Introduzione

La maggior parte delle informazioni trattate da un calcolatore sono in formato testuale. Si pensi ad esempio ai testi scritti in linguaggio naturale, ma anche a tutte quelle informazioni che sono rappresentabili mediante una sequenza finita di simboli (partiture musicali, immagini, sequenze di DNA). Inoltre poiché la loro quantità tende ad aumentare esponenzialmente anno dopo anno, la ricerca di algoritmi efficienti per il trattamento dei *testi* ha una grande valenza applicativa, oltre che teorica. Questi algoritmi vengono generalmente applicati a strumenti per la manipolazione dei dati (come ad esempio gli elaboratori di testi), ma riguardano anche il modo in cui questi dati vengono memorizzati o ricercati (algoritmi di compressione o di indicizzazione).

Uno dei problemi classici inerenti al trattamento dei testi è il problema del *Pattern-Matchin esatto*. Esistono anche altri problemi simili come il Pattern-Matching *multiplo* o il Pattern-Matching *approssimato*, ma poiché questi non verranno trattati in questa tesi, d'ora in avanti con il termine Pattern-Matching ci riferiremo sempre a quello esatto.

Il problema del Pattern-Matching consiste nel trovare tutte le occorrenze di una stringa (chiamata *pattern*) all'interno di un'altra di dimensioni maggiori (chiamata *testo*). Le soluzioni al problema del Pattern-Matching si dividono in due categorie in base all'applicazione. Nella prima il testo viene considerato variabile (si pensi ad esempio alla funzione di ricerca all'interno di un elaboratore di testi). Le soluzioni appartenenti a questa categoria sono generalmente basate su confronti o su automi a stati finiti, ed effettuano una preelaborazione del pattern per velocizzare l'operazione di ricerca. Nella seconda categoria invece il testo viene considerato immutabile (si pensi ad un documento all'interno di una banca dati). Le soluzioni appartenenti a

questa categoria effettuano una preelaborazione del testo che, nonostante sia più complessa rispetto a quelle che vengono fatte sul pattern, consente di effettuare le ricerche più rapidamente. In questa tesi ci occuperemo soltanto delle soluzioni appartenenti alla prima categoria.

Gli algoritmi basati su **confronti** lavorano allineando il primo carattere del pattern con un carattere all'interno del testo e verificando se a partire da quella posizione si ha l'occorrenza. Dopo il confronto l'algoritmo effettua uno spostamento verso destra allineando il pattern con un nuovo carattere. Le differenze tra i vari algoritmi consistono nel modo in cui vengono fatti gli allineamenti: gli algoritmi più efficienti sono generalmente quelli che riescono ad ottenere spostamenti più lunghi e che quindi effettuano un minor numero di confronti. L'algoritmo più semplice appartenente a questa categoria è quello **Naive** [CLRS05] che non effettua alcuna preelaborazione ma che (proprio per questo motivo) avanza nel testo un carattere alla volta.

Un approccio alternativo è stato proposto da **Karp** e **Rabin** [KR87]: viene calcolato un valore *hash* del pattern, quindi per ogni allineamento viene calcolato il valore hash della sottostringa del testo corrispondente. Se i due valori coincidono allora viene fatto un confronto carattere per carattere (necessario in quanto due stringhe diverse potrebbero produrre lo stesso valore hash). Poiché è possibile calcolare il valore hash di un allineamento in tempo costante a partire dal valore hash dell'allineamento precedente, questa tecnica offre un risparmio in termini di confronti fra singoli caratteri nel caso medio, nonostante esegua spostamenti unitari così come l'algoritmo Naive.

Il primo algoritmo basato su confronti con complessità lineare nel caso peggiore è stato proposto da **Knuth**, **Morris** e **Pratt** nel 1977 [KMP77] e verrà descritto nella sezione 1.3, mentre gli algoritmi più efficienti nella pratica sono quelli della famiglia **Boyer-Moore** [BM77, Hor80, HS91].

Una soluzione classica al problema che impiega tempo lineare ed offre buone prestazioni è quella basata sulla costruzione di **automi a stati finiti** [CLRS05]. Gli algoritmi basati su automi sono di due tipi: quelli che usano automi *deterministici* e quelli che usano automi *non deterministici*. In entrambi i casi si costruisce un automa sul pattern e si leggono tutti i caratteri del testo uno alla volta. Ogni volta che l'automa raggiunge uno stato

accentante viene riportata l'occorrenza del pattern. Questo tipo di algoritmi hanno nella fase di ricerca una complessità lineare sulla dimensione del testo in tutti i casi, mentre la costruzione dell'automa dipende dalla lunghezza del pattern e dalla dimensione dell'alfabeto.

Nel 1992 è stato presentato un algoritmo basato su automi non deterministici che sfrutta la tecnica del **bit-parallelism** [BYG92]. Questo, insieme ad altri algoritmi dello stesso tipo presentati successivamente [NR98, FG05, PT03], sfruttano il parallelismo intrinseco delle operazioni *bit a bit* per implementare efficientemente l'automa attraverso un'unica *word* del calcolatore. Tuttavia hanno un limite sulla dimensione del pattern imposta dalla dimensione della word.

La tesi è organizzata nel modo seguente: nel capitolo 1 verranno presentate alcune soluzioni classiche al problema del Pattern-Matching. Nel capitolo 2 verrà presentata la tecnica del bit-parallelism e verranno presentati alcuni algoritmi che fanno uso di questa tecnica. Nel capitolo 3 verrà proposta una nuova tecnica di rappresentazione dell'automa che permette di superare il limite sulla dimensione del pattern imposto dalla lunghezza della word. Inoltre verranno proposti alcuni algoritmi che fanno uso di questa nuova rappresentazione. Nel capitolo 4 verranno infine confrontate le prestazioni di questa nuova soluzione con alcune di quelle più efficienti presenti in letteratura.

Notazione e terminologia

Sia Σ un insieme finito di caratteri chiamato alfabeto, e siano $T[0...n-1]$ e $P[0...m-1]$ due sequenze di caratteri di Σ chiamate rispettivamente testo e pattern, di lunghezza pari a n e m ($n \geq m$). Diremo che il pattern P occorre nel testo T alla posizione s ($0 \leq s \leq n-m$) se $T[s...s+m-1] = P[0...m-1]$, ovvero se $T[s+i] = P[i]$ per $0 \leq i \leq m-1$. Si dice anche che s è uno *shift valido* in T per il pattern P.

Con Σ^* viene rappresentato l'insieme infinito di tutte le possibili stringhe costruite a partire da Σ (quindi $T, P \in \Sigma^*$). Con $|S|$ viene indicata la lunghezza in caratteri della stringa S. Il simbolo ε rappresenta la stringa nulla, ovvero tale che $|\varepsilon| = 0$. Per convenzione si ha che $\varepsilon \in \Sigma^*$.

Per indicare dei caratteri ripetuti all'interno di una stringa viene usata la

notazione esponenziale (ad esempio $aaabbc = a^3b^2c$). Questa stessa notazione verrà utilizzata anche con le stringhe di bit (quindi $0^2 1^3 0 1^2 = 00111011$). Con $S[i]$ $(0 \leq i \leq |S| - 1)$ si indica l'$(i + 1)$-esimo carattere della stringa S, mentre con $S[i...j]$ $(0 \leq i \leq j \leq |S| - 1)$ si indica la sottostringa formata dalla sequenza dei caratteri di S a partire dalla posizione i fino alla posizione j (ad esempio se $S = abcdef$, $S[2] = c$, $S[1...3] = bcd$). Con S_i si indica la sottostringa formata dai primi i caratteri della stringa S, quindi $S_i = S[0...i - 1]$. Con S^r indichiamo la stringa inversa di S, ad esempio se $S = abcdef$ allora $S^r = fedcba$.

Date due stringhe x e y si definisce concatenazione (e si indica con xy) la sequenza formata da tutti i caratteri di x seguita da tutti i caratteri di y. Data una stringa x, si dice che y è un prefisso di x (e si indica con $y \sqsubset x$) se $x = yw$, con $w \in \Sigma^*$. Analogamente si dice che y è un suffisso di x (e si indica con $y \sqsupset x$) se $x = wy$, con $w \in \Sigma^*$. Date due stringhe $x, y \in \Sigma^*$, si dice che x è un fattore di y se $y = uxv$, con $u, v \in \Sigma^*$. Con $Fact(x)$ indichiamo l'insieme di tutti i fattori di x.

Infine con la notazione $(x)_y$ indichiamo un numero x rappresentato nella notazione in base y, ad esempio $(15)_{10} = (1111)_2 = (F)_{16}$.

Capitolo 1

Le soluzioni classiche al problema

1.1 La soluzione Naive

L'algoritmo **Naive** [CLRS05] rappresenta la soluzione più semplice ed intuitiva al problema del Pattern-Matching esatto.

Algoritmo 1 NAIVE-MATCHER(T, P)

```
1: n ← length[T]
2: m ← length[P]
3: for s ← 0 to n − m do
4:    i ← 0
5:    while i < m and P[i] = T[s + i] do
6:       i ← i + 1
7:    end while
8:    if i = m then
9:       report match at s
10:   end if
11: end for
```

Per ogni posizione s del testo ($0 \leq s \leq n-m$) viene verificato se $T[s...s+m-1] = P[0...m-1]$. Poiché tutti gli spostamenti che vengono fatti sono unitari, nel caso peggiore (ad esempio con $T = a^n$ e $P = a^m$) si ha una complessità computazionale di $\Theta(n \cdot m)$, mentre nel caso migliore (ad esempio con $T = a^n$ e $P = b^m$) la complessità scende a $\Theta(n)$.

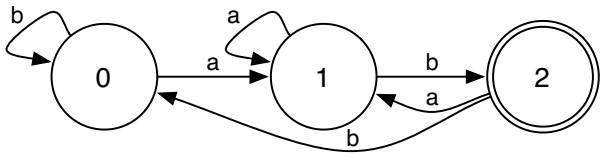

Figura 1.1: un semplice automa a tre stati che accetta la stringa ab in cui $Q = \{0, 1, 2\}$, $\Sigma = \{a, b\}$, $q_0 = 0$, $A = \{2\}$. Le frecce indicano le transizioni tra gli stati per ogni carattere.

Le prestazioni scadenti di questa soluzione sono dovute al fatto che l'algoritmo non effettua alcuna preelaborazione. Altri algoritmi che effettuano una preelaborazione del pattern possono migliorare gli avanzamenti (come ad esempio l'algoritmo di **Knuth-Morris-Pratt** [KMP77] o quello di **Boyer-Moore** [BM77]) o il tempo di testing (come ad esempio l'algoritmo di **Karp-Rabin** [KR87]).

1.2 La soluzione basata su automi a stati finiti

Questa soluzione, nonostante avanzi di un carattere alla volta come l'algoritmo Naive, ha una complessità lineare. Un automa a stati finiti deterministico M è una 5-tupla $(Q, q_0, A, \Sigma, \delta)$, dove

- Q è un insieme finito di stati
- $q_0 \in Q$ è lo stato iniziale
- $A \subseteq Q$ è un insieme distinto di stati accentanti
- Σ è un alfabeto finito di input
- δ è una funzione (detta funzione di transizione) definita in $Q \times \Sigma$ a valori in Q

L'automa parte dallo stato q_0 e legge una stringa un carattere alla volta, passando da uno stato all'altro attraverso la funzione δ: se l'automa si trova nello stato q, dopo aver letto il carattere $\gamma \in \Sigma$ si troverà nello stato $\delta(q, \gamma)$. Se M raggiunge uno stato accentante si dice che l'automa ha accettato la

stringa letta fino alla posizione corrente. Si definisce $L \subseteq \Sigma^*$ il linguaggio riconosciuto dall'automa, ovvero l'insieme di tutte le stringhe che vengono accettate dall'automa. Nel caso di automi costruiti per risolvere il problema del Pattern-Matching si ha $L = \{P\}$.

Dato il pattern P con $|P| = m$, l'automa deterministico ad esso associato viene costruito nel modo seguente: l'insieme degli stati è $Q = \{0, 1, ..., m\}$, $q_0 = 0$ è lo stato iniziale e m è l'unico stato accentante. Per definire la funzione di transizione introduciamo una nuova funzione detta funzione *suffisso*, definita in Σ^* a valori in $\{0, 1, ..., m\}$. Sia $x \in \Sigma^*$, $\sigma(x)$ è la lunghezza del prefisso più lungo di P che è anche suffisso di x:

$$\sigma(x) = max\{k : P_k \sqsupset x\} \tag{1.1}$$

A questo punto è possibile definire la funzione di transizione come:

$$\delta(q, a) = \sigma(P_q a) \qquad \forall q \in Q, \forall a \in \Sigma \tag{1.2}$$

L'algoritmo 2 fornisce una semplice procedura per calcolare la funzione di transizione. La procedura proposta ha una complessità pari a $O(m^3 \cdot |\Sigma|)$, tuttavia esistono altre procedure più efficienti [CH97] che riducono la complessità a $O(m \cdot |\Sigma|)$. L'algoritmo 3 riporta la procedura di ricerca. Essa prende in input il testo T, la funzione di transizione δ e lo stato accentante, e ha una complessità pari a $\Theta(n)$.

	0	1	2	3	4	5	6	7
a	1	1	3	1	5	1	7	1
b	0	2	0	4	0	4	0	2
c	0	0	0	0	0	6	0	0

Tabella 1.1: Funzione di transizione definita sul pattern $P = ababaca$ con $\Sigma = \{a, b, c\}$. L'automa corrispondente è rappresentato nella figura 1.2.

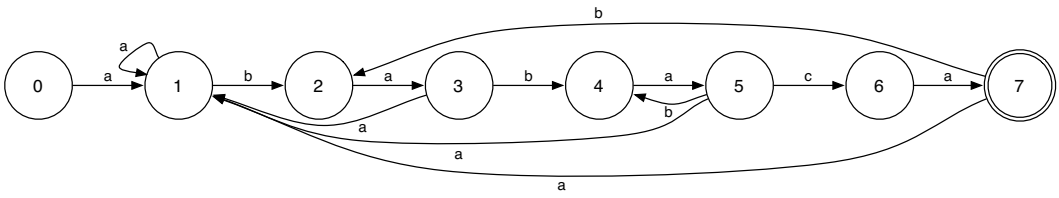

Figura 1.2: rappresentazione grafica di un automa costruito sul pattern $P = ababaca$ con $\Sigma = \{a, b, c\}$. Per semplicità di notazione sono state omesse nella figura tutte le transizioni che riportano allo stato iniziale.

Algoritmo 2 COMPUTE-TRANSITION-FUNCTION(P, Σ)

```
1: m ← length[P]
2: for q ← 0 to m do
3:    for all a ∈ Σ do
4:       k ← min(m + 1, q + 2)
5:       repeat
6:          k ← k − 1
7:       until P_k ⊐ P_q a
8:       δ(q, a) ← k
9:    end for
10: end for
11: return δ
```

Algoritmo 3 FINITE-AUTOMATON-MATCHER(T, Σ, m)

```
1: n ← length[T]
2: q ← 0
3: for i ← 0 to n − 1 do
4:    q ← δ(q, T[i])
5:    if q = m then
6:       report match at i − m
7:    end if
8: end for
```

1.3 L'algoritmo di Knuth-Morris-Pratt

L'algoritmo che verrà adesso presentato è stato il primo algoritmo di Pattern-Matching basato su confronti ad avere una complessità computazionale pari a $\Theta(n)$ [KMP77]. Così come avviene nella soluzione basata su automi, l'algoritmo di **Knuth-Morris-Pratt** effettua una preelaborazione del pattern al fine di calcolare una funzione chiamata *prefisso* (identificata con π). La funzione π viene utilizzata dall'algoritmo di ricerca per calcolare lo spostamento del pattern sul testo a seguito di un match del pattern o di un mismatch nel confronto tra due caratteri. Formalmente, dato un pattern $P[0...m-1]$, la funzione $\pi : \{0, 1, 2, ..., m\} \rightarrow \{-1, 0, 1, ..., m-1\}$ è definita come

$$\pi(q) = max(\{k : k < q, P_k \sqsupset P_q\} \cup \{-1\}) \tag{1.3}$$

Per capire come viene utilizzata la funzione prefisso si consideri la situazione rappresentata in figura 1.3. Inizialmente il primo carattere del pattern è allineato a $T[0]$ ed effettuando i confronti si verifica che $T_i = P_i = u$, mentre $T[i] \neq P[i]$. Sia k la lunghezza del più lungo prefisso di u che è anche un suo suffisso, se alliniamo il primo carattere del pattern con $T[i-k]$ è possibile riprendere i confronti dalla posizione $T[i]$ in quanto $P_k = T[i-k...i-1] = v$ (dato che $v \sqsupset u$). Inoltre, poiché v è il suffisso più lungo di u abbiamo la certezza di non saltare shift validi in T.

Per ottimizzare la procedura è possibile fare un'ulteriore ipotesi sul carattere $P[k]$. Si consideri ancora la figura 1.3; poiché sappiamo che $T[i] \neq P[i]$, quando calcoliamo la lunghezza k del suffisso possiamo imporre la condizione che $P[k] \neq P[i]$. Infatti se così non fosse all'allineamento successivo si avrebbe nuovamente $P[k] = P[i] \neq T[i]$. Si noti però che questa condizione non implica che $T[i] = P[k]$, quindi il controllo va comunque effettuato. Tuttavia in questo modo si esclude a priori un caso che sicuramente provocherebbe un mismatch.

0	1	2	3	4	5	6	7
-1	0	-1	0	-1	3	-1	1

Tabella 1.2: Funzione prefisso definita sul pattern $P = ababaca$.

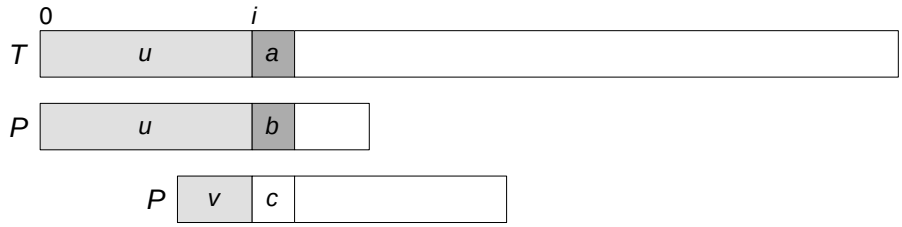

Figura 1.3: Nuovo allineamento del pattern a seguito di un mismatch nell'algoritmo di Knuth-Morris-Pratt, in cui $v \sqsupset u$.

L'algoritmo 4 riporta la procedura per il calcolo della funzione prefisso prendendo come input il pattern. La procedura ha una complessità computazionale $\Theta(m)$. L'algoritmo 5 invece riporta la procedura di ricerca che prende in input il testo, il pattern e la funzione prefisso calcolata in fase di preelaborazione. La procedura ha una complessità computazionale $\Theta(n)$.

Algoritmo 4 COMPUTE-PREFIX-FUNCTION(P)

```
1:  m ← length[P]
2:  π[0] ← 0
3:  k ← 0
4:  for q ← 1 to m − 1 do
5:     while k > 0 and P[k + 1] ≠ P[q] do
6:        k ← π[k]
7:        if P[k + 1] = P[q] then
8:           k ← k + 1
9:        end if
10:       π[q] ← k
11:    end while
12: end for
13:
14: return π
```

Algoritmo 5 KMP-MATCHER(T, P, π)

```
1:  n ← length[T]
2:  m ← length[P]
3:  q ← −1
4:  for i ← 0 to n − 1 do
5:     while q > −1 and P[q + 1] ≠ T[i] do
6:        q ← π[q]
7:     end while
8:     if P[q + 1] = T[i] then
9:        q ← q + 1
10:    end if
11:    if q = m − 1 then
12:       report match at i − m
13:       q ← π[q]
14:    end if
15: end for
```

Capitolo 2

Il bit-parallelism

2.1 Gli automi non deterministici

Un **automa a stati finiti non deterministico** M è una 5-tupla $(Q, q_0, A, \Sigma, \delta)$, dove

- Q è un insieme finito di stati
- $q_0 \in Q$ è lo stato iniziale
- $A \subseteq Q$ è un insieme distinto di stati accentanti
- Σ è un alfabeto finito di input
- δ è una funzione (detta funzione di transizione) definita in $Q \times \Sigma$ a valori in $\mathcal{P}(Q)$

La differenza tra gli automi deterministici (presentati nella sezione 1.2) e quelli non deterministici sta nella definizione della funzione δ. Infatti mentre nei primi la funzione definisce le transizioni da uno stato di Q ad un altro, in questi le transizioni sono definite da uno stato ad un sottoinsieme di stati di Q. Questo si traduce nella possibilità di avere nell'automa più stati attivi nello stesso istante. Gli automi deterministici si possono considerare dei casi particolari di automi non deterministici.

2.2 La tecnica del bit-parallelism

Il **bit parallelism** è una tecnica algoritmica che sfrutta la capacità del calcolatore di operare contemporaneamente su tutti i bit di una *word* attraverso

gli *operatori logici*, anche noti come operatori *bit a bit*. Una word è una sequenza di bit di una determinata lunghezza (32 o 64 nella maggior parte dei casi) e rappresenta l'unità di memorizzazione di un microprocessore. In base alla rappresentazione utilizzata (complemento a due, modulo e segno, ecc.) la stessa word può assumere diversi valori (intero con o senza segno, numero in virgola mobile e così via).

Gli operatori logici operano su tutti i bit di una word indipendentemente dalla rappresentazione utilizzata. Quelli più comunemente usati sono:

- OR, operatore binario rappresentato con il simbolo |
- AND, operatore binario rappresentato con il simbolo &
- NOT, operatore unario rappresentato con il simbolo $\sim$
- XOR, operatore binario rappresentato con il simbolo ^

A questi vanno aggiunti gli operatori di shift destro (rappresentato con $\gg$) e di shift sinistro (rappresentato con $\ll$) che, prendendo un intero l come parametro, spostano tutti i bit di una word rispettivamente a destra e a sinistra di l bit.

a	b	a | b	a & b	∼a	a ^ b
0	0	0	0	1	0
0	1	1	0	1	1
1	0	1	0	0	1
1	1	1	1	0	0

Tabella 2.1: tavola di verità di alcuni operatori logici

Nella tabella 2.1 sono riportati i risultati dei vari operatori applicati ai singoli bit. Quando gli operatori logici vengono applicati alle word, tutti i bit vengono processati parallelamente ma l'i-esimo bit del risultato dipende solamente di bit i-esimi dei due operandi. Per quanto riguarda gli operatori di shift invece i bit che superano i limiti della word vengono troncati, mentre nelle posizioni vuote viene inserito il valore 0.

La tecnica del bit parallelism consiste nel rappresentare più elementi all'interno di una word in modo tale che sia possibile aggiornarne i valori con un'unica operazione bit a bit, riducendo così la complessità di un algoritmo

Figura 2.1: rappresentazione grafica di un automa non deterministico costruito su un pattern $P[0...m-1]$.

di un fattore al più pari alla dimensione della word del calcolatore. Questa tecnica ha avuto una buona applicazione negli algoritmi di risoluzione del problema del Pattern-Matching basati su automi poiché uno stato attivo o non attivo di un automa può essere rappresentato con un semplice bit. Tuttavia l'applicazione di questa tecnica impone il vincolo che la lunghezza del pattern sia minore o uguale alla dimensione della word.

2.3 L'algoritmo Shift-Or

Lo **Shift-Or** presentato da Baeza-Yates e Gonnet [BYG92] è stato il primo algoritmo di Pattern-Matching ad utilizzare la tecnica del bit parallelism. Questa soluzione è basata su automi a stati finiti non deterministici.

A partire da un pattern $P[0...m-1]$ l'automa non deterministico viene costruito in questo modo: l'insieme degli stati è $Q = \{0, 1, ..., m\}$, lo stato iniziale è 0 ed è sempre attivo, mentre lo stato accentante è m. Le transizioni definite dalla funzione δ sono tutte del tipo

$$\delta(i-1, P[i-1]) = i \qquad \forall i \in \{1, 2, ..., m\} \tag{2.1}$$

L'automa viene codificato mediante una sequenza di m bit che indichiamo con $R = r_m r_{m-1} ... r_1$; il bit r_i in R è associato allo stato i-esimo dell'automa. Uno stato attivo viene codificato con il valore 0, mentre uno stato non attivo con il valore 1. Poiché lo stato iniziale è sempre attivo (e quindi ha sempre il valore 0) non è necessario memorizzarlo in R: questo significa che per codificare un automa costruito su un pattern di lunghezza m e che ha quindi $m+1$ stati occorrono esattamente m bit.

Sia R_j la configurazione dell'automa dopo la lettura del j-esimo carattere del testo, se lo stato i dell'automa è attivo (ovvero se $r_i = 0$) allora si ha che

i primi i caratteri del pattern corrispondono agli ultimi i caratteri letti del testo, ovvero

$$P_i = T[j - i + 1...j] \tag{2.2}$$

Poiché l'automa è non deterministico possono esserci più stati attivi, ad ognuno dei quali è associata la lunghezza di un prefisso del pattern che termina al j-esimo carattere del testo.

Inizialmente l'automa non ha nessuno stato attivo (se si esclude lo stato iniziale non memorizzato), quindi $R = 1^m$. Ogni volta che viene letto il carattere $T[j]$ ogni bit di R assume il seguente valore

$$r_i = \begin{cases} 0 & \text{se } i = 1 \text{ e se } P[0] = T[j] \\ 0 & \text{se } r_{i-1} = 0 \text{ in } R_{j-1} \text{ e se } P[i] = T[j] \\ 1 & \text{altrimenti} \end{cases} \tag{2.3}$$

Ogni volta che in R_j il bit $b_m = 0$ l'automa avrà raggiunto il suo stato accentante, ovvero si avrà l'occorrenza del pattern in $T[j - m + 1...j]$. Come si può notare facilmente questa procedura ha una complessità di $\Theta(n \cdot m)$ in quanto per ogni carattere letto del testo vanno aggiornati tutti gli stati dell'automa. Tuttavia grazie alla tecnica del bit parallelism è possibile aggiornare tutti gli stati contemporaneamente in tempo costante. Sia $\Sigma = \{c_1, ..., c_{|\Sigma|}\}$ l'alfabeto, per ogni carattere $c_i \in \Sigma$ viene costruita una maschera di bit $B[c_i] = b_{m-1}b_{m-2}...b_0$ dove

$$b_j = \begin{cases} 0 & \text{se } c_i = P[j] \\ 1 & \text{altrimenti} \end{cases} \tag{2.4}$$

`B[a] = 11010`	`B[b] = 10101`	`B[c] = 01111`	`B[d] = 11111`

Tabella 2.2: esempio dei valori di B calcolati per $P = ababc$ e $\Sigma = \{a, b, c, d\}$ nell'algoritmo Shift-Or

Per aggiornare tutti i bit di R secondo l'equazione 2.3 è sufficiente fare uno shift a sinistra di 1 bit e fare un'operazione di OR logico tra il risultato dello shift e la maschera di bit corrispondente al carattere letto dal testo: con l'operazione di shift viene propagata l'attività da ogni stato attivo al suo successivo, mentre la successiva operazione di OR logico disattiva gli stati che erano diventati attivi ma che non ne avevano diritto. Espresso in formula

$$R_j = (R_{j-1} \ll 1) \mid B[T[j]] \tag{2.5}$$

L'algoritmo 6 fornisce la procedura per la preelaborazione del pattern la cui complessità computazionale è di $\Theta(m + |\Sigma|)$, mentre la procedura di ricerca fornita dall'algoritmo 7 ha una complessità computazionale di $\Theta(n)$. Nel complesso questo algoritmo necessità di uno spazio aggiuntivo pari a $\Theta(|\Sigma|)$ per memorizzare le maschere di bit.

Algoritmo 6 SHIFT-OR-PREPROCESS(P, Σ)

```
1: m ← length[P]
2: for all c ∈ Σ do
3:    B[c] ←~ 0
4: end for
5: s ← 1
6: for i ← 0 to m − 1 do
7:    B[P[i]] ← B[P[i]] ^ s
8:    s = s ≪ 1
9: end for
10:
11: return B
```

Algoritmo 7 SHIFT-OR(T, P, B)

```
1: n ← length[T]
2: m ← length[P]
3: R ←~ 0
4: for j ← 0 to n − 1 do
5:    R = (R ≪ 1) | B[T[j]]
6:    if R & 10^(m−1) = 0 then
7:       report match at i − m + 1
8:    end if
9: end for
```

2.4 L'algoritmo BNDM

BNDM è l'acronimo di *Backward Nondeterministic Dawg Matching*, un algoritmo di Pattern-Matching presentato da Gonzalo Navarro e Mathieu Raffinot nel 1998 [NR98]. Questo algoritmo è basato sui *suffix automata*, ovvero degli automi a stati finiti che costruiti su un pattern, ne accettano tutti i

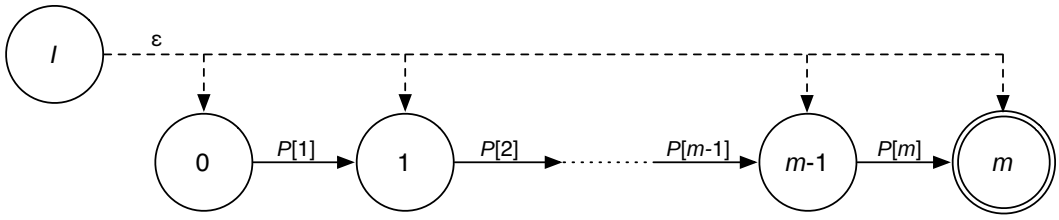

Figura 2.2: rappresentazione grafica di un *suffix automaton* non deterministico costruito su un pattern $P[0...m-1]$.

suffissi. La versione deterministica di questi automi è stata utilizzata nell'algoritmo **BDM** [CR94], tuttavia Navarro e Raffinot hanno utilizzato una versione non deterministica che, oltre ad avere una struttura più semplice rispetto alla prima, permette l'applicazione del bit parallelism.

Riferendoci alla figura 2.2 lo stato iniziale dell'automa è I. Poiché lo stato I ammette una transizione verso tutti gli altri stati leggendo il carattere ε, l'automa allo stato iniziale avrà tutti gli stati attivi. Successivamente si comporterà come un normale automa a stati finiti. Così come avviene nell'algoritmo Shift-Or, anche qui l'automa viene codificato con una sequenza di m bit (che indichiamo con $D = d_m d_{m-1}...d_1$) ma a differenza di quello uno stato attivo viene codificato con il valore 1, mentre uno stato non attivo con il valore 0.

La ricerca avviene facendo scorrere sul testo da sinistra verso destra una finestra che raggruppa un numero di caratteri pari a quelli del pattern. Ogni volta che la finestra raggiunge una nuova posizione nel testo (che indichiamo con *pos*) l'automa legge da destra verso sinistra i caratteri che vi si trovano all'interno. La lettura termina quando viene trovata un'occorrenza del pattern oppure quando tutti gli stati dell'automa sono disattivati e quindi non sono possibili ulteriori transizioni. A quel punto la finestra viene spostata in una nuova posizione del testo e il processo ricomincia.

L'inizializzazione avviene impostando $D = 1^m$ e viene effettuata ad ogni riposizionamento della finestra. All'iterazione k ($1 \leq k \leq m$) il bit d_i in D è pari a 1 se e solo se

$$P[m-i...m-i+k-1] = T[pos+m-k...pos+m-1] \qquad (2.6)$$

Dall'equazione 2.6 si può notare che se $d_m = 1$ all'iterazione k allora un

prefisso del pattern di lunghezza k termina alla posizione $pos + m - 1$ del testo. Grazie a questa condizione è possibile calcolare il valore più grande di k (con $k < m$) tale che $d_m = 1$, da poter sfruttare per effettuare l'allineamento successivo in modo efficiente, così come avviene ad esempio con la funzione prefisso nell'algoritmo di Knuth-Morris-Pratt. Infatti il vantaggio principale di BNDM rispetto a Shift-Or è che mentre quest'ultimo legge tutti i caratteri del testo, il primo salta quei caratteri che certamente non producono un match. La condizione per cui $k < m$ è giustificata dal fatto che se $d_m = 1$ per $k = m$ allora P occorre alla posizione pos di T, ovvero dall'equazione 2.6 si ottiene

$$P[0...m-1] = T[pos...pos + m - 1] \tag{2.7}$$

Così come nello Shift-Or, per effettuare le transizioni degli stati dell'automa in tempo costante l'algoritmo utilizza una tabella B che viene costruita nella fase di preelaborazione. Sia $\Sigma = \{c_1, ..., c_{|\Sigma|}\}$, per ogni carattere $c_i \in \Sigma$ viene costruita una maschera di bit $B[c_i] = b_0 b_1 ... b_{m-1}$ dove

$$b_j = \begin{cases} 1 & \text{se } c_i = P[j] \\ 0 & \text{altrimenti} \end{cases} \tag{2.8}$$

`B[a] = 10100`	`B[b] = 01010`	`B[c] = 00001`	`B[d] = 00000`

Tabella 2.3: esempio dei valori di B calcolati per $P = ababc$ e $\Sigma = \{a, b, c, d\}$ nell'algoritmo BNDM

Per aggiornare gli stati dell'automa è sufficiente fare un AND logico tra la rappresentazione D dell'automa e la maschera di bit corrispondente al carattere letto dal testo, e quindi fare uno shift a sinistra di 1 bit. In formula:

$$D' = (D \ \& \ B[T[j]]) \ll 1 \tag{2.9}$$

I controlli sul bit d_m vanno fatti prima di effettuare lo shift, poiché quest'ultimo serve solamente per preparare l'automa alla lettura del carattere successivo.

L'algoritmo 8 fornisce la procedura per la costruzione della tabella B la cui complessità è $\Theta(m + |\Sigma|)$. L'algoritmo 9 fornisce la procedura per la ricerca. Ogni volta che lo stato accentante dell'automa è attivo viene aggiornato il

valore *last*, che verrà poi usato per calcolare la posizione nel testo a cui allineare il primo carattere del pattern. La ricerca termina nel momento in cui l'automa ha tutti gli stati non attivi, condizione per cui lo stesso non è più in grado di effettuare ulteriori transizioni. La complessità della ricerca è di $\Theta(n/m)$ nel caso migliore (ad esempio dove $P = a^m$ e $T = b^n$), di $\Theta(n \cdot m)$ nel caso peggiore (ad esempio dove $P = a^m$ e $T = a^n$) e di $O((n/m) \cdot log_{|\Sigma|} m)$ nel caso medio.

Algoritmo 8 BNDM-PREPROCESS(P, Σ)

```
1: m ← length[P]
2: for all c ∈ Σ do
3:     B[c] ← 0
4: end for
5: s ← 1
6: for i ← m − 1 to 0 do
7:     B[P[i]] ← B[P[i]] | s
8:     s ← s ≪ 1
9: end for
10:
11: return B
```

2.5 L'algoritmo SBNDM

SBNDM è una variante dell'algoritmo BNDM proposta da Hannu Peltola e Jorma Tarhio [PT03]. L'idea che sta alla base di questa variante è la semplificazione della logica dell'algoritmo di ricerca allo scopo di eseguire un minor numero di istruzioni nel ciclo principale. Nonostante questa modifica possa portare ad allineamenti del pattern meno efficienti rispetto all'algoritmo originale, gli autori nei loro test hanno rilevato in alcuni casi prestazioni fino al 10% superiori rispetto a BNDM.

Ad ogni iterazione k l'algoritmo BNDM verifica se l'automa si trova nel suo stato accentante (algoritmo 9 riga 11). Questo controllo è indispensabile nell'ultima iterazione (la m-esima) per verificare se il pattern occorre per intero nel testo, mentre in tutte le iterazioni precedenti serve soltanto a verificare se la lunghezza dello spostamento successivo deve essere ridotta in modo

Algoritmo 9 BNDM(T, P, B)

```
1:  n ← length[T]
2:  m ← length[P]
3:  pos ← 0
4:  while pos ≤ n − m do
5:     j ← m
6:     last ← m
7:     D ← ~0
8:     while D ≠ 0 do
9:        D ← D & B[T[pos + j − 1]]
10:       j ← j − 1
11:       if D & 10^(m−1) ≠ 0 then
12:          if j > 0 then
13:             last ← j
14:          else
15:             report match at pos
16:          end if
17:       end if
18:    end while
19:    pos ← pos + last
20: end while
```

tale da allineare il pattern con il suo prefisso più lungo all'interno del testo. SBNDM invece ad ogni iterazione riduce lo spostamento di 1 indipendentemente dal fatto che l'automa si trovi in uno stato accentante oppure no. Con questa modifica la correttezza dell'algoritmo rimane garantita in quanto non è possibile saltare shift validi (gli spostamenti fatti da SBNDM sono sempre più piccoli o uguali a quelli fatti da BNDM). Tuttavia l'algoritmo necessita di un nuovo parametro (che indichiamo con s_0) che è la lunghezza del più lungo prefisso del pattern che è anche suo suffisso. Questo valore viene utilizzato per calcolare il nuovo allineamento a seguito di un match (senza di esso dopo un match verrebbe fatto uno spostamento unitario).

La procedura di preelaborazione è la stessa di quella utilizzata nell'algoritmo BNDM. In più va calcolato il valore di s_0 e questo può essere fatto in tempo $\Theta(m)$ applicando ad esempio una versione leggermente modificata di BNDM al pattern stesso (utilizzando la tabella B già calcolata per SBNDM) e recuperando il valore della variabile $last$. L'algoritmo 10 fornisce la procedura per la ricerca del pattern con SBNDM. Essa ha la stessa complessità asintotica di BNDM, anche se necessita di una fase di preelaborazione aggiuntiva per il calcolo di s_0. Come si può notare la logica nel ciclo principale risulta semplificata rispetto a quella dell'algoritmo 9.

Algoritmo 10 SBNDM(T, P, B, s_0)

```
1:  n ← length[T]
2:  m ← length[P]
3:  pos ← 0
4:  while pos ≤ n − m do
5:     j ← m
6:     D ← ~0
7:     repeat
8:        D ← (D ≪ 1) & B[T[pos + j − 1]]
9:        j ← j − 1
10:    until (D = 0) | (j = 0)
11:    if D ≠ 0 then
12:       report match at pos
13:       pos ← pos + s_0
14:    else
15:       pos ← pos + j + 1
16:    end if
17: end while
```