Assignment 3, Bioinformatica

Davide Cozzi, 829827

Indice

Capitolo 1

Esercizio

1.1 Svolgimento

Per semplicità si assume che la sequenza in input è data su alfabeto $\Sigma = \{A, C, G, T\}$.
Viene data in input la sequenza:

$$S = ACCGCGCTCGCGTACCTT$$

con l'obiettivo di dare una rappresentazione succinta del suo **grafo di De Bruijn** dato $k = 5$ (in modo da avere nodi di lunghezza pari a 4 come da specifica):

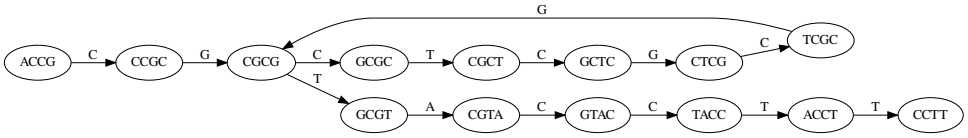


Figura 1.1: Grafo di De Bruijn della sequenza S.

Per poter avere una rappresentazione corretta del **grafo di De Bruijn succinto** bisogna in primis procedere con l'apposizione di una quantità di \$ pari a $k - 1$ in modo da ottenere un nodo composto da soli \$ nel nostro grafo e permettere la corretta visita dello stesso (come si vedrà dopo a causa dell'array F). Avendo $k = 5$ si ottiene quindi:

$$S = \$\$\$\$ACCGCGCTCGCGTACCTT$$

Si procede quindi alla rappresentazione succinta del grafo tramite:

- l'array *last*, per rappresentare la presenza multipli archi in uscita da un nodo
- l'array W, per rappresentare le etichette degli archi. Si segnala che si possono avere etichette marcate come *negative*, tramite il simbolo "-", che segnalano eventuali archi con la stessa etichetta che puntano allo stesso nodo partendo da due nodi diversi. Quest'ultimo dettaglio è comunque solo interno alla rappresentazione, non aspettandoci query, nello studio del grafo, con simboli già segnati negativi
- l'array F, per rappresentare in modo compatto l'array che sarebbe formato degli ultimi caratteri delle etichette dei nodi, ordinate in ordine lessicografico "da destra". Tale array F lo possiamo memorizzare in modo compatto, in "forma" analoga alla funzione C di un ipotetico **FM-index** costruito sull'array degli ultimi caratteri delle etichette dei nodi, grazie al fatto che i simboli sono ordinati lessicograficamente. Diventa quindi fondamentale avere il primo nodo del grafo con etichetta di soli \$, in modo da avere almeno un \$ nell'array degli ultimi simboli e poter costruire, nonché poi usare, correttamente l'array F

Si ottiene quindi rappresentazione del grafo come da tabella 1.1 (dove, dei valori in tabella, solo *last* e W sono in memoria mentre le altre informazioni sono mostrate per semplicità nello studio dell'esercizio).
Come detto in memoria si ha anche l'array F, una rappresentazione compatta di quello che nella tabella è indicato con *last char label*. L'array F prodotto per l'esercizio è quindi:

$$F[\$] = 0$$
$$F[A] = 1$$
$$F[C] = 3$$
$$F[G] = 11$$
$$F[T] = 15$$

Una rappresentazione grafica del grafo è visibile in figura 1.2.
Si cerca quindi di capire, dato un nodo v, come passare ad un nodo v', tramite uno degli archi uscenti da v, etichettato con l. Avendo quindi v e v' rappresentati con un indice, avendo che la numerazione degli stessi è indicizzata a partire da 0, si cerca di costruire una funzione $v' = outgoing(v, l)$

edge index	node index	***last***	***W***	label	last char label
0	0	1	A	$$$$	$
1	1	1	C	$$$A	A
2	2	1	C	CGTA	A
3	3	1	C	$$AC	C
4	4	1	C	GTAC	C
5	5	1	G	$ACC	C
6	6	1	T	TACC	C
7	7	1	G	CCGC	C
8	8	1	T	GCGC	C
9	9	1	G-	TCGC	C
10	10	1	G	GCTC	C
11	11	1	C	ACCG	G
12	12	0	C	CGCG	G
13	12	1	T	CGCG	G
14	13	1	C	CTCG	G
15	14	1	T	ACCT	T
16	15	1	C	CGCT	T
17	16	1	A	GCGT	T
18	17	1	$	CCTT	T

Tabella 1.1: Tabella con le informazioni in memoria, *last* e W, per l'esercizio con l'aggiunta di alcune informazioni extra. Come informazioni aggiuntive troviamo in primis gli indici dei nodi e gli indici degli archi, che verranno spesso usati nell'esercizio. Troviamo poi le label dei nodi e l'array degli ultimi simboli delle label dei nodi, ordinate lessicograficamente a partire da destra. Quest'ultima colonna viene memorizzata in modo compatto nell'array F.

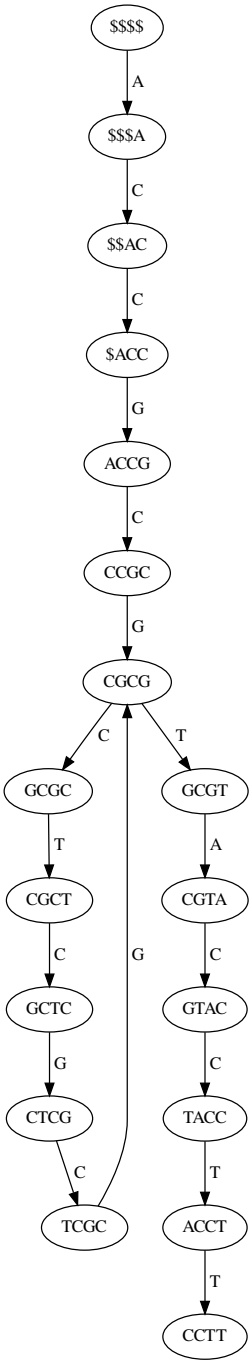


Figura 1.2: Rappresentazione grafica, comoda per visualizzare quanto trattato, del grafo di De Bruijn della sequenza S, con i dovuti \$ a sinistra e relativi nodi nel grafo docuti alla rappresentazione succinta del grafo stesso.

che, preso in input un nodo v, in forma di indice, e un'etichetta l, calcola v', qualora esista, che è il nodo, sempre in forma di indice, in cui si arriva partendo da v tramite un arco etichettato con l. Qualora tale nodo v' non esista si restituisce -1. Si ha quindi:

$$outgoing(v, l) = \begin{cases} v' & \text{se tale nodo } v' \text{ esiste} \\ -1 & \text{altrimenti} \end{cases}$$

Il calcolo di tale funzione, nonché in generale l'intero studio dei **grafi di De Bruijn succinti**, si basa su due funzioni che bisogna definire: **rank** e **select**.

Definizione 1. *Si definisce la funzione* $\mathbf{rank_s(x)}$ *come la funzione che permette di calcolare, in un vettore V di n elementi, il numero di occorrenze di un simbolo s tra la posizione 0 e una posizione x passata come argomento alla funzione.*

Definizione 2. *Si definisce la funzione* $\mathbf{select_s(x)}$ *come la funzione che permette di calcolare, in un vettore V di n elementi, l'indice dell'x-esimo simbolo uguale a s.*

Nel nostro caso parlando di $rank_s$ e $select_s$, per un simbolo s, si assume che si lavora sempre sull'array W mentre con $rank_1$ e $select_1$ si assume di lavorare sull'array last.
Per comodità, inoltre, si suppone di essere in grado di effettuare queste due operazioni anche su un simbolo s marcato come negativo senza ulteriori specifiche, come se fosse un simbolo come un altro, ovviamente diverso dal suo corrispondente non negato. A livello di codice, nella bozza implementata, il discorso in merito sarà "meno ottimizzato".
Per poter discutere della funzione $outgoing(v, l)$ abbiamo bisogno di un'altra operazione fondamentale per "navigare" un **grafo di De Bruijn** memorizzato in forma succinta, la funzione $forward(e)$.

Definizione 3. *Si definisce la funzione* $\mathbf{forward(e)}$ *come la funzione che restituisce l'indice dell'ultimo arco, che quindi presenta* 1 *nell'array last, del nodo a cui si punta tramite l'arco e. Qualora tale nodo non esista, non potendo quindi restituire l'indice del suo ultimo arco, si restituisce* -1.

Vediamo quindi indicativamente il funzionamento di tale funzione, notando come il ragionamento può essere pensato in modo a tratti simile, ma, come si vedrà, non in modo identico, a quanto si farebbe con LF-mapping tra l'array W e l'array chiamato *last char label*, in tabella 1.1:

- si estrae in primis il simbolo che etichetta l'arco e, banalmente usando l'array W:
$$s \leftarrow W[e]$$
Se tale simbolo è uguale a \$ si restiuisce -1 e si interrompe l'esecuzione in quanto sappiamo che non possiamo avere archi etichettati con \$, per costruzione, e che nell'array W essi rappresentano appunto un arco fittizio (rappresentando un arco ipotetico, non avendo quindi mancanza di informazione in W, uscente da un nodo "pozzo")

- si calcola quindi, sull'array W, quanti sono i simboli s presenti fino all'indice e, in modo da ottenere un ordine relativo del carattere s su W. Per farlo basta usare la funzione $rank_s$:
$$rindex \leftarrow rank_s(e)$$

- si cerca l'indice della prima occorrenza di s su *last char label*, tramite l'array F:
$$focc \leftarrow F[s]$$

- bisogna poi capire quanti nodi ci siano prima dei nodi "candidati", ovvero quei nodi che hanno etichette con ultimo simbolo pari a s. Per farlo si verificano quanti 1 ci sono sull'array *last* prima della prima occorrenza di un nodo "candidato", usando la funzione $rank_1$:
$$nprevs \leftarrow rank_1(focc - 1)$$

- infine si può ottenere l'indice dell'ultimo arco uscente dal nodo a cui si punta tramite l'arco e. Sapendo quanti simboli s precedono il simbolo voluto su W (che è stato definito sopra come $rindex$) e quanti nodi ho prima dei candidati possibili (che è stato definito sopra come $nprevs$) si può usare la funzione *select* sull'array *last*:
$$e' \leftarrow select_1(rindex + nprevs)$$
e restituire e'.

Un esempio di pseudocodice per la funzione $forward(e)$ è indicato nell'algoritmo 1.
Si è notato come in pratica si è fatto qualcosa di analogo all'LF-mapping ma tenendo traccia di eventuali nodi con più archi uscenti. Le difficoltà di usare direttamente LF-mapping vengono elencati alla fine dell'esercizio.

Algorithm 1 Pseudocodice della funzione $forward$

```
function FORWARD(e)
    s ← W[e]
    if s == '$' then
        return −1
    end if
    rindex ← rank_s(e)
    focc ← F[s]
    nprevs ← rank_1(focc − 1)
    return select_1(rindex + nprevs)
end function
```

Esempio 1. *Vediamo un esempio pratico di esecuzione della funzione $forward$. Si supponga $e = 13$, che corrisponde all'ultimo arco uscente dal nodo $CGCG$. Si ha quindi, come verificabile in tabella 1.1 e nella rappresentazione compatta di F:*

- $s = W[13] = T$
- $rindex = rank_T(13) = 3$
- $focc = F[T] = 15$
- $nprevs = rank_1(15 - 1) = rank_1(14) = 14$
- $e' = select_1(3 + 14) = select_1(17) = 17$

La conferma si ha, oltre che tramite figura 1.2, dal fatto che, l'arco di indice 17 *è l'ultimo, nonché il solo, relativo al nodo $GCGT$. L'arco* 13*, etichettato T, comporta infatti:*

$$CGCG \xrightarrow{T} GCGT$$

avendo conferma che la funzione è stata eseguita correttamente, avendo $GCGT$ l'ultimo arco, nonché l'unico, indicizzato 17 in tabella 1.1.
In merito al discorso dell'LF-mapping è interessante notare come effettivamente ci si sia spostati dalla terza T su W alla terza T su last char label.

Passiamo quindi allo studio effettivo della funzione $outgoing(v, l)$.
La funzione indicativamente esegue i seguenti passaggi:

- se simbolo in input è esattamente \$ allora la funzione termina, restituendo -1, per lo stesso discorso fatto in merito nella spiegazione della funzione $forward(e)$
- si identifica il range di indici che indicizzano gli archi uscenti dal nodo v, avendo che tali archi saranno consecutivi nella nostra rappresentazione succinta.
  Nel dettaglio questa operazione viene effettuata tramite due operazioni di $select_1$ sull'array $last$, array che appunto segnala qualora un nodo abbia più archi uscenti:

  $$noderange(v) \leftarrow (select_1(v)+1, select_1(v+1)) = (findex, lindex)$$

  con *findex* che abbrevia *first index* mentre *lindex* abbrevia *last index*. Avendo che i due estremi di tale intervallo, appunto $select_1(v)+1$ e $select_1(v+1)$ segnalano, rispettivamente, il primo e l'ultimo arco uscente dal nodo v (che si ricordi ha lo stesso *node index* per ogni arco uscente, come visibile in tabella). Si ha quindi una tupla di indici su archi
- a priori non si può sapere inoltre né se esista un arco etichettato con l che parta da v e arrivi in un qualche v' né tantomeno se, nella rappresentazione succinta, tale l sia marcato come negativo nella nostra rappresentazione, ricordando che in input avrò sicuramente un l non segnato negativo avendo "trasparenza" su questo aspetto. Il problema è capire quale sia il corretto arco da seguire. In input alla funzione si ha infatti un indice di nodo, nodo che potrebbe avere più archi uscenti, coi rispettivi indici diversi (ma consecutivi nella rappresentazione succinta) ed eventuali segnature negative. Si hanno quindi tre casistiche possibili, che coprono il fatto che tale arco esista (segnato negativo o meno) o che non esista:

  1. si considera l come non segnata. Tramite:

     $$nl \leftarrow rank_l(lindex)$$

     con $lindex$ calcolato nello step precedente tramite $noderange(v)$, conto quanti archi etichettati l ho fino a quell'indice. Bisogna quindi trovare l'indice dell'nl-esimo arco etichettato mediante l, tramite:

     $$edge \leftarrow select_l(nl)$$

Si è quindi ottenuto il corretto indice dell'arco uscente da v ed etichettato con l.
Bisogna quindi verificare che tale arco sia effettivamente nel range calcolato sopra, controllando che valga:

$$lindex \leq edge \leq findex$$

Se esiste un arco etichetto l che esce da v sicuramente è in quel range di posizioni sull'indice degli archi.
Qualora non valga si procede con la casistica 2 in quanto se fosse segnato negativo le funzioni rank e select lavorerebbero solo sul simbolo segnato negativo.
Qualora invece valga si procede a calcolare l'indice del nodo v' effettuando in primis:

$$edge' \leftarrow forward(edge)$$

per ottenere l'indice dell'ultimo dell'ultimo arco del nodo a cui si punta tramite l'arco $egde$. A questo punto basta usare la funzione $rank_1$ (con in aggiunta -1 per l'indicizzazione), sull'array $last$ per mappare l'indice dell'arco $edge'$ nel corrispettivo nodo di partenza v':

$$v' \leftarrow rank_1(edge') - 1$$

Si interrompe quindi l'esecuzione

2. si considera l come segnata negativa, trasformando quindi l in $l-$, e si effettuano i medesimi passaggi della casistica 1. Questa casistica copre quei casi per cui si abbiano più nodi con archi egualmente etichettati che puntano ad un nodo v'
3. se nessuno dei due casi precedenti è soddisfatto si restituisce -1, avendo che non esiste un arco etichettato con l che esce dal nodo v. Si interrompe quindi l'esecuzione

Sia nella funzione $forward(e)$ che nella funzione $outgoing(v, l)$ si assume, per semplicità, che gli indici di archi e nodi, nonché i simboli che etichettano gli archi, passati come argomento alle funzioni, siano coerenti con la rappresentazione succinta, non avendo indici "out of bound" o simboli non presenti nell'alfabeto usato.

Algorithm 2 Pseudocodice della funzione *outgoing*

function OUTGOING(v, l)
  if $l == \text{'\$'}$ **then**
    return -1
  end if
  $(findex, lindex) \leftarrow noderange(v)$
  $nl \leftarrow rank_l(lindex)$
  $edge \leftarrow select_l(nl)$
  if $lindex \leq edge \leq findex$ **then**
    $edge' \leftarrow forward(edge)$
    return $rank_1(edge') - 1$
  end if
  $l \leftarrow l + \text{' - '}$
  $nl \leftarrow rank_l(lindex)$
  $edge \leftarrow select_l(nl)$
  if $lindex \leq edge \leq findex$ **then**
    $edge' \leftarrow forward(edge)$
    return $rank_1(edge') - 1$
  end if
  return -1
end function

Vediamo alcuni esempi in modo da coprire le varie casistiche

Esempio 2. *Si prenda ad esempio il nodo 12, nell'immagine 1.2 etichettato con CGCG, come verificabile in tabella 1.2.*
Tale nodo presenta come noderange:

$$noderange(12) = (12, 13)$$

Avendo, infatti, complessivamente due archi uscenti dal nodo.
Vediamo quindi tre ipotetici archi uscenti, uno etichettato con T, uno con C e uno con A (che non esiste).
Iniziamo con T, considerata inizialmente non segnata:

- *si ha in primis:*
$$nl = rank_T(13) = 3$$
- *si ha quindi:*
$$edge = select_T(3) = 13$$
- *avendo che vale $12 \leq edge \leq 13$ calcolo (come visto all'esempio 1 per la funzione $forward$, esempio basato per completezza sugli stessi dati):*
$$edge' = forward(13) = 17$$
- *infine restituisco v':*
$$v' = rank_1(17) - 1 = 17 - 1 = 16$$
interrompendo l'esecuzione avendo:
$$outgoing(12, T) = 16$$

Si può verificare, tramite la tabella 1.2, che il nodo di indice 16 è etichettato con GCGT, che è corretto avendo:

$$CGCG \xrightarrow{T} GCGT$$

Vediamo il caso con C, considerata inizialmente non segnata:

- *si ha in primis:*
$$nl = rank_C(13) = 6$$
- *si ha quindi:*
$$edge = select_C(6) = 12$$

- *avendo che vale* $12 \leq edge \leq 13$ *calcolo:*

$$edge' = forward(12) = 8$$

- *infine restituisco* v'*:*

$$v' = rank_1(8) - 1 = 9 - 1 = 8$$

interrompendo l'esecuzione avendo:

$$outgoing(12, C) = 8$$

Si può verificare, tramite la tabella 1.2, che il nodo di indice 8 *è etichettato con* $GCGC$*, che è corretto avendo:*

$$CGCG \xrightarrow{C} GCGC$$

Vediamo infine il caso con A*, considerata inizialmente non segnata:*

- *si ha in primis:*

$$nl = rank_A(13) = 1$$

- *si ha quindi:*

$$edge = select_A(2) = 0$$

- *avendo che non vale* $12 \leq edge \leq 13$ *procedo a considerare* A *come se fosse segnata negativa*

Considero quindi $A-$*:*

- *si ha in primis:*

$$nl = rank_{A-}(13) = 0$$

- *si ha quindi:*

$$edge = select_{A-}(0) = -1$$

- *avendo anche in questo caso che non vale* $12 \leq edge \leq 13$*, passando quindi all'unico caso rimanente, la casistica 3, avendo:*

$$outgoing(12, A) = -1$$

e interrompendo l'esecuzione

La funzione infatti restituisce -1*, avendo che dal nodo* $CGCG$ *non si hanno archi uscenti etichettati con* A*, come verificabile nell'immagine 1.2 e nella tabella 1.1.*

Esempio 3. *Vediamo un ulteriore esempio, dove si prende il nodo 9, nell'immagine 1.2 etichettato con TCGC, come verificabile in tabella 1.1. Questa scelta non è casuale, infatti è uno dei due nodi, insieme a quello etichettato con CCGC, che, tramite un arco etichettato con G, punta al nodo etichettato con CGCG.*
Considero quindi di avere in input il simbolo G, avendo:

$$v' = outgoing(9, G)$$

Procediamo quindi con i medesimi conti di sopra, avendo innanzitutto che:

$$noderange(9) = (9, 9)$$

Avendo infatti un solo arco uscente da tale nodo.
Inizialmente G viene considerata non segnata:

- *si ha in primis:*
$$nl = rank_G(9) = 2$$
- *si ha quindi:*
$$edge = select_G(2) = 7$$
- *avendo che non vale* $9 \leq edge \leq 9$ *procedo a considerare l come se fosse segnata negativa*

Questo è un risultato potenzialmente atteso in quanto si hanno più archi etichettati con G che puntano al successore di v.
Considero quindi G−:

- *si ha in primis:*
$$nl = rank_{G-}(9) = 1$$
- *si ha quindi:*
$$edge = select_{G-}(1) = 9$$
- *avendo che vale* $12 \leq edge \leq 13$ *calcolo:*
$$edge' = forward(9) = 13$$
- *infine restituisco* v'*:*
$$v' = rank_1(13) - 1 = 13 - 1 = 12$$
interrompendo l'esecuzione avendo:
$$outgoing(9, G-) = 12$$

Si può verificare, tramite la tabella 1.2, che il nodo di indice 12 *è etichettato con CGCG, che è corretto avendo:*

$$TCGC \xrightarrow{G} CGCG$$

1.2 Considerazioni su LF-mapping

Come si è visto ho svolto l'esercizio seguendo quanto indicato da Alex Bowe. All'inizio ho provato a muovermi sulla rappresentazione succinta del grafo tramite l'**FM-index**, con C, Occ e **LF-mapping**, costruito sull'array W ma questo ha comportato diverse problematiche tra cui:

- la trattazione degli archi segnati negativi comportava diverse complicanze, ignorarli poteva comportare di avere simboli con cardinalità diverse tra W e l'array *last char label*. Ho provato qualche workaround ma non ne sono "venuto a capo" in modo accettabile
- l'eventuale presenza di più nodi finali comportava numeri diversi di \$, anche in questo caso vari workaround sono stati invano tentati
- i problemi sopra espressi si propagavano nel passaggio tra indici di archi e di nodi etc…
- i vari workaround, posto che, come anticipato, facendo test su vari grafi ho scoperto non essere nemmeno universalmente funzionanti, introducevano una complessità computazionale spesso eccessiva (soprattutto a fronte del *teorico* tempo costante delle funzioni *rank* e *select*) nonché un aumento di quella spaziale, dovendo tenere in memoria, ad esempio, informazioni aggiuntive sulle posizioni dei simboli \$ etc…

Non essendo riuscito a venire a capo, in modo soddifacente e universale, di queste problematiche ho scelto di seguire quanto detto da Alex Bowe.

1.3 Link al Codice e Considerazioni

Per lo svolgimento dell'esercizio, ho usato un abbozzato codice in Rust, disponibile al link `https://github.com/dlcgold/succint-dbg-rs`, che avevo scritto in queste settimane per comprendere al meglio il funzionamento dei grafi di De Bruijn succinti. Questo codice è un porting "semi-pariziale" di quanto fatto da Bowe, avendolo estratto dalla spiegazione sul sito e da una sua piccola implementazione "mockup" in Python.
Segnalo che tale implementazione non è comunque performante, a livello spaziale, quanto voluto in quanto tengo in memoria anche le strutture necessarie per effettuare le *rank* e le *select*, accessibili tramite un hashmap a seconda

del simbolo in analisi (con anche le strutture per le etichette segnate, dove, a livello di codice, l'essere o meno segnato è rappresentato con un array booleano e non direttamente in W per praticità, anche se si potrebbe fare un'analisi più approfondita in merito al potenziale vantaggio spaziale di memorizzare due `Vector`, uno di `char` e uno di `bool`, rispetto che ad uno di `str`).
Viene infatti calcolato un bitvector (non sembra esserci alternativa ad usare un bitvector nella libreria usata per *rank* e *select*) per ogni simbolo su W (con 1 quando si ha il simbolo e 0 altrimenti) su cui vengono costruite le varie strutture di rank/select offerte da BioRust (`https://docs.rs/bio/0.32.0/bio/data_structures/rank_select/struct.RankSelect.html`), implementazioni che, per di più, come specificato dalla documentazione della libreria, non sono esattamente in $\mathcal{O}(1)$.
La struttura implementata supporta tutte le funzioni elencate nel sito e vari test sembrano non segnalare problemi di correttezza anche se si segnalano, come già anticipato, problemi di performance spaziali e probabili problemi di performance temporali.
Sono state, infatti, implementate le seguenti funzioni:

- $forward(e)$
- $backward(e)$
- $outdegree(v)$
- $indegree(v)$
- $outgoing(v, l)$
- $incoming(v, l)$
- $successors(v)$
- $label(v)$

Oltre che alle funzioni di stampa su riga di comando delle informazioni memorizzate per rappresentare il **grafo di De Bruijn succinto** e le funzioni alla stampa (tramite uso della funzione $successors(n)$), in formato `.dot`, del grafo completo, del grafo senza i nodi con \$ e senza etichette (per non appesantire la visualizzazione con grafi grandi).
Sono stati implementati i costruttori del **grafo di De Bruijn succinto** a partire da stringhe o anche da file in formato FASTA e FASTQ.

Per eseguire i test:

```
> cargo test
```

Per visualizzare la piccola documentazione:

```
> cargo doc --open
```

Nel dettaglio i test sono basati sul grafo usato da Alex Bowe nella sua pagina. Sono inoltre presenti le stampe su file `.dot`, oltre che di quel grafo, generato a partire da una singola sequenza, anche di altri grafi, ovvero:

- il grafo presente nelle slide sulla tematica dei grafi di De Bruijn succinti, generato a partire da due sequenze
- il grafo usato nel primo assignment, generato a partire da due sequenza
- altri grafi di test a partire da file comunque non particolarmente grandi (sempre per il discorso fatto in precedenza dell'attuale non efficienza spaziale della bozza)

Capitolo 2

Domande di teoria

2.1 Domanda 1

*In questa risposta, per praticità, si assume di lavorare con un set di stringhe in input che sia **substring-free**, ovvero si assume che nessun testo è sotto-stringa di alcun altro testo.*
In primis si ha che:

Definizione 4. *Una stringa Q è definita come un **overlap** su un insieme di n testi $T_k, k \in [1, n]$ (\$ estesi) se esistono due sottoinsiemi $\mathcal{T}_s$ e $\mathcal{T}_p$ tali che, $\forall(T_i, T_J)$, coppia del prodotto cartesiano $\mathcal{T}_s \times \mathcal{T}_p$, si ha che T_i ha la stringa Q come suffisso, tranne il simbolo \$ (che sarebbe alla fine del suffisso ma viene escluso), e T_j ha la stringa Q come prefisso.*

Si ha che, se la stringa Q è un overlap per un insieme di testi, allora il **Q-interval** $[b, e]$ ha le seguenti proprietà:

1. ha ampiezza almeno due in quanto un overlap deve occorrere almeno in due testi

2. contiene un sottointervallo proprio $[b, e']$ nel quale sono indicizzati suffissi esattamente uguali a $Q\$$ infatti gli indici dei T_k (\$ estesi) letti dal suffix array S in $[b, e']$ sono i testi che appartengono a $\mathcal{T}_s$ (che hanno Q come suffisso, eslcuso il simbolo \$)

3. contiene almeno una posizione in $[e' + 1, e]$ tali per cui nella BWT B si ha, in quella posizione l, un simbolo \$, ovvero $B[l] = \$$. Infatti si ha che gli indici dei testi letti dal suffix array S in $[e' + 1, e]$, indici dove si ha il simbolo \$ in B, sono quelli relativi ai testi in $\mathcal{T}_p$, ovvero quelli che hanno Q come prefisso

Come anticipato ci si basa sull'assuzione che il set di stringhe sia *substring-free* in quanto ci permette di dire che, in ottica della seconda proprietà, che nell'intervallo $[b, e']$ non si abbiano valori della BWT B pari a \$. Inoltre, in merito alla terza proprietà, l'assunzione garantisce che la cardinalità dei simboli \$ nell'intervallo $[b, e]$ sulla BWT B sono è pari a quella in $[e' + 1, e]$.

2.1.1 Esempio

Vediamo quindi un esempio pratico di quanto detto.

Esempio 4. *Sia dato in input il seguente il seguente set di stringhe (\$ estese) substring-free:*

$$\mathcal{S} = \{tcat\$, tcag\$, gtca\$\} = \{T_1, T_2, T_3\}$$

e si sceglie:

$$Q = tca$$

Costruiamo quindi il ***generalized SA*** *e la* ***generalized BWT*** *(indicando per praticità anche il suffisso in analisi e indicando per completezza anche il* ***generalized LCP array****, che non viene usato nell'esercizio):*

	SA	**LCP**	**suffisso**	**BWT**
1	(5,1)	-1	\$	t
2	(5,2)	0	\$	g
3	(5,3)	0	\$	a
4	(4,3)	0	a\$	c
5	(3,2)	1	ag\$	c
6	(3,1)	1	at\$	c
7	(3,3)	0	ca\$	t
8	(2,2)	2	cag\$	t
9	(2,1)	2	cat\$	t
10	(4,2)	0	g\$	a
11	(1,3)	1	gtca\$	\$
12	(4,1)	0	t\$	a
13	**(2,3)**	**1**	**tca\$**	**g**
14	**(1,2)**	**3**	**tcag\$**	**\$**
15	**(1,1)**	**3**	**tcat\$**	**\$**

Per il Q scelto si ha:

$$Q = tca \implies |Q| = 3$$

che porta ad avere:

$$\mathcal{T}_p = \{T_1, T_2\} \text{ e } \mathcal{T}_s = \{T_3\}$$

Possiamo riconoscere nella tabella che il tca-interval $[b, e]$ *è:*

$$[b, e] = [13, 15]$$

Che avendo ampiezza tre, confermando la ***prima proprietà****.*
Inoltre all'indice 13 troviamo esattamente il suffisso $Q\$$*, ovvero tca*$\$$*, confermando anche quanto detto nella* ***seconda proprietà****.*
Studiando la ***terza proprietà*** *si ha, usando la stessa notazione usata nella spiegazione teorica, abbiamo:*

$$[b, e'] = [13, 13]$$

e di conseguenza

$$[e' + 1, e] = [14, 15]$$

che è un intervallo che contiene le posizioni $l = 14$ *e* $l = 15$ *che comportano, appunto,* $B[l] = \$$*. Inoltre, sempre per la* ***terza proprietà****, andando a vedere* $S[14]$ *e* $S[15]$ *notiamo che ci si riferisce, rispettivamente, a* T_2 *e* T_1*, che, come atteso, sono contenuti in* $\mathcal{T}_p$*.*
Tornando infine, ora che abbiamo definito e' *alla* ***seconda proprietà****, abbiamo conferma del fatto che il testo, ovvero* T_3*, di cui si ha suffisso in* $[b, e'] = [13, 13]$*, suffisso che nel dettaglio sappiamo essere* $Q\$$*, appartiene a* $\mathcal{T}_s$*, come atteso.*
Inoltre è verificabile anche quanto garantito dall'assunzione ***substring-free****:*

- *in* $l \in [b, e'] = [13, 13]$ *non si ha* $B[l] = \$$*, avendo solo* $B[13] = g$
- *nell'intervallo* $[b, e] = [13, 15]$ *ho due occorrenze di* $\$$*, come nell'intervallo* $[e' + 1, e] = [14, 15]$

2.2 Domanda 2

Si indicano le due definizioni viste di **lcp-interval** e **lcp-interval tree**.

Definizione 5. *Sia dato un testo* T *(*$\$$ *esteso), con il suo suffix array* S *e il suo array* LCP*.*
Definiamo ***lcp-interval*** *di valore* l *come l'intervallo di posizioni* $[i, j]$ *tale che vengano soddisfatte le seguenti proprietà:*

- $i < j$

- $LCP[i] < l$
- $LCP[j+1] < l$
- $LCP[k] \geq l$, *avendo* $i+1 \leq k \leq j$
- $LCP[k] = l$ *per almeno un* $k \in [i+1, j]$, *tale* k *è detto* l*-index*

Il valore l*, quindi, di un lcp-interval* $[i, j]$ *è uguale, per definizione, a (con RMQ ad indicare la funzione* ***Range Minumum Query****):*

$$RMQ(LCP, i+1, j)$$

L'lcp-interval $[i, j]$*, di valore* l*, viene quindi indicato con:*

$$l\text{-}[i, j]$$

In aggiunta alla definizione si hanno alcune osservazioni.
Il valore l è quindi la lunghezza del più lungo prefisso comune tra i suffissi di T che iniziano in posizione $S[i], S[i+1], \ldots, S[j]$, che viene denotato con ω.
Dato $l = 0$ si segnala, inoltre, che:

$$0\text{-}[1, n]$$

è lo 0-interval che ha come ω il prefisso nullo.
Si aggiungono alcune definizioni utili a trattare i **lcp-interval tree**:

Definizione 6. *Si dice che* l'*-*$[g, d]$ *è* ***child*** *di* l*-*$[i, j]$*, che eventualmente viene detto* ***parent****, sse non esiste un altro lcp-interval che racchiude* $[g, d]$ *e che a sua volta è racchiuso in* $[i, j]$*.*

Definizione 7. *Definisco* l'*-interval* $[g, d]$ *con prefisso* ω' *come* ***embedded*** *in un* l*-interval* $[i, j]$*, con prefisso* ω *sse:*

$$i \leq g < d \leq j$$

Quindi possiamo dire che l'intervallo $[i, j]$ *"racchiude" l'intervallo* $[g, d]$*.*
In tal caso si ha che :

$$l' > l$$

e quindi che ω *è prefisso di* ω'*.*

Possiamo quindi dare la definizione di **lcp-interval tree**.

Definizione 8. *Si definisce* ***lcp-interval tree*** *come un albero radicato* (V, A) *tale che:*

- *V è l'insieme degli lcp-interval*
- *A è l'insieme di tutte le relazioni "parent-child"*
- *la radice corrisponde a* 0-$[1, n]$

Per descrivere il legame tra **lcp-interval tree** e **suffix tree** definiamo quest'ultimo.

Definizione 9. *Dato un testo T ($ esteso) di n caratteri il* ***suffix tree*** *di T è un albero radicato con:*

- *n foglie etichettate da 1 a n*
- *ogni nodo interno ha almeno 2 child*
- *ogni arco è etichettato da una sottostringa di T*
- *non esistono due archi uscenti da un nodo con lo stesso carattere iniziale per l'etichetta*
- *la concatenazione delle etichette dalla radice alla foglia i è il suffisso i-esimo, $T[i:]$*

Per conludere la lista di consocenze necesarie a descrivere il legame tra tra **lcp-interval tree** e **suffix tree** definiamo il concetto di **path label**.

Definizione 10. *Dato un nodo w definisco la* ***path label*** *L_w come la concatenazione delle etichette dalla radice a w. Se w è una foglia i allora la path label è il suffisso $T[i:]$*

Possiamo quindi legare le due strutture.
Per farlo si consideri un suffix tree ST, costruito a partire dal testo T ($ esteso), al quale vengono virtualmente rimosse le foglie (e conseguentemente gli archi entranti in esse), ottenendo un albero che, per comodità, chiamo ST'. Si costruisca anche, sempre a partire dal testo T, un lcp-interval tree, che chiamo $LCPIT$.
È possibile notare come ST' e $LCPIT$ presentino la stessa struttura topologica. Inoltre, preso in $LCPIT$ il nodo etichettato l-$[i, j]$, con annesso prefisso ω, si ha tale nodo che esiste sse esiste un nodo su ST tale che la sua *path label* è proprio ω, e viceversa. Si ha una sorta di mapping tra le due strutture.

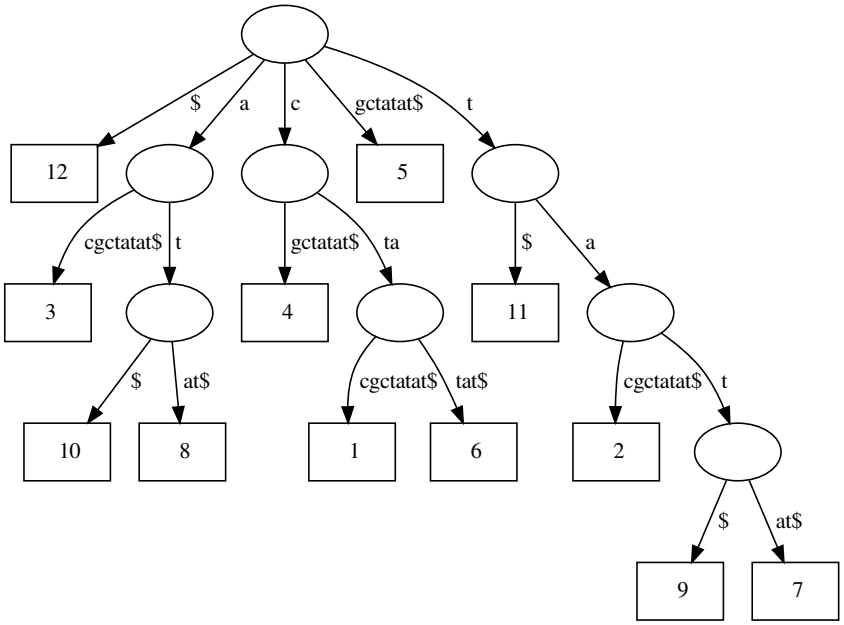
2.2.1 Esempio

Vediamo quindi un esempio pratico di quanto detto.

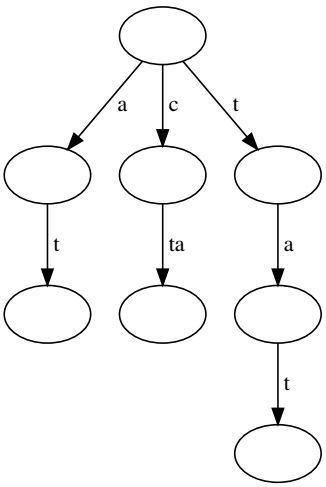
Esempio 5. *Considero in input il testo T ($ esteso):*

$$T = ctacgctatat\$$$

Vediamo quindi una rappresentazione del suffix tree di T:



Come detto visualizziamo anche ST', ovvero ST senza foglie e rispettivi archi entranti:



Passiamo quindi a studiare la costruzione dell'lcp-intervals tree.
Si ha (con il suffix array inverso riportato per completezza in quanto non usato in questo esempio):

	SA	**SA^{-1}**	**LCP**	**suffisso**
1	12	6	-1	$
2	3	10	0	acgctatat$
3	10	2	1	at$
4	8	5	2	atat$
5	4	8	0	cgctatat$
6	1	7	1	ctacgctatat$
7	6	12	3	ctatat$
8	5	4	0	gctatat$
9	11	11	0	t$
10	2	3	1	tacgctatat$
11	9	9	2	tat$
12	7	1	3	tatat$

Procediamo quindi con la costruzione dell'lcp-interval tree.
Si parte dalla radice, rappresentata dall'lcp-interval, con rispettivo valore l:

$$0\text{-}[1, 12]$$

che è quindi relativo alla stringa vuota ε. Possiamo quindi calcolare i relativi l-indices, ovvero:

$$l_0\text{-}indices = \{2, 5, 8, 9\}$$

Possiamo quindi dire che i child sono:

$$[2, 4], [5, 7], [8, 8], [9, 12]$$

Sappiamo inoltre che gli intervalli di ampiezza 1 non vengono considerati in fase di ricostruzione dell'lcp-interval tree in quanto sono facilmente individuabile in fase di percorrimento dell'lcp-interval tree anche senza averne espclito riferimento in memoria (inoltre si segnala che per questo motivo si ottiene una topologia analoga ad un suffix tree privo di foglie). Si hanno quindi i seguenti child, per i quali viene indicato anche il loro valore l, calcolato, dato il rispettivo intervallo $[b, e]$, tramite $RMQ(LCP, b + 1, e)$:

$$1\text{-}[2, 4], 1\text{-}[5, 7], 1\text{-}[9, 12]$$

Abbiamo quindi ottenuto i tre child del nodo root, di cui è possibile anche indicare ω, come definito precedentemente:

- 1-$[2, 4] \implies \omega = a$
- 1-$[5, 7] \implies \omega = c$
- 1-$[9, 12] \implies \omega = t$

Passiamo ad analizzare il primo child, 1-$[2, 4]$. *Calcoliamo i relativi l-indices, ovvero:*

$$l_1\text{-}indices = \{3\}$$

Avendo quindi, specificando anche il valore l e la relativa ω, solo:

$$2\text{-}[3, 4] \implies \omega = at$$

Essendo l'intervallo di ampiezza due non è necesario proseguire oltre.
Proseguiamo con il secondo child, 1-$[5, 7]$. *Calcoliamo i relativi l-indices, ovvero:*

$$l_2\text{-}indices = \{6\}$$

Avendo quindi, specificando anche il valore l e la relativa ω, solo:

$$3\text{-}[6, 7] \implies \omega = cta$$

Essendo l'intervallo di ampiezza due, anche in questo caso, non è necesario proseguire oltre.
Passiamo infine all'ultimo child, 1-$[9, 12]$. *Calcoliamo i relativi l-indices, ovvero:*

$$l_3\text{-}indices = \{10\}$$

Avendo quindi, specificando anche il valore l e la relativa ω, solo:

$$2\text{-}[10, 12] \implies \omega = ta$$

In questo caso ho un intervallo di ampiezza tre quindi si prosegue lo studio. Calcoliamo gli l-indices di 2-$[10, 12]$, *ovvero:*

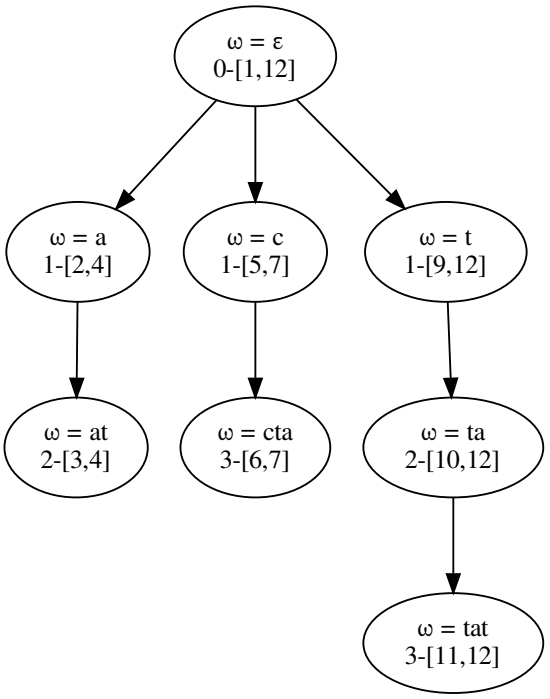
$$l_4\text{-}indices = \{11\}$$

Avendo quindi, specificando anche il valore l e la relativa ω, solo:

$$3\text{-}[11, 12] \implies \omega = tat$$

Essendo l'intervallo di ampiezza due non è necesario proseguire oltre.
Si segnala, inoltre, come valga quanto detto agli lcp-interval embedded, avendo che l'lcp-interval parent "racchiude", anche se non strettamente come visto nella definizione teorica, l'lcp-interval child e che valga che tutte le ω dei child contengano come prefisso l'ω del rispettivo parent.

Si ottiene quindi, nel complesso, la seguente rappresentazione dell'lcp-interval tree, a conclusione di quanto detto:



Dando una prova, anche "visuale", di quanto detto anche in merito alla relazione tra suffix tree e lcp-interval tree, notando il rapporto tra le path label del suffix tree e i ω dell'lcp-intervals tree, oltre alla medesima topologiacondivisa da quest'ultimo e dal suffix tree privo di foglie e archi entranti in esse. È interessante anche notare come dai nodi dell'lcp-intervals tree, tramite appunto gli lcp-intervals, si possibile risalire alla rappresentazione completa del suffix tree tramite il suffix array indicato in tabella.
Si prenda ad esempio il nodo, il più profondo nel a destra, nell'lcp-intervals tree etichettato con:

$$\omega = tat$$

$$3\text{-}[11, 12]$$

conferma di quanto detto sul legame tra ω e path label del suffix tree, che in quello stesso nodo arriva, dalla radice, con archi etichettati con t, a e t.

Inoltre, agli indici 11 e 12, abbiamo:

$$SA[11] = 9 \text{ } e \text{ } SA[12] = 7$$

che sono gli indici dei suffissi specificati nelle foglie "tagliate" sotto il corrispettivo nodo nel suffix tree, ad ulteriore conferma dello stretto rapporto tra le due strutture.
Con un ragionamento è possibile ricostruire anche le foglie collegate a nodi interni, sfruttando gli intervalli di ampiezza unitaria e i rispettivi valori in SA, sfruttando eventualmente i "buchi" lasciati dagli intervalli children di un nodo, come nel nostro caso per $[8, 8]$*, che è un "buco" tra* $[5, 7]$ *e* $[9, 12]$ *tra i children della radice. Inoltre, qualora si abbia un "buco" di ampiezza k maggiore stretta di 1, avendo per costruzione che solo gli intervalli di ampiezza 1 non vengono considerati, si può dire di avere k foglie. Ad esempio immaginiamo di modificare l'esempio visto sopra e ottenere come children effettivi della radice nell'lcp-interval tree i seguenti intervalli:*

$$1\text{-}[2, 4], 1\text{-}[5, 6], 1\text{-}[9, 12]$$

avendo un "buco" tra $[5, 6]$ *e* $[9, 12]$ *di ampiezza 2. In tal caso potremmo dire di avere due foglie, relative gli intervalli* $[7, 7]$ *e* $[8, 8]$*, con i relativi indici di suffisso ricavabili dal suffix array.*
Si completa così l'analisi del "mapping" tra le due strutture.

2.3 Domanda 3

Si può dimostrare che un algoritmo ottimo elimina fino a due breakpoint ogni passaggio. Indicando con $b(\pi)$ il numero dei breakpoint presenti nella permutazione π in input, possiamo quindi dire che il numero minimo di inversioni, n_{inv}, necessarie a riordinare π, ovvero atte a ottenere la permutazione identica a partire dalla permutazione π, risponde a:

$$n_{inv} \geq \left\lceil \frac{b(\pi)}{2} \right\rceil$$

Avendo però che ogni step elimina al più due breakpoint (e non, purtroppo, esattamente due) potrei avere comunque bisogno di più di $\left\lceil \frac{b(\pi)}{2} \right\rceil$ inversioni, anche se questo è il limite inferiore sotto al quale sicuramente non si può scendere.
La specifica d'uso dell'intero superiore, tramite $\lceil \cdot \rceil$, è data dal fatto che $\frac{b(\pi)}{2}$, qualora $b(\pi)$ fosse dispari, potrebbe avere valore decimale. Per quanto detto, avendo l'eliminazione di al più due breakpoint a inversione, in presenza

di un numero dispari di breakpoint, dovrei aggiungere almeno una singola inversione aggiuntiva, che rappresento nella disequazione tramite l'intero superiore. Infatti, bisognerebbe fare fare almeno $\frac{b(\pi)-1}{2}$ inversioni per i primi $b(\pi)-1$ breakpoint, che sono in numero pari avendo assunto $b(\pi)$ dispari, più l'inversione per il breakpoint mancante, da cui l'uso dell'intero superiore.

2.3.1 Esempio

Vediamo quindi un esempio pratico di quanto detto.

Esempio 6. *Prendiamo una permutazione* π*:*

$$\pi = [\ 6\ 1\ 5\ 4\ 3\ 2\ 7\ ]$$

Estendiamo π *con* 0 *all'inizio e con*

$$\max_{\pi} +1 = 7 + 1 = 8$$

alla fine, ottenendo:

$$\pi = [\ 0\ 6\ 1\ 5\ 4\ 3\ 2\ 7\ 8\ ]$$

Si indicano con " |" i breakpoint:

$$\pi = [\ 0\ |\ 6\ |\ 1\ |\ 5\ 4\ 3\ 2\ |\ 7\ 8\ ]$$

Avendo quindi:

$$b(\pi) = 4$$

Aspettandoci quindi un minimo numero di inversioni necessarie a ottenere la permutazione identica dalla permutazione π *del tipo:*

$$n_{inv} \geq \left\lceil \frac{4}{2} \right\rceil \geq 2$$

Si procede quindi con le inversioni.
Effettuiamo in primis l'inversione tra 6 e 1, rispettivamente di indice 1 e 2 (indicizzando da zero), che rimuove due breakpoint, ottenendo, tramite $r(1,2)$*:*

$$\pi = [\ 0\ 1\ |\ 6\ 5\ 4\ 3\ 2\ |\ 7\ 8\ ]$$

Infine si procede con $r(2,6)$*, rimuovendo anche i due restanti breakpoint, ottenendo la permutazione identica:*

$$\pi = [\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ ]$$

Come si è visto sono state effettuate 2 inversioni, sapendo, per il limite inferiore, che non avremmo potuto fare di meglio.
D'altro canto è bene notare come sarebbe stato possibile, a differenza dell'esempio mostrato, avere permutazioni, sempre con quattro breakpoint, che avrebbero necessitato di più di due inversioni per ottenere la permutazione identica.