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 2002-2003

Versione 1.3

Introduzione

Le stringhe, le sequenze e i testi elettronici rappresentano il formato più diffuso di informazione digitale in Internet insieme ai dati multimediali (immagini e suoni). Siamo letteralmente sommersi da questi dati testuali; a partire dal livello basso del calcolatore, tutto è codificato mediante una sequenza di 0 e 1. La crescita esplosiva dei dati sta cambiando il paradigma di ricerca e di indicizzazione dei testi. Prima, l'ambizione era di organizzare i dati in modo da convogliare *tutte* le possibili informazioni utili verso l'utente. Adesso, si pone il problema opposto: limitare, filtrare e classificare tali informazioni per fornirne solo una *parte* all'utente. La limitazione di banda al flusso di informazioni è ormai dovute all'utente stesso. Non sorprende affatto sapere che non basta una vita per leggere tutte le informazioni di proprio interesse in Internet e nel World Wide Web.

Nel nostro piccolo, immaginiamo di scrivere un testo con il nostro "editore" di testi preferito. A un certo punto, vogliamo cercare un paragrafo che contiene una frase o una parola chiave. Scopriremo che il nostro editore di testi supporta delle operazioni di ricerca più o meno raffinate che ci consentono di individuare velocemente il paragrafo. Queste operazioni sono spesso disponibili in librerie standard di software; per esempio, la funzione `strstr()` per cercare in una stringa è contenuta nella libreria standard `<string.h>` dei compilatori per il linguaggio di programmazione C. In mancanza di tali librerie, non è comunque impegnativo progettare un algoritmo di forza bruta per cercare una sequenza di caratteri in un testo.

Il lettore può giustamente chiedersi perché i testi in Internet non possono essere trattati alla stessa stregua. Il problema nasce dalla grande quantità di informazioni testuali distribuite geograficamente e disponibili elettronicamente. Esistono delle statistiche che affermano che questa quantità raddoppia regolarmente dopo alcuni mesi. Destinare centinaia di calcolatori alla elaborazione di milioni di ricerche su enormi collezioni di testi coinvolge ordini di grandezza decisamente superiori a quelli della ricerca di una parola con il nostro 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 non ha un

simile impatto nel nostro editore di testi. Le tecniche da impiegare per gestire le 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 a 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. In generale la qualità dei dati è bassa in quanto non è possibile controllare la qualità tipografica dei documenti resi pubblici in Internet e nel Web. I testi datati sono spesso riportati in versione elettronica usando termini ormai decaduti. Possono esistere più versioni modificate dello stesso testo.

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. Specializzare le ricerche ammettendo metacaratteri, espressioni regolari e un certo grado di imprecisione, così da renderle più selettive e mirate (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).

3. Presentare i testi individuati da una ricerca secondo un qualche ordine di rilevanza che sia efficacemente realizzabile tramite il calcolatore (cfr. Parte III: Indici testuali e motori di ricerca).

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 le loro connessioni con le attuali tecnologie. I campi di applicazione sono vari: la biologia computazionale, la compressione dei dati, l'estrazione di contenuti (data mining), la linguistica computazionale, il recupero veloce di testi, ecc. Per esempio, l'informazione sottostante la biochimica, la biologia cellulare e lo sviluppo degli organismi può essere rappresentata da una semplice sequenza di lettere {A, C, G, T} chiamata sequenza di DNA. È la struttura di dati alla base della biologia degli organismi, per cui la biologia computazionale cerca di ridurre fenomeni biologici complessi a interazioni tra sequenze di DNA.

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 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.

Le dispense descrivono un più ampio spettro di tecniche rispetto a quelle correntemente in uso in certi settori, scegliendo tra quelle che sembrano avere il più alto potenziale di applicazione (a giudizio personale dell'autore). Senza dubbio, parte del materiale descritto potrebbe non trovare mai diretta applicazione, per esempio, nei motori di ricerca. Tuttavia, una buona parte delle tecniche presentate dovrebbe risultare di immediata utilità per nuovi approcci nella gestione dei testi elettronici attraverso il loro confronto e recupero veloce, la compressione dei dati, l'inferenza logica e l'estrazione di contenuti dai testi.

La presentazione enfatizza le idee, gli algoritmi, la loro evoluzione e le loro conseguenti applicazioni, contenendo al minimo i tecnicismi tipici di questa area. Vengono descritti gli algoritmi e le strutture di dati 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 il punto di partenza per strumenti più sofisticati per la ricerca di contenuti. Un vantaggio della ricerca sintattica è la buona selettività quando la formulazione di una interrogazione è impostata correttamente. Questo dovrebbe dare l'impulso a conoscere meglio i principi algoritmici della ricerca sintattica per poterne sfruttare le potenzialità. Inoltre, vengono presentate alcune tecniche di ricerca per contenuti, in particolare quelle che permettono di filtrare e di indicizzare i testi per poi classificare i risultati delle ricerche. Lasciamo alle giovani menti il compito di aprire nuovi orizzonti verso metodi efficienti e selettivi per la ricerca semantica.

Chi l'ha vista?

Immaginiamo di voler vedere se `ananas` appare nel nome esotico `banananassata` di una nuova bibita. Nel gergo tecnico degli informatici, `ananas` e `banananassata` 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 $|\texttt{banananassata}| = 13$ caratteri. Una identica notazione vale per $|\Sigma|$ così da indicare il numero di simboli presenti nell'alfabeto delle stringhe. 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	$\lvert\Sigma\rvert$	uso
$\{0,1\}$	2	sequenze binarie
$\{A,\ldots,Z\}$	26	parole maiuscole
ASCII	256	sequenze alfanumeriche
Unicode	≥ 65536	sequenze alfanumeriche

Assumiamo 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). Senza perdita di generalità, assumiamo che i simboli dell'alfabeto Σ vengano codificati all'interno del calcolatore come numeri da 1 a $|\Sigma|$. 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 $1 \leq i \leq |P|$, e con $T[i]$ il carattere i-esimo del testo T, per $1 \leq i \leq |T|$. Continuando il nostro esempio, $P[3] = \texttt{a}$ e $T[3] = \texttt{n}$. Ma non vogliamo fermarci ai singoli caratteri. Per indicare porzioni contigue delle stringhe, chiamate *sottostringhe*, introduciamo la notazione $T[i..j]$ per denotare la sequenza di caratteri $T[i]T[i+1]\cdots T[j]$ dove $i \leq j$. Per esempio, $T[7..9] = \texttt{nas}$. Usiamo la coppia di interi (i, j) per rappresentare la sottostringa $T[i..j]$ con un numero costante di celle di memoria, indipendentemente dalla sua lunghezza $|T[i..j]| = j-i+1$. (La lunghezza di una sottostringa è chiaramente il numero di caratteri in essa contenuta.) Se $j < i$, 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 $1 \leq i \leq |T|$, la sottostringa $T[1..i]$ si chiama *prefisso* i-esimo; la sottostringa $T[i..|T|]$ si chiama *suffisso* i-esimo. Per esempio, il terzo prefisso di T è `ban`, mentre il suo terzo suffisso è `nananassata`.

Ritornando al problema di cercare il pattern $P = \texttt{ananas}$ nel testo $T = \texttt{banananassata}$, possiamo formularlo con la nostra terminologia. Vogliamo identificare le sottostringhe del testo che sono uguali al pattern. In altre parole, vogliamo individuare le posizioni j nel testo per cui $T[j..j+|P|-1] = P$. Nel nostro caso, poiché $T[4..9] = \texttt{ananas}$, abbiamo una occorrenza in posizione $j = 4$ 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 II 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 $|P|$ caselle che facciamo scorrere sul testo. Allineiamo la finestra su tutte le posizioni possibili del testo. Ad ogni istante, la finestra mostra esattamente $|P|$ caratteri contigui del testo, per esempio $T[j..j+|P|-1]$, su cui procedere con i confronti. Esistono $|T| - |P| + 1$ allineamenti 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 j. Se invece i confronti danno tutti esito positivo (tutti match), possiamo dichiarare di aver trovato una occorrenza in posizione 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 j quando la richiesta è enumerativa.

Mostreremo tecniche più sofisticate per rendere questo approccio efficiente e per ammettere pattern con errori e metacaratteri nella Parte I. Applicheremo le nozioni acquisite nelle Parti I e II alla indicizzazione nei motori di ricerca. Concludiamo questo capitolo dando i prerequisiti e le nozioni preliminari necessarie alla comprensione del materiale che discuteremo nei prossimi capitoli.

Prerequisiti e nozioni preliminari

Al fine di facilitare la propria comprensione, il lettore dovrebbe avere conoscenze di base nella programmazione dei calcolatori e nella 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 $a \leq b$ THEN $a \leftarrow b$ ELSE $b \leftarrow c$;
- ciclo while: WHILE $a \leq b$ AND $a > c$ DO $a \leftarrow a + 1$;
- ciclo for: FOR $i \leftarrow 1$ TO n DO $a \leftarrow a * i$;
- ciclo foreach: 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. Possiamo però forzare il ciclo foreach a simulare il ciclo for scrivendo FOREACH $i = 1, 2, \ldots, n$ DO $a \leftarrow a + 1$. Preferiamo mantenere comunque il ciclo for in quanto è di fatto uno dei costrutti standard.

Per quanto riguarda le strutture dati usiamo quelle elementari: vettori, array e tabelle (multidimensionali o meno); code e pile; liste con puntatori o riferimenti; alberi radicati e ordinati; grafi, cammini e 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 I 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 $\delta(q, c) =$ 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 *nondeterministico*.

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 (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. Il linguaggio riconosciuto dagli automi, ovvero l'insieme di tutte le stringhe accettate, coincide con le espressioni regolari che studieremo nel Capitolo 2.

Indice

Parte I

ALGORITMI DI RICERCA

Questa parte descrive gli algoritmi di ricerca testuali, in cui i testi vengono interamente esaminati a ogni richiesta. Le tecniche algoritmiche descritte sono comunque di utilità nelle parti successive per la costruzione di indici testuali che evitano la scansione completa del testo a ogni richiesta. Nel Capitolo 1, vengono descritti alcuni algoritmi di ricerca esatta di stringhe e sequenze, introducendo nozioni e strumenti che saranno utili negli altri capitoli. Nel Capitolo 2, vengono descritte le espressioni regolari e la loro potenza descrittiva nelle ricerche testuali. Il Capitolo 3 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 4, 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 1

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, $1 \leq j \leq n-m+1$, tale che la sottostringa $T[j..j+m-1]$ 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 di forza bruta per il problema della ricerca esatta, mostriamo come ottenere un algoritmo veloce. Prima eseguiamo una elaborazione preliminare (preprocessing) sul pattern usando tabelle opportune e automi deterministici; poi, sfruttiamo l'elaborazione preliminare per rendere più veloce la scansione del testo (scanning) alla ricerca di occorrenze del pattern. Durante la scansione, manteniamo una finestra di m caselle che facciamo scorrere sul testo. Allineiamo l'inizio della finestra alle posizioni $1 \leq j \leq n-m+1$ del testo. A ogni istante, la finestra mostra gli m caratteri nella sottostringa $T[j..j+m-1]$ 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 una 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; stampiamo la posizione j quando la ricerca è di tipo enumerativo. Alla fine del capitolo, mostriamo come estendere uno degli algoritmi efficienti al caso della ricerca simultanea di più pattern.

1.1 Algoritmo di forza bruta

Nel caso dell'algoritmo di forza bruta, denominato Ricerca⋄ForzaBruta, esaminiamo tutte le posizioni j della finestra scorrevole, e 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 $P =$ `ananas` e $T =$ `bananananassata`, dove $m = 6$ e $n = 13$. Iniziando con $j = 1$, confrontiamo solo il primo carattere `b` della finestra che dà luogo a un mismatch. Proseguiamo con $j = 2$ confrontando tutti i caratteri `ananan` della finestra corrente e scoprendo solo alla fine un mismatch. Per $j = 3$, abbiamo subito un mismatch con il primo carattere `n` della finestra corrente. Infine, con $j = 4$, abbiamo tutti match, per cui abbiamo trovato una occorrenza del pattern P.

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

Ricerca⋄ForzaBruta(P, 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 − m + 1..j] = P}
8: RETURN FALSE;
```

Per restituire tutte le occorrenze, o contarle, basta sostituire la linea 7 con la stampa del valore di 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 $O(mn)$ passi al caso pessimo. Una tale situazione si configura scegliendo, in maniera artificiale, il testo $T = \texttt{a} \cdots \texttt{aaa}$ come una sequenza ripetuta della stessa lettera e il pattern $P = \texttt{a} \cdots \texttt{aab}$ come una sequenza ripetuta della stessa lettera tranne che nell'ultima posizione, dove troviamo una lettera diversa. In tal caso, il corpo del ciclo for (linee 3–7) viene eseguito tutte le $n - m + 1$ volte e, ogni volta, il ciclo while interno (linee 4–5) richiede $O(m)$ tempo perché si accorge del mismatch $\texttt{a} \neq \texttt{b}$ solo quando $k = m - 1$. 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 P in modo che l'ultimo carattere da confrontare nell'ordine stabilito sia comunque `b`. Ne deriva che il numero totale di confronti di caratteri è $O(mn)$. Nelle sequenze di DNA, i valori tipici di m sono dell'ordine delle centinaia di caratteri, per cui il costo di RICERCA◇FORZABRUTA risulta eccessivo.

In pratica, va osservato che RICERCA◇FORZABRUTA 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 al più 3–4 caratteri. Il comportamento al caso pessimo di $O(mn)$ confronti è raro ed emerge con stringhe veramente particolari come quelle appena mostrate. Poiché il costo dell'algoritmo è dominato dal numero dei confronti nel ciclo while interno (linee 4–5), mostriamo che RICERCA◇FORZABRUTA esegue un totale di circa

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

confronti nel caso medio, dove $\sigma = |\Sigma| \geq 2$ è il numero di simboli nell'alfabeto delle stringhe. 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 e 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 $c \in \Sigma$. 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. 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. 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 parallelamente, cioè considerando il primo carattere del pattern e di ogni stringa nell'elenco, poi il secondo carattere del pattern e 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 iniziano con $P[1]$). Nel nostro esempio, se $P[1] = \texttt{C}$, 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 confronti. Aggiungiamo un parziale di σ^{m-1} confronti al totale. In maniera analoga a prima, possiamo concludere che solo una 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 $1 \leq i \leq m$, esaminiamo σ^{m-i+1} stringhe dando luogo a un parziale di σ^{m-i+1} confronti. Dalla somma dei parziali otteniamo il numero totale di confronti:

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

Si noti 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 uniforme. Certi simboli, come le vocali, occorrono più frequentemente di altri. In effetti, esistono delle statistiche su certe classi di testi (per esempio, la narrativa in lingua inglese di un certo periodo storico) in cui a ogni simbolo $c \in \Sigma$ viene associata la sua frequenza, o meglio la sua 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.

1.2 Scansione veloce del testo tramite l'automa

La sorgente di inefficienza nella ricerca del pattern P nel testo T nasce dalla eventualità che molti caratteri del testo vengano confrontati $O(m)$ volte ciascuno con alcuni dei caratteri del testo. Quando cerchiamo $P = \texttt{ananas}$ in $T = \texttt{bananaanassata}$, per esempio, confrontiamo tutti i caratteri del pattern con quelli del testo che si trovano a partire dalla posizione $j = 2$ (cioè `bananaanassata`). Alcuni di questi caratteri (`ana` in `bananaanassata`), pur essendo dei match, vengono nuovamente confrontati con i caratteri del pattern quando consideriamo la posizione $j = 4$ contenente una occorrenza. Vorremmo invece confrontare solo i rimanenti caratteri (cioè `nas` in `bananaanassata`) di cui non conosciamo l'esito come match o mismatch. Questo comporta lo spostamento della finestra scorrevole saltando le posizioni che non sono favorevoli ai match, senza però tralasciare alcuna occorrenza.

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, una elaborazione preliminare

	$\delta[i,c]$	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*					

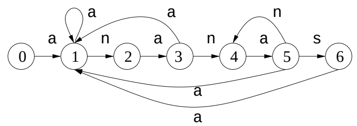


Figura 1.1: Tabella δ per il pattern $P =$ `ananas` e sua rappresentazione grafica. I simboli `-` denotano i caratteri del pattern che non sono stati ancora confrontati con successo all'interno della finestra scorrevole. Le caselle vuote contengono il valore 0, omesso dalla figura. Sono riportate le 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.

(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.

La tabella δ consiste di $m + 1$ righe e $|\Sigma|$ 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 $P[i + 1]$. Le colonne corrispondono ai simboli di Σ e prefigurano quale può essere il carattere c (senza accedere al testo!). Per esempio, la tabella δ per il pattern $P =$ `ananas` è riportata in Figura 1.1. L'elemento $\delta[i, c]$ della tabella, dove $0 \leq i \leq m$ e $c \in \Sigma$, rappresenta il fatto che abbiamo confrontato con successo i primi i caratteri del pattern con i primi 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 una occorrenza e quindi siamo nello stato finale. In generale, abbiamo due possibilità:

- $c = P[i + 1]$ per $i < m$ (numeri in **neretto** nella tabella in Figura 1.1). 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 $i = m$ (numeri in *corsivo* nella tabella in Figura 1.1). Sia $j \leq i$ il massimo intero per cui $P[1..j]$ è suffisso della stringa $P[1..i]\,c$, ottenuta concatenando $P[1..i]$ e c. La stringa $P[1..j]$ 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 $\delta[i, c] = j$ in questo caso.

La riga $i = 5$ della tabella in Figura 1.1 esemplifica le due possibilità. Tale riga corrisponde ad aver confrontato con successo i primi 5 caratteri `anana`. Consideriamo quindi la colonna

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

Esempio. L'importanza di avere una fase di elaborazione preliminare si rivela pienamente durante la fase di scansione del testo. Infatti, ciascun carattere del testo viene confrontato una *sola* volta. Prendiamo il testo $T =$ `bananassata`. Partendo dallo stato $i = 0$ e leggendo i caratteri del testo da sinistra a destra otteniamo la sequenza di transizioni: $0 \xrightarrow{\texttt{b}} 0 \xrightarrow{\texttt{a}} 1 \xrightarrow{\texttt{n}} 2 \xrightarrow{\texttt{a}} 3 \xrightarrow{\texttt{n}} 4 \xrightarrow{\texttt{a}} 5 \xrightarrow{\texttt{n}} 4 \xrightarrow{\texttt{a}} 5 \xrightarrow{\texttt{s}} 6 \xrightarrow{\texttt{s}} 0 \xrightarrow{\texttt{a}} 1 \xrightarrow{\texttt{t}} 0 \xrightarrow{\texttt{a}} 1$. Abbiamo una occorrenza quando lo stato raggiunto è $i = 6$. Dopo aver letto `banana` nel testo, siamo nello stato 5. Il carattere successivo del testo è `n` $\neq$ `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 da capo avendo confrontato con successo `anan`.

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

```
RICERCA◇AUTOMA(P, T):
m ← |P|; n ← |T|;
costruisci l'automa mediante la tabella δ;
i ← 0;
FOR j ← 1 TO n DO
   i ← δ[i, T[j]]
   IF i = m THEN
      RETURN TRUE   {Occorrenza T[j − m + 1..j] = P}
RETURN FALSE;
```

Complessità. Assumendo di aver già costruito la tabella δ il cui accesso all'elemento $\delta[i, c]$ richiede tempo costante, l'algoritmo RICERCA◇AUTOMA richiede $O(n)$ 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 gli ultimi m caratteri della finestra di scorrimento in memoria.

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 i-esimo del pattern, per $i = 1, 2, \ldots, m$, in modo da ottenere l'automa che trova le occorrenze di $P[1..i]$. Quando $i = m$, la tabella risultante è δ.

Esempio. Per renderci conto di come realizzare il passo i nella costruzione della tabella in Figura 1.1 per il pattern P = `ananas`, possiamo fissare il valore di $i = 5$ e vedere come ottenere la tabella che riconosce le occorrenze del pattern $P_5 = P[1..5]$ = `anana` partendo da quella per $P_4 = P[1..4]$ = `anan`:

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

Tabella per P_4 = `anan`.

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

Tabella per P_5 = `anana`

Osserviamo che lo stato finale per P_4 corrisponde alla riga $i-1$ e che per ottenere la tabella per P_5 dobbiamo inserire la riga $i = 5$. Partendo dallo stato $i - 1$, eseguiamo una transizione con il carattere i-esimo $P[5]$ = `a`, ottenendo lo stato $s = \delta[i-1, \mathtt{a}] = \mathit{3}$ nella tabella a sinistra. A questo punto, siccome vogliamo riconoscere P_5, dobbiamo porre $\delta[i-1, \mathtt{a}] = \mathbf{5}$ nella tabella a destra per definire la transizione al nuovo stato finale i. Poiché lo stato i, dopo aver letto `a`, si comporta come lo stato s (che non è finale!), è sufficiente copiare gli elementi dalla riga s alla riga i, ottenendo la tabella per P_5 a destra. Come ulteriore esempio, il lettore può eseguire il passo successivo $i = 6$, per ottenere la tabella in Figura 1.1 a partire dalla tabella di P_5 a destra.

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

```
Costruzione◇Automa(P):
m ← |P|;
FOREACH c ∈ Σ DO     {Stato iniziale 0}
  δ[0, c] ← 0;
FOR i ← 1 TO m DO     {Crea lo stato i}
  s ← δ[i − 1, P[i]];
  δ[i − 1, P[i]] ← i;
  FOREACH c ∈ Σ DO     {Copia la riga s nella riga i}
    δ[i, c] ← δ[s, c];
RETURN tabella δ;
```

Complessità. L'algoritmo Costruzione◇Automa esegue $m + 1$ passi, uno per riga, e riempie ciascuna riga in tempo $O(|\Sigma|)$. Il costo totale è quindi $O(m|\Sigma|)$ tempo e $O(m|\Sigma|)$ spazio. È possibile migliore 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 $|\Sigma_P| \leq m$ (nel nostro esempio, $\Sigma_P = \{\mathtt{a}, \mathtt{n}, \mathtt{s}\}$). Usando un vettore di appoggio C di lunghezza $|\Sigma|$, possiamo ridurre la tabella δ a sole $|\Sigma_P|+1$ 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 non appartiene a Σ_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 ottenuta eseguendo Costruzione◇Automa come elaborazione preliminare del pattern e Ricerca◇Automa come scansione del testo, richiede $O(m|\Sigma_P| + n)$ tempo e $O(m|\Sigma_P| + |\Sigma|)$ spazio aggiuntivo.

1.3 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◇ForzaBruta. Il problema è nel numero di colonne che può aumentare al crescere dell'alfabeto. Vediamo una maniera di 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 versone semplificata di quello originariamente proposta da Knuth, Morris e Pratt. Ha una estensione interessante a più stringhe che studieremo nella Sezione 1.6.

Il fulcro dell'algoritmo è dato da un automa di $m + 1$ righe che ha solo due colonne rapresentate 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 rappresenta una funzione da usare in corrispondenza di un mismatch e lo chiameremo φ. L'elemento $\varphi[i]$ memorizza la lunghezza del bordo di $P[1..i]$. Seppure sembra una lieve differenza rispetto alla definizione adottata nella Sezione 1.2, 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 1.1 per il pattern P = `ananas`. Mostriamo prima il vettore δ che raccoglie in modo compatto i numeri in neretto della tabella in Figura 1.1:

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 $i \geq 0$ 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 = `ananas` ha tre elementi maggiori di 0, in quanto solo tre prefissi di P hanno un bordo che non è vuoto. Il prefisso $P[1..3]$ = `ana` ha bordo `a` di lunghezza 1, il prefisso $P[1..4]$ = `anan` ha bordo `an` di lunghezza 2, e il prefisso $P[1..5]$ = `anana` ha bordo `ana` di lunghezza 3. Gli altri prefissi hanno bordo vuoto di lunghezza 0. Si usa inoltre il valore al contorno $\varphi[0] = -1$. I valori 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 $P =$ `ananas` nel testo $T =$ `banananassata`. 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. 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 $P[1..i] =$ `anana` del pattern in corrispondenza delle posizioni 2, 3, ..., 6, del testo (cioè in `banananassata`). Durante questi match abbiamo implicitamente usato il vettore δ. Il carattere successivo $T[j] =$ `n` del testo è diverso da quello $P[i+1] =$ `s` nel pattern. Essendo un mismatch, usiamo il vettore φ che sfrutta le proprietà dei bordi. In particolare, indica che il bordo di $P[1..i]$ è dato dai suoi primi $\varphi[i] = 3$ caratteri `ana`, per cui possiamo tranquillamente spostare la finestra scorrevole ponendo $i = \varphi[i] = 3$ senza perdere alcuna occorrenza. Adesso $P[1..i] =$ `ana` è allineato a sinistra di $T[j] =$ `n` (cioè `banananassata`) per cui possiamo ripetere il confronto con $P[i+1] =$ `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. Lo stesso carattere del testo può essere confrontato più di una volta con alcuni caratteri del pattern. Nel nostro esempio, il secondo confronto di $T[j]$ ha esito positivo per cui possiamo avanzare di stato fino a trovare l'occorrenza del pattern nel testo (cioè in `banananassata`).

Pseudocodice. Assumento di aver calcolato il vettore φ, manteniano la seguente invariante sulla finestra del testo prima di confrontare il carattere del testo $T[j]$ con un opportuno carattere del pattern. Per $i \geq 0$, i caratteri di $P[1..i]$ appaiono nella parte sinistra della finestra e il carattere $P[i+1]$ va confrontato con il carattere corrente del testo $T[j]$. L'invariante è chiaramente valida all'inizio della scansione e in generale quando $i = 0$ poiché $P[1..i]$ è la strigna vuota ϵ. Nel passo generico, confrontiamo $P[i+1]$ con $T[j]$, dove $0 \leq i \leq m$ e $1 \leq j \leq n$. 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 i e j, in quanto manteniamo l'invariante per il valore di i risultante dall'iterazione.

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 ⋄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;
```


Complessità. L'algoritmo RicercaSemplice ⋄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 alla linea 5, la variabile i viene decrementata nella linea 6 e $T[j]$ viene confrontato di nuovo. Fortunatamente, 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 (per 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 $2n$.

Non è difficile modificare il codice per eseguire ricerche di tipo quantitativo ed enumerativo. Basta sostituire la llinea 9 con l'incremento di una variabile contatore oppure con la stampa della posizione $j - m + 1$.

Rimane da discutere come costruire il vettore φ. Il valore per $j = 0$ è uguale a -1. 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. Vediamo come mettere in relazione questo con il bordo di $P[1..j] = P[1..j-1]P[j]$.

Esempio. Prendiamo il pattern $P =$ `ananas` e fissamo $j = 5$ supponendo di avere calcolato il bordo `an` di $P[1..j-1] =$ `anan` per cui $\varphi[j-1] = i = 2$. Nel considerare il bordo di $P[1..j] =$ `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 $\varphi[j] = i+1$. In questo caso siamo fortunati, perché il bordo è `ana` in quanto $P[3] = P[5] =$ `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 $P[4] =$ `n` $\neq$ `s` $= P[5]$, per cui il bordo di $P[1..6] =$ `ananas` non può essere $i+1$ ma un più *piccolo* prefisso e suffisso che termina con `s`. La proprietà interessante è che se rimuoviamo la `s` otteniamo un bordo del bordo! In altre parole trasformiamo il problema di cercare il bordo in $P[1..j] = P[1..j-1]P[j]$ nel problema di trovare il bordo in $P[1..i]P[j]$. Iterando con $i = \varphi[i]$ scaliamo ogni volta il bordo fino a che non troviamo il bordo vuoto, che è il caso per $j = 6$.

Pseudocodice. L'algoritmo per calcolare il vettore φ si basa sui bordi. Poniamo $\varphi[0] = -1$ e, per $j = 1, 2, \ldots, m$, calcoliamo $\varphi[j]$ usando il valore $i = \varphi[j-1]$ calcolato al passo precedente. Estendere il bordo precedente significa verificare che $P[i+1] = T[j]$. 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 poniamo $\varphi[j] = i+1$.

CostruzioneSemplice ⋄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 ⋄KnuthMorrisPratt e RicercaSemplice ⋄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 in uesto 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 scorrevvole, 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 $O(m+n)$ indipendente 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⋄Automa.

L'analogia tra lo pseudocodice di CostruzioneSemplice ⋄KnuthMorrisPratt e di RicercaSemplice ⋄KnuthMorrisPratt suggerisce che, volendo, si può utilizzare solo l'algoritmo di CostruzioneSemplice ⋄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 ⋄KnuthMorrisPratt(S) per trovare i bordi di S. Lasciamo come esercizio al lettore convincersi 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 $\varphi[\,] \leq m$ nelle linee 4, 6 e 7 di CostruzioneSemplice ⋄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`. Produrrà sicuramente un mismatch dovuto al fatto che anch'esso è seguito da `a`. Il codice risultante differisce in parte da quello di RicercaSemplice ⋄KnuthMorrisPratt e

CostruzioneSemplice ⋄KnuthMorrisPratt, anche se l'idea algoritmica rimane identica nella sostanza.

1.4 Praticamente veloce: algoritmo di Boyer, Moore e Horspool

Gli algoritmi presentati finora hanno tutti la caratteristica comune di esaminare, come minimo, *tutti* i caratteri del testo nel 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 nel caso pessimo. Nulla vieta però che si possano esaminare meno caratteri nel caso medio. Purtroppo gli algoritmi delle sezioni precedenti non hanno questa proprietà. Presentiamo perciò un algoritmo veloce, inizialmente proposto da Boyer e Moore in contemporanea a quello di Knuth, Morris e Pratt, che ha il vantaggio di richiedere $O(m + (n/m)\log m)$ tempo in media. In altre parole, per ogni porzione di m caratteri del testo 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, fornendo 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, partendo dalla fine della finestra. Un potenziale vantaggio di procedere in questa maniera è che probabilmente, in caso di mismatch, riusciamo a spostare la finestra più velocemente 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. Assumiamo di aver confrontato con successo zero o più caratteri della finestra, denotati da u, a partire da destra e siamo incappati in un mismatch tra il carattere a nel pattern e l'omologo carattere $b \neq a$ nella finestra. La Figura 1.2.a illustra una tale situazione generale 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 allineando la sottostringa cu del pattern con quella bu del testo provochiamo uno spostamento della finestra verso sinistra (cfr. Figura 1.2.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 1.2.c mostra una tale situazione. In ogni caso abbiamo uno scorrimento della finestra a sinistra.

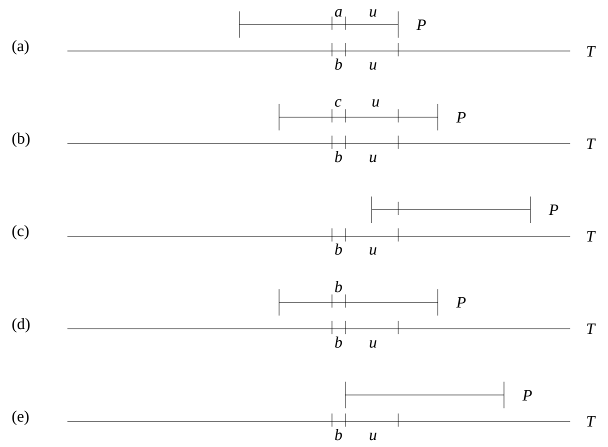


Figura 1.2: Situazione di mismatch nella finestra con l'algoritmo di Boyer e Moore. (a) Dopo aver confrontato con successo una sottostringa u (possibilmente vuota) a partire da destra, troviamo due caratteri $a \neq b$. (b)–(c) Spostamento della finestra con la prima tabella. (d)–(e) Spostamento della finestra con la seconda tabella.

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 1.2.d). Chiaramente, se b non appare nel pattern, spostiamo la finestra completamente a destra di b (cfr. Figura 1.2.e).

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 con 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 $O(m + (n/m) \log n)$, motivando così la sua velocità in pratica.

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

Esempio. Proviamo a calcolare il vettore λ per il pattern $P =$ `ananas`, escludendo l'ultimo carattere `s` dalla computazione. Vogliamo usare λ per far spostare la finestra, il cui ultimo carattere è c. Di conseguenza, se c non appare nelle prime $m-1$ posizioni del pattern dobbiamo porre $\lambda[c] = m = 6$ a indicare lo spostamento completo della finestra a destra di c. Se invece c appare in i come ultima posizione nel pattern, dobbiamo spostare la finestra di $m-i$ posizioni per allineare 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 il suo valore di λ uguale 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 il suo valore di λ uguale a $m - i = 2$. Otteniamo il seguente vettore, dove le colonne non specificate contengono il valore $m = 6$:

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

Ora 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 troviamo 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. Per esempio, i caratteri iniziali `ban` e quelli finali `sata` non sono stati confrontati.

Pseudocodice. L'algoritmo che presentiamo è molto veloce in pratica poiché mantiene le prestazioni pratiche dell'algoritmo originale di Boyer e Morre senza la complicazione delle due tabelle. La sua sorprendente semplicità è illustrata nel codice descritto di seguito che prende il nome anche del suo proponente 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 fienstra è di $m-i$ posizioni in avanti. I caratteri che non compaiono nel pattern danno luogo a spostamenti di m posizioni. Si noti che l'ultimo carattere $P[m]$ del pattern non va considerato per garantire uno spostamento di almeno una posizione.

Costruzione ⋄BoyerMooreHorspool(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: $\quad\lambda[P[i]] \leftarrow m - i$;
6: return tabella λ;

Il vettore λ viene 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 gli i-esimi caratteri da destra $P[m-i]$ e $T[j-i]$. Lo spostamento della finestra di almeno una posizione avviene alla linea 11.

Ricerca⋄BoyerMooreHorspool(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⋄BoyerMooreHorspool può effettuare uno spostamento maggiore in alcuni casi con una semplice modifica. Anziché usare l'ultima carattere $T[j]$ della finestra, possiamo usare il successivo $T[j+1]$ 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. Quindi, in Costruzione ⋄BoyerMooreHorspool, l'assegnamento alla linea 3 impiega il valore $m+1$, il ciclo for alla linea 4 arriva fina a m e l'assegnamento alla linea 3 impiega il valore $m-i+1$.

Complessità. La complessità al caso pessimo di Ricerca⋄BoyerMooreHorspool è $O(mn)$ 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 di forza bruta descritta nella Sezione 1.1, in cui un match ha probalilità $1/|\Sigma|$ e un mismatch ha probalilità $1-(1/|\Sigma|)$.

Mostriamo infatti che Ricerca⋄BoyerMooreHorspool ha un costo medio di $O(m + n/|\Sigma|)$, ovvero esamina mediamente uno ogni $|\Sigma|$ 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 di forza bruta in Sezione 1.1 e peggiore di quella media $O(m + (n/m)\log m)$ dell'algoritmo di Boyer e Moore, in quanto $|\Sigma|$ è fissato.

Il costo di Costruzione ⋄BoyerMooreHorspool è comunque $O(m)$, per cui concentriamo l'analisi sul resto della computazione in Ricerca⋄BoyerMooreHorspool per mostrare il suo costo medio $O(m/\sigma)$, dove $\sigma = |\Sigma|$. 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 1.1 il numero medio di confronti all'interno di una finestra:

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

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 Ricerca⋄BoyerMooreHorspooldobbiamo 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 di $O(n/|\Sigma|)$ passi.

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 accade con probabilitá $[1-(1/\sigma)]^{m-1}$.
- $\lambda[c] = m-i$, per $1 \leq i \leq m-1$. In altre parole, c è uguale al carattere $P[i]$, mentre è diverso dagli altri caratteri in $P[i+1, m]$. Questo accade con probabilità $(1/\sigma)[1-(1/\sigma)]^{m-i-1}$.

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

$$\bar{s} = m[1-(1/\sigma)]^{m-1} + \sum_{i=1}^{m-1}(m-i)(1/\sigma)[1-(1/\sigma)]^{m-i-1} \tag{1.1}$$

Per i virtuosi delle manipolazioni algebriche, mostriamo come semplificare l'espressione in (1.1). Usiamo la nota formula

$$\sum_{j=1}^{m-1} jc^j = \frac{c + (m-1)c^{m+1} - mc^m}{(c-1)^2}, \tag{1.2}$$

dimostrabile per induzione su m. Partendo dalla formula (1.1), mettiamo in evidenza $\frac{1/\sigma}{1-(1/\sigma)}$ nella sommatoria e sostituiamo $m-i$ con j, per cui la formula diventa

$$m[1-(1/\sigma)]^{m-1} + \frac{1/\sigma}{1-(1/\sigma)}\sum_{j=1}^{m-1} j[1-(1/\sigma)]^j. \tag{1.3}$$

Applicando la formula nota (1.2) con $c = 1-(1/\sigma)$, otteniamo

$$m[1-(1/\sigma)]^{m-1} + \frac{1/\sigma}{1-(1/\sigma)}\frac{[1-(1/\sigma)] + (m-1)[1-(1/\sigma)]^{m+1} - m[1-(1/\sigma)]^m}{1/\sigma^2}. \tag{1.4}$$

Semplificando le due frazioni otteniamo

$$m[1-(1/\sigma)]^{m-1} + \sigma\{1 + (m-1)[1-(1/\sigma)]^m - m[1-(1/\sigma)]^{m-1}\}. \tag{1.5}$$

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

$$\sigma\{1-[1-(1/\sigma)]^m(-(m/\sigma)/[1-(1/\sigma)]-(m-1)+m/[1-(1/\sigma)])\}. \tag{1.6}$$

A questo punto, non è difficile semplificare $-(m/\sigma)/[1-(1/\sigma)]-(m-1)+m/[1-(1/\sigma)]=1$, ottenendo che la (1.1) si riduce a

$$\bar{s}=\sigma\left[1-\left(1-\frac{1}{\sigma}\right)^m\right]. \tag{1.7}$$

1.5 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 dell'alfabeto e dalla lunghezza del pattern.

Da osservazioni sperimentali risulta che Ricerca⋄ForzaBruta 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⋄BoyerMooreHorspool 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⋄ForzaBruta 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 al più 3-4 caratteri. In tal caso, il costo addizionale di Costruzione ⋄Automa, CostruzioneSemplice ⋄KnuthMorrisPratt e Costruzione ⋄BoyerMooreHorspool 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.

1.6 Ricerca simultanea di più stringhe: algoritmo di Aho e Corasick

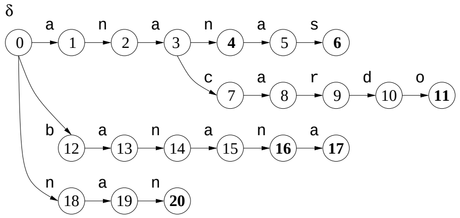
Diverse applicazioni richiedono la ricerca simultanea di più pattern $P_1, \ldots, P_k$ durante la scansione del testo T, per esempio, per espandere gli acronimi o per verificare la correttezza

delle citazioni bibliografiche. Un altro esempio è l'individuazione di file infettati da virus informatici. Molti virus lasciano una "impronta digitale" della loro infezione riconoscibile come occorrenza di una certa sequenza di byte nel codice eseguibile delle applicazioni. Il riconoscimento di impronte equivale al problema dello string matching. 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, vorremmo verificare velocemente se almeno una delle migliaia di impronte digitali occorre nel file.

Usando la 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 $O(M+kn)$ più il numero di occorrenze riportate. Aho e Corasick hanno proposto un metodo, coevo a quello di Knuth, Morris e Pratt, che richiede solo $O(M+n)$ tempo più il numero di occorrenze riportate. In altre parole, la fase di scansione del testo è k volte più veloce, per cui è il metodo è preferibile rispetto al metodo che ripete k ricerche indipendenti. Si ha un guadagno notevole in termini di velocità per un grande numero k di pattern da cercare. Questo metodo di ricerca multipla è realizzato nel programma di utilità `fgrep` 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 volerli cercare indipendentemente, per cui costruiamo i vettori δ_i e φ_i dei rispettivi automi con l'algoritmo CostruzioneSemplice ⋄KnuthMorrisPratt, applicato singolarmente a ciascun pattern P_i, dove $1 \le i \le k$. Volendo realizzare la ricerca su tutti questi automi in una singola passata, possiamo applicare un qualche meccanismo di fusione dei vettori δ_i e φ_i, ottenendo due tabelle uniche δ e φ che permettono la ricerca simultanea di tutti i pattern. La fusione avviene in due parti. Nella prima parte, costruiamo uno *scheletro* dell'automa in cui fondiamo i vettori δ_i, in modo da condividere le porzioni iniziali dei vettori che corrispondono ai prefissi uguali dei pattern. Nella seconda parte, fondiamo i vettori φ_i usando la nozione di bordo, introdotta nella Sezione 1.2 e qui estesa a un insieme di stringhe. Informalmente, il bordo è un suffisso che è anche prefisso di almeno uno dei 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` (questo ultimo introdotto perché occorre all'interno di P_1 e P_3). Mostriamo prima la prima parte, in cui costruiamo lo scheletro usando una sola tabella δ per tutti i pattern da cercare. Con riferimento alla Figura 1.3, il pattern P_1 viene creato allocando la riga i (stato) della tabella δ per il carattere $P_1[i]$, analogamente a quanto visto finora. Nel trattare P_2, cerchiamo il suo più lungo prefisso, `ana`, in comune con i pattern fino al quel momento esaminati, ovvero P_1. Tale prefisso corrisponde ad avere attraversato gli stati 0, 1, 2, e 3, che quindi vengono condivisi con gli altri pattern. È sufficiente allocare le righe 7–11 per sistemare i rimanenti caratteri in `cardo`. I rimanenti due pattern P_3 e P_4 non condividono prefissi comuni (a parte quello vuoto), per cui allochiamo una nuova riga per ciascun carattere. Gli stati finali sono marcati in neretto; manteniamo un vettore *final* di valori Booleani, tale che *final*$[i]$ = TRUE se e solo se lo stato i è finale (nel nostro caso, abbiamo 4, 6, 11, 16, 17 e 20 come stati finali).



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

Figura 1.3: 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, analoga all'algoritmo RicercaSemplice ⋄KnuthMorrisPratt.

Pseudocodice. L'algoritmo di costruzione dello scheletro inizializza a NIL gli elementi della tabella δ e a FALSE gli elementi in *final* (di fatto, crea lo stato 0). Prende un pattern P_j alla volta (ciclo for alle linee 6–14) e cerca il suo più lungo prefisso in δ (linee 7–9), identificabile partendo dallo stato $i = 0$ e seguendo gli stati indicati da δ alla riga i con il carattere $P_j[p]$. Non appena un valore NIL viene trovato e il pattern P_j ha ancora caratteri non esaminati, crea gli stati necessari al resto del pattern (linee 10–13). Infine, marca lo stato finale ottenuto (linea 14). Si noti che l'algoritmo non marca ancora gli stati interni 4 e 16 come finali in quanto tale compito è posticipato alla costruzione dell'altra tabella φ.

Costruzione ⋄Scheletro ⋄AhoCorasick($P_1, \ldots, P_k$):

1: $last \leftarrow 0$;
2: FOREACH $i \geq 0$ DO
3:    $final[i] \leftarrow$ FALSE;
4: FOREACH $i \geq 0$ AND $c \in \Sigma$ DO
5:    $\delta[i, c] \leftarrow$ NIL;
6: FOR $j \leftarrow 1$ TO k DO
7:    $i \leftarrow 0$; $p \leftarrow 1$;
8:    WHILE $p \leq |P_j|$ AND $\delta[i, P_j[p]] \neq$ NIL DO
9:       $i \leftarrow \delta[i, P_j[p]]$;
10:       $p \leftarrow p + 1$;
11:    WHILE $p \leq |P_j|$ DO
12:       $last \leftarrow last + 1$;
13:       $\delta[i, P_j[p]] \leftarrow last$;
14:       $i \leftarrow last$; $p \leftarrow p + 1$;
15:    $final[i] \leftarrow$ TRUE;
16: RETURN tabelle δ, *final*;

Dal nostro esempio, possiamo già evincere alcune proprietà dello scheletro.

- 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 vertice v corrisponde al prefisso comune di uno o più pattern; concatenando i caratteri lungo il cammino dal vertice 0 al vertice v otteniamo il prefisso.
- Se un pattern non appare all'interno degli altri pattern, corrisponde a un unico cammino dalla radice a una foglia.
- Se un pattern occorre in altri pattern, allora corrisponde a uno o più nodi interni.

Esempio. Illustriamo la seconda parte, in cui calcoliamo una tabella φ analoga a quella calcolata con CostruzioneSemplice ⋄KnuthMorrisPratt, per cui fissiamo il valore $\varphi[0] = -1$. Per $i > 0$, sia $P_j[1..p]$ il prefisso associato al vertice i. Dobbiamo trovare il bordo di $P_j[1..p]$, definito come il suo più lungo suffisso che prefisso di almeno uno dei pattern. Il vertice che memorizza tale bordo, in quanto prefisso di uno o più pattern, è il valore da porre in $\varphi[i]$. Si noti che il bordo è ben definito perché per ogni prefisso esiste un solo nodo che lo rappresenta. Nel nostro caso, con $i = 7$, abbiamo un bordo vuoto per $P_j[1..p] =$ `anac` e poniamo $\varphi[7] = 0$. Con $i = 4$, il bordo di $P_j[1..p] =$ `anan` è `nan` a cui corrisponde il vertice 20. Poniamo quindi $\varphi[4] = 20$. Per trovare i valori di φ usiamo la stessa tecnica di CostruzioneSemplice ⋄KnuthMorrisPratt, con la differenza che se il prefisso $P_j[1..p]$ è memorizzato nel nodo i, allora $P_j[1..p-1]$ è memorizzato nel nodo r, con $\delta[r, P_j[p]] = i$. Non è detto che sia sempre $r = i - 1$. Per esempio, $r = 3$ con $i = 7$, e $r = 3$ con $i = 4$. Rimandiamo il lettore alla discussione di CostruzioneSemplice ⋄KnuthMorrisPratt per i dettagli tecnici, mostrando in questo esempio come calcolare $\varphi[4] = 20$ e $\varphi[7] = 0$. Per $i = 4$, abbiamo $r = 3$. Prendiamo il suo bordo `na` $\varphi[3] = 19$ e vediamo che si estende con il carattere $P_j[p] =$ `n` ottenendo `na` che ha sede nel vertice 20. Per $i = 7$, abbiamo $r = 3$. Tuttavia, il bordo `na` rappresentato da $\varphi[3] = 19$ non si estende con $P_j[p] =$ `c`. Passiamo perciò a considerare il bordo `a` rappresentato da $\varphi[19] = 1$, e anch'esso non si estende con $P_j[p] =$ `c`. Rimane da iterare con il bordo vuoto rappresentato da $\varphi[1] = 0$, e anch'esso non si estende. Possiamo perciò concludere che il bordo di `anac` è vuoto per cui $\varphi[7] = 0$.

Pseudocodice. L'algoritmo pone $\varphi[0] = -1$. Poi esamina i vertici per livelli, dove il vertice 0 ha livello 0. In generale, il vertice $i > 0$ ha livello l se il suo predecessore r ha livello $l - 1$. I valori di φ vengono quindi fissati per livelli, partendo dal livello 1. In tal modo i vertici dei livelli precedenti hanno il valore φ corretto e possono contribuire i valori per il livello corrente. A parte questa differenza, il calcolo di φ (linee 6–11) procede come in CostruzioneSemplice ⋄KnuthMorrisPratt. Le linee 12–14 si occupano invece dei pattern che occorrono in altri pattern. Come si può notare in Figura 1.3, questi possono dare luogo a più di uno stato finale. Fortunatamente, tali stati sono facilmente individuabili per un tale pattern. Partendo dallo stato v in cui il cammino dallo stato 0 a v corrisponde al pattern, tutti gli stati finali u corrispondenti hanno la proprietà che l'applicazione ripetuta di $e \geq 1$ volte di φ a partire da u restituisce v. In altri termini, denotando con $\varphi^e[u]$ lo stato raggiungibile a partire da r mediante e applicazioni di $u \leftarrow \varphi[u]$, abbiamo che $\varphi^e[u] = v$.

CostruzioneSemplice ⋄AhoCorasick($P_1, \ldots, P_k$):

1: $\varphi[0] \leftarrow -1$;
2: costruisci lo scheletro per ottenere le tabelle δ e *final*;
3: FOREACH livello $l > 0$ AND stato s a livello l AND carattere a tale che $\delta[s, a] \neq$ NIL DO

$r \leftarrow \delta[s, a]$;
$i \leftarrow \varphi[s]$;
WHILE $i \geq 0$ AND $\delta[i, a] =$ NIL DO
  $i \leftarrow \varphi[i]$;
IF $i \geq 0$ THEN
  $\varphi[r] \leftarrow \delta[i, a]$;
ELSE
  $\varphi[r] \leftarrow 0$;
FOREACH $i \geq 0$ DO
  IF $\mathit{final}[\varphi^e[i]] =$ TRUE AND $e \geq 1$ THEN
    $\mathit{final}[i] =$ TRUE
RETURN tabelle δ, φ, *final*;

Complessità. Il costo totale della costruzione dell'automa è $O(M)$ tempo assumendo che $|\Sigma|$ è costante, altrimenti c'è un fattore moltiplicativo logaritmico in $|\Sigma|$. Lo spazio richiesto è $O(M)$ perché manteniamo in realtà solo i valori $\delta[] \neq$ NIL per cui δ forma un albero che ha radice nello stato 0. La complessità della costruzione di φ è identica, realizzando CostruzioneSemplice $\diamond$ AhoCorasick mediante una opportuna visita per livelli dello scheletro.

Pseudocodice. A questo punto possiamo proporre l'algoritmo di ricerca, che è una estensione di RicercaSemplice $\diamond$ KnuthMorrisPratt.

RicercaSemplice $\diamond$ AhoCorasick($P_1, \ldots, P_k, T$):
1: $M \leftarrow |P_1| + \cdots + |P_k|$; $n \leftarrow |T|$;
2: costruisci le tabelle δ, φ e *final*;
3: $i \leftarrow 0$;
4: FOR $j \leftarrow 1$ TO n DO
5:   WHILE $i \geq 0$ AND $\delta[i, T[j]] =$ NIL DO
6:     $i \leftarrow \varphi[i]$;
7:   IF $i \geq 0$ THEN
8:     $i \leftarrow \delta[i, T[j]]$;
9:   ELSE
10:     $i \leftarrow 0$;
11:   IF $\mathit{final}[i]$ THEN
12:     RETURN TRUE; $\{$*Occorrenza* $T[j - m + 1..j] = P$ *per pattern* P *associato a riga* $i\}$
13: RETURN FALSE;

Per restituire tutte le occorrenze a partire da un certo stato i finale, basta sostituire la riga 12 con un ciclo che procede a elencare tutti gli stati finali raggiungibili da i attraverso φ:

$f \leftarrow i$;
WHILE $f \geq 0$ AND $\mathit{final}[f]$ DO $\{$*Occorrenza* $T[j - m + 1..j] = P$ *per pattern* P *associato a riga* $f\}$
  $f \leftarrow \varphi[f]$;

Complessità. Il costo della scansione è $O(n)$ più il numero di occorrenze riportate, a cui va aggiunto il costo $O(M)$ per la costruzione delle tabelle δ e φ. L'argomento è analogo a RicercaSemplice $\diamond$ KnuthMorrisPratt: abbiamo una finestra che scorre su una posizione diversa ad ogni iterazione di uno dei due cicli for e while. Segue dal fatto che ci sono al più n posizioni.

Capitolo 2

Ricerca con Espressioni Regolari e Caratteri Speciali

In questo capitolo trattiamo un tipo di ricerca molto flessibile basata sulle espressioni regolari. Spesso i testi hanno un formato con delle particolari strutture sintattiche come i formati HTML o XML ampiamente diffusi in Internet, oppure alcune porzioni di testo sono caratterizzate da certe regolarità che vogliamo sfruttare 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` (ebbene sì!, 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. L'iterazione 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, le espressioni regolari servono a descrivere in maniera concisa un *insieme* di pattern da cercare, per cui il termine pattern indica, in tale contesto, 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 come `grep` di Unix/Linux, diversi utenti non ne sfruttano appieno le possibilità in quanto la loro formulazione appare ostica a prima vista. Il presente capitolo è un invito al lettore ad approfondire meglio questo tipo di ricerche che, se opportunamente utilizzate, permettono di ottenere risultati più selettivi rispetto alla ricerca esatta descritta nel Capitolo 1. 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 ai loro simboli precedenti e successivi.

2.1 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. La definizione 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.

1. Le espressioni regolari semplici sono i simboli dell'alfabeto Σ. Ogni espressione regolare $c \in \Sigma$ rappresenta il simbolo c stesso; per esempio, l'espressione regolare `a` rappresenta il carattere `a`. Fanno eccezione i simboli speciali $|, (), *, \epsilon$ che vengono usati per comporre le espressioni regolari. In particolare, ϵ rappresenta la stringa vuota, che non contiene alcun carattere.

2. Due espressioni regolari r_1 e r_2 possono essere composte attraverso due regole:
   (a) $(r_1|r_2)$ è l'espressione regolare di alternativa a cui corrispondono le stringhe rappresentate da r_1 oppure da r_2. Per esempio, l'espressione (`a`|`b`) rappresenta sia la stringa `a` che la stringa `b`.
   (b) $(r_1)(r_2)$ è l'espressione regolare di concatenazione a cui corrispondono le stringhe xy ottenute concatenando una qualsiasi stringa x rappresentata da r_1 con una qualsiasi stringa y rappresentata da r_2. Per esempio, l'espressione (`a`)(`b`) rappresenta la stringa `ab`, e (`a`)(`b`|`c`) rappresenta le stringhe `ab` e `ac` (ovvero `a` seguita da una `b` oppure una `c`).

3. Una espressione regolare r può essere concatenata con se stessa zero, una o più volte mediante due regole:
   (a) $(r)^*$ è l'espressione regolare a cui corrispondono le stringhe della forma $x_1 x_2 \cdots x_s$ per ogni intero $s \geq 0$, dove l'operatore $*$ viene chiamato stella di Kleene. Se $s = 0$, allora la stringa risultante è ϵ. Altrimenti, ciascuna x_i è una qualsiasi stringa rappresentata da r, e $x_1 x_2 \cdots x_s$ rappresenta tutte le possibili concatenazioni di stringhe rappresentate da r. 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.
   (b) (r) è l'espressione regolare a cui corrispondono le stesse stringhe rappresentate da r.

Adottiamo la convenzione di omettere l'uso delle parentesi nelle espressioni regolari quando questa scelta non dia luogo ad ambiguità di interpretazione. Inoltre, usiamo ϵ per indicare sia il carattere vuoto che la stringa vuota (formalmente andrebbero usati due simboli diversi; nel nostro caso possiamo usare la stessa notazione senza generare confusione nel lettore).

Per comprendere la potenzialità delle espressioni regolari vediamo alcuni esempi. L'espressione (`0`|`1`)(`0`|`1`)* rappresenta tutte le sequenze binarie non vuote, ovvero `0` oppure `1` seguito da zero o più cifre binarie. I termini `aiuole` e `aiole` rappresentano due varianti della stessa parola che possono essere rappresentate con l'espressione `ai(uo|o)le`. 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.

Le espressioni regolari servono anche a specializzare la ricerca (esatta o meno) di una stringa pattern P. Basta specificare due espressioni regolari r_1 e r_2 che delimitano il contesto sinistro e il contesto destro delle occorrenze di P all'interno del testo, utilizzando il pattern esteso $r_1\,P\,r_2$ 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 $P = \mathtt{vendesi}$, specificando l'espressione regolare $<\mathtt{B}|\mathtt{b}>\Sigma^*\,\mathtt{vendesi}\,\Sigma^*<\mathtt{B}|\mathtt{b}>$. In tale espressione ammettiamo maiuscole `B` e minuscole `b` nei marcatori, e la parola `vendesi` si trova racchiusa tra due sequenze arbitrarie Σ^*, eventualmente vuote, dove Σ inequivocabilmente denota tutti i caratteri dell'alfabeto $(\mathtt{a}|\mathtt{b}|\cdots)$.

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, il connettivo OR è equivalente all'operatore $|$ di alternativa; il complemento di una espressione regolare è ancora una espressione regolare, per cui è semplice realizzare il NOT. Infine, per specificare due termini t_1 e t_2 in AND basta cercare $(t_1\,\Sigma^*\,t_2 \mid t_2\,\Sigma^*\,t_1)$.

Il successo delle espressioni regolari nelle applicazioni, a partire dal loro impiego nella progettazione dei compilatori, ha motivato una serie di modifiche ed estensioni per renderne più semplice e compatta la specifica. I simboli $|, (,), *$ sono aggiunti all'alfabeto Σ e possono essere specificati mediante le espressioni regolari $\backslash|, \backslash(, \backslash)$ e $\backslash*$, rispettivamente. Inoltre, $\backslash$ stesso viene specificato mediante $\backslash\backslash$. Le notazioni $\hat{}$ e \$ indicano rispettivamente l'*inizio* e la *fine* di una riga o di una parola. La coppia di parentesi quadre [ ] serve a indicare un *insieme* di caratteri (di conseguenza bisogna fare uso di $\backslash[$ e $\backslash]$ per specificare tali simboli come espressioni regolari). Per esempio $[\mathtt{abc}\backslash],]$ è equivalente a specificare $(\mathtt{a}|\mathtt{b}|\mathtt{c}|\,]\,|,)$ nella notazione originaria delle espressioni regolari. Si può inoltre definire il *complemento* di un insieme di caratteri mediante la notazione $[\hat{}]$. Per esempio, $[\hat{}\mathtt{abc}]$ rappresenta tutti i caratteri diversi da $\mathtt{a}, \mathtt{b}, \mathtt{c}$. Un *intervallo* di caratteri può essere specificato mediante il trattino; per esempio, $[\mathtt{A}-\mathtt{Z}]$ rappresenta tutte le lettere maiuscole e $[\hat{}\,\mathtt{A}-\mathtt{Z}\,\mathtt{a}-\mathtt{z}]$ rappresenta un qualunque carattere che non sia una lettera. Per indicare tutti i caratteri dell'alfabeto Σ, ovvero un simbolo qualsiasi, si può usare il punto ".". Esistono ulteriori notazioni, varianti della stella di Kleene, per specificare quante volte una espressione regolare deve essere ripetuta; per esempio, $(r)^+ \equiv (r)(r)^*$ indica che r deve essere ripetuta una o più volte. Lo standard IEEE Posix fornisce tutti i dettagli della notazione da adottare per le estensioni sintattiche delle espressioni regolari.

Come esempio finale 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: $[\hat{}\,\mathtt{a}-\mathtt{z}\,\mathtt{A}-\mathtt{Z}\,\mathtt{0}-\mathtt{9}]^*[\mathtt{a}-\mathtt{z}\,\mathtt{A}-\mathtt{Z}][\mathtt{a}-\mathtt{z}\,\mathtt{A}-\mathtt{Z}\,\mathtt{0}-\mathtt{9}]^*[\hat{}\,\mathtt{a}-\mathtt{z}\,\mathtt{A}-\mathtt{Z}\,\mathtt{0}-\mathtt{9}]^*$, ovvero una lettera in $[\mathtt{a}-\mathtt{z}\,\mathtt{A}-\mathtt{Z}]$ seguita da una sequenza alfanumerica in $[\mathtt{a}-\mathtt{z}\,\mathtt{A}-\mathtt{Z}\,\mathtt{0}-\mathtt{9}]^*$ possibilmente vuota, il tutto racchiuso tra due sequenze non-alfanumeriche

in $[\hat{}\,\mathtt{a}-\mathtt{z}\,\mathtt{A}-\mathtt{Z}\,\mathtt{0}-\mathtt{9}]^*$ possibilmente vuote se siamo all'inizio o alla fine del codice sorgente.

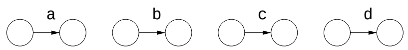
Osserviamo che le estensioni discusse finora semplificano la formulazione delle stringhe ma possono essere sempre ottenute con la notazione originaria delle espressioni regolari. In altre parole, non cambia il linguaggio rappresentato con le espressioni regolari. Non è il caso dei *riferimenti all'indietro*, in quanto il problema del loro riconoscimento diventa computazionalmente più arduo da trattare (precisamente, NP-completo).

2.2 Costruzione dell'automa finito nondeterministico

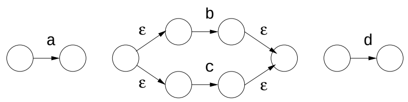
La ricerca delle espressioni regolari richiede la costruzione di un automa finito, deterministico o non. Per semplificare la discussione, mostriamo come costruire un automa nondeterministico. La costruzione segue fedelmente la definizione ricorsiva delle espressioni regolari. Il caso base è dato dalle espressioni semplici, ovvero i caratteri dell'alfabeto Σ come riportato nel punto 1 della Sezione 2.1. Il passo induttivo è dato dalla composizione delle espressioni regolari secondo quanto descritto nei punti 2 e 3 della Sezione 2.1.

Esempio. Supponiamo di voler costruire l'automa per l'espressione regolare $r = \mathtt{a}(\mathtt{b}|\mathtt{c})^*\mathtt{d}$. Vediamo come ottenere r per costruire quindi l'automa corrispondente. Prima riconosciamo `a`, `b`, `c` e `d` come espressioni regolari semplici. Poi componiamo `b` e `c` ottenendo $(\mathtt{b}|\mathtt{c})$. Su tale espressione applichiamo la stella di Kleene $(\mathtt{b}|\mathtt{c})^*$; infine, concateniamo il risultato con `a` e `d` per ottenere r.

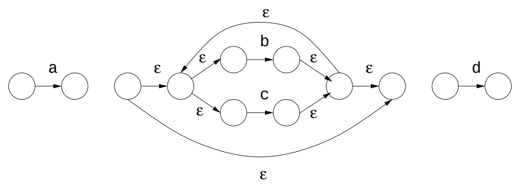
Procediamo alla stessa maniera per la costruzione dell'automa corrispondente a r. Come base di partenza, costruiamo gli automi per le espressioni semplici `a`, `b`, `c` e `d`:



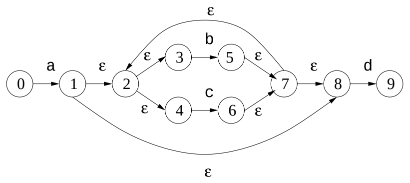
Ciascuno di questi automi ha un proprio stato iniziale e uno finale. Quando andiamo a comporre gli automi per `b` e `c` così da ottenere l'alternativa $(\mathtt{b}|\mathtt{c})$, dobbiamo porre i due automi in parallelo, creando un nuovo stato iniziale e un nuovo stato finale. Il nuovo stato iniziale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



Possiamo adesso applicare la stella di Kleene, collegando lo stato finale dell'automa per $(\mathtt{b}|\mathtt{c})$ 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. Il nuovo stato finale è collegato in uscita allo stato finale dell'automa:



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:



Ricordiamo che gli archi aggiunti durante la creazione degli automi sono tutti etichettati con ϵ e chiamati ϵ-*transizioni*. Fanno eccezione gli archi impiegati negli automi per le espressioni semplici `a`, `b`, `c` e `d`. 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, 4 e 8 in quanto raggiungibili dallo stato 7 mediante ϵ-transizioni.

Dal nostro esempio illustrativo possiamo già evincere alcune proprietà dell'automa risultante. Innanzi tutto, abbiamo un solo stato iniziale e un solo stato finale. Inoltre, ogni stato o ha un solo arco uscente etichettato con un carattere $c \in \Sigma$, oppure ha al più due ϵ-transizioni. Infine, il numero di stati è limitato superiormente da $2|r|$, dove $|r|$ è la lunghezza dell'espressione r calcolata come il numero di caratteri e di simboli $|$ e $*$ che appaiono in r. Le suddette proprietà possono essere dimostrate per induzione. Per esempio, la limitazione superiore di $2r$ al numero di stati deriva dal seguente ragionamento. Ciascun carattere c dà luogo a due stati. 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 $2r$.

Pseudocodice. La costruzione dell'automa finito nondeterministico segue la definizione ricorsiva dell'espressione regolare r:

Costruzione $\diamond$AutomaFinitoNonDeterministico(r):

1. Caso base $r \equiv c$ per un carattere $c \in \Sigma$. Restituisci l'automa composto da due stati con una freccia etichettata c dallo stato iniziale allo stato finale.

2. Caso induttivo in cui r è ottenuta dalla composizione di due espressioni regolari r_1 e r_2. Chiama ricorsivamente Costruzione $\diamond$AutomaFinitoNonDeterministico su r_1 e r_2 per ottenere i corrispondenti automi δ_1 e δ_2.
   (a) Caso $r \equiv (r_1|r_2)$. 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 $r \equiv (r_1)(r_2)$. Metti δ_1 e δ_2 in serie facendo coincidere lo stato finale di δ_1 con lo stato iniziale di δ_2.
3. Caso induttivo in cui r è ottenuta da una espressione regolare r_1. Chiama ricorsivamente Costruzione $\diamond$AutomaFinitoNonDeterministico(r_1) per ottenere il corrispondente automa δ_1.
   (a) Caso $r \equiv (r_1)^*$. 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 $r \equiv (r_1)$. Restituisci δ_1 senza modifiche.

Complessità. Sia $|r|$ la lunghezza dell'espressione regolare r. L'algoritmo di costruzione dell'automa esegue $O(|r|)$ passi di costruzione di automi semplici e $O(|r|)$ 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 $O(|r|)$ tempo e spazio.

2.3 Ricerca di espressioni regolari con l'automa

Vediamo come utilizzare l'automa nondeterministico δ per la ricerca di una espressione regolare r di lunghezza m in un testo T di lunghezza 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 alla espressione regolare r.

Esempio. Consideriamo l'automa δ per l'espressione $r = \mathtt{a}(\mathtt{b}|\mathtt{c})^*\mathtt{d}$, in cui aggiungiamo una transizione dallo stato iniziale verso se stesso etichettata con Σ. Simuliamo il comportamento dell'automa in corrispondenza del testo $T = \mathtt{aacbcd}$. Manteniamo un insieme Q di stati attivi, partendo con Q che contiene il solo stato iniziale 0. 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, $Q = \{0, 1, 2, 3, 4, 8\}$. 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 $Q = \{0, 2, 3, 4, 6, 7, 8\}$. Leggendo quindi `b`, abbiamo che $Q' = \{0, 5\}$ e $Q = \{0, 2, 3, 4, 5, 7, 8\}$. Il successivo carattere `c` fornisce i precedenti valori di $Q' = \{0, 6\}$ e $Q = \{0, 2, 3, 4, 6, 7, 8\}$. Infine, la lettura di `d` fornisce $Q = Q' = \{0, 9\}$, raggiungendo lo stato finale 9. Possiamo concludere che nella posizione corrente del testo T esiste una occorrenza di r.

Pseudocodice. L'algoritmo che ci accingiamo a descrivere segue la prima versione del programma di utilità `grep` nei sistemi operativi Linux e Unix. Consiste nella simulazione di un automa nondeterministico 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, memorizzato in un insieme Q. Nella generica iterazione, Q contiene tutti gli stati attivi in δ. La transizione multipla dell'automa in base al prossimo carattere $T[j]$ letto nel testo avviene in due passi. Nel primo passo, per ciascuno stato $s \in Q$ prendiamo il suo prossimo stato $\delta[s, T[j]]$ (ricordiamo che l'automa nondeterministico δ è tale per cui ci sono $|\delta[s, T[j]]| \leq 1$ stati oppure $|\delta[s, \epsilon]| \leq 2$ stati). In tal modo otteniamo un nuovo insieme di stati attivi Q'. Nel secondo passo, i valori di Q vengono ricalcolati propagando gli stati attivi in Q' attraverso le 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 attraverso una visita in profondità $DFS(s, \delta)$ dell'automa δ a partire da s. La visita interpreta δ come un grafo diretto in cui i vertici sono gli stati e gli archi sono le sole ϵ-transizioni, e inizia la sua computazione a partire dal nodo s. Il seguente algoritmo restituisce tutti gli stati raggiungibili con zero o più ϵ-transizioni in δ a partire dagli stati in Q'.

EPSILON(Q', δ):
1: $Q'' \leftarrow \{\}$; {*Insieme vuoto*}
2: FOREACH $s \in Q'$ DO
3: $\quad Q'' \leftarrow Q'' \cup DFS(s, \delta)$;
4: RETURN Q'';

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

RICERCA⋄AUTOMAFINITONONDETERMINISTICO(r, T):
1: $m \leftarrow |r|$; $n \leftarrow |T|$;
2: costruisci l'automa finito nondeterministico δ per r;
3: $Q \leftarrow \{0\}$; {*Stato iniziale* 0 *dell'automa*}
4: $Q \leftarrow$ EPSILON(Q, δ);
5: IF Q contiene lo stato finale THEN
6: $\quad$ RETURN TRUE;
7: FOR $j \leftarrow 1$ TO n DO
8: $\quad Q' \leftarrow \{\}$; {*Insieme vuoto*}
9: $\quad$ FOREACH $s \in Q$ DO
10: $\quad\quad Q' \leftarrow Q' \cup \delta[s, T[j]]$;
11: $\quad Q \leftarrow$ EPSILON(Q', δ);
12: $\quad$ IF Q contiene stato finale THEN
13: $\quad\quad$ RETURN TRUE;
14: RETURN FALSE;

Dopo aver costruito l'automa δ e 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) 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) 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 stata una realizzazione molto efficiente di tali idee.

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. Associamo un vettore binario agli stati dell'automa che sono al più $2m$, in modo che l'i-esimo elemento del vettore corrisponde allo stato i. L'insieme Q è mantenuto ponendo a 1 gli elementi corrispondenti agli stati in Q. In tal modo, possiamo eseguire l'algoritmo EPSILON in $O(m)$ tempo, interrompendo la visita in un vertice s se $s \in Q$ (possiamo verificare facilmente la condizione usando il vettore binario). Poiché ogni vertice ha al più due archi, il numero totale di nodi visitati da tutte le visite DFS è al più due volte il numero totale di stati. Ogni nodo richiede tempo costante per la sua visita, per cui una invocazione di EPSILON impiega $O(m)$ tempo. Un ragionamento analogo vale per il primo passo (linee 8–10) di ogni iterazione, in cui si pongono a 1 i soli elementi del vettore binario corrispondenti ai nuovi stati attivi. In conclusione, ogni iterazione richiede tempo $O(m)$. Dopo aver costruito l'automa δ in $O(m)$ tempo, RICERCA⋄AUTOMAFINITONONDETERMINISTICO esegue n iterazioni per un costo totale di $O(mn)$ tempo al caso pessimo.

La ricerca di una espressione regolare può essere effettuata, in alternativa, con un automa deterministico. Nella teoria dei linguaggi formali, le espressioni regolari, gli automi finiti nondeterministici 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 nondeterministici ad automi deterministici che può essere applicata all'automa risultante da COSTRUZIONE ⋄AUTOMAFINITONONDETERMINISTICO. L'algoritmo di ricerca diventa quindi la scansione (real-time) del testo T mediante automa in tempo lineare, analogamente alla procedura RICERCA⋄AUTOMA discussa nel Capitolo 1. Sfortunatamente, l'automa deterministico può avere un numero esponenziale in m di stati al caso pessimo. Per esempio, l'espressione regolare $(\mathtt{a}|\mathtt{b})^*\mathtt{a}(\mathtt{a}|\mathtt{b})\cdots(\mathtt{a}|\mathtt{b})$, in cui appaiono $k = \Theta(m)$ espressioni $(\mathtt{a}|\mathtt{b})$ alla sua destra, rappresenta una qualunque stringa che abbia il simbolo `a` a $k+1$ posizioni di distanza dalla fine. Il corrispondente automa deterministico richiede $\Omega(2^{\Theta(m)})$ stati, mentre per l'automa nondeterministico ne bastano $O(m)$. Di conseguenza, l'algoritmo di ricerca con automa deterministico costa $O(2^m + n)$ al caso pessimo e richiede spazio $O(2^m)$.

Esiste una elegante costruzione diretta di un automa deterministico a partire da una espressione regolare, senza passare per l'automa nondeterministico. 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 una area temporanea di appoggio per gli stati (cache). Anche se il caso pessimo non migliora, il comportamento pratico è lineare 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. La concatenazione ordinata di tali simboli fornisce la stringa P e, visto che la sequenza rappresenta un percorso obbligato, ogni occorrenza dell'espressione regolare r deve contenere P. L'idea è di impiegare come filtro la ricerca esatta di P impiegando algoritmi veloci di tipo Boyer-Moore-Hoorspool. Le posizioni del testo che non contengono P possono essere quindi ignorate; le rimanenti indicano che l'espressione regolare r potrebbe apparire. Per ogni occorrenza di P, scegliamo un intorno di posizioni adiacenti sufficientemente lungo, esaminando i corrispondenti caratteri con l'automa per r. In questa maniera l'algoritmo identifica tutte le occorrenze di r utilizzando le occorrenze di P come base di partenza, e può guadagnare un fattore di almeno 10 in velocità.

Capitolo 3

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! Possono essere di varia natura, rendendo inefficaci gli algoritmi di ricerca esatta descritta nel Capitolo 1. 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, in quanto spesso usano termini ormai decaduti.

Esistono tuttavia delle situazioni in cui gli errori sono inevitabili. I testi che si riferiscono a diversi modelli linguistici possono dar luogo a forme eterogenee che vanno trattare; 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 alla evoluzione molecolare rendono inefficaci le tecniche di ricerca esatta in quanto una porzione di sequenza può occorrere più volte variata con diversi cambiamenti.

La possibilità di ammettere errori e imprecisioni non solo allevia i problemi di ricerca appena descritti, ma permette una certa flessibilità nella specifica. Quante volte non ricordiamo esattamente come si scrive un nome straniero di luogo o di persona? Spesso alcuni di questi nomi ammettono traslitterazioni diverse. 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. Maggiore è la distanza, maggiore è l'errore. Dopo aver definito alcune distanze, 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 k errori. Tale ricerca può essere vista come un caso speciale di ricerca con espressioni regolari, per cui mostriamo anche un automa nondeterministico a stati finiti in grado di eseguirla.

3.1 Definizione di errore e di distanza

La definizione di errore può essere formulata matematicamente collegandola al concetto di *distanza* tra stringhe. Una distanza d è una funzione definita su coppie di stringhe dell'alfabeto Σ che restituisce un reale nonnegativo, $d : \Sigma^* \times \Sigma^* \to I\!R^+ \cup \{0\}$. 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: $d(x, x) = 0$ per ogni $x \in \Sigma^*$.
- Simmetrica: $d(x, y) = d(y, x)$ per ogni $x, y \in \Sigma^*$.
- Triangolare: $d(x, z) \leq d(x, y) + d(y, z)$ per ogni $x, y, z \in \Sigma^*$.

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 la stessa lunghezza m e misura il numero di caratteri che non corrispondono (chiamati *mismatch*): $d(x, y) = |\{i \mid x[i] \neq y[i] \text{ per } 1 \leq i \leq m\}|$. Per esempio, $d(\texttt{stringa}, \texttt{spranga}) = 2$ 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 x in una stringa y (qui ϵ denota il carattere vuoto):

- $\epsilon \to a$: inserzione di un singolo carattere a.
- $a \to \epsilon$: cancellazione di un singolo carattere a.

- $a \to b$: sostituzione (*mismatch*) di un singolo carattere a con un altro carattere $b \neq a$.

La distanza di edit $d(x, y)$ è quindi definita in termini della *minima* sequenza di operazioni di edit per trasformare x in y. Per convenzione, $d(x, y)$ denota il solo numero di operazioni nella sequenza minima, e tale numero può variare da 0 al massimo tra le lunghezze delle stringhe $\max\{|x|, |y|\}$. Assumiamo d'ora in poi che $|x| = m \leq n = |y|$, per cui $0 \leq d(x, y) \leq n$. Per esempio, $d(\texttt{ananas}, \texttt{banane}) = 3$, in quanto possiamo trasformare $\texttt{ananas}$ in $\texttt{banane}$ con tre operazioni di edit: $\texttt{ananas} \xrightarrow{\epsilon \to \texttt{b}} \texttt{\underline{b}ananas} \xrightarrow{\texttt{a} \to \texttt{e}} \texttt{banan\underline{e}s} \xrightarrow{\texttt{s} \to \epsilon} \texttt{banane}$. Poiché la distanza è simmetrica, cioè $d(x, y) = d(y, x)$, le inserzioni in x possono essere viste parimenti come cancellazioni da y, e le cancellazioni da x come inserzioni in y. Infatti, $d(\texttt{banane}, \texttt{ananas}) = 3$ e la corrispondente sequenza di edit è ottenuta per inversione: $\texttt{banane} \xrightarrow{\texttt{b} \to \epsilon} \texttt{anane} \xrightarrow{\texttt{e} \to \texttt{a}} \texttt{anan\underline{a}} \xrightarrow{\epsilon \to \texttt{s}} \texttt{anana\underline{s}}$.

Una maniera alternativa di visualizzare la sequenza di edit che non dipende dalla direzione di trasformazione (da x a y oppure da y a 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 $\sqcup$:

$$\begin{array}{ccccccc} \sqcup & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{n} & \underline{\texttt{a}} & \texttt{s} \\ \texttt{b} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{n} & \underline{\texttt{e}} & \sqcup \end{array}$$

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

$$\begin{array}{ccccccc} \sqcup & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{n} & \texttt{a} & \underline{\texttt{s}} \\ \texttt{b} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{n} & \sqcup & \underline{\texttt{e}} \end{array}$$

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:

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

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 q (chiamate q-grammi), oppure sulla possibilità di scambiare due o più caratteri in posizioni arbitrarie, come generalizzazione dell'operazione di trasposizione.

Tuttavia, la distanza di edit è sicuramente 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_{ab} in cui si specifica il costo di sostituzione del carattere a con il carattere b. Le inserzioni e le cancellazioni vengono viste come una sostituzione in cui il carattere vuoto ϵ è coinvolto, per cui $W_{\epsilon a}$ è il peso dell'inserimento visto come sostituzione di ϵ con il carattere a, e $W_{a\epsilon}$ è il peso della cancellazione vista come la sostituzione del carattere a con ϵ. Tali matrici sono impiegate dai biologi molecolari che ne fissano i valori in modo che le operazioni di edit più probabili 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.

3.2 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 $n \geq m$ 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, utilizzando 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 $0 \leq i \leq m$ e $0 \leq j \leq n$. 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 ci 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 $1 \leq i \leq m$ 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 $1 \leq j \leq n$ in quanto trasformare la stringa vuota in una stringa di lunghezza j richiede l'inserzione di j caratteri.

Di diversa natura è la regola per riempire l'elemento $D[i,j]$ quando $i,j > 0$.

$$D[i,j] = \min\{\, D[i-1,j-1] + [\![x[i] \neq y[j]]\!],\; D[i-1,j]+1,\; D[i,j-1]+1 \,\}, \tag{3.1}$$

dove l'espressione $[\![x[i] \neq y[j]]\!]$ vale 1 se i due caratteri $x[i]$ e $y[j]$ sono diversi, e vale 0 se sono uguali. La regola si basa sull'induzione, dove la colonna 0 e la riga 0 delle condizioni al contorno rappresentano il caso base. Assumiamo di aver calcolato correttamente i valori di D che si trovano nel quadrante in alto a sinistra di $D[i,j]$ (escluso), ovvero conosciamo tutti i valori corretti di $D[i',j']$ per $0 \leq i' \leq i$ e $0 \leq j' \leq j$ tranne $D[i,j]$ (che vogliamo calcolare). Mostriamo la validità della regola (3.1).

Sia A un allineamento tra $x[1..i]$ e $y[1..j]$ che corrisponde alla loro distanza di edit. In A prendiamo l'ultima colonna ed esaminiamo il suo contenuto dall'alto in basso, dando luogo a quattro casi possibili:

- La colonna contiene due simboli che sono uguali. Questi simboli devono essere $x[i]$ e $y[j]$. Ne consegue che la distanza di edit rimane uguale a $D[i-1,j-1]$. Per assurdo, se fosse minore, lo sarebbe anche $D[i-1,j-1]$!

[1] Il termine storico di programmazione dinamica è un po' fuorviante. In realtà si tratta di una tabella riempita (dinamicamente?) con una regola ricorsiva.

- La colonna contiene due simboli che sono diversi e nessuno dei due è uno spazio bianco ␣. Abbiamo un mismatch tra $x[i]$ e $y[j]$; quindi dobbiamo incrementare $D[i-1,j-1]$ di 1 per ottenere la distanza di edit. Per assurdo, se fosse minore, lo sarebbe anche $D[i-1,j-1]$.
- 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 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.
- 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 risultante è perciò $D[i,j-1]$ incrementata di 1 a causa dell'inserzione. Non può essere minore perché altrimenti lo sarebbe anche $D[i,j-1]$.

Come si può notare, l'ipotesi induttiva viene usata per dichiarare che i valori delle distanze $D[i-1,j-1]$, $D[i-1,j]$ e $D[i,j-1]$ sono quelli minimi, per cui un valore di $D[i,j]$ inferiore a quelli discussi sopra porterebbe per assurdo a rendere inferiori anche i valori di $D[i-1,j-1]$, $D[i-1,j]$ e $D[i,j-1]$, rispettivamente. Tuttavia, non siamo a conoscenza dell'allineamento A al momento del calcolo di $D[i,j]$, e non possiamo applicare i singoli casi della casistica riportata sopra. Sappiamo però che deve essere uno dei valori nella formula (3.1). Prendendo il minimo tra tali distanze, tutte valide, siamo certi di scegliere il valore corretto per la distanza $D[i,j]$.

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 (3.1) 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 a sinistra.

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

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 sono in ordine non decrescente (e chiaramente differiscono di al più 1).

Lo pseudocodice per il calcolo della tabella D di programmazione dinamica segue fedelmente la formula (3.1) 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 $D[i,j-1]$ sono stati correttamente trovati.

DistanzaEdit(x, y):
```
m ← |x|; n ← |y|;
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];
```

Volendo ammettere anche le trasposizioni tra le operazioni possibili di edit, basta aggiungere un nuovo caso alle righe 8–11, 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:     D[i,j] ← 1 + MINIMO(D[i−1, j−1], D[i−1, j], D[i, j−1]);
```

Nel caso più generale di distanza con i pesi W_{ab}, 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 $D[-1,-1] = 0$ in modo che $D[0,0]$ sia correttamente posto a 0. Assumiamo inoltre che $x[0] = y[0] = \epsilon$. 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];
```

In tutti i casi visti sopra, ogni elemento della tabella D può essere riempito in tempo costante. Essendoci $O(mn)$ elementi in D, la complessità risultante è di $O(mn)$ tempo e spazio. Lo spazio può essere ridotto a $O(m)$ manenendo le ultime due colonne calcolate $j-1$ e j.

Una volta stabilita la distanza di edit $d(x,y)$ mediante la tabella D, vorremmo poter ottenere anche le sequenze di trasformazione da x a y (o i loro allineneamenti). È possibile utilizzare la matrice D a ritroso, partendo dall'elemento $D[m,n]$. Discutiamo il caso di distanza con costi uniformi, in quanto il caso della distanza pesata è analogo. Dato un elemento $D[i,j]$, possiamo ricostruire le operazioni di edit in x:

- se $D[i,j] = D[i-1, j-1]$, non ci sono operazioni di edit;
- se $D[i,j] = D[i-1, j-1] + 1$, abbiamo una sostituzione di $x[i]$ con $y[j]$;
- se $D[i,j] = D[i-1, j] + 1$, abbiamo una cancellazione di $x[i]$;
- se $D[i,j] = D[i, j-1] + 1$, abbiamo una inserzione di $y[j]$.

Più casi possono valere simultaneamente per cui vanno tutti considerati in quanto non sappiamo quali porterranno a soluzione. 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. Ogni volta che raggiungiamo l'elemento $D[0,0]$, abbiamo una distinta sequenza di trasformazione.

Una maniera concettualmente più semplice per ricostruire la sequenza di trasformazione consiste nell'utilizzare il *grafo* della edit distance. I vertici del grafo sono i punti di intersezione di una griglia di taglia $(m+1) \times (n+1)$, per cui il vertice che si trova all'incrocio della riga i e della colonna j viene indicato con (i,j). Oltre agli archi orizzontali diretti da $(i, j-1)$ a (i,j) e agli archi verticali diretti da $(i-1, j)$ a (i,j), vi sono gli archi diagonali che sono diretti da $(i-1, j-1)$ a (i,j). Gli archi hanno costo 1, tranne gli archi diagonali da $(i-1, j-1)$ a (i,j) tali che $x[i] = y[j]$, il cui costo è 0. In 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 $d(x,y)$ rappresentano tutte le possibili sequenze di trasformazioni di x in y con una regola analoga a quelle citate sopra in base all'arco attraversato:

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

3.3 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 riga viene considerata come un simbolo dell'alfabeto. Non essendo difficile stabilire se due righe di testo sono uguali o meno, la distanza di edit ci fornisce il minimo numero di cambiamenti di riga per trasformare un file in un altro. In realtà, vorremmo trovare anche le righe comuni tra due file, analogamente al comando `comp` in Unix. A tal fine, possiamo riformulare il nostro problema di distanza in termini di *sottosequenza* che, a differenza della sottostringa, è una sequenza in cui i caratteri non sono necessariamente contigui nella stringa di partenza. La distinzione non è solo concettuale, in quanto le tecniche e i risultati ottenibili per le sottostringhe sono diversi da quelli per le sottosequenze (in generale più difficili da trattare). Quindi, 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. Siamo interessati a trovare la più lunga, chiamata appunto la *sottosequenza comune più lunga* $s(x, y)$ tra x e y. Esiste una precisa relazione tra la lunghezza $|s(x, y)|$ e la distanza di edit $d'(x, y)$ in cui sono ammesse *solo* inserzioni e cancellazioni:

$$d'(x, y) = |x| + |y| - 2|s(x, y)|.$$

In altre parole, il numero minimo di operazioni di edit può essere calcolato prendendo la sottosequenza comune più lunga $s(x, y)$, per cui sappiamo che $2|s(x, y)|$ caratteri combaciano a coppie e gli altri caratteri vanno inseriti o cancellati. La sottosequenza $s(x, y)$ 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 $d'(x, y)$, rendiamo massima la lunghezza $|s(x, y)|$ della sottosequenza comune.

È possibile calcolare direttamente $|s(x, y)|$ usando una tabella di programmazione dinamica S, tale che l'elemento $S[i, j]$ contiene la lunghezza della sottosequenza comune più lunga tra $x[1..i]$ e $y[1..j]$, dove $0 \leq i \leq m$ e $0 \leq j \leq n$. Le condizioni al contorno sono $S[i, 0] = 0$ e $S[0, j] = 0$ poiché se una delle due stringhe è vuota allora l'unica sottosequenza comune è quella vuota. Per $i, j > 0$, abbiamo $S[i, j] = S[i-1, j-1] + 1$ se $x[i] = y[j]$; altrimenti $S[i, j] = \max\{S[i-1, j], S[i, j-1]\}$ se $x[i] \neq y[j]$. 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 $O((r + n) \log n)$, dove r è il numero di possibili coppie di caratteri uguali nelle stringhe x e y, dove il primo elemento di ogni coppia appartiene a x e il secondo a y. Pur essendo $r = O(n^2)$ al caso pessimo, in pratica $r = O(n)$ quando applichiamo l'algoritmo alla differenza tra file, in cui i caratteri dell'alfabeto sono le righe dei file. In tal caso, trovare la sottosequenza comune più lunga richiede solo $O(n \log n)$ tempo e $O(n)$ spazio, portando a una realizzazione alternativa di `comp` senza cambiare funzionalità ma migliorandone decisamente le prestazioni.

3.4 Ricerca con al più k errori

La distanza di edit serve nel problema della ricerca approssimata con k errori o differenze, in cui vogliamo individuare le sottostringhe del testo T che sono a distanza al più k dal pattern P. Tuttavia, essendoci potenzialmente un numero quadratico di tali sottostringhe, siamo interessati a sapere in quali posizioni j del testo terminano le occorrenze di tali sottostringhe. Avendo al più n posizioni, riduciamo la dimensione dei dati in uscita prodotti dalle soluzioni al problema. Formalmente, dati una stringa testo T di lunghezza n, una stringa pattern P di lunghezza m e fissata una soglia massima $k < m$ per la distanza di edit $d()$, vogliamo trovare trovare tutte le posizioni j tali che $d(P, T[j'..j]) \leq k$ per almeno un valore j' con $1 \leq j' \leq j \leq n$. Si noti che, a differenza della ricerca esatta, la finestra di ricerca nel testo

ha una dimensione variabile $m - k \leq j - j' + 1 \leq m + k$, dove $m - k$ corrisponde ad avere tutte cancellazioni e $m + k$ ad avere tutte inserzioni. Nelle applicazioni reali, l'intervallo di valori tipicamente usati sono $5 \leq m \leq c \cdot 100$ per la lunghezza del pattern, per una qualche costante $c > 0$, con i pattern corti più frequenti nel recupero dei documenti e i pattern lunghi diffusi nella biologia molecolare; abbiamo quindi $1 \leq k \leq m/2$ per il numero di errori, e $c \cdot 10^3 \leq n \leq c \cdot 10^9$ per la lunghezza del testo.

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

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 nel caso della ricerca approssimata. Nell'iterazione j usiamo il vettore C per memorizzare la colonna $j - 1$ della matrice D e il vettore C' per calcolare i valori della colonna j. Durante ogni iterazione, manteniamo l'invariante che $C[h] = D[h, j-1]$ e $C'[h] = D[h, j]$ per $0 \leq h \leq m$. 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 una occorrenza approssimata di P con al più k errori che termina nella posizione j.

```
RicercaApprossimata(P, T, k):
m ← |P|; n ← |T|;
FOR i ← 0 TO m DO
  C[i] ← i;   {Colonna 0}
FOR j ← 1 TO n DO
  FOR i ← 1 TO m DO
    IF P[i] = T[j] THEN
      C'[i] ← C[i − 1];
    ELSE
      C'[i] ← 1 + Minimo(C[i − 1], C'[i − 1], C[i]);
  C ← C';
  IF C[m] ≤ k THEN
    RETURN TRUE;   {Occorrenza approssimata di P che termina in j}
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. La complessità di RicercaApprossimata rimane $O(nm)$ tempo al caso pessimo mentre lo spazio si riduce a $O(m)$. Esistono soluzioni recenti che migliorano il tempo al caso pessimo, per esempio ottenendo un costo di $O(nk)$, ma non sono discusse in queste dispense.

L'algoritmo RicercaApprossimata(P, T, k) viene utilizzato anche per il cosiddetto metodo di "best match", in cui si trova il minimo $\hat{k} \geq 0$ per cui c'è un'occorrenza di P con $\hat{k}$ errori. Possiamo iniziare fissando $k = 1, 2, 4, \ldots$ per le potenze del due fino a che non troviamo una occorrenza per $k = 2^r$. A questo punto sappiamo che $2^{r-1} < \hat{k} \leq 2^r$, per cui basta eseguire una variante della ricerca binaria su k, in cui ogni volta che fissiamo un valore di k eseguiamo

RicercaApprossimata(P, T, k). Se troviamo una occorrenza decrementiamo k di una opportuna potenza del due; altrimenti lo incrementiamo. Alla fine, dobbiamo trovare $k = \hat{k}$. Se utilizziamo l'algoritmo con costo $O(nk)$, il costo totale è proporzionale a n moltiplicato una somma geometrica proporzionale a $\hat{k}$, per cui il costo finale è comunque $O(n\hat{k})$ invece che $O(nm)$, anche se non conosciamo a priori il valore di $\hat{k}$

3.5 Automi per la ricerca con errori

La ricerca con k differenze di un pattern P può essere vista come caso speciale della ricerca con le espressioni regolari vista nel Capitolo 2. Per convincerci di ciò, mostriamo un automa nondeterministico che permette di risolvere il problema della ricerca con k differenze. È anche possibile definire un automa deterministico ma richiede tempo e spazio esponenziale (si vedano le espressioni regolari).

Gli stati dell'automa nondeterministico sono organizzati su $k+1$ righe, dove ciascuna riga è composta da $m+1$ stati. Le righe sono numerate da 0 a k, e gli stati in ciascuna riga sono numerati da 0 a m in aggiunta al numero di riga. In generale, lo stato (e, i) denota lo stato numero i nella riga e, ed è attivo se i primi i caratteri $P[1..i]$ del pattern occorrono nel testo con al più e errori terminando nella posizione corrente del testo. Gli stati finali nell'automa sono quelli dell'ultima colonna a destra oppure dell'ultima diagonale completa a destra.

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, se uno stato (e, i) è attivo, lo sono tutti quelli (e', i) con $e \leq e' \leq k+1$, in quanto avere al più e errori implica avere al più $e+1$ errori, $e+2$ errori, ecc. 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 per ogni i il valore minimo e per cui lo stato (e, i) è attivo, otteniamo esattamente $D[i, j] = e$.

La complessità della simulazione dell'automa nondeterministico per P con il testo T richiede $O(mn)$ tempo, come nelle espressioni regolari. Lo schema è però di per sé interessante e viene adottato nel comando `agrep`. 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 visto nel Capitolo 1.

Capitolo 4

Ricerca Senza Confronti Diretti

Gli algoritmi di ricerca discussi finora si basano sui confronti tra caratteri per stabilire se un dato pattern occorre nel testo, senza sfruttare le particolarità dell'alfabeto Σ. In realtà, l'alfabeto usato è codificato in binario, come ASCII e Unicode/UTF8, per cui è possibile impiegare i caratteri in operazioni che manipolano i bit delle parole di memoria. In questo capitolo, 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`. Descriviamo prima le operazioni di base per manipolare i bit di cui abbiamo bisogno. Poi mostriamo come risolvere il problema della ricerca esatta con tali meccanismi per poi estenderli alla distanza di edit e alle espressioni regolari.

4.1 Operazioni logiche e algoritmi aritmetici

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 k operazioni AND , OR e NOT bit a bit, siamo in grado di verificare simultaneamente w condizioni booleane al costo di k operazioni. Gli algoritmi aritmetici portano all'estremo questa possibilità di manipolazioni dei bit. La tabella seguente riporta alcune operazioni logiche comuni, adottate per esempio nel linguaggio di programmazione C.

$\vert$	OR logico in parallelo bit a bit
&	AND logico in parallelo bit a bit
$\wedge$	XOR logico in parallelo bit a bit
$\sim$	NOT logico in parallelo bit a bit
$<< r$	spostamento dei bit di r posizioni a sinistra
$>> r$	spostamento dei bit di 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 $\mathit{rshift}(x)$ che effettua uno spostamento a destra di tutti i bit di x mettendo 1 nella posizione più a sinistra lasciata libera. Ciò si ottiene eseguendo $(x >> 1 | 10 \cdots 0)$, dove $x >> 1$ effettua lo spostamento a destra di una posizione; l'OR con $10 \cdots 0$ forza ad avere 1 in tale posizione.

Nel seguito indicheremo con $b_1 \cdots b_w$ i bit in una parola di memoria, dove b_1 è il bit di peso maggiore e 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.

4.2 Paradigma Shift-And

Un 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 una 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 Ricerca⋄BoyerMooreHorspool. Discutiamo tale paradigma perché è la base di un algoritmo di ricerca più complesso descritto in seguito.

Nella descrizione dell'algoritmo, assumiamo per semplicità che $m = w$. Vedremo poi come rimuovere questa assunzione. L'algoritmo mantiene una parola di memoria $D = d_1 \ldots d_m$ per lo stato corrente, con la seguente invariante: dopo aver letto il carattere corrente $T[j]$ dal testo, per $1 \le j \le n$, vale $d_i = 1$ per $1 \le i \le m$ se e solo se i primi caratteri $P[1..i]$ del pattern occorrono come suffisso dei primi caratteri $T[1..j]$ del testo. Per esempio, ponendo $P =$ `ananas` e $T =$ `banananassata` otteniamo le seguenti configurazioni di D, in cui sono evidenziati i prefissi del pattern che occorrono terminando 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 nel 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` occorrono come suffissi di $T[1..7] =$ `bananan`. Quando $d_m = 1$, abbiamo una 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 . Innanzi tutto, D viene inizializzata in modo da contenere tutti 0. Poi viene creata una tabella B composta da $|\Sigma|$ parole di memoria. La parola $B[c] = b_1 \ldots b_m$ corrispondente al carattere $c \in \Sigma$, 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 1 segnalano la presenza del carattere c nel pattern P. Nel nostro esempio:

	`ananas`
$B[$`a`$]$	`101010`
...	`000000`
$B[$`n`$]$	`010100`
...	`000000`
$B[$`s`$]$	`000001`
...	`000000`

Al passo j, supponiamo di aver calcolato correttamente D fino al passo $j-1$ e di voler calcolare il nuovo valore di D, denotato da D', dopo il passo j. Quando leggiamo il carattere corrente $T[j]$ del testo, abbiamo che $P[1..i]$ occorre come suffisso di $T[1..j]$ se e solo se $P[1..i-1]$ occorre come suffisso di $T[1..j-1]$ e $P[i] = T[j]$. Fortunatamente, possiamo esprimere le due condizioni mediante D e B, usando le operazioni logiche sui bit. Innanzi tutto, vale che l'$(i-1)$-esimo bit in D vale 1 se e solo se $P[1..i-1]$ occorre come suffisso di $T[1..j-1]$. Inoltre, l'i-esimo bit della parola $B[T[j]]$ vale 1 se e solo se $P[i] = T[j]$. Vogliamo mettere in AND queste due condizioni ponendo a 1 l'i-esimo bit di D' se e solo se entrambe valgono. Basta allora spostare i bit di D di una posizione a destra (in modo che l'$(i-1)$-esimo bit in D occupi la posizione i) mettendoli in AND parallelo con la parola $B[T[j]]$. Per $1 \le i \le m$, se i bit in posizione i sono entrambi 1 abbiamo che entrambe le condizioni sono soddisfatte. Possiamo concludere che $P[1..i]$ occorre come suffisso di $T[1..j]$. Per calcolare D' basta quindi eseguire

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

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

RICERCA◇SHIFTAND(P, T):
$m \leftarrow |P|$; $n \leftarrow |T|$;
FOREACH $c \in \Sigma$ DO
  $B[c] \leftarrow 0 \cdots 0$;
FOR $i \leftarrow 1$ TO m DO
  i-esimo bit di $B[P[i]] \leftarrow 1$;
$D \leftarrow 0 \cdots 0$;
FOR $j \leftarrow 1$ TO n DO
  $D \leftarrow \mathit{rshift}(D) \,\&\, B[T[j]]$
  IF m-esimo bit di D è 1 THEN
    RETURN TRUE; $\{$*Occorrenza* $T[j-m+1..j] = P\}$
RETURN FALSE;

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 $P = \mathtt{a}[\mathtt{n},\mathtt{s}]\mathtt{anas}$. Per trattare una ricerca di questo tipo, basta modificare le maschere B. Il resto dell'algoritmo non cambia. Nel nostro esempio, il secondo bit di $B[\mathtt{s}]$ va posto a 1, per cui $B[\mathtt{s}] = 010001$. 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), 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 maniera 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 $c \in \Sigma$ che sia diverso $c \neq c_1, \ldots, c_s$.

La complessità di RICERCA◇SHIFTAND è di $O(|\Sigma| + m + n) = O(n)$ tempo in quanto tutte le istruzioni nelle singole linee richiedono tempo costante. Abbiamo finora assunto che $m = w$. Se $m < w$, possiamo utilizzare i primi m bit delle parole ponendo a 0 i restanti. Se $m > w$, possiamo dividere ciascuna sequenza di m bit in $\lceil m/w \rceil$ parole di w bit ciascuna. La simulazione di una operazione logica su m bit ora richiede tempo $O(\lceil m/w \rceil)$, che va moltiplicato per la complessità precedente dando luogo a $O(\lceil m/w \rceil n)$ tempo totale. L'algoritmo ha complessità lineare quando $w = O(m)$. Lo spazio richiesto è $O(|\Sigma| + m)$.

4.3 Estensione del paradigma: algoritmi per `agrep`

Il paradigma SHIFT-AND può essere esteso per trattare espressioni regolari e ricerche approssimate con al più k errori, ottenendo degli algoritmi impiegati in `agrep`, una estensione di `grep` di Wu e Manber. [1] 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 (si veda il Capitolo 3 per le definizioni). In un certo senso, viene simulato l'automa non deterministico descritto nella Sezione 3.5.

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

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. Per $0 \leq e \leq k$, la parola R_e codifica le occorrenze del pattern con al più e errori, analogamente alla riga e dell'automa non deterministico. In particolare, dopo aver letto il carattere corrente $T[j]$ del testo, l'i-esimo bit di R_e vale 1 se e solo se $P[1..i]$ occorre come suffisso di $T[1..j]$ con al più e errori. Inizialmente poniamo $R_e = 1^e 0^{m-e}$, ovvero e copie di 1 seguite da $m - e$ copie di 0. Per risolvere il problema della ricerca approssimata, basta inoltre verificare che l'ultimo bit in R_k valga 1.

Anche se siamo interessati a R_k, le altre parole di memoria servono a calcolare efficientemente R_k. Diamo perciò le regole per aggiornare correttamente le parole R_e, indicando con R'_e il risultato dell'aggiornamento dopo aver letto il carattere corrente $T[j]$ del testo.

La parola R_0 corrisponde alle occorrenze esatte e quindi viene aggiornata come discusso nella Sezione 4.2:

$$R'_0 \leftarrow \mathit{rshift}(R_0) \,\&\, B[T[j]].$$

La parola R_e per $e > 0$ deve mettere in OR i quattro casi possibili (match, sostituzione, cancellazione e inserzione) analogamente alle frecce dell'automa non deterministico descritto nella Sezione 3.5:

- Match tra $P[i]$ e $T[j]$ (freccia orizzontale etichettata con $P[i]$): qui $P[1..i-1]$ occorre con al più e errori come suffisso di $T[1..j-1]$ e vale $P[i] = T[j]$. L'espressione corrispondente è analoga a quella di R_0, ovvero $\mathit{rshift}(R_e) \,\&\, B[T[j]]$.
- Sostituzione di $P[i]$ con $T[j]$ (freccia diagonale etichettata con Σ): abbiamo che $P[1..i-1]$ deve occorrere con al più $(e-1)$ errori come suffisso di $T[1..j-1]$ e deve valere $P[i] \neq T[j]$. L'espressione corrispondente alla prima condizione è $\mathit{rshift}(R_{e-1})$. Si noti che possiamo evitare di metterla in AND con il complemento $\sim B[T[j]]$ corrispondente alla seconda condizione (perché?).
- Cancellazione di $P[i]$ (freccia diagonale etichettata con ϵ): qui $P[1..i-1]$ deve occorrere con al più $(e-1)$ errori come suffisso di $T[1..j]$, per cui l'espressione è $\mathit{rshift}(R'_{e-1})$.
- Inserzione di $T[j]$ (freccia verticale etichettata con Σ): in questo caso, $P[1..i]$ deve occorrere con al più $(e-1)$ errori come suffisso di $T[1..j-1]$, per cui l'espressione è R_{e-1}.

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

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

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

Lo pseudocodice per la realizzazione dell'algoritmo è analogo a RICERCA◇SHIFTAND, con la differenza che si gestiscono $k+1$ parole invece che una:

RICERCAAPPROSSIMATA◇SHIFTAND(P, T, k):
$m \leftarrow |P|$; $n \leftarrow |T|$;
FOREACH $c \in \Sigma$ DO
  $B[c] \leftarrow 0$;

```
4: FOR i ← 1 TO m DO
5:    i-esimo bit di B[P[i]] ← 1;
6: FOR e ← 0 TO k DO
7:    R_e ← 1^e 0^(m−e);
8: FOR j ← 1 TO n DO
9:    R_0 ← rshift(R_0) & B[T[j]]
10:   FOR e ← 1 TO k DO
11:      R_e ← rshift(R_e) & B[T[j]] | rshift(R_{e−1} | R'_{e−1}) | R_{e−1};
12:   IF m-esimo bit di R_k è 1 THEN
13:      RETURN TRUE;   {Occorrenza approssimata di P che termina in j}
14: RETURN FALSE;
```

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

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 una automobile attraverso la sua targa `AX318YZ`, di cui però non ricordiamo se il numero è esattamente `318`, possiamo racchiudere tra parentesi angolari le posizioni del pattern $\langle$`AX`$\rangle$318$\langle$`YZ`$\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 11 dello pseudocodice diventa

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

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

Il paradigma SHIFT-AND è utile anche con le espressioni regolari viste nel Capitolo 2. Data una espressione regolare r, costruiamo il suo automa non deterministico δ e numeriamo i suoi stati da 0 a m, in cui lo stato iniziale è 0 e lo stato finale è m, dove $m \leq 2|r|$ in quanto il numero di stati è al più due volte la lunghezza dell'espressione regolare. La numerazione va fatta mediante una visita sul grafo indotto dall'automa δ, in modo da garantire che due stati hanno numeri consecutivi quando sono collegati da una transizione etichettata con un simbolo dell'alfabeto Σ. Tale numerazione è sempre possibile perché abbiamo visto nella Sezione 2.2 che ogni stato ha in uscita o una singola transizione etichettata con un simbolo dell'alfabeto Σ oppure al più due ϵ-transizioni. Associamo quindi una parola di memoria $R = r_1 \cdots r_m$ all'automa δ, in modo che l'i-esimo bit $r_i = 1$ se e solo se lo stato i è attivo; non c'è bisogno del bit r_0 in quanto lo stato 0 è sempre attivo. Quando $r_m = 1$ abbiamo una occorrenza dell'espressione regolare che termina nella posizione corrente del testo.

Il punto è come aggiornare R al passo j, ottenendo il nuovo valore R' dopo aver letto il carattere $T[j]$ dal testo. Fortunatamente, le transizioni multiple dell'automa in base a $T[j]$ possono essere gestite come le parole D' e R'_0 negli algoritmi precedenti. Con riferimento alla Sezione 2.3, avremo che la relazione tra Q' (ovvero l'insieme i prossimi stati dell'automa nell'algoritmo RICERCA⋄AUTOMAFINITONONDETERMINISTICO) e R' si esplica osservando che lo stato i appartiene a Q' se e solo se l'i-esimo bit di R' vale 1.

La difficoltà concettuale risiede quindi nella simulazione della procedura EPSILON(Q', δ) descritta nella Sezione 2.3 per la propagazione attraverso le ϵ-transizioni di δ a partire dagli stati in Q', ottenendo l'insieme Q'' indicato nella procedura. L'idea di base è di porre a 1 i bit di R' corrispondenti agli stati in Q'', calcolandosi in modo preliminare delle tabelle ausiliari.

Come elaborazione preliminare, costruiamo prima una maschera binaria $E = e_1 \cdots e_m$ tale che $e_i = 1$ se e solo se lo stato i ha ϵ-transizioni. Poi dividiamo E in $\lceil m/8 \rceil$ byte (eventualmente appendendo un opportuno numero di 0 in fondo a E) e costruiamo $\lceil m/8 \rceil$ tabelle ausiliari. La k-esima tabella ausiliare corrisponde al k-esimo byte di E. In particolare, la tabella ha al più 256 parole di m bit, una parola per ogni possibile combinazione di valori 1 in E. Osserviamo che, non sapendo a priori quali gruppi di stati saranno attivi, dobbiamo considerare tutte le possibilità, che sono però al più 256 per il k-esimo byte di E. Eseguiamo quindi EPSILON su ciascuna combinazione e, una volta trovati gli stati raggiungibili con ϵ-transizioni, poniamo a 1 i rispettivi bit nella parola della tabella associata alla combinazione. In totale, abbiamo al più $256\lceil m/8 \rceil = O(m)$ parole nelle tabelle (in pratica sono molto meno). Possiamo inizializzarle in $O(m^2)$ tempo in modo ovvio; con opportuni trucchi tale costo può essere reso $O(m)$.

Terminata l'elaborazione preliminare, passiamo a simulare l'automa calcolando R'. Abbiamo già visto che non è un problema calcolare le transizioni multiple per $T[j]$; eseguiamo $R' \leftarrow \mathit{rshift}(R) \,\&\, B[T[j]]$ come negli algoritmi precedenti. Mostriamo quindi come simulare la propagazione, tramite ϵ-transizioni, dei bit 1 in R' per gli stati attivi. Mascheriamo R' con E, generando la parola di memoria $S = R' \,\&\, E$ che seleziona i soli stati con ϵ-transizioni (gli altri non hanno bisogno di essere propagati con EPSILON). Dividiamo S in byte e, preso il k-esimo byte, accediamo alla k-esima tabella ausiliare per recuperare la parola S_k associata alla combinazione di bit 1 presente nel k-esimo byte. La proprietà interessante è che facendo l'OR delle $\lceil m/8 \rceil$ parole S_k così ottenute, creiamo una parola U tale che l'i-esimo bit di U vale 1 se e solo se lo stato i appartiene a Q''. In altre parole, abbiamo propagato gli stati attivi attraverso le ϵ-transizioni. A questo punto basta aggiornare R ponendolo uguale a $R' \,\&\, U$. Lavorando su $O(\lceil m/w \rceil)$ sequenze di $\Theta(w)$ bit in S invece che sui byte di S, possiamo realizzare un passo di simulazione dell'automa in $O(\lceil m/w \rceil)$ tempo.

Il costo dell'intera computazione per trattare le espressioni regolari può essere analizzato in maniera analoga alle altre ricerche, per cui otteniamo $O(\lceil m/w \rceil n)$ tempo e $O(|\Sigma| + m)$ spazio. È possibile combinare gli approcci finora visti per cercare espressioni regolari con errori, ma tale argomento non viene discusso in queste dispense.

Vediamo infine come l'applicazione `agrep`, molto diffusa nel campo della ricerca testuale, adatta gli algoritmi 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 1.4); 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 1.6). 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 3.4) in cui si trova il minimo $k \geq 0$ per cui c'è un'occorrenza del pattern con k errori.

Parte II

STRUTTURE DI DATI PER TESTI

Questa parte descrive le strutture di dati più diffuse nella costruzione di indici per testi. Quasi tutte le strutture esposte realizzano il concetto di ricerca per prefissi, in cui vengono individuate le chiavi che hanno un certo prefisso. Precisamente, dato un insieme S di stringhe, vogliamo costruire delle strutture di dati per memorizzare S. Le strutture di dati devono supportare delle ricerche in cui vanno identificate tutte le stringhe in S aventi come prefisso una stringa arbitraria P fornita in ingresso, ovvero i caratteri iniziali delle stringhe così trovate devono essere quelli specificati in P. Per esempio, la ricerca di $P =$ `VE` nell'insieme S composto dai nomi di alcune province italiane

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

fornisce come risultato le stringhe `VERBANIA`, `VERCELLI` e `VERONA`. Nate per ricerche di questo tipo, le strutture di dati impiegate a questo scopo sono talmente flessibili da risultare utili per altri tipi di ricerche più complesse che sono via via discusse nei capitoli successivi; quasi tutte, inoltre, permettono di ottenere la sequenza di chiavi in ordine lessicografico mediante una loro semplice visita.

Tali strutture vengono introdotte nella Parte II per la loro importanza nella costruzione di indici testuali che evitano di esaminare interamente i testi a ogni richiesta, contrariamente agli algoritmi di ricerca testuali descritti nella Parte I che necessitano ogni volta una scansione completa dei testi. Il loro impiego nella costruzione degli indici si basa sulla seguente *proprietà fondamentale*:

> Un pattern P occorre in posizione i del testo T se e solo se P è prefisso del suffisso $T[i..n]$.

In altre parole, il problema della ricerca di un pattern descritto nella Parte I può essere formulato in termini della ricerca per prefissi che viene discussa nella Parte II. In tale contesto, l'insieme S non è costituito da stringhe indipendenti, bensì dai *suffissi* del testo. Per esempio, se $P =$ `ananas` e $T =$ `bananananassata`, fissiamo il contenuto dell'insieme S in modo da avere $n = 13$ stringhe scelte come i suffissi di T, dove il suffisso $T[i..n]$ ha numero i per $1 \leq 1 \leq n$:

1. `bananananassata` 2. `anananassata` 3. `nananassata` 4. `ananassata` 5. `nanassata`
6. `anassata` 7. `nassata` 8. `assata` 9. `ssata` 10. `sata` 11. `ata` 12. `ta` 13. `a`

La proprietà fondamentale riduce quindi il problema di cercare P in T al problema di individuare le stringhe di S che hanno P come prefisso. Nel nostro caso, troviamo una sola stringa, il suffisso `ananassata` con numero $i = 4$, e possiamo fornire in uscita la risposta che P occorre in posizione $i = 4$ di T. È quindi importante realizzare l'operazione di ricerca per prefissi nel modo più efficiente possibile.

La proprietà fondamentale può essere ristretta a un sottoinsieme dei suffissi (per esempio gli inizi di parola nel testo), restringendo così l'insieme delle possibili occorrenze del pattern alle sole posizioni del testo associate ai suffissi del sottoinsieme. Continuando l'esempio, supponiamo di considerare le occorrenze dei pattern nelle sole posizioni pari del testo. Fissiamo perciò S come l'insieme contenente

2. `anananassata` 4. `ananassata` 6. `anassata` 8. `assata` 10. `sata` 12. `ta`

e procediamo nella ricerca come prima. Una tale soluzione permette di risparmiare spazio nelle applicazioni che richiedono la ricerca di pattern soltanto in alcune porzioni del testo.

Inoltre, è interessante notare che la proprietà fondamentale vale non solo per la ricerca esatta vista nel Capitolo 1, ma anche per la ricerca con espressioni regolari vista nel Capitolo 2 e per la ricerca con errori vista nel Capitolo 3. Di conseguenza quasi tutte le strutture dati che sono discusse nei capitoli seguenti possono essere impiegate efficientemente come indici di testi in quanto supportano le ricerche con prefissi. Basta impiegarle per memorizzare i suffissi del testo.

Capitolo 5

Alberi Digitali di Ricerca

5.1 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. 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 descritta nella Parte II. 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 dei router in Internet.

Un esempio di trie, già incontrato in incognito nella Sezione 1.6, è lo scheletro dell'automa

di Aho-Corasick per la ricerca simultanea di più stringhe (si veda la Figura 1.3, in alto). L'insieme di stringhe ivi memorizzate è dato da $S = \{\texttt{ananas}, \texttt{anacardo}, \texttt{banana}, \texttt{nan}\}$. 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 $c \in \{\texttt{a}, \texttt{b}, \texttt{c}\}$, 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 $c = \texttt{a}$, osserviamo che hanno in comune anche il secondo e il terzo carattere e 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 ␣, 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 $|\Sigma|$ 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. Ai soli fini della definizione ricorsiva del trie, il carattere iniziale c comune alle stringhe in S_c viene ignorato (e temporaneamente rimosso). 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 per $c \in \Sigma$ (alcuni dei figli possono essere vuoti).

Mostriamo nella Figura 5.1, 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 5.1, le foglie del trie sono numerate da 1 a 7, e 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 l allora i suoi figli hanno livello $l + 1$, e per stabilire quale tra i figli scegliere, bisogna guardare al carattere in posizione $l + 1$ delle stringhe.

Il massimo tra i livelli assegnati 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

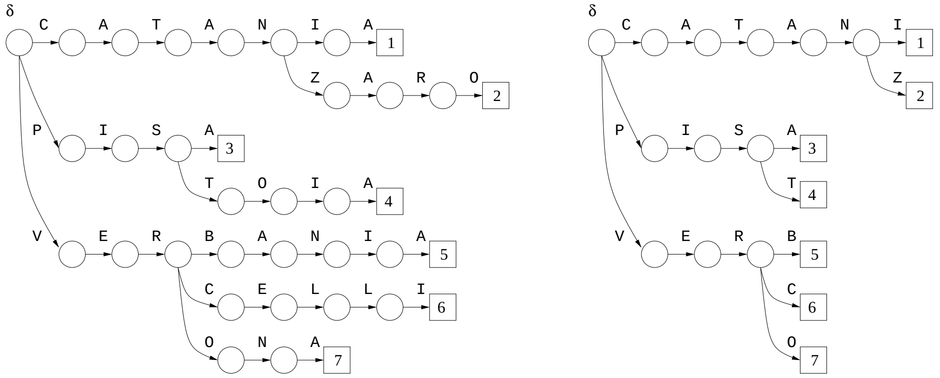


Figura 5.1: A sinistra, il trie per memorizzare i nomi di alcune province; a destra, la versione potata del trie.

portano a una sola foglia, l'altezza media non dipende dalla lunghezza delle stringhe, ma è limitata superiormente da $2\log_{|\Sigma|}|S| + O(1)$, dove $|S|$ 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 5.1, 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 $N+1$. 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 $|S|$ 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 $1.44\,|S|$.

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 u hanno in comune il prefisso associato a u. Nel nostro esempio, le foglie discendenti dal nodo che memorizza il prefisso `PI` hanno associate le stringhe `PISA` e `PISTOIA`.

- 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 k caratteri in comune (come `VERBANIA` e `VERONA` con $k = 3$), condividono un cammino dalla radice al nodo di livello k che corrisponde a quel prefisso.

- 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.

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 $|\Sigma|$ 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 $|\Sigma|$ 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. 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.

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.

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: $u \leftarrow root(\delta)$; $m \leftarrow |P|$;

2: FOR $i \leftarrow 1$ TO m DO
3:   IF $\delta(u, P[i]) =$ NIL THEN
4:     RETURN FALSE;
5:   ELSE
6:     $u \leftarrow \delta(u, P[i])$;
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.

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 $\log_2(26^9 \times 10^7) - 1 \geq 64$ 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 $y = \text{␣}$ per poter associare una foglia a P, nel caso P non sia stata già inserita in precedenza.

INSERZIONE◇TRIE(P, δ):
1: $u \leftarrow root(\delta)$; $m \leftarrow |P|$;
2: FOR $i \leftarrow 1$ TO m DO
3:   IF $\delta(u, P[i]) =$ NIL THEN {*Il prefisso x occorre in u, dove $P = xy$*}
4:     WHILE $i \leq m$ DO {*Crea il trie per y*}
5:       alloca un nuovo nodo v;
6:       $\delta(u, P[i]) \leftarrow v$;
7:       $i \leftarrow i + 1$;
8:       $u \leftarrow v$;
9:     RETURN FALSE;
10:   ELSE
11:     $u \leftarrow \delta(u, P[i])$;
12: IF u è nodo interno THEN {*P occorre come prefisso x, quindi $y = \text{␣}$*}

```
    alloca una nuova foglia v etichettata con P;
    δ(u, ␣) ← v;
    RETURN FALSE;
ELSE
    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 2. 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 $\delta_t(u, c)$, 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 riporato di seguito. Facciamo uso della procedura EPSILON descritta nella Sezione 2.3, e di una procedura ausiliare VISITA⋄FOGLIE(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.

RICERCA⋄ESPRESSIONIREGOLARI⋄TRIE(δ_r, δ_t):

```
Q ← {0};   {Stato iniziale di δr}
u ← root(δt);
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):

```
IF Q contiene lo stato finale di δr THEN
    Visita⋄Foglie(u);
ELSE
    FOREACH carattere c ∈ Σ tale che δt(u, c) ≠ NIL DO
        v ← δt(u, c);   {Figlio non vuoto di u con etichetta c}
        Q' ← {};   {Insieme vuoto}
        FOREACH s ∈ Q DO
            Q' ← Q' ∪ δr(s, c);   {Transizioni dell'automa δr}
        Q' ← Epsilon(Q', δr);
        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 3 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 pat-

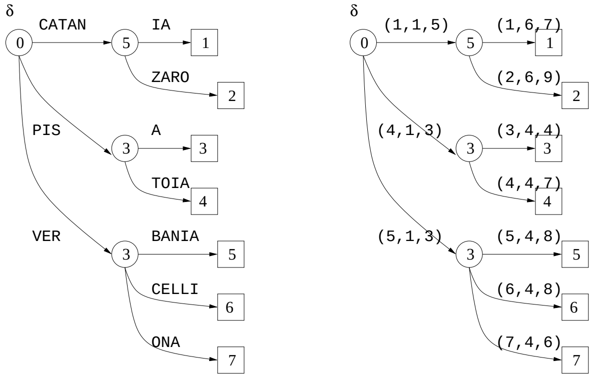


Figura 5.2: A sinistra, il trie compatto per memorizzare i nomi di alcune province; a destra, la versione con le sottostringhe rappresentate mediante triple.

tern 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 `e205140` $\Rightarrow$ `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.

5.2 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 5.2. 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 $2 \leq i \leq k$. 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 5.2 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 5.2 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 p associato a u, entrambi pa e pb sono prefissi delle stringhe in S per almeno due caratteri $a \neq b$ dell'alfabeto (incluso il marcatore $\sqcup$).

Riassumendo, ciascuna foglia nel trie compatto memorizza la stringa ad essa associata. Ciascun nodo interno 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 u a u stesso. Se u è la radice, tale valore è 0.
- Un insieme di $F(u)$ archi etichettati, dove $F(u) \geq 2$ è il numero di figli di u. Ciascun arco etichettato consiste in una stringa rappresentata come (i, j, 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 S di stringhe di lunghezza totale N è $O(|S|)$ invece che $O(N)$ 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 $|S|$ foglie nel trie compatto, la sua dimensione definita come numero totale di nodi è al più $2|S|$.

La ricerca di una stringa P di lunghezza 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 u come scegliere la sottostringa giusta con cui confrontare una parte di P? Nel trie guardiamo i singoli caratteri che etichettano gli archi uscenti da 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 P. In altre parole, se siamo arrivati al nodo u con il carattere $c = P[i]$, scegliamo l'arco uscente da u che ha come primo carattere 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 P a partire dalla posizione $i+1$. Se l'intera sottostringa viene confrontata con successo oppure i caratteri di P sono tutti esaminati, raggiungiamo il figlio di u indicato dall'arco. Altrimenti, possiamo concludere la ricerca con un fallimento. Se alla fine la ricerca ha esito positivo, raggiungiamo un nodo u tale che il prefisso associato a u ha P come prefisso (non è detto che debbano essere uguali; per esempio si prenda $P = \texttt{VERB}$ nel trie compatto di Figura 5.2). Le stringhe in S che hanno prefisso P sono quelle memorizzate nelle foglie discendenti da u. Il costo della ricerca è $O(m)$ se $|\Sigma| = O(1)$; altrimenti è $O(m \log |\Sigma|)$.

Nella ricerca appena descritta, in caso di successo confrontiamo m caratteri del pattern P e $h \leq m+1$ nodi per localizzare u. In certi contesti, pur sapendo che P occorre nel trie compatto, non conosciamo quale è il corrispondente nodo u. Per localizzare il nodo u, possiamo impiegare un numero inferiore di confronti rispetto alla ricerca descritta sopra. In effetti, essendo certi a priori che P occorre, possiamo raggiungere 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 $O(m)$ a $O(h)$. 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 5.1.

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 $P = xy$. Sia u il nodo raggiunto. Se il prefisso associato a u coincide con x, allora creiamo una nuova foglia etichettata con P. Colleghiamo il nodo u a tale foglia con un arco etichettato con y. Tuttavia, se x non coincide, dobbiamo spezzare in due l'arco entrante in u, inserendo un nuovo nodo u' come nuovo padre di u e associando il prefisso x a u'. Procediamo quindi con u' alla stessa stregua indicata sopra per u, ovvero creando una foglia e collegandola a u' con etichetta y. Il costo dell'inserzione è $O(m)$ se $|\Sigma| = O(1)$; altrimenti è $O(m \log |\Sigma|)$.

La cancellazione di P richiede la rimozione della foglia etichettata con P, nonché dell'arco che collega la foglia a suo padre u. Se u diventa unario, allora bisogna rimuovere 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 $O(m)$ se $|\Sigma| = O(1)$; altrimenti è $O(m \log |\Sigma|)$.

Appendice A

Licenza per Documentazione Libera GNU (traduzione)

Preambolo

Lo scopo di questa licenza è di rendere un manuale, un testo o altri documenti scritti "liberi" nel senso di assicurare a tutti la libertà effettiva di copiarli e redistribuirli, con o senza modifiche, a fini di lucro o no. In secondo luogo questa licenza prevede per autori ed editori il modo per ottenere il giusto riconoscimento del proprio lavoro, preservandoli dall'essere considerati responsabili per modifiche apportate da altri.

Questa licenza è un "copyleft": ciò vuol dire che i lavori che derivano dal documento originale devono essere ugualmente liberi. È il complemento alla Licenza Pubblica Generale GNU, che è una licenza di tipo "copyleft" pensata per il software libero.

Abbiamo progettato questa licenza al fine di applicarla alla documentazione del software libero, perché il software libero ha bisogno di documentazione libera: un programma libero dovrebbe accompagnarsi a manuali che forniscano la stessa libertà del software. Ma questa licenza non è limitata alla documentazione del software; può essere utilizzata per ogni testo che tratti un qualsiasi argomento e al di là dell'avvenuta pubblicazione cartacea. Raccomandiamo principalmente questa licenza per opere che abbiano fini didattici o per manuali di consultazione.

A.1 Applicabilità e definizioni

Questa licenza si applica a qualsiasi manuale o altra opera che contenga una nota messa dal detentore del copyright che dica che si può distribuire nei termini di questa licenza. Con "Documento", in seguito ci si riferisce a qualsiasi manuale o opera. Ogni fruitore è un destinatario della licenza e viene indicato con "voi".

Una "versione modificata" di un documento è ogni opera contenente il documento stesso o parte di esso, sia riprodotto alla lettera che con modifiche, oppure traduzioni in un'altra lingua.

Una "sezione secondaria" è un'appendice cui si fa riferimento o una premessa del documento e riguarda esclusivamente il rapporto dell'editore o dell'autore del documento con l'argomento generale del documento stesso (o argomenti affini) e non contiene nulla che possa essere compreso nell'argomento principale. (Per esempio, se il documento è in parte un manuale di matematica, una sezione secondaria non può contenere spiegazioni di matematica). Il rapporto con l'argomento può essere un tema collegato storicamente con il soggetto principale o con soggetti affini, o essere costituito da argomentazioni legali, commerciali, filosofiche, etiche o politiche pertinenti.

Le "sezioni non modificabili" sono alcune sezioni secondarie i cui titoli sono esplicitamente dichiarati essere sezioni non modificabili, nella nota che indica che il documento è realizzato sotto questa licenza.

I "testi copertina" sono dei brevi brani di testo che sono elencati nella nota che indica che il documento è realizzato sotto questa licenza.

Una copia "trasparente" del documento indica una copia leggibile da un calcolatore, codificata in un formato le cui specifiche sono disponibili pubblicamente, i cui contenuti possono essere visti e modificati direttamente, ora e in futuro, con generici editor di testi o (per immagini composte da pixel) con generici editor di immagini o (per i disegni) con qualche editor di disegni ampiamente diffuso, e la copia deve essere adatta al trattamento per la formattazione o per la conversione in una varietà di formati atti alla successiva formattazione. Una copia fatta in un altro formato di file trasparente il cui markup è stato progettato per intralciare o scoraggiare modifiche future da parte dei lettori non è trasparente. Una copia che non è trasparente è "opaca".

Esempi di formati adatti per copie trasparenti sono l'ASCII puro senza markup, il formato di input per Texinfo, il formato di input per LaTex, SGML o XML accoppiati ad una DTD pubblica e disponibile, e semplice HTML conforme agli standard e progettato per essere modificato manualmente. Formati opachi sono PostScript, PDF, formati proprietari che possono essere letti e modificati solo con word processor proprietari, SGML o XML per cui non è in genere disponibile la DTD o gli strumenti per il trattamento, e HTML generato automaticamente da qualche word processor per il solo output.

La "pagina del titolo" di un libro stampato indica la pagina del titolo stessa, più qualche pagina seguente per quanto necessario a contenere in modo leggibile, il materiale che la licenza prevede che compaia nella pagina del titolo. Per opere in formati in cui non sia contemplata esplicitamente la pagina del titolo, con "pagina del titolo" si intende il testo prossimo al titolo dell'opera, precedente l'inizio del corpo del testo.

A.2 Copie alla lettera

Si può copiare e distribuire il documento con l'ausilio di qualsiasi mezzo, per fini di lucro e non, fornendo per tutte le copie questa licenza, le note sul copyright e l'avviso che questa licenza si applica al documento, e che non si aggiungono altre condizioni al di fuori di quelle della licenza stessa. Non si possono usare misure tecniche per impedire o controllare la lettura o la produzione di copie successive alle copie che si producono o distribuiscono. Però si possono ricavare compensi per le copie fornite. Se si distribuiscono un numero sufficiente di copie si

devono seguire anche le condizioni della sezione A.3.

Si possono anche prestare copie e con le stesse condizioni sopra menzionate possono essere utilizzate in pubblico.

A.3 Copiare in notevoli quantità

Se si pubblicano a mezzo stampa più di 100 copie del documento, e la nota della licenza indica che esistono uno o più testi copertina, si devono includere nelle copie, in modo chiaro e leggibile, tutti i testi copertina indicati: il testo della prima di copertina in prima di copertina e il testo di quarta di copertina in quarta di copertina. Ambedue devono identificare l'editore che pubblica il documento. La prima di copertina deve presentare il titolo completo con tutte le parole che lo compongono egualmente visibili ed evidenti. Si può aggiungere altro materiale alle copertine. Il copiare con modifiche limitate alle sole copertine, purché si preservino il titolo e le altre condizioni viste in precedenza, è considerato alla stregua di copiare alla lettera.

Se il testo richiesto per le copertine è troppo voluminoso per essere riprodotto in modo leggibile, se ne può mettere una prima parte per quanto ragionevolmente può stare in copertina, e continuare nelle pagine immediatamente seguenti.

Se si pubblicano o distribuiscono copie opache del documento in numero superiore a 100, si deve anche includere una copia trasparente leggibile da un calcolatore per ogni copia o menzionare per ogni copia opaca un indirizzo di una rete di calcolatori pubblicamente accessibile in cui vi sia una copia trasparente completa del documento, spogliato di materiale aggiuntivo, e a cui si possa accedere anonimamente e gratuitamente per scaricare il documento usando i protocolli standard e pubblici generalmente usati. Se si adotta l'ultima opzione, si deve prestare la giusta attenzione, nel momento in cui si inizia la distribuzione in quantità elevata di copie opache, ad assicurarsi che la copia trasparente rimanga accessibile all'indirizzo stabilito fino ad almeno un anno di distanza dall'ultima distribuzione (direttamente o attraverso rivenditori) di quell'edizione al pubblico.

È caldamente consigliato, benché non obbligatorio, contattare l'autore del documento prima di distribuirne un numero considerevole di copie, per metterlo in grado di fornire una versione aggiornata del documento.

A.4 Modifiche

Si possono copiare e distribuire versioni modificate del documento rispettando le condizioni delle precedenti sezioni A.2 e A.3, purché la versione modificata sia realizzata seguendo scrupolosamente questa stessa licenza, con la versione modificata che svolga il ruolo del "documento", così da estendere la licenza sulla distribuzione e la modifica a chiunque ne possieda una copia. Inoltre nelle versioni modificate si deve:

1. Usare nella pagina del titolo (e nelle copertine se ce ne sono) un titolo diverso da quello del documento, e da quelli di versioni precedenti (che devono essere elencati nella sezione storia del documento ove presenti). Si può usare lo stesso titolo di una versione precedente se l'editore di quella versione originale ne ha dato il permesso.

2. Elencare nella pagina del titolo, come autori, una o più persone o gruppi responsabili in

qualità di autori delle modifiche nella versione modificata, insieme ad almeno cinque fra i principali autori del documento (tutti gli autori principali se sono meno di cinque).

3. Dichiarare nella pagina del titolo il nome dell'editore della versione modificata in qualità di editore.
4. Conservare tutte le note sul copyright del documento originale.
5. Aggiungere un'appropriata licenza per le modifiche di seguito alle altre licenze sui copyright.
6. Includere immediatamente dopo la nota di copyright, un avviso di licenza che dia pubblicamente il permesso di usare la versione modificata nei termini di questa licenza, nella forma mostrata nell'addendum alla fine di questo testo.
7. Preservare in questo avviso di licenza l'intera lista di sezioni non modificabili e testi copertina richieste come previsto dalla licenza del documento.
8. Includere una copia non modificata di questa licenza.
9. Conservare la sezione intitolata "Storia", e il suo titolo, e aggiungere a questa un elemento che riporti al minimo il titolo, l'anno, i nuovi autori, e gli editori della versione modificata come figurano nella pagina del titolo. Se non ci sono sezioni intitolate "Storia" nel documento, createne una che riporti il titolo, gli autori, gli editori del documento come figurano nella pagina del titolo, quindi aggiungete un elemento che descriva la versione modificata come detto in precedenza.
10. Conservare l'indirizzo in rete riportato nel documento, se c'è, al fine del pubblico accesso ad una copia trasparente, e possibilmente l'indirizzo in rete per le precedenti versioni su cui ci si è basati. Questi possono essere collocati nella sezione "Storia". Si può omettere un indirizzo di rete per un'opera pubblicata almeno quattro anni prima del documento stesso, o se l'originario editore della versione cui ci si riferisce ne dà il permesso.
11. In ogni sezione di "Ringraziamenti" o "Dediche", si conservino il titolo, il senso, il tono della sezione stessa.
12. Si conservino inalterate le sezioni non modificabili del documento, nei propri testi e nei propri titoli. I numeri della sezione o equivalenti non sono considerati parte del titolo della sezione.
13. Si cancelli ogni sezione intitolata "Riconoscimenti". Solo questa sezione può non essere inclusa nella versione modificata.
14. Non si modifichi il titolo di sezioni esistenti come "miglioria" o per creare confusione con i titoli di sezioni non modificabili.

Se la versione modificata comprende nuove sezioni di primaria importanza o appendici che ricadono in "sezioni secondarie", e non contengono materiale copiato dal documento, si ha facoltà di rendere non modificabili quante sezioni si voglia. Per fare ciò si aggiunga il loro titolo alla lista delle sezioni immutabili nella nota di copyright della versione modificata. Questi titoli devono essere diversi dai titoli di ogni altra sezione.

Si può aggiungere una sezione intitolata "Riconoscimenti", a patto che non contenga altro che le approvazioni alla versione modificata prodotte da vari soggetti–per esempio, affermazioni di revisione o che il testo è stato approvato da una organizzazione come la definizione normativa di uno standard.

Si può aggiungere un brano fino a cinque parole come Testo Copertina, e un brano fino a 25 parole come Testo di Retro Copertina, alla fine dell'elenco dei Testi Copertina nella versione modificata. Solamente un brano del Testo Copertina e uno del Testo di Retro Copertina possono essere aggiunti (anche con adattamenti) da ciascuna persona o organizzazione. Se il documento include già un testo copertina per la stessa copertina, precedentemente aggiunto o adattato da voi o dalla stessa organizzazione nel nome della quale si agisce, non se ne può aggiungere un altro, ma si può rimpiazzare il vecchio ottenendo l'esplicita autorizzazione dall'editore precedente che aveva aggiunto il testo copertina.

L'autore/i e l'editore/i del "documento" non ottengono da questa licenza il permesso di usare i propri nomi per pubblicizzare la versione modificata o rivendicare l'approvazione di ogni versione modificata.

A.5 Unione di documenti

Si può unire il documento con altri realizzati sotto questa licenza, seguendo i termini definiti nella precedente sezione A.4 per le versioni modificate, a patto che si includa l'insieme di tutte le Sezioni Invarianti di tutti i documenti originali, senza modifiche, e si elenchino tutte come Sezioni Invarianti della sintesi di documenti nella licenza della stessa.

Nella sintesi è necessaria una sola copia di questa licenza, e multiple sezioni invarianti possono essere rimpiazzate da una singola copia se identiche. Se ci sono multiple Sezioni Invarianti con lo stesso nome ma contenuti differenti, si renda unico il titolo di ciascuna sezione aggiungendovi alla fine e fra parentesi, il nome dell'autore o editore della sezione, se noti, o altrimenti un numero distintivo. Si facciano gli stessi aggiustamenti ai titoli delle sezioni nell'elenco delle Sezioni Invarianti nella nota di copiright della sintesi.

Nella sintesi si devono unire le varie sezioni intitolate "storia" nei vari documenti originali di partenza per formare una unica sezione intitolata "storia"; allo stesso modo si unisca ogni sezione intitolata "Ringraziamenti", e ogni sezione intitolata "Dediche". Si devono eliminare tutte le sezioni intitolate "Riconoscimenti".

A.6 Raccolte di documenti

Si può produrre una raccolta che consista del documento e di altri realizzati sotto questa licenza; e rimpiazzare le singole copie di questa licenza nei vari documenti con una sola inclusa nella raccolta, solamente se si seguono le regole fissate da questa licenza per le copie alla lettera come se si applicassero a ciascun documento.

Si può estrarre un singolo documento da una raccolta e distribuirlo individualmente sotto questa licenza, solo se si inserisce una copia di questa licenza nel documento estratto e se si seguono tutte le altre regole fissate da questa licenza per le copie alla lettera del documento.

A.7 Raccogliere insieme a lavori indipendenti

Una raccolta del documento o sue derivazioni con altri documenti o lavori separati o indipendenti, all'interno di o a formare un archivio o un supporto per la distribuzione, non è una "versione modificata" del documento nella sua interezza, se non ci sono copiright per l'intera raccolta. Ciascuna raccolta si chiama allora "aggregato" e questa licenza non si applica agli altri lavori contenuti in essa che ne sono parte, per il solo fatto di essere raccolti insieme, qualora non siano però loro stessi lavori derivati dal documento.

Se le esigenze del Testo Copertina della sezione A.3 sono applicabili a queste copie del documento allora, se il documento è inferiore ad un quarto dell'intero aggregato i Testi Copertina del documento possono essere piazzati in copertine che delimitano solo il documento all'interno dell'aggregato. Altrimenti devono apparire nella copertina dell'intero aggregato.

A.8 Traduzione

La traduzione è considerata un tipo di modifica, e di conseguenza si possono distribuire traduzioni del documento seguendo i termini della sezione A.4. Rimpiazzare sezioni non modificabili con traduzioni richiede un particolare permesso da parte dei detentori del diritto d'autore, ma si possono includere traduzioni di una o più sezioni non modificabili in aggiunta alle versioni originali di queste sezioni immutabili. Si può fornire una traduzione della presente licenza a patto che si includa anche l'originale versione inglese di questa licenza. In caso di discordanza fra la traduzione e l'originale inglese di questa licenza la versione originale inglese prevale sempre.

A.9 Termini

Non si può applicare un'altra licenza al documento, copiarlo, modificarlo, o distribuirlo al di fuori dei termini espressamente previsti da questa licenza. Ogni altro tentativo di applicare un'altra licenza al documento, copiarlo, modificarlo, o distribuirlo è deprecato e pone fine automaticamente ai diritti previsti da questa licenza. Comunque, per quanti abbiano ricevuto copie o abbiano diritti coperti da questa licenza, essi non ne cessano se si rimane perfettamente coerenti con quanto previsto dalla stessa.

A.10 Revisioni future di questa licenza

La Free Software Foundation può pubblicare nuove, rivedute versioni della Licenza per Documentazione Libera GNU volta per volta. Qualche nuova versione potrebbe essere simile nello spirito alla versione attuale ma differire in dettagli per affrontare nuovi problemi e concetti. Si veda `http://www.gnu.org/copyleft/`.

Ad ogni versione della licenza viene dato un numero che distingue la versione stessa. Se il documento specifica che si riferisce ad una versione particolare della licenza contraddistinta dal numero o "ogni versione successiva", si ha la possibilità di seguire termini e condizioni sia della versione specificata che di ogni versione successiva pubblicata (non come bozza) dalla Free Software Foundation. Se il documento non specifica un numero di versione particolare di

questa licenza, si può scegliere ogni versione pubblicata (non come bozza) dalla Free Software Foundation.