

SZEGEDI TUDOMÁNYEGYETEM

Természettudományi és Informatikai Kar

Számítógépes Algoritmusok és Mesterséges Intelligencia Tanszék

Szakdolgozat

String matchig algoritmusok

Kardos Bálint

programtervező informatikus

Témavezető: Csirik János, Professzor Emeritus

Szeged

2023

Feladatkiírás

A szakdolgozat célja a String matching algoritmusok megismerése és bemutatása. A dolgozat ismertesse a legfontosabb és legismertebb algoritmusokat, amelyek segítségével szövegben lehet mintázatokot keresni és azonosítani. Az algoritmusok működését és alkalmazási területeit is vizsgálja meg.

Tartalmi összefoglaló

- **A téma megnevezése:**

String matching algoritmusok.

- **A megadott feladat megfogalmazása:**

A szakdolgozat célja a String matching algoritmusok megismerése és bemutatása. A dolgozat ismertesse a legfontosabb és legismertebb algoritmusokat, amelyek segítségével szövegben lehet mintázatokat keresni és azonosítani. Az algoritmusok működését és alkalmazási területeit is vizsgálja meg.

- **A megoldási mód:**

Ismertettem mi a string matching probléma és hol használják. Ezután leírtam a naiv algoritmus elméletét, és hogy miért is kell ezt gyorsítani. Tárgyaltam kicsit okosabb változatát, mint például a Not so naiv vagy a Hamming távolságos naiv algoritmus. Ezek után részleteztem a Boyer-Moore algoritmust, Knuth-Morris-Pratt algoritmust, az Apostolico-Giancarlo algoritmust, az Aho-Corasick algoritmust, illetve két metódust, ami bit operandusokat használ a keresésre, mint a Shift-or vagy a Karp and Rabin algoritmus. Suffix fát ismertettem, és hogy ez miért hasznos string matching-hez. Részleteztem a program felépítését, kinézetét. A program működését és a megvalósított algoritmusokat részletesen ismertettem.

- **Alkalmazott eszközök, módszerek:**

A JavaScript nyelven történő algoritmusok implementációját követően létrehoztam egy weboldalt, amely az algoritmusok eredményeit szemlélteti és lehetővé teszi a használatukat. Az algoritmusokat harmonikusan integráltam a weboldalba, melyet HTML és CSS segítségével formáztam meg. A felhasználói felület dinamikusságát és egyszerűségét React.js keretrendszerrel értem el, lehetővé téve a teljes funkcionalitás megjelenítését egyetlen oldalon.

- **Elért eredmények:**

Egy weblap, ahol ki lehet választani a kereső algoritmust (Naiv, Not So Naiv, Naiv H, Boyer-Moore, Knuth-Morris-Pratt, Aho-Corasick, Shift-or, Karp and Rabin). Meg lehet adni a szöveget amin szeretnénk keresni, és azt a szöveget amit szeretnénk keresni, hogy megtalálható-e a szövegben. A honlap elvégzi a keresést, megadja a keresés idejét, és kiemeli a szövegben a talált mintát.

- **Kulcsszavak:**

Naiv, Not So Naiv, Hamming távolság, Boyer-Moore, Knuth-Morris-Pratt, Apostolico-Giancarlo, Aho-Corasick, Shift-or, Karp and Rabin, Suffix fa, string matching

Tartalomjegyzék

Feladatkiírás.....	1
Tartalmi összefoglaló.....	2
Tartalomjegyzék	3
1. Bevezetés	5
1.1 Pontos illesztés: mi is a probléma?	5
1.2 A pontos illesztés fontossága	6
1.3 Naiv algoritmus.....	6
1.4 A Naiv algoritmus felgyorsítása	7
1.5 Előfeldolgozási megközelítés	7
1.6 Not So Naive algoritmus.....	7
1.7 Naiv algoritmus Hamming távolsággal	8
2. Klasszikus összehasonlításra alapuló módszerek	9
2.1 Bevezető.....	9
2.2 Boyer-Moore algoritmus.....	9
2.3 Knuth-Morris-Pratt algoritmus	11
2.4 Apostolico-Giancarlo algoritmus.....	13
2.5 Aho-Corasick algoritmus	16
3. Seminumerical string matching	18
3.1 Bevezető.....	18
3.2 Shift-or módszer.....	18
3.3 Karp and Rabin	20
4. Suffix fa és használata	22
4.1 Bevezető.....	22
4.2 Suffix fa	23
5. A program bemutatása	24

5.1 A program felépítése.....	24
5.2 Program kinézete	26
5.3 Naiv algoritmus.....	26
5.4 Naiv H algoritmus.....	26
5.5. Not So Naiv algoritmus	27
5.6 Boyer-Moore algoritmus.....	27
5.7 Knuth-Morris-Pratt algoritmus	28
5.8 Aho-Corasick algoritmus	28
5.9. Shift-Or algoritmus	30
5.10. Karp and Rabin algoritmus	30
6. A program teljes futására egy példa.....	31
7. Összefoglalás	33
Nyilatkozat.....	34
Irodalomjegyzék	35
Köszönetnyilvánítás.....	36
Melléklet útmutató	37

1. Bevezetés

A stringek az információ tárolásának és feldolgozásának egyik legegyszerűbb, és az egyik legolcsóbb módja, ezért a mai napig nagyon elterjedt tárolási módszer. A string egy karakter sorozat lánc. A string algoritmusok pedig olyan algoritmusok, amiket stringek kezelésére terveztek.

A string algoritmusok a mai napig fontos szerepet játszanak a mindennapi életben pont azért, mert hatékonyak és gyorsak. Webes keresés esetén, gépi tanulás során, szöveg feldolgozása közben, antivírusoknál vagy éppen DNS-szekvenálásnál, és még sok más helyen is találkozhatunk string algoritmusokkal.

A szakdolgozat célja, hogy bemutassa a string algoritmusokat, legnagyobb hangsúlyt a string illesztő eljárásokra fekteti. Ezeket foglalja össze és mutatja be.

1.1 Pontos illesztés: mi is a probléma?

A mintaillesztés lényege az, hogy egy adott szövegben vagy sorozatban megkeressük egy másik, általában rövidebb szövegminta előfordulásait. Ezt a folyamatot gyakran stringkeresésnek is nevezik. Az ilyen típusú keresést általában arra használjuk, hogy megtaláljuk egy adott minta pontos helyét vagy az összes előfordulását egy hosszabb szövegben vagy sorozatban.

Fontos bevezetni alapfogalmakat. A dolgozatban a keresett mintára P -vel, a szövegre, amiben keresünk meg T -vel fogok hivatkozni. P hosszára n -el, és T hosszára pedig m -el.

A minta előfordulásának fogalma: Ha k egy szám 0 és $n-m$ közötti. Akkor azt mondjuk, hogy P minta illeszkedik a $k+1$ -dik pozíción illeszkedik a T szövegre, vagy a P minta k eltolással illeszkedik T -re, illetve k érvényes eltolás, ha $T[k+1 \dots k+m] = P[1 \dots m]$.

Legyen például a $P=aba$ és $T=bbabaxababay$ akkor P megtalálható a T -ben a 3., 7. és 9. pozíciókban. P két előfordulása átfedésben van egymással, mint itt a 7 és 9 helyen, itt $k=6$ és 8 Itt $m=3$ és $n=12$.

1.2 A pontos illesztés fontossága

A pontos illesztés problémája mindenkinek alapvetőnek kellene lennie, aki használt valaha számítógépet. A probléma nagyon változatos alkalmazásokban merül fel, amelyet nehéz lenne teljes egészében felsorolni. Néhány gyakori felhasználása a szövegfeldolgozás, a könyvtárak kereső programjai, internetes keresőprogramok, egy szimpla oldalon való keresés ctrl+F. A molekuláris biológiában több száz speciális adatbázis található, nyers DNS, RNS és aminosav karakterláncok, vagy a nyers karakterláncból származó feldolgozott minták. Bár a tipikus szöveg feldolgozásban mára valószínűleg elég kevés probléma maradt. Az exact matching megoldott erre a célra. De a történet változik más felhasználásnál. Az algoritmusok működésének megértése a kulcs ahhoz, hogy a feladatot hatékonyan oldjuk meg.

1.3 Naiv algoritmus

Általában az exact matching problémát a naiv módszerrel szoktuk kezdeni, ezért én is így teszek. Mivel egy nagyon szimpla és buta algoritmusról van szó, ezért egyszerű-mintaillesztésnek vagy brute force-nak is szokták hívni. Az algoritmus a szöveg 0 és n közötti minden pontján ellenőrzi, hogy a minta előfordulása ott kezdődik-e. Ha nem akkor egyet lép előre a szövegben és újra próbálkozik. Ha igen, akkor a szöveg következő elemét a minta következő elemével hasonlítja össze, amíg igaz. Ha megtalálja a mintát kiírja, ha nem akkor a szövegbe megint egyet lép, míg a szöveg végéig nem ér. Ez egy hosszú folyamat, legrosszabb esetben $O(m*n)$ a futási ideje, ahol n a szöveg mérete és m a minta mérete. Tárhely igénye $O(1)$. (Farkas, 2017) (Charras & Lecroq, Brute force algorithm, 1997)

EGYSZERŰ-MINTAILLESZTŐ(T, P)

```
1   $n \leftarrow \text{hossz}[T]$ 
2   $m \leftarrow \text{hossz}[P]$ 
3  for  $s \leftarrow 0$  to  $n - m$ 
4      do if  $P[1..m] = T[s + 1..s + m]$ 
5          then print „A minta illeszkedik az („ $s + 1$ „)-edik pozícióra”
```

1.3.1. ábra: Naiv algoritmus

(Farkas, 2017)

1.4 A Naiv algoritmus felgyorsítása

A Naiv algoritmus egy egyszerű, de időigényes algoritmus, ezért fel lehet gyorsítani úgy, hogy minél kevesebb lépésből jussunk el a végeredményhez. Azt szeretnénk elérni, hogy minél nagyobb részeket tudjunk átugrani a szövegben úgy, hogy közben véletlenül se ugorjunk át illeszkedést. Ha többet lépünk előre, mint egy pozíció, az gyorsítja az összehasonlítást, mivel gyorsabban halad P T -ben. Némelyik algoritmus úgy próbál elérni nagy ugrásokat, hogy a mintában próbál meg ugrani minden összehasonlítás után. Ezeket fogom a következőkben bővebben is kifejteni.

1.5 Előfeldolgozási megközelítés

Az egyik lehetséges módszer a folyamat felgyorsítására az, ha két részre osztjuk a problémát. Először viszonylag kevesebb időt töltve feldolgozzuk a mintát vagy a szöveget. A folyamat során az algoritmusnak nem is kell tudnia a másik stringről. Ezt a részét nevezzük előfeldolgozási szakasznak. Ezt követően a keresési szakaszba lép az algoritmus. Az ilyen előfeldolgozás hasznos lehet azért is, mert ha egy mintát vagy egy szöveget többször akarunk felhasználni, akkor az előfeldolgozási szakaszt az első után meg tudjuk spórolni, ezzel is gyorsítva a keresést.

1.6 Not So Naive algoritmus

A naiv algoritmust fel lehet gyorsítani egy minimális minta előfeldolgozással. A Not So Naive algoritmus ezt úgy éri el, hogy a keresési fázisban P minta pozícióit a következő sorrendben vizsgálja: $1, 2, 3, \dots, m-2, m-1, 0$.

Összehasonlítása a mintapozíciókkal a következő sorrendben történik: $T[j \dots j+m-1]$: ha $P[0]=P[1]$ és $P[1] \neq T[j+1]$, ha $P[0] \neq P[1]$ és $P[1]=T[j+1]$ a minta 2 pozícióval eltolódik a kísérlet végén, egyébként 1-gyel.

Így az előfeldolgozási fázis állandó időben és térben végezhető el. A Not So Naive algoritmusnak a keresési fázis idő komplexitása legrosszabb esetben négyzetesen nő, de átlagosan majdnem lineáris növekedésű. (Charras & Lecroq, Not So Naive algorithm, 1997)

1.7 Naiv algoritmus Hamming távolsággal

Hamming-távolság alatt két azonos hosszúságú bináris jelsorozat eltérő bitjeinek a számát értjük. A fogalmat kiterjeszthetjük két azonos hosszúságú szöveges karaktersorozatra is. Jelen esetben azt jelenti, hogy a vizsgált szövegen a minta hány karakterrel tér el.

Az alap naiv algoritmust nagyon egyszerű úgy átalakítani, hogy tudja kezelni a kódot. Egy változóval számoljuk, hogy hány eltérés volt az összehasonlításnál és csak akkor csúsztatjuk arrébb a mintánkat, ha az eltérés már nagyobb, mint a Hamming távolság.

A futási ideje $O(n*m)$ idő komplexitással írható le, ahol n a szöveg hossza, m pedig a minta hossza.

Olyan esetekben hasznos lehet, ha a mintánk vagy a szövegünk nem tiszta. Például a szekvenálás közben hibás lett a genom kódja, vagy ha az adott szövegben vannak helyesírási hibák, akkor is meg tudjuk találni, amit keresünk.

2. Klasszikus összehasonlításon alapuló módszerek

2.1 Bevezető

Ez a fejezet bemutat több klasszikus összehasonlításon alapuló módszert a pontos illesztés problémára. Megfelelő kiterjesztéssel ezek az algoritmusok megvalósíthatóak legrosszabb esetben is lineáris idő alatt és melyek a minta előfeldolgozásával érik el ezt. A szöveg előfeldolgozás módszerével szellemileg azonos, de fogalmi nehézségeiben meglehetősen eltérő. Néhány klasszikus módszer elég bonyolult, ezért ezekkel ebbe a fejezetbe nem foglalkozunk. Itt az előző fejezetbe tárgyalt előfeldolgozási módszerekre alapuló pontos illesztést tárgyaljuk, mint a Boyer-Moore algoritmus vagy a Knuth-Morris-Patt.

2.2 Boyer-Moore algoritmus

Hasonlóképpen, mint naiv algoritmus a Boyer-Moore algoritmus is a balról jobbra halad végig a T szövegen, és amikor nem egyezik, akkor jobbra csúsztatja a P mintát. Viszont három okos megoldást alkalmaz, amit a naiv nem alkalmazott: a jobbról balra való vizsgálata, rossz karakter szabály, és a jó utótag eltolási szabály. Ezeket együtt alkalmazva általában kevesebb mint $m+12$ összehasonlítást végez, és legrosszabb esetben is lineáris időben lefut. Az algoritmust a leghatékonyabb karakterlánc-illesztés algoritmusnak tekintik. Ennek egyszerűsített változatát, vagy a teljes algoritmust gyakran implementálják alkalmazásokban. Minden P T -vel való összehasonlítást az algoritmus jobbról balra végzi el ellentétesen, mint a naiv algoritmus. A mikor nem talál egyezést, eggyel jobbra tolja relatív a T -hez képest. Ez magában véve nem jelenet semmiféle gyorsulást, mert így ugyanúgy $O(mn)$ idő alatt fut le az algoritmus. Ezért kell mellé a két okos eltolási szabály.

A rossz karakter szabály (bad character rule) egy olyan heurisztika, amely jelentősen segítheti az algoritmus gyors lefolyását. A heurisztika megértéséhez tegyük fel, hogy az utolsó karakter, amit P -ben vizsgáltunk y és T -ben egy x karakterrel áll együtt ($x \neq y$). Amikor ez az eltérés előfordul, mi tudjuk a leg jobb oldalon lévő előfordulását az x karakternek a P -ben, ezért a mintát biztonságosan eltolhatjuk annyival, hogy ez megegyezzen. Minden kisebb eltolás rögtön egy eltérésbe fulladna, ezért a hosszabb eltolás helyes, úgy nem ugrunk át semmilyen egyezést. Ha az x teljesen nem található meg a P -ben akkor az egész mintát eltolhatjuk, hogy teljesen

átmenjen az eltérés pontjára. Ebben az esetben sok karakter egyáltalán nem is lesz vizsgálva a T -ben, így sublinearis időben lefut a programunk.

Definíció: Az abc minden x karakteréhez legyen $R(x)$ az x karakter jobb szélső előfordulási helye a P -ben. $R(x)$ értéke nulla, ha x nem fordul elő P -ben.

Ez az előfeldolgozás nem igényel nagy erőfeszítést $O(n)$ idő alatt létre lehet hozni az $R(x)$ -et. Az előfeldolgozási szakaszban az érdemes egy rossz karakter táblázatot létrehozni, és ebben tárolni az eredményét. Az eltérésnél így a tábla vizsgálata legrosszabb esetben is $n-1$, ami nagyjából annyi, mint amennyi karakter egyezett. Szóval a legrosszabb esetben is, maximum duplázza a Boyer-Moore algoritmus futási idejét. A legtöbb probléma esetén azonban a hozzáadott idő jóval kevesebb lesz, mint a duplája. A heurisztika jól működik a legtöbb esetben, főleg angol abcével. Nem olyan hatékony rövid a minta esetén, vagy ha a szöveg sok olyan karaktert tartalmaz, mint a minta.

A jó suffix szabály (Good Suffix rule) bevezetésével ezt javítjuk. A klasszikus előfeldolgozási módszer a good suffix rule-ra nehéznek tekinthető. A heurisztika úgy működik, létrehozunk egy táblázatot, amely minden minta-utótaghoz megadja annak leghosszabb megfelelő utótagjának hosszát. A megfelelő utótag egy olyan utótag, amely a minta végén kezdődik, és a minta többi részét is tartalmazza. Ha a szövegben egy minta-utótagot találunk, akkor a táblázatban megnézzük annak a leghosszabb megfelelő utótagjának hosszát. A szöveget a megfelelő utótag hosszával előrelepettjük.

A két táblázat segítségével sok karakter vizsgálatot át tudunk lépni. Mivel fentiek szerint egyik se fog potenciális egyezést átugrani ezért minden eltérés esetén azt a lépést alkalmazzuk, ami nagyobb lép előre. Ezért most prezentálni lehet a teljes algoritmust.

A Boyer-Moore algoritmus: Előfeldolgozási fázis: A minta minden pozíciójához kiszámítjuk, hogy az adott karakter utolsó előfordulása hol van a mintában. Kiszámítjuk, hogy minden karakter milyen távolságra van a mintától a szövegben. Keresési fázis: A szövegben balról jobbra haladva, a mintán jobbról balra haladva, minden egyes pozícióban megvizsgáljuk, hogy a minta aktuális karaktere megegyezik-e a szöveg aktuális karakterével. Ha igen, akkor a következő karaktereket is megnézzük. Ha nem, akkor a következő pozícióba toljuk a mintát, attól függően melyik szabály alapján lépünk nagyobb.

The Boyer–Moore algorithm

```
{Preprocessing stage}
  Given the pattern  $P$ ,
  Compute  $L'(i)$  and  $l'(i)$  for each position  $i$  of  $P$ ,
  and compute  $R(x)$  for each character  $x \in \Sigma$ .
{Search stage}
   $k := n$ ;
  while  $k \leq m$  do
    begin
       $i := n$ ;
       $h := k$ ;
      while  $i > 0$  and  $P(i) = T(h)$  do
        begin
           $i := i - 1$ ;
           $h := h - 1$ ;
        end;
      if  $i = 0$  then
        begin
          report an occurrence of  $P$  in  $T$  ending at position  $k$ .
           $k := k + n - l'(2)$ ;
        end
      else
        shift  $P$  (increase  $k$ ) by the maximum amount determined by the
        (extended) bad character rule and the good suffix rule.
    end;
```

2.2.1. ábra: Boyer-Moore algoritmus

(Gusfield, 1997)

Legyen a szöveg hossza n , a keresett string hossza m . Ekkor a Boyer-Moore algoritmus legrosszabb esetben $O(mn)$, ha a minta megtalálható a szövegben és $O(m+n)$ ha a nem jelenik meg a szövegbe. (Gusfield, 1997), (Charras & Lecroq, Boyer-Moore algorithm, 1997)

2.3 Knuth-Morris-Pratt algoritmus

A legjobban ismert lineáris idejű algoritmus az exact matching problémára az Knuth, Morris és Patt miatt van. Habár ez a megoldás ritka választás, alsóbbrendű, mint a Boyer-Moore vagy másik metódusok. Egyszerűen magyarázható, és lineáris időkorlátja könnyen igazolható. Az algoritmus a jól ismert Aho-Corasick algoritmus alapját is képezi, amely hatékonyan megtalálja a szövegben előforduló bármely részkifejezést egy mintakészletből.

Egy adott P minta igazítása T szöveghez, naiv algoritmus összehasonlítja i számú karaktert a P -ben a T -hez igazítva, és eltér a következő hasonlításra. A naiv algoritmus a P eltolja egyvel és balról kezdi az összehasonlítást újból. De egy nagyobb eltolás is lehetséges lehet. Vegyük például $P=abcxabcde$ és az eltérés a 8. pozícióban van P -ben. Nagyon egyszerűen lehet levezetni, hogy P -t el lehet tolni 4 pozícióval anélkül, hogy átugornánk egyetlen P illeszkedést is T -ben. Észrevehető, hogy a levezetéshez nem kell tudnunk T szöveget vagy, hogy a P

pontosan hogyan illeszkedik T -re. Csak azt tudjuk, hogy P milyen pozícióban tért el. Erre épül a Knuth-Morris-Pratt algoritmus.

Szóval a KMP algoritmus az előfeldolgozás során egy úgynevezett prefix függvényt számol ki, amellyel azt mondja meg, hogy a minta mely részei egyeznek meg önmagukkal. Az így létrehozott tömböt szokták jump táblának, vagy next táblának nevezni. Ez a tömb azt tárolja, hogy a résszstringhez milyen hosszú a leghosszabb valódi prefix, ami egyezik a string azonos hosszú szuffixéval. Amikor karakternél eltérést észlelünk, a mintát jobbra toljuk a táblázat segítségével, és így tudhatjuk, hogy a már megvizsgált karakter egyezni fog a mintával.

Knuth-Morris-Pratt algorithm

```

begin
Preprocess  $P$  to find  $F'(k) = sp'_{k-1} + 1$  for  $k$  from 1 to  $n + 1$ .
   $c := 1$ ;
   $p := 1$ ;
  While  $c + (n - p) \leq m$ 
  do begin
    While  $P(p) = T(c)$  and  $p \leq n$ 
    do begin
       $p := p + 1$ ;
       $c := c + 1$ ;
    end;
    if  $p = n + 1$  then
      report an occurrence of  $P$  starting at position  $c - n$  of  $T$ .
    if  $p := 1$  then  $c := c + 1$ 
     $p := F'(p)$ ;
  end;
end.
```

2.3.1. ábra: Knuth-Morris-Pratt algoritmus
(Gusfield, 1997)

Legyen a szöveg hossza n , a keresett string hossza m . Ekkor a Knuth-Morris-Pratt algoritmus előfeldolgozás eljárás lépésszáma $O(m)$. Tegyük fel, hogy $m \ll n$, ekkor a keresés műveletigénye legrosszabb és legjobb esetben is egyaránt $O(n)$. Ezért a KMP algoritmust stabil eljárásnak mondjuk.

Az eltolásnál viszont az algoritmus egy T -ben lévő karaktert többször is vizsgálhat ezért az eljárás nem valós idejű algoritmus. Hogy valami valós idejű legyen, ahhoz konstans mennyiségű munka szükséges az első vizsgálat és akármelyik utolsó vizsgálat között eltelt időben. A Knuth-Morris-Pratt módszerben, ha a pozíció a T -ben egy egyezésben van, akkor az többé nem lesz újra vizsgálva. Viszont, ha a pozíció egy eltérésben szerepelt, akkor ez már nem igaz. Meg kell jegyezni, hogy a valós idejű probléma csak a keresési fázisba izgat minket. Az előfeldolgozás lehet nem valós idejű. Ha az algoritmus valós idejű, akkor bizonyosan lineáris

idejű is. El kell ismerni viszont, hogy a valós idejű algoritmusok melletti érvek a lineáris idejű módszerekkel szemben némileg kevesek. A valós idejű algoritmus inkább elméleti, mint gyakorlati. (Charras & Lecroq, Knuth-Morris-Pratt algorithm, 1997) (Gusfield, 1997)

2.4 Apostolico-Giancarlo algoritmus

A Boyer-Moore algoritmust nehéz elemezni, mert minden próbálkozás után elfelejti az összes karaktert, amelyhez már illesztett. A mindkét metódus ugyanazokat az eltéréseket találja meg és ugyanazokat az eltolásokat végezné el, viszont az algoritmus kerüli az explicit egyezéseket. Két módosítással éri el ezt.

Először az Apostolico és Giancarlo egy olyan algoritmust alkotott meg, amely a keresési fázisban minden kísérlet végén megjegyzi a leghosszabb suffix hosszát. Ezeket az információkat egy Skip táblázatban tárolja. Gondoljunk egy fázisra, ahol a jobb végén a P -nek egy j pozícióhoz van igazítva T -ben, és tegyük fel P és T megegyeznek egy helyen, de nem hosszabban. Ilyenkor legyen $M(j)$ értéke $k \leq l$. $M(j)$ rögzíti, hogy egy k hosszúságú P suffix fordul elő T -ben, és pontosan a j pozícióban végződik. Ahogy az algoritmus halad előre, egy $M(j)$ értéket beállít egy j pozícióra T -ben, amely P jobb végéhez igazodik. $M(j)$ definiálatlan az összes többi pozícióra.

A második módosítás az N és M vektorokat kihasználva felgyorsítja a Boyer-Moore algoritmust, bizonyos egyezésekre és eltérésekre következtetve. Az ötlet megértéséhez tegyük fel, hogy a Boyer-Moore algoritmus $P(i)$ és $T(h)$ karaktereket készül összehasonlítani, és tegyük fel, hogy tudja, hogy $M(h) > N_i$. Ez azt jelenti, hogy a P egy N_i hosszúságú substring az i pozícióban végződik, és illeszkedik a P suffix-éhez, míg egy T hosszúságú $M(h)$ substring a h pozícióba ér, és megegyezik a P utótagjával. Tehát az N , hosszúságú utótagjai ennek a két részstringnek meg kell egyeznie, és arra a következtetésre juthatunk, hogy a következő N_i -összehasonlítások a Boyer-Moore algoritmusban egyezések lennének. Továbbá, ha $N_i = i$, akkor P előfordulását találtuk T -ben, és ha $N_i < i$, akkor biztosak lehetünk benne, hogy a következő összehasonlítás hibás lesz. Ennélfogva Boyer-Moore szimulálásakor, ha $M(h) > N_i$ elkerülhettünk legalább N_i explicit összehasonlítást. Természetesen azt $M(h) > N_i$ nem igaz az összes esetben.

```

APOSTOLICO-GIANCARLO( $x, m, y, n$ )
1   $j \leftarrow 0$ 
2  while  $j \leq n - m$ 
3      do  $i \leftarrow m - 1$ 
4          while  $i \geq 0$ 
5              do if  $skip[i + j] = 0$ 
6                  then if  $x[i] = y[i + j]$ 
7                      then  $i \leftarrow i - 1$ 
8                      else BREAK()
9                  elseif  $skip[i + j] > suff[i]$ 
10                     then  $\triangleright$  Cases 1 and 2
11                          $i \leftarrow i - suff[i]$ 
12                         BREAK()
13                  elseif  $skip[i + j] < suff[i]$ 
14                     then  $\triangleright$  Case 3
15                          $i \leftarrow i - skip[i + j]$ 
16                         BREAK()
17                  else  $\triangleright$  Case 4
18                       $i \leftarrow i - suff[i]$ 
19           $skip[i + j] \leftarrow m - i - 1$ 
20      if  $i < 0$ 
21          then REPORT( $j$ )
22           $j \leftarrow j + sMatch[0]$ 
23      else  $j \leftarrow j + \max(sMatch[i], occ[y[i + j]] - m + i + 1)$ 

```

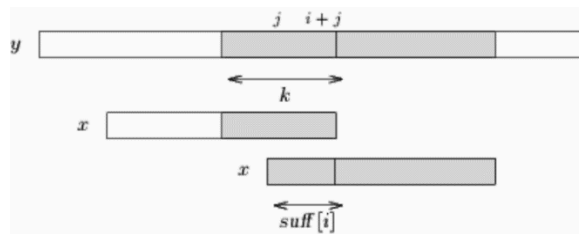
2.4.1.ábra: Apostolico-Giancarlo algoritmus

(Gusfield, 1997)

A j pozícióra tett kísérlet során, ha az algoritmus sikeresen összehasonlítja az $y[i \dots j-1 \dots j+m-1]$ szöveg faktort, akkor négy eset adódik.

Ilyen esetek:

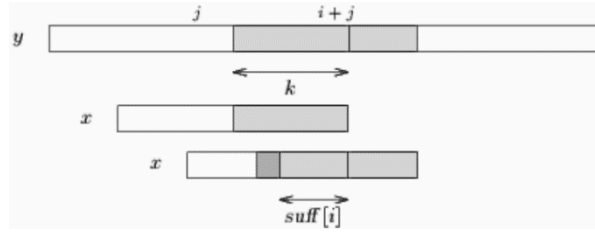
Eset 1: $k > suff[i]$ és $suff[i] = i+1$. Ez azt jelenti, hogy x előfordulása a j és pozícióban található és $skip[j+m-1]$ m lesz (lásd 6.ábra). $per(x)$ hosszúságú eltolás jön létre.



2.4.2.ábra: AG. 1. eset

(Charras & Lecroq, Apostolico-Giancarlo algorithm, 1997)

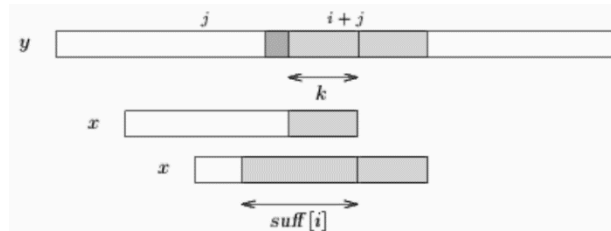
Eset 2: $k > suff[i]$ és $suff[i] \leq i$. Ez azt jelenti, hogy az eltérés $x[i-suff[i]]$ és $y[i+j-suff[i]]$ között van, és $skip[j+m-1]$ értéke $m-1-i+suff[i]$ (lásd 7.ábra). Az eltolás $bmBc[y[i+j-suff[i]]]$ és $bmGs[i-suff[i]+1]$ alapján megy végbe.



2.4.3.ábra: AG. 2. eset

(Charras & Lecroq, Apostolico-Giancarlo algorithm, 1997)

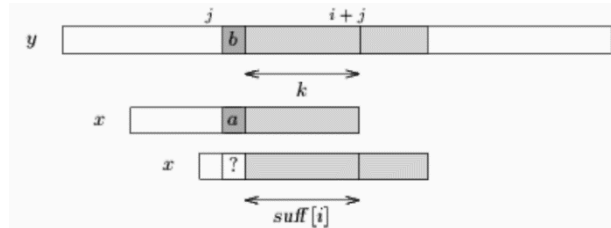
Eset 3: $k < \text{suff}[i]$. Ez azt jelenti, hogy az eltérés $x[i-k]$ és $y[i+j-k]$ között van, és $\text{skip}[j+m-1]$ értéke $m-1-i+k$ (lásd 8.ábra). Az eltolás $\text{bmBc}[y[i+j-k]]$ és $\text{bmGs}[i-k+1]$ alapján megy végbe.



2.4.4.ábra: AG. 3. eset

(Charras & Lecroq, Apostolico-Giancarlo algorithm, 1997)

Eset 4: $k = \text{suff}[i]$. Ez az egyetlen eset, amikor "ugrást" kell végrehajtani az $y[i+j-k+1 .. i+j]$ szöveg tényezőn az $y[i+j-k]$ és $x[i-k]$ karakterek összehasonlításának folytatásához (lásd 2.4.5.ábra).



2.4.5.ábra: AG. 4. eset

(Charras & Lecroq, Apostolico-Giancarlo algorithm, 1997)

Az Apostolico-Giancarlo algoritmus előfeldolgozási fázisának térben és időben történő bonyolultsága megegyezik a Boyer-Moore algoritmuséval. A keresési fázisban minden próbálkozásnál csak az utolsó m információra van szükség a skip táblához, így a tábla kihagyás mérete $O(m)$ -re csökkenthető. Az Apostolico-Giancarlo algoritmus a legrosszabb esetben legfeljebb $3/2n$ karakter-összehasonlítást hajt végre. (Charras & Lecroq, Apostolico-Giancarlo algorithm, 1997)

2.5 Aho-Corasick algoritmus

Az eddig tárgyalt algoritmusok mind-mind egyetlen mintát próbáltak illeszteni a szövegre. Felmerülhet olyan lehetőség sok esetben, hogy nem egy mintát keresünk, hanem több mintát egyszerre. Ilyen lehet például, ha táblázatunk káromkodásokból áll, és ezeket mind ki szeretnénk szűrni, vagy DNS szekvenciákat szeretnénk keresni egy genomon, vagy vannak előre beállított címkék és azok alapján szeretnénk keresni. Ebben az esetben az előző algoritmusoknak minden egyes esetben ahány különböző dolgot szeretnénk keresni, annyszor újra és újra le kell futnia a kereséseknek. Aho-Corasick erre a problémára kínál egyfajta megoldást.

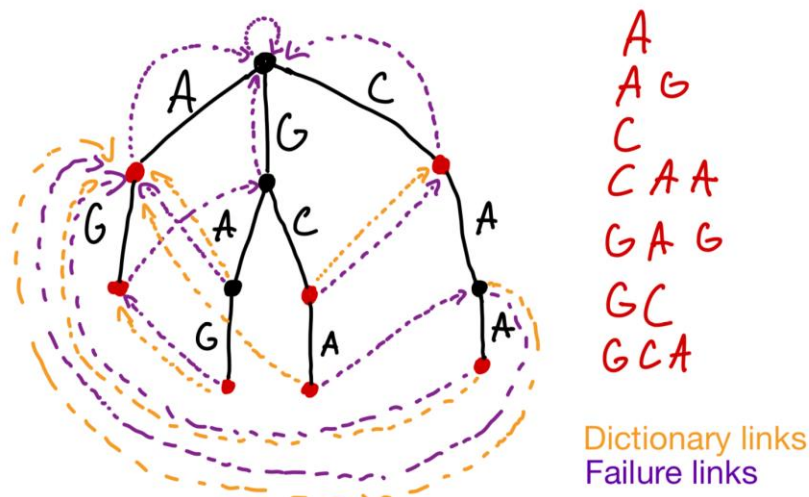
Ez az algoritmus is az előfeldolgozási fázisban a mintát dolgozza fel, csak itt egyszerre többet. A mintákból felépít egy véges állapotú automatát, ezen automata segítségével a szövegen egyszer végig menve megtaláljuk az összes előfordulást. Az algoritmus úgy is felfogható, mint a Knuth-Morris-Pratt algoritmus általánosítása a fákon.

A véges állapotú automata a számítás matematikai modellje. Egy absztrakt gép, amely egy adott időpontban véges számú állapotból pontosan egyben lehet. Definíciója: Az $A = \langle A, X, Y, \delta, a_0, \mu \rangle$ hatost megjelölt állapotú véges, determinisztikus automatának nevezzük, ahol A : véges halmaz, az automata állapotainak halmaza, X : ábécé, az automata bemenő jeleinek halmaza, Y : ábécé, az automata kimenő jeleinek halmaza, $\delta: A \times X \rightarrow A$ alakú, mindenütt értelmezett leképezés, az állapot-átmeneti függvény, $a_0: a_0 \in A$, az automata kezdő állapota, $\mu: A \rightarrow Y$ alakú, mindenütt értelmezett leképezés, a jelölő függvény.

Prefixfa struktúrába tárolja a mintát, ahol minden út végpont egy hozzá tartozó minta. Ehhez a fához Failure Linkeket, vagyis hiba láncokat rendel minden csomópontba. A hibaláncok segítik az algoritmust az átfedések kezelésében, és a hatékony visszalépésben, amikor egy karakter nem felel meg a jelenlegi minta következő karakterének.

Van olyan eset, amikor az egyik minta teljes átfedése másik mintának. Ilyenkor a fában nem a levélben lesz találat, hanem egy csomópontban. Ez felvet olyan problémákat, hogy bizonyos esetekben a hiba láncok nem elegendőek a megtaláláshoz. Ezért be kell vezetni dictionary linkeket is minden egyes csomópontba.

Egy ilyen automata reprezentációja található a képen (2.5.1. ábra). A minták halmaza: "a,ag,c,caa,gag,gc,gca". Fa csomópontjai pirosak amikor az automata abban az állapotban talál egyezést, és feketék, ha épp keresési állapotában van. A narancssárga szaggatott nyilak reprezentálják a dictionary linkeket, a lila szaggatott nyilak pedig a failure linkeket.



2.5.1.ábra: Aho-Corasick algoritmus automatája

Az automatát felépítve kész az előfeldolgozási szakasz. Jól látni, hogy keresési fázisban az lesz a dolgunk, hogy lépésről lépésre végig lépkedünk az automatán, és az automata bemente pedig T szöveg lesz. Ennek az eszköznek a segítségével a műveleti lépéseket rögtön le is csökkentettük szöveg hosszára vagyis $O(n)$ -re. Egyetlen egyszer kell végig mennünk T szövegen, és megtaláljuk az összes kereset mintát, jelentősen csökkentve az algoritmus futási idejét. (Gusfield, 1997)

Algorithm full AC search

```

 $l := 1;$ 
 $c := 1;$ 
 $w := \text{root};$ 
repeat
  While there is an edge  $(w, w')$  labeled  $T(c)$ 
  begin
    if  $w'$  is numbered by pattern  $i$  or there is
    a directed path of failure links from  $w'$  to a node numbered with  $i$ 
    then report that  $P_i$  occurs in  $T$  ending at position  $c$ ;
     $w := w'$  and  $c := c + 1$ ;
  end;
   $w := n_w$  and  $l := c - lp(w)$ ;
until  $c > n$ ;

```

2.5.2.ábra: Aho-Corasick algoritmus
(Gusfield, 1997)

3. Seminumerical string matching

3.1 Bevezető

Az eddig tárgyalt exact matching problémák mind összehasonlításon alapuló metódusok. A fő primitív operandus az összes algoritmusban két karakter összehasonlítása volt. Viszont vannak olyan string matching algoritmusok, amik bit operandusokon alapszanak. Ezek teljesen más megközelítést képviselnek, mint a karakter összehasonlító. Ebben a fejezetben két ilyen módszert fogok ismertetni: a shift-and metódust, és a random fingerprint metódusát Karpnak és Rabinnek.

3.2 Shift-or metódus

R. Baeza-Yates és G. Gonnet kitalált egy egyszerű, bitorientált metódust, ami megoldja az exact matching problémát. Nagyon effektíven működik relatíve kicsi mintán (egy átlagos angol szó hossza). Ezt a metódust Shift-Or metódusnak nevezték el, de van aki shift-and metódusnak hívja.

A metódus egy véges állapotú nemdeterminisztikus automatát használ. Ha minta P hossza n , és a szöveg T hossza m , akkor a definíciója: Legyen M egy $n \times m$ -gyel bináris értékű tömb, amiben index i 1-től n -ig fut, és j index 1-től m -ig. A belépés $M(i, j)$ akkor és csak akkor 1, ha

P első i karaktere pontosan megegyezik T j karakterre végződő i karaktereivel. Más esetben a belépés nulla.

```
shiftOr(pattern,text){
  for (0 .. pattern hossza) {
    mask[pattern[i]] = mask[pattern[i]] | (1 << i)
  }
  for (0 .. text hossza) {
    state = (state << 1) + 1
    state = state & mask[text[i]]
    ha ((state & (1 << (m - 1))) != 0) {
      egyezés megtalálva
    }
  }
}
```

3.2.1.ábra: Shift-or algoritmus

A maszk tábla készítése egy for ciklus (lásd 3.2.1.ábra) ami végigmegy a minta elemein i indexel. $Mask[pattern[i]]$ elemét egy bit művelettel az i -edik bitet eltolja balra, a többi nulla.

A keresés végigmegy a szövegen. A state változót egy balra shifteléssel frissítjük, majd hozzáadunk egyet, ezzel szimulálva a minta további karakterekkel való összehasonlítását. A frissített $state$ értéket egy bitenkénti $\&$ művelettel kombináljuk a $mask$ tömbben tárolt értékkel, így alkalmazva a mintázatot a szövegre. Ellenőrizzük, hogy az állapot m -edik bitje beállt-e. Ha igen, akkor találtunk egy mintát a szövegben, és az aktuális pozíciót hozzáadjuk a *matches* tömbhöz. Ha nem akkor folytatjuk a for ciklust.

Ez alapján látni, hogy hasonló módon, mint a naiv algoritmus volt ez is egyesével végigmegy a szöveg elemén, és van olyan karakter, amin még többször is. Viszont mivel a bitműveletet használ így ez szignifikánsan gyorsabb, mint egy karakter összehasonlító algoritmus. Hiába, hogy legrosszabb esetben $O(mn)$ db bit műveletet kell végrehajtani, nagyon effektív ha n kisebb, mint egy egyszerű számítógépes szó. Ebben az esetben a művelethez elég egy egyszavas művelet. Még akkor is, ha a minta nagysága többszöröse is az egyszerű szó, néhány művelet kell csak. Egyszavas esetén nagyon effektív, mind időben és tárigényben. Teoretikus

szempontból nézve egy nem egy egyenes vonalú metódus, de egy praktikus és sok esetben használható algoritmus.

(Charras & Lecroq, Shift Or algorithm, 1997) (Gusfield, 1997)

3.3 Karp and Rabin

A Shift-or algoritmus feltételezte, hogy hatékonyan tudjuk shiftelni bitek vektorát, és azt is, hogy tudjuk eggyel növelni. A Rabin-Karp eljárás a mintát, valamint a szöveg ugyanolyan hosszúságú szakaszait nagy egész számokkal kódolja, így az illeszkedés vizsgálat két szám egyenlőségének ellenőrzése.

Az algoritmus emiatt elveszíti a közvetlen kapcsolatát a szöveg egyes karaktereivel, így nem lesz képes nagy lépésekre, hanem mindig csak egy pozíción csúsztatja jobbra a mintát a szövegben, ahol minden helyzet egy új egész számot határoz meg, mint a lefedett karakterek kódját. Tekinthetjük úgy az eljárást, hogy egy egész számokból álló sorozaton lineáris kereséssel keressünk egy szám első előfordulását, ahol ez a keresett szám a mintának a kódolásából származik.

Ettől a módszer hatékony lesz, mivel az egész számokkal való műveletek gyorsan végrehajthatóak. Mivel azonban igen nagy számokról lehet szó, az egyenlőségük fogalma és ellenőrzése összetettebb, mint a megszokott nagyságrendekben, így a hatékonyság nem magától értetődik.

$$H(P) = \sum_{i=1}^{i=n} 2^{n-i} P(i).$$

$$H(T_r) = \sum_{i=1}^{i=n} 2^{n-i} T(r+i-1).$$

3.3.1.ábra: Hash függvények

(Gusfield, 1997)

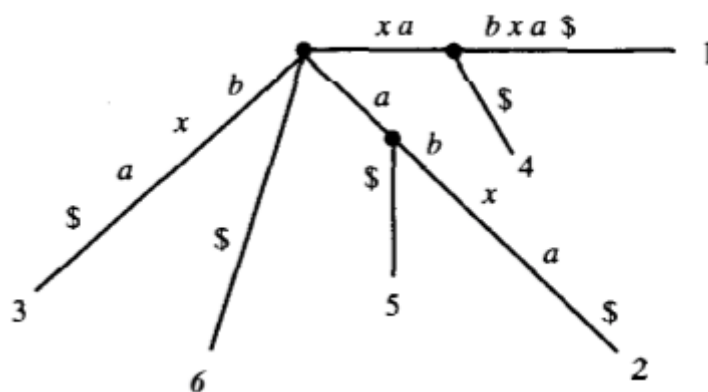
Működése során, hogy bitekké alakítsuk a P mintát és T szöveget hash funkciót kell alkalmazni (lásd 3.3.1.ábra). Akkor találtuk meg a P minta kezdő pozícióját, akkor és csak akkor, ha $H(P)=H(T_r)$. A helyett, hogy nagy számokkal dolgoznánk $H(P)$ -nél és $H(T_r)$ -nél, egy relatíve kisebb számmal fogunk dolgozni, mert redukáljuk modulo p -vel. Ez az aritmetika nem vesz igénybe nagy számolást, szóval hatékony lesz. Az így létrejött számot hívjuk ujjlenyomatomnak.

A Karp-Rabin fingerprint algoritmus legrosszabb esetben lineáris idejű, de nem nulla (de extrémén kicsi) eséllyel lesz hibás. Látunk más alternatívákat, amik nem hibáznak és ugyanilyen gyorsan tudnak működni, így jogosan merül fel a kérdés mi értelme van ennek? Három válasz van erre a kérdésre. Először is, gyakorlati szempontból a módszer egyszerű, és más problémákra is kiterjeszthető, mint például a two dimensional part time matching with odd pattern shapes. Másodszor, a módszert konkrét bizonyítékok kísérik, amelyek a módszer teljesítményének jelentős tulajdonságait állapítják meg. Az ujjlenyomatokhoz hasonló módszerek a Karp-Rabin módszert megelőzően léteznek, de a Karp-Rabin módszerrel ellentétben általában hiányzik belőlük az elméleti elemzés. A teljesítményükről keveset bizonyítottak. A fő vonzereje az, hogy a módszer egészen más elképzeléseken alapul, mint a hiba mentességet garantáló lineáris idejű módszerek. A módszer tehát azért került ide, mert ennek a dolgozatnak az a központi célja, hogy bemutassa a különféle technikákban, algoritmusokban és bizonyításokban használt ötletek sokféle gyűjteményét. (Charras & Lecroq, Karp-Rabin algorithm, 1997) (Gusfield, 1997)

4. Suffix fa és használata

4.1 Bevezető

A suffix tree (vagy fa) egy adat struktúra, ami feltárja a string belső szerkezetét. A suffix fák használhatóak exact matching probléma megoldására lineáris időben, de az igazi erénye, hogy használható komplexebb problémákra is, szintén lineáris időben végrehajtva. Ráadásul ezek a fák egy hidat képeznek az exact matching és az inexact matching között, amire a dolgozatban kevésbé fókuszálunk.



4.1.1.ábra: Suffix fa "xabxa\$" stringnek

(Gusfield, 1997)

Klasszikus felhasználása a suffix fáknak a substring probléma. Adott egy szöveg T , ami m hosszúságú. Ezután $O(m)$, vagy lineáris előfeldolgozási idővel fel kell készíteni arra, hogy akármilyen nem ismert P string-et, aminek a hossza n , $O(n)$ idő alatt találjuk meg P -t T -ben vagy mondjuk meg, hogy nincs benne. Az előfeldolgozási szakasz ideje megengedett, hogy arányos legyen a T szöveg hosszával, de ez után viszont a P keresése már P hosszával kell hogy arányos legyen. Ezeket a határokat a suffix fa segítségével tudjuk elérni. A suffix fa $O(m)$ idő alatt építhető fel az előfeldolgozási szakaszban, ezután bármikor egy n hosszúságú bemenetel az algoritmus $O(n)$ idő alatt végzi el a keresést a fán.

Az eddigi problémák mind arra épültek, hogy a P mintát dolgozták fel előre és így gyorsították a keresést. Ha viszont az adatunk óriási és fix, mint például The Human Genome Project által létrehozott emberei DNS, akkor az ebben való keresést gyorsíthatjuk, ha szöveget dolgozzuk fel először. Az előző illesztési problémáknak az volt a problémája, hogy úgy növekedett a keresés ideje, ahogy a szöveg növekedett. Ezt pedig egy Suffix fa képes megoldani.

4.2 Suffix fa

A fát több fajta módon is fel lehet építeni. Ha naivan állunk hozzá, és először egy suffix trie-t csinálunk, majd ezt redukáljuk le tree-re akkor elveszik az egész lényege, amiért ezt a módszert választottuk.

Ezt a problémát a finn Esko Ukkonen 1995-ben megoldotta. Az algoritmus azon az elgondoláson alapul, hogy az suffix fát fokozatosan bővítjük, miközben a bemeneti stringet egy karakterenként dolgozzuk fel. Ukkonen kulcsfontosságú meglátása az, hogy az „implicit reprezentációnak” nevezett technikát használja, hogy elkerülje az összes utótag explicit reprezentációját a fában. Ehelyett az algoritmus menet közben építi fel a fát, miközben feldolgozza a bemeneti karakterlánc minden karakterét. Használata széles körben elterjedt. $O(n)$ időbe képes felépíteni a teljes fát.

High-level Ukkonen algorithm

```
Construct tree  $\mathcal{T}_1$ .
For  $i$  from 1 to  $m - 1$  do
  begin {phase  $i + 1$ }
    For  $j$  from 1 to  $i + 1$ 
      begin {extension  $j$ }
        Find the end of the path from the root labeled  $S[j..i]$  in the
        current tree. If needed, extend that path by adding character  $S(i + 1)$ ,
        thus assuring that string  $S[j..i + 1]$  is in the tree.
      end;
    end;
  end;
```

4.2.1. ábra Ukkonen algoritmus

(Gusfield, 1997)

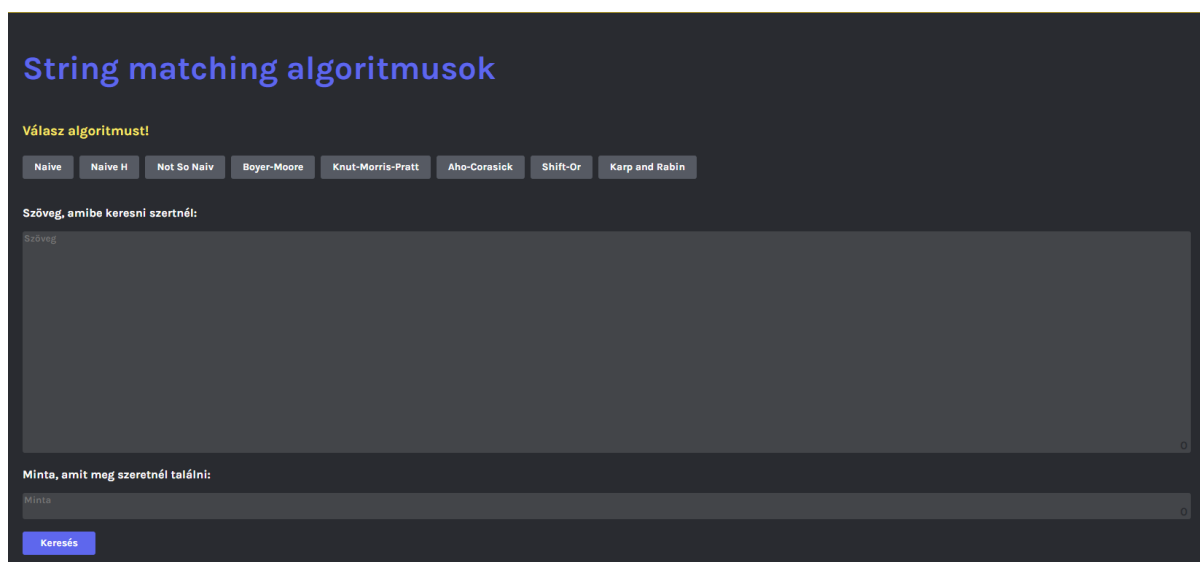
Az Ukkonen fajta algoritmus a legelterjedtebb megoldás, viszont másféleképpen is meg lehet oldani. Weinernek és McCreighnak is van suffix fa algoritmus.

Mint a bevezetőben is említettem, a suffix fa használható másra is, mint a matching probléma. Mivel a dolgozatnak nem célja ezek bemutatása, csak a teljesség igénye nélkül megemlítek pár további problémát. Substring ellenőrzésre alkalmas. Meg lehet rajta nézni mi a leghosszabb ismételt substring. Suffix tömb építésre is alkalmas. Ha két stringből építjük fel a fát, alkalmas megmondani a leghosszabb közös substring-et vagy pedig a leghosszabb palindrom substring-et. (Gusfield, 1997)

5. A program bemutatása

5.1 A program felépítése

Az programom egy weblap, aminek a segítségével mutatok be néhány string matching algoritmust. Név szerint a Naiv, Naiv with Hamming distance, Not so Naiv, Boyer-Moore, Knuth-Morris-Pratt, Aho-Corasick, Shift-Or, és a Karp and Rabin algoritmust, amiket JavaScript nyelven valósítottam meg.

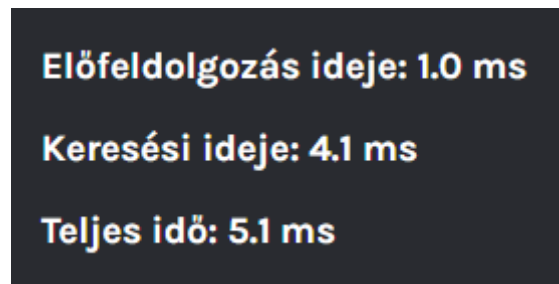
The screenshot shows a web application titled "String matching algoritmusok" in a dark-themed interface. Below the title, there is a yellow prompt "Válassz algoritmust!". A row of buttons allows selecting an algorithm: "Naive", "Naive H", "Not So Naiv", "Boyer-Moore", "Knuth-Morris-Pratt", "Aho-Corasick", "Shift-Or", and "Karp and Rabin". Below these buttons, there are two input sections. The first is labeled "Szöveg, amibe keresni szeretnél:" and contains a large, empty text area. The second is labeled "Minta, amit meg szeretnél találni:" and contains a smaller, empty text input field. At the bottom left, there is a blue button labeled "Keresés".

5.1.1. ábra: kezdő állapota

Kezdő állapotban a cím, mint String matching algoritmusok fogad. Alatta egy sárga szöveg, hogy válasszunk algoritmust. Két beviteli mező. Egy nagyobb, amibe a szöveget kell írni, amiben keresni szeretnénk. És egy kisebb, amibe a keresendő mintát lehet írni. Mindkét bevitelmező jobb alsó sarkában található egy szám, ami az adott mezőbe írt karakterek számát mutatja. A honlap alján pedig egy kék keresés gomb található.

Abban az esetben, ha kiválasztjuk az algoritmusunkat, az adott algoritmus gomb színe zöld lesz. Mindig csak a kiválasztott algoritmus színe lesz zöld. Minden algoritmushoz kapunk egy nagyon leegyszerűsített leírást, mi is az adott algoritmus. Ezek után, ha szövegbeviteli mező üres felette sárgán jelezni fogja a program, hogy nem lehet üres. Amint írunk bele, eltűnik a figyelmeztetés. Hasonlóképpen működik a minta beviteli mező is. Viszont ez csak akkor figyelmeztet, ha ki van választva algoritmus, és a szöveg mező sem üres.

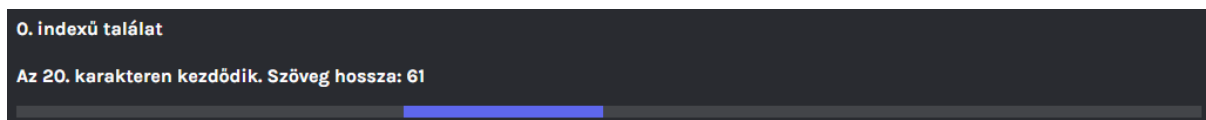
„Keresés” gomb megnyomása küld alertet üres beviteli mező esetén. Ha nem üres, lefuttatja a keresést. Kiírja a kereséshez szükséges előfeldolgozási időt, a feldolgozás utáni keresési időt és a kettő összegét. Mindhármat milliszekundumban 2 tizedesjegyre kerekítve.



Előfeldolgozás ideje: 1.0 ms
Keresési ideje: 4.1 ms
Teljes idő: 5.1 ms

5.1.2.ábra: idő kiírás

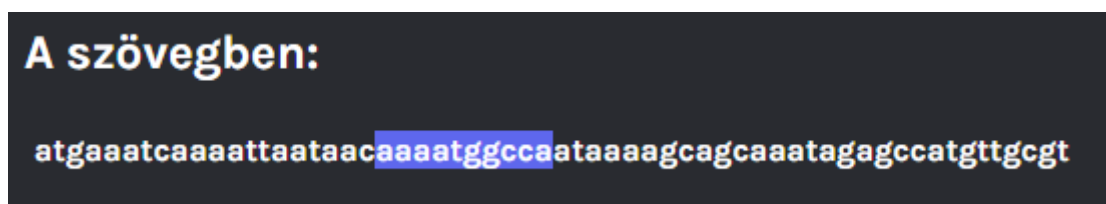
Ha a keresés eredményes, azaz illeszthető a minta az adott szövegre, akkor minden helyes találatot kiír a következőképpen. Az összes találatot fel indexeli 0-val kezdve, és kiírja illesztés kezdő karakterének helyét, és a szöveg hosszát. Alatta található egy szürke csík, ami a szöveg 100%-át reprezentálja. A csíkon lévő kék szakasz azt reprezentálja, hogy a talált minta a szövegben hol található százalékosan.



0. indexű találat
Az 20. karakteren kezdődik. Szöveg hossza: 61

5.1.3.ábra: ábrázolás

A találatok alatti részben, egybegyűjtve az összes találatot, kiemeli kék háttérrel a mintákat a szövegből.



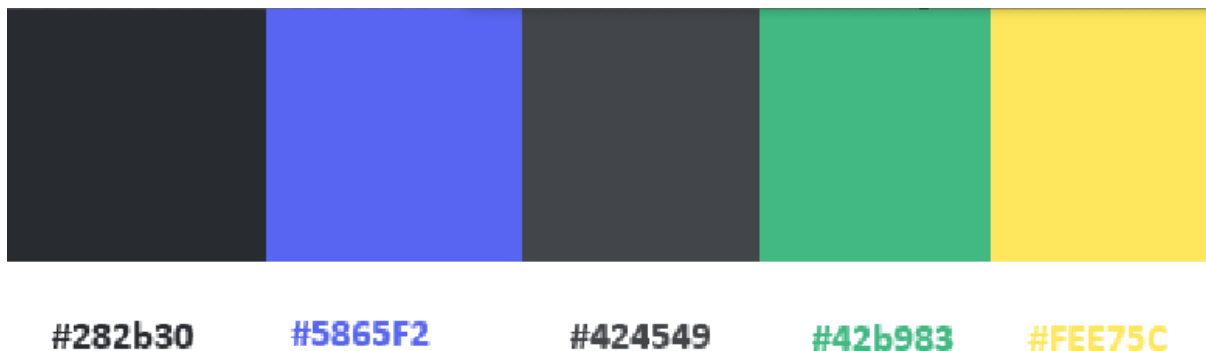
A szövegben:
atgaaatcaaaattaataacaaaatggccaataaaagcagcaaataagagccatgttgct

5.1.4.ábra: kiemelt szöveg

A honlap kliens oldalon számol, semmit nem tölt fel a szerverre, így biztonságosan használható. Viszont mivel kliens oldali számításokat végez JavaScriptben, és a megjelenítés is ezen alapszik. Ha a böngésző nem támogatja, esetleg a felhasználó kikapcsolta a JavaScriptet, akkor a program használhatatlan. Az adott keresések gyorsasága emiatt függ a felhasználó eszköztől is. Korlát még a JavaScript egyetlen szálon futása, ami azt jelenti, hogy nincs igazi többszálúság a nyelvben.

5.2 Program kinézete

Egy egyszerű letisztult megjelenést szerettem volna elérni. Minimalista megjelenés érdekében egy sötét háttér színt választottam (#282b30). A fő szín a kék (#5865F2) választottam, ezzel íródott a cím, illetve a kiemelések, keretek is ezzel vannak kiemelve. A szöveg színe fehér. A beviteli mezők és a gombok szürkék, mint a háttér (#424549). Kiválasztott algoritmus színe zöld (#42b983), a figyelmeztetések színe sárga (#FEE75C).



5.2.1.ábra: színek

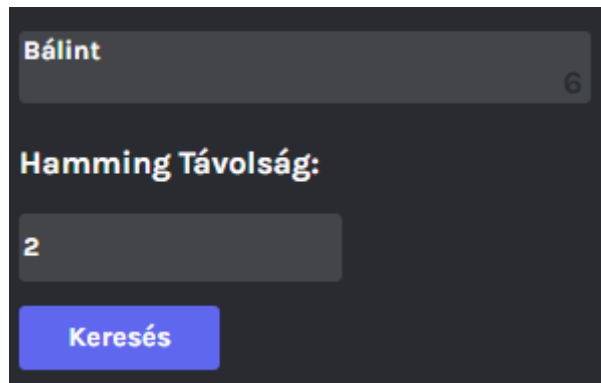
A gombok sarkai minimálisan lekerekített (border-radius: 3px). A használt betűtípus is kerek stílus: Karla típusú.

5.3 Naiv algoritmus

Naiv algoritmus esetén két alap beviteli mező van. A szövegmező, amiben szeretnénk keresni, és a minta mező, amit szeretnénk keresni. A keresés a naiv modult hívja meg segítségül, és azon belül a naiv algoritmust. Ez két stringet vár. A mintát és a szöveget, és egy tömbbel tér vissza. A tömb elemeiben a helyes mintaillesztés első karakterének helye van. Mivel nincs előfeldolgozás, az algoritmusnál 0ms-ot fog kiírni erre az értékre. Keresés ideje a naiv függvény futási ideje lesz. A találatokat az egyesével kiírja (lásd 5.1.3. ábra), és a teljes szöveget is kiemelve a találatokkal (lásd 5.1.4. ábra). Az algoritmus rövid leírása a programban: minden lehetséges helyen megpróbálja illeszteni a keresett stringet a szövegre.

5.4 Naiv H algoritmus

Naiv algoritmushoz képest annyit változik, hogy az alap két, minta és szöveg, beviteli mezők alatt megjelenik egy harmadik mező, amibe a Hamming távolságot lehet megadni, nulla vagy pozitív egész számként (lásd 5.4.1. ábra).

The image shows a dark-themed web application interface. At the top, there is a text input field containing the word "Bálint" and a small number "6" to its right. Below this, the text "Hamming Távolság:" is displayed. Underneath, there is another text input field containing the number "2". At the bottom of the interface, there is a blue button with the text "Keresés" (Search) in white.

5.4.1.ábra: Hamming távolság mező

A keresés a naiv modult hívja meg segítségül, azon belül a naiveH függvényt, ami 3 bemenetet vár a mintát, a szöveget és a távolságot. Egy tömbbel tér vissza, a tömb elemeiben a helyes mintaillesztés első karakterének helye van. Mivel nincs előfeldolgozás az algoritmusnál 0ms-ot fog kiírni erre az értékre. Keresés ideje a naiv függvény futási ideje lesz. A találatokat az egyesével kiírja (lásd 5.1.3. ábra), és a teljes szöveget is kiemelve a találatokkal (lásd 5.1.1.ábra). Az algoritmus rövid leírása a programban: Brute force-al keresi a mintát de Hamming távolságnyi eltérést enged.

5.5. Not So Naiv algoritmus

Hasonlóképpen, mint naiv algoritmusnál teljesen hasonlóan néz ki a felépítése és a végeredmény ábrázolása, csak a keresési fázisba a Not So Naiv algoritmussal keres, aminek szintúgy nincs előfeldolgozási ideje. Az algoritmus rövid leírása a programban: A naiv algoritmus azzal kiegészítve, hogy a mintaillesztést a második karaktertől kezdve kezdi vizsgálni.

5.6 Boyer-Moore algoritmus

Ennél a beviteli mezők teljesen hasonlóak, mint az előző algoritmusoknál, szöveg és minta mező található meg. A keresés előtt viszont létre kell hoznia mintából készült táblázatokat, ami alapján gyorsítja a keresést. Táblázatok létrehozási idejét, ami ezen függvények futását jelenti, méri az előfeldolgozási idő. A keresés alatt már a függvényt a kész táblázatokkal hívjuk meg, ezért a tényleges keresés idejét is tudjuk külön mérni. Végeredmény ábrázolás a szokásos módon zajlik, találatok külön és a szöveg kiemelve velük. Az algoritmus rövid leírása a

programban: A hatékony ugrás érdekében két szabály alapján ugrik előre a szövegben. Szöveget balról jobbra, mintát jobbról balra olvassa. Bad character szabály. Good Suffix szabály.

5.7 Knuth-Morris-Pratt algoritmus

Teljesen hasonló, mint a Boyer-Moore algoritmusoknál. Szöveg és minta mező található meg. A keresés előtt viszont létre kell hoznia mintából készült táblázatot, ami alapján gyorsítja a keresést. `kmpPrefix` függvény az előfeldolgozás, ennek a végeredménye kell még a `kmp` függvénynek a mintán és a szövegen kívül. Végeredmény ábrázolás a szokásos módon zajlik, találatok külön és a szöveg kiemelve velük. Az algoritmus rövid leírása a programban: Az algoritmus a minta előzetes vizsgálatával előre meghatározza, hogy ha a minta adott karaktere nem illeszkedik a szövegre, akkor mekkora az a legkisebb ugrás, amíg biztosan nem lesz egyezés.

5.8 Aho-Corasick algoritmus

Mivel a AC keresés több keresendő mintából épít egy automatát, ez segíti a gyors keresést. Itt nem egy mintát lehet megadni, hanem ezen felül meg lehet adni még 4 másikat is.

A kereséshez minimum az első minta mezőjébe kell írni. A többi mező opcionális. Az előfeldolgozási időbe az automata felépítési ideje tartozik. Majd a keresés és ennek az idejét is rögzítjük.

Minta, amit meg szeretnél találni:

Minta mező nem lehet üres

Minta 0

Minta 2:

Minta 2 0

Minta 3:

Minta 3 0

Minta 4:

Minta 4 0

Minta 5:

Minta 5 0

Keresés

5.81.ábra: AC plusz beviteli mezői

Az algoritmus több mintának a helyét adja vissza, ezért a visszatérési tömb nem számokból áll, hanem minta és index párokból álló objektumból. Így minden találatnál azt is tudjuk melyik mintát találtuk meg. Ennek hatására úgy kell módosítani a megjelenítést, hogy feltüntetjük melyik mintát találtuk meg az adott helyen.

Találat "aaaa"

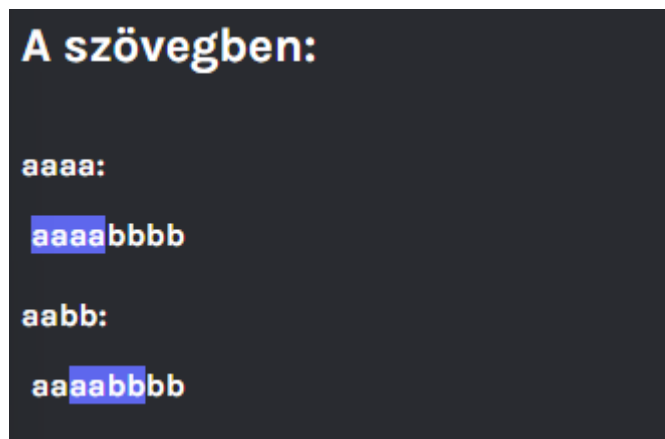
Az 0. karakteren kezdődik. Szöveg hossza: 8

Találat "aabb"

Az 2. karakteren kezdődik. Szöveg hossza: 8

5.8.2.ábra: AC alg .találatok megjelenítése

Ezen felül mikor a szövegben kiemeljük a mintánkat, akkor többször jelenítjük meg a szöveget, attól függően, hogy melyik mintát találtuk meg.



5.8.3.ábra: AC alg. találatok megjelenítése 2.

Az algoritmus rövid leírása a programban: Az algoritmus egy automatát épít a mintákból, és ezzel egyszerre több mintát is lehet keresni egy szövegben gyorsan és hatékonyan.

5.9. Shift-Or algoritmus

A Shift-or algoritmus teljesen alapértelmezett bevitellel és kimenettel rendelkezik. Egy szövegmező és egy minta mező van. Nincs elő feldolgozási idő. A találat indexét és helyét ábrázolja. Az algoritmus rövid leírása a programban: Az algoritmus egy hatékony mintaillesztési algoritmus, amely egy biteltolást használ.

5.10. Karp and Rabin algoritmus

A Karp and Rabin algoritmus teljesen alapértelmezett bevitellel és kimenettel rendelkezik. Egy szövegmező és egy minta mező van. Előfeldolgozás alatt létrehoz egy tömböt, és ezzel hívjuk meg a keresést. A szokásos módon a találat indexét és helyét ábrázolja. Az algoritmus rövid leírása a programban: Az algoritmus a minta és a szöveg összehasonlításához egy hash függvényt használ, így csak a hashértékek egyenlőségének ellenőrzésére van szükség, ami gyorsabb, mint a karakterek egyezőségének ellenőrzése.

6. A program teljes futására egy példa

Legyen a keresendő szavunk: **aabb** és legyen a szöveg, amin keresni szeretnénk az: **aaaaabbbbb**. Legyen az algoritmus Naiv algoritmus.

Az algoritmusoknál rákattintunk Naiv gombra. A gomb háttere zöldre változik. A válasz algoritmusok felirat eltűnik. Megjelenik a felhívás, hogy nem lehet üres a szövegmező. Megjelenik az algoritmus egyszerűsített leírása: Minden lehetséges helyen megpróbálja illeszteni a keresett sztringeket a szövegre.

1. Beírjuk a szöveget(aaaabbbb) a szövegmezőbe.

Eltűnik a felhívás az üres mezőről, megjelenik a minta mező felett, hogy az a mező üres. A beírt szöveg mezőjének jobb alsó sarkában a szám 10-re változik, mivel az adott mezőbe lévő szöveg hosszát jelzi.

2. Kitöltjük a minta mezőt is(aabb).

Felhívás eltűnik. Jobb alsó sarkában a szám 4-re vált.

3. Keresés gombra kattintva elindul a keresés.

Ellenőrzi a program, hogy nem üres-e valamelyik mező. Nem talál hibát, így tovább lép. Eltárolja az időt, amit a performance.now() függvény segítségével kapott. meghívja a naiv függvényt a két stringgel.

A naiv algoritmus elkezd a 0. indextől illeszteni a mintát a szövegre. A 2. karakter után lesz eltérés a mintában, ezért tol a mintán egyet. Itt is a 2. el fog térni ezért még egyet tol a mintán. A szöveg 2. karakterétől indulva is eltérés lesz a minta 2. eleménél, szóval még egyet tol rajta. 3. karakternél végig tudja illeszteni a mintát ezért a tömbhöz adja 3-as számot. Majd lép egyesével még 3at de mindegyik esetben eltérésbe fullad. A függvény visszatért egy tömbbel, aminek egyetlen egy eleme van a 3. Eltárolja az időt amit a performance.now() függvény segítségével kapott. A két eltérő idő különbségéből meg is kapjuk a függvény futási idejét, ami 0.1ms. (Ilyen rövid példa esetén nem igen érdemes nézni a függvény hatékonyságát.)

A program a keresés gomb alatt kiírja az előző feldolgozási időt, ami ugye nem volt ezért 0ms, a keresés idejét, 0.10ms. És a kettő összegét. Alatta megjeleníti az összes találatot, ami jelenleg 1 db. A tömb 0 elemén található. A 3. karakteren kezdődik. Kiírja a szöveg hosszát, ami 10. Egy szürke vonalat rajzol, ami a szöveget reprezentálja és rajta egy kék csíkot ami a szürke vonal 30%-tól indul a és a 60%-ig tart, ami azt jelenti a minta a szövegben ezen a szakaszon található meg. Legvégül kiírja a szövegeket és kék háttér színnel kiemeli benne a mintát.

String matching algoritmusok

Naive

Naive H

Not So Naiv

Boyer-Moore

Knut-Morris-Pratt

Aho-Corasick

Shift-Or

Karp and Rabin

A naiv algoritmus
Minden lehetséges helyen megpróbálja illeszteni a keresett sztringeket a szövegre

Szöveg, amibe keresni szeretnél:

aaaaabbbb10

Minta, amit meg szeretnél találni:

aabb4

Keresés

Előfeldolgozás ideje: 0.0 ms
Keresési ideje: 0.0 ms
Teljes idő: 0.0 ms

0. indexű találat
Az 3. karakteren kezdődik. Szöveg hossza: 10

A szövegben:

aaa**a**bbbb

6.1.ábra: naiv algoritmus képernyő kép

7. Összefoglalás

A szakdolgozatom témája a string matching algoritmusok elemzése. Ehez részletesen minél több megközelítési módszert próbáltam bemutatni. Ahhoz, hogy bonyolult algoritmusokkal foglalkozzak elengedhetetlennek tartottam, hogy ismertessem mi is a probléma, és miért érdemes foglalkozni vele.

Először egyszerű naiv algoritmuson keresztül mutattam be azt, hogy milyen igények, illetve milyen elvárások léphetnek fel a megfelelő string matching algoritmus megszerkesztésénél. A nem pontos illesztés esetét is tárgyaltam Hamming távolsággal, és hogy a naiv algoritmus, hogyan alakítható át erre.

Majd a klasszikus összehasonlításon alapuló illesztő algoritmusok közül leírtam a Boyer-Moore-tól kezdve a Knuth-Morris-Pratt és a Apostolico-Giancarlo féle megközelítéseket végül az Aho-Corasick algoritmust.

Következő fejezetben a seminumerical string matching-et a shift-ro és a Karp and Rabin algoritmusokon keresztül mutattam be.

A negyedik fejezetben ismertettem a suiffix fát, és azt hogy milyen szerepe van a string matching-ben.

A program bemutatásánál részletesen leírtam annak felépítését. Működését képek segítségével illusztráltam. Részleteztem nyolc különböző algoritmus implementációját a programomban.

A végén egy konkrét példán bemutattam az applikáció működését.

Nyilatkozat

Alulírott Kardos Balint.....programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Számítógépes Algoritmusok és Mesterséges Intelligencia Tanszékén készítettem, programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel. Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumban tárolja.

2023.12.14.

dátum



aláírás

Irodalomjegyzék

- Charras, C., & Lecroq, T. (1997, 1 14). *Apostolico-Giancarlo algorithm*. IGM. Retrieved 9 6, 2023, from <https://www-igm.univ-mlv.fr/~lecroq/string/node16.html#SECTION00160>
- Charras, C., & Lecroq, T. (1997, 1 14). *Boyer-Moore algorithm*. IGM. Retrieved 9 5, 2023, from <https://www-igm.univ-mlv.fr/~lecroq/string/node14.html>
- Charras, C., & Lecroq, T. (1997, 1 14). *Brute force algorithm*. IGM. Retrieved 9 6, 2023, from <https://www-igm.univ-mlv.fr/~lecroq/string/node3.html#SECTION0030>
- Charras, C., & Lecroq, T. (1997, 1 14). *Karp-Rabin algorithm*. IGM. Retrieved 9 6, 2023, from <https://www-igm.univ-mlv.fr/~lecroq/string/node5.html#SECTION0050>
- Charras, C., & Lecroq, T. (1997, 1 14). *Knuth-Morris-Pratt algorithm*. IGM. Retrieved 9 6, 2023, from <https://www-igm.univ-mlv.fr/~lecroq/string/node8.html#SECTION0080>
- Charras, C., & Lecroq, T. (1997, 1 14). *Not So Naive algorithm*. IGM. Retrieved 9 10, 2023, from <https://www-igm.univ-mlv.fr/~lecroq/string/node13.html#SECTION00130>
- Charras, C., & Lecroq, T. (1997, 1 14). *Shift Or algorithm*. IGM. Retrieved 9 6, 2023, from <https://www-igm.univ-mlv.fr/~lecroq/string/node6.html#SECTION0060>
- Farkas, R. (2017). *11_Sztringalgoritmusok* [ppt]. <https://www.inf.u-szeged.hu/~rfarkas/Alga17/>.
- Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

Köszönetnyilvánítás

Szeretnék szívből köszönetet mondani Csirik Jánosnak, a témavezetőmnek, aki nélkülözhetetlen segítséget nyújtott szakdolgozatom elkészítése során. Az értékes tanácsai, türelme és szakmai iránymutatása nélkül a dolgozatom nem állhatott volna össze ilyen színvonalasan. Köszönöm, hogy mindvégig mellettem állt és hozzájárult a munkám sikeréhez. Nagy megbecsüléssel tartozom szüleimnek is, akik mindvégig támogattak az egyetemi éveim során. Köszönöm, hogy mindig mellettem voltak, és bizalmukkal erőt adtak az elérni kívánt célokhoz. Nélkülük ez az út nem lett volna ilyen sikeres és értelmes.

Külön köszönet illeti nagypapámat, dr.Adamkovich Istvánt, aki bölcsességével és támogatásával mindig inspirált engem. Az értékes szakmai tapasztalatai és az általa közvetített értékrend hozzájárultak ahhoz, hogy a szakdolgozatom ne csak egy tanulmány legyen, hanem egy értékes tapasztalat is az életem során.

Mindezen segítség és támogatás nélkül a szakdolgozatom elkészítése nem lett volna ilyen eredményes. Hálásan köszönöm mindenkinek.

Melléklet útmutató

A dolgozathoz tartozó elektronikus melléklet rövid leírása:

KardosBálintSzakdolgozat.zip tartalma:

szakdolgozat-react mappa:

src mappa:

alg mappa:

az algoritmusok implementációja JavaScripten

React komponensek és a style.css

public mappa:

index.html, manifest.json

további fájlok amik a react futáshoz kellenek (package-lock.json, package.json..)

Szakdolgozat ugyanezek a fájlok továbbá elérhető a GitHubon is:

<https://github.com/balintkardos/Szakdolgozat>

Szakdolgozat hoz készült react program fut a szerveren is:

<https://balintkardos.github.io/Szakdolgozat/>