Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Bajić

Vzporedni algoritmi za problem Levenshteinove razdalje in najdaljše skupno podzaporedje

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Jurij Mihelič

Ljubljana, 2024

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Kandidat: Luka Bajić

Naslov: Vzporedni algoritmi za problem Levenshteinove razdalje in najdaljše skupno podzaporedje

Vrsta naloge: Diplomska naloga na visokošolskem programu prve stopnje Računalništvo in informatika

Mentor: izr. prof. dr. Jurij Mihelič

Opis:

Na področju algoritmov je znanih več različnih pristopov za izračun razdalje med dvema nizoma. Med njimi sta tudi izračun Levenshteinove razdalje in najdaljšega skupnega podzaporedja, ki oba temeljita na dinamičnem programiranju. V diplomski nalogi predstavite oba problema in osnovne algoritme za njuno reševanje. Nato za reševanje razvijte vzporedne algoritme in jih eksperimentalno ovrednotite.

Title: Parallel algorithms for the Levenshtein distance and the longest common subsequence problems

Description:

In the field of algorithms, several different approaches for calculating the distance between two strings are known. Among them are the calculation of the Levenshtein distance and the longest common subsequence, both of which are based on dynamic programming. In the thesis, present both problems and the basic algorithms for solving them. Then develop parallel algorithms for solving them and evaluate them experimentally.

Zahvaljujem se mentorju izr. prof. dr. Juriju Miheliču za koristne nasvete in pomoč pri izdelavi diplomske naloge.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
ASCII	American standard code for information exchange	Ameriški standardni nabor za izmenjavo informacij
DDR	double data rate	dvojna hitrost podatkov
JSON	JavaScript object notation	JavaScript objektna notacija
LCS	longest common subsequence	najdaljše skupno podzaporedje

Povzetek

Naslov: Vzporedni algoritmi za problem Levenshteinove razdalje in najdaljše skupno podzaporedje

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 vodenje izvedb izvorne kode (npr. Git). Diplomsko delo opiše različne pristope optimizacije dveh tovrstnih algoritmov: Levenshteinove razdalje in najdaljšega skupnega podzaporedja, z uporabo koncepta večnitnosti in dveh temeljnih pristopov vzporejanja – diagonalni in naprej-nazaj. Glavni cilj je časovna pohitritev algoritmov, ki jo ustrezno izmerimo in primerjamo nove implementacije z obstoječimi. Tako za diagonalni pristop kot za pristop naprej-nazaj razvijemo tudi prostorsko optimizirano metodo, ki je smiselna za uporabo pri velikih količinah vhodnih podatkov.

Ključne besede: vzporedni algoritmi, večnitnost, Levenshteinova razdalja, najdaljše skupno podzaporedje, dinamično programiranje.

Abstract

Title: Parallel algorithms for the Levenshtein distance and the longest common subsequence problems

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 both diagonal and forward-backward approaches we develop an additional space optimization, which is useful for extremely large amounts of input data.

Keywords: parallel algorithms, multithreading, Levenshtein distance, longest common subsequence, dynamic programming.

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. Gre za dva zelo pogosto uporabljena algoritma za potrebe primerjave nizov oziroma računanje razdalje med nizi, ki ju običajno rešujemo z dinamičnim programiranjem (za potrebe tega diplomskega dela smo se omejili na razdalje med točno dvema nizoma, ki je tudi najpogosteje uporabljena v praktičnih aplikacijah). S pomočjo koncepta večnitnosti smo razvili več različic algoritmov, ki dosežejo časovno pohitritev v primerjavi z osnovno implementacijo. Pri tem smo uporabili dva različna načina povzporejanja: diagonalni in naprej-nazaj. Meritve smo izvajali na naključno generiranih nizih, velikosti med 10000 in 400000 znakov. Poleg časovne pohitritve smo iskali tudi možnosti za prostorske optimizacije, torej algoritme, ki zasedejo čimmanjšo količino delovnega pomnilnika, kar je uporabno predvsem za vhodne podatke velikih velikosti, ki jih ni smiselno, oziroma pogosto celo nemogoče v celoti hraniti v delovnem pomnilniku.

1.1 Motivacija

Iskanje podobnosti med nizi se na prvi pogled zdi dokaj enostaven problem, vendar gre v resnici za obsežno in raznoliko skupino pristopov, 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 [4], kjer pogosto naletimo na probleme z ozkim naborom znakov in lahko algoritme temu primerno tudi prilagodimo. Kot drugi primer vzamemo orodja za kontrolo različic [2]. 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).

Iz tovrstnih praktičnih uporab vidimo, da je vsaka optimizacija algoritmov za iskanje razdalje med nizi zelo koristna, zato smo v tem diplomskem delu preučili nekaj tovrstnih pristopov za algoritma za računanje LCS in Levenshteinove razdalje. Omenjeni primeri so sicer zelo odvisni od specifične domene, v našem delu pa smo se osredotočili na iskanje splošnih rešitev, ki pridejo v poštev ne glede na tip vhodnih podatkov, predpostavljamo samo, da naprava, na kateri algoritme izvajamo, omogoča večnitnost.

Kot predlogo smo uporabili konceptualne ideje, razvite v diplomskem delu [8]. Gre za povzporejanje algoritma za računanje razdalje med časovnimi vrstami, ki ga običajno implementiramo z uporabo dinamičnega programiranja na podoben način kot računanje Levenshteinove razdalje in razdalje LCS.

1.2 Struktura

Diplomsko delo je razdeljeno na pet poglavij. Po uvodnem delu sledi opis obeh temeljnih algoritmov, njune praktične uporabnosti in njunih implementacij z uporabo dinamičnega programiranja. Tretje poglavje predstavi dva glavna pristopa povzporejanja: diagonalni in naprej-nazaj. Pri diagonalnem

je predstavljen tudi način optimizacije prenosov iz glavnega pomnilnika v predpomnilnik, ki je ključen za hitrejše delovanje algoritma. Poleg tega za vsako implementacijo opišemo še 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 sklepne ugotovitve in morebitne ideje za nadaljnji razvoj.

Poglavje 2

Metode in problemi

V tem poglavju je predstavljena metoda razvoja algoritmov dinamičnega programiranja, ki služi kot temelj za vse algoritme, ki so predstavljeni v nadaljevanju. Nato sta podrobneje opisana še osnovna algoritma za računanje razdalje LCS in Levenshteinove razdalje.

2.1 Dinamično programiranje

Dinamično programiranje je algoritmična metoda, s katero skušamo optimizirati izvajanje algoritmov, ki imajo takoimenovano optimalno podstrukturo, kar pomeni, da je do rešitve problema možno priti preko optimalne rešitve njegovih podproblemov. To dosežemo s hranjenjem vmesnih rezultatov v tabeli velikosti $n \times m$, če n in m predstavljata dolžini vhodnih nizov. V našem primeru smo problem primerjanja dveh nizov tako razbili na enostavnejše podprobleme primerjanja dveh znakov. Pri osnovni implementaciji se končni rezultat nahaja v celici (n, m), iz preostanka tabele pa lahko tudi rekonstruiramo pot, po kateri smo prišli do rezultata.

2.2 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 [6], vendar ker to nima občutnega vpliva na časovno zahtevnost algoritma, v nadaljevanju predpostavimo, da imajo vse tri operacije enako utež. S tem naborom operacij obstaja teoretično neskončno možnih načinov za transformacijo nekega niza v poljuben drugi niz, Levenshteinova razdalja pa predstavlja ceno najkrajše tovrstne transformacije.

Izračunamo jo tako, da najprej inicializiramo prvo vrstico in stolpec dvodimenzionalne matrike M. Dimenziji matrike sta odvisni od vhodnih podatkov: če imamo dva niza dolžin n in m, bo velikost matrike enaka $(n + 1) \times (m + 1)$. Ker gre tukaj dejansko za primerjavo niza s praznim nizom, je ta korak trivialen in vedno enak: definiramo spremenljivki i, ki iterira od 0 do $n + 1$, in j, ki iterira od 0 do $m + 1$. V prvo vrstico vnesemo vrednosti spremenljivke j, v prvi stolpec pa vrednosti spremenljivke i, kot je prikazano na sliki 2.1 v rdeči barvi.

		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 računanje Levenshteinove razdalje za dva naključna niza.

Nato vrednosti za preostale celice v matriki izračunamo po formuli 2.1, za $i = 1, ..., n, n+1$ in $j = 1, ..., m, m+1$, kjer $s1$ in $s2$ predstavljata vhodna niza dolžin n in m. Končni rezultat dobimo v celici na poziciji $(n+1, m+1)$.

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

Sama implementacija je precej intuitivna (izvorna koda 2.1): z dvojno zanko iteriramo po globalni tabeli *arr* in računamo vrednosti po formuli 2.1 glede na enakost trenutnih dveh znakov v vhodnih nizih $str1$ in $str2$. Pri tem moramo biti pozorni, da pri indeksiranju upoštevamo dodatno vrstico in stolpec, ki smo ju vstavili za inicializacijo začetnih vrednosti (vrstica 8 v kodi 2.1).

```
1 int forward_levenshtein (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)
5                 arr[i][j] = j;
6             else if (j == 0)
7                 arr[i][j] = i;
8             else if (str1[i-1] == str2[j-1])
9                 arr[i][j] = arr[i-1][j-1];
10            else
11                arr[i][j] = 1 + min(arr[i][j-1], min(arr[i
    -1][j], arr[i-1][j-1]));
12        }
13    }
14    return arr[row][column];
15 }
```

Izvorna koda 2.1: Algoritem za izračun Levenshteinove razdalje.

2.2.1 Primer uporabe

Avtorja članka [3] 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.

2.3 LCS

Podzaporedje definiramo kot zaporedje znakov, ki se nahajajo v nizu, v istem vrstnem redu, vendar ne nujno strnjeno. Torej, niz "$ABCD$" vsebuje podzaporedje "ACD", ne vsebuje pa podzaporedja "DCB". Najdaljše skupno podzaporedje predstavlja najdaljšo možno podzaporedje, ki se nahaja v dveh vhodnih nizih. V naših algoritmih iščemo dolžino tega podzaporedja. Algoritem lahko sicer posplošimo na poljubno število nizov [1], vendar smo se v naši implementaciji omejili na točno dva niza.

Pristop reševanja tega problema je podoben kot za Levenshteinovo razdaljo v prejšnjem poglavju: ponovno uporabljamo dvodimenzionalno matriko M, dimenzij $(n+1) \times (m+1)$. Začetne vrednosti za prvo vrstico in prvi stolpec so v tem primeru kar ničle, saj je pri praznem nizu dolžina podzaporedja vedno 0. Preostanek tabele pa izračunamo po formuli 2.2, za $i = 1, ..., n, n+1$ in $j = 1, ..., m, m+1$, kjer $s1$ in $s2$ predstavljata vhodna niza dolžin n in m.

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

Končni rezultat ponovno dobimo v celici $(n+1, m+1)$, kot je prikazano na sliki 2.2.

		a	b	c	d
	0	0	0	0	0
b	0	0	1	1	1
c	0	0	1	2	2
a	0	1	1	2	2

Slika 2.2: Primer izračunane matrike za iskanje LCS za dva naključna niza.

Implementacija algoritma je prav tako podobna Levenshteinovi (koda 2.1), razlika je le v formuli za izračun vrednosti v posameznih celicah. V tem primeru bi lahko celo izpustili inicializacijo začetnih vrednosti (vrstici 4 in 5 v kodi 2.2), če bi bili prepričani, da je prazna tabela že predhodno napolnjena z ničlami.

```
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      return arr[row][column];
11  }
```

Izvorna koda 2.2: Algoritem za izračun razdalje LCS.

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 $O(n \times m)$. 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 20 različic algoritmov:

- LCS naprej
- LCS naprej s prostorsko optimizacijo
- LCS nazaj
- LCS nazaj s prostorsko optimizacijo
- LCS naprej-nazaj (2 niti)
- LCS naprej-nazaj s prostorsko optimizacijo (2 niti)
- LCS diagonalni

- LCS diagonalni s prostorsko optimizacijo
- LCS diagonalni vzporedni (poljubno število niti)
- LCS diagonalni vzporedni s prostorsko optimizacijo (poljubno število niti)
- Levenshteinova razdalja naprej
- Levenshteinova razdalja naprej s prostorsko optimizacijo
- Levenshteinova razdalja nazaj
- Levenshteinova razdalja nazaj s prostorsko optimizacijo
- Levenshteinova razdalja naprej-nazaj (2 niti)
- Levenshteinova razdalja naprej-nazaj s prostorsko optimizacijo (2 niti)
- Levenshteinova razdalja diagonalni
- Levenshteinova razdalja diagonalni s prostorsko optimizacijo
- Levenshteinova razdalja diagonalni vzporedni (poljubno število niti)
- Levenshteinova razdalja diagonalni vzporedni s prostorsko optimizacijo (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 $i = 1, 2, ..., n, n+1$ in $j = 1, 2, ..., m, m+1$, nakar smo prišli do končne rešitve na poziciji $(n+1, m+1)$. 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 (slika 3.1).

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

Slika 3.1: Primer izračunane matrike za Levenshteinovo razdaljo s pristopom nazaj.

Izkaže se, da lahko z uporabo dveh niti poženemo oba pristopa istočasno (izvorna koda 3.1). Najprej ustvarimo spremenljivko podatkovnega tipa *struct args* (vrstice 3–7 v kodi 3.1), s katero podamo vse potrebne argumente funkcijama, ki ju prva in druga nit nato vzporedno izvedeta (vrstici 9 in 10 v kodi 3.1). Najpomembnejši argument je h, ki predstavlja indeks srednje vrstice tabele. Nastavimo ga lahko sicer na poljubno vrednost, vendar ker je najbolj smiselno, da obe niti izvedeta približno enako število izračunov (v primeru lihega števila vrstic pride do razlike za eno vrstico), ga definiramo kot $row/2$.

```
1  int fb_LCS (string str1, string str2, int row, int column) {
2      int h = row/2;
3      struct args a;
4      a.s1 = str1;
5      a.s2 = str2;
6      a.row = h;
7      a.col = column;
8
9      thread t1(topHalf_LCS, ref(a));
10     thread t2(bottomHalf_LCS, ref(a));
11
12     t1.join();
13     t2.join();
14
15     return merge_LCS(h, row, column);
```

```
16 }
```

Izvorna koda 3.1: Algoritem LCS naprej-nazaj.

V prejšnjem koraku smo tabelo v logičnem smislu prepolovili na dva dela: ena nit računa zgornjo polovico, druga pa spodnjo (izvorna koda 3.2). Na ta način, vsaj teoretično, dosežemo dvakratno pohitritev na dovolj velikih vhodnih podatkih.

```
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     if(nrows == 1)
15         return;
16
17     for(int i = nrows+1; i > a.row; i--)
18         for(int j = a.col; j > 0; j--)
19             if (i == nrows+1 || j == a.col)
20                 arr[i][j] = 0;
21             else if(a.s1[i-1] == a.s2[j-1])
22                 arr[i][j] = 1 + arr[i+1][j+1];
23             else
24                 arr[i][j] = max(arr[i][j+1], arr[i+1][j]);
25 }
```

Izvorna koda 3.2: Funkciji za računanje zgornje in spodnje polovice tabele za LCS.

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 3.2.

		a	b	b	d	c	c	
	0	0	0	0	0	0	0	/
b	0	0	1	1	1	1	1	/
c	0	0	1	1	1	2	2	/
a	0	1	1	1	1	2	2	/
a	/	3	2	2	2	1	1	0
d	/	2	2	2	2	1	1	0
c	/	1	1	1	1	1	1	0
	/	0	0	0	0	0	0	0

Slika 3.2: Primer končnega izračuna algoritma LCS naprej-nazaj na naključnih nizih.

Da dobimo dejansko razdaljo, moramo še združiti zadnjo vrstico prve niti z zadnjo vrstico druge niti z združevalno 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 iteriramo po dveh stičnih vrsticah in računamo vsoto parov po diagonalah (torej $M(h,0)+M(h+1,1)$, $M(h,1)+M(h+1,2)$, in tako dalje do indeksa m). Pri Levenshteinovi razdalji iščemo najmanjšo vsoto izmed teh naračunanih elementov (vrstice 1–12 v kodi 3.3), pri LCS pa največjo (vrstice 14–25 v kodi 3.3).

```
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      return currentMin;
12  }
13
14  int merge_LCS (int h, int row, int column) {
15      int temp = 0;
16      int currentMax = 0;
17
18      for(int i = 1; i <= column; i++) {
19          temp = arr[h][i-1] + arr[h+1][i];
20
21          if(temp > currentMax)
22              currentMax = temp;
23      }
24      return currentMax;
25  }
```

Izvorna koda 3.3: Združevalni funkciji za Levenshteinovo razdaljo in LCS.

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 50000 se izkaže, da če želimo hraniti celotno dvodimenzionalno tabelo v pomnilniku, mora biti njegova velikost približno 10 GB, 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 vsekakor smiselen za vhodne nize krajše od dolžine 65535.

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. Ta postopek implementiramo na sledeč način (izvorna koda 3.4): namesto dvodimenzionalne tabele *arr* iz prejšnjih implementacij, tukaj uporabljamo dve enodimenzionalni tabeli (*temp*1 in *current*), katerih dolžina je enaka številu stolpcev (torej dolžini drugega niza+1). Zanke in pogoji so enaki kot v osnovni implemetaciji, razlika je le, da rezultate tukaj shranjujemo v enodimenzionalno tabelo namesto v dvodimenzionalno. Kadar pridemo do konca vrstice, moramo te rezultate še prepisati v začasno tabelo $temp1P$, da lahko z njimi računamo v naslednji iteraciji. Prepis vrednosti najlažje implementiramo z uporabo dveh kazalcev $temp1P$ in $currentP$ ter spremenljivke c, ki poskrbi za ustrezni prepis v vrsticah 24–31.

```
1  int backward_levenshtein_space_optimization (string str1,
       string str2, int row, int column) {
2       temp1 = new int[column];
3       current = new int[column];
4
5       int* temp1P = temp1;
6       int* currentP = current;
7       int c = 0;
8       int currentSolution = 0;
9
10      for(int i = row-1; i >= 0; i--) {
11          for(int j = column-1; j >= 0; j--) {
12              if (i == row-1)
13                  currentP[j] = column-j-1;
14              else if (j == column-1)
15                  currentP[j] = row-i-1;
16              else if(str1[i] == str2[j])
17                  currentP[j] = temp1P[j+1];
```

```
18              else
19                  currentP[j] = 1 + min(currentP[j+1], min(
    temp1P[j], temp1P[j+1]));
20          }
21          currentSolution = currentP[0];
22
23          c = (c+1) % 2;
24          if(c == 0) {
25              temp1P = temp1;
26              currentP = current;
27          }
28          else {
29              temp1P = current;
30              currentP = temp1;
31          }
32      }
33
34      return currentSolution;
35  }
```

Izvorna koda 3.4: Prostorska optimizacija za računanje Levenshteinove razdalje s pristopom nazaj.

Kot je razvidno iz rezultatov v 4. poglavju, smo s tem dosegli dodatno pohitritev, ker smo z uporabo kazalcev implementirali prepis vrednosti s konstantno in praktično zanemarljivo časovno zahtevnostjo. Glavna slabost tega pristopa pa je, da glede na to, da ne hranimo vmesnih izračunov v pomnilniku, na koncu tudi ne moremo rekonstruirati poti, po kateri smo prišli do rezultata. Drugače povedano, algoritem vrne samo vrednost razdalje 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 velike vhodne podatke, kadar hranjenje dvodimenzionalne tabele v pomnilniku ne pride v poštev.

3.3 Diagonalni pristop

Pomanjkljivost pristopa naprej-nazaj je, da dejansko uporablja samo dve niti, oziroma ga je težko dodatno vzporejati, ker se morajo niti v tem primeru med seboj čakati in posledično ne dosežemo izboljšanega časa izvajanja.

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.

3.4 Optimizacija pomnilniškega dostopa

Prednost diagonalnega pristopa v primerjavi z naprej-nazaj je, da deluje s poljubnim številom niti namesto s samo dvema, vendar se izkaže, da pride pri tem do nepričakovanega ozkega grla pri dostopih do delovnega pomnilnika. Namreč, ko računamo celice na posamezni diagonali, je potrebno za izračun iz delovnega pomnilnika v predpomnilnik prenesti celice, ki se nahajajo zgoraj, levo in levo diagonalno v relaciji s trenutnim elementom. Večina modernih računalniških arhitektur iz glavnega pomnilnika v predpomnilnik prenese blok določene dolžine, težava pri tem pa je, da ker se premikamo po diagonali, ostale niti potrebujejo v danem trenutku vsebine celic, ki se nahajajo v nekem čisto drugem delu glavnega pomnilnika, zato mora vsaka nit v večini iteracij prenašati svoj blok v predpomnilnik, kar je zelo časovno potratno v primerjavi s samimi izračuni [5].

		x	y	z	w
	1	2	3	4	5
a	6	7	8	9	10
b	11	12	13	14	15
c	16	17	18	19	20
d	21	22	23	24	25

(a) Običajna matrika.

		x	y	z	w
	1				
a	6	2			
b	11	7	3		
c	16	12	8	4	
d	21	17	13	9	5
	22	18	14	10	
	23	19	15		
	24	20			
	25				

(b) Optimizirana matrika.

Slika 3.3: Primer preoblikovanja matrike za optimizacijo pomnilniških dostopov za dva niza dolžine 4.

Temu problemu se izognemo tako, da matriko zarotiramo v desno za 45 stopinj in s tem dosežemo, da je vsaka diagonala iz osnovne matrike zdaj predstavljena kot vrstica v novi matriki, kot je prikazano na sliki 3.3 (prikazane niso izračunane vrednosti, temveč samo intuitivne oznake posameznih celic, da jasno vidimo, kam se posamezna celica preslika). S tem smo zmanjšali število prenosov v predpomnilnik, ker niti zdaj izvajajo operacije nad elementi tabele, ki so si medsebojno bližji (vedno se nahajajo le v prejšnji ali predprejšnji vrstici, torej na indeksih $i-1$ ali $i-2$).

Hitro opazimo, da se je število vrstic podvojilo, oziroma natančneje: novo število vrstic je enako $n+m-1$. Če sta vhodna niza enako dolga (torej $n = m$), se dejanska prostorska zahtevnost ne poslabša, saj lahko na enostaven način hranimo enako število celic. Potrebno je le, da v programski kodi pri inicializaciji tabele upoštevamo nove dolžine vrstic glede na indeks vrstice. To dosežemo s programsko kodo 3.5:

```
1 int newRow = row + column - 1;
```

```
2
3 arrM = new unsigned short int*[newRow];
4
5 for (int i = 0; i < newRow; i++)
6     if(i < row)
7         arrM[i] = new unsigned short int[i+1];
8     else
9         arrM[i] = new unsigned short int[row + column - i -
    1];
```

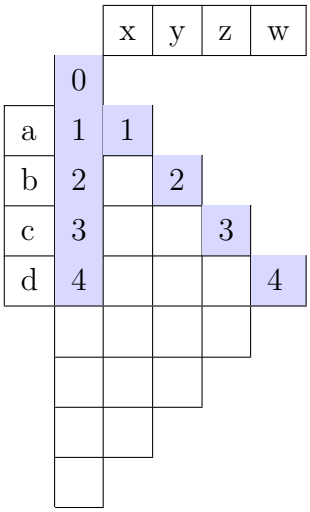
Izvorna koda 3.5: Inicializacija optimizirane tabele.

Ker smo spremenili strukturo tabele, moramo temu primerno prilagoditi tudi programsko kodo. Namesto da iteriramo po diagonalah, zdaj iteriramo po vrsticah, vendar moramo upoštevati, da so indeksi soležnih celic, ki jih potrebujemo za računanje vrednosti v posamezni celici zdaj drugačni, oziroma celice dejansko niti niso več soležne. Indeksi se razlikujejo tudi glede na to, v katerem delu tabele se trenutno nahajamo, zato je najbolj smiselno kodo razdeliti na štiri dele, ki so podrobneje opisani v nadaljevanju. Za lažje razumevanje, si delovanje posameznega dela ogledamo na konkretnem primeru računanja Levenshteinove razdalje za dva niza dolžine 4 (za LCS je sama struktura programa enaka, razlika je le v pogojih in izračunih vrednosti celic). Najprej grafično prikažemo, katere celice posamezen del algoritma računa in nato podamo še formulo za računanje vrednosti, ki upošteva spremenjene indekse zaradi rotacije:

- Inicializacija prve vrstice in stolpca

  Prvi korak je še vedno precej intuitiven: stolpec ostane enak kot pri osnovni implementaciji, vrstica pa zdaj postane diagonala, kot je prikazano na sliki 3.4. Vrednosti računamo po formuli 3.1 za stolpec in 3.2 za vrstico, kjer n predstavlja dolžino prvega niza, m pa dolžino drugega niza.

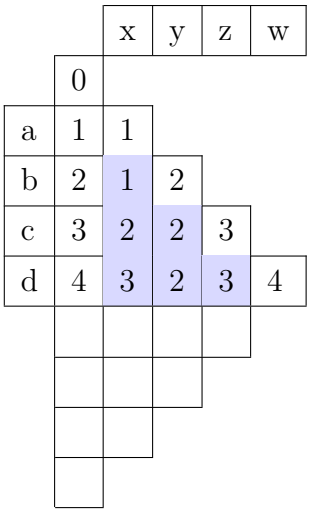
$$M(i, 0) = i, \quad \text{za } i = 0, 1, ..., n, n + 1 \tag{3.1}$$



Slika 3.4: Začetne vrednosti.

$$M(j,j) = j, \quad \text{za } j = 0, 1, ..., m, m+1 \tag{3.2}$$

- Zgornji trikotnik



Slika 3.5: Zgornji trikotnik.

Drugi korak je računanje vrednosti za celice v zgornjem trikotniku, ki je prikazan na sliki 3.5.

Za lažjo vizualizacijo sprememb indeksov si pomagamo s sliko 3.3: kot primer vzamemo celico št. 7 in vidimo, da je v običajni matriki njena diagonalno sosednja celica št. 1 (torej indeks $(i-1, j-1)$), v optimizirani matriki pa se celica št. 1 nahaja na indeksu $(i-2, j-1)$. Po istem principu poiščemo nove indekse še za levo sosednjo celico in zgoraj sosednjo celico, ter na ta način pridemo do formule 3.3, ki jo uporabimo za računanje vrednosti v celotnem zgornjem trikotniku, torej za $i = 2, ..., n+1$ in $j = 1, ..., i$.

$$M(i,j) = \begin{cases} M(i-2, j-1), \\ \qquad \text{če } s1[i-j-1] = s2[j-1] \\ 1 + \min(M(i-2, j-1), M(i-1, j-1), M(i-1, j)), \\ \qquad \text{če } s1[i-j-1] \neq s2[j-1] \end{cases} \tag{3.3}$$

Tako kot za indekse celic, moramo biti pozorni tudi na spremembo indeksov znakov v nizih $s1$ in $s2$. Če ponovno vzamemo za primer celico št. 7 na sliki 3.3, vidimo da v osnovni matriki ta celica predstavlja primerjavo med znakoma "a"in "x", v optimizirani pa med "b"in "x", kar je seveda nepravilno, zato moramo indekse za pogoj v formuli 3.3 ustrezno prilagoditi: namesto da iz niza $s1$ vzamemo znak na indeksu $(i-1)$, zdaj vzamemo znak na indeksu $(i-j-1)$, indeks za $s2$ pa ostane enak kot v osnovni formuli.

- Vmesne vrednosti in zamik

  Ta korak je sestavljen iz dveh delov, pri čemer pride prvi del v poštev samo kadar $n \neq m$. V takem primeru uporabimo enako formulo kot v prejšnjem razdelku (torej 3.3), le da tukaj iteriramo po $i = m, m+1, ..., n$ in $j = 1, 2, ..., m$.

  Nato moramo še poskrbeti, da so celice pravilno zamaknjene v levo, kot je prikazano na sliki 3.6, kar dosežemo s formulo 3.4, če n in m

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

Slika 3.6: Vmesne vrednosti in zamik.

predstavljata dolžino nizov $s1$ oziroma $s2$ in $j = 0, 1, ..., m$.

$$M(n+1, j) = \begin{cases} M(n-1, j), \\ \qquad \text{če } s1[n-j-1] = s2[j] \\ 1 + \min(M(n-1, j), M(n, j+1), M(n, j)), \\ \qquad \text{če } s1[n-j-1] \neq s2[j] \end{cases} \tag{3.4}$$

- Spodnji trikotnik

  Kot zadnji korak moramo izračunati še vrednosti v spodnjem trikotniku, ki je prikazan na sliki 3.7, nakar dobimo končni rezultat v celici na indeksu $(n + m - 1, 0)$. Indeksi v tem delu tabele so ponovno drugačni, zato moramo vrednosti računati po novi enačbi 3.5, za $i = n + 1, n + 2, ..., n + m - 1$ in $j = 0, 1, ..., n + m - i - 1$.

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

Slika 3.7: Spodnji trikotnik.

$$M(i,j) = \begin{cases} M(i-2, j+1), \\ \qquad \text{če } s1[n-j-1] = s2[j+i-n+1] \\ 1 + \min(M(i-2, j+1), M(i-1, j+1), M(i-1, j)), \\ \qquad \text{če } s1[n-j-1] \neq s2[j+i-n+1] \end{cases} \tag{3.5}$$

3.5 Vzporejanje diagonalnega algoritma

V prejšnjem poglavju smo razvili algoritem z optimiziranim pomnilniškim dostopom, v tem poglavju pa si ogledamo proces vzporejanja tega algoritma. Ključni koncept, ki ga za to potrebujemo je sinhronizacija niti.

Kot vemo, je glavna ideja večnitnosti to, da istočasno poganjamo več niti, ki izvajajo vsaka svoje operacije, običajno nad nekimi skupnimi podatki (v našem primeru je to dvodimenzionalna tabela). Če imamo n niti, lahko v idealnem primeru torej dosežemo n–kratno pohitritev. Do težav običajno pride zaradi medsebojne odvisnosti podatkov, kar pomeni, da določena nit v nekem trenutku ne more nadaljevati izvajanja, ker mora čakati rezultat, ki

ga računa ena izmed ostalih niti. V takšnih situacijah obvezno potrebujemo sinhronizacijo, za kar obstaja več možnih implementacij. V našem primeru smo koristili oviro (angl. barrier), ki je že vgrajena v standardno knjižnico programskega jezika C++ od verzije 20 dalje. V programski kodi 3.6 se poslužujemo ovire vsakič, ko kličemo funkcijo arrive_and_wait(). Izkaže se, da moramo sinhronizacijo izvesti vsakič, ko pridemo z zanko do konca posamezne vrstice v tabeli.

Če si podrobneje ogledamo kodo 3.6, vidimo, da je razdeljena na štiri dele, ki smo jih opisali v prejšnjem poglavju:

- Inicializacija prve vrstice in stolpca: vrstice 9–12
- Zgornji trikotnik: vrstice 14–22
- Vmesne vrednosti in zamik: vrstice 24–40
- Spodnji trikotnik: vrstice 42–50

Za potrebe vzporejanja smo morali implementirati prej omenjene ovire, poleg tega pa je potrebno še določiti, kako si bodo niti delile posamezno vrstico. Tudi za ta korak obstajajo različni pristopi, za naše algoritme pa smo vzeli kar najbolj intuitivno delitev: dolžino vrstice delimo s številom niti in rezultat zaokrožimo navzgor, da zajamemo še morebiten ostanek, če se deljenje ne izide (vrstica 6 v kodi 3.6). S tem dobimo število elementov (oziroma celic), ki naj jih posamezna nit izračuna, določiti moramo samo še začetno točko, kar najlažje dosežemo tako, da vsaki niti določimo unikaten id, od 0 do $n\_thr - 1$. Začetno točko nato dobimo z enostavnim množenjem id-ja in števila elementov (vrstica 7 v kodi 3.6).

```
1 int n_thr = 8;
2 barrier bar { n_thr };
3
4 void diagonal_levenshtein_memory_optimization_thread (diag& a
    ) {
```

```
5     int newRow = a.row + a.col - 1;
6     int chunkSize = ceil((double)(a.col)/a.n_total);
7     int startIndex = a.id * chunkSize;
8
9     for (int i = startIndex; i < a.row && i < startIndex+
    chunkSize; i++)
10        arrM[i][0] = i;
11    for (int j = startIndex+1; j < a.col && j <= startIndex+
    chunkSize; j++)
12        arrM[j][j] = j;
13
14    for(int i = 2; i < a.col; i++) {
15        bar.arrive_and_wait();
16        for(int j = startIndex+1; j < i && j <= startIndex+
    chunkSize; j++) {
17            if(a.s1[i-j-1] == a.s2[j-1])
18                arrM[i][j] = arrM[i-2][j-1];
19            else
20                arrM[i][j] = 1 + min(arrM[i-2][j-1], min(arrM
    [i-1][j-1], arrM[i-1][j]));
21        }
22    }
23
24    for(int i = a.col; i < a.row; i++) {
25        bar.arrive_and_wait();
26        for(int j = startIndex+1; j < a.col && j <=
    startIndex+chunkSize; j++) {
27            if(a.s1[i-j-1] == a.s2[j-1])
28                arrM[i][j] = arrM[i-2][j-1];
29            else
30                arrM[i][j] = 1 + min(arrM[i-1][j], min(arrM[i
    -1][j-1], arrM[i-2][j-1]));
31        }
32    }
33
34    bar.arrive_and_wait();
35    for(int j = startIndex; j < a.col - 1 && j < startIndex+
    chunkSize; j++) {
```

```
36          if(a.s1[a.row-j-2] == a.s2[j])
37              arrM[a.row][j] = arrM[a.row-2][j];
38          else
39              arrM[a.row][j] = 1 + min(arrM[a.row-2][j], min(
    arrM[a.row-1][j+1], arrM[a.row-1][j]));
40      }
41
42      for(int i = a.row+1; i < newRow; i++) {
43          bar.arrive_and_wait();
44          for(int j = startIndex; j < newRow - i && j <
    startIndex+chunkSize; j++) {
45              if(a.s1[a.row-j-2] == a.s2[j+i-a.row])
46                  arrM[i][j] = arrM[i-2][j+1];
47              else
48                  arrM[i][j] = 1 + min(arrM[i-2][j+1], min(arrM
    [i-1][j+1], arrM[i-1][j]));
49          }
50      }
51 }
52
```

Izvorna koda 3.6: Vzporedni diagonalni algoritem za računanje Levenshteinove razdalje.

3.6 Prostorska optimizacija diagonalnega algoritma

Na tem mestu na kratko predstavimo še prostorsko optimizacijo diagonalnih pristopov. Ideja je konceptualno enaka kot pri algoritmih naprej, nazaj in naprej-nazaj: namesto dvodimenzionalne tabele uporabimo enodimenzionalni tabeli, za hrambo trenutne in prejšnje vrstice, edina razlika je, da moramo pri diagonalnih algoritmih z optimizacijo predpomnilnika hraniti še predprejšnjo vrstico (kar je hitro razvidno iz samega algoritma, saj v njem dostopamo tudi do indeksa $i-2$). Prostorska zahtevnost je v tem primeru torej $3n$, namesto $2n$. Pri vzporednih algoritmih moramo biti dodatno pozorni

tudi na sinhronizacijo niti pred prepisom vrednosti. Postopek je sledeč:

- vzporedni izračun vrstice
- sinhronizacija niti
- prepis vrednosti (z uporabo kazalcev)
- sinhronizacija niti

Za razumevanje praktične implementacije tega postopka si lahko ogledamo krajši izsek programske kode za računanje zgornjega trikotnika pri vzporednem algoritmu za računanje razdalje LCS s prostorsko in pomnilniško optimizacijo (koda 3.7). Izračunane vrednosti shranjujemo v 3 enodimenzionalne tabele $diagCurrent$, $diagTemp1$ in $diagTemp2$, po podobni logiki kot pri prostorski optimizaciji algoritma nazaj (3.4), le da spremenljivka c v tem primeru kroži med vrednostmi 0, 1 in 2, ker tukaj operiramo s tremi tabelami namesto z dvema. Ključnega pomena je, da pred prepisom vrednosti sinhroniziramo niti (vrstica 9 v kodi 3.7), v nasprotnem primeru se lahko zgodi da pride do prepisa preden so vse niti končale s svojimi izračuni in tako dobimo nepopolne oziroma napačne vrednosti. Da ne pride do nepredvidljivih rezultatov, moramo biti tudi pozorni, da prepis izvaja le ena izmed niti (vrstica 10 v kodi 3.7). To si lahko privoščimo, ker je časovna zahtevnost prepisa s kazalci zanemarljivo majhna.

```
1 for(int i = 2; i < a.col; i++) {
2         bar.arrive_and_wait();
3         for(int j = startIndex+1; j < i && j <= startIndex+
    chunkSize; j++)
4             if(a.s1[i-j-1] == a.s2[j-1])
5                 diagCurrentP[j] = 1 + diagTemp1P[j-1];
6             else
7                 diagCurrentP[j] = max(diagTemp2[j-1],
    diagTemp2[j]);
8
```

```
9           bar.arrive_and_wait();
10          if(a.id == 0) {
11              c = (c+1) % 3;
12              if(c == 0) {
13                  diagTemp1P = diagTemp1;
14                  diagTemp2 = diagTemp2;
15                  diagCurrentP = diagCurrent;
16              } else if (c == 1) {
17                  diagTemp1P = diagTemp2;
18                  diagTemp2 = diagCurrent;
19                  diagCurrentP = diagTemp1;
20              } else {
21                  diagTemp1P = diagCurrent;
22                  diagTemp2 = diagTemp1;
23                  diagCurrentP = diagTemp2;
24              }
25          }
26      }
```

Izvorna koda 3.7: Del kode za računanje razdalje LCS s prostorsko in pomnilniško optimizacijo.

Poglavje 4

Eksperimentalno ovrednotenje

V tem poglavju najprej predstavimo postopek generiranja podatkov in na kratko opišemo sistem, na katerem smo izvajali testne meritve, nato pa prikažemo meritve časov izvajanja za posamezne algoritme in njihovo medsebojno primerjavo. Glede na to, da je večnitenje zelo odvisno od same arhitekture sistema na katerem izvajamo programsko kodo, kot tudi od raznih zunanjih dejavnikov (npr. razvrščanje procesov operacijskega sistema), je predvsem pri tabelah manjših velikosti smiselno, da posamezno funkcijo kličemo večkrat in kot končni rezultat vzamemo povprečje vseh meritev.

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 s 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.

4.1 Testno okolje

Vse meritve, ki so predstavljene v naslednjih podpoglavjih, smo izvajali na sistemu s sledečimi specifikacijami:

- operacijski sistem: Windows 11 Home, 64-bitni
- procesor: AMD Ryzen 7 6800H:
  - 3.2 GHz
  - 8 jeder
  - 16 MB L3 predpomnilnika
- delovni pomnilnik: 16 GB DDR5

Za vizualizacijo rezultatov smo uporabili knjižnico Matplotlib v programskem jeziku Python, preostanek programske kode (torej implementacija vseh 20 različic algoritmov in generiranje naključnih podatkov) je spisan v programskem jeziku C++20. Kot urejevalnik smo uporabljali Visual Studio Code, ki ima vgrajen prevajalnik gcc, verzije 13.2.0. Ključnega pomena je, da prevajalniku dodamo optimizacijsko stikalo -O3 [7] v JSON datoteki `tasks.json`. S tem nekoliko podaljšamo čas prevajanja, vendar je čas dejanskega izvajanja programa občutno hitrejši.

Algoritmi pravilno delujejo za vhodne podatke različnih dimenzij (dolžin), vendar smo za potrebe enostavnejših oziroma bolj preglednih grafičnih prikazov rezultatov meritve izvajali samo nad dvema nizoma enake dolžine. Drugače povedano, na x osi na vseh grafih v naslednih podpoglavjih je navedena samo ena dolžina niza, kar pa dejansko pomeni, da smo algoritme izvajali nad dvema nizoma te dolžine. Pri prostorsko optimiziranih algoritmih to pomeni, da delamo z dvema oziroma tremi tabelami dolžine n, pri ostalih pa gre za matriko velikosti $n \times n$.

Za implementacije, ki hranijo celotno tabelo v delovnem pomnilniku, smo meritve izvedli za nize dolžin med 10000 in 60000, s korakom 10000, za

implemetacije s prostorsko optimizacijo pa za nize dolžin med 100000 in 400000, s korakom 50000.

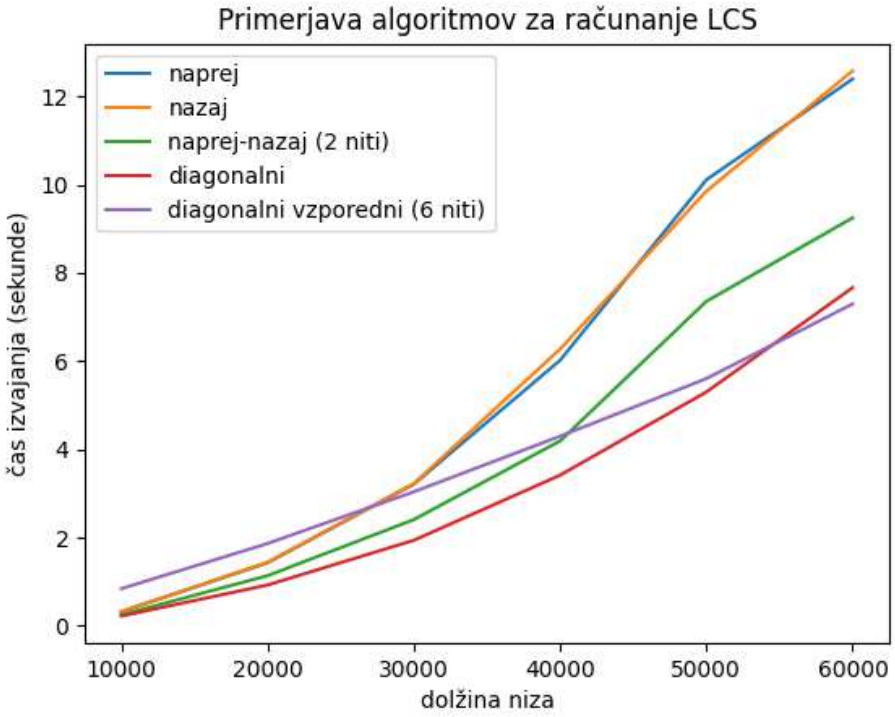
4.2 Primerjava algoritmov za LCS

V tem poglavju predstavimo meritve časov izvajanja algoritmov za računanje LCS. Implementacije z optimzacijo prostora in implementacije brez optimizacije prostora obravnavamo kot ločeni entiteti, zato rezultate meritev prikažemo v ločenih podpoglavjih.

4.2.1 Primerjava algoritmov brez prostorske optimizacije

Najprej si ogledamo čase izvajanja za algoritme brez prostorske optimizacije (slika 4.1). V tej kategoriji imamo 5 algoritmov, od tega sta 2 vzporedna: naprej-nazaj in diagonalni vzporedni (ta na našem sistemu v povprečju dosega najboljše čase pri uporabi 6 niti). Kot smo pričakovali, je čas izvajanja za pristopa naprej in nazaj praktično enak, ker gre dejansko za isti algoritem, le z različnim vrstnim redom računanja celic. Z vzporednim pristopom naprej-nazaj pa dosežemo pohitritev že na meritvi z nizi dolžine 10000 (slika 4.1). Izkaže pa se, da pohitritev ni ravno dvakratna, kot bi teoretično pričakovali pri uporabi dveh niti. Težava je dejansko pri alokaciji pomnilnika, torej pri sami inicializaciji dvodimenzionalne tabele. To je relativno časovno potraten korak, ki ga vsebujejo vsi algoritmi, ki so prikazani na grafu, vendar ga ni mogoče izvesti vzporedno.

Primer: za vhodna niza dolžin 50000 alokacija pomnilnika traja približno 3.5 sekund, celotno izvajanje vzporednega algoritma naprej-nazaj pa traja približno 7 sekund. Enako velja za osnovni algoritem (naprej). Če torej odštejemo čas alokacije od algoritmov naprej in naprej-nazaj, dobimo za vhodna niza dolžin 50000 časa izvajanja približno 6.5 sekund in 3.5 sekund, kar pomeni, da je izvajanje računskih operacij z uporabo dveh niti dejansko



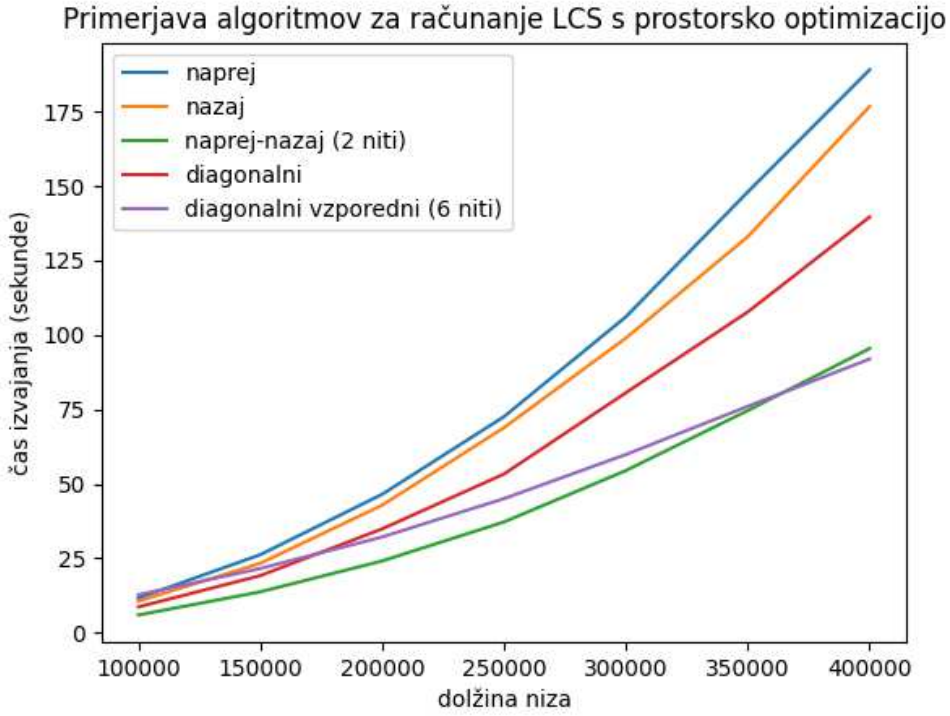
Slika 4.1: Primerjava časov izvajanja LCS algoritmov brez prostorske optimizacije.

res skoraj dvakrat hitrejše. Kot smo že omenili v prejšnjih poglavjih, je čas izvajanja združevalne funkcije praktično zanemarljiv, čas kreiranja in združevanja dveh niti pa je prav tako relativno kratek.

Zanimiva ugotovitev je tudi, da se diagonalni algoritem dobro izkaže že brez paralelizacije in je celo hitrejši od vzporednega algoritma s 6 nitmi za nize do velikosti približno 50000. Sklepamo, da do tega pride zaradi obvezne sinhronizacije niti na koncu vsake vrstice pri vzporednem algoritmu. Pri tem pride do neizogibnega medsebojnega čakanja med nitmi, saj ne smejo nadaljevati izvajanja dokler ni trenutna vrstica do konca izračunana. Na neki točki pa vzporedni algoritem postane hitrejši, ker postane cena sinhronizacije manjša od cene računanja vseh elementov z uporabo samo ene niti.

4.2.2 Primerjava algoritmov s prostorsko optimizacijo

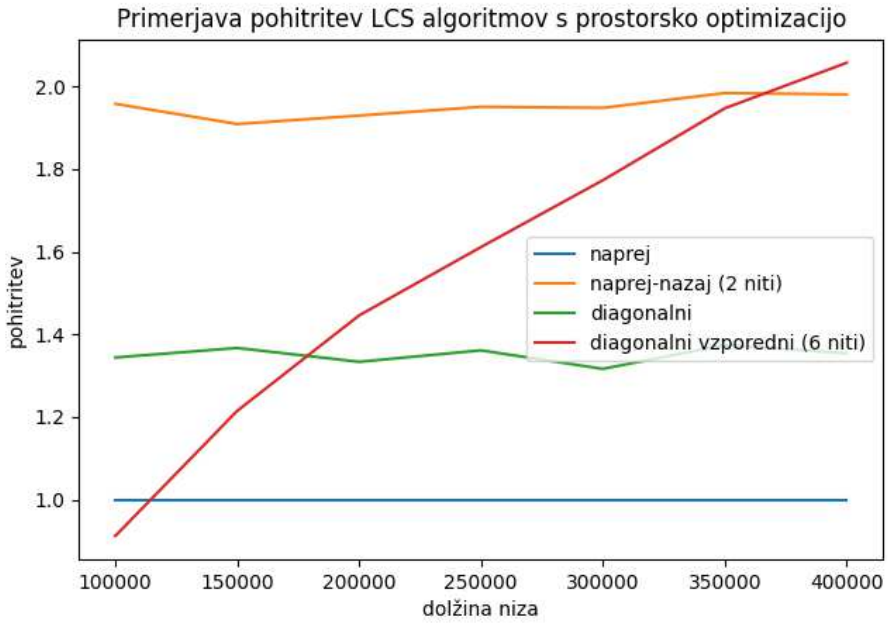
Pri primerjavi prostorsko optimiziranih algoritmov (slika 4.2) ravno tako vidimo, da ne obstaja bistvena razlika med pristopoma naprej in nazaj, pristop naprej-nazaj pa je v tem primeru res skoraj dvakrat hitrejši od osnovnega algoritma, ker tukaj nimamo težav z alokacijo pomnilnika.



Slika 4.2: Primerjava časov izvajanja LCS algoritmov s prostorsko optimizacijo.

Zanimiva ugotovitev je, da se pristop naprej-nazaj izkaže bolje od diagonalnega tudi za precej velike vhodne nize (do približno 350000 x 350000). Tukaj gre dejansko za primerjavo med algoritmom z dvema nitima, ki sta medseboj neodvisni (torej ločeno računata vsaka svojo polovico tabele) in algoritmom s šestimi nitmi, ki se morajo sinhronizirati na koncu vsake vrstice, pri čemer neizogibno pride do medsebojnega čakanja. Vseeno pa lahko pričakujemo, da bo na še večjih vhodnih nizih diagonalni algoritem postal

občutno hitrejši od naprej-nazaj. Iz slike 4.2 to sicer ni intuitivno razvidno, zato si za lažje razumevanje raje ogledamo graf pohitritev (4.3):

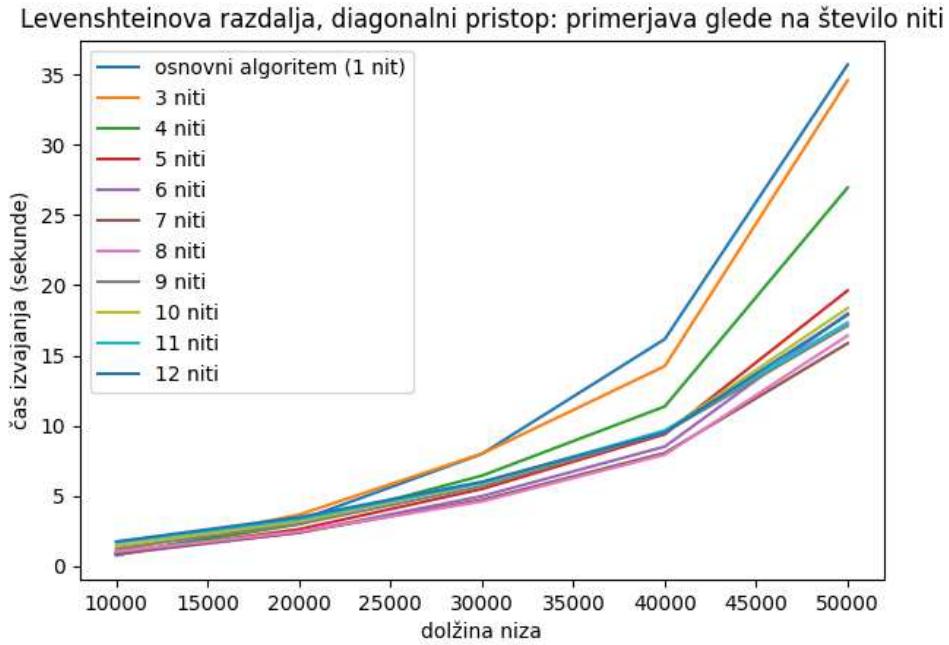


Slika 4.3: Primerjava pohitritev algoritmov za računanje razdalje LCS s prostorsko optimizacijo.

Na x-osi je še vedno dolžina vhodnih nizov, na y-osi pa prikažemo pohitritev. Izračunamo jo tako, da vzamemo algoritem naprej kot osnovo in čase izvajanja osnovnega algoritma delimo s časi izvajanja ostalih algoritmov. S tem dobimo vrednosti pohitritev glede na velikost vhodnih nizov. Tukaj je že na prvi pogled razvidno, da je pohitritev algoritma naprej-nazaj bolj ali manj konstantna (kar je intuitivno smiselno, ker gre za dve medsebojno neodvisni niti in pohitritev v tem primeru ne more biti več kot dvakratna), medtem ko pohitritev diagonalnega vzporednega algoritma raste z velikostjo vhodnih podatkov.

4.3 Primerjava algoritmov za Levenshteinovo razdaljo

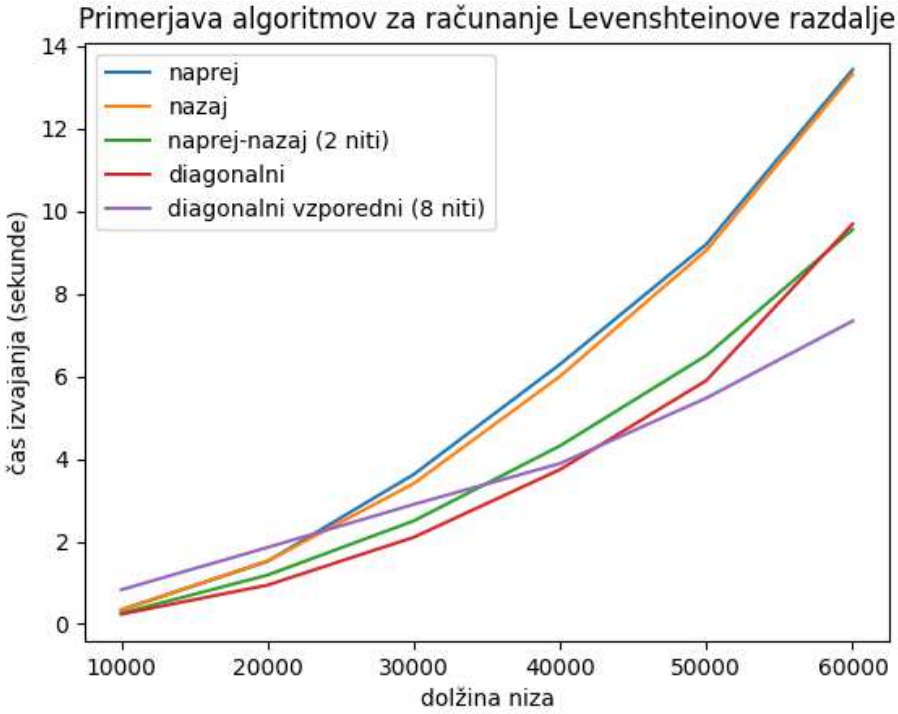
Na tem mestu je smiselno, da si ogledamo še primerjavo časov izvajanja diagonalnega algoritma glede na število niti (slika 4.4). Iz specifikacij sistema vemo, da lahko hkrati vzporedno izvajamo največ 8 niti, vendar vidimo, da je tudi večje število niti (npr. 12) še vedno hitrejše od osnovnega algoritma. Ker pride v tem primeru do dodatnega čakanja med nitmi, seveda ni smiselno, da uporabimo število niti večje od 8, zato smo za meritve za Levenshteinovo razdaljo vedno uporabljali 8 niti, za LCS pa se v povprečju bolje izkaže samo 6 niti.



Slika 4.4: Primerjava časov izvajanja diagonalnih algoritmov glede na število niti.

4.3.1 Primerjava algoritmov brez prostorske optimizacije

Ker sta algoritma za računanje razdalje LCS in Levenshteinove razdalje konceptualno zelo podobna, lahko pričakujemo, da bo to razvidno tudi iz praktičnih meritev. Že na prvi pogled vidimo, da so razmerja med krivuljami res približno enaka (če primerjamo npr. sliko 4.5 s sliko 4.1). Dejanski časi izvajanja so tukaj nekoliko daljši, predvidoma zaradi dodatnega pogoja in gnezdenega klica funkcije min(), ki nista prisotna v algoritmu za računanje razdalje LCS.

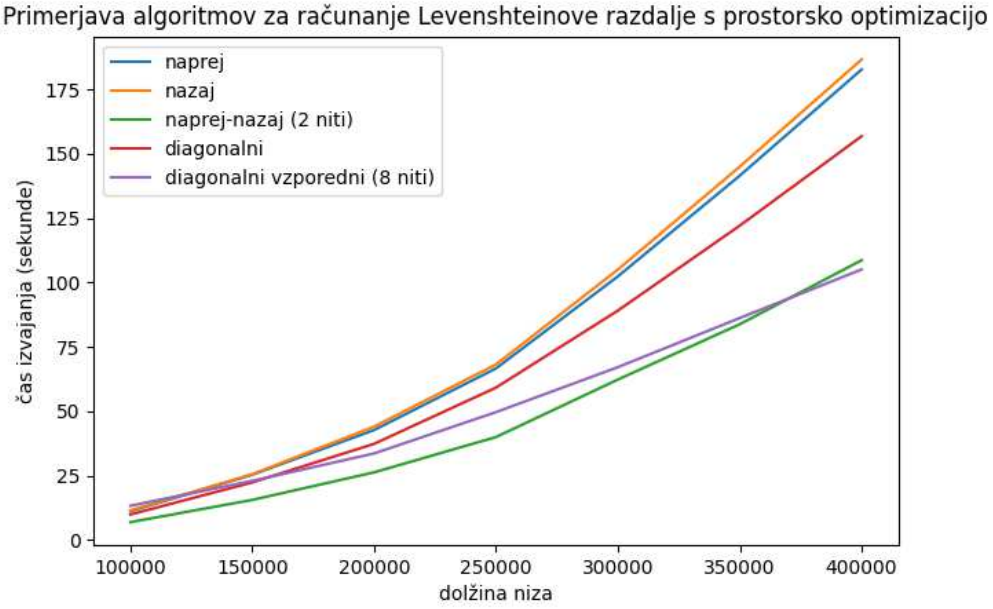


Slika 4.5: Primerjava časov izvajanja algoritmov za računanje Levenshteinove razdalje brez prostorske optimizacije.

Zaradi daljših časov izvajanja se tudi izkaže, da občutimo uporabnost vzporednih algoritmov že na krajših dolžinah vhodnih nizov. Primer: pri

LCS algoritmih (slika 4.1) je osnovni diagonalni algoritem hitrejši od vzporednega do nizov dolžine približno 50000, pri Levenshteinovi razdalji (slika 4.5) pa sta časa izvajanja približno enaka že pri nizih dolžine 40000.

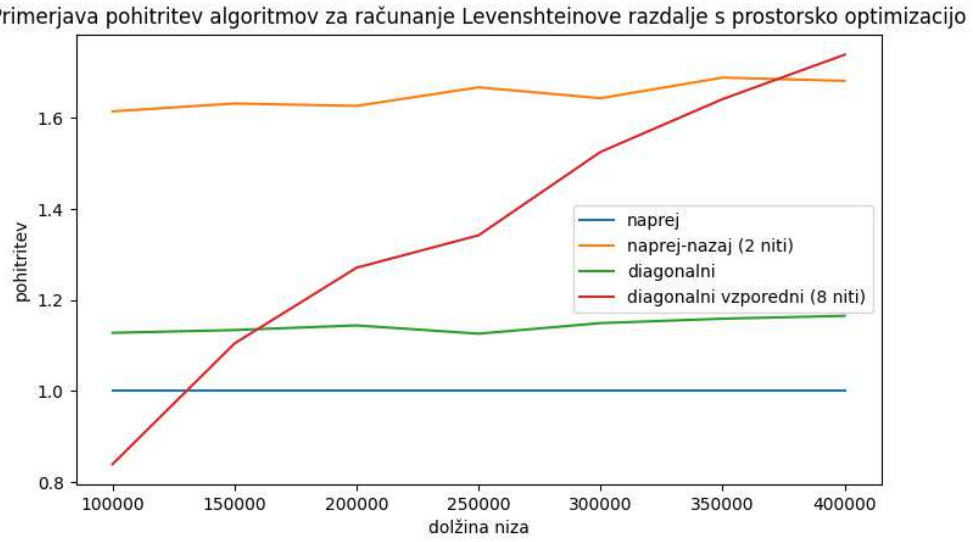
4.3.2 Primerjava algoritmov s prostorsko optimizacijo



Slika 4.6: Primerjava časov izvajanja algoritmov za računanje Levenshteinove razdalje s prostorsko optimizacijo.

Na sliki 4.6 je prikazana primerjava časov izvajanja algoritmov za iskanje Levenshteinove razdalje s prostorsko optimizacijo. Ponovno vidimo, da sta algoritma naprej in nazaj praktično enako hitra, diagonalni brez vzporejanja pa doseže majhno, vendar ne ravno zanemarljivo pohitritev. Najhitrejša algoritma sta po pričakovanjih naprej-nazaj in diagonalni vzporedni. Slednji postane hitrejši na približno isti točki kot pri razdalji LCS, torej približno med dolžinami nizov 350000 in 400000.

Za bolj intuitiven prikaz pohitritev si ogledamo sliko 4.7. Za osnovni algoritem ponovno vzamemo pristop naprej in izračunamo pohitritve ostalih



Slika 4.7: Primerjava pohitritev algoritmov za računanje Levenshteinove razdalje s prostorsko optimizacijo.

algoritmov po istem principu kot pri razdalji LCS (torej z enostavnim deljenjem časov izvajanja). Ponovno vidimo, da je pohitritev pri vzporednem algoritmu naprej-nazaj bolj ali manj konstantna, medtem ko pohitritev diagonalnega vzporednega algoritma narašča z dolžinami vhodnih nizov.

Poglavje 5

Zaključki

V tem poglavju predstavimo sklepne ugotovitve diplomskega dela in predlagamo nekaj možnih idej za nadaljnji razvoj.

5.1 Sklepi

Izkaže se, da obstajajo različni možni načini povzporejanja algoritmov za računanje dolžine LCS in Levenshteinove razdalje, kljub temu, da pri obeh problemih obstaja velika medsebojna odvisnost med izračunanimi rezultati v matriki.

Veliko pohitritev dosežemo že z uporabo dveh niti, s pristopom naprej-nazaj. Izboljšave so vidne že na relativno majhnih vhodnih podatkih. Če nas zanima samo dolžina LCS ali Levenshteinove razdalje, in nas ne zanima rekonstrukcija poti, po kateri smo prišli do rezultata, pa lahko implementiramo še občutno izboljšano porabo prostora v pomnilniku, s tem da se izognemo uporabi dvodimenzionalnih matrik. Prostorsko zahtevnost tako izboljšamo iz $O(n^2)$ na $O(n)$.

Za vzporedno izvajanje več kot dveh niti, se moramo poslužiti diagonalnega pristopa, pri katerem lahko izberemo poljubno število niti glede na omejitve strojne opreme na sistemu, na katerem se izvaja programska koda, ali pa glede na velikost vhodnih podatkov.

5.2 Nadaljnji razvoj

Z vzporednimi algoritmi smo dosegli občutno pohitritev že na običajnih osebnih računalnikih. V nadaljnje 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. Tukaj bi lahko eksperimentirali tudi z različnim številom niti glede na dolžino vrstice pri diagonalnem pristopu, glede na to, da so dolžine vrstic daljše na srednjem delu tabele in krajše na začetku in koncu.

Zanimivo bi bilo tudi ugotoviti, ali je algoritme mogoče preurediti tako, da na nek način upoštevajo dolžine vhodnih nizov, predvsem če pride do situacije, ko je en niz bistveno krajši od drugega. Predvidevamo, da bi bilo v tem primeru smiselno krajši niz postaviti v vrstice, daljši niz pa v stolpce matrike, da bi s tem zmanjšali število sinhronizacij niti pri diagonalnem pristopu.

Literatura

[1] Chinmay Bepery, Sk. Abdullah-Al-Mamun in M. Sohel Rahman. "Computing a longest common subsequence for multiple sequences". V: *2015 2nd International Conference on Electrical Information and Communication Technologies (EICT)*. 2015, str. 118–129. DOI: 10.1109/EICT.2015.7391933.

[2] Gerardo Canfora, Luigi Cerulo in Massimiliano Di Penta. "Identifying Changed Source Code Lines from Version Repositories". V: jun. 2007, str. 14–14. ISBN: 0-7695-2950-X. DOI: 10.1109/MSR.2007.14.

[3] 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.

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

[5] Danijela Efnusheva, Ana Cholakoska in Aristotel Tentov. "A Survey of Different Approaches for Overcoming the Processor - Memory Bottleneck". V: *International Journal of Information Technology and Computer Science* 9 (apr. 2017), str. 151. DOI: 10.5121/ijcsit.2017.9214.

[6] 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.

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

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