

Kovács Bertalan

REGULÁRIS KIFEJEZÉST ELLENŐRZŐ FORDÍTÁSI IDŐBEN GENERÁLT ALGORITMUSOK ÖSSZEHASONLÍTÁSA

Témavezető: Dr. Bertók Botond

Műszaki Informatikai Kar, Rendszer- és Számítástudományi Tanszék

Pannon Egyetem
Intézményi Tudományos Diákköri Konferencia
2024



Reguláris kifejezést ellenőrző fordítási időben generált algoritmusok összehasonlítása

Készítette: Kovács Bertalan

Programtervező informatikus MSc

Műszaki Informatikai Kar, Rendszer- és Számítástudományi Tanszék

Témavezető: Dr. Bertók Botond

A reguláris kifejezések olyan szövegek, amelyek segítségével más szövegekben tudunk valamilyen minta szerint keresni. A gyakorlatban sokszor használjuk őket nagy szövegben történő mintaillesztésre, vagy minták alapján történő strukturális cserére¹.

A reguláris kifejezések alapján történő kereséskor először mindig el kell készítenünk az olvasót, amellyel az adott kifejezésre illeszkedő mintákat tudjuk keresni. Ezt az olvasót a gyakorlatban futási időben készítjük el, ahol számít a futási idő, az olvasó felépítésének ideje és a felépítés közben használt memória mennyisége is.

Amennyiben a reguláris kifejezés előre ismert, ezt az olvasót el tudjuk készíteni fordítási időben. Ebben az esetben kevésbé fontos az olvasó készítésének ideje, az olvasó készítése közben használt memória, nagyobb prioritást élvez helyette a futási idő és a futás közbeni memóriahasználat minimalizálása.

A dolgozat a szakdolgozatomhoz korábban írt kódgenerátort használja és bővíti. Az eredetileg elkészített kódgenerátor Java forráskódot készít, Java nyelven íródott. A dolgozat célja megvizsgálni azt, hogy alacsonyabb szintű nyelveken (C++) elérhető-e alacsonyabb futási idő, illetve azt, hogy hogyan viszonyul a generált kód futási ideje reguláris kifejezés feldolgozó könyvtárak használata során mért futási időhöz.

A dolgozat során bemutatjuk a használt reguláris kifejezés nyelvtanát, az eredeti kód generálásának folyamatát. A Java kódgenerátort módosítjuk C++ és Python kód generálására, áttekintjük a forráskódok közötti architektúráis különbségeket, nyelv-specifikus optimalizációkat. Összehasonlítjuk az elkészült C++, Java és Python kódok futási idejét, majd összevetjük ezeket C++ nyelven használható reguláris kifejezés parser könyvtárak futási sebességével.

¹ *Search and replace*

Tartalomjegyzék

1. Bevezetés	5
2. Irodalmi áttekintés	6
2.1. Reguláris kifejezések	6
2.2. Módszerek reguláris kifejezések alapján történő keresésre	6
2.2.1. DFA alapú algoritmus	6
2.2.2. NFA alapú algoritmus	7
2.2.3. Visszalépéses alapú algoritmus	7
2.3. Reguláris kifejezés nyelvtana	7
2.3.1. Zárójel	8
2.3.2. Tetszőleges karakter	8
2.3.3. Elágazás	8
2.3.4. Karaktercsoport	8
2.3.5. Negált karaktercsoport	9
2.3.6. Legalább egy ismétlődés	9
2.3.7. Legfeljebb egy ismétlődés	9
2.3.8. Tetszőleges számú ismétlődés	9
2.3.9. Adott számú ismétlődés	9
3. Kód generálása Java nyelven	11
3.1. Kódgenerálás folyamata	11
3.1.1. Nemdeterminisztikus állapotgép készítése	13
3.1.2. Determinisztikus állapotgép elkészítése	14
3.2. Generált kód általános felépítése	14
3.3. Optimalizálás Java környezetben	16
3.3.1. Több fájlra szétszedés	17
3.3.2. Statikus függvények használata	19

3.3.3.	Ismétlődő kód kiemelése	19
3.3.4.	Egyéb optimalizációk	20
4.	Kód optimalizálása más nyelveken	22
4.1.	Kód felépítése C++ nyelven	22
4.1.1.	Osztályok összevonása	22
4.1.2.	Kétszeres elágazás összevonása	24
4.1.3.	Fordítás	24
4.2.	Python nyelvű implementáció problémái	24
5.	Teljesítmény vizsgálata	27
5.1.	Java, C++, Python kód összehasonlítása	28
5.1.1.	Futásidő változása a bemenet hosszának függvényében	28
5.1.2.	Futási idő a teszt regexek esetén	29
5.2.	C++-ban generált re2, pcre2 összehasonlítása	29
5.2.1.	Egyszerű regexek	29
5.2.2.	Egymást fedő halmazokat tartalmazó regexek	30
5.3.	C++-ban és .NET-ben generált kód összehasonlítása	30
6.	Összegzés	35
6.1.	Tapasztalatok	35
6.2.	További munkák	36
	Irodalomjegyzék	37

1. fejezet

Bevezetés

A reguláris kifejezések olyan szövegek, amelyek segítségével más szövegekben tudunk valamilyen minta szerint keresni. A legtöbb programozási nyelvben van egyszerűen, beépítetten lehetőség regex alapján történő keresésre.

A reguláris kifejezések alapján történő kereséskor el kell készítenünk az olvasót, amellyel az adott reguláris kifejezésre illeszkedő mintákat tudjuk keresni. Ezt az olvasót el tudjuk készíteni olyan módszert használva, amely fordítási időben több időt vehet igénybe, de minimalizálja a futási időt és a memória-használatot.

A dolgozat célja megvizsgálni azt, hogy különböző programozási nyelvek esetén egymáshoz viszonyítva milyen teljesítmény érhető el előre generált regex motort használva, valamint hogyan viszonyul az előre generált olvasó teljesítménye futtatási időben működő megoldásokhoz. A .NET 7 óta a .NET keretrendszer is képes ilyen módon regex olvasót generálni, így annak a teljesítményét is összevetjük az általunk generált olvasók sebességével.

A dolgozat elkészítéséhez a következő célokból volt mesterséges intelligencia alapú rendszer (ChatGPT) használva:

- Egyes részek átfordítása a tesztkörnyezet forráskódjából Java, C++ és Python nyelvek között
- re2 és pcre2 C++ könyvtárak használata
- Az elkészült reguláris nyelvhez Backus-Naur forma szintaktikai megfogalmazása
- L^AT_EX környezetben felmerült hibák javítása, egyes könyvtárak használata

2. fejezet

Irodalmi áttekintés

2.1. Reguláris kifejezések

A reguláris kifejezések (a későbbiekben regexek) olyan stringek, amelyek definiálnak stringek egy halmazát, vagy sorbarendezett string-párok halmazát. [5] Reguláris kifejezéseket gyakran használunk szövegben történő keresésre, szövegek részeinek kicserélésére.

Regexek feldolgozásánál többféle megközelítést alkalmaznak, a legelterjedtebb háromféle módszer: backtracking¹ módszer, NFA², vagy DFA³ alapján történő bejárás.

2.2. Módszerek reguláris kifejezések alapján történő keresésre

2.2.1. DFA alapú algoritmus

Minden reguláris kifejezés átalakítható determinisztikus véges automatává[3, 78-80. oldal]. Az így készített DFA-nak kötelezően van kezdő állapota és legalább egy elfogadó állapota. A keresés kezdetekor a kezdő állapotból indulunk, majd a bemeneti karakter és a véges automata jelenlegi állapotának függvényében változtatjuk a DFA állapotát.

A DFA készítésekor a memória és az időigény nagy lehet ($\mathcal{O}(2^n)$), ezért a gyakorlatban ritkán használják. A kódgenerálásnál ezt a megközelítést alkalmazzuk.

¹Visszalépéses keresés

²Nemdeterminisztikus véges automata

³Determinisztikus véges automata

2.2.2. NFA alapú algoritmus

Több módszer segítségével is átalakítható a reguláris kifejezés NFA-vá[7], amelyet hasonlóképpen használunk, mint az előzőleg említett DFA-t. A fő különbség az algoritmusok között, hogy ebben az esetben nem elég egy jelenlegi állapotot tárolni, el kell tárolnunk az összes olyan állapotot, amelyben a rendszer jelenleg lehet. A futási idő $\mathcal{O}(nm)$ -re nő, viszont gyorsabb a véges automata elkészítése.

Ezt a megközelítést alkalmazza a Google által karbantartott re2 könyvtár.

2.2.3. Visszalépéses alapú algoritmus

Ez a megközelítés az NFA alapú algoritmus egy változata, amely nem tárolja el minden lehetséges állapotát a véges automatának, ehelyett amikor elágazáshoz jut, valamelyik lehetséges állapottal továbbhalad és ha nem sikeres a találat, akkor visszatér az elágazáshoz és egy alternatív állapottal folytatja a keresést.

Egyes regex nyelvtani elemek nem lehetségesek visszalépés használata nélkül⁴. A Java nyelvbe beépített regex motor visszalépéses algoritmust használ.

2.3. Reguláris kifejezés nyelvtana

Az elkészült kód a korábban megírt szakdolgozatomban[1] ismertetett nyelvtant használja, amelyet itt röviden ismertetek. Az ebben a dolgozatban használt reguláris kifejezések kétféle karakterből épülnek fel: literálokból és metakarakterekből. A literálok bármilyen karakterek, amelyeket a keresett szövegnek tartalmaznia kell a megadott sorrendben. Ezek speciális szerepet is tölthetnek be egyes metakarakterek hatására. A metakarakterek különleges jelentéssel bírnak, a keresett szövegben nem literálként fognak szerepelni. Használhatóak literálként a metakarakterek a regexben, amennyiben egy `\` karakter előzi meg őket. A kódgenerátor a következő metakaraktereket ismeri:

⁴Ezek a nyelvi elemek túlmutatnak a reguláris nyelveken.

2.3.1. Zárójel

Zárójel () használható egyes elemek csoportosítására. Önmagában a zárójel nem bír jelentéssel, viszont a zárójelben lévő elemek egy egységet fognak alkotni. Példaképpen az $(ab)^+$ reguláris kifejezés tetszőlegesen sok ab -re illeszkedik, míg az ab^+ egy a betőre, majd tetszőlegesen sok b betőre illeszkedik.

2.3.2. Tetszőleges karakter

A `.` karakter tetszőleges karaktert jelöl. A `al.a` reguláris kifejezés az `alfa`, `alga` és `alma` szavakra is illeszkedik.

2.3.3. Elágazás

A `|` karakter elágazást jelöl, miszerint a kifejezés bal és jobb oldalán lévő kifejezések közül bármelyik elfogadható. Az `al(g|m)a` kifejezés az `alma` és az `alga` szavakra is érvényes.

2.3.4. Karaktercsoport

A szögletes zárójel (`[]`) karakter csoportokat jelöl. A karakter csoportban található bármelyik karakter illeszkedik rá. Karakter csoportban megadható több karakter, vagy karaktereknek egy sora. Kétféle jelölést tartalmazhat: lehetnek benne literálok egymás után (`[abcd]`), vagy kötőjellel elválasztva (`[a-z]`). A kötőjellel megadott karaktertartomány tartalmazza a két megadott karaktert, valamint az összes karaktert, amely a két határoló karakter között helyezkedik el. Ez a kétféle jelölés keverhető is (`[abc-fA-Z]` ⁵). A karaktercsoport tartalmazhat kötőjelet literálként, amennyiben az első, vagy az utolsó megadott karakter. A karaktercsoport tartalmazhat `]` karaktert, amennyiben az első megadott karakter. Ha egy karaktercsoport tartalmaz kötőjelet és `]`-t is, akkor azt a következőképpen lehet megadni: `[]-regex]`.

⁵Minden angol nagybetű, vagy kisbetű `a` és `f` között

2.3.5. Negált karaktercsoport

Amennyiben egy karaktercsoport első karaktere `^`, akkor azt negált karaktercsoportként kezeljük. A negált karaktercsoport illeszkedik minden karakterre a karaktercsoport bármelyik elemén kívül. A benne található elemek megadásának szintaxisa megegyezik a 2.3.4 részben meghatározottal.

2.3.6. Legalább egy ismétlődés

A Legalább egy ismétlődés metakarakter `(+)` önmagában nem bír jelentéssel. Kizárólag más, önmagában jelentéssel bíró elem után tehető (literál, karaktercsoport, negált karaktercsoport, tetszőleges karakter, vagy zárójel). Ilyenkor annak az elemnek a bemenetben legalább egyszer ismétlődnie kell. A `alm+a` kifejezés illeszkedik az `alma`, `almma` szavakra, de nem illeszkedik az `ala` szóra.

2.3.7. Legfeljebb egy ismétlődés

A Legfeljebb egy ismétlődés metakarakter `(?)` önmagában nem bír jelentéssel. Kizárólag más, önmagában jelentéssel bíró elem után tehető (literál, karaktercsoport, negált karaktercsoport, tetszőleges karakter, vagy zárójel). Ilyenkor az elem, amelyre a metakarakter vonatkozik legfeljebb egyszer fordulhat elő a bemenetben. Az `alm?a` kifejezés illeszkedik az `alma` és az `ala` szavakra, de nem illeszkedik az `almma` szóra.

2.3.8. Tetszőleges számú ismétlődés

A Tetszőleges számú ismétlődés metakarakter `(*)` önmagában nem bír jelentéssel. Kizárólag más, önmagában jelentéssel bíró elem után tehető (literál, karaktercsoport, negált karaktercsoport, tetszőleges karakter, vagy zárójel). Ilyenkor az elem, amelyre a metakarakter vonatkozik tetszőleges mennyiségben előfordulhat a bemenetben, vagy hiányozhat is. Az `alm*a` kifejezés illeszkedik az `alma`, `ala` és az `almma` szavakra is.

2.3.9. Adott számú ismétlődés

Az Adott számú ismétlődés metakarakter `{ }` önmagában nem bír jelentéssel. Kizárólag más, önmagában jelentéssel bíró elem után tehető (literál,

karaktercsoport, negált karaktercsoport, tetszőleges karakter, vagy zárójel). Ilyenkor az elemnek, amelyre a metakarakter vonatkozik a kapcsos zárójelek között megadott mennyiségben kell ismétlődnie. A mennyiség négyféleképpen megadható:

- $\{n\}$: Pontos mennyiségként (pontosan n -szer)
- $\{n, \}$: Alsó határként (legalább n -szer)
- $\{, m\}$: Felső határként (legfeljebb m -szer)
- $\{n, m\}$: Intervallumként (legalább n -szer, de legfeljebb m -szer)

Az $alm\{2,4\}a$ kifejezés illeszkedik az $almma$, $almma$ és az $almmmma$ szavakra, de az $alma$, vagy az $almmmma$ szavakra már nem.

3. fejezet

Kód generálása Java nyelven

A generált kód eredetileg Java nyelvre lett fejlesztve. Az NFA elkészítéséhez a McNaughton–Yamada–Thompson algoritmus[6] egy módosított változata lett implementálva, majd az ebből elkészült DFA állapotaiból készült Java forráskód. A generált forráskód több lépésben került optimalizálásra, végső futási ideje megközelíti a Java nyelvbe beépített regex motort, sok esetben gyorsabb nála.

3.1. Kódgenerálás folyamata

A kódgenerálás folyamata három fő lépésből áll:

1. Regex beolvasása
2. Regexhez tartozó DFA elkészítése
 - (a) ϵ -NFA elkészítése
 - (b) ϵ -NFA-ból DFA elkészítése
 - (c) DFA minimalizálása
3. DFA-ból Java forráskód elkészítése

A regexek beolvasása egy egyszerű rekurzív módszerrel történik, amely a szakdolgozatomban[1] bővebben ki van fejtve. A reguláris kifejezések nyelvtanát a következő Backus–Naur-forma¹ írja le:

¹Környezetfüggetlen nyelvtanok leírására használható metaszintaxis

$$\begin{aligned}
 \langle \text{regex} \rangle & \models \langle \text{expression} \rangle \\
 \langle \text{expression} \rangle & \models \langle \text{term} \rangle \mid \langle \text{term} \rangle \langle \text{expression} \rangle \mid \langle \text{term} \rangle " \mid " \langle \text{expression} \rangle \\
 \langle \text{term} \rangle & \models \langle \text{factor} \rangle \mid \langle \text{factor} \rangle \langle \text{quantifier} \rangle \\
 \langle \text{factor} \rangle & \models \langle \text{character} \rangle \mid \\
 & \quad "." \mid \\
 & \quad "(" \langle \text{expression} \rangle ")" \mid \\
 & \quad "[" \langle \text{characterclass} \rangle "]" \\
 \langle \text{character} \rangle & \models \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \mid \langle \text{specialcharacter} \rangle \\
 \langle \text{escapedcharacter} \rangle & \models "\backslash" \langle \text{character} \rangle \\
 \langle \text{characterclass} \rangle & \models \langle \text{classcontent} \rangle \mid \\
 & \quad "^" \langle \text{classcontent} \rangle \mid \\
 & \quad "]" \langle \text{classcontent} \rangle \mid \\
 & \quad "]" - " \langle \text{classcontent} \rangle \\
 \langle \text{classcontent} \rangle & \models \langle \text{classelement} \rangle \mid \langle \text{classelement} \rangle \langle \text{classcontent} \rangle \\
 \langle \text{classelement} \rangle & \models \langle \text{character} \rangle \mid \\
 & \quad \langle \text{character} \rangle "-" \langle \text{character} \rangle \mid \\
 & \quad \langle \text{escapedcharacter} \rangle \\
 \langle \text{quantifier} \rangle & \models "+" \mid \\
 & \quad "?" \mid \\
 & \quad "*" \mid \\
 & \quad "{" \langle \text{integer} \rangle " ," \langle \text{integer} \rangle "}" \mid \\
 & \quad "{" \langle \text{integer} \rangle " ," }" \mid \\
 & \quad "{" , " \langle \text{integer} \rangle "}" \mid \\
 & \quad "{" \langle \text{integer} \rangle "}" \\
 \langle \text{letter} \rangle & \models "a" \dots "z" \mid "A" \dots "Z" \\
 \langle \text{digit} \rangle & \models "0" \dots "9"
 \end{aligned}$$

$$\begin{aligned} \langle \text{specialcharacter} \rangle \models & \text{"+"} \mid \\ & \text{"?"} \mid \\ & \text{"*"} \mid \\ & \text{"."} \mid \\ & \text{"("} \mid \\ & \text{")"} \mid \\ & \text{"["} \mid \\ & \text{"]"} \mid \\ & \text{"^"} \mid \\ & \text{"{"} \mid \\ & \text{"}" } \mid \\ & \text{"|"} \mid \\ & \text{"_"} \end{aligned}$$

$$\langle \text{integer} \rangle \models \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{integer} \rangle$$

3.1.1. Nemdeterminisztikus állapotgép készítése

Első lépésként elkészítjük a regexhez tartozó NFA-t. A folyamat a szakdolgozatomban[1] leírtak alapján történik.

Az algoritmus által létrejött NFA tartalmaz ϵ -éleket. Ilyen állapotgép esetén nem egyértelmű, hogy melyik állapotban vagyunk, mert az ϵ -éleken keresztül az állapot megváltozhat bemenet nélkül.

A reguláris kifejezések egyes részeiből kis méretű véges automatákat készítünk, amelyeket később össze tudunk csatolni a következő módon:

1. Legyen N_1, N_2 ϵ -NFA
2. Legyen N_1s N_1 kezdő csúcsa
3. N_1e N_1 elfogadó csúcsa
4. N_2s N_2 kezdő csúcsa
5. N_2e N_2 elfogadó csúcsa.

6. Elágazunk attól függően, hogy mi a kapcsolat N_1 és N_2 között.

- (a) **Összefűzés** (N_2 N_1 után következik): Vegyünk egy ϵ élet N_1e -ből N_2s -be.
- (b) **Unió** (N_2 és N_1 párhuzamosan következnek): Vegyünk fel ϵ éleket a kiinduló csúcsból a N_1s -be és N_2s -be, majd vegyünk fel ϵ éleket N_1e -ből és N_2e -ből az elfogadó csúcsba.

3.1.2. Determinisztikus állapotgép elkészítése

Az NFA-ból determinisztikus állapotgépet készítünk, majd azt minimalizáljuk. Az algoritmus a következőképpen működik:

Minden NFA-beli csúcsra lefuttatjuk a következő iterációt:

1. Összegyűjtjük az összes olyan NFA-beli A csúcsot, amelyekbe ϵ -élekkel eljuthatunk
2. Felveszünk minden karakterre élt, amely karakterrel van él valamely NFA-beli csúcsból
3. Az élekhez összegyűjtjük az összes olyan NFA-beli csúcsot, amelybe az éleken keresztül juthatunk
4. Ezekből az NFA-beli csúcshalmazokból készítünk egy új DFA csúcsot².
5. Kiszámoljuk az alapértelmezett élet (ez az eredeti szakdolgozatban[1] bővebben ki van fejtve)

3.2. Generált kód általános felépítése

A generált kód eredetileg Java nyelvre lett fejlesztve. Java környezetben a bemenet String típusú, az elvárt kimenet pedig egy iterálható halmaz, amely megadja az egyes találatok helyét. Egy Java osztályt generálunk, amely tartalmaz egy *findMatches* függvényt. Ez a függvény adja vissza a bemeneti szöveghez tartozó talált mintákat.

A *findMatches* függvény a következőképpen működik:

²Amennyiben már létezik olyan DFA-beli csúcs, amely pontosan ezeket az NFA-beli csúcsokat tartalmazza, akkor azt vesszük, nem hozunk létre újat

1. Inicializáljuk a folyamat során használt változókat. A következő változók vannak jelen a keresés során:
 - **Index:** A bemeneti szövegben a jelenlegi karakter indexe.
 - **Allapot azonosítója:** A véges automatában a jelenlegi állapot azonosítója.
 - **Aktuális keresés kezdete:** A jelenlegi keresés melyik indexnél kezdődött a bemeneti szövegben
 - **Megtalált minták:** A megtalált minták kezdő- és záró indexei. A minta $[i_{\text{kezdő}}, i_{\text{záró}})$ intervallumban található.
 - **Legutolsó sikeres találat indexe:** Kereséskor minél hosszabb találatokat keresünk, ezért eltároljuk a legutolsó találat végét és keresés újraindításakor ez az index lesz a találat vége.
2. Végigmegyünk a bemeneti szövegen karakterről karakterre
 - (a) Elágazunk a jelenlegi állapot alapján
 - (b) Elágazunk a jelenlegi karakter alapján
 - i. Továbbmegyünk a következő állapotba
 - ii. Amennyiben van találat, eltároljuk a találat indexét
 - iii. Amennyiben újraindul a keresés és volt találat, eltároljuk a megtalált minták között
 - (c) Növeljük a jelenlegi karakter indexét
3. Visszatérünk a megtalált mintákkal

A függvény Java nyelven a következőképpen épül fel:

```
public class Parser_Impl implements RegexParser {
    public ParseResult findMatches(String inputString) {
        // Initialize variables
        while (currentIndex < inputString.length()) {
            currentCharacter = inputString.charAt(currentIndex);
            switch (currentState) {
                case 0: {
                    switch (currentCharacter) {
```

```

        case 'a': {
            currentState = 2;
            break;
        }
        // Other inputs
    }
    break;
}
// Accepting state
case 1: {
    lastSuccessfulMatchAt = currentIndex;
    switch (currentCharacter) {
        case 'a': {
            // Restarting search
            currentState = 0;
            foundMatches = new ParseResultMatch(
                currentMatchStartedAt,
                lastSuccessfulMatchAt
            );
            currentIndex = lastSuccessfulMatchAt > currentMatchStartedAt
                ? lastSuccessfulMatchAt - 1
                : currentMatchStartedAt;
            currentMatchStartedAt = currentIndex + 1;
            break;
        }
        // Other inputs
    }
    break;
}
// Other states
}
currentIndex++;
}
return new foundMatches;
}
}

```

3.3. Optimalizálás Java környezetben

A felvázolt működés két okból nem lesz optimális a gyakorlatban:

- A generált kód egy függvényt alkot, amely nagyon nagy is lehet. Java függvény nem állhat több, mint 2^{16} utasításból[4, 4.7.3. bekezdés] A gyakorlatban 8000 sor feletti forráskódot a Java fordító már nem fordít, így a fordítási fázison nem jutunk túl.
- A generált kódot nem gyorsítja a JIT³ fordító olyan mértékben, mint elvárnánk. Habár egyes nyelvekben minden optimalizáció fordítási időben megtörténik, Java esetén futás közben is optimalizálásra kerül a kód. Az elkészülő függvény túl nagy és a gyakorlatban a fordító nem talál olyan mintát futás közben, amit optimalizálhat.

Az optimalizált Java kód rövid függvényeket tartalmaz, minél kevesebb példányosított osztállyal. Java kód optimalizálásakor nehéz megállapítani, hogy milyen strukturális átalakítások fognak gyorsabb kódot eredményezni. A kódot túlságosan rövid, vagy hosszú függvényekre szétszedni ugyanolyan káros lehet a futási idő szempontjából. Az osztályok példányosításának elkerülése gyorsítja a kódot, mert nem kell a virtuális függvénytáblához hozzáférnie a az alkalmazásnak futás közben.

A JIT fordító optimalizálja azokat a függvényeket, amelyekben a program sok időt tölt el. Ezeket a függvényeket `inline`-olja, elágazásokat egyszerűsít, vagy akár natív kódot generál. Amennyiben a függvényeink egy példányban léteznek (egy külön osztályba ki vannak emelve), az optimalizációs lépés függvényenként egyszer fog lefutni, így minimalizáljuk a fordítás idejét.

3.3.1. Több fájlra szétszedés

Az első felmerülő problémát megoldja az, ha az állapotgép egyes állapotait külön osztályokra bontjuk le. Ezek az osztályok nem csinálnak mást, mint a jelenlegi karakter szerint léptetik az állapotgépet.

Az így szétbontott osztályok hossza nem érheti el a sok ezer sort (a bennük található elágazásnak maximum 256 esete lehet), így a fordítási probléma nem lép fel.

Az egyes állapotok osztályai a következőképpen épülnek fel:

³Just In Time

```
public class Parser_impl_state_1 {  
    public void run() {  
        switch (currentCharacter) {  
            case 'a': {  
                currentState = 2;  
                break;  
            }  
            // Other inputs  
        }  
    }  
}
```

Továbbra is fennáll a probléma, hogy pár ezer, vagy akár pár száz állapotos állapotgép esetén el tudjuk érni a maximális osztályméretet. Ezért bevezetünk egy köztes osztályt, amely valamennyi állapotot (2^4) összevon és a megfelelő állapotkezelőnek továbbítja a függvényhívást, így csökkentve a fő osztályban lévő kódmennyiséget.

Az egyes köztes osztályok felépítése a következő:

```
public class Parser_impl_part_0 {  
    public void run() {  
        switch (currentState) {  
            case 0: {  
                Parser_impl_state_1.run();  
                break;  
            }  
            case 1: {  
                Parser_impl_state_1.run();  
                break;  
            }  
        }  
    }  
}
```

Bekerül a fő osztályba egy új változó, amely megadja, hogy melyik köztes osztályt kell meghívunk. Emellett a jelenlegi állapothoz tartozó azonosító inntől csak a köztes osztályon belüli azonosítót fogja jelenteni (*mod* 2^4).

A fő osztály javított felépítése:

```
public class Parser_Impl implements RegexParser {  
    public ParseResult findMatches(String inputString) {
```

```
// Initialize variables
while (currentIndex < inputString.length()) {
    currentCharacter = inputString.charAt(currentIndex);
    switch (currentStateHandler) {
        case 0: {
            TesParser_impl_part_0.run();
            break;
        }
    }
    currentIndex++;
}
return new foundMatches;
}
```

Az itt számított `currentStateHandler` a korábbi `currentState` 4-el jobbra shift-elt értéke, a `currentState` pedig innentől a korábbi `currentState` utolsó 4 bitje lesz.

3.3.2. Statikus függvények használata

Amennyiben szétszedjük a forráskódot állapotonként egy-egy osztályra, a kód már fordítható, viszont meglehetősen lassú. Legfőbb oka a `vtable`⁴ olvasásával töltött idő, amelynek forrása az, hogy az osztályokat példányosítjuk és az egyes példányok függvényeit használjuk.

Erre megoldást jelent az, ha minden osztály statikus függvényeket és változókat definiál. Az így újrastrukturált kódban az egyes állapotkezelők és köztes osztályok nincsenek példányosítva, nem kell a `vtable`-ben keresnie a JVM⁵-nek az egyes függvények helyét a memóriában.

3.3.3. Ismétlődő kód kiemelése

Az egyes függvények belsejében található kódrészeket külön-külön `bytecode`⁶-nak felelnek meg. Ezeket a különböző kódrészeket a JIT fordító nem tudja összevonni, nem tudja egységesen optimalizálni. Amennyiben egy állapotból

⁴virtuális függvénytábla

⁵Java virtuális gép, a fordított kódot futtató környezet

⁶Java bytecode, a JVM által értelmezhető utasításkészlet

egy másikba lépünk, ugyanaz a kódrészlet fog lefutni, ezeket ki tudjuk emelni. Amennyiben ezeket a kódrészleteket összevonjuk és minden állapotátmenetnél ugyanazon függvényt használjuk, jelentős teljesítmény-javulást érhetünk el.

Létrehozunk egy segédosztályt, amely tartalmaz háromféle függvényt:

- Állapotgép léptetése adott állapotba
- Keresés újratekzdése
- Találat mentése

A segédosztály használatával az egyes állapotkezelők felépítése a következővé változik:

```
public class Parser_impl_state_1 {
    public void run() {
        switch (Parser_Impl.currentCharacter) {
            case 'a': {
                Parser_Impl_util.jumpToState2();
                break;
            }
            // Other inputs
            default: {
                Parser_Impl_util.restartSearch();
                break;
            }
        }
    }
}
```

3.3.4. Egyéb optimalizációk

A fő while ciklusban tudunk még optimalizálni a futási időn két sor átírásával:

- A while ciklus feltételében lévő `inputString.length()` függvény értékét elég a változók inicializálásakor elvégezni, így a while ciklusban ez nincs fölöslegesen kiértékelve
- A `currentCharacter` számolása esetén egyes esetekben gyorsabb, ha az `inputString.charAt(currentIndex)` helyett a `String`-hez tartozó

`char []`-t használjuk, amelyet előre kiszámítunk az `inputString.toCharArray()` hívással.

4. fejezet

Kód optimalizálása más nyelveken

4.1. Kód felépítése C++ nyelven

A Java nyelvű kód átilleszthető egyéb strukturális változtatás nélkül C++ nyelvre, viszont teljesítményjavulást nem tapasztalunk a kód felépítésének megváltoztatása nélkül (a generált C++ kód kevesebb, mint 10% teljesítményjavulást eredményez). C++ nyelvben kevésbé hatékony a függvényhívás (a kontextusváltás drága műveletnek számít, sok CPU utasítást igényel) és nincsen korlát fájlok, osztályok, vagy függvények hosszára. A generált kód újrastrukturálásával jelentős gyorsítások érhetőek el.

4.1.1. Osztályok összevonása

Az optimalizált Java kóddal ellentétben nem ideális újabb osztályokat létrehozni, vagy külön függvényekbe gyűjteni a kód különböző részeit. A fő osztály egyetlen függvényének törzsébe rakhatjuk közvetlenül az állapot szerinti elágazást, majd azon belülre a jelenlegi karakter alapján történőt. Így elkerüljük a változók másolását és minimalizáljuk a fordító által okozott optimalizálási hibák lehetőségét.

```
class GeneratedParser {
public:
    static std::deque<std::pair<int, int>>
        findMatches(std::string inputString) {
        // Initialize variables
        while (currentIndex < stringLength) {
            currentCharacter = characters[currentIndex];
            switch (currentState) {
                case 0:
```

```

lastSuccessfulMatchAt = currentIndex;
switch (currentCharacter) {
    case ' ':
        currentState = 2;
        break;
    default:
        currentState = 1;
        break;
}
break;
// Handle other states
case 5:
    switch (currentCharacter) {
        case ' ':
            currentState = 3;
            break;
        default:
            currentState = 1;
            if (lastSuccessfulMatchAt > currentMatchStartedAt) {
                found.push_back(
                    std::pair<int, int>(
                        currentMatchStartedAt,
                        lastSuccessfulMatchAt
                    )
                );
            }
            currentIndex = lastSuccessfulMatchAt > currentMatchStartedAt
                ? lastSuccessfulMatchAt - 1
                : currentMatchStartedAt;
            currentMatchStartedAt = currentIndex + 1;
            break;
        }
    }
}
}
}
}

```

4.1.2. Kétszeres elágazás összevonása

A jelenlegi megoldás kettő elágazást tartalmaz: a jelenlegi állapot, majd a bemenet jelenlegi karaktere alapján. Amennyiben kevesebb, mint 2^{24} állapota van a véges automatának, megoldható az elágazás egy összevont **switch-case** használatával, ahol az elágazó változó első 24 bitje a jelenlegi állapot, az utolsó 8 bitje pedig a jelenlegi bemeneti karakter.

Ez a megoldás nem eredményez gyorsabb futási időt a gyakorlatban az előzőleg felvázolthoz képest. Ahhoz, hogy egy **byte**-ot és egy 32 bites számot össze tudjunk vonni, a 8 bites tagot át kell másolnunk a megfelelő memória-helyre, ki kell töltenünk 0-val, majd csak utána tudunk vele dolgozni. Emellett kettő darab kisebb **switch-case** helyett egy hatalmas **switch-case** kerül be a forráskódba, amelyet sokkal nehezebb hatékonyan kihasználni.

Ezzel a megközelítéssel a futási idő az optimalizálatlan kódhoz képest is lassul ($\sim 60\%$ -al megnő a futási idő ".5,32_" regex esetén).

4.1.3. Fordítás

Fordításnál az -O1 fordító flaget alkalmaztuk. Optimalizáló flagek nélkül a futási idő $\sim 100\%$ -kal, magasabb optimalizáló flagekkel (-O2, -O3, vagy -Ofast) $\sim 15\text{-}20\%$ -kal lassabb futási időt érünk el (" .5,32_" regex esetén).

Az így fordított alkalmazás esetében a fordító felismeri azt, hogy egyes változók értékét fölöslegesen eltárolni és kizárólag az elágazásnál vannak használva (`currentCharacter`, `stringLength`).

4.2. Python nyelvű implementáció problémái

A Python kód felépítése eltér mindkét korábban említettől. A Python a 3.10-es verzió óta támogat **match-case** típusú elágazást¹, amely hasonlít a **switch-case** típusúhoz, viszont jelentősen lassabb kiszámítani. Ez két helyen lassítja a futást:

- A jelenlegi állapot alapján történő elágazás
- A jelenlegi karakter alapján történő elágazás

¹Bővebben a Python 3.10 release log-ban lehet erről olvasni

Mindkét esetben $\mathcal{O}(1)$ -nél lassabb lesz az elágazás. Kétféleképpen tudjuk megközelíteni az elágazások helyettesítését: egymásba ágyazott `if-elseif` feltétel-vizsgálattal, vagy `match-case`-el.

A gyakorlatban a leggyorsabb megoldást az nyújtja, ha példányosítunk minden állapothoz egy-egy osztályt, amelyen belül `match-case` alapján határozzuk meg azt, hogy milyen módon kell a jelenlegi állapotot változtatnunk. Amennyiben a jelenlegi állapot esetében is `match-case`-t alkalmazunk, a futó alkalmazás jelentősen nagyobb futási időt eredményez (~ 3 -szoros futási idő ".5,32_" regex esetén).

A használt Python kód struktúrája a következő:

```
class parser_6:
    class State_0:
        def __init__(self, p):
            self.parent = p
            # In handle(), a "self.parent." prefix is added
            # to each parameter, but it is omitted here
            # for the sake of readability
        def handle(self, c):
            lastSuccessfulMatchAt = currentIndex;
            match c:
                case ' ':
                    currentState = 2
                    currentStateHandler = state_2
                case _:
                    currentState = 1

    class State_5:
        def __init__(self, p):
            self.parent = p
        def handle(self, c):
            lastSuccessfulMatchAt = currentIndex;
            match c:
                case ' ':
                    currentState = 3
                    currentStateHandler = state_3
                case _:
                    currentState = 1
                    currentStateHandler = state_1
```

```
        if (lastSuccessfulMatchAt > currentMatchStartedAt):
            found.append(
                (currentMatchStartedAt, lastSuccessfulMatchAt)
            )
        currentIndex = ( lastSuccessfulMatchAt - 1
            if lastSuccessfulMatchAt > currentMatchStartedAt
            else currentMatchStartedAt )
        currentMatchStartedAt = currentIndex + 1

def __init__(self):
    # Instantiate state handler objects
    self.state_0 = self.State_0(self)
    self.state_1 = self.State_1(self)
    # (...)

def findMatches(self, inputString):
    # Initialize variables
    while (self.currentIndex < self.stringLength) :
        self.currentCharacter = self.characters[self.currentIndex]
        self.currentStateHandler.handle(self.currentCharacter)
        self.currentIndex = self.currentIndex + 1
    return self.found
```

5. fejezet

Teljesítmény vizsgálata

Az elkészült kód sebességét kétféleképpen vizsgáljuk meg. Először összehasonlítjuk a generált C++ és Python kódot a Java kóddal és megvizsgáljuk, hogy az alacsonyabb nyelven készített kód javít-e a teljesítményen, illetve, hogy mennyire befolyásolják a futási időt a Python nyelv limitációi.

Következőnek összehasonlítjuk a generált C++ kódot már meglévő regex könyvtárak feldolgozási sebességével, majd a .NET által készített generált regex olvasó futási idejével.

Mindkét esetben a következő 7 regexet használjuk:

- `.5,32_`
- `([a-zA-F0-9]{8}:){3}[a-zA-F0-9]{8}`
- `.{2,6}[^A-Z]{5,8}ABCD`
- `AB[A-Z]{2,}[a-z]{0,5}C`
- `.{0,30}ABCDE`
- `[a-f]+[g-q]*([a-q])?a*`
- `AB([A-K]{3,}|[C-Q]{0,7}|[N-Z]+)C`

Ezek a regex-ek lefedik a 2.3 pontban meghatározott nyelvtan egészét.

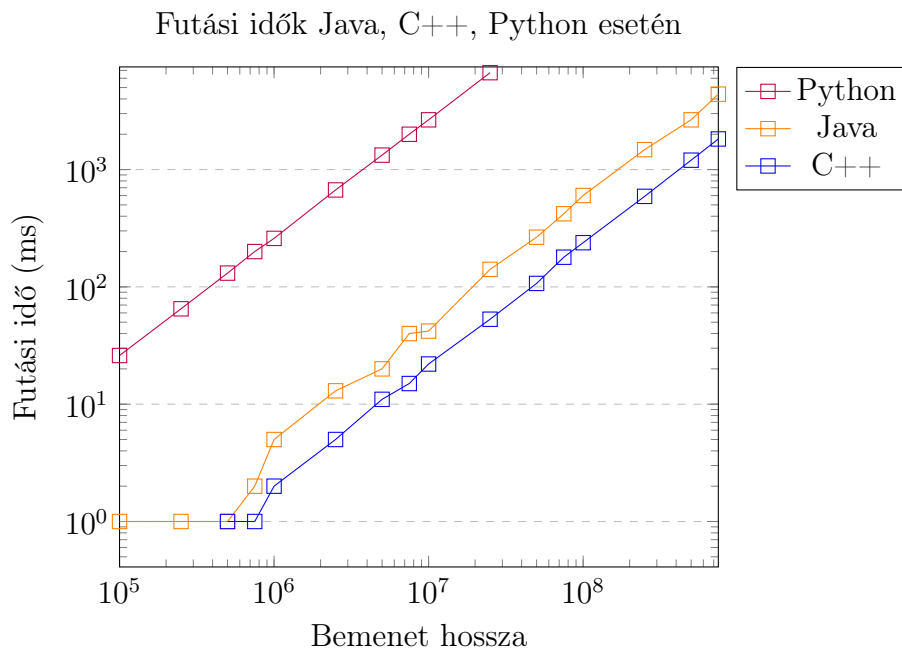
Az egyes regexek esetén generálunk olyan bemenetet, amelyről tudjuk, hogy az adott minta illeszkedik rá. A minták illeszkedésének száma akár több nagyságrenddel is eltérhet a különböző regexek esetén. Minden esetben háromféle bemenettel teszteljük a futási időket a minta illeszkedéseinek számától függően:

kevés illeszkedés, közepes számú illeszkedés, sok illeszkedés. Az illeszkedések eloszlása homogén a bemeneten belül.

Minden regex esetén többször futtatunk tesztet, amelyeknek mediánját vesszük az adott regex esetén a futási időnek. Java és Python kód esetén futtatunk "bemelegítő" köröket, amely lehetőséget ad a futási idejű optimalizációra. Az ilyen bemelegítő körök idejét nem számoljuk bele a végső futási időbe.

5.1. Java, C++, Python kód összehasonlítása

5.1.1. Futásidő változása a bemenet hosszának függvényében



5.1. ábra. Futási változása bement függvényében

A bemenet függvényében a futási idő lineáris (gyakorlatban közel lineáris), amely látható az 5.1 ábrán. A memóriahasználat arányos a bemenetben található minták számával, amely homogén bemenetben arányos a bemenet hosszával.

5.1.2. Futási idő a teszt regexek esetén

A Python kód nagyságrendekkel lassabb volt, ezért kisebb bemeneten futtatuk és a mért eredményeket felskáláztuk (2.5×10^7 helyett 2.5×10^6 hosszú bemenetet használtunk és a futási időket beszoroztuk 10-zel). A futási idő linearitása és a bemenetek generálásának módja miatt ez pontos eredményt ad.

A futási időknél az tapasztalható, hogy a C++ kód minden esetben kevesebb idő alatt végez, mint a generált Java kód, amely minden esetben gyorsabb, mint a Python kód.

A Python kód a C++ kódhoz képest 57-171-szer magasabb futási időt eredményezett, míg a Java kód 1.61-7.7-szer volt lassabb, mint a C++ kód.

5.2. C++-ban generált re2, pcre2 összehasonlítása

Kettő C++ nyelven használható regex könyvtár teljesítményét hasonlítottuk össze a generált kóddal: a Google által karbantartott re2¹ és a PCRE2Project által karbantartott pcre2² könyvtárat.

5.2.1. Egyszerű regexek

Az első három regex esetén (5.2, 5.3, 5.4 ábrák) a futási időkből ugyanaz a fellépő minta: amennyiben a bemenet nagyobb része illeszkedik a regex-re (több mintát találunk benne), a futási idő a generált kódban és a pcre2 könyvtárt használva csökkennek, míg az re2 könyvtár futási ideje nő. Nem tudtuk egyértelműen megállapítani, hogy az re2 és a pcre könyvtárak futási idői miért így változnak, viszont a generált kód esetében igen.

A generált kód esetén a futási idő arányos a keresés újraindításainak számaival. Keresés újraindításakor naiv módszerrel keressük meg azt az indexet, amelyről a keresést újraindítjuk³, amely nem hatékony abban az esetben, ha a bemeneti szöveg kevészer, vagy ritkán tartalmazza a regex által leírt mintát.

¹<https://github.com/google/re2>

²<https://github.com/PCRE2Project/pcre2>

³Amennyiben volt mintára illeszkedés, akkor az illeszkedés végétől, egyébként a jelenlegi keresés kezdetének indexe után egyel.

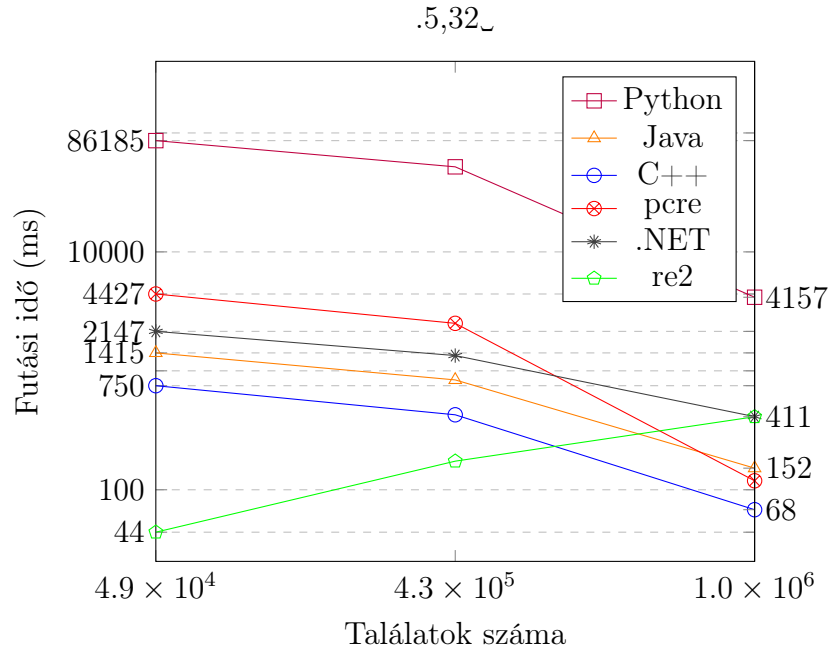
5.2.2. Egymást fedő halmazokat tartalmazó regexek

Amennyiben a regexek tartalmaztak olyan állapotot, amely esetén nehéz eldönteni azt, hogy a regexen belül melyik állapotban vagyunk (5.5, 5.6, 5.7, 5.8 ábrák), az re2 és a pcre teljesítménye kimutathatóan rosszabb volt, mint a generált kódé azokban a tesztesetekben, ahol a keresett szövegben a minták száma megfelelően nagy. A generált kódban a nem egyértelmű állapotok a DFA-vá történő átalakítás során összevonásra kerülnek, így elkerüljük az állapotok közötti ütközésből eredő bizonytalanságot. Ez a bizonytalanság lassítani tudja az NFA-alapú és a visszalépéses algoritmust használó regex motorokat, amelyet a mérési eredmény is alátámaszt.

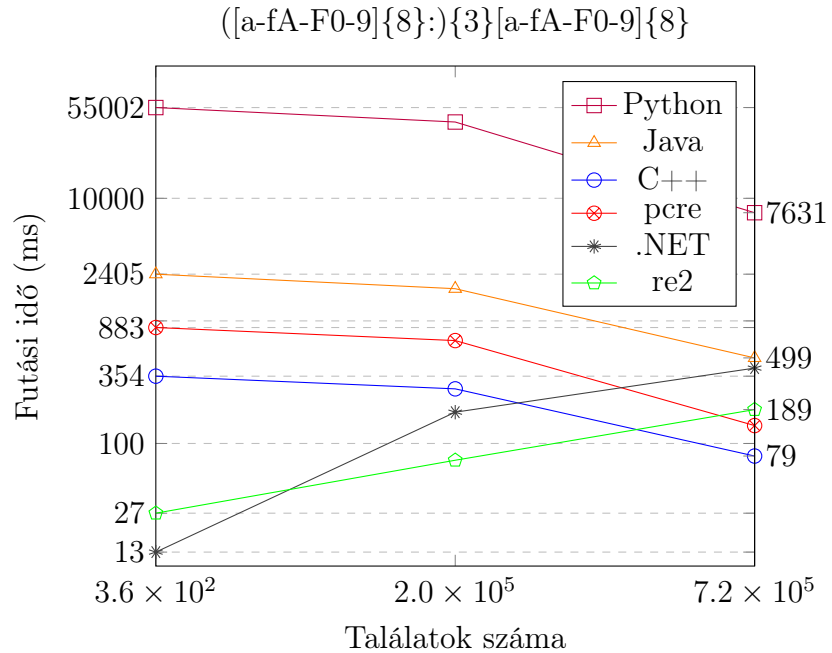
5.3. C++-ban és .NET-ben generált kód összehasonlítása

A .NET által generált kód futási időiben az általunk használt teszt-regexek alapján nehéz mintát találni. Ez a program eredményezte a legrövidebb futási időt mindegyik teszteset közül (5.3 ábra), ugyanakkor a legtöbb esetben lassabban kaptunk eredményt, mint a Java nyelven generált kód esetében. A mérési eredmények alapján kevésbé alkalmazható a generált C++ kód esetében látható tendencia, miszerint a találatok számának növekedésével csökken a futási idő.

Teljesítmény vizsgálata

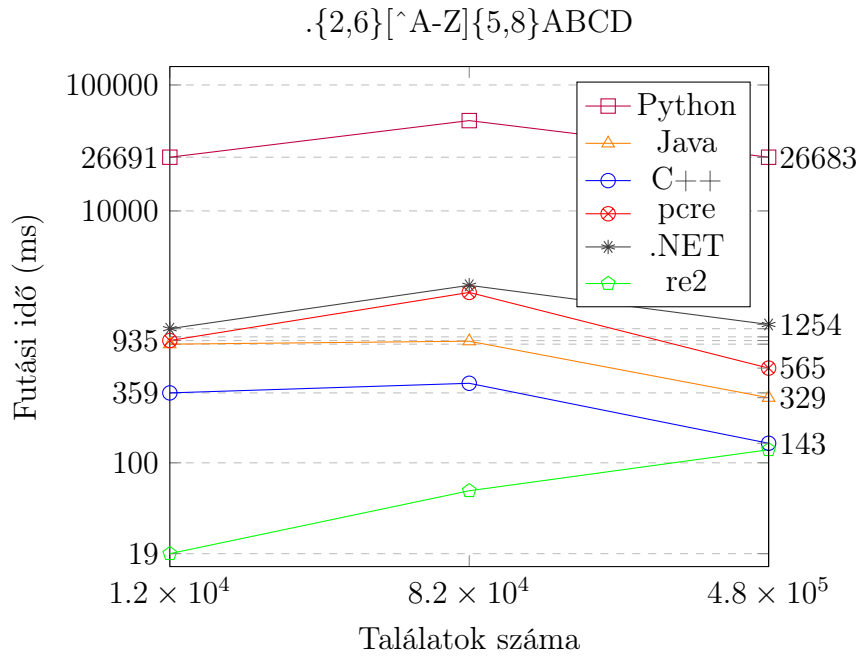


5.2. ábra. Futási idők .5,32_ regex, 2.5×10^7 hosszú bemenet esetén

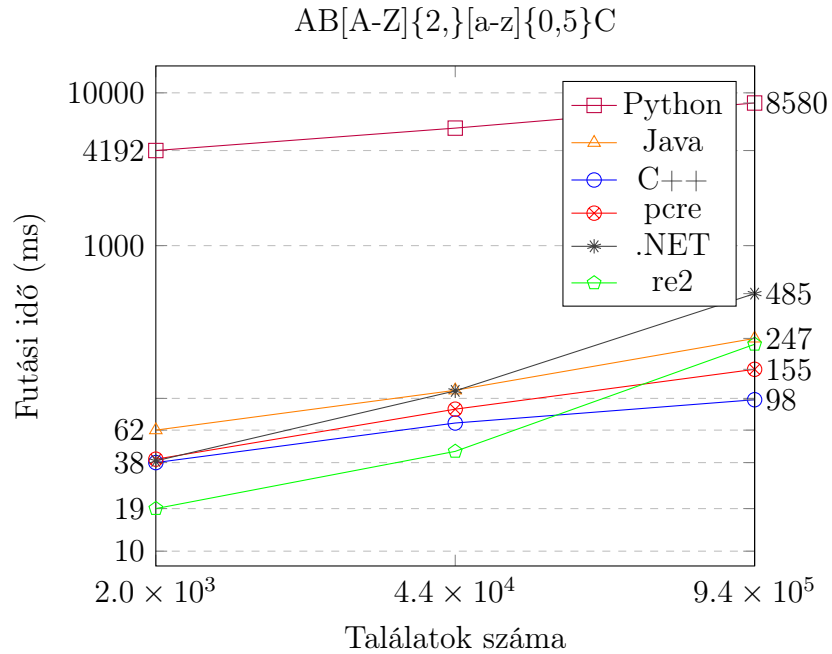


5.3. ábra. Futási idők ([a-zA-Z0-9]{8}:){3}[a-zA-Z0-9]{8} regex, 2.5×10^7 hosszú bemenet esetén

Teljesítmény vizsgálata

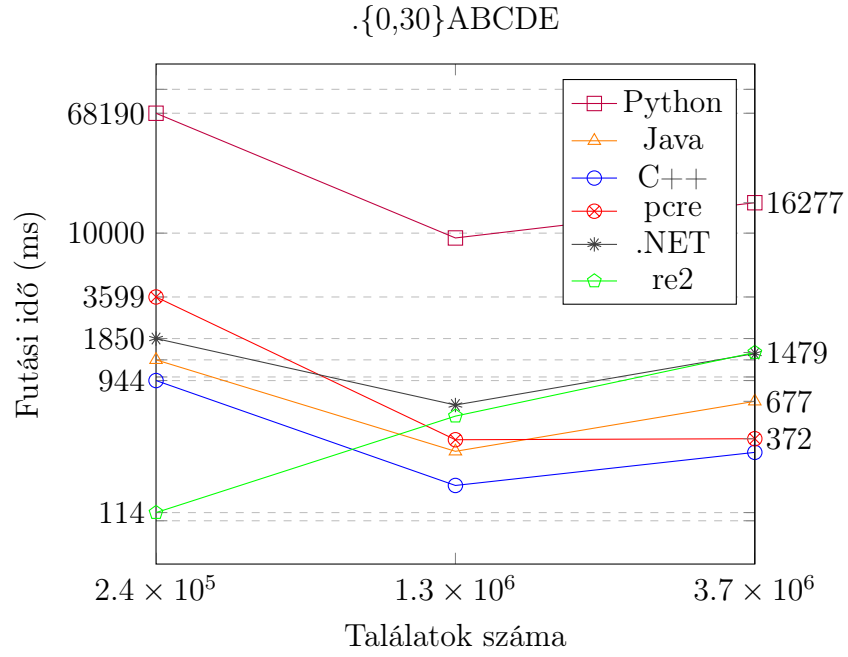


5.4. ábra. Futási idők `.{2,6}[^A-Z]{5,8}ABCD` regex, 1.0×10^7 hosszú bemenet esetén

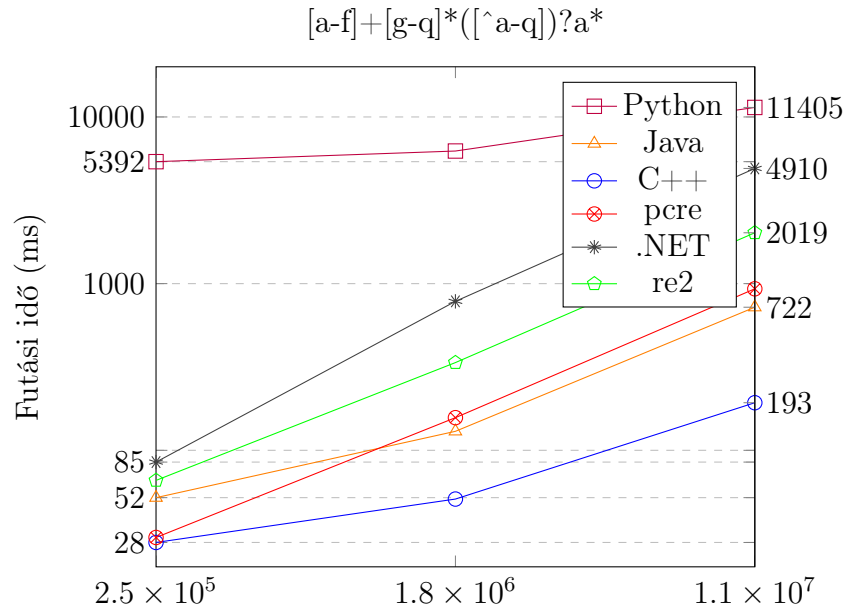


5.5. ábra. Futási idők `AB[A-Z]{2,}[a-z]{0,5}C` regex, 2.5×10^7 hosszú bemenet esetén

Teljesítmény vizsgálata

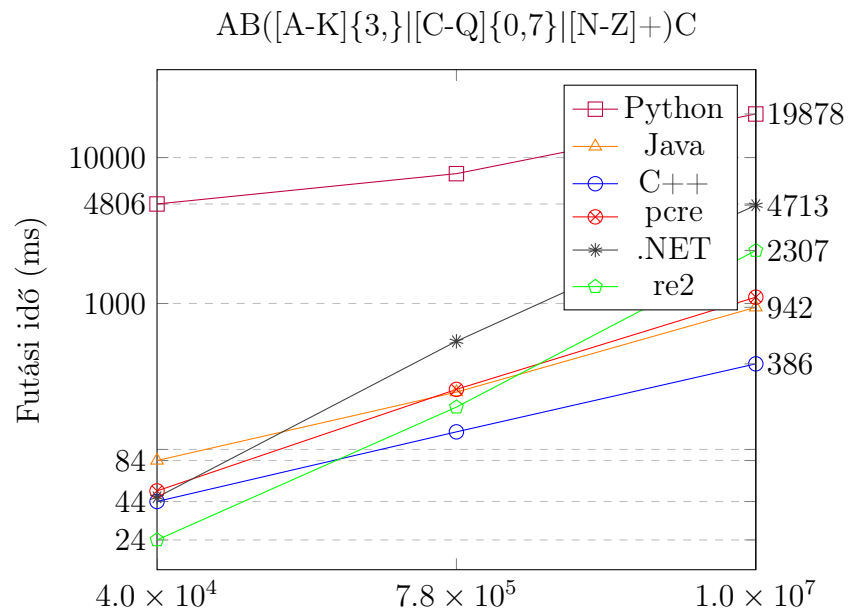


5.6. ábra. Futási idők $\{0,30\}$ ABCDE regex, 2.5×10^7 hosszú bemenet esetén



5.7. ábra. Futási idők $[a-f] + [g-q]^*(\wedge a-q)?a^*$ regex, 2.5×10^7 hosszú bemenet esetén

Teljesítmény vizsgálata



5.8. ábra. Futási idők AB([A-K]{3,}|[C-Q]{0,7}|[N-Z]+)C regex esetén

6. fejezet

Összegzés

6.1. Tapasztalatok

A mérési eredmények alapján a következőket tapasztaltuk:

- A teljesítmény javítható alacsonyabb szintű nyelvek használatával. A vizsgált nyelvek esetében minden tesztesetre igaz volt, hogy az alacsonyabb nyelven elkészített program (amely kevesebb absztrakciót és indirekciót tartalmaz) gyorsabban dolgozta fel a szöveget.
- Van olyan bemenet, amely esetén a DFA-alapú generált megoldás jelentősen gyorsabban képes szöveget feldolgozni, mint az re2, vagy a pcre2 könyvtár. A megfelelő felhasználási körülmények mellett érdemes a generált olvasót használni, amellyel valódi alkalmazások sebességét tudnánk gyorsítani.
- A C++ nyelven generált olvasó a legtöbb esetben gyorsabban adott eredményt, mint a .NET-ben generált verzió. A .NET által generált kód potenciálisan javítható az általunk használt módszerekkel.
- Az re2 könyvtár a generált kódhoz hasonló módon véges automata alapú, így a kettő megközelítés esetén arra számítottunk, hogy a futási idő is hasonló módon fog változni a bemenet függvényében. A gyakorlati tapasztalat viszont az, hogy az re2 által készített olvasó más típusú bemenetek esetén volt gyors, mint a generált. A generált kód struktúrája potenciálisan javítható az re2 működésének vizsgálatával.

A választott teszt-regexek alapján azt tapasztaltuk, hogy a nem egyértelmű

regexek esetén¹ jobban teljesít a DFA-alapú algoritmus, mint az re2, vagy a pcre könyvtárak. Ez viszont nem állapítható meg általánosan a két könyvtár forráskódjának alaposabb vizsgálata és alaposabb tesztelés nélkül.

6.2. További munkák

A tapasztalatok alapján érdemes további kutatást végezni a regex motor futási idejű generálásával kapcsolatban. Az re2 könyvtár keresés-újraindítási stratégiáját alkalmazva feltételezhetően jelentősen csökkenthető a futási idő olyan bemenetek esetén, amelyek kevésszer tartalmazzák a mintát.

Más futási idejű feldolgozó esetén is alkalmazható a kódgeneráló módszer. Olyan alkalmazások esetében, ahol deklaratív módon van leírva az alkalmazás API²-ja hatékonyabb, ha a működéshez szükséges kódot típusbiztosan, kizárólag az alkalmazás igényeit kiszolgálva készítjük el általános keretrendszer használata nélkül. Így elkerüljük a keretrendszerek absztrakcióit, indirekcióit.

¹Van olyan bemenet, amely esetén több állapotba is kerülhet

²Alkalmazásprogramozási felület

Irodalomjegyzék

- [1] Kovács Bertalan. Reguláris kifejezések hatékony kezelése java környezetben. Bsc diplomamunka, Budapesti Műszaki és Gazdaságtudományi Egyetem, 2022.
- [2] CyberZHG. toolbox (encoding and parsing tools). <https://github.com/CyberZHG/toolbox>, 2021.
- [3] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1998.
- [4] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The java virtual machine specification, 2015. <https://docs.oracle.com/javase/specs/jvms/se8/html/index.html>.
- [5] Ruslan Mitkov. *The Oxford Handbook of Computational Linguistics*. Oxford University Press, 2003.
- [6] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.
- [7] Bruce W. Watson. A taxonomy of finite automata construction algorithms. 1993.