

UNIVERZA V LJUBLJANI  
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Luka Bajić

# Inženiring vzporednih algoritmov

DIPLOMSKO DELO

VISOKOŠOLSKI ŠTUDIJSKI PROGRAM  
PRVE STOPNJE  
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR:izr. prof. dr. Jurij Mihelič

Ljubljana, 2024

To delo je ponujeno pod licenco *Creative Commons Priznanje avtorstva-Deljenje pod enakimi pogoji 2.5 Slovenija* (ali novejšo različico). To pomeni, da se tako besedilo, slike, grafi in druge sestavine dela kot tudi rezultati diplomskega dela lahko prosto distribuirajo, reproducirajo, uporabljajo, priobčujejo javnosti in predelujejo, pod pogojem, da se jasno in vidno navede avtorja in naslov tega dela in da se v primeru spremembe, preoblikovanja ali uporabe tega dela v svojem delu, lahko distribuira predelava le pod licenco, ki je enaka tej. Podrobnosti licence so dostopne na spletni strani [creativecommons.si](http://creativecommons.si) ali na Inštitutu za intelektualno lastnino, Streliška 1, 1000 Ljubljana.



Izvorna koda diplomskega dela, njeni rezultati in v ta namen razvita programska oprema je ponujena pod licenco GNU General Public License, različica 3 (ali novejša). To pomeni, da se lahko prosto distribuira in/ali predeluje pod njenimi pogoji. Podrobnosti licence so dostopne na spletni strani <http://www.gnu.org/licenses/>.

*Besedilo je oblikovano z urejevalnikom besedil L<sup>A</sup>T<sub>E</sub>X.*

**Kandidat:** Luka Bajić

**Naslov:** Inženiring vzporednih algoritmov

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

**Mentor:** izr. prof. dr. Jurij Mihelič

**Opis:**

V diplomski nalogi [8] 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 izr. prof. dr. Juriju Miheliču za pomoč pri izdelavi  
diplomske naloge in Urošu Koritniku za moralno podporo in koristne nasvete  
tekom študija.*









# Kazalo

**Povzetek**

**Abstract**

<b>1</b>	<b>Uvod</b>	<b>1</b>
1.1	Motivacija . . . . .	2
1.2	Struktura . . . . .	2
<b>2</b>	<b>Metode in problemi</b>	<b>5</b>
2.1	Dinamično programiranje . . . . .	5
2.2	Levenshteinova razdalja . . . . .	5
2.3	LCS . . . . .	8
<b>3</b>	<b>Vzporedni algoritmi</b>	<b>11</b>
3.1	Pristop naprej-nazaj . . . . .	12
3.2	Prostorska optimizacija . . . . .	16
3.3	Diagonalni pristop . . . . .	18
3.4	Optimizacija pomnilniškega dostopa . . . . .	19
3.5	Vzporejanje diagonalnega algoritma . . . . .	24
<b>4</b>	<b>Eksperimentalno ovrednotenje</b>	<b>31</b>
4.1	Testno okolje . . . . .	32
4.2	Primerjava algoritmov za LCS . . . . .	33
4.3	Primerjava algoritmov za Levenshteinovo razdaljo . . . . .	36

<b>5 Zaključki</b>	<b>41</b>
5.1 Sklepi . . . . .	41
5.2 Nadaljnji razvoj . . . . .	42
<b>Literatura</b>	<b>43</b>

# Seznam uporabljenih kratic

kratica	angleško	slovensko
<b>LCS</b>	longest common subsequence	najdaljše skupno podzaporedje
<b>JSON</b>	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 temeljnih 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 approach 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. 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 večitnosti 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 ogromne 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.

## 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 dejansko ključen za hitro 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 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 njihovih podproblemov. To dosežemo s hranjenjem vmesnih rezultatov v tabeli velikosti  $n \times m$ . 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 Levenshteinova razdalja

Predpostavimo, da lahko nad vsakim znakom v nizu izvedemo tri osnovne operacije: brisanje, vstavljanje in spreminjanje. Hipotetično lahko posame-

znim 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  $n$ , in  $j$ , ki iterira od 0 do  $m$ . 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  $i = 1, \dots, n, n + 1$  in  $j = 1, \dots, m, m + 1$ , kjer  $s1$  in  $s2$  predstavljata vhodna niza dolžin  $n$  in  $m$ . Končni rezultat dobimo v celici na poziciji  $(n + 1, m + 1)$ .

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

Sama implementacija je precej intuitivna (glej kodo 2.1): z dvojno zanko iteriramo po globalni tabeli in računamo vrednosti po formuli glede na enakost trenutnih dveh znakov v vhodnih nizih *str1* in *str2*, pri čemer 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 (ali več [1]) vhodnih nizih. V naših algoritmih iščemo dolžino tega podzaporedja.

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  $i = 1, \dots, n, n + 1$  in  $j = 1, \dots, m, m + 1$ , kjer  $s_1$  in  $s_2$  predstavljata vhodna niza dolžin  $n$  in  $m$ .



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

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 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  $i = 0, 1, \dots, n$  in  $j = 0, 1, \dots, m$ , nakar smo prišli do končne rešitve na poziciji  $(n, m)$ . Brez težav lahko počnemo ravno obratno: iteriramo od  $i = n, n - 1, \dots, 0$  in  $j = m, m - 1, \dots, 0$  in v tem primeru dobimo rešitev na poziciji  $(0, 0)$ . Temu rečemo pristop nazaj (glej sliko 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 iskanje Levenshteinove razdalje s pristopom nazaj

Izkaže se, da lahko z uporabo dveh niti poženemo oba pristopa istočasno (glej izvorno kodo 3.1). Najprej ustvarimo spremenljivko podatkovnega tipa `struct` (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

Skratka, v prejšnjem koraku smo tabelo v logičnem smislu prepolovili na dva dela: ena nit računa zgornjo polovico, druga pa spodnjo (glej kodo 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 dvo-dimenzionalna 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 pri Levenshteinovi razdalji iščemo najmanjšo vsoto istoležnih elementov po diagonali (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 20000 se izkaže, da če želimo hraniti celotno dvodimenzionalno tabelo v pomnilniku, mora biti njegova velikost približno 16 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 smislen za vhodne nize krajše od dolžine 65536.

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 (glej kodo 3.4): namesto dvodimenzionalne tabele *arr* iz prejšnjih implementacij, tukaj uporabljamo dve enodimenzionalni tabeli (*temp1* 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 shranjujemo v enodimenzionalno tabelo *current*. Kadar pridemo do konca vrstice, moramo te rezultate še prepisati v začasno tabelo *temp1*, da lahko z njimi računamo v naslednji iteraciji (vrstici 17 in 18 v kodi 3.4).

```
1 int backward_levenshtein_space_optimization (string str1,
    string str2, int row, int column) {
2     int temp1[column];
3     int current[column];
4
5     for(int i = row-1; i >= 0; i--) {
6         for(int j = column-1; j >= 0; j--) {
7             if (i == row-1)
8                 current[j] = column-j-1;
9             else if (j == column-1)
10                 current[j] = row-i-1;
11             else if(str1[i] == str2[j])
12                 current[j] = temp1[j+1];
13             else
14                 current[j] = 1 + min(current[j+1], min(temp1[
15 j], temp1[j+1]));
16         }
17         for(int j = 0; j < column; j++)
18             temp1[j] = current[j];
19     }
20     return current[0];
```

21 }

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, kljub temu, da mora program v vsaki iteraciji prepisovati vrednosti med tabelama. 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 ogromne 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 izboljšane 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 diagonalni.

### 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 levo, nad in levo diagonalno nad 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].

Temu problemu se izognemo tako, da matriko zarotiramo v desno za 45 stopinj in s tem dosežemo, da je vsaka diagonalna iz stare 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.

Hitro opazimo, da se je število vrstic podvojilo (oziroma natančneje: novo število vrstic je enako  $n + m - 1$ ), vendar se dejanska prostorska zahtevnost ni poslabšala, saj dejansko še vedno hranimo enako število celic. Potrebno je le, da v programski kodi pri inicializaciji tabele upoštevamo nove dolžine stolpcev 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
```

(a) Običajna matrika

(b) Optimizirana matrika

```
9         arrM[i] = new unsigned short int[row + column - i -
1];
```

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

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

Slika 3.4: Začetne vrednosti

Prvi korak je še vedno precej intuitiven: stolpec ostane enak kot pri osnovni implemetaciji, 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.

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

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

- 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

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

Slika 3.5: Zgornji trikotnik

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  $i = 2, \dots, n + 1$  in  $j = 1, \dots, i$ .

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

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  $(i - j - 1)$ ,  $s2$  pa ostane enak kot v osnovni formuli.

- Vmesne vrednosti in zamik

		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

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  $i = m, m + 1, \dots, n$  in  $j = 1, 2, \dots, m$ .

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

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

		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

- 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  $i = n + 1, n + 2, \dots, n + m - 1$  in  $j = 0, 1, \dots, n + m - i - 1$ .

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

### 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 9–11
- 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  $n\_thr$ . 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

### 3.5.1 Prostorska optimizacija diagonalnih pristopov

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 enodimenzionalne tabele, 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
- vzporedni prepis vrednosti
- sinhronizacija niti

Za razumevanje praktične implementacije tega postopka si lahko ogledamo krajši izsek programske kode za računanje zgornjega trikotnika pri vzporednem algoritmu za LCS s prostorsko in pomnilniško optimizacijo (koda 3.7). Izračunane vrednosti shranjujemo v 3 enodimenzionalne tabele *diagCurrent*, *diagTemp1* in *diagTemp2*, ključnega pomena pa je, da pred prepisom vrednosti sinhroniziramo niti (vrstica 10 v kodi 3.7).

```

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              diagCurrent[j] = 1 + diagTemp1[j-1];
6          else
7              diagCurrent[j] = max(diagTemp2[j-1], diagTemp2[j
            ]);
8      }
9
10     bar.arrive_and_wait();
11     for(int j = startIndex; j < a.col && j < startIndex+
        chunkSize; j++) {
12         diagTemp1[j] = diagTemp2[j];
13         diagTemp2[j] = diagCurrent[j];

```

```
14     }
```

```
15 }
```

Izvorna koda 3.7: Del kode za računanje 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 16 različic algoritmov in generiranje naključnih podatkov) je spisan v programskem jeziku C++<sup>20</sup>. 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 povečamo čas prevajanja, vendar je čas dejanskega izvajanja programa občutno hitrejši.

Algoritmi seveda 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 v vseh grafikonih v naslednjih 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  $n \times n$ .

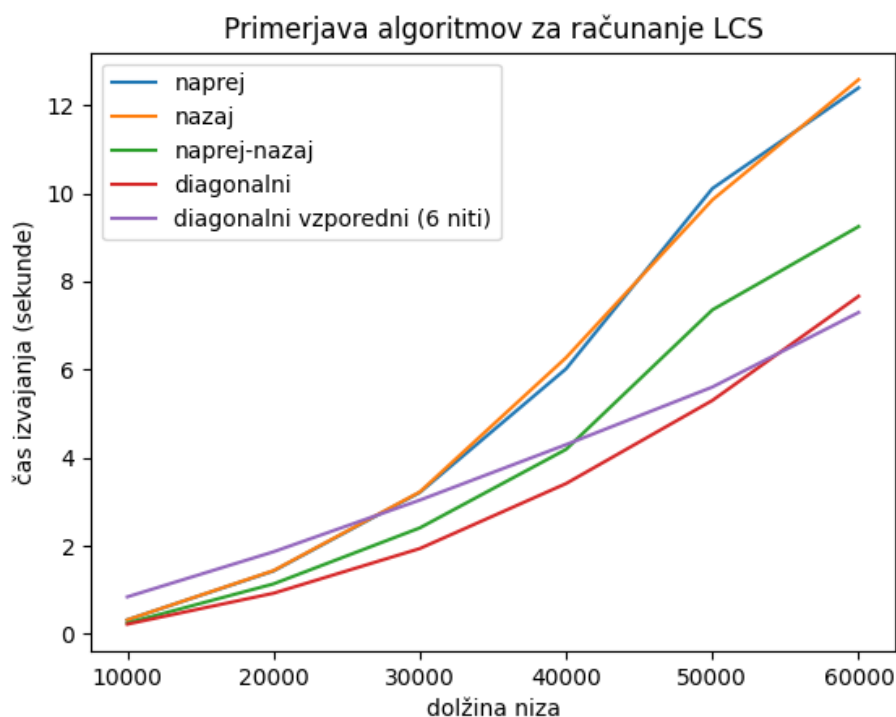
Za implementacije, ki hranijo celotno tabelo v delovnem pomnilniku smo meritve izvedli za nize dolžin med 10000 in 60000, s korakom 10000, za



implementacije 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 optimizacijo prostora in implementacije brez optimizacije prostora obravnavamo kot ločeni entiteti, zato rezultate meritev prikažemo na ločenih grafih.

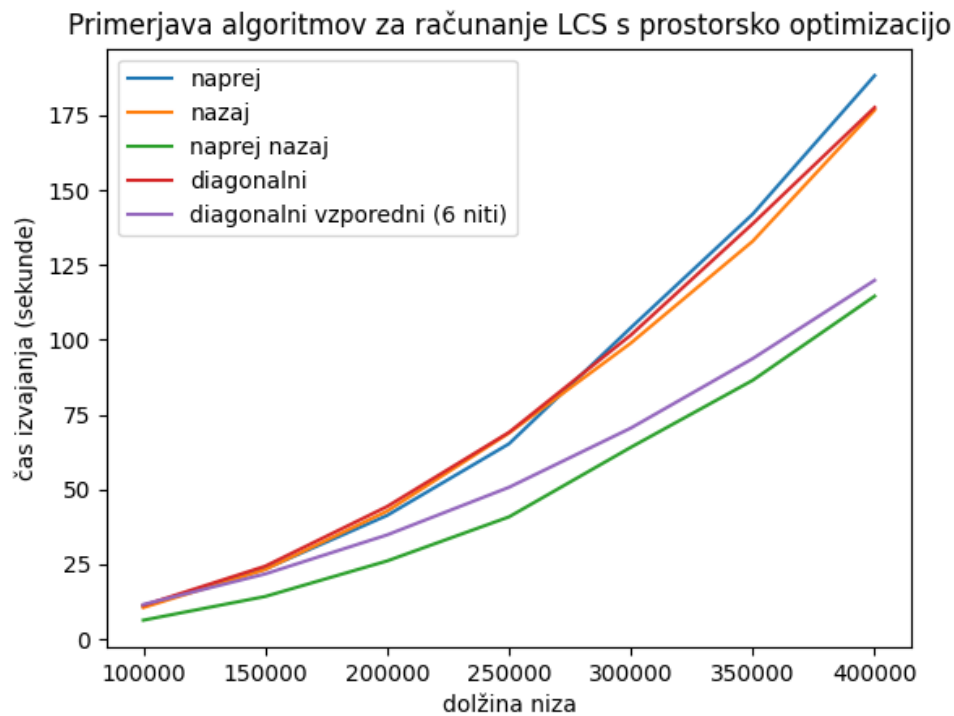


Slika 4.1: Primerjava časov izvajanja LCS algoritmov brez prostorske optimizacije

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 do-

sega 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.

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. Na neki točki pa vzporedni algoritem postane hitrejši, ker se cena sinhronizacije ne povečuje polinomske.



Slika 4.2: Primerjava časov izvajanja LCS algoritmov s prostorsko optimizacijo

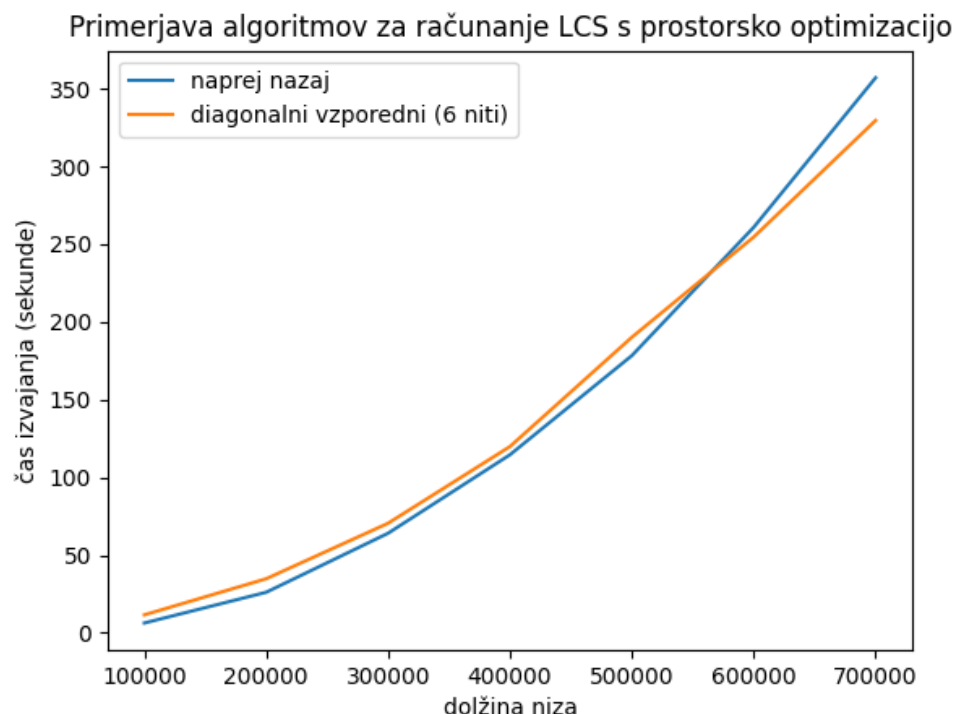
Pri primerjavi prostorsko optimiziranih funkcij (slika 4.2) ravno tako vi-

dimo, da ne obstaja bistvena razlika med pristopoma naprej in nazaj, je pa tukaj pristop naprej-nazaj res skoraj dvakrat hitrejši, iz česar lahko sklepamo, da je ozko grlo pri prenašanju dvodimenzionalnih tabel v predpomnilnik.

Pri nizih dolžine približno 200000 začnemo opazovati občutno razliko med tremi osnovnimi algoritmi (naprej, nazaj in diagonalni) in vzporednima algoritmoma (naprej-nazaj in diagonalni vzporedni). Nekoliko presenetljiv rezultat je, da je tudi na relativno velikih dolžinah vhodnih podatkih pristop naprej-nazaj rahlo hitrejši od diagonalnega. 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 smo pričakovali, da obstaja neka točka na kateri bo diagonalni algoritem postal hitrejši od naprej-nazaj, zato smo izvedli dodatne teste na še večjih dolžinah vhodnih nizov. Ker gre za metode s prostorsko optimizacijo tukaj nismo pretirano zaskrbljeni glede porabe pomnilnika: za tri enodimenzionalne tabele dolžine 700000 potrebujemo približno 8 GB pomnilniškega prostora (če podatkovni tip `int` zasede 4 bajte v pomnilniku), kar je sprejemljivo za specifikacije naprave na kateri smo izvajali teste.

Kot je razvidno iz slike 4.3, smo res našli točko, kjer je cena sinhronizacije 6 niti dovolj majhna, da postane diagonalni algoritem hitejši od dveh neodvisnih niti - ta je približno med 500000 in 600000, pri 700000 pa je diagonalni algoritem hitrejši že za skoraj pol minute.

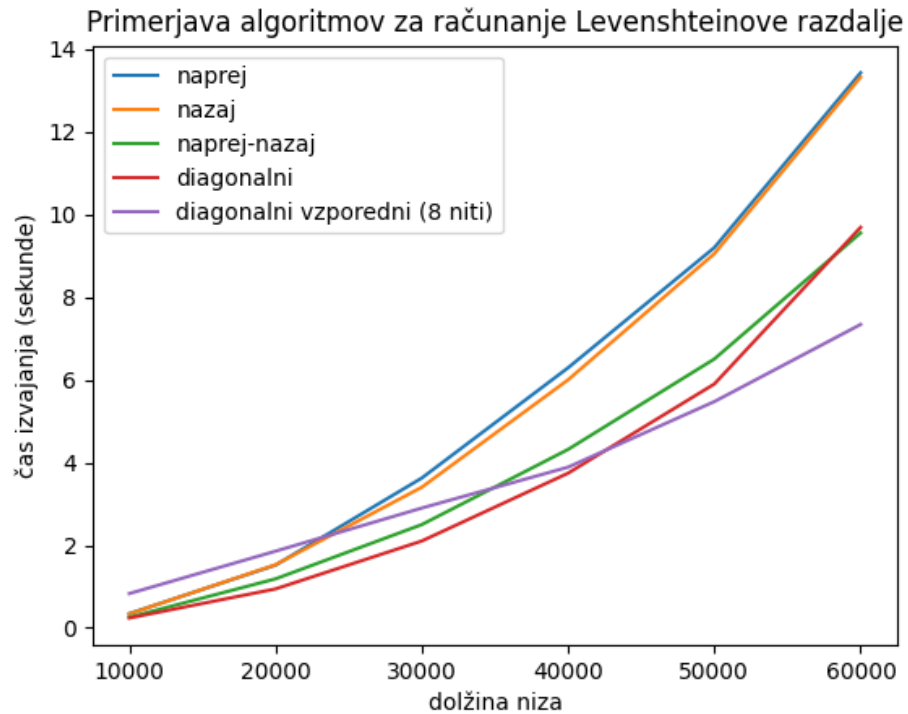


Slika 4.3: Dodatna primerjava časov izvajanja LCS algoritmov s prostorsko optimizacijo

### 4.3 Primerjava algoritmov za Levenshteinovo razdaljo

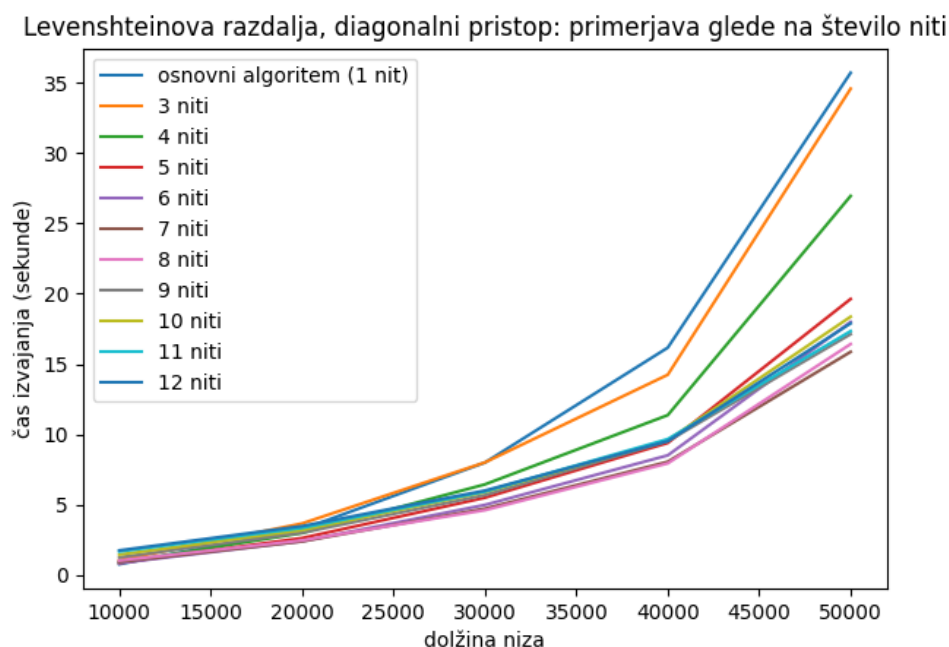
Ker sta algoritma za iskanje LCS in Levenshteinove razdalje konceptualno zelo podobna, lahko pričakujemo, da bo to razvidno tudi iz praktičnih meritev. Že na prvi pogled vidimo, da so razmerja med krivuljami res približno enaka (če primerjamo npr. sliko 4.4 s sliko 4.1). Dejanski časi izvajanja so tukaj nekoliko daljši, predvidoma zaradi gnezdenega klica funkcije `min()`, ki ni prisoten v algoritmu za računanje LCS.

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



Slika 4.4: Primerjava časov izvajanja algoritmov za iskanje Levenshteinove razdalje brez prostorske optimizacije

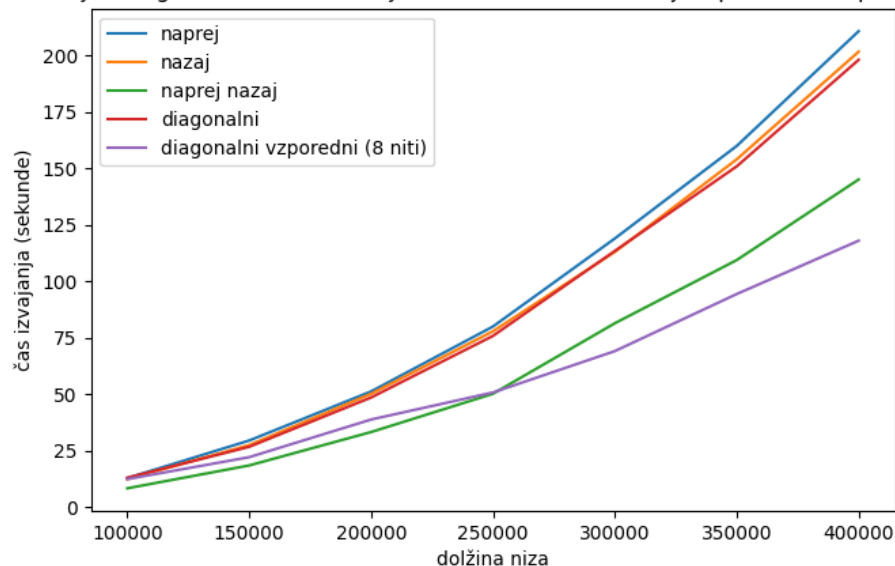
LCS algoritmi (slika 4.1) je osnovni diagonalni algoritem hitrejši od vzporednega do nizov dolžine približno 50000, pri Levenshteinovi razdalji (slika 4.4) pa sta časa izvajanja približno enaka že pri nizih dolžine 40000.



Slika 4.5: Primerjava časov izvajanja diagonalnih algoritmov glede na število niti

Na tem mestu je smiselno, da si ogledamo še primerjavo časov izvajanja diagonalnega algoritma glede na število niti (slika 4.5). Iz specifikacij sistema vemo, da lahko hkrati poganjamo največ 8 niti, vendar vidimo, da je tudi večje število niti (npr. 12) še vedno hitrejša 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.

Primerjava algoritmov za računanje Levenshteinove razdalje s prostorsko optimizacijo



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

Na sliki 4.6 je prikazana še primerjava časov izvajanja algoritmov s prostorsko optimizacijo. Ponovno vidimo, da je diagonalni vzporedni algoritem občutno hitrejši na bistveno krajših dolžinah nizov kot pri LCS (250000 proti 600000), zato tukaj niti ni potrebno, da izvajamo dodatne meritve nad dolžinami 400000.





# 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 naprejnazaj. 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 kolilčino jeder, oziroma ali je mogoče algoritme še dodatno prilagoditi za tovrstne sisteme.

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 Breiting. “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*. Diplomaska naloga. Fakulteta za  
računalništvo in informatiko, Univerza v Ljubljani, 2020.