Assignment 1, Bioinformatica

Davide Cozzi, 829827

Indice

Capitolo 1

esercizio 1

1.1 Versione 1

Si assumano stringhe su $\Sigma = \{a, c, g, t, A, C, G, T\}$ indicizzate a partire dalla posizione 0, per comodità.
L'algoritmo si divide in due parti:

1. divisione in "token" delle due stringhe. A partire da ogni stringa si ottiene un vettore di sottostringhe di uguali caratteri
2. confronto dei due vettori di sottostringhe

Si procede specificando che:

- $length(X)$ restituisce la lunghezza di X, che sia una stringa o un vettore
- $push(X, Y)$ effettua l'operazione di `push` di Y nel vettore/stringa X
- $string(Y)$ effettua il cast di Y a `string`
- $Y[i, e]$ specifica la sottostringa di Y che va dall'indice i incluso all'indice e incluso
- $lowercase(Y)$ porta Y in *lowercase*

Si ha quindi la prima parte dell'algoritmo, che effettua la divisione in "token".

Esempio 1. *Vediamo un esempio di input e output della funzione* `Splitter`.

- ***input****: "aaataaaggggccccctttttttttttttttcc"*
- ***output****: ["aaa", "t", "aaa", "gggg", "ccccc", "ttttttttttttttt", "cc"]*

Si ha quindi lo pseudocodice:

Algorithm 1 Algoritmo per lo split in "token" delle stringhe

```
function SPLITTER(s)
    result ← [ ]
    n ← length(s)
    last_mismatch ← 0
    if n == 1 then
        push(result, string(s[0]))
        return result
    end if
    tmp ← ""
    for i ← 0 to n do
        if i > 0 and s[i − 1] ≠ s[i] then
            push(result, tmp)
            tmp ← ""
        end if
        push(tmp, s[i])
        if i == (n − 1) then
            push(result, tmp)
        end if
    end for
    return result
end function
```

Una volta ottenuto il vettore dei "token" delle due sequenze in input basta confrontare i due vettori.
Prima di vedere il confronto si specifica che:

- le due sequenze vengono trasformate in *lowercase* per praticità
- si assume che le due sequenze non possono essere stringhe vuote ε

Fatte queste assunzioni si può fare ancora un ultima osservazione prima di procedere con l'algoritmo. Qualora i due vettori prodotti dalla funzione

`Splitter` fossero di lunghezza diversa allora l'algoritmo non potrà in nessun caso dire che ci sia stata un"'infezione" in quanto si assume che non ci siano né *gaps* né *inserimenti*. In altri termini qualora i due vettori siano di diversa cardinalità sicuramente, con queste premesse, non è avvenuta alcuna infezione. Vediamo ora le altre casistiche. Si itera contemporaneamente sull'elemento i-esimo dei due vettori e:

- se i due elementi i-esimi presentano il primo carattere diverso allora, date le premesse, si specifica che non può essere avvenuta un'infezione
- se i due elementi i-esimi presentano il primo carattere uguale allora si procede confrontando i due elementi i-esimi (si assuma X vettore relativo alla "sequenza originale" e Y vettore relativo alla sequenze che si vuole dimostrare l'eventuale infezione):
  - se il primo carattere è "a" allora la lunghezza di $Y[i]$ deve essere minore o uguale a 5 volte quella di $X[i]$, in quanto per ogni "a" in $X[i]$ posso avere al più 5 "a" in $Y[i]$
  - se il primo carattere è "t" allora la lunghezza di $Y[i]$ deve essere minore o uguale a 10 volte quella di $X[i]$, in quanto per ogni "t" in $X[i]$ posso avere al più 10 "t" in $Y[i]$
  - se il primo carattere è "c" o "g" allora la lunghezza di $Y[i]$ deve essere maggiore o uguale di quella di $X[i]$, in quanto per ogni "c" o "g" in in $X[i]$ posso avere un numero indefinito di "c" o "g" in $Y[i]$

Si ha quindi il seguente pseudocodice dell'algoritmo, che ritorna $\top$ o $\bot$ a seconda che sia possibile che $seq2$ sia una versione infettata di $seq1$:

Algorithm 2 algoritmo di verifica dell'infezione

```
1:  function CheckInfection(seq1, seq2)
2:      if length(seq1) == 0 or length(seq2) == 0 then
3:          return ⊥
4:      end if
5:      vecseq1 ← Splitter(lowercase(seq1))
6:      vecseq2 ← Splitter(lowercase(seq2))
7:      check ← ⊤
8:      if length(vecseq1) ≠ length(vecseq2) then
9:          return ⊥
10:     end if
11:     j ← length(vecseq1)
12:     for i ← 0 to j do
13:         if vseq1[i][0] ≠ vseq2[i][0] or ¬check then
14:             return ⊥
15:         end if
16:         if vecseq1[i][0] == 'a' then
17:             check ← (length(vecseq2[i]) ≤ 5 · length(vecseq1[i]))
18:         else if vecseq1[i][0] == 't' then
19:             check ← (length(vecseq2[i]) ≤ 10 · length(vecseq1[i]))
20:         else
21:             check ← (length(vecseq2[i]) ≥ length(vecseq1[i]))
22:         end if
23:     end for
24:     return check
25: end function
```

Esempio 2. *Qualche esempio:*

- ***input**: "ATAGCTC" e "AAATAAAGGGGCCCCCTTTTTTTCC"*
- ***output**: ⊤*

infatti (si usano i colori per rappresentare i vari "token" dei vettori):

$$ATAGCTC$$
$$AAATAAAGGGGCCCCCTTTTTTTCC$$

Avendo che tutti i vincoli sono rispettati.

D'altro canto vediamo un esempio in cui non si può dire di avere una mutazione:

- ***input**: "ATAGCTC" e "AAATAAAAAAGGGGCCCCCTTTTTTTCC"*
- ***output**: $\perp$*

infatti (si usano i colori per rappresentare i vari "token" dei vettori):

$$ATAGCTC$$
$$AAATAAAAAAGGGGCCCCCTTTTTTTCC$$

Avendo che il terzo "token" della prima stringa è A mentre quello della seconda stringa è $AAAAAA$, avendo che viene rotto il vincolo per il quale per ogni A posso avere al più 5 A nella mutazione.
Un altro esempio in cui non si può dire di avere una mutazione è:

- ***input**: "ATAGCTC" e "AAACCTAAAAAAGGGGCCCCCTTTTTTT"*
- ***output**: $\perp$*

infatti (si usano i colori per rappresentare i vari "token" dei vettori):

$$ATAGCTC$$
$$AAACCTAAAAAAGGGGCCCCCTTTTTTT$$

In quanto i due secondi "token", T e CC, corrispondono a caratteri diversi.

1.2 Versione 2

Si assumano stringhe su $\Sigma = \{a, c, g, t, A, C, G, T\}$ indicizzate a partire dalla posizione 0, per comodità.
Indico anche una versione intuitivamente più semplice. In questo caso si scorre la sequenze che si suppone essere originale, tenendo conto dei caratteri uguali ripetuti. Appena si ha un cambio di carattere si verifica se nella sequenza che si vuole dimostrare essere una mutazione di quella originale si ha una corretta sequenza di caratteri secondo la specifica.
Si ha che:

- per comodità le sequenze sono riportate in lowercase e si assume che, in presenza di anche solo una sequenza nulla, l'algoritmo restituisce $\perp$
- `seq1` e `seq2` sono rispettivamente la sequenza originale al sequenza che si vuole dimostrare essere la mutazione
- `co` e `cm` sono le variabili che ogni volta accumulano il conteggio dei caratteri uguali consecutivi rispettivamente sulla sequenza originale e su quella che si suppone mutata
- i e j sono rispettivamente gli indici per la prima e per la seconda sequenza

In poche parole si itera sulla sequenza originale, aggiornando di volta in volta il contatore relativo finché si ha lo stesso carattere. Nel momento in cui si ha un cambiamento o si è arrivato all'ultimo carattere della sequenza si ferma il conteggio e si verifica il conteggio sulla seconda sequenza (anche in questo caso facendo attenzione a non andare *out of bounds*). Qualora la seconda sequenza presenti, all'indice a cui si è arrivati con l'algoritmo, un carattere diverso da quello della prima si può restituire $\perp$. Controllando la seconda sequenza si aggiorna il rispettivo contatore e l'indice. Una volta che anche sulla seconda sequenza si ha un cambio di carattere o si è arrivati alla fine si confrontano i due contatori secondo le specifiche (ad esempio se nella stringa originale avevo due "a" consecutive ne posso avere al più 10 in quella che si vuole verificare essere la mutazione). Finito questo controllo si azzerano i contatori e si verifica che, qualora la sequenza originale sia stata visitata interamente, anche la seconda sia conclusa. In caso contrario si restituisce $\perp$.
Si ha quindi il seguente pseudocodice.

Algorithm 3 algoritmo di verifica dell'infezione, seconda versione

```
1: function CheckInfection(seq1, seq2)
2:     m ← length(seq1)
3:     n ← length(seq2)
4:     if m == 0 or n == 0 then
5:         return ⊥
6:     end if
7:     seq1 ← lowercase(seq1)
8:     seq2 ← lowercase(seq2)
9:     co ← 0
10:    cm ← 0
11:    j ← 0
12:    for i ← 0 to m do
13:        co ← co + 1
14:        if i == n − 1 or seq1[i] ≠ seq[i + 1 then
15:            if seq1[i] ≠ seq2[j] then
16:                return ⊥
17:            end if
18:            while j ≠ n − 1 and seq2[j] == seq2[j + 1] do
19:                cm ← cm + 1
20:                j ← j + 1
21:            end while
22:            cm ← cm + 1
23:            j ← j + 1
24:            if seq1[i] == 'a' and cm ≤ 5 · co then
25:                check ← ⊤
26:            else if seq1[i] == 't' and cm ≤ 10 · co then
27:                check ← ⊤
28:            else if (seq1[i] == 'c' or seq1[i] == 'g') and cm ≥ co then
29:                check ← ⊤
30:            else
31:                return ⊥
32:            end if
33:            co ← 0
34:            cm ← 0
35:            if i == m − 1 and j ≠ n then
36:                return ⊥
37:            end if
38:        end if
39:    end for
40:    return check
41: end function
```

Dal punto di vista pratico questo secondo algoritmo evita la fase di preprocessing vista nel primo algoritmo con la funzione `Splitter`. Anziché avere a priori le sottostringhe di cui confrontare le lunghezze tiene conto di volta in volta dei caratteri uguali consecutivi, confrontando i contatori (motivo per cui anche gli esempi possono essere riadattati anche a questa seconda versione). Dal punto di vista computazionale, assumendo che la funzione `length` abbia costo lineare si ha in entrambi i casi un tempo quadratico nel caso peggiore, anche se si potrebbe fare uno studio più approfondito.

Capitolo 2

Esercizio 2

2.1 Versione 1

Si assumano stringhe su $\Sigma = \{a, c, g, t, A, C, G, T\}$ (anche se come nell'esercizio precedente poi si procederà con il `lowercase`) indicizzate a partire dalla posizione 0, per comodità.
Si fanno le seguenti assunzioni:

- si assume che le mutazione siano solo cambi di base, non avendo quindi inserzioni o delezioni
- si assume che, non avendo inserzioni o delezioni, le due sequenze siano di egual lunghezza per poter avere un input valido per l'algoritmo
- si assume che si può avere una mutazione solo dopo almeno 5 basi non mutate
- si assume che la prima base della sequenza può mutare
- si assume che le sequenze siano tali per cui il loro *kmer-set* coincida con il loro *spettro*. In altri termini le sequenze sono tali per cui si hanno solo *kmer* univoci
- si assume, per pura semplicità, di avere in input sequenze lunghe almeno quanto un singolo *kmer*

In base alle assunzioni precedenti si può anche assumere che, calcolati i due *spettri*, che sono *spettri senza ripetizioni*, relativi alle due sequenze, si abbia che essi siano di cardinalità uguale. Per verificare che il *kmer-set* sia della

stessa cardinalità dello spettro basta verificare che, assumendo $kmers$ il *kmer-set* (calcolato per un certo k) di una certa sequenza seq, non valga:

$$length(kmers) \neq length(seq) - (k - 1)$$

Procediamo quindi descrivendo l'idea dietro l'algoritmo. L'idea principale è quella di calcolare i *kmer* di una lunghezza k non causale e di evitare confronti inutili. Assumendo di avere che una mutazione è possibile sse almeno le 5 basi precedenti non sono mutazioni, in quanto dopo una mutazione abbiamo assunto esserci almeno 5 basi non mutate, sono di fronte ad un caso "limite" del tipo:

$$MBBBBBM$$

Indicando con M le mutazioni e con B le basi non mutate. Ne segue immediatamente che *kmer* calcolati con $k = 6$ sono i *kmer* di cardinalità massima tali per cui posso avere al più una singola mutazione per *kmer*.
Indicando con $seq1$ la sequenza di riferimento e con $seq2$ la sequenza mutata si ha quindi che, per le assunzioni fatte, posso fare un confronto "1:1" dei *kmer-set*, sapendo che essi sono di cardinalità uguale.
Si noti però che un confronto diretto sarebbe "inutile" in quanto, avendo sicuramente almeno 5 basi non mutate dopo una mutazione posso saltare il confronto di 5 *kmer*, in quanto tutti e 5 aggiungerebbero in coda una base sicuramente non mutata. Inoltre, per le assunzioni fatte, è possibile anche solo confrontare, di volta in volta, i due simboli finali dei due *kmer* e non i *kmer* interi in quanto tutto il prefisso del *kmer*, che termina in penultima posizione della stringa, è stato già verificato negli step precedenti. Qualora il simbolo finale sia diverso si è di fronte ad una mutazione e, avendo *kmer-set* di uguale cardinalità e quindi confrontando sempre i due *kmer* i-esimi, è possibile risalire all'indice della mutazione basandosi su tale i e sulla lunghezza dei *kmer*, quindi 6.
Una volta individuata la mutazione si procede a salvare in una struttura adeguata le seguenti informazioni:

- base presente nella sequenza di riferimento, `b1`
- base mutata presente nella seconda sequenza, `b2`
- indice della mutazione, `i`

In merito si assume che `newMutation(b1, b2, i)` restituisca una mutazione con le specifiche indicate come argomento.
Bisogna fare un'ultima osservazione. Si è assunto che la primissima base

possa subire mutazione. Si verifica quindi se le due basi iniziali siano diverse e, qualora lo fossero, si aggiunge la mutazione e ci si sposta ai due *kmer* di posizione 1 nei due *kmer-set*, sapendo che essi non conterranno sicuramente la prima base delle rispettive sequenze e quindi, per le assunzioni fatte, continueranno a contenere al più una mutazione, nell'ultimo carattere. Dopo questa osservazione si procede regolarmente con l'algoritmo.
Per praticità l'algoritmo restituisce una tupla contenente:

1. il vettore delle mutazioni
2. un booleano che segnala se i vincoli del problema sono stati tutti rispettati

Quindi si hanno vari casi:

- un vettore di mutazioni non nullo (in tal caso sicuramente il booleano presenta $\top$)
- un vettore di mutazioni nullo e in tal caso:
  - se il booleano è $\top$ significa che le due sequenze rispettano tutti i vincoli ma non si hanno mutazioni
  - se il booleano è $\bot$ significa che qualche vincolo è stato infranto, ovvero stringhe di lunghezza diversa, stringhe nulle e stringhe che prevedono *kmer* non univoci

L'assunzione di avere distanza 5 tra ogni mutazione non viene verificata e viene assunta come assioma.
Vediamo quindi prima lo pseudocodice e poi qualche esempio.
Esplicitiamo prima la funzione `getKmers` che, data una sequenza *seq* e un intero k, restituisce un vettore di stringhe, che per le assunzioni è sia il *kmer-set* che lo *spettro*, mantenendo l'ordine, quindi il *kmer* che nel vettore ha indice i è il *kmer* che inizia all'indice i della sequenza. Quest'ultimo ragionamento è possibile solo grazie alle assunzioni fatte: *kmer* non univoci non permetterebbero questo. Con $s[i, j]$ si intende la sottostringa di s dall'indice i all'indice j inclusi. Con $s[i,]$ si intende il suffisso di s che inizia all'indice i.
Si noti che la funzione `getKmers`, così esplicitata, consumerebbe la sequenza quindi la si "copia" in una sequenza temporanea tmp.

Algorithm 4 Funzione di calcolo dei *kmer*

```
1: function getKmers(seq, k)
2:     tmp ← seq
3:     kmers ← [ ]
4:     while length(tmp) ≥ k do
5:         push(kmers, tmp[0, k − 1])
6:         tmp ← tmp[1, ]
7:     end while
8:     return kmers
9: end function
```

Algorithm 5 Algoritmo basato su *kmer* per mutazioni

```
1: function CheckMutation(seq1, seq2)
2:     if length(seq1) ≠ length(seq2) then
3:         return ([ ], ⊥)
4:     end if
5:     if length(seq1) == 0 or length(seq2) == 0 then
6:         return ([ ], ⊥)
7:     end if
8:     muts ← [ ]
9:     i ← 0
10:    n ← length(seq1)
11:    seq1 ← lowercase(seq1)
12:    seq2 ← lowercase(seq2)
13:    if seq1[0] ≠ seq2[0] then
14:        push(muts, newMutation(seq1[0], seq2[0], 0))
15:        i ← index + 1
16:    end if
17:    kmers1 ← getKmers(seq1, 6)
18:    kmers2 ← getKmers(seq2, 6)
19:    if length(kmers1) ≠ n − 5 or length(kmers2) ≠ n − 5 then
20:        return ([ ], ⊥)
21:    end if
22:    while i < length(kmers1) do
23:        if kmers1[i][5] ≠ kmers2[i][5] then
24:            push(muts, newMutation(kmers1[i][5], kmers2[i][5], 5 + i))
25:            i ← i + 6
26:        else
27:            i ← i + 1
28:        end if
29:    end while
30:    return (muts, ⊤)
31: end function
```

Dove con i teniamo traccia dei *kmer* da confrontare, eventualmente saltando i confronti superflui sopra descritti.
Vediamo quindi qualche esempio.

Esempio 3. *Siano date in input:*

- *atcttgcattaccgccccaatc*
- *atcttacattaccgtcccaacc*

Le due sequenze rispettano tutte le assunzioni.
Calcolati i kmer si procede con il confronto, sapendo che le due sequenze iniziano con la stessa base. L'indice dell'enumerazione corrisponde all'indice i dell'algoritmo.

0. *confronto "atcttg" con "atctta". Ho un mismatch tra gli ultimi simboli "g" e "a", che sono all'indice 5 essendo noi al kmer di indice 0 (si ha infatti $5 + 0 = 5$). Aggiungo quindi la mutazione* `(g,a,5)` *e aggiorno l'indice al valore $i + 6$, ovvero, in questo caso,* 6. *Salto quindi tutti i confronti e riparto dal 6*
1. ~~*confronto "tcttgc" con "tcttac"*~~
2. ~~*confronto "cttgca" con "cttaca"*~~
3. ~~*confronto "ttgcat" con "ttacat"*~~
4. ~~*confronto "tgcatt" con "tacatt"*~~
5. ~~*confronto "gcatta" con "acatta"*~~
6. *confronto "cattac" con "cattac". Non ho mismatch quindi faccio $i + 1$ e basta andando al confronto successivo*
7. *confronto "attacc" con "attacc". Non ho mismatch quindi faccio $i + 1$ e basta andando al confronto successivo*
8. *confronto "ttaccg" con "ttaccg". Non ho mismatch quindi faccio $i + 1$ e basta andando al confronto successivo*
9. *confronto "taccgc" con "taccgt". Ho un mismatch tra gli ultimi simboli "c" e "t", che sono all'indice 14 essendo noi al kmer di indice 9 (si ha infatti $5 + 9 = 14$). Aggiungo quindi la mutazione* `(c,t,14)` *e aggiorno l'indice al valore $i + 6$, ovvero, in questo caso,* 15. *Salto quindi tutti i confronti e riparto dal 15*

10. ~~*confronto "accgcc" con "accgtc"*~~
11. ~~*confronto "ccgccc" con "ccgtcc"*~~
12. ~~*confronto "cgcccc" con "cgtccc"*~~
13. ~~*confronto "gccccc" con "gtcccc"*~~
14. ~~*confronto "ccccaa" con "tcccaa"*~~
15. *confronto "cccaat" con "cccaac". Ho un mismatch tra gli ultimi simboli "t" e "c", che sono all'indice 20 essendo noi al kmer di indice 15 (si ha infatti* $5+15 = 20$*). Aggiungo quindi la mutazione* `(t,c,20)` *e aggiorno l'indice al valore* $i+6$*, ovvero, in questo caso,* 26*. Essendo l'indice maggiore stretto della cardinalità del kmer-set, interrompo l'esecuzione, sapendo che comunque non potrei avere in ogni caso ulteriori mutazioni*
16. ~~*confronto "ccaatc" con "ccaacc"*~~

Esempio 4. *Più brevemente vediamo un altro input:*

- *tatcttgcattaccgccccaatc*
- *gatcttacattaccgtcccaacc*

In questo casi si noti come la prima base non combaci, si procede quindi salvando la mutazione `(t,g,0)` *e facendo iniziare il confronto tra i kmer di indice 1, ovvero "atcttg" e "atctta", proseguendo poi come nell'esempio precedente.*

2.2 Versione 2

Per pura curiosità si può fare anche un piccolo ragionamento ulteriore. Con le assunzioni date si può costruire un **grafo di De Bruijn** delle due stringhe e vedere che non si hanno cicli, non essendoci *kmer* non univoci all'interno di ciascuna stringa. Le mutazioni possono essere quindi viste come le coppie di caratteri che etichettano l'inizio di una **bubble** nel grafo, ovvero le etichette degli archi di un nodo che ha due archi uscenti, Vista l'essenza di cicli, si ha che il primo *kmer* calcolato per ciascuna delle due sequenze (potrebbero essere due diversi) è un nodo privo di archi entranti, essendo quindi un nodo *source*, e l'ultimo nodo calcolato per ciascuna delle due sequenze (potrebbero essere due diversi) è un nodo privo di archi uscenti, essendo un nodo *sink*.

Basta quindi percorrere un cammino dal primo nodo (al più del caso con mismatch in prima base che verrà trattato dopo) fino ad un nodo *sink*, privo di figli, e ogni volta che si incontra un nodo con due figli salvare la mutazione e proseguire su uno dei due cammini della **bubble**, indifferentemente. Ogni avanzamento di nodo comporta l'incremento di indice utile a salvare la posizione della mutazione.
Si termina quando si arriva ad un nodo privo di figli. Qualora si abbia una mutazione nella prima base allora si avranno due nodi privi di archi entranti che puntano però allo stesso nodo. Si procede quindi aggiungendo la prima base come mutazione e proseguendo con il ragionamento visto sopra a partire da questo nodo a cui puntano.
I due grafi sono visualizzabili in figura 2.1. Facciamo le seguenti assunzioni per comodità:

- la funzione `createDBG(v, k)` prende in input un vettore v di stringhe e un intero k e restituisce un oggetto rappresentante il *grafo di De Bruijn* costruito a partire dalle sequenze contenute in v. Tali sequenze sono in questo caso due e sono supposte valide secondo le assunzioni fatte all'inizio

- si supponga di avere un oggetto `dbg` rappresentante il *grafo di De Bruijn*. Si ha che la funzione `startNodes(dbg)` restituisce un vettore contenente i puntatori ai nodi iniziali/sources, ovvero i nodi senza archi entranti del *grafo di De Bruijn*. Per le assunzioni fatte, ho almeno un nodo iniziale e al più due

- si supponga che ogni nodo sia rappresentato da un oggetto e che, dato un nodo n si abbiano le seguenti funzioni:

  - `label(n)`, che restituisce il *(k-1)-mer* che etichetta il nodo n
  - `outDegree(n)`, che restituisce il numero di archi uscenti dal nodo n. Se $n == null$ il metodo restituisce comunque 0
  - `outLabel(n)`, che restituisce un vettore con le etichette degli archi uscenti dal nodo n
  - `nextNodes(n)`, che fornisce un vettore di puntatori ai nodi successivi/figli di n

Ovviamente avendo due sequenze in input che garantiscono le assunzioni iniziali quando si parla di vettori si parla sempre di vettori che possono essere o di lunghezza uno o due.

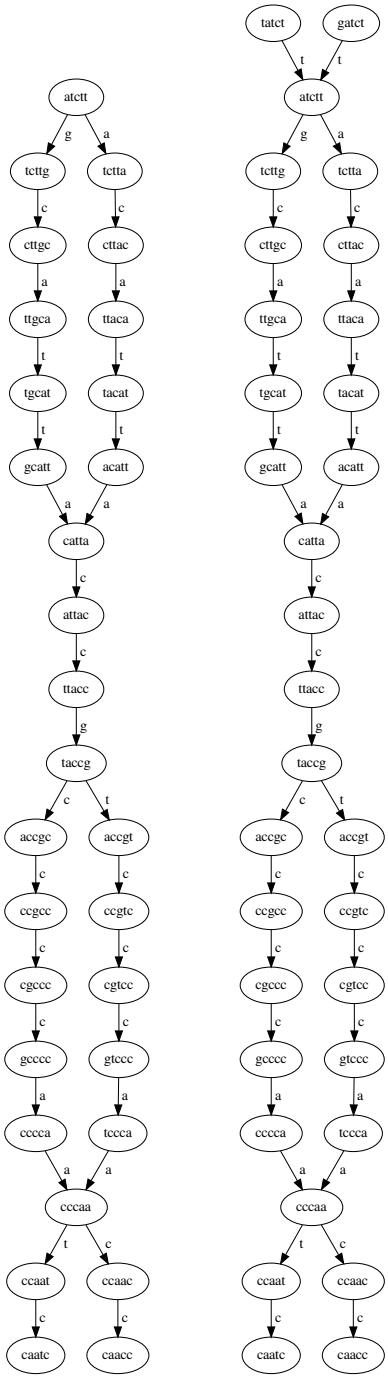


Figura 2.1: Grafi di De Bruijn relativi, rispettivamente, ai due esempi della "versione 1"

Vediamo quindi uno pseudocodice possibile per trovare le mutazione usando il *grafo di De Bruijn*. L'output è dello stesso formato del precedente algoritmo.

Algorithm 6 Algoritmo basato su *kmer* e *grafo di De Bruijn* per mutazioni

```
1: function CheckMutation(seq1, seq2)
2:     if length(seq1) ≠ length(seq2) then
3:         return ([ ], ⊥)
4:     end if
5:     if length(seq1) == 0 or length(seq2) == 0 then
6:         return ([ ], ⊥)
7:     end if
8:     muts ← [ ]
9:     i ← 0
10:    seq1 ← lowercase(seq1)
11:    seq2 ← lowercase(seq2)
12:    dbg ← createDBG([seq1, seq2], 6)
13:    s ← startNodes(dbg)
14:    if length(s) ≠ 1 then
15:        push(muts, newMutation(label(s[0])[0], label(s[1])[0], 0))
16:        i ← i + 1
17:    end if
18:    curr ← s[0]
19:    while ⊤ do
20:        if outDegree(curr) == 2 then
21:            l ← outLabel(curr)
22:            push(muts, newMutation(l[0], l[1], i + 5))
23:            curr ← nextNodes(curr)[0]
24:        else
25:            i ← i + 1
26:            curr ← nextNodes(curr)[0]
27:        end if
28:        if outdegree(curr) == 0 then
29:            break
30:        end if
31:    end while
32:    return (muts, ⊤)
33: end function
```

Si noti che senza ulteriori assunzioni si perde esplicita referenza di quale base della mutazione appartiene alla sequenza di riferimento o a quella mutata, ma un controllo per capirlo è facilmente eseguibile usando l'indice.

2.3 Versione 3

Potenzialmente, in fase di creazione del *grafo di De Bruijn*, potremmo tenere traccia degli archi che "saltano" le *bubble*, come in figura 2.2, per poter simulare qualcosa di analogo alla versione 1 dell'algoritmo, evitando di visitare archi inutili, saltando direttamente dall'inizio di una **bubble**, dopo aver aggiunto la mutazione, alla sua fine, tramite una funzione adeguata.
Ipotizziamo quindi che in fase di costruzione del *grafo di De Bruijn* ogni qual volta si riconosca l'inizio di una **bubble** si provveda a creare, qualora la **bubble** si chiuda, a creare un arco, di tipologia diversa rispetto a quelli standard del grafo (anche solo banalmente indicandolo con 2 anziché 1 nell'ipotetica matrice di adiacenza), che punti alla fine della **bubble stessa**. Ipotizziamo quindi l'aggiunta un metodo, dato un nodo n inizio di una **bubble** (quindi con `outDegree(n)==2`), ovvero `toEndBubble(n)`. Tale metodo restituisce il nodo n' di fine **bubble** (quello successivo con due archi entranti). Qualora `toEndBubble(n)` restituisca *null* si ha che la **bubble** non viene chiusa prima della fine del grafo (indicando che non si ha una mutazione nelle ultime 6 basi).
Il caso di prima base mutata rimane analogo alla versione precedente.
Il salto alla fine della **bubble** comporta un comportamento pari a quello della prima versione dell'esercizio, comportando il medesimo aumento di indice per poter tener corretta traccia della posizione della mutazione.

Possiamo quindi indicare il nuovo pseudocodice:

Algorithm 7 Algoritmo basato su *kmer*, *grafo di De Bruijn* e *bubble* per mutazioni

```
1: function CheckMutation(seq1, seq2)
2:     if length(seq1) ≠ length(seq2) then
3:         return ([ ], ⊥)
4:     end if
5:     if length(seq1) == 0 or length(seq2) == 0 then
6:         return ([ ], ⊥)
7:     end if
8:     muts ← [ ]
9:     i ← 0
10:    seq1 ← lowercase(seq1)
11:    seq2 ← lowercase(seq2)
12:    dbg ← createDBG([seq1, seq2], 6)
13:    s ← startNodes(dbg)
14:    if length(s) ≠ 1 then
15:        push(muts, newMutation(label(s[0])[0], label(s[1])[0], 0))
16:        i ← i + 1
17:    end if
18:    curr ← s[0]
19:    while ⊤ do
20:        if outDegree(curr) ≠ 2 then
21:            i ← i + 1
22:            curr ← nextNodes(curr)[0]
23:        else
24:            l ← outLabel(curr)
25:            push(muts, newMutation(l[0], l[1], i + 5))
26:            curr ← toEndBubble(curr)
27:            i ← i + 6
28:        end if
29:        if outDegree(curr) == 0 then
30:            break
31:        end if
32:    end while
33:    return (muts, ⊤)
34: end function
```

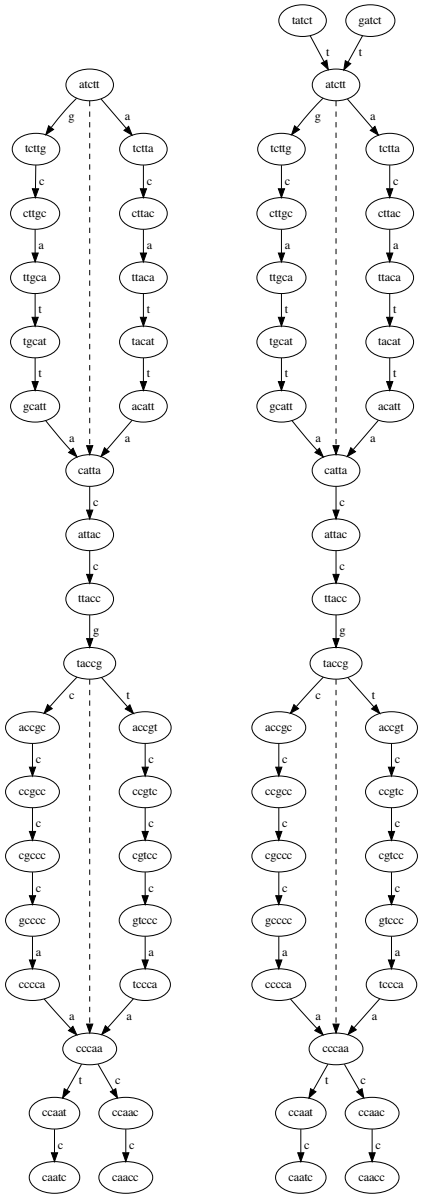


Figura 2.2: Grafi di De Bruijn relativi con i "salti" delle **bubble**, rispettivamente, ai due esempi della "versione 1"

Capitolo 3

Esercizio 3

La **distanza di Hamming** tra due stringhe, che si assumono di uguale lunghezza, è il numero di indici per i quali i due caratteri associati sulle due stringhe sono diversi. È quindi un conteggio delle sostituzioni necessarie per passare da una stringa all'altra.
Ipotizzando di voler studiare la distanza di Hamming tra due sequenze tramite la griglia estesa si può ipotizzare di ragionare solo in termini di una delle due sequenze, ovvero considerare i pesi degli archi in ottica di una sola delle due sequenze, come in figura 3.1. Si associano quindi i seguenti pesi:

- $w(diagonale) = 0$
- $w(verticale) = 0$
- $w(orizzontale) = 1$

o, in modo speculare se si vuole ragionare sull'altra sequenza:

- $w(diagonale) = 0$
- $w(verticale) = 1$
- $w(orizzontale) = 0$

Dove, in entrambi i casi, gli archi diagonali rappresentano un match mentre, a seconda, gli archi verticali o quelli orizzontali tengono conto dei mismatch.
A questo punto si cerca il cammino di peso minimo che parte dal nodo *source*, posto in alto a sinistra, e arriva al nodo *sink*, posto in basso a destra. Essendo il nodo sink in posizione (n, n), assumendo le due sequenze lunghe n, garantisco che, calcolando il cammino che parte dal nodo source $(0, 0)$ e arriva in quel nodo, si abbia il valore della distanza di Hamming, che prevede stringhe

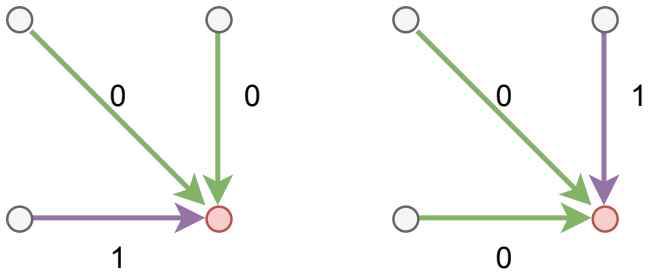
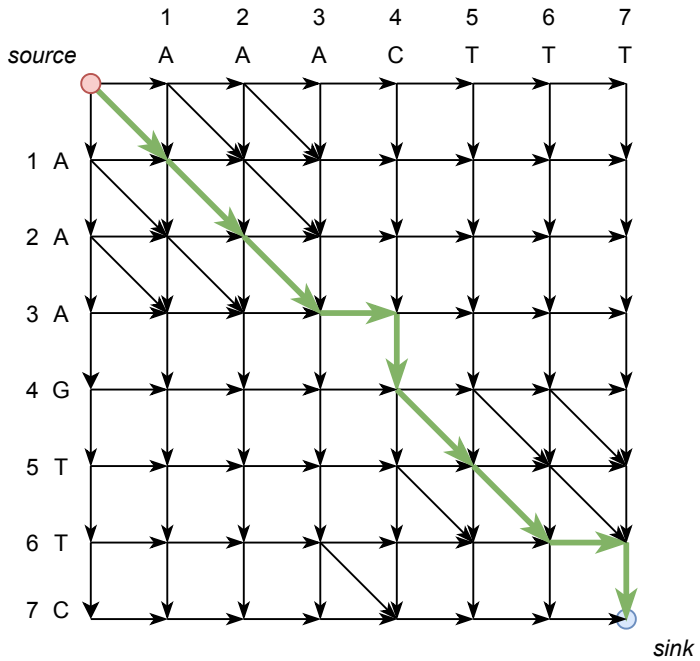


Figura 3.1: Rappresentazione grafica dei pesi degli archi.

di ugual lunghezza. I vari valori degli altri nodi sono da considerarsi "temporanei" e non rappresentanti una distanza di Hamming. Potenzialmente però tutti e soli i nodi (i, i), con $i \in [1, n)$, oltre quindi al nodo sink, potrebbero essere i nodi di fine di potenziali cammini dal source per calcolare la distanza di Hamming tra le sottostringhe di lunghezza i delle due stringhe in input. Prendendo, ad esempio, in input:

- AAAGTTC
- AAACTTT

Si ha un cammino minimo (non l'unico), assumendo costo degli archi in orizzontale pari a 1 e in verticale pari a 0, del tipo:



Dove in verde è segnato il cammino minimo scelto. Avendo solo due archi orizzontali, che ricordiamo avere peso 1 mentre gli altri hanno peso nullo, possiamo concludere che la distanza di Hamming tra le due stringhe è pari a 2.
Per il ragionamento fatto sopra, se mi fermassi in $(3, 3)$, per $i = 3$, avrei la distanza di Hamming tra AAA e AAA. Il cammino dal source a $(3, 3)$ è formato da sole diagonali e quindi ha costo 0, che è appunto al distanza di Hamming tra le due sottostringhe.

Capitolo 4

Esercizio 4

Il problema della **longest common substring** si pone l'obiettivo di estrarre, a partire da due stringhe di lunghezza arbitraria, anche non uguale, la sottostringa, comune ad entrambe, più lunga.
Dal punto di vista della griglia si assegnano i seguenti pesi (come da figura 4.1):

- $w(diagonale) = 1$
- $w(verticale) = -i$
- $w(orizzontale) = -j$

Dove gli archi diagonali rappresentano un match ed i e j sono gli indici che scorrono le due stringhe, ovvero rispettivamente sulle righe e sulle colonne.
Si assume che i nodi abbiano valore $x \geq 0$ quindi **qualora si ottenga un valore negativo come risultato dopo l'attraversamento dell'arco viene messo il valore 0**.
Quando si ha un match quindi il nodo finale ha valore pari a uno più il valore

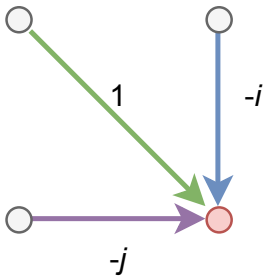


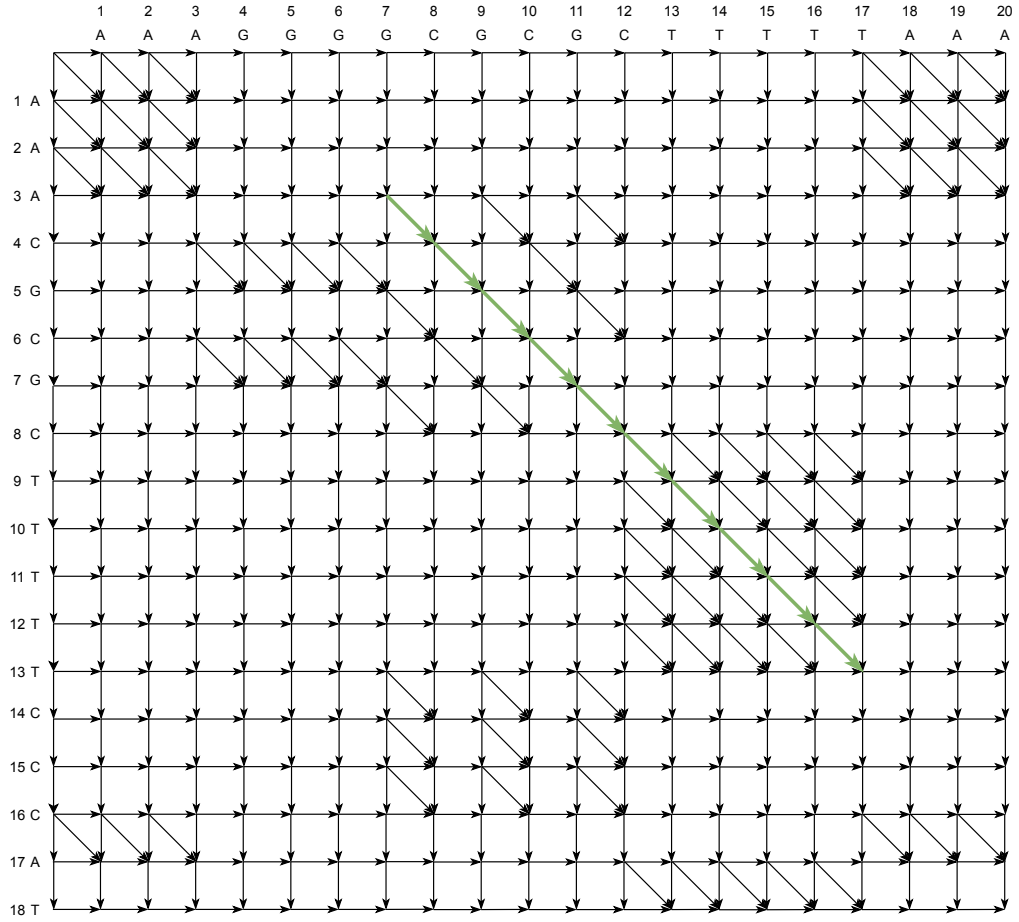
Figura 4.1: Rappresentazione grafica dei pesi degli archi. Si ricorda che nel nodo finale, in rosso, posso comunque avere solo valori ≥ 0

del nodo sorgente dell'arco diagonale.
Si ragiona quindi tenendo traccia del valore massimo raggiungo, tenendo traccia, per una maggior efficienza, anche delle coordinate. Una volta completata la griglia si avrà che il valore massimo corrisponde al nodo *sink* del cammino composto dalla più lunga sequenza possibile di archi diagonali consecutivi. Qualsiasi altra "operazione", con archi orizzontali o diagonali, comporta infatti una perdita di punteggio e ogni volta che un cammino diagonale termina viene azzerato il punteggio, tramite pesi negativi pesati sugli indici. Azzerando ogni volta si permette di avere la costruzione di un eventuale cammino diagonale che assegna man mano il valore corrispondente alla lunghezza del cammino stesso, avendo che la diagonale pesa 1. Il valore massimo quindi altro non è che la lunghezza della *longest common substring*. Sapendo che archi diagonali corrispondono a match tra le due sequenze si ottiene che tale cammino corrisponde alla *longest common substring*.
Volendo si può scegliere di salvare in un vettore i valori massimi qualora coincidano, per ottenere eventuali più *longest common substring* qualora ce ne siano.
In altri termini si cerca il sottocammino più pesante.
ipotizzando di non "azzerare" ogni volta in caso di mismatch otterrei comunque una griglia dover il nodo massimo mi permetterebbe di risalire la diagonale ritrovando la *longest common substring* ma tale nodo non potrebbe avere come valore la lunghezza della stessa, in quanto non ricomincerei il conto da 0 ogni volta che creo un nuovo cammino fatto di diagonali. Per ricostruire la *longest common substring* si parte dal nodo *sink*, come detto definito dal valore massimo calcolato. Si aggiunge il carattere corrispondente e si risale tutto il cammino diagonale, aggiungendo di volta in volta in testa il carattere letto. Ci si ferma quindi quando non si ha più un nodo il cui punteggio è stato ottenuto tramite un arco diagonale.

Vediamo quindi un esempio pratico. Siano date:

- AAACGCGCTTTTTCCCAT
- AAAGGGGCGCGCTTTTTAAA

Si costruisce quindi la griglia e si identifica il cammino diagonale più lungo:



Ricostruendo si ha che tale cammino identifica un massimo pari a 10. Il sottocammino più pesante è quindi quello rappresentato in verde. Ipotizzando quindi di poter 'risalire" la dialogale si ricostruisce:

CGCGCTTTTT

che è appunto la *longest common substring* delle due sequenze in input.

Capitolo 5

Link al codice

Durante lo svolgimento dell'assignment è stato elaborato anche il codice in *Rust* per i vari esercizi, con i vari unit test correlati. L'unica eccezione è l'algoritmo di studio del *grafo di De Bruijn* del quale comunque è presente una bozza di codice per creare lo stesso e generare il relativo file `.dot`.
Il codice, insieme al T_EX di questa relazione, è disponibile all'indirizzo: `https://github.com/dlcgold/es_bio`.
Per eseguire i test:

```
> cd 1/esbio
> cargo test
```

Per eseguire i test con alcune stampe:

```
> cd 1/esbio
> cargo test -- --nocapture
```

Per visualizzare la piccola documentazione:

```
> cd 1/esbio
> cargo doc --open
```