbolo c stesso, ossia l'insieme di stringhe $\{c\}$; per esempio, l'espressione regolare `a` rappresenta l'insieme $\{a\}$. L'espressione regolare ϵ rappresenta l'insieme vuoto di stringhe.
(2) Due espressioni regolari r_1 e r_2 possono essere composte attraverso due regole.
   (a) Alternativa: $(r_1|r_2)$ è l'espressione regolare di alternativa a cui corrispondono le stringhe rappresentate da r_1 oppure da r_2, quindi il corrispondente insieme è dato dall'unione degli insiemi di stringhe per r_1 e r_2. Per esempio, l'espressione `(a|b)` rappresenta l'insieme $\{a, b\}$.
   (b) Concatenazione: $(r_1)(r_2)$ è l'espressione regolare di giustapposizione a cui corrisponde l'insieme della stringhe xy ottenute concatenando ciascuna stringa x dell'insieme per r_1 con ciascuna stringa y dell'insieme per r_2. Per esempio, l'espressione `(a)(b)` rappresenta l'insieme $\{ab\}$, e `(a|b)(c|d)` rappresenta l'insieme $\{ac, ad, bc, bd\}$.
(3) Un'espressione regolare r può essere infine ottenuta con due ulteriori regole.
   (a) Stella di Kleene: r^* è l'espressione regolare ottenuta mediante la concatenazione con se stessa zero o più volte, a cui corrisponde l'insieme delle stringhe della forma $x_1x_2\cdots x_s$ per ogni intero $s \geqslant 0$. Se $s = 0$, allora la stringa risultante è ϵ. Altrimenti, ciascuna x_i è una qualsiasi stringa dell'insieme rappresentato da r e, quindi, le stringhe $x_1x_2\cdots x_s$, $s \leqslant 0$, rappresentano le possibili concatenazioni di stringhe rappresentate da r. In altre parole, l'insieme per r^* è chiuso rispetto all'operazione di concatenazione. Per esempio, l'espressione $(a)^*$ rappresenta l'insieme infinito di stringhe ϵ, `a`, `aa`, `aaa`, ..., mentre $(a|b)^*$ rappresenta ϵ, `a`, `b`, `aa`, `ab`, `ba`, `bb`, `aaa`, ..., `bbb`, e così via all'infinito. L'operatore $*$ viene chiamato stella di Kleene e rende infinita la cardinalità dell'insieme risultante.
   (b) Parentesi tonde: (r) è equivalente all'espressione regolare r, ossia entrambe rappresentano lo stesso insieme di stringhe.

Adottiamo la convenzione di omettere l'uso delle parentesi nelle espressioni regolari quando questa scelta non genera ambiguità di interpretazione: la stella di Kleene ha la priorità massima, seguita dalla concatenazione e poi dall'alternativa (analogamente alla precedenza dell'elevamento a potenza, della moltiplicazione e della somma nelle espressioni aritmetiche).

L'uso delle espressioni regolari nella pratica ha generato una sintassi più ricca che può essere comunque espressa nella notazione di base descritta sopra (tranne alcuni casi che discuteremo in seguito). Useremo Σ come abbreviazione della regola di alternativa tra tutti i simboli in Σ e useremo r^s come abbreviazione della concatenazione di r con se stessa per $s \geqslant 1$ volte.

Nella Figura 3.1 riportiamo parte della sintassi estesa definita dallo standard POSIX, basato sulla sintassi originariamente sviluppata per `grep`. La concatenazione e l'alternativa mantengono la stessa notazione, anche se l'alternativa tra singoli simboli può essere anche espressa con $[a_1a_2\cdots a_k]$. Esistono altre abbreviazioni come `[: alnum :]` per indicare i simboli alfanumerici `[A–Za–z0–9]` oppure `[: alpha :]` per indicare i soli caratteri `[A–Za–z]` e `[: digit :]` per indicare le sole cifre decimali `[0–9]`. È possibile inoltre imporre che l'occorrenza sia all'inizio o alla fine di una riga specificando il simbolo `^` o `$`, rispettivamente.

POSIX	esempio	notazione di base	esempio
.	.	Σ	(a|b|c|$\cdots$|z)
[$a_1a_2\cdots a_k$]	[acfq−t]	(a_1|a_2|$\cdots$|a_k)	(a|c|f|q|r|s|t)
[^$a_1a_2\cdots a_k$]	[^a−fhjm−z]	$\Sigma - (a_1\|a_2\|\cdots\|a_k)$	(g|i|k|l)
r^*	[^ad−y]*	r^*	(b|c|z)*
r+	[^ad−y]+	r^*r	(b|c|z)*(b|c|z)
r?	[^ad−y]?	(ϵ|r)	(ϵ|b|c|z)
r{i, j}	ab{2, 4}	($r^i\|r^{i+1}\|r^{i+2}\|\cdots\|r^j$)	(abab|ababab|abababab)

FIGURA 3.1. Esempi di sintassi POSIX e loro rappresentazione nella notazione di base dove, per scopi illustrativi, supponiamo che $\Sigma = \{a, b, c, \ldots, z\}$.

3.2. Uso delle espressioni regolari

Per comprendere la potenzialità delle espressioni regolari vediamo alcuni esempi. L'espressione [ch]?at|but corrisponde all'insieme {at, but, cat, hat}. Ricordando che l'espressione [01]+ è equivalente a (0|1)*(0|1), otteniamo l'insieme delle sequenze binarie non vuote. I termini aiuole e aiole possono essere rappresentati con l'espressione aiu?ole. Gli studiosi di papiri e di iscrizioni hanno accesso a raccolte elettroniche di trascrizioni in cui ogni carattere che risulta parzialmente o totalmente mancante come simbolo grafico viene rappresentato da più caratteri (le possibili interpretazioni di ciò che resta del simbolo grafico). In tale contesto, l'uso delle espressioni regolari facilita la ricerca in quanto permette di introdurre alternative e lacune nella specifica del pattern. In maniera simile ma applicandole sulle sequenze di DNA, i biologi usano le espressioni regolari senza la stella di Kleene, chiamandole *network expression*.

Nei linguaggi di programmazione, gli identificatori di variabili sono sequenze alfanumeriche di lunghezza massima in cui il primo carattere è una lettera. Possiamo usare le espressioni regolari per definirli come [a−z A−Z][a−z A−Z 0−9]*, ovvero una lettera in [a − z A − Z] seguita da una sequenza alfanumerica in [a − z A − Z 0 − 9]* possibilmente vuota.

Per identificare un generico indirizzo IP, per esempio 131.114.3.12, possiamo definire un'espressione di appoggio n = [0−9] | [1−9][0−9] | [1−9][0−9][0−9] | 2[0−4][0−9] | 25[0−5] per denotare un numero da 0 a 255 in decimale e usare r = n(\.n){3} per esprimere un generico numero IP. In tale espressione regolare, \. rappresenta il semplice carattere '.' e non il metacarattere riportato nella Figura 3.1 e il simbolo \ viene detto di *escape* in quanto permette di specificare i metacaratteri come semplici simboli, per esempio, \) o *; inoltre, x{3} è un'abbreviazione per x{3}{3}, ossia x concatenata tre volte.

Controintuitivamente, \< e \> non sono semplici simboli come < e >, anche se è presente il simbolo di *escape*, ma metacaratteri! Analogamente a ^ e $, \< e \> richiedono che l'occorrenza sia a inizio o fine parola. Imponendo che ogni testo sia implicitamente preceduto e terminato da un carattere fittizio non alfanumerico (per esempio, '\n' ossia il simbolo per andare a capo), possiamo tradurre \< e \> come [^ : alnum :] posto all'inizio o alla fine, rispettivamente, dell'espressione da cercare; in maniera analoga, possiamo sostituire ^ e $ con '\n' prima o dopo l'espressione regolare, rispettivamente. La sintassi POSIX è piuttosto ricca e abbiamo visto solo qualche esempio. Alcune abbreviazioni sono utili come '\w', che è equivalente a [[: alnum :]] e '\b' che può essere usato al posto di \< e \>. La sintassi estesa POSIX può essere usata, per esempio, con Perl oppure lanciando il comando grep -E.

Un generico indirizzo di posta elettronica nei domini .it, .com o .org, per esempio, può essere catturato mediante expressioni regolari. Possiamo scrivere la seguente espressione \b \w[-_.\w]* \@ [-_\w]+(\.[-_\w]+)*\.(it|com|org) \b, dove \w[-_.\w]* indica

lo *username* che inizia con un carattere alfanumerico e può usare trattini e puntini (notare che `-` appare come primo simbolo nelle parentesi quadre per essere interpretato come semplice carattere e che il simbolo `.` non ha bisogno di *escape* all'interno delle quadre), `\@` indica la chiocciola (senza *escape* è un metacarattere) e `[-_\w]+(\.[-_\w]+)*\.(it|com|org)` rappresenta l'*hostname* visto come concatenazione di sequenze alfanumeriche ed eventuali trattini, separate da punti e terminate con uno dei domini sopra. Possiamo individuare l'inizio di un indirizzo URL nei domini sopra, identificando il suo *hostname* con l'espressione `\bhttp://[-_\w]+(\.[-_\w]+)*\.(it|com|org)`. In questi esempi, rischiamo di catturare più del dovuto, per cui bisogna esplicitare ulteriormente le espressioni regolari: rimandiamo a testi specialistici per mostrare come usare appieno la sintassi POSIX.

Le espressioni regolari servono anche a specializzare la ricerca (esatta o meno) di una stringa pattern `P`. Possiamo infatti interpretare una tale ricerca equivalentemente come quella dell'espressione regolare $\mathtt{r} = \mathtt{.*\,P\,.*}$. Per determinare una condizione particolare sui caratteri che precedono e succedono un'occorrenza di `P`, possiamo specificare due espressioni regolari $\mathtt{r}_1$ e $\mathtt{r}_2$ che ne delimitano il contesto sinistro e destro all'interno del testo, utilizzando l'espressione $\mathtt{r}_1\,\mathtt{P}\,\mathtt{r}_2$ per ridurre il numero di occorrenze trovate.

Per esempio, nei testi in formato HTML, le porzioni di testo in neretto sono identificate dalla coppia di marcatori in maiuscolo `<B>` e `</B>` oppure in minuscolo `<b>` e `</b>` oppure una combinazione di maiuscolo e minuscolo. Il testo compreso tra i due marcatori viene mostrato in neretto e quindi degno di una certa rilevanza. Possiamo verificare se un documento contiene una parola in neretto, per esempio `P = vendesi`, specificando l'espressione `r = <[bB]>.*\bvendesi\b.*</[bB]>`: in tale espressione ammettiamo maiuscole `B` e minuscole `b` nei marcatori, e la parola `vendesi` (delimitata da inizio e fine parola `\b`) si trova racchiusa tra due sequenze arbitrarie `.*`, eventualmente vuote. Potremmo però catturare erroneamente un'occorrenza della forma `<b>avviso</b> vendesi casa in <b>Toscana</b>`. Per evitare ciò, possiamo sostituire ciascuna `.*` con l'espressione `.*(?<![bB]>|</[bB]>)` che indica qualunque sequenza, anche vuota, purché non contenga `<[bB]>` o `</[bB]>`: a tal fine, usiamo il costrutto $\mathtt{r}_1\mathtt{(?<!}\mathtt{r}_2\mathtt{)}$ per trovare $\mathtt{r}_1$ purchè al suo interno l'occorrenza non verifichi $\mathtt{r}_2$.

In realtà la valutazione delle espressioni regolari è *greedy*, nel senso che vengono "mangiati nel testo" il maggior numero di caratteri che soddisfano i metacaratteri come `*`. Volendo catturare le porzioni in neretto in un testo HTML, saremmo tentati di scrivere un'espressione come `r = <[bB]>.*</[bB]>`. Tuttavia, verrebbe catturata la *più grande* porzione di una riga che soddisfa tale espressione: `questo <b>esempio</b> illustra quanto <b>scritto</b> sopra`, restituendo `<b>esempio</b> illustra quanto <b>scritto</b>` come occorrenza. La soluzione più semplice è quella di richiedere che la valutazione dell'espressione segua la modalità *lazy* (ossia non *greedy*), in modo che venga scelta la *più corta* porzione di testo che soddisfa l'espressione. Nel nostro esempio, basta segnalare ciò aggiungendo il simbolo `?` dopo il metacarattere, ossia scrivendo `r = <[bB]>.*?</[bB]>`.

Nei motori di ricerca sono molto diffuse le ricerche di tipo booleano, che mostriamo essere dei casi speciali di espressioni regolari. Ricordiamo che una ricerca di tipo booleano usa termini connessi in AND, OR e NOT. Con due o più termini specificati in AND vogliamo trovare i documenti che contengono *tutti* i termini ivi specificati. Con la ricerca in OR intendiamo trovare i documenti che contengono *almeno uno* dei termini. Con la ricerca in NOT, vogliamo i documenti che *non* contengono i termini specificati dopo il NOT. In termini di espressioni regolari, per specificare due termini $\mathtt{t}_1$ e $\mathtt{t}_2$ in AND basta cercare $\mathtt{r} = \left(\mathtt{t}_1\,\mathtt{.*}\,\mathtt{t}_2 \;\middle|\; \mathtt{t}_2\,\mathtt{.*}\,\mathtt{t}_1\right)$; il connettivo OR è equivalente all'operatore | di alternativa; il complemento di un'espressione regolare è ancora un'espressione regolare, per cui in questo modo è possibile realizzare il NOT.

Osserviamo che le estensioni discusse finora semplificano la formulazione delle espressioni regolari ma possono essere sempre ottenute con la notazione di base. In altre parole, non cambia il potere espressivo del linguaggio rappresentato da tali varianti sintattiche delle espressioni regolari. Non è il caso dei *riferimenti all'indietro*, in cui le espressioni tra parentesi tonde sono numerate consecutivamente da sinistra a destra, a partire da 1, ed è possibile richiarmarle sucessivamente.

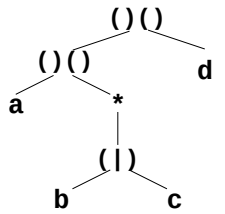
Per esempio, `(ab)a(c)\1` equivale a ripetere l'occorrenza nella prima coppia di parentesi (segnalato da `\i`, dove `i` $= 1$ è la `i`-esima coppia di parentesi scelta), riconoscendo `abacab` come occorrenza. Quest'estensione non è affatto innocua e permette di formulare `r` $=$ `(.+)\1`, ossia di riconoscere le occorrenze aventi la forma `xx` come `tata`, dove `x` è una stringa non vuota: tale linguaggio, chiamato dei quadrati, non può essere espresso mediante la notazione di base; anzi, non è neppure un linguaggio libero dal contesto (al contrario dei palindromi $x\hat{x}$ come `abba`, dove $\hat{x}$ è la stringa speculare a `x`).

È stato pertanto coniato il termine `regexp` per indicare questa classe estesa che include strettamente quella delle espressioni regolari. In diverse realizzazioni pratiche, sono ammessi un numero massimo di riferimenti all'indietro e si usano tecniche di *backtracking* che sono esponenziali al caso pessimo, ma sono ottimizzate per alcune configurazioni in pratica. Notare che il problema del riconoscimento delle `regexp` con un numero arbitrario di riferimenti all'indietro, diventa computazionalmente più arduo da trattare (NP-hard).

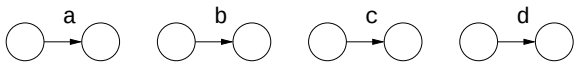
3.3. Costruzione dell'automa finito non deterministico

La ricerca delle espressioni regolari richiede la costruzione di un automa finito, deterministico o non. Per semplificare la discussione, mostriamo come costruire un automa non deterministico. La costruzione segue fedelmente la definizione ricorsiva delle espressioni regolari. Il caso base è dato dalle espressioni semplici, ovvero i caratteri dell'alfabeto Σ o il simbolo ϵ come riportato nel punto 1 della Sezione 3.1. Il passo induttivo è dato dalla composizione delle espressioni regolari secondo quanto descritto nei successivi punti 2 e 3.

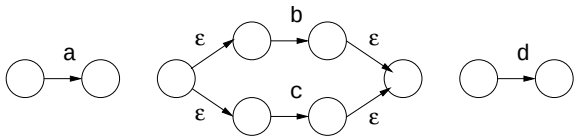
Esempio. Supponiamo di voler costruire l'automa per l'espressione regolare `r` $=$ `a(b|c)*d`. Vediamo come ottenere `r` per costruire quindi l'automa corrispondente, immaginando l'ordine di visita posticipata dell'albero sintattico per `r`: prima riconosciamo `a`, `b`, `c` e `d` come espressioni regolari semplici; poi componiamo `b` e `c` ottenendo (`b|c`); su tale espressione applichiamo la stella di Kleene (`b|c`)*; infine, concateniamo il risultato con `a` e `d` per ottenere `r`.



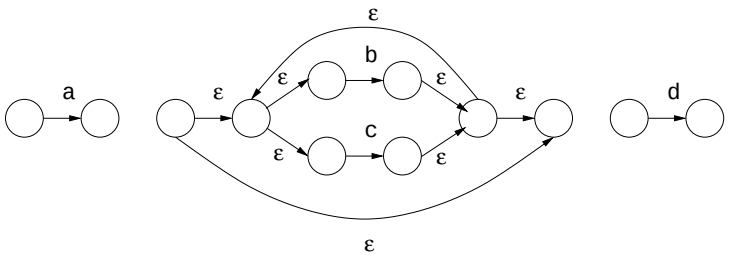
Procediamo alla stessa maniera per la costruzione dell'automa corrispondente a `r`. Come base di partenza, costruiamo gli automi per le espressioni semplici `a`, `b`, `c` e `d`:



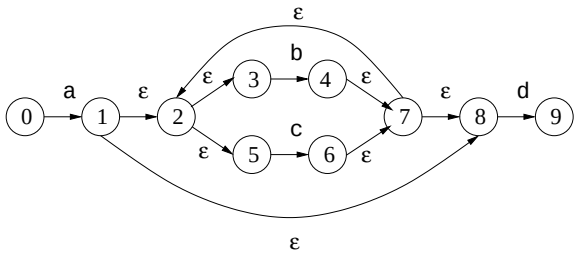
Ciascuno di questi automi ha un proprio stato iniziale e uno finale. Quando andiamo a comporre gli automi per `b` e `c` così da ottenere l'alternativa (`b|c`), dobbiamo porre i due automi in parallelo, creando un nuovo stato iniziale e un nuovo stato finale. Il nuovo stato iniziale va collegato agli stati iniziali dei due automi. Il nuovo stato finale è collegato alle uscite degli stati finali dei due automi. L'automa risultante corrisponde all'espressione (`b|c`), che ha un solo stato iniziale e uno finale:



Possiamo adesso applicare la stella di Kleene, collegando lo stato finale dell'automa per (b|c) al suo stato iniziale. Inoltre, creiamo un nuovo stato iniziale e un nuovo stato finale. Il nuovo stato iniziale va collegato sia allo stato iniziale dell'automa che al nuovo stato finale. Lo stato finale dell'automa è collegato in uscita al nuovo stato finale:



Possiamo finalmente concatenare i tre automi finora ottenuti, mettendoli in sequenza così da richiedere il loro attraversamento in serie. Lo stato finale di ogni automa, tranne l'ultimo, viene fatto coincidere con lo stato iniziale dell'automa successivo. Ancora una volta, l'automa risultante ha un solo stato iniziale 0 e un solo stato finale 9:



Nella definizione ricorsiva delle espressioni regolari, il caso base per i caratteri $\mathtt{c} \in \Sigma$ (ossia le foglie nell'albero sintattico) genera gli archi per le espressioni semplici `a`, `b`, `c` e `d`. Gli archi aggiunti durante i passi induttivi (ossia i nodi interni dell'albero sintattico) sono tutti etichettati con ϵ e sono chiamati ϵ-*transizioni*. Le ϵ-transizioni sono introdotte perché possano venire attraversate *senza* dover leggere alcun carattere. In altre parole, se uno stato `s` è attivo durante il riconoscimento di stringhe con l'automa, allora lo sono tutti gli stati raggiungibili a partire da `s` con una o più ϵ-transizioni. Per esempio, se lo stato 7 dell'automa per `r` è attivo, lo sono anche gli stati 2, 3, 5 e 8 in quanto raggiungibili dallo stato 7 mediante ϵ-transizioni.

Notare che il non determinismo dell'automa risultante è dovuto alla presenza delle ϵ-transizioni, perché gli archi etichettati con caratteri di Σ non danno luogo a non determinismo. Inoltre, avremmo potuto usare un numero inferiore di stati: tuttavia, la descrizione informale della costruzione appena vista è di facile realizzazione seguendo le notazioni di base.

Dal nostro esempio illustrativo possiamo dedurre alcune proprietà dell'automa risultante:

- esistono un solo stato iniziale e un solo stato finale;
- ogni stato ha un solo arco uscente etichettato con un carattere $\mathtt{c} \in \Sigma$, oppure ha al più due ϵ-transizioni in uscita: quindi la transizione con $\mathtt{c} \in \Sigma$ richiede $O(1)$ tempo;
- il numero di stati è limitato superiormente da $2|\mathtt{r}|$, dove $|\mathtt{r}|$ è la lunghezza dell'espressione `r` calcolata come il numero di costanti e operatori che appaiono in `r`.

Tali proprietà possono essere dimostrate per induzione strutturale seguendo lo schema ricorsivo di costruzione delle espressioni regolari. Per esempio, la limitazione superiore di $2|\mathtt{r}|$ al numero di stati deriva dal seguente ragionamento. Ciascun carattere $\mathtt{c}$ dà luogo a due stati nel caso base. Durante la composizione, il numero di stati viene incrementato di due unità solo in corrispondenza dei simboli $|$ e $*$. Negli altri casi, rimane lo stesso o addirittura diminuisce di una unità nel caso della concatenazione in serie. Di conseguenza, il numero di stati non può eccedere $2|\mathtt{r}|$.

Pseudocodice. La costruzione dell'automa finito non deterministico segue quindi la definizione ricorsiva dell'espressione regolare $\mathtt{r}$:

COSTRUZIONE$\diamond$AUTOMAFINITONONDETERMINISTICO($\mathtt{r}$):

(1) Caso base $\mathtt{r} \equiv \mathtt{c}$ per un carattere $\mathtt{c} \in \Sigma$ oppure $\mathtt{r} \equiv \epsilon$: restituisci l'automa composto da due stati con una freccia etichettata $\mathtt{c}$ (oppure ϵ) dallo stato iniziale allo stato finale.
(2) Caso induttivo in cui $\mathtt{r}$ è ottenuta dalla composizione di due espressioni regolari $\mathtt{r}_1$ e $\mathtt{r}_2$. Chiama ricorsivamente COSTRUZIONE$\diamond$AUTOMAFINITONONDETERMINISTICO su $\mathtt{r}_1$ e $\mathtt{r}_2$ per ottenere i corrispondenti automi δ_1 e δ_2.
   (a) Caso $\mathtt{r} \equiv (\mathtt{r}_1|\mathtt{r}_2)$: crea un nuovo stato iniziale e un nuovo stato finale per mettere δ_1 e δ_2 in parallelo. Il nuovo stato iniziale va collegato agli stati iniziali di δ_1 e δ_2; i loro stati finali sono collegati al nuovo stato finale. I collegamenti sono tutti effettuati tramite ϵ-transizioni.
   (b) Caso $\mathtt{r} \equiv (\mathtt{r}_1)(\mathtt{r}_2)$: metti δ_1 e δ_2 in serie facendo coincidere lo stato finale di δ_1 con lo stato iniziale di δ_2. Ne risulta che lo stato inziale è quello di δ_1 e quello finale è quello di δ_2.
(3) Caso induttivo in cui $\mathtt{r}$ è ottenuta da un'espressione regolare $\mathtt{r}_1$. Chiama ricorsivamente COSTRUZIONE$\diamond$AUTOMAFINITONONDETERMINISTICO($\mathtt{r}_1$) per ottenere il corrispondente automa δ_1.
   (a) Caso $\mathtt{r} \equiv (\mathtt{r}_1)^*$. Poni una ϵ-transizione dallo stato finale di δ_1 al suo stato iniziale. Crea un nuovo stato iniziale e un nuovo stato finale. Poni due ϵ-transizioni dal nuovo stato iniziale, una diretta al nuovo stato finale e l'altra allo stato iniziale di δ_1. Poni una ϵ-transizione dallo stato finale di δ_1 al nuovo stato finale.
   (b) Caso $\mathtt{r} \equiv (\mathtt{r}_1)$. Restituisci δ_1 senza modifiche.

L'implementazione immediata dell'approccio sopra è quello di costruire esplicitamente l'automa. Esistono metodi di ricerca molto più efficienti che invece generano direttamente il codice da eseguire dalle regole ricorsive esposte sopra: si parla infatti di compilazione di espressioni regolari proprio perché dall'espressione $\mathtt{r}$ otteniamo direttamente il codice da compilare invece che un automa non deterministico da simulare.

Complessità. Sia $|\mathtt{r}|$ la lunghezza dell'espressione regolare $\mathtt{r}$. L'algoritmo di costruzione dell'automa esegue $O(|\mathtt{r}|)$ passi di costruzione di automi semplici e $O(|\mathtt{r}|)$ passi di loro successiva composizione. Ogni passo richiede tempo costante per la creazione dei nuovi stati e per il collegamento degli stati iniziali/finali di al più due automi. Complessivamente, il costo computazionale è di $O(|\mathtt{r}|)$ tempo e spazio.

3.4. Ricerca di espressioni regolari con l'automa

Vediamo come utilizzare l'automa non deterministico δ per la ricerca di un'espressione regolare $\mathtt{r}$ di lunghezza $\mathtt{m}$ in un testo $\mathtt{T}$ di lunghezza $\mathtt{n}$. A tal fine, dobbiamo aggiungere una freccia dallo stato iniziale di δ a se stesso etichettata con tutti i simboli di Σ, in quanto vogliamo

che lo stato iniziale sia sempre attivo. Questa semplice estensione permette di verificare se esiste una sottostringa di T che corrisponde all'espressione regolare r.

Esempio. Consideriamo l'automa δ per l'espressione $r = a(b|c)^*d$, in cui aggiungiamo una transizione dallo stato iniziale verso se stesso etichettata con Σ. Simuliamo il comportamento dell'automa in corrispondenza del testo $T = aacbcd$. Manteniamo un insieme Q di stati attivi, partendo con Q che contiene il solo stato iniziale 0 (in quanto non ci sono ϵ-transizioni in uscita da esso). Dopo aver letto il primo carattere di T, l'insieme ausiliario Q' contiene gli stati raggiungibili da quelli in Q tramite una transizione etichettata con a, ottenendo $Q' = \{0, 1\}$. A questo punto aggiorniamo Q con tutti gli stati che sono raggiungibili da Q' attraverso zero o più ϵ-transizioni. Nel nostro caso, $Q = \{0, 1, 2, 3, 5, 8\}$. Iteriamo il procedimento con la lettura del successivo carattere di T, che risulta essere di nuovo a, per cui gli insiemi Q' e Q non cambiano. Leggendo c, abbiamo che $Q' = \{0, 6\}$ e $Q = \{0, 2, 3, 5, 6, 7, 8\}$. Leggendo quindi b, abbiamo che $Q' = \{0, 4\}$ e $Q = \{0, 2, 3, 4, 5, 7, 8\}$. Il successivo carattere c fornisce i precedenti valori di $Q' = \{0, 6\}$ e $Q = \{0, 2, 3, 5, 6, 7, 8\}$. Infine, la lettura di d fornisce $Q = Q' = \{0, 9\}$, raggiungendo lo stato finale 9. Possiamo concludere che nella posizione corrente del testo T termina un'occorrenza di r.

Pseudocodice. L'algoritmo che ci accingiamo a descrivere segue la prima versione del programma di utilità `grep` sviluppato da Thompson. Consiste nella simulazione di un automa non deterministico δ per l'espressione regolare r, assumendo che gli stati siano numerati a partire da 0 (lo stato iniziale).

Inizialmente l'unico stato attivo di δ è quello iniziale e tutti gli stati da esso raggiungibili con zero o più ϵ-transizioni, memorizzati in un insieme Q. Nella generica iterazione, Q contiene tutti gli stati attivi in δ. La transizione di (molteplici) stati dell'automa in base al prossimo carattere T[j] letto nel testo avviene in due passi. Ricordiamo che l'automa non deterministico δ è tale per cui ci sono $\left|\delta\left[s, T[j]\right]\right| \leqslant 1$ stati oppure $\left|\delta[s, \epsilon]\right| \leqslant 2$ stati.

(1) Per ciascuno stato $s \in Q$ prendiamo il suo prossimo stato $\delta\left[s, T[j]\right]$. In tal modo otteniamo un nuovo insieme di stati attivi Q' e viene reso vuoto Q.
(2) Gli stati di Q' attivano tutti gli stati raggiungibili attraverso ϵ-transizioni: se uno stato s' è raggiungibile attraverso zero o più ϵ-transizioni a partire da uno stato $s \in Q'$, allora s' viene aggiunto a Q. (Notare che $Q' \subseteq Q$ per definizione.)

La propagazione dagli stati in Q' a tutti gli stati in Q avviene attraverso l'esecuzione di EpsilonIngenua, descritta sotto: essa considera implicitamente il grafo diretto i cui vertici sono gli stati dell'automa δ e i cui archi sono le *sole* ϵ-transizioni; effettua quindi una visita di tale grafo sparso, per esempio attraverso una visita in ampiezza $BFS(s, \delta)$ a partire dal vertice che corrisponde allo stato s di δ. Ipotizziamo che $BFS(s, \delta)$ restituisca l'insieme degli stati raggiungibili da s (incluso).

EpsilonIngenua(Q', δ):

```
1: Q'' ← {};    {insieme vuoto}
2: FOREACH s ∈ Q' DO
3:     Q'' ← Q'' ∪ BFS(s, δ);
4: RETURN Q'';
```

Tecnicamente, EpsilonIngenua calcola quella che si chiama ϵ-*chiusura* di Q' in δ. L'algoritmo può essere inefficiente se attraversa più volte gli stessi archi: teoricamente, al caso pessimo, il tempo di esecuzione può essere quadratico nel numero di stati di δ perché EpsilonIngenua effettua tante visite del grafo quanti sono gli stati in Q' (e Q' può contenere una frazione costante degli stati di δ).

Tuttavia, una semplice modifica permette all'algoritmo di ϵ-chiusura di richiedere solo tempo lineare, sfruttando il fatto che il numero di archi è linearmente proporzionale al numero di stati. Normalmente, siamo abituati a implementare la visita $BFS(s, \delta)$ usando una coda che viene inizializzata con s: è pertanto sufficiente estendere tale visita mettendo nella coda *tutti* gli stati in Q' prima della partenza della visita. L'effetto della visita risultante è quindi quello di restituire tutti gli stati raggiungibili a partire da quelli in Q' (inclusi).

In altre parole, abbiamo che la visita adesso richiede tempo lineare nel numero di stati dell'automa δ perché ogni arco viene attraversato una sola volta e ogni vertice u appare nella coda tante volte quanti sono gli archi entranti in u, più un'ulteriore volta se $u \in Q'$. Ricordiamo che il numero degli archi cresce linearmente con il numero di stati, per concludere l'analisi del costo lineare al caso pessimo.

Il seguente algoritmo implementa tale schema efficiente, utilizzando un vettore booleano `attivo` e una coda C che fornisce le operazioni *enqueue* (metti in coda) e *dequeue* (estrai dalla coda), ciascuna in tempo costante.

EPSILON(Q', δ):

```
 1: FOREACH stato u in δ DO
 2:    attivo[u] ← FALSE;
 3: C ← {}, Q'' ← {};    {coda vuota e insieme degli stati vuoto}
 4: FOREACH s ∈ Q' DO
 5:    C ← enqueue(C, s);
 6: WHILE C ≠ {} DO
 7:    u ← dequeue(C);
 8:    IF NOT attivo[u] THEN
 9:       attivo[u] = TRUE;
10:       Q'' ← Q'' ∪ {u};
11:       FOREACH v ∈ δ[u, ε] DO
12:          C ← enqueue(C, v);
13: RETURN Q'';
```

Possiamo adesso impiegare l'algoritmo EPSILON nella ricerca di un'espressione regolare r nel testo T.

RICERCA⋄AUTOMAFINITONONDETERMINISTICO(r, T):

```
 1: m ← |r|; n ← |T|;
 2: costruisci l'automa finito non deterministico δ per r;
 3: Q ← {0};    {stato iniziale 0 dell'automa}
 4: Q ← EPSILON(Q, δ);
 5: IF Q contiene lo stato finale THEN
 6:    RETURN TRUE;
 7: FOR j ← 1 TO n DO
 8:    Q' ← {};    {insieme vuoto}
 9:    FOREACH s ∈ Q DO
10:       Q' ← Q' ∪ δ[s, T[j]];
11:    Q ← EPSILON(Q', δ);
12:    IF Q contiene stato finale THEN
13:       RETURN TRUE;
14: RETURN FALSE;
```

Dopo aver costruito l'automa δ e aver inizializzato le variabili e l'insieme Q con i dovuti controlli (linee 1–6), l'algoritmo RICERCA⋄AUTOMAFINITONONDETERMINISTICO procede a

eseguire n iterazioni (linee 7–13), dove l'iterazione j è divisa in due passi, come discusso precedentemente. Nel primo passo (linee 8–10), essa effettua le transizioni multiple a partire dagli stati in Q con il carattere $T[j]$, ottenendo l'insieme ausiliario Q'. Nel secondo passo (linea 11), essa propaga gli stati attivi in Q' mediante le ϵ-transizioni, aggiornando Q. Infine, verifica se lo stato finale è stato raggiunto (linee 12–13). Se ciò non accade al termine di tutte le iterazioni, conclude che l'espressione r non appare in T (linea 14).

La prima versione di `grep` è praticamente una realizzazione molto efficiente di tali idee. Successivamente la simulazione dell'automa è stata effettuata diversamente, ossia generando direttamente del codice da compilare durante la costruzione dell'automa. In questo modo, sono state migliorate le prestazioni in pratica anche se la complessità teorica rimane la stessa. Nel Capitolo 5 vedremo un modo alternativo per realizzare efficientemente la ricerca con espressioni regolari usando operazioni logiche per manipolare i bit delle parole di memoria.

Complessità. L'algoritmo RICERCA◇AUTOMAFINITONONDETERMINISTICO può essere eseguito in $O(mn)$ tempo al caso pessimo, dove $m = |r|$ è la lunghezza dell'espressione regolare r. Infatti, abbiamo n passi di simulazione e ciascun passo richiede $O(m)$ tempo, in quanto è dominato dal costo dell'algoritmo EPSILON.

3.5. Considerazioni finali

La ricerca di un'espressione regolare può essere effettuata, in alternativa, con un automa deterministico. Nella teoria dei linguaggi formali, le espressioni regolari, gli automi finiti non deterministici e gli automi finiti deterministici rappresentano lo stesso linguaggio. Ovvero, per ogni espressione regolare esiste un automa (deterministico e non) che riconosce il suo linguaggio e viceversa: tale importante risultato è noto in letteratura come Teorema di Kleene, 1956.

In tale contesto, esiste una tecnica generale di trasformazione da automi non deterministici ad automi deterministici che può essere applicata all'automa δ. L'algoritmo di ricerca diventa quindi la scansione (in tempo reale) del testo T mediante automa, in tempo lineare, analogamente alla procedura RICERCA◇AUTOMA discussa nel Capitolo 2. Sfortunatamente, l'automa deterministico può avere un numero esponenziale di stati al caso pessimo. Per esempio, l'espressione regolare $(a|b)^*a(a|b)\cdots(a|b)$, in cui appaiono k espressioni $(a|b)$ alla sua destra, dove $k < m$ e $k = \Theta(m)$, rappresenta una qualunque stringa che abbia il simbolo a a $k+1$ posizioni di distanza dalla fine. Il corrispondente automa deterministico richiede $\Omega(2^k)$ stati, mentre per l'automa non deterministico ne bastano $O(m)$. Infatti, l'algoritmo di ricerca con automa deterministico impiega $O(2^m + n)$ tempo al caso pessimo e richiede spazio $O(2^m)$.

Esiste un'elegante costruzione diretta di un automa deterministico a partire da un'espressione regolare, senza passare per l'automa non deterministico. Partendo dall'osservazione che non tutti gli stati sono utilizzati durante la ricerca in un dato testo, questa costruzione può essere realizzata in maniera "pigra". Durante la ricerca, gli stati dell'automa vengono costruiti man mano che sono raggiunti a partire dagli stati generati fino a quel momento. Vengono così creati e usati solo una parte dei possibili stati. Tale idea è realizzata in modo efficiente in `egrep` mediante l'uso di un'area temporanea di appoggio per gli stati (cache). Anche se il caso pessimo non migliora, il comportamento pratico è buono sia in termini di tempo che in termini di spazio.

La storia di `grep`/`egrep` non è finita. Tra i vari trucchi per rendere veloci questi algoritmi in pratica, va menzionato quello di cercare nei rispettivi automi una sequenza massimale di stati senza ϵ-transizioni, con la proprietà che tale sequenza è un percorso obbligato per raggiungere lo stato finale. Nel caso che tale sequenza esista, gli stati nella sequenza hanno una sola transizione etichettata con un simbolo di Σ. Pertanto, la concatenazione ordinata di

tali simboli fornisce una stringa $\mathtt{P}$ e, visto che la sequenza rappresenta un percorso obbligato, ogni occorrenza dell'espressione regolare $\mathtt{r}$ deve contenere $\mathtt{P}$. L'idea è di impiegare come filtro la ricerca esatta di $\mathtt{P}$ attraverso algoritmi veloci di tipo Boyer-Moore-Gosper-Horspool. Le posizioni del testo che non contengono $\mathtt{P}$ possono essere quindi ignorate: le rimanenti indicano che l'espressione regolare $\mathtt{r}$ potrebbe apparire. Per ogni occorrenza di $\mathtt{P}$, viene quindi scelto un intorno sufficientemente lungo di posizioni adiacenti, esaminando i corrispondenti caratteri con l'automa per $\mathtt{r}$. In questa maniera l'algoritmo identifica tutte le occorrenze di $\mathtt{r}$ utilizzando le occorrenze di $\mathtt{P}$ come base di partenza e può guadagnare un fattore di almeno 10 in velocità, perché riesce a scartare buona parte del testo.

Un altro trucco che funziona bene in pratica è quello di abbandonare la simulazione dell'automa non deterministico a favore di un *parser* (o analizzatore sintattico) a discesa ricorsiva in grado di riconoscere il linguaggio delle `regexp` (che tra l'altro non può essere riconosciuto dagli automi). Poiché il tempo d'esecuzione può essere esponenziale, in tale *parser* vengono applicate delle euristiche di taglio oltre a quella basata su Boyer-Moore-Gosper-Horspool descritta sopra. Riferendoci all'albero sintattico mostrato all'inizio del Capitolo 3.3, il *parser* procede in modalità di *backtracking* in concomitanza dei nodi che corrispondono all'alternativa e alla stella di Kleene, in quanto quest'ultime generano più scelte possibili a causa delle ϵ-transizioni: tali scelte vengono esplorate via via con il *backtracking* a cui si applicano dei tagli per limitare (ma non eliminare!) l'esplosione combinatoria dei casi da esaminare. È possibile generare delle istanze in cui la simulazione efficiente dell'automa va enormemente meglio di quella del parser con *backtracking*, per cui non è tuttora chiaro quale sia il modo ottimo per cercare le espressioni regolari in un testo.

Discutiamo infine un caso ristretto di espressione regolare molto utile nella pratica, chiamato *pattern con don't care*, ad esempio come `act..t.cagc....ctc`. Un pattern $\mathtt{r}$ di questo tipo è composto da caratteri in Σ e da metacaratteri '`.`', chiamati *don't care*: può essere quindi visto come una sequenza alternata di stringhe massimali e non vuote di simboli in Σ^* e di *don't care*, ossia $\mathtt{r} = \mathtt{p}_1 .^{\ell_1} \mathtt{p}_2 .^{\ell_2} \cdots \mathtt{p}_k$ (nel nostro esempio, $\mathtt{p}_1 = \mathtt{act}$, $\mathtt{p}_2 = \mathtt{t}$, $\mathtt{p}_3 = \mathtt{cagc}$ e $\mathtt{p}_4 = \mathtt{ctc}$).

Possiamo impiegare l'algoritmo RicercaSemplice$\diamond$AhoCorasick con le stringhe $\mathtt{p}_1$, $\mathtt{p}_2$, ..., $\mathtt{p}_k$ e un array C di contatori, per cercare le occorrenze di $\mathtt{r}$ in un testo $\mathtt{T}$. Calcoliamo, per ciascun $\mathtt{p}_i$, il numero $\mathtt{n}_i$ di caratteri che lo precedono (incluso i *don't care*) e inizializziamo a zero tutti gli elementi dell'array $\mathtt{C}$. Eseguiamo poi la ricerca delle stringhe $\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_k$: quando troviamo un'occorrenza di una stringa $\mathtt{p}_i$ in posizione $\mathtt{j}$ del testo, incrementiamo di 1 il contatore $\mathtt{C}[\mathtt{j} - \mathtt{n}_i]$. Alla fine di questa scansione, possiamo determinare se $\mathtt{r}$ occorre in posizione $\mathtt{j}$ verificando che $\mathtt{C}[\mathtt{j}] = \mathtt{k}$: infatti vuol dire che, a partire dalla posizione $\mathtt{j}$, tutte le stringhe $\mathtt{p}_i$ occorrono, ciascuna alla propria distanza $\mathtt{n}_i$ da $\mathtt{j}$. È possibile mantenere $\mathtt{C}$ di dimensione $O(\mathtt{m})$ invece che $O(\mathtt{n})$, dove $\mathtt{m} = |\mathtt{r}|$. Il costo dell'algoritmo è pari a $O(\mathtt{n}\mathtt{k})$ tempo, al caso pessimo, e risulta essere migliore del costo di simulazione dell'automa non deterministico per l'espressione $\mathtt{r}$ quando $\mathtt{k} \ll \mathtt{m}$.

CAPITOLO 4

Ricerca con Errori e Confronto tra Sequenze

In questo capitolo presentiamo alcuni algoritmi di ricerca con errori basati sul confronto tra sequenze. Gli errori si riferiscono ovviamente al testo `T` da ricercare (e non agli algoritmi!) e possono essere di varia natura, rendendo inefficaci gli algoritmi di ricerca esatta (Capitolo 2). Esistono diverse situazioni in cui, pur specificando una stringa pattern `P` di ricerca, non abbiamo garanzie che `P` appaia esattamente nel testo `T`. L'introduzione di errori nel testo `T` può essere dovuta a errori umani di battitura durante l'immissione dei testi. La qualità dei dati disponibili nel Web e in Internet è bassa in quanto ogni persona con accesso alla rete può potenzialmente pubblicare un proprio testo elettronico. Non essendoci un "comitato di redazione" del Web per controllare la qualità tipografica e stilistica di tali testi, è impensabile uniformarli a un qualche standard lessicale, grammatico e stilistico. Le parole in tali testi rischiano di non poter essere trovate altrimenti con i motori di ricerca. Un problema analogo emerge anche nei testi datati che vengono riportati in versione elettronica, spesso usando termini ormai decaduti.

Esistono tuttavia delle situazioni in cui gli errori sono inevitabili. I testi che hanno diversi contesti linguistici possono dar luogo a forme eterogenee che vanno trattate: per esempio, si pensi alle forme lessicali diverse per le medesime parole nella lingua inglese e americana. Se poi i dati sono trasmessi attraverso canali di trasmissione non perfetti, possono venire introdotti errori di trasmissione. Anche i testi prodotti da strumenti di misurazione e di riconoscimento hanno implicito un errore di misura. Nel riconoscimento della voce, si vuole trasformare un messaggio verbale ottenuto da un segnale audio in un messaggio testuale o, più semplicemente, riconoscere una parola udita tra quelle memorizzate in un opportuno dizionario. Nel riconoscimento della scrittura, un messaggio grafico deve essere trasformato in un messaggio testuale attraverso il riconoscimento automatico dei caratteri. Analoghe situazioni si hanno nei verificatori ortografici, nelle interfacce uomo-macchina in linguaggio naturale e nell'apprendimento automatico delle lingue. Quando i testi sono di natura biologica, come le sequenze di DNA, le mutazioni dovute all'evoluzione molecolare rendono inefficaci le tecniche di ricerca esatta in quanto una porzione di sequenza può occorrere più volte ma variata in seguito ai mutamenti.

La possibilità di ammettere errori e imprecisioni non solo allevia i problemi di ricerca appena descritti, ma permette una certa flessibilità nella specifica (si pensi all'opzione *did you mean?* di Google). Quante volte non ricordiamo esattamente come si scrive un nome straniero di luogo o di persona? Spesso alcuni di questi nomi ammettono traslitterazioni diverse (in alcune lingue, le vocali non sono sempre scritte). Le tecniche alla base della ricerca con errori,

chiamata anche *ricerca approssimata*, trovano utile applicazione anche in altri contesti. Per esempio, avendo più versioni modificate dello stesso testo, vogliamo ricostruire le modifiche apportate al testo originario attraverso le varie versioni. Oppure, usando le parole come simboli indivisibili dell'alfabeto, possiamo vedere una frase come una stringa di parole. Ammettere errori di ricerca su tali simboli/parole equivale a cercare una frase che contiene più o meno le parole specificate (e possibilmente un concetto simile), eventualmente variando l'ordine delle parole. Tecniche simili sono adottate anche per il rilevamento della presenza di virus o di intrusioni nei sistemi informatici perché caratterizzati da certe sequenze di operazioni mescolate ad altre.

Tipicamente la nozione di errore è legata a una qualche nozione di distanza tra le stringhe, presa in prestito dalla teoria dei codici a correzione di errore: maggiore è la distanza, maggiore è l'errore. Dopo aver introdotto alcune di nozioni di distanza note in letteratura, ci concentriamo nel resto del capitolo sulla distanza di edit e il suo calcolo efficiente, collegandola alla nozione di sottosequenza comune più lunga e alla ricerca con al più $\mathtt{k}$ errori. Tale ricerca può essere vista come un caso speciale di ricerca con espressioni regolari, per cui mostriamo anche un automa non deterministico a stati finiti in grado di eseguirla.

4.1. Definizione di errore e di distanza

La definizione di errore può essere formulata matematicamente collegandola al concetto di *distanza* tra stringhe. Una distanza $\mathtt{d}$ è una funzione definita su coppie di stringhe dell'alfabeto Σ e restituisce un reale non negativo, ossia $\mathtt{d} : \Sigma^* \times \Sigma^* \to \mathbf{R}^+ \cup \{0\}$. Essa deve soddisfare le tre seguenti proprietà per dare luogo al cosiddetto spazio metrico (analogamente alla distanza Euclidea che tutti noi usiamo con la carta geografica):

- riflessiva: $\mathtt{d}(\mathtt{x}, \mathtt{x}) = 0$ per ogni $\mathtt{x} \in \Sigma^*$;
- simmetrica: $\mathtt{d}(\mathtt{x}, \mathtt{y}) = \mathtt{d}(\mathtt{y}, \mathtt{x})$ per ogni $\mathtt{x}, \mathtt{y} \in \Sigma^*$;
- triangolare: $\mathtt{d}(\mathtt{x}, \mathtt{z}) \leqslant \mathtt{d}(\mathtt{x}, \mathtt{y}) + \mathtt{d}(\mathtt{y}, \mathtt{z})$ per ogni $\mathtt{x}, \mathtt{y}, \mathtt{z} \in \Sigma^*$.

Esempi di distanze per stringhe sono la *distanza di Hamming* e la *distanza di Levenshtein*. La distanza di Hamming viene usata spesso nella teoria dei codici a correzione di errore. È definita per stringhe aventi pari lunghezza $\mathtt{m}$ e viene impiegata per misurare il numero di caratteri che non corrispondono (chiamati *mismatch*): $\mathtt{d}(\mathtt{x}, \mathtt{y}) = \big|\{\mathtt{i} \mid \mathtt{x}[\mathtt{i}] \neq \mathtt{y}[\mathtt{i}] \text{ per } 1 \leqslant \mathtt{i} \leqslant \mathtt{m}\}\big|$. Per esempio, $\mathtt{d}(\mathtt{stringa}, \mathtt{spranga}) = 2$ in quanto il secondo e il quarto carattere differiscono. La distanza di Levenshtein, nota come *distanza di edit*, si basa invece su tre operazioni di *edit* per trasformare una stringa $\mathtt{x}$ in una stringa $\mathtt{y}$ (qui ϵ denota il carattere vuoto):

- $\epsilon \to \mathtt{a}$: inserzione di un singolo carattere $\mathtt{a}$;
- $\mathtt{a} \to \epsilon$: cancellazione di un singolo carattere $\mathtt{a}$;
- $\mathtt{a} \to \mathtt{b}$: sostituzione (*mismatch*) di un singolo carattere $\mathtt{a}$ con un altro carattere $\mathtt{b} \neq \mathtt{a}$.

La distanza di edit $\mathtt{d}(\mathtt{x}, \mathtt{y})$ è quindi definita in termini della *minima* sequenza di operazioni di edit per trasformare $\mathtt{x}$ in $\mathtt{y}$. Per convenzione, $\mathtt{d}(\mathtt{x}, \mathtt{y})$ denota il solo numero di operazioni nella sequenza minima e tale numero può variare da 0 al massimo tra le lunghezze delle stringhe $\max\{|\mathtt{x}|, |\mathtt{y}|\}$. Assumiamo d'ora in poi che $|\mathtt{x}| = \mathtt{m} \leqslant \mathtt{n} = |\mathtt{y}|$, per cui $0 \leqslant \mathtt{d}(\mathtt{x}, \mathtt{y}) \leqslant \mathtt{n}$. Per esempio, $\mathtt{d}(\mathtt{ananas}, \mathtt{banane}) = 3$, in quanto possiamo trasformare `ananas` in `banane` con tre operazioni di edit: $\mathtt{ananas} \xrightarrow{\epsilon \to \mathtt{b}} \underline{\mathtt{b}}\mathtt{ananas} \xrightarrow{\mathtt{a} \to \mathtt{e}} \mathtt{banan}\underline{\mathtt{e}}\mathtt{s} \xrightarrow{\mathtt{s} \to \epsilon} \mathtt{banane}$.

La distanza è banalmente riflessiva perché non occorre alcuna operazione di edit per ottenere la stringa stessa. Vale la proprietà simmetrica, cioè $\mathtt{d}(\mathtt{x}, \mathtt{y}) = \mathtt{d}(\mathtt{y}, \mathtt{x})$: infatti, le inserzioni in $\mathtt{x}$ possono essere viste parimenti come cancellazioni da $\mathtt{y}$ e le cancellazioni da $\mathtt{x}$ come

inserzioni in $\mathtt{y}$; gli argomenti delle sostituzioni possono essere invertiti nell'ordine. Nel nostro esempio per $\mathtt{d}(\mathtt{banane}, \mathtt{ananas}) = 3$, la corrispondente sequenza di edit è ottenuta per inversione rispetto a quella descritta prima: $\mathtt{banane} \xrightarrow{\epsilon\to\mathtt{s}} \mathtt{bananes} \xrightarrow{\mathtt{e}\to\mathtt{a}} \mathtt{bananas} \xrightarrow{\mathtt{b}\to\epsilon} \mathtt{ananas}$.

Infine, la distanza soddisfa anche la proprietà triangolare, ossia $\mathtt{d}(\mathtt{x}, \mathtt{z}) \leqslant \mathtt{d}(\mathtt{x}, \mathtt{y}) + \mathtt{d}(\mathtt{y}, \mathtt{z})$. Per trasformare $\mathtt{x}$ in $\mathtt{z}$, possiamo sempre trasformare prima $\mathtt{x}$ in $\mathtt{y}$ e poi $\mathtt{y}$ in $\mathtt{z}$: quindi la trasformazione minima diretta da $\mathtt{x}$ a $\mathtt{y}$ può impiegare un numero di operazioni di edit pari a quelle impiegate nelle due trasformazioni o minore; certamente non può impiegarne un numero maggiore, perché contraddirebbe la condizione di minimalità di $\mathtt{d}(\mathtt{x}, \mathtt{y})$ e $\mathtt{d}(\mathtt{y}, \mathtt{z})$.

Una maniera alternativa di visualizzare la sequenza di edit, che non dipende dalla direzione di trasformazione (da $\mathtt{x}$ a $\mathtt{y}$ oppure da $\mathtt{y}$ a $\mathtt{x}$) e dall'ordine delle operazioni di edit ivi contenute, è basata sul concetto di *allineamento*, in cui le due stringhe sono sovrapposte verticalmente, allineando per colonne i loro caratteri con le seguenti regole: se due caratteri combaciano oppure sono dei mismatch, sono posti sulla stessa colonna, sottolineando i mismatch; altrimenti creiamo uno spazio vuoto da porre sulla stessa colonna del carattere inserito o cancellato. Nel nostro esempio abbiamo la seguente situazione in cui lo spazio bianco è indicato da $\sqcup$:

$$\begin{array}{cccccccc} \sqcup & \mathtt{a} & \mathtt{n} & \mathtt{a} & \mathtt{n} & \underline{\mathtt{a}} & \mathtt{s} \\ \mathtt{b} & \mathtt{a} & \mathtt{n} & \mathtt{a} & \mathtt{n} & \underline{\mathtt{e}} & \sqcup \end{array}$$

In realtà, possono esserci più sequenze della stessa lunghezza minima, ma questo non cambia la definizione di $\mathtt{d}(\mathtt{x}, \mathtt{y})$ in quanto è basata sulla lunghezza comune di tali sequenze. Nel nostro esempio, possiamo trasformare $\mathtt{ananas}$ in $\mathtt{banane}$ con un'altra sequenza di tre operazioni di edit: $\mathtt{ananas} \xrightarrow{\epsilon\to\mathtt{b}} \underline{\mathtt{b}}\mathtt{ananas} \xrightarrow{\mathtt{a}\to\epsilon} \mathtt{banans} \xrightarrow{\mathtt{s}\to\mathtt{e}} \mathtt{banan}\underline{\mathtt{e}}$, il cui allineamento è

$$\begin{array}{cccccccc} \sqcup & \mathtt{a} & \mathtt{n} & \mathtt{a} & \mathtt{n} & \mathtt{a} & \underline{\mathtt{s}} \\ \mathtt{b} & \mathtt{a} & \mathtt{n} & \mathtt{a} & \mathtt{n} & \sqcup & \underline{\mathtt{e}} \end{array}$$

Non è difficile convincersi che restringendo le operazioni di edit alle sole sostituzioni, la distanza di edit produce la distanza di Hamming come caso particolare. È inoltre possibile estendere il repertorio delle operazioni di edit aggiungendo la trasposizione di caratteri adiacenti:

- $\mathtt{ab} \to \mathtt{ba}$: trasposizione di due caratteri adiacenti $\mathtt{a} \neq \mathtt{b}$.

Per completezza, va detto che esistono altre definizioni di distanza tra stringhe, alcune delle quali portano a problemi considerati computazionalmente difficili (NP-completi). Alcune permettono di invertire la sequenza dei caratteri nelle sottostringhe, oppure di riorganizzare e permutare le sottostringhe in posizioni arbitrarie. Altre sono basate sulla scoperta di sottostringhe comuni di lunghezza prefissata $\mathtt{q}$ (chiamate $\mathtt{q}$-grammi), oppure sulla possibilità di scambiare due o più caratteri in posizioni arbitrarie, come generalizzazione dell'operazione di trasposizione.

Nonostante alcune sue limitazioni, la distanza di edit è tuttora quella maggiormente adottata, anche nella sua forma *pesata*. Ogni operazione di edit ha un costo, chiamato *peso*, che dipende da quali caratteri sono coinvolti. Tali pesi sono riportati in una matrice $W_{\mathtt{ab}}$ in cui viene specificato il costo di sostituzione del carattere $\mathtt{a}$ con il carattere $\mathtt{b}$. Le inserzioni e le cancellazioni vengono viste come una sostituzione, in cui il carattere vuoto ϵ è coinvolto, per cui $W_{\epsilon\mathtt{a}}$ è il peso dell'inserimento, visto come sostituzione di ϵ con il carattere $\mathtt{a}$, e $W_{\mathtt{a}\epsilon}$ è il peso della cancellazione, vista come la sostituzione del carattere $\mathtt{a}$ con ϵ.

Tali matrici pesate sono impiegate, per esempio, dai biologi molecolari che ne fissano i valori in modo che le operazioni di edit più probabili (e meno costose da un punto di vista di legami chimici) abbiano un costo inferiore a quelle meno probabili. La distanza di edit pesata è quindi definita in termini della sequenza di *costo minimo* (non necessariamente di lunghezza

minima!), dove il costo di una sequenza di edit è dato dalla somma dei pesi delle operazioni di edit in essa contenute. Fissando i pesi in modo che $W_{aa} = 0$ e i rimanenti W_{ab} siano tutti uguali e positivi, otteniamo la distanza di edit a costo uniforme, come caso particolare.

4.2. Calcolo efficiente della distanza di edit

Le possibili sequenze di operazioni di edit per trasformare una stringa x di lunghezza m in una stringa y di lunghezza $n \geqslant m$ possono essere in numero esponenziale. È quindi inefficiente generarle tutte per poi scegliere quella di lunghezza minima o di costo minimo. Per il momento, concentriamoci sulla distanza di edit a costo uniforme per cui cerchiamo la sequenza di operazioni di lunghezza minima.

Usiamo una tecnica di programmazione dinamica[1] per calcolare la distanza in $O(mn)$ tempo, impiegando una tabella D con $m+1$ righe numerate da 0 a m e con $n+1$ colonne numerate da 0 a n. L'elemento $D[i,j]$ contiene il minimo numero di operazioni di edit per trasformare i primi i caratteri $x[1..i]$ di x nei primi j caratteri $y[1..j]$ di y, dove $0 \leqslant i \leqslant m$ e $0 \leqslant j \leqslant n$. In realtà siamo interessati al solo valore $D[m,n]$ in quanto fornisce la distanza di edit $d(x,y)$. Tuttavia, risolviamo un problema più generale, ovvero quello di calcolare la distanza di edit tra tutti i prefissi di x e tutti i prefissi di y, perché questo aiuta notevolmente a trovare $d(x,y)$.

La colonna 0 e la riga 0 di D possono essere definite a priori come condizioni al contorno. Infatti, $D[i,0] = i$ per $1 \leqslant i \leqslant m$, in quanto trasformare una stringa di lunghezza i nella stringa vuota con una sequenza minima di operazioni richiede di cancellare tutti gli i caratteri della stringa. Analogamente, $D[0,j] = j$ per $1 \leqslant j \leqslant n$, in quanto trasformare la stringa vuota in una stringa di lunghezza j richiede l'inserzione di j caratteri.

Discutiamo ora come ottenere la seguente regola per riempire l'elemento $D[i,j]$, per $i,j > 0$,

$$D[i,j] = \min\big\{\, D[i-1,j-1] + [\![x[i] \neq y[j]]\!],\ D[i-1,j] + 1,\ D[i,j-1] + 1 \,\big\}, \tag{8}$$

dove $[\![x[i] \neq y[j]]\!]$ vale 1 se i due caratteri $x[i]$ e $y[j]$ sono diversi, mentre vale 0 se sono uguali.

La regola (8) si basa sull'induzione per numero di riga e di colonna, dove la colonna 0 e la riga 0 delle condizioni al contorno rappresentano il caso base, e considera i tre tipi di operazioni di edit che possono venire applicati. Mostriamo la correttezza di tale regola per ciascun $D[i,j]$, ipotizzando di aver calcolato correttamente i valori che si trovano immediatamente nella sua diagonale in alto a sinistra, in alto e poi a sinistra, ovvero $D[i-1,j-1]$, $D[i-1,j]$ e $D[i,j-1]$.

A tale scopo, consideriamo un allineamento minimo tra $x[1..i]$ e $y[1..j]$, che corrisponde alla loro distanza di edit, ed esaminiamo i due simboli che appaiono nell'ultima colonna di tale allineamento, che non possono essere entrambi $\sqcup$, ottenendo così quattro possibili situazioni.

(1) La colonna contiene due simboli che sono uguali: questi simboli devono essere necessariamente $x[i]$ e $y[j]$. Ne consegue che la distanza di edit $D[i,j]$ rimane uguale a $D[i-1,j-1]$. Per assurdo, se $D[i,j]$ fosse minore, lo sarebbe anche $D[i-1,j-1]$!
(2) La colonna contiene due simboli che sono diversi e nessuno dei due è uno spazio bianco $\sqcup$. Abbiamo un mismatch $x[i] \neq y[j]$; quindi dobbiamo incrementare $D[i-1,j-1]$ di 1 per ottenere la distanza di edit. Per assurdo, se $D[i,j]$ fosse minore, lo sarebbe anche $D[i-1,j-1]$.

[1] Il termine di programmazione dinamica è un po' fuorviante. In realtà si tratta di una tabella riempita (dinamicamente?) con una regola ricorsiva (di programmazione matematica). Per esempio, nel calcolare i numeri di Fibonacci $F_k = F_{k-1} + F_{k-2}$, con $F_0 = F_1 = 0$, la ricorsione dà luogo a un'esplosione combinaria. Invece, la programmazione dinamica tabula i valori di $F_0, F_1, \ldots, F_{k-1}$ e calcola il successivo F_k in tempo $O(1)$. Per completezza esiste un metodo ancora più veloce basato sulla moltiplicazione di opportune matrici 2×2.

(3) La posizione contiene, nell'ordine, un simbolo e uno spazio ␣. In tal caso, il simbolo è $x[i]$, che viene cancellato. La distanza di edit risultante $D[i, j]$ è perciò $D[i-1, j]$ incrementata di 1 per tener conto della cancellazione. Non può essere minore perché avremmo come assurdo che anche $D[i-1, j]$ lo sarebbe.

(4) La posizione contiene, nell'ordine, uno spazio ␣ e un simbolo. In tal caso, il simbolo è $y[j]$, che viene inserito. La distanza di edit $D[i, j]$ è perciò $D[i, j-1]$ incrementata di 1 a causa dell'inserzione. Non può essere minore perché lo sarebbe $D[i, j-1]$.

Come si può notare, l'ipotesi induttiva viene adoperata per la minimalità delle distanze $D[i-1, j-1]$, $D[i-1, j]$ e $D[i, j-1]$, per cui un valore di $D[i, j]$ inferiore a quelli discussi sopra porterebbe alla contraddizione di rendere inferiori anche i valori di $D[i-1, j-1]$, $D[i-1, j]$ o $D[i, j-1]$, rispettivamente. Purtroppo, per il calcolo effettivo di $D[i, j]$, non siamo a conoscenza del corrispondente allineamento minimo e non possiamo quindi applicare individualmente i casi 1–4 riportati sopra. Fortunatamente, $D[i, j]$ deve essere uno dei valori formulati in tali casi e, prendendone il minimo, siamo certi di scegliere il valore corretto per la distanza $D[i, j]$. Questa scelta viene pertanto descritta sinteticamente nella formula (8).

Mostriamo un esempio di calcolo della tabella D di programmazione dinamica con le stringhe $x =$ `ananas` e $y =$ `banane`. Riempiamo prima la colonna 0 e la riga 0 con i valori al contorno. Quindi applichiamo la regola (8) per calcolare gli altri elementi. Durante il calcolo di un elemento, dobbiamo esaminare solo i tre elementi adiacenti che si trovano, rispettivamente, in diagonale in alto a sinistra, in alto e poi a sinistra.

$D[i, j]$		b	a	n	a	n	e
	0	*1*	*2*	*3*	*4*	*5*	*6*
a	*1*	1	1	2	3	4	5
n	*2*	2	2	1	2	3	4
a	*3*	3	2	2	1	2	3
n	*4*	4	3	2	2	1	2
a	*5*	5	4	3	2	2	2
s	*6*	6	5	4	3	3	**3**

Notare che i valori degli elementi adiacenti nella tabella D differiscono di al più 1. Inoltre, definendo la diagonale d come l'insieme degli elementi $D[i, j]$ tali che $j - i = d$, possiamo notare che gli elementi su ogni diagonale d, dove $-m \leqslant d \leqslant n$, sono in ordine non decrescente (e chiaramente differiscono di al più 1). Sfrutteremo questa proprietà in seguito.

Lo pseudocodice per il calcolo della tabella D di programmazione dinamica segue fedelmente la formula (8) e procede riempiendo D per colonne (alternativamente, per righe) in modo da garantire che al momento del calcolo di $D[i, j]$, i valori dei suoi elementi adiacenti ($D[i-1, j-1]$, $D[i-1, j]$ e $D[i, j-1]$) siano stati correttamente trovati.

DistanzaEdit(x, y):

```
1: m ← |x|; n ← |y|;
2: FOR i ← 0 TO m DO
3:    D[i, 0] ← i;    {colonna 0}
4: FOR j ← 0 TO n DO
5:    D[0, j] ← j;    {riga 0}
6: FOR j ← 1 TO n DO
7:    FOR i ← 1 TO m DO
8:       IF x[i] = y[j] THEN
9:          D[i, j] ← D[i − 1, j − 1];
```

```
10:     ELSE
11:         D[i, j] ← 1 + MINIMO(D[i − 1, j − 1], D[i − 1, j], D[i, j − 1]);
12: RETURN D[m, n];
```

In tutti i casi visti sopra, ogni elemento della tabella D può essere riempito in tempo costante, una volta calcolati gli opportuni elementi ad esso adiacenti. Essendoci $O(mn)$ elementi in D, la complessità risultante è di $O(mn)$ tempo e spazio. Lo spazio può essere ridotto a $O(m)$ semplicemente mantenendo le ultime due colonne calcolate ($j-1$ e j) in D, in quanto $D[i, j]$ necessita solo di esse: tuttavia, un tale approccio permette solo di calcolare $d(x, y)$ ma non di trovare uno dei corrispondenti allineamenti.

Rimane da vedere, una volta stabilita la distanza di edit $d(x, y)$ mediante la tabella D, come ottenere anche una delle sequenze di trasformazione da x a y o il corrispondente allineamento. È possibile utilizzare la matrice D a ritroso, partendo dall'elemento $D[m, n]$. Dato un elemento $D[i, j]$, possiamo ricostruire le operazioni di edit in x:

(1) se $D[i, j] = D[i-1, j-1]$ e $x[i] = y[j]$, non ci sono operazioni di edit;
(2) se $D[i, j] = D[i-1, j-1] + 1$, abbiamo una sostituzione di $x[i]$ con $y[j]$;
(3) se $D[i, j] = D[i-1, j] + 1$, abbiamo una cancellazione di $x[i]$;
(4) se $D[i, j] = D[i, j-1] + 1$, abbiamo un'inserzione di $y[j]$.

Più casi possono valere simultaneamente per cui vanno tutti considerati in quanto ognuno conduce a un diverso allineamento. Immaginiamo quindi partire da $D[m, n]$. Per ogni caso che vale, visitiamo l'elemento corrispondente (rispettivamente $D[m-1, n-1]$, $D[m-1, n]$, $D[m, n-1]$). Procedendo ricorsivamente in tale maniera sul resto degli elementi, generiamo una visita a ritroso della tabella D seguendo però solo i casi descritti sopra. Osserviamo che possiamo sempre raggiungere l'elemento $D[0, 0]$ poiché almeno uno dei quattro casi sopra vale sempre: ogni volta che raggiungiamo $D[0, 0]$, abbiamo una distinta sequenza di trasformazione.

Nella tabella D del nostro esempio è riportato uno di tali cammini (numeri in neretto) e il rispettivo allineamento, dove segnaliamo l'inserimento di `b` per $D[0, 1]$, la cancellazione di `a` per $D[5, 5]$ e la sostituzione di `s` con `e` per $D[6, 6]$.

D[i, j]		b	a	n	a	n	e
	0	**1**	*2*	*3*	*4*	*5*	*6*
a	*1*	1	**1**	2	3	4	5
n	*2*	2	2	**1**	2	3	4
a	*3*	3	2	2	**1**	2	3
n	*4*	4	3	2	2	**1**	2
a	*5*	5	4	3	2	**2**	2
s	*6*	6	5	4	3	3	**3**

```
␣ a n a n a s
b a n a n ␣ e
```

4.3. Estensioni del calcolo della distanza di edit

Una maniera concettualmente più semplice per ricostruire la sequenza di trasformazione e il corrispondente allineamento consiste nell'utilizzare il *grafo della distanza di edit*.

(1) I vertici del grafo sono i punti di intersezione di una griglia di taglia $(m+1) \times (n+1)$, in cui il vertice (i, j) si trova all'incrocio della riga i e della colonna j.
(2) Il grafo è orientato e aciclico: gli archi orizzontali vanno da $(i, j-1)$ a (i, j); quelli verticali vanno da $(i-1, j)$ a (i, j); quelli diagonali vanno da $(i-1, j-1)$ a (i, j).
(3) Gli archi hanno costo pari a 1, tranne gli archi diagonali da $(i-1, j-1)$ a (i, j) tali che $x[i] = y[j]$, il cui costo è pari a 0.

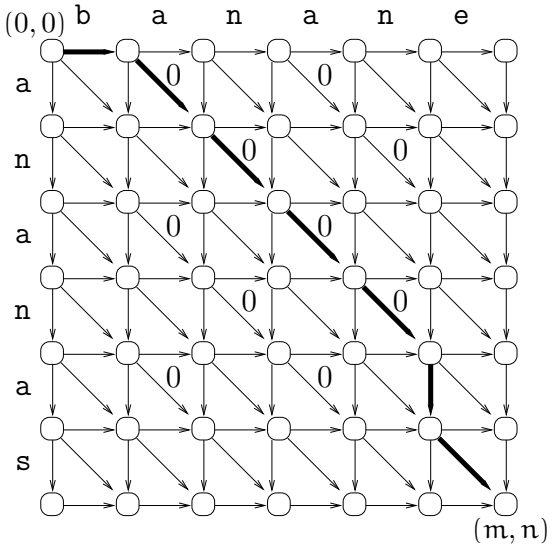


FIGURA 4.1. Un esempio di grafo della distanza di edit $d(x, y) = 3$, con un cammino minimo evidenziato, per $x =$ `ananas` e $y =$ `banane`. Gli archi hanno un costo pari a 1 di default, mentre i costi pari a 0 sono esplicitamente mostrati.

Un esempio di grafo della distanza di edit è mostrato in Figura 4.1 in cui è evidenziato un cammino minimo. In un tale grafo, consideriamo i cammini dal vertice $(0, 0)$ al vertice (m, n) e i loro costi calcolati come somma dei costi degli archi attraversati. Il minimo tra questi costi fornisce la distanza di edit $d(x, y)$. Inoltre, tutti i cammini di costo minimo $d(x, y)$ (non necessariamente di lunghezza minima in termini di numero di archi attraversati!) rappresentano tutte le possibili sequenze di trasformazioni di x in y con delle regole (analoghe a quelle citate nella Sezione 4.2), in base all'arco attraversato:

(1) se l'arco attraversato da $(i-1, j-1)$ a (i, j) ha costo 0, non ci sono operazioni di edit;
(2) se l'arco da $(i-1, j-1)$ a (i, j) ha costo 1, abbiamo una sostituzione di $x[i]$ con $y[j]$;
(3) se l'arco attraversato va da $(i-1, j)$ a (i, j), abbiamo una cancellazione di $x[i]$;
(4) se l'arco attraversato va da $(i, j-1)$ a (i, j), abbiamo un'inserzione di $y[j]$.

Lasciamo, come esercizio al lettore, la scrittura del codice che trova tali cammini minimi nel grafo della distanza di edit. Notare che tale grafo non va esplicitamente costruito, ma i suoi archi possono essere facilmente dedotti guardando ai caratteri delle due stringhe x e y di cui si vuole calcolare $d(x, y)$.

Mostriamo invece come verificare, date due stringhe x e y e un intero k tale che $0 \leqslant k \leqslant n$, se vale $d(x, y) \leqslant k$ in $O(nk)$ tempo e spazio, usando implicitamente il grafo della distanza di edit e la proprietà sulle diagonali non decrescenti della matrice D. Tuttavia, non è garantito che l'algoritmo calcoli sempre il valore $d(x, y)$: sicuramente è in grado di farlo se $d(x, y) \leqslant k$; se invece $d(x, y) > k$, restituisce un valore maggiore di k ma non necessariamente $d(x, y)$.

Questa caratteristica non è una vera limitazione perché possiamo eseguire l'algoritmo $O(\log k^*) = O(\log d(x, y))$ volte, per $k = 1, 2, 4, 8, \ldots, k^*$, dove $k^*/2 < d(x, y) \leqslant k^*$. Il costo totale è una somma geometrica $O(\sum_{k=1,2,4,8,\ldots,k^*} nk) = O(nk^*) = O(n\, d(x, y))$. Una semplice modifica dell'algoritmo ci permette anche di ricostruire i cammini minimi e, quindi, gli allineamenti di costo $d(x, y)$. In realtà, poiché l'algoritmo al caso pessimo esamina tutto il grafo e poi calcola $d(x, y)$ e l'allineamento, la sua complessità in tempo e spazio è $O(n \min\{m, d(x, y)\})$. Questo algoritmo è particolarmente efficiente per stringhe x e y che sono poco distanti l'una dall'altra, ossia con $d(x, y) \ll m$.

L'algoritmo di costo $O(nk)$ è ricorsivo e ha parametri che soddisfano le condizioni $0 \leqslant i \leqslant m$ e $0 \leqslant j \leqslant n$ a significare che vogliamo calcolare il costo di un cammino minimo nel grafo dal vertice (i, j) al vertice (m, n). Il caso base per l'algoritmo è rappresentato dal vertice (m, n), il cui costo minimo da se stesso è ovviamente pari a 0

Inizialmente, l'algoritmo viene invocato con $i = j = 0$ (insieme a x, y, m, n e k, che omettiamo come parametri). Durante la sua esecuzione, esamina i quattro possibili casi, così come sono stati discussi finora, memorizzando i valori minimi man mano calcolati in una tabella C di $2k+1$ righe e n colonne, dove $C[j-i, i]$ contiene il valore per il vertice (i, j). Tale tecnica, denominata *memoization*, è considerata equivalente alla programmazione dinamica.

CamminoCostoMin(i, j):

```
1: IF i = m AND j = n THEN
2:    RETURN 0;
3: IF |j − i| > k OR i > m OR j > n THEN
4:    RETURN k+1;
5: IF C[j − i, i] è stato memorizzato THEN
6:    RETURN C[j − i, i];
7: IF x[i + 1] = y[j + 1] THEN
8:    d = CamminoCostoMin(i + 1, j + 1);
9: ELSE
10:   d = CamminoCostoMin(i + 1, j + 1) + 1;
11: u = CamminoCostoMin(i + 1, j) + 1;
12: l = CamminoCostoMin(i, j + 1) + 1;
13: C[j − i, i] = Minimo(d, u, l);
14: RETURN C[j − i, i];
```

Se $d(x, y) \leqslant k$, l'algoritmo correttamente calcola $d(x, y)$. Innanzitutto, possiamo dedurre dalla tabella D che i valori non decrescenti sulle diagonali, che vanno da $-m$ a $-k-1$ e da $k+1$ a n, sono tutti strettamente maggiori di k e quindi non possono contribuire al calcolo di $d(x, y)$. In tal caso, l'algoritmo restituisce un valore $k+1$ a rappresentanza di ciò. Di conseguenza, i valori che possono contribuire a $d(x, y)$ sono quelli nelle $2k+1$ diagonali da $-k$ a k, ciascuna composta da al più n elementi. L'algoritmo restringe perciò la visita (implicita) del grafo ai soli vertici (i, j) lungo tale diagonale, ossia quelli per cui $|j - i| \leqslant k$. Tali vertici sono in numero $O(nk)$ per cui l'algoritmo impiega $O(nk)$ tempo e occupa $O(nk)$ spazio per la tabella C.

Se invece $d(x, y) > k$, allora un valore maggiore di k deve contribuire al calcolo di $d(x, y)$. Se tale valore è in una delle $2k+1$ diagonali suddette, l'algoritmo calcola tale valore. Se invece è in una delle restanti diagonali, l'algoritmo sostuisce tale valore con $k+1$ ma non pregiudica la proprietà che $d(x, y) > k$: alla fine il risultato potrebbe non essere $d(x, y)$ ma sicuramente è un valore strettamente maggiore di k. La complessità rimane pari a $O(nk)$ tempo e spazio.

Concludiamo esaminando come trattare il caso delle trasposizioni e la distanza pesata (Sezione 4.1), mantenendo lo stesso costo computazionale delle soluzioni per il calcolo di $d(x, y)$. Volendo ammettere anche le trasposizioni tra le operazioni possibili di edit, basta aggiungere un nuovo caso alle righe 8–11 di DistanzaEdit, che vanno sostituite con

```
8: IF x[i] = y[j] THEN
9:    D[i, j] ← D[i − 1, j − 1];
10: ELSE IF i > 1 AND j > 1 AND x[i − 1] x[i] = y[j] y[j − 1] THEN
11:   D[i, j] ← 1 + D[i − 2, j − 2];
12: ELSE
```

13: $\mathtt{D}[\mathtt{i},\mathtt{j}] \leftarrow 1 + \textsc{Minimo}\big(\mathtt{D}[\mathtt{i}-1,\mathtt{j}-1], \mathtt{D}[\mathtt{i}-1,\mathtt{j}], \mathtt{D}[\mathtt{i},\mathtt{j}-1]\big)$;

Nel caso più generale di distanza con i pesi $W_{\mathtt{a}\mathtt{b}}$, le condizioni al contorno cambiano ponendo il valore speciale ∞, più grande di qualunque reale impiegato, nella riga -1 e nella colonna -1, che possono essere anche mantenute implicitamente. Poniamo $\mathtt{D}[-1,-1] = 0$ in modo che $\mathtt{D}[0,0]$ sia correttamente posto a 0. Assumiamo inoltre che $\mathtt{x}[0] = \mathtt{y}[0] = \epsilon$. Il resto del codice è simile a DistanzaEdit in quanto è sufficiente sostituire l'incremento di 1 corrispondente a ogni operazione di edit con il corrispondente peso:

DistanzaEditPesata(x, y):

```
 1: m ← |x|; n ← |y|;
 2: for i ← 0 to m do
 3:    D[i, −1] ← ∞;    {condizione al contorno per le colonne}
 4: for j ← 0 to n do
 5:    D[−1, j] ← ∞;    {condizione al contorno per le righe}
 6: D[−1, −1] ← 0;
 7: for j ← 0 to n do
 8:    for i ← 0 to m do
 9:       if x[i] = y[j] then
10:          D[i, j] ← D[i − 1, j − 1];
11:       else
12:          D[i, j] ← Minimo(D[i − 1, j − 1] + W_{x[i] y[j]}, D[i − 1, j] + W_{x[i] ε}, D[i, j − 1] + W_{ε y[j]});
13: return D[m, n];
```

Per quanto riguarda il grafo della distanza di edit, cambiano solo i pesi degli archi: gli archi orizzontali da $(\mathtt{i},\mathtt{j}-1)$ a $(\mathtt{i},\mathtt{j})$ hanno costo pari a $W_{\epsilon\,\mathtt{y}[\mathtt{j}]}$; quelli verticali da $(\mathtt{i}-1,\mathtt{j})$ a $(\mathtt{i},\mathtt{j})$ hanno costo pari a $W_{\mathtt{x}[\mathtt{i}]\,\epsilon}$; infine, quelli diagonali da $(\mathtt{i}-1,\mathtt{j}-1)$ a $(\mathtt{i},\mathtt{j})$ hanno costo $W_{\mathtt{x}[\mathtt{i}]\,\mathtt{y}[\mathtt{j}]}$. Il resto degli algoritmi descritto in questa sezione rimane valido.

4.4. Distanza di edit e sottosequenza comune più lunga

La tecnica di programmazione dinamica per la distanza di edit risulta utile, tra l'altro, a trovare le differenze tra due documenti testuali. Possiamo applicare uno degli algoritmi visti sopra, in particolare DistanzaEdit, per stabilire la differenza. In effetti, il comando `diff` in Unix procede in questa maniera: un file viene visto come una sequenza di righe e ogni intera riga viene considerata come un simbolo dell'alfabeto Σ. Non essendo difficile stabilire se due righe di testo sono uguali o meno, la distanza di edit ci fornisce il minimo numero di inserimenti, cancellazioni e sostituzioni di intere righe per trasformare un file in un altro.

In realtà, vorremmo trovare anche le righe comuni tra due file. A tal fine, possiamo riformulare il nostro problema di distanza in termini di *sottosequenza* che, a differenza della sottostringa, è una sequenza i cui caratteri non sono necessariamente contigui nella stringa di partenza. Per esempio, `abc` è una sottosequenza di a**b**a**c**a**b**. La distinzione tra sottostringa e sottosequenza non è superficiale, in quanto le tecniche e i risultati ottenibili per le sottostringhe sono diversi da quelli per le sottosequenze (quest'ultime, in generale, più difficili da trattare). Riassumendo: nella terminologia corrente, stringa e sequenza rappresentano lo stesso concetto mentre sottostringa e sottosequenza indicano due concetti diversi. Una sottostringa è un caso particolare di sottosequenza, ma non vale il viceversa.

Una sottosequenza di una stringa può essere vista come il risultato della cancellazione di zero o più caratteri dalla stringa di partenza. Una sottosequenza *comune* a due stringhe `x` e `y` è una stringa che risulta sottosequenza di entrambe `x` e `y`. Per esempio `abc` è una sottosequenza comune a a**b**a**c**a**b** e a**b**r**a**c**a**d. Siamo interessati a trovare la più lunga, chiamata appunto la

sottosequenza comune più lunga $\mathtt{s}(\mathtt{x},\mathtt{y})$ tra $\mathtt{x}$ e $\mathtt{y}$: a tal fine, possiamo sfruttare una precisa relazione tra la lunghezza $|\mathtt{s}(\mathtt{x},\mathtt{y})|$ e la distanza di edit $\mathtt{d}'(\mathtt{x},\mathtt{y})$ in cui sono ammesse *solo* inserzioni e cancellazioni. Per esempio, consideriamo il seguente allineamento, dove i mismatch non sono ammessi:

```
a b r a c a d ␣
a b ␣ a c a ␣ b
```

La sottosequenza comune più lunga è $\mathtt{s}(\mathtt{x},\mathtt{y}) = \mathtt{abaca}$ e la sua lunghezza è pari a 5, mentre $\mathtt{d}'(\mathtt{x},\mathtt{y}) = 3$ (ossia il numero di colonne in cui appare ␣). Nell'allineamento mostrato sopra, i caratteri di $\mathtt{s}(\mathtt{x},\mathtt{y})$ appaiono due volte mentre quelli non appartenenti a $\mathtt{s}(\mathtt{x},\mathtt{y})$ appaiono nella riga superiore oppure in quella inferiore. Il totale dei caratteri contribuisce alla lunghezza totale delle stringhe $\mathtt{x}$ e $\mathtt{y}$, per cui vale la relazione $\mathtt{d}'(\mathtt{x},\mathtt{y}) = |\mathtt{x}| + |\mathtt{y}| - 2|\mathtt{s}(\mathtt{x},\mathtt{y})|$.

In altre parole, il numero minimo $\mathtt{d}'(\mathtt{x},\mathtt{y})$ di operazioni di edit può essere calcolato prendendo la sottosequenza comune più lunga $\mathtt{s}(\mathtt{x},\mathtt{y})$, per cui sappiamo che $2|\mathtt{s}(\mathtt{x},\mathtt{y})|$ caratteri combaciano a coppie e gli altri caratteri vanno inseriti o cancellati. La sottosequenza $\mathtt{s}(\mathtt{x},\mathtt{y})$ può essere determinata ponendo a 1 i costi delle inserzioni e delle cancellazioni, mentre il costo delle sostituzioni viene impostato al valore speciale ∞, più grande di ogni altro (è un modo per dire che ammettiamo solo inserzioni e cancellazioni). Minimizzando tale distanza di edit $\mathtt{d}'(\mathtt{x},\mathtt{y})$, rendiamo massima la lunghezza $|\mathtt{s}(\mathtt{x},\mathtt{y})|$ della sottosequenza comune.

È possibile calcolare direttamente $|\mathtt{s}(\mathtt{x},\mathtt{y})|$ usando una tabella di programmazione dinamica $\mathtt{S}$, tale che l'elemento $\mathtt{S}[\mathtt{i},\mathtt{j}]$ contiene la lunghezza della sottosequenza comune più lunga tra $\mathtt{x}[1..\mathtt{i}]$ e $\mathtt{y}[1..\mathtt{j}]$, dove $0 \leqslant \mathtt{i} \leqslant \mathtt{m}$ e $0 \leqslant \mathtt{j} \leqslant \mathtt{n}$. Le condizioni al contorno sono $\mathtt{S}[\mathtt{i},0] = \mathtt{S}[0,\mathtt{j}] = 0$ poiché se una delle due stringhe è vuota allora l'unica sottosequenza comune è quella vuota. Per $\mathtt{i},\mathtt{j} > 0$, abbiamo

- $\mathtt{S}[\mathtt{i},\mathtt{j}] = \mathtt{S}[\mathtt{i}-1,\mathtt{j}-1] + 1$ se $\mathtt{x}[\mathtt{i}] = \mathtt{y}[\mathtt{j}]$;
- $\mathtt{S}[\mathtt{i},\mathtt{j}] = \max\bigl\{\mathtt{S}[\mathtt{i}-1,\mathtt{j}], \mathtt{S}[\mathtt{i},\mathtt{j}-1]\bigr\}$ se $\mathtt{x}[\mathtt{i}] \neq \mathtt{y}[\mathtt{j}]$.

Lasciamo al lettore la dimostrazione della correttezza delle regole di programmazione dinamica appena esposte e la loro realizzazione in pseudocodice.

Il vantaggio di considerare il problema della differenza tra due file in termini di sottosequenza comune più lunga risiede nella possibilità di risolverlo in tempo $O((\mathtt{r}+\mathtt{n})\log\mathtt{n})$, dove $\mathtt{r}$ è il numero di possibili coppie di caratteri uguali nelle stringhe $\mathtt{x}$ e $\mathtt{y}$, dove il primo elemento di ogni coppia è un carattere di $\mathtt{x}$ e il secondo lo è di $\mathtt{y}$. Pur essendo $\mathtt{r} = O(\mathtt{m}\mathtt{n})$ al caso pessimo, in pratica $\mathtt{r} = O(\mathtt{n})$ quando applichiamo l'algoritmo alla differenza tra file, in cui i caratteri dell'alfabeto sono le intere righe dei file. In tal caso, trovare la sottosequenza comune più lunga richiede solo $O(\mathtt{n}\log\mathtt{n})$ tempo e $O(\mathtt{n})$ spazio, portando a una realizzazione alternativa senza cambiare funzionalità ma migliorandone decisamente le prestazioni.

4.5. Ricerca con al più k errori

Vediamo ora come impiegare la distanza di edit nel problema della ricerca approssimata con $\mathtt{k}$ errori (o differenze) di un pattern $\mathtt{P}$ nel testo $\mathtt{T}$: in tale problema, vogliamo individuare le sottostringhe $\mathtt{T}[\mathtt{j}'..\mathtt{j}]$ che sono a distanza al più $\mathtt{k}$ dal pattern $\mathtt{P}$, ossia $\mathtt{d}(\mathtt{P},\mathtt{T}[\mathtt{j}'..\mathtt{j}]) \leqslant \mathtt{k}$. Tuttavia, essendoci potenzialmente un numero quadratico di tali sottostringhe, siamo interessati a individuare in quali posizioni $\mathtt{j}$ del testo terminano le occorrenze di tali sottostringhe. Avendo al più $\mathtt{n}$ posizioni, conteniamo la dimensione dei dati in uscita così prodotti dalle soluzioni.

Formalmente, data una stringa testo $\mathtt{T}$ di lunghezza $\mathtt{n}$ e una stringa pattern $\mathtt{P}$ di lunghezza $\mathtt{m}$ e fissata una soglia massima $\mathtt{k} < \mathtt{m}$ per la distanza di edit $\mathtt{d}()$, vogliamo trovare tutte le posizioni $\mathtt{j}$, tali che $\mathtt{d}\bigl(\mathtt{P},\mathtt{T}[\mathtt{j}'..\mathtt{j}]\bigr) \leqslant \mathtt{k}$ per almeno un valore $\mathtt{j}'$ con $1 \leqslant \mathtt{j}' \leqslant \mathtt{j} \leqslant \mathtt{n}$.

Notare che, a differenza della ricerca esatta, la finestra di ricerca nel testo ha una dimensione variabile $m-k \leqslant j-j'+1 \leqslant m+k$, dove $m-k$ corrisponde ad avere tutte cancellazioni e $m+k$ ad avere tutte inserzioni. Nelle applicazioni reali, l'intervallo di valori usati per m sono dell'ordine delle migliaia al massimo, con pattern corti più frequenti nel recupero dei documenti e pattern lunghi diffusi nella biologia molecolare; solitamente, $1 \leqslant k \leqslant m/2$ per il numero di errori, mentre il testo può avere un lunghezza n dell'ordine dei milioni di caratteri o più, a seconda delle applicazioni.

Per risolvere il problema bisogna cambiare la tabella D di programmazione dinamica, introdotta nella Sezione 4.2. Sorprendentemente, il cambiamento è solo sulle condizioni al contorno per la riga 0, ponendo $D[0, j'] = 0$ per $0 \leqslant j' \leqslant n$ nella linea 5 di DistanzaEdit(P, T). In questo modo rendiamo ogni posizione j' un possibile inizio di un'occorrenza $T[j'..j]$ con errori. Equivalentemente, il pattern vuoto ha distanza zero in ogni posizione. Il resto della computazione in DistanzaEdit è esattamente come prima. Alla luce di queste considerazioni, vale la proprietà che $D[m, j] \leqslant k$ se e solo se c'è un'occorrenza approssimata di P che termina nella posizione j.

Come abbiamo visto, lo spazio utilizzato da DistanzaEdit è $O(nm)$ per cui è auspicabile ridurlo a $O(m)$ usando l'idea, menzionata prima, di mantenere le ultime due colonne calcolate $j-1$ e j. Vediamo meglio i dettagli di come applicare quest'idea al caso della ricerca approssimata. Nell'iterazione j, usiamo il vettore C per memorizzare il contenuto della colonna $j-1$ della tabella D e il vettore C' per calcolare i valori della colonna j. Durante ogni iterazione, manteniamo l'invariante che $C[i] = D[i, j-1]$ e $C'[i] = D[i, j]$ per $0 \leqslant i \leqslant m$. Prima di passare all'iterazione successiva, il contenuto di C' viene copiato in C per poter calcolare il nuovo valore di C'. Poichè $C[m] = D[m, j]$ dopo la copia tra vettori, se $C[m]$ ha un valore minore o uguale a k allora siamo certi che c'è un'occorrenza approssimata di P con al più k errori che termina nella posizione j.

RicercaApprossimata(P, T, k):

```
 1: m ← |P|; n ← |T|;
 2: FOR i ← 0 TO m DO
 3:    C[i] ← i;    {colonna 0}
 4: FOR j ← 1 TO n DO
 5:    C'[0] = 0;    {riga 0}
 6:    FOR i ← 1 TO m DO
 7:       IF P[i] = T[j] THEN
 8:          C'[i] ← C[i − 1];
 9:       ELSE
10:          C'[i] ← 1 + Minimo(C[i − 1], C'[i − 1], C[i]);
11:    C ← C';
12:    IF C[m] ⩽ k THEN
13:       RETURN TRUE;    {occorrenza approssimata di P che termina in j}
14: RETURN FALSE;
```

Non è difficile modificare lo pseudocodice per far stampare tutte le posizioni delle occorrenze approssimate di P in T con al più k differenze. Volendo costruire l'allineamento di un'occorrenza in posizione j con P, possiamo utilizzare le tecniche viste nelle Sezioni 4.2 e 4.3, applicandole su P e gli $m+k$ caratteri in T che precedono la posizione j (inclusa), procedendo a ritroso (per esempio, invertendo l'ordine dei caratteri in entrambe le stringhe).

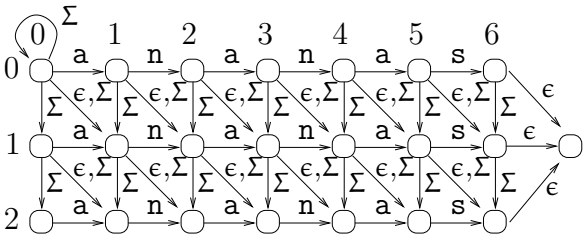
La complessità di RicercaApprossimata rimane $O(nm)$ tempo al caso pessimo mentre lo spazio si riduce a $O(m)$. Esistono soluzioni che migliorano il tempo al caso pessimo, per esempio ottenendo un costo di $O(nk)$ tempo, ma non sono qui discusse.

L'algoritmo RicercaApprossimata(P, T, k) viene utilizzato anche per il cosiddetto metodo di *best match*, in cui si trova il minimo $k^* \geqslant 0$ per cui c'è un'occorrenza di P con k^* errori. Possiamo iniziare fissando $k = 1, 2, 4, 8, \ldots$ per le potenze del due fino a che non troviamo un'occorrenza per $k = 2^r$ tale che $2^{r-1} < k^* \leqslant 2^r$: esisterà una posizione j tale che $C[m] = k^*$, per cui il costo totale è proporzionale a n moltiplicato una somma geometrica proporzionale a k^* (cfr. la Sezione 4.3 per un'analisi simile). Il costo finale è quindi $O(nk^*)$ invece che $O(nm)$, anche se non conosciamo a priori il valore di k^*.

4.6. Automi per la ricerca con errori

La ricerca con al più k errori di un pattern P può essere vista come caso speciale della ricerca con le espressioni regolari (Capitolo 3). Per convincerci di ciò, mostriamo un automa non deterministico che permette di risolvere il problema della ricerca approssimata. È anche possibile definire un automa deterministico ma richiede tempo e spazio esponenziale (come già discusso a proposito delle espressioni regolari).

Gli stati dell'automa non deterministico sono organizzati su $k + 1$ righe, dove ciascuna riga è composta da $m + 1$ stati. Le righe sono numerate da 0 a k: gli stati in ciascuna riga hanno un numero di colonna da 0 a m. In generale, lo stato (e, i) denota lo stato nella riga e e nella colonna i ed è attivo se i primi i caratteri $P[1..i]$ del pattern occorrono nella posizione corrente del testo con al più e errori. Gli stati dell'ultima colonna a destra nell'automa sono collegati allo stato finale mediante ϵ-transizioni (in realtà bastano quelli dell'ultima diagonale completa a destra) e lo stato iniziale è $(0, 0)$. Un esempio di automa per il pattern $P = \texttt{ananas}$ con al più $k = 2$ errori è riportato di seguito.



Dall'esempio, possiamo notare che le transizioni sono orizzontali, verticali e diagonali, a parte lo stato iniziale $(0, 0)$ che ha una transizione verso se stesso etichettata con Σ per far sì che sia sempre attivo in ogni posizione del testo. La transizione orizzontale dallo stato $(e, i-1)$ allo stato (e, i) è etichettata con il carattere $P[i]$ e rappresenta un match del carattere. La transizione verticale da $(e - 1, i)$ a (e, i) e la transizione diagonale da $(e - 1, i - 1)$ a (e, i) sono etichettate con Σ e rappresentano, rispettivamente, l'inserzione nel pattern del carattere correntemente letto nel testo e il suo mismatch con il carattere $P[i]$ del pattern. Esiste una ulteriore transizione diagonale da $(e - 1, i - 1)$ a (e, i) etichettata con ϵ per rappresentare la cancellazione del carattere $P[i]$ dal pattern.

Durante la computazione con l'automa, eseguiamo la simulazione dell'automa non deterministico visto nella Sezione 3.4. La complessità della simulazione dell'automa non deterministico per P con il testo T richiede $O(mn)$ tempo, come nelle espressioni regolari.

In generale, possiamo osservare che, se uno stato (e, i) è attivo, lo sono tutti quelli (e', i) con $e \leqslant e' \leqslant k$, in quanto avere al più e errori implica avere al più $e + 1$ errori, $e + 2$ errori e così via. È interessante notare la seguente relazione con la tabella D di programmazione dinamica: se dopo aver letto il carattere corrente $T[j]$ nel testo, prendiamo il valore minimo e per cui lo stato (e, i) è attivo, per $1 \leqslant i \leqslant m$, otteniamo che vale $D[i, j] = e$.

Rimuovendo l'arco che dallo stato iniziale va in se stesso, otteniamo un automa per P che possiamo comporre secondo le regole induttive viste per le espressioni regolari, generando un'espressione r che ci permette di raffinare la ricerca di P con al più k errori in base al contesto delle sue occorrenze. Per esempio, possiamo cercare la parola P = vendesi con al più k = 2 errori all'interno delle porzioni in neretto di un file HTML, con l'automa derivante dall'espressiore r = <[bB]>.*(?<!<[bB]>|</[bB]>)\bP\b.*(?<!<[bB]>|</[bB]>)</[bB]>, dove l'automa per P è quello descritto in questa sezione e viene composto con gli automi via via generati per la parte rimanente di r.

Tale schema è di per sé interessante perché combina la ricerca con errori con quella mediante espressioni regolari e viene adottato nel comando agrep. Nel Capitolo 5 vedremo un modo per realizzare efficientemente la ricerca approssimata usando operazioni logiche per manipolare i bit delle parole di memoria, analogamente a quanto succede con agrep.

Osserviamo infine che l'automa può essere costruito sul pattern con i caratteri in ordine inverso, per eseguire una simulazione dell'automa secondo lo stile dell'algoritmo di Boyer-Moore-Gosper-Horspool visto nel Capitolo 2.

4.7. Considerazioni pratiche

Per rendere veloci in pratica gli algoritmi per la ricerca approssimata sono stati studiati dei metodi di filtro. L'idea di base è quella di far girare prima un algoritmo semplice e veloce che permette di scartare diverse posizioni del testo T dove il pattern P sicuramente non può occorrere con al più k errori. Viene poi eseguito un algoritmo per la ricerca approssimata, più costoso computazionalmente, solo sulle zone selezionate che sopravvivono al filtro.

Discutiamo una semplice tecnica di questo tipo. Supponiamo di voler cercare P = ananas con al più k = 1 errori. A tale scopo, dividiamo P in due parti uguali: se né ana né nas occorrono nel testo, certamente non può apparire nemmemo P con al più un errore: infatti tale errore dovrebbe trovarsi o nella metà sinistra (ana) o nella metà destra (nas) e quindi l'altra metà deve occorrere esattamente.

In generale, con k errori, dividiamo P in $k+1$ parti uguali (o circa uguali se m non è multiplo di $k+1$). Seppure ciascuna parte può contenere errori, certamente deve esistere una parte che *non* contiene errori: se così non fosse, avremmo almeno $k+1$ errori e tali occorrenze non ci interessano. Di conseguenza, nelle zone di testo dove non appare *nessuna* delle $k+1$ parti possiamo concludere che non può esserci un'occorrenza approssimata. Viceversa, nelle rimanenti zone deve apparire almeno una parte. Per cercare simultaneamente le $k+1$ parti, possiamo impiegare l'algoritmo RicercaSemplice⋄AhoCorasick.

A questo punto, per ciascuna posizione j del testo in cui c'è un'occorrenza di almeno una delle $k+1$ parti, prendiamo un suo intorno di $2m+2k+1$ caratteri $T[j-m-k..j+m+k]$ ed eseguiamo uno degli algoritmi descritti in questo capitolo per verificare effettivamente se ci sia un'occorrenza di P con errori. Con una programmazione attenta è possibile evitare di riesaminare porzioni comuni di testo.

Questa tecnica funziona piuttosto bene con pattern di lunghezza moderata e un numero di errori non elevato, in quanto permette di scartare buona parte del testo attraverso il filtro. Per pattern lunghi esistono altre tecniche più sofisticate: dal punto di vista teorico, il migliore di tali algoritmi richiede $O(n(k+\log\sigma)/m)$ tempo in media per un numero di errori $k \leqslant m(1-e/\sqrt{\sigma})$.

CAPITOLO 5

Ricerca senza Uso di Confronti Diretti

Gli algoritmi di ricerca discussi finora adottano indirettamente il modello per confronti (*comparison model*), ossia si basano sui confronti diretti tra caratteri per stabilire se un dato pattern occorre nel testo, senza sfruttare altre ipotesi sull'alfabeto Σ: dati due simboli $\mathtt{a}, \mathtt{b} \in \Sigma$, le operazioni applicate sono sempre confronti del tipo $\mathtt{a} < \mathtt{b}$, $\mathtt{a} \leqslant \mathtt{b}$, $\mathtt{a} = \mathtt{b}$ e così via. In pratica, l'alfabeto usato è codificato in binario (ASCII e Unicode/UTF8) e, come già sottolineato in precedenza, possiamo vederlo come un intervallo di interi $\Sigma = \{1, 2, \ldots, \sigma\}$ (oppure $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ ma questo non cambia la sostanza delle cose). Abbiamo già usato i caratteri come indici di array e di tabelle per accedere al prossimo stato nelle transizioni di alcuni degli automi visti nel Capitolo 2 (fa eccezione l'algoritmo di Knuth-Morris-Pratt che è puramente per confronti). [1]

In questo capitolo, mostriamo invece come sfruttare il fatto che $\Sigma = \{1, 2, \ldots, \sigma\}$, utilizzando i caratteri come indici di array e tabelle. Descriviamo alcuni metodi di ricerca che sfruttano tale possibilità fornendo nuovi algoritmi che non sono basati sul confronto di caratteri, dando luogo ad applicazioni di indubbia utilità come `agrep`. In particolare, intendiamo impiegare le operazioni logiche e aritmetiche disponibili nei calcolatori per effettuare delle manipolazioni parallele sui bit delle parole di memoria, in modo da poter risolvere prima il problema della ricerca esatta con tali meccanismi e quindi estenderli alla ricerca approssima con distanza di edit e a quelle con le espressioni regolari.

5.1. Manipolazione logiche sui bit e modello di calcolo con parole a w bit

I calcolatori moderni offrono una serie di operazioni logiche per manipolare in parallelo i bit dei registri. Gli algoritmi aritmetici, oltre a usare le classiche operazioni logico-aritmetiche, sfruttano le operazioni del calcolatore per operare sui bit in parallelo, assumendo di avere a disposizione parole di memoria composte da w bit ciascuna. Per esempio, effettuando una serie di $\mathtt{k}$ operazioni AND , OR e NOT bit a bit, siamo in grado di verificare simultaneamente w condizioni booleane al costo di $\mathtt{k}$ operazioni. Gli algoritmi aritmetici portano all'estremo questa possibilità di manipolazioni dei bit. La tabella seguente elenca alcune delle più comuni operazioni logiche, adottate per esempio nel linguaggio di programmazione C.

[1] Va sottolineato che, volendo utilizzare tali algoritmi nel puro modello per confronti, il calcolo del prossimo stato nelle transizioni richiede $O(\log \sigma)$ tempo, tranne Knuth-Morris-Pratt.

$\mid$	OR logico in parallelo bit a bit
&	AND logico in parallelo bit a bit
$\wedge$	XOR (or esclusivo) in parallelo bit a bit
$\sim$	NOT (complemento) in parallelo bit a bit
$<< \mathtt{r}$	spostamento dei bit di $\mathtt{r}$ posizioni a sinistra
$>> \mathtt{r}$	spostamento dei bit di $\mathtt{r}$ posizioni a destra

Nello spostamento dei bit, le posizioni lasciate libere vengono solitamente riempite con 0. In tal caso, è utile definire una funzione *rshift*($\mathtt{x}$) che effettua uno spostamento a destra di tutti i bit di $\mathtt{x}$ mettendo 1 nella posizione più a sinistra lasciata libera. Ciò si ottiene eseguendo $(\mathtt{x} >> 1 | 10\cdots0)$, dove $\mathtt{x} >> 1$ effettua lo spostamento a destra di una posizione; l'OR con $10\cdots0$ forza ad avere 1 in tale posizione.

Nel seguito indicheremo con $\mathtt{b}_1\cdots\mathtt{b}_w$ i bit in una parola di memoria, dove $\mathtt{b}_1$ è il bit di peso maggiore e $\mathtt{b}_w$ è quello di peso minore. Attualmente, i calcolatori di uso comune hanno parole di memoria di $w = 32$ o $w = 64$ bit; non è impensabile prevedere $w = 128$ in un prossimo futuro.

Come esempio di applicazione delle operazioni suddette, consideriamo il problema di calcolare il numero di bit pari a 1 in una variabile intera $\mathtt{b} \geqslant 0$. Possiamo realizzare semplicemente ciò utilizzando un ciclo di w iterazioni sui suoi $\mathtt{b}_1\cdots\mathtt{b}_w$:

1: $mask \leftarrow 000\cdots01$; $\mathtt{c} \leftarrow 0$;
2: FOR $\mathtt{i} \leftarrow w$ DOWNTO 1 DO
3: IF $(\mathtt{b}\,\&\,mask) \neq 0$ THEN
4: $\mathtt{c} \leftarrow \mathtt{c} + 1$;
5: $mask \leftarrow mask << 1$;

Il comando condizionale verifica che $\mathtt{b}_\mathtt{i} \neq 0$ e, in tal caso, incrementa il contatore $\mathtt{c}$. A ogni iterazione, l'unico 1 nella maschera *mask* viene spostato di una posizione a sinistra. Poiché viene iterato per $\mathtt{i} = w, w-1, \ldots, 1$, il costo è proporzionale a w, indipendentemente dal numero w_1 di bit pari a 1 contenuti in $\mathtt{b}$.

È tuttavia possibile impiegare un numero di iterazioni che dipendono soltanto dal numero w_1 (ossia dai bit pari a 1, quelli che vogliamo contare).

1: c $\leftarrow 0$;
2: WHILE $\mathtt{b} \neq 0$ DO
3: $\mathtt{c} \leftarrow \mathtt{c} + 1$;
4: $\mathtt{b} \leftarrow \mathtt{b}\,\&\,(\mathtt{b} - 1)$;

A ogni iterazione del ciclo while, la prima istruzione incrementa il contatore $\mathtt{c}$ e la seconda istruzione azzera l'uno più a destra (meno significativo) di $\mathtt{b}$. Per esempio, se la rappresentazione in base 2 di $\mathtt{b}$ è 00101100, allora quella di $\mathtt{b}' = \mathtt{b} - 1$ è 00101011, per cui $\mathtt{b}\,\&\,\mathtt{b}'$ fornisce 00101000 in binario, azzerano di fatto il terzo bit da destra. In generale, se $\mathtt{b}_\mathtt{i} = 1$ e $\mathtt{b}_\mathtt{j} = 0$ per $\mathtt{j} > \mathtt{i}$, osserviamo che i primi $\mathtt{i} - 1$ bit di $\mathtt{b}' = \mathtt{b} - 1$ rimangono inalterati ($\mathtt{b}'_\mathtt{j} = \mathtt{b}_\mathtt{j}$ per $\mathtt{j} < \mathtt{i}$) mentre $\mathtt{b}_\mathtt{i} = 0$ e $\mathtt{b}_\mathtt{j} = 1$ per $\mathtt{j} > \mathtt{i}$. Di conseguenza, effettuando l'and logico tra $\mathtt{b}$ e $\mathtt{b}'$ otteniamo come risultato la sequenza in cui i primi $\mathtt{i} - 1$ bit sono uguali a $\mathtt{b}_1, \ldots, \mathtt{b}_{\mathtt{i}-1}$ mentre il resto dei bit sono pari a 0.

Questo metodo (secondo D.E. Knuth, introdotto indipendentemente da B. Kernighan, D. Lehmer e P. Wegner) mostra come l'uso attento delle manipolazioni parallele dei bit possa portare dei vantaggi nelle prestazioni. (Per completezza, esistono metodi ancora più efficienti per tale conteggio). Vediamo come impiegarle per la ricerca di pattern.

5.2. Paradigma Shift-And

Un semplice esempio di algoritmo aritmetico è dato dal paradigma Shift-And introdotto da Baeza-Yates e Gonnet per la ricerca esatta di un pattern P di lunghezza m in un testo T di lunghezza n. Permette di ottenere un tempo di ricerca $O(n\lceil m/w\rceil)$, per cui è un'interessante alternativa per pattern medio o piccoli, con $m = O(w)$, come nel caso del recupero di testi, anche se non vengono raggiunte le prestazioni dell'algoritmo RicercaBoyerMooreGosperHorspool. Discutiamo tale paradigma perché è la base degli algoritmi di ricerca più complessi descritti in seguito.

Nella descrizione dell'algoritmo, assumiamo per semplicità che sia $m = w$, in quanto vedremo come rimuovere questa assunzione. Useremo la notazione $P[1..i] = \mathit{suff}(T[1..j])$ per indicare che il prefisso $P[1..i]$ del pattern occorre come *suffisso* del prefisso $T[1..j]$ del testo. L'algoritmo mantiene una parola di memoria $D = d_1 \dots d_m$ per lo stato corrente, con la seguente invariante: dopo aver letto il carattere corrente $T[j]$ dal testo ($1 \leqslant j \leqslant n$), per $1 \leqslant i \leqslant m$ vale la proprietà

$$d_i = 1 \text{ se e solo se } P[1..i] = \mathit{suff}(T[1..j]) \tag{9}$$

Per esempio, ponendo $P =$ `ananas` e $T =$ `bananananassata` otteniamo le seguenti configurazioni di D, in cui sono evidenziati i prefissi del pattern che hanno un'occorrenza che termina nella posizione corrente del testo.

j	$T[j]$	$D = d_1d_2d_3d_4d_5d_6$	Prefissi di P che terminano in $T[1..j]$
1	b	0 0 0 0 0 0	
2	a	1 0 0 0 0 0	a
3	n	0 1 0 0 0 0	an
4	a	1 0 1 0 0 0	a, ana
5	n	0 1 0 1 0 0	an, anan
6	a	1 0 1 0 1 0	a, ana, anana
7	n	0 1 0 1 0 0	an, anan
8	a	1 0 1 0 1 0	a, ana, anana
9	s	0 0 0 0 0 1	ananas!
10	s	0 0 0 0 0 0	
11	a	1 0 0 0 0 0	a
12	t	0 0 0 0 0 0	
13	a	1 0 0 0 0 0	a

Come si può notare, i valori 1 in D al passo j corrispondono ai prefissi del pattern che occorrono come suffissi di $T[1..j]$. Al passo $j = 7$, vale $d_2 = d_4 = 1$ a indicare che $P[1..2] =$ `an` e $P[1..4] =$ `anan` sono suffissi di $T[1..7] =$ `bananan`. Quando $d_m = 1$, abbiamo un'occorrenza di P che termina nella posizione j del testo; per esempio, questo accade al passo $j = 9$.

Mostriamo come il paradigma Shift-And permette di calcolare il valore corretto di D in tempo costante, a ogni passo j. A tal fine, viene creata una tabella B composta da σ parole di memoria. La parola $B[c] = b_1 \dots b_m$ riveste il ruolo di maschera per le occorrenze del carattere $c \in \Sigma$ all'interno di P. A tal fine, $B[c]$ ha il bit i-esimo $b_i = 1$ se solo se $P[i] = c$. In altre parole, $B[c]$ è una maschera binaria i cui bit a 1 segnalano la presenza del carattere c nel pattern P. Nel caso che c non appaia in P, abbiamo tutti i bit a 0. Nel nostro esempio:

	ananas
...	000000
B[a]	101010
...	000000

	ananas
...	000000
B[n]	010100
...	000000

	ananas
...	000000
B[s]	000001
...	000000

Innanzi tutto, il passo 0 corrisponde a inizializzare D in modo da contenere tutti 0. Per aggiornare D, supponiamo di aver calcolato correttamente D fino al passo $\mathsf{j}-1$ e di voler calcolare il nuovo valore di D, denotato da D', al passo $\mathsf{j} \geqslant 1$. In altre parole, $\mathsf{D} = \mathsf{d}_1\mathsf{d}_2\cdots\mathsf{d}_m$ si riferisce alla posizione $\mathsf{j}-1$ e $\mathsf{D}' = \mathsf{d}'_1\mathsf{d}'_2\cdots\mathsf{d}'_m$ alla posizione j. Quando leggiamo il carattere corrente $\mathsf{T}[\mathsf{j}]$ nel testo, utilizziamo la seguente proprietà

$$\underbrace{\mathsf{P}[1..\mathsf{i}] = \mathit{suff}(\mathsf{T}[1..\mathsf{j}])}_{\mathsf{d}'_\mathsf{i}=1} \Leftrightarrow \underbrace{\mathsf{P}[1..\mathsf{i}-1] = \mathit{suff}(\mathsf{T}[1..\mathsf{j}-1])}_{\mathsf{d}_{\mathsf{i}-1}=1} \;\&\; \underbrace{\mathsf{P}[\mathsf{i}] = \mathsf{T}[\mathsf{j}]}_{\mathsf{B}[\mathsf{T}[\mathsf{j}]]_\mathsf{i}=1} \tag{10}$$

Fortunatamente, possiamo esprimere le due condizioni dell'equazione (10) mediante D e B, usando le operazioni logiche sui bit. Innanzi tutto, vale che l'$(\mathsf{i}-1)$-esimo bit in D vale 1 se e solo se $\mathsf{P}[1..\mathsf{i}-1]$ occorre come suffisso di $\mathsf{T}[1..\mathsf{j}-1]$. Inoltre, l'$\mathsf{i}$-esimo bit della parola $\mathsf{B}\big[\mathsf{T}[\mathsf{j}]\big]$ vale 1 se e solo se $\mathsf{P}[\mathsf{i}] = \mathsf{T}[\mathsf{j}]$. Vogliamo mettere in and queste due condizioni ponendo a 1 l'i-esimo bit di D' se e solo se entrambe valgono.

Per realizzare l'equazione (10) utilizzando le operazioni logiche, basta spostare i bit di D di una posizione a destra (in modo che l'$(\mathsf{i}-1)$-esimo bit in D occupi la posizione i) mettendoli in and logico con la parola $\mathsf{B}\big[\mathsf{T}[\mathsf{j}]\big]$. Per $1 \leqslant \mathsf{i} \leqslant \mathsf{m}$, se i bit in posizione i sono entrambi 1 abbiamo che entrambe le condizioni sono soddisfatte. Possiamo concludere che $\mathsf{P}[1..\mathsf{i}]$ occorre come suffisso di $\mathsf{T}[1..\mathsf{j}]$. In sintesi, per calcolare D' basta quindi eseguire

$$\mathsf{D}' \leftarrow \mathit{rshift}(\mathsf{D}) \,\&\, \mathsf{B}\big[\mathsf{T}[\mathsf{j}]\big].$$

Nel nostro esempio $\mathsf{D} = 101010$ prima del passo 6, per cui D' al passo $\mathsf{j} = 7$ è ottenuto eseguendo $\mathit{rshift}(\mathsf{D}) = 110101$, il cui risultato è messo in and con $\mathsf{B}[\mathsf{n}] = 0101000$, fornendo così $\mathsf{D}' = 010100$ come risultato. Lo pseudocodice per il paradigma Shift-And applicato alla ricerca esatta è riportato di seguito.

RicercaShiftAnd(P, T):

```
1: m ← |P|; n ← |T|;
2: FOREACH c ∈ Σ DO
3:     B[c] ← 00···0;
4: mask ← 10···0;
5: FOR i ← 1 TO m DO
6:     B[P[i]] ← mask | B[P[i]];
7:     mask ← mask >> 1;
8: D ← 00···0; mask ← 0···01;
9: FOR j ← 1 TO n DO
10:    D ← rshift(D) & B[T[j]]
11:    IF (D & mask) ≠ 0 THEN
12:        RETURN TRUE;    {occorrenza T[j − m + 1..j] = P}
13: RETURN FALSE;
```

Notare che il ciclo for alle linee 5–7 pone $\mathsf{B}\,[\mathsf{P}[\mathsf{i}]]_\mathsf{i} = 1$, di fatto costruendo la tabella B. Inoltre, il testo alla linea 11 permette di stabilire se $\mathsf{d}_\mathsf{m} = 0$.

La complessità di RicercaShiftAnd è di $\mathsf{O}(\sigma + \mathsf{m} + \mathsf{n}) = \mathsf{O}(\mathsf{n})$ tempo in quanto tutte le istruzioni nelle singole linee richiedono tempo costante, ipotizzando che $\mathsf{m} = w$. Se $\mathsf{m} < w$, possiamo utilizzare i primi m bit delle parole ponendo a 0 i restanti. Se invece $\mathsf{m} > w$, possiamo dividere ciascuna sequenza di m bit in $\lceil \mathsf{m}/w \rceil$ parole di w bit ciascuna. La simulazione di una operazione logica su m bit ora richiede tempo $\mathsf{O}(\lceil \mathsf{m}/w \rceil)$, che va moltiplicato per la complessità precedente. In sintesi, l'algoritmo ha complessità $\mathsf{O}(\lceil \mathsf{m}/w \rceil \mathsf{n})$ tempo, che diventa lineare quando $w = \mathsf{O}(\mathsf{m})$. Lo spazio richiesto è $\mathsf{O}(\sigma + \mathsf{m})$.

Non è difficile estendere la ricerca esatta in RICERCA◇SHIFTAND permettendo la specifica di più caratteri in una data posizione. Per esempio, volendo cercare sia `ananas` che `asanas`, possiamo specificare il pattern $P = a[ns]anas$. Per trattare una ricerca di questo tipo, basta modificare le maschere B. Nel nostro esempio, il secondo bit di $B[s]$ va posto a 1, per cui $B[s] = 010001$. Il resto dell'algoritmo non cambia. In generale, per ogni posizione i del pattern che specifica un insieme di caratteri $[c_1, \ldots, c_s]$, poniamo a 1 l'i-esimo bit nelle maschere $B[c_1]$, $\ldots$, $B[c_s]$.

Se la posizione i del pattern specifica un qualunque carattere (quindi, contiene il carattere speciale DONTCARE $= \backslash.$, indicato anche con '?', delle espressioni regolari), poniamo a 1 l'i-esimo bit in tutte le maschere; se DONTCARE appare nel testo, definiamo per lui la maschera $B[\text{DONTCARE}] = 1 \cdots 1$ contenente tutti 1.

Infine, possiamo trattare alla stessa stregua anche il complemento: se la posizione i del pattern non deve contenere i caratteri $c_1, \ldots, c_s$, poniamo a 1 l'i-esimo bit nelle maschere $B[c]$ per ogni carattere $c \in \Sigma - \{c_1, \ldots, c_s\}$ che sia diverso. In tutti questi casi, la complessità non cambia e rimane di $O(\lceil m/w \rceil n)$ tempo e $O(\sigma + m)$ spazio.

Non possiamo trattare però pattern più flessibili, come nelle espressioni regolari, per cui è necessario estendere tale paradigma ad altri tipi di ricerca.

5.3. Estensione del paradigma ad `agrep`: distanza di edit

Il paradigma SHIFT-AND può essere ampliato per trattare le ricerche approssimate con al più k errori, ottenendo degli algoritmi impiegati in `agrep`, una estensione di `grep` di Wu e Manber. [2] Vediamo come applicare il paradigma alle ricerche di un pattern P di lunghezza m in un testo T di lunghezza n usando la distanza di edit $d()$ introdotta nel Capitolo 4. In un certo senso, vogliamo simulare l'automa non deterministico descritto nella Sezione 4.6.

La parola di memoria D per rappresentare lo stato corrente viene sostituita con $k+1$ parole $R_0, \ldots R_k$ per simulare le $k+1$ righe dell'automa non deterministico (in quanto ammettiamo occorrenze che abbiano distanza di edit al più k dal pattern P). In particolare, per $0 \leqslant e \leqslant k$, la parola R_e codifica le occorrenze del pattern con al più e errori (analogamente alla riga e dell'automa non deterministico). Dopo aver letto il carattere corrente $T[j]$ del testo, viene mantenuta la seguente invariante, dove $(R_e)_i$ indica l'i-esimo bit di R_e, per $1 \leqslant i \leqslant m$:

$$(R_e)_i = 1 \text{ se e solo se } d(P[1..i], \mathit{suff}(T[1..j])) \leqslant e. \tag{11}$$

In altre parole, se l'i-esimo bit di R_e vale 1, sappiamo che $P[1..i]$ occorre come suffisso di $T[1..j]$ con al più e errori. Notare che la definizione di R_0 coincide con quella di D vista nella Sezione 5.2.

Ipotizzando di aver correttamente calcolato i valori di ciascun R_e, possiamo risolvere il problema della ricerca approssimata, verificando che l'ultimo bit in R_k valga 1 (ossia $(R_k)_m = 1$). Anche se siamo interessati a R_k, le altre parole di memoria R_e servono a calcolare efficientemente R_k in modo induttivo. L'induzione è doppia, nel senso che c'è un'induzione sulla posizione j del testo (analogamente alla Sezione 5.2) e, per un valore j fissato, c'è un'induzione sul numero di errori e.

Inizialmente, per il passo 0, poniamo $R_e = 1^e 0^{m-e}$ per $0 \leqslant e \leqslant k$, ovvero e copie di 1 seguite da $m-e$ copie di 0, perché un prefisso di P di lunghezza al più e occorre sempre con al più e errori. Per il passo $j = 1, 2, \ldots, n$, ipotizziamo di aver correttamente calcolato le parole R_e per il passo $j-1$ e mostriamo come ottenere il loro aggiornamento, indicato con R'_e, dopo aver letto il carattere corrente $T[j]$ del testo.

[2] Disponibile nel Web all'indirizzo `http://manber.com/software.html`

Poiché la parola R_0 corrisponde alle occorrenze esatte, viene quindi aggiornata come D, secondo quanto discusso nella Sezione 5.2:

$$\mathsf{R}'_0 \leftarrow \mathit{rshift}(\mathsf{R}_0) \,\&\, \mathsf{B}\big[\mathsf{T}[j]\big].$$

La parola R_e per $e = 1, 2, \ldots, k$ deve invece effettuare l'or logico di quattro casi possibili (match, sostituzione, cancellazione e inserzione) in quanto non conosciamo a priori quale di questi avverrà (anche se certamente almeno uno di loro accadrà), analogamente a quanto espresso dalle frecce dell'automa non deterministico descritto nella Sezione 4.6. Quando leggiamo il carattere corrente $\mathsf{T}[j]$, per ciascun $e = 1, 2, \ldots, k$, abbiamo quindi i seguenti casi affinché $\mathsf{P}[1..i]$ occorra con al più e errori come suffisso di $\mathsf{T}[1..j]$ (ossia $\mathsf{d}(\mathsf{P}[1..i], \mathit{suff}(\mathsf{T}[1..j])) \leqslant e$). A tal fine, ricordiamo che R_e si riferisce a $\mathsf{T}[1..j-1]$ mentre R'_e si riferisce a $\mathsf{T}[1..j]$; inoltre, il bit in posizione i sia di R_e che di R'_e si riferisce a $\mathsf{P}[1..i]$.

- Match tra $\mathsf{P}[i]$ e $\mathsf{T}[j]$:[3] qui $\mathsf{P}[1..i-1]$ occorre con al più e errori come suffisso di $\mathsf{T}[1..j-1]$ e vale $\mathsf{P}[i] = \mathsf{T}[j]$. L'espressione corrispondente è analoga a quella di R_0, ovvero $\mathit{rshift}(\mathsf{R}_e) \,\&\, \mathsf{B}\big[\mathsf{T}[j]\big]$, perché deriva da una proprietà analoga alla (10), ossia
$$\underbrace{\mathsf{d}(\mathsf{P}[1..i], \mathit{suff}(\mathsf{T}[1..j])) \leqslant e}_{(\mathsf{R}'_e)_i=1} \Leftrightarrow \underbrace{\mathsf{d}(\mathsf{P}[1..i-1], \mathit{suff}(\mathsf{T}[1..j-1])) \leqslant e}_{(\mathsf{R}_e)_{i-1}=1} \;\&\; \underbrace{\mathsf{P}[i] = \mathsf{T}[j]}_{\mathsf{B}[\mathsf{T}[j]]_i=1} \tag{12}$$
- Sostituzione di $\mathsf{P}[i]$ con $\mathsf{T}[j]$:[4] abbiamo che $\mathsf{P}[1..i-1]$ deve occorrere con al più $(e-1)$ errori come suffisso di $\mathsf{T}[1..j-1]$ e vale $\mathsf{P}[i] \neq \mathsf{T}[j]$. In tal caso, possiamo esprimere tale proprietà come
$$\underbrace{\mathsf{d}(\mathsf{P}[1..i], \mathit{suff}(\mathsf{T}[1..j])) \leqslant e}_{(\mathsf{R}'_e)_i=1} \Leftrightarrow \underbrace{\mathsf{d}(\mathsf{P}[1..i-1], \mathit{suff}(\mathsf{T}[1..j-1])) \leqslant e-1}_{(\mathsf{R}_{e-1})_{i-1}=1} \;\&\; \underbrace{\mathsf{P}[i] \neq \mathsf{T}[j]}_{\mathsf{B}[\mathsf{T}[j]]_i=0} \tag{13}$$
  L'espressione corrispondente alla prima condizione a destra dell'uguaglianza in (13) è $\mathit{rshift}(\mathsf{R}_{e-1})$. Si noti che possiamo evitare di metterla in and logico con il complemento $\sim \mathsf{B}\big[\mathsf{T}[j]\big]$ corrispondente alla seconda condizione (perché se non vale abbiamo un match e rientriamo nel caso precedente).
- Cancellazione di $\mathsf{P}[i]$:[5] qui $\mathsf{P}[1..i-1]$ deve occorrere con al più $(e-1)$ errori come suffisso di $\mathsf{T}[1..j]$, per cui l'espressione è $\mathit{rshift}(\mathsf{R}'_{e-1})$, come indicato dalla proprietà
$$\underbrace{\mathsf{d}(\mathsf{P}[1..i], \mathit{suff}(\mathsf{T}[1..j])) \leqslant e}_{(\mathsf{R}'_e)_i=1} \Leftrightarrow \underbrace{\mathsf{d}(\mathsf{P}[1..i-1], \mathit{suff}(\mathsf{T}[1..j])) \leqslant e-1}_{(\mathsf{R}'_{e-1})_{i-1}=1} \tag{14}$$
- Inserzione di $\mathsf{T}[j]$:[6] in questo caso, $\mathsf{P}[1..i]$ deve occorrere con al più $(e-1)$ errori come suffisso di $\mathsf{T}[1..j-1]$, per cui l'espressione è R_{e-1}, come indicato dalla proprietà
$$\underbrace{\mathsf{d}(\mathsf{P}[1..i], \mathit{suff}(\mathsf{T}[1..j])) \leqslant e}_{(\mathsf{R}'_e)_i=1} \Leftrightarrow \underbrace{\mathsf{d}(\mathsf{P}[1..i], \mathit{suff}(\mathsf{T}[1..j-1])) \leqslant e-1}_{(\mathsf{R}_{e-1})_i=1} \tag{15}$$

Possiamo quindi mettere le quattro espressioni in or logico, ottenendo $\mathsf{R}'_e \leftarrow \mathit{rshift}(\mathsf{R}_e) \,\&\, \mathsf{B}\big[\mathsf{T}[j]\big] \;\big|\; \mathit{rshift}(\mathsf{R}_{e-1}) \;\big|\; \mathit{rshift}(\mathsf{R}'_{e-1}) \;\big|\; \mathsf{R}_{e-1}$, che può essere riscritta mettendo in evidenza *rshift* nel secondo e terzo termine dell'OR come

$$\mathsf{R}'_e \leftarrow \mathit{rshift}(\mathsf{R}_e) \,\&\, \mathsf{B}\big[\mathsf{T}[j]\big] \;\big|\; \mathit{rshift}(\mathsf{R}_{e-1} \,|\, \mathsf{R}'_{e-1}) \;\big|\; \mathsf{R}_{e-1}.$$

Si noti che l'aggiornamento di R'_e richiede soltanto due operazioni di spostamento (*rshift*), una operazione di and e tre operazioni di or.

[3] Equivale alla freccia orizzontale etichettata con $\mathsf{P}[i]$ nell'automa della Sezione 4.6.

[4] Equivale alla freccia diagonale etichettata con Σ nell'automa della Sezione 4.6.

[5] Equivale alla freccia diagonale etichettata con ϵ nell'automa della Sezione 4.6.

[6] Equivale alla freccia verticale etichettata con Σ nell'automa della Sezione 4.6.

Per esempio, fissando $k = 1$ per il pattern $P =$ `ananas` nel testo $T =$ `bananananassata`, otteniamo che le configurazioni di R_0 sono esattamente quelle di D descritte nella Sezione 5.2. Concentriamoci quindi su R_1, mostrando come cambia quando i caratteri di T vengono letti.

j	$T[j]$	R_0	R_1	Prefissi $P[1..i]$ tali che $d(P[1..i], \mathit{suff}(T[1..j])) \leqslant 1$
0		0 0 0 0 0 0	1 0 0 0 0 0	
1	b	0 0 0 0 0 0	1 0 0 0 0 0	
2	a	1 0 0 0 0 0	1 1 0 0 0 0	a, an
3	n	0 1 0 0 0 0	1 1 1 0 0 0	a, an, ana
4	a	1 0 1 0 0 0	1 1 1 1 0 0	a, an, ana, anan
5	n	0 1 0 1 0 0	1 1 1 1 1 0	a, an, ana, anan, anana
6	a	1 0 1 0 1 0	1 1 1 1 1 1	a, an, ana, anan, anana, ananas!
7	n	0 1 0 1 0 0	1 1 1 1 1 1	a, an, ana, anan, anana, ananas!
8	a	1 0 1 0 1 0	1 1 1 1 1 1	a, an, ana, anan, anana, ananas!
9	s	0 0 0 0 0 1	1 1 1 1 1 1	a, an, ana, anan, anana, ananas!
10	s	0 0 0 0 0 0	1 0 0 0 0 1	a, ananas!
11	a	1 0 0 0 0 0	1 1 0 0 0 0	a, an
12	t	0 0 0 0 0 0	1 1 0 0 0 0	a, an
13	a	1 0 0 0 0 0	1 1 1 0 0 0	a, an, ana

Lo pseudocodice per la realizzazione dell'algoritmo è analogo a RICERCA⋄SHIFTAND, con la differenza che si gestiscono $k + 1$ parole invece che una. La doppia induzione è realizzata attraverso un ciclo for per tutti i valori $j = 1, 2, \ldots, n$, contenente al suo interno un ciclo for per tutti i valori $e = 1, 2, \ldots, k$.

RICERCAAPPROSSIMATA⋄SHIFTAND(P, T, k):

```
1:  m ← |P|; n ← |T|;
2:  FOREACH c ∈ Σ DO
3:      B[c] ← 00···0;
4:  mask ← 10···0;
5:  FOR i ← 1 TO m DO
6:      B[P[i]] ← mask | B[P[i]];
7:      mask ← mask >> 1;
8:  mask ← 00···0;
9:  FOR e ← 0 TO k DO
10:     R_e ← mask;
11:     mask ← rshift(mask);
12: mask ← 0···01;
13: FOR j ← 1 TO n DO
14:     R'_0 ← rshift(R_0) & B[T[j]]
15:     FOR e ← 1 TO k DO
16:         R'_e ← rshift(R_e) & B[T[j]] | rshift(R_{e-1} | R'_{e-1}) | R_{e-1};
17:     FOR e ← 0 TO k DO
18:         R_e ← R'_e;
19:     IF (R_k & mask) ≠ 0 THEN
20:         RETURN TRUE;    {occorrenza approssimata con ⩽ k errori di P che termina in j}
21: RETURN FALSE;
```

Le linee 1–7 sono analoghe a quelle di RICERCA⋄SHIFTAND e permettono di inizializzare le maschere B. Le successive linee 8–11 inizializzano le parole $R_e = 1^e 0^{m-e}$, per $0 \leqslant e \leqslant k$. Dopo aver inizializzato *mask* nella linea 12 per verificare l'ultimo bit di R_k (successivamente,

nella linea 19), inizia il ciclo for principale nelle linee 13–20 che realizza le operazioni logiche appena discusse nei quattro casi sopra.

La complessità dell'algoritmo è proporzionale a k volte quella di RICERCA⋄SHIFTAND, per cui richiede $O\big(k\lceil m/w\rceil n\big)$ tempo al caso pessimo e lo spazio occupato è $O(m+(\sigma+k)\lceil m/w\rceil)$ per la memorizzazione delle parole R_e. Nei casi pratici di recupero dei testi, solitamente vale $m = O(w)$, per cui il costo diventa $O(nk)$ tempo e $O(m+\sigma)$ spazio. Nel caso di pattern lunghi (come nelle sequenze biologiche) esistono altri metodi più efficaci.

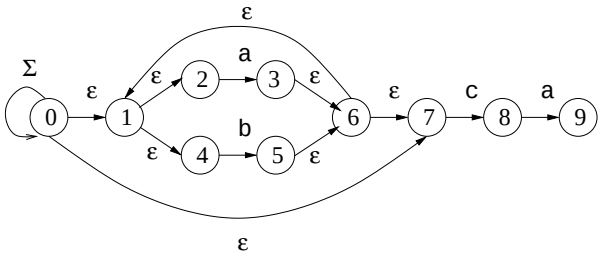
Concludiamo osservando che l'algoritmo RICERCAAPPROSSIMATA⋄SHIFTAND effettua una ricerca del pattern P ammettendo al più k errori in qualunque posizione di P. Ci sono situazioni in cui sono ammessi errori solo in alcune posizioni di P. Per esempio, volendo rintracciare un'automobile attraverso la sua targa `XY313WZ`, di cui però non ricordiamo se il numero è esattamente `313`, possiamo racchiudere tra parentesi angolari le posizioni del pattern $\langle$`XY`$\rangle$`313`$\langle$`WZ`$\rangle$ che non devono contenere errori. In generale, oltre al pattern possiamo specificare un insieme I di posizioni nel pattern in cui non si ammettono errori; nel nostro esempio, $I = \{1, 2, 6, 7\}$. Estendiamo di conseguenza l'algoritmo RICERCAAPPROSSIMATA⋄SHIFTAND, definendo una maschera binaria $M = m_1 \cdots m_m$ tale che $m_i = 1$ se e solo se $i \notin I$. La linea 16 dello pseudocodice diventa

16: $R'_e \leftarrow \mathit{rshift}(R_e)\,\&\,B\big[T[j]\big] \mid \big((\mathit{rshift}(R_{e-1} \mid R'_{e-1}) \mid R_{e-1}\big)\ \&\ M\big)$,

ovvero gli errori sono ammessi solo nelle posizioni in cui M vale 1, non coinvolgendo così le posizioni in I.

5.4. Estensione del paradigma ad `agrep`: espressioni regolari

Il paradigma SHIFT-AND è utile per le espressioni regolari viste nel Capitolo 3. Data un'espressione regolare r, costruiamo il suo automa non deterministico δ. Dalla Sezione 3.3, sappiamo come costruire l'automa per $r = (a|b)^*ca$, riportato di seguito (dove si assume una transizione dallo stato 0 a se stesso per ogni simbolo $c \in \Sigma$).



Come precedentemente notato, il non determinismo dell'automa risultante è dovuto alla presenza delle ϵ-transizioni, perché gli archi etichettati con caratteri di Σ non danno luogo a non determinismo. Valgono infatti le seguenti proprietà discusse in precedenza:

(1) esistono un solo stato iniziale e un solo stato finale;
(2) ogni stato (escluso quello inziale) ha un solo arco uscente etichettato con un carattere $c \in \Sigma$, oppure ha al più due ϵ-transizioni in uscita;
(3) il numero m di stati (escluso quello iniziale) è limitato superiormente da $2|r|$, dove $|r|$ è la lunghezza dell'espressione r calcolata come il numero di costanti e operatori che appaiono in r.

In particolare, utilizziamo tali proprietà per numerare gli stati da 0 a m, in modo tale che

- lo stato iniziale è 0;
- lo stato finale è m, dove $m \leqslant 2|r|$;

- in ogni coppia di stati collegati da una transizione etichettata con un simbolo di Σ, i due stati hanno numeri consecutivi (con la dovuta eccezione della transizione dallo stato 0 a se stesso).

La suddetta numerazione esiste sempre perché, se due stati sono collegati da una transizione etichettata con un simbolo dell'alfabeto Σ, allora non esistono altre transizioni che li collega (vedere la proprietà (2) di prima). Una semplice visita del grafo indotto dall'automa δ permette, quindi, di assegnare tale numerazione in tempo $O(m)$.

Per simulare l'automa δ con le tecniche descritte in questo capitolo, associamo una parola di memoria $R = r_0 \cdots r_m$ all'automa δ, in modo che il bit $r_i = 1$ se e solo se lo stato i è attivo (notare che il bit $r_0 = 1$ in quanto lo stato 0 è sempre attivo con qualunque carattere $c \in \Sigma$). Quando $r_m = 1$ abbiamo un'occorrenza dell'espressione regolare che termina nella posizione corrente del testo.

Possiamo quindi applicare il paradigma SHIFT-AND in cui, dopo aver letto il carattere $T[j]$ dal testo al passo $j = 1, 2, \ldots, n$, aggiorniamo R ottenendo il nuovo valore R'. A tal fine, simuliamo l'automa δ sulla falsariga di quanto riportato nella Sezione 3.4.

Come prima fase, dato l'insieme Q degli stati attivi (ossia $Q = \{i : 0 \leqslant i \leqslant m \text{ e } r_i = 1\}$ è ricavabile implicitamente da R), dobbiamo calcolare l'insieme Q' dei *prossimi stati* (come nell'algoritmo RICERCA⋄AUTOMAFINITONONDETERMINISTICO). A tale scopo, utilizziamo l'osservazione sopra che in ogni coppia di stati collegati da una transizione etichettata con un simbolo di Σ, i due stati hanno numeri consecutivi. Pertanto, possiamo costruire delle maschere B tali che, per ogni $c \in \Sigma$ e $1 \leqslant i \leqslant m$, vale $B[c]_i = 1$ (il bit in posizione i della maschera $B[c]$ è pari a 1) se e soltanto se esiste una transizione dallo stato $(i-1)$ allo stato i etichettata con il simbolo c. Nel nostro esempio, abbiamo le seguenti maschere:

	0123456789
...	1000000000
B[a]	1001000001
B[b]	1000010000

	0123456789
B[c]	1000000010
...	1000000000
...	1000000000

Notare che, in generale, ci possono essere più bit posti a 1 in una maschera $B[c]$ e che il primo bit è sempre pari a 1 perché lo stato 0 è sempre attivo. A questo punto, le transizioni multiple dell'automa a partire dagli stati in Q con la lettura del simbolo $T[j]$ possono essere gestite come le parole D' e R'_0 negli algoritmi precedenti. In altri termini,

$$R' \leftarrow \mathit{rshift}(R) \,\&\, B\big[T[j]\big].$$

produce l'effetto desiderato, ottenendo $Q' = \{i : 0 \leqslant i \leqslant m \text{ e } r'_i = 1\}$. Infatti, se uno stato $(i-1)$ è attivo e c'è una transizione etichettata con un simbolo $T[j]$, allora questa deve condurre allo stato i, secondo quanto discusso prima. L'operazione di *rshift* e il primo bit (a 1) di $B\big[T[j]\big]$ fanno sì che lo stato 0 rimanga sempre attivo.

Come seconda fase, dobbiamo determinare l'ϵ-*chiusura* di Q': la difficoltà concettuale risiede nella propagazione attraverso le ϵ-transizioni di δ a partire dagli stati in Q' (come nella procedura EPSILON(Q', δ) vista nella Sezione 3.4). L'idea di base è di calcolare in modo preliminare delle tabelle ausiliari per facilitare questo compito.

Costruiamo prima $m + 1$ maschere intermedie M^k, una per ogni stato $0 \leqslant k \leqslant m$. In particolare, il bit $M^k_i = 1$ segnala che dallo stato k possiamo raggiungere lo stato i attraverso zero o più ϵ-transizioni. Nel nostro esempio, abbiamo le seguenti maschere, osservando che M^7, M^8 e M^9 non hanno ϵ-transizioni e quindi la loro maschera è composta da tutti 0:

	0123456789
M^0	1110100100
M^1	0110100000

	0123456789
M^2	0010000000
M^3	0111101100
M^4	0000100000

	0123456789
M^5	0110111100
M^6	0110101100

Per esempio, M^3 rappresenta il fatto che gli stati raggiungibili con zero o più ϵ-transizioni a partire da 3 sono 1, 2, 3, 4, 6 e 7. Come elaborazione preliminare, fissiamo un parametro L e dividiamo gli stati in $\lceil m/L \rceil$ gruppi, ciascun gruppo composto da L stati numerati consecutivamente (a parte l'ultimo gruppo che può averne di meno). Nel nostro esempio, fissiamo $L = 3$, ottenendo i gruppi $\{0, 1, 2\}$, $\{3, 4, 5\}$, $\{6, 7, 8\}$ e $\{9\}$.

Per ciascun gruppo costruiamo una tabella di 2^L elementi, in corrispondenza di tutte le possibili configurazioni di quegli stati attivi o meno. Per esempio, per il primo gruppo, la configurazione di L bit `101` indica che gli stati attivi sono 0 e 2, mentre la stessa configurazione per il secondo gruppo indica che sono attivi gli stati 3 e 5. Di questi stati, facciamo l'or logico delle loro maschere ($M^0 \,|\, M^2 =$ `1110100100` e $M^3 \,|\, M^5 =$ `0111111100`) e le memorizziamo in posizione `101` (ossia posizione 3) della corrispondente tabella. In tal modo, intendiamo codificare il fatto che l'ϵ-chiusura dell'insieme di stati $\{0, 2\}$ è $\{0, 1, 2, 4, 7\}$ mentre quella di $\{3, 5\}$ è $\{1, 2, 3, 4, 5, 6, 7\}$. Indicando con T_i la tabella dell'i-esimo gruppo, otteniamo le seguenti tabelle T_1, T_2 e T_3 (la tabella T_4 è superflua perché l'unico stato nel quarto gruppo non ha ϵ-transizioni e quindi tutte le sue righe sono composte da soli bit pari a 0):

012	0123456789
000	0000000000
001	0010000000
010	0110100000
011	0110100000
100	1110100100
101	1110100100
110	1110100100
111	1110100100

T_1

345	0123456789
000	0000000000
001	0110111100
010	0000100000
011	0110111100
100	0111101100
101	0111111100
110	0111101100
111	0111111100

T_2

678	0123456789
000	0000000000
001	0000000000
010	0000000000
011	0000000000
100	0110101100
101	0110101100
110	0110101100
111	0110101100

T_3

Possiamo costruire ciascuna tabella T_i in $O(2^L \lceil m/w \rceil)$ tempo utilizzando le maschere intermedie M^k ed esaminando le 2^L configurazioni in ordine cresente di bit pari a 1 in modo che per ogni configurazione ce ne sia sempre una precedente che ha un bit 1 in meno (nel nostro esempio, `000`, `001`, `010`, `100`, `011`, `110`, `101`, `111`). Essendoci $\lceil m/L \rceil$ tali tabelle, abbiamo un costo totale di $O(2^L m^2/wL)$ tempo, che diventa $O(m^3/w \log m)$ fissando $L = \lfloor \log_2 m \rfloor$.

Possiamo finalmente calcolare l'ϵ-chiusura dell'insieme Q' utilizzando le suddette tabelle direttamente sulla parola R' calcolata nella prima fase. Spezziamo R' in $\lceil m/L \rceil$ gruppi di L bit consecutivi (l'ultimo gruppo può avere meno di L bit) e per l'i-esimo gruppo accediamo alla maschera memorizzata nella tabella T_i. Continuando il nostro esempio, se $R' =$ `1000010000` dopo aver letto i caratteri `aaab` nel testo, in quanto gli stati attivi sono quelli in $Q' = \{0, 5\}$, possiamo trovare l'ϵ-chiusura di Q' dividendo R' in quattro gruppi `100 001 000 0` ed effettuando l'or logico tra $T_1[100] =$ `1110100100`, $T_2[001] =$ `0110111100`, $T_3[000] = T_4[000] =$ `0000000000`, ottenendo così $R =$ `1110111100`, ossia il prossimo insieme di stati $Q = \{0, 1, 2, 4, 5, 6, 7\}$, come possiamo verificare direttamente dall'automa δ.

In sintesi, nella prima fase creiamo l'automa δ e le maschere B in $O(m\sigma)$ tempo e spazio e nella seconda fase costruiamo le tabelle T_i in $O(m^3/w \log m)$ tempo e spazio. Quindi la preelaborazione (*preprocessing*) dell'espressione regolare r richiede $O(m\sigma + m^3/w \log m)$ tempo e spazio. Per la scansione operiamo come descritto sopra, calcolandoci R' da R (prima fase) e poi ottenendo il nuovo R come ϵ-chiusura di R' (seconda fase). Per ciascuno degli n caratteri

del testo, impieghiamo $O(m/L \times m/w) = O(m^2/w \log m)$ tempo. Ipotizzando che $m \leqslant n$, ne deriva che il costo totale è $O(m\sigma + nm^2/w \log m)$, il quale diventa $O(m\sigma + nm/\log m)$ quando $m = O(w)$: in tal caso, le prestazioni sono migliori di quelle ottenute dall'algoritmo RICERCA◇AUTOMAFINITONONDETERMINISTICO. È possibile combinare gli approcci finora visti per cercare espressioni regolari con errori, ma tale argomento non viene discusso ulteriormente in queste dispense.

5.5. Considerazioni finali su `agrep`

L'applicazione `agrep`, molto diffusa nel campo della ricerca testuale, si basa in parte su algoritmi simili a quelli visti in questi capitoli. Il software progettato da Wu e Manber procede in base al tipo di pattern specificato. Se il pattern è esatto, usa una variante di Boyer-Moore (Sezione 2.4); altrimenti usa le tecniche descritte in questo capitolo (denominate `bitap`). Tuttavia, in alcuni casi, ne aggiunge di altre descritte di seguito.

Se vogliamo cercare un insieme di pattern, viene impiegata una variante di Boyer-Moore applicata a più pattern (denominata `mgrep`) in alternativa all'algoritmo di Aho-Corasick (Sezione 2.6). Viene calcolata una funzione hash applicata a blocchi di caratteri di lunghezza $b = \log_\sigma(2m)$, dove m è la lunghezza del pattern più corto. Viene quindi applicata una variante del metodo di *shift* nello stile di Boyer-Moore basandosi però sui b caratteri trattati come unità indivisibile. Se viene trovata una eventuale occorrenza dei b caratteri, allora viene calcolata la funzione hash su un intorno di m caratteri nel testo. La funziona hash seleziona un sottoinsieme dei pattern che contengono gli m caratteri e tali pattern vengono confrontati a uno a uno. In pratica questa ultima situazione occorre poche volte, per cui l'algoritmo complessivamente va più veloce di Aho-Corasick.

Se gli errori riguardano un solo pattern lungo, viene adottato un algoritmo (denominato `amonkey`) ancora in stile Boyer-Moore che opera con errori per filtrare il testo. Se il pattern è moderatamente corto, viene usato il trucco di partizionarlo in $k+1$ parti di eguale lunghezza; poichè siamo interessati alle occorrenze con al più k errori, almeno delle parti deve occorrere *senza* errori. Ne risulta un filtro di ricerca (denominato `mmonkey`) che è particolarmente efficace in quanto si basa sulla ricerca esatta. Una volta individuate le porzioni di testo in cui occorre almeno una delle $k+1$ parti del pattern, si procede con la ricerca con errori solo in un intorno di $m+k$ caratteri per quelle porzioni, risparmiando tempo considerevole. Viene inoltre adottato il metodo di "best match" (Sezione 4.5) in cui si trova il minimo $k \geqslant 0$ per cui c'è un'occorrenza del pattern con k errori.

Parte 2

STRUTTURE DI DATI PER TESTI
(ossia ricerca senza scansione del testo)

CAPITOLO 6

Liste Invertite

I documenti testuali presenti nei sistemi di *information retrieval* sono relativamente stabili e memorizzati in grandi quantità (terabyte e oltre). Per poter fornire una risposta veloce alle numerose ricerche degli utenti, è impensabile scandire tutti i documenti per ogni ricerca, utilizzando le tecniche viste nei capitoli precedenti: invece, conviene effettuare una fase di elaborazione preliminare per costruire delle strutture di dati, dette *indici testuali*, in cui le ricerche siano molto veloci, di fatto evitando di scandire l'intera collezione dei documenti a ogni ricerca. Il prezzo da pagare è l'utilizzo di ulteriore memoria per memorizzare gli indici testuali, ma il tempo di calcolo richiesto nella costruzione di tali indici ripaga ampiamente l'enorme miglioramento nella velocità di risposta delle ricerche degli utenti. Esistono due approcci principali alla costruzione di indici testuali:

- *wold-level text indexing*: ipotizza che si possa definire in modo preciso la nozione di *termine* (inteso come una parola o una sequenza massimale alfanumerica), in modo da partizionare ciascun documento in segmenti disgiunti, ciascuno corrispondente a un termine. In altre parole, un documento è visto come una sequenza di termini e la ricerca avviene specificando uno o più termini. In alcuni casi, i termini troppo comuni (*stop word*) vengono esclusi dall'indicizzazione. Le liste invertite (chiamante anche file invertiti, file dei *posting* o concordanze) sono un esempio di tale tipo di indici testuali che discutiamo in questo capitolo.
- *full text indexing*: non assume alcuna ipotesi sulla struttura dei documenti, che vengono visti semplicemente come sequenze di caratteri presi dall'alfabeto Σ, permettendo di cercare una *qualsiasi* sottostringa in esse. Un esempio di tale indice è l'albero dei suffissi o *suffix tree* (Capitolo **??**) che viene impiegato laddove non sia possibile partizionare un documento in una sequenza di termini. Per esempio, nelle sequenze biologiche e nelle lingue asiatiche, non è semplice trovare una regola per suddividere il testo in parole. Questo compito non è affatto banale ed è prono agli errori, sia che venga eseguito da persone che venga svolto in modo automatico da un calcolatore.

6.1. Organizzazione logica dei dati

In questo capitolo, definiamo le liste invertite per una collezione $\mathsf{D} = \{\mathsf{D}_1, \mathsf{D}_2, \ldots, \mathsf{D}_\mathsf{d}\}$ di documenti. Più che una struttura di dati possono essere viste come un'organizzazione logica dei dati, in quanto le componenti possono essere realizzate utilizzando strutture di dati differenti:

- il vocabolario o lessico V contiene l'insieme dei termini distinti che appaiono nei documenti in D;
- la lista invertita L_P, per ciascun termine $P \in V$, contiene un riferimento alla posizione iniziale di ciascuna occorrenza di P nei documenti in D: in altre parole, la lista per P contiene la coppia $\langle j, i \rangle$ se la sottostringa $D_j[i, i + |P| - 1]$ del documento $D_j \in D$ dà luogo a un termine che è proprio uguale a P.

Per esempio, assumendo che D contenga un solo documento D_1 il cui testo (con relativa numerazione dei suoi caratteri) è

```
sopra la panca la capra canta sotto la panca la capra crepa
          1         2         3         4         5
12345678901234567890123456789012345678901234567890123456789
```

possiamo osservare che il corrispondente vocabolario è $V = \{$`canta`, `capra`, `crepa`, `la`, `panca`, `sopra`, `sotto`$\}$. La lista invertita per il termine `capra` è $L_{\texttt{capra}} = \{19, 49\}$ mentre per il termine `la` è $L_{\texttt{la}} = \{6, 16, 37, 46\}$ (stiamo omettendo l'indice di D_1 in quanto è l'unico documento in D).

6.2. Memorizzazione e costruzione

Essendo ordinate al loro interno, le liste invertite sono spesso mantenute in forma compressa per ridurre lo spazio a circa il 10–25% del testo e, inoltre, le occorrenze sono spesso riferite al numero di linea piuttosto che alla posizione precisa nel documento. Nel nostro esempio, $L_{\texttt{la}} = \{7, 16, 37, 46\}$ si riferisce alle posizioni nel documento e viene rappresentata in modo differenziale come $L_{\texttt{la}} = \{7, 9, 21, 9\}$, sottraendo ciascun intero (tranne il primo) con quello immediatamente precedente. Per ricostruire gli elementi basta scandire la lista dal suo inizio e sommare. Per esempio, per ricostruire il terzo elemento di $L_{\texttt{la}}$ basta sommare $7+9+21 = 37$.

Il vantaggio di tale approccio è che gli interi generati sono piccoli e possono essere rappresentati con codici prefissi a lunghezza variabile, invece che mediante codifiche a lunghezza fissata di 32 o 64 bit. Due semplici modi per rappresentare un intero positivo x sono il codice gamma e quello delta.

Il *codice gamma* rappresenta x in due parti: la prima codifica il valore $e = \lfloor \log_2 x \rfloor$ usando una semplice rappresentazione unaria (ossia e copie di 0 seguite da un 1), a cui segue il valore $r = x - 2^{\lfloor \log_2 x \rfloor}$ usando la rappresentazione binaria su e bit (per un totale di $1+2e = 1+2\lfloor \log_2 x \rfloor$ bit). In altre parole, x viene visto come $x = 2^e + r$ per il massimo valore possibile di e e viene rappresentato come 0^e1 seguito dalla codifica binaria di r su e bit successivi. Per esempio, il codice gamma per $x = 1, 2, 3, 4, 5, \ldots$ è rispettivamente dato da $1, 01\,0, 01\,1, 001\,00, 001\,01, \ldots$. La lista $L_{\texttt{la}} = \{7, 9, 21, 9\}$ viene quindi rappresentata dalla sequenza binaria

$$\underbrace{001\,11}_{7=2^2+3}\ \underbrace{0001\,001}_{9=2^3+1}\ \underbrace{00001\,0101}_{21=2^4+5}\ \underbrace{0001\,001}_{9=2^3+1}$$

Notare che, essendo codice prefissi, possiamo decodificare in modo univoco la sequenza binaria così prodotta: partiamo dal primo bit e contiamo il numero e di zeri fino a incontrare il primo uno. Leggiamo quindi i successivi e bit che rappresentano il valore r. I $2e + 1$ bit così letti sono la rappresentazione binaria di $x = 2^e + r$. Riapplichiamo iterativamente tale procedimento sulla parte rimanente dei bit non ancora letti, fino a ultimare la lettura dell'intera sequenza.

Il *codice delta* richiede teoricamente meno bit del codice gamma (anche se in pratica non è sempre vero quando gli interi sono molto piccoli). Utilizza sempre la decomposizione di $x = 2^e + r$ per l'esponente e massimo, solo che utilizza il codice gamma (anziché quello unario) per rappresentare e, richiedendo in questo modo $1 + 2\lfloor \log_2 e \rfloor + e \leqslant 1 + 2\lfloor \log_2 \log_2 x \rfloor +$

$\lfloor \log_2 x \rfloor$ bit. Per esempio, il codice delta per $x = 1, 2, 3, 4, 5, \ldots$ è rispettivamente dato da $1, 010\,0, 010\,1, 011\,00, 011\,01, \ldots$. La lista $L_{la} = \{7, 9, 21, 9\}$ viene quindi rappresentata dalla sequenza binaria

$$\underbrace{010\,11}_{7=2^2+3}\ \underbrace{011\,001}_{9=2^3+1}\ \underbrace{00100\,0101}_{21=2^4+5}\ \underbrace{011\,001}_{9=2^3+1}$$

Per interi di grandi dimensioni, viene usato anche il codice allineato al byte. Preso l'intero $x \geqslant 1$, sia $b(x) \geqslant 1 + \lfloor \log_2 x \rfloor$ il minimo numero di bit per rappresentarlo in binario, tale che $b(x)$ è multiplo di 7 (ossia $b(x)$ è il minimo esponente e tale che $x < 2^e$ ed $e \mod 7 = 0$). I $b(x)$ bit che rappresentano x vengono partizionati in gruppi di 7 bit consecutivi. Ciascun gruppo viene memorizzato in un byte in cui il bit più significativo viene usato per indicare se tale gruppo è l'ultimo (bit a 1) oppure no (bit a 0). Per esempio, $x = 2007$ viene rappresentato con $b(x) = 14$ bit pari a 0001111 1010111. Quindi, la sua rappresentazione allineata richiede due byte 0 0001111 e 1 1010111, dove il primo bit (più significativo) viene detto *bit di continuazione*.

Esistono anche altri modi più sofisticati ed efficienti per rappresentare ciascuna lista invertita L_P, che memorizzano ulteriori informazioni come la sua lunghezza in quanto rappresenta la frequenza del termine P nei documenti in D.

La memorizzazione delle liste invertite prevede anche l'utilizzo di un dizionario (come una tabella hash, un albero di ricerca oppure una delle strutture di dati che vedremo nei prossimi capitoli) per memorizzare i termini P nel vocabolario V.

Per la costruzione, osserviamo che una semplice scansione dei documenti in D ci permette di effettuare una fase di parsing del testo per scomporlo in una sequenza di termini. Per ogni termine P così identificato, verifichiamo se P appartiene al vocabolario V (aggiornato durante la costruzione). Se $P \notin V$, creaiamo una lista vuota L_P e inseriamo P in V associandogli L_P. In ogni caso, prendiamo il documento corrente D_j e la posizione i di inizio del termine P in tale documento, in modo da aggiungere la coppia $\langle j, i \rangle$ in fondo alla lista L_P (così da mantenerla ordinata). Notare che, in una singola passata sui documenti in D, possiamo costruire sia V che le liste L_P per $P \in V$, anche in formato compresso. Se $t(m)$ indica il costo di ricerca e di inserimento di un termine nel vocabolario V e m è il numero totale di termini esaminati nei vari documenti di D, il costo totale è di $O\big(m\, t(m)\big)$ tempo.

Purtroppo, la grande dimensione dei dati coinvolti, permette raramente di usare soltanto la memoria principale. Anche se le capacità delle memorie aumentano rapidamente, la quantità di documenti da trattare aumenta di pari passo. È pertanto preferibile adottare uno schema a più passate, valido sia per la memoria principale che per quella secondaria:

(1) Scandisci i documenti in D e crea soltanto il vocabolario V (ma non le liste invertite).
(2) Scandisci il vocabolario V ed enumera i suoi termini coerentemente con il loro ordine lessicografico: ossia, per ogni coppia di termini $P_r, P_s \in V$, vale $P_r \leqslant P_s$ sse $r \leqslant s$.
(3) Scandisci nuovamente i documenti in D: per ogni termine P così esaminato, sia i la posizione della sua occorrenza e D_j il documento di appartenenza. Accedi a V e determina il numero lessicografico r associato a P (ossia $P \equiv P_r$). Appendi la tripla $\langle r, j, i \rangle$ in un file di appoggio F.
(4) Ordina il file F in modo lessicografico sulle triple. In questo modo, le triple referenti alla stesso termine P si troveranno in posizioni contigue di F ordinato. Inoltre, le occorrenze di P saranno ordinate prima in ordine di documento e, a parità di documento, in ordine di posizione all'interno del documento.
(5) Scandisci F a partire dalla prima tripla in esso. Per le triple corrispondenti allo stesso termine P (ossia con la stessa prima componente), crea la corrispondente lista invertita L_P e associala a P in V.

Indicando con $sort(\mathtt{m})$ il costo computazionale per ordinare $\mathtt{m}$ triple lessicograficamente, il costo totale dell'algoritmo di costruzione descritto sopra è pari a $\mathtt{O}\big(sort(\mathtt{m}) + \mathtt{m}\,\mathtt{t}(\mathtt{m})\big)$ tempo, in quanto il costo delle scansioni non supera $sort(\mathtt{m})$: solitamente, $sort(\mathtt{m})$ è il costo dominante dell'intera costruzione. Usando la variante del mergesort per la memoria esterna, in cui vengono fuse $\mathtt{R}$ parti ordinate di $\mathtt{F}$ alla volta, abbiamo che $sort(\mathtt{m}) = \mathtt{O}\big(\frac{\mathtt{m}}{\mathtt{B}} \log_{\mathtt{R}} \frac{\mathtt{m}}{\mathtt{B}}\big)$ operazioni di I/O, dove ciascuna operazione di I/O trasferisce un blocco di $\mathtt{B}$ triple da o verso la memoria estera. Notare che la complessità dell'ordinamento con una memoria principale di capacità $\mathtt{M}$ è $\Theta\big(\frac{\mathtt{m}}{\mathtt{B}} \log_{\mathtt{M}/\mathtt{B}} \frac{\mathtt{m}}{\mathtt{B}}\big)$ operazioni di I/O (equivale a scegliere $\mathtt{R} = (\mathtt{M}/\mathtt{B})^{\epsilon}$ per una costante $\epsilon \leqslant 1$). [1]

Il metodo di costruzione appena esposto ipotizza che $\mathtt{V}$ possa essere contenuto in memoria principale. Nel caso che ciò non sia possibile, è consigliabile eseguire l'algoritmo di costruzione in fasi. La prima fase costruisce solo i primi $\mathtt{h}$ termini di $\mathtt{V}$, dove $\mathtt{h}$ massimizza la porzione di $\mathtt{V}$ che può risiedere in memoria principale. Dopo la prima fase, abbiamo individuato i primi $\mathtt{h}$ termini di $\mathtt{V}$ e abbiamo costruito le relative liste invertite. Procediamo quindi con la seconda fase che si occupa dei secondi $\mathtt{h}$ termini in $\mathtt{V}$ e così via, utilizzando un numero di fasi sufficiente a costruire l'intero vocabolario e le sue liste invertite. Il costo precedente va quindi moltiplicato per il numero $\mathtt{O}(|\mathtt{V}|/\mathtt{h})$ delle fasi impiegate.

I moderni motori di ricerca impiegano un *cluster*, composto da un numero elevato di calcolatori, per costruire le liste invertite: utilizzano la vasta memoria principale distribuita tra tutti quei calcolatori, sia per il vocabolario che per le liste invertite. La compressione delle liste invertite è cruciale per poterle mantenere in memoria principale, allievando così il problema del tempo di accesso alla memoria secondaria (che è di diversi ordini di grandezza superiore a quello della memoria principale). Inoltre, il basso costo della memoria principale utilizzata e la grande quantità di calcolatori utilizzati rendono altamente probabile la possibilità di un errore di lettura o scrittura, di fatto richiedendo metodi di indicizzazione tolleranti ai guasti.

6.3. Operazioni di ricerca

È interessante notare che le tecniche di ricerca viste nei capitoli precedenti, pur richiedendo una completa scansione dei documenti, possono essere usare proficuamente insieme alle liste invertite. Iniziamo a esaminare le ricerche più semplici, riguardanti le richieste effettuate in diversi motori di ricerca, le quali prevedono l'uso di termini collegati tra loro mediante gli operatori booleani:

- l'operatore di congiunzione ($\mathtt{P}$ AND $\mathtt{Q}$) tra due termini prevede che entrambi i termini occorrano nel documento: tale operazione fornisce come risultato l'intersezione tra le liste invertite corrispondenti ai termini ($\mathtt{L}_{\mathtt{P}} \cap \mathtt{L}_{\mathtt{Q}}$);
- l'operatore di disgiunzione ($\mathtt{P}$ OR $\mathtt{Q}$) prevede che almeno uno dei termini occorra: tale operazione viene tradotta come unione (senza ripetizioni!) delle liste invertite corrispondenti ai termini ($\mathtt{L}_{\mathtt{P}} \cup \mathtt{L}_{\mathtt{Q}}$);
- l'operatore di negazione (NOT $\mathtt{P}$) indica che il termine non debba occorrere: equivale a prendere il complemento delle corrispondente lista invertita ($\overline{\mathtt{L}_{\mathtt{P}}}$).

È possibile usufruire anche dell'operatore di vicinanza (NEAR) come estensione dell'operatore di congiunzione, in cui viene espresso che i termini specificati occorrano a poche posizioni l'uno dall'altro. Le interrogazioni booleane possono essere composte come una qualunque

[1] La complessità computazionale in memoria esterna viene espressa in termini del numero di blocchi memorizzati su disco, dove ciascun blocco contiene $\mathtt{B}$ elementi. Per esempio, leggere un file di $\mathtt{m}$ elementi richiede $\Theta(\mathtt{m}/\mathtt{B})$ operazioni di I/O.

espressione, anche se le statistiche mostrano che la maggior parte delle interrogazioni nei motori di ricerca consiste nella ricerca di due termini connessi dall'operatore di congiunzione.

Le corrispondenti operazioni sulle liste invertite possono essere svolte efficientemente utilizzando una variante dell'algoritmo di fusione adoperato per il *mergesort*, essendo gà ordinate individualmente. La ricerca con la congiunzione (AND) calcolata come intersezione di L_P e L_Q richiede $O(|L_P| + |L_Q|)$ tempo: da notare però che possiamo effettuare l'intersezione attraverso la ricerca binaria di ogni elemento della lista più corta tra L_P e L_Q nella lista (ordinata) della più lunga. Quindi, se $m = \min\{|L_P|, |L_Q|\}$ e $n = \max\{|L_P|, |L_Q|\}$, il costo è pari a $O(m \log n)$ tempo, il quale è inferiore al costo $O(m + n)$ della fusione, quando m è molto minore di n. In generale, il costo ottimo per l'intersezione è pari $\Theta(m \log(n/m))$ che è sempre migliore sia di $O(m \log n)$ che di $O(m + n)$. Possiamo ottenere tale costo utilizzando degli opportuni alberi di ricerca (*finger search tree*).

Tale costo favorisce chiaramente le liste più corte e motiva la strategia di risoluzione delle ricerche in cui c'è la congiunzione di $t > 2$ termini (invece che di soli due): prima ordiniamo le t liste invertite dei termini in ordine crescente di frequenza (pari alla loro lunghezza); poi, calcoliamo l'intersezione tra le prime due liste e, dalla terza in poi, effettuiamo l'intersezione tra quella e il risultato parziale calcolato fino a quel momento con le precedenti liste già intersecate. Nel considerare anche le espressioni in disgiunzione (OR), procediamo in modo analogo per ordine di frequenza delle liste (intermedie o meno), utilizzando una stima basata sulla somma delle loro lunghezze.

Alternativamente le liste invertite possono essere mantenute ordinate in base a un ordine o un rango di importanza delle occorrenze (per esempio, il *PageRank* di Google se trattiamo documenti del Web). Se tale ordine è preservato in modo coerente attraverso tutte le liste, possiamo ancora applicare la fusione del *mergesort*, come sopra, terminando la scansione delle liste non appena raggiungiamo un numero sufficiente di occorrenze "rilevanti" per l'utente.

Concludiamo il capitolo, discutendo come ottenere delle ricerche più flessibili per il *word-level text indexing* utilizzando le tecniche viste nei capitoli precedenti per le ricerche approssimate e le espressioni regolari. Sfruttiamo il fatto che la dimensione del vocabolario, solitamente di poche decine o centinaia di megabyte, è decisamente inferiore a quella totale dei documenti in D, solitamente dell'ordine dei terabyte (cioè $|V| \ll \sum_{j=1}^{d} |D_j|$).

L'idea è quella di cercare un pattern P nei documenti in D, usando la distanza di edit oppure specificando un'espressione regolare, applicando gli algoritmi di scansione sequenziale (Capitoli 2–5) non direttamente ai documenti in D, ma piuttosto ai termini memorizzati in V. Questo passo concettuale è importante e rivitalizza le tecniche sequenziali viste nei capitoli precedenti: *le liste invertite permettono di rendere molto più veloci le ricerche riducendo il problema della scansione dei documenti in* D *al problema della scansione del loro vocabolario* V.

La scansione di un terabyte di dati richiede molte ore mentre quella di pochi megabyte necessita solo di pochi secondi. Per un dato pattern P, siano $P_1, P_2, \ldots, P_s$ i termini in V trovati attraverso la ricerca approssimata o mediante un'espressione regolare. Per ottenere tutte le occorrenze di P nei vari documenti di D è sufficiente fornire in uscita l'unione delle corrispondenti liste $L_{P_1} \cup L_{P_2} \cup \cdots L_{P_s}$ utilizzando le idee appena descritte sopra (per il calcolo dell'OR). L'idea è semplice ma molto efficace per come utilizza gli svariati concetti visti finora (programmazione dinamica, simulazione di automi e così via). Osserviamo che il risultato di tale ricerca è comunque una lista, per cui può essere combinata in un'espressione booleana con altri termini, come già discusso nel capitolo.

Un esempio dell'utilizzo flessibile di questo tipo di ricerca, che riduce al tempo stesso lo spazio richiesto dalle liste invertite, è realizzato in `glimpse`. L'idea di base segue le linee appena descritte: in più, per risparmiare spazio, ciascun documento di D viene diviso in

blocchi della stessa dimensione (tranne l'ultimo blocco, che può avere dimensione inferiore). In questo modo, gli interi memorizzati nelle liste sono ancora più piccoli perché indirizzano i blocchi piuttosto che le singole posizioni all'interno dei blocchi. Il risparmio in spazio deriva anche dal fatto che, se lo stesso termine P occorre più volte nello stesso blocco, memorizziamo il riferimento al blocco una sola volta in L_P. In questo modo, lo spazio totale richiesto dalle liste invertite si riduce ulteriormente a circa il 2–4% dello spazio richiesto dai documenti in D.

La controparte per tale vantaggio è che la ricerca di P nel vocabolario, dopo aver fornito l'unione delle liste $\mathsf{L}_{\mathsf{P}_1} \cup \mathsf{L}_{\mathsf{P}_2} \cup \cdots \mathsf{L}_{\mathsf{P}_s}$, richiede un'ulteriore fase di validazione per accedere ai blocchi ivi indicati. Ciascun blocco deve essere scandito sequenzialmente, per cercare effettivamente le occorrenze che soddisfano la ricerca di P. Il vantaggio di tale metodo è che, per ricerche in cui il numero di blocchi risultanti è relativamente basso, risparmia spazio mantenendo una velocità di ricerca paragonabile a quella delle liste invertite standard. Invece, non funziona bene con le ricerche che coinvolgono termini frequenti, in quanto c'è il rischio di selezionare un numero elevato di blocchi (che devono essere scanditi).

Nel prossimo capitolo esaminiamo delle strutture di dati che possono essere impiegate per memorizzare il vocabolario V. In tal modo, le ricerche flessibili, menzionate sopra, non necessitano più della completa scansione dei termini nell'intero vocabolario V, ma utilizzano questi indici costruiti sui termini dello "indice" V per rendere le ricerche ancora più veloci rispetto a quanto discusso finora.

CAPITOLO 7

Alberi Digitali di Ricerca

7.1. Trie per stringhe

Il *trie* o albero digitale è una struttura di dati largamente impiegata per memorizzare un insieme S di stringhe utilizzando la loro decomposizione come sequenze di caratteri (per esempio, S potrebbe essere il vocabolario V utilizzato nelle liste invertite descritte nel Capitolo 6). Il termine si pronuncia come la parola inglese *try* e deriva dalla parola inglese *retrieval* utilizzata per descrivere il recupero delle informazioni. I trie hanno innumerevoli applicazioni in quanto permettono la *ricerca per prefissi* di un pattern P, ossia permettono di identificare le stringhe dell'insieme S che hanno P come prefisso. Tale tipo ricerca svelerà tutta la sua potenza espressiva quando discuteremo il *full text indexing* (si vedano i Capitoli 6 e **??**).

I trie sono utilizzati nella compressione dei dati, nei compilatori e negli analizzatori sintattici. Servono a completare termini specificati solo in parte; per esempio, i comandi nella shell, le parole nella composizione dei testi, i numeri telefonici e i messaggi SMS nei telefoni cellulari, gli indirizzi del Web o della posta elettronica. Permettono la realizzazione efficiente di correttori ortografici, di analizzatori di linguaggio naturale, e di sistemi per il recupero di informazioni mediante basi di conoscenza. Supportano inoltre ricerche più complesse di quella per prefissi, come la ricerca con espressioni regolari e con errori. Permettono di individuare ripetizioni nelle stringhe (per esempio, utili nell'analisi degli stili di scrittura di vari autori), di recuperare le stringhe comprese in un certo intervallo e di trovare parole che occorrono a distanza limitata in un testo, quando i trie sono utilizzati in indici costruiti in modo automatico. Le loro prestazioni ne hanno favorito l'impiego anche nel trattamento di dati multidimensionali, nella elaborazione dei segnali e nelle telecomunicazioni. Per esempio sono utilmente impiegati nella codifica e decodifica dei messaggi, nella risoluzione dei conflitti nell'accesso ai canali e nell'istradamento veloce nei router di Internet.

Un esempio di trie, già incontrato nella Sezione 2.6, è lo scheletro dell'automa di Aho-Corasick per la ricerca simultanea di più stringhe (si veda la Figura 2.4, in alto). L'insieme di stringhe ivi memorizzate è dato da $\mathsf{S} = \{\texttt{ananas}, \texttt{anacardo}, \texttt{banana}, \texttt{nan}\}$. Come si può notare, partiamo dal nodo iniziale 0, chiamato *radice* nei trie, per distinguere le stringhe in base al loro primo carattere: `a`, `b` oppure `n`. Per ciascun carattere $\texttt{c} \in \{\texttt{a}, \texttt{b}, \texttt{c}\}$, avremo un gruppo di stringhe composto da tutte e sole quelle che hanno `c` come primo carattere. Preso uno di questi gruppi, ad esempio, quello composto dalle stringhe `ananas` e `anacardo` associate al carattere $\texttt{c} = \texttt{a}$, osserviamo che hanno in comune anche il secondo e il terzo carattere e

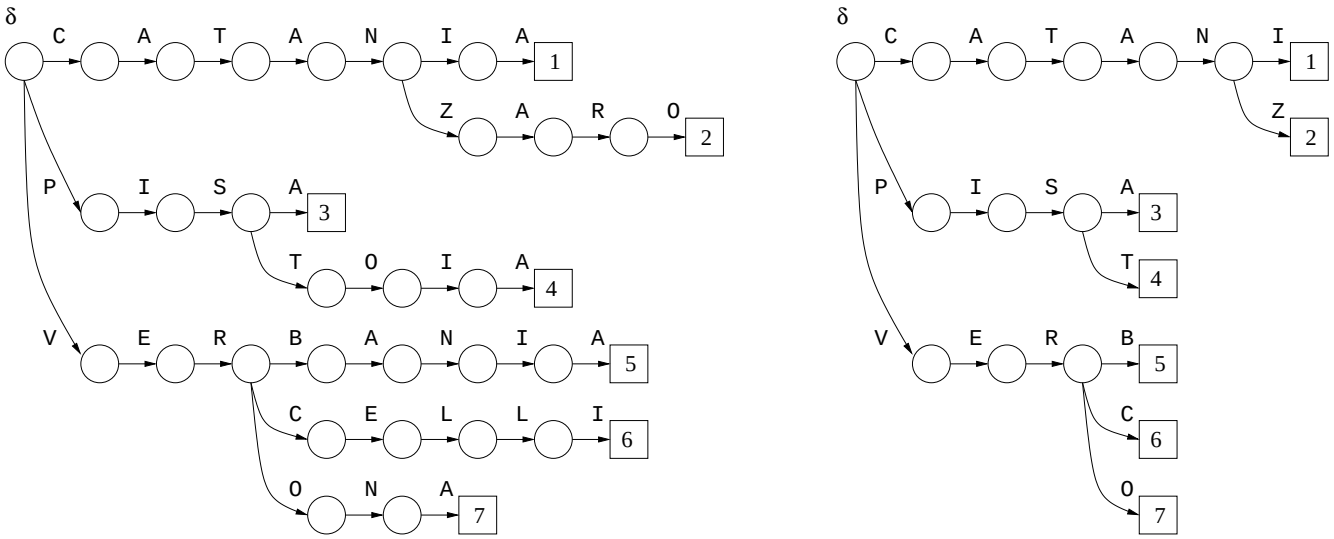


FIGURA 7.1. A sinistra, il trie per memorizzare i nomi di alcune province; a destra, la versione potata del trie.

si distinguono invece per il quarto carattere (c oppure n). Formiamo perciò dei sottogruppi aventi lo stesso carattere in quarta posizione e così via. In tal modo visitiamo i nodi interni del trie. Procediamo in questo modo fino ad ottenere gruppi vuoti, dopo aver raggiunto le foglie. Il trie è quindi chiamato anche albero digitale perché la sua struttura simula il comportamento di una ricerca digitale. Prima identifichiamo le parole che iniziano con un certo carattere. Poi, tra queste, identifichiamo quelle che hanno un certo carattere in seconda posizione, e così via.

In generale, assumiamo che le stringhe nell'insieme S non siano una prefisso dell'altra; se così non fosse, possiamo aggiungere all'alfabeto Σ un carattere speciale, il marcatore $\sqcup$, da utilizzare unicamente in fondo alle stringhe per far sì che soddisfino la nostra assunzione (tale marcatore ha la stessa funzione del terminatore di stringa '\0' nel linguaggio di programmazione C). Il trie δ per l'insieme S è un albero i cui i nodi hanno al più σ figli non vuoti, e la cui definizione è data ricorsivamente in termini di S:

(1) Se l'insieme S delle stringhe è vuoto, il trie δ è vuoto. Viene rappresentato con il simbolo NIL per indicare che è vuoto.
(2) Altrimenti, S non è vuoto e, per ogni carattere $c \in \Sigma$, sia S_c l'insieme delle stringhe in S aventi c come carattere iniziale. Sia S'_c l'insieme delle stringhe in S_c private del loro carattere iniziale c, comune a tutte. Il trie δ è quindi composto da un nodo chiamato *radice*, il cui c-esimo figlio è a sua volta il trie per le stringhe in S'_c, dove $c \in \Sigma$ (alcuni dei figli possono essere vuoti).

Mostriamo nella Figura 7.1, a sinistra, il trie δ corrispondente all'insieme S di stringhe composto da alcuni nomi di provincia:

1. CATANIA 2. CATANZARO 3. PISA 4. PISTOIA 5. VERBANIA 6. VERCELLI 7. VERONA

Ogni foglia del trie identifica univocamente una stringa in S e, viceversa, ogni stringa in S permette di individuare una foglia. Per esempio, la foglia 6 corrisponde a VERCELLI. In base a tale proprietà, le stringhe in S vengono enumerate e associate in modo univoco alle foglie di δ. In Figura 7.1, le foglie del trie sono numerate da 1 a 7, dove la foglia numero i memorizza il nome della i-esima provincia.

Coerentemente alla notazione sugli automi, dato un nodo u nel trie δ indichiamo con $\delta(u, c)$ il nodo raggiungibile da u mediante il carattere c. Se $\delta(u, c) = \text{NIL}$, allora il sottoalbero

corrispondente è vuoto. Un trie viene tipicamente caratterizzato da alcuni parametri. Innanzi tutto, viene assegnato un livello ai suoi nodi. La radice ha livello 0 e i suoi figli hanno livello 1. In generale, se un nodo ha livello $\mathtt{l}$ allora i suoi figli hanno livello $\mathtt{l}+1$: per stabilire quale tra i figli scegliere, bisogna guardare al carattere in posizione $\mathtt{l}+1$ delle stringhe.

Come si può notare dall'esempio, i trie esibiscono delle interessanti proprietà nonostante la semplicità con cui gestiscono le stringhe, viste come chiavi a lunghezza variabile:

- In generale, i nodi del trie rappresentato *prefissi* delle stringhe, e le stringhe memorizzate nelle foglie discendenti da un nodo $\mathtt{u}$ hanno in comune il prefisso associato a $\mathtt{u}$. Nel nostro esempio, le foglie discendenti dal nodo che memorizza il prefisso `PI` hanno associate le stringhe `PISA` e `PISTOIA`. In sintesi: cammino radice-nodo condiviso se e solo se prefisso comune nelle stringhe raggiungibili da quel cammino.
- La ricerca per prefissi risulta facilitata in quanto consiste nella individuazione di un nodo e delle sue foglie discendenti. La ricerca è guidata dalla stringa stessa e il tempo di ricerca è limitato superiormente dalla lunghezza della stringa da cercare e delle stringhe recuperate, invece che dalla dimensione del trie. I trie forniscono quindi un accesso veloce competitivo con l'hash e con gli alberi binari di ricerca. L'hash non permette inoltre la ricerca per prefissi.
- I trie preservano l'ordine delle stringhe, che possono essere elencate lessicograficamente attraverso una semplice visita simmetrica. La forma del trie è inoltre determinata univocamente dalle chiavi e non dal loro ordine di inserimento (contrariamente agli alberi di ricerca). Non sono necessarie operazioni di ristrutturazione per mantenere una qualche forma di bilanciamento. Forniscono quindi delle prestazioni interessanti al caso pessimo senza dover affrontare ristrutturazioni.

Il massimo tra i livelli del trie è denominato altezza. Si noti che l'altezza è data dalla lunghezza massima tra le stringhe. Nel nostro esempio, l'altezza del trie è 9 in quanto la stringa più lunga nell'insieme è `CATANZARO`. È stato dimostrato che, potando i sottoalberi che contengono una sola foglia, l'altezza media non dipende dalla lunghezza delle stringhe, ma è limitata superiormente da $2\log_\sigma|\mathsf{S}| + \mathsf{O}(1)$, dove $|\mathsf{S}|$ è il numero di stringhe nell'insieme S. Tale potatura assume che le stringhe siano memorizzate altrove, per cui è possibile ricostruire il sottoalbero potato in quanto è un filamento di nodi contenente la sequenza di caratteri finali di una delle stringhe (si veda Figura 7.1, a destra, per un esempio). Alternativamente, tali caratteri possono essere memorizzati nella foglia ottenuta dalla potatura. (Alcuni autori indicano con trie la sua versione potata.)

La dimensione del trie è il numero totale di nodi. Indicando con N la lunghezza totale delle stringhe in S, ovvero la somma delle loro lunghezze, abbiamo che la dimensione di un trie è al più $\mathsf{N}+1$. Tale valore viene raggiunto quando ciascuna stringa in S ha il primo carattere differente da quello delle altre, per cui il trie è composto da una radice e poi da $|\mathsf{S}|$ filamenti di nodi, ciascun filamento corrispondente a una stringa. Tuttavia, se si adotta la versione potata del trie, è stato dimostrato che la dimensione media non dipende dalla lunghezza delle stringhe e vale all'incira $1.44\,|\mathsf{S}|$.

I trie rappresentano una semplice forma di compressione dei prefissi comuni alle stringhe, in quanto parte delle chiavi sono codificate nella struttura stessa una sola volta. Le stringhe che hanno un prefisso di $\mathtt{k}$ caratteri in comune (come `VERBANIA` e `VERONA` con $\mathtt{k}=3$), condividono un cammino dalla radice al nodo di livello $\mathtt{k}$ che corrisponde a quel prefisso.

7.2. Rappresentazione in memoria

Purtroppo gli svantaggi dei trie sono che possono essere inefficienti dal punto di vista dello spazio occupato e che le prestazioni possono peggiorare se il linguaggio di programmazione adottato non rende disponibile un accesso efficiente ai singoli caratteri delle stringhe. Esistono diverse alternative per l'effettiva rappresentazione in memoria del trie δ, che in fondo può essere visto come un albero k-ario (generalizzazione dell'albero binario) in cui $k = \sigma$.

Rappresentazione tabulare. Il trie δ è una tabella con un numero di righe pari al numero di nodi in δ (al più $N+1$) e un numero di colonne uguale alla dimensione dell'alfabeto Σ. Il nodo generico è una delle righe, tipicamente rappresentata come un array di σ riferimenti, in cui il c-esimo riferimento punta al sottoalbero corrispondente al carattere $c \in \Sigma$. Un riferimento NIL indica che il sottoalbero è vuoto. La notazione $\delta(u, c)$ si traduce nel semplice accesso all'elemento della tabella che si trova nella riga u in posizione c. Il costo di tale operazione è costante se Σ è codificato come la sequenza dei numeri da 1 a Σ. Le righe corrispondenti alle foglie non sono strettamente necessarie e possono essere sostituite dalle stringhe dell'insieme S. Ad ogni modo, lo spazio occupato rimane proporzionale a $O(N\sigma)$.

Esistono tecniche sofisticate per risparmiare spazio cercando di rendere il più possibile compatta la tabella δ, che diventa un vettore unidimensionale in cui le varie righe sono porzioni contigue di σ elementi. Ogni riga viene perciò identificata dalla posizione del suo primo elemento all'interno del vettore. Preso un nodo, la sua riga viene vista come una sequenza di valori uguali o diversi da NIL al fine di sovrapporla parzialmente con le altre righe purché non vengano sovrapposti due elementi diversi da NIL per evitare inconsistenze (in maniera figurata, la riga è un pettine in cui i valori NIL sono i denti mancanti). In generale il problema di compattare i trie in questo modo è collegato al problema della superstringa comune più corta, che ricade nella classe dei problemi computazionalmente NP-ardui. Esistono però delle tecniche che offrono soluzioni approssimate soddisfacenti.

Rappresentazione mediante liste. Se Σ è particolarmente grande mentre i figli non vuoti nei vari nodi sono relativamente pochi, conviene mantenere in una lista concatenata i soli riferimenti ai figli non vuoti. Per ogni figlio f di u, se c è il carattere associato a f, la lista contiene la coppia (c, f) in cui c è la chiave di ricerca. Le foglie sono rappresentate mettendo il solo riferimento alla stringa corrispondente. La rappresentazione richiede meno spazio rispetto agli array, in quanto sono sufficienti $O(N)$ celle di memoria. Tuttavia, la notazione $\delta(u, c)$ si traduce in una scansione della lista del nodo u con la chiave c per trovare l'eventuale figlio associato al carattere c. Indicando con $F(u)$ il numero di figli del nodo u, il costo è $O(F(u)) = O(\sigma)$ tempo al caso pessimo.

Rappresentazione mediante alberi di ricerca. È un'idea analoga alla rappresentazione mediante liste solo che si usano alberi bilanciati con chiavi $c \in \Sigma$, o addirittura dei piccoli trie sulla rappresentazione binaria dei caratteri c. Lo spazio rimane $O(N)$ mentre il tempo si riduce a $O(\log F(u))$ oppure $O(\log \sigma)$ nel caso pessimo. Si noti che, pur essendo teoricamente migliore, tale soluzione è poco adottata in pratica per valori moderati di $F(u)$.

Rappresentazione mediante tabelle hash. È un'idea analoga alla rappresentazione mediante liste solo che i nodi sono mantenuti in una tabella hash. È possibile usare una tabella hash per nodo u usando i caratteri c come chiavi di ricerca, oppure prevedere una tabella globale in cui le coppie (u, c) sono le chiavi di ricerca. Lo spazio rimane proporzionale a $O(N)$. La notazione $\delta(u, c)$ viene tradotta in un accesso alla tabella hash in tempo medio costante (a meno di usare lo hash perfetto che richiede tempo costante al caso pessimo). Tuttavia, non è possibile memorizzare i figli in ordine lessicografico secondo il carattere c associato. In pratica si usa la tabella globale, e quindi la scansione dei figli di un nodo u richiede la scansione

completa della tabella in $O(N)$ tempo invece che $O(F(u))$ tempo.

Rappresentazione senza puntatori o succinta. Nel caso di trie statici, i nodi sono memorizzati a livelli da sinistra a destra a partire dal livello 0 della radice. Per ogni livello indichiamo esplicitamente il primo nodo, mentre gli altri nodo sullo stesso livello indicano solo la loro distanza come numero di nodi rispetto al primo. Lo spazio si riduce ulteriormente ma la realizzazione di $\delta(u, c)$ può essere piuttosto costosa. Esistono tecniche più sofisticate per rappresentare un trie binario mediante una sequenza di $O(N)$ bit in cui le operazioni di attraversamento da padre a figli e viceversa richiedono tempo costante. [1]

Rappresentazione ibrida. È particolarmente efficiente in pratica. Nei livelli vicino alla radice si usa la rappresentazione tabulare o mediante tabelle hash; nei livelli intermedi si usa la rappresentazione mediante liste; infine, nei livelli vicini alle foglie si usano degli array compatti. Precisamente, i nodi vicino alla radice hanno un numero di figli elevato per cui la rappresentazione tabulare, che permette l'accesso più veloce, non spreca troppo spazio. Nel livelli intermedi, le liste concatenate sono composte da piccoli array compatti di taglia prefissata. Quando un array si riempie, ne viene allocato un altro da aggiungere alla lista. I nodi vicini alle foglie usano singoli array compatti di taglia prefissata. Quando l'array si riempie, viene gestito a lista come nei livelli intermedi.

7.3. Operazioni sui trie

La ricerca per prefissi di una stringa pattern P di lunghezza m nell'insieme di stringhe S attraverso il trie δ ricalca la definizione ricorsiva di trie. Partiamo dalla radice per decidere quale figlio scegliere a livello $i = 1$. Al generico passo in cui dobbiamo scegliere un figlio a livello i del nodo corrente u, esaminiamo il carattere $c = P[i]$. Se il c-esimo sottoalbero del nodo corrente non è vuoto, passiamo a visitate il nodo $\delta(u, c)$ incrementando il livello $i = i + 1$. Se invece il sottoalbero è vuoto, la ricerca termina con fallimento. Se $i = m$, abbiamo esaminato tutta la stringa P con successo arrivando al nodo u. Tutte le stringhe in S che hanno P come prefisso sono memorizzate nelle foglie discendenti da u e possono essere recuperate tramite una semplice visita. Diamo lo pseudocodice per l'identificazione del nodo u in quanto la visita del sottoalbero radicato in u è standard.

RICERCA◇TRIE(P, δ):

```
1: u ← root(δ); m ← |P|;
2: FOR i ← 1 TO m DO
3:    IF δ(u, P[i]) = NIL THEN
4:       RETURN FALSE;
5:    ELSE
6:       u ← δ(u, P[i]);
7: RETURN TRUE;    {le stringhe discendono dal nodo u}
```

È interessante notare che, in caso di fallimento alla linea 4, abbiamo che il prefisso corrispondente al nodo u rappresenta il più lungo prefisso di P che è comune a una o più stringhe di S. Inoltre le ricerche con fallimento di stringhe sufficientemente lunghe sono più veloci delle ricerche mediante tabelle hash in quanto la ricerca nei trie si ferma non appena trova un prefisso di P che non occorre nel trie, mentre la funzione hash va comunque calcolata esaminando tutta la stringa.

[1] Per il lettore interessato, la rappresentazione succinta degli alberi (e quindi dei trie) è descritta nel Capitolo 4 del libro "Strutture di dati e algoritmi: progettazione, analisi e visualizzazione" di P. Crescenzi, G. Gambosi, R. Grossi, ed. Pearson Addison-Wesley, 2006.

Non è difficile calcolare il costo della ricerca, in quanto vengono visitati $O(m)$ nodi invocando il metodo che realizza $\delta(u, c)$. Assumendo che il suo calcolo richieda tempo costante, abbiamo $O(m)$ tempo di ricerca. Per alfabeti molto grandi, il calcolo richiede $O(\log \sigma)$ tempo e il costo diventa $O(m \log \sigma)$. Il numero di nodi visitati oltre alla radice non eccede l'altezza del trie.

Diamo un piccolo esempio quantitativo sulla velocità di ricerca dei trie. Assumiamo di volere memorizzare i codici fiscali in un trie. Ricordiamo che un codice fiscale contiene 9 lettere prese dall'alfabeto `A` ... `Z` di 26 caratteri, e 7 cifre prese dall'alfabeto `0` ... `9` di 10 caratteri, per un totale di $26^9 \times 10^7$ possibili codici fiscali. Cercare un codice fiscale in un trie richiede di attraversare al più 16 nodi, indipendentemente dal numero di codici fiscali memorizzati nel trie, in quanto l'altezza del trie è 16. In alternativa ai trie potremmo usare la ricerca binaria, per esempio, ma avremmo una dipendenza dal numero di chiavi. Nel caso estremo, memorizzando metà dei possibili codici fiscali in un array ordinato, la ricerca binaria richiederebbe circa $\log_2(26^9 \times 10^7) - 1 \geqslant 64$ confronti tra chiavi!

I trie possono essere mantenuti per un insieme dinamico S di stringhe, in cui vogliamo inserire e cancellare stringhe oltre che a cercarle. L'inserzione di una nuova stringa P nel trie δ cerca prima il suo più lungo prefisso x che occorre in un nodo u di δ analogamente alla procedura RICERCA◇TRIE. Decompone quindi P come $P = xy$, e sostituisce il sottoalbero vuoto raggiunto con la ricerca di x mettendo al suo posto il trie per y. Se P occorre interamente come prefisso, abbiamo che $P = x$ e poniamo $y = \sqcup$ per poter associare una foglia a P, nel caso P non sia stata già inserita in precedenza.

INSERZIONE◇TRIE(P, δ):

```
 1: u ← root(δ); m ← |P|;
 2: FOR i ← 1 TO m DO
 3:    IF δ(u, P[i]) = NIL THEN    {Il prefisso x occorre in u, dove P = xy}
 4:       WHILE i ⩽ m DO    {Crea il trie per y}
 5:          alloca un nuovo nodo v;
 6:          δ(u, P[i]) ← v;
 7:          i ← i + 1;
 8:          u ← v;
 9:       RETURN FALSE;
10:    ELSE
11:       u ← δ(u, P[i]);
12: IF u è nodo interno THEN    {P occorre come prefisso x, quindi y = ⊔}
13:    alloca una nuova foglia v etichettata con P;
14:    δ(u, ⊔) ← v;
15:    RETURN FALSE;
16: ELSE
17:    RETURN TRUE;    {P è già nella foglia u}
```

Il costo dell'inserzione è $O(m)$ tempo assumendo che il calcolo e l'inizializzazione di $\delta(u, c)$ richiedano tempo costante. La cancellazione di P viene trattata in modo semplice. Cerchiamo la foglia corrispondente a P, e cancelliamo la foglia e i suoi antenati fino a che non giungiamo a un antenato che ha più di un figlio. La procedura termina riducendo di uno il numero di figli di tale antenato. Il costo è $O(m)$.

La duttilità dei trie si esprime attraverso vari tipi di ricerca oltre a quella per prefissi. Abbiamo visto finora che permettono di trovare il prefisso più lungo del pattern P, utilizzato per l'accesso veloce alle informazioni di istradamento (IP lookup) nei router e nella gestione

della *cache* quando si accede a vari indirizzi Web. Discutiamo altri tipi di ricerche.

Nella ricerca per *intervalli*, date due stringhe P_1 e P_2, vogliamo identificare tutte le stringhe in S che sono lessicograficamente comprese tra queste due stringhe. Utilizzando i trie, cerchiamo separatamente le due stringhe pervenendo a due nodi u_1 e u_2, indipendentemente dal fatto che le ricerche terminino con successo o meno. Quindi forniamo in uscita le stringhe che sono nelle foglie comprese tra u_1 e u_2 secondo una visita anticipata del trie. Si noti che la visita può essere eseguita attraversando solo i nodi che sono antenati di tali foglie.

Nella ricerca simultanea di un *insieme* C di chiavi nell'insieme S, utilizziamo non solo il trie δ per S, ma costruiamo anche il trie δ_{C} per C. Invece di cercare le stringhe in C separatamente, sovrapponiamo i due trie δ e δ_{C} in tempo proporzionale alla dimensione di δ_{C}. Nella sovrapposizione (virtuale!) dei due trie, le foglie di δ che discendono dalle foglie di δ_{C} rappresentano le stringhe da fornire in uscita. Anche se il miglioramento asintotico della complessità non è evidente, si migliorano le prestazioni in quanto solitamente δ è di dimensioni considerevoli, per cui visitando i suoi nodi una sola volta riduciamo il numero di *cache miss* nel processore. Tale metodo può essere applicato anche alle espressioni regolari che rappresentano un piccolo insieme di stringhe.

Nella ricerca di *espressioni regolari* r, in generale, costruiamo un automa δ_{r} come discusso nel Capitolo 3. Invece di scandire le stringhe di S a una a una con δ_{r} per vedere se un loro prefisso soddisfa l'espressione regolare r, applichiamo una visita cumulativa al trie δ_{t} per S utilizzando una pila. L'idea principale è di associare uno o più stati dell'automa δ_{r} a ciascun nodo del trie δ_{t} e di metterli nella pila man mano che procediamo nella visita del trie. Quando incappiamo in un nodo u del trie a cui è associato uno stato finale dell'automa, forniamo in uscita tutte le stringhe memorizzate nelle foglie discendenti da u e non procediamo oltre con l'automa nei nodi discendenti da u. Altrimenti, quando passiamo dal nodo u a un suo figlio $\delta_{\mathsf{t}}(u, \mathsf{c})$, facciamo avanzare l'automa δ_{r} come se avessimo letto c nel testo. Mettiamo gli stati associati a u nella pila e associamo a tale figlio i nuovi stati attivi dopo la lettura di c. Quando risaliamo da un figlio a suo padre u, ripristiniamo gli stati di u dalla pila usata per la visita. Se l'automa è deterministico, abbiamo uno stato per nodo nella pila. In generale, la pila deve avere capacità massima pari all'altezza del trie moltiplicata per il massimo numero di stati che possono essere associati a un nodo. Il vantaggio di tale metodo è che non tutti i nodi vengono necessariamente visitati; in media, il loro numero è sublineare nella dimensione del trie. Il punto cruciale è che la scansione dell'automa sui prefissi comuni associati ai nodi del trie δ_{t} viene effettuata una sola volta, invece che essere ripetuta ogni volta per singola stringa. Tale vantaggio deriva dalla proprietà dei trie di comprimere, in un certo senso, i prefissi comuni.

Per completezza, forniamo alcuni particolari dei passi descritti con lo pseudocodice riportato di seguito. Facciamo uso della procedura Epsilon descritta nella Sezione 3.4, e di una procedura ausiliare VisitaFoglie(u) che effettua la visita del trie a partire dal ndo u, recuperando le foglie discendenti da u. Assumiamo che l'automa δ_{r} abbia stato iniziale 0 senza ϵ-transizioni a se stesso, e rimuoviamo dallo stato finale le transizioni in quanto non sono necessarie perché vogliamo trovare solo i prefissi che soddisfano r.

RicercaEspressioniRegolariTrie($\delta_{\mathsf{r}}, \delta_{\mathsf{t}}$):

1: $\mathsf{Q} \leftarrow \{0\}$; $\quad$ {*stato iniziale di* δ_{r}}
2: $u \leftarrow root(\delta_{\mathsf{t}})$;
3: RicercaRicorsiva(Q, u);

La procedura RicercaRicorsiva assume che gli stati attivi dell'automa δ_{r} sono in Q e vanno associati al nodo u del trie δ_{t}. Ogni invocazione di RicercaRicorsiva salva Q nella pila di sistema per le chiamate ricorsive, in modo che risalendo da figlio in padre ritrova

l'insieme Q degli stati associati al padre.

RicercaRicorsiva(Q, u):

```
1: IF Q contiene lo stato finale di δ_r THEN
2:    VisitaFoglie(u);
3: ELSE
4:    FOREACH carattere c ∈ Σ tale che δ_t(u, c) ≠ NIL DO
5:       v ← δ_t(u, c);    {figlio non vuoto di u con etichetta c}
6:       Q' ← {};    {insieme vuoto}
7:       FOREACH s ∈ Q DO
8:          Q' ← Q' ∪ δ_r(s, c);    {transizioni dell'automa δ_r}
9:       Q' ← Epsilon(Q', δ_r);
10:      RicercaRicorsiva(Q', v);
```

Nella ricerca con *errori*, siamo interessati a trovare le stringhe in S che hanno un prefisso che differisce dal pattern P per al più k errori (si veda il Capitolo 4 per la nozione di errore). L'idea è simile a quella delle espressioni regolari, per cui effettuiamo una visita del trie δ associando a ogni nodo u l'ultima colonna della tabella D_u di programmazione dinamica calcolata tra P e il prefisso associato al nodo u. Se la colonna contiene un valore minore o uguale a k in ultima posizione, le stringhe associate alle foglie discendenti da u hanno un prefisso che differisce da P per al più k errori. Sospendiamo il calcolo della programmazione dinamica e visitiamo il sottoalbero di u, riprendendo il calcolo senza attraversare i discendenti di u. Altrimenti, la colonna associata a u viene messa nella pila quando visitiamo un figlio $v = \delta(u, c)$ di u. Tuttavia, per calcolare la nuova tabella D_v per il prefisso associato a v, dobbiamo aggiungere una sola colonna a D_u in quanto abbiamo esteso il prefisso di u con il carattere c per ottenere il prefisso associato a v. Anche qui stiamo sfruttando la proprietà dei trie di far condividere prefissi comuni, per cui non ricalcoliamo ogni volta la tabella di programmazione dinamica. Quando risaliamo dal figlio al nodo u, ripristiniamo la colonna associata a u dalla pila. Si noti che è sufficiente attraversare i nodi u di livello al più $m + k$ per il calcolo delle tabelle D_u.

Nella ricerca *ortografica* effettuata nei correttori ortografici, vogliamo verificare che il pattern P sia una parola corretta dal punto di vista ortografico e, nel caso non lo sia, vogliamo individuare in S le parole che potrebbero essere la forma corretta di P. Dagli esperimenti effettuati dai linguisti computazionali risulta che l'80% degli errori ortografici sono dovuti a $k = 1$ errori. In tal caso, possiamo usare la tecnica descritta sopra per le ricerche con al più k errori (il programma `aspell` procede più o meno in questo modo). Esistono però anche errori fonetici, per esempio nella lingua inglese. A tale proposito è interessante vedere come funziona il SOUNDEX, un metodo brevettato nel 1922 per trasformare parole inglesi con suono simile in stringhe uguali. Il codice SOUNDEX consiste nella prima lettera delle stringhe da codificare seguita da una sequenza di cifre (spesso troncata a lunghezza 3). Le cifre sono assegnate alle lettere secondo la tabella seguente:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	0	1	2	0	0	2	2	4	5	5	0	1	2	6	2	3	0	1	0	2	0	2

Gli zeri sono quindi rimossi e le cifre ripetute sono ridotte a una singola cifra. Per esempio, il codice SOUNDEX per `ecxsample` è `e22205140` $\Rightarrow$ `e2514` e quello per `ecxample` è `e2205140` $\Rightarrow$ `e2514`. Memorizzando i codici SOUNDEX delle parole in un trie siamo in grado di trovare le parole con un suono simile. Quando cerchiamo una parola nel trie, calcoliamo il suo codice SOUNDEX e lo cerchiamo nel trie. In alternativa, possiamo costruire il trie direttamente sulle parole, calcolandoci via via il codice SOUNDEX dei suoi prefissi man mano che lo attraversiamo, anche se è meno efficiente.

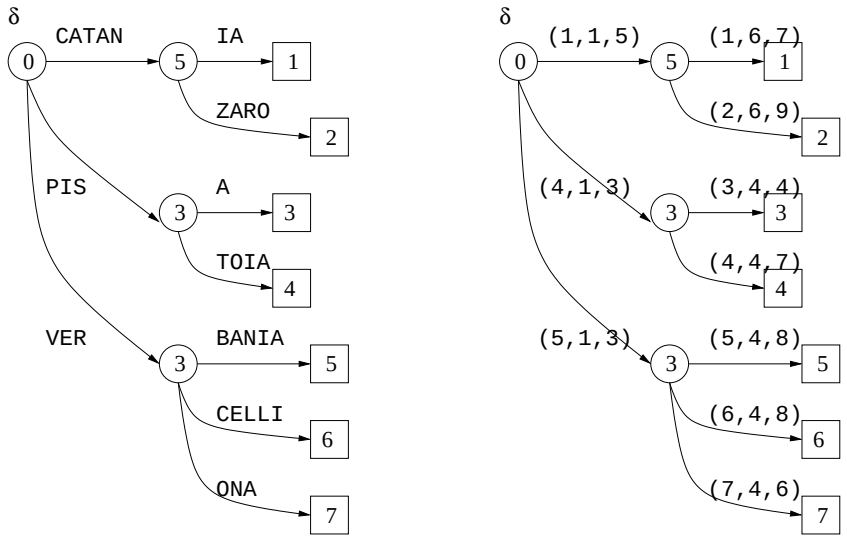


FIGURA 7.2. A sinistra, il trie compatto per memorizzare i nomi di alcune province; a destra, la versione con le sottostringhe rappresentate mediante triple.

7.4. Trie compatti

I trie discussi finora hanno in realtà alcuni nodi che non sono strettamente necessari. Precisamente, i nodi con un solo figlio rappresentano una scelta obbligata e non raffinano ulteriormente la ricerca, al contrario dei nodi aventi almeno due figli. Il *trie compatto* è informalmente ottenuto dal trie per compattazione, ovvero rimozione dei nodi con un solo figlio. Dopo tale operazione, gli archi sono etichettati con sottostringhe invece che singoli caratteri. Un esempio di trie compattato per l'insieme S

1. CATANIA 2. CATANZARO 3. PISA 4. PISTOIA 5. VERBANIA 6. VERCELLI 7. VERONA

è mostrato in Figura 7.2. Formalmente classifichiamo un nodo di un trie come *unario* se ha un solo figlio. Una *catena* di nodi unari è una sequenza massimale $u_1, \ldots, u_k$ di nodi nel trie tale che u_i è l'unico figlio di u_{i-1} per $2 \leqslant i \leqslant k$. Si noti che u_1 potrebbe essere la radice, oppure u_k potrebbe essere una foglia.

Definiamo l'operazione di compattazione di una catena $u_1, \ldots, u_k$ come segue. Sia α la sottostringa ottenuta concatenando i caratteri nella catena incontrati percorrendo gli archi da u_1 a u_k. Sostituiamo l'intera catena con la coppia di nodi u_1 e u_k collegati da un *singolo* arco etichettato con la sottostringa α. Il nodo u_k viene inoltre esteso così da contenere la lunghezza di α. Fanno eccezione la radice (etichettata con 0 per rappresentare la sottostringa vuota e a cui è permesso di avere un solo figlio) e le foglie (etichettate con le stringhe che memorizzano). Il trie compatto è ottenuto dal trie applicando l'operazione di compattazione a tutte le catene. Si veda Figura 7.2 per un esempio di trie compattato.

Nella rappresentazione di un trie compatto le sottostringhe negli archi sono rappresentate per mezzo di triple (i, j, k), a indicare che la sottostringa appare nella stringa numero i dalla posizione j alla posizione k. Il vantaggio di tale rappresentazione è che richiede spazio costante indipendentemente dalla lunghezza della sottostringa, purché le stringhe nell'insieme S siano memorizzata a parte. Si veda Figura 7.2 per un esempio di tale rappresentazione.

Adottiamo la convenzione di mantenere ordinati gli archi che partono dallo stesso nodo, in base al primo carattere della loro etichetta. Tali caratteri sono distinti perché lo erano anche nel trie prima della compattazione. In questo modo, una visita simmetrica fornisce le stringhe in ordine lessicografico. Vale inoltre la proprietà che un nodo u diverso dalle foglie appare nel

trie compatto se e solo se, prendendo il prefisso $\mathtt{p}$ associato a $\mathtt{u}$, entrambi $\mathtt{pa}$ e $\mathtt{pb}$ sono prefissi delle stringhe in $\mathtt{S}$ per almeno due caratteri $\mathtt{a} \neq \mathtt{b}$ dell'alfabeto (incluso il marcatore $\sqcup$).

Riassumendo, ciascuna foglia nel trie compatto memorizza la stringa ad essa associata. Ciascun nodo interno $\mathtt{u}$ ha almeno due figli (tranne la radice che può averne uno) e memorizza le seguenti informazioni:

- Un campo chiamato valore di *skip*, per memorizzare la lunghezza della sottostringa che etichetta l'arco che collega il padre di $\mathtt{u}$ a $\mathtt{u}$ stesso. Se $\mathtt{u}$ è la radice, tale valore è 0.
- Un insieme di $\mathtt{F}(\mathtt{u})$ archi etichettati, dove $\mathtt{F}(\mathtt{u}) \geqslant 2$ è il numero di figli di $\mathtt{u}$. Ciascun arco etichettato consiste in una stringa rappresentata come $(\mathtt{i}, \mathtt{j}, \mathtt{k})$ e in un puntatore al figlio. Gli archi sono mantenuti ordinati in base al primo carattere delle sottostringhe.

È interessante notare che lo spazio richiesto dal trie compatto per un insieme $\mathtt{S}$ di stringhe di lunghezza totale $\mathtt{N}$ è $\mathtt{O}(|\mathtt{S}|)$ invece che $\mathtt{O}(\mathtt{N})$ al caso pessimo; dipende cioè solo dal numero delle stringhe e non dalla somma delle loro lunghezze, contrariamente al trie. Tale proprietà deriva dal fatto che, in un albero avente i nodi interni con almeno due figli ciascuno, il numero di nodi interni è inferiore al numero di foglie. Essendoci $|\mathtt{S}|$ foglie nel trie compatto, la sua dimensione definita come numero totale di nodi è al più $2|\mathtt{S}|$.

7.5. Operazioni sui trie compatti

La ricerca di una stringa $\mathtt{P}$ di lunghezza $\mathtt{m}$ in un trie compatto simula la ricerca in un trie. A tutti gli effetti, quando attraversiamo un arco etichettato con una sottostringa, stiamo attraversando implicitamente la catena di nodi nel trie prima della compattazione. L'unico accorgimento è che la sottostringa va di volta in volta ricostruita attraverso la sua rappresentazione come tripla. Dato un nodo $\mathtt{u}$ come scegliere la sottostringa giusta con cui confrontare una parte di $\mathtt{P}$? Nel trie guardiamo i singoli caratteri che etichettano gli archi uscenti da $\mathtt{u}$. Nel trie compatto, sappiamo che tali caratteri appaiono come caratteri iniziali delle sottostringhe. Essendo distinti e ordinati, possiamo individuare la sottostringa opportuna con un singolo carattere di $\mathtt{P}$. In altre parole, se siamo arrivati al nodo $\mathtt{u}$ con il carattere $\mathtt{c} = \mathtt{P}[\mathtt{i}]$, scegliamo l'arco uscente da $\mathtt{u}$ che ha come primo carattere $\mathtt{c}$. Se tale arco non esiste, possiamo concludere la ricerca con un fallimento. Se esiste, l'arco è univocamente identificato. Allora confrontiamo il resto della sottostringa a lui associata con i caratteri di $\mathtt{P}$ a partire dalla posizione $\mathtt{i}+1$. Se l'intera sottostringa viene confrontata con successo oppure i caratteri di $\mathtt{P}$ sono tutti esaminati, raggiungiamo il figlio di $\mathtt{u}$ indicato dall'arco. Altrimenti, possiamo concludere la ricerca con un fallimento. Se alla fine la ricerca ha esito positivo, raggiungiamo un nodo $\mathtt{u}$ tale che il prefisso associato a $\mathtt{u}$ ha $\mathtt{P}$ come prefisso (non è detto che debbano essere uguali; per esempio si prenda $\mathtt{P} = \mathtt{VERB}$ nel trie compatto di Figura 7.2). Le stringhe in $\mathtt{S}$ che hanno prefisso $\mathtt{P}$ sono quelle memorizzate nelle foglie discendenti da $\mathtt{u}$. Il costo della ricerca è $\mathtt{O}(\mathtt{m})$ se $\sigma = \mathtt{O}(1)$; altrimenti è $\mathtt{O}(\mathtt{m} \log \sigma)$.

Nella ricerca appena descritta, in caso di successo confrontiamo $\mathtt{m}$ caratteri del pattern $\mathtt{P}$ e $\mathtt{h} \leqslant \mathtt{m}+1$ nodi per localizzare $\mathtt{u}$. In certi contesti, pur sapendo che $\mathtt{P}$ occorre nel trie compatto, non conosciamo quale è il corrispondente nodo $\mathtt{u}$. Per localizzare il nodo $\mathtt{u}$, possiamo impiegare un numero inferiore di confronti rispetto alla ricerca descritta sopra. In effetti, essendo certi a priori che $\mathtt{P}$ occorre, possiamo raggiungere $\mathtt{u}$ confrontando un *solo* carattere per nodo, in quanto il confronto con il resto della sottostringa sicuramente ha successo. In tal modo, il costo della ricerca scende da $\mathtt{O}(\mathtt{m})$ a $\mathtt{O}(\mathtt{h})$. La situazione può essere trattata analogamente anche se partiamo da un nodo interno invece che dalla radice. Una tale modalità di ricerca sarà fondamentale in uno dei capitoli successivi.

Osserviamo che, essendo il trie compatto una rappresentazione succinta del trie, eredita i vari tipi di ricerca (per intervalli, di un insieme, di espressioni regolari, ecc.) descritti nella Sezione 7.1.

La modifica del trie compatto ricalca quella del trie, tenendo conto però che gli archi sono etichettati con le sottostringhe.

Nella fattispecie, l'inserzione di una stringa P consiste ancora una volta nelle ricerca del suo più lungo prefisso x che occorre nel trie compatto, dove $\mathsf{P} = \mathsf{xy}$. Sia $\mathfrak{u}$ il nodo raggiunto. Se il prefisso associato a $\mathfrak{u}$ coincide con x, allora creiamo una nuova foglia etichettata con P. Colleghiamo il nodo $\mathfrak{u}$ a tale foglia con un arco etichettato con y. Tuttavia, se x non coincide, dobbiamo spezzare in due l'arco entrante in $\mathfrak{u}$, inserendo un nuovo nodo $\mathfrak{u}'$ come nuovo padre di $\mathfrak{u}$ e associando il prefisso x a $\mathfrak{u}'$. Procediamo quindi con $\mathfrak{u}'$ alla stessa stregua indicata sopra per $\mathfrak{u}$, ovvero creando una foglia e collegandola a $\mathfrak{u}'$ con etichetta y. Il costo dell'inserzione è $\mathsf{O}(\mathfrak{m})$ se $\sigma = \mathsf{O}(1)$; altrimenti è $\mathsf{O}(\mathfrak{m} \log \sigma)$.

La cancellazione di P richiede la rimozione della foglia etichettata con P, nonché dell'arco che collega la foglia a suo padre $\mathfrak{u}$. Se $\mathfrak{u}$ diventa unario, allora bisogna rimuovere $\mathfrak{u}$, il suo arco entrante e il suo arco uscente, sostituendoli con un unico arco la cui etichetta è la concatenazione della sottostringa nell'arco entrante con quella nell'arco uscente. Tuttavia bisogna stare attenti a non usare etichette nel trie compattato che fanno riferimento a stringhe cancellate. Il costo della cancellazione è $\mathsf{O}(\mathfrak{m})$ se $\sigma = \mathsf{O}(1)$; altrimenti è $\mathsf{O}(\mathfrak{m} \log \sigma)$.