In questo capitolo vedremo alcuni degli algoritmi fondamentali utilizzabili per analizzare ed elaborare in modo efficiente grandi insiemi di dati di tipo testuale. Oltre ad avere applicazioni interessanti, gli algoritmi di elaborazione di testi sono anche adatti a illustrare alcuni importanti schemi progettuali per algoritmi.

Iniziamo prendendo in esame il problema della ricerca di una stringa (detta *pattern*, cioè "schema", "esempio", "modello" o "campione") come sottostringa di una porzione di testo di dimensioni maggiori, come, ad esempio, la ricerca di una parola in un documento. Questo problema, chiamato del *pattern matching* (cioè della ricerca di una corrispondenza per un campione di stringa in un testo), può essere risolto con il cosiddetto *metodo della forza bruta* (*brute-force method*), che risulta spesso inefficiente ma ha un campo di applicabilità generale. Continueremo descrivendo algoritmi più efficienti per risolvere il problema del *pattern matching*, esaminando anche alcune strutture dati aventi l'obiettivo specifico di organizzare in modo migliore i dati di tipo testo, allo scopo di consentire lo sviluppo di ricerche più efficienti.

A causa delle enormi dimensioni che spesso caratterizzano gli insiemi di dati di questo tipo, assume grande importanza il problema della compressione dei dati, tanto per minimizzare il numero di bit richiesto per trasferire i dati attraverso la rete quanto per ridurre l'occupazione di spazio a lungo termine all'interno degli archivi. Per la compressione dei testi, possiamo progettare algoritmi usando il *metodo greedy* ("goloso"), che spesso ci consente di trovare soluzioni approssimate a problemi difficili, e per alcuni problemi (come la compressione del testo) è anche in grado di generare algoritmi ottimali.

Infine, parleremo di *programmazione dinamica*, una tecnica algoritmica applicabile in determinate situazioni per risolvere un problema in un tempo polinomiale, mentre a prima vista sembrerebbe richiedere un tempo d'esecuzione esponenziale. Presenteremo un'applicazione di questa tecnica al problema di individuare corrispondenze parziali tra stringhe che possono essere simili ma non perfettamente allineate. Questo problema emerge quando, durante la scrittura di un documento, si cercano suggerimenti per una parola scritta in modo non corretto, oppure quando si cercano somiglianze tra campioni di materiale genetico.

13.1.1 Notazioni per stringhe di caratteri

Parlando di algoritmi per l'elaborazione di testi, si usano stringhe di caratteri per rappresentare, appunto, un modello per un testo. Le stringhe di caratteri possono provenire da una grande varietà di sorgenti di informazione, comprese diverse applicazioni scientifiche, linguistiche o per Internet. Ecco alcuni esempi di stringhe:

S = "CGTAAACTGCTTTAATCAAACGC"
T = "http://www.wiley.com"

La prima stringa, S, proviene da applicazioni di elaborazione del DNA, mentre la seconda, T, è l'indirizzo Internet (URL, *uniform resource locator*) dell'editore della versione originale di questo libro.

Per consentire nelle descrizioni dei nostri algoritmi l'utilizzo di un concetto di stringa sufficientemente generale, ipotizziamo soltanto che i caratteri appartenenti a una stringa provengano da un *alfabeto* noto, che indichiamo con Σ. Ad esempio, nel contesto di applicazioni che riguardano il DNA, l'alfabeto standard è costituito da quattro simboli, Σ =

$\{A, C, G, T\}$. L'alfabeto Σ può, ovviamente, essere un sottoinsieme dell'insieme di caratteri ASCII o Unicode, ma potrebbe anche essere più generale. Anche se ipotizziamo che un alfabeto abbia una dimensione finita e prefissata, indicata con $|\Sigma|$, tale dimensione può essere rilevante, come nel caso dell'utilizzo, nel linguaggio di programmazione Java, dell'alfabeto Unicode, che contiene più di un milione di caratteri diversi. Nell'analisi asintotica degli algoritmi di elaborazione di testi, quindi, prenderemo in considerazione l'influenza di $|\Sigma|$.

Come detto nel Paragrafo 1.3, la classe `String` di Java fornisce supporto alla rappresentazione di una sequenza ***immutabile*** di caratteri, mentre la classe `StringBuilder` viene usata per rappresentare sequenze di caratteri ***modificabili***. Nella maggior parte di questo capitolo, ci baseremo su una rappresentazione più primitiva delle stringhe come array di `char`, principalmente perché questo consente l'utilizzo della notazione standard mediante indice, come $S[i]$, invece della sintassi decisamente più ingombrante che sarebbe richiesta dalla classe `String`, con espressioni come `S.charAt(i)`.

Per indicare porzioni di stringa, introduciamo il concetto di ***sottostringa*** (***substring***) di una stringa P avente n caratteri come di una stringa descritta nella forma $P[i]P[i+1]P[i+2]\ldots P[j]$, con $0 \leq i \leq j \leq n-1$. Per semplificare la notazione, indicheremo con $P[i..j]$ la sottostringa di P che va dall'indice i all'indice j, entrambi inclusi. Osserviamo che una stringa è, dal punto di vista tecnico, una sottostringa di se stessa (prendendo $i = 0$ e $j = n-1$), quindi, quando vogliamo escludere questa possibilità, dobbiamo restringere la definizione alle sottostringhe ***proprie***, nelle quali $i > 0$ oppure $j < n-1$. Per convenzione, poi, ammettiamo che, quando $i > j$, la sottostringa $P[i..j]$ sia uguale alla ***stringa nulla***, che ha lunghezza 0.

Inoltre, per introdurre alcune categorie di stringhe speciali, quando una sottostringa ha la forma $P[0..j]$, con $0 \leq j \leq n-1$, la chiamiamo ***prefisso*** di P, mentre una sottostringa come $P[i..n-1]$, con $0 \leq i \leq n-1$, è un ***suffisso*** di P. Ad esempio, se consideriamo di nuovo che P sia la stringa di DNA presentata all'inizio del paragrafo, allora `"CGTAA"` è un prefisso di P, `"CGC"` è un suffisso di P e `"TTAATC"` è una sottostringa (propria) di P. Osserviamo che la stringa nulla è sia un prefisso sia un suffisso di qualsiasi altra stringa.

13.2 Algoritmi di *pattern matching*

Nel problema del pattern matching classico, ci viene fornita una stringa di ***testo*** di lunghezza n e una stringa, che svolge il ruolo di ***pattern*** da cercare, di lunghezza $m \leq n$: dobbiamo determinare se il ***pattern*** sia una sottostringa del ***testo***. Se lo è, possiamo voler trovare il minimo indice all'interno del testo che rappresenta l'inizio di una occorrenza del pattern, oppure ***tutti*** gli indici, nel testo, in cui inizia un'occorrenza del pattern.

Il problema del pattern matching è inerente a molti dei comportamenti della classe `String`, come `text.contains(pattern)` e `text.indexOf(pattern)`, ed è anche un sotto-problema di operazioni più complesse di elaborazione di stringhe, come `text.replace(pattern, substitute)` e `text.split(pattern)`.

In questo paragrafo presenteremo tre algoritmi di pattern matching, con livello di difficoltà crescente. Le nostre implementazioni restituiscono l'indice in cui inizia l'occorrenza del pattern più a sinistra nel testo, se ce n'è almeno una. In caso di ricerca infruttuosa, adottiamo la convenzione prevista dal metodo `indexOf` della classe `String` di Java, restituendo -1 come valore sentinella.

13.2.1 Forza bruta

Lo schema che consente di progettare algoritmi mediante *forza bruta* è una potente tecnica di progettazione di algoritmi utilizzabile quando vogliamo cercare qualcosa o quando vogliamo ottimizzare una determinata funzione. In generale, applicando questa tecnica tipicamente elenchiamo tutte le possibili configurazioni dei dati da elaborare e, tra queste, scegliamo la migliore di tutte.

Applicando questa tecnica alla progettazione di un algoritmo di pattern matching, otteniamo quello che probabilmente è il primo algoritmo che ci viene in mente per risolvere il problema: semplicemente, verifichiamo tutti i possibili posizionamenti del pattern all'interno del testo. Il Codice 13.1 mostra un'implementazione di questo algoritmo.

Codice 13.1: Un'implementazione dell'algoritmo di pattern matching mediante forza bruta (per semplificare la notazione, usiamo array di caratteri invece di stringhe).

```
1  /** Restituisce l'indice minimo in cui inizia pattern nel testo (oppure -1). */
2  public static int findBrute(char[] text, char[] pattern) {
3    int n = text.length;
4    int m = pattern.length;
5    for (int i=0; i <= n - m; i++) { // prova tutti gli indici iniziali nel testo
6      int k = 0;                     // k è un indice nel pattern
7      while (k < m && text[i+k] == pattern[k]) // il k-esimo carattere corrisponde
8        k++;
9      if (k == m)              // se abbiamo raggiunto la fine del pattern
10       return i;              // la sottostringa text[i..i+m-1] corrisponde al pattern
11   }
12   return -1;                 // ricerca fallita
13 }
```

Prestazioni

L'analisi dell'algoritmo di pattern matching mediante forza bruta non potrebbe essere più semplice. È costituito da due cicli annidati, con il ciclo esterno che scandisce tutti i possibili indici iniziali del pattern nel testo e il ciclo interno che analizza ciascun carattere del pattern, confrontandolo con il carattere potenzialmente corrispondente nel testo. Quindi, la correttezza di questo algoritmo discende immediatamente da questo approccio di ricerca esaustivo.

Tuttavia, nel caso peggiore, il tempo d'esecuzione dell'algoritmo di pattern matching mediante forza bruta non è buono, perché per ogni verifica di un possibile allineamento del pattern nel testo posso essere necessari fino a m confronti tra coppie di caratteri. Facendo riferimento al Codice 13.1, vediamo che il ciclo `for`, quello esterno, viene eseguito al massimo $n - m + 1$ volte, mentre il ciclo `while`, quello interno, viene eseguito al massimo m volte. Quindi, il tempo d'esecuzione di questo algoritmo, nel caso peggiore, è $O(nm)$.

Esempio 13.1: *Supponiamo che, in un problema di pattern matching, il testo sia la stringa seguente:*

```
text = "abacaabaccabacabaabb"
```

e il pattern sia:

```
pattern = "abacab"
```

La Figura 13.1 mostra l'esecuzione dell'algoritmo di pattern matching mediante forza bruta per questo problema.

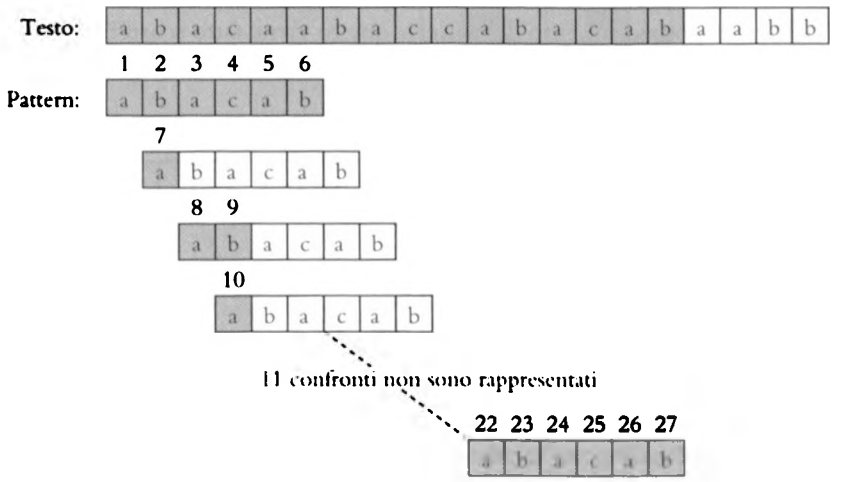


Figura 13.1: Esempio di esecuzione dell'algoritmo di pattern matching mediante forza bruta. L'algoritmo effettua 27 confronti tra caratteri, indicati con etichette numerate.

13.2.2 L'algoritmo di Boyer-Moore

A prima vista si potrebbe pensare che, per poter individuare il pattern come sottostringa di un testo o per decidere sulla sua assenza, sia sempre necessario esaminare tutti i caratteri del testo, ma non è così. L'algoritmo di *Boyer-Moore* per il pattern matching, che studieremo in questo paragrafo in una versione semplificata, a volte può evitare l'esame di una frazione rilevante dei caratteri del testo.

L'idea principale su cui si basa l'algoritmo di Boyer-Moore è quella di migliorare il tempo d'esecuzione dell'algoritmo a forza bruta aggiungendo due strategie euristiche che possono far risparmiare tempo. In sintesi, si tratta di questo:

Euristica Looking-Glass: (euristica che "guarda allo specchio") Quando si verifica un possibile posizionamento del pattern nel testo, i confronti tra le coppie di caratteri vengono effettuati procedendo da destra a sinistra.

Euristica Character-Jump: (euristica che "salta dei caratteri") Durante la verifica di un possibile posizionamento del pattern nel testo, una differenza tra il carattere `text[i]` = *c* e il corrispondente carattere `pattern[k]` viene gestita in questo modo: se *c* non è presente in alcuna posizione del pattern, si fa scorrere il pattern completamente a destra della posizione `text[i]`; altrimenti, si fa scorrere il pattern verso destra finché l'occorrenza di *c* che si trova più a destra nel pattern risulta essere allineata con `text[i]`.

A breve definiremo in modo più formale queste strategie euristiche, ma, a livello intuitivo, collaborano tra loro, come in una squadra, per consentirci di evitare confronti con intere sequenze di caratteri all'interno del testo. In particolare, quando si trova una mancata

corrispondenza tra caratteri in una posizione vicina alla fine del pattern, possiamo trovarci a riallineare il pattern oltre tale posizione, senza dove mai esaminare i molti caratteri del testo che precedono la posizione in questione. Ad esempio, la Figura 13.2 mostra alcune applicazioni di queste euristiche. Si osservi che, quando i caratteri e e i non corrispondono, alla fine del primo posizionamento del pattern, questo viene fatto scorrere completamente oltre la posizione di mancata corrispondenza, evitando così l'esame dei primi quattro caratteri del testo.

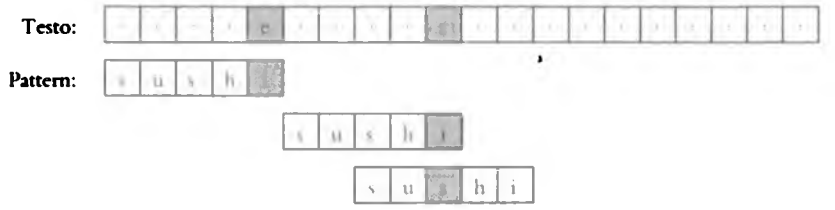


Figura 13.2: Un semplice esempio che illustra l'intuizione su cui si basa l'algoritmo di pattern matching di Boyer-Moore. Il primo confronto evidenzia una mancata corrispondenza con il carattere *e* del testo. Dato che quel carattere non è presente in nessuna posizione del pattern, l'intero pattern viene fatto scorrere oltre quella posizione. Il secondo confronto è ancora una mancata corrispondenza, ma il carattere del testo che non ha trovato corrispondenza è *s*, che compare nel pattern. Quindi, il pattern viene fatto scorrere in modo che l'ultima occorrenza di *s* nel pattern si trovi allineata con la *s* che si stava esaminando nel testo. La parte restante della procedura non è rappresentata nella figura.

L'esempio della Figura 13.2 è abbastanza semplice, perché riguarda solamente corrispondenze mancate con l'ultimo carattere del pattern. Più in generale, quando l'ultimo carattere del pattern trova invece corrispondenza nel testo, l'algoritmo procede cercando di estendere la porzione identica, passando al penultimo carattere del pattern senza cambiare il suo posizionamento rispetto al testo. Il processo continua finché si trova nel testo una corrispondenza per l'intero pattern oppure si trova una coppia di caratteri diversi in qualche posizione del pattern diversa dall'ultima.

Se si trova una mancata corrispondenza tra una coppia di caratteri e il carattere preso in esame nel testo non ricorre in alcuna posizione del pattern, facciamo scorrere l'intero pattern a destra, oltre tale posizione esaminata, come avviene nel primo caso della Figura 13.2. Se, invece, il carattere preso in esame nel testo è presente in qualche posizione del pattern, dobbiamo considerare due possibili sotto-casi, in relazione al fatto che la sua occorrenza più a destra (cioè, come si dice, l'*ultima* occorrenza) si trovi a sinistra o a destra del carattere che, nel pattern, è stato esaminato per ultimo e ha provocato la mancata corrispondenza. Questi due casi sono proprio illustrati nella Figura 13.3.

Nel caso della Figura 13.3(b), facciamo scorrere il pattern di una sola posizione. Sarebbe più efficace farlo scorrere verso destra fino a quando non si trova nel pattern un'altra occorrenza del carattere `text[`i`]`, ma non vogliamo cercare tale altra occorrenza. L'efficienza dell'algoritmo di Boyer-Moore sta nel fatto che si possa determinare velocemente se un carattere che ha provocato una mancata corrispondenza è presente da qualche parte nel pattern. In particolare, definiamo la funzione `last(`c`)` in questo modo:

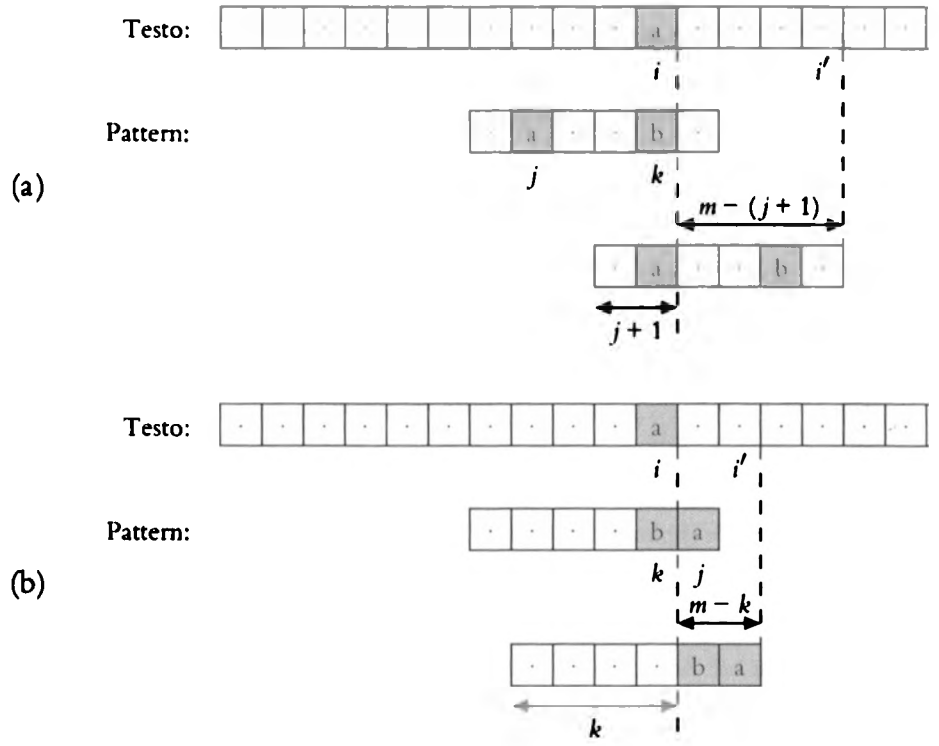


Figura 13.3: Ulteriori regole per l'euristica *character-jump* dell'algoritmo Boyer-Moore. Indichiamo con i l'indice, nel testo, del carattere che ha provocato la mancata corrispondenza, mentre k è l'indice corrispondente nel pattern e j rappresenta l'indice della posizione più a destra di `text[i]` nel pattern. Distinguiamo due casi: (a) $j < k$, nel qual caso facciamo scorrere il pattern di $k - j$ posizioni e, quindi, l'indice i aumenta di $m - (j + 1)$ unità; (b) $j > k$, nel qual caso facciamo scorrere il pattern di una sola posizione e, quindi, l'indice i aumenta di $m - k$ unità.

- Se c appartiene al pattern, `last`(c) è l'indice dell'occorrenza più a destra di c nel pattern (cioè l'*ultima*, appunto "last"); altrimenti, definiamo per convenzione `last`(c) = −1.

Ipotizzando che l'alfabeto abbia dimensione finita e prefissata e che i caratteri possano essere usati come indici in un array (ad esempio, usando il codice corrispondente al carattere), si può facilmente implementare la funzione `last` come una tabella di ricerca, con un tempo d'accesso al valore `last`(c) che, nel caso peggiore, è $O(1)$. La tabella, però, avrebbe una dimensione uguale alla dimensione dell'alfabeto (non correlata alla dimensione del pattern) e ci vorrebbe tempo per inizializzare l'intera tabella.

Preferiamo, quindi, rappresentare la funzione `last` con una tabella hash, che realizza una mappa contenente soltanto i caratteri effettivamente presenti nel pattern. Lo spazio di memoria utilizzato da questo approccio è proporzionale al numero di simboli dell'alfabeto distinti che compaiono nel pattern, quindi è $O(\max(m, |\Sigma|))$. Il tempo atteso per la ricerca rimane $O(1)$, così come il suo caso peggiore, se consideriamo che $|\Sigma|$ sia un valore costante. Il Codice 13.2 presenta la nostra implementazione completa dell'algoritmo di pattern matching di Boyer-Moore.

Codice 13.2: Un'implementazione dell'algoritmo di Boyer-Moore.

```
/** Restituisce l'indice minimo in cui inizia pattern nel testo (oppure -1). */
public static int findBoyerMoore(char[] text, char[] pattern) {
  int n = text.length;
  int m = pattern.length;
  if (m == 0) return 0;       // ricerca banale di una stringa vuota
  Map<Character,Integer> last = new HashMap<>(); // la mappa della funzione 'last'
  for (int i=0; i < n; i++)
    last.put(text[i], -1);    // usa -1 come default per tutti i caratteri del testo
  for (int k=0; k < m; k++)
    last.put(pattern[k], k); // l'occorrenza più a destra sarà l'ultima memorizzata
  // inizia con la fine del pattern allineata con l'indice m-1 del testo
  int i = m-1;                // indice nel testo
  int k = m-1;                // indice nel pattern
  while (i < n) {
    if (text[i] == pattern[k]) { // trovato un carattere che corrisponde
      if (k == 0) return i;      // trovato un intero pattern che corrisponde
      i--;                       // altrimenti, esamina il carattere precedente
      k--;                       // tanto nel testo quanto nel pattern
    } else {
      i += m - Math.min(k, 1 + last.get(text[i])); // decide come fare il salto
      k = m - 1;                 // riparte dalla fine del pattern
    }
  }
  return -1;                  // ricerca fallita
}
```

La correttezza dell'algoritmo di pattern matching di Boyer-Moore discende dal fatto che, ogni volta che il metodo fa scorrere il pattern in avanti, si ha la garanzia di non "saltare" la posizione di una possibile corrispondenza completa tra il pattern e il testo, cioè, come anche si dice, un *match*, perché la funzione last(c) indica la posizione dell'***ultima*** occorrenza di c nel pattern. Nella Figura 13.4 illustriamo l'esecuzione dell'algoritmo di pattern matching di Boyer-Moore applicato a una stringa simile a quella dell'Esempio 13.1.

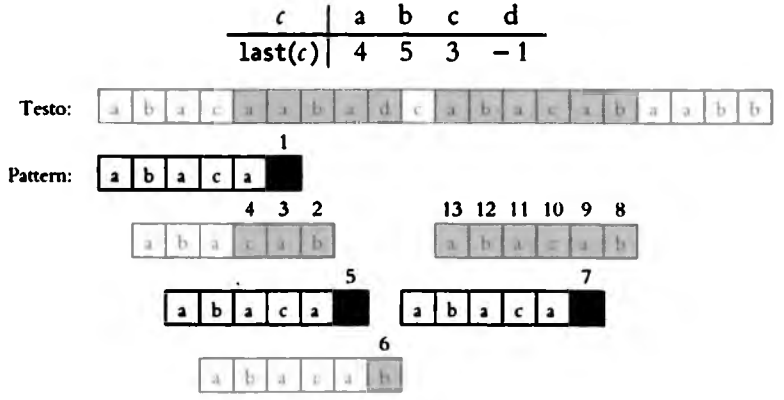


Figura 13.4: Rappresentazione grafica dell'esecuzione dell'algoritmo di pattern matching di Boyer-Moore; è riportata anche la funzione last(c). L'algoritmo effettua 13 confronti tra coppie di caratteri, indicati con etichette numerate.

Prestazioni

Se si usa una tabella di ricerca tradizionale, il caso peggiore del tempo d'esecuzione dell'algoritmo di Boyer-Moore è $O(nm + |\Sigma|)$. Il calcolo della funzione last richiede un tempo $O(m + |\Sigma|)$, sebbene la dipendenza da $|\Sigma|$ possa essere rimossa usando una tabella hash. La ricerca di un pattern richiede un tempo $O(nm)$ nel caso peggiore, esattamente come avviene con l'algoritmo a forza bruta. Un esempio che costituisce un caso peggiore per Boyer-Moore è questo:

$$\text{testo} = \overbrace{aaaaaa\cdots a}^{n}$$

$$\text{pattern} = \overbrace{baa\cdots a}^{m-1}$$

La situazione del caso peggiore, però, accade molto raramente se si fanno ricerche in un testo in lingua inglese: in quel caso, l'algoritmo di Boyer-Moore è spesso in grado di saltare porzioni di testo di grandi dimensioni. Esperimenti condotti su testi in lingua inglese evidenziano come il numero medio di confronti effettuati per ciascun carattere del testo sia 0.24 nel caso di un pattern di lunghezza cinque.

Abbiamo presentato una versione semplificata dell'algoritmo di Boyer-Moore. L'algoritmo originale è in grado di ottenere un tempo d'esecuzione che, nel caso peggiore, è $O(n + m + |\Sigma|)$, usando una strategia di scorrimento euristica alternativa per le porzioni di testo che costituiscono una corrispondenza parziale con il pattern, con efficacia maggiore dell'euristica *character-jump* illustrata qui. Questa strategia alternativa è basata sull'applicazione della stessa idea fondamentale che caratterizza l'algoritmo di pattern matching dovuto a Knuth, Morris e Pratt, di cui parleremo ora.

13.2.3 L'algoritmo di Knuth-Morris-Pratt

Analizzando le prestazioni nel caso peggiore degli algoritmi di pattern matching di Boyer-Moore e a forza bruta applicati a esemplari specifici del problema, come quello dell'Esempio 13.1, dovrebbe risultare evidente una grande inefficienza (almeno nel caso peggiore): per alcuni posizionamenti del pattern, anche se riscontriamo parecchi caratteri corretti prima di trovare un carattere non corrispondente, ignoriamo poi tutte le informazioni ottenute dai confronti andati a buon fine e ripartiamo semplicemente con il posizionamento successivo del pattern.

L'algoritmo KMP (dalle iniziali degli inventori, Knuth-Morris-Pratt), discusso in questo paragrafo, evita questo spreco di informazioni e, così facendo, riesce a raggiungere prestazioni $O(n + m)$ nel caso peggiore, che sono asintoticamente ottime, perché, nel caso peggiore, qualunque algoritmo di pattern matching dovrà esaminare almeno una volta tutti i caratteri del testo e tutti i caratteri del pattern. L'idea principale su cui si basa l'algoritmo KMP è quella di pre-calcolare le auto-sovrapposizioni tra porzioni del pattern in modo che, quando in una determinata posizione si riscontra una mancata corrispondenza, si possa sapere immediatamente lo spostamento massimo verso destra a cui si può sottoporre il pattern prima di continuare la ricerca. La Figura 13.5 mostra un esempio che motiva questo approccio.

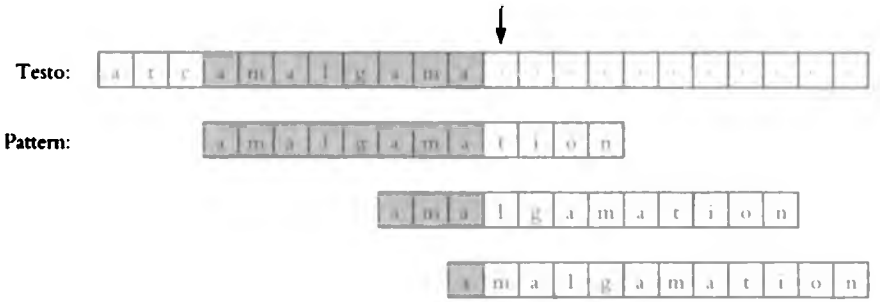


Figura 13.5: Un esempio che motiva la strategia utilizzata dall'algoritmo KMP. Se nella posizione indicata dalla freccia si riscontra una mancata corrispondenza tra il carattere del pattern e quello del testo, si può far scorrere il pattern fino a raggiungere la seconda configurazione, senza che ci sia bisogno di verificare di nuovo in modo esplicito la corrispondenza parziale che era stata individuata nel prefisso `ama`. Se il carattere del testo posto in corrispondenza della freccia (che aveva causato il primo errore di matching) non è un carattere `l`, allora il prossimo posizionamento del pattern, alla ricerca di un potenziale allineamento, può trarre vantaggio dal carattere a condiviso tra la parte iniziale del pattern e il testo.

La *funzione fallimento*

Per implementare l'algoritmo KMP, dovremo pre-calcolare la cosiddetta *funzione fallimento* (*failure function*), f, che fornisce il valore corretto per lo scorrimento verso destra del pattern in caso di fallimento di un confronto. Nello specifico, la funzione fallimento $f(k)$ è definita come la dimensione del più lungo prefisso del pattern che è anche suffisso della sottostringa `pattern[1..k]` (si osservi che *non* abbiamo incluso il carattere `pattern[0]` nella sottostringa, perché faremo certamente scorrere il pattern almeno di una posizione). Intuitivamente, se riscontriamo un confronto fallito per il carattere `pattern[k+1]`, la funzione $f(k)$ ci dice quanti dei caratteri consecutivi immediatamente precedenti a `pattern[k+1]`possono essere riutilizzati come "validi" per far ripartire il confronto da una posizione più avanzata della prima. L'Esempio 13.2 descrive il valore della funzione fallimento per il pattern usato nell'esempio della Figura 13.5.

Esempio 13.2: *Consideriamo il pattern* P = `"amalgamation"` *della Figura 13.5. La funzione fallimento dell'algoritmo KMP,* $f(k)$*, per la stringa* P *è quella presentata nella tabella seguente:*

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	l	g	a	m	a	t	i	o	n
$f(k)$	0	0	1	0	0	1	2	3	0	0	0	0

Implementazione

Il Codice 13.3 presenta la nostra implementazione dell'algoritmo di pattern matching KMP: si basa su un metodo ausiliario, `computeFailKMP`, che calcola in modo efficiente la funzione fallimento, come vedremo in seguito.

La parte principale dell'algoritmo KMP è il suo ciclo `while`, ciascuna iterazione del quale esegue un confronto tra il carattere che si trova nel testo all'indice j e il carattere del pattern all'indice k. Se il risultato di questo confronto è una corrispondenza, l'algoritmo si sposta

al carattere successivo tanto nel testo quanto nel pattern (o restituisce una segnalazione di *match* se ha raggiunto la fine del pattern). Se, invece, il confronto dà esito negativo, l'algoritmo consulta la funzione fallimento per trovare nel pattern un nuovo carattere candidato al confronto, oppure, se il fallimento è avvenuto in corrispondenza del primo carattere del pattern, riparte dall'indice successivo nel testo, perché non ci sono informazioni da riutilizzare.

Codice 13.3: Un'implementazione dell'algoritmo di pattern matching KMP. Il metodo ausiliario `computeFailKMP` è descritto nel Codice 13.4.

```
/** Restituisce l'indice minimo in cui inizia pattern nel testo (oppure -1). */
public static int findKMP(char[] text, char[] pattern) {
  int n = text.length;
  int m = pattern.length;
  if (m == 0) return 0;       // ricerca banale di una stringa vuota
  int[] fail = computeFailKMP(pattern); // funzione calcolata da un metodo privato
  int j = 0;                  // indice nel testo
  int k = 0;                  // indice nel pattern
  while (j < n) {
    if (text[j] == pattern[k]) { // il pattern[0..k] corrisponde fin qui
      if (k == m - 1) return j - m + 1; // corrispondenza completa
      j++;                    // altrimenti cerca di estendere la corrispondenza
      k++;
    } else if (k > 0)
      k = fail[k - 1];        // riutilizza il suffisso di P[0..k-1]
    else
      j++;
  }
  return -1;                  // ricerca fallita
}
```

Costruzione della funzione fallimento di KMP

Per costruire la funzione fallimento usiamo il metodo presentato nel Codice 13.4, che è una sorta di processo "di bootstrap" (che "sta in piedi da solo"...) che confronta il pattern con se stesso secondo quanto previsto dall'algoritmo KMP. Ogni volta che due caratteri corrispondono, eseguiamo l'assegnazione $f(j) = k + 1$. Osserviamo che, essendo $j > k$ durante l'intera esecuzione dell'algoritmo, il valore di $f(k - 1)$ è sempre già stato definito nel momento in cui viene utilizzato.

Codice 13.4: Un'implementazione del metodo ausiliario `computeFailKMP`, a supporto dell'algoritmo di pattern matching KMP. Si osservi che l'algoritmo utilizza i valori della funzione fallimento calcolati in precedenza per calcolare in modo efficiente i nuovi valori.

```
private static int[] computeFailKMP(char[] pattern) {
  int m = pattern.length;
  int[] fail = new int[m];   // tutte le sovrapposizioni sono così uguali a zero
  int j = 1;
  int k = 0;
  while (j < m) {            // in questo passo calcola fail[j], se non è zero
    if (pattern[j] == pattern[k]) { // finora k+1 caratteri corrispondono
      fail[j] = k + 1;
      j++;
      k++;
    } else if (k > 0)        // k segue un prefisso che corrisponde
```

```
            k = fail[k - 1];
          else
            j++;
        }
        return fail;
    }
```

Prestazioni

Escludendo il calcolo della funzione fallimento, il tempo d'esecuzione dell'algoritmo KMP è chiaramente proporzionale al numero di iterazioni eseguite dal ciclo `while`. Per agevolare l'analisi, definiamo $s = j - k$: intuitivamente, s è lo spostamento complessivo del pattern verso destra rispetto alla posizione iniziale. Osserviamo che, durante l'intera esecuzione dell'algoritmo, si ha $s \leq n$. Durante ciascuna iterazione del ciclo si verifica uno dei tre casi seguenti:

- Se `text[j] = pattern[k]`, allora j e k aumentano di un'unità, lasciando così inalterato s.
- Se `text[j] ≠ pattern[k]` e `k > 0`, allora j non cambia e s aumenta almeno di un'unità, perché in questo caso s passa da $j - k$ a $j - f(k-1)$; osserviamo che questo significa un aumento di s di una quantità uguale a $k - f(k-1)$, che è certamente positiva perché $f(k-1) < k$.
- Se `text[j] ≠ pattern[k]` e `k = 0`, allora j aumenta di un'unità e s diminuisce di un'unità, perché k non cambia.

Quindi, durante ciascuna iterazione del ciclo, j o s aumenta di almeno un'unità (eventualmente aumentano entrambi); di conseguenza, il numero totale di iterazioni del ciclo `while` dell'algoritmo di pattern matching KMP è al massimo $2n$. Per ottenere questo risultato, però, bisogna ovviamente aver già calcolato la funzione di fallimento relativa al pattern.

L'algoritmo che calcola la funzione di fallimento richiede un tempo $O(m)$ e la sua analisi è analoga a quella dell'algoritmo KMP principale, anche se con un pattern di lunghezza m confrontato con se stesso. Abbiamo, quindi:

Proposizione 13.3: *L'algoritmo KMP risolve il problema del pattern matching, cercando un pattern di lunghezza m in un testo di lunghezza n, in un tempo $O(n + m)$.*

La correttezza di questo algoritmo discende dalla definizione stessa di funzione fallimento. Qualunque confronto che venga evitato è effettivamente non necessario, perché la funzione fallimento garantisce che tutti i confronti ignorati siano ridondanti: si tratterebbe di rifare confronti già fatti, con esito positivo, tra gli stessi caratteri.

Nella Figura 13.6 mostriamo l'esecuzione dell'algoritmo di pattern matching KMP applicato agli stessi dati (testo e pattern) dell'Esempio 13.1. Si noti l'uso della funzione fallimento per evitare di rifare alcuni confronti tra un carattere del pattern e un carattere del testo. Si noti, ancora, che l'algoritmo esegue complessivamente un numero di confronti inferiore a quello richiesto dall'algoritmo a forza bruta eseguito con le stesse stringhe (si veda la Figura 13.1).

Funzione di fallimento:

k	0	1	2	3	4	5
pattern $[k]$	a	b	a	c	a	b
$f(k)$	0	0	1	0	1	2

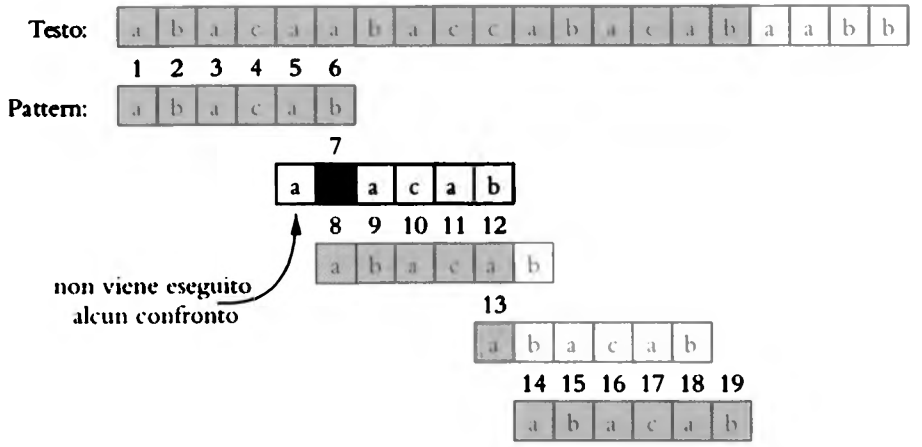


Figura 13.6: Visualizzazione del funzionamento dell'algoritmo di pattern matching KMP. L'algoritmo principale esegue 19 confronti tra caratteri, indicati con etichette numeriche (durante il calcolo della funzione fallimento vengono eseguiti altri confronti).

13.3 Trie

Gli algoritmi di pattern matching presentati nel Paragrafo 13.2 rendono più veloce la ricerca in un testo effettuando un'elaborazione preventiva del pattern (nell'algoritmo Boyer-Moore per calcolare la funzione `last` e nell'algoritmo KMP per calcolare la funzione fallimento). In questo paragrafo vedremo un approccio complementare: presentiamo algoritmi di pattern matching che elaborano preventivamente il testo, piuttosto che il pattern. Questo approccio è adatto a quelle applicazioni che eseguono ripetute ricerche all'interno di un testo prefissato, così che il costo iniziale della pre-elaborazione del testo sia compensato da un aumento della velocità di ciascuna ricerca successiva: ad esempio, un sito web che offra la possibilità di cercare pattern all'interno dell'opera *Hamlet* di Shakespeare, oppure un motore di ricerca che presenti all'utente pagine web contenenti il termine *Hamlet*).

Un *trie* (parola pronunciata come l'inglese "try" ma individuata come porzione della parola "re*trie*val") è una struttura dati ad albero per memorizzare stringhe al fine di agevolare un pattern matching veloce. L'applicazione principale per i trie è nel settore dell'*information retrieval* (da cui il nome), che significa "recupero dell'informazione" da una base di dati: un esempio tipico è la ricerca di una determinata sequenza di DNA all'interno di una base di dati genomica, che è costituita da una raccolta S di stringhe, tutte definite usando lo stesso alfabeto. Il tipo di interrogazione principale messo a disposizione da un trie è il pattern matching e il *prefix matching*: quest'ultima operazione prevede di cercare, tra tutte le stringhe appartenenti a un insieme S, quelle che iniziano con la stringa X, cioè che hanno X come prefisso.

13.3.1 Trie standard

Sia S un insieme di s stringhe, definite sull'alfabeto Σ, con la proprietà aggiuntiva che nessuna stringa di S sia un prefisso di un'altra stringa di S. Un ***trie standard*** per S è un albero ordinato T con le seguenti proprietà (illustrate nella Figura 13.7):

- Ogni nodo di T, tranne la radice, è etichettato con un carattere di Σ.
- I figli di un nodo interno di T hanno etichette distinte.
- T ha s foglie, ciascuna associata a una stringa di S, in modo che la concatenazione delle etichette dei nodi che si trovano lungo il percorso che va dalla radice a una foglia v di T generi la stringa di S che è associata a v.

Quindi, un trie T rappresenta le stringhe di S mediante percorsi che vanno dalla radice a una foglia di T. Si noti l'importanza di aver ipotizzato che nessuna stringa di S sia un prefisso di un'altra stringa di S: questo garantisce che ciascuna stringa di S sia associata in modo univoco a una foglia di T (questa ipotesi è simile al vincolo sui codici dei prefissi nella codifica di Huffman, che sarà descritta nel Paragrafo 13.4). Possiamo sempre rendere vera questa ipotesi aggiungendo alla fine di ciascuna stringa un carattere speciale che non appartenga all'alfabeto originario Σ.

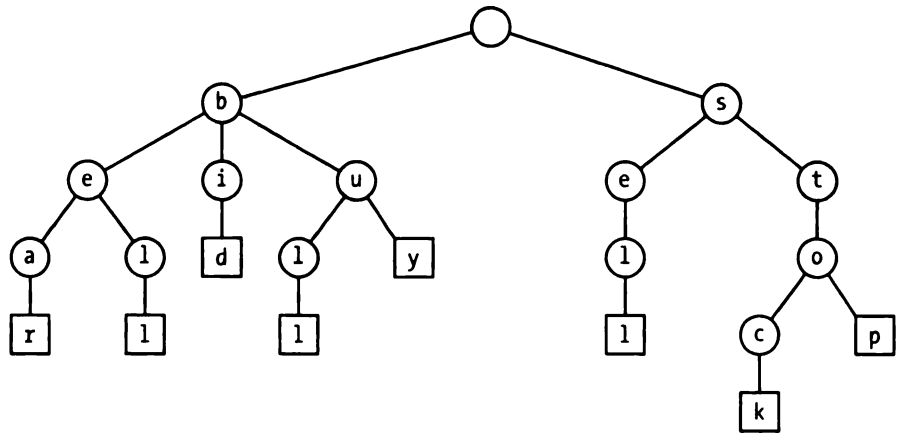


Figura 13.7: Trie standard per l'insieme di stringhe {bear, bell, bid, bull, buy, sell, stock, stop}.

Un nodo interno di un trie standard T può avere un numero di figli qualsiasi, compreso tra 1 e $|\Sigma|$. Per ogni carattere che è il carattere iniziale in almeno una stringa di S esiste un ramo che va dalla radice r a uno dei suoi figli. Inoltre, un percorso che vada dalla radice di T a un nodo interno v di profondità k corrisponde a un prefisso di lunghezza k, $X[0..k-1]$, di una stringa X di S. Inoltre, per ogni carattere c che può seguire il prefisso $X[0..k-1]$ in una delle stringhe dell'insieme S, esiste un figlio di v etichettato con c. In questo modo, un trie memorizza in modo sintetico i prefissi comuni esistenti all'interno di un insieme di stringhe.

Come caso speciale, se nell'alfabeto esistono soltanto due caratteri, il trie è fondamentalmente un albero binario, con alcuni nodi interni che possono avere un solo figlio

(cioè può essere un albero binario improprio). In generale, pur essendo possibile che un nodo interno abbia fino a $|\Sigma|$ figli, in pratica il grado medio di tali nodi è probabilmente molto inferiore: ad esempio, il trie della Figura 13.7 ha molti nodi interni con un solo figlio. Per insiemi di dati di maggiori dimensioni, è probabile che il grado medio dei nodi diventi minore nei livelli più profondi dell'albero, perché saranno sempre meno le stringhe che condividono un prefisso sempre più lungo e, quindi, sempre minori le possibilità di allungare quel prefisso. Inoltre, in molti linguaggi, in particolare in quelli naturali, esistono combinazioni di caratteri il cui verificarsi è veramente improbabile.

La proposizione seguente elenca alcune proprietà strutturali importanti di un trie standard.

Proposizione 13.4: *Un trie standard T che memorizza una raccolta S di s stringhe di lunghezza totale n i cui caratteri appartengono all'alfabeto Σ gode delle seguenti proprietà:*

- L'altezza di T è uguale alla lunghezza della più lunga stringa presente in S.
- Ogni nodo interno di T ha al massimo $|\Sigma|$ figli.
- T ha s foglie.
- Il numero di nodi di T è al massimo $n + 1$.

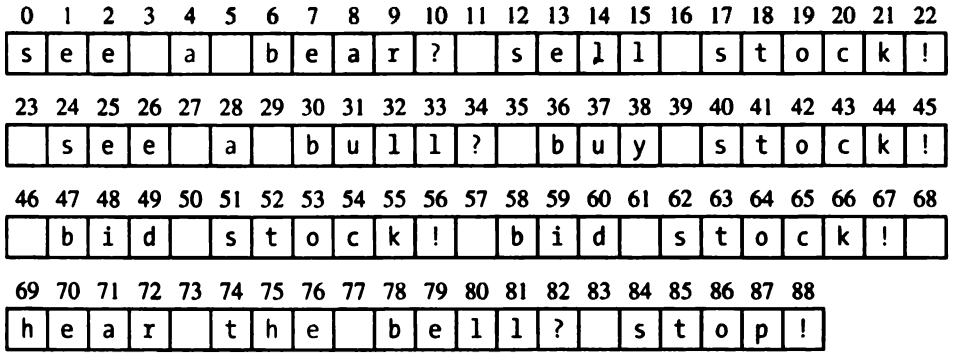
Il caso peggiore per il numero di nodi di un trie si verifica quando nessuna coppia di stringhe condivide un prefisso comune non vuoto: in questo caso, tranne la radice, tutti i nodi interni hanno un solo figlio.

Un trie T per un insieme S di stringhe può essere utilizzato per implementare un insieme o una mappa le cui chiavi siano le stringhe di S. In particolare, per cercare una stringa X in T è sufficiente scendere dalla radice lungo il percorso individuato dai caratteri di X: se tale percorso esiste e termina in una foglia, allora sappiamo che X è una delle stringhe di S. Ad esempio, nel trie della Figura 13.7, scendendo lungo il percorso corrispondente a "bull" si termina in una foglia. Se, invece, il percorso non esiste oppure esiste ma termina in un nodo interno, allora X non è una delle stringhe di S. Nell'esempio della Figura 13.7, il percorso corrispondente a "bet" non esiste e il percorso di "be" termina in un nodo interno: nessuna delle due parole appartiene all'insieme S.

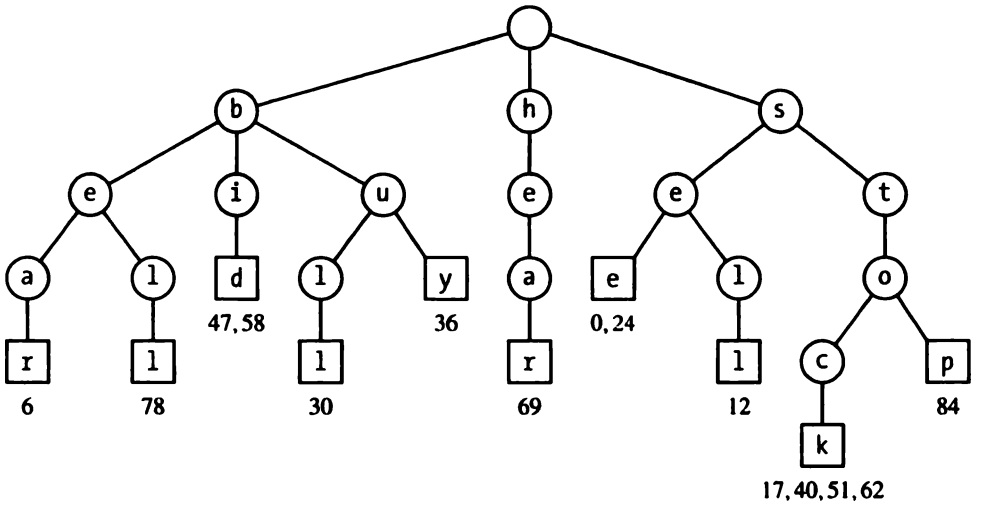
È facile vedere che il tempo d'esecuzione della ricerca di una stringa di lunghezza m è $O(m \cdot |\Sigma|)$, perché si visitano al massimo $m + 1$ nodi di T e in ciascun nodo si impiega un tempo $O(|\Sigma|)$ per determinare quale figlio abbia come etichetta il carattere successivo della stringa che si sta cercando. Si può effettuare quest'ultima ricerca in un tempo limitato da $O(|\Sigma|)$, anche se i figli dei nodi non sono ordinati, perché in ciascun nodo il numero massimo di figli è $|\Sigma|$; questo tempo può essere migliorato e reso $O(\log |\Sigma|)$ o anche $O(1)$, mettendo in corrispondenza i caratteri e i figli mediante una tabella di ricerca o una tabella hash secondaria in ciascun nodo, oppure usando una tabella di ricerca diretta di dimensione $|\Sigma|$, se $|\Sigma|$ è sufficientemente piccolo (come nel caso, ad esempio, di stringhe di DNA). Per queste ragioni, solitamente ci si aspetta che la ricerca di una stringa di lunghezza m venga eseguita in un tempo $O(m)$.

Dalla discussione precedente consegue che possiamo usare un trie per eseguire una forma speciale di pattern matching, chiamata ***word matching***, con il quale si vuole determinare se un dato pattern sia esattamente uguale a una delle parole di un testo. Il *word matching* differisce dal pattern matching standard perché il pattern non può essere una sottostringa

qualsiasi del testo: può essere solamente una delle sue parole. Per risolvere questo problema, ciascuna singola parola del documento originale deve essere aggiunta al trie (come si può vedere nella Figura 13.8). Una semplice estensione di questo schema consente di eseguire anche ricerche di *prefix-matching*, tuttavia non si riescono a eseguire in modo efficiente ricerche di pattern qualsiasi all'interno di un testo (ad esempio, quando il pattern è un suffisso proprio di una parola o si estende su due parole).



(a)



(b)

Figura 13.8: Esecuzione di word-matching in un trie standard: (a) testo in cui cercare (vengono escluse la cosiddette *stop word*, cioè articoli e preposizioni, perché porterebbero a frequenti violazioni del vincolo sull'assenza di prefissi all'interno dell'insieme); (b) trie standard per le parole del testo. Nelle foglie del trie sono state aggiunte alcune informazioni: gli indici nel testo in cui inizia la parola rappresentata dal nodo. Ad esempio, la foglia che corrisponde alla parola "stock" contiene un'annotazione che dice che tale parola è presente nel testo a partire dagli indici 17, 40, 51 e 62.

Per costruire un trie standard T per un insieme S di stringhe, possiamo usare un algoritmo incrementale che inserisce una stringa alla volta. Ricordiamo l'ipotesi relativa al fatto che nessuna stringa di S sia un prefisso di un'altra stringa di S. Per inserire una stringa X nel trie T, scendiamo lungo il percorso associato a X in T, partendo dalla radice e, quando rimaniamo bloccati perché non si riesce più a scendere, creiamo una nuova catena di nodi che memorizzino i caratteri rimanenti di X e la inseriamo in modo che il suo nodo iniziale diventi figlio del nodo in cui ci siamo fermati. Il tempo necessario per inserire una parola X di lunghezza m è simile a quello richiesto per la sua ricerca, con caso peggiore $O(m \cdot |\Sigma|)$, oppure valore atteso $O(m)$ se si usa in ciascun nodo una tabella hash secondaria. Di conseguenza, la costruzione dell'intero trie per l'insieme S richiede un tempo atteso $O(n)$, dove n è la lunghezza totale delle stringhe di S.

L'albero trie soffre di una potenziale inefficienza nell'uso dello spazio in memoria e questo ha suggerito lo sviluppo del *trie compresso* (*compressed trie*), nodo anche (per ragioni storiche) come *Patricia trie*. Nello specifico, il problema del trie standard sta nel fatto che possono esserci molti nodi che hanno un solo figlio, la cui esistenza è uno spreco di spazio. Ne parleremo nel prossimo paragrafo.

13.3.2 Trie compresso

Un *trie compresso* (*compressed trie*) è simile al trie standard, ma garantisce che ogni nodo interno del trie abbia almeno due figli. Come si può vedere nella Figura 13.9, questa regola viene messa in atto *comprimendo* in un unico ramo catene di nodi aventi un solo figlio. Sia T un trie standard. Diciamo che un nodo interno v di T è *ridondante* se v non è la radice e ha un solo figlio. Ad esempio, il trie della Figura 13.7 ha otto nodi ridondanti. Inoltre, diciamo che una catena di nodi avente $k \geq 2$ rami, $(v_0, v_1)\ (v_1, v_2) \ldots (v_{k-1}, v_k)$, è *ridondante* se:

- v_i è ridondante per $i = 1, \ldots, k-1$.
- v_0 e v_k non sono ridondanti.

Un trie standard T può essere trasformato in un trie compresso sostituendo ciascuna catena ridondante $(v_0, v_1)\ (v_1, v_2) \ldots (v_{k-1}, v_k)$ costituita da $k \geq 2$ rami con un unico nodo (v_0, v_k) e assegnando a v_k una nuova etichetta che sia la concatenazione delle etichette dei nodi $v_1, \ldots, v_k$.

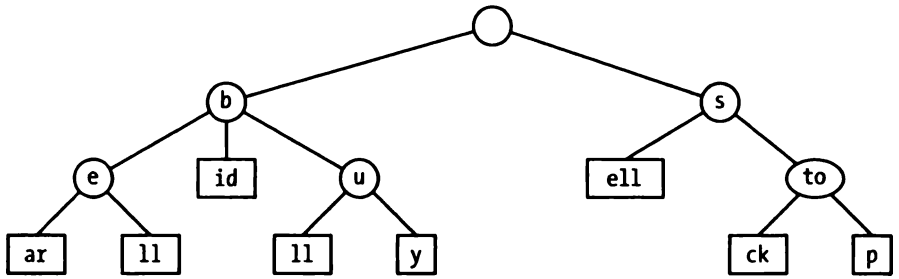


Figura 13.9: Trie compresso per l'insieme di stringhe {bear, bell, bid, bull, buy, sell, stock, stop}, da confrontare con il trie standard per lo stesso insieme, visto nella Figura 13.7. Si osservi che, oltre alla compressione avvenuta nelle foglie, è stato compresso anche un nodo interno, quello con etichetta "to", che è condiviso dalle parole "stock" e "stop".

I nodi compressi, quindi, sono etichettati con stringhe, che sono sottostringhe dell'insieme dato, e non più con singoli caratteri. Il vantaggio di un trie compresso, rispetto a un trie standard, è che il numero di nodi del trie compresso è proporzionale al numero di stringhe e non più alla loro lunghezza totale, come affermato dalla proposizione seguente (da confrontare con la Proposizione 13.4).

Proposizione 13.5: *Un trie compresso T che memorizza una raccolta S di s stringhe i cui caratteri appartengono all'alfabeto Σ di dimensione d gode delle seguenti proprietà:*

- Ogni nodo interno di T ha almeno due figli e al massimo d figli.
- T ha s foglie.
- Il numero di nodi di T è $O(s)$.

Il lettore attento si chiederà se la compressione dei percorsi fornisca effettivamente qualche vantaggio, dal momento che è controbilanciata dal corrispondente allungamento delle etichette dei nodi. In effetti, un trie compresso è davvero vantaggioso soltanto quando viene utilizzato come struttura di indicizzazione ***ausiliaria*** per un insieme di stringhe già memorizzato in una struttura principale, perché in tal modo non è più necessario memorizzare nel trie tutti i caratteri delle stringhe.

Supponiamo, ad esempio, che l'insieme S di stringhe sia memorizzato in un array, in modo che le stringhe siano $S[0], S[1], \ldots, S[s-1]$. Invece di memorizzare l'etichetta X di un nodo in modo esplicito, la rappresentiamo semplicemente come combinazione di tre numeri interi (i, j, k), tali che $X = S[i][j..k]$, cioè X è la sottostringa di $S[i]$ costituita dai caratteri che vanno dall'indice j all'indice k, inclusi (si veda un esempio nella Figura 13.10, confrontandolo anche con il trie standard della Figura 13.8).

Questo ulteriore schema di compressione ci consente di ridurre lo spazio totale occupato dal trie, passando da $O(n)$ per il trie standard a $O(s)$ per il trie compresso, essendo n la lunghezza totale delle stringhe di S e s il numero di stringhe di S. Abbiamo ovviamente ancora bisogno di memorizzare da qualche parte le diverse stringhe che appartengono a S, ma abbiamo ridotto lo spazio occupato dal trie.

La ricerca in un trie compresso non è necessariamente più veloce di quella in un trie standard, perché c'è ancora bisogno di confrontare ciascun carattere del pattern cercato con le etichette, potenzialmente costituite da più caratteri, che si incontrano attraversando i percorsi nel trie.

13.3.3 Trie dei suffissi

Una delle applicazioni principali degli alberi trie riguarda il caso in cui le stringhe dell'insieme S sono tutti i suffissi di una data stringa X. Un tale trie è chiamato ***trie dei suffissi*** (*suffix trie*) o ***albero dei suffissi*** della stringa X. Ad esempio, la Figura 13.11a mostra il trie dei suffissi per la stringa "minimize", che hà otto suffissi. Nel caso di un trie dei suffissi, la rappresentazione compatta che abbiamo visto nel paragrafo precedente può essere ulteriormente semplificata: l'etichetta di ciascun nodo è una coppia "j..k" e rappresenta la stringa $X[j..k]$ (come si può vedere nella Figura 13.11b). Per soddisfare il vincolo, tipico di tutti i trie, che nessun suffisso di X sia un prefisso di un altro suffisso di X, possiamo aggiungere alla fine di X (e, quindi, alla fine di ciascun suo suffisso) un carattere speciale, spesso indicato

con \$, che non faccia parte dell'alfabeto originario del problema, Σ. Quindi, se la stringa X ha lunghezza n, costruiamo un trie per l'insieme delle n stringhe $X[j..n-1]\$$, con $j = 0, \ldots, n-1$.

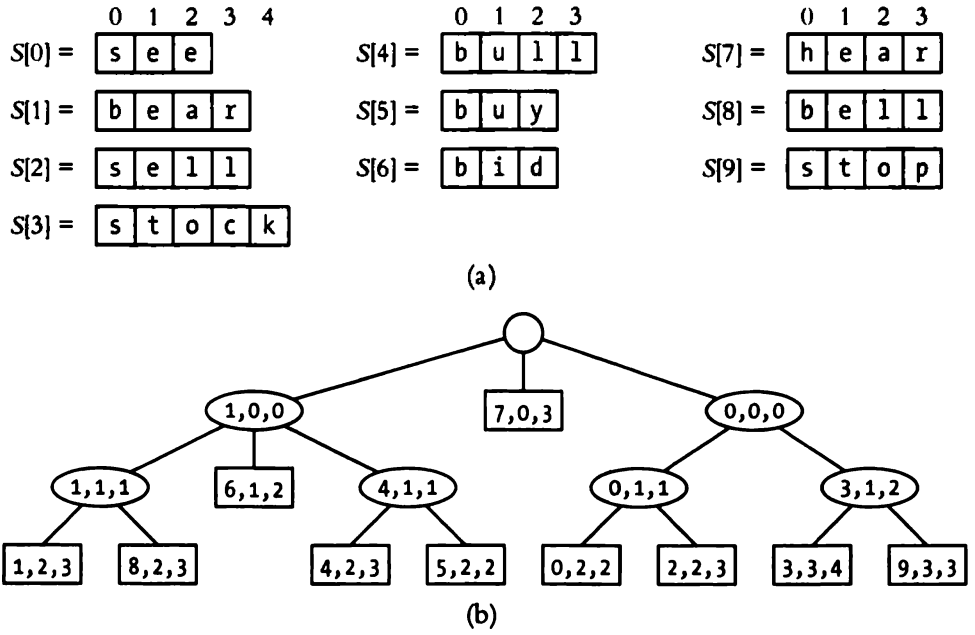


Figura 13.10: (a) Insieme di stringhe memorizzato in un array. (b) Rappresentazione compatta del trie compresso di S.

Risparmiare spazio

L'uso di un trie dei suffissi ci consente di risparmiare spazio rispetto a un trie standard, per via di alcune tecniche di compressione che si possono usare, tra le quali quelle già viste per i trie compressi.

Il vantaggio di una rappresentazione compatta di un trie risulta evidente nel caso dei trie dei suffissi. Dato che la lunghezza totale dei suffissi di una stringa X di lunghezza n è:

$$1+2+\cdots+n=\frac{n(n+1)}{2},$$

la memorizzazione in un trie di tutti i suffissi di X in modo esplicito richiederebbe uno spazio $O(n^2)$, mentre il trie dei suffissi rappresenta queste stringhe in modo implicito occupando uno spazio $O(n)$, come asserito in modo formale dalla proposizione che segue.

Proposizione 13.6: *La rappresentazione compatta di un trie dei suffissi T per la stringa X di lunghezza n usa uno spazio $O(n)$.*

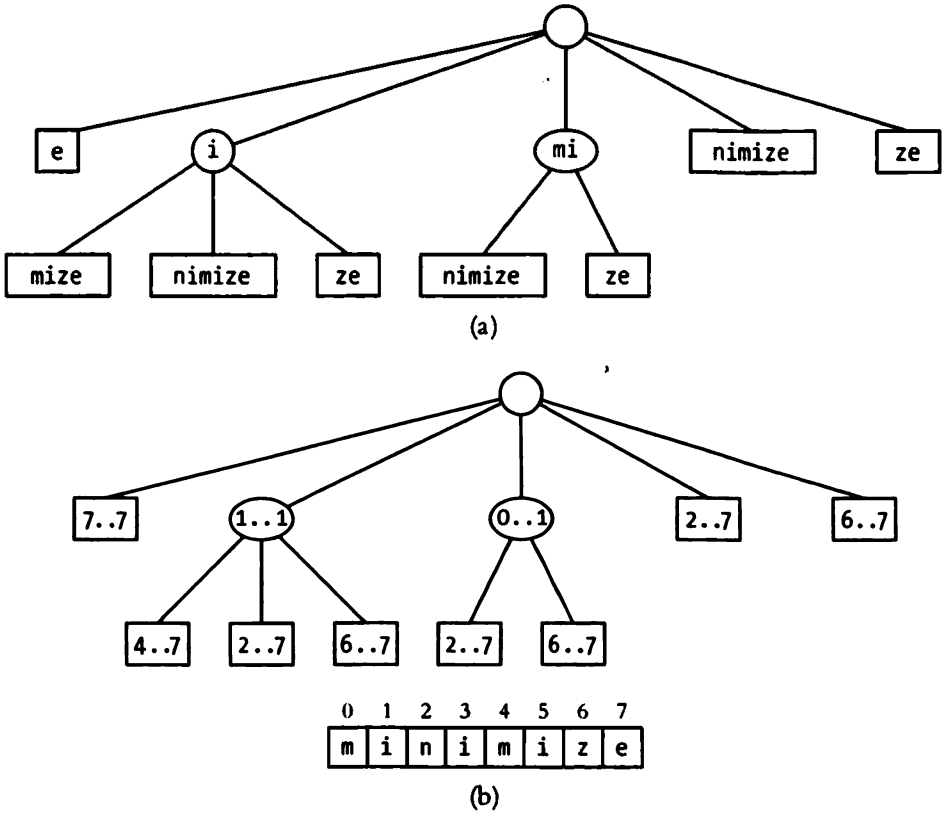


Figura 13.11: (a) Trie dei suffissi T per la stringa X = "minimize". (b) Rappresentazione compatta di T, dove la coppia $j..k$ rappresenta la sottostringa $X[j..k]$ della stringa di riferimento.

Costruzione

Possiamo costruire il trie dei suffissi per una stringa di lunghezza n usando un algoritmo incrementale come quello visto nel Paragrafo 13.3.1. Questa costruzione richiede un tempo $O(|\Sigma|n^2)$ perché la lunghezza totale dei suffissi è una funzione quadratica di n, però il trie dei suffissi (compatto) per una stringa di lunghezza n può essere costruito in un tempo $O(n)$ usando un algoritmo specifico, diverso da quello usato per i trie standard. Questo algoritmo di costruzione in un tempo lineare è, però, piuttosto complesso e non lo riportiamo qui. Comunque, quando vogliamo usare un trie dei suffissi per risolvere problemi, possiamo basarci sull'esistenza di questo algoritmo di costruzione veloce.

Uso di un trie dei suffissi

Il trie dei suffissi T per una stringa X può essere utilizzato per risolvere in modo efficiente il problema del pattern matching sul testo X: possiamo, infatti, determinare se un pattern P è una sottostringa di X cercando di seguire un percorso associato a P in T, perché P è una sottostringa di X se e solo se è possibile seguire tale percorso. La ricerca all'interno del trie T seguendo un percorso in discesa a partire dalla radice presuppone che i nodi di T memorizzino alcune informazioni aggiuntive, rispetto alla rappresentazione compatta che abbiamo visto per il trie dei suffissi:

Se il nodo v ha etichetta j..k e Y è la stringa di lunghezza y associata al percorso che va dalla radice a v (incluso), allora $X[k - y + 1..k] = Y$.

Questa proprietà garantisce che, se il pattern è presente nel testo, in un tempo $O(m)$ possiamo calcolare l'indice iniziale della sua occorrenza nel testo.

13.3.4 Indicizzazione nei motori di ricerca

Il World Wide Web contiene un'enorme quantità di documenti di testo (le pagine web). Le informazioni relative a queste pagine possono essere raccolte usando un programma chiamato *Web crawler*, che poi memorizza queste informazioni in una base di dati. Un *motore di ricerca* (*search engine*) per il Web permette agli utenti di recuperare da tale base di dati le informazioni più rilevanti, perché è in grado di individuare, tra le pagine del Web, le più rilevanti tra quelle che contengono le parole chiave (*keyword*) specificate. In questo paragrafo presenteremo un modello semplificato di un motore di ricerca.

Indici inversi

L'informazione fondamentale memorizzata da un motore di ricerca è una mappa, detta *indice inverso* (*inverted index*) o *file inverso* (*inverted file*), che contiene coppie chiave-valore del tipo (w, L), dove w è una parola e L è un contenitore di pagine che contengono la parola w. Le chiavi (cioè le parole) di questa mappa sono chiamate *termini indice* (*index term*) e dovrebbero costituire un insieme di voci di un vocabolario e di nomi propri, con la maggiore dimensione possibile. I valori di questa mappa sono detti *liste di occorrenza* (*occurrence list*) e dovrebbero contenere il massimo numero possibile di pagine web.

Un indice inverso può essere facilmente implementato con una struttura dati di questo tipo:

1. Un array che memorizza le liste di occorrenza associate ai singoli termini indice (senza rispettare alcun ordinamento particolare).
2. Un trie compresso associato all'insieme dei termini indice (cioè delle chiavi), con le foglie che memorizzano il valore dell'indice in corrispondenza del quale si trova, nell'array, la lista di occorrenza associata al termine indice.

Le liste di occorrenza vengono memorizzate al di fuori del trie per fare in modo che la dimensione della struttura dati che realizza il trie sia sufficientemente piccola da trovar posto nella memoria interna, mentre, per via della grande dimensione totale che hanno, le liste di occorrenza devono essere memorizzate sul disco, nella memoria di massa.

Con questa struttura dati, la ricerca di una singola parola chiave è simile a un problema di *word matching* (visto nel Paragrafo 13.3.1): si cerca la parola nel trie e si restituisce la lista di occorrenza associata.

Quando l'utente fornisce più parole chiave e desidera ottenere un elenco delle pagine che contengono *tutte* le parole chiave date, si recupera la lista di occorrenza di ciascuna parola usando il trie e si restituisce la loro intersezione. Per agevolare il calcolo dell'intersezione, le liste di occorrenza devono essere implementate con una sequenza ordinata in base all'indirizzo della pagina o con una mappa, in modo da consentire un'esecuzione efficiente delle operazioni tra insiemi.