Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Bajić

Inženiring vzporednih algoritmov

diplomsko delo

visokošolski študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2024

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Kandidat: Luka Bajić
Naslov: Inženiring vzporednih algoritmov
Vrsta naloge: Diplomska naloga na visokošolskem programu prve stopnje Računalništvo in informatika
Mentor: doc. dr. Jurij Mihelič

Opis:
V diplomski nalogi [5] je študent preizkusil različne tehnike za povzporejanje algoritma za izračun DTW (dynamic time warping) razdalje, ki sem jih nato še dopolnil. Predvidevam, da bi enake tehnike bile uporabne tudi za nekatere druge probleme, kot je razdalja LCS (longest common subsequence) ali Levenshteinova razdalja. V nalogi je torej treba te tehnike sprogramirati za omenjena problema (lahko le en problem, če bo veliko dela), nato pa jih smiselno eksperimentalno ovrednotiti.

Zahvaljujem se mentorju doc. dr. Juriju Miheliču za pomoč pri izdelavi diplomske naloge in Urošu Koritniku za moralno podporo in koristne nasvete tekom študija.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
LCS	longest common subsequence	najdaljše skupno podzaporedje
JSON	JavaScript Object Notation	JavaScript objektna notacija

Povzetek

Naslov: Inženiring vzporednih algoritmov

Avtor: Luka Bajić

Algoritmi za iskanje podobnosti med nizi so pomemben gradnik raznovrstnih aplikacij, ki so dandanes v vsakodnevni uporabi, od črkovalnikov do orodij za kontrolo različic (npr. Git). Diplomsko delo opiše različne pristope optimizacije dveh tovrstnih algoritmov: Levenshteinove razdalje in najdaljšega skupnega podzaporedja, z uporabo koncepta večnitnosti in dveh temelnjih pristopov paralelizacije - diagonalni in naprej-nazaj. Glavni cilj je časovna pohitritev algoritmov, ki jo ustrezno izmerimo in primerjamo nove implementacije z obstoječimi. Za pristop naprej-nazaj razvijemo tudi prostorsko optimizirano metodo, ki je smiselna za uporabo pri ogromnih količinah vhodnih podatkov.

Ključne besede: vzporedni algoritmi, večnitnost, Levenshteinova razdalja, LCS.

Abstract

Title: Parallel Algorithm Engineering

Author: Luka Bajić

Algorithms for finding similarity between strings are a building block of a large variety of modern applications, from spell checkers to version control tools (e.g. Git). This BSc thesis describes various approaches for optimizing two such algorithms: Levenshtein distance and Longest Common Subsequence, utilizing the concept of multithreading and two fundamental parallelization approaches - diagonal and forward-backward. Main objective is improved execution time of said algorithms, which we measure and compare our results with existing implementations. For forward-backward apporach we develop an additional space optimization, which is useful for extremely large amounts of input data.

Keywords: computer, computer, computer.

Poglavje 1

Uvod

V diplomskem delu so opisani različni pristopi za optimizacijo algoritmov za računanje Levenshteinove razdalje in najdaljšega skupnega podzaporedja (angl. Longest Common Subsequence - LCS) z uporabo vzporednega procesiranja.

1.1 Motivacija

Iskanje podobnosti med nizi se na prvi pogled zdi dokaj enostaven problem, vendar gre v resnici za obsežno in raznoliko skupino algoritmov, ki služijo kot gradniki aplikacij na številnih področjih in v različnih domenah, v katerih se običajno srečujemo z zelo specifičnimi potrebami oz. zahtevami. Tipičen primer je bioinformatika ali genetika [2], kjer pogosto naletimo na probleme z ozkim naborom znakov in lahko algoritme temu primerno tudi prilagodimo. Kot drugi primer vzamemo orodja za kontrolo različic. Tukaj lahko pričakujemo, da bosta dve datoteki, ki ju primerjamo, večinoma zelo podobni, morda samo z nekaj spremenjenimi vrsticami, posledično lahko razvijemo algoritem, ki bo tovrstna pričakovanja upošteval in se morda na nek način obnašal bolj uporabniku prijazno (npr. z nekimi hitrimi, vmesnimi aproksimacijami, če je celotno izvajanje preveč dolgotrajno). Vsekakor pa vemo, da je vsaka optimizacija algoritmov za primerjavo nizov zelo kori-

stna, zato smo v tem diplomskem delu preučili nekaj tovrstnih pristopov za algoritma za računanje LCS in Levenshteinove razdalje.

1.2 Struktura

Diplomsko delo je razdeljeno na pet poglavij. Po uvodnem delu sledi opis obeh temelnjih algoritmov, njune praktične uporabnosti in njunih implementacij z uporabo dinamičnega programiranja. Tretje poglavje predstavi dva glavna pristopa povzporejanja - diagonalni in naprej-nazaj. Pri slednjem je prikazan tudi primer prostorske optimizacije. V četrtem poglavju so predstavljeni rezultati meritev na različnih velikostih vhodnih podatkov in primerjave časov izvajanja posameznih različic algoritmov. V petem poglavju sledijo še sklepi in ideje za nadalnji razvoj.

Poglavje 2

Metode in paradigme

V tem poglavju je predstavljena algoritmična paradigma dinamičnega programiranja, ki služi kot temelj za vse algoritme, ki smo jih implementirali. Nato sta podrobneje predstavljena še osnovna algoritma za računanje LCS in Levenshteinove razdalje.

2.1 Dinamično programiranje

Dinamično programiranje je algoritmična paradigma, s katero skušamo optimizirati izvajanje problemov, ki imajo takoimenovano optimalno podstrukturo, kar pomeni, da je do rešitve problema možno priti preko optimalne rešitve njegovih podproblemov. Za to obstajata dva glavna načina: rekurzivni (z uporabo memoizacije) in iterativni (z uporabo tabeliranja). Za našo implementacijo smo uporabili iterativni pristop, kar pomeni, da hranimo vmesne rezultate v tabeli velikosti n x m. Problem primerjanja dveh nizov smo tako razbili na enostavnejše podprobleme primerjanja dveh znakov. Končni rezultat se nahaja v celici (n, m), vendar lahko iz preostanka tabele tudi rekonstruiramo pot, po kateri smo prišli do rezultata.

2.2 LCS

Po sledeči enačbi (2.1)...

$$M(i,j) = \begin{cases} 1 + M(i-1, j-1), \\ \qquad \text{če } s1[i-1] = s2[j-1] \\ max(M(i-1,j), M(i,j-1)), \\ \qquad \text{če } s1[i-1] \neq s2[j-1] \end{cases} \tag{2.1}$$

```
1 int forward_LCS (string str1, string str2, int row, int
      column) {
2     for(int i = 0; i < row; i++)
3         for (int j = 0; j < column; j++)
4             if (i == 0 || j == 0)
5                 arr[i][j] = 0;
6             else if (str1[i-1] == str2 [j-1])
7                 arr[i][j] = 1 + arr[i-1][j-1];
8             else
9                 arr[i][j] = max(arr[i-1][j], arr[i][j-1]);
10
11    return arr[row-1][column-1];
12 }
```

Izvorna koda 2.1: Algoritem LCS naprej

2.3 Levenshteinova razdalja

Predpostavimo, da lahko nad vsakim znakom v nizu izvedemo tri osnovne operacije: brisanje, vstavljanje in spreminjanje. Hipotetično lahko posameznim operacijam dodelimo tudi uteži [3], vendar ker to nima občutnega vpliva na časovno zahtevnost algoritma, v nadaljevanju sklepamo, da imajo vse tri enako utež. S tem naborom operacij obstaja teoretično neskončno možnih načinov za transformacijo nekega niza v poljuben drugi niz, Levenshteinova razdalja pa predstavlja dolžino najkrajše tovrstne transformacije.

Izračunamo jo tako, da najprej inicializiramo prvo vrstico in stolpec matrike. Ker gre tukaj za primerjavo niza s praznim nizom, je ta korak trivialen in vedno enak: v prvo vrstico vnesemo vrednosti spremenljivke j, v prvi stolpec pa vrednosti spremenljivke i, kot je prikazano na sliki (2.1) v rdečem fontu.

		a	b	c	d
	0	1	2	3	4
x	1	1	2	3	4
y	2	2	2	3	4
z	3	3	3	3	4

Slika 2.1: Primer izračunane matrike za iskanje Levenshteinove razdalje

Nato vrednosti za preostale celice v matriki izračunamo po formuli (2.2), za $i = 1, ..., n$ in $j = 1, ..., m$, kjer $s1$ in $s2$ predstavljata vhodna niza dolžin n in m. Končni rezultat dobimo v celici na poziciji (n, m).

$$M(i,j) = \begin{cases} M(i-1, j-1), \\ \qquad \text{če } s1[i-1] = s2[j-1] \\ 1 + min(M(i, j-1), min(M(i-1, j), M(i-1, j-1))), \\ \qquad \text{če } s1[i-1] \neq s2[j-1] \end{cases} \tag{2.2}$$

2.3.1 Primer uporabe

Avtorja članka [1] sta razvila svojo zgoščevalno funkcijo za potrebe primerjanja podobnosti med dvema datotekama. Ideja je, da algoritem datoteki, ki sta potencialno dolžine več megabajtov, skrči na neko manjšo velikost (npr. nekaj kilobajtov) in na koncu nad tema krajšima reprezentacijama požene iskanje Levenshteinove razdalje. Dejansko gre torej za aproksimacijo razlike oz. podobnosti med datotekama, vendar se izkaže, da je ta zadnji korak (torej

dejansko iskanje podobnosti) kljub vsemu ozko grlo celotnega procesa, tako da je čimbolj optimalna implementacija algoritma za iskanje Levenshteinove razdalje ključnega pomena.

Poglavje 3

Vzporedni algoritmi

Osnovna struktura vzporednega algoritma za oba problema je enaka kot njuna rešitev iz prejšnjega poglavja - še vedno uporabljamo dvodimenzionalno tabelo za hranjenje vmesnih vrednosti, torej je prostorska zahtevnost enaka - $O(n * m)$. Razlika je samo v tem, da vrednosti, ki niso medsebojno odvisne, lahko računamo vzporedno z uporabo koncepta večnitnosti (angl. multithreading). Za to obstajata dva glavna pristopa - diagonalni in naprej-nazaj. Poleg tega smo razvili tudi prostorsko optimizirane algoritme, ki ne hranijo celotne tabele v pomnilniku, temveč samo trenutno in prejšnjo vrstico.

Vsega skupaj smo implementirali 14 različic algoritmov:

- LCS naprej (dinamično programiranje)
- LCS naprej s prostorsko optimizacijo
- LCS nazaj (dinamično programiranje)
- LCS nazaj s prostorsko optimizacijo
- LCS naprej-nazaj (2 niti)
- LCS naprej-nazaj s prostorsko optimizacijo (2 niti)
- LCS diagonalno (poljubno število niti)

- Levenshteinova razdalja naprej (dinamično programiranje)
- Levenshteinova razdalja naprej s prostorsko optimizacijo
- Levenshteinova razdalja nazaj (dinamično programiranje)
- Levenshteinova razdalja nazaj s prostorsko optimizacijo
- Levenshteinova razdalja naprej-nazaj (2 niti)
- Levenshteinova razdalja naprej-nazaj s prostorsko optimizacijo (2 niti)
- Levenshteinova razdalja diagonalno (poljubno število niti)

3.1 Pristop naprej-nazaj

Pristop naprej smo že srečali v poglavjih 2.2 in 2.3, ko smo po tabeli iterirali od leve proti desni, ter od vrha navzdol, torej za $i = 0, 1, ..., n$ in $j = 0, 1, ..., m$, nakar smo prišli do končne rešitve na poziciji (i, j). Brez težav lahko počnemo ravno obratno - iteriramo od $i = n, n-1, ...0$ in $j = m, m-1, ..., 0$ in v tem primeru dobimo rešitev na poziciji $(0, 0)$ - temu rečemo pristop nazaj. Izkaže se, da lahko z uporabo dveh niti poženemo oba pristopa istočasno (3.1).

```
1  int fb_LCS (string str1, string str2, int row, int column) {
2
3      int h = row / 2;
4      struct args a;
5      a.s1 = str1;
6      a.s2 = str2;
7      a.row = h;
8      a.col = column;
9
10     thread t1(topHalf_LCS, ref(a));
11     thread t2(bottomHalf_LCS, ref(a));
12
13     t1.join();
```

```
14      t2.join();
15
16      //merge results to find the actual distance
17      return merge_LCS(h, row, column);
18  }
```

Izvorna koda 3.1: Algoritem LCS naprej-nazaj

Tabelo v abstraktnem smislu prepolovimo na dva dela - ena nit računa zgornjo polovico, druga pa spodnjo 3.2. Na ta način, vsaj teoretično, dosežemo dvakratno pohitritev.

```
1  void topHalf_LCS (args& a) {
2      for(int i = 0; i <= a.row; i++)
3          for(int j = 0; j < a.col; j++)
4              if(i == 0 || j == 0)
5                  arr[i][j] = 0;
6              else if(a.s1[i-1] == a.s2[j-1])
7                  arr[i][j] = 1 + arr[i-1][j-1];
8              else
9                  arr[i][j] = max(arr[i][j-1], arr[i-1][j]);
10 }
11
12 void bottomHalf_LCS (args& a) {
13     int nrows = a.s1.length();
14
15     //in case there's nothing for this thread to do
16     if(nrows == 1)
17         return;
18
19     for(int i = nrows+1; i > a.row; i--)
20         for(int j = a.col; j > 0; j--)
21             if (i == nrows+1 || j == a.col)
22                 arr[i][j] = 0;
23             else if(a.s1[i-1] == a.s2[j-1])
24                 arr[i][j] = 1 + arr[i+1][j+1];
25             else
26                 arr[i][j] = max(arr[i][j+1], arr[i+1][j]);
```

```
27 }
```

Izvorna koda 3.2: Vsaka nit računa svojo polovico tabele

Kadar obe niti zaključita izvajanje, je rezultat funkcije naprej-nazaj dvodimenzionalna tabela, pri kateri sta relevantni samo dve vrstici. Primer za dva naključna niza je prikazan na sliki (3.1):

		a	b	c	d	e	f	
	0	0	0	0	0	0	0	0
x	0	1	2	3	4	5	6	0
y	0	2	2	3	4	5	6	0
z	0	3	3	3	4	5	6	0
x	0	1	1	2	3	4	5	0
y	0	2	2	3	4	5	5	0
z	0	3	3	3	4	5	5	0
	0	0	0	0	0	0	0	0

Slika 3.1: Primer končnega izračuna algoritma naprej-nazaj na naključnih nizih

Da dobimo končno razdaljo, moramo še združiti zadnjo vrstico prve niti z zadnjo vrstico druge niti z združevalni funkcijo, ki pa je časovne zahtevnosti $O(n)$, ker gre dejansko samo za dve enodimenzionalni tabeli, tako da nima bistvenega vpliva na čas izvajanja celotnega algoritma.

Za združevanje pri Levenshteinovi razdalji (3.3) iščemo najmanjšo vsoto istoležnih elementov, pri LCS pa največjo.

```
1 int merge_levenshtein (int h, int row, int column) {
2     int temp = 0;
3     int currentMin = INT_MAX;
4
5     for(int i = 1; i <= column; i++) {
6         temp = arr[h][i-1] + arr[h+1][i];
7
```

```
8           if(temp < currentMin)
9               currentMin = temp;
10      }
11
12      return currentMin;
13  }
```

Izvorna koda 3.3: Združevalna funkcija za Levenshteinovo razdaljo

3.2 Prostorska optimizacija

Pri delu z ogromno količino vhodnih podatkov se lahko zgodi da velikost tabele presega kapaciteto pomnilnika. Že za 2 niza dolžine 20000 se izkaže, da če želimo hraniti celotno dvodimenzionalno tabelo v pomnilniku, mora biti velikost le-tega približno 16GB, če predpostavimo da spremenljivke podatkovnega tipa int zasedejo 4 bajte na danem sistemu. Pri naših meritvah, ki so podrobneje opisane v poglavju 4, smo se temu problemu izognili z uporabo spremenljivk podatkovnega tipa unsigned short int, ki zasedajo samo 2 bajta v pomnilniku. Pri tem smo morali biti pozorni, da ne pride do preliva (angl. overflow), saj lahko z dvema bajtoma hranimo maksimalno vrednost 65535, vendar ker vemo, da je tako za LCS kot za Levenshteinovo razdaljo zgornja meja rezultata manjša ali enaka dolžini daljšega izmed vhodnih nizov, je ta pristop vseeno smiselen za vhodne nize krajše od dolžine 65536 (čeprav nizi te dolžine kljub vsemu presegajo kapaciteto pomnilnika na povprečnem osebnem računalniku, na katerem smo poganjali teste).

Vseeno pa je smiselno najti splošnejšo rešitev. Izkaže se, da pri pristopu naprej-nazaj, oziroma tudi brez uporabe večnitnosti pri pristopih naprej in nazaj, lahko bistveno zmanjšamo porabo prostora tako, da v pomnilniku hranimo samo vrstico, ki jo trenutno računamo in prejšnjo vrstico, ostale so nepotrebne in jih lahko zavržemo (3.4). Kot je razvidno iz rezultatov v 4. poglavju, smo s tem dosegli dodatno pohitritev, kljub temu, da mora program v vsaki iteraciji prepisovati vrednosti med tabelama. Glavna slabost tega pristopa pa je, da ker ne hranimo vmesnih izračunov v pomnilniku, na

koncu tudi ne moremo rekonstruirati poti po kateri smo prišli do rezultata. Drugače povedano, algoritem vrne samo razdaljo med nizoma, ne moremo pa ugotoviti kako dejansko pretvoriti prvi niz v drugega (kar sicer ni bilo relevantno pri naši analizi, je pa vseeno pomemben koncept pri marsikaterih praktičnih aplikacijah tovrstnih algoritmov). Vsekakor pa je ta način zelo uporaben za ogromne vhodne podatke, kadar hranjenje dvodimenzionalne tabele v pomnilniku ne pride v poštev.

```
1 int backward_levenshtein_space_optimization (string str1,
    string str2, int row, int column) {
2     int temp1[column];
3     int current[column];
4 
5     //initialize zeros
6     for (int i = 0; i < column; i++) {
7         temp1[i] = 0;
8         current[i] = 0;
9     }
10 
11     for(int i = row-1; i >= 0; i--) {
12         for(int j = column-1; j >= 0; j--) {
13             if (i == row-1)
14                 current[j] = column-j-1;
15             else if (j == column-1)
16                 current[j] = row-i-1;
17             else if(str1[i] == str2[j])
18                 current[j] = temp1[j+1];
19             else
20                 current[j] = 1 + min(current[j+1], min(temp1[
    j], temp1[j+1]));
21         }
22 
23         //rewrite data
24         for(int j = 0; j < column; j++)
25             temp1[j] = current[j];
26 
27     }
```

```
28
29      return current[0];
30  }
```

Izvorna koda 3.4: Prostorska optimizacija

3.3 Diagonalni pristop

Pomanjkljivost pristopa naprej-nazaj je, da dejansko uporablja samo dve niti, oziroma ga je težko dodatno paralelizirati, ker se morajo niti v tem primeru med seboj čakati in ne dosežemo maksimalne optimalnosti, oziroma se čas morda celo poslabša zaradi prevelikega overheada.

Zato če želimo uporabiti več kot dve niti, raje koristimo diagonalni pristop, ki nam teoretično omogoča hkratno uporabo do k niti (k = dolžina diagonale = dolžina krajšega izmed dveh nizov). Ta pristop deluje, ker se izkaže, da so elementi na diagonalah vedno medsebojno neodvisni, algoritem namreč dostopa samo do podatkov v celicah (i, j-1), (i-1, j) in (i-1, j-1), paziti moramo samo, da se diagonale računajo v pravem vrstnem redu, torej da začnemo s celico (0, 0) in končamo s celico (n, m). To najlažje dosežemo tako, da z zunanjo zanko iteriramo po diagonalah (teh je vedno n+m-1) in nato v vsaki iteraciji vzporedno računamo vrednosti vseh celic na posamezni diagonali.

Poglavje 4

Rezultati

V tem poglavju so prikazane meritve časov izvajanja za posamezne algoritme in medsebojna primerjava le-teh. Glede na to, da je večnitenje zelo odvisno od same arhitekture sistema na katerem izvajamo programsko kodo, pa tudi od raznih zunanjih dejavnikov (npr. časovno razvrščanje operacijskega sistema), je predvsem pri tabelah manjših velikosti smiselno, da posamezno funkcijo kličemo večkrat in kot končni rezultat vzamemo povprečje vseh meritev.

4.1 Generiranje podatkov

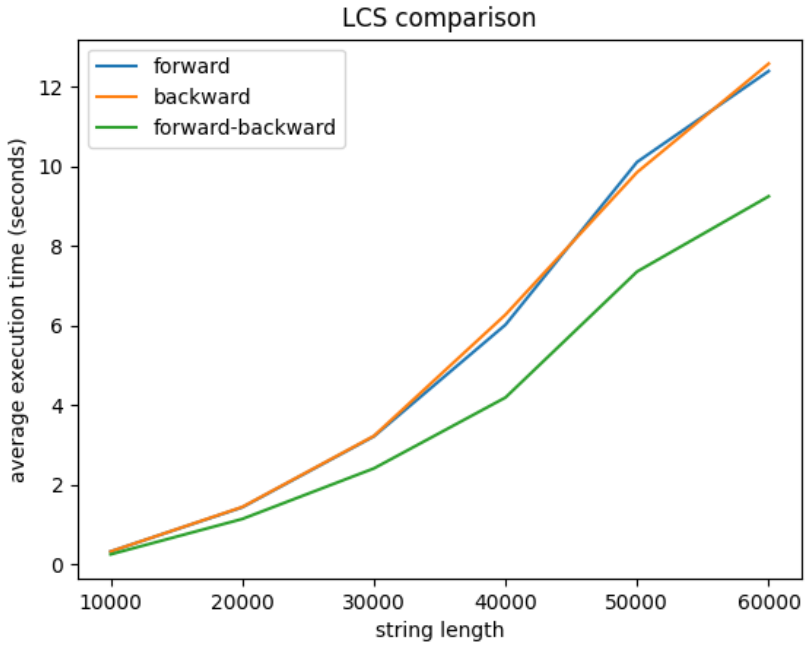
Vhodni podatki so enakega tipa tako za LCS kot za Levenshteinovo razdaljo: teoretično gre za poljubno število znakovnih nizov, vendar smo se za potrebe tega diplomskega dela omejili na delo z točno dvema nizoma, ki vsebujeta znake iz angleške abecede. Na same algoritme ne bi imelo nobenega vpliva, če bi uporabili celotno ASCII tabelo ali katerikoli drug nabor znakov, edina razlika bi bila v tem, da bi bile izračunane razdalje v povprečju krajše zaradi manjše teoretične verjetnosti da se posamezen znak pojavi v obeh nizih. Posledično bi se izvedlo manj klicev funkcij min oz. max, vendar je kar se tiče časovne zahtevnosti ta pohitritev zanemarljiva.

4.2 Delovno okolje

VSCode gcc -O3 [4] AMD 16GB JSON

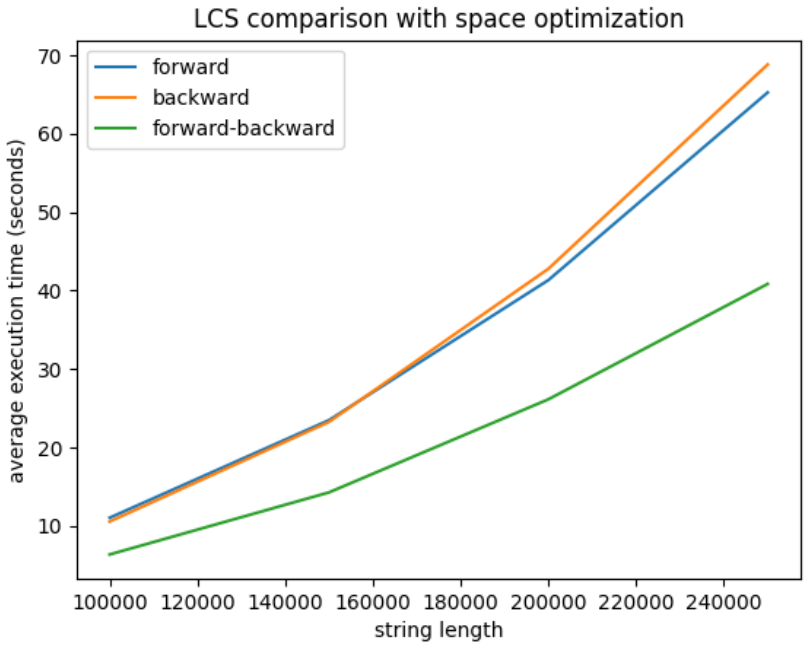
4.3 Primerjava sekvenčnih algoritmov

4.4 Primerjava vzporednih algoritmov

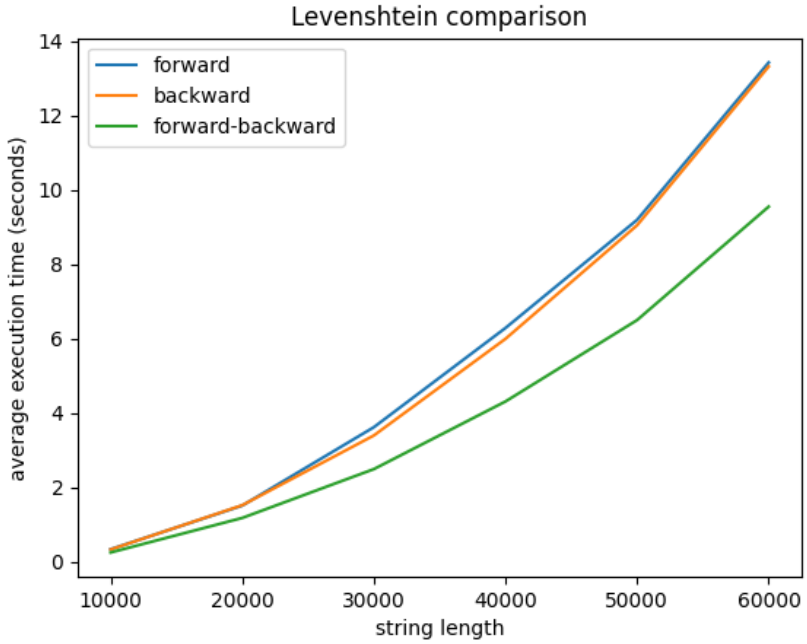


Slika 4.1: Primerjava LCS algoritmov

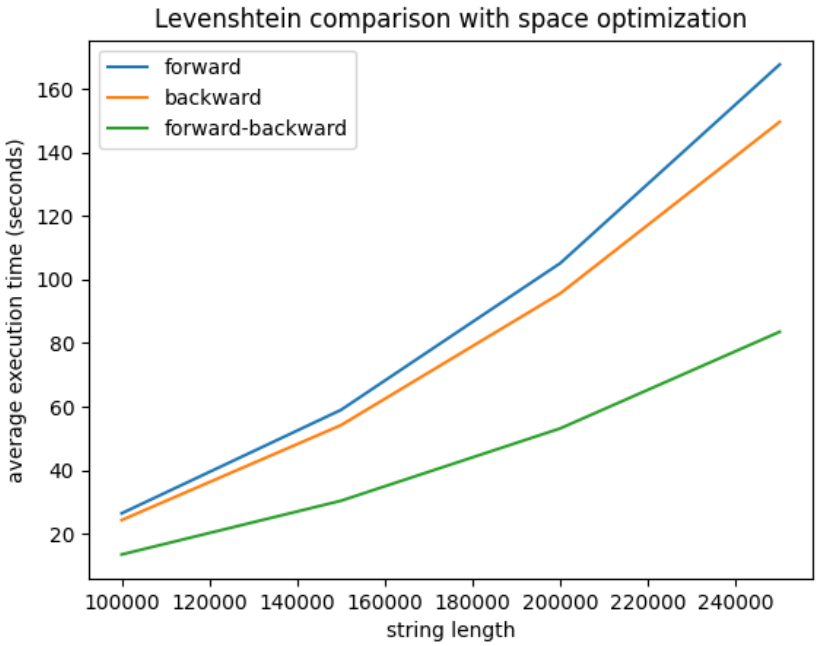
4.5 Celotna primerjava



Slika 4.2: Primerjava LCS algoritmov s prostorsko optimizacijo



Slika 4.3: Primerjava algoritmov za iskanje Levenshteinove razdalje



Slika 4.4: Primerjava algoritmov za iskanje Levenshteinove razdalje s prostorsko optimizacijo

Poglavje 5

Zaključki

5.1 Sklepi

5.2 Nadalnji razvoj

Z vzporednimi algoritmi dosegli občutno pohitritev že na običajnih osebnih računalnikih. V nadalnje bi bilo zanimivo preveriti ali je čas izvajanja morda še krajši na zmogljivejših sistemih z ogromno količino jeder, oziroma ali je mogoče algoritme še dodatno prilagoditi za tovrstne sisteme.

Literatura

[1] Peter Coates in Frank Breitinger. "Identifying document similarity using a fast estimation of the Levenshtein Distance based on compression and signatures". V: 2022. DOI: 10.48550/arXiv.2307.11496.

[2] Thomas H Cormen in sod. *Introduction to Algorithms*. The MIT Press, 2022.

[3] Lionel Fontan in sod. "Using Phonologically Weighted Levenshtein Distances for the Prediction of Microscopic Intelligibility". V: *Proc. Interspeech 2016*. 2016, str. 650–654. DOI: 10.21437/Interspeech.2016-431.

[4] GNU. *Using the GNU Compiler Collection (GCC)*. URL: https://gcc.gnu.org/onlinedocs/gcc-4.6.4/gcc/ (pridobljeno 16. 7. 2024).

[5] Leon Premk. *Algoritmi za izračun razdalje med časovnimi vrstami z dinamičnim prilagajanjem časa*. Diplomska naloga. Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, 2020.