Università degli Studi di Udine

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in Informatica

Tesi di Laurea

Algoritmo di Landau-Vishkin: implementazione

Relatore:

Prof. Alberto Policriti

Laureando:

Gianluca Demartini

Anno Accademico 2002 - 2003

INDICE

1. Introduzione

Il confronto approssimato tra stringhe è un settore molto importante in biologia computazionale. Questa importanza è spiegata dal fatto che i dati che si utilizzano non possono essere sempre corretti e quindi è intrinseca la presenza di errori nei dati che si hanno riguardo le molecole, e dal fatto che gli allineamenti spesso forniscono importanti informazioni riguardo la funzione di geni e proteine, infatti una forte somiglianza tra sequenze biomolecolari (DNA, RNA, sequenze di amminoacidi) di solito comporta una certa somiglianza strutturale o funzionale tra esse. Ovviamente queste sequenze codificano e riflettono la più complessa struttura che si ha in realtà a livello della cellula.

Un esempio in cui è appropriato l'uso di limitare il numero di differenze nell'allineamento di due sequenze biologiche è quello della localizzazione di geni le cui mutazioni causano, o contribuiscono, a certe malattie genetiche. Vengono confrontate sequenze di DNA che rappresentano geni di persone malate e di persone sane per trovare le differenze consistenti dato che molte malattie genetiche sono causate da cambiamenti molto piccoli in un gene.

Per risolvere questi problemi sono stati prodotti diversi algoritmi prima di quello proposto da Landau e da Vishkin, come ad esempio quelli basati su tecniche di programmazione dinamica o quello proposto da E. Ukonnen con complessità comunque peggiori di quello qui studiato.

Si vuole implementare l'algoritmo descritto da Gad M. Landau e Uzi Vishkin nell'articolo intitolato "Fast String Matching with k differences" e pubblicato nel 1988 sul "Journal of Computer and System Sciences". L'obiettivo di questo algoritmo è di scoprire tutte le occorrenze di una stringa P in un testo T accettando al più *k* errori durante il confronto.

In questo documento verrà dapprima presentato il problema generale dell'allineamento tra stringhe con le necessarie definizioni dei concetti essenziali, e quindi verranno presentati degli algoritmi che sono poi stati migliorati nel tempo fino a descrivere in dettaglio l'algoritmo di Landau-Vishkin presentando quali sono i suoi vantaggi rispetto ai precedenti. In fine vi sarà una descrizione del codice prodotto per realizzare l'implementazione dell'algoritmo studiato.

Per produrre l'implementazione dell'algoritmo di Landau-Vishkin si è per prima cosa implementato un algoritmo più semplice che risolve lo stesso problema con complessità peggiore (quello di Ukonnen), per poi modificare la parte fondamentale mantenendo la struttura del precedente, come fatto da Landau e da Vishkin con le loro idee. Un vantaggio dell'implementazione proposta è che i risultati vengono inseriti in un file e quindi possono essere riutilizzabili successivamente da altre applicazioni. Il difetto principale dell'implementazione invece è la sua complessità quadratica. Oltre al necessario miglioramento della complessità sarebbe utile sostituire l'utilizzo di file con l'utilizzo

di database sia per contenere le stringhe di input sia per contenere i risultati in output, in quanto la dimensione di questi, nell'utilizzo che ci si propone di effettuare, si suppone sia molto elevata e quindi si presenterebbero dei problemi di memoria fisica insufficiente a contenere i dati.

2. Allineamento di sequenze con errori

In accordo da quanto scritto da D. Gusfield [3] definiamo i concetti importanti utili a descrivere il problema che l'algoritmo studiato risolve.
L'obiettivo dell'allineamento di sequenze con errori è quello di confrontare tra loro due stringhe di caratteri e di ricercare le occorrenze di una all'interno dell'altra, accettando però anche le occorrenze con alcuni errori di confronto.

Distinguiamo 3 tipi di errori che possiamo accettare durante il confronto:

- *Mismatch* (*R*), un carattere nel testo non coincide con il rispettivo carattere nel pattern;
- *Inserimenti* (*I*), un carattere presente nel pattern non è presente al rispettivo posto nel testo;
- *Cancellazioni* (*D*), un carattere presente nel testo non è presente al rispettivo posto nel pattern;

Esempio:

```
T: a b c d e f g h  l
     R   D     I
P: a a c d  f g h i l
```

Possiamo ora definire la *distanza di Levensthein* come il numero di *R I D* che è necessario eseguire per ottenere una stringa *A* da una stringa *B*.
Si può inoltre definire il problema dell'**allineamento globale** come la determinazione della *distanza di Levensthein* date due stringhe.
Se poi interessa la posizione dei *R I D* allora sarà necessario costruire l'*edit transcript* delle due stringhe che sarà una sequenza di R I D, ad esempio *RRRID*…

Per **allineamento locale** invece si intende la ricerca degli allineamenti globali di una stringa con tutte le sottostringhe dell'altra. L'allineamento locale a distanza minore o uguale a *k* è quindi la ricerca di tutte le occorrenze di una stringa all'interno dell'altra con meno di *k* errori.

3. Contesto storico

Nel 1970 Needelman-Wrush costruiscono un algoritmo cubico per l'allineamento globale di stringhe con errori. Nel 1981 Smith e Waterman costruiscono un algoritmo quadratico (basato sulla tecnica della programmazione dinamica) per l'allineamento locale. Nel 1983 E. Ukonnen crea un algoritmo che si pone l'obbiettivo di migliorare quello basato sulla programmazione dinamica cercando di computare meno valori, ma il risultato è che la complessità nel caso peggiore risulta asintoticamente equivalente. Mentre nel 1986 Landau-Vishkin producono un algoritmo lineare per l'allineamento locale a distanza minore o uguale a *k* sfruttando le idee presenti nell'algoritmo di Knuth Morris-Pratt, cioè di sfruttare il lavoro (i match tra sottostringhe) che è stato eseguito in fasi precedenti e utilizzare i suffix tree per estrarre più velocemente informazioni riguardanti stringhe. Il problema dell'allineamento locale a distanza minore o uguale a *k* è stato successivamente risolto in tempo mediamente sublineare nel caso in cui le stringhe abbiano certe caratteristiche particolari (algoritmo di Wu-Mauber, algoritmo di Chang-Lowler, algoritmo di Meyers).

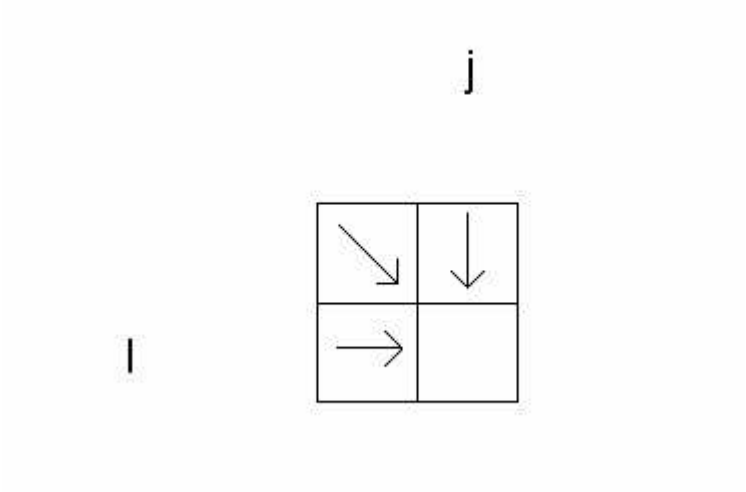
4. Un algoritmo inefficiente per l'allineamento di stringhe con errori

Partiamo con la descrizione di un algoritmo precedente e meno efficiente di quello di Landau-Vishkin. Un metodo per l'allineamento globale tra due stringhe è quello che si basa sulla *programmazione dinamica*. Questo algoritmo, date due stringhe P e T di lunghezza *m* ed *n* rispettivamente, consiste nel costruire una matrice D di dimensione |P|x|T| così definita

$D_{l,j}$ = minimo numero di differenze che posso ottenere confrontando le stringhe P[1..l] e T[1..j].

La cosa più interessante è che la prima riga e la prima colonna sono facilmente inizializzabili e il valore di ogni elemento della matrice è facilmente assegnato a partire dal valore di tre elementi adiacenti:

$$D_{l,j} = \begin{cases} \min(D_{l-1,j}+1,\ D_{l,j-1}+1,\ D_{l-1,j-1}) \text{ se } P[l]=T[j] \\ D_{l-1,j-1}+1 \text{ altrimenti} \end{cases}$$



Questo algoritmo è facilmente adattabile per l'allineamento locale semplicemente costruendo la matrice $D_{l,j}$ per $n-m+k$ volte vedendo così il problema dell'allineamento locale come una sequenza di più allineamenti globali, quindi avremo per $i=0\ldots n-m+k$ delle matrici $D^{(i)}_{l,j}$ che ci daranno le seguenti informazioni

$D^{(i)}_{l,j}$ =minimo numero di differenze ottenibile allineando le stringhe P[1..l] e T[i+1..i+j].

Ora, per scoprire quali sono le occorrenze di P in T con al più *k* differenze, sarà sufficiente controllare le ultime righe delle matrici e se

$$D^{(i)}_{m,j} \leq k$$

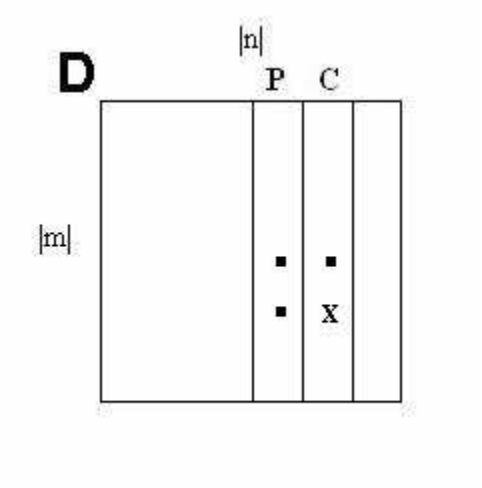
allora P[1..m] e T[i+1..i+j] sono allineabili con al più *k* errori e quindi in posizione *i+1* del testo inizia un'occorrenza valida del pattern.

Lo pseudo-codice di un algoritmo per l'allineamento locale basato sulla programmazione dinamica è il seguente:

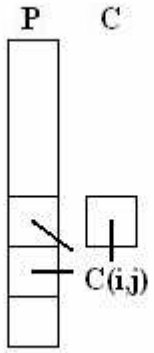
```
for i:=0 to n-m+k do
        D[0][0]:=0
        for j:=1 to m+k do D[0][j]:=j od
```

```
        for l:=1 to m do D[l][0]:=l od
        for l:=1 to m do
                for j:=1 to m+k do
                        if (P[l]=T[i+j]) then
                        D[l][j]:=min( D[l-1][j]+1, D[l][j-1]+1, D[l-1][j-1] )
                        else
                        D[l][j]:=D[l-1][j-1]+1
                od
        od
        for j:=m-k to m+k do
                if (D[m][j]=k) then print("Trovata un'occorrenza che inizia in T[i+1]")
        od
od
```

Visto questo interessante modo di costruire la matrice *D*, è possibile produrre un metodo che consenta di non memorizzare interamente la matrice. Lo spazio richiesto da *D* è O(mn), che è possibile ridurre a *2m*. Quando si assegnano i valori agli elementi della riga *i* di *D* sono necessari soltanto i valori degli elementi della riga *i-1*, quindi è possibile implementare l'algoritmo basato su programmazione dinamica utilizzando solamente due colonne, *C* (corrente) e *P* (precedente), cioè O(m) spazio. La colonna *C* è calcolata usando *P*, le due stringhe e i valori già calcolati di *C*.



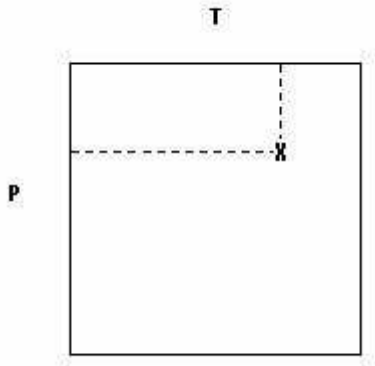
Un ulteriore miglioramento dello spazio necessario è possibile utilizzando una sola colonna e una cella invece delle due colonne, in quanto per calcolare un elemento di *C* è necessario, oltre *P* e le due stringhe, solamente il precedente valore calcolato in *C* (memorizzabile nella cella). Ovviamente se $n<m$ lo spazio può essere ridotto a O(n) utilizzando righe al posto di colonne.



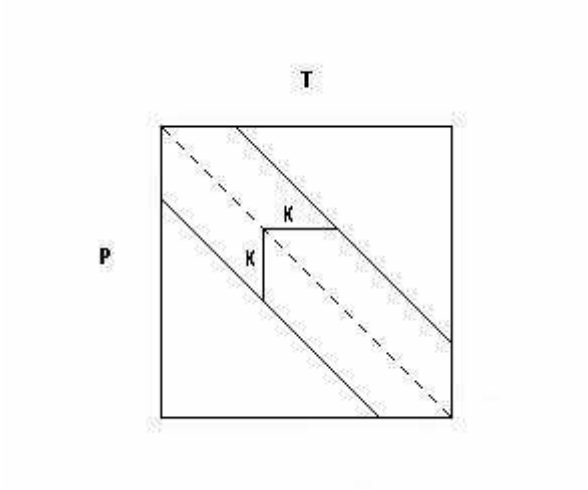
L' algoritmo di programmazione dinamica però ha il difetto di costruire l'intera matrice *D* il cui costo sarà O(mn) anche senza la necessità di memorizzarla interamente (in quanto i valori sono assegnati in base a quelli degli elementi contigui). Quindi il costo dell'allineamento globale sarà quadratico e quello per l'allineamento locale sarà cubico.
La prima cosa a cui si può pensare per migliorare la complessità è quella di non costruire l'intera matrice *D* e quindi di non calcolare tutti i valori dei suoi elementi. Vedremo che questa idea è effettivamente stata proposta, ma vedremo anche che non sarà sufficiente a raggiungere la complessità lineare in quanto sarà necessario, inoltre, sfruttare informazioni riguardanti le computazioni già svolte al fine di evitare di ripeterle nuovamente.

5. Algoritmo di Landau-Vishkin

Come detto, l'algoritmo di Landau-Vishkin risolve il problema dell'allineamento locale con errori di sequenze, trova cioè tutte le occorrenze di un pattern *P* in un testo *T* con al più *k* errori del tipo inserimento, cancellazione o mismatch.
Nell'algoritmo di programmazione dinamica viene costruita una matrice |P|x|T|,



il che comporta il fatto di computare valori che non interessano (allineamenti con più di *k* errori). Il difetto di calcolare valori non interessanti è stato risolto successivamente con un algoritmo che evita le computazioni di valori inutili che si trovano al di sopra e al di sotto di certe diagonali della matrice (in particolare interessano *2k+1* diagonali della matrice perché altrimenti avrei più di *k* errori) descritto in "On approximate string matching" pubblicato nel 1983 da E. Ukonnen su Lecture Notes in Computer Science 158.



Diciamo che la coppia di indici *(i,j)* di una matrice si trova sulla diagonale *d* se e soltanto se *j-i=d*. Per evitare di calcolare valori non utili della matrice D, Ukkonen considera, e costruisce, una matrice L che conterrà le informazioni utili presenti in D

$$L^{(i)}_{d,e} = l \leftrightarrow D^{(i)}_{l,j} = e \text{ dove } j=l+d$$

L avrà dimensioni *2(k+1)* x *2(k+1)* e in posizione *(d,e)* ci indicherà il più lungo prefisso di P che sappiamo allineare con T a costo *e*. Quindi

$$L^{(i)}_{d,e} = l \leftrightarrow P[1..l] \approx T[i+1..i+l+d] \text{ con e errori}$$

Analogamente a prima, se ci interessa l'allineamento locale dovremmo costruire *n-m+k* matrici L e, se ad un certo punto scriviamo in un elemento della matrice il valore della lunghezza del pattern, vorrà dire che riusciamo ad allineare l'intero pattern con una sottostringa del testo commettendo *e* errori, e, dato che calcoliamo solo i valori di $e \leq k$, cioè quelli utili, che abbiamo trovato un'occorrenza valida.

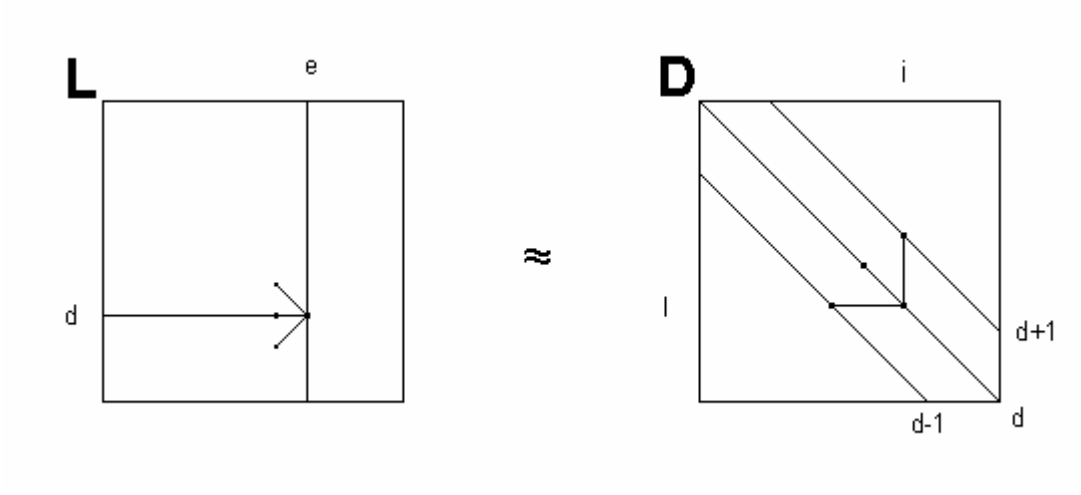
Inizializziamo $L_{d,e}$ con $-\infty$ per $d = -(k+1)..(k+1)$ ed $e = |d|-2$ il che indica l'impossibilità di allineare il pattern con il testo trovandosi sulla diagonale *d* e commettendo $|d|-2$ errori in quanto

trovandosi sulla diagonale *d* sarà già necessario compiere almeno |*d*| errori per riallineare le stringhe. Il secondo passo dell'inizializzazione prevede di assegnare il valore |*d*|*-1* a $L_{d,|d|-1}$ se *d<0* perché ciò significa che il prefisso di pattern è più lungo della sottostringa del testo considerata, in particolare è lungo |*d*|*-1*, quindi con al massimo |*d*|*-1* errori esso corrisponderà alla stringa del testo. Se invece *d≥0*, $L_{d,|d|-1}$ prenderà valore -1 perché in questo caso il prefisso di pattern e più corto della sottostringa del testo e quindi il valore -1 indica che non è possibile allineare con |*d*|*-1* errori due stringhe di lunghezza che differisce di *d*, ma, considerando una porzione più piccola di testo, potrei riuscire ad allinearle.

Dopo l'inizializzazione, attraverso due cicli annidati con $e \leq k$ e $|d| \leq e$ per non uscire fuori delle diagonali interessanti, assegniamo i valori ai restanti elementi di L. Si inizializza una variabile *row* come segue

$$row := \max(L_{d,e-1}+1,\ L_{d-1,e-1},\ L_{d+1,e-1}+1)$$

row, debitamente incrementata, sarà il valore che assegneremo a $L_{d,e}$.



Da *d-1* in d non scendo di riga, *l* resta uguale, mentre da *d* e da *d+1*, *l* viene incrementato di 1.

Il valore iniziale di *row* è il massimo di quei valori perché si sceglie il più grande prefisso di P che so allineare con T commettendo *e-1* errori e se il valore proviene da *d* o da *d+1* allora il valore verrà incrementato di uno in quanto aumenta la dimensione del prefisso del pattern, altrimenti resta uguale.

A questo punto si esegue un ciclo while che confronta i successivi caratteri del pattern con i successivi caratteri del testo e, se il confronto va a buon fine, si incrementa il valore di *row* fino a che il pattern ed il testo coincidono oppure finché non si raggiunge la fine di P o di T. Ora il valore di *row* viene assegnato a $L_{d,e}$, e se questo coincide con *m* di sarà trovata un'occorrenza valida.

Con questi accorgimenti però la complessità resta ancora O(m*n) (dove *m* è la lunghezza del pattern ed *n* è la lunghezza del testo) in quanto il ciclo while viene eseguito al più *m* volte e si trova all'interno di una funzione che viene chiamata O(n) volte.
L'ingrediente decisivo per ottenere una performance lineare O(m), introdotto nell'algoritmo di Landau-Vishkin, è quello di sfruttare i match che sono già stati eseguiti in fasi precedenti per evitare di calcolare i valori di tutte le diagonali.
La soluzione proposta è quella di memorizzare i match (parziali) già effettuati in una tabella *m*n* chiamata *MaxLength* utilizzata per saltare match già eseguiti. L'informazione contenuta in *MaxLength* può venire memorizzata o mediante una matrice *m*n* (costruibile in tempo quadratico) oppure mediante un suffix tree (costruibile in tempo lineare mediante l'algoritmo di Ukonnen) (vedi 5.2) utilizzando poi l'algoritmo di Harel-Tarjan per trovare il Lowest-Common-Ancestor in tempo costante.
Per introdurre il problema di trovare il Lowest-Common-Ancestor è necessario dire che in un albero *T*, un nodo *u* è **predecessore** di un nodo *v* se e solo se *u* si trova sull'unico cammino tra la radice dell'albero *T* e il nodo *v*. In un albero *T* il **Lowest-Common-Ancestor** di due nodi *x* e *y* è il più profondo nodo in *T* che è predecessore sia di *x* che di *y*.

5.1 Struttura dell'algoritmo di Landau-Vishkin

L'algoritmo si compone di una parte di analisi del pattern, che consiste nella costruzione della matrice *MaxLength* o del suffix tree, e di una parte di analisi del testo che consiste nel ricercare effettivamente le occorrenze.
L'implementazione dell'analisi del pattern consiste nel costruire la matrice *MaxLength* che ci dirà, per ogni posizione *(i,j)*, la lunghezza del più lungo allineamento esatto di prefissi dei suffissi del pattern inizianti all'indice *i* e *j*, cioè se MaxLength[i][j]=f vorrà dire che P[i+1…i+f]=P[j+1…j+f] e che P[i+f+1]≠P[j+f+1]. Ad esempio, se il pattern sarà dato dalla stringa *acaagttaccag* allora nella matrice in posizione (0,7) sarà presente il valore 2 in quanto P[1,2]=*ac*=P[8,9] e P[3] ≠P[10].

L'analisi del testo consiste in *n-m+k* chiamate alla funzione Check(i) che controlla se alla posizione *i+1* del testo inizia un'occorrenza, con al più *k* errori, del pattern.
Per risolvere il problema di complessità dell'algoritmo di Ukonnen viene inserito un ciclo che controllerà qual è il più lungo match esatto a partire da P[row+1] e da T[i+row+d+1] (come il ciclo while di Ukonnen), senza però eseguire i confronti, sfruttando cioè soltanto le informazioni in possesso: l'output dell'analisi del pattern ed i match esatti eseguiti nelle iterazioni precedenti di Check. Questa prima parte di Check controlla il numero di differenze tra T[i+1…j] e un prefisso del pattern. Se già in questa fase si scoprono più di *k* differenze si salta all'iterazione successiva risparmiando in complessità in quanto non eseguiremo il ciclo che effettua realmente il confronto carattere per carattere.

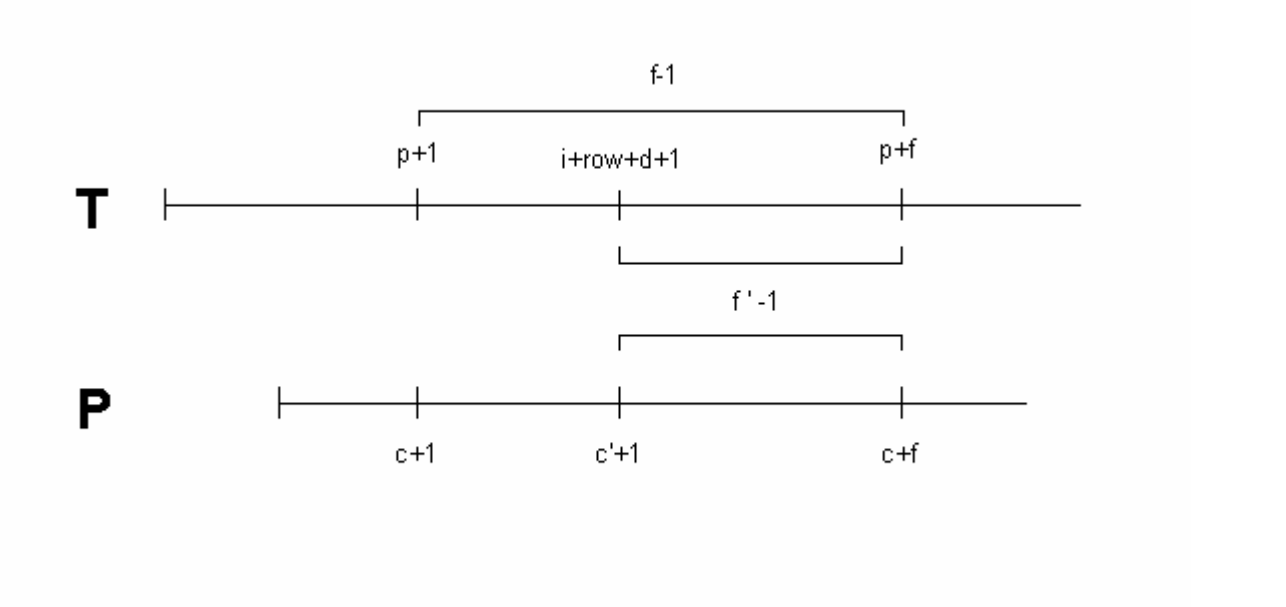
Supponiamo che nell'iterazione *i-1* di Check sia stato calcolato, per ogni posizione nel testo *X=1…i* il più lungo prefisso di T[x…i] che si allinea con un prefisso del pattern con al più *k* errori. Sia T[x…j(x)] questo prefisso di T[x…i], chiamo *j* il massimo dei j(1)…j(i) che sarà la posizione più a destra raggiunta nel testo nella precedente iterazione.

Spieghiamo ora come vengono mantenute le informazioni dall'iterazione precedente. Se una sottostringa del testo T[p+1…p+f] concorda con una sottostringa del pattern P[c+1…c+f] e T[p+f+1]≠P[c+f+1] indichiamo questa corrispondenza con la tripla *(p,c,f)*. Indichiamo inoltre i simboli nel testo che non sono presenti nella sottostringa del pattern con la tripla *(h,0,0)* se questo simbolo è T[h+1]. La corrispondenza tra T[i+1…j] e un suffisso di un prefisso del pattern può essere descritto da una sequenza di triple chiamata S_{ij} lunga al più *2k+2*.

E' necessario definire inoltre la *cover* di un indice del testo. Questa sarà una tripla del tipo *(p,c,f)* tale che, se sarà la *cover* di T[h+1], $p \leq h \leq p+f$. Inoltre se $i \leq h \leq j$ allora esiste in S_{ij} una tripla che copre T[h+1].

A questo punto di Check(i) sappiamo allineare P[1..row+1] con T[i+1..i+row+d+1] con *e* errori. Ora si cerca il più lungo match esatto a partire da P[row+1] e da T[i+row+d+1] finché non si troveranno e+1 errori oppure si raggiungerà j dopo il quale non potremmo sfruttare informazioni di match precedenti ma dovremmo procedere al confronto carattere per carattere.
Si estrae da S_{ij} (memoria dei match precedenti) la cover di T[i+row+d+1] e si ricavano da questa c' ed f ' tali che T[i+row+d+1..i+row+d+f ']=P[c'+1..c'+f ']. Calcoliamo i valori di c' ed f ':

La cover (p,c,f) di T[i+row+d+1] ci dirà che T[p+1..p+f]=P[c+1..c+f] e sappiamo che p ≤ i+row+d ≤ p+f



Quindi f ’-1=p+f-(i+row+d+1) → f ’=p+f-(i+row+d) e voglio
c’+1=c+f-(f’-1) → c’=c+f-f ’.
Ora se f ’=0 e T[i+row+d+1] ≠ P[row+1] abbiamo scoperto e+1 errori e quindi possiamo andare alla fine del calcolo dell’elemento di $L_{d,e}$, se invece il confronto è andato a buon fine possiamo incrementare la lunghezza di P e confrontare il prossimo carattere del testo.
Se invece f ’≠ 0 possiamo utilizzare le informazioni che abbiamo sul pattern controllando il valore di *MaxLength*[c][row]. A questo punto se f ’ e *MaxLength*[c][row] sono diversi si incrementa il valore di row con il minimo tra i due e si salta alla fine del calcolo dell’elemento di $L_{d,e}$ perché sono stati trovati e+1 errori in quanto il carattere successivo da confrontare tra P e T darà un errore; se invece f ‘ e *MaxLength*[c][row] sono uguali, si incrementa row con il valore di f ‘ e si procede con il confronto tra P e T in quanto non è detto che il successivo carattere di P e T non garantisca un confronto che vada a buon fine.

S_{ij} viene costruita al termine di Check(i) nel momento in cui viene controllato se va modificato il valore di *j*. Se alla fine dell’i-esima iterazione di Check è stato raggiunto un nuovo carattere del testo allora costruiamo una nuova sequenza al posto di S_{ij}. Per fare ciò, nel corso dell’iterazione si tiene in memoria per ogni $L_{d,e}$ una sequenza di triple $T_{d,e}$ tale che se $L_{d,e} = l$ la sequenza descrive l’allineamento con *e* errori tra $t_{i+1} \ldots t_{i+l+d}$ con $a_i \ldots a_l$. Al termine dell’iterazione *i* si controlla quale delle sequenze ha raggiunto il simbolo più a destra nel testo e, se questo è maggiore di *j*, si prende questo come nuovo valore di *j* e la sua sequenza di triple come nuovo S_{ij}.

5.2 Suffix Tree

Un'alternativa alla costruzione della matrice *MaxLength* è, come detto, la costruzione di un suffix tree. La complessità della fase di analisi del pattern costruendo la matrice *MaxLength* è pari a $O(m^2)$, dove *m* è la lunghezza del pattern, in quanto la matrice è composta da *m* righe e da *m* colonne. La complessità della analisi del pattern costruendo un suffix tree sarebbe minore in quanto esistono diversi algoritmi, ad esempio quello descritto da P. Weiner in "Linear pattern matching algorithm" nel 1973, che richiedono tempo O(m) per la costruzione di un suffix tree per una stringa di lunghezza *m*, in questo caso il nostro pattern.

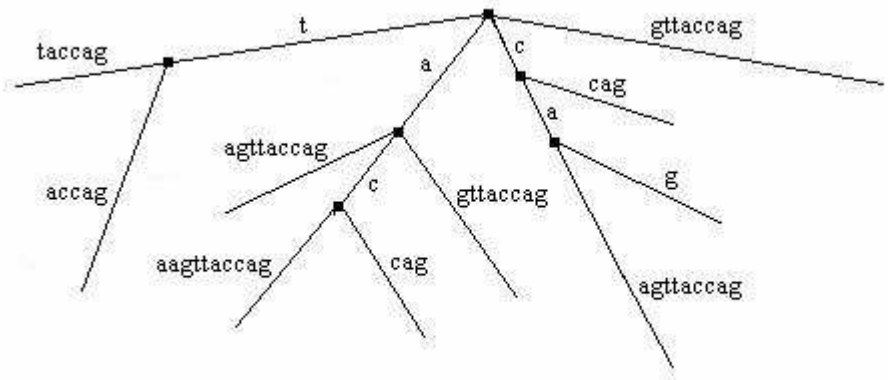
Quindi utilizzando un suffix tree la complessità dell'intero algoritmo resterà lineare, mentre la costruzione della matrice rende l'intero algoritmo quadratico nella dimensione del pattern.

Un *suffix tree* è una struttura dati che espone la struttura interna di una stringa, e può essere usato per la soluzione lineare di molti problemi complessi su stringhe, come ad esempio il matching inesatto. Anche se già prima di lui sono state proposte delle soluzioni lineari per la costruzione di suffix tree, l'algoritmo più usato è quello proposto da Ukonnen data la sua maggior semplicità.

Definiamo un *suffix tree* per una stringa *S* di *m* caratteri, un albero con esattamente *m* foglie numerate da *1* ad *m*. Ogni nodo interno ha almeno due figli e ogni arco è etichettato con una sottostringa non vuota di *S*. Le etichette di due archi uscenti dallo stesso nodo non possono cominciare con la stessa lettera.

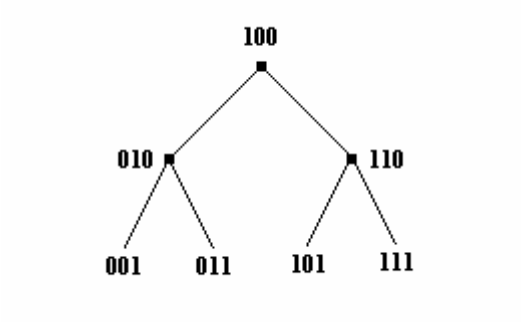
Per ogni foglia *i*, la concatenazione delle etichette degli archi del cammino radice-foglia, è esattamente il suffisso di *S* che parte dalla posizione *i*, cioè $S[i \ldots m]$.

Ad esempio se la stringa è *acaagttaccag* il suffix tree relativo sarà dato da:



5.2.1 Trovare il Lowest-Common-Ancestor in tempo costante

Una volta costruito il suffix tree in tempo lineare, ad esempio con l'algoritmo di Ukonnen, sarà necessario utilizzarlo durante l'analisi del testo al posto della matrice *MaxLength*. Affinché il suffix tree ci dia le stesse informazioni che ci può dare la matrice *MaxLength*, cioè la lunghezza del più lungo allineamento esatto di prefissi di suffissi del pattern, sarà necessario utilizzare un algoritmo per trovare il Lowest-Common-Ancestor di due nodi di un albero, ad esempio l'algoritmo di Harel-Tarjan descritto in "Fast algorithms for finding neares common ancestors" del 1984. Questo algoritmo trova il Lowest-Common-Ancestor in tempo costante O(1).
Un albero viene processato in tempo lineare O(m) effettuando una visita in profondità dell'albero allo scopo di mappare tutti i suoi nodi in quelli di un albero binario completo. A questo punto si ha a disposizione un albero con *d* archi in cui ogni nodo interno ha esattamente due figli e ogni nodo ha assegnato un numero binario di dimensione *d+1* bit che identifica l'unico cammino dalla radice a quel nodo in quanto l'i-esimo bit da sinistra del numero ci dirà se l'i-esimo arco sul cammino radice-nodo va al figlio sinistro (0) oppure al figlio destro (1).



Per trovare il Lowest-Common-Ancestor di due nodi *i* e *j* viene fatto un **XOR** dei numeri associati ai nodi *i* e *j* e si cerca, nel risultato, il bit più a sinistra che sia settato a 1. Ciò significa che i cammini per raggiungere *i* e *j* concordano per i primi *k-1* archi, se il bit più a sinistra che sia settato a 1 si trova in posizione *k*. Il che vuol dire che Lowest-Common-Ancestor(i,j) sarà il nodo il cui numero associato è formato dai *k-1* bit di *i*, seguiti da un 1 e da *d+1-k* zeri.

5.3 Pseudo-codice dell'algoritmo di Landau-Vishkin

Analisi del Pattern

```
begin
    Inizializzazione
        for d:=0 to m-1 do
            MaxLength[m-1-d][m-1]:=1 se P[m-d]=P[m]
            MaxLength[m-1-d][m-1]:=0 altrimenti
        od
    for d:=0 to m-1 do
        for i:=m-2-d to 0 do
            if P[i+1]=P[i+d+1]
                then MaxLength[i][i+d]:=1 + MaxLength[i+1][i+d+1]
                else MaxLength[i][i+d]:=0
        od
    od
end
```

Il passo di inizializzazione consiste nel dare i valori all'ultima colonna della matrice. Si definisce la coppia *(i,j)* sulla diagonale *d* se *j-i=d* dove $-(m+1) \leq d \leq m-1$.

Analisi del Testo

```
begin
    for i:=0 to n-m+k do
        Check(i)
    od
end
```

Funzione Check

begin

[1] Inizializzazione

for d:=-(k+1) **to** (k+1) **do**

L[d][|d|-2]:=-∞

if d<0

then L[d][|d|-1]:=|d|-1

else L[d][|d|-1]:=-1

od

[2] **for** e:=0 **to** k **do**

for d:=-e **to** e **do**

[3] row:= max(L[d][e-1]+1, L[d-1][e-1], L[d+1][e-1]+1)

[4.new]Estrarre da S_{ij} informazioni sufficienti a trovare e+1

differenze sulla diagonale d o raggiungere la posizione j.

while i+row+d+1 ≤ j **do**

[4.new.1] Estrarre da S_{ij} la tripla che copre T[i+row+d+1].

Ricavare c ed f tali che T[i+row+d+1…i+row+d+f]=P[c+1…c+f].

[4.new.2] **if** f=0

then *[4.new.3]* **if** T[i+row+d+1]≠P[row+1]

then go to [5]

else row:=row+1

else *[4.new.4]* **if** f≠MaxLength[c][row]

then row:=row+min(f,

MaxLength[c][row])

go to [5]

else row:=row+f

od

[4.old] Leggere del testo addizionale per raggiungere e+1 differenze sulla

diagonale d, la fine del testo o la fine del pattern.

while row<m and i+row+d<n and P[row+1]=T[i+row+d+1] **do**

Row:=row+1

od

[5] L[d][e]:=row

[6] **if** L[d][e]=m

then print(“Trovata un’occorrenza che inizia in T[i+1]”)

od

od

[7] Se sono stati raggiunti nuovi simboli nel testo costruisci una nuova S_{ij}.

end

Nella fase 4.new di Check si controlla se è possibile saltare la parte 4.old che porta complessità maggiore. Il ciclo while di 4.new serve per cercare il più lungo match esatto tra una sottostringa del testo iniziante in T[i+row+d+1] e una sottostringa del pattern iniziante in P[row+1].
All’inizio dell’iterazione *i* di Check le liste $T_{d,e}$ sono vuote. Per trovare la sequenza di triple $T_{d,e}[d][e]$aggiungiamo triple alla fine della sequenza del predecessore di *L[d][e]*, cioè quello che da il valore nell’inizializzazione di row (punto [3]). Chiamiamo r1 il valore iniziale di row. Se row prende il suo valore iniziale da L[d-1][e-1] oppure da L[d][e-1] aggiungiamo la tripla (i+r1+d-1, 0, 0) indicante che per T[i+row+d] non ci sono simboli corrispondenti nel prefisso del pattern. Successivamente all’istruzione [5], se L[d][e]>r1 allora aggiungiamo la tripla (i+r1+d, r1, L[d][e]-r1) che indicherà il pezzo di match esatto tra i caratteri del testo ed i caratteri del pattern. Al termine dell’iterazione *i* si controlla quale delle *2(k+1)* x *2(k+1)* sequenze ha raggiunto il simbolo più a destra nel testo. Se l’indice di questo simbolo è maggiore di *j* allora prendiamo questo indice come nuovo *j* e la sequenza come nuovo S_{ij}.

Se, invece di utilizzare la matrice *MaxLength*, costruiamo un suffix tree allora consideriamo il LCA delle foglie *i* e *j* che corrispondeono ai suffissi P[i+1…m] e P[j+1…m]. A questo punto MaxLength[i][j] sarà semplicemente Length(LCA(i,j)).

5.4 Complessità dell'algoritmo di Landau-Vishkin

La funzione Check richiede tempo $O(k^2)$, quindi l'analisi del testo richiede $O(n*k^2)$ tempo.
L'istruzione [4.old] viene chiamata ogni volta che è stato raggiunto un nuovo simbolo nel testo. Abbiamo O(k) diagonali e può essere necessario confrontare il nuovo simbolo per ognuna di esse, con complessità O(k) ogni volta per l'istruzione [4.old].
In ogni iterazione vengono computate O(k) diagonali, e S_{ij} ha al più $2k+2$ triple; possiamo assegnare ogni operazione effettuata su diagonali a una differenza scoperta (al più $k+1$) o a una tripla esaminata (al più $2k+2$), quindi ci sono O(k) operazioni per diagonale e $O(k^2)$ per iterazione in [4.new].
Anche l'istruzione [7] richiede tempo $O(k^2)$ perché le liste T_{de} sono create aggiungendo triple alle vecchie liste e ci sono $O(k^2)$ triple in tutto.

5.4.1 Complessità utilizzando un suffix tree

Ogni interrogazione alla matrice *MaxLength* richiede tempo costante e dato che l'analisi del testo richiede tempo $O(n*k^2)$, vengono fatte solo $O(n*k^2)$ interrogazioni alla matrice.
Il suffix tree restituito dall'analisi del pattern ha O(m) nodi. L'algoritmo di Harel-Tarjan effettua un pre-processing del suffix tree in tempo O(m) e quindi restituisce il Lowest-Common-Ancestor in tempo costante, mentre utilizzando la matrice avevamo tempo $O(m^2)$ per la sua costruzione.

6. Descrizione del codice sorgente

L'implementazione dell'algoritmo è composta da 5 files scritti in linguaggio C.

L'header-file di nome "Alg.H" contiene il caricamento delle librerie necessarie, l'elenco di tutte le funzioni, che verranno definite ed utilizzate nei file sorgente, e la definizione delle strutture dati e delle costanti (vedi 6.1).

Il file "Main.C" contiene la procedura principale main la quale effettua le chiamate alle procedure openpattern e textpattern, ed in seguito chiama le procedure che si occupano dell'analisi del pattern e dell'analisi del testo (vedi 6.2).

Il file "Triple.C" contiene le funzioni che operano sulla struttura dati "lista di triple" definita nel file "Alg.H"; queste serviranno ad aggiungere un elemento alla lista, e di cercare all'interno di una lista la tripla che "copre" un certo indice del testo (vedi 6.3).

In file "Buildmaxlen.C" contiene la procedura che si occupa dell'analisi del pattern. In questo caso la procedura si occupa di costruire la matrice che ci dirà, in posizione *(i,j)*, la lunghezza del più lungo allineamento esatto di prefissi dei suffissi del pattern inizianti all'indice *i* e *j* (vedi 6.4).

Il file "File.C" contiene le funzioni addette all'accesso ai file contenenti il testo ed il pattern e si preoccuperà di caricarli in dei vettori di caratteri. Inoltre, questo file contiene una procedura che si occupa, ogni qualvolta venga trovata un'occorrenza con al più *k* errori, di scrivere in un file "results.txt" l'indice nel testo in cui inizia quella occorrenza, ed una funzione che restituisce la dimensione del file il cui nome viene passato per parametro (vedi 6.5).

Infine c'è il file "Check.C" che contiene la parte essenziale dell'algoritmo, cioè la funzione Check(i) che si occuperà di verificare se alla posizione *i* del testo può iniziare un'occorrenza del pattern con al più *k* errori (vedi 6.6).

Descriviamo ora in dettaglio il contenuto di ogni file e le proprie funzioni.

6.1 File Alg.H

Il file contiene l'inclusione di determinate librerie che serviranno all'accesso ai file contenenti testo, pattern e risultati, ad allocare zone di memoria per le liste di triple e a misurare la dimensione dei file.
Contiene poi la definizione delle costanti *pathname* dei file contenenti testo e pattern, la variabile numerica *k* indicante il numero di errori massimo accettabile durante il confronto tra testo e pattern, e due variabili ausiliarie che serviranno ad eseguire cicli.
In seguito è presente la dichiarazione della struttura dati "lista di triple" che sarà una struttura contenente tre variabili di tipo intero, di nome "p", "c" ed "f", e una variabile puntatore dello stesso tipo che stiamo dichiarando, che svolgerà il compito di puntatore al prossimo elemento della lista; per convenzione abbiamo stabilito che se il valore del puntatore è NULL, vuol dire che siamo al termine della lista.

```
struct Triple{
    int p;
    int c;
    int f;
    struct Triple *pointer;
};
```

E' presente quindi la definizione della matrice *L* utilizzata nella procedura Check(i) la cui dimensione varia in funzione del parametro *k*, il che obbliga il valore di *k* ad essere passato come parametro di input al programma. Segue la definizione di una matrice di liste "Triple" e di una ulteriore lista "Triple" che serviranno a memorizzare informazioni riguardo i confronti svolti durante le iterazioni del programma.
C'è poi la definizione di una matrice di puntatori e di un singolo puntatore di tipo "Triple" che serviranno a tenere in memoria qual è l'inizio delle liste "Triple" nel momento in cui sarà necessario scorrerle perdendo così la testa della lista.
In fine sono presenti le definizioni dei vettori di caratteri che conterranno il testo ed il pattern, della variabile intera "j" che indicherà l'indice del testo più a destra che è stato raggiunto nel corso delle chiamate alla procedura Check(i), delle variabili intere "m" ed "n" che indicheranno rispettivamente la lunghezza del pattern e del testo rispettivamente, e di tutte le funzioni utilizzate nel programma.

6.2 File Main.C

Il file contiene l'inclusione del file "Alg.H", la definizione delle funzioni max e min, che dati due interi restituiscono l'intero maggiore o minore dei due rispettivamente, e la funzione principale main.

6.2.1 La funzione main

Nella funzione main per prima cosa viene acquisito il parametro iniziale "k" e associato ad una variabile. Quindi viene allocato lo spazio per la matrice "L" in funzione del valore di "k" e viene allocato lo spazio in memoria per la matrice contenente ad ogni elemento una lista di triple *(p,c,f)* ognuna delle quali viene inizializzata a NULL indicando così che la lista è vuota. Anche per la lista *Sij* viene eseguito lo stesso lavoro, viene quindi allocato lo spazio per la lista ed essa viene inizializzata a NULL. Analogamente si inizializzano i puntatori all'inizio delle liste "Triple".

A questo punto viene definita la variabile intera "i", che servirà a scandire il ciclo delle chiamate della funzione Check, la dimensione del testo e del pattern, grazie alla funzione "fsize", e allocato lo spazio per i vettori che dovranno contenerli.

Vengono chiamate le procedure openpattern e opentext (vedi 6.5.1 e 6.5.2).

Viene ora fatta l'analisi del pattern attraverso la chiamata alla funzione build_maxlen che assegnerà i corretti valori numerici agli elementi della matrice di dimensione *m*m* precedentemente definita nella funzione main (vedi 6.4.1).

Ora, dopo aver inizializzato la variabile j a *0*, viene fatta l'analisi del testo, viene quindi eseguito un ciclo da *0* a *n-m+k* che effettua le chiamate alla funzione Check.

Al termine di questo ciclo il programma termina.

6.3 File Triple.C

Il file contiene l'inclusione del file "Alg.H", e la definizione delle funzioni addTriple e findCover utilizzate nella funzione Check.

6.3.1 La funzione addTriple

La funzione addTriple aggiunge la tripla di interi, passati come parametri, alla lista, anch'essa passata come parametro, e restituisce un puntatore alla lista con la tripla in più.
Per prima cosa la funzione controlla se la tripla puntata in questo momento è NULL, il che indicherebbe che ci troviamo alla fine della lista. In questo caso viene allocato lo spazio per una nuova tripla nella lista che viene inzializzata con i valori passati in ingresso alla funzione, viene inizializzato a NULL il valore del puntatore al prossimo elemento e viene restituito il puntatore alla lista modificata. Altrimenti, se la tripla puntata in questo momento non è NULL, allora vuol dire che non ci troviamo alla fine della lista e quindi viene chiamata ricorsivamente la funzione addTriple su il prossimo elemento della lista; questo però avviene solo dopo aver controllato che la tripla da inserire non sia già presente nella lista, nel qual caso la funzione terminerebbe.

6.3.2 La funzione findCover

La funzione findCover, dato un indice in ingresso, cerca tra le triple della lista "Sij" quella che "copre" l'indice passato come parametro.
La funzione controlla se la tripla attualmente puntata da "Sij" è la cover tramite la condizione

```
p <= index-1 && index-1 <= (p+f)
```

Se la condizione è verificata allora viene istanziata una tripla con i valori *p*, *c* ed *f* corretti e restituita come output della funzione, altrimenti si passa al prossimo elemento della lista "Sij" e si chiama ricorsivamente la funzione findCover che verificherà se la tripla successiva è la cover dell'indice passato in input.

6.4 File BuildMaxLen.C

Il file contiene l'inclusione del file "Alg.H", e la definizione della funzione build_maxlen utilizzata da main.

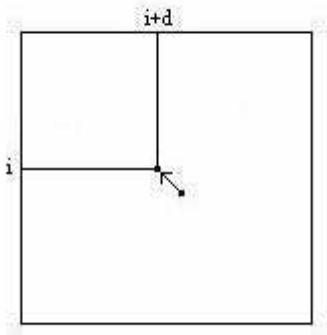
6.4.1 La funzione build_maxlen

La funzione build_maxlen, data una matrice in ingresso, inserisce i valori corretti in base ai valori del vettore P che contiene il pattern.

Per prima cosa c'è un ciclo da *0* a *m* (lunghezza del pattern) che inizializza l'ultima colonna della matrice con *1* o con *0* a seconda se l'ultimo carattere del pattern è uguale al carattere indicato dall'indice della riga della matrice.

Quindi vengono due cicli innestati, per '*d*' da *0* ad *m* e da *m-2-d* a *0* rispettivamente, che a seconda del fatto che due caratteri nel pattern si uguagliano o meno, aumenta di *1* il valore della cella della matrice diagonalmente più in basso oppure inserisce uno *0* se i due caratteri sono diversi.

```
for (d=0;d<m;d++){
    for(i=m-2-d; i>=0 ; i--){
        if (P[i+1] == P[i+d+1]){
                maxlen[i][i+d]= (1 + (int) maxlen[i+1][i+d+1]);
                maxlen[i+d][i]= (1 + (int) maxlen[i+1][i+d+1]);
        }else{
                maxlen[i][i+d]=0;maxlen[i+d][i]=0;
        }
    }
}
```



Inoltre, ogni volta che viene inserito un numero in un elemento della matrice, viene inserito lo stesso valore nell'elemento con indici scambiati. Questo perché la matrice è simmetrica rispetto la diagonale principale.

6.5 File File.C

Il file contiene l'inclusione del file "Alg.H", la definizione delle funzioni openpattern, opentext che riempiono i vettore P e T con il contenuto dei file che contengono il pattern ed il testo rispettivamente, la definizione della funzione writeresults che scrive in un file "results.txt" l'indice del testo al quale inizia un'occorrenza valida del pattern, e la definizione della funzione fsize che restituisce la dimensione del file il cui nome viene passato per parametro.

6.5.1 La funzione openpattern

La procedura openpattern, che non ha parametri in ingresso, definisce per prima cosa un puntatore ad un file che viene inizializzato attraverso la funzione *fopen* con parametro il *pathname* del file, contenente il pattern, e la modalità *read*.
Il primo elemento del vettore P viene inizializzato con il carattere *blank* in modo tale da utilizzare la stringa con indici da *1* a *m*. Quindi è presente un ciclo *while* che ha come guardia la condizione

```
P[i-1] != EOF
```

che verificherà se l'ultimo carattere inserito non è quello di fine del file. All'interno del ciclo, grazie alla funzione *fgetc*, viene inserito un carattere in posizione *i* nel vettore P e viene incrementata la variabile "i".
Al termine del ciclo viene assegnato il valore alla variabile "m" che conterrà la lunghezza della stringa del pattern.

6.5.2 La funzione opentext

La procedura opentext, che non ha parametri in ingresso, definisce per prima cosa un puntatore ad un file che viene inizializzato attraverso la funzione *fopen* con parametro il *pathname* del file, contenente il testo, e la modalità *read*.
Il primo elemento del vettore T viene inizializzato con il carattere *blank* in modo tale da utilizzare la stringa con indici da *1* a *n*. Quindi è presente un ciclo *while* che ha come guardia la condizione

```
T[i-1] != EOF
```

che verificherà se l'ultimo carattere inserito non è quello di fine del file. All'interno del ciclo, grazie alla funzione *fgetc*, viene inserito un carattere in posizione *i* nel vettore T e viene incrementata la variabile "i".
Al termine del ciclo viene assegnato il valore alla variabile "n" che conterrà la lunghezza della stringa del testo.

6.5.3 La funzione writeresults

La funzione writeresults, dato un valore intero in ingresso lo scrive in un file.
Viene definito un puntatore ad un file che viene inizializzato attraverso la funzione *fopen* con parametro il nome del file "results.txt" e la modalità *write*. Quindi attraverso la funzione *fprintf* viene scritto l'indice, ricevuto come parametro, nel file aperto ed infine viene chiuso il file.

6.5.4 La funzione fsize

La funzione fsize, grazie a delle funzioni di basso livello utilizzabili dopo aver importato certe librerie predefinite, misura la dimensione di un file il cui *pathname* viene passato in input e restituisce in output un valore intero che indica il numero di caratteri che contiene il file.

6.6 File Check.C

Il file contiene l'inclusione del file "Alg.H" e la definizione della procedura Check(i) che verificherà se alla posizione *i* del testo parte un'occorrenza valida del pattern.

6.6.1 La funzione check

La funzione Check, dopo una serie di definizioni di variabili intere, procede, grazie a due cicli innestati, a scandire la matrice di liste Tde e a impostare a NULL il valore di ciascun elemento, il che implica lo svuotamento di tutte le liste nella matrice.
Dopo ciò viene inizializzata la matrice L con un ciclo da *–k-1 a k+1*; il valore -∞ presente nell'algoritmo è sostituito nel programma dal valore *–d-5* in quanto sarà più piccolo di ogni altro valore inserito.
A questo punto ci sono due cicli innestati,

```
for(e=0;e<=k;e++){
      for(d=-e;d<=e;d++){
```

nei quali viene per prima cosa inizializzata la variabile "row" con il massimo tra 3 elementi della matrice L, ed il suo valore viene memorizzato nella variabile "r1".
A questo punto viene copiata nella posizione *(d,e)* della matrice Tde la lista relativa alle coordinate dell'elemento di L da cui ha preso il valore "row", scandendo la lista di origine e chiamando la

funzione addTriple per aggiungere l'elemento alla lista (d,e). Al termine della scansione viene ripristinato il puntatore all'inizio della lista.

Dopo aver fatto ciò, se "row" ha preso il suo valore da certi elementi della matrice L, viene aggiunta una tripla alla lista *(d,e)* indicante il che un certo carattere nel testo non è presente nel pattern; quindi con la funzione addTriple viene aggiunta la tripla corretta.

Ora, se è verificata la guardia

```
i+row+d+1<=j
```

viene cercata la cover dell'indice i+row+d+1 tramite la funzione findCover. Al termine di findCover viene ripristinato il puntatore all'inizio della lista "Sij" in quanto è stata scandita per la ricerca della cover. Vengono quindi settate le variabili "c" ed "f" con i valori calcolati in base alla cover trovata. In base ai valori di "c" ed "f" viene modificato il valore della variabile "row", si accede alla matrice MaxLength, e si salta ad un punto del codice evitando di eseguire un ciclo che agisce sulla variabile "row". Se questo salto non avviene, viene eseguito il ciclo che confronterà caratteri del pattern con caratteri del testo incrementando la variabile "row" ogni qualvolta il confronto andrà a buon fine.

A questo punto il valore di "row" viene assegnato all'elemento in posizione *(d,e)* della matrice L e, se il valore attuale di "row" è diverso dal suo valore iniziale (memorizzato nella variabile "r1"), cioè se "row" è stato incrementato, viene aggiunta una tripla alla lista in posizione *(d,e)* indicante il match esatto tra una parte del testo ed una parte del pattern.

Se il valore inserito nella matrice L è uguale ad *m*, allora è stata trovata un'occorrenza del pattern nel testo con al più *k* errori; quindi viene chiamata la procedura writeresults(i) che scriverà il valore di "i" nel file "results.txt".

A questo punto i due cicli innestati terminano e si procederà a verificare se sarà necessario costruire una nuova lista Sij e aggiornare il valore della variabile "j".

Si cerca, scandendo con due cicli innestati la matrice L, il valore massimo di $L[d][e]+d+i$ che, se maggiore di *j*, sarà il nuovo indice del carattere più a destra raggiunto nel testo, e le coordinate *(d,e)* indicheranno che la nuova lista Sij sarà quella in posizione *(d,e)* nella matrice Tde.

Per copiare la lista da Tde a Sij scandisco con un ciclo *while* gli elementi della lista in Tde e, per ogni elemento, chiamo la funzione addTriple per inserire l'elemento nella lista Sij (precedentemente cancellata assegnandole il valore NULL).

Dopo aver scandito la lista in Tde ripristino il puntatore all'inizio della lista.

A questo punto si è pronti per un'altra iterazione della funzione Check.

7. Testing dell'implementazione

A questo punto testiamo l'algoritmo implementato per verificarne l'effettivo funzionamento. Tutti i test sono stati svolti con una macchina Toshiba Satellite 1410-604 dotato di processore Mobile Intel Celeron 1.8 GHz, 256MB SDRAM, sistema operativo Microsoft Windows XP Home Edition. Procediamo con una batteria di test che verifichi i casi limite per poter giudicare la correttezza dell'implementazione su dei valori di input significativi.

Proviamo a fornire in input al programma i seguenti dati:

Testo *abcdefghi*

Pattern *bxdyegh*

k *3*

L'output del programma ci dice che, con questi parametri, l'unica occorrenza del pattern inizia in corrispondenza del secondo carattere del testo:

```
k vale 3

Il pattern e':
bxdyegh

Il testo e':
abcdefghi

m= 7
n= 9
inizio check 0
inizio check 1

An occurence with <= 3 differences of the pattern starts at T[2]

inizio check 2
inizio check 3
inizio check 4
inizio check 5

Fatta l'analisi del testo
```

La risposta è corretta in quanto i tre errori sono dati rispettivamente da un mismatch al terzo carattere del testo, un inserimento tra il quarto e quinto carattere del testo, ed una cancellazione del sesto carattere del testo.

Proviamo ora a variare il valore del parametro k (proviamo il valore 2) per vedere se con un numero minore di errori ammissibili non vengono trovate occorrenze valide:

```
k vale 2

Il pattern e':
bxdyegh
```

```
Il testo e':
abcdefghi

m= 7
n= 9
inizio check 0
inizio check 1
inizio check 2
inizio check 3
inizio check 4

Fatta l'analisi del testo
```

Ora invece, mantenendo sempre le due stesse stringhe, aumentiamo il valore di k (proviamo il valore 4) per vedere se effettivamente vengono trovate più occorrenze:

```
k vale 4

Il pattern e':
bxdyegh

Il testo e':
abcdefghi

m= 7
n= 9
inizio check 0

An occurence with <= 4 differences of the pattern starts at T[1]

inizio check 1

An occurence with <= 4 differences of the pattern starts at T[2]


An occurence with <= 4 differences of the pattern starts at T[2]


An occurence with <= 4 differences of the pattern starts at T[2]


An occurence with <= 4 differences of the pattern starts at T[2]

inizio check 2

An occurence with <= 4 differences of the pattern starts at T[3]

inizio check 3

An occurence with <= 4 differences of the pattern starts at T[4]

inizio check 4
inizio check 5
inizio check 6

Fatta l'analisi del testo
```

Ora cambiamo le stringhe di input provando a restringere l'alfabeto a soli quattro caratteri:

```
k vale 1
```

```
Il pattern e':
acagtt

Il testo e':
acaagtt

m= 6
n= 7
inizio check 0

An occurence with <= 1 differences of the pattern starts at T[1]

inizio check 1
inizio check 2

An occurence with <= 1 differences of the pattern starts at T[3]


Fatta l'analisi del testo
```

L’errore ammesso per l’occorrenza iniziante al primo carattere del testo corrisponde a una cancellazione di un carattere ‘a’ nella terza o nella quarta posizione del testo. L’errore ammesso per l’occorrenza iniziante al terzo carattere del testo corrisponde a un inserimento di un carattere ‘c’ tra la terza e la quarta posizione del testo.

Analogamente a quanto fatto prima proviamo a modificare il valore del parametro k per vedere l’effetto prodotto sulle occorrenze trovate dal programma. Proviamo con un k = 0:

```
k vale 0

Il pattern e':
acagtt

Il testo e':
acaagtt

m= 6
n= 7
inizio check 0
inizio check 1

Fatta l'analisi del testo
```

Giustamente non viene rilevata alcuna occorrenza in quanto il pattern non ricorre esattamente come sottostringa del testo.

Proviamo ora a testare il programma con un pattern la cui occorrenza non inizi proprio ai primi caratteri del testo ma al centro della stringa:

```
k vale 2

Il pattern e':
agagtgt

Il testo e':
acgttagtgagtac
```

```
m= 7
n= 14
inizio check 0
inizio check 1
inizio check 2
inizio check 3
inizio check 4
inizio check 5

An occurence with <= 2 differences of the pattern starts at T[6]

inizio check 6
inizio check 7
inizio check 8
inizio check 9

Fatta l'analisi del testo
```

Qui gli errori corrispondenti all'occorrenza valida rilevata sono due mismatch in corrispondenza della posizione 8 e 10 del testo.

Infine proviamo il caso limite della riduzione al confronto esatto tra stringhe ponendo k uguale a zero e fornendo come stringa di pattern una stringa che ricorre senza errori più volte nel testo:

```
k vale 0

Il pattern e':
xx

Il testo e':
abxcdxxefxxxg

m= 2
n= 13
inizio check 0
inizio check 1
inizio check 2
inizio check 3
inizio check 4
inizio check 5

An occurence with <= 0 differences of the pattern starts at T[6]

inizio check 6
inizio check 7
inizio check 8
inizio check 9

An occurence with <= 0 differences of the pattern starts at T[10]

inizio check 10

An occurence with <= 0 differences of the pattern starts at T[11]

inizio check 11

Fatta l'analisi del testo
```

Dopo queste prove, eseguite con dei valori di input considerati sufficientemente adatti a testare il funzionamento dell'algoritmo, siamo in grado di dire che l'implementazione fornita è corretta.
A questo punto testiamo l'algoritmo con delle stringhe significative, cerchiamo, cioè, di identificare la presenza del codice di geni all'interno del codice di cromosomi, ed inoltre cerchiamo di fare un confronto con un'implementazione di un algoritmo asintoticamente peggiore, cioè quello basato esclusivamente sulla programmazione dinamica (vedi pseudo-codice a pagina 8-9), misurando i tempi di esecuzione delle due implementazioni. Per fare ciò ci si è affidato alle informazioni, riguardanti i codici di geni e cromosomi, fornite da uno strumento [5] prodotto dal European Bioinformatics Institute (EBI), un organizzazione non-profit facente parte del European Molecular Biology Laboratory (EMBL).
Le dimensioni dei file di testo tra cui sono stati effettuati i confronti sono state regolate in base alla limitata memoria fisica, che ha dovuto contenere le strutture dati necessarie ad eseguire i programmi per il confronto, a disposizione nella macchina.
Abbiamo provato a cercare il gene denominato ACT4_BOMMO all'interno del cromosoma X della specie *Anopheles gambiae* (zanzara), più precisamente nella sezione del cromosoma contenente i caratteri dal 11645158 al 11670259. I risultati delle misurazioni sono espressi nella seguente tabella:

Ricerca del gene ACT4_BOMMO all'interno del cromosoma X.11645158-11670259 della specie *Anopheles gambiae*.			
Dimensione testo 25104	Dimesione pattern 3978	Numero errori ammessi 2	Indice nel testo dell'occorrenza trovata 10215
Tempo (medio) di esecuzione dell'algoritmo di Landau-Vishkin= **7,81 secondi** Tempo (medio) di esecuzione dell'algoritmo di Programmazione dinamica= **10362,86 secondi**			

In seguito abbiamo provato la ricerca del gene denominato ASB12 all'interno del cromosoma X dell'uomo (specie *Homo sapiens*) in un tratto di codice compreso tra la base 62310867 e la base 62317278. I risultati sono i seguenti:

Ricerca del gene ASB12 all'interno del cromosoma X.62310867-62317278 della specie *Homo sapiens*.			
Dimensione testo 7192	Dimesione pattern 6414	Numero errori ammessi 2	Indice nel testo dell'occorrenza trovata 538
Tempo (medio) di esecuzione dell'algoritmo di Landau-Vishkin= **21,92 secondi** Tempo (medio) di esecuzione dell'algoritmo di Programmazione dinamica= **1918,19 secondi**			

Da questi risultati possiamo quindi riconoscere che l'implementazione dell'algoritmo di Landau-Vishkin è comunque migliore di un'implementazione banale di un algoritmo che sfrutta solamente la programmazione dinamica. Quello che è stato comunque possibile osservare durante l'esecuzione è che il tempo di esecuzione dell'algoritmo di programmazione dinamica aumenta in funzione alla differenza tra dimensione del testo e dimensione del pattern, mentre il tempo di esecuzione dell'algoritmo di Landau-Vishkin aumenta in funzione alla dimensione del pattern in quanto il tempo di esecuzione è quasi totalmente implicato dalla fase di analisi del pattern.

Misurazione	Tempo analisi del pattern	Tempo analisi del testo	Tempo totale
1	21,010	0,321	21,331
2	22,521	0,291	22,812
3	21,691	0,601	22,292
4	21,000	0,261	21,261

Dati relativi alls Ricerca del gene ASB12 all'interno del cromosoma X.62310867-62317278 della specie *Homo sapiens*, quindi con dimensione del pattern 6414.

8. Conclusioni

L'implementazione proposta purtroppo sfrutta la costruzione della matrice *MaxLength* il che porta l'intero algoritmo ad una complessità quadratica invece che lineare come descritto nell'articolo di Landau-Vishkin [1] che propone l'utilizzo di un suffix tree nella fase di analisi del pattern.

A questo punto sarà necessario integrare l'implementazione proposta con una implementazione di un algoritmo per la costruzione di suffix tree e fornire una implementazione dell'algoritmo di Harel-

Tarjan per trovare il Lowest-Common-Ancestor in tempo costante, dopo di che si vedrà raggiunta la complessità lineare O(kn) per l'intero algoritmo di Landau-Vishkin come progettato dai suoi due inventori.
Un'altra modifica utile sarà quella di utilizzare database al posto dei file di testo usati per contenere le stringhe di input e le occorrenze in output.

Bibliografia:

- [1] Gad M. Landau, Uzi Vishkin **Fast String Matching with k Differences** (1986).
- [2] Brian W. Kernighan, Dennis M. Ritchie, **Linguaggio C** Seconda edizione (1990).
- [3] Dan Gusfield, **Algorithms on strings, trees, and sequences** (1997).
- [4] Cormen T.H., Leiserson C.E., Rivest R.L, **Introduction to Algorithms**, MIT Press, (1990).
- [5] http://www.ensembl.org (**Ensembl Genome Browser**)
- [6] http://bioinf.dimi.uniud.it/ (**Bioinformatics Lab's Portal**)