Chapter 4

Algoritmi su stringhe

Le stringhe sono particolari sequenze, cioè successioni lineari di elementi di tipo omogeneo. In una stringa i singoli elementi vengono detti *caratteri* e l'insieme di tutti i caratteri ammessi in una data stringa è detto *alfabeto*. In questo appunti considereremo solo alfabeti finiti, cioè costituiti da un numero finito di simboli. Ciò che caratterizza le stringhe, e che ne giustifica uno studio "a parte" rispetto alle sequenze, è la particolarità delle operazioni che su di esse si vogliono comunemente effettuare, e che sono differenti rispetto alle manipolazioni che abbiamo visto per le sequenze; ad esempio, quasi mai abbiamo interesse, nelle applicazioni, a procedere all'ordinamento di una stringa né alla ricerca di un suo dato elemento. Il particolare uso cui sono soggette le stringhe in ormai numerossimi ambiti applicativi dell'Informatica si riflette poi, ed anche questo è motivo di interesse per studiarle in un corso di livello universitario, in una notevole quantità di strutture dati e algoritmi sviluppati per rappresentarle e manipolarle efficientemente.

4.1 Definizioni di base

Stringhe e operazioni sulle stringhe

Indicheremo con la lettera greca Σ un generico alfabeto e con le lettere finali dell'alfabeto latino (x, y e z) simboli generici del generico alfabeto. Casi particolarmente interessanti si hanno quando $\Sigma = \mathcal{B} = \{\mathtt{0}, \mathtt{1}\}$ o quando Σ coincide con l'insieme dei simboli ASCII. Altri casi importanti, per la vastità e la rilevanza delle applicazioni alla Biologia molecolare, si hanno quando le stringhe rappresentano sequenze di DNA (o di RNA), e dunque

$\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ (o $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$), oppure ancora sequenze di amminoacidi, pertanto $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{K}, \mathtt{L}, \mathtt{M}, \mathtt{N}, \mathtt{P}, \mathtt{Q}, \mathtt{R}, \mathtt{S}, \mathtt{T}, \mathtt{V}, \mathtt{W}, \mathtt{Y}\}$. Si noti che i caratteri dei vari alfabeti che utilizzeremo in questi appunti saranno scritti utilizzando il font `typewriter`. Il simbolo x come variabile o come "generico carattere" sarà dunque diverso dal simbolo `x` inteso come ben preciso carattere di un alfabeto assegnato. Qualora possano sorgere ambiguità, ricorreremo anche alla notazione, tipicamente utilizzata nei linguaggi di programmazione, che prevede di racchiudere costanti di tipo carattere se stringhe fra apici o doppi apici.

Per indicare stringhe generiche utilizzeremo le prime lettere dell'alfabeto greco (α, β, γ), in minuscolo. Scriveremo $\alpha \in \Sigma^*$ per indicare che la stringa α è definita sull'alfabeto Σ, e cioè che i suoi caratteri appartengono a Σ[1]. La stringa vuota, cioè la stringa composta da zero caratteri, è definita su qualunque alfabeto e per indicarla useremo il simbolo ϵ.

La *concatenazione* è l'operazione che, date due stringhe α e β su uno stesso alfabeto Σ, produce la stringa $\gamma = \alpha\beta$ su Σ costituita dalla giustapposizione di α e β, nell'ordine. La concatenazione è un'operazione associativa: $(\alpha\beta)\gamma = \alpha(\beta\gamma)$.

La *lunghezza* di una stringa è il numero di caratteri che compongono la stringa stessa. La lunghezza della stringa vuota è 0. La lunghezza di una stringa α verrà spesso indicata con $|\alpha|$. La lunghezza della stringa γ ottenuta dalla concatenazione delle due stringhe α e β è $|\gamma| = |\alpha| + |\beta|$.

Individueremo i caratteri di una stringa α utilizzando indici compresi fra 0 ed $n-1$ (anziché fra 1 e n), secondo quanto avviene nella gran parte dei linguaggi di programmazione moderni. Ne consegue che il carattere di indice i è l'$(i+1)$-esimo; per indicarlo useremo a seconda dei casi le notazioni $\alpha[i]$, di stile "programmativo", e α_i, di stile matematico.

Per indicare la stringa ottenuta dalla concatenazione, k volte, di uno stesso simbolo x utilizzeremo spesso la notazione x^k. Ad esempio, la stringa $\mathtt{0}\mathtt{1}^k\mathtt{0}$ indica la stringa costituita dalla concatenazione del simbolo `0`, seguito dalla stringa di k `1`, seguita ancora dal simbolo `0`.

Una stringa β è *sottostringa* di α se esistono due stringhe γ e δ (eventualmente vuote) tali che $\alpha = \gamma\beta\delta$. Se $\beta \neq \epsilon$ e almeno una fra γ e δ è diversa dalla stringa vuota, diremo dice che β è una sottostringa *propria* di α. Se γ (δ) è la stringa vuota, allora β è *prefisso* (*suffisso*) di α. Prefissi e suffissi costituiti da k caratteri vengono anche detti prefissi e suffissi di *ordine* k.

[1] Giustificheremo poco oltre questa scrittura, parlando delle espressioni regolari.

Diremo poi che una stringa β è un *bordo* di α se β è contemporaneamente prefisso e suffisso di α. Ad esempio, la stringa `re` è un bordo della stringa `restituire`.

Data una stringa α, utilizzeremo la notazione $\alpha[i..j]$ (o, alternativamente, la notazione $\alpha_{i..j}$) per indicare la sottostringa di α costituita dai caratteri $\alpha[i], \alpha[i+1], \ldots, \alpha[j]$; nel caso $i > j$, $\alpha[i..j]$ indica la stringa vuota. Utilizzeremo anche la notazione $\alpha^{(j)}$ come alternativa a $\alpha_{0..j-1}$, cioè per indicare il prefisso di ordine j di α.

Se β è una sottostringa di α diremo anche che α contiene (almeno) una *occorrenza* di β. Se $\alpha = \gamma\beta\delta$ e $|\gamma| = k$, diremo che β *occorre* alla posizione k di α o anche che β ha un *match* alla posizione k di α, $k \geq 0$. La prima occorrenza di una stringa β in α è quella che occorre alla posizione minima k, per la quale cioè non esiste $k' < k$ tale che β occorre anche alla posizione k'.

Un *allineamento* di una stringa α alla posizione i di β, con $i \leq |\beta| - |\alpha|$, è l'operazione mediante la quale si sovrappone (idealmente) α con la sottostringa $\beta[i..i + |\alpha| - 1]$. L'allineamento viene utilizzato negli algoritmi di string matching per verificare se α occorre ad una data posizione di β. In questo contesto, diremo che un allineamento di α con β *ha successo* se tutti i caratteri allineati coincidono, cioè se α occorre in β alla posizione di allineamento; in caso contrario diremo che l'allineamento *fallisce*.

Linguaggi formali

Un *linguaggio formale* L su un alfabeto Σ è un insieme di stringhe definite sull'alfabeto Σ. L'aggettivo "formale" si riferisce al fatto che l'appartenenza di una data stringa ad L è definita in modo rigoroso, ad esempio descrivendo matematicamente le proprietà che deve soddisfare una generica stringa del linguaggio.

Esempio 4.41. Il linguaggio L così definito:

$$L = \{\alpha \in \{0,1\}^* | \exists \beta \in \{0,1\}^*, \alpha = \mathtt{1}\beta\mathtt{1}\}$$

include tutte e sole le stringhe binarie che iniziano e terminano con `1`

Tipicamente, nel contesto delle applicazioni informatiche, sono tuttavia altre le principali modalità con cui vengono definiti i linguaggi; e precisamente una modalità *generativa* e una *riconoscitiva*. Secondo la prima modalità,

sono date regole formali mediante le quali è possibile "costruire" o "generare" tutte le stringhe del linguaggio che si intende definire; con la seconda viene invece dato un algoritmo in grado di "decidere" se una data stringa appartiene al linguaggio[2] In questi appunti studieremo le *espressioni regolari.* che sono un formalismo generativo, e gli automi a stati finiti, che sono un formalismo riconoscitivo.

Espressioni regolari

Le *espressioni regolari* su un alfabeto Σ sono un formalismo per descrivere insiemi di stringhe su Σ. Le espressioni regolari su Σ sono definite ricorsivamente nel modo seguente.

Base. Per ogni carattere $\mathtt{a} \in \Sigma$, $\mathtt{a}$ è un'espressione regolare che denota l'insieme $\{\mathtt{a}\}$. Anche ϵ è un'espressione regolare, che denota l'insieme vuoto.

Parentesi. Se $\mathcal{E}$ è un'espressione regolare, la scrittura $(\mathcal{E})$ è un'espressione regolare *equivalente* alla prima, cioè che denota lo stesso insieme di stringhe.

Concatenazione Se $\mathcal{E}$ ed $\mathcal{F}$ sono espressioni regolari, la scrittura $\mathcal{E}\mathcal{F}$ è un'espressione regolare che denota l'insieme di stringhe che possono essere scritte concatenando una stringa di $\mathcal{E}$ con una stringa di $\mathcal{F}$.

Unione Se $\mathcal{E}$ ed $\mathcal{F}$ sono espressioni regolari, la scrittura $\mathcal{E}|\mathcal{F}$ è un'espressione regolare che denota l'unione delle di $\mathcal{E}$ e di $\mathcal{F}$.

Chiusura Se $\mathcal{E}$ è un'espressione regolare, la scrittura $\mathcal{E}^*$ è un'espressione regolare che denota l'insieme delle stringhe ottenute concatenando stringhe di $\mathcal{E}$ un numero arbitrario $i \geq 0$ di volte.

Lo scopo delle parentesi è di forzare un ordine di composizione delle espressioni diverso da quello standard (che prevede che la chiusura abbia più alta priorità, seguita dalla concatenazione e infine dall'unione).

[2] Ovviamente esistono linguaggi per cui non è sempre possibile decidere l'appartenenza o non appartenenza di una stringa, ma nessuno di tali *linguaggi indecidibili* sarà considerato in questi appunti.

Esempio 4.42. Consideriamo l'alfabeto binario $\mathcal{B} = \{\mathtt{0}, \mathtt{1}\}$. L'espressione regolare $E_{\mathcal{B}} = \mathtt{0} + \mathtt{1}$ denota il linguaggio $L_{\mathcal{B}} = \{\mathtt{0}, \mathtt{1}\}$, le cui stringhe sono gli stessi caratteri dell'alfabeto. Abbiamo cioè, dal punto di vista insiemistico, $L_{\mathcal{B}} = \mathcal{B}$. Consideriamo ora l'espressione regolare $E^*_{\mathcal{B}} = (\mathtt{0} + \mathtt{1})^*$; dalla definizione dell'operatore di chiusura sappiamo che essa denota l'insieme di stringhe ottenute concatenando le stringhe (caratteri) `0` e `1` un numero arbitrario di volte. In altri termini, dunque, $E^*_{\mathcal{B}}$ denota il linguaggio di tutte le possibili stringhe binarie. Questo giustifica la notazione $\mathcal{B}^*$ per indicare tale insieme di tutte le stringhe binarie. Un'analoga costruzione per il generico alfabeto Σ giustifica la notazione Σ^* per indicare l'insieme di tutte le possibili stringhe su Σ.

Esempio 4.43. L'espressione regolare $\mathtt{0}|\mathtt{1}^*\mathtt{10}$ su $\mathcal{B}$ va interpretata come $\mathtt{0}|((\mathtt{1}^*)\mathtt{10})$. Essa identifica quindi l'insieme di stringhe $S = \{\mathtt{0}, \mathtt{10}, \mathtt{110}, \ldots\}$.

Esempio 4.44. Posto $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, la scrittura $\mathtt{a}(\mathtt{b}|\mathtt{c})^*\mathtt{a}$ denota l'insieme delle stringhe su Σ che iniziano e terminano con il carattere `a` e che contengono un numero arbitrario (incluso 0) di caratteri `b` e `c` comunque alternati.

Esempio 4.45. La scrittura $(\mathtt{0}|\mathtt{1})(\mathtt{1}|\mathtt{10})^*$ denota l'insieme delle stringhe su $\mathcal{B}$ che non contengono due `0` consecutivi.

A coloro che utilizzano il sistema operativo Unix/Linux, le espressioni regolari saranno probabilmente note per via del comando `grep` (tra gli altri). Al riguardo, la pagina di manuale di questa utility (richiamabile sotto Linux mediante il comando `man grep`) include un'introduzione alle espressioni regolari molto semplice e ben fatta.

Un insieme di stringhe definite su un alfabeto Σ è detto *linguaggio* su Σ. L'insieme può essere definito mediante elencazione esplicita delle stringhe che lo compongono, come nel caso di $\{\mathtt{0}, \mathtt{1}, \mathtt{11}, \mathtt{1001}\}$, ovvero può essere identificato mediante regole formali, come nel caso delle espressioni regolari. Infatti, poiché un'espressione regolare identifica un insieme di stringhe, essa denota proprio un linguaggio. Un linguaggio che può essere definito mediante espressioni regolari è detto *linguaggio regolare*.

Esempio 4.46. Gli insiemi di strighe introdotti negli esempi 4.43, 4.44 e 4.45 sono linguaggi regolari perché definiti mediante espressioni regolari su un alfabeto opportuno.

Multiset

Un *multiset* è una collezione in cui i singoli elementi componenti possono essere ripetuti. Ad esempio, $\{1, 1, 2, 3, 3, 3\}$ è un multiset in cui gli elementi 1 e 3 compaiono con molteplicità maggiore di uno. È evidente che un insieme è un caso particolare di multiset. Dati due multiset A e B, definiamo l'operazione di unione $\cup_m$ che produce un multiset costituito dagli elementi in A o in B, ciascuno dei quali con molteplicità pari alla somma delle molteplicità che ha in A e B. Ad esempio, se A è il multiset indicato in precedenza e $B = \{1, 2, 2, 3, 3, 4\}$, abbiamo $A \cup_m B = \{1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 3, 4\}$.

4.2 String matching

Il problema fondamentale dello string matching può essere espresso nel modo seguente.

> Data una stringa $T \in \Sigma^*$ di lunghezza n, detta *testo*, ed una stringa $P \in \Sigma^*$ di lunghezza $m \leq n$, detta *pattern*, determinare tutte le (eventuali) occorrenze di P in T.

Prima di considerare algoritmi che risolvono il problema sopra enunciato, facciamo qualche semplice riflessione sulla sua *complessità*. Al riguardo, ci concentreremo solo sulle operazioni di confronto fra caratteri, trascurando altre operazioni che, come ad esempio l'aggiornamento degli indici utilizzati per scorrere gli array, non determinano il costo asintotico complessivo del codice eseguibile.

Per quanto possa a prima vista sembrare sorprendente, è possibile progettare algoritmi che, in dipendenza del particolare input, esaminano significativamente meno di n caratteri del testo (si veda l'Esercizio 4.50). Tuttavia, nel *caso più sfavorevole* (*worst-case*), tutti i caratteri di T e di P devono essere analizzati. Si consideri, ad esempio, la situazione in cui $T = \mathtt{1}^n$ e $P = \mathtt{1}^m$. Si tratta, naturalmente, di un caso limite che però serve a provare la precedente affermazione. Infatti, poiché ogni posizione $T[i]$, $i = 0, \ldots, n-m$, identifica un match, ogni carattere del testo è coinvolto in almeno un match col pattern. Ne consegue che, non esaminando anche un solo carattere di T si incorrerebbe nella possibilità di non rilevare almeno una occorrenza di P in T. Testo e pattern devono quindi essere esaminati per intero, con costo $\Omega(n + m) = \Omega(n)$ (poiché $n \geq m$).

La limitazione $\Omega(n)$ è il riferimento per valutare la bontà degli algoritmi

in senso asintotico. Va però precisato che l'efficienza pratica dipende non solo dal costo computazionale asintotico, ma anche da altri fattori teorici (semplicità, uso di strutture dati ausiliarie, cardinalità dell'alfabeto Σ su cui sono definiti testo e pattern) oltreché, ovviamente, dalla bontà della codifica.

In questa sezione prenderemo in considerazione algoritmi che possono pre-elaborare il pattern prima di procedere alla lettura del testo. Poiché, spesso, risulta $n >> m$, una elaborazione preliminare sul pattern può essere conveniente quando serva a ridurre il costo di match col testo, anche nel caso in cui tale elaborazione preliminare abbia costo relativamente elevato rispetto ad m.

4.2.1 Algoritmo a forza bruta

Si chiama in questo modo, in generale, un algoritmo che prova tutte le potenziali soluzioni fino a trovarne una "buona". Nel caso dello string matching, ciò significa provare tutte le possibili collocazioni del pattern nel testo, cioè verificare se P occorre alla posizione i di T, per $i = 0, 1, \ldots, n - m$. Questa descrizione si traduce nel seguente codice, che stampa tutte (le posizioni de) le occorrenze di P in T. L'algoritmo non prevede alcuna elaborazione preliminare del pattern.

```
BRUTE-FORCE(T, P)
 1  n ← |T|
 2  m ← |P|
 3  i ← 0
 4  while i ≤ n − m
 5        j ← 0
          // Verifica se P = T[i..i + m − 1]
 6        while j < m and P[j] = T[i + j]
 7              j ← j + 1
 8        if j = m
 9              print i
10        i ← i + 1
```

Si noti che l'iterazione sul testo è di lunghezza determinata. Per tale motivo, alla riga 4 sarebbe più appropriato utilizare il costrutto **for**, anziché il **while**. Abbiamo preferito questa seconda soluzione perché in questo modo

Brute-Force risulta più direttamente confrontabile con un codice che presenteremo nella prossima sezione e che ne rappresenterà una evoluzione.

Per molti problemi di natura combinatoriale, gli algoritmi che utilizzano forza bruta sono estremamente inefficienti in quanto il numero di potenziali soluzioni da controllare è enorme. Nel caso in questione, le possibili occorrenze del pattern nel testo sono al più n e dunque non c'è la temuta esplosione combinatoriale e l'analisi del costo computazionale dell'algoritmo risulta molto semplice. Le operazioni di confronto tra caratteri delle due stringhe vengono eseguite solo nella riga 6. Il corpo del costrutto **while** più esterno viene eseguito $n - m$ volte e, per ognuna di esse, il confronto di riga 6 viene eseguito al più m volte, per un totale di al più $nm - m^2$ confronti. Nel caso in cui m sia costituito da pochi caratteri, oppure quando risulti $m \approx n$, il costo computazionale dell'algoritmo Brute-Force può essere considerato lineare in n e dunque ottimale in senso asintotico. Viceversa, se m è una frazione costante di n, ad esempio, $m = n/2$, il costo computazionale dell'algoritmo è $\Theta(n^2)$, risulta cioè quadratico nella dimensione del testo.

Esercizio 4.47. Modificare lo pseudo-codice di Brute-Force in modo che restituisca una lista con tutte le occorrenze, anziché stamparle.

Esercizio 4.48. Si fornisca un'istanza di input, testo e pattern, che porta l'algoritmo Brute-Force ad eseguire esattamente $nm - m^2 + m$ confronti di caratteri.

Esercizio 4.49. Si modifichi l'algoritmo Brute-Force nel modo seguente: per un dato allineamento di P con T, cioè fissato il valore dell'indice i, il confronto dei caratteri viene eseguito da destra verso sinistra (con riferimento allo pseudocodice, il ciclo interno decrementa j partendo dal valore $m - 1$). Se P occorre alla posizione i di T, allora l'algoritmo incrementa i di 1. Se invece si verifica un mismatch per un dato $j \geq 0$, allora incrementa i della quantità $\max\{1, j - \mathcal{C}(T[i+j])\}$. La funzione $\mathcal{C}$ è definita nel modo seguente: per $x \in \Sigma$, $\mathcal{C}(x)$ è l'indice dell'ultima occorrenza di x in P, ovvero $\mathcal{C}(x) = -1$ se x non occorre in P. Ad esempio, se $P =$ `aacagaag` sull'alfabeto del DNA, risulta $\mathcal{C}(\texttt{a}) = 6, \mathcal{C}(\texttt{c}) = 2, \mathcal{C}(\texttt{g}) = 7$ e $\mathcal{C}(\texttt{t}) = -1$. Si dimostri che l'algoritmo così modificato è corretto e si fornisca un esempio in cui l'algoritmo non esamina tutti i caratteri di T.

Esercizio 4.50. Con riferimento all'algoritmo Brute-Force modificato come nell'Esercizio 4.49, si determini una classe di esempi (stringhe T e P) che comporta l'analisi di $o(n)$ caratteri del testo.

4.2.2 Algoritmo di Knuth, Morris e Pratt

L'inefficienza dell'algoritmo a forza bruta risiede nel fatto che tutte le possibili posizioni di match fra pattern e testo vengono provate; in altri termini, l'indice i che stabilisce la posizione di allineamento viene sempre incrementato di un'unità, anche quando un incremento maggiore non provocherebbe il mancato rilevamento di eventuali match. Prima di mostrare un caso molto semplice in cui ciò è possibile, diamo qualche ulteriore definizione.

Definizione 4.51. Con la locuzione *shift (del pattern) di s posizioni* indicheremo l'operazione in base alla quale il pattern viene riallineato sul testo alla posizione $i + s$, dove i è la posizione di allineamento attuale. Se $s > 1$, gli allineamenti alle posizioni $i + 1, \ldots, i + s - 1$ non vengono considerati.

Definizione 4.52. Se risulta $P_{0..j-1} = T_{i..i+j-1}$ e $P_j \neq T_{i+j}$, $0 \leq i \leq n - m$ e $0 \leq j < m$, si dice che la posizione i è una *falsa partenza* di lunghezza j.

Esempio 4.53. Per il testo $T =$ `perdindirindina` e il pattern $P =$ `dindina` le posizioni 3 e 6 sono false partenze di lunghezza, rispettivamente, 5 e 2. La posizione 11 non è una falsa partenza perché $11 > n - m = 15 - 7 = 8$, cioè 11 non è una posizione di allineamento completa del pattern sul testo.

Nel seguito sarà utile fare riferimento anche ad un match completo del pattern come ad una falsa partenza di lunghezza m.

Esempio 4.54. Sia $P = \mathtt{a}^m$ e sia i una posizione di falsa partenza di lunghezza $j < m$. Questo significa, in particolare, che $T_{i+j} \neq P_j$, ma poiché $P_0 = P_1 = \ldots = P_j$, ne consegue che nessuno shift di lunghezza $s \leq j$ potrà produrre un match completo del pattern; con un tale shift, infatti, il carattere $P_{j-s} = \mathtt{a}$ viene allineato a $T_{i+j} \neq \mathtt{a}$ e dunque si può stabilire anticipatamente che il confronto fallisce. Sulla base dell'informazione disponibile, il prossimo tentativo che può avere successo è alla posizione $i + j + 1$. La situazione è illustrata in Figura 4.1, per il caso particolare $m = 5$ e $j = 3$.

Pattern formati dalla ripetizione di uno stesso carattere sono certamente molto particolari. Tuttavia l'esempio suggerisce come valga la pena verificare se esistono altre strutture che possano condurre a concreti risparmi computazionali, che consentano cioè valori di shift del pattern maggiori di 1.

Procediamo innanzitutto (seguendo la metodologia *Top-Down*) con una modifica del codice di Brute-force in modo da predisporlo ad incorporare arbitrari valori di shift. La nuova formulazione, Generic-shift-PM,

Figure 4.1: Dall'alto. Mismatch al quarto confronto nell'attuale allineamento. Uno shift del pattern di una posizione genera in modo predicibile un mismatch al terzo confronto. Il prossimo allineamento con possibilità di successo si ha solo con uno shift di quattro posizioni del pattern sul testo.

utilizza al suo interno una funzione SHIFT$_P$ che richiede in input un valore intero $j \in \{0, 1, \ldots, |P|\}$ e restituisce due valori, pure di tipo intero.

Generic-shift-PM(T, P)

```
 1  n ← |T|
 2  m ← |P|
 3  i = j = 0
 4  while i ≤ n − m
 5        while j < m and P[j] = T[i + j]
 6              j ← j + 1
 7        if j = m
 8              print i
 9        s, k = SHIFT_P(j) // Variabile k usata solo per chiarezza
10        i = i + s
11        j = k
```

Come si vede dal codice, i due valori restituiti da SHIFT$_P$ vengono utilizzati per incrementare l'indice i, che definisce la posizione attuale di allinea-

mento (come in BRUTE-FORCE), e per ri-inizializzare l'indice j che (ancora come in BRUTE-FORCE) viene utilizato per individuare le coppie di caratteri di pattern e testo da confrontare. La figura 4.2 illustra graficamente l significato dei due valori.

$\ldots\ T_i\ T_{i+1}\ T_{i+2}\ T_{i+3}\ T_{i+4}\ T_{i+5}\ T_{i+6}\ T_{i+7}\ T_{i+8}\ \ldots$

? ? ? ? ? ?

$P_0\ P_1\ P_2\ P_3\ P_4\ P_5\ \ldots$

$\ldots\ T_i\ T_{i+1}\ T_{i+2}\ T_{i+3}\ T_{i+4}\ T_{i+5}\ T_{i+6}\ T_{i+7}\ T_{i+8}\ \ldots$

? ? ? ? ?

$\rightarrow\ \rightarrow\ P_0\ P_1\ P_2\ P_3\ P_4\ P_5\ \ldots$

Figure 4.2: Utilizzo delle quantità restituite da SHIFT$_P$. L'esempio è relativo ai particolari valori $s = 2$ e $k = 1$. Sopra. Attuale posizione di allineamento del pattern sul testo. Sotto. Nuova posizione di allineamento; come si vede, il pattern è posizionato alla posizione $i+s = i+2$; inoltre, i confronti partono dalla posizione k-esima del pattern e dunque, in questo caso, dalla posizione 1. Nella figura questo fatto è evidenziato dalla mancanza del punto interogativo fra P_0 e T_{i+2}.

Il codice di GENERIC-SHIFT-PM è una generalizzazione di BRUTE-FORCE. Infatti, se la procedura SHIFT$_P$ è implementata in modo da restituire sempre la coppia di valori $s = 1$ e $k = 0$, ciò che si ottiene è esattamente il comportamento di BRUTE-FORCE: l'indice i viene incrementato di una unità ad ogni iterazione sul testo e i confronti col pattern iniziano sempre dalla posizione 0 di quest'ultimo.

Siamo ovviamente interessati a valori di shift maggiori, che aumentano la velocità dell'algoritmo, in particolare dell'incremento s della posizione di allineamento. Naturalmente ciò non deve venire a scapito della correttezza. In altri termini, non dobbiamo "perdere" occorrenze. Allo scopo è necessario che il valore di s restituito da SHIFT$_P$ soddisfi le due seguenti condizioni:

correttezza nessuno shift di $s' < s$ posizioni può dare origine ad un match completo del pattern;

ammissibilità sulla base di ciò che può essere dedotto dallo studio preliminare del pattern (la sua pre-elaborazione) e i confronti *già effettuati* con il testo, è possibile che lo shift di s posizioni produca un match completo.

Si noti che ogni shift di lunghezza maggiore di $|P|$ è sempre ammissibile, semplicemente perché nell'attuale posizione di allineamento, nessun confronto coinvolge caratteri del testo "così lontani". Più in generale, dopo una falsa partenza di lunghezza j, ogni shift di lunghezza superiore a j è ammissibile. Il seguente risultato ci aiuta invece a capire quali sono gli shift ammissibili di lunghezza non maggiore rispetto alla lunghezza della falsa partenza, che sono i più interessanti.

Teorema 4.55. Sia m la lunghezza del pattern P e supponiamo che l'attuale posizione i di allineamento di P sul testo T dia origine ad una falsa partenza di lunghezza $j \leq m$. Uno shift $s = j - b$ è ammissibile alla posizione i se e solo se $P^{(j)}$ ha un bordo di lunghezza b e (nel caso $j < m$) $P^{(j+1)}$ non ha un bordo di lunghezza $b + 1$.

Dim. Per l'ipotesi sulla falsa partenza risulta:

$$\begin{array}{rcl} P_0 & = & T_i \\ P_1 & = & T_{i+1} \\ & \dots & \\ P_{j-b} & = & T_{i+j-b} \\ & \dots & \\ P_{j-1} & = & T_{i+j-1} \end{array} \tag{4.1}$$

D'altra parte il riallineamento alla posizione $i + s = i + j - b$ determina, fra gli altri, i seguenti confronti:

$$\begin{array}{rcl} P_0 & \stackrel{?}{=} & T_{i+j-b} \\ P_1 & \stackrel{?}{=} & T_{i+j-b+1} \\ & \dots & \\ P_{b-1} & \stackrel{?}{=} & T_{i+j-1} \end{array} \tag{4.2}$$

Da (4.1) e (4.2) possiamo concludere che il riallineamento ha possibilità di condurre ad un match completo solo se:

$$\begin{array}{rcl} P_0 & = & P_{j-b} \\ P_1 & = & P_{j-b+1} \\ & \dots & \\ P_{b-1} & = & P_{j-1} \end{array}$$

cioè solo se il prefisso di ordine j di P ha un bordo di lunghezza b. Se ciò non accade, almeno un confronto è destinato a fallire in modo prevedibile e lo shift non è ammissibile. Viceversa, in presenza del bordo, se $j = m$ lo shift è certamente ammissibile perché i confronti sono soddisfatti (l'argomentazione precedente è del tutto reversibile). Se invece $j < m$, allora non possiamo ignorare il fatto che è stato effettuato un ulteriore confronto, fra P_j e T_{i+j}, del quale sappiamo anche l'esito alla luce dell'ipotesi sulla falsa partenza, e cioè $P_j \neq T_{i+j}$. Il riallineamento alla posizione $i + j - b$ fa sì che il carattere T_{i+j} venga confrontato con P_b. Se dunque risulta $P_b = P_j$ è evidente che il nuovo allineamento è destinato a fallire, cioè non è ammissibile. Se invece $P_b \neq P_j$ allora, poiché la disuguaglianza non è transitiva, nulla possiamo concludere a priori sull'esito del confronto fra P_j e T_{i+j} e dunque lo shift è ammissibile. Si noti che la condizione $P_b = P_j$, unitamente all'ipotesi che $P^{(j)}$ abbia un bordo di lunghezza b, implica che $P^{(j+1)}$ ha un bordo di lunghezza $b + 1$. Ciò completa la dimostrazione. □

La figura 4.3 illustra graficamente le argomentazioni del teorema.

Sulla base del Teorema 4.55, e delle osservazioni che lo hanno preceduto, possiamo definire il seguente predicato $A_P(j, s)$ che, per valori interi positivi $0 \leq j \leq |P|$ e $s \geq 1$, vale TRUE se e soltanto se s è uno shift ammissibile per una falsa partenza di lunghezza j.

$$A_P(j,s) = \begin{cases} \textsc{True} & \text{se } s \geq j+1 \\ \textsc{True} & \text{se } (s \leq j < |P|) \wedge \\ & \big(P^{(j)} \text{ ha un bordo di lunghezza } j-s\big) \wedge \\ & \big(P^{(j+1)} \text{ non ha un bordo di lunghezza } j-s+1\big) \\ \textsc{True} & \text{se } (s \leq j = |P|) \wedge \\ & (P \text{ ha un bordo di lunghezza } j-s) \\ \textsc{False} & \text{altrimenti} \end{cases}$$

Il calcolo del predicato $A_P(j, s)$ può essere banalmente effettuato in tempo $O(\max\{1, j - s\})$ utilizzando la mera definizione.

Il Teorema 4.55 e il conseguente predicato A_P caratterizzano gli shift ammissibili. Che cosa possiamo dire sulla correttezza? Qual è lo shift corretto da applicare in presenza di più shift ammissibili? La risposta in realtà è molto

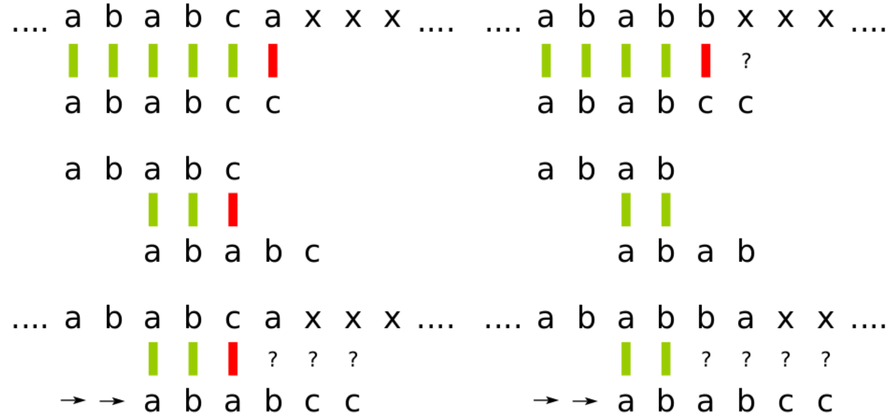


Figure 4.3: Sinistra. Falsa partenza di lunghezza 5 (in alto). Poiché il prefisso di ordine 5 del pattern non ha un bordo di lunghezza 3 (riga centrale), sarebbe inutile tentare uno shift di 2=5-3 posizioni perché il mismatch `c` $\neq$ `a` (ultima riga) è predicibile. Destra. Falsa partenza di lunghezza 4 (in alto). In questo caso il prefisso di ordine 4 del pattern ha un bordo di lunghezza 2 (riga centrale) ed inoltre il prefisso di ordine 5 del pattern non ha un bordo di lunghezza 3 (`aba` $\neq$ `abc`). Uno shift di 2 posizioni è dunque ammissibile (ultima riga). Si noti, infatti, che il mismatch che avrà origine alla terza posizione del pattern dopo il riallineamento (indicato con ?) non è prevedibile con l'informazione acquisita (sappiamo solo che il carattere del testo coinvolto nel confronto non è una `c`).

semplice. Se due o più shift risultano ammissibili solo quello di lunghezza minima è anche corretto. Questo ovviamente perché utilizzare uno shift più lungo può portare al mancato rilevamento di un match.

Il Teorema 4.55 può essere sfruttato in chiave computazionale perché consente di stabilire, a seguito di una falsa partenza di lunghezza j, quali shift abbiano possibilità di successo, cioè siano ammissibili. Al riguardo, si può precalcolare il valore del predicato $A_P(j, s)$ per ogni valore consentito dei parametri j ed s e memorizzare il risultato in una tabella ad accesso veloce. Tuttavia, per ogni possibile valore di j ci possono essere più valori s per cui s è uno shift ammissibie (si veda l'esempio di Figura 4.4). È facile rendersi conto che, in tal caso, il valore corretto da inserire nella tabella è quello *minimo*, perché questo è l'unico che corrisponde ad uno shift che soddisfa anche la proprietà di correttezza (shift corretto, per brevità).

Possiamo riassumere quanto esposto sopra nella seguente *proprietà*, che costituisce la base per l'implementazione della funzione $\text{SHIFT}_P(j)$.

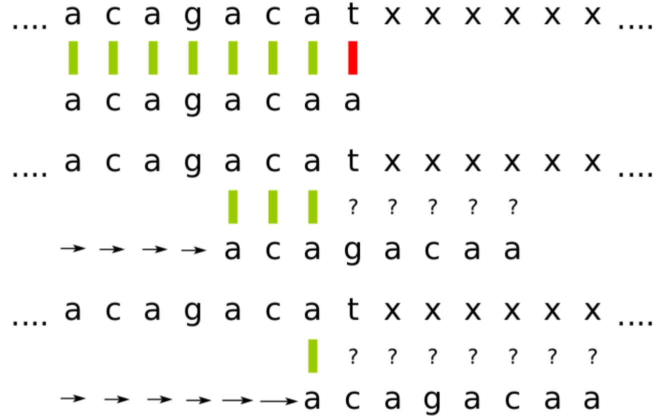


Figure 4.4: Il prefisso di ordine 7 del pattern, cioè la stringa `acagaca`, ha un bordo di lunghezza 1 e un bordo di lunghezza 3, cui corrispondono i due shift ammissibili $s = 7 - 3 = 4$ e $s = 7 - 1 = 6$. Il secondo shift, che sarebbe auspicabile dal punto di vista computazionale, in quanto esclude un maggior numero di allineamenti, non è tuttavia accettabile perché potrebbe condurre alla non identificazione di un match.

Proprietà 4.56. Supponiamo che la posizione i del testo T sia una falsa partenza di lunghezza $j \in \{0, 1, \ldots, |P|\}$ per il pattern P. Lo shift

$$s_j = \min_{1 \leq s \leq j+1} \{s | A_P(j, s)\}$$

è corretto e ammissibile e, in particolare, (se $s_j \leq j$) garantisce che i confronti:

$$\begin{array}{rcl} P_0 & \stackrel{?}{=} & T_{i+s_j} \\ P_1 & \stackrel{?}{=} & P_{i+s_j+1} \\ & \cdots & \\ P_{j-s_j-1} & \stackrel{?}{=} & T_{i+j-1} \end{array}$$

sono tutti soddisfatti.

La quantità s_j è precisamente il primo valore che deve essere restituito dalla funzione $\text{SHIFT}_P(j)$. Inoltre, la precedente proprietà stabilisce che, sulla nuova posizione di allineamento, i primi $j - s_j$ confronti saranno certamente soddisfatti. Si può quindi ripartire ri-inizializzando j al valore $j - s_j$ anziché

a zero. Dunque proprio $d_j = \max\{0, j - s_j\}$ è il secondo valore che deve essere restituito da $\textsc{shift}_P(j)$.

La funzione Best-Prefix calcola le quantità s_j, per valori di j nel range $1, \ldots, |P|$. Poiché ogni "chiamata" del predicato $A_P(j, s)$ impiega tempo $O(\max\{1, j - s\}$, è immediato rendersi conto che il costo computazionale di Best-Prefix è cubico rispetto ad m (per la presenza di 3 cicli annidati, uno dei quali "nascosto" nell'implementazione del predicato $A_P(j, s)$). Per i valori tipici di m in molte (anche se non certo nella totalità delle) applicazioni, una tale complessità non crea alcun tipo di problema.

```
Best-Prefix(P)
1  m ← |P|
2  for j ← 0 to m
3       s_j = j + 1
4       for s ← 1 to j
5            if A_P(j, s)
6                 s_j = s
7                 break
8  return ⟨s_0, s_1, ..., s_m⟩
```

Sulla base dei valori calcolati da Best-Prefix(P) è poi possibile costruire la tabella (di $m+1$ colonne) che contiene le due quantità s_j e d_j.

Esempio 4.57. Nel caso del pattern $P =$ `dindina` la tabella (che chiameremo tabella *Best-Prefix*, perché costruita a partire dai valori calcolati dalla funzione Best-Prefix) è mostrata in Figura 4.5.

A questo punto la funzione $\textsc{shift}_P(j)$ corrisponde ad un semplice table look-up. L'algoritmo Generic-shift-PM che ne consegue prende il nome di (algoritmo di) Knuth-Morris-Pratt, dal nome dei tre ricercatori che per primi lo hanno proposto. Riportiamo di seguito l'algoritmo completo che calcola tutte le occorrenze di P in T, inserendole in una lista L.

j (lunghezza falsa partenza)	Prefisso di ordine j di P	s_j	d_j
0		1	0
1	*d*	1	0
2	*di*	2	0
3	*din*	4	0
4	*dind*	4	0
5	*dindi*	5	0
6	*dindin*	3	3
7	*dindina*	7	0

Figure 4.5: Valori di s_j e d_j relativi al pattern $P = dindina$.

```
KNUTH-MORRIS-PRATT(T, P)
 1  n = |T|
 2  m = |P|
 3  L = []
 4  ⟨s_0, s_1, ..., s_m⟩ = BEST-PREFIX(P)
 5  for j = 0 to
 6      d_j = max{0, j − s_j}
 7  i = j = 0
 8  while i ≤ n − m
 9      while j < m and P[j] = T[i + j]
10          j = j + 1
11      if j = m
12          APPEND(L, i)
13      i = i + s_j
14      j = d_j
15  return L
```

L'esercizio 4.61 chiede di dimostrare che la complessità dell'algoritmo di Knuth, Morris e Pratt è $\Theta(n)$, cioè è lineare nella dimensione del testo.

Esercizio 4.58. Simulare in modo dettagliato l'esecuzione dell'algoritmo Knuth-Morris-Pratt su input $T =$ `perdindirindina` e $P =$ `dindina`.

Esercizio 4.59. Si fornisca la tabella *Best-Prefix* corrspondente al patterm $P =$ `abracadabra`.

Esercizio 4.60. Si fornisca la tabella *Best-Prefix* corrspondente al patterm $P =$ `acagacgtacag`.

Esercizio 4.61. Si provi che la complessità dell'algoritmo Knuth-Morris-Pratt è $\Theta(n + \alpha(m))$ nel worst-case, dove $\alpha(m)$ è il costo della fase di preprocessing. Suggerimento: si rifletta sul valore della quantità $s_j + d_j$.

Esercizio 4.62. Se l'alfabeto su cui sono definiti pattern e testo è binario, l'occorrenza di un mismatch $P[j] \neq T[i + j]$ determina comunque l'identità del carattere $T[i + j]$. Si ridefinisca il valore di s_j considerando il caso di alfabeti binari.

Esercizio 4.63. Si consideri il caso in cui il pattern può contenere un carattere jolly e si modifichi opportunamente l'implementazione delle funzioni Best-Prefix e Knuth-Morris-Pratt.

4.2.3 Matching di un insieme di pattern

L'algoritmo di Knuth, Morris e Pratt è meno efficiente, in pratica, di altri algoritmi che sono stati proposti per il problema dello string matching. Fra le ragioni che ci hanno indotto a presentarlo (oltre al merito di essere stato il primo algoritmo proposto fra quelli carattarizzati da un tempo di esecuzione lineare nella lunghezza del testo), particolarmente importante è il fatto che esso può essere generalizzato al caso del cosiddetto *set matching*, cioè al problema che chiede di determinare tutte le occorrenze di un insieme di pattern in un testo. In tale circostanza l'algoritmo, che prende il nome di algoritmo di *Aho-Corasick*, risulta competitivo anche sul piano pratico.

L'algoritmo di Aho-Corasick si definisce in modo naturale utilizzando un automa a stati finiti. Procederemo quindi in questo modo; rivisiteremo dapprima l'algoritmo di Knuth, Morris e Pratt in termini di automi, quindi estenderemo quest'ultimo in modo naturale al caso di un insieme di pattern.

Algoritmo KMP come automa a stati finiti

Gli *automi a stati finiti* sono modelli in grado di descrivere il comportamento di semplici dispositivi di calcolo. Sono ampiamente utilizzati in molti settori dell'Informatica, ad esempio nello studio dei linguaggi formali e dei protocolli di comunicazione.

Informalmente, un automa a stati finiti deterministico, o semplicemente *ASFD*, può essere visto come un calcolatore elementare dotato di stato interno che, ad ogni istante di tempo, "legge" un simbolo da un dispositivo di input e, in dipendenza di tale simbolo e dello stato interno corrente, cambia il valore di quest'ultimo. Ad ogni stato q possono poi essere associate varie azioni che l'automa esegue quando transita in q.

In modo formale, un *ASFD* F è una quintupla

$$F = \{\Sigma, Q, q_0, Q_f, \delta\},$$

in cui

- Σ è un insieme finito di *simboli*;
- $Q = \{q_0, q_1, \ldots, q_{m-1}\}$ è un insieme finito i cui elementi sono gli *stati* dell'automa;
- q_0 è un elemento speciale di Q, detto *stato iniziale*;

- $Q_f \subseteq Q$ è un insieme di stati detti *stati finali*;

- δ è la funzione che determina le *transizioni* di stato. Essa mappa coppie $\langle stato, simbolo \rangle$ in stati: $\delta : Q \times \Sigma \to Q$.

Alcune precisazioni sono opportune a questo punto. Innanzitutto osserviamo che gli elementi di Σ possono rappresentare entità molto diverse. Ad esempio, in applicazioni di text-processing, essi possono semplicemente essere i caratteri dell'alfabeto su cui è definito il testo, mentre in applicazioni di controllo possono benissimo rappresentare l'insieme dei possibili messaggi/segnali che l'unità di controllo riceve da un sensore. Analogamente, le azioni (eventualmente) associate agli stati interni possono essere molto diverse, sempre a seconda delle applicazioni; si può andare dalla semplice stampa di messaggi all'invio di segnali di controllo ad opportuni attuatori.

Le *computazioni* di un automa sono definibili in modo molto intuitivo: ad ogni passo, l'automa si trova in uno stato q (inizialmente $q = q_0$), legge un simbolo x dall'input e transita nello stato $\delta(q, x)$. La computazione termina quando si verifica una delle due seguenti condizioni:

1. non vi sono più simboli di input, oppure,

2. in corrispondenza dello stato attuale e del simbolo letto, la funzione di transizione non risulta specificata.

Il numero di transizioni effettuate prima della terminazione è detto *lunghezza* della computazione e ne rappresenta in modo naturale una misura di complessità.

Un contesto molto importante di utilizzo degli automi è proprio nel campo dei linguaggi formali e dell'elaborazione di stringhe. In tali ambiti applicativi essi sono etichettati come *automi riconoscitori* proprio perché vengono utilizzati per *riconoscere* stringhe con particolari proprietà. Tale ambito è precisamente quello di nostro interesse in questi appunti.

Per gli automi riconoscitori i simboli di input sono chiaramente i caratteri dell'alfabeto Σ. La sequenza di simboli presentati in input all'automa forma la *stringa* di input. Si dice che l'automa *riconosce* o *accetta* una stringa $\alpha = \alpha_0\alpha_1 \ldots \alpha_{n-1}$ se e solo se:

1. a partire dallo stato iniziale q_0, ogni transizione di stato risulta definita, cioè non si verifica mai la condizione 2 di terminazione; detto ancora

in termini più precisi, se q^i indica lo stato prima della lettura dell'i-esimo carattere in input (con $q^0 = q_0$), allora il valore $q^{i+1} = \delta(q^i, \alpha_i)$ è sempre definito.

2. $q^n \in Q_f$, cioè lo stato interno dopo la lettura di tutti i caratteri di input è uno degli stati finali, che nel caso di automi riconoscitori sono anche detti di *accettazione* dell'input.

Esempio 4.64. Si consideri l'ASFD definito nel modo seguente: $\Sigma = \mathcal{B} = \{\mathtt{0}, \mathtt{1}\}$, $Q = \{P, D\}$, $q_0 = P$, $Q_f = \{P\}$ e

$$\delta(P, \mathtt{1}) = \delta(D, \mathtt{0}) = D, \qquad \delta(P, \mathtt{0}) = \delta(D, \mathtt{1}) = P.$$

Notiamo che la funzione di transizione è specificata per ogni possibile coppia $\langle stato, simbolo \rangle$; ne consegue che le transizioni di stato sono sempre definite, qualunque sia la stringa in input. Consideriamo, ad esempio, la stringa, $\alpha = \mathtt{1011}$. Le corrispondenti transizioni di stato sono:

$$\begin{aligned} \delta(q^0, \mathtt{1}) &= \delta(P, \mathtt{1}) = D = q^1 \\ \delta(q^1, \mathtt{0}) &= \delta(D, \mathtt{0}) = D = q^2 \\ \delta(q^2, \mathtt{1}) &= \delta(D, \mathtt{1}) = P = q^3 \\ \delta(q^3, \mathtt{1}) &= \delta(P, \mathtt{1}) = D = q^4 \end{aligned}$$

Poiché $q^4 = D \notin Q_f$, la stringa non è accettata. È facile dimostrare per induzione sulla lunghezza della stringa in input che questa viene riconosciuta se e solo se contiene un numero pari di `1`.

Ricordando che un linguaggio formale altro non è che un insieme di stringhe, possiamo affermare che l'automa dell'esempio 4.64 *riconosce il linguaggio* L_P così definito:

$$L_P = \{x \in \mathcal{B}^* | x \text{ contiene un numero pari di } \mathtt{1}\}$$

proprio perché riconosce tutte e sole le stringhe di L_P. Per inciso, un tale automa può essere utilizzato (nel campo delle comunicazioni di segnali binari) per fare il cosiddetto *parity checking*, che è forse la più semplice tecnica di riconoscimento di errori di trasmissione.

Un formalismo comune per rappresentare un ASFD è quello dei *grafi di transizione*, i cui nodi ed archi rappresentano, rispettivamente, stati e transizioni. Ogni arco è etichettato da un simbolo di input. Lo stato iniziale

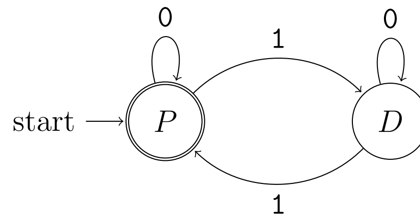


Figure 4.6: Grafo di transizione per l'ASFD dell'esempio 4.64.

viene evidenziato mediante una freccia entrante (e non uscente da alcun altro nodo) mentre gli stati finali sono indicati tramite doppia cerchiatura. La figura 4.6 descrive l'automa dell'esempio 4.64.

Ritorniamo ora al problema fondamentale dello string matching. Ad ogni pattern P può essere associato, in modo elementare, un opportuno automa a stati finiti F_P che riconosce P. Si noti infatti che si tratta di riconoscere una sola stringa (o, se si preferisce, un linguaggio costituito da una sola stringa). Ad esempio, l'automa di figura 4.7 riconosce il pattern `mamma` e solo tale pattern.

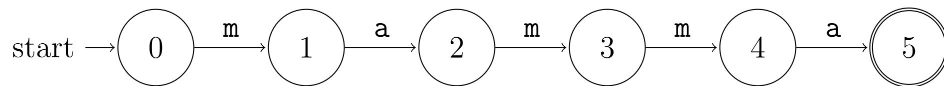


Figure 4.7: Automa che riconosce la stringa `mamma`.

Si noti che la funzione di transizione dell'automa di figura 4.7 è quasi sempre indefinita. Ad esempio, nello stato iniziale, qualsiasi simbolo $x \neq \texttt{m}$ provoca la terminazione della computazione (e il mancato riconoscimento dell'input) in quanto $\delta(0, x)$ è indefinita.

Utilizziamo proprio il pattern `mamma` come esempio e modifichiamo l'*automa base* di figura 4.7 allo scopo che possa operare il riconoscimento di tutte le occorrenze del pattern all'interno di un testo arbitrario.

Per ottenere il risultato che ci siamo prefissi dovremo ovviamente associare allo stato di accettazione un'azione di *notifica* che è stato rilevato un match del pattern nel testo. Questa però è l'aspetto meno rilevante, perché non riguarda la struttura dell'automa. Sia dunque $A_P = \{\Sigma, Q, q_0, Q_f, \delta\}$

l'automa che vogliamo realizzare; le singole componenti sono definite come segue.

- L'alfabeto Σ di input di F_P coincide con l'alfabeto sul quale sono definiti il pattern P e il testo.
- L'insieme Q include $m+1$ stati, che possiamo denotare con i numeri interi da 0 a m, dove, al solito, m è la lunghezza del pattern ($m = |P|$).
- Senza perdita di generalità, ma anzi per facilitare la comprensione della struttura dell'automa, possiamo stabilire che lo stato iniziale sia proprio lo stato 0 e che l'unico stato di accettazione, o di riconoscimento, sia lo stato m, cioè $q_0 = 0$ e $Q_f = \{m\}$.
- La funzione di transizione $\delta(q, x)$ è definita per ogni possibile coppia $\langle \textit{stato}, \textit{simbolo} \rangle$, perché la computazione si dovrà arrestare solo in base alla condizione 2, cioè per esaurimento dell'input. La definizione di δ è chiaramente la parte più complessa dell'automa e costituisce la fondamentale differenza con l'automa base, in cui essa è invece quasi sempre indefinita, come già osservato a proposito dell'automa di figura 4.7. Vedremo che, con poche varianti, la funzione di transizione implementa il calcolo delle quantità s_j definite dalla funzione BEST-PREFIX.

Per comprendere la definizione che daremo della funzione δ, precisiamo dapprima qual è il "significato" che si intende associare ad ogni singolo stato. Nel caso di questi automi il senso è molto semplice: lo stato j corrisponde alla situazione in cui l'automa ha riconosciuto esattamente j caratteri del pattern nel testo; o, ancora più precisamente, la situazione in cui gli ultimi j caratteri letti dall'automa corrispondono ai primi j caratteri del pattern. Sempre con riferimento all'automa base della figura 4.7, se, ad esempio, l'automa si trova nello stato 3, ciò significa che gli ultimi 3 caratteri letti sono, nell'ordine, `m`, `a` e `m`.

Stabilito il significato, ci chiediamo quale debba essere il comportamento dell'automa nel generico stato j su input il generico carattere x dell'alfabeto. La situazione più semplice si verifica quando $x = P_j$, cioè quando l'input coincide il $(j+1)$-esimo carattere del pattern. In tal caso l'automa deve transitare nello stato $j+1$, in accordo al significato attribuito agli stati. Ad esempio, se nello stato 3 l'input è il carattere $P_3 = \mathtt{m}$, l'automa di figura 4.7 deve transitare nello stato 4. Consideriamo ora il caso in cui $x \neq P_j$. In tale

situazione, evidentemente, l'automa non sta rilevando un match fra pattern e testo, ma la domanda fondamentale è: da dove (cioè da quale stato) deve ripartire l'automa nella ricerca del prossimo match?

Il ragionamento corretto qui è molto simile a quello già fatto per determinare l'entità dello shift per l'algoritmo KNUTH-MORRIS-PRATT, e forse anche un poco più semplice. Poiché l'automa si trova nello stato j e l'input è il carattere x, gli ultimi $j+1$ caratteri letti dall'automa formano la stringa $P_{0..j-1}x$. Ora, se tale stringa ha un bordo di lunghezza k, si può anche affermare che gli ultimi k caratteri letti coincidono con la stringa $P_{0..k-1}$ e dunque l'automa può transitare nello stato k. Ovviamente, anche in questo caso bisogna considerare il bordo più lungo, nel caso i bordi siano più d'uno. Se non ci sono bordi l'automa deve ripartire dallo stato 0.

Consideriamo nuovamente l'automa di figura 4.7. Supponiamo che esso si trovi nello stato 3 e che il prossimo input sia la lettera `a`. Questo vuol dire che gli ultimi 4 caratteri letti formano la stringa $P_{0..2}\texttt{a} = \texttt{mama}$. Tale stringa ha un (solo) bordo di lunghezza 2 e dunque è corretto per l'automa transitare nello stato 2, che appunto rappresenta la situazione in cui gli ultimi 2 caratteri letti sono `m` e `a`. Rientra in questo caso, e dunque non va trattato a parte, anche il comportamento dell'automa nello stato di accettazione. Se l'automa di figura 4.7 si trova nello stato 6 (e dunque ha appena rilevato un match) e il prossimo carattere in input fosse una m, esso dovrebbe transitare nello stato 3 e non nello stato 1, perché la stringa $P_{0..m-1}\texttt{m} = \texttt{mammam}$ ha un bordo di lunghezza 1 e uno di lunghezza 3 e va data preferenza a quest'ultimo, che descrive una situazione "più avanzata" nel processo di riconoscimento della prossima occorrenza del pattern.

La figura 4.8 mostra l'automa (quasi) completo per il riconoscimento del pattern `mamma`. Nel diagramma mancano tantissimi collegamenti; abbiamo detto infatti che la funzione di transizione è definita per ogni coppia $\langle \textit{stato}, \textit{simbolo} \rangle$ e dunque, ipotizzando che l'alfabeto non sia limitato alle due sole lettere `a` e `m`, mancano tutte le transizioni etichettate con differenti caratteri. Ovviamente queste riportano l'automa nello stato iniziale e dunque, per esigenze di chiarezza, tutte le transizioni mancanti devono essere intese come entranti nello stato 0.

Gli argomenti riportati sopra si riassumono nel seguente Teorema.

Teorema 4.65. Per ogni stato j e per ogni simbolo di input x, la funzione di transizione δ dell'ASFD F_P corrispondente al pattern P è così definita

$$\delta(j,x) = \max\{k \leq m | P_{0..k-1} \text{ è suffisso di } P_{0..j-1}x\}.$$

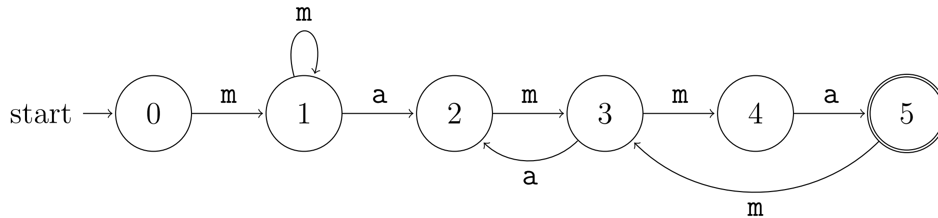


Figure 4.8: Grafo di transizione per l'ASFD che riconosce la stringa `mamma` all'interno di un testo. Tutte le transizioni mancanti (da ogni stato e per ogni possibile carattere in input) si devono intendere entranti nel nodo 0.

Si noti che la funzione $\delta(j, x)$ del Teorema 4.65 è ben definita in quanto considera tutti i suffissi (non solo quelli propri). In particolare, se $x = P_j$ risulta ovviamente $P_{0..j} = P_{0..j-1}x$ (il prefisso è improprio) e dunque $\delta(j, x) = j+1$, come previsto. Se invece $x \neq P_j$ e per nessun $k > 0$ risulta che $P_{0..k-1}$ è suffisso proprio di $P_{0..j-1}x$, allora il teorema prevede di considerare la stringa vuota ϵ (che è prefisso e suffisso improprio di qualunque stringa), e poiché $|\epsilon| = 0$ il valore della funzione è, correttamente, $\delta(j, x) = 0$.

Esempio 4.66. Costruiamo l'ASFD per la ricerca di occorrenze del pattern $P = 001001$ in un testo sull'alfabeto $\mathcal{B} = \{0, 1\}$. Possiamo subito costruire la prima porzione del grafo, con gli stati e le transizioni ovvie (figura 4.9). Per ogni stato, dobbiamo ora capire in quale altro stato deve transitare l'automa quando legge il simbolo di input non specificato in figura 4.9. Per il teorema 4.65, per ogni stato $j < m$, dobbiamo considerare il più lungo prefisso $P_{0..k-1}$ che è anche suffisso di $P_{0..j-1}x$ (dove x è il bit complementare di P_j). Tale valore k indica il nuovo stato. Per $j = m$, dobbiamo invece considerare entrambi i simboli di input. Per la determinazione effettiva della transizione ci aiutiamo con la tabella 4.1. L'automa completo è riportato in figura 4.10.

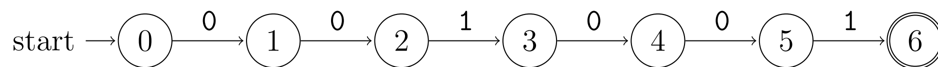


Figure 4.9: Grafo di transizione "parziale" relativo all'automa dell'esempio 4.66.

Per la costruzione della funzione δ relativa ad un pattern arbitrario si procede comunque come indicato nell'esempio 4.66. Ovviamente è necessario

j	$P_{0..j-1}$	x	$P_{0..j-1}x$	$P_{0..k-1}$	$\delta(j,x)$
0	ϵ	1	1	ϵ	0
1	0	1	01	ϵ	0
2	00	0	000	00	2
3	001	1	0011	ϵ	0
4	0010	1	00101	ϵ	0
5	00100	0	001000	00	2
6	001001	0	0010010	0010	4
6	001001	1	0010011	ϵ	0

Table 4.1: Determinazione della transizione "non banale" per l'automa dell'esempio 4.66. Nel caso $k < m$, x indica il simbolo che, in figura 4.9, *non* porta nello stato $k+1$. Per $k = m$ sono invece stati considerati entrambi i simboli.

considerare, per ogni stato, tutti i possibili caratteri di input. La procedura Transition implementa l'algoritmo delineato.

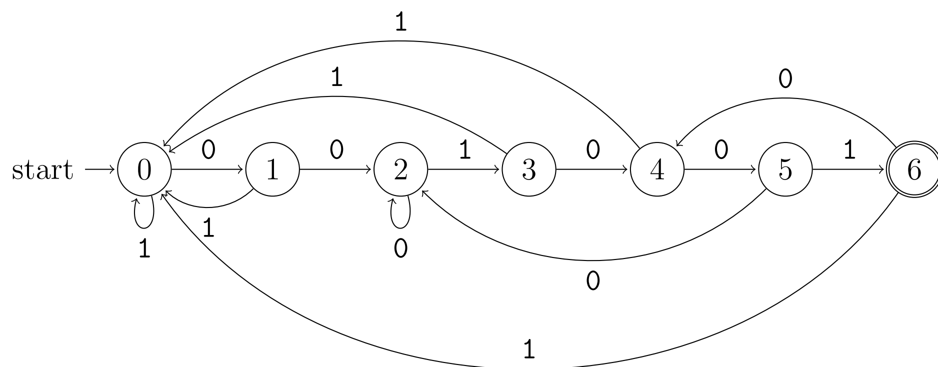


Figure 4.10: Grafo di transizione completo relativo all'automa dell'esempio 4.66.

```
TRANSITION(P, Σ)
1  for j ← 0 to |P|
2      for all x ∈ Σ
3          z ← P[0..j − 1]x
4          k ← min(j + 1, m)
5          while k ≥ 0 and P[0..k − 1] ≠ z[j + 1 − k..j]
6              k ← k − 1
7          δ(j, x) ← k
8  return δ
```

Il costo computazionale dell'algoritmo TRANSITION è alquanto elevato: come nel caso dell'algoritmo BEST-PREFIX, abbiamo tre cicli annidati, di lunghezza rispettivamente $m+1$, $|\Sigma|$ e $\min(j+1,m)$, per $k = 0, 1, \ldots, m$; inoltre il test del ciclo più interno richiede tempo proporzionale a k, $k = 0, 1, \ldots, \min(j+1, m)$. Nel caso più sfavorevole, il costo di esecuzione è dato dunque dalla seguente somma:

$$
\begin{aligned}
\Delta(m) &= \sum_{j=0}^{m} \sum_{x \in \Sigma} \sum_{k=0}^{\min(j+1,m)} O(k) \\
&= \sum_{x \in \Sigma} \sum_{j=0}^{m} O(\min(j+1,m)^2) \\
&= |\Sigma| O(m^3)
\end{aligned}
$$

Anche dal punto di vista dello spazio, la memorizzazione della funzione δ completa risulta abbastanza onerosa. Si tratta infatti di memorizzare una tabella di m righe e $|\Sigma|$ colonne, il che implica spazio $\Theta(m|\Sigma|)$. Si possono tuttavia fornire soluzioni migliori sia in termini di tempo che di spazio. Studieremo una tale soluzione nell'Esercizio 4.70.

Avendo a disposizione la funzione di transizione, la scrittura di un algoritmo che "simula" il comportamento dell'automa a stati finiti diviene un semplice esercizio. Lo pseudo-codice è riportato in figura 4.11.

Una digressione tecnica è però necessaria per ragioni di correttezza. Gli automi a stati finiti sono modelli che non prevedono uso di una memoria di lettura e scrittura. Una semplice conseguenza di questo fatto, che qui ci interessa, è che in un tale modello è impossibile "contare" i caratteri di un testo, proprio perché non possiamo utilizzare una variabile contatore. Per

un automa riconoscitore è dunqe impossibile dichiare le posizioni alle quali si è verificato un match. Il codice di figura 4.11 è dunque strettamente più potente degli automi simulati. L'esercizio 4.73 chiede di modificare il codice del simulatore in modo che rappresenti l'automa in senso stretto.

```
ASFD-MATCHER(P, T, Σ)
1  δ ← TRANSITION(P, Σ)
2  m ← |P|
3  q ← i ← 0
4  while not end-of-input
5       x ← next input character
6       i ← i + 1
7       q ← δ(q, x)
8       if q = |P|
9            Riporta un'occorrenza alla posizione i − |P|
```

Figure 4.11: Simulatore di un generico automa per il pattern matching in un testo.

Esercizio 4.67. Si definisca l'ASFD che riconosce le stringhe su $\mathcal{B}$ costituite da uno o più `0` seguiti da uno o più `1`.

Esercizio 4.68. Si definisca l'ASFD che riconosce le stringhe su $\mathcal{B}$ costituite da una o più "copie" della stringa `10010`.

Esercizio 4.69. Sia $\Sigma = \mathcal{D} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Si disegni il grafo di transizione completo dell'ASFD che riconosce le occorrenze di $P = \mathtt{ACCGA}$ in una qualunque stringa (testo) su Σ.

Esercizio 4.70. Abbiamo visto che, se l'alfabeto Σ ha elevata cardinalità, memorizzare la funzione di transizione dell'automa riconoscitore mediante una tabella esplicita può essere molto oneroso (si pensi al caso di un pattern di qualche migliaio di caratteri sull'alfabeto Unicode). Per migliorare la situazione, osserviamo innanzitutto come la parte onerosa sia costituita dalla apparente necessità di dover specificare il comportamento dell'automa in corrispondenza di tutti i possibili mismatch, quindi per tutti i simboli

dell'alfabeto meno uno. Una possibile alternativa consiste invece nel considerare due sole possibilità: match e mismatch. Più precisamente, se nel generico stato j si verifica un mismatch fra il carattere di input x e P_j, la funzione rimanda ad uno stato k_j, *indipendente da* x, così determinato:

$$k_j = \delta(j, x) = \max\{k < j | P_{0..k-1} \text{ è suffisso di } P_{0..j-1}\}.$$

(si notino e si rifletta bene sulle due differenze con il valore $\delta(j, x)$ previsto dal Teorema 4.65). In termini di grafo di transizione, ciò si traduce nel fatto che, dal generico stato $j < m$, usciranno due sole frecce, una etichettata con P_j e l'altra con un generico (meta)carattere $\bar{P}_j$, che indica un qualunque carattere in $\Sigma - \{P_j\}$. Il carattere x viene poi *riletto* nel nuovo stato. Si verifichi che in questo modo l'automa riconosce comunque correttamente le occorrenze del pattern nel testo e che, indipendentemente dalla cardinalità di Σ, la sua complessità in spazio è $O(m)$.

Esercizio 4.71. Si ripeta la costruzione dell'Esercizio 4.69 alla luce delle considerazioni espresse nell'Esercizio 4.70.

Esercizio 4.72. Si disegni il grafo di transizione dell'automa che riconosce le occorrenze del pattern $P =$ `href` in un testo sull'alfabeto ASCII. Tale automa potrebbe essere utilizzato per localizzare tutti gli hyperlink contenuti in un file HTML.

Esercizio 4.73. Un automa a stati finiti non può contare ma, in ogni stato, può compiere azioni come stampare semplici messaggi. Se dunque un automa riconoscitore non può direttamente riportare le posizioni di match del pattern nel testo, può comunque comunicare informazione che, con opportuna post-elaborazione, consente di ricostruire le posizioni di match. Si modifichi il codice di figura 4.11 in tal senso, in modo cioè che simuli l'automa in modo stretto e che comunichi l'informazione necessaria per risalire alle posizioni di match.

4.2.4 Algoritmo di Aho e Corasick

La costruzione di un automa che riconosce un insieme finito $\mathcal{P}$ pattern su un alfabeto Σ è relativamente semplice ed estende in modo naturale il caso di un singolo pattern. Distinguiamo anche qui due fasi nella definizione del grafo di transizione. Il grafo costruito nella prima fase è un *trie* su Σ[3], una

[3]Un trie su un alfabeto Σ è un albero i cui archi sono etichettati con simboli di Σ.

struttura che generalizza, al caso di più pattern, le strutture lineari di figura 4.7 e 4.9.

Sia T un trie su Σ e sia q un nodo di T. Definiamo *path label* $\ell(q)$ di q la stringa ottenuta dalla concatenazione delle etichette sugli archi dell'unico cammino dalla radice di T a q (nell'ordine). Dato un insieme $\mathcal{P}$ di pattern, il corrispondente trie $T_{\mathcal{P}}$ costruito nella fase I ha tanti nodi quanti sono i possibili *prefissi distinti* dei pattern in $\mathcal{P}$, ed ogni nodo q è associato al prefisso che ne costituisce la path label.

Esempio 4.74. Consideriamo l'insieme

$$\mathcal{P} = \{\texttt{algebra}, \texttt{algoritmi}, \texttt{economia}, \texttt{geologia}, \texttt{geometria}\}.$$

L'automa risultante dalla fase I è mostrato in figura 4.12.

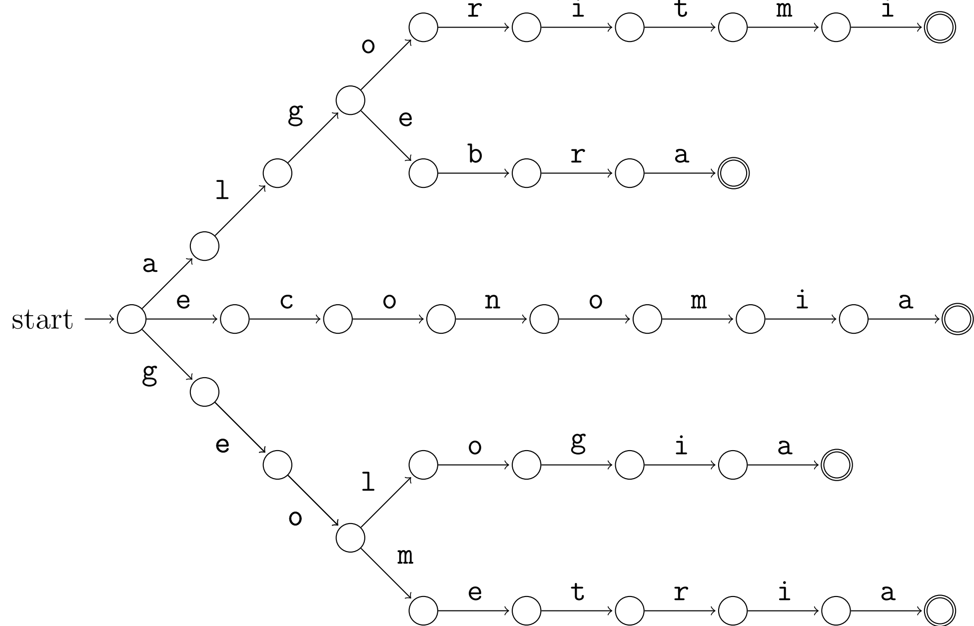


Figure 4.12: Trie relativo all'insieme di pattern S dell'esempio 4.74. Per migliorare la leggibilità, l'albero è stato disegnato in orizzontale, anziché in verticale. Inoltre, gli stati dell'automa non sono stati esplicitamente nominati.

È utile avere anche una notazione che ci permetta di passare da un prefisso α allo stato corrispondente (cioè quello stato che ha α come path label).

Useremo dunque la scrittura s_α per indicare lo stato dell'automa tale che $\ell(s_\alpha) = \alpha$. Si noti che, nel caso di singolo pattern e secondo la convenzione che abbiamo (implicitamente) adottato per numerare gli stati, vale $s_\alpha = |\alpha|$. Naturalmente, questa regola di numerazione non può valere nel caso di più pattern; tuttavia, anche in quest'ultimo caso assumeremo $s_\epsilon = 0$.

Lo pseudocodice di figura 4.13 descrive in dettaglio la fase di costruzione del trie (fase I). Si noti, in particolare, che alcuni stati vengono resi terminali; come già nel caso di singolo pattern (cf. sezione 4.2.3), in tali stati l'automa emetterà un messaggio di riconoscimento senza terminare l'esecuzione. La funzione Ahead, utilizzata nello pseudocodice, è definita come segue:

$$\textsc{Ahead}(q,x) = \begin{cases} r & \text{se nel trie esiste l'arco } (q,r) \text{ con etichetta } x \\ \texttt{nil} & \text{altrimenti} \end{cases}$$

L'esercizio 4.82 chiede di definire un'implementazione della funzione Ahead.

```
Build-Trie(𝒫)
     // New-State() restituisce un nuovo stato (nodo del trie)
 1   root = New-State()
 2   for each P in 𝒫                        // I pattern sono elaborati in sequenza
 3       q = root
 4       for i = 0 to |P| − 1
 5           x = P[i]
 6           r = Ahead(q, x)
 7           if r = nil
 8               r = New-State()
                 // Inserisce l'arco q → r con etichetta x
 9               New-Edge(q, r, x)          // Equivale a porre δ(q, x) = r
10           q = r
11       accepting[q] = True                // q stato di accettazione
```

Figure 4.13: Fase di costruzione del trie per l'algoritmo di Aho-Corasick.

Esempio 4.75. (Comportamento delle funzioni Ahead e Build-Trie) Si consideri l'insieme di pattern dell'esempio 4.74. Dopo l'elaborazione del pat-

tern `algebra`, l'automa si presenta come nella figura 4.14 (in alto). Se il pattern successivo è `algoritmi`, le successive 4 chiamate della funzione Ahead restituiranno i seguenti valori:

$$1, 2, 3, \texttt{nil},$$

e l'automa (parziale) costruito dalla funzione Build-Trie si presenterà come in figura 4.14 (in basso).

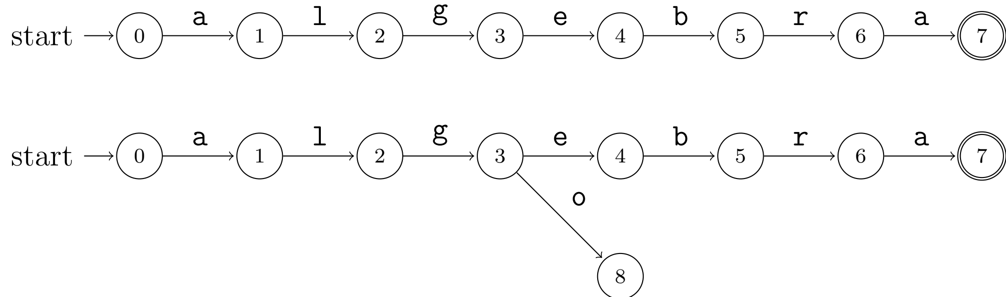


Figure 4.14: Momenti della costruzione (fase I) del grafo di transizione relativo all'automa dell'esempio 4.74. Per illustrare il comportamento della funzione Ahead è stato necessario nominare gli stati

La fase II procede al completamento dell'automa, inserendo tutte le altre transizioni, cioè quelle che fanno seguito ad un mismatch o al rilevamento dell'occorrenza di un pattern.

Riconsideriamo il caso del riconoscimento di un singolo pattern P (Sezione 4.2.3). Sappiamo che, in corrispondenza di uno stato $q < m$ esiste un solo arco "in avanti", etichettato con $P[q]$; per ogni altro simbolo $x \neq P[q]$, esiste una transizione che porta nello stato r dove r è la lunghezza del bordo più lungo di $P[0..q-1]x$. Analogamente, nel caso di più pattern, l'automa deve transitare dallo stato s_α, associato al prefisso α, allo stato s_β, associato a β, dove β è il suffisso più lungo αx che è anche prefisso di uno dei pattern di $\mathcal{P}$. Si noti che β può essere prefisso comune ad uno o più pattern. Ad esempio, nel caso di figura 4.12, dallo stato $s_{\texttt{alge}}$, su input `o` dovremmo transitare nello stato $s_{\texttt{geo}}$, che corrisponde al prefisso `geo` comune ai pattern `geologia` e `geometria`.

Per formalizzare i concetti sopra esposti, fissato l'insieme $\mathcal{P}$ di pattern, introduciamo la funzione $\textsc{suffix} : \Sigma^* \to \Sigma^*$, definita nel modo seguente:

$$\textsc{suffix}(\alpha) = \max_{P \in \mathcal{P}} \left\{ |\beta| : \exists i, 0 \leq i \leq |P|, \beta = \alpha_{i..|P|-1} \wedge \beta = P_{0..|P|-i-1} \right\} \tag{4.3}$$

A parole, $\textsc{suffix}(\alpha)$ è il suffisso più lungo di α che è anche un prefisso di uno o più pattern di $\mathcal{P}$. Ovviamente, se la stringa α è essa stessa prefisso di uno dei pattern risulta $\textsc{suffix}(\alpha) = \alpha$. Ad esempio, con riferimento all'insieme di pattern già considerati nell'esempio 4.74, abbiamo $\textsc{suffix}(\texttt{geo}) = \texttt{geo}$. Si noti che, volendo rendere evidente la dipendenza da $\mathcal{P}$, dovremmo più correttamente scrivere $\textsc{suffix}_{\mathcal{P}}()$, anziché $\textsc{suffix}()$. Si noti anche che la definizione (4.3), includendo il caso $i = |P|$, tiene conto della eventualità che $\textsc{suffix}(\alpha) = \epsilon$, cioè che nessun suffisso non vuoto di α sia prefisso di uno dei pattern.

Utilizzando la (4.3) possiamo esprimere la funzione di transizione $\delta(q, x)$ dell'automa nel modo seguente:

$$\delta(q, x) = s_{\textsc{suffix}(\ell(q)x)}$$

dove $\ell(q)x$ è la concatenzione di $\ell(q)$ con il carattere x.

La funzione così definita tiene conto anche delle transizioni già inserite nella fase 1; infatti, se $\textsc{Ahead}(q, x) = r \neq \texttt{nil}$, allora $\ell(r) = \ell(q)x$ è un prefisso di almeno uno dei pattern e dunque

$$s_{\textsc{suffix}(\ell(q)x)} = s_{\ell(q)x} = s_{\ell(r)} = r.$$

Il calcolo di $\textsc{suffix}(\alpha)$ può essere effettuato con l'aiuto del solo trie costruito nella fase I utilizzando la definizione (4.3): si considerano cioè suffissi via via più corti di α e, per ogni suffisso β così individuato, si "ripercorre" l'automa a partire dallo stato iniziale utilizzando solamente la funzione Ahead. Lo pseudocodice è illustrato in Figura 4.15. L'esercizio 4.81 chiede di valutare la complessità di questa soluzione.

Come già osservato nella Sezione 4.2.3, memorizzare esplicitamente ogni possibile transizione richiede spazio proporzionale a $O(|\Sigma|m)$, dove in questo caso m è la lunghezza complessiva dell'insieme di tutti i pattern[4]. Si noti che lo spazio occupato dalla funzione di transizione è "distribuito" fra tutti i nodi;

[4]Se ci sono molti prefissi comuni a più pattern, il numero di stati potrebbe essere significativamente minore. Tuttavia, nel worst-case, esso è proprio $m + 1$.

```
SUFFIX(α)
1  l = |α|
2  for i = 0 to l − 1
3        q = 0
4        j = i
5        while j < l and AHEAD(q, α[j]) ≠ nil
6              q = AHEAD(q, α[j])
7        if j == l
8              return α[i..l − 1]
9  return ϵ
```

Figure 4.15: Pseudocodice per la funzione SUFFIX().

abbiamo cioè $O(m)$ nodi, ognuno dei quali richiede spazio $\Theta(|\Sigma|)$. L'esercizio 4.70 suggerisce la possibilità di ridurre l'occupazione di spazio a $O(m)$ nel caso di singolo pattern, pagando un prezzo (in generale modesto) nel tempo richiesto per il riconoscimento del pattern nel testo. Nel caso di alfabeti di elevata cardinalità, ciò risulta decisamente vantaggioso. L'algoritmo di Aho-Corasick estende al caso di un insieme di pattern l'idea suggerita nell'esercizio 4.70 per la costruzione di una funzione di transizione più economica in spazio. Di seguito sviluppiamo per esteso tale idea.

Consideriamo un generico stato q. Da q escono un certo numero di archi inseriti durante la fase I (eventualmente nessuno, nel caso di stati terminali). Tali archi sono etichettati da quei simboli x tali che $\text{AHEAD}(q, x) \neq \texttt{nil}$. Consideriamo ora un simbolo z per il quale vale $\text{AHEAD}(q, z) = \texttt{nil}$. In tal caso decidiamo che l'automa non utilizzi il simbolo z (o almeno non subito) per determinare il prossimo stato: z viene in qualche modo "bufferizzato" per essere riutilizzato in un secondo momento[5]. Lo stato r successivo viene quindi deciso solo in funzione di q e, dunque, sulla base della path label $\ell(q)$ di q, che descrive gli ultimi simboli letti prima di z. Introduciamo la notazione $\text{FAILURE}(q)$ per individuare tale stato r che dipende da q ma non da z. Eccezion fatta per il carattere z di input, valgono comunque tutte le con-

[5]In senso stretto, un automa a stati finiti non ha la capacità di bufferizzare l'input per poterlo riconsiderare più volte. Questa capacità rende "l'automa" che stiamo definendo più potente di un ASFD.

siderazioni fatte in precedenza e dunque possiamo affermare che $\text{FAILURE}(q)$ dovrà indicare quello stato r associato al suffisso più lungo di $\ell(q)$ che è anche prefisso di almeno un pattern di $\mathcal{P}$. C'è però una non secondaria differenza fra i due casi; non è infatti corretto porre $\text{FAILURE}(q) = s_{\text{SUFFIX}(\ell(q))}$. Questo perché $\ell(q)$, in quanto path label di un nodo del trie, è prefisso di almeno un pattern e dunque è un "punto fisso" di SUFFIX; vale cioè $\text{SUFFIX}(\ell(q)) = \ell(q)$. Questo ovviamente non va bene perché porterebbe a concludere che $\text{FAILURE}(q) = s_{\text{SUFFIX}(\ell(q))} = s_{\ell(q)} = q$, provocando un ciclo infinito nell'automa riconoscitore. Dobbiamo quindi escludere il primo carattere di $\ell(q)$ e dunque considerarne solo prefissi propri (incluso però ϵ). Dobbiamo cioè porre:

$$\text{FAILURE}(q) = s_{\text{SUFFIX}(\ell(q)_{1..|\ell(q)-1|})} \tag{4.4}$$

Per semplicità di notazione, introduciamo la funzione PROPERSUFFIX così definita:

$$\text{PROPERSUFFIX}(\alpha) = \text{SUFFIX}(\alpha_{1..|\alpha|-1})$$

Si noti che includiamo anche la stringa vuota fra i prefissi propri di una generica stringa α non vuota (escludiamo solo α stessa).

Esempio 4.76. Con riferimento al trie di figura 4.12, se $q = s_{\texttt{alge}}$ allora $r = \text{FAILURE}(q) = s_{\texttt{ge}}$ perché `ge` è il più lungo suffisso di `alge` che è prefisso di due pattern (`geologia` e `geometria`). Se $q = s_{\texttt{algo}}$ allora $r = \text{FAILURE}(q) = 0$ perché nessun suffisso proprio di `algo` è prefisso di un pattern.

Nello stato $r = \text{FAILURE}(q)$ l'automa andrà a "riconsiderare" il carattere di input z, bufferizzato in precedenza, e procederà esattamente come nello stato q: se vale $\text{AHEAD}(r, z) \neq \texttt{nil}$, l'automa transiterà proprio nello stato $\text{AHEAD}(r, z)$, altrimenti in $t = \text{FAILURE}(r)$. Procedendo in questo modo è possibile che, al verificarsi di un mismatch nello stato q, l'automa giunga a transitare nello stato 0 in al più $|\ell(q)|$ passi. Si noti, infatti, che, per ogni stato $q \neq 0$, vale $\ell(\text{FAILURE}(q)) < \ell(q)$, cioè lo stato $\text{FAILURE}(q)$ è "più vicino" alla radice del trie di quanto non lo sia q. Questo semplicemente perché $\ell(\text{FAILURE}(q))$ è un suffisso di $\ell(q)$ che non può essere $\ell(q)$ stesso. Si noti poi che $\text{FAILURE}(0) = 0$.

Continuando con l'esempio di figura 4.12, se nello stato $r = s_{\texttt{ge}}$ l'input fosse il carattere `o`, varrebbe $\text{AHEAD}(r, \texttt{o}) = s_{\texttt{geo}}$. Se invece l'input bufferizzato `z` fosse diverso da `o`, cioè se valesse $\text{AHEAD}(r, \texttt{z}) = \texttt{nil}$, riapplicando

la funzione FAILURE(r) l'automa transiterebbe nello stato $t = s_{\mathtt{e}}$. In t, riconsiderando l'input z bufferizzato (e non ancora "consumato") l'automa transiterebbe in $s_{\mathtt{ec}}$ se $z = \mathtt{c}$, altrimenti in FAILURE(t) = 0.

Utilizzando le funzioni AHEAD e FAILURE possiamo realizzare la funzione di transizione dell'automa (memorizzato nel trie T) nel modo descritto dal seguente pseudocodice, che riflette le argomentazioni sviluppate sopra. L'algoritmo suppone che inizialmente l'automa si trovi nello stato 0. Le funzioni AHEAD e FAILURE lavorano sul trie T (che non è indicato esplicitamente).

```
AC-T_T(q, x)
1  while q ≠ nil and AHEAD(q, x) = nil
2       q = FAILURE(q)
3  if q = nil
4       return 0
5  else return AHEAD(q, x)
```

La funzione FAILURE(q) può essere memorizzata utilizzando spazio $\Theta(m)$, dove $m = O(\sum_{P \in \mathcal{P}} |P|)$ è il numero di stati dell'automa. Come già nel caso di $\delta(q, x)$, la memorizzazione è "distribuita" negli stati dell'automa; tuttavia qui la richiesta di spazio è costante per ogni stato. Il calcolo preliminare di tutti i valori FAILURE(q) può essere fatto ricorrendo alla definizione (4.4) e, dunque, all'algoritmo per il calcolo di SUFFIX (figura 4.15). Nel seguito forniamo un secondo che si basa sul seguente semplice risultato.

Teorema 4.77. Sia q un generico stato dell'automa diverso dallo stato iniziale e dagli stati ad esso direttamente collegati (cioè i figli dello stato iniziale nel trie). Sia $\alpha = \ell(q)$ la path label di q e sia z l'ultimo carattere di α: $\alpha = \beta z$. Posto δ = PROPERSUFFIX(β) e, dunque $p = s_\delta$ = FAILURE(s_β), se risulta AHEAD(p, z) $= r \neq$ **nil** allora vale FAILURE(q) $= r$. La figura 4.16 visualizza la situazione.

Dim. Dalle ipotesi su q necessariamente vale $|\alpha| \geq 2$ e dunque $\beta \neq \epsilon$. Per definizione δ è il più lungo suffisso proprio di β che è anche prefisso di almeno un pattern. Si noti che δ può essere la stringa vuota; in tal caso varrebbe $q = 0$. Ora, se risulta AHEAD(p, z) $\neq$ **nil**, allora anche $\gamma = \delta z$ è prefisso di almeno uno dei pattern; ma γ è anche suffisso proprio di α e necessariamente anche il più lungo fra quelli che sono prefisso di almeno un pattern, altrimenti

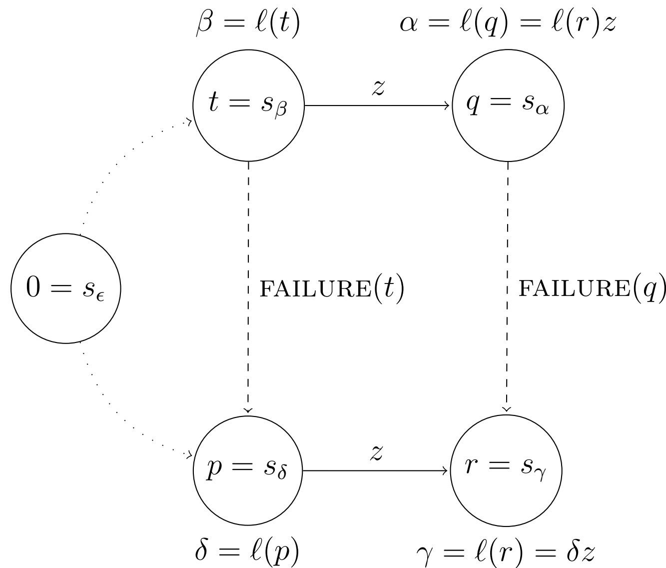


Figure 4.16: Calcolo di FAILURE(q) nelle ipotesi del teorema 4.77

verrebbe contraddetta l'assunzione che $\delta =$ PROPERSUFFIX(β) (si veda anche la figura 4.17). Ne consegue che $s_\gamma =$ AHEAD(p, z) è il valore corretto di FAILURE(q). □

Se q è uno stato non considerato dal teorema 4.77, cioè è lo stato iniziale oppure un suo discendente diretto nel trie, allora evidentemente vale FAILURE(q) $= 0$. Che cosa accade invece se, con riferimento alle posizioni fatte nel teorema, AHEAD(t, z) $=$ **nil**? In tal caso $\gamma = \delta z$ non è suffisso di α. È tuttavia ancora possibile che "accorciando il suffisso" di β che abbiamo

$\alpha =$ | β | z |

$\gamma =$ | δ | z |

Figure 4.17: Nell'ipotesi del teorema 4.77, $\alpha = \beta z$ e $\delta =$ PROPERSUFFIX(β) e dunque $\gamma = \delta z =$ PROPERSUFFIX(α). Se fosse PROPERSUFFIX(α) $= \zeta$, con $|\zeta| > |\gamma|$, allora verrebbe contraddetta l'ipotesi che $\delta =$ PROPERSUFFIX(β).

considerato, e cioè prendendo un suffisso proprio di δ, si ottenga, dopo la concatenazione con z, un prefisso di uno dei pattern. Al riguardo si consideri il seguente esempio.

Esempio 4.78. Sia $\mathcal{P}$ ={palestra,alessia,esca,stoffa} l'insieme di pattern e sia q lo stato associato al prefisso $\alpha =$ palest. Con le posizioni del teorema 4.77, risulta $\beta =$ pales, $z =$ t e $\delta =$ PROPERSUFFIX(pales) $=$ ales, perché ales è prefisso del pattern alessia. Tuttavia $\gamma = \delta z =$ alest non è prefisso di alcun pattern. Se ora consideriamo il suffisso PROPERSUFFIX(δ) abbiamo PROPERSUFFIX(ales) $=$ es, perché es è prefisso del pattern esca. Anche in questo caso, però, se concateniamo $z =$ t a es otteniamo la stringa est che non è suffisso di alcun pattern (equivalentemente, potremmo dire che nello stato $s_{\texttt{es}}$ risulta AHEAD($s_{\texttt{es}}, z$) $=$ **nil**). Se ora, con un ulteriore "passaggio", consideriamo PROPERSUFFIX(es) otteniamo s che, concatenato con il carattere $z =$ t, produce la stringa st che è prefisso del pattern stoffa. Vale dunque FAILURE(q) $= s_{\texttt{st}}$.

Le considerazioni e l'esempio precedenti suggeriscono quindi che si proceda ricorsivamente nel calcolo del valore di FAILURE(q) fino ad arrivare ad uno stato p tale che AHEAD(p, z) $\neq$ **nil**, ovvero si giunga allo stato iniziale. L'algoritmo completo è mostrato con lo pseudo-codice di figura 4.18. L'algoritmo calcola il valore FAILURE(q) di ogni stato dell'automa e lo memorizza nel corrispondente nodo del trie (per il successivo uso, quando l'automa verrà impiegato per riconoscere i pattern in un testo). Si noti che l'ordine con cui vengono precalcolati i valori di FAILURE() è determinato da una visita in ampiezza del trie. La visita in ampiezza garantisce che, se il calcolo del valore di FAILURE(q) dipende dal valore di FAILURE(t), quest'ultimo è già stato calcolato. Abbiamo già osservato, infatti, che la lunghezza di $\ell(t)$ è necessariamente minore di $\ell(q)$ ($\ell(t)$ è un suffisso proprio di $\ell(q)$) e questo implica che t è "più vicino" alla radice del trie rispetto a q).

Si noti ancora che, nel codice di figura 4.18, la scrittura FAILURE[q] $\leftarrow$ r (cioè un assegnamento in cui il simbolo di funzione compare a sinistra dell'operatore $\leftarrow$) va quindi intesa come la memorizzazione del valore di FAILURE() nello stato q. Un assegnamento del tipo $r =$ FAILURE[q] (dove il simbolo di funzione compare invece a destra) corrisponde invece ad un *accesso* alla variabile che memorizza il valore pre-calcolato FAILURE(q).

Le righe 16 e 17 richiedono forse qualche spiegazione aggiuntiva. Si ricordi che, nella fase di costruzione del trie (algoritmo BUILD-TRIE di pagina 4-123), ogni stato la cui path label coincide con uno dei pattern viene marcato

```
MAKE-FAILURE(T)
 1  FAILURE[0] = nil // Non si può andare oltre lo stato iniziale
    // Per la visita del trie utilizziamo una coda semplice
    // inizializzata con i diretti discendenti della radice del trie
 2  Q = NEW QUEUE()
 3  for each q discendente diretto di 0 in T
 4       FAILURE[q] = 0
 5       Q.ENQUEUE(q)
 6  while not Q.EMPTY()
 7       s = Q.DEQUEUE()
 8       for each z ∈ Σ s.t. AHEAD(s, z) ≠ nil
 9            q = AHEAD(s, z)
10            t = FAILURE(s)
11            while t ≠ nil and AHEAD(t, z) == nil
12                 t = FAILURE(t)
13            if t == nil
14                 FAILURE[q] = 0
15            else FAILURE[q] = AHEAD(t, z)
16                 if accepting[q]
17                      accepting[FAILURE[q]] = True
```

Figure 4.18: Un algoritmo più efficiente per la definizione della funzione AHEAD.

come terminale. Questo accade anche nel caso in cui uno dei pattern sia un prefisso di un altro pattern. Ad esempio, se $S = \{\texttt{ozi}, \texttt{oziare}\}$, il trie risultante dalla fase I apparirà come in figura 4.19.

Si consideri ora l'insieme $S = \{\texttt{oziare}, \texttt{zia}\}$. Al termine della fase I il trie è quello di figura 4.20. Lo stato 5 non è stato reso terminale perchè la sua path label (`amore`) non corrisponde ad alcun pattern di S. Tuttavia, il suffisso più lungo di `amore` (la stringa `more`) è un prefisso di S che coincide con un pattern. Ne consegue che FAILURE(5) è terminale e che nello stato 5 l'automa ha appena riconosciuto il pattern associato proprio a FAILURE(5). Dunque lo stato 5 deve correttamente essere marcato come terminale.

La figura 4.21 illustra il grafo di transizione completo per l'automa che riconosce l'insieme di pattern dell'esempio 4.74. Le linee disegnate in rosso

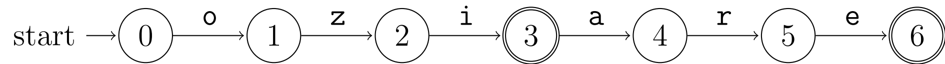


Figure 4.19: Trie relativo a $\mathcal{P} = \{\texttt{oziare}, \texttt{ozi}\}$. Il nodo 3 è correttamente etichettato come terminale.

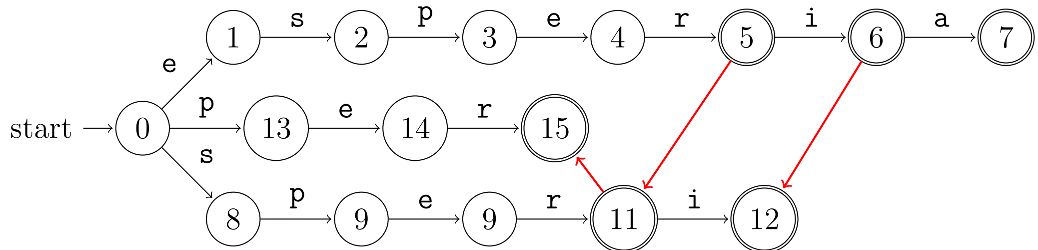


Figure 4.20: Trie relativo a $\mathcal{P} = \{\texttt{esperia}, \texttt{speri}, \texttt{per}\}$. Gli archi in rosso si riferiscono alla funzione Failure (non sono stati riportati tutti). Il nodo 11 deve essere marcato come terminale perchè Failure(11) = 15 e 15 è terminale. A sua volta, il nodo 5 deve essere marcato come terminale perché Failure(5) = 11. Si noti qui l'importanza della visita in ampiezza del trie. Anche il nodo 6 è marcato come terminale perché Failure(6) è terminale.

si riferiscono alla funzione FAILURE. Per tutti gli stati per i quali la transizione "all'indietro" non è mostrata, si deve intendere che essa porti allo stato iniziale.

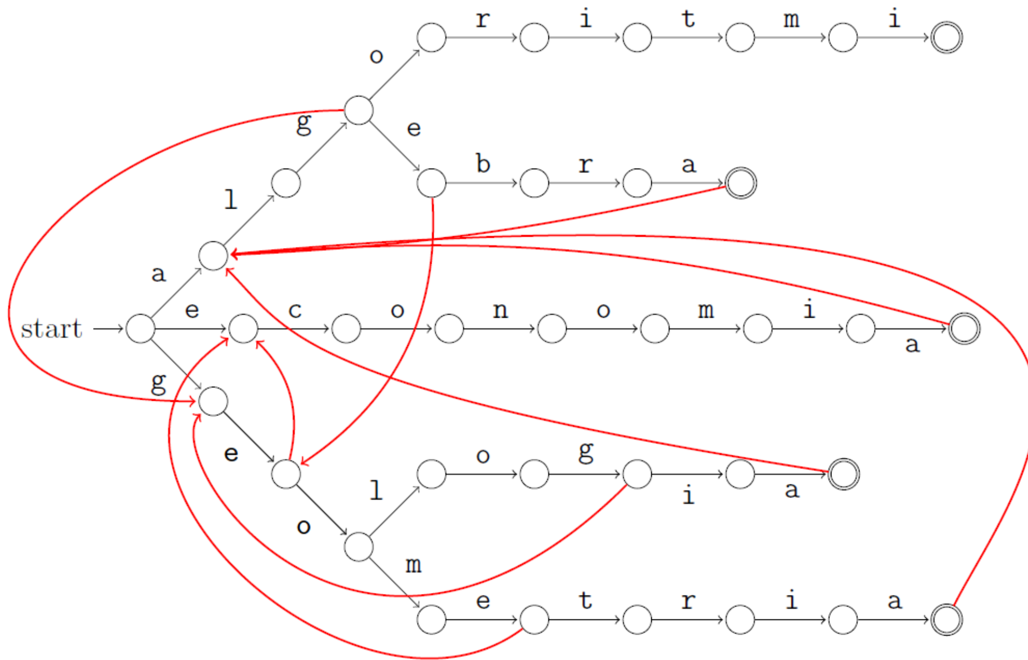


Figure 4.21: Automa che riconosce l'insieme di pattern $\mathcal{P}$ dell'esempio 4.74. Come per la figura 4.20, gli archi in rosso si riferiscono alla funzione FAILURE.

A questo punto, lo pseudocodice dell'algoritmo di Aho e Corasick risulta molto semplice. Esso utilizza le tre funzioni già definite: (1) per la costruzione del trie, (2) per la costruzione della funzione FAILURE, (3) per la definizione delle transizioni.

```
AHO-CORASICK(𝒫, T)
1  𝒯 = BUILD-TRIE(𝒫)
2  MAKE-FAILURE(𝒯)
3  q = 0
4  n = |T|
5  for i = 0 to n − 1
6        q = AC-T_𝒯(q, T[i])
7        if accepting[q]
8              Riporta un'occorrenza del pattern ℓ(q) alla pos. i − |ℓ(q)| + 1
```

Esercizio 4.79. Si costruiscano gli automi corrispondenti agli insiemi $S_1 = \{\texttt{he, she, his, hers}\}$ e $S_2 = \{\texttt{gtagct}, \texttt{tag}, \texttt{agct}, \texttt{ctagt}\}$.

Esercizio 4.80. Si implementi nel linguaggio preferito l'algoritmo Build-Trie, definendo un'opportuna struttura dati per il trie.

Esercizio 4.81. Si dimostri che la complessità dello algoritmo 4.15 è $O(|\alpha|^2)$. Si esibisca una stringa α che comporta un numero di confronti proporzionale a $|\alpha|^2$.

Esercizio 4.82. Si implementi la funzione Ahead facendo riferimento alla realizzazione concreta del trie fornita nell'Esercizio 4.80.

Esercizio 4.83. Utilizzando i risultati nel testo e negli esercizi 4.80 e 4.82, si dimostri che il costo in spazio dell'algoritmo Aho-Corasick è $O(m)$, dove m è la lunghezza complessiva dei pattern.

Esercizio 4.84. Si dimostri che la complessità dell'algoritmo di figura 4.18 è $O\left(\left(\sum_{P\in\mathcal{P}} |P|\right)\left(\max_{P\in\mathcal{P}} |P|\right)\right)$.

4.3 Pattern specificati mediante espressioni regolari

Obiettivo di questa sezione è di presentare il processo di progettazione di un algoritmo che sia in grado di riconoscere lo stesso linguaggio (cioè lo stesso insieme di stringhe) definito mediante una generica espressione regolare E, un caso molto frequente in pratica e pertanto di grande interesse applicativo. Un risultato fondamentale nella teoria dei linguaggi formali è l'equivalenza

fra espressioni regolari e automi finiti. Equivalenza vuol proprio dire che ogni linguaggio definibile mediante un'espressione regolare E è anche riconoscibile da un opportuno automa a stati finiti deterministico D_E. Idealmente, quindi, l'algoritmo cui facevamo riferimento poco sopra è proprio un'automa finito, o meglio, una *famiglia* di automi, ognuno specifico per una data espressione regolare.

Il passaggio diretto da espressioni regolari ad automi deterministici non è però agevole. Il modo migliore per rendersi conto della verità di questa afffermazione è di provare ad operare tale trasformazione per espressioni regolari diverse (si veda l'esercizio **??**). Il problema che "sembra" emergere è che ogni espressione regolare vada trattata, e trasformata, in modo ad-hoc.

Come vedremo, il problema viene superato utilizzando un "modello di calcolo intermedio" rappresentato dagli automi finiti *non deterministici*. In un tale modello, esistono stati q (almeno uno) in cui lo stato successivo non è univocamente determinato da q e dal prossimo carattere in input; da q l'automa può cioè transitare indifferentemente in almeno due stati distinti, diciamo q' e q''[6]. Modelli di calcolo non deterministici giocano un ruolo importante nel campo dell'Informatica teorica ma non sono "operativi". Limitandoci al caso degli automi finiti, che qui ci interessa, vedremo che essi consentono di risolvere proprio il problema che ci siamo posti, in quanto:

1. data un'espressione regolare E esiste un processo algoritmico che consente di definire un automa a stati finiti non deterministico N_E equivalente ad E;

2. dato un automa non deterministico N_E esiste un procedimento algoritmico che consente di definire un automa deterministico D_E ad esso equivalente (cioè che riconosce lo stesso insieme di stringhe);

ovvero:

2. esiste un algoritmo A che, data la descrizione di un qualunque automa non deterministico N_E e una stringa α per N_E, fornisce lo stesso output che fornirebbe N_E su input α.

Noi seguiremo proprio questa seconda strada per arrivare al riconoscimento di pattern specificati mediante espressioni regolari.

[6]Si noti bene che alla scelta non è associata alcuna nozione di probabilità. L'automa non sceglie q' o q'' sulla base, poniamo, del risultato del lancio di una moneta.

4.3.1 Automi non deterministici con ϵ transizioni

Gli automi non deterministici che andremo a considerare saranno solo di un particolare tipo, precisamente quello che (come vedremo) risulta dalla trasformazione di espressioni regolari. Diamo la seguente definizione formale:

Definizione 4.85. Un automa a stati finiti non deterministico $\mathcal{A}$ è una quintupla

$$\mathcal{A} = \{\Sigma, Q, q_0, Q_f, \delta\}$$

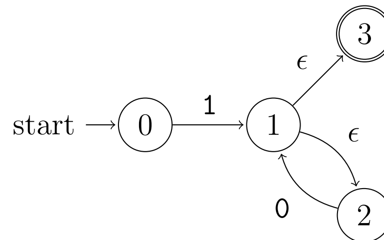
in cui Σ, Q, q_0 e Q_f sono definiti come nel caso degli automi deterministici, mentre

$$\delta : \delta' \cup \delta'', \qquad \delta' : Q_1 \times \Sigma \rightarrow Q, \qquad \delta'' : Q_2 \times \{\epsilon\} \rightarrow 2^Q$$

dove $Q_1, Q_2 \subseteq Q$ e $Q_1 \cap Q_2 = \emptyset$.

In altri termini, gli stati di Q sono suddivisi in due sottoinsiemi disgiunti[7]: dagli stati di Q_1 l'automa procede solo leggendo un simbolo dal dispositivo di input, mentre dagli stati di Q_2 procede "leggendo ϵ" e, dunque, senza in realtà consumare simboli in input. Chiameremo *deterministico* uno stato $q \in Q_1$ perché $|\delta(q, x)| = |\delta'(q, x)| \leq 1$; in termini di grafo di transizione, possiamo cioè affermare che da uno stato $q \in Q_1$ esce al più una transizione etichettata con uno stesso simbolo dell'alfabeto. Diremo invece che gli stati di Q_2 sono *non deterministici* perché da essi possono uscire più transizioni etichettate con ϵ; ciò è consentito in quanto il codominio di δ'' è l'insieme potenza di Q, ovvero l'insieme di tutti i possibili sottoinsiemi di Q.

Esempio 4.86. Il seguente automa è non deterministico in quanto dallo stato 1 è possibile transitare arbitrariamente nello stato 2 o nello stato 3 (senza consumare input), risulta cioè $\delta(1, \epsilon) = \{2, 3\}$.



[7]Non si tratta necessriamente di una partizione; è infatti possibile che in alcuni stati la funzione non sia definita.

Come vedremo, l'automa riconosce il linguaggio definito dall'espressione regolare `10*`, cioè l'insieme delle stringhe costituite da un `1` seguito da un numero arbitrario di `0` (incluso nessuno).

Una computazione di un automa non deterministico su input α, $|\alpha| = n$, è una sequenza di stati

$$q_0, q_1, q_2, \ldots$$

tale che:

1. q_0 è lo stato iniziale;

e, per $i = 1, 2, \ldots$.

2. se $q_i \in Q_1$, allora $q_{i+1} = \delta(q, \alpha_j)$, dove $j < n$ è il "prossimo" simbolo in input ($\alpha_{0..j-1}$ è la porzione di input già letta dall'automa);
3. se $q_i \in Q_2$, allora $q_{i+1} \in \delta(q, \epsilon)$.

La computazione di un automa non deterministico può non terminare, ma questo può accadere solo se nel grafo esiste un ciclo etichettato solamente con ϵ, una situazione che denota un progetto non accurato dell'automa stesso. La computazione termina invece, come nel caso di un automa deterministico, per assenza di input oppure perché nello stato corrente, ed eventualmente in dipendenza del simbolo di input, la funzione di transizione non è definita. La nozione di *stringa accettata* (o *riconosciuta*) è invece significativamente diversa rispetto al caso deterministico. Al riguardo, riflettiamo ancora sull'esempio 4.86.

Consideriamo dapprima la stringa di input $\alpha =$ `10`. Partendo dallo stato iniziale (stato 0), l'automa transita nello stato 1 leggendo il simbolo `1`. A questo punto però l'automa può decidere *non deterministicamente* se transitare nello stato 2 oppure nello stato 3, in entrambi i casi senza leggere simboli di input. Il passaggio nello stato 3 provocherebbe il non riconoscimento dell'input poiché per $\delta(3, 0)$ non è definita. Il passaggio nello stato 2 provoca invece il successivo ritorno allo stato 1 a seguito della lettura del simbolo `0`. Come in precedenza, dallo stato 1 l'automa può transitare negli stati 2 o 3. Questa volta la scelta "giusta" è la transizione nello stato 3; infatti il passaggio allo stato 2 provocherebbe l'arresto della computazione per input esaurito, e lo stato 2 non è di accettazione. Invece, il passaggio allo stato 3 provoca l'arresto in uno stato di accettazione.

Come si comporta invece l'automa su input 110? Come nel caso precedente, dopo essere entrato nello stato 1, a seguito della lettura del primo simbolo 1, l'automa si trova a dover decidere fra il passaggio nello stato 2 o nello stato 3. È però facile verificare che entrambe le scelte portano alla terminazione in uno stato non di accettazione.

Che cosa possiamo dedurre di generale da quanto appena osservato riguardo lo specifico automa dell'esempio 4.86? Esistono input (come 110 per l'automa dell'esempio) per i quali non è possibile trovare una sequenza di transizioni che porta l'automa in uno stato di accettazione a seguito della lettura di tutta la stringa di input; vi sono invece input (come 10) per i quali, "azzeccando" le scelte giuste, è possibile arrivare al riconoscimento. Questa è esattamente la definizione di accettazione da parte di un ASFND.

Definizione 4.87. Un ASFND $\mathcal{A}$ accetta un input α se e solo se *esiste* almeno una computazione che, su input α, porta $\mathcal{A}$ in uno stato di accettazione con la lettura di tutta la stringa α. Come nel caso deterministico, il linguaggio $L_{\mathcal{A}}$ riconosciuto da $\mathcal{A}$ è l'insieme delle stringhe da esso accettate.

È facile rendersi conto che, sulla base della definizione appena data, l'automa dell'esempio 4.86 riconosce tutte e sole le stringhe definite dall'espressione regolare 10*. Infatti, su input una stringa della forma 10^i (e solo di questa forma), l'automa ha la possibilità di arrivare nello stato di accettazione dopo aver "ciclato" $i \geq 0$ volte fra gli stati 1 e 2.

Definizione 4.88. Se $\alpha \in L_{\mathcal{A}}$, il costo computazionale (tempo di calcolo) di $\mathcal{A}$ su input α è definito come la lunghezza della computazione più corta che porta $\mathcal{A}$ ad accettare α.

È necessario fare qualche riflessione sul calcolo non deterministico perché è molto probabile che lo studente che incontra per la prima volta a questo concetto rimanga disorientato. Innanzitutto la definizione 4.87 non sembra operativa; a sua volta la definizione 4.88 sembra invece conferire al modello non deterministico una maggiore potenza rispetto a quello deterministico. In "soldoni", infatti, la 4.87 afferma che se nel grafo di transizione c'è un percorso che porta all'accettazione, anche fra moltissimi possibili, l'automa lo trova; la 4.88 arriva invece a dire che il prezzo computazionale da pagare per arrivare a tale risultato equivale al costo di seguire sempre percorso più corto, come se questo si potesse "imbroccare" senza esitazioni e senza preoccuparsi degli altri. Una tale definizione di costo potrebbe applicarsi solo ad una

macchina "fortunata" che, appunto, ogni volta che deve operare una scelta azzecca sempre quella giusta. È evidente che una tale macchina non esiste.

In effetti, il tentativo di implementare in qualche modo computazioni non deterministiche sembra svelare una sorta di "coperta corta": o si usano più risorse di calcolo oppure si impiega molto più tempo rispetto a ciò che le due definizioni 4.87 e 4.88 prese insieme affermano.

Un primo modo di pensare all'esecuzione di un calcolo non deterministico consiste infatti nell'immaginare che, ad ogni possibile biforcazione della computazione (come nel caso dello stato 1 dell'automa dell'esempio 4.86), si abbia la possibilità di utilizzare altrettanti *thread*, ognuno con il compito di percorrere una possibile alternativa. Sulla base di questa interpretazione, la definizione 4.88 appare corretta; in effetti il primo thread che arriva ad accettare l'input può mandare un segnale di terminazione a tutti gli altri. Tuttavia è lecito chiedersi se ciò non nasconda un "trucco"; non è infatti possibile che il numero di thread diventi enorme, situazione che in qualche modo evidenzia la necessità di usare hardware con un numero irrealistico di processori o core? In effetti, se i primi n passi di una computazione non deterministica corrispondessero ad altrettanti punti di biforcazione, il numero di thread impiegati potrebbe teoricamente arrivare a 2^n.

Un approccio alternativo ricorre invece alla tecnica ideata da Arianna e utilizzata da Teseo per uscire dal labirinto di Cnosso; tecnica che in Informatica prende il nome di *backtracking*. In base ad essa, ad ogni possibile biforcazione si sceglie una qualunque strada e, se questa non porta all'uscita del labirinto (nel nostro caso, all'accettazione dell'input), si torna all'ultimo punto di biforcazione che presenta alternative ancora non esplorate, per poi proseguire su una di queste. Tale interpretazione del non determinismo non richiede hardware aggiuntivo; tuttavia essa fa apparire poco sensata la definizione 4.88, questa volta in modo ancora più evidente. Se nel primo caso essa sembrava "nascondere" la necessità di risorse di calcolo in numero spropositato, adesso ad andare in crisi è la nozione stessa di tempo di calcolo. Infatti, è ben possibile che la strada giusta per uscire dal labirinto non sia ne' la più corta ne' tantomeno la prima tentata. Ancora una volta, se ci sono n punti di biforcazione, la via giusta potrebbe essere trovata solo dopo aver esplorato tutte le 2^n alternative.

Come si esce da questa situazione? La risposta è che in generale non se ne esce! In effetti, per modelli di calcolo più potenti rispetto agli automi finiti, come ad esempio gli automi a pila o le macchine di Turing, il problema del rapporto fra versioni deterministiche (realizzabili in concreto, via hardware

o software) e modelli non deterministici è ancora aperto. In tali contesti, i modelli non deterministici assolvono un ruolo importante ma principalmente di natura teorica. Fanno eccezione proprio gli automi finiti!

Nel caso di automi finiti, l'esplosione esponenziale nel tempo calcolo necessario per la simulazione con backtracking, ovvero della quantità di hardware nel caso di simulazione parallela con thread, non si verifica. In particolare, come vedremo nella Sezione 4.3.3, è possibile progettare un algoritmo (deterministico) "efficiente" per uscire dal labirinto, cioè per simulare il comportamento di un generico automa non deterministico su input una qualsiasi stringa α. Tale algoritmo è efficiente nel senso che la sua complessità è entro un fattore costante rispetto a quanto indicato nella definizione 4.88.

Sarebbe pure possibile mostrare, anche se non lo faremo in questi appunti, che si può direttamente trasformare un automa non deterministico in uno deterministico equivalente. Quest'ultimo, pur potendo avere molti più stati rispetto all'automa di partenza, ne ha comunque un numero che è indipendente dalla lunghezza delle stringhe in input, dunque costante. Questo equivale a dimostrare che anche rispetto alla prima possibilità di simulare un processo non deterministico, quella cioè che ricorre all'uso di risorse di calcolo aggiuntive, la temuta esplosione esponenziale non si verifica[8], neppure nel caso in cui la computazione non deterministica abbia n punti di biforcazione.

4.3.2 Dalle espressioni regolari agli automi

Gli automi corrispondenti alle espressiomi regolari che costruiremo rientrano in quelli descritti nella definizione 4.85 con l'ulteriore proprietà che anche lo stato finale è unico e che da esso non si dipartono transizioni.

Il processo di costruzione utilizza dei *componenti base* costituiti da semplici automi per il riconoscimento di una singola lettera, come evidenziato in figura 4.22.

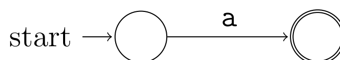


Figure 4.22: Automa base per il riconoscimento di una singola lettera.

[8] La quantità di "hardware" per un automa è proprio misurata in ternimi di numero di stati e di transizioni.

Utilizzando gli automi base si costruiscono poi automi che riconoscono le varie "parti" dell'espressione regolare fino all'automa completo. Per far ciò si utilizzano tre regole di composizione degli automi che corrispondono alle tre regole di formazione delle espressioni regolari, ovvero concatenazione, unione e chiusura. Nella composizione si sfrutta il fatto che gli automi componenti hanno un solo stato iniziale e un solo stato finale. La struttura del generico automa utilizzato nella composizione è illustrata in figura 4.23. Sono evidenziati lo stato iniziale, quello finale e il "corpo" dell'automa che, nella composizione, non gioca alcun ruolo (nel caso di automa per una singola lettera x il corpo è costituito da una sola transizione, etichettata con x).

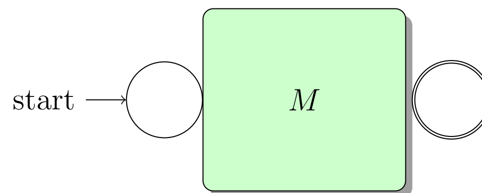


Figure 4.23: Struttura del generico automa non deterministico M corrispondente ad un'espressione regolare. Gli stati evidenziati (iniziale e di accettazione) sono parte di M.

Le regole di composizione corrispondenti a concatenzione, unione e chiusura sono rispettivamente illustrate nelle figure 4.24, 4.25 e 4.26. Si noti come gli automi risultanti abbiano sempre la stessa struttura generale indicata in figura 4.23.

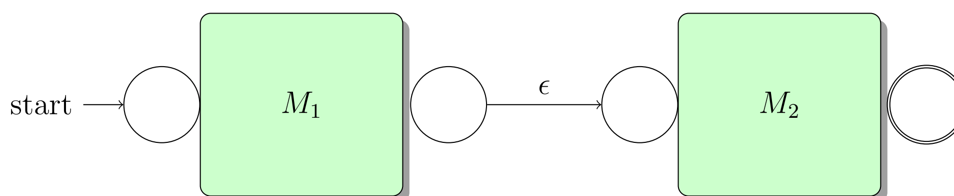


Figure 4.24: ASFND corrispondente all'espressione regolare $\mathcal{E}_1\mathcal{E}_2$ a partire dagli automi M_1 e M_2 per $\mathcal{E}_1$ e $\mathcal{E}_2$.

Esempio 4.89. La figura 4.27 mostra l'automa completo per il riconoscimento dell'espressione regolare $E = \texttt{b}^*\texttt{a}|\texttt{ab}^*$. Si noti la numerazione degli

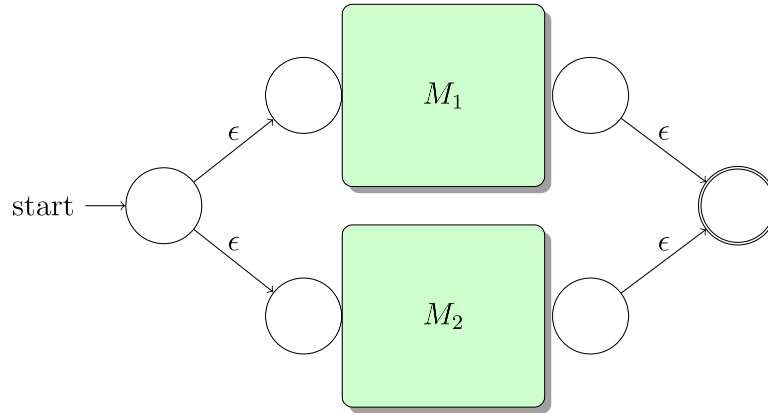


Figure 4.25: ASFND M' corrispondente all'espressione regolare $\mathcal{E}_1 + \mathcal{E}_2$ a partire dagli automi M_1 ed M_2 per $\mathcal{E}_1$ ed $\mathcal{E}_2$. M' ha due nuovi stati, iniziale e finale.

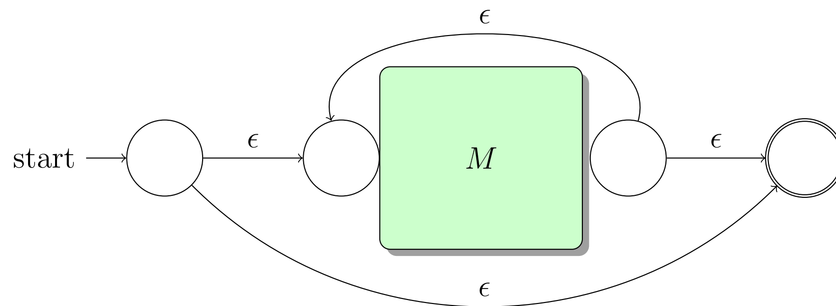


Figure 4.26: ASFND M' corrispondente all'espressione regolare $\mathcal{E}^*$ a partire dall'automa M per $\mathcal{E}$. M' ha due nuovi stati, iniziale e finale.

stati che riflette la struttura di E a partire dalle espressioni regolari più semplici che la compongono, nel rispetto delle precedenze di operatore: il primo automa che viene considerato è quello per il riconoscimento di `b` (stati 0 e 1); a questo viene poi applicata la regola di chiusura per formare l'automa che riconosce `b`* (stati 2, 0, 1 e 3); a questo viene concatenato l'automa per il riconoscimento di `a` (stati 4 e 5) ottenendo l'automa per `b`*`a` (stati 2, 0, 1, 3, 4 e 5); e così via.

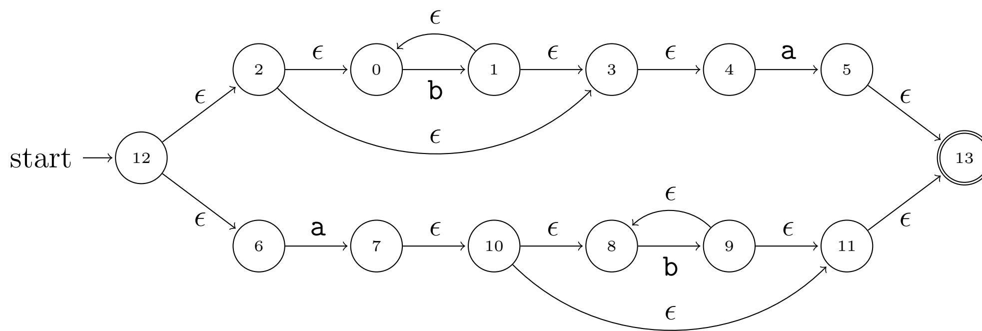


Figure 4.27: ASFND corrispondente all'espressione regolare `b`*`a`|`ab`*.

Gli automi che scaturiscono dal processo di costruzione descritto in questa sezione non sono molto efficienti, nel senso che sono caratterizzati da un elevato numero di stati e di transizioni, molte delle quali "inutili". Tuttavia, esattamente come avviene nel processo di compilazione di un programma scritto in linguaggio ad alto livello, il passaggio completo dalla descrizione iniziale al "prodotto" ottimizzato è più efficacemente affrontato se suddiviso in due passi: (1) realizzazione di un prodotto intermedio corretto, e (2) sua ottimizzazione. Nel caso dei compilatori, il "prodotto" intermedio può essere, ad esempio, un codice indipendente dalla particolare architettura o anche una rappresentazione ad albero del programma, mentre il prodotto finale è il codice macchina ottimizzato per una particolare architettura. Questi due passaggi sono affrontati separatamente nel front-end e nel back-end del compilatore. Nel caso delle espressioni regolari, il prodotto intermedio è già un automa "corretto" mentre il risultato finale potrebbe essere un automa con il numero minimo di stati. La figura 4.28 illustra un automa equivalente a quello di figura 4.89 in cui sono state eliminate tutte le ϵ-transizioni inutili;

l'eliminazione di una ϵ-transizione "inutile" che collega il nodo i al nodo j porta anche all'identificazione di i e j (in altri termini, uno dei due nodi viene rimosso). Per i nostri scopi, tuttavia, non andremo a considerare il problema della minimizzazione del numero di stati.

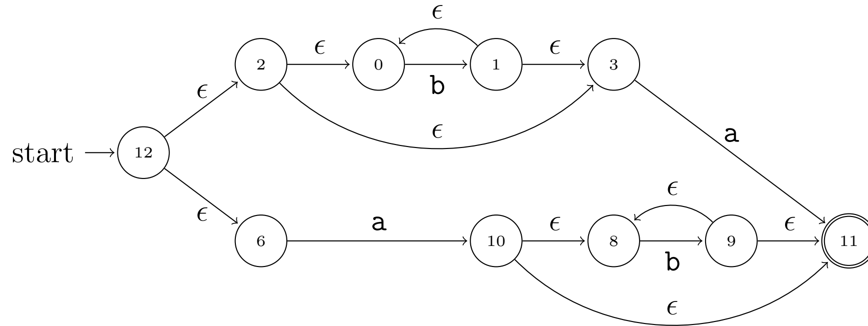


Figure 4.28: Automa dell'esempio 4.89 "ottimizzato". Gli stati 3, 4 sono stati fusi in un solo stato (arbitrariamente etichettato come nodo "3") come pure gli stati 7, 10 e, nell'ordine, 13, 11 e 5, 11. Se uno dei due stati fusi è di accettazione anche lo stato risultante è di accettazione.

4.3.3 Simulazione di automi non deterministici

Descriviamo ora un algoritmo che, dato un generico automa riconoscitore a stati finiti non deterministico M e una stringa I di input, di lunghezza n, verifica se M riconosce (accetta) I.

L'idea di fondo per simulare M consiste semplicemente nel provare tutte le possibilità derivanti dalla presenza di alternative nel grafo di transizione. Per come abbiamo strutturato i nostri automi, queste ultime sono limitate a quegli stati dai quali si dipartono due (e solo due) ϵ-transizioni. Al riguardo, diamo la seguente definizione.

Definizione 4.90. Diremo che uno stato q dell'automa di M è *compatibile* con la porzione di input $I_{0..j-1}$ (per un qualche $j \geq 0$) se l'automa può trovarsi nello stato q dopo aver letto i primi j simboli di I.

L'algoritmo utilizza un'opportuna struttura dati per memorizzare gli stati che sono compatibili con una data porzione di input già vista. Più precisa-

mente, per ogni valore di $j \in \{0, 1, \ldots, n-1\}$, l'algoritmo analizza (estraendoli dalla struttura dati) gli stati compatibili con $I_{0..j-1}$. Detto q un tale stato, ci sono solo due possibilità.

1. $q \in Q_1$, e dunque da q esce una sola transizione, etichettata con un carattere $x \in \Sigma$, che conduce in uno stato che chiameremo q'. In questo caso, se il carattere di input corrente I_j coincide con x, allora q' viene inserito nella struttura dati come stato compatibile con $I_{0..j}$, altrimenti non viene inserito (si noti che q' potrebbe ancora essere compatibile con $I_{0..j}$, ma non per il tramite di q).

2. $q \in Q_2$ e dunque da q escono (al più) due transizioni etichettate con ϵ che conducono negli stati q' e q''. In tal caso, questi ultimi sono ovviamente "equivalenti" a q (perché nel passaggio da q a q' o q'' non si consuma input) e dunque devono essere inseriti nella struttura fra gli stati compatibili con $I_{0..j-1}$.

Quando nella struttura dati non ci sono più stati compatibili con $I_{0..j-1}$, l'algoritmo fa avanzare il puntatore di input (pone cioè $j = j+1$) e ricomincia ad analizzare gli stati compatibili con $I_{0..j-1}$. L'algoritmo termina quando la struttura si svuota completamente ovvero quando si raggiunge lo stato finale.

La precedente descrizione impone certi requisiti alla struttura dati. Per ogni dato valore di j, essa deve infatti consentire l'inserzione di stati compatibili con $I_{0..j-1}$ o con $I_{0..j}$ nonché l'estrazione di stati compatibili con $I_{0..j-1}$. Al riguardo, nell'algoritmo dettagliato utilizzeremo una struttura nota con il nome di *double ended queue* (o semplicemente *deque*). Questa è un "misto" di coda e pila e consente quindi l'inserzione di elementi sia in fondo (come nella coda) sia in cima alla struttura (come nella pila) e l'estrazione degli elementi dalla sola cima (come in entrambe le strutture base). Utilizzando tale struttura, nel caso 1 lo stato viene inserito in fondo, mentre nel caso 2 gli stati vengono inseriti in cima alla deque e l'ultimo stato inserito è anche il primo che viene estratto. Si noti che questa è una scelta di comodo; infatti ciò che è richiesto dall'algoritmo è solo che gli stati compatibili con $I_{0..j-1}$ siano estratti *prima* degli stati compatibili con $I_{0..j}$. L'ordine preciso non è importante.

Per semplicità, nell'algoritmo dettagliato indicheremo con S_1 ed S_2 gli insiemi di stati compatibili con $I_{0..j-1}$ e $I_{0..j}$, rispettivamente (la dipendenza da j è tenuta implicita). Porremo poi $S = S_1 \cup_m S_2$; si noti quindi che S è

un multiset perché uno stato può comparire sia in S_1 che in S_2 e vogliamo che le due "copie" dello stato siano conservate nella struttura dati.

Esempio 4.91. Si consideri nuovamente l'automa di figura 4.27 e si supponga che $I = \mathtt{a}\alpha$, cioè che I inizi con il carattere $\mathtt{a}$. Inizialmente, per $j = 0$ gli stati compatibili con $I_{0..-1} = \epsilon$ sono tutti quelli che possono essere raggiunti a partire dallo stato iniziale senza leggere caratteri di input, e dunque seguendo solo ϵ-transizioni; risulta dunque $S_1 = \{12, 2, 0, 3, 4, 6\}$. Gli stati compatibili con $I_{0..0} = \mathtt{a}$ sono invece quelli che possono essere raggiunti seguendo dapprima una transizione etichettata con il carattere $\mathtt{a}$ che si diparta da uno degli stati di S_1, quindi (eventualmente) da ϵ-transizioni. È facile vedere che $S_2 = \{5, 13, 7, 10, 8, 11\}$.

Inizialmente l'algoritmo pone $j = 0$, $S_1 = \{q_0\}$, $S_2 = \{\}$. Il solo stato iniziale q_0 viene dunque inserito nella struttura e tale stato è in effetti compatibile con $I_{0..j-1} = \epsilon$. Si ripetono quindi le seguenti azioni.

1. Se $S = \{\}$ oppure $j \geq n$ la simulazione termina con insuccesso.
2. Se $S_1 = \{\}$ e $S_2 \neq \{\}$, si pone $j = j + 1$, $S_1 = S_2$ e $S_2 = \{\}$.
3. Se $q \in S_1$:
   (a) se $q \in Q_F$, cioè q è lo stato finale, la simulazione termina con successo se e solo se $j = n - 1$.
   (b) se $q \in Q_1$ e $\delta(q, I_j)$ è definita, si pone $S_2 = S_2 \cup \{\delta(q, I_j)\}$, $S_1 = S_1 \backslash \{q\}$.
   (c) Se $q \in Q_1$ ma $\delta(q, I_j)$ non è definita, si pone $S_1 = S_1 \backslash \{q\}$.
   (d) Se $q \in Q_2$, si pone $S_1 = (S_1 \backslash \{q\}) \cup \delta(q, \epsilon)$.

Nello pseudo-codice di figura 4.29, gli automi da simulare sono implementati mediante tre array paralleli, la cui lunghezza è pari al numero di stati. Infatti, per ogni stato dobbiamo rappresentare al più tre informazioni: (1) il carattere atteso, se si tratta di stato di Q_1, o altrimenti la stringa vuota; (2) uno o due stati successori. Si veda la tabella 4.2, che rappresenta l'ASFND di figura 4.27.

Lo pseudo-codice di figura 4.29 utilizza la struttura *deque* sulla quale sono definite le operazioni: put(v), che inserisce v in coda; push(v), che inserisce v in testa; pop(), che estrae un elemento dalla testa; empty(), che controlla

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
`ic`	`b`	ϵ	ϵ	ϵ	`a`	ϵ	`a`	ϵ	`b`	ϵ	ϵ	ϵ	ϵ	\$
`state1`	1	0	0	4	5	13	7	10	9	8	8	13	2	
`state2`		3	3	4		13		10		11	11	13	6	

Table 4.2: Descrizione tabellare dell'ASFND di figura 4.28 mediante tre array: `ic`, `state1` e `state2`. Si noti che, per uno stato $q \in Q_2$, è possibile stabilire se da q si dipartono due o una sola transizione perché, nel secondo caso, i valori di `state1` e `state2` coincidono. In un'implementazione concreta, i simboli ϵ e \$, quest'ultimo che indica lo stato finale, possono essere rappresentati da qualsiasi simbolo che non faccia parte dell'alfabeto di input. Oltre agli array paralleli, la rappresentazione dell'automa richiede una singola variabile che indica lo stato iniziale, nel caso dell'esempio lo stato 12.

se la struttura è vuota. Si osservi come viene implementata la separazione fra gli stati S_1 ed S_2. Si utilizza un simbolo particolare chiamato *read* (-1, nella fattispecie), che si suppone non essere parte dell'alfabeto di input. Tale simbolo è il primo inserito nella struttura. Ogni operazione push() inserisce un elemento in testa, e quindi "prima" di *read*; viceversa, ogni operazione put() inserisce un elemento in coda e dunque "dopo" *read*. Quando *read* viene estratto, vuol dire che non si sono più elementi davanti (cioè S_1 è vuoto); *read* viene dunque inserito nuovamente in fondo con il risultato che gli elementi che lo seguivano (l'insieme S_2) ora diventa il nuovo S_1 mentre S_2 diviene vuoto.

ASFND-MATCHER$(T, ic, state1, state2)$

```
 1  read = −1                       // Simbolo di separazione fra stati di S1 e di S2
 2  Deque dq                                 // Definizione di una struttura deque
 3  n = length(I)                                        // Lunghezza dell'input
 4  j = 0
 5  q = 0
 6  dq.put(read)
 7  while j < n
 8      if q == read
        // Non ci sono stati in S1, si passa al successivo carattere di input
 9          j = j + 1
10          dq.put(read)
11      else if ic[q] == $ and j == n − 1
12              return accept
13          else if ic[q] == I[j]                                // Stato in Q1
14                  dq.put(state1[q])
15              else if ic[q] == ϵ                               // Stato in Q2
16                      dq.push(state1[q])
17                      if state2[q] ≠ state1[q]
18                          dq.push(state2[q])
19      if dq.empty()
20          return reject
21      q = dq.pop()
22  return reject
```

Figure 4.29: Algoritmo di simulazione di un ASFND.