Bioinformatikai algoritmusok

Készítette: Németh Andrea
Nappali tagozat
Programtervező matematikus szak

Témavezető: Dr. Iványi Antal Miklós

2009. június 6.

Tartalomjegyzék

1. Bevezetés **2**

2. Alap definíciók, összefüggések **4**

3. Algoritmusok bonyolultsága **7**

4. Mintaillesztés **9**

4.1. Egyszerű mintaillesztés (Brute–Force algoritmus) 10

4.2. Boyer–Moore algoritmus 12

4.3. Gyors Boyer–Moore algoritmus 15

4.4. Apostolico–Giancarlo algoritmus 17

4.5. Szimulációs eredmények 19

4.5.1. Az algoritmusok implementációja 23

4.6. Szuffix-fák 27

4.7. Mintaillesztés kompakt szuffix-fa segítségével 31

4.8. Mintaillesztés párhuzamosításának lehetőségei 33

4.8.1. Szimulációs eredmények 34

5. Szekvenciaillesztés (Egyszeres) **36**

5.1. Globális illesztés (Needleman–Wunsch algoritmus) 39

5.2. Lokális és szemiglobális illesztés (Smith–Waterman algoritmus) . . . 43

5.3. Büntetőfüggvények 46

5.4. Szubsztituciós mátrixok 47

5.4.1. PAM mátrix 48

5.4.2. BLOSUM mátrix 50

5.5. Heurisztikus keresési módszerek 51

5.5.1. FASTA heurisztika 51

5.5.2. BLAST heurisztika 54

6. Többszörös szekvenciaillesztés **56**

6.1. A többszörös illesztés pontos előállítása 59

7. Összefoglalás **62**

1. Bevezetés

A bioinformatika biológiai problémák informatikai vizsgálatával foglalkozó, interdiszciplináris tudományág, a biológia és az informatika határterülete. A nukleinsavak az élő szervezetek egyik legfontosabb alkotórészei. Ezekben tárolódnak ugyanis az öröklődéshez, és a fehérjeszintézishez szükséges információk. A DNS szál a bináris adatokhoz hasonló formában négy bázissal (nukleotiddal) kódolja az információt, ezek az adenin, timin, citozin, és a guanin, röviden A, T, C, és G. Mivel ezek a bázisok 0.35 nanométerenként helyezkednek el a molekulában, az adatsűrűség körülbelül 20 TB lehet négyzetcentiméterenként. Ezen szekvenciák vizsgálata során sok olyan probléma merült fel, mely könnyen visszavezethető matematikai feladatokra. Ez lehet az egyik oka annak, hogy ez egy nagyon fontos kutatási terület napjainkban.

Watson és Crick több, mint 50 éve tett felfedezése, miszerint a DNS kettős hélix szerkezetű új korszakot nyitott a molekuláris biológia történetében. Azóta a biológiai struktúrákkal és eljárásokkal kapcsolatos ismereteink hatalmas léptekben fejlődtek és egyre érdekesebbé váltak a kutatók számára. Ezt bizonyítja többek között az 1990-es években útjára indított Humán Genom Projekt is, melynek célja őseink génjeinek felismerése, szűrése és géntérképük előállítása volt. A tudósok 2000-ben jelentették be, hogy sikeresen feltárták a teljes emberi genomot egészen a nukleotidok szintjéig és azonosították a benne található összes gént. A teljes humán genom bázissorrendjét nagyjából 2001-re határozták meg. Ez mintegy 3 gigabázisnyi ($3\ \text{Gb} = 3 \cdot 10^9\ \text{bp}$) betűt és körülbelül 30 ezer gént jelent. A 2006 májusában befejeződött program eredményeként megszületett az első klónozott bárány, aki a Dolly nevet kapta.

Az évek során és napjainkban is zajló projektek hihetetlen mennyiségű adatot termelnek, melyeket rendezni kell ahhoz, hogy hasznossá váljanak az olyan területű kutatások számára, mint az orvosi genetika, farmakológia, immunológia, belgyógyászat, vagy a biotechnológia. A számítástechnika vezető szerepet játszik a bioinformatika több tudományágat is érintő kutatási területeiben. Internet nélkül szinte megoldhatatlan feladat lenne az évek során felhalmozódott adatok tárolása és elérése.

Az egyik legismertebb probléma a DNS szekvenálás, melynek célja azon nukleotid sorozatok meghatározása, melyek alapvető építőkövei az emberi DNS-nek. Ezek szerepe nagyon fontos, hiszen az öröklődés során átadott információkat tárolják.

Az informatika szemszögéből nézve a nukleotid bázisokat egy betűkből álló sorozattal lehet a leghatékonyabban reprezentálni. A szekvenálást végző eljárások már az 1980-as évek óta ismertek, viszont az általuk feltérképezhető DNS molekulák hossza korlátos. A jelenlegi algoritmusok körülbelül 1000 elemből álló sorozat vizsgálatát teszik lehetővé. Ezzel szemben a kutatások során több százezres nagyságrendű láncok tulajdonságairól szeretnénk információt gyűjteni. Hogyan tudjuk mégis elérni céljainkat? Az egyik lehetőség, hogy másolatokat készítünk a tanulmányozni kívánt molekuláról, majd ezeket véletlenszerűen darabokra bontjuk. Nagy valószínűséggel a felosztás során kapott részek átfedik egymást, de elméletileg elég rövidek ahhoz, hogy közvetlenül szekvenálni tudjuk őket az általunk már ismert eljárásokkal. Az egymástól függetlenül elvégzett műveletek során több olyan darabot is kiszűrhetünk, melyről tudjuk, hogy része egy másiknak, vagy átfedi azt. Sajnos arra vonatkozóan, hogy a teljes DNS szekvenciát az egységek mely sorrendjével nyerhetjük vissza nincsenek információink, mivel az eljárás során a sorrendre vonatkozó adatok elvesznek. Emiatt újra kell rendezni szórészletek ezreit. Ez tipikusan olyan feladat, mely elvégzéséhez az informatikai eszközök elengedhetetlenek.

Ezen probléma orvosolható a legkisebb olyan sztring megtalálásával, mely tartalmazza az összes többit. Sajnos ez a megoldás nem olyan egyszerű, mint amilyen egyszerűen hangzik, ugyanis a töredékek lehetséges sorrendjének számos variációja létezik. Ráadásul ha meg is találnánk a megfelelő szót még mindig nyitva marad a kérdés, hogy az valóban megegyezik-e a kutatott DNS szekvenciával, vagy további kereséssel találhatunk-e pontosabb, finomabb megoldást.

A molekuláris biológia által felvetett kutatási területek sajnos nincsenek teljesen formalizálva. Ezzel szemben, egyrészt konkrét elképzelések adottak az eredményekkel kapcsolatban, másrészt viszont csak bizonytalan feltételezéseink vannak arról, hogy hogyan lehet a kitűzött célokat és eredményeket elérni. A különböző biotechnológiai alkalmazások eltérő szemléletet igényelnek. Ezen szemléletek és eljárások részletes megismerése és megértése a legtöbb esetben csak biológusok számára lehetséges. Ennek ellenére az alapelvek és összefüggések ismerete nagyon hasznos az ezen a területen kutató informatikusok számára. Csak így lehetséges a már létező, alapvető informatikai algoritmusok módosítása és hatékonyabbá tétele. Az egyik legnagyobb gondot az okozza, hogy minden biológiai adat természeténél fogva pontatlan. Ez azt vonja maga után, hogy a kísérleti eljárások esetében a hiba valószínűsége mindig nagy, tehát figyelembe kell venni őket a további műveletek

elvégzésekor.
Másrészt minden informatikai eljárásból eredő megoldási javaslat valójában csak egy tudományos feltevés. Csak további kutatások igazolhatják, hogy a hipotézisek valóban releváns biológiai megoldást nyújtanak. A probléma formális specifikálása elengedhetetlen feltétele az algoritmusok kutatás során való felhasználásának. Ez különösen fontos már a célok meghatározásánál és az ismert adatok vizsgálatánál is. Csak modellek segítségével érhetjük el a kívánt eredményeket. Természetesen egy probléma megoldására több modell is létezhet, így lényeges ezek kiértékelése aszerint, hogy mennyire tudják szemléltetni a fontos jellemzőket és hatásokat.
Például a DNS molekula szerkezetére vonatkozó kutatások során is több lehetőség vetődött fel, ezek közül a ma már szinte mindenki által ismert kettős hélix szerkezet volt a legszemléletesebb, mely betűk sorozataként ábrázolta a molekulát. A legtöbb esetben ez a modell ésszerű és segítségével több lehetőség nyílik informatikai eljárások bevonására a további kísérletek során, viszont a térbeli tulajdonságok vizsgálatához ez a modell már nem megfelelő.
Miután megfelelően formalizáltuk a problémát a megoldás következő lépéseként irányelvként használva a leszögezett elveket és szabályokat algoritmikus szemszögből vizsgáljuk a kérdést. Ennek eredményeként születnek meg a napjainkban is használt metódusok.

2. Alap definíciók, összefüggések

Az algoritmusok tárgyalása előtt elengedhetetlen a szükséges definíciók és szabályok ismertetése. Egyes biológiai struktúrákat legegyszerűbben sztringekkel lehet reprezentálni.

1. **Sztring**
   Σ *ábécé* egy véges, nem üres halmaz, melynek elemeit karaktereknek, betűknek, vagy szimbólumoknak szokás nevezni.
   Egy Σ (véges) ábécé feletti *szó* Σ elemeiből képzett véges sorozat.
   Egy s *szó hossza* megegyezik az azt alkotó betűk számával. Jele: $|s|$.
   Minden x szimbólum esetén meghatározhatjuk annak *előfordulását* az s szóban. Jele: $|s|_x$.
   A definíció következménye: $|s| = \sum_{x \in \Sigma} |s|_x$.
   Az üres sztring jele: λ, hossza: 0 .
   A Σ ábécé feletti n hosszú szavak halmazát Σ_n -el jelöljük és

$$\Sigma^* = \bigcup_{i \geq 0} \Sigma^i$$

azaz a Σ véges ábécé feletti összes szó halmaza.
A bioinformatikával kapcsolatos eddigi irodalomban a „sztring" és a „sorozat" szavakat szinonímaként használták.
Megjegyzések:

(a) Legyen s és t szó a Σ ábécé felett. Ekkor

- st-vel jelöljük e két szó *konkatenációját.*
- s a *részszava* t-nek, ha $\exists\ u \wedge v \in \Sigma : t = usv$.
- s a *prefixe* t-nek, ha $\exists\ v \in \Sigma : t = sv$.
- s a *suffixe* t-nek, ha $\exists\ u \in \Sigma : t = us$.
- s a t *valódi részszava*, ha prefixe t-nek és $s \neq t$.
- s a *részsorozata* t-nek, ha s minden betűje ugyanabban a sorrendben megjelenik t-ben is, de nem feltétlenül folytonosan.
- az üres szó prefixe minden szónak.

(b) Annak ellenére, hogy a „sztring" vagy „szó" és a „sorozat" kifejezéseket szinonímaként használjuk, jelentésük néha eltérő lehet. Az alábbi példák ezt is szemléltetik.

- Legyen $\Sigma = \{a, b, c\}$. Ekkor *„abcabc"* egy szó Σ felett, $|abcabc| = 6$ és $|abcabc|_a = |abcabc|_b = |abcabc|_c = 2$.
- Legyen *„abc"* és *„cba"* két szó Σ felett. Ekkor *„abccba"* a konkatenációjuk, továbbá $\lambda abc = abc$ és $cba\lambda = cba$.
- Az *„abc"* szó részszava önmagának, de nem valódi részszó. A *„bc"* viszont már az.
- Az *„aa"* szó részsorozata az *„abca"* -nak, de az *„acb"* már nem az.

(c) Legyen Σ ábécé és $s = s_1 \dots s_n$ egy szó, ahol $s_1, .., s_n \in \Sigma$ betűk. $\forall i, j \in \{1, .., n\} : i < j$: az $s_i \dots s_j$ részszót $s[i, j]$ -vel jelöljük.

(d) Legyen s és t szó Σ ábécé felett. Ha mindkettőnek létezik felbontása a következő tulajdonságokkal:

- $s = xy$,
- $t = yz$,
- $x \neq \lambda$ és $z \neq \lambda$ és

- $|y|$ maximális,

  akkor y-t az s és t *átfedésének* nevezzük. Jele: $Ov(s,t)$.

(e) Ebben az esetben

- xyz-t s és t *egyesítésének* nevezzük. Jelölése: $< s,t >$.
- $x = Pref(s,t)$
- $z = Szuff(s,t)$
- ha $Ov(s,t)$ hossza 0, akkor s és t átfedését üresnek nevezzük és egyesítésük megegyezik konkatenációjukkal.
- Legyen $s =$ *„cccababab”* és $t =$ *„abababddd”* szavak a $\Sigma = \{a,b,c,d\}$ ábécé felett. Ekkor $< s,t > =$ *„cccababababddd”*, $Ov(s,t) =$ *„ababab”*, $Pref(s,t) =$ *„ccc”* és $Szuff(s,t) =$ *„ddd”*.

(f) Adottak Σ_1 és Σ_2 ábécék, és h: $\Sigma_1^* \to \Sigma_2^*$ függvény, mely
homomofizmus $\Leftrightarrow \forall x,y \in \Sigma_1^* : h(x \cdot y) = h(x) \cdot h(y)$.

2. **Gráf**

A legtöbb bioinformatikai algoritmus számára az egyik legfontosabb adatstruktúra a sztring mellett a gráf, sőt a gráfokon belül is jelentős szerepet töltenek be a fák.
$G = (V,E)$ *irányítatlan gráf* alatt egy párt értünk, ahol V a csúcsok véges halmaza és $E \subseteq \{\{x,y\} | x,y \in V \wedge x \neq y\}$ az élek halmaza.
Irányított gráf esetén azon élek számát, melyeknek a csúcs a kezdőpontja a csúcs *kifokának* nevezzük, azon élek számát pedig, melyeknek a végpontja a csúcs *befokának* nevezzük. Minden irányított gráfból könnyen nyerhetünk irányítatlant az irányok elhagyásával.

(a) Két csúcsot *szomszédosnak* nevezünk, ha $x,y \in V : x,y \in E$.

(b) Egy x csúcs *fokán* a rá illeszkedő élek számát értjük és $deg(x)$-el jelöljük. Egy gráf foka megegyezik az összes benne található csúcs fokának a maximumával.

(c) Egy $G = (V,E)$ gráf *teljes*, ha $E = \{\{x,y\} | x,y, \in V \wedge x \neq y\}$, azaz ha minden csúcs között létezik él.

(d) Legyen $G_1 = (V_1,E_1)$ és $G_2 = (V_2,E_2)$ gráf. G_2 *részgráfja* G_1-nek, ha $V_2 \subseteq V_1$ és $E_2 \subseteq E_1$. Legyen $G = (V,E)$ egy gráf és $V' \subseteq V$. Ekkor a V' által generált $G' = (V',E')$ részgráf éleinek halmaza:
$E' = \{\{x,y\} \in E | x,y \in V'\}$.

(e) Egy $G = (V, E)$ gráfon értelmezett *út* a gráf páronként különböző csúcsaiból képzett sorozat, azaz
$P = x_1, x_2, .., x_m$, ahol $x_i \in V \wedge x_i \neq x_j \ \forall i, j \in \{1, .., m\}$ és $\{x_i, x_{i+1}\} \in E \ \forall i \in \{1, .., m-1\}$.
Ha $x_1 = x_m$ az utat *körnek* nevezzük.
Azt a kört, mely egy gráf összes csúcsát tartalmazza *Hamilton–körnek*, amely pedig pontosan egyszer tartalmazza egy gráf összes élét *Euler–körnek* nevezzük.
Két nem szomszédos csúcsot összekötő él a *húr*.
Összefüggő egy gráf, ha bármely két csúcsa összeköthető úttal.

3. **Fa**
A $T = (V, E)$ gráf *fa*, ha összefüggő és körmentes.
Ha $x \in V : deg(x) = 1$, akkor x *levél*, ha $deg(x) \geq 2$ akkor *belső csúcs*. Az általunk vizsgált fák rendelkeznek egy speciális csúccsal, melyet *gyökérnek* nevezünk és a befoka 0.

(a) Legyen $G = (V, E)$ gráf és $T = (V', E')$ fa. Ha $V = V'$ és $E' \subseteq E$, akkor T-t a G gráf *feszítőfájának* nevezzük.

(b) Az *irányított bináris fa* egy olyan irányított fa, melyben minden belső csúcs kifoka maximum 2 lehet. Irányítatlan bináris fa esetén egyik csúcs fokszáma sem lehet nagyobb, mint 3.

3. Algoritmusok bonyolultsága

Sok különböző típusú problémára nyújthatunk megoldást algoritmusok segítségével. Ezek közül négy jelentős kategóriát emelnék ki:

- **Eldöntendő probléma**
Ilyen probléma esetén a metódus megvizsgálja az adott feltételek alapján a bemenő adatokat, és ha azok megfelelnek a feltételeknek *„Igaz”*, ha nem felelnek meg *„Hamis”* értékkel tér vissza. A feladat meghatározásához elegendő megadni a megengedett bemeneteket és azon feltételek halmazát, melyek alapján a tesztelést végre kell hajtani. Ehhez a típushoz tartozik például a Hamilton–kör probléma, melynek a bemenő paramétere egy irányítatlan gráf,

a kimenő paramétere pedig ha létezik benne Hamilton–kör *„Igaz”*, különben *„Hamis”*.

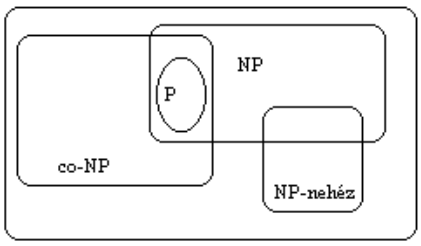
Az eldöntési problémák azon osztályát, amelyek megoldhatók a bemenő adat méretének polinomiális függvényével felülről becsült lépésszámban, *P-beli problémáknak* nevezzük.

Tehát minden probléma, amely megoldható determinisztikus Turing-gép segítségével ehhez az osztályhoz tartozik.

Vannak esetek, amikor csak úgy adhatunk választ egy kérdésre, hogy az összes lehetséges bemenetre bebizonyítjuk, hogy megfelel, vagy, hogy nem felel meg a feltételeknek. Természetesen ez a vizsgálat nagyon hosszú ideig is eltarthat. Az eldöntési problémák azon osztályát, amelyben az igenlő választ ezzel a módszerrel polinomiális időben be tudjuk bizonyítani, *NP-nek* nevezzük.

Azaz minden probléma, amely csak nemdeterminisztikus Turing-géppel oldható meg ide tartozik. Az ugyanezzel a módszerrel a negatív állítást bebizonyító algoritmusokat a *co-NP* osztályba soroljuk.

Nyitott kérdés, hogy $P \subset NP$ vagy $P = NP$. Nagyon sok olyan feladat van, melyet vissza lehet vezetni erre a problémára.



A probléma osztályok egy lehetséges viszonya.

- **Optimalizációs probléma**

  Ha egy ilyen feladattal találkozunk minden bemenő adathoz meg kell határoznunk a lehetséges megoldások halmazát, azok költségfüggvényével együtt. Célunk, hogy minimális, vagy maximális költségű eredményt találjunk.

  A feladat specifikációjához az alábbi négy paraméter meghatározása szükséges:

  – input esetek halmaza,

  – minden esethez az összes lehetséges megoldás halmaza,

– a lehetséges megoldások költségfüggvényei és

– optimalizálási cél

Ebbe a kategóriába sorolható az úgynevezett Utazóügynök–Probléma, melynek lényege, hogy adva van n város, illetve az utiköltség bármely két város között. Keressük a legolcsóbb utat egy adott pontból indulva, amely minden várost pontosan egyszer érint, majd a kiindulási helyre ér vissza. $\frac{(n-1)!}{2}$ út közül kell választanunk, ez ugyanis a Hamilton–körök száma az n pontú teljes gráfban $n \geq 2$ esetén. Bemenő paraméterként kapunk egy teljes, irányítatlan, súlyozott gráfot, melynek n csúcsa van és egy költségfüggvényt.

- **Informatikailag kezelhetetlen probléma**
  Azok a problémák amelyek a futási idő tekintetében a feladat méretének exponenciális függvényével jellemezhetők, számítástechnikai szempontból kezelhetetlenek.

- **Algoritmikusan eldönthetetlen probléma**
  Vannak problémák melyekhez nem konstruálható olyan Turing-gép, mely biztosan megáll. Ezeket algoritmikusan eldönthetetleneknek nevezzük. Annak megállapítása, hogy valamely probléma algoritmikusan eldönthetetlen-e algoritmikusan eldönthetetlen probléma.

4. Mintaillesztés

A következő metódusok a biológiai szekvenciák pontos összehasonlítását és az azokban található ismétlődő minták megtalálását tűzik ki célul. Az egyik legelemibb és leggyakrabban szavakkal kapcsolatosan előforduló probléma a mintaillesztés problémája.

Feladat: Annak eldöntése, hogy a bemenő adatként adott minta megtalálható-e az ugyancsak bemenő adatként adott szövegben, és ha megtalálható mely pozícióktól kezdve jelenik meg. Ez a probléma nagyon sok olyan területen jelenik meg, ahol szükség van egy viszonylag hosszú szövegben való keresésre. A molekuláris biológiában többnyire akkor használják, amikor egy szekvenált DNS részletet keresnek egy adatbázisban.

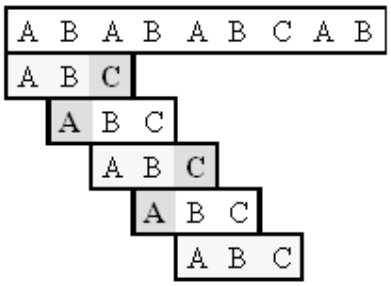
Definíció: Legyen Σ egy tetszőleges ábécé, továbbá fel kell tenni, hogy S és M szavakon megengedett művelet az indexelés.

- *Input:* $S = s_1 \dots s_n$ egy n hosszú Σ feletti véges szó és
  $M = m_1 \dots m_m$ egy ugyancsak Σ feletti véges szó, mely a keresés mintája.
- *Output:* Egy halmaz, mely tartalmazza az összes olyan S-beli pozíciót, melytől kezdve az M szó megtalálható az S szövegben, azaz
  $I \subseteq \{1, .., n - m + 1\}$ és $\forall i \in I : s_i \dots s_{i+m-1} = m_1 \dots m_m$.
- Legyen $k \in \{0, .., n-m\}$ Az M minta a $k+1$-edik pozícióban illeszkedik az S szövegre, ha $s_{k+1} \dots s_{k+m} = m_1 \dots m_m$.

Algoritmusok:

4.1. Egyszerű mintaillesztés (Brute–Force algoritmus)

Ez a naiv algoritmus egy a mintát tartalmazó csúszó ablakot tologat az S szöveg fölött, amelyben keresünk. Minden egyes pozícióban ellenőrzi, hogy a minta azonose az aktuális részszóval. Ez a vizsgálat hatékonyabbá tehető, ha az első olyan $j \in [1, .., m]$ pozíciót keressük, melyre $s_{k+j} \neq m_j$. Az eljárás tulajdonképpen egy lineáris keresésbe ágyazott lineáris keresést valósít meg.
Példa:



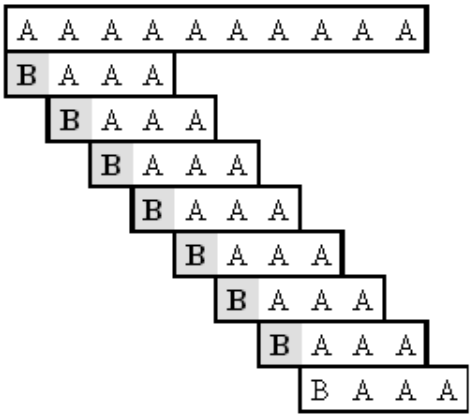
Mielőtt elkezdjük a hatékonyságelemzést tisztáznunk kell, hogy pontosan milyen információra van szükségünk. Ha a mintánk előfordulásai közül csak az első kezdő pozíciójára vagyunk kíváncsiak, akkor műveletigény szempontjából a legjobb eset, amikor a minta az első pozíción illeszkedik a szövegre. Ekkor az összehasonlítások száma minden mintaillesztő algoritmusnál m lesz. Nem érdemes azonban ezt az esetet legjobb esetként tekinteni, mivel nem ad az algoritmus sebességére vonatkozóan valósághű jellemzést. A továbbiakban a mintaillesztési algoritmusok vizsgálata során mindig feltesszük, hogy a minta nem fordul elő a szövegben, így az algoritmusnak fel kell dolgoznia a „teljes” bemenő adatot (vagy a dokumentum legalább akkora részét, amelyből már következik, hogy a minta nem fordul elő benne). A

különböző algoritmusok hatékonysága a szöveg teljes feldolgozásának gyorsaságában nyilvánul meg. Ha a minta összes előfordulásának helyét akarjuk megtudni, akkor eleve szükség van a teljes, vagy majdnem teljes szöveg vizsgálatára. Bármelyik esetet is választjuk fel kell tennünk, hogy a minta hossza nagyságrendben nem nagyobb, mint a szöveg hossza, azaz $m = O(n)$. Ha a fenti feltételeket figyelembe vesszük a legjobb esetben a minta első karaktere nincs a szövegben, így minden k eltolásnál már $j = 1$ esetben elromlik az illeszkedés.

Tehát minden eltolásnál csak egy összehasonlítást végzünk el, azaz az összehasonlítások száma megegyezik az eltolások számával, $n - m + 1$ -gyel.
Ebből következik, hogy a minimális összehasonlítási szám:

$$n - m + 1 = \Theta(n)$$

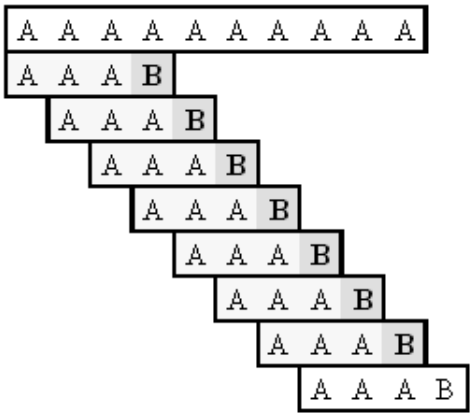
Példa:



A legrosszabb esetben minden eltolásánál csak a minta utolsó karakterénél romlik el az illeszkedés. Ekkor minden eltolásnál m összehasonlítást végzünk, így a műveletigény az eltolások számának m-szerese. Így a maximális összehasonlítások száma

$$(n - m + 1) \cdot m = \Theta(n \cdot m)$$

Példa:

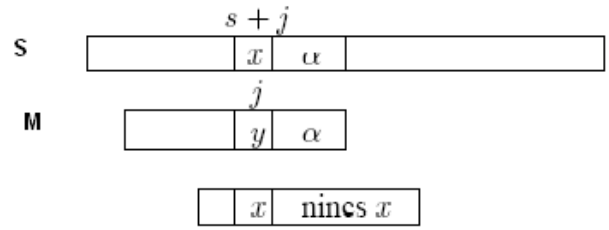


Sajnos ezen algoritmus műveletigénye a legrosszabb esetben nagyon nagy, ez a futási idő növekedését is maga után vonja. Ezt a lentebb található szimulációs eredmények is igazolják. Így elsődleges célunk, hogy hatékonyabb megoldást keressünk a problémára.

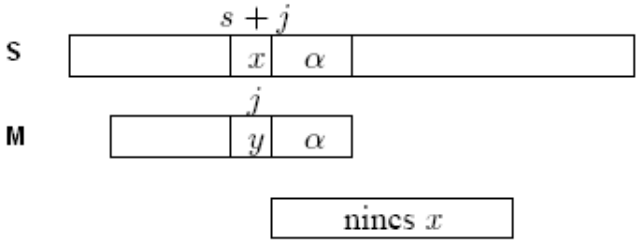
4.2. Boyer–Moore algoritmus

Ez az eljárás is ugyanazon alapelvek alapján működik, mint az előbbi, viszont sok összehasonlítást takarít meg egy hatékony előfeldolgozási metódus alkalmazásával. Annak ellenére, hogy a legrosszabb esetben ez az eljárás is hasonló eredményeket produkál, a gyakorlati alkalmazások során a futási ideje sok esetben határozottan rövidebb, így gyakrabban találkozhatunk vele. Alapvetően itt is a keresett mintát toljuk el balról jobbra a szöveg felett, viszont az összehasonlítást az egyszerű mintaillesztéssel ellentétben jobbról balra végezzük, tehát amikor a $s_{j+1} \dots s_{j+m}$ részszóhoz akarjuk hasonlítani a $m_1 \dots m_m$ sablonunkat, akkor először az $s_{j+m} = m_m$ állítást ellenőrizzük, és így tovább visszafelé haladva egészen addig, míg $s_{j+i} \neq m_i$, vagy a minta elejére nem érünk. Amint eltérést találunk jobbra toljuk az ablakot. Az eltolás nagyságát a következő két szabály határozza meg:

Utolsó pozíció heurisztika: Minden ábécébeli betűhöz megadja, hogy hol fordult elő utoljára a mintában. Ha a szöveg aktuálisan vizsgált betűje nem található meg a mintában, akkor a sablont eltolhatjuk a karakter utánra. Ha szerepel benne, akkor a legutolsó előfordulását illesztjük a szöveg megfelelő karakteréhez. Ha ebben az esetben negatív, azaz bal irányba kellene eltolni az ablakot a másik szabályt alkalmazzuk.

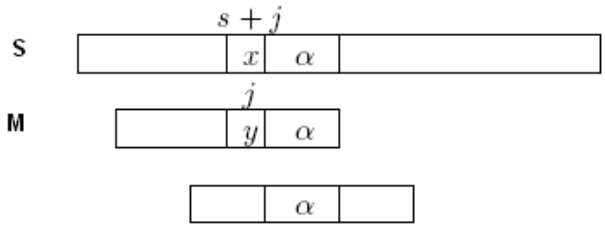


Utolsó pozíció heurisztika, amikor a nem egyező karakter szerepel a mintában

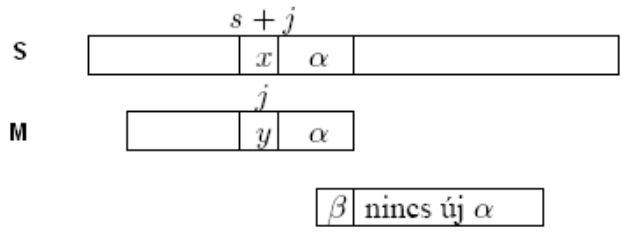


Utolsó pozíció heurisztika, amikor a nem egyező karakter nem szerepel a mintában

Jó szuffix heurisztika: A mintát úgy mozgatjuk, hogy a szövegnek arra a részére, amelyre már illeszkedett a minta vizsgált része, újra (legalább részben) illeszkedjen. Ha az illeszkedő részminta (α) újra előfordul a szövegben, akkor ezt a részt kell illeszteni, egyébként eme minta egy szuffixét (β) kell illeszteni a szöveghez.

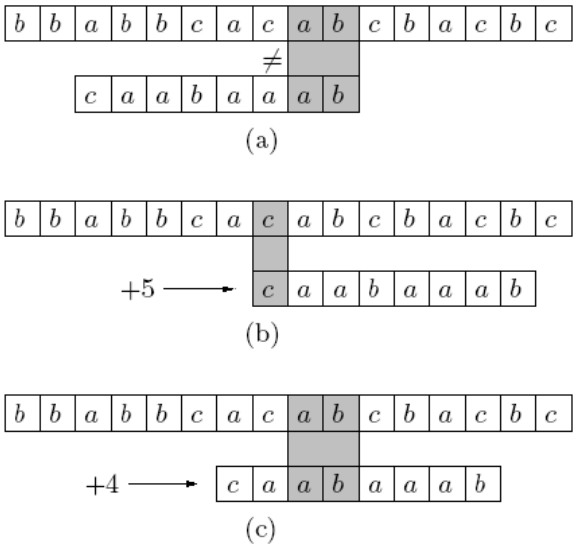


A Jó szuffix heurisztika, mikor az α újra előfordul



A Jó szuffix heurisztika, mikor az α nem fordul elő újra

Példa:



A Boyer–Moore algoritmus nem illeszkedés esetén a két heurisztika közül azt választja, amellyel nagyobbat léphet. Noha a módszer bonyolultsága $O(n \cdot m)$, a legrosszabb eset csak kivételes esetekben fordul elő, s kellően nagy ábécé esetén az algoritmus rendszerint nagyon gyors. Ez az oka annak, hogy napjainkban szinte minden szövegszerkesztőben ezt a módszert implementálták.

```
Function UTOLSÓ-POZÍCIÓ(P,m,S)
  Input: P minta, m minta hossza,S ábécé
  Output: u utolsó pozíció heurisztika értékei
1 forall the a ∈ S do u[a] ← 0
2 for j ← 1 to m do u[P[j]] ← j
3 return u
```

```
Function JÓ-SZUFFIX(P,m)
  Input: P minta, m minta hossza
  Output: g jó szuffix heurisztika értékei
1 p1 ← PREFIX-FÜGGVÉNY-SZÁMÍTÁS(P)
2 P' ← FORDÍT(P)
3 p2 ← PREFIX-FÜGGVÉNY-SZÁMÍTÁS(P')
4 for j ← 0 to m do g[j] ← m − p1[m]
5 for l ← 1 to m do
6 |   j ← m − p2[l]
7 |   if g[j] > l − p2[l] then g[j] ← l − p2[l]
8 endfor
9 return g
```

```
Procedure BOYER-MOORE-ILLESZTŐ(T,P,S)
  Input: T szöveg, P minta, S ábécé
  Eredmény: illeszkedés esetén figyelmeztetés
1  n ← T.hossz
2  m ← P.hossz
3  u ← UTOLSÓ-POZÍCIÓ(P,m,S)
4  g ← JÓ-SZUFFIX(P,m)
5  s ← 0
6  while s ≤ n − m do
7  |   j ← m
8  |   while j > 0 and P[j] = T[s + j] do j ← j − 1
9  |   if j = 0 then
10 |   |   kiír illeszkedés az s pozíción
11 |   |   s ← s + g[0]
12 |   else
13 |   |   s ← s + max(g[j], j − u(T[s + j]))
14 |   endif
15 endw
```

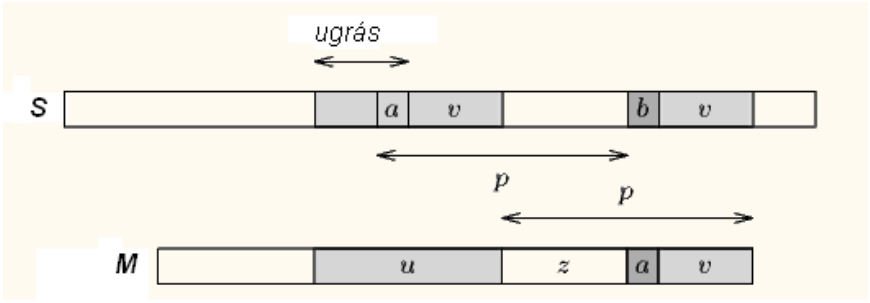
Megjegyzés: A pszeudokódok az [1] forrásból származnak.

Ez az algoritmus példa arra, hogy az előfeldolgozó eljárás milyen nagy mértékben képes növelni egy algoritmus hatékonyságát.

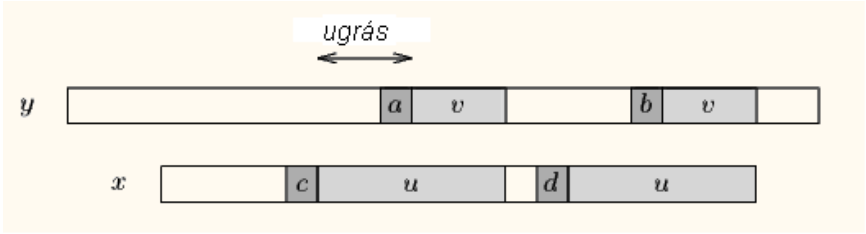
4.3. Gyors Boyer–Moore algoritmus

A Boyer–Moore algoritmus egy javított változata, mely nem igényli további előfeldolgozó eljárás alkalmazását, csak a memóriaigénye nagyobb, de a többlet adatok eltárolásához csupán nagyságrendileg konstans méretű memória szükséges. Azzal egészíti ki a már korábban tárgyalt eljárást, hogy minden Jó szuffix szabály

alkalmazás után megjegyzi a szövegnek azt a részét, mely megegyezett a minta szuffixével. Így lehetővé válik, hogy egyes esetekben a szöveg ezen részét átugorva jelentősen csökkentsük az összehasonlítások számát és ezzel a futási időt elég nagy bemenő adatok esetében. Ez az ugrás azonban csak akkor lehetséges, ha az aktuális összehasonlítás során kapott, a minta szuffixével megegyező szövegrészlet hossza kisebb, mint a korábban megjegyzett szuffix.
Legyen u a legutóbb elmentett szufix és v az aktuális illesztés során talált szuffix. A lenti ábrán is látszik, hogy uzv a szuffixe M mintának. Legyenek a és b azok a karakterek, melyek nem egyeztek meg az aktuális összehasonlítás során. Ekkor av egy szuffixe M-nek. A két nem egyező karakter előfordulása p távolságra van egymástól a szövegben és M $|uzv|$ hosszúságú szuffixe $p = |zv|$ hosszúságú periódusonként jelenik meg, mivel u az uzv határán helyezkedik el p távolságon belül nem tudja lefedni az a és b karaktereket egyszerre a szövegben. A legkisebb ugrás hossza ebben az esetben $|u| - |v|$



Abban az esetben, amikor $|v| < |u|$ teljesül és az utolsó pozíció heurisztika szerinti ugrás nagyobb, mint a Jó szuffix heuriszikáé és az előbb leírt ugrásé, akkor az aktuális ugrásnak $|u| + 1$ -től hosszabbnak kell lenni. Ezt az esetet a következő ábra szemlélteti.



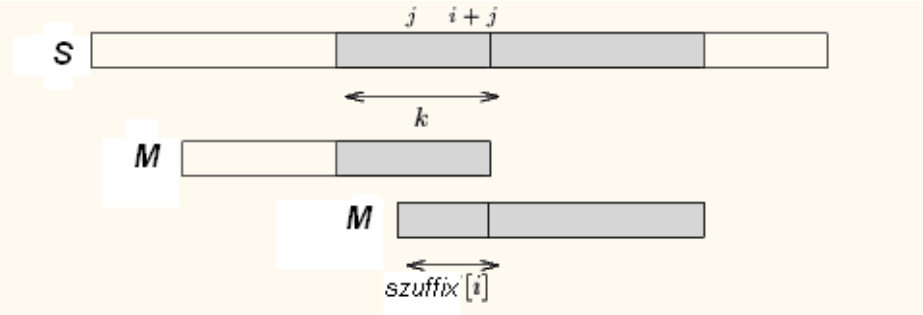
Tehát amikor $c \neq d$ nem hasonlíthatjuk össze őket v-ben ugyanazzal a karakterrel, de ha nem kötnénk ki, hogy ebben az esetben legalább $|u| + 1$ hosszú ugrást kell tenni, akkor megtörténne az összehasonlítás.

Ennél az algoritmusoknál az összehasonlítások számát $O(2n)$-el becsülhetjük.

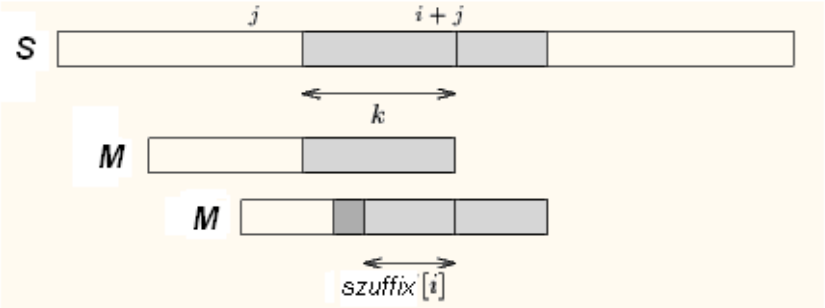
4.4. Apostolico–Giancarlo algoritmus

Ez az eljárás is a Boyer–Moore algoritmus egy változata. Apostolico és Giancarlo egy olyan algoritmust tervezett, mely minden kísérlet végén megjegyzi a leghosszabb illesztett szuffix hosszát. Ezeket az információkat egy vektorban tároljuk, amit most nevezzünk *„shift"*-nek. Tegyük fel, hogy a j-edik pozíció előtti illesztéshez tartozó szuffix k hosszú.
Ekkor $shift[i+j] = k$. Vezessünk be egy másik vektort, melyet nevezzünk *„szuffix"*-nek. A $szuffix[i]$-ben tároljuk a minta azon leghosszabb szuffixének a hosszát, mely az i-edik pozícióban ér véget ($0 \leq i < m$). Amikor a j-edik pozícióban végezzük az illesztést az algoritmus az $s_{i+j+1} \ldots s_{j+m-1}$ részszót vizsgálja.
Ekkor 4 eset lehetséges:

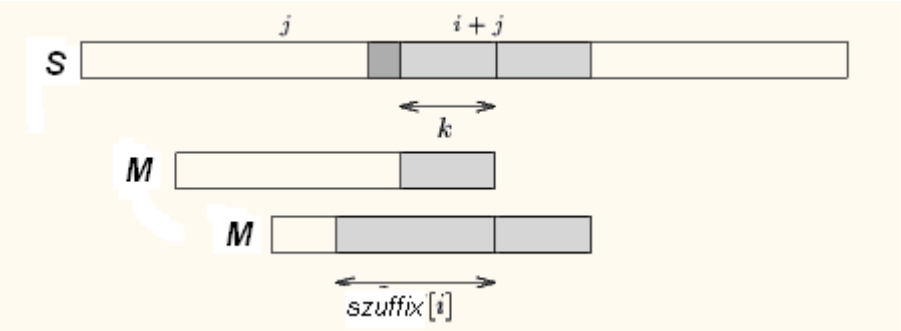
1. eset: $k > szuffix[i]$ és $szuffix[i] = i$.
   Ez azt jelenti, hogy megtaláltuk M egy előfordulását a j-edik pozícióban és $skip[j + m - 1]$ értékét m-re állítjuk.



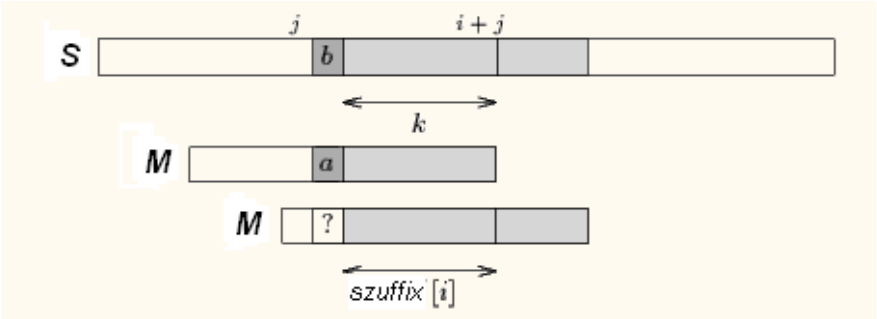
2. eset: $k > szuffix[i]$ és $szuffix[i] \leq i$.
   Ez azt jelenti, hogy $m_{i-szuffix[i]} \neq s_{i+j-szuffix[i]}$
   és $skip[j + m - 1]$ értékét beállítjuk $m - 1 - i + szuffix[i]$-re.



3. eset: $k < szuffix[i]$.
   Ez azt jelenti, hogy $m_{i-k} \neq s_{i+j-k}$ és $skip[j+m-1]$ értékét $m-1-i+k$-ra állítjuk.



4. eset: $k = szuffix[i]$.
   Ez az egyetlen eset, amikor átugorhatjuk az $s_{i+j-k+1} \dots s_{i+j}$ részszót és folytathatjuk az illesztést az s_{i+j-k} és m_{i-k} karakterek illesztésével.



A két vektort minden egyes pozícióban aktualizálni kell, mégpedig a következő módon:

$$skip[j+m-1] = \max\{k | m_{m-k} \dots m_{m-1} = s_{j+m-k} \dots s_{j+m-1}\}.$$
$$szuffix[i] = \max\{k | m_{i-k+1} \dots m_i = s_{m-k} \dots m_{m-1}\}.$$

Ezen algoritmus futási ideje megegyezik az eredeti Boyer–Moore algoritmus futási idejével, viszont az összehasonlítások száma a legrosszabb esetben $O(\frac{3}{2}n)$.
Megjegyzés: A fejezethez az [1, 3, 4, 5] forrásokat használtam fel.

4.5. Szimulációs eredmények

Ebben az alfejezetben az előbbiekben tárgyalt algoritmusokkal kapcsolatos, általam írt program alapján zajló szimuláció eredményeit foglalom össze. Az implementált eljárások a minta minden előfordulását megkeresik, ebből adódóan minden esetben feldolgozásra kerül a teljes szöveg, vagy annak elegendően nagy része. A bemenő paramétereknél figyelembe vettem azok hosszúságát, valamint három különböző elemszámú ábécén is futtattam az algoritmusokat. Egy 20 és egy 4 elemű ábécébeli szavakat vizsgáltam. Választásom azért esett e két ábécére, mert a biológiai szekvenciák (protein és DNS szekvenciák) is ilyen ábécék feletti szóként értelmezhetők.Igyekeztem szemléltetni az ábécé számosságának befolyásolási tényezőjét is. Annak érdekében, hogy a tesztadataim valósághűek legyenek konkrét, létező DNS és protein szekvencia részleteken hajtottam végre a szimulációt. A DNS szekvenciákat a GenBank adatbázisból vettem. Az itt található szekvenciák felépítésére egy példa a [12] honlapon található. A protein szekvencia részleteket a SwissProt adatbázisból választottam. Fasta formátumú adatokat vizsgáltam. Ezek struktúrájára a [13] forrásban találhatunk példát. Mindkét ábécéből 20 szöveg-minta párt vettem, a táblázatokban sorszámmal azonosítottam őket. A 4 elemszámú ábécén végzett szimuláció eredményeit a következő táblázatokban foglalom össze.

Jelölések: BF = Brute–Force algoritmus, BM = Boyer–Moore algoritmus, GyBM = Gyors Boyer-Moore algoritmus, AG = Apostolico–Giancarlo algoritmus.

			Futási idők			
Sorszám	Minta hossz	Szöveg hossz	BF	BM	GyBM	AG
1.	5	10	5,2E-06	5,2E-06	4,6E-06	4,6E-06
2.	10	20	1,05E-05	3,9E-06	3,3E-06	4,6E-06
3.	10	40	3,4E-05	7,9E-06	8,58E-06	1,5E-05
4.	10	80	4,6E-05	1,8E-05	1,9E-05	2,0E-05
5.	10	120	9,3E-05	2,3E-05	1,8E-05	4,6E-05
6.	20	100	6,27E-05	2,04E-05	1,8E-05	1,6E-05
7.	50	120	6,27E-05	5,2E-06	9.24E-06	8,58E-06
8.	50	240	1,6E-04	1,2E-05	2,0E-05	2,3E-05
9.	50	280	1,9E-04	2.9E-05	2,8E-05	2,7E-05
10.	50	320	2,6E-04	5,4E-05	5,9E-05	3,2E-05
11.	60	270	2,0E-04	1,32E-05	2,3E-05	3,4E-05
12.	60	320	2,5E-04	4,4E-05	1,2E-05	5,2E-05
13.	70	280	1,9E-04	2,0E-05	3,4E-05	5,28E-05
14.	80	120	3,7E-05	5,94E-06	5,94E-06	8,58E-06
15.	80	240	1,6E-04	1,6E-05	1,6E-05	1,58E-05
16.	80	320	2,23E-04	4,35E-05	4,2E-05	4,5E-05
17.	95	240	1,37E-04	1,5E-05	1,05E-05	2,7E-05
18.	95	320	2,72E-04	4,22E-05	1,05E-05	5,8E-05
19.	100	280	1,69E-04	4,6E-05	1,05E-05	6,4E-05
20.	100	350	2,4E-04	1,5E-05	1,5E-05	4,62E-05

			Összehasonlítások száma			
Sorszám	Minta hossz	Szöveg hossz	BF	BM	GyBM	AG
1.	5	10	8	8	7	7
2.	10	20	16	6	5	7
3.	10	40	52	12	13	23
4.	10	80	71	30	29	31
5.	10	120	142	36	28	71
6.	20	100	95	31	30	25
7.	50	120	95	8	14	13
8.	50	240	249	32	31	35
9.	50	280	302	44	43	41
10.	50	320	393	82	90	49
11.	60	270	308	20	36	58
12.	60	320	380	68	19	79
13.	70	280	297	31	52	80
14.	80	120	57	9	9	13
15.	80	240	255	25	25	24
16.	80	320	338	66	64	75
17.	95	240	208	16	16	41
18.	95	320	413	64	16	89
19.	100	280	257	70	16	98
20.	100	350	368	25	25	70

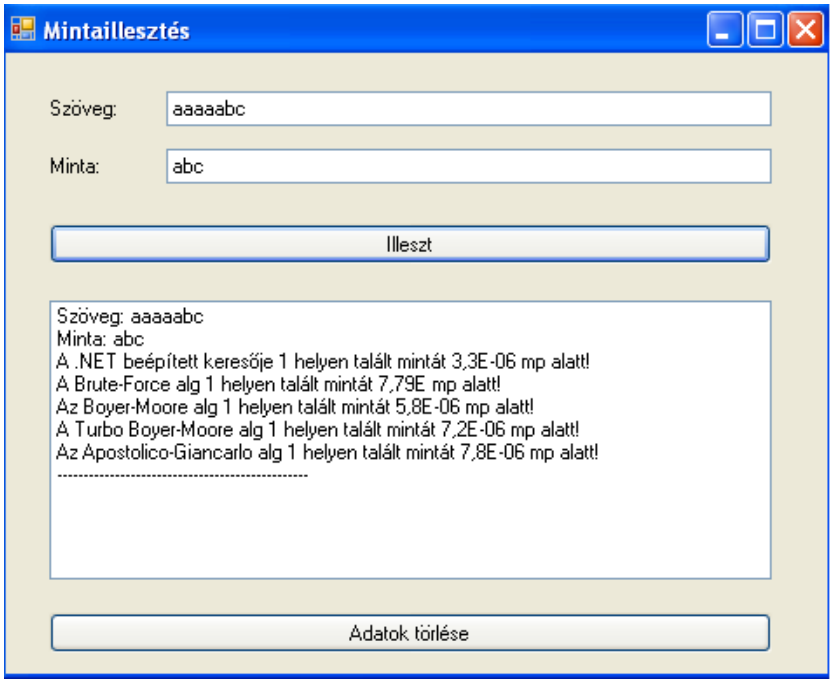
			Menetek száma			
Sorszám	Minta hossz	Szöveg hossz	BF	BM	GyBM	AG
1.	5	10	6	3	3	3
2.	10	20	11	2	2	3
3.	10	40	41	4	4	5
4.	10	80	85	8	8	15
5.	10	120	121	12	22	17
6.	20	100	81	14	20	18
7.	50	120	71	14	8	8
8.	50	240	191	14	14	23
9.	50	280	231	32	28	27
10.	50	320	271	23	23	31
11.	60	270	211	33	16	15
12.	60	320	261	25	18	33
13.	70	280	211	18	18	10
14.	80	120	41	9	9	13
15.	80	240	161	19	19	18
16.	80	320	241	28	27	29
17.	95	240	147	11	8	47
18.	95	320	227	22	8	46
19.	100	280	183	33	8	49
20.	100	350	254	16	16	54

Összefoglalás: Az adatok alapján látható, hogy az elméletben az összehasonlításokkal kapcsolatban megfogalmazott korlátok értékeit csak nagyon ritkán közelítik meg az eredmények. A Brute–Force algoritmus összehasonlítási száma és ebből adódóan futási ideje egyes esetekben kiugróan magas, míg a Boyer–Moore és változatai sokkal hatékonyabban, lényegesen kisebb futási idő alatt oldják meg a problémát. Az eredményekből én azt a következtetést vonom le, hogy a Gyors Boyer–Moore és az Apostolico–Giancarlo algoritmus összehasonlítási számára vonatkozó felső korlát csökkenthető. A táblázat alapján a gyakorlatban előforduló esetebben a Gyors Boyer–Moore algoritmus nem éri el az $O(nm)$ összehasonlítási számot, és még a legrosszabb esetben sem végez a Apostolico–Giancarlo algoritmus $O(\frac{3}{2}n)$ összehasonlítást. Az is szembetűnő, hogy a Boyer–Moore algoritmus változatai elég nagy bemenő adatok

esetében különösen hatékonnyá tudják tenni a feldolgozást. Ez a két eljárás a legtöbb esetben közel egyforma értékeket produkál.
Fontos megemlíteni, hogy az ábécék elemszáma is befolyásolja az algoritmusok hatékonyságát. Ez elsősorban az előfeldolgozást és így nagyobb ugrásokat végrehajtó eljárások esetében lehet megfigyelni.

4.5.1. Az algoritmusok implementációja

A fenti szimulációkat a következő *C#* nyelven megírt eljárásokkal valósítottam meg. A lenti kód részletek felelnek az egyes algoritmusokért.



Brute–Force algoritmus:

```
public IEnumerable<int> BruteForce(string text, int startingIndex)
{
    int patternLength = m_pattern.Length;
    int textLength = text.Length;
    int talalat = 0;

    for (int index = startingIndex; index < textLength - patternLength + 1; index++)
    {
        for (int j = 0; j < patternLength && m_pattern[j] == text[index + j]; j++)
        {
            if (j >= patternLength - 1)
            {
                talalat += 1;
            }
        }
    }
   yield return talalat;
}
```

Boyer–Moore algoritmus és az előfeldolgozást végző eljárások:

```
private int[] BadCharacterShift(string pattern)
{
  int[] badCharacterShift = new int[256];

  for (int c = 0; c < badCharacterShift.Length; ++c)
    badCharacterShift[c] = pattern.Length;
  for (int i = 0; i < pattern.Length - 1; ++i)
    badCharacterShift[pattern[i]] = pattern.Length - i - 1;

  return badCharacterShift;
}
```

```
private int[] GoodSuffixShift(string pattern, int[] suff)
{
  int patternLength = pattern.Length;
  int[] goodSuffixShift = new int[pattern.Length + 1];

  for (int i = 0; i < patternLength; ++i)
    goodSuffixShift[i] = patternLength;
  int j = 0;
  for (int i = patternLength - 1; i >= -1; --i)
    if (i == -1 || suff[i] == i + 1)
      for (; j < patternLength - 1 - i; ++j)
        if (goodSuffixShift[j] == patternLength)
          goodSuffixShift[j] = patternLength - 1 - i;
  for (int i = 0; i <= patternLength - 2; ++i)
    goodSuffixShift[patternLength - 1 - suff[i]] = patternLength - 1 - i;

  return goodSuffixShift;
}
```

```
public IEnumerable<int> BoyerMooreMatch(string text, int startingIndex)
{
  int patternLength = m_pattern.Length;
  int textLength = text.Length;

  /* Searching */
  int index = startingIndex;
  while (index <= textLength - patternLength)
  {
    int unmatched;
    for (unmatched = patternLength - 1;
      unmatched >= 0 && (m_pattern[unmatched] == text[unmatched + index]);
      --unmatched
    )
      ; // empty

    if (unmatched < 0)
    {
      yield return index;
      index += m_goodSuffixShift[0];
    }
    else
      index += Math.Max(m_goodSuffixShift[unmatched],
        m_badCharacterShift[text[unmatched + index]] - patternLength + 1 + unmatched)
  }
}
```

Gyors Boyer–Moore algoritmus:

```
public IEnumerable<int> GyorsBoyerMooreMatch(string text, int startingIndex)
{
  int patternLength = m_pattern.Length;
  int textLength = text.Length;

  /* Keresés */
  int index = startingIndex;
  int overlap = 0;
  int shift = patternLength;
  while (index <= textLength - patternLength)
  {
    int unmatched = patternLength - 1;

    while (unmatched >= 0 && (m_pattern[unmatched] == text[unmatched + index]))
    {
      --unmatched;
      if (overlap != 0 && unmatched == patternLength - 1 - shift)
        unmatched -= overlap;
    }
    if (unmatched < 0)
    {
      yield return index;
      shift = m_goodSuffixShift[0];
      overlap = patternLength - shift;
    }
```

```
    else
    {
      int matched = patternLength - 1 - unmatched;
      int turboShift = overlap - matched;
      int bcShift = m_badCharacterShift[text[unmatched + index]] - patternLength + 1
      shift = Math.Max(turboShift, bcShift);
      shift = Math.Max(shift, m_goodSuffixShift[unmatched]);
      if (shift == m_goodSuffixShift[unmatched])
        overlap = Math.Min(patternLength - shift, matched);
      else
      {
        if (turboShift < bcShift)
          shift = Math.Max(shift, overlap + 1);
        overlap = 0;
      }
    }

      index += shift;
    }
  }
```

Apostolico–Giancarlo algoritmus:

```
  public IEnumerable<int> ApostolicoGiancarloMatch(string text, int startingIndex)
  {
    int patternLength = m_pattern.Length;
    int textLength = text.Length;
    int[] skip = new int[patternLength];
    int shift;
    int index = startingIndex;
    while (index <= textLength - patternLength)
    {
      int unmatched = patternLength - 1;
      while (unmatched >= 0)
      {
        int skipLength = skip[unmatched];
        int suffixLength = m_suffixes[unmatched];
        if (skipLength > 0)
          if (skipLength > suffixLength)
          {
            if (unmatched + 1 == suffixLength)
              unmatched = (-1);
            else
              unmatched -= suffixLength;
            break;
          }
          else
          {
            unmatched -= skipLength;
            if (skipLength < suffixLength)
              break;
          }
        else
        {
          if (m_pattern[unmatched] == text[unmatched + index])
            --unmatched;
          else
            break;
        }
      }
```

```
      if (unmatched < 0)
      {
        yield return index;
        skip[patternLength - 1] = patternLength;
        shift = m_goodSuffixShift[0];
      }
      else
      {
        skip[patternLength - 1] = patternLength - 1 - unmatched;
        shift = Math.Max(m_goodSuffixShift[unmatched],
          m_badCharacterShift[text[unmatched + index]] - patternLength + 1 + unmatched
          );
      }
      index += shift;

      for (int copy = 0; copy < patternLength - shift; ++copy)
        skip[copy] = skip[copy + shift];

      for (int clear = 0; clear < shift; ++clear)
        skip[patternLength - shift + clear] = 0;
    }
  }
```

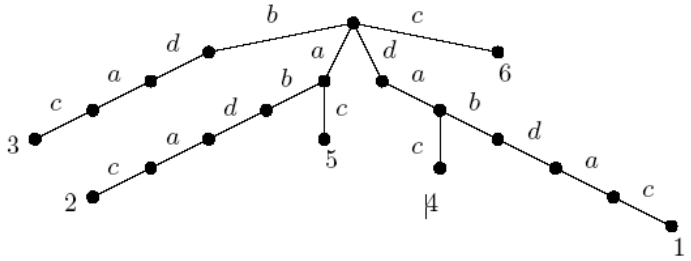
4.6. Szuffix-fák

Ennek a struktúrának a segítségével elérhetjük, hogy a minta hosszának figyelembevételével lineáris idő alatt megoldást kapjunk a mintaillesztés problémájára. Ezzel a metódussal nagy mértékű gyorsulást tudunk elérni, különösen abban az esetben, amikor több különböző mintát is szeretnénk egy szöveghez hasonlítani. Ez a probléma gyakran felvetődik a molekuláris biológiában, például amikor több szekvenált DNS töredéket szeretnénk összevetni adatbázisban tárolt korábban előállított DNS sorozatokkal. Ezen fák segítségével hatékonyan tárolhatunk információkat egy szöveg szuffixéről elősegítve ezzel a gyorsabb mintával való összehasonlítást.

Definíció: Legyen $S = s_1 \dots s_n \in \Sigma^n$ egy szó Σ ábécé felett és $T_s = (V, E)$ egy irányított fa r gyökérrel, melyet S egyszerű *szuffix-fájának* nevezünk, ha eleget tesz a következő követelményeknek:

1. A fának pontosan n db levele van, melyeknek címkéi rendre $1, \dots, n$.
2. A fa élei a Σ ábécé betűivel vannak címkézve.
3. Minden belső csúcsból annak gyermekeibe mutató él páronként különböző szimbólumokkal rendelkezik.

4. Az utat a fa gyökerétől az i-edik levélig az azt alkotó élek címkéinek összefűzéséből kapjuk.



Egy S szöveghez akkor és csak akkor létezik egyszerű szuffix-fa, ha nem tartalmaz olyan szuffix részt, mely prefixe lenne egy másik szuffixének. Ellenkező esetben nem tenne eleget a definícióban leírt követelményeknek, hisz ekkor a belső csúcsokból kifutó élek címkéi nem lennének páronként különbözőek. Szerencsére van egy egyszerű stratégiánk ennek a gondnak a kiküszöbölésére, melynek lényege, hogy bevezetünk egy új „$” szimbólumot az S szövegbe, mely nem található meg a Σ ábécében és a szuffix-fát az S\$-ra építjük fel. Így garantáljuk, hogy egy szuffix sem lesz egy másik szuffix prefixe.

Tétel. Legyen $S = s_1 \dots s_n \in \Sigma^n$ egy szó és $\$ \notin \Sigma$. Ekkor a következő algoritmus megadja az S\$ egyszerű szuffix-fáját.

Algoritmus. $\Sigma' = \Sigma \cup \$$ és $S' = S\$$

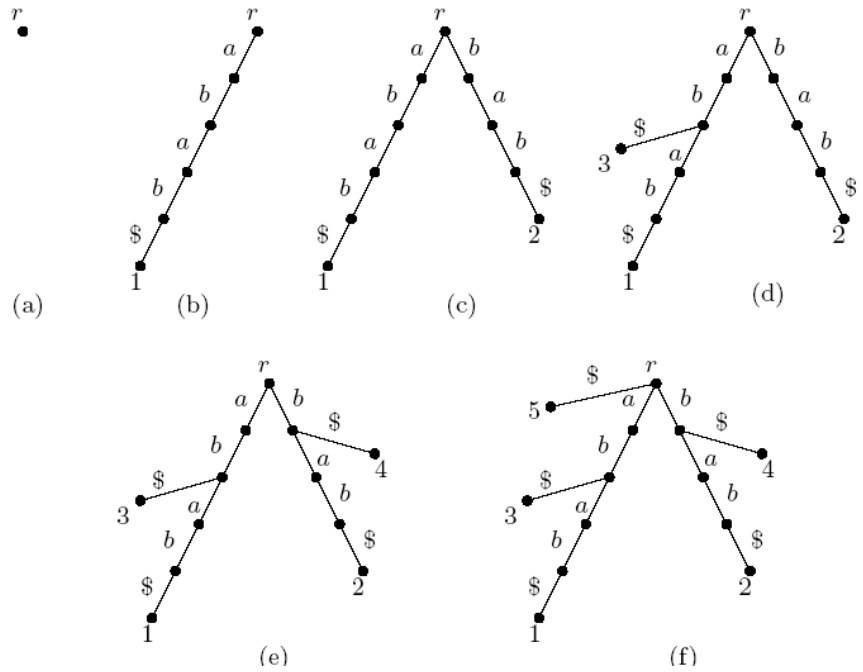
Inicializáljuk a T_s fát r gyökérrel és az élek üres halmazával
for $i = 1$ to n do
A gyökértől kezdve keressük a maximális prefix utat, melynek címkéje $s_i, .., s_{j_i}$ és végpontja x_i a $T_{s'}$ fában. Ez az út egyértelműen megadható.
Ezt követően hozzá adjuk az $x_i, y_{i_{j_{i+1}}}, .., y_{i_n}, y_{i_{n+1}}$ utat, melynek címkéje $s_{j_i+1} \dots s_n\$$ lesz. $y_{i_{j_{i+1}}}, .., y_{i_n}, y_{i_{n+1}}$ új csúcsok a fában.
Az $y_{i_{n+1}}$ csúcsot i-nek nevezzük. Végül kimenetként megkapjuk a teljes szuffix-fát.

Bizonyítás. Minden lépésben csupán egyetlen úthoz adunk hozzá egy újabb csúcsot egy új él felhasználásával. Mivel egy egyetlen csúcsból álló gráfból indulunk ki, és az egyes lépések után is teljesülnek a fa definíciójában adott tulajdonságok abban biztosak lehetünk, hogy az eljárás végén egy fát kapunk.

Igazolni kell továbbá, hogy a 4.6 pontban meghatározott tulajdoságok teljesülnek:

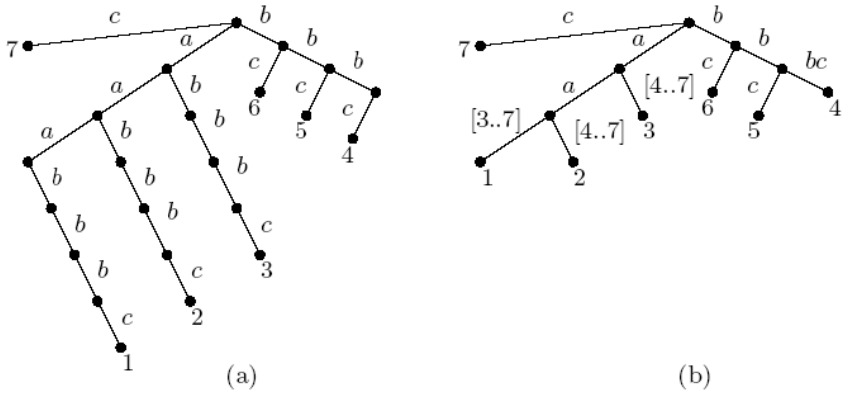
1. A szuffixek a hosszuk alapján csökkenő sorrendben kerülnek beszúrásra a fába, ez biztosítja, hogy minden i-edik lépésben az s_i-edik csúcshoz tartozó utat határozzuk meg és s_i nem levél. Továbbá mivel egyik szuffix sem lehet bármely más szuffix prefixe az út, melyet a lépések során hozzáadunk a fához nem lehet üres és minden lépésben pontosan egy új levél jön létre. Mindezekből adódik, hogy a fának n db levele van és ezek rendre az $1, .., n$ címkéket kapják.
2. Mivel $\Sigma' = \Sigma \cup \$$, ez teljesül.
3. Ez a tulajdonság abból következik, hogy minden i-edik lépésben az algoritmus a maximális hosszúságú utat keresi, mely a gyökérből indul, $s_i \dots s_n\$$ prefixével van címkézve és az új út ezen maximális út végpontjához fog tartozni. Ha az algoritmus egy olyan élt adna hozzá az s_i-edik csúcshoz, mely már létezik egy már kimenő élen az út nem lenne maximális.
4. Ez az algoritmus közvetlen következménye. □

Példa:



Az „*abab$*” szó szuffix-fájának felépítése a tétel alapján

Ha egy S szöveghez ismerünk egy egyszerű szuffix-fát egy véges hosszúságú mintáról könnyen meg tudjuk állapítani, hogy megtalálható-e a szövegben, mégpedig úgy, hogy elindulunk a fa gyökerétől és egy a minta által címkézett utat keresünk. Ha találunk ilyet a minta szerepel a szövegben, ellenkező esetben nem.
Ezt az eljárást alkalmazva $O(m)$ idő alatt el lehet dönteni a problémát. Ha az M minta minden előfordulásának kezdő pozíciójára kíváncsiak vagyunk a futási idő a fa azon részfájának nagyságától függ, mely tartalmazza a mintát.



a) Az $S = a^3b^3c$ szó egyszerű szuffix-fája
b) Az S szó kompakt szuffix-fája

Sajnos egy S szöveg egyszerű szuffix-fája elérheti akár a $\Omega(|S|^2)$méretet is, így a már említett részfa túlságosan nagy a hatékony bejárás megvalósításához. Ez a probléma orvosolható a szuffix-fák egy hatékonyabb reprezentációjával, a kompakt szuffix-fával, mely nem jelöli azokat a csúcsokat, melyeknek a fokszáma 1, helyettük az S szövegbeli kezdő és végpontjukat jeleníti meg.
Egy k hosszúságú út megjelenítését akkor tudjuk ily módon csökkenteni, ha

$$2 \cdot \log_2 n \leq k \cdot \log_2 |\Sigma|$$
$$k \geq 2 \cdot \log_2(n - |\Sigma|).$$

Definíció: $S = s_1 \dots s_n \in \Sigma^n$ egy szó. Egy $T_S = (V, E)$ irányított fa, melynek gyökere r *S kompakt szuffix-fája*, ha eleget tesz a következő kritériumoknak:

- Pontosan n darab levele van, melyeknek a címkéi rendre $1, .., n$.
- Minden belső csúcsnak a fában legalább két gyereke van.

- A fa élei az S-ben található részszavakkal vannak felcímkézve, és minden k hosszúságú részszó az S-beli kezdő és végpontjával van reprezentálva, ahol $k \geq 2 \cdot \log_2(n - |\Sigma|)$ teljesül.
- Minden belső csúcsból kiinduló él címkéje páronként különböző szimbólumokkal kezdődik.
- Az út a gyökértől az i-edik levélig $s_i \dots s_n$ címkével van ellátva.

Lemma. Legyen S = $s_1 \dots s_n \in \Sigma^n$ egy szó. Az S-hez tartozó kompakt szuffix-fa $O(n \cdot \log_2 n)$ méretű.

Bizonyítás. Minden szuffix-fának pontosan n levele van. Mivel T_S egy kompakt szuffix-fa minden belső csúcsának legalább 2 gyereke van. Ebből következik, hogy T_S-nek legfeljebb $n-1$ belső csúcsa van, így legfeljebb $2n-1$ csúcsa és $2n-2$ éle lehet. Az élek címkéinek hossza megközelítőleg $O(\log_2 n)$, tehát a fa reprezentációjához $O(n \cdot \log_2 n)$ helyre van szükség. □

4.7. Mintaillesztés kompakt szuffix-fa segítségével

Az algoritmus 12 lépésének informális leírása:
Input: $M = m_1 \dots m_m$ minta és $S = s_1 \dots s_n$ szöveg Σ ábécé felett.

1. Felépítjük az S' szöveg $T_{S'} = (V, E)$ kompakt szuffix-fáját, ahol $S' = S\$$ és $\$ \notin \Sigma$.

2. Inicializálás: $x := r$ (Aktuális csúcs, kezdetben a fa gyökere)
$i := 1$ (Aktuális pozíció a mintában)
talált := $false$ (A mintát még nem találtuk meg)
megtalálható := $true$ (A mintát még megtalálhatjuk a szövegben)

3. while not *talált* és megtalálható Keressük azt az élt, mely x csúcsból indul és m_i-vel kezdődik.
egyezés: = $false$ (Még nem találtuk meg)
U:= {x azon gyermekei, melyeket még nem vizsgáltunk}

4. while not *egyezés* és *U* nem üres Választunk egy $v \in U$ csúcsot

5. if *címke'*$(x, v) = m_i\alpha$,ahol $\alpha \in (\Sigma \cup \$)^*$ **then** *egyezés*:= $true$
címke:= *címke'*(x, v)

6. else if *címke'*$(x, v) = (k..l)$ és $s_k = m_i$ then *egyezés*:= *true*
l':= $\min\{l, k + m - i\}$
címke:= $m_k \dots m_{l'}$

7. else U:= U - $\{v\}$

8. Összehasonlítjuk a címkét a már olvasott mintarészlettel. majd a következőket vizsgáljuk

9. if ($m_i, .., m_m$ **és *címke* egyike sem pefixe a másiknak) then**
megtalálható:= *false*

10. else if *címke* prefixe $m_i \dots m_m$**-nek then** x:= v
i:=i + $|$*címke*$|$

11. else $m_i \dots m_m$ **prefixe a címkének then** x:= v
talált:= *true*

12. if *talált* then az x gyökerű részfa leveleit belerakjuk az I halmazba

Output: I halmaz, mely tartalmazza azon pozíciókat, melyektől kezdve a minta megtalálható a szövegben.

Tétel. A fenti algoritmus a Σ ábécé feletti $S = s_1 \dots s_n$ szövegre és $M = m_1 \dots m_m$ mintára $O(n \cdot \log_2 n + m \cdot |\Sigma| + k)$ idő alatt oldja meg a mintaillesztés problémát, ahol a minta k alkalommal fordul elő a szövegben, és $|\Sigma| < \frac{n}{c}$ teljesül minden $c \geq 2$ esetén.

Bizonyítás. A kompakt szuffix-fa generálása $O(n \cdot \log_2 n)$ időt vesz igénybe, ezt már korábban beláttuk.

Az inicializálás (2. lépés) konstans idő alatt lezajlik, így már csak a külső ciklus (3. lépés) műveletigényének meghatározása marad hátra. Első dolgunk annak kiszámolása, hogy mennyi időt vesz igénybe egy csúcs megfelelő gyermekének kiválasztása a belső ciklusban (4. lépés). Tudjuk, hogy egy $\Omega(\log_2 n)$ hosszúságú részszó tömörített címkéjének a hossza minimum $2 \cdot \log_2(n - |\Sigma|)$. A legrosszabb esetben minden aktuális csúcsnak közel $|\Sigma|$ számú gyereke van. Ha a keresés során először a nem tömörített címkéket vizsgáljuk, és utána a tömörítetteket, akkor garantálhatjuk, hogy a tömörített címkék vizsgálatára csak akkor kerül sor, ha a keresett csúcs tömörített. A megfelelő sorrend felállítása $O(|\Sigma|)$ ideig tart. Ezt követően a tömörített címkék olvasása

$$O(\tfrac{m}{\log_2 n} \cdot \log_2 n \cdot |\Sigma|) = O(|\Sigma| \cdot m)$$

idő alatt történik. Minden nem tömörített címke esetén elegendő csak az első szimbólumot olvasni ahhoz, hogy megtaláljuk a megfelelő csúcsot. Minden a gyökérből a csúcsokba vezető út, melyben a hasonlítandó betű megjelenik legfeljebb m élt tartalmaz, így legfeljebb $O(|\Sigma| \cdot m)$ időre van szükség a tömörítetlen csúcsok olvasásához. Ebből következik, hogy ugyanennyi idő kell a megfelelő csúcs kiválasztásához. A minta összehasonlítása a megfelelő ág címkéivel (9–12. lépésig) $O(m)$ időt igényel. Már csak azt kell meghatároznunk, hogy mennyi idő alatt lehet megvizsgálni azt a részfát, melynek levelei tartalmazzák azokat a kezdő pozíciókat, melyektől a minta megjelenik a szövegben. Ennek a részfának pontosan k levele van így legfeljebb 2k-1 csúcsa, mivel minden belső csúcsnak legfeljebb 2 gyereke lehet. Az algoritmusnak ezen része $O(k)$ idő alatt teljesíthető. Összegezve az eredményeket bizonyítottuk, hogy az algoritmus futási idejének becsült értéke

$$O(n \cdot \log_2 n + m \cdot |\Sigma| + k). \square$$

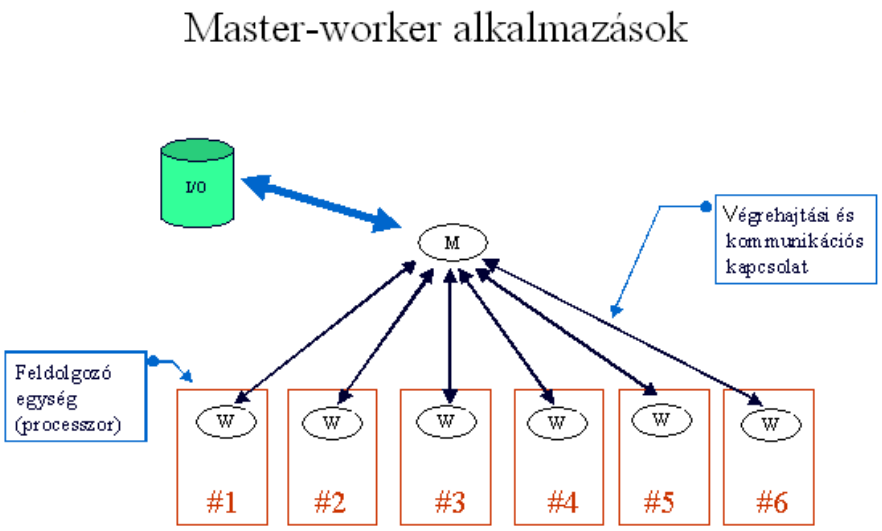
Megjegyzés: Ehhez a fejezethez a [1, 8, 9, 10] forrásokat használtam fel.

4.8. Mintaillesztés párhuzamosításának lehetőségei

Ennek a problémának a megoldása is hatékonyabbá tehető egyszerre több processzor alkalmazásával. Ebben az esetben viszont meg kell terveznünk a processzorok munkamegosztását. Az egyik lehetséges megoldás, hogy a szöveget, melyben a mintát keressük felosztjuk r darab független részre úgy, hogy

$$\lceil \tfrac{n-m+1}{r} \rceil + m - 1$$

legyen az egyes szövegrészletek karakterszáma, és ezek a részletek folytonosan kövessék egymást. Ekkor r darab $m-1$ hosszúságú átfedés alakul ki az egyes részek között. Tehát $r \cdot (m-1)$ redundáns karakter jön létre. Ezt követően mindegyik szövegdarabon függetlenül egymástól végrehajtjuk a mintaillesztés soros algoritmusát. Ennek eredményeként megkapjuk, hogy az egyes részekben mely pozíciótól kezdve jelenik meg a minta. Végül a részszámításokat végző processzorok globális kommunikáció segítségével összegzik az eredményeket. Ennél az eljárásnál fontos kérdés, hogy miként valósítjuk meg a kommunikációt. Erre a legalkalmasabbak az úgynevezett „Master-worker” alkalmazások, melyek felépítését a következő ábra szemlélteti:



Itt egy kitüntetett processzor (M) inicializálja és elindítja a többi a lokális illesztéseket függetlenül végrehajtó processzort. *Statikusnak* nevezzük azt az esetet, amikor ugyanannyi processzorunk van, mint részszavunk, legyen ez most p. Mindegyik egység rendelkezik egy egyértelmű azonosítóval. Azt is feltételezhetjük, hogy az egyes részszavakat a helyi lemezen tároljuk és minden processzor ismeri a mintát. Az eljárás akkor ér véget, amikor az összes lokális mintaillesztés befejeződik, és az eredményeiket összesíti a kitüntetett (M) processzor.

4.8.1. Szimulációs eredmények

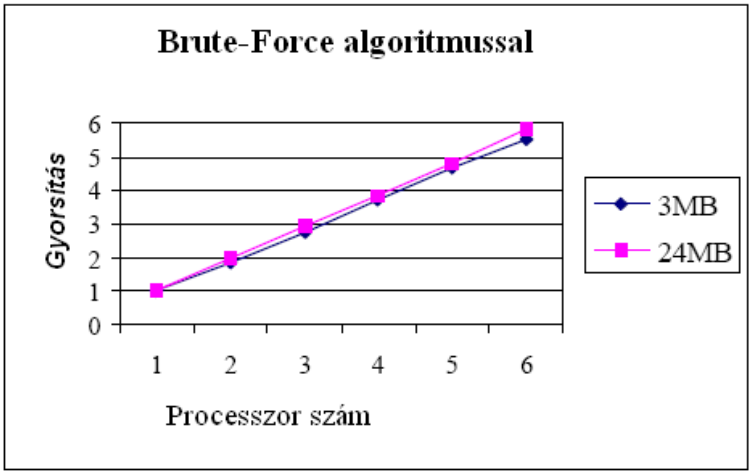
A következőkben összefoglalt szimulációs eredményeket egy 6 személyi számítógépből álló klaszteren futtattam. Külön vizsgáltam az 1, 2, 3, 4, 5 és 6 processzor együttes munkájából adódó eredményeket. Ezek összegzését a lenti táblázatok szemléltetik. Az eredményeket elsősorban a processzorok száma, a minta hossza, és az egyes részszavak hossza befolyásolta. Ezen paraméterek változtatása és variálása képezte a szimuláció alapját. A Brute–Force algoritmust felhasználva a következő eredmények születtek:

P/M	5	10	30	60
1	9.237	8.724	8.513	9.284
2	0.622	0.596	0.584	0.631
3	0.419	0.405	0.396	0.428
4	0.312	0.301	0.296	0.318
5	0.252	0.247	0.242	0.266
6	0.203	0.205	0.197	0.212

Egy 3MB nagyságú szöveghez tartozó futási idők másodpercben megadva, ahol az oszlopok a processzorszámot, a sorok pedig a minták hosszát tartalmazzák.

P/M	5	10	30	60
1	9.237	8.724	8.513	9.284
2	4.642	4.460	4.375	4.769
3	3.095	2.969	2.926	3.201
4	2.334	2.260	2.225	2.421
5	1.875	1.801	1.785	1.962
6	1.558	1.492	1.472	1.631

Egy 24MB nagyságú szöveghez tartozó futási idők másodpercben megadva, ahol az oszlopok a processzorszámot, a sorok pedig a minták hosszát tartalmazzák.



Az ábra a fenti adatok alapján szemlélteti a hatékonyságban elért javulást. Itt gyorsítás alatt az 1 processzoros eset és a több processzoros eset idejének a hányadosát értjük.

A párhuzamosítás egy másik lehetséges módja, hogy a szöveget egyszerre két ablak segítségével vizsgáljuk. Az egyikkel a szöveg elejéről indulunk, a másikkal pedig a szöveg végéről indulunk. Ekkor 2 processzorra van szükségünk.

5. Szekvenciaillesztés (Egyszeres)

Az alapvető mintaillesztési algoritmusok vizsgálata során felvetődik a kérdés, hogy a hibás adatok esetén hogyan lehetne hatékonyan megállapítani, hogy a szóban forgó minta hasonlít-e a szöveg valamely részletéhez. Ehhez meg kell határozni azokat a tulajdonságokat, melyekkel a hasonló biológiai szekvenciák rendelkeznek. A szekvenciaillesztés problémája különösen sokrétű, és több a molekuláris biológiában felmerülő kérdést lehet rá visszavezetni.
Az evolúciós fejlődés során általában már meglévő fajok mutálódtak, így ha a vizsgált szekvenciához találunk egy másik olyan szekvenciát, melynek tulajdonságai és funkciói ismertek, tudunk következtetéseket levonni.
Például fontos szerepet játszik a DNS szekvenálás során létrejövő átfedések vizsgálatánál, vagy két szekvencia közti rokoni kapcsolat megállapításánál.
Az öröklődés során a DNS molekulák még a sejt kettéosztódása előtt duplikálódnak, ekkor a legtöbb esetben az eredeti sejttel megegyező két molekula jön létre. Az utódsejtek mindegyike tartalmazza a teljes genetikai információkészletet. Azonban vannak esetek, amikor a DNS replikálódása során az információk kissé megváltoznak, mutálódnak. Tulajdonképpen ezen a tényen alapul az evolúció. Az évmilliók során folyamatosan jelentek meg új fajok alkalmazkodva a megváltozott környezeti hatásokhoz.
Két szekvencia rokonságának feltárása során azt a mutáció sorozatot akarjuk meghatározni, melynek alapján az egyikből kialakulhatott a másik. Ezt a folyamatot nevezzük evolúciós történetnek. A vizsgálat során feltesszük, hogy az egyes elváltozások egymástól függetlenek, így az egymást követő mutációsorozatok valószínűsége az egyes tagok valószínűségének szorzata. A gyakorlatban vannak olyan változások, melyeknek a valószínűsége nagyobb, mint a többié, és olyanok is, melyek csak nagyon ritkán fordulnak elő. A probléma ésszerű modellezésének érdekében úgynevezett súlyfüggvényeket vezetünk be. A nagy valószínűségű mutációk kisebb, a kis valószínűségű mutációk pedig nagyobb súlyt kapnak. A napjainkban használt algoritmusok legtöbbször az általuk használt függvényben térnek el.
Természetesen az eljárások a biológusok által is alkalmazott minimális törzsfejlő-

dés elvét figyelembe véve működnek.

Definíciók:

1. Legyenek $S = s_1 \dots s_m$ és $T = t_1 \dots t_n$ szavak Σ ábécé felett. $\{-\} \notin \Sigma$. Ez a jel fogja szimbolzálni a réseket.
   Legyen $\Sigma' = \Sigma \cup \{-\}$, és $h : (\Sigma')^* \to \Sigma^*$ egy homomorfizmus, melyre igaz, hogy $\forall a \in \Sigma : h(a) = a$ és $h(-) = \lambda$. S és T egy illesztésén azt az (S', T') Σ' ábécébeli szavakból álló párt értjük, mely tagjainak hossza l $\geq \max\{m, n\}$ és a következő tulajdonságokkal rendelkezik:

   - $|S'| = |T'| \geq \max\{|S|, |T|\}$,
   - $h(S') = S$,
   - $h(T') = T$ és
   - S'-ben és T'-ben nem található ugyanabban a pozícióban rés, azaz $\forall i \in \{1, .., l\} : s'_i \neq -$ vagy $t'_i \neq -$.

   Példa:
   Legyen $S =$ *„GACCGATTATG”* és $T =$ *„GATCGGAATAG”*.
   Egy lehetséges illesztése e két szónak:

   S' = GA–CGGA<u>T</u>TATG
   T' = GATCGGA<u>A</u>TA–G

   Ebben az esetben egy rést láthatunk S-ben a második pozíció után és az utolsó előtti helyen a T-ben, valamint a két aláhúzott betűben különbözik még a két szó. A konvenciók alapján a felső szó az ősi szekvencia, az alsó pedig a leszármazott.
   Két szó illesztését hatékonyan lehet ábrázolni egy $(2 \times l)$ nagyságú mátrix segítségével, melynek a mezői a következők lehetnek:

   - Egy a szimbólum beszúrása egy i pozíció elé. Jelölése: $a \leftarrow^i$
   - Egy a szimbólum törlése egy i pozícióról. Jelölése: $- \leftarrow^i$
   - Egy a szimbólum cseréje b szimbólumra egy i pozícióban.
     Jelölése: $b \leftarrow^i a$

   A transzformációk kompozícióján az egymás utáni végrehajtásukat értjük. Ez azt jelenti, hogy egy evolúciós történet leírható törlések, beszúrások és betűk

cseréjének sorozataként. Természetesen a minimális törzsfejlődés elvének figyelembevételéhez szükségünk van az illesztések kategorizálására, ehhez viszont elengedhetetlen egy súlyozás bevezetése. A lehetséges illesztések száma exponenciálisan nő a szekvenciák hosszával.

2. Legyenek S és T szavak a Σ ábécé felett, $p(a,b) \in \mathbb{Q}\ \forall a,b \in \Sigma$, $g \in \mathbb{Q}$, δ egy súlyfüggvény és cél$_\delta \in \{$ min, max $\}$ a célfüggvény, mely mindig a feladat aktuális paraméterezésétől függ.
   Ekkor
   $\forall$x,y$\in \Sigma$:

   - $\delta(x,y) = p(x,y)$,
   - $\delta(-,y) = \delta(y,-) = g$

   Ezzel páronként megadjuk a betűkre a csere súlyát. Ha feltesszük, hogy

   - $p(a,b) = p(b,a)$,
   - $p(a,a) = 0$ és
   - $p(a,b) + p(b,c) \geq p(a,c)$

   Egy metrikát kapunk, így ebben az esetben csak azokat a transzformációkat kell vizsgálnunk, melyek során egy betű legfeljebb egyszer transzformálódik.
   A fentiek alapján egy $(S^{'}, T^{'})$ illesztés értéke , ahol $S^{'} = s_1^{'}, .., s_l^{'}$ és
   $T^{'} = t_1^{'}, .., t_l^{'}$:

   $$\delta(S^{'}, T^{'}) = \sum_{i=1}^{l} \delta(s_i^{'}, t_i^{'})$$

   A gyakorlatban a távolság meghatározásakor leginkább használt heurisztika a *Levensthein-távolság*, melynél $p(a,b) = 1$, ha $a \neq b$ és $p(a,a) = 0\ \forall a,b \in \Sigma$ esetén, valamint $g = 1$.
   Egy másik gyakran használt úgynevezett *szerkesztési távolság* paraméterezése szerint $p(a,a) = 1$, $p(a,b) = -1$, ha $a \neq b$ és $g = -2$.

3. Legyen S és T két szó a Σ ábécé felett, és δ egy illesztési függvény.
   S és T δ szerinti illesztése ($\delta(S,T)$) akkor és csak akkor *optimális*, ha a hozzájuk rendelt mutációsorozat súlya az optimalizációs függvény szerint optimális, azaz

$$\delta(S,T) = \text{cél}_\delta\{\delta(S',T')|(S',T')\ S \text{ és } T \text{ szavak egy illesztése}\}.$$

Jelöljük $\alpha(S,T)$-vel a két szó optimális illesztéseinek halmazát. A fenti definíciókat és állításokat felhasználva meg tudjuk állapítani két szekvencia hasonlóságát egy optimalizálási feladat megoldásával.

5.1. Globális illesztés (Needleman–Wunsch algoritmus)

Input: S és T szavak a Σ ábécé felett, δ illesztési függvény és a célfüggvény, ami az aktuális paraméterezés alapján vagy maximalizálás, vagy minimalizálás.

A problémát egy dinamikus programozási mátrix segítségével oldjuk meg. Az alap ötlet az, hogy az adott szavak minden prefix párjára kiszámítjuk az optimális értéket.
Legyenek $S = s_1 \dots s_m$ és $T = t_1 \dots t_n$ szavak a Σ ábécé felett.
A „Gyors" algoritmus arra épül, hogy ha ismerjük

$$p(\alpha(S_{i-1}, T_j)),\ p(\alpha(S_i, T_{j-1})) \text{ és } p(\alpha(S_{i-1}, T_{j-1}))$$

értékét, akkor konstans idő alatt ki tudjuk számítani $p(\alpha(S_i, T_j))$ értékét is. Ez egy rekurzív eljárás.
Első lépésként készítünk egy $M(m+1) \times (n+1)$ úgynevezett hasonlósági mátrixot az összes lehetséges prefix vizsgálatához, melynek inicializálásánál az itt következő értékekkel látjuk el a megfelelő mezőket.

- $M_{0,0} = 0$
- $M_{i,0} = i \cdot g\ \forall i \in \{1..m\}$
- $M_{0,j} = j \cdot g\ \forall j \in \{1..n\}$

A mátrix kitöltése az alábbi képlet alapján zajlik:

$$p(\alpha(S_i, T_j) = \text{cél}_\delta\{p(\alpha(S_{i-1}, T_j) + g\ ;\ p(\alpha(S_i, T_{j-1}) + g\ ;\ p(\alpha(S_{i-1}, T_{j-1}) + p(x,y)\}.$$

A mátrix megfelelő elemei adják az egyes optimális értékeket, kitöltése $\Theta(mn)$ időt igényel. Az $M_{i,j}$ elem meg fogja nekünk adni az $s_1 \dots s_i$ és $t_1 \dots t_j$ prefixek hasonlóságát. A végén az $M_{n,m}$ értéktől elindulva minden lépésben

az optimalizációs függvény alapján értéket választva kell eljutnunk az $M_{0,0}$ csúcshoz, és így megkapunk egy optimális illesztést. A következő eljárás ezért a funkcióért felel. A két szó hasonlóságának megállapítása a következő eljárások segítségével történik:

Procedure *MÁTRIX-FELTÖLTÉS*

Input: S és T szavak
Output: M(m,n)

1. **Mátrix feltöltése 0 elemekkel**

```
for i = 0 to m do
    for j = 0 to n do
        M(i,j) := 0
```

2. **Első sor és oszlop inicializálása**

```
for i = 0 to m do
    M(i,0) = i · g
for j = 0 to n do
    M(0,j) = j · g
```

3. **Mátrix kitöltése**

```
for i = 1 to m do
    for j = 1 to n do
        M(i,j) := cél_δ {M(i-1,j) + g,
                         M(i,j-1) + g,
                         M(i-1,j-1) + p(s_i, s_j)}
```

Procedure *ILLESZTÉS (M)*

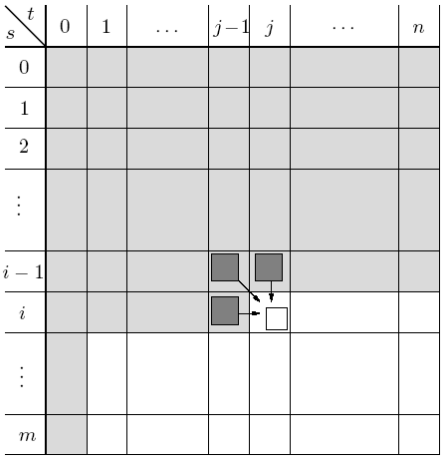
Input: S és T hasonlósági mátrixa (M)
Output: (s',t') illesztése
run Align(m,n)

```
Procedure Align(i,j):
    if i = 0 and j = 0 then
        s' := λ
        t' := λ
    else
        if M(i,j) = M(i-1,j) + g then
            (s̄, t̄) := Align(i-1, j)
            s' := s̄ · s_i
            t' := t̄ · -
        else if M(i,j) = M(i,j-1) + g then
            (s̄, t̄) := Align(i, j-1)
            s' := s̄ · -
            t' := t̄ · t_j
        else {M(i,j) = M(i-1,j-1) + p(s_i, t_j)}
            (s̄, t̄) := Align(i-1, j-1)
            s' := s̄ · s_i
            t' := t̄ · t_j
    return (s', t')
```

Az optimális illesztések száma exponenciálisan nőhet a szekvenciák hosszával. Ez az algoritmus $n + m$ alkalommal hívja meg rekurzívan önmagát. A megfelelő érték kiválasztása konstans időben történik, így az algoritmus futási ideje $O(m + n)$-el becsülhető.

A fenti eljárás bővítésével elérhetjük, hogy az összes optimális illesztés halmazával térjen vissza, azonban vannak esetek, amikor ennek a halmaznak a számossága nagyon nagy is lehet.

Például ha $S = A^{2n}$ és $T = A^n$ szavakat választjuk, ahol $n \in \mathbb{N}$, akkor a lehetséges optimális illesztések száma $\binom{2n}{n}$ és ebben az esetben még nem vettük figyelembe a rések elhelyezkedését. Ebből adódik, hogy egy, az összes optimális illesztést megadó algoritmus futása legrosszabb esetben akár exponenciális időt is igénybe vehet.



Egy dinamikus programozási mátrix kitöltése

Példa:

s \ t		0	A 1	G 2	T 3
	0	0	−2	−4	−6
A	1	−2	1	−1	−3
A	2	−4	−1	0	−2
A	3	−6	−3	−2	−1
T	4	−8	−5	−4	−1

(a)

s \ t		0	A 1	G 2	T 3
	0	0	−2	−4	−6
A	1	−2	1	−1	−3
A	2	−4	−1	0	−2
A	3	−6	−3	−2	−1
T	4	−8	−5	−4	−1

(b)

Az S=„$AAAT$” és T = „AGT” szavak illesztése
$p(a,a) = 1$, $p(a,b) = -1$, ha $a \neq b$ és $g = -2$ paraméterezéssel.
Az a) ábra a már kitöltött mátrixot ábrázolja.
A b) ábra pedig az összes optimális illesztés meghatározásának módját szemlélteti.

Az összes optimális illesztés:

```
AAAT   AAAT   AAAT
-AGT   A-GT   AG-T
```

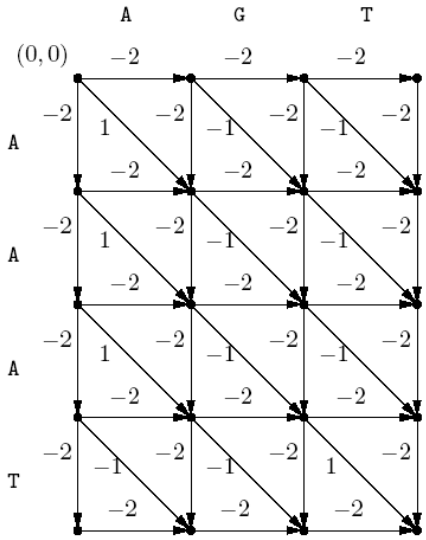
Egy másik hatékony módszer segítségével polinomiális időben megoldható az eredmények ábrázolása. Ez a módszer egy irányított gráfot épít fel, melynek a gyökere az $M_{0,0}$ csúcs és minden élt tartalmaz. Ebben a gráfban az $M_{n,m}$ csúcsból kiinduló és $M_{0,0}$-ba vezető utak egy-egy optimális illesztést szimbolizálnak.

Definíció: Legyenek $S = s_1 \dots s_m$ és $T = t_1 \dots t_n$ szavak egy Σ ábécé felett, és δ egy illesztési függvény $p(a,b)$ és g paraméterekkel, $\forall$ a,b $\in \Sigma$ -ra. A $G_\delta(s,t) = (V,E,c)$ irányított, súlyozott gráf, melyre igazak a következő állítások:

- $V = \{0,..,m\} \times \{0,..,n\}$,

- $E = \{((i,j),(i,j+1))|0 \leq i \leq m \text{ és } 0 \leq j \leq n-1\}$
  $\cup \{((i,j),(i+1,j))|0 \leq i \leq m-1 \text{ és } 0 \leq j \leq n\}$
  $\cup \{((i,j),(i+1,j+1))|0 \leq i \leq m-1 \text{ és } 0 \leq j \leq n-1\}\}$, és
- $c : E \to \mathbb{Q}$,
  c((i,j),(i,j+1)) = c((i,j),(i+1,j)) = g és
  c((i,j),(i+1,j+1)) = p($s_{i+1,t_{j+1}}$).

A már korábban szemléltetett példához tartozó gráf a következő:



A gráf vízszintes élei az S szóban, a függőlegesek pedig a T szóban található réseknek feleltethetők meg. Az átlós élek az aktuális betűk egyezését vizsgálják. Az optimális illesztések meghatározásához már csak annyi dolgunk maradt hátra, hogy megkeressük a $G(0,0)$ csúcsból a $G(m,n)$ csúcsba vezető a célfüggvénynek megfelelő összköltségű utakat. Ehhez számos alapvető algoritmus áll a rendelkezésünkre.

5.2. Lokális és szemiglobális illesztés (Smith–Waterman algoritmus)

A lokális illesztés problémájával találkozunk például amikor egy ismeretlen DNS vagy protein szekvenciát szeretnénk összehasonlítani az adatbázisokban tárolt korábbi adatokkal. Ezeknek a sorozatoknak csak egyes részei egyeznek meg. Ezek az egyezőségek globális illesztéssel nem találhatók meg. Ezt szemlélteti a következő példa:

Legyen $S =$ „$AAAAACTCTCTCT$" és $T =$ „$GCGCGCGCAAAAA$" és a paraméterek a következők: $p(a,a) = 1$, $p(a,b) = -1$, ahol $(a \neq b)$ és
$g = -2$, maximum célfüggvénnyel. Mindkét szekvencia tartalmazza az „$AAAAA$" részszót. A lokális optimális illesztés eredménye:

```
          AAAAA(CTCTCTCT)
(GCGCGCGC)AAAAA
```

Látható, hogy a paraméterezések alapján az illesztés súlya 5.
Ezzel szemben S és T globális illesztése a következő:

```
AAAAACTCTCTCT
GCGCGCGCAAAAA
```

Ebben az esetben az illesztés súlya -11.

Definíció: Legyenek $S = s_1 \dots s_m$ és $T = t_1 \dots t_n$ szavak egy Σ ábécé felett, δ súlyfüggvény és cél_δ az optimalizációs célfüggvény.
A két sztring *lokális illesztésén* azok részszavainak globális illesztését értjük, azaz $\overline{s} = s_{i_1} \dots s_{i_2}$ és $\overline{t} = t_{j_1} \dots t_{j_2}$ részszavak. Egy $A = (\overline{s}', \overline{t}')$ $\overline{s}$ és $\overline{t}$ részszavak illesztése a δ szerinti optimális lokális illesztése S-nek és T-nek, ha a következő teljesül

$$\delta(\text{A}) = \text{cél}_\delta\{\delta(\overline{s}, \overline{t}) | \overline{s} \text{ része az } S, \overline{t} \text{ pedig a } T \text{ sztringnek}\}$$

Definíció: Az *optimális lokális illesztés* megtalálása a következő optimalizációs probléma:

Input: S és T szavak, δ súlyfüggvény és cél_δ célfüggvény.

Lehetséges megoldások: S és T szavak összes lokális illesztése, azaz minden $(\overline{s}, \overline{t})$ globális illesztése.

Illesztés súlya: Minden $A = (\overline{s}, \overline{t})$ lokális illesztés esetén $\delta(A)$.

Optimalizációs cél: Maximálás

A legtöbb esetben a célfüggvény maximálás az optimális eredmény kiszámolásánál, ha viszont a korábban tárgyalt szerkesztési távolság paraméterezését alkalmazzuk, melynek optimalizációs célja minimalizálás az optimális illesztés mindig nagyon

rövid lesz, a legrosszabb esetben csak 1 betű és ez a legtöbb esetben nem ad elfogadható információt.
Az optimális lokális illesztés meghatározásához elegendő a már globális esetben tárgyalt algoritmust módosítani. Itt is egy $M(m+1) \times (n+1)$ méretű hasonlósági mátrixot generálunk, azonban most a szavaknak nem a prefixeit vizsgáljuk, hanem a szuffixeit, és a kitöltésének szabályai is némiképp változnak:

$$M(i,j) = \text{cél}_\delta\{M(i-1,j)+g \ ; \ M(i,j-1)+g \ ; \ M(i-1,j-i)+p(s_i,t_j) \ ; \ 0 \ \}.$$

Végül a mátrixban található cél_δ függvény szerinti érték lesz az optimális illesztés súlya, magát az illesztést pedig úgy kaphatjuk meg, hogy a megtalált maximális érték pozíciójától visszalépkedünk az első 0 értékű pozícióig.
A globális illesztés egy másik alkalmazása az úgynevezett szemiglobális illesztés, mely során a teljes szavakat illesztjük, azonban ha a szavak elején vagy a végén helyezkednek el rések azok nem számítanak bele az illesztés súlyába. Ezt az eljárást főleg akkor alkalmazzák, amikor az összehasonlítandó szavak hossza között nagy eltérés van. Itt is találhatunk olyan példákat, melyeknél az egyik korábbi illesztés sem ad megfelelő eredményt.
Például: $S =$ „$ACTTTATGCCTGCT$” és $T =$ „$ACAGGCT$” a már előbbi példánál is alkalmazott paraméterezéssel.
Az optimális globális illesztés súlya -7.

```
ACTTTATGCCTGCT
AC---A-G---GCT
```

Ha a következő -13 súlyú illesztésben a T elejéről és végéről levágnánk a réseket és így nem számítanánk őket bele a súlyozásba jelentősen javítanánk az értékeket.

```
ACTTTAT-GCCTGCT
-----ACAGGCT---
```

Viszont ezzel az eljárással sem érnénk el az optimális illesztést, mely a következő:

```
(ACTTTATGCCT)GCT
      (ACAGG)GCT
```

Ha belegondolunk szemiglobális illesztés esetén a globálishoz képest csak az M hasonlósági mátrix kitöltése módosul.

Ha az első szó elején található réseket levágjuk, olyan, mintha az M első sorát 0-kal töltenénk fel. Ha ugyanennek a szekvenciának a végét szeljük le az M utolsó sorának maximális értéke megadja az illesztés értékét. Ha a második szó elejét nem vesszük figyelembe az M mátrix első oszlopát töltjük fel 0-kal, ha a végét akkor az utolsó oszlop értékeinek maximuma határozza meg az illesztés súlyát. Ha elhagyjuk a réseket az második szekvencia elejéről valamint az első szekvencia végéről és elvégezzük a globális optimális illesztésnél tárgyalt algoritmust a két szó átfedését közelítő eredményeket kapunk. Ez egy példája a szemiglobális illesztés felhasználásának.

5.3. Büntetőfüggvények

A korábbi fejezetekben csak egyszerű büntetőfüggvényeket alkalmaztunk. Nem vettük figyelembe például, ha egymás után több rés állt, pedig a gyakorlatban gyakran előfordul, hogy egy mutáció komplett blokkokat változtat meg egy szóban. Az eddig tárgyalt algoritmusok ezekben az esetekben túlzottan büntették az egymás utáni réseket. Ez az oka annak, hogy az évek során a tapasztalatok alapján különböző büntetőfüggvények alakultak ki, egyre hatékonyabban tükrözve a valóságban végbemenő folyamatokat.
Célunk továbbra is az, hogy megtaláljuk azt a minimális mutáció sorozatot, mely az egyik szekvenciát a másikba transzformálja. Például ha szeretnénk, hogy a hasonlósági mátrix kitöltésekor ne csak a betűket közvetlenül megelőző részszavak értékeit vegyük figyelembe, a következő módon kell módosítanunk az M mátrix kitöltésének szabályát:

$$M(i,j) = \text{cél}_{\delta}\{\text{cél}_{\delta_{0\le k\le i}} M(k,j) + g_{i-k}\ ;\ \text{cél}_{\delta_{0\le l\le j}} M(i,l) + g_{j-l}\ ;\\ M(i-1,j-i) + p(s_i,t_j)\ ;\ 0\ \}.$$

Ekkor a teljes táblázat kitöltése $\Theta(n+m)$ ideig tart, ebből következik, hogy az optimális illesztés súlyának kiszámításához $\Theta(nm(n+m))$ idő szükséges. Ebből is látszik, hogy ennek a módosult algoritmusnak a futási ideje két közel egyforma szó esetén megközelíti a szavak hosszának a köbét. Ez nem elég gyors algoritmus, de hatékonyabbá tehető más résbüntető függvény alkalmazásával.

Definíció: Egy g résbüntető függvényt *affinnak* hívunk, ha

$$\forall k \in \mathbb{N} : g_k = ku + v, \text{ ahol u,v} \ge 0.$$

A büntető függvények ezen típusának nincs megalapozott biológiai háttere, előnye, hogy hatékonyan átindexelhető, így a Gotoh algoritmus, mely ezt használja gyors. Ennél az eljárásnál az előbbiekhez képest a következő módosításokat hajtjuk végre. A mátrix kitöltésénél használt képlet átindexelése minimalizálás esetén:

$$\begin{aligned} min_{0\leq k\leq i}\{M_{i-k,j}+g_k\} &= \min\{M_{i-1,j}+g_1, \min_{1\leq k\leq i}\{M_{i-k,j}+g_k\} \\ &= \min\{M_{i-1,j}+g_1, \min_{0\leq k\leq i-1}\{M_{i-1-k,j}+g_{k+1}\} \\ &= \min\{M_{i-1,j}+g_1, \min_{0\leq k\leq i-1}\{M_{i-1-k,j}+g_k+u\}. \end{aligned}$$

A számításnál használt másik értékre is elvégezve az eredmény:

$$\min_{0\leq l\leq j}\{M_{i,j-l}+g_l\} = \min\{M_{i,j-1}+g_1, \min_{0\leq l\leq j-1}\{M_{i,j-l-1}+g_l+u\}.$$

Definíció: Egy g résbüntető függvényt *konkávnak* nevezünk, ha

$$\forall i\in\mathbb{N}: g_{i+1}-g_i\leq g_i-g_{i-1}.$$

Tehát mennél nagyobb egy rés, annál kevésbé büntetjük. A negatív súlyok elkerülésének érdekében ki szoktuk kötni, hogy a függvény szigorúan monoton növekvő. Ehhez a típusú büntetőfüggvényhez létezik $O(n\cdot m(log_2 n+log_2 m))$ futási idejű algoritmus. *Megjegyzés:* A [2] forrást használtam fel.

5.4. Szubsztituciós mátrixok

Szekvenciaillesztés során gyakran találkozhatunk olyan aminosavakkal, melyek jobban hasonlítanak egymásra funkcionalitásban és szerkezetben is, mint bármely másikra. Ha egy mutáció során egy aminosav egy olyan másik aminosavvá alakul át, melynek funkciói és tulajdonságai nagyon eltérnek az övétől, azt is eredményezheti, hogy a protein elveszti biológiai funkcióját. Ebben az esetben kicsi az esélye annak, hogy átöröklődik a következő generációra.
Ahhoz, hogy mérni tudjunk két szekvencia közti különbséget meg kell határoznunk minden egyes lehetséges csere értékét, azaz a korábbi jelöléseket alkalmazva p(a,b)-t. Ezen értékeket szintén egy $(|\Sigma|)\times(|\Sigma|)$ méretű úgynevezett szubsztituciós, vagy mutációs gyakoriságok alapján definiált mátrixban tároljuk, ahol Σ az az ábécé, mely felett a szekvenciák értelmezve vannak. Például két protein összehasonlítása során egy $(20)\times(20)$ nagyságú mátixot használunk.
A következőkben részletezem ezek típusait, felépítését és a generálásukat végző eljárásokat.

5.4.1. PAM mátrix

Definíció: Egy *mutációt jóváhagyottnak* tekintünk, ha az ős szekvenciát csak annyira változtatja meg, hogy annak funkciója csak kis mértékben módosul, így öröklődhet a következő generációra.

Definíció: *APM = PAM = Accepted Point Mutation* a szelekció által jóváhagyott mutáció, és a szekvenciák közti távolság mérésére szolgáló egység. S és T protein szekvenciák *1 PAM távolságra vannak* egymástól, ha egy jóváhagyott mutáció sorozat során, mely nem tartalmaz sem törlést, sem beszúrást átlagosan 100 aminosavanként történik 1 helyen cserélődés.

Logikusan következik, hogy a k PAM távolságú proteinek esetében átlagosan k helyen történik mutáció 100 aminosavanként. A fenti definíció elméleti, a gyakorlatban egy pozíciót egymás után több mutáció is megváltoztathat. Ettől függetlenül ez a technika jól hasznosítható k PAM távolságú protein szekvenciák összehasonlításánál.

PAM mátrixok generálása:

Először tisztáznunk kell a homológ protein szekvenciák definícióját.
Két protein sorozat *homológ*, ha a bennük található megfelelő proteinek funkciója megegyezik. Tegyük fel, hogy ismerünk homológ protein párokat, melyekről tudjuk, hogy k PAM távolságúak. Mivel a távolságot a cserék számával jellemeztük fel kell tennünk, hogy ismerjük a szekvenciák páronkénti optimális illesztését, azaz a rések pontos helyzetét.
Legyen A halmaz a vizsgált sorozatpárok összes lehetséges illesztésének a halmaza, valamint $Col(A)$ egy halmaz, melyben azon A-beli elemek találhatók, melyek nem tartalmaznak rést. Ezt követően minden aminosav szimbólumpárra meghatározzuk a relatív gyakoriságát:

$$freq(a_i, a_j) = \frac{|(a_i,a_j)|+|(a_j,a_i)|:(a_i,a_j),(a_j,a_i)\in Col(A)}{2\cdot|\,Col(A)|}.$$

Az egyes aminosavakhoz is definiálhatunk relatív gyakoriságot azok előfordulásának száma és a szekvenciák teljes hosszának hányadosaként. Ezeket a definíciókat felhasználva le tudjuk írni a PAM_1 mátrix egyes elemeinek előállításához szükséges képletet:

$$\mathrm{PAM}_1 = \log_2 \frac{freq(a_i,a_j)}{freq(a_i)\cdot freq(a_j)}.$$

Tehát először meghatározzuk annak valószínűségét, hogy a_i a_j szimbólummá változik egy mutáció során, majd ezt az értéket elosztjuk annak a valószínűségével, hogy a_i és a_j egyszerre jelenik meg egy illesztésben. A logaritmus alkalmazásával leegyszerűsíthető a kifejezés, hisz így az egyes illesztések szorzata helyett elég azok logaritmusának az összegét venni. A gyakorlatban a mátrix egyes elemeit 10-zel beszorozzák és ezt követően még kerekítik is azokat.

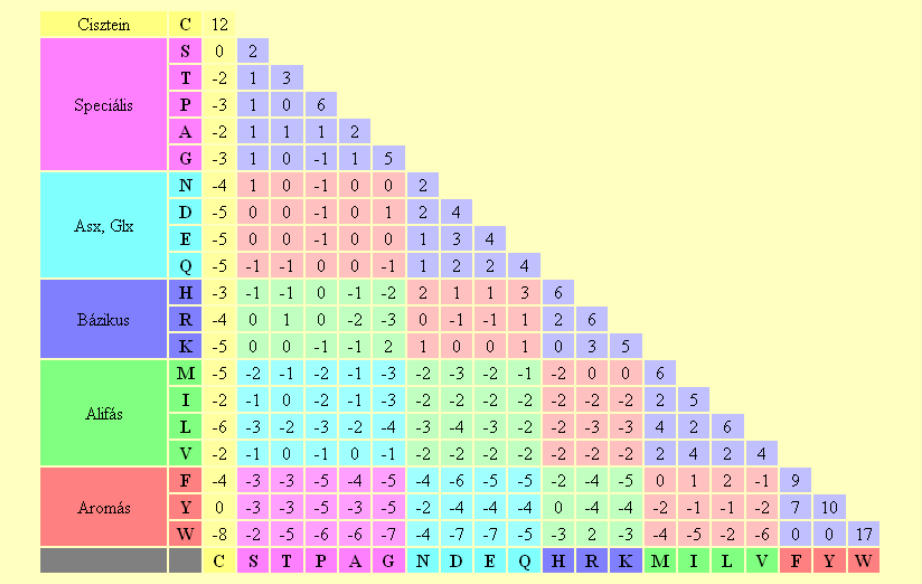
A Margaret Dayhoff és kollégái által kidolgozott eljárások először olyan szekvenciákat vizsgálnak, melyek közös őstől származnak és csak 1 PAM távolságra vannak egymástól. Ezek az összerendezett fehérjék minimum 85%-ban azonosak. A nagy mértékű hasonlóság miatt még a hosszúságuk is közel egyforma, így könnyű meghatározni az optimális illesztésekben található rések kezdő pozícióját. Ezen adatok ismeretében már fel tudunk építeni egy 1 PAM mátrixot.

A PAM_k mátrixok előállításához a PAM_1 mátrixon kívül szükségünk van még egy $F(20\times20)$ mátrixra, melynek $F(i,j)$ eleme tárolja, hogy mennyi esély van arra, hogy a_i szimbólum a_j-re változik függetlenül az aktuális előfordulási értékektől. F hatványozásával megkaphatjuk a k-PAM távolságú szekvenciákban lezajló cserék valószínűségét. Felhasználva ezeket az adatokat a következő módon tudjuk előállítani a PAM_k mátrixokat:

$$\mathrm{PAM}_k = \log_2 \frac{freq(a_i)\cdot F^k(i,j)}{freq(a_i)\cdot freq(a_j)} = \log_2 \frac{F^k(i,j)}{freq(a_j)}.$$

Ezzel az eljárással egy szimmetrikus mátrixot kapunk. A gyakorlatban a $k = \{40,\ 100,\ 250\ \}$ nagyságú mátrixok használata elterjedt.

Probléma: Ahhoz, hogy tudjuk melyik PAM mátrixot kell használnunk az adott esetben, ismernünk kell a szekvenciaazonosság mértékét. Ezt viszont csak az összerendezés után tudjuk meg pontosan, amihez a PAM mátrixot szeretnénk felhasználni.



Egy PAM_{250} mátrix, melyben az aminosavak tulajdonságaik szerint vannak csoportosítva

5.4.2. BLOSUM mátrix

A BLOSUM mátrix a szubsztituciós mátrixok másik típusa, mellyel gyakrabban találkozhatunk a gyakorlatban, mint a PAM mátrixokkal, mert távoli homológok vizsgálatára is alkalmas, így nagyobb adatkészletet képes feldolgozni és biológiailag helyesebb eredményeket ad. Feltöltéséhez úgynevezett BLOCKS adatbázis szükséges, mely fehérjecsaládok többszörösen összerendezett szekvenciablokkjait tartalmazza. A szekvenciákat klaszterekre bontják. Azok a szavak kerülnek egy csoportba, melyek hasonlósága nagyobb, mint 62%. Az itt található információ többszörös szevenciaillesztéssel is megkapható. A klaszterekre jellemző, hogy rövid, folytonos, résmentes szekvencia részleteket tartalmaznak, melyek egy szekvenciaillesztés darabjai. Méretük alsó határa meg van szabva. Egy teljes szekvenciaillesztést mátrix segítségével lehet ábrázolni. A blokkok egyes sorai függetlenek is lehetnek egymástól. A feltöltéshez megadjuk az összes összehasonlítandó szekvenciapár halmazát, ezt követően kiszámoljuk az összes aminosav relatív gyakoriságát, azaz $freq(a_i)$-t és a $freq(a_i, a_j)$ értéket is minden elemre, ahol (a_i, a_j) egy halmazbeli aminosav pár. Ekkor a mátrix kitöltésének képlete:

$$\text{BLOSSUM}_x(i,j) = 2 \cdot \log_2 \frac{freq(a_i,a_j)}{freq(a_i \cdot freq(a_j))}.$$

Ezt az értéket is kerekíteni szokták.

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W	
C	9																				C
S	-1	4																			S
T	-1	1	5																		T
P	-3	-1	-1	7																	P
A	0	1	0	-1	4																A
G	-3	0	-2	-2	0	6															G
N	-3	1	0	-2	-2	0	6														N
D	-3	0	-1	-1	-2	-1	1	6													D
E	-4	0	-1	-1	-1	-2	0	2	5												E
Q	-3	0	-1	-1	-1	-2	0	0	2	5											Q
H	-3	-1	-2	-2	-2	-2	1	-1	0	0	8										H
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	5									R
K	-3	0	-1	-1	-1	-2	0	-1	1	1	-1	2	5								K
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	5							M
I	-1	-2	-1	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	4						I
L	-1	-2	-1	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	4					L
V	-1	-2	0	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	4				V
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	6			F
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	7		Y
W	-2	-3	-2	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	11	W

Egy BLOSSUM62 mátrix

5.5. Heurisztikus keresési módszerek

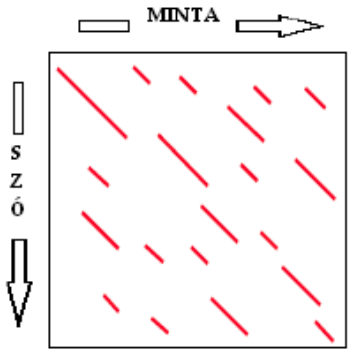
A már korábban bemutatott polinomiális futási idejű illesztési algoritmusok (mint például a Needleman–Wunsch vagy Smith–Waterman algoritmus) sajnos nem bizonyulnak elég hatékonynak a nagy méretű DNS vagy protein adatbázisokban való keresés során. Ez az oka annak, hogy a gyakorlatban elsősorban heurisztikus eljárásokat alkalmaznak, melyek ugyan nem garantálják az optimális megoldást, de lényegesen gyorsabbak.
Alapvetően két programcsalád terjedt el, melyek a legtöbb esetben jól hasznosítható, közel optimális eredményt adnak.

5.5.1. FASTA heurisztika

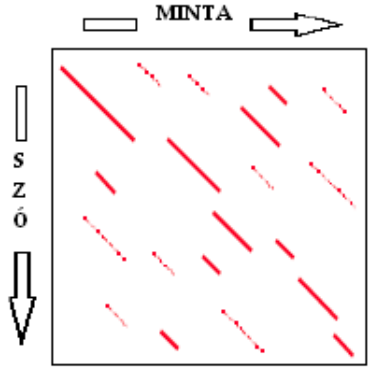
A FASTA elnevezés a *„fast all”* angol kifejezésből ered, mely arra utal, hogy ez a program DNS és protein szekvenciákon is alkalmazható, ellentétben az elődjével, amit FASTP-nek hívnak és csak proteineken működik. Az itt tárgyalt programok folyamatosan fejlesztés alatt állnak és több variációjuk is létezik. Ez az eljárás az általunk keresett mintát összehasonlítja az adatbázisban tárolt összes szekvenciával. A futása során a következő 4 lépés történik:

1. Meghatározzuk a k paramétert, mely megadja, hogy milyen hosszúságú részszavakat keresünk a mintában és az adatbázisbeli szekvenciákban egyaránt. Az egyező részeket *„hot spot”*-nak nevezzük és a két szekvenciabeli kezdő pozíciójukból alkotott párral azonosítjuk őket. DNS szekvenálás során a tapaszta-

latok alapján kialakult ideális k érték 6, proteineknél pedig 2. Ebben a lépésben hasznosíthatjuk a már tárgyalt mintaillesztő algoritmusokat, különösen azokat, melyek előfeldolgozó eljárással teszik hatékonyabbá a kereséseket. Ilyen például a Boyer–Moore algoritmus.

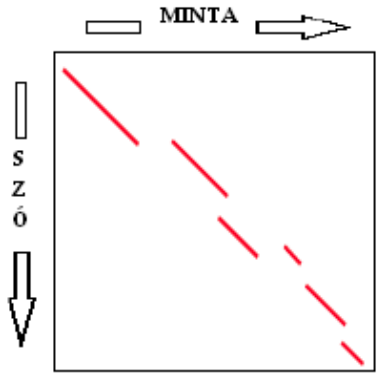


2. A kapott hot spot részeket megpróbáljuk csoportosítani. Ehhez egy a globális illesztésnél már alkalmazott hasonlósági mátrixot használunk, melyben a sorokhoz a minta (M) az oszlopokhoz pedig az adatbázisbeli szó (S) tartozik, és $M(i,j) = 1$, ha $m_i = s_j$ különben pedig 0. Ebből adódik, hogy minden megegyező rész az átlókon helyezkedik el. Mi csak azokat a diagonál részleteket vizsgáljuk, melyek hot spot részletekkel kezdődnek és végződnek. Ezeket súlyozzuk, mégpedig úgy, hogy minden egyező rész növeli, és minden köztük található rés annak hosszától függően csökkenti a súlyt. Ez a lépés hatékonyan végrehajtható
$O(|\text{hot spot}|)$ idő alatt. Ekkor részszó párokat kapunk, melyekre az algoritmus optimális lokális illesztést ad a már korábban részletezett eljárások segítségével.

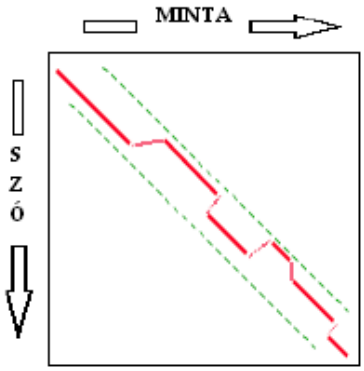


A vastagabb vonalak jelzik a magas súlyú illesztéseket

3. Ha a részszavak illesztése során azok súlya meghaladja a határértéket, az algoritmus megpróbálja kibővíteni azokat, hogy az értékeiket a megfelelő korlátok közé szorítsa. Ez a feladat leginkább gráfelméleti problémaként szemléltethető. Tekintsük a 2. lépésben meghatározott illesztéseket egy gráf csúcsainak, a hozzájuk tartozó értkekkel pedig címkézzük fel a gráf éleit. Legyen u és v egy-egy illesztés, u az (i, j) pontban ér véget, v pedig az (i', j') csúcsnál kezdődik. Irányított él akkor, és csak akkor található a két csúcs között, ha $i < i'$ és $j < j'$ teljesül, azaz, ha mindkét illesztés egy másik, hosszabbhoz csatlakozik. Az u-t és v-t összekötő él egy rést szimbolizál, így súlya negatív. Ekkor az eddigi illesztések által alkotott utak közül az optimálisak egy-egy optimális illesztést határoznak meg.



4. A 2. lépésben meghatározott lokális optimális illesztést tartalmazó, korlátos méretű blokkra lefuttatja a Smith–Waterman algoritmust. Így egy alternatív megoldást is nyújt a problémára.



Az eljárás ezt a 4 lépést végzi el az összes adatbázisban található szekvenciára. Tehát a FASTA metódus statisztikai adatok alapján becsüli meg a lehetséges szig-

nifikáns megoldásokat, de a legmagasabb súlyú illesztések közül csak az egyiket adja meg.

5.5.2. BLAST heurisztika

A BLAST az egyik legszélesebb körben elterjedt programcsalád. Ezt gyors futási idejének köszönheti. Célja egy olyan összefüggő DNA és protein szekvenciák hasonlóságát vizsgáló algoritmus megalkotása, mely gyorsabb, mint az előzőekben használt FASTA algoritmusok, viszont legalább olyan hatékony, mint azok. A korábbi eljárásokhoz nagy számítási teljesítményre volt szükség a megfelelő eredmények eléréséhez, gondoljunk csak a dinamikus programozás hardver követelményeire, főleg nagy adatmennyiségek feldolgozása során. Emiatt is az egyik legfontosabb szempont a használni kívánt algoritmus kiválasztásakor a hatékony tárolás, alacsony memóriafoglalási igény és a gyors futási idő.
A most vizsgált algoritmus két komponensből áll. Egy kereső algoritmusból és egy statisztikai becsléseket tartalmazó mátrixból, mely általában a már korábban tárgyalt PAM vagy BLOSUM mátrix.
Az eljárás lépései:

1. Az algoritmus adott hosszúságú (w) hasonló részszavakat keres a mintában és az éppen vizsgált adatbázisbeli szóban. A gyakorlatban tipikusan használt paraméter a DNS szekvenciáknál $w = 11$ és a protein szekvenciák esetében $w = 3$. Ellentétben a FASTA eljárással ez a metódus nemcsak azonos w hosszúságú részszavakat keres, hanem olyan rést nem tartalmazó lokális illesztéseket is vizsgál, melyek értékei meghaladják a határértéket, amit a szubsztituciós mátrix határoz meg. Alkalmas adatstruktúrák felhasználásával ez a lépés lineáris idő alatt elvégezhető.

2. Olyan hasonló részszó párokat keresünk, melyek legfeljebb d távolságra vannak egymástól. Azokat a párokat, melyek nem felelnek meg ennek a követelménynek az algoritmus figyelmen kívül hagyja. A d paraméter értéke mindig a w értékétől függ, például $w = 2$ esetén általában $d = 16$ értékkel szoktak számolni.

3. Megpróbáljuk kiterjeszteni mindkét irányba a talált párokat úgy, hogy mindkét végpontjukhoz további illesztéseket csatolunk egészen addig, amíg már az illesztés súlya nem nő tovább. Az eredeti BLAST eljárás nem enged

meg réseket, de a későbbi verziók már ezt a problémát is kiküszöbölik. Ezt követően a határérték alatti pontozással rendelkező párokat (melyeket *HSP = High Scoring Pair*-nek nevezzük) csökkenő sorrendben adja meg az algoritmus.

Ez az eljárás több szempontból is hatékonyabb, mint a FASTA. Megadja az összes legjobb részleges illeszkedést, és kezelni tudja a réseket is. Egy párhuzamos változatot is fejlesztettek már, mely az adatbázis szegmentálásával, párhuzamos lekérdezésekkel és összehasonlításokkal éri el a még gyorsabb működést. A BLAST algoritmust megvalósították FPGA (*Field Programmable Gate Array*) áramkörrel is, mely még gyorsabb, mint a szoftveres implenetáció. Az algoritmus felhasználhatósága is széleskörű. Találkozhatunk vele DNA és protein szekvenciák adatbáisbeli vizsgálatánál, motívum kereséseknél, gén azonosításoknál és többszörös régiók hosszú DNA szekvenciákban történő keresésénél is. Ennek megfelelően a BLAST mára programcsomaggá fejlődött, mely a következő komponensekből áll:

BLAST program	Szekvencia	Adatbázis
BLASTP	Fehérje	Fehérje
BLASTN	Nukleotid	Nukleotid
BLASTX	6 keretben lefordított nukleotid	Fehérje
TBLASTN	Fehérje	6 keretben lefordított nukleotid
TBLASTX	6 keretben lefordított nukleotid	6 keretben lefordított nukleotid

6. Többszörös szekvenciaillesztés

A korábbi fejezetekben tárgyalt algoritmusok csak két szekvencia illesztését tették lehetővé. Most azt vizsgáljuk, hogy mely eljárásokkal lehet a lehető leghatékonyabban elvégezni több szekvencia összehasonlítását. Ennek a feladatnak az elvégzése algoritmikailag már lényegesen bonyolultabb, mint a korábbiaké. Napjainkban a bioinformatika egyik központi feladatának tekinthetjük. Alkalmazhatjuk szekvenciák adatbázisokban történő keresésére, hasonlóságaik vagy különbségeik mértékének szemléltetésére, evolúciós leszármazások vizsgálatára, egy szekvencia család konzervatív régióinak megkeresésére (azokat a pozíciókat kutatjuk, melyek az adott csoport funkcionális tulajdonságait kialakítják).

> „Amit két homológ szekvencia suttog, azt egy többszörös illesztés hangosan kiáltja"
>
> Arthur Lesk

Definíció: Legyen $s_1 = s_{11} \dots s_{1m_1}, .., s_k = s_{k1} \dots s_{km_k}$ k darab sztring egy Σ ábécé felett, „–" $\notin \Sigma$. Továbbá legyen $\Sigma' = \Sigma \cup \{-\}$ és $h{:}(\Sigma')^* \to \Sigma^*$ egy homomorfizmus, melyre $h(a) = a$ $\forall a \in \Sigma$ esetén és $h(-) = \lambda$.
Az $s_1 \dots s_k$ szavak *többszörös illesztését* egymás alá írt k-asokként $(s_1', .., s_k')$ lehet modellezni. Ezeket *illesztett k-asoknak* nevezzük. Az alkotók hossza $l \geq \max\{m_i | 1 \leq i \geq k\}$ és a következő követelményeknek kell teljesülnie:

1. $|s_1'| = |s_2'| = \dots = |s_k'|$,
2. $h(s_i')= s_i$ $\forall i \in \{1, .., k\}$ és
3. nem létezik olyan pozíció, melyben
   $\forall j \in \{1, .., l\} : \exists i \in \{1, .., k\} : s_{i,j}' =$ „–".

A második tulajdonság informálisan azt jelenti, hogy az s_i' szóból kitörölve a réseket az s_i szóból kapjuk vissza.
A következő fontos kérdés, hogy hogyan súlyozzuk az illesztéseket? A páronkénti illesztés algoritmusának egyszerű általánosításával könnyen kaphatunk egy megoldást.

Azt is tisztáznunk kell, hogy hogyan definiáljuk egy többszörös illesztés súlyozását. Erre két lehetőségünk is van. Az első megoldáshoz definiálnunk kell egy többszörös szekvenciaillesztés konszenzusát. Ennek előállítása informálisan leírva

úgy történik, hogy minden oszlopból kiválasztjuk a leggyakrabban előforduló szimbólumot. A többszörös illesztés és konszenzusának távolsága megegyezik azoknak a szimbólumoknak a számával, melyekben különböznek.

Formális definíció: Legyen $(s_1^{'}, .., s_k^{'})$ egy többszörös szekvenciaillesztése az $s_1, .., s_k \in \Sigma^*$ szavaknak, és legyen $l = |s_1^{'}|$ az illesztés hossza.
Ekkor a $c = c_1 \dots c_l \in \Sigma^l$ szó a *konszenzusa* $(s_1^{'}, .., s_k^{'})$-nek, ha

$$c_j = \max_{a \in \Sigma} |\{s_{ij}^{'} = a | 1 \leq i \leq k\}| \ \forall \ 1 \leq i \leq l.$$

Egy c konszenus és egy $(s_1^{'}, .., s_k^{'})$ illesztés közti *távolságon* a következőt értjük:

$$d(c, (s_1^{'}, .., s_k^{'})) = \sum_{j=1}^{l} |\{s_{ij}^{'}| \ 1 \leq i \leq k, s_{ij}^{'} \neq c_j\}|.$$

Két ugyanazon többszörös illesztéshez tartozó konszenzus kapcsolatát a következő lemma írja le.

Lemma. Legyen $(s_1^{'}, .., s_k^{'})$ egy többszörös szekvenciaillesztés, és c, valamint $\overline{c}$ a hozzá tartozó konszenzusok. Ekkor

$$d(c, (s_1^{'}, .., s_k^{'})) = d(\overline{c}, (s_1^{'}, .., s_k^{'})).$$

Bizonyítás. Vegyük az illesztés egy j oszlopát. Ha $c_j \neq \overline{c}_j$ teljesül az azt jelenti, hogy c_j ugyanannyiszor fordul elő a $j - dik$ oszlopban, mint $\overline{c}_j$ Ebből az következik, hogy

$$|\{s_{ij}^{'} \mid 1 \leq i \leq k, \ s_{ij}^{'} \neq c_j\}| = |\{s_{ij}^{'} \mid 1 \leq i \leq k, \ s_{ij}^{'} \neq \overline{c}_j\}|.$$

Ez az egyenlőség az illesztés minden oszlopára igaz, így bizonyítottuk a lemmát. □

Mivel egy többszörös illesztés és egy konszenzus távolsága független az általunk választott konszenzustól, a távolságot tetszőleges konszenzus segítségével definiálhatjuk.
Ez a távolság fogalom hatékonyan felhasználható a már korábban említett súlyozás definiálásához, hiszen mennél kisebb a távolság annál jobb az illesztés. A következő példa egy konszenzus meghatározását szemlélteti.

$$\begin{aligned} s_1' &= \texttt{AATGCT} \\ s_2' &= \texttt{A-TTC-} \\ s_3' &= \texttt{---TCC} \\ \hline c &= \texttt{AATTCT} \end{aligned}$$

$$d(c,(s_1',s_2',s_3')) = 1+2+1+1+0+2 = 7.$$

Most már minden adott a formális definícióhoz.

Formális definíció: A többszörös szekvenciaillesztés minimális távolságú konszenzusok alapján a következő optimalizálási probléma:

Input: Egy $S = s_1,..,s_k$ Σ ábécé feletti szavakat tartalmazó halmaz.

Lehetséges megoldások: S minden többszörös illesztése.

Költségfüggvény: $d(c,(s_1',..,s_k'))$.

Optimalizációs cél: Minimalizálás.

Itt kell megjegyeznünk, hogy a fent leírt konszenzus definíció csak egy a sok közül, ezt nevezik *„Többségi szavazás"*-nak. A kapcsolódó irodalomban gyakran megfigyelhető, hogy a konszenzust bővebb értelemben tekintik, ugyanis azokat a szavakat is hozzá sorolják, melyek többszörös illesztésből származnak. A súlyozás definiálásának másik módja a probléma páronkénti illesztésre való leredukálásán alapul.

Definíció: Legyen Σ egy ábécé, $- \notin \Sigma$ a rések szimbóluma. Legyen δ egy páronkénti illesztés súlyfüggvénye a Σ ábécé felett, kiegészítve $\delta(-,-)$ definíciójával. Az optimalizációs cél legyen most minimalizálás. Egy l hosszúságú $(s_1',..,s_k')$ többszörös illesztés súlyozásához (δ_{TI}) először megadjuk egy oszlop súlyának kiszámolási módját.

$$\delta_{TI}(s_{1j}',..,s_{kj}') = \sum_{i=1}^{k}\sum_{p=i+1}^{k} \delta(s_{ij}',s_{pj}') \; \forall 1 \le j \le l.$$

Ezt felhasználva a teljes illesztés súlyán a következő összeget értjük:

$$\delta_{TI}(s_1',..,s_k') = \sum_{j=1}^{l} \delta_{TI}(s_{1j}',..,s_{kj}').$$

A következő példa ezt a módszert szemlélteti. Itt $\delta(a,b) = 0$, ha $a = b$ és $\delta(a,b) = 1$, ha $a \neq b$ $\forall a,b \in \{A,C,G,T,-\}$. A korábbi példára alkalmazva ezt a módszert a következőket kapjuk:

$$s'_1 = \texttt{AATGCT}$$
$$s'_2 = \texttt{A-TTC-}$$
$$s'_3 = \texttt{---TCC}$$

Ebben az esetben

$$\delta_{TI}(s'_1, s'_2, s'_3) = \sum_{j=1}^{6} \sum_{i=1}^{3} \sum_{p=i+1}^{3} \delta_{TI}(s'_{ij}, s'_{pj})$$
$$= \sum_{j=1}^{6} (\delta_{TI}(s'_{1j}, s'_{2j}) + \delta_{TI}(s'_{1j}, s'_{3j}) + \delta_{TI}(s'_{2j}, s'_{3j}))$$
$$= (0+1+1) + (1+1+0) + (0+1+1)$$
$$+(1+1+0) + (0+0+0) + (1+1+1) = 11.$$

Formális definíció: A többszörös illesztés problémája optimális δ_{TI} súlyozással a következő optimalizálási feladat:

Input: Egy $S = s_1, .., s_k$ Σ ábécé feletti szavakat tartalmazó halmaz.

Lehetséges megoldások: S minden többszörös illesztése.

Költségfüggvény: $\delta_{TI}(s'_1, .., s'_k)$.

Optimalizációs cél: Minimalizálás.

6.1. A többszörös illesztés pontos előállítása

A többszörös illesztés előállításához ismét a dinamikus programozás eszközeit hívjuk segítségül, melyekkel már korábban az egyszeres illesztésnél találkoztunk. Adott k darab sztring, $(s_1, .., s_k)$, ezeket szeretnénk illeszteni. Ismét egy M k dimenziós tömböt használunk, melyben a prefixek optimális többszörös szekvenciaillesztésének értékeit tároljuk.
Sajnos ezen eljárás használatakor nehézségek léphetnek fel, ha a k paraméter elég nagy. A másik jelentős probléma, hogy k értéke része a bemenő adatoknak és ennek a programnak az idő és tárigénye exponenciálisan függ ettől a paramétertől. Ahhoz, hogy megállapítsuk a kitöltéshez szükséges minimális értéket $2^k - 1$ érték közül kell kiválasztani a minimumot. A $k = 2$ esetben láttuk, hogy 3 érték közül kellett választani aszerint, hogy a beszúrás, törlés vagy a csere műveletéből származó érték volt a legkisebb, vagy ha az optimalizációs cél úgy kívánja a legnagyobb.
A többszörös illesztés esetében minden oszlopban figyelembe kell vennünk a rések elhelyezkedésének összes lehetséges kombinációját. Ez minden esetben bonyolulttá teszi az egyes értékek kiszámítását is. Például, ha figyelembe veszzük a már tárgyalt δ_{TI} súlyozást, akkor is a futási idő nagyságrendileg k^2 lesz. Ha feltesszük, hogy

minden bemenő szó hossza azonos (n), akkor n^k eleme lesz M-nek és a feltöltése $O(k^2 \cdot 2^k \cdot n^k)$ időbe telik. Ebből következik, hogy a dinamikus programozási eszközöket használó eljárások ebben az esetben csak nagyon kevés, szinte csak maximum $3-4$ bemenő sztringre esetén lesz hatékony. Alapvetően nem tudunk jobb eljárást, mely pontosan meg tudja határozni a többszörös illesztést.
Sőt ez a probléma NP-teljes, ha a bemenő szavak száma (k) része az inputnak. Az algoritmusok bonyolultságát tárgyaló fejezetben meghatároztuk az NP-teljes probléma definícióját. Tulajdonképpen minden probléma, amely csak nemdeterminisztikus Turing-géppel oldható meg ebbe az osztályba tartozik. Tehát, ha be akarjuk bizonyítani, hogy a többszörös illesztés az NP osztályba tartozik definiálnunk kell egy döntési eljárást.

Definíció: A többszörös illesztés problémához tartozó δ_{TI} *súlyozást alkalmazó döntési eljárás* a következő:

Input: Egy k pozitív szám , $S = s_1, .., s_k$ halmaz, mely Σ ábécébeli szavakat tartalmaz, $\delta : (\Sigma \cup \{ - \})^2 \longrightarrow \mathbb{Q}$ súlyfüggvény és egy d pozitív szám.

Output: *„Igaz”*, ha az S halmazbeli szavaknak létezik olyan többszörös illesztésük, melynek δ_{TI} szerinti súlya kisebb, vagy egyenlő, mint d. Különben a visszatérési érték *„Hamis”*.

Definíció: Az $S = s_1, .., s_k$ halmazban található szavakból képzett *szupersztring* egy sztring, mely $\forall i \in \{1, .., k\}$ esetén s_i -t részszóként tartalmazza. A legrövidebb közös $\Sigma = \{0, 1\}$ ábécé feletti szupersztring meghatározásának problémájához tartozó döntési eljárás a következő:

Input: Egy k pozitív szám, $S = s_1, .., s_k$ halmaz, mely $\Sigma = \{0, 1\}$ ábécébeli szavakat tartalmaz, és egy m pozitív szám.

Output: *„Igaz”*, ha létezik t az S halmazbeli elemekből képzett legrövidebb közös szupersztring, melynek hossza kisebb, mint m. Különben a visszatérési érték *„Hamis”*.

Lemma. A legrövidebb közös szupersztring meghatározásának problémája NP-teljes.

Erre a problémára is több bioinformatikai kérdést vezethetünk vissza. A megoldására számos közelítő algoritmus született. Egy mohó algoritmus minden sztringpárra

megkeresi a maximális átfedéseket, majd ezeket próbálja összefűzni egy legrövidebb szupersztringgé. Az algoritmus futási ideje $O(nm)$, ahol n a szekvenciák száma, m pedig a szekvenciák összhossza. Az így megtalált szupersztring mérete bizonyítottan kisebb, mint a legrövidebb szupersztring hosszának négyszerese.

Mint már korábban láttuk a többszörös szekvenciaillesztés dinamikus programozású módszerekkel történő végrehajtása nem hatékony, különösen nagy számú bemenő paraméter esetén. Ez a fő oka annak, hogy a gyakorlatban ennek a problémának a megoldására is inkább heurisztikus módszereket alkalmaznak, melyek bár nem adnak pontos eredményt, jól megközelítik azt. Az illesztéseknek ezt a típusát *progresszívnek* nevezzük. A hierarchikus módszer az egyik legelterjedtebb napjainkban. Ezt használja fel például a Clustal algoritmus, melynek több verziója is létezik (ClustalW, ClustalX).

Ez az eljárás első lépésként minden egyes szekvencia között végrehajt egy gyors páronkénti illesztést és ez alapján készít egy távolság mátrixot. Ezt felhasználva létrehoz egy úgynevezett *„vezér fát"*, majd végighaladva a fán a szomszédos szekvenciákat illeszti. Mindig a két egymáshoz leginkább hasonlítót vizsgálja először, majd ehhez illeszti a következőt. Az itt kapott eredmények ismét felhasználhatóak egy újabb fa készítéséhez. Egészen addig hajtja végre ezeket a lépéseket, míg az illesztés már nem változik.

Előnyök: Az eljárás gyors és egyszerű, általában helyes eredményt ad.

Hátrányok: Ha a kezdeti lépések során hiba lép fel, azt már nem tudjuk korrigálni. Ha a vizsgált szekvenciák nagyon távol állnak egymástól romlik a hatékonysága. lehet hozni. Ezt a módszert valósítja meg a Clustal-algoritmus is.

7. Összefoglalás

A mintaillesztő algoritmusok közül elemeztük és különböző ábécék feletti bemenő adatok segítségével szimuláltuk a Brute–Force, Boyer–Moore, Gyors Boyer–Moore és Apostolico–Giancarlo algoritmusokat. Az eredmények alapján megfogalmaztuk azt a sejtést, miszerint a Gyors Boyer–Moore és az Apostolico–Giancarlo algoritmusok összehasonlítási számára adott felső korlát csökkentjető. A soros megoldások implementálása mellett általánosan megfogalmaztunk egy a problémát párhuzamosan megoldó eljárást, majd ezt is megvalósítottuk és 6 processzor segítségével szimulációt hajtottunk végre. Összehasonlítottuk az 1, 2, 3, 4, 5 és 6 processzor együttes munkájával kapott futási eredményeket. Emellett megvizsgáltuk, hogyan lehet szuffix-fák segítségével megoldani a mintaillesztés problémáját. Megnéztük az egyszeres szekvenciaillesztésnél alkalmazott lokális illesztést végrehajtó Smith–Waterman, és a globális illesztést végző Needlemann-Wunsch algoritmust. Összegyűjtöttük a gyakorlatban leginkább használt büntetőfüggvényeket. Szó esett a Szubsztituciós mátrixok jelentőségéről és a heurisztikus keresési algoritmusokról is. Végül megvizsgáltuk a többszörös szekvenciaillesztés elméleti alapjait.

Irodalomjegyzék:

1. Hans-Joachim Bockenhauer, Dirk Bongartz: Algorithmic Aspects of Bioinformatics.
   Springer Verlag, 2007. 406 oldal, ISBN: 9783540719120, ISBN-10: 3540719121.

2. Miklós István: Bioinformatika. Informatikai algoritmusok 1. kötet 13. fejezet (Szerkesztő: Iványi Antal) ELTE Eötvös Kiadó, 2005. ISBN: 963 463 775 2.
   http://compalg.inf.elte.hu/~tony/Elektronikus/Informatikai/Infalg1E.pdf

3. Ricardo A. Baeza-Yates: Efficient Text Searching. Waterloo, Ontario, 1989.
   http://www.cs.uwaterloo.ca/research/tr/1989/CS-89-17.pdf

4. Brenda S. Baker: Parameterized Pattern Matching by Boyer–Moore-type Algorithms.
   http://reference.kfupm.edu.sa/content/p/a///parameterized_pattern_matching_by_boyer__2671001.pdf

5. R. F. Zhu and T. Takaoka: On improving the average case of the Boyer–Moore string matching algorithm.
   J. Inform. Process., 10(3):173–177,1987.

6. A. Apostolico, R. Giancarlo: Sequence alignment in molecular biology
   Journal of Computational Biology, Vol. 5, pag. 173-196, 1998.

7. Adam R. Galper, Douglas L. Brutlag: Parallel Similarity Search and Alignment with the Dynamic Programming Method. Stanford, California. 1990.
   http://cmgm.stanford.edu/~brutlag/Papers/galper90.pdf

8. A. Andersson and S. Nilsson: Efficient Implementation of Suffix Trees.
   Sofware–Practice and Experience, Vol. 25(2), 129-141 (February 1995)

9. Arthur Dardia, Suzanne Matthews: Suffix Tree Construction.
   Rensselaer Polytechnic Institute
   http://www.cs.rpi.edu/~matths3/documents/suffix_tree.pdf

10. Arne Andersson, N. Jesper Larsson, Kurt Swanson: Suffix Trees on Words
    Dept. of Computer Science, Lund University.

11. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ: Basic local alignment search tool.
(1990).J Mol Biol 215 (3): 403-410. PMID 2231712.

12. http://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html

13. http://www3.imperial.ac.uk/bioinfsupport/help/prot_formats#fasta