

String matching con automi a stati finiti

Lorenzo Modica

Algoritmi e Laboratorio

Indice

1.Introduzione

Il problema dello string matching

Possibili soluzioni

Soluzione naive

2.Automa a stati finiti

Definizione

Esempi generici

Implementazione

3.String matching con ASF

Descrizione della soluzione

Pseudocodice

Implementazione

Esempi

4. Ulteriori funzionalità

Confronto

Gestione file di esempio

1. Introduzione

Il problema dello string matching

Il termine *string matching*, anche detto *pattern matching*, si riferisce al processo che permette di individuare tutte le occorrenze di una stringa, cioè una sequenza di caratteri, all'interno di un testo.

Tale testo potrebbe essere di dimensioni notevoli, ad esempio un documento in cui bisogna trovare una determinata parola o una sequenza di DNA da analizzare, per cui è necessario che si applichino algoritmi efficienti capaci di migliorare significativamente le tempistiche per ottenere il risultato.

Al fine di descrivere il problema formalmente, diremo che:

- Σ sia un alfabeto finito, ad esempio $\Sigma = \{a, b\}$.
- Σ^* sia l'insieme di tutte le possibili stringhe ottenute utilizzando i caratteri dell'alfabeto Σ .
- $T \in \Sigma^*$ sia il testo, di lunghezza $n \in \mathbb{N}$.
- $P \in \Sigma^*$ sia il pattern (la stringa), di lunghezza $m \in \mathbb{N}$, con $m \leq n$.
- ε sia una stringa di lunghezza 0, detta *stringa nulla*; si ha che $\varepsilon \in \Sigma^*$.

Inoltre, avremo che la lunghezza di una stringa x è indicata con $|x|$ e che la concatenazione di due stringhe x e y , indicata con xy , ha lunghezza di $|x| + |y|$ ed è composta dai caratteri di x seguiti dai caratteri di y .

La relazione $w \sqsubset x$, tra due stringhe $w, x \in \Sigma$, indica che w è un **prefisso** di x , cioè si avrà che $x = wy$ per qualche stringa $y \in \Sigma^*$.

Invece, diremo che w è **suffisso** di x , indicandolo con $w \sqsupset x$, se $x = yw$ per qualche stringa $y \in \Sigma^*$.

Entrambe le relazioni, \sqsubset e \sqsupset sono transitive.

Ad esempio, si ha che $p \sqsubset \text{pane}$ e che $\text{pane} \sqsubset \text{panettone}$, quindi $p \sqsubset \text{panettone}$.

Allo stesso modo, $\text{nettone} \sqsupset \text{panettone}$ e $\text{one} \sqsupset \text{nettone}$, quindi $\text{one} \sqsupset \text{panettone}$.

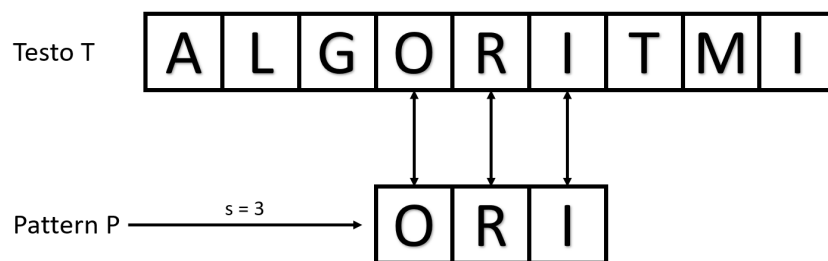
La stringa nulla è suffisso e prefisso di qualsiasi stringa; infatti, si ha che $\varepsilon \sqsubset x$, per qualsiasi

$x \in \Sigma^*$, ponendo, nella condizione del prefisso $x = wy$, $x = y$ e $\varepsilon = w$.

Per comodità, si indicherà con S_k il prefisso di k caratteri di una generica stringa S .

Ad esempio, S_0 corrisponde a ε e $S_{\dim S}$ corrisponde alla stringa S ($\dim S$ è la dimensione di S).

Quindi, il problema dello string matching può essere definito come la ricerca di tutte le possibili occorrenze di s , detti *spostamenti*, nell'intervallo $0 \leq s \leq n - m$ tali che $P \sqsupset T_{s+m}$.



L'intervallo di validità di s è ridotto a $n - m$, e non n , dato che il prefisso ottenuto dal testo T al massimo può avere una dimensione pari alla dimensione del testo stesso, non superiore.

Figura 1

La figura 1 presenta un esempio di string matching tra un pattern P , "ORI", e un testo T , "ALGORITMI".

Al quarto tentativo, con $s = 3$, lo spostamento risulta essere valido in quanto P è suffisso di $T_{s+m} = T_{3+3=6}$, che corrisponde a "ALGORI", cioè $ORI \sqsupset ALGORI$.

Per il suddetto esempio l'unico spostamento corretto è $s = 3$, ma ciò non vieta, in altre situazioni, di poter trovare più occorrenze valide all'interno dello stesso testo.

Possibili soluzioni

Esistono molteplici algoritmi di *string matching* con complessità differenti, calcolati in funzione della lunghezza del testo e della lunghezza del pattern, rispettivamente n e m .

Tali complessità sono ricavate dalla somma delle complessità di due fasi distinte che caratterizzano l'algoritmo: preelaborazione e matching.

Lo stadio di preelaborazione prevede una serie di operazioni effettuate sul pattern in modo da rendere più celere l'esecuzione della fase successiva, che invece riguarda l'effettiva ricerca degli spostamenti validi.

Ad esempio, uno dei possibili algoritmi di string matching prevede, durante la fase di preelaborazione, la generazione di un automa a stati finiti(ASF) appositamente progettato per ricercare le occorrenze di un dato pattern P in un testo generico.

Non tutti gli algoritmi, però, prevedono una fase di preelaborazione.

L'algoritmo ingenuo di forza bruta(*naive*), è uno di questi. Esso, di fatti, prova ogni possibile spostamento, classificandolo valido in caso di matching.

Di seguito, il testo T e il pattern P verranno trattati come degli array, $T[1 .. n]$ e $P[1 .. m]$, di lunghezza n e m .

Soluzione naive

L'algoritmo naive trova tutti gli spostamenti verificando iterativamente se, per ciascuno degli $n - m + 1$ valori possibili di s , la condizione $P[1 .. m] = T[s + 1 .. n]$ è vera.

Di seguito viene riportato lo pseudocodice della procedura.

NAIVE-STRING-MATCHER(T, P)

```
1      n = T.length
2      m = P.length
3      for s = 0 to n - m
4          if P[1 .. m] == T[s + 1 .. s + m]
5              stampa "spostamento valido: " + s
```

Nelle prime due righe vengono assegnate le lunghezze del testo T e del pattern P alle rispettive variabili n e m .

Nella riga 3, il ciclo for considera ogni possibile spostamento e, all'interno del ciclo(righe 4 e 5), si valuta se il singolo spostamento è effettivamente valido.

La condizione nella riga 4 richiede un ciclo per essere effettuato in quanto bisogna confrontare i caratteri nelle posizioni corrispondenti fino a quando si sono confrontate tutte le posizioni o si verifica un *mismatch*(i due caratteri confrontati non corrispondono).

Tale ciclo implicito ha complessità lineare in funzione di m , cioè $O(m)$, e si ha che:

- Il caso ottimo è costante e avviene quando le due stringhe confrontate non corrispondono a partire dal primo carattere, quindi si avrà subito un mismatch.
- Il caso pessimo si verifica quando lo spostamento è valido, cioè quando le due stringhe confrontate coincidono, perché il ciclo dovrà essere eseguito m volte per stabilire se lo spostamento è valido.

La procedura NAIVE-STRING-MATCHER richiede, in totale, un tempo di $O((n - m + 1)m)$, dove $n - m + 1$ è il numero di volte che viene eseguito il ciclo interno, che ha complessità pari a $O(m)$.

La complessità nel caso peggiore è quindi $\Theta(n^2)$ se $m = n/2$.

Segue il codice relativo all'implementazione della procedura.

```
#include<iostream>
#include<vector>

using namespace std;

bool string_match(string a, string b){
    return (!a.compare(b));
}

vector<int>* naive_string_matching(string T, string p){
    vector<int>* sol = new vector<int>();
    int n = T.length();
    int m = p.length();
    for(int s = 0; s < n - m + 1; s++){
        if(string_match(T.substr(s,m),p))
            sol->push_back(s);
    }
    return sol;
}
```

Il codice sopraindicato è presente nel file header “naive_string_matching.h”.

Nello stesso folder è situato un test case per provare l'implementazione dell'algoritmo.

Tale algoritmo non rappresenta una soluzione efficiente per il problema dello string matching a differenza di altri algoritmi che, invece di ignorare possibili informazioni estrapolate dalla stringa

P, le usano per migliorare le prestazioni nella fase di matching.

Un algoritmo che utilizza queste informazioni a suo vantaggio è l'algoritmo di string matching con ASF.

Prima di descrivere questo algoritmo viene proposta una breve trattazione generale degli automi a stati finiti.

2. Automa a stati finiti

Definizione

Un automa a stati finiti viene rappresentato tramite una 5-upla $(Q, q_0, A, \Sigma, \delta)$, dove:

- Q è un insieme finito di **stati**
- q_0 è lo **stato iniziale**
- $A \subseteq Q$ è l'insieme degli **stati accettanti**
- Σ è un alfabeto di input finito
- δ è una funzione da $Q \times \Sigma$ in Q , detta **funzione di transizione** di M

La funzione di transizione può essere rappresentata graficamente tramite una tabella o tramite un diagramma con nodi e archi.

L'ASF inizia la sua esecuzione nello stato iniziale q_0 e riceve in input una sequenza di caratteri facenti parte dell'alfabeto Σ che legge uno alla volta.

Ad ogni carattere c letto l'automa effettua una transizione dallo stato in cui si trova q allo stato $\delta(q, c)$.

Inoltre, se lo stato corrente q :

- Appartiene a A , si dice che l'automa ha **accettato** la stringa finora letta.
- Non appartiene a A , l'input letto viene **rifiutato**.

Inoltre, indichiamo con φ una funzione detta **funzione stato finale**, da Σ^* a Q tale che $\varphi(w)$,

con $w \in \Sigma^*$, è lo stato in cui finisce l'ASF dopo aver ispezionato la stringa w .

Quindi, l'automa accetta w se e soltanto se $\varphi(w) \in A$.

La funzione φ è definita dalla relazione ricorsiva

$$\varphi(\varepsilon) = q_0$$

$$\varphi(wc) = \delta(\varphi(w), c) \quad \text{per } w \in \Sigma^*, a \in \Sigma$$

Esempi generici

Vengono presentati due esempi di ASF:

1. Viene creato un ASF M1 il cui scopo è di accettare le stringhe con un numero pari di 'a'.

M1 è definito dalla 5-tupla $(Q, q_0, A, \Sigma, \delta)$, dove:

- $Q = \{0, 1\}$
- $q_0 = 0$
- $A = \{0\}$
- $\Sigma = \{a, b\}$
- δ viene definita sulla base della seguente tabella:

δ	a	b
0	1	0
1	0	1

Input d'esempio: aaba

Passaggi per l'esecuzione della funzione stato finale sull'input d'esempio:

$\varphi(aaba)$
 $\delta(\varphi(aab), a)$
 $\delta(\delta(\varphi(aa), b), a)$
 $\delta(\delta(\delta(\varphi(a), a), b), a)$
 $\delta(\delta(\delta(\delta(\varphi(\varepsilon), a), a), b), a)$
 $\delta(\delta(\delta(\delta(0, a), a), b), a)$
 $\delta(\delta(\delta(1, a), b), a)$
 $\delta(\delta(0, b), a)$
 $\delta(0, a) = 1$
 $1 \notin A$, quindi l'automa rifiuta l'input.

2. Viene creato un ASF M2 il cui scopo è di accettare le stringhe che terminano con 'abc'.

M2 è definito dalla 5-tupla $(Q, q_0, A, \Sigma, \delta)$, dove:

- $Q = \{0, 1, 2, 3\}$
- $q_0 = 0$
- $A = \{3\}$
- $\Sigma = \{a, b, c\}$
- δ viene definita sulla base della seguente tabella:

δ	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	1	0	0

Input d'esempio: cababc

Passaggi per l'esecuzione della funzione stato finale sull'input d'esempio:

```

 $\varphi(cababc)$ 
 $\delta(\varphi(cabab), c)$ 
 $\delta(\delta(\varphi(caba), b), c)$ 
 $\delta(\delta(\delta(\varphi(cab), a), b), c)$ 
 $\delta(\delta(\delta(\delta(\varphi(ca), b), a), b), c)$ 
 $\delta(\delta(\delta(\delta(\delta(\varphi(c), a), b), a), b), c)$ 
 $\delta(\delta(\delta(\delta(\delta(\delta(\varphi(\varepsilon), c), a), b), a), b), c)$ 
 $\delta(\delta(\delta(\delta(\delta(0, c), a), b), a), b), c)$ 
 $\delta(\delta(\delta(\delta(0, a), b), a), b), c)$ 
 $\delta(\delta(\delta(1, b), a), b), c)$ 
 $\delta(\delta(2, a), b), c)$ 
 $\delta(\delta(1, b), c)$ 
 $\delta(2, c) = 3$ 
 $3 \in A$ , quindi l'automa accetta l'input

```

Implementazione

Di seguito si presenta una versione parziale e semplificata del codice presente nel file “automata.h”.

```

#include<iostream>
#include<unordered_map>
#define MISSING 0
using namespace std;
class automata{
private:
    char* sigma;
    int Q;
    int **delta;
    int s0;
    int *F;
    int dim_sigma;
    int dim_F;
    unordered_map<char,int> indexOf;

```

```

    bool isFinal(int state){
        for(int i=0;i<dim_F;i++)
            if(state == F[i])
                return true;
        return false;
    }

public:
    //valuta la stringa in input
    bool evaluate(string s){
        int q = s0;//stato corrente
        for(int i=0;i<s.length();i++)
            q = delta[q][indexOf[s.at(i)]];
        return isFinal(q);
    }
};

```

Nella porzione di codice riportato sono presenti soltanto due metodi, **isFinal** e **evaluate**. **isFinal** prende in input uno stato e ritorna *true* se tale stato fa parte degli stati accettanti, altrimenti ritorna *false*.

Evaluate prende in input una stringa *s* e su di essa calcola, in maniera iterativa, ciò che viene fatto ricorsivamente tramite la funzione stato finale, quindi ritorna in output un boolean che indica se l'automata ha accettato o meno la stringa letta.

Inoltre, è presente un file, “esempi_automata1.cpp”, che permette di testare la classe *automata* implementata in “automata.h”.

Sebbene gli esempi previsti in “esempi_automata1.cpp” siano due, viene creata una sola istanza di *automata(a1)* che muta la propria funzione di transizione δ sulla base dell'esempio da rappresentare.

Tale passaggio dalla funzione di transizione del primo esempio alla funzione del secondo esempio(e viceversa) è reso possibile dal metodo *change_delta*, che consente di modificare, richiamandolo una volta per ogni possibile combinazione *stato* – *carattere*, l'intera funzione di transizione.

Il metodo richiede 3 parametri: lo stato *q*, il carattere *c* e il nuovo valore di $\delta(q, c)$.

L'utente può invocare le modifiche che permettono di eseguire un altro esempio scrivendo il comando “change”.

In alternativa, se *a1* è impostato sull'esempio desiderato, l'utente può direttamente scrivere il testo *s* da dare in input all'automata tramite il metodo *evaluate*.

Nel primo esempio proposto viene generato un automa il cui compito è accettare le stringhe contenenti un numero pari di 'a', nel secondo esempio una stringa viene accettata se contiene un numero pari di 'b'.

L'immagine successiva raffigura un'esecuzione di prova di "esempi_automata1.cpp".

```
--Automa che accetta le stringhe con un numero pari di 'a'--
bbbbaaabb
La stringa non e' accettata
aaaab
La stringa e' accettata
change
--Automa che accetta le stringhe con un numero pari di 'b'--
b
La stringa non e' accettata
ababa
La stringa e' accettata
stop

-----
Process exited after 48.54 seconds with return value 0
Premere un tasto per continuare . . .
```

Inizialmente l'automa accetta stringhe con un numero pari di 'a'. Infatti, inserendo la stringa "bbbbaaabb" l'automa rifiuta la stringa, mentre inserendo "aaaab", che contiene un numero pari di 'a', l'automa l'accetta.

In seguito, viene inserito il comando *change* per passare da un esempio all'altro.

Adesso l'automa è impostato per riconoscere un numero pari di 'b'.

Quindi l'automa rifiuta "b" e accetta "ababa".

Per terminare l'esecuzione del ciclo while, e quindi del programma, si può scrivere "stop".

3.String matching con ASF

Descrizione della soluzione

Gli automi di string matching esaminano un carattere del testo una sola volta, impiegando un tempo costante per ogni carattere. Il tempo di matching impiegato, dopo il tempo di preelaborazione del pattern per costruire l'automa, è quindi $\Theta(n)$.

Tuttavia, il tempo per costruire l'automa può essere grande, se l'alfabeto Σ è grande.

Per specificare l'automa di string matching che corrisponde a un dato pattern $P[1..m]$, definiamo prima una funzione ausiliaria σ , detta **funzione suffisso**, associata a P . La funzione σ è una corrispondenza da Σ^* a $\{0, 1, \dots, m\}$ tale che $\sigma(x)$ è la lunghezza del prefisso più lungo di P che è un suffisso di x :

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

Da notare che per un pattern P di lunghezza m , si ha $\sigma(x) = m$ se e soltanto se $P \sqsupset x$.

Inoltre, dalla definizione della funzione suffisso segue che, se $x \sqsupset y$, allora $\sigma(x) \leq \sigma(y)$.

Definiamo l'automa di string matching che corrisponde a un dato pattern $P[1..m]$ nel modo seguente.

- L'insieme degli stati Q è $\{0, 1, \dots, m\}$. Lo stato iniziale q_0 è lo stato 0; lo stato m è l'unico stato accettante. Allo stesso tempo, lo stato $i \in Q$ sta ad indicare che mancano esattamente $m - i$ caratteri (corretti) affinché l'automa rilevi uno spostamento valido. Quindi, tale sequenza di caratteri mancanti, indichiamola con L , di lunghezza $k = m - i$ deve essere suffisso di P , cioè $L \sqsupset P$.

- La funzione di transizione δ è definita nel seguente modo:

$$\delta(q, a) = \sigma(P_q a) \quad \forall q \in Q, \forall a \in A$$

L'intento è di sapere, ad ogni passo, la lunghezza del più lungo prefisso del pattern P che fino a quel momento è suffisso di T .

Avendo già letto l' i -esimo carattere di T , se l'automa è nello stato q , ovvero se la proposizione $P_q \sqsupset T_i \wedge \neg \exists k > q: P_k \sqsupset T_i$ è vera, una volta letto il prossimo carattere $T[i + 1] = a$ vogliamo che il valore dello stato dell'automa venga aggiornato alla lunghezza del prefisso più lungo di P che è suffisso di $T_{i+1} = T_i a$, cioè $\sigma(T_i a)$.

Il valore ottenuto con $\sigma(T_i a)$ equivale a $\sigma(P_q a)$ dato che P_q è il più lungo prefisso di P che è suffisso di T_i , tale equivalenza verrà dimostrata nel Lemma 3.2.

Quindi, sostituendo $\sigma(T_i a)$ con $\sigma(P_q a)$, possiamo calcolare la funzione di transizione

esclusivamente in funzione del pattern P (come previsto), senza dover considerare il testo T su cui, in seguito, verrà effettuato il matching.

Ci sono due possibili situazioni, sulla base del valore di $T[i + 1] = a$:

- $a = P[q + 1]$; quindi il carattere a continua a coincidere con il pattern; in questo caso l'automa passerà dallo stato q allo stato $q+1$, cioè avremo che $\sigma(T_i a) = \sigma(T_i) + 1$.
- $a \neq P[q + 1]$; quindi a non coincide con il pattern. In questo caso si deve trovare un prefisso di P più corto che sia anche suffisso di $T_i a$. Ciò comporta che $\sigma(T_i a) \leq \sigma(T_i)$.

In generale, considerando entrambe le situazioni, vale la disequazione $\sigma(T_i a) \leq \sigma(T_i) + 1$, che tratteremo nel lemma successivo.

Lemma 3.1 (Disuguaglianza della funzione suffisso)

Per ogni stringa x e carattere a , si ha $\sigma(xa) \leq \sigma(x) + 1$.

Dimostrazione

Poniamo $r = \sigma(xa)$. Se $r = 0$, allora la condizione $\sigma(xa) = r \leq \sigma(x) + 1$ è banalmente soddisfatta, perché il valore di $\sigma(x)$ non è mai negativo. Supponiamo che $r > 0$.

Si ha $P_r \sqsupset xa$, per la definizione di σ . Quindi $P_{r-1} \sqsupset x$, escludendo la a alla fine di P_r e alla fine di xa . Ne consegue che $r - 1 \leq \sigma(x)$, perché $\sigma(x)$ è il valore massimo di k tale $P_k \sqsupset x$, quindi

$$\sigma(xa) = r \leq \sigma(x) + 1.$$

Lemma 3.2 (Ricorsione della funzione suffisso)

Per ogni stringa x e carattere a , se $q = \sigma(x)$, allora $\sigma(xa) = \sigma(P_q a)$.

Dimostrazione

Dalla definizione di σ , cioè $\sigma(x) = \max\{k : P_k \sqsupseteq x\}$, si ha che P_q è il prefisso di P più lungo ad essere suffisso di x . Si ha anche $P_q a \sqsupseteq xa$. Se poniamo $r = \sigma(xa)$, allora $r \leq q + 1$ per il Lemma 3.1. Poiché $P_q a \sqsupseteq xa$, $P_r \sqsupseteq xa$ e $|P_r| \leq |P_q a|$, abbiamo che $P_r \sqsupseteq P_q a$. Quindi $r = \sigma(xa) \leq \sigma(P_q a)$; ma si ha anche $\sigma(P_q a) \leq \sigma(xa)$, perchè $\sigma(P_q a) \sqsupseteq xa$. Dunque $\sigma(xa) = \sigma(P_q a)$.

Segue un teorema che dimostra che l'automa tiene costantemente traccia, tramite il proprio stato, del più lungo prefisso del pattern che è un suffisso di quanto è stato letto fino a quel momento. In altre parole, per tutto il tempo l'automa conserva l'invariante $\varphi(T_i) = \sigma(T_i)$.

Teorema 3.3

Se φ è la funzione dello stato finale di un automa di string matching per un dato pattern P e $T[1..n]$ è un testo di input per l'automa, allora $\varphi(T_i) = \sigma(T_i)$ per $i = 0, 1, \dots, n$.

Dimostrazione

La dimostrazione è per induzione su i . Per $i=0$, il teorema è banalmente vero, perché $T_0 = \varepsilon$. Quindi, $\varphi(\varepsilon) = \varphi(T_0)$ è 0 dato che lo stato iniziale è 0 e non avverrà nessuna transizione e $\sigma(\varepsilon) = \sigma(T_0)$ è pure 0 perché l'unico prefisso di P ad essere suffisso di ε è P_0 , cioè ε stesso.

Adesso supponiamo che $\varphi(T_i) = \sigma(T_i)$ e proviamo che $\varphi(T_{i+1}) = \sigma(T_{i+1})$.

Indichiamo $\varphi(T_i)$ con q e $T[i + 1]$ con a . Allora si ha

$$\begin{aligned}
 \varphi(T_{i+1}) &= \varphi(T_i a) && \text{(per le definizioni di } T_{i+1} \text{ e } a) \\
 &= \delta(\varphi(T_i), a) && \text{(per la definizione ricorsiva di } \varphi) \\
 &= \delta(q, a) && \text{(per la definizione di } q) \\
 &= \sigma(P_q a) && \text{(per la definizione di } \delta) \\
 &= \sigma(T_i a) && \text{(per il lemma 3.2 e per induzione)} \\
 &= \sigma(T_{i+1}) && \text{(per la definizione di } T_{i+1})
 \end{aligned}$$

Quindi, per tale teorema, se l'automa si trova nello stato q , allora q è il più grande valore tale che $P_q \sqsupseteq T_i$. Quindi, si ha $q = m$ se e soltanto se è stata appena ispezionata un'occorrenza del pattern P .

Pseudocodice

Viene riportato lo pseudocodice della funzione di preelaborazione e della procedura di matching. Il seguente codice permette di calcolare la funzione di transizione δ da un dato pattern $P[1..m]$.

COMPUTE-TRANSITION-FUNCTION(P, Σ)

```

1      m = P.length
2      for q = 0 to m
3          for ogni carattere a ∈ Σ
4              k = min(m, q + 1) + 1
5              repeat
6                  k = k - 1
7              until  $P_k \sqsupseteq P_q a$ 
8                   $\delta(q, a) = k$ 
9      return  $\delta$ 
```

I cicli annidati nelle righe 2 e 3 considerano ogni combinazione stato-carattere possibile in modo da generare una funzione di transizione δ completa.

Le righe da 4 a 8 assegnano a $\delta(q, a)$ il più grande valore di k tale che $P_k \sqsupseteq P_q a$.

Il codice inizia con il massimo valore possibile di k , che è il minimo tra la dimensione del pattern P , cioè m , che corrisponde con lo stato più grande che l'automa può assumere e $q+1$, che sarebbe lo stato successivo a quello in cui si trova l'automa.

A tale valore di k viene aggiunto 1 perchè il ciclo seguente prevede che k venga decrementato prima di effettuare il controllo $P_k \sqsupseteq P_q a$.

Il tempo di esecuzione della funzione è $O(m^3 |\Sigma|)$, dato che i cicli esterni contribuiscono con un fattore $m|\Sigma|$, il ciclo **repeat** interno può essere eseguito al più $m + 1$ volte e il controllo $P_k \sqsupseteq P_q a$ richiede un controllo implicito sui carattere che può prevedere fino a m confronti.

La successiva procedura sfrutta la funzione di transizione δ , calcolata dalla funzione COMPUTE-TRANSITION-FUNCTION, per effettuare la fase di matching, cioè trovare le occorrenze del pattern P in un testo di input $T[1..n]$.

FINITE-AUTOMATON-MATCHER(T, δ, Σ)


```

1      n = T.length
2      q = 0
3      for i = 1 to n
4          q =  $\delta(q, T[i])$ 
5          if q == m
6              stampa "Occorrenza del pattern con spostamento" i - m

```

Nella funzione è previsto un solo ciclo, eseguito n volte, le restanti istruzioni richiedono tempo costante. Quindi il suo tempo di esecuzione è lineare in funzione della lunghezza del testo in input T , cioè $\Theta(n)$.

Implementazione

L'implementazione completa dello String Matching Automata (*SMA*) si trova nel file "string_matching_automata.h".

La classe *SMA* è una specializzazione della classe *automata*, dato che *SMA* è anch'esso un automa, ma con l'intento specifico di risolvere il problema dello string matching.

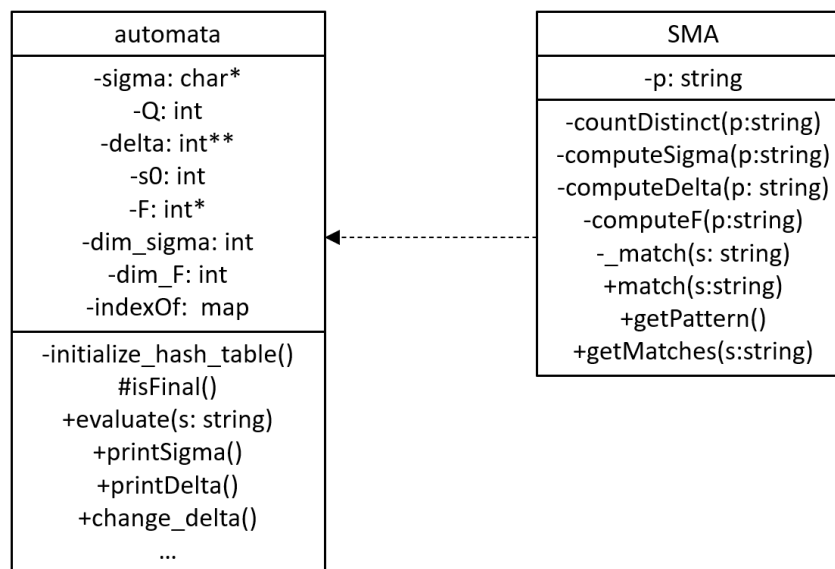


Figura 2

La figura 2 rappresenta l'UML che descrive la classe madre(*automata*), la classe figlia(*SMA*) e la relazione di ereditarietà che le lega.

Mentre la superclasse definisce gli attributi fondamentali per la costruzione di un automa, come gli elementi della 5-upla e le dimensioni dei relativi array, la sottoclasse *SMA* si limita ad aggiungere un solo attributo al set di attributi ereditati, cioè il pattern dell'automata *p*.

Inoltre, *SMA* prevede molteplici metodi:

- *CountDistinct*, viene utilizzato per contare il numero di caratteri distinti nel pattern *p*, serve per sapere la dimensione dell'array *sigma*, cioè $|\Sigma|$.
- *ComputeSigma*, *ComputeDelta* e *ComputeF* hanno il compito di calcolare rispettivamente l'alfabeto Σ , la funzione di transizione δ e l'insieme F .
- *_match*, *match* e *getMatches* hanno lo scopo di segnalare gli spostamenti validi del pattern *p* nel testo in input.
- *getPattern* ritorna il pattern *p*.

Di seguito viene riportato il codice di *ComputeDelta*, il cui obiettivo è calcolare δ .

```

1      int** computeDelta(string p){
2          int _Q = getQ();
3          char* _sigma = getSigma();
4          int n_sigma = getSigmaDim();
5          int** delta = new int*[_Q];

6          for(int i = 0; i < _Q; i++){
7              int k = i < p.size()-1 ? i : p.size()-1;
8              string sub_p = p.substr(0, 1 + k);
9              delta[i] = new int[n_sigma + 1];
10             delta[i][0] = 0;
11             for(int j = 0; j < n_sigma; j++){
12                 string str = p.substr(0,i);
13                 str += _sigma[j];
14                 delta[i][j + 1] = bestSubSuffix(sub_p, str);
15             }
16         }
17         return delta;
18     }
```

L'array *delta*, per ogni stato *i*, riserva una posizione dedicata a tutti i caratteri presenti nel testo che non fanno parte dell'alfabeto Σ . Tale posizione è sempre *delta[i][0]* e viene posta a 0 dato

che il carattere in questione non è sicuramente presente in p , quindi non c'è alcun prefisso di p , eccetto ϵ , ad essere suffisso del testo finora letto. Abbiamo che $\epsilon = p_0$ quindi, in questi casi, il nuovo stato dell'automa sarà 0.

Come si può notare a riga 14, il valore di `delta[i][j+1]` viene deciso dalla funzione *bestSubSuffix*, il cui comportamento è analogo a ciò che avviene in COMPUTE-TRANSITION-FUNCTION nelle righe 5,6 e 7: controlla qual è il prefisso più grande di p che è suffisso della stringa ottenuta dalla concatenazione tra il prefisso di p , di lunghezza pari allo stato attuale, e un carattere dell'alfabeto Σ (scelto dall'array `_sigma` in posizione j).

```
//confronta due stringhe: ritorna true se a è suffisso di b, 0 altrimenti
bool isSuffix(string a, string b){
    for(int i = 1; i <= a.length(); i++)
        if(a.at(a.length() - i) != b.at(b.length() - i))
            return false;
    return true;
}

/*Trova la dimensione della più grande sottostringa di sf che è suffisso
di p*/
int bestSubSuffix(string sf, string p){
    for(int i = sf.length(); i > 0; i--)
        if(isSuffix(sf.substr(0,i), p))
            return i;
    return 0;
}
```

La funzione *bestSubSuffix* effettua delle chiamate a *isSuffix*, che verifica se una stringa è suffisso di un'altra.

Sia *bestSubSuffix* che *isSuffix* si trovano in “suffixes.h”.

Il corrispondente di FINITE-AUTOMATON-MATCHER è `_match()`, permette di trovare le occorrenze di p nel testo s e salva i relativi spostamenti in un vettore di interi.

```

vector<int>* _match(string s){
    vector<int>* v = new vector<int>();
    int q = getS0();//stato iniziale(0)
    for(int i=0;i<s.length();i++){
        q = getDelta()[q][getIndexof(s.at(i))];
        if(isFinal(q)) v->push_back(i - p.size() + 1);
    }
    return v;
}

```

Il metodo *getIndexof* permette di ritornare il numero che identifica un determinato carattere $s[i]$ in modo da poter assegnare allo stato q il valore $\delta(q, s[i])$.

GetIndexof viene implementato interrogando un oggetto *indexof* di tipo `unordered_map<char,int>`.

Esempi

Il file “main.cpp” presenta degli esempi di string matching con automi a stati finiti.

Nel sorgente vengono create due istanze della classe SMA, a e b .

Il pattern assegnato al primo automa è "nanna"; tramite la funzione $a.printDelta()$ viene stampata a video la funzione di transizione δ di a , dove le colonne rappresentano i due caratteri dell'alfabeto, 'n' e 'a', mentre le righe rappresentano i possibili stati dell'automa, da 0 alla lunghezza del pattern.

Funzione di transizione di a:

	n	a
0)	1	0
1)	1	2
2)	3	0
3)	4	2
4)	1	5
5)	3	0

In seguito, tramite il metodo `match(s)`, viene eseguita la fase di matching sul testo "ninna nanna nonfj nannik nanannannana".

L'automa trova le due occorrenze di "nanna" presenti nel testo e le segnala specificando lo spostamento.

```
RICERCA DEL PATTERN: "nanna" NEL TESTO: "ninna nanna nonfj nannik nanannannana"
OCCORRENZE:
        6        29
```

4. Ulteriori funzionalità

Confronto

Al fine di confrontare i due algoritmi di string matching presentati, naive e con automa a stati finiti, è stato creato il file "compare.cpp", inserito nel folder "compare_algorithms".

Compare.cpp permette di eseguire i due algoritmi sugli stessi dati in input in modo da ottenere i risultati e i tempi di esecuzione.

Il programma darà la possibilità di scegliere tra un testo in input inserito dall'utente e dei file già presenti nelle cartelle `util\samples` e `util\longer_samples`.

La prima cartella presenta dei file di testo di esempio di lunghezza più contenuta rispetto alla seconda. Infatti, utilizzando uno dei file di testo presenti nella seconda cartella, il programma riporterà anche i tempi di esecuzione degli algoritmi (in microsecondi) e tramite un semplice confronto decreterà il più rapido.

Per la gestione dei file nelle cartelle si usa un'istanza della classe `samplesManager`, presente nel

file samplesManager.h.

Per la visualizzazione dei file presenti nelle cartelle e delle relative dimensioni viene utilizzato il metodo printStarryInfo().

Inoltre, il programma stampa a video ogni singolo spostamento valido trovato dall'algoritmo(sia per il Naive algorithm che per il Finite Automata algorithm) e li confronta per verificare che i due algoritmi abbiano ottenuto gli stessi risultati.

All'inizio, il programma chiede se si preferisce utilizzare un file di testo o inserire il testo da tastiera.

Specificando S o s, al prossimo passaggio si potrà scegliere tra dei file di testo di dimensioni notevoli o dei file di dimensioni minori, in questo caso scegliamo i file di dimensioni maggiori in modo da poter confrontare i tempi di esecuzione dei due algoritmi, come da figura.

```
Confronto tra algoritmi di string matching
Gli algoritmi considerati sono: Naive algorithm; Finite Automata algorithm;

Vuoi utilizzare dei campioni di testo prelevati da file?(S/N): S

Vuoi utilizzare dei campioni abbastanza grandi da poter fare considerazioni sul tempo di esecuzione degli algoritmi?(S/N): S
      FILE      DIMENSIONE
1) textalice29.txt      *
2) textasyoulik.txt      *
3) textlctet10.txt      ****
4) textplrabn12.txt      *****

Scegli l'id del file da selezionare:
```

Proseguiamo scegliendo il quarto file di testo, specificando il numero 4.

Ora è possibile inserire il pattern da ricercare nel testo, in questo esempio il pattern sarà la stringa “prof”.

Il programma prima eseguirà l'algoritmo naive, successivamente l'algoritmo con l'automa.

Come si potrà notare dall'immagine successiva, il programma stampa il numero di occorrenze, ogni singola occorrenza trovata e il tempo di esecuzione dell'algoritmo.

```
Gli algoritmi considerati sono: Naive algorithm; Finite Automata algorithm;

Vuoi utilizzare dei campioni di testo prelevati da file?(S/N): S

Vuoi utilizzare dei campioni abbastanza grandi da poter fare considerazioni sul tempo di esecuzione degli algoritmi?(S/N): S
      FILE      DIMENSIONE
1) textalice29.txt      *
2) textasyoulik.txt      *
3) textlctet10.txt      ****
4) textplrabn12.txt      *****

Scegli l'id del file da selezionare: 4

Inserisci pattern: prof

Naive algorithm:
Number of occurrences: 18
Occurrences: 1778      14067      20243      56998      57576      64360      76424      81802      128975      160548      160690      244213      254328      254522      285504      297898      334955      461930
Execution time: 58531

Finite Automata algorithm:
Number of occurrences: 18
Occurrences: 1778      14067      20243      56998      57576      64360      76424      81802      128975      160548      160690      244213      254328      254522      285504      297898      334955      461930
Execution time: 39256
```

Alla fine viene riportato che il tempo di esecuzione di Naive algorithm(58531 μ s) è maggiore rispetto al tempo di Finite Automata algorithm(39256 μ s), questa condizione si è presentata in quasi tutti i test da me svolti, facendo un particolare riferimento ai test effettuati con testi di grandi dimensioni e con l'alfabeto Σ di dimensioni ridotte, in cui la differenza in termini di tempo di esecuzione tra Naive algorithm e Finite Automata algorithm andava ad accentuarsi sempre più.

Infatti, una dimensione del testo abbastanza grande non implica un importante decremento delle prestazioni dell'automa dato che il tempo di matching nel testo è lineare($\Theta(n)$), come invece accade nell'algoritmo ingenuo.

Gestione file di esempio

Per gestire i file di esempio è stata scritta la classe samplesManager.

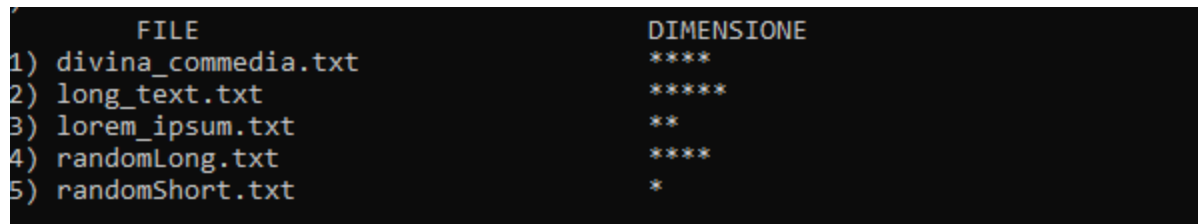
Lo scopo è di poter gestire l'utilizzo dei campioni di testo presenti nelle cartelle 'samples' e 'longer_samples' (ma samplesManager funziona con una directory generica, non si limita al nome di queste cartelle specifiche).

In questo modo, quando bisogna comparare gli algoritmi di string matching(specialmente per quanto riguarda i tempi di esecuzione) si potranno importare a run-time il testo che più si preferisce, potendo basarsi sulla dimensione dei vari file(espressa con gli asterischi, richiamando il metodo printStarryInfo()).

Così facendo si possono fare test utilizzando testi molto grandi e che contengono un alfabeto di caratteri alfanumerici vasto(interessante specialmente dal punto di vista dell'automa a stati finiti per quanto concerne il computo della funzione di transizione).

Ad ogni nuova esecuzione, l'istanza della classe va a prelevare i nomi dei file di testo(potrebbe

prendere anche altri tipi di file, ma non ne abbiamo bisogno) presenti nella cartella specificata. Quindi, in qualsiasi momento si possono inserire nuovi file campione per effettuare nuovi test.



```
FILE                                DIMENSIONE
1) divina_commedia.txt             *****
2) long_text.txt                   *****
3) lorem_ipsum.txt                 **
4) randomLong.txt                  *****
5) randomShort.txt                 *
```

L'immagine mostra l'esecuzione del metodo `printStarryInfo()` sulla cartella `samples`.

La classe `samplesManager` si trova nel file `"samplesManager.h"` nella cartella `"util"`.

Nella stessa cartella si trova un piccolo programmino, `"samplesManager.cpp"`, che permette di visualizzare, tramite `printStarryInfo()`, la lista dei file di testo presenti in `"samples"` e, indicando l'id di uno dei file, è possibile visualizzarne il contenuto.

Riferimenti

Introduzione agli algoritmi e strutture dati. Thomas H. Cormen, Charles E. Leiserson, Ronald L.

Automa a stati finiti. [Wikipedia](#)

Algoritmo di pattern matching su stringhe. [Wikipedia](#)