NYILATKOZAT

Név: Nagy Boldizsár

ELTE Természettudományi Kar, szak: Matematika BSc

NEPTUN azonosító: IJAKT3

Szakdolgozat címe:
Mintaillesztési algoritmusok

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2023.06.07

Nagy Boldizsár

a hallgató aláírása

Eötvös Loránd Tudományegyetem
Természettudományi Kar
Matematikai Intézet

Mintaillesztési algoritmusok

— Szakdolgozat —

Nagy Boldizsár
Matematika BSc
Alkalmazott matematikus szakirány

Témavezetők: Király Zoltán
Schwarcz Tamás Bence

Számítógéptudományi Tanszék

Budapest, 2023

Tartalomjegyzék

Bevezetés

A szakdolgozatomban különböző mintaillesztési algoritmusokat mutatok be és hasonlítok össze. A mintaillesztés problémája arról szól, hogy egy vagy több rövidebb mintának megkeressük a helyét egy szövegben. Különböző alkalmazásoknál különböző lehet, hogy mit tekintünk találatnak, van ahol csak a pontosan megegyező szeleteit a szövegnek, és van, hogy szükségünk van azokra a szövegrészletekre is melyek nagyon hasonlítanak a mintára, de nem egyeznek meg vele. Az első két fejezetben pontos egyezést kereső algoritmusokról lesz szó, a harmadik fejezetben tárgyalt algoritmusok megengednek különbségeket a minta és a találat között. Való életben is nagyon sok területen alkalmazhatóak ezek az algoritmusok, például DNS szekvenciák keresésére [1], vírusok detektálására [2] és az adatbányászat területén is hasznosnak [3] bizonyulnak.

1. fejezet

Mintaillesztés pontos egyezés alapján

Van egy m hosszú szavunk s, illetve egy n hosszú szövegünk t. A feladat, hogy megtaláljuk a szó első előfordulását a szövegben (hasonló feladat, hogy az összes előfordulását). Könnyen látható, hogy minden algoritmusnak legalább $\lfloor \frac{n}{m} \rfloor$ lépésre lesz szüksége, mivel ennél kevesebb esetén biztosan lesz a szövegnek egy legalább m hosszú szelete, amelynek egyetlen betűjét sem olvastuk el. Vannak algoritmusok, amik bizonyos minták és szövegek esetén elérik ezt és összesen $\lfloor \frac{n}{m} \rfloor$ lépést tesznek csak meg.

1.1. Brute force

1.1.1. Leírás

Az algoritmus végigiterál t összes m hosszú szövegrészletén és megvizsgálja, hogy bármelyik megegyezik-e s-sel.

1. algoritmus Brute force algoritmus

Funct BRUTEFORCE(s, t)

```
1:  for i = 1, ..., (n − m + 1) do
2:      a := TRUE
3:      for j = 1, ..., m do
4:          if s_j ≠ t_{i+j−1} then
5:              a := FALSE
6:          end if
7:      end for
8:      if a then
9:          return i
10:     end if
11: end for
```

1.1.2. Lépésszám

$(n - m + 1) \cdot m$ db összehasonlítást végez az algoritmus, tehát $O(nm)$ a futási ideje.

1.2. Horspool-algoritmus

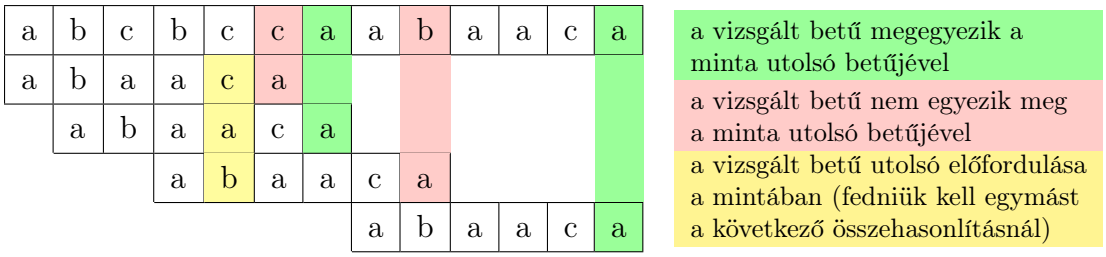
1.2.1. Leírás

Horspool-algoritmusa [4] előre feldolgozza az s mintát úgy, hogy előállít egy E szótárat, amiben a Σ ábécé összes betűjéhez eltárol egy számot, mely a betű távolsága a minta végétől. Amennyiben egy betű többször is előfordul a mintában, úgy a legutolsó előfordulástól vett távolságot tárolja, illetve ha egy betű nincs benne a mintában, akkor a távolsága a minta végétől m, azaz a minta hossza. Az algoritmus mindig a minta utolsó karakterét hasonlítja össze a szöveg i-edik karakterével, ha nem egyezik meg, akkor i-t növeljük az előfeldolgozás alapján kapott értékkel, hogy megvizsgáljuk a következő lehetséges helyet. Amennyiben megegyezik, akkor a szöveg i-ben végződő m hosszú részletét összehasonlítja a teljes mintával, ha megegyezik, akkor visszaadja $(i - m + 1)$-et. Az összehasonlítást a szöveg elejéről kezdjük.

2. algoritmus Horspool-algoritmus az előfeldolgozás nélkül

Funct HORSPOOL(s, t)

```
1:  i := m
2:  while i ≤ n do
3:      if t_i = s_m then
4:          a := TRUE
5:          for j = 1, ... m − 1 do
6:              if s_{m−j} ≠ t_{i−j} then
7:                  a := FALSE
8:              end if
9:          end for
10:         if a then
11:             return i − m + 1
12:         end if
13:     end if
14:     i = i + E[t_i]
15: end while
```



1.1. ábra. Egy példa, amiben a Horspool-algoritmus megkeresi az "abaaca" mintát egy szövegben.

1.2.2. A Horspool-algoritmus helyessége

Bizonyítás. Indirekt tegyük fel, hogy az algoritmus nem vizsgál meg egy esetet, amiben a szövegrész megegyezik a mintával, így rossz eredményt ad. Ehhez az kell, hogy túl nagyot lépjen és áthaladjon felette. Legyen az eltolás nagysága w, az utoljára vizsgált szövegrész és a találat távolsága pedig x, ez azt jelenti, hogy az utoljára összehasonlított betű távolsága a találatban levő helyétől is x. Ha túl nagyot lépett az algoritmus, az azt jelenti, hogy $w > x$, de w az előfeldolgozás szerint a legkisebb távolság, amivel eltolva a mintát jó helyre kerül az utoljára vizsgált betű, tehát nem lehet kisebb, mint x, így ellentmondásra jutottunk, azaz az algoritmus nem léphet túl nagyot. □

1.2.3. Előfeldolgozás

Az E szótárat úgy állítjuk elő, hogy a Σ ábécé összes betűjére a távolságot m-re állítjuk, majd végigmegyünk a minta összes betűjén, és frissítjük a betűhöz tartozó távolságot.

3. algoritmus A Horspool-algoritmus előfeldolgozása

Funct TÁV(s, Σ)

1: **for** $b \in \Sigma$ **do**
2: $\quad E[b] := m$
3: **end for**
4: **for** $i = 1, \ldots, (m-1)$ **do**
5: $\quad E[s_i] := m - i$
6: **end for**

1.2.4. Lépésszám

Az algoritmus legrosszabb esetben mindig eggyel tolja el a mintát és mindig passzol az utolsó karakter, így összehasonlítja a teljes szövegrészt a mintával (pl.: "baaa" keresése az "aaaaaaaaaaaaaaaaabaaa"-ban), ez $(n - m) \cdot (m - 1)$ összehasonlítás, tehát $O(nm)$ lépés. Ha a minta nincs benne a szövegben, akkor legjobb esetben az algoritmus mindig m-mel tolja el a mintát, (pl.:"aaaa" keresése a "bbbbbbbbbbbb"-ben), ekkor $\left\lfloor \frac{n}{m} \right\rfloor$ összehasonlítást végez.

Az előfeldolgozás lépésszáma

Az előfeldolgozás végigiterál az Σ ábécé összes elemén, majd a minta összes betűjén, tehát $|\Sigma| + m$ beállítást/frissítést végez, azaz a lépésszáma $O(|\Sigma| + m)$.

1.3. Knuth–Morris–Pratt-algoritmus

1. Definíció. A q az s szó lábfeje, amennyiben q az s leghosszabb valódi kezdőszelete, ami egyben végszelet is. Avagy $\exists y, z : |y| = |z| > 0 \wedge qy = s = zq$ és $|y|$ minimális.

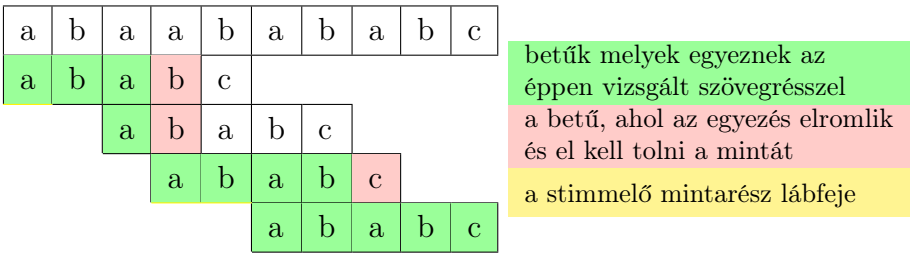
1.3.1. Leírás

A Knuth–Morris–Pratt-algoritmusnak [5] van előfeldolgozási része, ahol az s szó minden kezdőszeletéhez eltároljuk a lábfejének hosszát egy P tömbben. Ezt felhasználva fogjuk eltolni a mintát, kihasználva azt, hogy tudjuk, az éppen vizsgált részszónak mekkora kezdőszelete egyezik meg egy ugyanakkora méretű végszeletével.

4. algoritmus Knuth–Morris–Pratt-algoritmus az előfeldolgozás nélkül

Funct KMP(s, t)

```
1:  i := 1
2:  j := 0
3:  while i + j ≤ n do                    ▷ Tudjuk, hogy s_1 ... s_j = t_i ... t_{i+j-1}
4:      if j = m then
5:          return i
6:      end if
7:      if t_{i+j} = s_{j+1} then
8:          j = j + 1
9:      else
10:         if j = 0 then
11:             i = i + 1
12:         else
13:             i = i + j − P[j]
14:             j = P[j]
15:         end if
16:     end if
17: end while
```



1.2. ábra. Egy példa, amiben a Knuth–Morris–Pratt-algoritmus megkeresi az "ababc" mintát egy szövegben

1.3.2. A Knuth–Morris–Pratt-algoritmus helyessége

Bizonyítás. Indirekt tegyük fel, hogy az algoritmus nem vizsgál meg egy esetet, amiben a szövegrész megegyezik a mintával, így rossz eredményt ad. Legyen a minta helye a szövegben k, akkor ad rossz eredményt az algoritmus, ha az i sose veszi fel a k értéket. Azaz vagy $i < k < i+1$, vagy $i < k < i+j-P[j]$, az első nem lehetséges, mivel $i, k \in \mathbb{Z}$, a második eset azért nem lehetséges, mert $P[j]$ a leghosszabb kezdő- és végszelet, tehát $j - P[j]$ a legkisebb eltolás, aminél a minta j hosszú kezdőszeletének lábfeje jó helyre kerül.

Még azt kell ellenőrizni, hogy ott ahol az algoritmus előfordulást állít, tényleg a minta van a szövegben. Ehhez elég belátni, hogy az algoritmus futása során mindig igaz lesz, hogy a szöveg i-ben kezdődő j hosszú szelete megegyezik a minta j hosszú

kezdőszeletével. Inicializáláskor $i = j = 0$, ez stimmel, ezért ha mindig úgy változtatjuk a két változót, hogy ha eddig megvolt ez a tulajdonság, akkor továbbra is megmarad, akkor végig teljesülni fog. 3 féleképpen változtatjuk i-t és j-t:

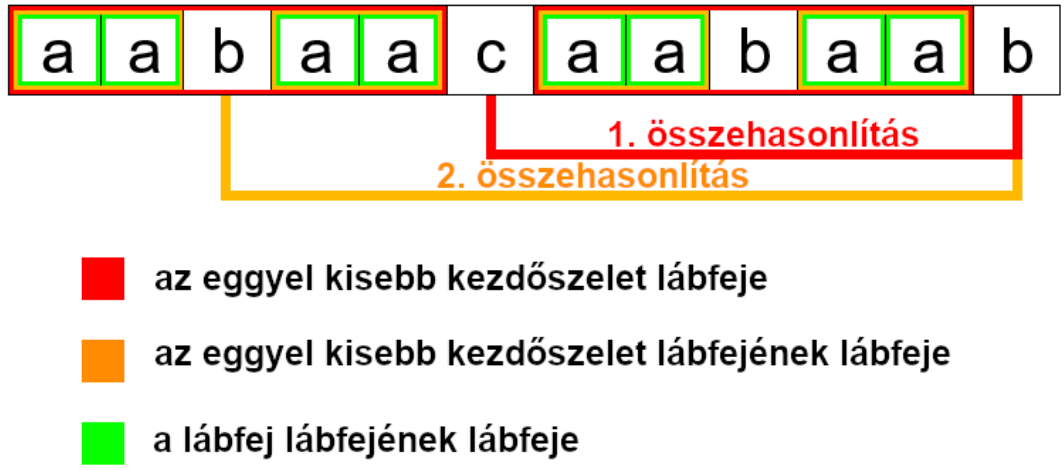
- ha $t_{i+j} = s_{j+1}$, akkor j-t 1-gyel növeljük, ez triviális, hogy nem rontja el a tulajdonságot,
- ha $t_{i+j} \neq s_{j+1} \wedge j = 0$, akkor nem növeljük j-t, tehát 0 marad, és a 0 hosszú szeletek mindig megegyeznek,
- ha $t_{i+j} \neq s_{j+1} \wedge j \neq 0$, akkor i-t annyival növeljük, hogy a $P[j]$ hosszú kezdőszelet, oda kerüljön, ahol eddig a $P[j]$ hosszú végszelet volt és j-t lecsökkentjük $P[j]$-re, ez a két szelet megegyezik, mivel a lábfeje $s_1 ... s_j$-nek, tehát ez sem rontja el a tulajdonságot.

Az algoritmus csak akkor állít előfordulást, ha $j = m$, tehát s m hosszú kezdőszeletét tartalmazza a szöveg i-től $i + m - 1$-ig, és a minta m hosszú kezdőszelete az önmaga. □

1.3.3. Előfeldolgozás

Az előfeldolgozás során szeretnénk az s minta minden kezdőszeletének meghatározni a lábfejét. Ez naiv módon úgy történne, hogy végigiterálunk a kezdőszeleteken, és minden kezdőszelet kezdőszeletein is végigiterálunk, és megnézzük, hogy megegyezik-e a kezdőszelet azonos hosszú végszeletével, ez $O(m^3)$ lépésszámú ami nem hatékony.

Jelölje pref_i az $s_1 \dots s_i$ kezdőszeletet, q_i a pref_i lábfejét és k a q_i hosszát. Az algoritmus minden lábfejet az eggyel rövidebb kezdőszelet lábfejéből számít ki, mivel $P[i+1] = P[i]+1$, ha $q_i s_{i+1} = q_i s_{k+1}$, illetve ha ez nem teljesül, akkor k-t megváltoztatjuk $P[k]$-ra és újra ellenőrizzük $q_i s_{i+1} = q_i s_{k+1}$ feltételt, ha teljesül megtaláltuk $P[i+1]$-t, ha nem akkor megint megváltoztatjuk k-t $P[k]$-ra, ha elérünk $k = 0$-ig, akkor megállunk. Azért $P[k]$-ra változtatjuk k-t, mert egy szó lábfejének a lábfejére is igaz lesz, hogy kezdő- és egyben végszelete is a szónak (csak nem a maximális méretű ilyen tulajdonságú szelet, mert az a lábfej). Azaz elég lesz az ilyen lábfejjelöltek közül kiválasztani a legnagyobbat amelyikre, teljesül, hogy a pref_i-ben utána következő betű megegyezik pref_{i+1} utolsó betűjével, azaz s_{i+1}-gyel.



1.3. ábra. Az "aabaacaabaab" kezdőszelet lábfejjelöltjei

5. algoritmus A Knuth–Morris–Pratt-algoritmus előfeldolgozása

Funct labfejtomb(s)

```
1:  P[0] = 0
2:  P[1] = 0
3:  for i = 2...m do
4:      k = i − 1
5:      P[i] = 0
6:      while k > 0 do
7:          if s_i = s_{P[k]+1} then
8:              P[i] = P[k] + 1
9:              break
10:         else
11:             k = P[k]
12:         end if
13:     end while
14: end for
```

1.3.4. Lépésszám

Három eset van, mikor megváltoztatjuk i-t vagy j-t ($j = j+1$ vagy $i = i+1$ vagy $i = i+j-P[j] \wedge j = P[j]$), ezek közül minden iteráció során pontosan egy valósul meg. Ebből látszik, hogy i monoton nő, mivel sosem csökkentjük az értékét, $i+j$ szintén monoton nő, mivel vagy i vagy j nő vagy $i+j = (i+j-P[j])+P[j] = i+j$, azaz nem csökken sosem. Az is látszik, hogy vagy i vagy $i+j$ szigorúan monoton nő, mivel $(j = j+1)$ esetén $i+j$ nő, illetve $i = i+1$ és $i = i+j-P[j] \wedge j = P[j]$ megvalósulása esetén i nő. Tehát $2i+j$ szigorúan monoton nő. Az algoritmus kezdetén $2i+j = 2$

a végén pedig $2i + j \leq 2n$, azaz maximum $2n - 2$ db összehasonlítást végez az algoritmus, így a lépésszáma $O(n)$.

Az előfeldolgozás lépésszáma

Legyen k_j az, hogy $P[j]$ kiszámításához hány lépésre van szükség. Mikor $P[j]$-t kiszámítjuk $P[j-1]$-ből, az i minden körben szigorúan monoton csökken, legalább k_j-szer, tehát a végén $i \leq P[j-1] - k_j + 1$ és ekkor $P[j] = i$ vagy $i + 1$, tehát $P[j] \leq P[j-1] - k_j + 2$. Így

$$\sum_{j=1}^{m} k_j = P[1] - P[m] + 2 \cdot (m-1) \leq 2(m-1),$$

mivel $P[1] = 0$ és $P[m] \geq 0$.

1.4. Karp–Rabin-algoritmus

1.4.1. Leírás

A Karp–Rabin-algoritmus [6] nem a mintát hasonlítja össze a szöveggel, mint a korábban tárgyalt algoritmusok, hanem a minta ujjlenyomatát az éppen vizsgált szövegrész ujjlenyomatával. Először kiszámítja a minta ujjlenyomatát, ezután pedig végigiterál a szöveg összes m hosszú szeletén, és azoknak is kiszámítja az ujjlenyomatát és összehasonlítja minta ujjlenyomatával. Az ujjlenyomatok kiszámítására „gördülő" hash függvényt használ melynek lényege, hogy ha tudjuk $t_i t_{i+1} \dots t_{i+m-1}$ ujjlenyomatát, akkor konstans időben ki tudjuk számolni $t_{i+1} t_{i+2} \dots t_{i+m}$ ujjlenyomatát, azaz adott m hosszú szelet ujjlenyomatát meg lehet kapni az eggyel korábban kezdődő m hosszú szelet ujjlenyomatából konstans időben. Tehát itt az előfeldolgozás azt jelenti, hogy kiszámoljuk a minta ujjlenyomatát, ez m lépés.

1.4.2. Példa hash függvényre

Egy megfelelő hash függvény, ha minden szóhoz egy számot rendelünk, úgy hogy a Σ ábécé betűihez hozzárendelünk egész számokat 0-tól $|\Sigma| - 1$-ig, és a szó betűit lecseréljük a nekik megfelelő számokra, mintha számjegyek lennének $a = |\Sigma|$ alapú számrendszerben.

x	val(x)
a	0
b	1
c	2
d	3
e	4

b	e	e	a	b
1	4	4	0	1

$\mathrm{h}(beeab) = 14401_5 = 1226_{10}$

1.4. ábra. Egy megfelelő hash függvény értéke *beeab*-ben, $\Sigma = \{a, b, c, d, e\}$ ábécé mellett.

Legyen a betűhöz számot rendelő függvény $val(x)$, ekkor a $t_i t_{i+1} \dots t_{i+m-1}$ szóra a hash függvény értéke $val(t_i) \cdot a^{m-1} + val(t_{i+1}) \cdot a^{m-2} + ... + val(t_{i+m-1}) \cdot a^0$. A $t_{i+1} t_{i+2} \dots t_{i+m}$ szóra hasonlóan $val(t_{i+1}) \cdot a^{m-1} + val(t_{i+2}) \cdot a^{m-2} + ... + val(t_{i+m}) \cdot a^0$ a hash függvény értéke, azaz $val(t_i) \cdot a^{m-1} + val(t_{i+1}) \cdot a^{m-2} + ... + val(t_{i+m-1}) \cdot a^0$-ból kivonva $val(t_i) \cdot a^{m-1}$-t, megszorozva a-val és hozzáadva $val(t_{i+m})$-et megkapjuk $val(t_{i+1}) \cdot a^{m-1} + val(t_{i+2}) \cdot a^{m-2} + ... + val(t_{i+m}) \cdot a^0$-t azaz a következő szelet ujjlenyomatát. A tárhely és a sebesség érdekében, a hash függvényt módosíthatjuk úgy, hogy a végén még egy lépést csinál, ahol a szám maradékát veszi egy nagy prímszámra M-re, azért választunk prímet, mert akkor ritkábban lesz két különböző szeletnek ugyanaz az értéke. A modulo M nem rontja el a "gördülési" tulajdonságot mivel $a = q \cdot M + r$ ahol $r < M$ esetén $a + b \equiv r + b \bmod M$.

6. algoritmus hash függvény

Funct h(s)

1: $a = 0$
2: **for** $i = 1...m$ **do**
3: $\quad a = (a \cdot alap + val(s_i)) \bmod M$
4: **end for**
5: **return** a

7. algoritmus Karp–Rabin-algoritmus

Funct KARPRABIN(s, t)

```
1:  H = h(s)
2:  v = h(t_1...t_m)
3:  Q = |Σ|^{m-1}
4:  for i = 2, ...m do
5:      v = (v − val(t_{i−1}) · Q) mod M
6:      v = (v · m + val(t_{i+m−1})) mod M
7:      if H = v then
8:          return i
9:      end if
10: end for
```

1.4.3. A Karp–Rabin-algoritmus helyessége

A betű-szám hozzárendelés és a szó felírása is kölcsönösen egyértelmű, a $\bmod M$ leképezés egyértelmű, így két azonos szeletnek mindig ugyanaz lesz az ujjlenyomata, tehát nem lehetséges, hogy egy találatot nem talál meg az algoritmus. Mivel a $\bmod M$ leképezés nem injektív, így ritkán (M-től függ) lehetséges, hogy két különböző szeletnek megegyezik az ujjlenyomata, és ekkor ott jelezhet találatot az algoritmus, ahol valójában nincs. Ez kiküszöbölhető úgy, hogy ahol találatot jelez az algoritmus ott összehasonlítjuk a szeletet a mintával, ez növeli a lépésszámot legfeljebb $O(m \cdot x)$ lépéssel, ahol x a mintával megegyező ujjlenyomatú szeletek száma a szövegben.

1.4.4. Lépésszám

Az n hosszú szövegnek $n - m$ db m hosszú szelete van, illetve az első szelet ujjlenyomatát teljes egészében ki kell számolnunk, mivel abból indulunk ki és ez még m lépés, tehát az algoritmus lépésszáma $O(n)$.

1.5. A futásidők mérése

1.5.1. A mérések módszere

A mérésekhez használt implementációk megtalálhatóak ezen a linken. A mérések tisztaságának érdekében az implementációk az összes előfordulást megkeresik, így egy korai találat nem okoz kiugró eredményt.

Futtatási környezet

Operációs rendszer:	Windows 10 Pro x64bit
Processzor:	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
Memória:	8.00 GB DDR4
Fordító:	MinGW (gcc 11.2.0)

Időmérés

Az időmérést a C++ `chrono` library-jének a `high_resolution_clock` osztályával valósítom meg, minden tesztesetet 10 alkalommal futtatok és kiválasztom a leggyorsabbat.

Véletlenszerű tesztesetek

Minden $(n, m, |\Sigma|)$ paraméter-szethez 20 db szöveget és 20 db mintát generálok egyenletes eloszlással visszatevéssel választva az ábécéből, és mind a 400 minta-szöveg tesztesetet lefuttatom, az eredményeket sorbarendezem és a középső 80% átlagát veszem.

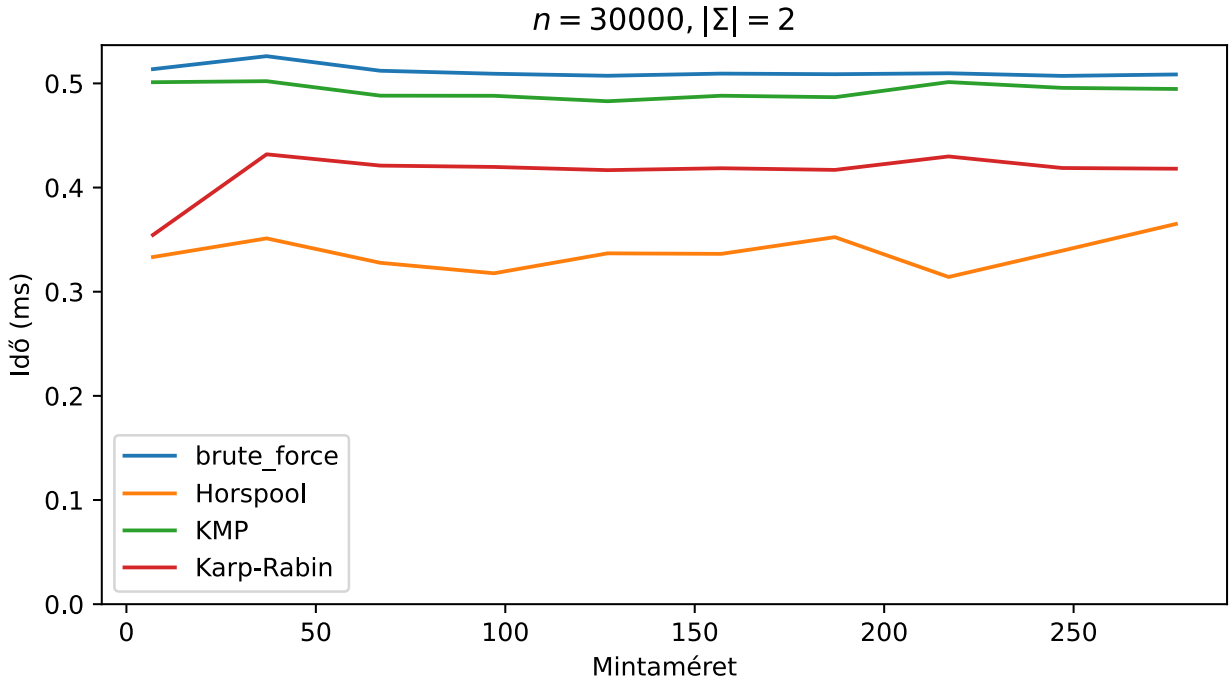
A Karp-Rabin implementáció

A Karp-Rabin implementációmban a használt M modulus 1000000007. Azt a változatot tesztelem, ahol ha a szelet és a minta ujjlenyomata egyezik, akkor leellenőrzi az algoritmus, hogy maga a szelet és a minta is megegyezik, így nincsenek hamis találatok. A számrendszer alapszáma az ábécémérettől függetlenül mindig 64 az implementációmban.

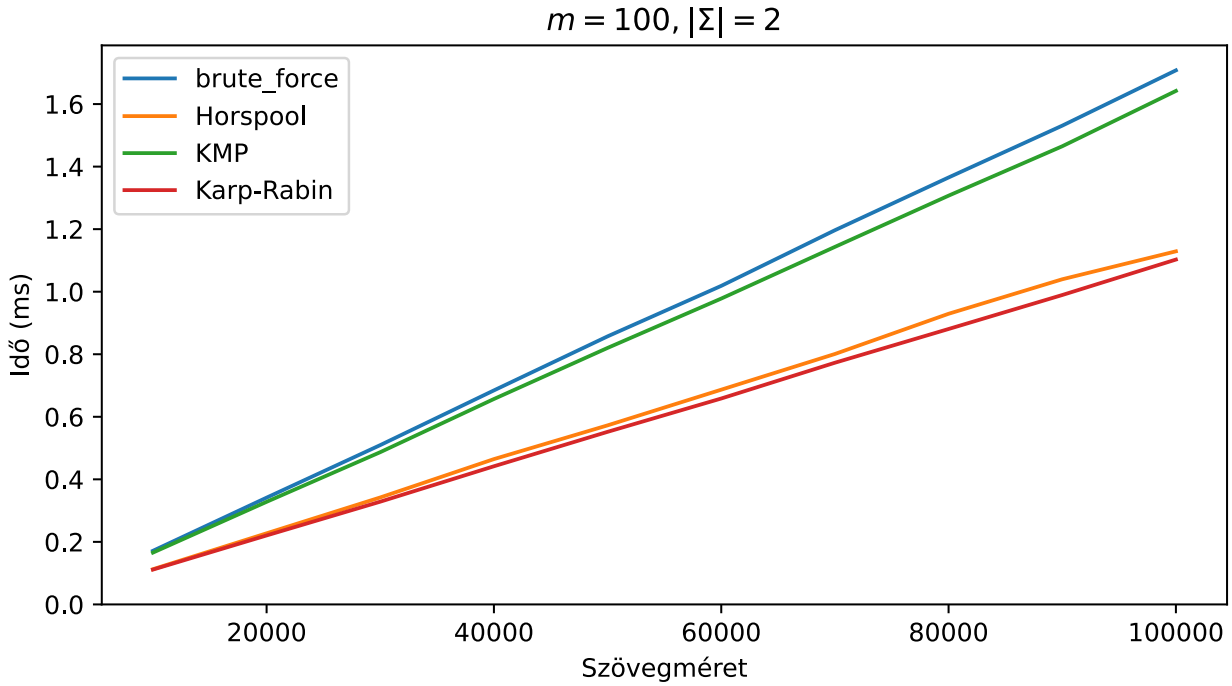
1.6. A mérések eredménye

1.6.1. 2 elemű ábécé

A 2 elemű ábécék kiemelkedően fontosak, mivel nagyon sokszor bitsorozatokkal tárolunk információkat, így hasznos, ha gyorsan tudunk bennük keresni.



1.5. ábra. A különböző algoritmusok futásideje 2 elemű ábécé és 30000 hosszú szöveg esetén, a mintaméret függvényében

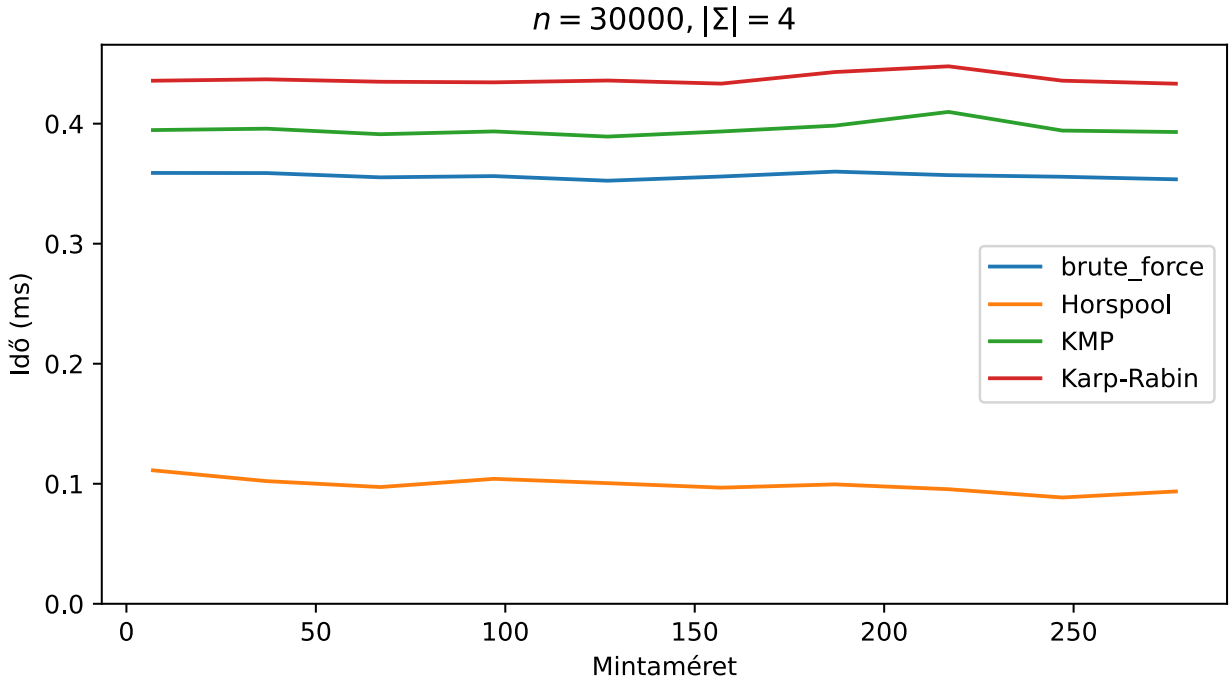


1.6. ábra. A különböző algoritmusok futásideje 2 elemű ábécé és 100 hosszú minta esetén, a szövegméret függvényében

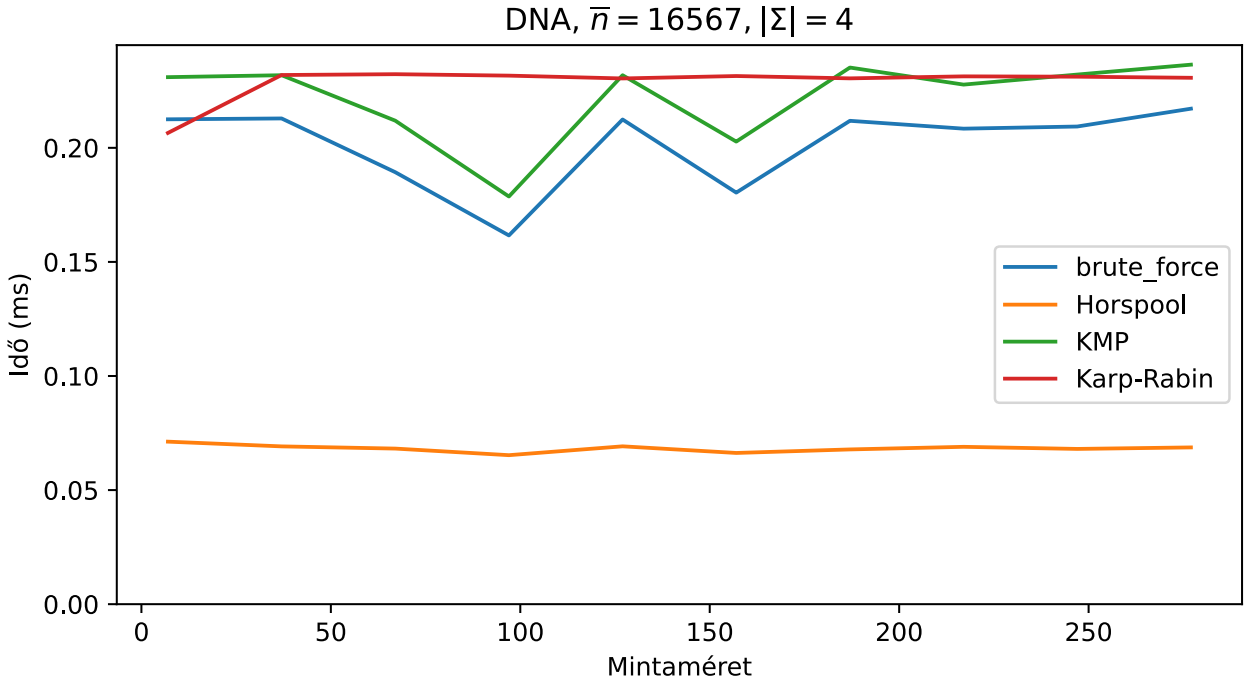
A szövegméret növekedésével lineárisan nő az algoritmusok futásideje is, míg a mintaméret növekedésének nincs számottevő hatása a futásidőkre.

1.6.2. 4 elemű ábécé

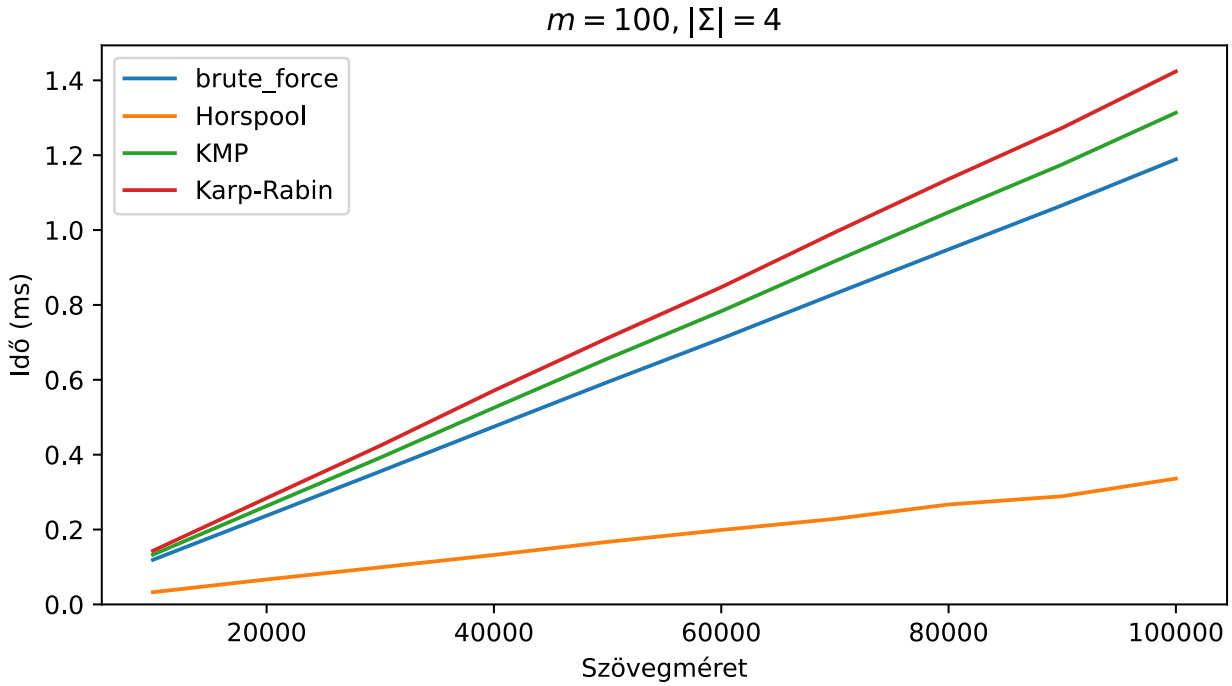
A 4 elemű ábécé is nagyon fontos bioinformatikában, mivel a DNS szekvencia 4 féle nukleobázis (citozin (C), guanin (G), adenin (A), timin (T)) sorozata. Itt nem csak véletlen eseteken teszteltem, hanem valós adatokon [7] is , ezek elérhetőek itt. A valós tesztekhez 10 db körülbelül azonos hosszúságú ($kb.16560$) hosszú szekvenciát használtam szövegnek, és minden m mintaérethez generáltam 10 db m hosszú mintát, úgy hogy mind a 10 szöveg utolsó m karakterét vettem. Ezután mind a 100 minta-szöveg párt lefuttattam és a középső 80% átlagát vettem.



1.7. ábra. A különböző algoritmusok futásideje 4 elemű ábécé és 30000 hosszú szöveg esetén, a mintaméret függvényében



1.8. ábra. A különböző algoritmusok futásideje valós szövegeken, a mintaméret függvényében

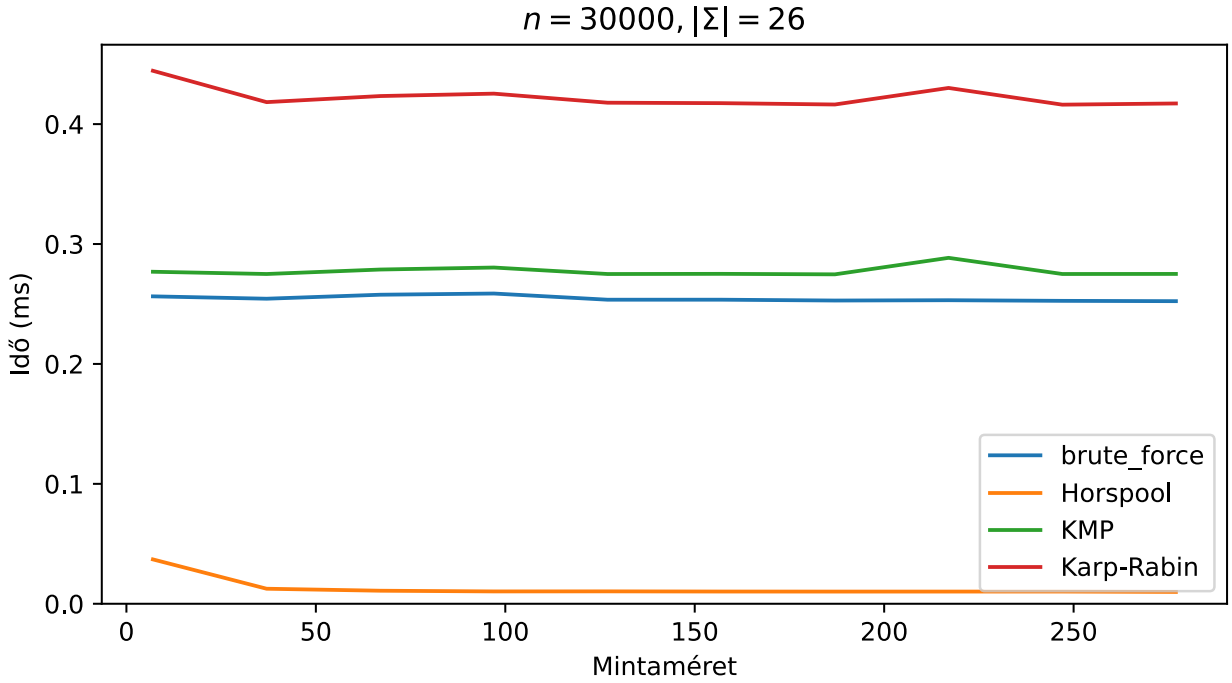


1.9. ábra. A különböző algoritmusok futásideje 4 elemű ábécé és 100 hosszú minta esetén, a szövegméret függvényében

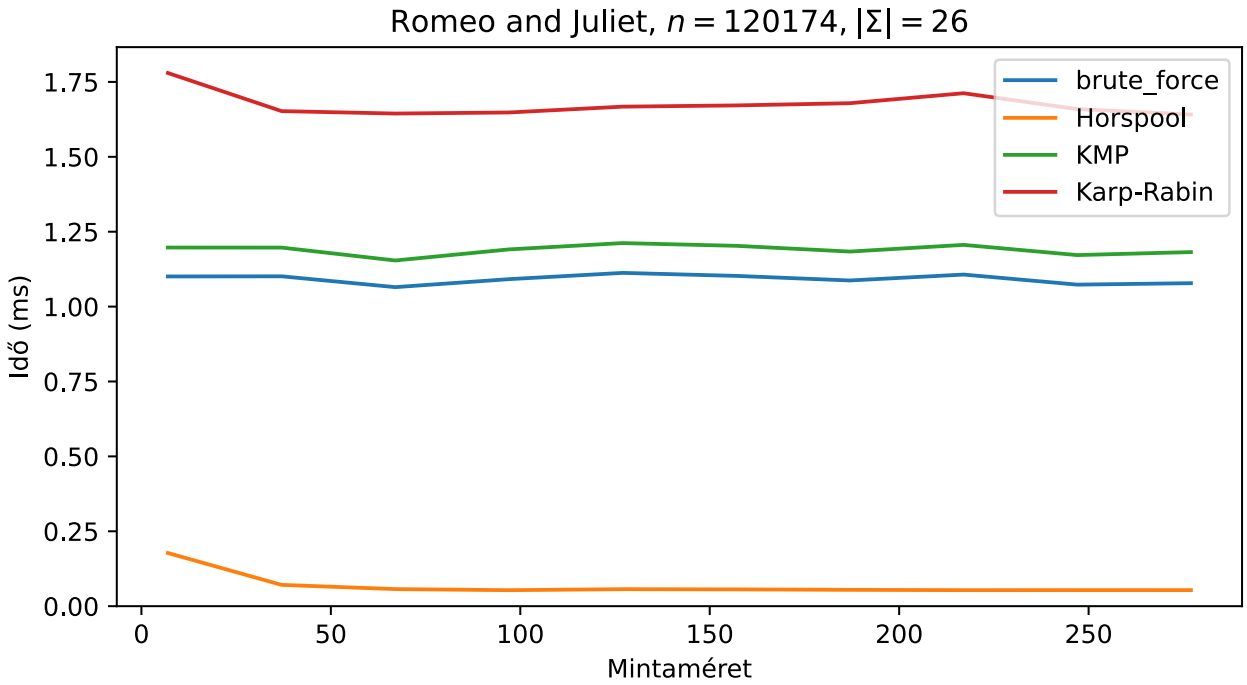
Jól látszik, hogy a szövegméret növekedése itt is a futásidők lineáris növekedésével jár, míg a mintaméret kevésbé befolyásolja a futásidőket 4 elemű ábécé esetén.

1.6.3. 26 elemű ábécé

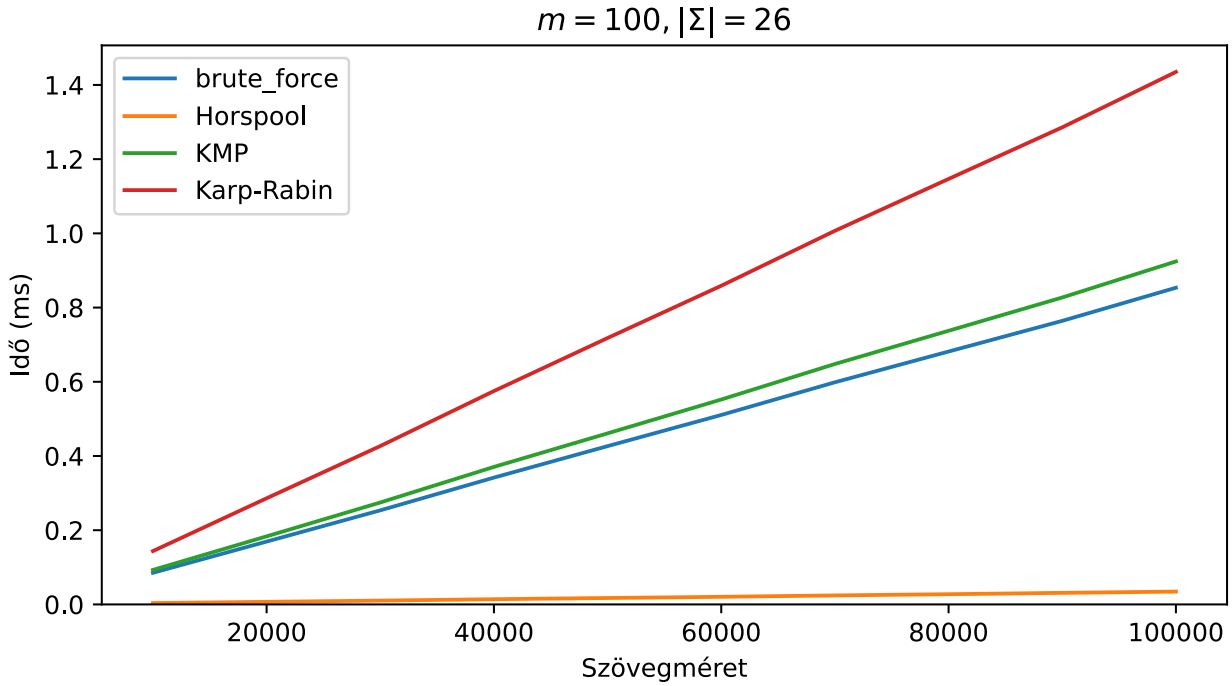
Az angol ábécé 26 betűt tartalmaz, ezért a 26 elemű ábécének is kiemelt szerepe van. Itt a ”Rómeó és Júlia” [8] angol szövegét használtam a teszteléshez, eltávolítva belőle az összes nem az angol ábécébe tartozó karaktert (pl. ”.”, ”-”, stb.) és kisbetűssé téve az egészet, hogy jól összehasonlítható legyen a véletlenszerűen generált szövegekkel.



1.10. ábra. A különböző algoritmusok futásideje angol ábécé és 30000 hosszú szöveg esetén, a mintaméret függvényében



1.11. ábra. A különböző algoritmusok futásideje, mikor a Rómeó és Júlia angol szövegében különböző méretű részleteket keresnek

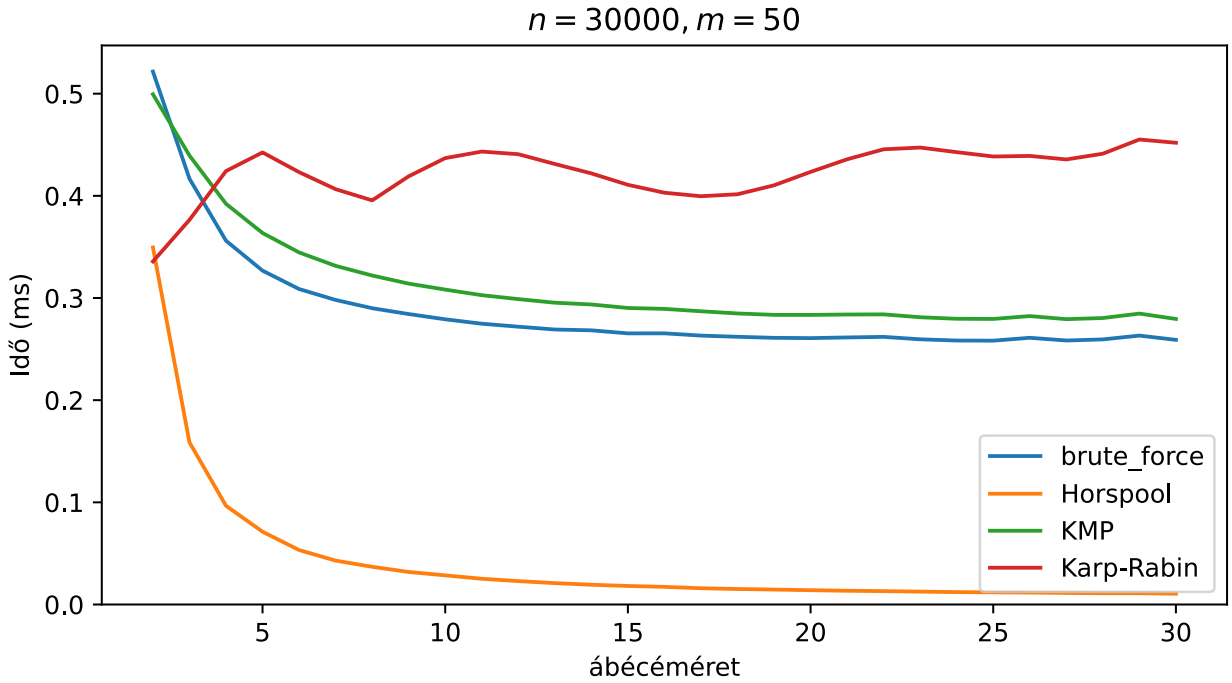


1.12. ábra. A különböző algoritmusok futásideje angol ábécé és 100 hosszú minta esetén, a szövegméret függvényében

A szövegméret növekedésével itt is nő minden algoritmus futásideje. A mintaméret hatása a futásidőre itt sem számottevő. A Horspool ekkora ábécénél mind

véletlenszerűen generált szövegen, mind a ”Rómeó és Júlia” angol szövegén sokkal gyorsabb a többinél, a gyakoribb és nagyobb eltolásoknak köszönhetően.

1.6.4. Különböző ábécéméretek



1.13. ábra. A különböző algoritmusok futásideje 50 hosszú minta és 30000 hosszú szöveg esetén, az ábécéméret függvényében

A Horspool esetén tapasztalható a legnagyobb gyorsulás, mivel minél nagyobb az ábécé annál ritkábban egyezik meg a minta utolsó és a szöveg éppen vizsgált betűje, így ritkábban nézi végig a teljes mintát, illetve az eltolások is nagyobbak, mivel többféle betűt tartalmazhat a minta.
A KMP futásidejére is hatással van az ábécéméret, nagyobb ábécék esetén gyorsabb.
A Karp–Rabinnál az ábécéméret nem befolyásolja igazán a futásidőt.

1.6.5. Összegzés

A Horspool legrosszabb esetben $O(nm)$ lépésszámú, a gyakorlatban azonban, nagyon nagyon gyors, főleg nagyobb ábécéméretek esetén.
A KMP és a brute force az ábécéméretre valamelyest érzékeny, de a gyakorlati futásideje leginkább a szöveghossztól függ csak, ahogy azt elméletben láttuk.
A Karp–Rabin futásideje is a szöveghossztól függ leginkább. Fontos megjegyezni,

hogy a futásidők nagyon függnek a megvalósítástól is. Az implementációk elkészítése során az elméletben konstans idejű műveletek optimalizálásával, nagyon nagy javulást tudtam elérni.

2. fejezet

Több minta keresése pontos egyezés alapján

Van egy n hosszú szöveg, és k darab $m_1, m_2, \ldots m_k$ hosszú minta, amiknek az összes előfordulását a szövegben meg szeretnék találni. A fejezetben legyen $M := \sum_{i=1}^{k} m_i$.

2.1. Egy mintát kereső algoritmusok felhasználása

Az a probléma az eddig vizsgált algoritmusokkal, hogy a Karp–Rabint leszámítva nem tudnak párhuzamosan egyszerre több mintát keresni, így k db minta esetén k-szor kell futtatni az algoritmust, minden mintára egyszer. A Karp–Rabin-algoritmust egy kis módosítással át lehet írni több minta keresésére úgy, hogy az előfeldogozás során minden minta ujjlenyomatát kiszámítja és egy hashmapben eltárolja. A hashmapet a C++ standard libraryjének `<unordered_map>` osztályával valósítom meg. A keresés ugyanúgy folyik mint az egy mintát kereső esetben, azzal a különbséggel, hogy itt az éppen vizsgált részlet ujjlenyomatát nem "a minta" ujjlenyomatával hasonlítja össze, hanem megnézi, hogy megtalálható-e a hashmapben. A hashmapben a lekérdezés átlagosan $O(1)$ idejű, így több mintára is $O(n)$ lesz a Karp–Rabin-algoritmus futásideje. Fontos megjegyezni azonban, hogy ha a keresendő minták nem azonos hosszúságúak, akkor annyiszor kell végigmenni a szövegen ahány különböző mintahossz van, hogy minden lehetséges találatot megvizsgáljunk.

Algoritmus	**Lépésszám k darab minta esetén**
Brute force	$\sum_{i=1}^{k} O(nm_i) = O(nM)$
Horspool	$\sum_{i=1}^{k} nm_i = O(nM)$
KMP	$\sum_{i=1}^{k} (2n-2) = O(nk)$
Karp–Rabin	$O(n)$

2.1. táblázat. Egy mintát kereső algoritmusok lépésszáma legrosszabb esetben több minta esetén

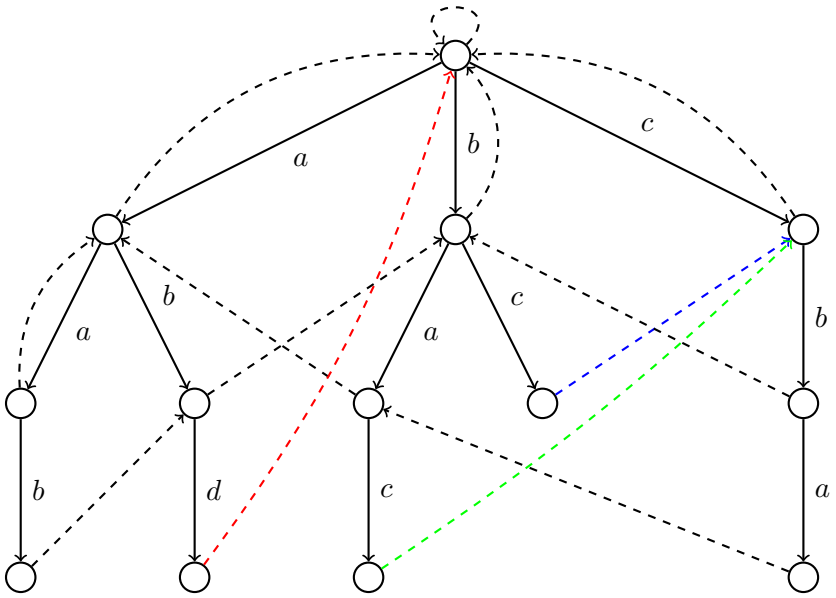
2.2. Aho–Corasick

2.2.1. Leírás

Az Aho–Corasick-algoritmus [9] a Knuth–Morris–Pratt-algoritmushoz hasonlóan az azonos mintarészleteket használja ki. A linuxos `grep` program is ezt az algoritmust használja több minta egyidejű keresésére [10]. Az előfeldolgozás során építünk egy véges automatát, és az algoritmus során végigiterálunk a szöveg betűin és közben a betűk alapján lépkedünk az automatában.

2.2.2. Az automata adatstruktúrája

Az automata adatstruktúrája csúcsokból áll, amiket élekkel összekötünk. Egy csúcs tartalmaz egy szomszédsági listát, mint egy egyszerű gráfcsúcs, de ezenfelül van egy külön hibaéle is, amiben azt tároljuk, hogy hova kell lépnünk, ha a következő betű szerint nem tudnánk továbblépni.
Minden él tartalmaz egy karaktert (betűt). A hibaélek azért kellenek, mert ha egy mintának egy első kezdőszeletét tudtuk illeszteni, de nem az egészet, akkor nem biztos, hogy újra a start pozícióból kell kezdenünk, mert lehetséges hogy a minta stimmelő kezdőszeletének van egy olyan végszelete amely egy másik minta kezdőszelete, és ekkor úgy kell folytatódnia az algoritmusnak, hogy ezt a mintát is megtalálja, tehát abból az állapotból kell továbblépni.



2.1. ábra. Egy automata, az aab, abd, bac, bc, cba mintákból.

2.2.3. Előfeldolgozás – Az automata építése

Az automatának az építés elején egy állapota van, a kezdőállapot. Ezután végigiterálunk a mintákon és minden mintát hozzáadunk úgy, hogy a kezdőállapotból indulunk és végigiterálunk az adott minta betűin. Ha tudunk az éppen vizsgált betű szerint lépni, akkor lépünk, ha nem, akkor hozzáadunk az automatához egy új élet abból a csúcsból ahonnan nem tudtunk továbblépni egy új állapotba és erre az élre az éppen vizsgált betűt írjuk.

Ezután hozzáadjuk a hibaéleket. A kezdőállapot hibaéle a kezdőállapotba mutat. A többi hibaélet úgy adjuk hozzá, hogy fentről lefelé szintenként végigiterálunk a fa csúcsain. Legyen az x csúcsban végződő mintarészlet $P = p_1 \dots p_k$, kell P leghosszabb valódi végszelete, amely valamelyik minta kezdőszelete, ennek a kezdőszeletnek a végébe kell, hogy mutasson a hibaélünk. Itt kihasználhatjuk, hogy $p_1 \dots p_{k-1}$-ről már tudjuk melyik a leghosszabb megfelelő végszelete, mivel szintenkét haladunk, tehát ha az x szülőjének hibaélének végén lévő csúcsból (y-ból) megy p_k jelű él, akkor az abban végződő minta kezdőszelet, így az él megfelelő hibaél lesz (pl.: a 2.1-es ábrán a kék hibaél). Ha nincs p_k jelű él, akkor rekurzívan továbbhaladunk: a hibaélen lépünk tovább a következő állapotba és újra megnézzük, hogy tudunk-e p_k jelű élen lépni, ha igen, lépünk és megállunk, ide fog mutatni a hibaélünk (pl. a 2.1-es ábrán a zöld él). Lehetséges, hogy egyet lépünk a kezdőállapot hibaélén önmagába, ekkor is megállunk, hogy ne kerüljünk végtelen ciklusba, ez azt jelenti, hogy P-nek nincs

olyan végszelete, amely egy minta kezdőszelete, (pl. a 2.1-es ábrán a piros él). Azért haladunk a hibaéleken, mert P minden végszeletének a végszelete a P végszelete is.

2.2.4. Lépésszám

Az algoritmus lépésszáma $O(n)$. $O(n)$ összehasonlítást végzünk, mivel mindig mikor rendes élen vagy a start (hurok)hibaélén lépünk, akkor lépünk egyet a szövegen, ha (nem start) hibaélen lépünk, akkor nem lépünk a szövegben, ezért legalább n lépés lesz, de kevesebb, mint $2n$, mivel maximum annyiszor léphetünk (nem start) hibaélen ahányszor léptünk rendes élen, mivel a hibaélek mindig a fában szigorúan nagyobb szintről mennek kisebb szintre.
Az automata építésének első része $O(M)$ lépés, mivel minden egyes mintát hozzáadunk a fához végigiterálva a minták betűin. Legfeljebb $M+1$ db hibaél van, mivel minden állapothoz pontosan egy tartozik. Legyen az i. minta k hosszú kezdőszeletének megfelelő állapot $p_{i,k}$, az ebből kimenő hibaél végén lévő csúcs pedig $h_{i,k}$. A $h_{i,k}$ szintje $d(start, h_{i,k})$. A $p_{i,k}$ hibaélének keresésénél először a szülője hibaélének végén lévő csúcsot vizsgáljuk meg $h_{i,k-1}$-t, utoljára pedig $h_{i,k}$-t, minden szinten legfeljebb egy csúcsot vizsgál meg az algoritmus, mivel a hibaélek szigorúan kisebb szintre mutatnak. Így az i. mintához tartozó hibaélek kiszámításához szükséges lépések száma legfeljebb

$$\sum_{k=1}^{m_i}\Big(d(start, h_{i,k-1}) + 1 - d(start, h_{i,k})\Big) = d(start, h_{i,0}) + m_i - d(start, h_{i,m_i}) =$$

$$= 0 + m_i - d(start, h_{i,m_i}) \leq m_i$$

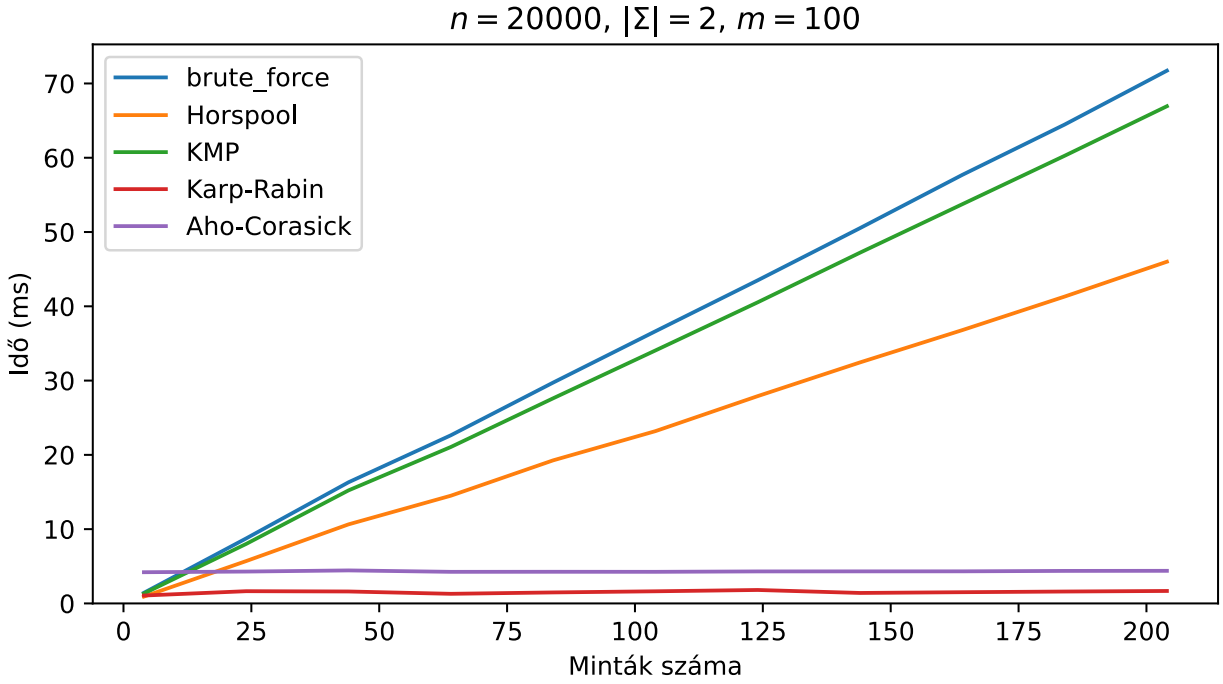
A hibaélek kiszámításához szükséges lépések száma legfeljebb $O(\sum m_i) = O(M)$.

2.3. Az Aho–Corasick teljesítménye, az egy mintát kereső algoritmusokkal szemben

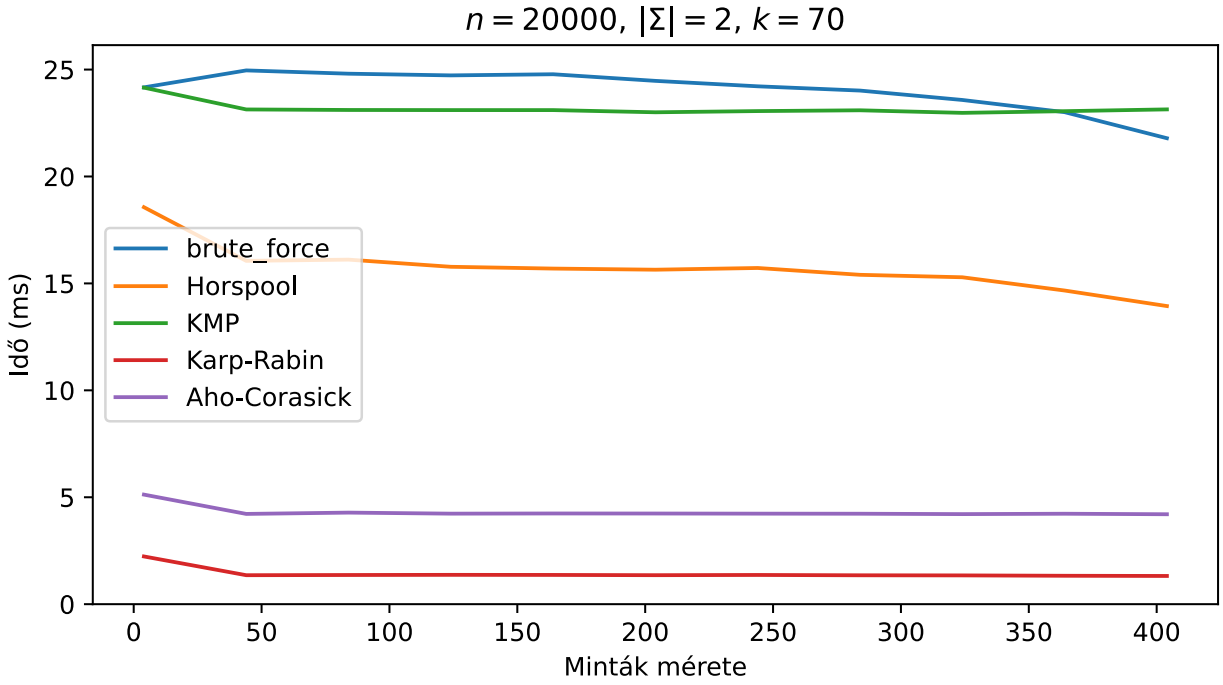
A mérésekhez használt implementációk megtalálhatóak ezen a linken. A mérések tisztaságának érdekében az implementációk az összes előfordulást megkeresik, így egy korai találat nem okoz kiugró eredményt. Az brute force, KMP és Horspool algoritmusok minden egyes mintát külön megkeresnek a szövegben, azaz k minta

esetén k különböző keresést végeznek és ezek futásidejének összege a teljes futásidő, azonban a Karp–Rabinnak a hashmapes módosítását tesztelem ebben a fejezetben.

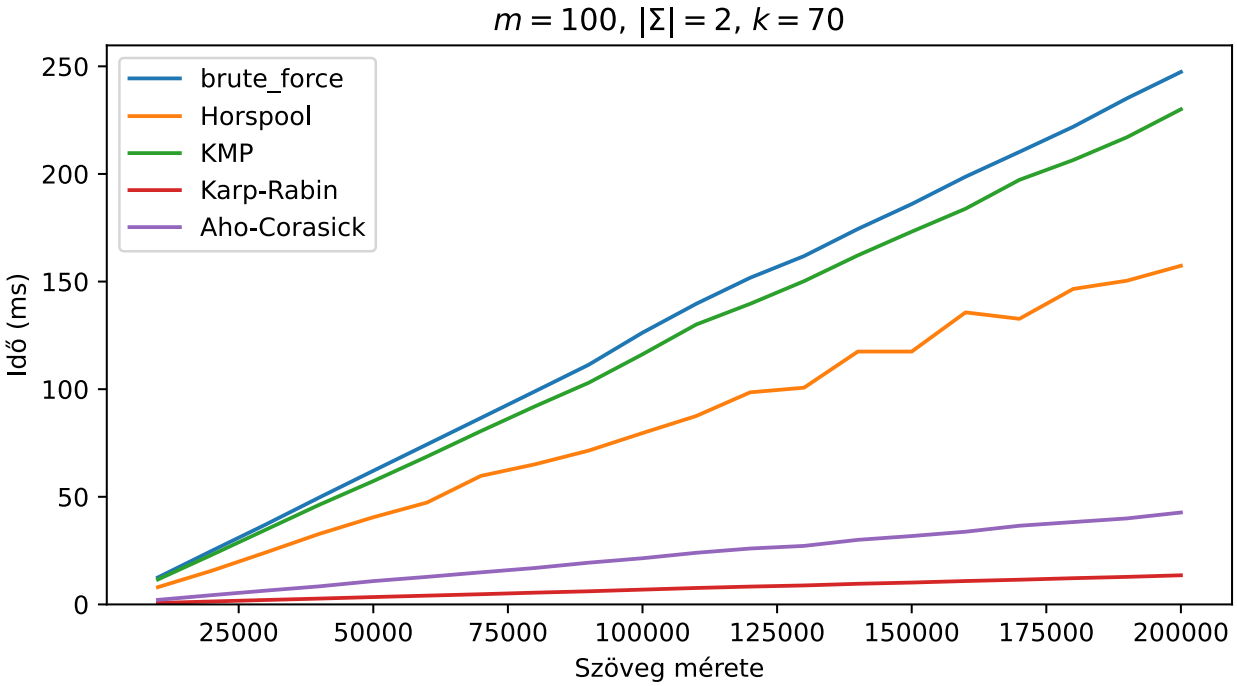
2.3.1. 2 elemű ábécé



2.2. ábra. A különböző algoritmusok futásideje két elemű ábécé esetén, 100 hosszú mintákkal, a minták számának függvényében



2.3. ábra. A különböző algoritmusok futásideje 2 elemű ábécé és 70 db minta esetén, a mintaméret függvényében

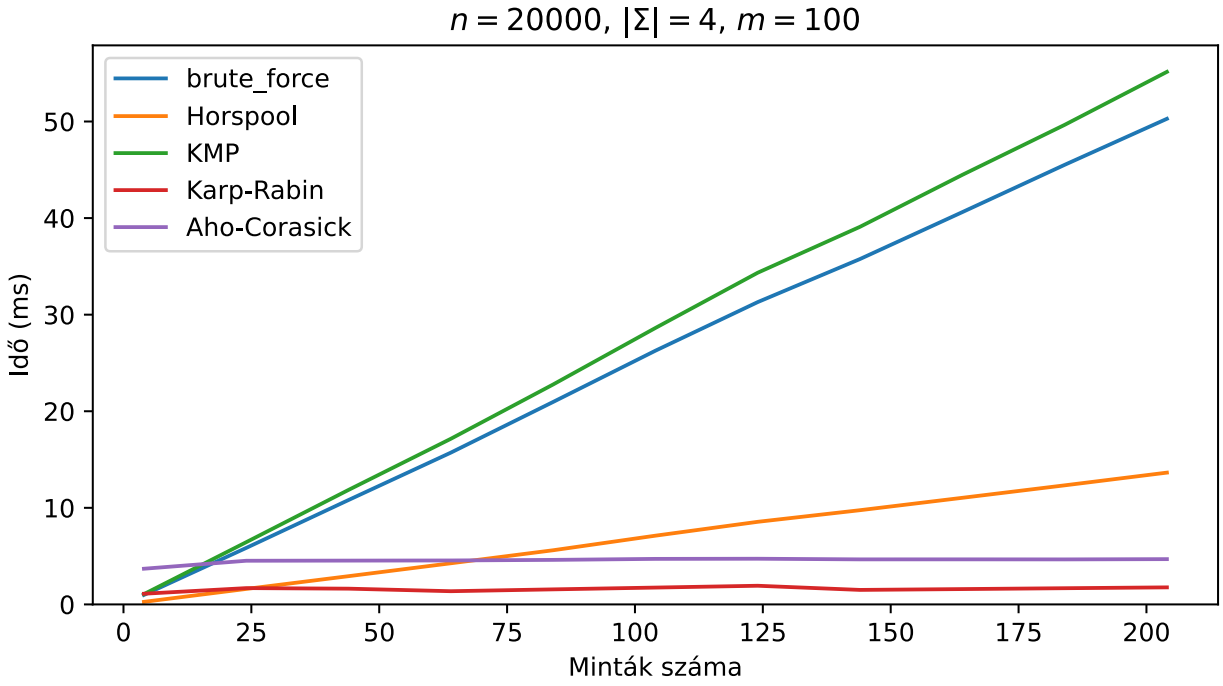


2.4. ábra. A különböző algoritmusok futásideje két elemű ábécé és 70 db 100 hosszú minta esetén a szöveg hosszának függvényében

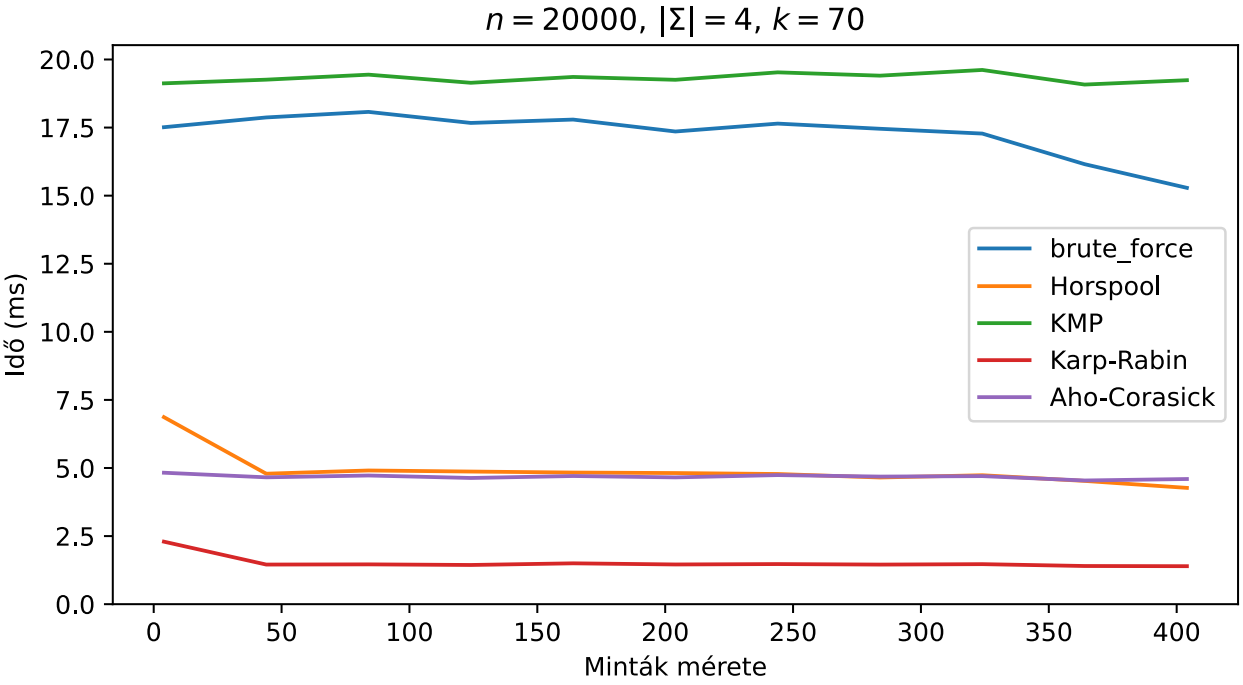
Jól látszik, hogy sem az Aho–Corasicknek sem a Karp–Rabinnak a futásideje a két elemű ábécéknél nem függ a minták számától, akár rövidek akár hosszúak, a

szövegmérettől függ csak a gyakorlati futásidejük.

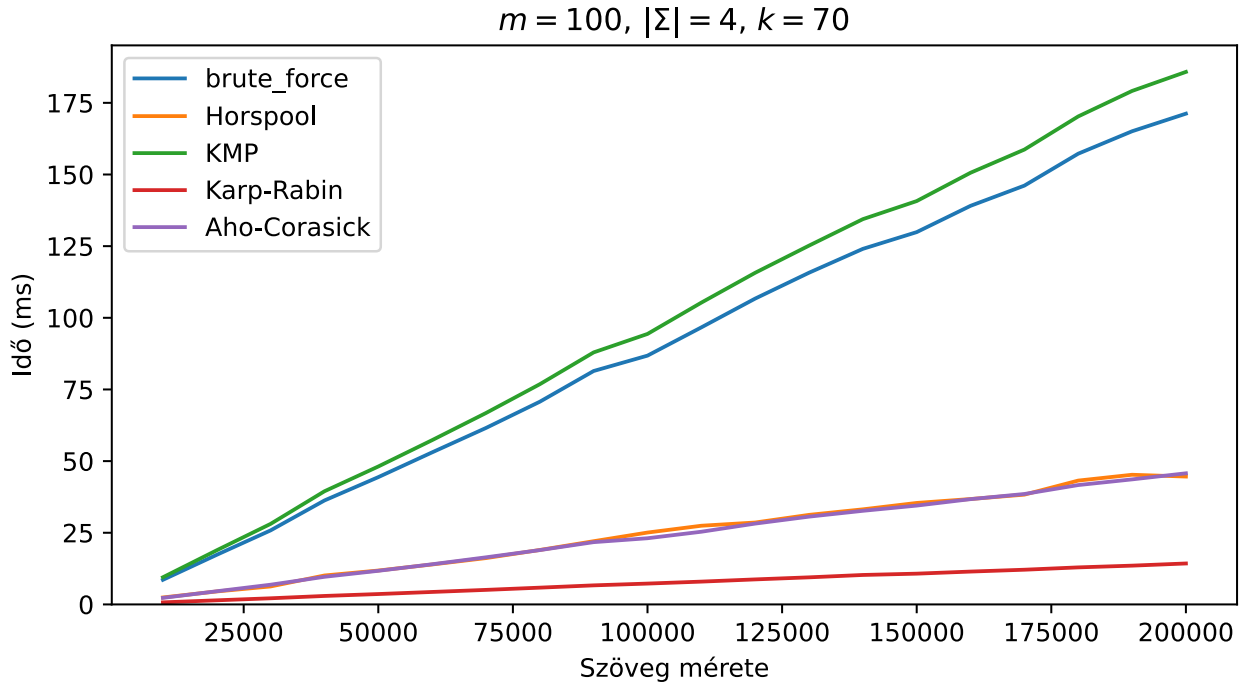
2.3.2. 4 elemű ábécé



2.5. ábra. A különböző algoritmusok futásideje négy elemű ábécé esetén, 100 hosszú mintákkal



2.6. ábra. A különböző algoritmusok futásideje négy elemű ábécé és 70 db minta esetén, a mintaméret függvényében

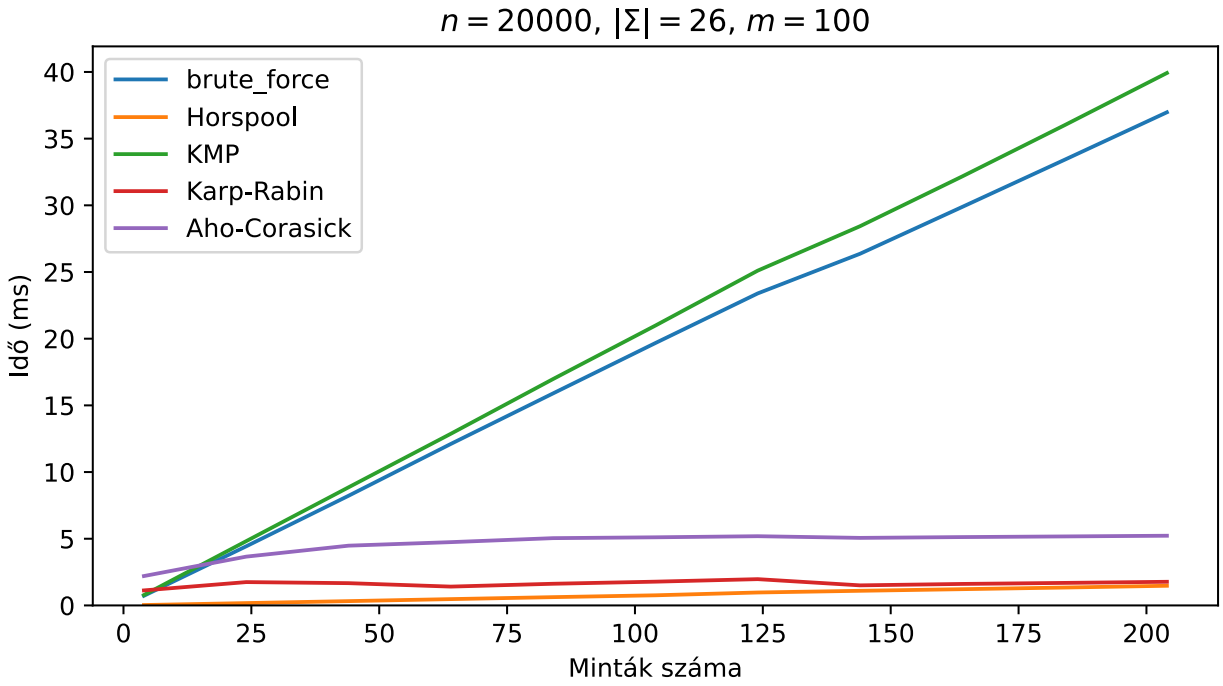


2.7. ábra. A különböző algoritmusok futásideje négy elemű ábécé és 50 db 100 hosszú minta esetén a szöveg hosszának függvényében

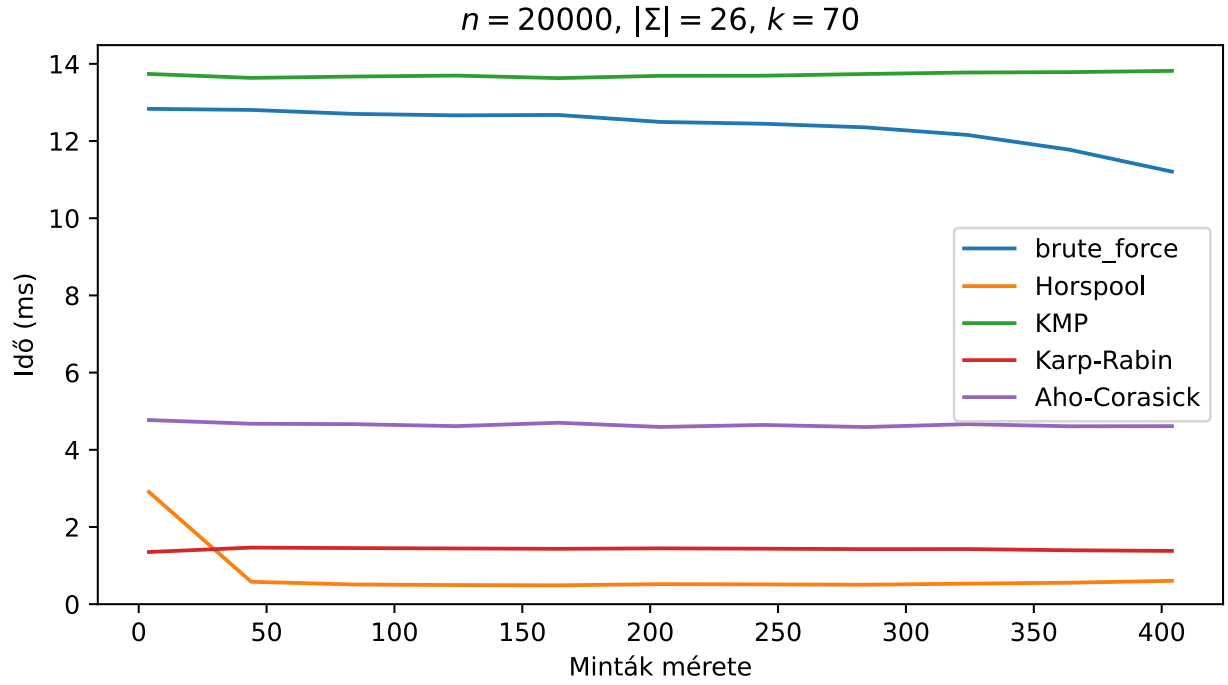
Itt is jól látszik, hogy minél több különböző mintát szeretnénk megtalálni a szövegben, annál lassabb megtalálni őket az egy mintát kereső algoritmusokkal, míg

az Aho–Corasick és a Karp–Rabin futásidejét a minták száma itt sem befolyásolja, sem rövid sem hosszú minták esetén.

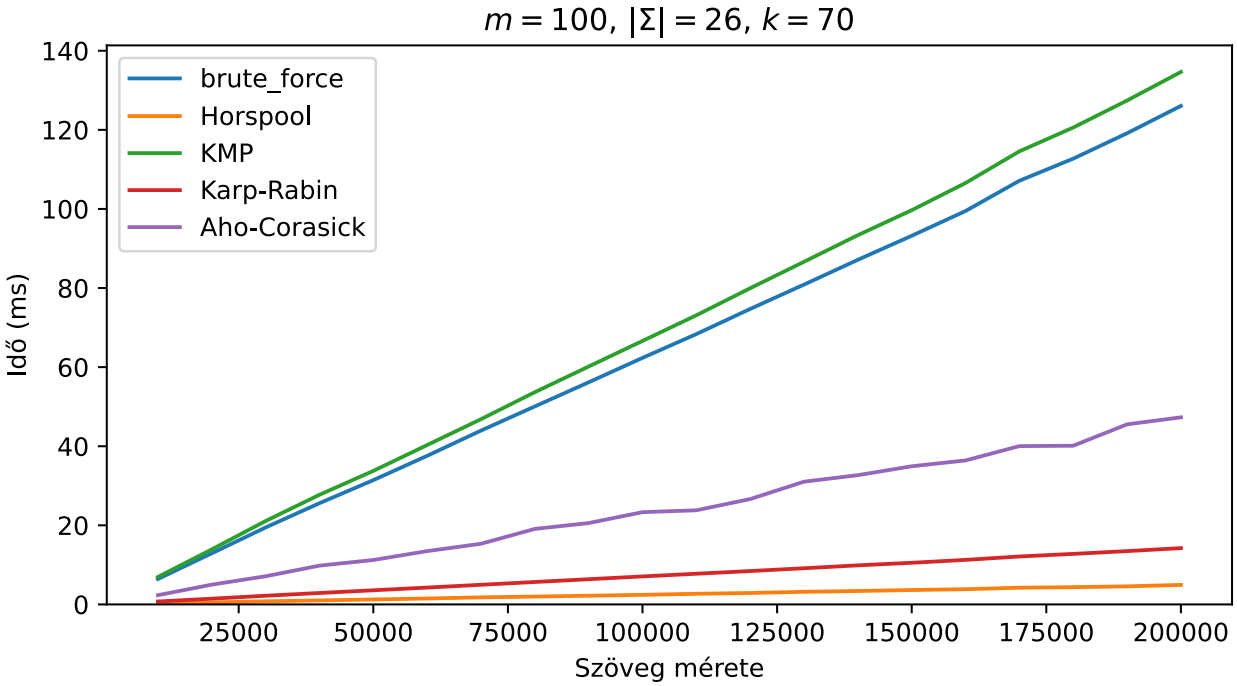
2.3.3. 26 elemű ábécé



2.8. ábra. A különböző algoritmusok futásideje angol ábécé esetén, 100 hosszú mintákkal, a minták számának függvényében



2.9. ábra. A különböző algoritmusok futásideje angol ábécé és 70 db minta esetén, a mintaméret függvényében



2.10. ábra. A különböző algoritmusok futásideje az angol ábécé és 50 db 100 hosszú minta esetén a szöveg hosszának függvényében

Ember által alkotott szövegekben több szó vagy kifejezés keresésére is nagyon jól működik az Aho–Corasick, itt is a szövegtől függ csak a sebessége. A Horspool

nagy ábécék esetén olyan gyors a gyakorlatban, hogy minden egyes mintára külön lefuttatva is nagyon jó eredményeket ér el, azonban a minták számának növelésével lineárisan nő a futási ideje.

2.3.4. Összegzés

Az Aho–Corasick futásideje valóban nem függ a minták számától, a szöveg méretétől függ leginkább a futásideje a gyakorlatban is. A Horspool még több minta esetén is nagyon jól teljesít, főleg ha nagy az ábécé, azonban minél több a minta, annál lassabb lesz. A Karp–Rabin hashmapes módosítása is nagyon jó a gyakorlatban több minta egyidejű keresésére, de az Aho–Corasickkel szemben megvan az a hátránya, hogy csak azonos hosszúságú minták esetén gyors, mivel minden különböző mintahosszra külön végig kell iterálnia a szövegen. A többi egyszerre egy mintát kereső algoritmus nem tudja felvenni a versenyt az Aho–Corasickkel, ha egyszerre több mintát szeretnénk megtalálni a szövegben.

3. fejezet

Mintaillesztés k-különbséggel

Ezt a fejezetet Alberto Apostolico és Zvi Galill Pattern Matching Algorithms [11] című tankönyve alapján írtam. A feladat hasonló a pontos egyezés alapú mintaillesztéshez (a $k = 0$ eset az a pontos egyezés). A célunk most is egy m hosszú s szó megkeresése egy n hosszú t szövegben, azonban megengedünk különbségeket is a minta és a találat között legfeljebb k darabot, (k is az input része). Fontos, hogy a szöveg egy részszeletének távolsága a mintától nagyban függ az általunk választott szó metrikától, különböző mértékek alapján a mintaillesztés teljesen különböző találatokat adhat. Ebben a fejezetben a Levenshtein-távolságát (szerkesztési távolságát) fogjuk vizsgálni a részszeleteknek a mintától. A megengedett különbségek:

(a) a minta egyik betűjéhez a szövegben egy másik betű tartozik,

(b) a minta egyik betűjéhez a szövegben nem tartozik betű,

(c) a szöveg egyik betűjéhez a mintában nem tartozik betű.

Van k-eltérés probléma is, ahol csak az (a) típusú különbséget engedjük meg, (azaz a Hamming-távolság alapján keresünk), ez egy egyszerűbb feladat, azonban mégsem ismerni gyorsabb algoritmust rá, mint a k-különbség problémára [11], amely egy általánosabb probléma.

3.1. Szó metrikák

Ebben az alfejezetben két szó metrikáról lesz szó, a Hamming-távolságról és a Levenshtein-távolságról. Közös lesz a távolságfüggvényekben, hogy teljesíteniük kell a metrikák alaptulajdonságait, azaz

1. Két szó távolsága pontosan akkor 0, ha a két szó megegyezik.

2. Szimmetrikus, azaz s_1 szó távolsága s_2-től egyenlő s_2 távolságával s_1-től.

3. Teljesíti a háromszög-egyenlőtlenséget, azaz s_1 távolsága s_3-tól nem lehet nagyobb, mint az s_1-s_2 és az s_2-s_3 távolságok összege.

3.1.1. Hamming-távolság

2. Definíció. [12] Két szó Hamming-távolságán az azonos pozíciókban lévő különböző betűk számát értjük.

b	u	d	a	p	e	s	t
b	u	k	a	r	e	s	t

3.1. ábra. „budapest” és „bukarest” Hamming-távolsága 2.

Tulajdonságok

A Hamming-távolság metrika mivel,

1. ha két szó távolsága 0 az azt jelenti, hogy nincs különböző betűjük ugyanabban a pozícióban, azaz a két szó megegyezik,

2. minden cserének egy csere az inverze, tehát szimmetrikus,

3. teljesíti a háromszög-egyenlőséget is, mivel ha u és v Hamming-távolsága x_1, illetve v és w Hamming-távolsága x_2, akkor u és w maximum $x_1 + x_2$ helyen különbözhet (azokon a helyeken, ahol u,v vagy v,w különbözött).

Kiszámítás

Két n hosszú szó Hamming-távolságának kiszámításához, mindenképp végig kell néznünk az összes betűpárt, tehát legalább $O(n)$ lépésre van szükség, és ennyi elég is. Szimplán végigiterálunk a két szón párhuzamosan és megszámoljuk a különbségeket.

3.1.2. Levenshtein-távolság

Két szó Levenshtein-távolsága [13] a legrövidebb átalakítás sorozat elemszáma, amellyel megkaphatjuk az egyik szóból a másikat. Megengedi a betűk kicserélését, mint a Hamming-távolság, de ezenkívül még megengedi betűk beszúrását, illetve törlését is, így különböző méretű szavak között is értelmezhető.

3. Definíció. Legyen a és b a két szavunk, jelölje egy n hosszú s szó $n-1$ hosszú végszeletét $tl(s)$,

$$lev(a,b) = \begin{cases} |a|, & \text{ha } |b| = 0 \\ |b|, & \text{ha } |a| = 0 \\ lev(tl(a), tl(b)), & \text{ha } a_1 = b_1 \\ 1 + \min \begin{cases} lev\big(tl(a), b\big) \\ lev\big(a, tl(b)\big) \\ lev\big(tl(a), tl(b)\big) \end{cases} & \text{egyébként} \end{cases}$$

Tulajdonságok

A definícióban minden eset megfeleltethető egy szerkesztésnek (vagy szerkesztéssorozatnak).

1. Ha b az üres szó, akkor triviális, hogy a betűit kell hozzáadnunk, és ennél kevesebb szerkesztés nem elég.

2. Ugyanez igaz, fordított esetben, mikor a az üres szó.

3. Ha a két kezdőbetű megegyezik, akkor elég a két maradék végszelet távolságát kiszámítani, mivel a két kezdőbetű elhagyása nem változtatja meg a többi betű relatív helyét a két szóban.

4. Ha nem egyezik meg a két kezdőbetű, akkor 1 lépésben ezt javítanunk kell, ez a lépés lehet a elejéről törlés, a elejére beszúrás, vagy a kezdőbetűjének átírása, ezek közül választjuk ki azt, ami optimálisabb.

A Levenshtein-távolság metrika, mivel

1. két szó távolsága csak akkor lehet 0, ha nincs különböző betűjük ugyanabban a pozícióban,

2. a törlésnek a beszúrás, a beszúrásnak a törlés és a cserének egy másik csere az inverze, tehát szimmetrikus,

3. teljesíti a háromszög-egyenlőtlenséget is, mivel ha u és v távolsága x_1, illetve v és w távolsága x_2, akkor u-ból megszerkeszthetjük v-t és abból megszerkeszthetjük w-t, tehát $x_1 + x_2$ szerkesztés biztosan elég.

Becslések

Legyen a n hosszú és b m hosszú szó, ekkor $\max(n, m) \geq lev(a, b) \geq |n - m|$, mivel maximum annyi betűt kell átírnunk/törölnünk, mint a hosszabb szó hossza, és legalább annyi betűt törölnünk kell, mint a két szó hosszának különbsége.
Két azonos méretű szó Levenshtein-távolsága legfeljebb annyi, mint a két szó Hamming-távolsága, mivel itt is végrehajthatjuk ugyanazokat a cseréket, mint amik a Hamming-távolságnál megengedettek, de itt törölhetünk és hozzáadhatunk is, így gyakran sokkal kisebb lesz a Levenshtein-távolság, mint a Hamming-távolság.

3.2. ábra. A „penge” és „enged” szavak Hamming-távolsága 5, míg a Levenshtein-távolságuk csak 2.

Kiszámítás

A definíció alapján rekurzív megvalósítással is kiszámítható, de ez nagyon lassú, mivel rengetegszer ki kell számítanunk ugyanannak a két szeletnek a Levenshtein-távolságát.
Dinamikus programozással [14] kiszámíthatjuk úgy, hogy egy M mátrixban tároljuk a megfelelő kezdőszeletek Levenshtein-távolságát. $M_{i,j}$-ben tároljuk az a i hosszú és a b j hosszú kezdőszeletének Levenshtein-távolságát. A mátrixot 0-tól indexeljük, a 0. sorát feltöltjük 0-tól n-ig az egész számokkal, mivel $lev(a, b) = |a|$, ha $|b| = 0$, ugyanígy a 0. oszlopot is feltöltjük. Ezután végigmegyünk fentről lefele, balról jobbra és kiszámoljuk a mátrixot a definíció szerint, ha a i. és b j. betűje megegyezik, akkor

$M_{i,j} = M_{i-1,j-1}$, ha nem akkor $M_{i,j} = \min(M_{i-1,j}, M_{i,j-1}, M_{i-1,j-1}) + 1$. Látszik, hogy adott érték kiszámításához csak a felette lévő sorra van szükség, ezért elég mindig csak a legutolsó két sort eltárolni, és így a szükséges tár $O(n)$ lesz $O(nm)$ helyett.

8. algoritmus Levenshtein-távolság kiszámítása dinamikus programozással

Funct LEV(a, b)

```
1:  for j = 0...n do
2:      M_1[j] := j                          ▷ Az első sora a mátrixnak, tehát |b| = 0
3:  end for
4:  for i = 1...m do
5:      M_2[0] := i                          ▷ Az első oszlopa a mátrixnak, tehát |a| = 0
6:      for j = 1...n do
7:          if b_i = a_j then
8:              M_2[j] := M_1[j − 1]
9:          else
10:             M_2[j] := min(M_1[j], M_1[j − 1], M_2[j − 1]) + 1
11:         end if
12:     end for
13:     M_1 := M_2                           ▷ M_2-t (az i. sort) betöltjük M_1-be
14: end for
15: return M_2[n]
```

3.2. k-különbség probléma megoldása dinamikus programozással

Ha az előbb vizsgált Levenshtein-távolságot kiszámító dinamikus programozási algoritmusban, a minta 0 hosszú kezdőszeletéhez tartozó sort 0-kal töltjük fel, (ahelyett hogy 0-tól n-ig az egész számokkal), akkor az algoritmus az $M_{i,j}$-be a legkisebb távolságot írja, mely $s_1, s_2 \dots s_i$ és a szöveg egy t_j-ben végződő szelete között lehet, azaz ha $M_{m,j} \leq k$, akkor van egy találatunk, melynek t_j-ben van a vége. Ez azért működik, mert azzal, hogy csupa 0-t írtunk a 0 hosszú mintakezdőszelethez tartozó sorba, 0 értéket adtunk azoknak a szerkesztéseknek, melyek a szöveg egy kezdőszeletének elejéről törlik a betűket.

Ebben az alfejezetben egy alternatív dinamikus programozási algoritmust [11] írok le, mely ugyanennek a mátrixnak az értékeit számítja ki, de az átlók mentén, ezt a logikát követi majd az ezután következő algoritmus is.

4. Definíció. Az M mátrix d. átlójának elemei $\{M_{i,j} : j - i = d\}$.

5. Definíció. Minden ℓ különbségre és d. átlóra, $L_{d,\ell}$ legyen a legnagyobb sor, melyre $M_{i,j} = \ell$ és $M_{i,j}$ a d. átlón van. $L_{d,\ell} = \max\{i : M_{i,d+i} = \ell\}$.

A definícióban megadott feltételek miatt, ℓ a legkisebb távolság $s_1 \dots s_{L_{d,\ell}}$ és a szöveg bármely $t_{L_{d,\ell}+d}$-ben végződő kezdőszelete között, illetve mivel $L_{d,\ell}$ a legnagyobb sor melyre teljesülnek a feltételek, ezért $s_{L_{d,\ell}+1} \neq t_{L_{d,\ell}+d+1}$. Számunkra azon $L_{d,\ell}$-k érdekesek, melyeknél $\ell \leq k$. Ha egy $L_{d,\ell} = m$, az azt jelenti, hogy $t_{L_{d,\ell}+d}$-ben van vége egy találatnak.

3.2.1. Leírás

Az algoritmus az $L_{d,\ell}$-ket számítja ki úgy, hogy felteszi minden $x < \ell$-re és y-ra, $L_{y,x}$ már megvan. Ha $L_{d,\ell} = i$, akkor tudjuk, hogy $M_{i,d+i} = \ell$ és a 8. algoritmusból tudjuk, hogy ezt az értéket kétféleképpen kaphattuk:

(a) $s_i \neq t_{d+i}$ és ($M_{i-1,d+i-1} = \ell - 1$ vagy $M_{i-1,d+i} = \ell - 1$ vagy $M_{i,d+i-1} = \ell - 1$),

(b) $s_i = t_{d+i}$ és $M_{i-1,d+i-1} = \ell$.

Ebből látszik, hogy $L_{d,\ell}$ értéke legalább $\max(L_{d,\ell-1} + 1, L_{d-1,\ell-1}, L_{d+1,\ell-1})$, mivel ha $L_{d,\ell-1} = x$, akkor a definíció miatt mivel az a legnagyobb sor M-ben $l - 1$ értékkel a d. átlón, ezért az $x + 1$ sorban biztosan l van, ha $L_{d-1,\ell-1} = x$, akkor M-ben a $d - 1$. átlón az x. sorban $l - 1$ van, tehát ugyanezen sorban mellette legfeljebb l lehet, $L_{d+1,\ell-1} = x$-re is hasonlóan láthatjuk ezt. Ezért erre a $\max(L_{d,\ell-1}+1, L_{d-1,\ell-1}, L_{d+1,\ell-1})$ értékre beállítjuk a *sor* nevű változót és a d. átlón addig lépkedünk lefele amíg (b) fentáll (mivel ekkor nem nő az ℓ érték), és minden lépésben a *sor*-t növeljük 1-gyel, amint $s_i \neq t_{d-i}$ megállunk, $L_{d,\ell} := sor$. Az algoritmus elején inicializálni kell az alapeseteket, melyekből számolni fogjuk a többi értéket.

A $d < -k$ átlók nem lesznek hasznosak, mivel ezen átlók értékei a szöveg maximum $m - k - 1$ hosszú kezdőszeleteihez tartoznak, tehát a távolságuk a mintától mindenképp több, mint k.

		b	a	a	b	c	c	c	c	b	b	b	a	a
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	0	1	1	0	1	1	1	1	0	0	0	1	1
b	2	1	1	2	1	1	2	2	2	1	0	0	1	2
a	3	2	1	1	2	2	2	3	3	2	1	1	0	1
c	4	3	2	2	2	2	2	2	3	3	2	2	1	1

3.1. táblázat. *bbac* szó és a *baabccccbbbaa* szöveg mátrixa

9. algoritmus Az $L_{d,\ell}$ értékek kiszámítása dinamikus programozással

Funct ATLOK(a, b)

1: **for** $d = 0...n$ **do**
2: $\quad L_{d,-1} = -1$ ▷ így 0-ról fog kezdődni $L_{d,0}$ értékének keresése
3: **end for**
4: **for** $d = -(k+1)... - 1$ **do**
5: $\quad L_{d,|d|-1} = |d| - 1$
6: $\quad L_{d,|d|-2} = |d| - 2$
7: **end for**
8: **for** $\ell = -1...k$ **do**
9: $\quad L_{n+1,\ell} = -1$
10: **end for**
11: **for** $\ell = 0...k$ **do**
12: $\quad$ **for** $d = -\ell...n$ **do**
13: $\quad\quad sor := \max(L_{d,\ell-1} + 1, L_{d-1,\ell-1}, L_{d+1,\ell-1})$
14: $\quad\quad$ **while** $sor < m$ & $sor < n - d$ & $a_{sor+1} = t_{sor+1+d}$ **do**
15: $\quad\quad\quad sor := sor + 1$ ▷ amíg nem nő a távolság, addig növeljük
16: $\quad\quad$ **end while**
17: $\quad\quad L_{d,\ell} = sor$
18: $\quad\quad$ **if** $L_{d,\ell} = m$ **then**
19: $\quad\quad\quad$ **print** $d + m$ ▷ a találat végének indexét írjuk ki
20: $\quad\quad$ **end if**
21: $\quad$ **end for**
22: **end for**

3.2.2. Az algoritmus helyessége

$\ell = 0$ esetén $sor = 0$ inicializáláskor, a 13. sorban lévő utasítás miatt. Ezután a 14. sorban a ciklus addig növeli a sor értékét, amíg $a_1, a_2 \dots a_{L_{d,0}} = t_{d+1}, t_{d+2} \dots t_{d+L_{d,0}}$ teljesül, ha a következő karakter már nem stimmel akkor kilép a ciklusból, tehát $L_{d,0}$ a megfelelő értéket kapta, mert a következő sorban ezen az átlón már egy nagyobb távolság van. Indukcióval $\ell = \ell_0$ esetén tegyük fel, hogy már

$\forall d : L_{d,\ell_0-1}$-nek helyes az értéke, ekkor 13. sor miatt *sor* a legnagyobb sor a d átlón amire igaz, hogy $M_{sor,d+sor} = \ell_0$ értéket kaphatott az (a) feltétel által, tehát innen kell indítanunk a keresést, és a következő ciklus meg is találja a legnagyobb megfelelő sort, ugyanúgy ahogy $L_{d,0}$-nál. Még kell, hogy a mátrix határainak inicializálása helyes, az $L_{d,-1}$-t -1-re állítjuk minden 0 és n közti d-re, hogy 0-ról kezdődjön az $L_{d,0}$ értékének keresése, az $L_{-d,|d|-1}$ értékeket $|d|-1$-re, az $L_{-d,|d|-2}$ értékeket $|d|-2$-re, hogy az $L_{-\ell,\ell}$ értékek keresése legalább $|d|-1$-ről, az $L_{-\ell,\ell+1}$ értékek keresése pedig legalább $|d|-2$-ről kezdődjön. Azért állítjuk, az $L_{n+1,\ell}$ értékeket -1-re, hogy minden $L_{\ell,n}$ érték keresése 0-tól induljon.

3.2.3. Lépésszám

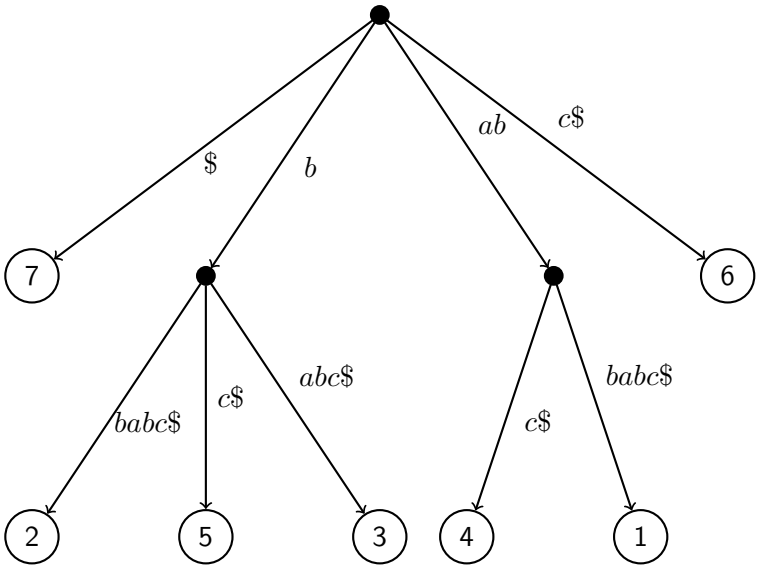
A lépésszáma ugyanannyi, mint a korábbi dinamikus programozási algoritmusnak, $n+k+1$ átlón számítjuk ki az $L_{d,\ell}$-ket, és legfeljebb m féle értéket kaphat a *sor* változó, így $O((n+k+1)\cdot m) = O(nm)$ lépésszámú lesz.

3.3. A hatékony algoritmus

A hatékony algoritmus [11] az előző algoritmus futásidejét $O(nk)$-ra javítja, $O(n+m)$ lépésszámú előfeldolgozás mellett. Az algoritmusnak két része van, először a szöveg és a minta egymásután írásával létrehozunk egy új stringet és elkészítjük ennek a stringnek az végszelet-fáját. Az algoritmus második felében a korábbi dinamikus programozási algoritmushoz hasonlóan ki fogjuk számítani az $L_{d,\ell}$ értékeket, azonban itt a végszelet-fa segítségével visszavezetjük a „legalacsonyabb közös ős" (LCA) feladatra.

3.3.1. Végszelet-fa

6. Definíció. Egy m hosszú s szó végszelet-fája, egy olyan fa melynek m levele van 1-től m-ig számozva, minden éléhez s egy nem üres szelete tartozik, méghozzá úgy, hogy a fában a gyökértől az i. levélhez tartozó úton az éleket összeolvasva $s_i, s_{i+1} \dots s_m$-t kapjuk. Ezenkívül az egy csúcsból kimenő élekhez nem tartozhat ugyanolyan kezdőbetűjű szelet, és minden nem gyökér és nem levél csúcsnak legalább két gyereke van.



3.3. ábra. Az *abbabc*$ szó végszelet-fája

Megjegyzés. Nem levél csúcsa a fának maximum m db lehet, mivel minden nem levél és nem gyökér csúcsnak legalább két gyereke van,

3.3.2. Leírás

Elsőként megépítjük az $s+t$ végszelet-fáját, ezután az algoritmus ugyanaz, mint az előző fejezetben lévő, leszámítva a ciklust amivel növeljük a *sor* változót. A kiindulási értéke itt is $sor := \max(L_{d,\ell-1}+1, L_{d-1,\ell-1}, L_{d+1,\ell-1})$ lesz, és itt is keressük a maximális q-t, amire $s_{sor+1} \dots s_{sor+q} = t_{d+sor+1} \dots t_{d+sor+q}$ teljesül. A ciklus helyett q-t úgy találjuk meg, hogy a végszelet-fában megkeressük a sor. levél és a d. levél legközelebbi közös ősét.
Sorban hozzáadhatnánk a végszeleteket a fához, ahogy a 2. fejezetben a faépítő algoritmus is tette (így $O(n^2)$ lépésszámú lenne), de akkor nem használnánk ki, hogy ezek a végszeletek tartalmazzák egymást.

3.3.3. McCreight algoritmusa [15] a fa megépítésére

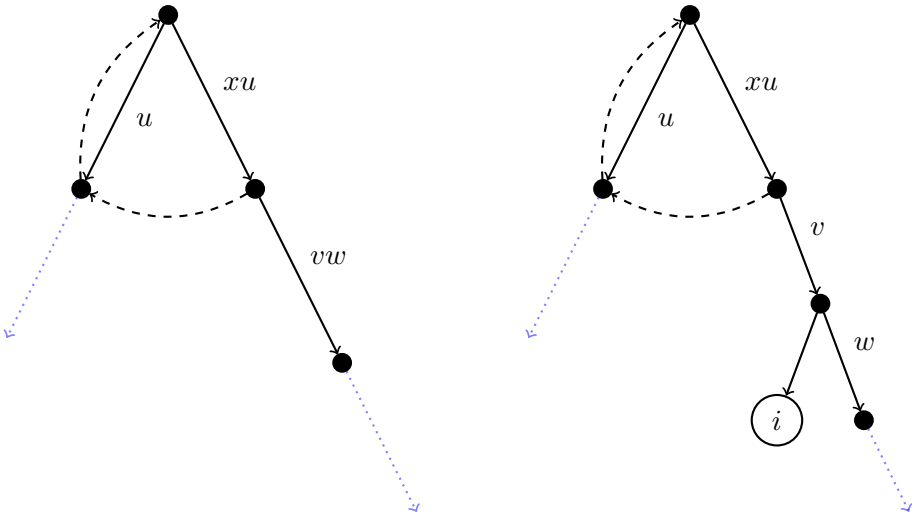
Legyen az s m hosszú szó végszelet-fája T. Jelölje $\underline{v}$ a p csúcsot, ha a gyökérből a p csúcsba vezető úton kiolvasható szelet v. Fontos, hogy a szó utolsó betűje egyedi legyen, különben lenne olyan végszelete s-nek, mely egy másik végszeletének a kezdőszelete, tehát nem egy levélben végződne. Jelölje suf_i az $s_i \dots s_m$ végszeletet, és ennek a végszeletnek a feje legyen a leghosszabb kezdőszelete, amely s egy másik hosszabb végszeletének is a kezdőszelete, suf_i töve pedig legyen a nem fej része.

Például $s = abbabc\$$, $suf_4 = abc\$$ feje ab, mivel $s_1 s_2 = ab$ és $s_3 \neq c$. Az algoritmus végigiterál s végszeletein, az i. körben hozzáadja suf_i-t a gráfhoz, méghozzá úgy, hogy ne duplázza meg egyik kezdőszeletét sem a fában, ezért kell a feje, a leghosszabb kezdőszelete, ami már benne van a fában. Jelölje T_i a fa állapotát az i. iteráció után. A fa építésénél az élekre nem fogjuk ráírni a hozzájuk tartozó $s_i \dots s_j$ szeletet, hanem csak egy indexpárt (i, j), ami a szelet elejét és végét jelöli. Ez azért jó, mert így konstans időben tudunk hozzáadni élt vagy csúcsot a fához.

1. Tétel. *Ha suf_{i-1} feje xu, ahol x egy betű és u egy tetszőleges szelet, akkor u suf_i fejének a kezdőszelete.*

Bizonyítás. Ha suf_{i-1} feje xu, akkor a definícióból következik, hogy $\exists j : suf_{j-1}$ kezdőszelete xu, emiatt u kezdőszelete suf_j-nek (és triviális, hogy kezdőszelete suf_i-nek is). Tehát mivel van egy hosszabb végszelet, aminek a kezdőszelete u, ezért u biztosan benne lesz suf_i fejében. □

Ezt a tulajdonságot segédélekkel használjuk ki, úgy hogy $\underline{xu}$-ból segédélet húzunk $\underline{u}$-ba, lehetséges, hogy u-t kiolvasva a fán, nem egy csúcsban ér véget, hanem egy él közepén, ekkor ezen él elejébe húzzuk a segédélet. Ez hasznos lesz, mert így az i. körben meg tudjuk találni suf_i fejének a kezdőszeletét, suf_{i-1} fejének csúcsából kimenő segédél végén. Mivel a keresett fej kezdőszelete már megvan, innentől csak rekurzívan továbblépkedünk lefele a fában, amíg az egész fej meg nem lesz, ha nincs ilyen akkor beszúrunk egy új csúcsot. Ha megtaláltuk a fejet, akkor húzunk belőle egy új élet egy levélbe, az élre suf_i tövét írjuk.



3.4. ábra. T_{i-1}-be beillesztjük suf_i-t. suf_{i-1} feje xuv, tehát uv suf_i fejének kezdőszelete.

3.3.4. A legközelebbi közös ős feladat megoldása

7. Definíció. Az LCA-feladat [16] (Legközelebbi Közös ős feladat), hogy egy fában adott két csúcs u és v, akkor $LCA(u, v)$ visszaadja u és v legközelebbi közös ősét, ez a csúcs a gyökértől a legtávolabbi közös ősük.

Az RMQ [16] avagy Intervallum minimum lekérdezés feladat

Van egy n méretű A tömbünk, a feladat, hogy egy $RMQ(i, j)$ kérdésre megadjuk azt a $i \leq k \leq j$ indexet, amire $A[k]$ minimális.
A feladat egy speciális esete a $\pm 1 - RMQ$ feladat, mikor a tömb szomszédos elemeinek különbsége ± 1, azaz $\forall\ 2 \leq i \leq n : |A[i] - A[i-1]| = 1$.

2. Tétel. *Ha a $\pm 1 - RMQ$ feladatra van (O(n), O(1)) megoldásunk, akkor az LCA feladatra is kaphatunk (O(n), O(1)) megoldást [16].*

Bizonyítás. A fa éleit megduplázzuk és keresünk benne egy Euler-kört a gyökérből, ilyen biztosan létezni fog, mivel a párhuzamos élek miatt minden csúcs fokszáma páros lesz. Az Euler-kör csúcsait feljegyezzük sorban egy E tömbbe, $2 \cdot (n-1)$ éle lesz az új gráfnak és még a végén újra belerakjuk a gyökért, tehát a tömb $2n-1$ méretű lesz. Feljegyezzük a csúcsok mélységét is egy L tömbben a bejárás közben ($L[i] := d(E[i], gyoker)$). Ezenkívül egy R n méretű tömbben eltároljuk minden csúcshoz az első előfordulásának az indexét E-ben. Könnyen látható, hogy $E[RMQ_L(R[u], R[v])]$

u és v legközelebbi közös őse, mivel ez az Euler-körben a gyökérhez legközelebbi csúcs az u és v közötti szakaszon. Az L tömbre teljesül $\pm 1 - RMQ$ feladat feltétele, mivel az új gráfban a szomszédos csúcsok szintkülönbsége ± 1 az eredeti fában. □

3. Tétel. *Létezik (O(n), O(1)) megoldás a* $\pm 1 - RMQ$ *feladatra.*

A bizonyítás megtalálható Király Zoltán Adatstruktúrák [16] című jegyzetében.

3.3.5. Lépésszám

A csúcsok és élek beszúrása konstans idejű és maximum $2(n+m)$ csúcs lehet a fában, mivel minden belső csúcsnak legalább két gyereke van.
Ha az i. körben a fej keresésekor áthaladunk egy csúcson, akkor az $i+1$. körben azt a csúcsot biztosan nem fogjuk megvizsgálni, mivel minden i. körben vizsgált csúcs már benne volt T_{i-1}-ben, ezért van már segédél, ami kimegy belőle. Az i. körben a fej keresésénél $|fej(suf_1)| - |fej(suf_{i-1})| + 1$ betűt vizsgálunk meg (a segédélnek köszönhetően). Tehát összesen

$$\sum_{i=0}^{n+m} |fej(suf_i)| - |fej(suf_{i-1})| + 1 = suf_{n+m} - suf_0 + n = 0 - 0 + n + m$$

összehasonlítást végzünk maximum a fejkereséseknél.
Az LCA feladatot megoldó algoritmus $O(n+m)$ feldolgozás után, minden lekérdezésre konstans időben tud válaszolni, így minden legalacsonyabb közös őst konstans időben meg tudunk találni. Összesen $n+k+1$ átlón számolunk ki k db $L_{d,l}$ értéket, tehát ez összesen $O(nk)$ idő lesz, és ez az idő dominálja az algoritmus többi szakaszának idejét, tehát a hatékony algoritmus futásideje csak a szöveg hosszától és a különbségek számától függ.

3.4. A dinamikus programozási algoritmus a gyakorlatban

A dinamikus programozási algoritmust itt a brute force algoritmussal hasonlítom össze, bár a brute force csak pontos egyezéseket talál meg.

Levenshtein-távolság	brute_force	Dinamikus
2	0.28	29.89
4	0.28	29.62
6	0.29	29.91
8	0.28	30.15
10	0.28	30.35
12	0.29	30.77
14	0.28	29.29
16	0.28	30.09
18	0.30	30.38
20	0.28	30.56

3.2. táblázat. A dinamikus programozási algoritmus futásideje angol ábécé, 30000 hosszú szöveg és 50 hosszú minta esetén

Mintaméret	brute_force	Dinamikus
40	0.28	23.73
80	0.28	46.72
120	0.27	68.51
160	0.27	92.30
200	0.27	114.94
240	0.27	137.54
280	0.27	159.21
320	0.27	181.82
360	0.27	205.22
400	0.26	227.09

3.3. táblázat. A dinamikus programozási algoritmus futásideje angol ábécé, 30000 hosszú szöveg és legfeljebb 3 Levenshtein-távolság esetén

Szövegméret	brute_force	Dinamikus
10000	0.09	19.60
20000	0.18	38.93
30000	0.28	58.50
40000	0.37	78.84
50000	0.47	99.98
60000	0.56	119.11
70000	0.65	137.15
80000	0.75	161.44
90000	0.86	188.24
100000	0.94	203.27

3.4. táblázat. A dinamikus programozási algoritmus futásideje angol ábécé 100 hosszú minta és legfeljebb 3 Levenshtein-távolság esetén

ábécéméret	brute_force	Dinamikus
4	0.42	36.62
8	0.34	38.07
12	0.31	33.51
16	0.29	31.10
20	0.28	30.24
24	0.28	29.75
28	0.28	29.68
32	0.27	29.14
36	0.28	29.63
40	0.29	30.33

3.5. táblázat. A dinamikus programozási algoritmus futásideje 30000 hosszú szöveg, 50 hosszú minta és legfeljebb 3 Levenshtein-távolság esetén

Jól látszik, hogy a dinamikus programozási algoritmus sokkal lassabb még a brute force-nál is, mivel a pontos találatokon kívül megtalál a mintához hasonló

szövegrészleteket is.

A futásidejét csupán a szöveg és a minta mérete befolyásolja, a megengedett k, illetve az ábécé mérete nem.

Irodalomjegyzék

[1] Peyman Neamatollahi, Montassir Hadi és Mahmoud Naghibzadeh. „Simple and efficient pattern matching algorithms for biological sequences". *IEEE Access* 8 (2020), 23838–23846. old.

[2] Tsern-Huei Lee. „Generalized aho-corasick algorithm for signature based anti-virus applications". *2007 16th International conference on computer communications and networks*. IEEE. 2007, 792–797. old.

[3] Bing Liu, Robert Grossman és Yanhong Zhai. „Mining data records in Web pages". *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2003, 601–606. old.

[4] R Nigel Horspool. „Practical fast searching in strings". *Software: Practice and Experience* 10.6 (1980), 501–506. old.

[5] Donald E Knuth, James H Morris Jr és Vaughan R Pratt. „Fast pattern matching in strings". *SIAM Journal on Computing* 6.2 (1977), 323–350. old.

[6] Richard M Karp és Michael O Rabin. „Efficient randomized pattern-matching algorithms". *IBM Journal of Research and Development* 31.2 (1987), 249–260. old.

[7] Edvard Ehler és tsai. „AmtDB: a database of ancient human mitochondrial genomes". *Nucleic Acids Research* 47.D1 (2018. szept.), D29–D32. old.

[8] William Shakespeare. *Romeo and Juliet*. 1595. URL: `https://www.gutenberg.org/files/1112/1112.txt`.

[9] Alfred V Aho és Margaret J Corasick. „Efficient string matching: an aid to bibliographic search". *Communications of the ACM* 18.6 (1975), 333–340. old.

[10] *GNU Grep: Print lines matching a pattern*. URL: `https://www.gnu.org/software/grep/manual/`.

[11] Alberto Apostolico és Zvi Galil. *Pattern matching algorithms*. Oxford University Press on Demand, 1997.

[12] Richard W Hamming. „Error detecting and error correcting codes". *The Bell System Technical Journal* 29.2 (1950), 147–160. old.

[13] Vladimir I Levenshtein. „Binary codes capable of correcting deletions, insertions, and reversals". *Soviet Physics Doklady*. 10. köt. 8. 1966, 707–710. old.

[14] Robert A Wagner és Michael J Fischer. „The string-to-string correction problem". *Journal of the ACM* 21.1 (1974), 168–173. old.

[15] Edward M McCreight. „A space-economical suffix tree construction algorithm". *Journal of the ACM* 23.2 (1976), 262–272. old.

[16] Király Zoltán. *Adatstruktúrák*. `http://zkiraly.web.elte.hu/Adatstrukturak.pdf`. 2022.

Ábrák jegyzéke

Táblázatok jegyzéke

Algoritmusjegyzék