Minimum Substring Partitioning

Yang Li Pegah Kamousi Fangqiu Han Shengqi Yang
Xifeng Yan Subhash Suri

Tartalomjegyzék

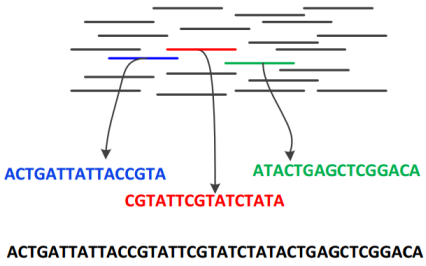
1. Absztrakt

A masszívan párhuzamos DNS szekvenciáló technikák újradefiniálják a génkutatást. Milliárdnyi rövid, alacsony költségű olvasásból állítanak össze teljes genomokat. Sajnos ezen technikáknak hatalmas memóriagényük van, például emlősök esetén. Próbálunk tehát a de Bruijn gráfok építésének memóriaproblémáinak utánajárni, ami sokszor akár több száz Gb-nyi adatot eredményez a memóriában. Egy merevlemez-alapú partícionáló eljárást adunk, ami ugyanezt a feladatot legfeljebb 10 Gb memória felhasználásával oldja meg, komolyabb időveszteség nélkül.
Az algoritmust Minimum Substring Partitioning-nek (MSP-nek) nevezzünk. Az MSP ezeket a rövid olvasásokat több kicsi, diszjunkt partícióra bontja fel úgy, hogy ezen partíciókat be tudjuk tölteni memóriába, majd fel tudjuk dolgozni őket egyenként, hogy később a többivel összekapcsolva megalkossák a de Bruijn gráfot.
Kihasználva a k-hosszú résszavak közötti ismétlődést, az MSP elképesztő tömörítési arányt ér el: a partíciók teljes méretét redukálja $\theta(kn)$ -ről $\theta(n)$-re, ahol n a rövid olvasás hossza, k pedig a k-hosszú részszó hossza. A kísérleti eredmények megmutatták, hogy az eljárás képes nagy méretű adathalmazokon, akár otthoni gépeken dolgozni.

2. Bevezetés

A magas minőségű genom szekvenciálás rendkívül fontos biológiai és egészségügyi problémák megoldásánál. A legnagyobb probléma a különböző fajok teljes genomját összeállítani, ami például bizonyos betegségek tüneteinek megállapításánál kritikus lenne. Sajnos a már létező alacsony költségű eljárások nem képesek a teljes genomot előállítani, így a legnagyobb kihívás most a rövid olvasások összeillesztése úgy, hogy visszaadják a teljes genomot, ami egy *Big Data* probléma. Az ilyen rövid olvasások száma akár elérheti az egy milliárdot is, és egy ilyen olvasás 10 és 100 közötti bázist tartalmaz. Az 1. Ábra példa egy ilyen szekvencia összeállító eljárásra, ahol 3 rövidebb szekvenciát egyesítünk egy hosszabbá.

Ezt az eljárást De novo összeállításnak nevezzünk, az elmúlt évtizedekben ezt rengetegen kutatták. Két megközelítése van: a fedési-elhelyezkedési, illetve a



1. ábra. Szekvenciák összeillesztése

de Bruijn gráfos megközelítés. Előbbi gráfot épít az átfedések alapján, ami már az átfedési-gráf puszta mérete miatt csak kisebb genomok esetén használható. A de Bruijn gráfos megközelítés k-merekre bontja a rövid olvasásokat, majd az átfedések alapján köti össze őket. Nagy mennyiségű olvasás esetén a de Bruijn gráfos megközelítés a legnépszerűbb.
Népszerűsége ellenére ez a megközelítés a memória-igénye miatt problémás. Ebben a cikkben ezt a problémát oldjuk meg lemez-alapú eljárás felhasználásával, ami kevesebb, mint 10 Gb memóriát használ a futási idő romlása nélkül.
A de Bruijn gráfban minden csúcs egy k-mer. Ahhoz, hogy felépítsük ezt a gráfot, meg kell találnunk a szétszórt, de azonos k-mereket különböző olvasásokból. Erre a legegyszerűbb megoldás egy hash-tábla használata. Minden A, C, G és T szimbólumot 2 biten kódolunk. Egy 137 Gb méretű adathalmaz esetén, ahol $k = 59$, 11.8 milliárd különböző k-mer lesz, a fordított komplemenseket is beleszámolva, ami így egy 283 Gb méretű hash-táblát fog építeni, ami túl nagy. Használhatnánk azonban az adatbázisok világában sokat használt merevlemez-alapú partíció-egyesítő megközelítést. Egy S rövid olvasás halmazból két klasszikus módja van a duplikátumok felfedezésének:

1. Vízszintesen partícionáljuk S-t $S_1, S_2..., S_t$ diszjunkt részhalmazokra, mindegyik S_i-re előállítjuk a H_i hash-táblát a k-merekből, rendezzük H-t, lemezre írjuk, végül pedig egyesítjük $H_1, H_2, ..., H_t$-t;

2. Partícionáljunk minden k-mert S-ből $S_1, S_2, ..., S_t$ diszjunkt részhalmazokba az utolsó pár szimbólum alapján, majd minden S_i-re hozzuk létre a H_i hash-táblát, építsük meg a k-mer mapelést és írjuk ki H_i-t lemezre, majd kombináljuk őket.

Mindkét eljárás hatékony memória használat szempontjából, azonban nagyon lassúak. Az első teljesen reménytelen, a második pedig hatalmas mennyiségű k-mert állít elő az első lépésben. Egy 258.7 Gb méretű adathalmazra, $k = 59$-re, a lemezre írt k-merek mérete közel 3 Tb volt, és a duplikátumok megkeresése 30 óráig tartott.
Az eljárásunk a második megoldást veszi megint szemügyre. Sok olyan k-mert állítunk ezzel elő, amik azonos rövid olvasásokból állnak elő és nagy ismétlődések találhatóak közöttük, azonban külön partíciókba kerülnek, ami nagy overheadet okoz. Ebből kifolyólag vezetjük be a Minimum Substring Partitioning-et, ami k-merek helyett darabokra osztja fel a rövid olvasásokat, ahol minden darab k-mereket tartalmaz közös minimális p-hosszú részszóval, ahol $p \leq k$. Ennek a végeredménye gyakorlatilag azonos a k-merek tömörítésével. Bemutatjuk, hogy ez a tömörítés nem okoz szignifikáns számítási overheadet, de akár 10-15-ször kisebb partíciókat eredményezhet. Megmutatjuk, hogy a partíciók számát sikerül $\theta(kn)$-ről $\theta(n)$-re csökkenteni. Ezen felül belátjuk, hogy a legnagyobb partíció mérete exponenciálisan csökken a p-nek megfelelően, ami mutatja, hogy az eljárás nagyon memória-takarékos. Amikor $p = 12$, a memória-használat kevesebb mint 10 GB, minden tesztelt esetben.

3. Előismeretek

1. Definíció (Rövid olvasás). *Egy rövid olvasás egy szó a σ ábécé fölött.*

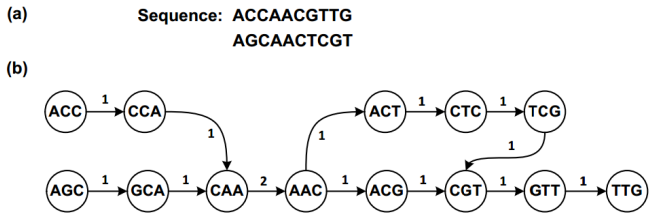
2. Definíció (K-mer). *Egy k-mer egy szó, aminek hossza k.*

3. Definíció (Szomszédos k-merek). *Adott s rövid olvasásra, jelölje $s[i, j]$ a résszóit az s i-edik és j-edik betűje között. s felbontható $m - k + 1$ k-merre, kifejtve $s[1, k], s[2, k + 1], ..., s[m - k + 1, m]$. Az $s[i, k + i - 1], s[i + 1, k + i]$ k-mereket szomszédosnak nevezzük s-ben.*

Egy s rövid olvasásra tekinthetjük a generált k-mereket egy k-hosszú ablaknak, ami végigcsúszik az s fölött. Két α és β k-mer szomszédos, ha α utolsó $k - 1$ résszava a β első $k - 1$ résszava. Egy s_i-ből kinyert k-mert jelölhetjük $s_{i,j}$-nek.

4. Definíció (de Bruijn gráf). *Adott S rövid olvasás halmazra, a $G = \{V, E\}$ de Bruijn gráf úgy áll elő, hogy létrehozunk egy csúcsot minden különböző k-*

merre S-ben, és élet húzunk bármely két csúcs között, ha a hozzájuk tartozó k-merek szomszédosak legalább egy rövid olvasásban.



2. ábra. Egy de Bruijn gráf, $k = 3$ esetén.

3.1. K-mer mapelés

Adott rövid olvasásokból álló adathalmazra, ahhoz, hogy felépíthessük a hozzá tartozó de Bruijn gráfot, mapelni kell az összes duplikált k-mert a különböző olvasásokból ugyanarra a csúcsra. Ha a csúcsokat egész számokkal azonosítjuk, akkor a duplikált k-merekhez is ezt az egészet fogjuk rendelni. Ezt az eljárást k-mer mapelésnek nevezzük. Amint a mapelés megtörtént, a rövid olvasások végigfutásával természetesen előállíthatjuk a de Bruijn gráfot, amiből következik, hogy ezen gráf építése a mapelésből és a csúcsok összekötéséből áll.

3.2. Scatter/Gather

A memória probléma egyik megoldása, hogy az adatot kisebb egységekre partícionáljuk a lemezen, majd így dolgozzuk fel. Ennek megvalósítására két scatter/gather eljárást ismertetünk. Az elsőt vízszintes partícionálásnak (Horizontal Partition, H-Partition) nevezzük.

1. Osszuk fel a rövid olvasásokból álló S adahalmazt diszjunkt, azonos méretű partíciókra úgy, hogy minden partíciót be tudjunk tölteni a memóriába.

2. Minden S_i partícióra, szúrjuk be a k-mereket a H_i hash-táblába. A beszúrási sorrendnek megfelelően rendeljünk hozzá egy sorszámot a különböző k-merekhez 1-től kezdődően. Legyen M_i a lokális k-mer mapelő eljárás S_i-ben.

3. Minden S_i partícióra, írjuk ki az összes $s_{i,j}$ partíciót és a hozzá tartozó indexet. Legyen P_i a kimeneti szekvencia.

4. Fésüljük össze P_i-t úgy, hogy egy globális mapelő eljárást, M-et használunk úgy, hogy az kielégíti a következő megszorításokat: akármely S_j-ből kinyert γ k-merre, legyen S_i a partíció a legkisebb i-vel, ami még tartalmazza γ-t. Ekkor $M(\gamma) = M_i(\gamma)$.

A harmadik lépés mérete $\theta(kn)$, ahol n a rövid olvasások mennyisége, k pedig a k-merek hossza. A negyedik lépés nagyon költséges, mivel egy rendező/összefésülő eljárást kell használnia, hogy felismerje a különböző partíciókból származó duplikált k-mereket.
A legnagyobb probléma ezzel az eljárással, hogy a különböző előfordulásai egy adott k-mernek nem ugyanabba a partícióba kerülnek. Ahhoz, hogy ezt megoldjuk, az egyik megoldás a vödör partícionálás. Ennek a menete a következő:

1. Nyerjük ki az összes k-mert S-ből, és rakjuk őket diszjunkt $S_1, S_2, ..., S_t$ partíciókba $H(s_{i,j})modt$-nek megfelelően.

2. Minden S_i partícióra, szúrjuk be a k-mereket a H_i hash-táblába és rendeljünk hozzájuk egy folyamatosan növelt egész azonosítót, kezdve $\sum_{j=1}^{i-1} S_j$-től a beszúrás sorrendjének megfelelően, ahol S_j a különböző k-merek száma az S_j partícióban. Legyen M a k-mer mapelő eljárás. Nyilvánvaló, hogy M globális mapelés: minden különböző k-merre egyedi azonosítót ad.

3. Minden S_i partícióra írjuk ki az összes $s_{i,j}$ k-mert (ebben az esetben elég az (i, j) index a teljes szó helyett) a hozzátartozó azonosítóval, (i, j) szerint növekvő sorrendben. Legyen P_i a kimeneti szekvencia.

4. Fésüljük össze az összes P_i-t (i, j) szerinti növekvő sorrendben.

Bár a negyedik lépés itt gyorsabb, mint a vízszintes partícionálás esetén, az összes partíció teljes mérete szintén $\theta(kn)$, ami nagyobb genomok esetén könnyen Tb-os méretű lehet. Most bevezetünk egy új partícionáló eljárást, a Minimális részszó Partícionálást, ami ezt a számot $\theta(n)$-re csökkenti.

4. Minimum Substring Partitioning

A vödör partícionálásnak overheadje igen nagy, mivel a szomszédos k-merek elég valószínű, hogy külön partíciókba kerülnek, kivéve ha $H(s_{i,j})modt = H(s_{i,j+1})modt)$.

Erre a problémára más megközelítést használva adunk megoldást.

5. Definíció (Minimum p-részszó). *Adott s szóra, a p-hosszú részszó r minimum p-részszava (vagy pivot részszava) s-nek, ha minden s'-re s' a p-hosszú részszava s-nek, s.t.-nek és $r \leq s'$-nek ($\leq$ lexikografikus rendezés). s-re ekkor mondhatjuk, hogy r lefedi. A minimális p-részszavát s-nek jelölhetjük $min_p(s)$-sel.*

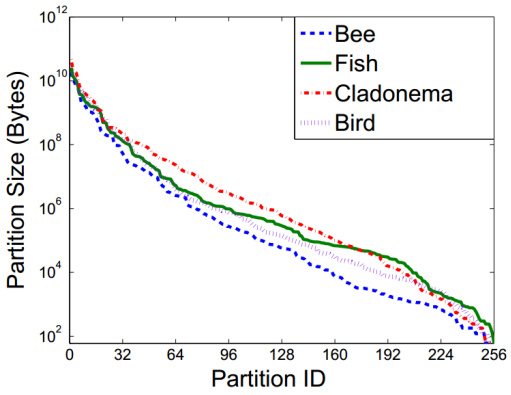
3. ábra. Minimum Substring Partitioning

Mivel két szomszédos k-mer k-1 hosszan fedi le a másikat, az esély, hogy tartalmazzák ugyanazt a minimális p-részszót ($p < k$) potenciálisan igen nagy. Az ábrán láthatunk erre egy példát, ahol az első 5 k-mernek azonos a minimális 4-részszava. Ilyen esetekben ahelyett, hogy generálnánk ezt az 5 k-mert külön-külön, egyszerűen tömöríthetjük az eredeti rövid olvasás használatával, ACTGATTATTAACCGTACAA-val, és kiírhatjuk az AACC minimális részszóhoz tartozó partícióba. Formálisan: adott $s = s_1 s_2 s_3 ... s_m$ rövid olvasásra, ha a szomszédos j k-mer $s[i, i+k-1]$ és $s[i+j-1, i+j+k-2]$ ugyanazzal a minimális p-részszóval rendelkeznek, amit jelöljünk r-rel, akkor kiírhatjuk egyszerűen a $s_i s_{i+1} ... s_{i+j+k-2}$ részszót a $H(r) mod t$ partícióba anélkül, hogy j darab k-merre bontanánk. Ha j nagy, ez a tömörítési stratégia jelentősen csökkenti a partíciók méretét és a futási időt.

6. Definíció (Minimum Substring Partitioning). *Adott egy szó, $s = s_1, s_2, ..., s_m$, $p \leq k \leq m$, a minimális résszó partícionálás ezt az s-t maximális hosszúságú részszavakra bontja, hogy $\{s[i,j] | i + k - 1 \leq j, 1 \leq i, j \leq m\}$, s.t., ekkor minden k-mer $s[i,j]$-ben ugyanazzal a minimális p-részszóval rendelkeznek. $s[i,j]$-t szu-*

per k-mernek nevezzünk.

A minimális partícionálás szerint a nagyobb p-k jó eséllyel fogják megtörni a szekvenciát több szegmensre, különböző minimális p-részszavakkal, ezzel növelve a partíciók összméretét. Másfelől, a kisebb p-k nagyobb partíciókat eredményeznek, amik esetleg nem férnek be memóriába.



4. ábra. Partíciók méretének eloszlása

Az ábrán a partíciók méretének eloszlását láthatjuk, $p = 4$ esetén különböző fajokhoz tartozó adathalmazokra. Van néhány nagy, domináns partíció. A p értéke megszabja a partíciók összméretét és a várható legnagyobb partíciók méretét is. Két technikát alkalmazhatunk, hogy a partíciók asszimetrikus eloszlásával megbirkózzunk. Az első, hogy ha p nem túl kicsi (pl. 10), a legnagyobb partíciók mérete hasonló lesz. A második, a partíciók feltekerése (*Wrapped Partition*), később kerül bemutatásra.
A következőkben egy random string modell segítségével fogjuk megvizsgálni az MSP-t. Kimondásra kerül több tétel, köztük az is, ami kimondja, hogy a partíciók várható mérete $\theta(n)$, ami szignifikánsan jobb $\theta(kn)$-nél. Ezen kívül megadjuk az alsó és a felső határát az MSP által előállított partíciók méretének, ami exponenciálisan fog csökkeni p-nek megfelelően, mutatva, hogy az MSP nagyon memória-hatékony.

4.1. Partíciók száma

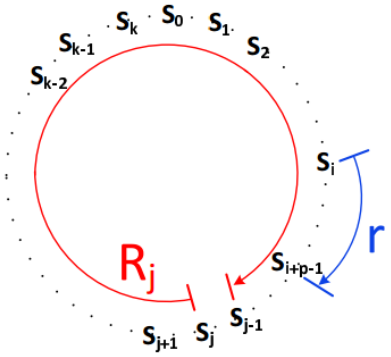
A következő tételek bizonyítás nélkül szerepelnek. Ezen bizonyítások megtalálhatóak a szerzők eredeti cikkében.

1. Tétel. *Az össz partíció méret* $\theta(\frac{lk}{m}n + n)$.

2. Tétel. *Legyen* $l(m, k, p)$ *a törések átlagos száma egy MSP-ben. Egy random string modellben* $l(m, k, p) \propto (m - k)$.

3. Tétel. *Egy random string modellben* $P_1(kp, p) \leq \frac{p+1}{k+1}$.

1. Következmény. *Egy random string modellben a partíciók összmérete* $O(pn)$.



5. ábra. A 3. tétel illusztrációja.

4. Tétel. *Egy random string modellben adott* $a > 0$ *egészre* $P_1(k + a, p + a) \leq 2P_1(k, p) + \frac{p+1}{4P}$.

5. Tétel. *Egy random string modellben azon különböző k-merek százalékára, amiket lefed egy p-részszó, a felső korlát* $\frac{3k}{4p+1}, p \geq 2$.

5. Fordított komplemensek

A DNS szekvenciákat két irányból olvashatjuk, előrefelé, és visszafelé úgy, hogy az adott szimbólumot a Watson-Crick komplemensére cseréljük. Ezeket fordított komplemenseknek hívjuk és ekvivalensnek tekinti őket a bioinformatika. A

legtöbb szekvenciáló eljárás mindkét iránnyal dolgozik. Az összerakási folyamat során minden szekvenciát kétszer kell olvasni, a két különböző irányban.
A fordított komplemensek nem okoznak problémát a vödör partícionálás esetében: ha egy k-mert beolvasunk a memóriába abból már könnyen elő tudjuk állítani a fordított komplemensét. Ugyanez azonban trükkös MSP-vel, aminek a célja az egymást követő k-merek tömörítése, amennyiben tartalmazzák ugyanazt a minimális p-részszót. Sajnos a fordíott komplemenseik nem feltétlen tartalmazzák ugyanazt a minimális p-részszót, ami arra kényszerít minket, hogy explicit módon előállítsuk a fordított komplemenseket minden rövid olvasásra, ami értelemszerűen duplázni fogja az I/O költségeket.

7. Definíció (Minimális részszó fordított komplemensekkel). *Adott s szó, egy p-hosszú t részszava s-nek a minimális p-részszava s-nek, ha* $\forall s', s'$ *egy p-hosszú részszava s-nek, vagy* s' *fordított komplemens, hogy s.t.,* $t \leq s'$ *(ahol* $\leq$ *lexikografikus rendezés).*

Ez a definíció újradefiniálja a minimális részszót úgy, hogy számol az adott k-mer fordított komplemensével. Ezzel az új definícióval nem kell sem explicit módon kiírnunk a fordított komplemenseket, sem megváltoztatni a MSP-t előállító eljárást. A következőkben ezzel a problémával nem számolunk.

6. Algoritmusok

Most kitérünk arra, hogy hogyan kell egy de Bruijn gráfot felépíteni. Ennek három lépése van: partícionálás, mapelés és összefésülés. Minden lépés egy lemezre kiírt adathalmazzal indul, és egy módosíott, szintén lemezre írt adathalmaz előállításával ér véget. Az első lépés bemenete nyilván egy rövid olvasásokból álló adathalmaz, az utolsó lépés kimenete pedig azonosítókból áll, amik mapelve vannak a megfelelő k-merekhez.

6.1. Partícionálás

Az első lépés a rövid olvasások partícionálása MSP használatával. Az eljárás naív változata így néz ki:

1. Adott s rövid olvasásra, csúsztassunk végig egy k hosszú ablakot az s-en, hogy előállítsuk a k-mereket.

2. Minden k-merre állítsuk elő a minimális p-részszót.

3. Találjuk meg a szuper k-mereket s-ben.

Ez a megoldás azonban nem használja ki az átfedéseket a szomszédos k-merek között. Amikor a k hosszú ablakot végigcsúsztatjuk s-en, nyilvántarthatunk egy elsőbbségi sort a p-részszavakról, amik az ablakban vannak. Minden jobbra csúsztatáskor berakjuk a jelenlegi ablak utolsó p-részszavát ebbe a sorba, illetve kivesszük belőle az előző ablakból származó első részszót. Mivel az ablakban levő p-részszavak száma $k-p+1$ and $m-p+1$ p-részszó van s-ben, az összehasonlítások száma $O((m-p+1)\log(k-p+1)) = O(m \log k)$.
Bár a prioritásos sor elméletben jó, az overhead amit bevezet igen nagy is tud lenni. Ezt a problémát a Simple Scan algoritmussal kerüljük ki (1. Algoritmus). Ez az algoritmus végignézi az ablakot az első szimbólumból, hogy megtalálja a minimum p-részszót, nevezzük ezt $min_p s$-nek, illetve az ehhez tartozó kezdőpozíciót, ezt pedig nevezzük *min.pos*-nak. Ezután csúsztat jobbra, szimbólumonként egyesével, egészen a rövid olvasás végéig. Minden csúsztatás után ellenőrzi, hogy *min.pos* még benne van-e az ablakban, ha nem, akkor megkeresi az új $min_p s$-t és *min.pos*-t. Egyébként leellenőrzi, hogy az utolsó p-részszó a jelenlegi ablakból kisebb-e, mint a jelenlegi $min_p s$. Ha igen, ez a p-részszó lesz az új $min_p s$, illetve értelemszerűen a *min.pos*-t is átállítjuk. Mivel a szomszédos k-mereknek esélyesen ugyanaz a minimum p-részszavuk, ezért ezt az újrakeresést viszonylag ritkán kell végrehajtani. Bár a legrosszabb esetben a p-részszavak összehasonlításának száma $O(mk)$, a következő tétel alapján láthatjuk, hogy ez gyakorlatban valószínűleg kevesebb.

6. Tétel. *Adott m-hosszú szóra, tegyük fel, hogy az MSP s-t $l+1$ részszóra bontja. Az 1. algoritmusnak ekkor legfeljebb $\theta(m+lk)$ p-részszó összehasonlításra van szüksége.*

Algoritmus 1. SimpleScan

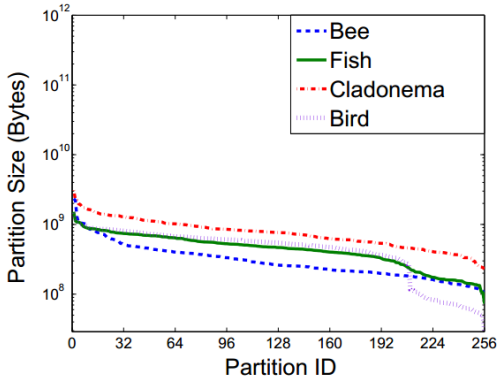
```
Input: String s[1..m], integer k; p.
min_s := the minimum p-substring of s[1; k]
min_pos := the start position of min_s in s
for all i from 2 to m-k+1 do
    if i > min pos then
        min_s := the minimum p-substring of s[i; i+k-1]
        min_pos := the start position of min_s in s
    else
        if the last p-substring of s[i; i+k-1] < min_s then
            min_s := the last p-substring of s[i; i+k-1]
            min_pos := the start position of min_s in s
        end if
    end if
```

8. Definíció (Feltekert partíciók). *Adott $\{s_1\}$ szó halmazra, H hash-függvényre, legyen t a partíciók száma. Bármely k-merre $s_{i,j}$-ben a partíció feltekerése $s_{i,j}$ minimális részszavát $H(min_p(s_{i,j}))$ mod t partícióhoz rendeli.*

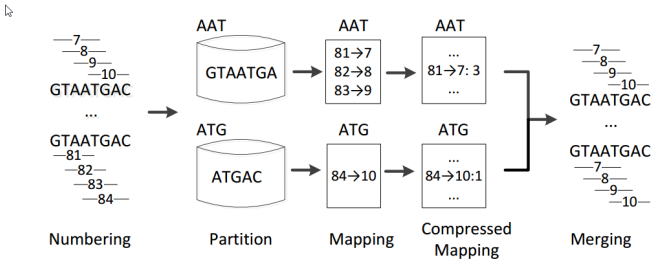
Mivel minden p-részszó egy partícióhoz tartozik, ezért a partíciók teljes száma 4^p. Ahogy p nő, ez a szám exponenciálisan megemelkedik. Ahhoz, hogy ezt elkerüljük, bevezethetünk egy hashelő függvényt, ami feltekeri a partíciókat egy felhasználó által megadott számig. Ebben az esetben minden partíciót, amit egy p-részszóból állítottunk elő, random módon beletesszük egy feltekert partícióba. Ennek egyik eredménye az lesz, hogy a partíciók mérete közötti szórás csökken. Az ábra mutatja a partíciók méretét $p = 10$-re, ha a feltekert partíciók mérete 256-ra van állítva. A partíciók száma megegyezik a $p = 4$ esetten, feltekerés nélkül, de a partíciók méretei kisebb eltéréseket mutatnak.

6.2. Mapelés

Ebben a lépésben minden különböző k-merhez egy egyedi azonosítót rendelünk, ami meg fog egyezni a de Bruijn gráfban levő csúcs azonosítójával. Bármikor, amikor olyan k-mert találunk, ami addig nem volt a táblában, új azonosítót rendelünk hozzá. A kezdő azonosító adott táblában mindig az előző tábla maximális azonosítójánál egyel nagyobb, amiből következik, hogy a partíciók és a hozzájuk tartozó hash-táblák diszjunktak. Minden feldolgozott partíció után, létrehozzuk a kimeneti fájlt (vagy $id\ file$-t), és beleírjuk a hash-táblát.



6. ábra. Partíciók feltekerése



7. ábra. ID-k cseréje és mergelése

Az összefésülő lépésben újra végignézzük a rövid olvasásokat, hogy be tudjuk húzni a szomszédos k-merek közötti éleket a gráfban. Minden szomszédos k-mer párra meg kell keresnünk a hozzájuk tartozó azonosítókat az id fájlokból. Mivel ezen fájlok mérete akkora, mint a partíciók mérete, ami komoly I/O overheadet okoz. Ahhoz, hogy ezt megoldjuk, létrehozunk egy azonosító lecserélő stratégiát, ami az ábrán látható. A partícionáló lépés során minden k-merhez előre hozzárendelünk egy egészet 1-től kezdve. Ugyanazok a k-merek különböző rövid olvasásokból külön azonosítót kapnak. Minden partícióra, ha k-mert látunk, először megnézzünk a hash-táblában, hogy létezik-e már. Ha létezik, akkor egy $< kmer, id >$ rekord helyett egy felcserélő rekordot írunk az azonosító felcserélő fájlba, $< current\_id, first\_id >$ alakban, jelezve, hogy a k-mer duplikátum, ezért az előre hozzárendelt azonosítót, a current_id-t le kell cserélnünk az első találati azonosítóra, a first_id-re. Szintén az ábrán látható erre egy példa.

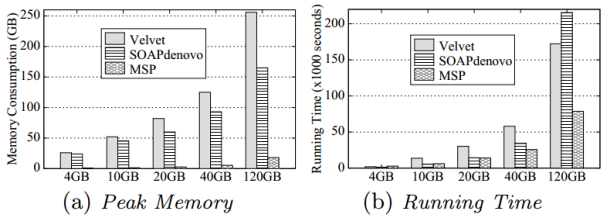
6.3. Összefésülés

Miután előállítottuk az azonosító felcserélő fájlokat, az utolsó lépés az összefésülés. Ebben a lépésben egyszerűen összefésüljük az összes felcserélő fájlt, hogy előállítsuk az azonosítók olyan szekvenciáját, ami már az eredeti rövid olvasásokra mutat, az eredeti sorrendben. Nyissuk meg az összes felcserélő fájlt. Ekkor nyilván mindegyik fájl header az első felcserélő rekordra mutat, amik természetes módon rendezve vannak a first_id szerint növekvő sorrendben, így megkereshetjük a legkisebb lecserélendő azonosítót a file header-ökben. Felsoroljuk az azonosítókat 1-től kezdve, és lecserélünk bármely olyan azonosítót, amire van felcserélő rekord, majd a következő felcserélő rekordra lépünk, és ezt ismételgetjük. Ha ezzel végeztünk, egy szekvenciát kapunk, ahol az azonosítók a k-merekhez kapcsolhatók rövid olvasásokból, azonos sorrendben. Ez egyébként egy lemez-alapú de Bruijn gráfot fog adni, amit szétoszthatunk gépek között, végigfuthatjuk, vagy akár tömörítve memóriába olvashatjuk.

7. Kísérletek, eredmények

7.1. Hatékonyság

Elsőként tekintsük a de Bruijn gráfok építésének sebességét. Az MSP-t a SOAPdenovo és a Velvet algoritmusokkal fogjuk összehasonlítani. Könnyen látható, hogy mind memóriaigény, mind sebesség szempontjából az MSP szignifikánssan gyorsabb a másik két algoritmusnál.



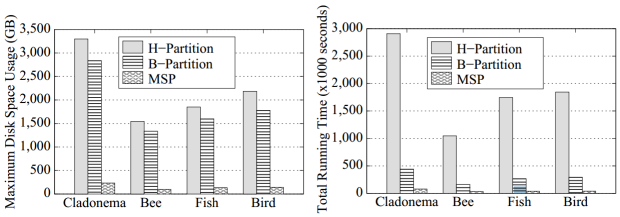
(a) *Peak Memory* (b) *Running Time*

8. ábra. Velvet, SOAPdenovo és MSP.

Nézzük most a partícionálási eredményeket. Itt az MSP-t a H-partícionálással és a B-partícionálással hasonlítjuk össze. Mindegyik eljárás esetében 1000 partíciót hozunk létre, a k-merek hosszát pedig 59-re állítjuk. Mindhárom algoritmus hasonló mennyiségű memóriát használt, a lemezhasználat viszont, ahogy látható,

egyértelműen az MSP-nek kedvez.

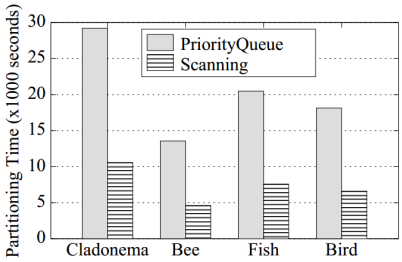
Végül tekintsük a pásztázó eljárást (1. algoritmus). Ezt az algoritmust az



(a) *Max Disk Space Usage* (b) *Running Time*

9. ábra. H-Partícionálás, B-Partícionálás és MSP.

elsőbbségi soros változattal vetjük össze. Látható, hogy az MSP ismét sokkal gyorsabb az alternatívánál.

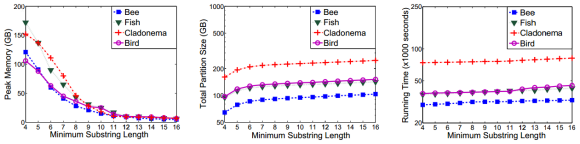


10. ábra. Pásztázó eljárások.

7.2. Skálázhatóság

Most hasonlítsuk össze az MSP skálázhatóságát a Velvet, illetve SOAPdenovo algoritmusokkal. A cladonema adathalmazt fogjuk használni, az MSP-t 1000 partícióra és $p = 10$-re állítva.

Mindhárom algoritmus lineárisan fog nőni mind memóriaigény, mind futási idő tekintetében. Itt is az MSP teljesített a legjobban.



(a) *Peak Memory* (b) *Total Partition Size* (c) *Running Time*

11. ábra. Velvet, SOAPdenovo és MSP skálázhatósága.

8. Összegzés

Bemutattuk a Minimum Substring Partitioning eljárást, ami a duplikált k-merek hatékonyabb összefésülését hivatott megvalósítani, kihasználva, hogy a k-merek között komoly méretű átfedések lehetnek. Láttuk, hogy az MSP nem csak gazdaságosabban bánik a memóriával, de ezen túl még gyorsabb is a jelenleg legfejlettebb algoritmusoknál. Valós DNS-szekvenciákon való kísérletek kimutatták, hogy valós alkalmazások esetén is kiválóan megállja a helyét az algoritmus.