Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Bajić

Inženiring vzporednih algoritmov

diplomsko delo

visokošolski študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2024

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 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 LaTeX.

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

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

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

Kazalo

Seznam uporabljenih kratic

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

Povzetek

Naslov: Inženiring vzporednih algoritmov

Avtor: Luka Bajić

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

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

Abstract

Title: Parallel Algorithm Engineering

Author: Luka Bajić

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

Keywords: computer, computer, computer.

Poglavje 1

Uvod

V diplomskem delu so opisani različni pristopi za optimizacijo algoritmov za računanje Levenshteinove razdalje in najdaljšega skupnega podzaporedja (angl. Longest Common Subsequence - LCS) z uporabo vzporednega procesiranja. Gre za dva zelo pogosto uporabljena algoritma za potrebe primerjave nizov oziroma iskanje 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 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 algoritmov, ki služijo kot gradniki aplikacij na številnih področjih in v različnih domenah, v katerih se običajno srečujemo z zelo specifičnimi potrebami oz. zahtevami. Tipičen primer je bioinformatika ali genetika [2], kjer pogosto naletimo na probleme z ozkim naborom znakov in lahko algoritme temu primerno tudi prilagodimo. Kot drugi primer vzamemo orodja za kontrolo različic. Tukaj lahko pričakujemo, da bosta dve datoteki, ki ju primerjamo, večinoma zelo podobni, morda samo z nekaj spremenjenimi vrsticami, posledično lahko razvijemo algoritem, ki bo tovrstna pričakovanja upošteval in se morda na nek način obnašal bolj uporabniku prijazno (npr. z nekimi hitrimi, vmesnimi aproksimacijami, če je celotno izvajanje preveč dolgotrajno). Vsekakor pa vemo, da je vsaka optimizacija algoritmov za primerjavo nizov zelo koristna, zato smo v tem diplomskem delu preučili nekaj tovrstnih pristopov za algoritma za računanje LCS in Levenshteinove razdalje.

1.2 Struktura

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

Poglavje 2

Metode in paradigme

V tem poglavju je predstavljena algoritmična paradigma 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 paradigma, s katero skušamo optimizirati izvajanje problemov, ki imajo takoimenovano optimalno podstrukturo, kar pomeni, da je do rešitve problema možno priti preko optimalne rešitve njihovih podproblemov. Za to obstajata dva glavna načina: rekurzivni (z uporabo memoizacije) in iterativni (z uporabo tabeliranja). Za našo implementacijo smo uporabili iterativni pristop, kar pomeni, da hranimo vmesne rezultate v tabeli velikosti $n \times m$. 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 posameznim operacijam dodelimo tudi uteži [3], vendar ker to nima občutnega vpliva na časovno zahtevnost algoritma, v nadaljevanju sklepamo, da imajo vse tri enako utež. S tem naborom operacij obstaja teoretično neskončno možnih načinov za transformacijo nekega niza v poljuben drugi niz, Levenshteinova razdalja pa predstavlja dolžino najkrajše tovrstne transformacije.

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

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

Slika 2.1: Primer izračunane matrike za iskanje Levenshteinove razdalje za dva naključna niza

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

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

Sama implementacija je precej intuitivna - 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
15    return arr[row][column];
16 }
```

Izvorna koda 2.1: Algoritem za izračun Levenshteinove razdalje

2.2.1 Primer uporabe

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

2.3 LCS

Podsekvenco definiramo kot zaporedje znakov, ki se nahajajo v nizu, v istem vrstnem redu, vendar ne nujno zaporedno. Torej, niz "$ABCD$" vsebuje podsekvenco "ACD", ne vsebuje pa podsekvence "DCB". Najdaljša skupna podsekvenca, oziroma LCS, predstavlja najdaljšo možno podsekvenco, ki se nahaja v dveh (ali več) vhodnih nizih, v naših algoritmih pa iščemo dolžino te podsekvence.

Pristop reševanja tega problema je podoben kot za Levenshteinovo razdaljo v prejšnjem poglavju: začetne vrednosti za prvo vrstico in prvi stolpec so v tem primeru kar ničle, saj je pri praznem nizu dolžina podsekvence vedno 0. Preostanek tabele pa izračunamo po formuli (2.2), za $i = 1, ..., n$ in $j = 1, ..., m$, kjer $s1$ in $s2$ predstavljata vhodna niza dolžin n in m. Končni rezultat ponovno dobimo v celici (n, m), kot je prikazano na sliki 2.2.

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

		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 (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
11    return arr[row][column];
12 }
```

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 diagonalno

- LCS diagonalno s prostorsko optimizacijo
- LCS diagonalno vzporedno (poljubno število niti)
- LCS diagonalno vzporedno 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 diagonalno
- Levenshteinova razdalja diagonalno s prostorsko optimizacijo
- Levenshteinova razdalja diagonalno vzporedno (poljubno število niti)
- Levenshteinova razdalja diagonalno vzporedno 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 $i = 0, 1, ..., n$ in $j = 0, 1, ..., m$, nakar smo prišli do končne rešitve na poziciji (n, m). 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 (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 (3.1).

```
1 int fb_LCS (string str1, string str2, int row, int column) {
2
3     int h = row / 2;
4     struct args a;
5     a.s1 = str1;
6     a.s2 = str2;
7     a.row = h;
8     a.col = column;
9
10    thread t1(topHalf_LCS, ref(a));
11    thread t2(bottomHalf_LCS, ref(a));
12
13    t1.join();
14    t2.join();
15
16    //merge results to find the actual distance
17    return merge_LCS(h, row, column);
18 }
```

Izvorna koda 3.1: Algoritem LCS naprej-nazaj

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

```
1 void topHalf_LCS (args& a) {
2     for(int i = 0; i <= a.row; i++)
3         for(int j = 0; j < a.col; j++)
4             if(i == 0 || j == 0)
5                 arr[i][j] = 0;
6             else if(a.s1[i-1] == a.s2[j-1])
7                 arr[i][j] = 1 + arr[i-1][j-1];
8             else
9                 arr[i][j] = max(arr[i][j-1], arr[i-1][j]);
10 }
11
12 void bottomHalf_LCS (args& a) {
13     int nrows = a.s1.length();
14
15     //in case there's nothing for this thread to do
16     if(nrows == 1)
17         return;
18
19     for(int i = nrows+1; i > a.row; i--)
20         for(int j = a.col; j > 0; j--)
21             if (i == nrows+1 || j == a.col)
22                 arr[i][j] = 0;
23             else if(a.s1[i-1] == a.s2[j-1])
24                 arr[i][j] = 1 + arr[i+1][j+1];
25             else
26                 arr[i][j] = max(arr[i][j+1], arr[i+1][j]);
27 }
```

Izvorna koda 3.2: Vsaka nit računa svojo polovico tabele

Kadar obe niti zaključita izvajanje, je rezultat funkcije naprej-nazaj dvodimenzionalna tabela, pri kateri sta relevantni samo dve vrstici. Primer za dva naključna niza je prikazan na sliki (3.2).

Da dobimo končno 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.

		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 naprej-nazaj na naključnih nizih

Za združevanje pri Levenshteinovi razdalji iščemo najmanjšo vsoto istoležnih elementov po diagonali, kot je prikazano v kodi (3.3), pri LCS pa največjo.

```
1 int merge_levenshtein (int h, int row, int column) {
2     int temp = 0;
3     int currentMin = INT_MAX;
4
5     for(int i = 1; i <= column; i++) {
6         temp = arr[h][i-1] + arr[h+1][i];
7
8         if(temp < currentMin)
9             currentMin = temp;
10    }
11
12    return currentMin;
13 }
```

Izvorna koda 3.3: Združevalna funkcija za Levenshteinovo razdaljo

3.2 Prostorska optimizacija

Pri delu z ogromno količino vhodnih podatkov se lahko zgodi da velikost tabele presega kapaciteto pomnilnika. Že za 2 niza dolžine 20000 se izkaže, da če želimo hraniti celotno dvodimenzionalno tabelo v pomnilniku, mora biti velikost le-tega približno 16GB, če predpostavimo da spremenljivke podatkovnega tipa int zasedejo 4 bajte na danem sistemu. Pri naših meritvah, ki so podrobneje opisane v poglavju 4, smo se temu problemu izognili z uporabo spremenljivk podatkovnega tipa unsigned short int, ki zasedajo samo 2 bajta v pomnilniku. Pri tem smo morali biti pozorni, da ne pride do preliva (angl. overflow), saj lahko z dvema bajtoma hranimo maksimalno vrednost 65535, vendar ker vemo, da je tako za LCS kot za Levenshteinovo razdaljo zgornja meja rezultata manjša ali enaka dolžini daljšega izmed vhodnih nizov, je ta pristop vsekakor smiselen 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 (3.4). Kot je razvidno iz rezultatov v 4. poglavju, smo s tem dosegli dodatno pohitritev, kljub temu, da mora program v vsaki iteraciji prepisovati vrednosti med tabelama. Glavna slabost tega pristopa pa je, da 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 razdaljo med nizoma, ne moremo pa ugotoviti kako dejansko pretvoriti prvi niz v drugega (kar sicer ni bilo relevantno pri naši analizi, je pa vseeno pomemben koncept pri marsikaterih praktičnih aplikacijah tovrstnih algoritmov). Vsekakor pa je ta način zelo uporaben za ogromne vhodne podatke, kadar hranjenje dvodimenzionalne tabele v pomnilniku ne pride v poštev.

```
1 int backward_levenshtein_space_optimization (string str1,
    string str2, int row, int column) {
```

```
2     int temp1[column];
3     int current[column];
4
5     //initialize zeros
6     for (int i = 0; i < column; i++) {
7         temp1[i] = 0;
8         current[i] = 0;
9     }
10
11    for(int i = row-1; i >= 0; i--) {
12        for(int j = column-1; j >= 0; j--) {
13            if (i == row-1)
14                current[j] = column-j-1;
15            else if (j == column-1)
16                current[j] = row-i-1;
17            else if(str1[i] == str2[j])
18                current[j] = temp1[j+1];
19            else
20                current[j] = 1 + min(current[j+1], min(temp1[
    j], temp1[j+1]));
21        }
22
23        //rewrite data
24        for(int j = 0; j < column; j++)
25            temp1[j] = current[j];
26    }
27
28    return current[0];
29 }
```

Izvorna koda 3.4: Prostorska optimizacija

3.3 Diagonalni pristop

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

morda celo poslabša zaradi prevelikega overheada.

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

3.4 Optimizacija pomnilniških dostopov

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.

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

		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

Hitro opazimo, da se je število vrstic podvojilo, 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 na sledeč način (3.5):

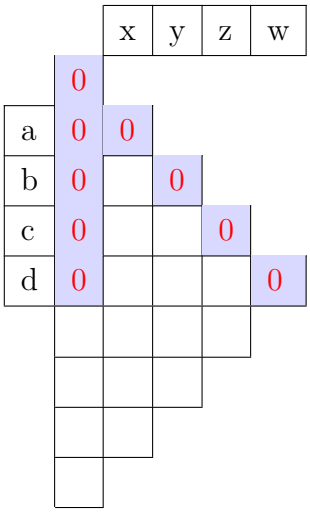
```
1 int newRow = row + column - 1;
2
3 arrMemory = new unsigned short int*[newRow];
4
5 for (int i = 0; i < newRow; i++)
6     if(i < row)
7         arrMemory[i] = new unsigned short int[i+1];
8     else
9         arrMemory[i] = new unsigned short int[row + column -
    i - 1];
```

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 razdeliti kodo na štiri dele:

- Prva vrstica in stolpec (vrstica je zdaj dejansko diagonala)



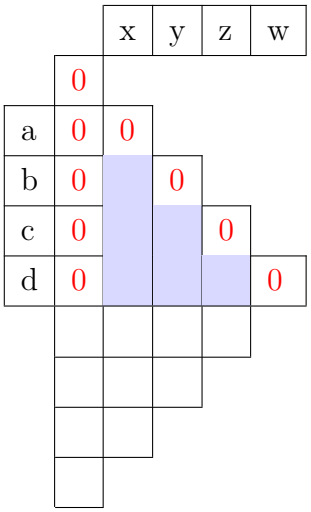
Slika 3.4: Začetne vrednosti

```
1         for (int i = 0; i < row; i++)
2             arrMemory[i][0] = 0;
3         for (int j = 1; j < column; j++)
4             arrMemory[j][j] = 0;
5
```

Izvorna koda 3.6: Inicializacija začetnih vrednosti za LCS

- Zgornji trikotnik

```
1         for(int i = 2; i < column; i++) {
2             for(int j = 1; j < i; j++) {
3                 if(str1[i-j-1] == str2[j-1])
```



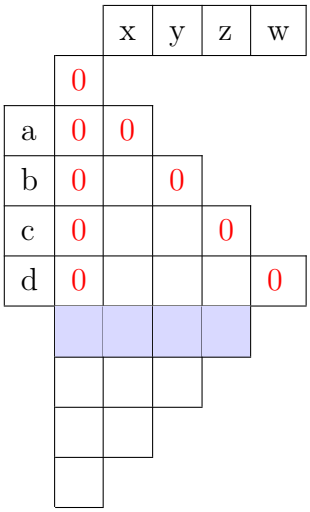
Slika 3.5: Zgornji trikotnik

```
4                       arrMemory[i][j] = 1 + arrMemory[i-2][
    j-1];
5                   else
6                       arrMemory[i][j] = max(arrMemory[i-1][
    j-1], arrMemory[i-1][j]);
7               }
8           }
9
```

Izvorna koda 3.7: Izračun zgornjega trikotnika za LCS

- Vmesne vrednosti

```
1           for(int i = column; i < row; i++) {
2               for(int j = 1; j < column; j++) {
3                   if(str1[i-j-1] == str2[j-1])
4                       arrMemory[i][j] = 1 + arrMemory[i-2][
    j-1];
5                   else
6                       arrMemory[i][j] = max(arrMemory[i-1][
    j], arrMemory[i-1][j-1]);
7               }
8           }
```



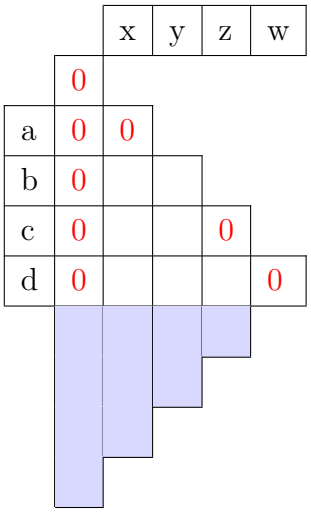
Slika 3.6: Vmesne vrednosti

```
9
10          for(int j = 0; j < column - 1; j++) {
11              if(str1[row-j-2] == str2[j])
12                  arrMemory[row][j] = 1 + arrMemory[row-2][
    j];
13              else
14                  arrMemory[row][j] = max(arrMemory[row-1][
    j+1], arrMemory[row-1][j]);
15          }
16
```

Izvorna koda 3.8: Izračun vmesnih vrednosti za LCS

- Spodnji trikotnik

```
1           for(int i = row+1; i < newRow; i++) {
2               for(int j = 0; j < newRow - i; j++) {
3                   if(str1[row-j-2] == str2[j+i-row])
4                       arrMemory[i][j] = 1 + arrMemory[i-2][
    j+1];
5                   else
6                       arrMemory[i][j] = max(arrMemory[i-1][
    j+1], arrMemory[i-1][j]);
```



Slika 3.7: Spodnji trikotnik

```
7                 }
8             }
9
```

Izvorna koda 3.9: Izračun spodnjega trikotnika za LCS

Poglavje 4

Rezultati

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 medsebojno primerjavo le-teh. Glede na to, da je večnitenje zelo odvisno od same arhitekture sistema na katerem izvajamo programsko kodo, kot tudi od raznih zunanjih dejavnikov (npr. časovno razvrščanje operacijskega sistema), je predvsem pri tabelah manjših velikosti smiselno, da posamezno funkcijo kličemo večkrat in kot končni rezultat vzamemo povprečje vseh meritev.

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 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 $n \times n$.

4.1 Generiranje podatkov

Vhodni podatki so enakega tipa tako za LCS kot za Levenshteinovo razdaljo: teoretično gre za poljubno število znakovnih nizov, vendar smo se za potrebe tega diplomskega dela omejili na delo 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.

4.2 Delovno okolje

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

- operacijski sistem: Windows 11 Home, 64-bitni
- procesor: AMD Ryzen 7 6800H:
  - 3.2GHz
  - 8 jeder
  - 16MB L3 predpomnilnika
- delovni pomnilnik: 16GB 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++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 [4] v JSON datoteki tasks.json. S tem nekoliko povečamo čas prevajanja, vendar je čas dejanskega izvajanja programa občutno hitrejši.

4.3 Primerjava algoritmov za LCS

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.

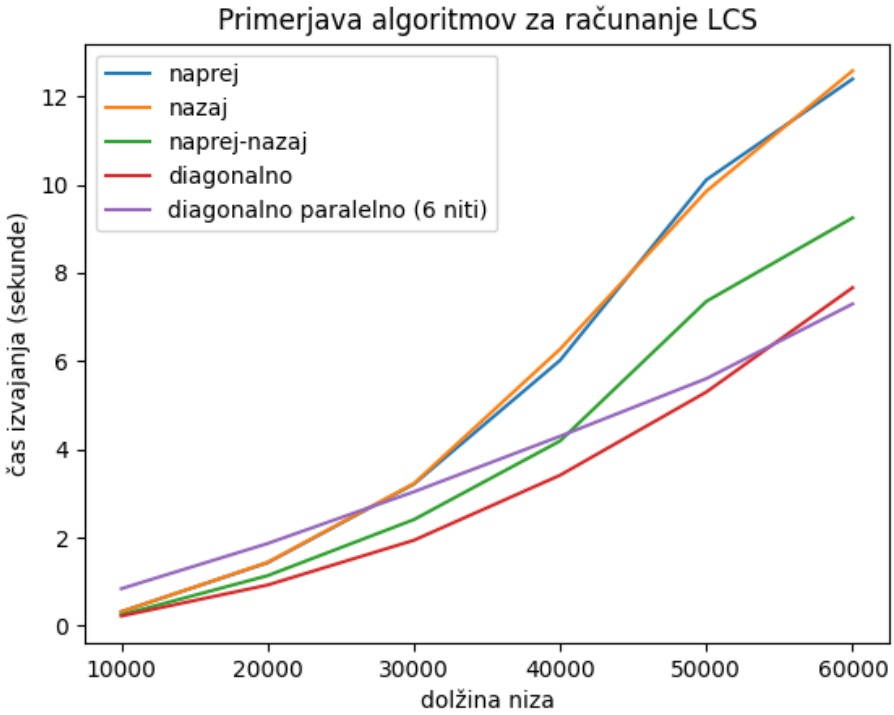
Kot smo pričakovali, je čas izvajanja za pristopa naprej in nazaj praktično enak, ker gre dejansko za isti algoritem, le z različnim vrstnim redom računanja celic. S pristopom naprej-nazaj pa dosežemo pohitritev že na meritvi z nizi dolžine 10000 (4.1). Izkaže pa se, da pohitritev ni ravno dvakratna, 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.

Pri primerjavi prostorsko optimiziranih funkcij (4.2) ravno tako vidimo, 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 opažati občutno razliko med tremi osnovnimi algoritmi (naprej, nazaj in diagonalno) in vzporednima algoritmoma (naprej-nazaj in diagonalno paralelno). 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.

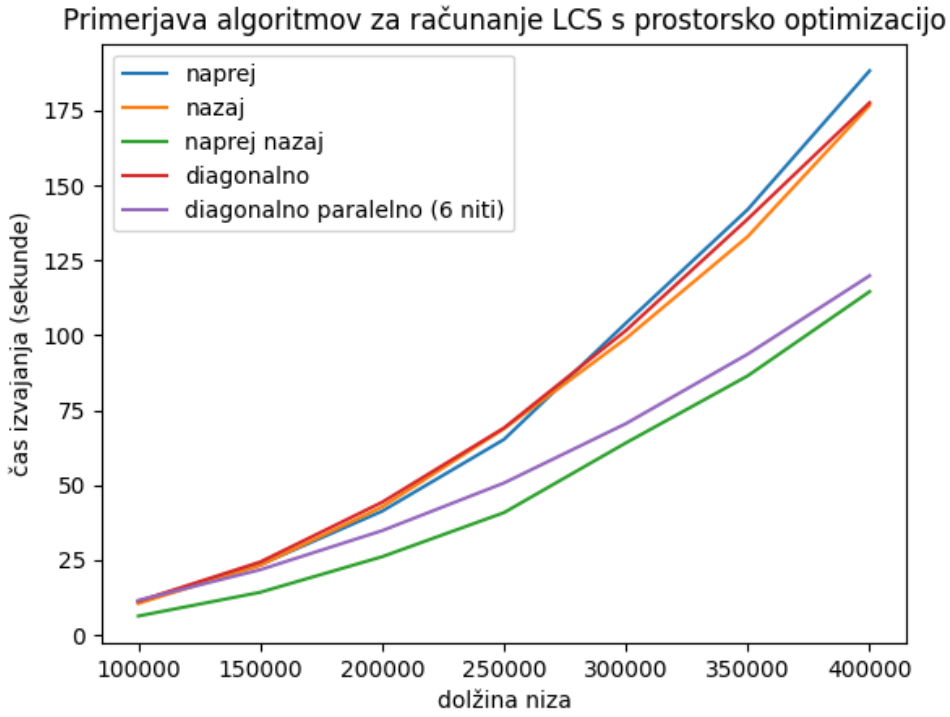
Vseeno pa smo pričakovali, da obstaja neka točka na kateri bo diagonalni algoritem postal hitrejši od naprej-nazaj, zato smo izvedli dodatne teste na še večjih dolžinah vhodnih nizov. Ker gre za metode s prostorsko optimizacijo



Slika 4.1: Primerjava časov izvajanja LCS algoritmov brez prostorske optimizacije

tukaj nismo pretirano zaskrbljeni glede porabe pomnilnika: za tri enodimenzionalne tabele dolžine 700000 potrebujemo približno 8GB 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 vzporedni algoritem hitrejši že za skoraj pol minute.

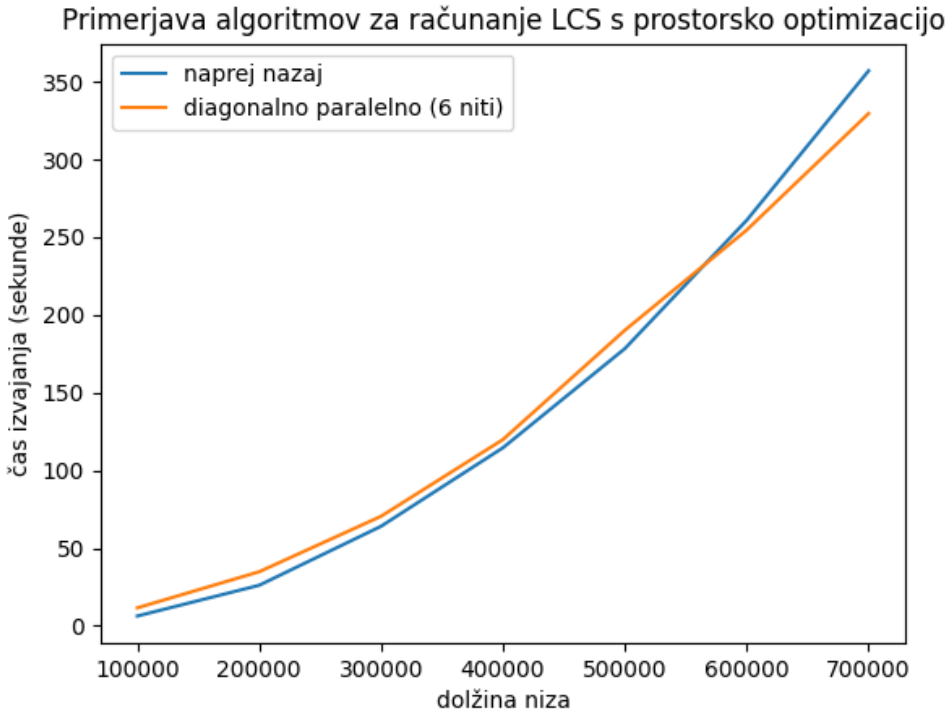


Slika 4.2: Primerjava časov izvajanja LCS algoritmov s prostorsko optimizacijo

4.4 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 (4.4). 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

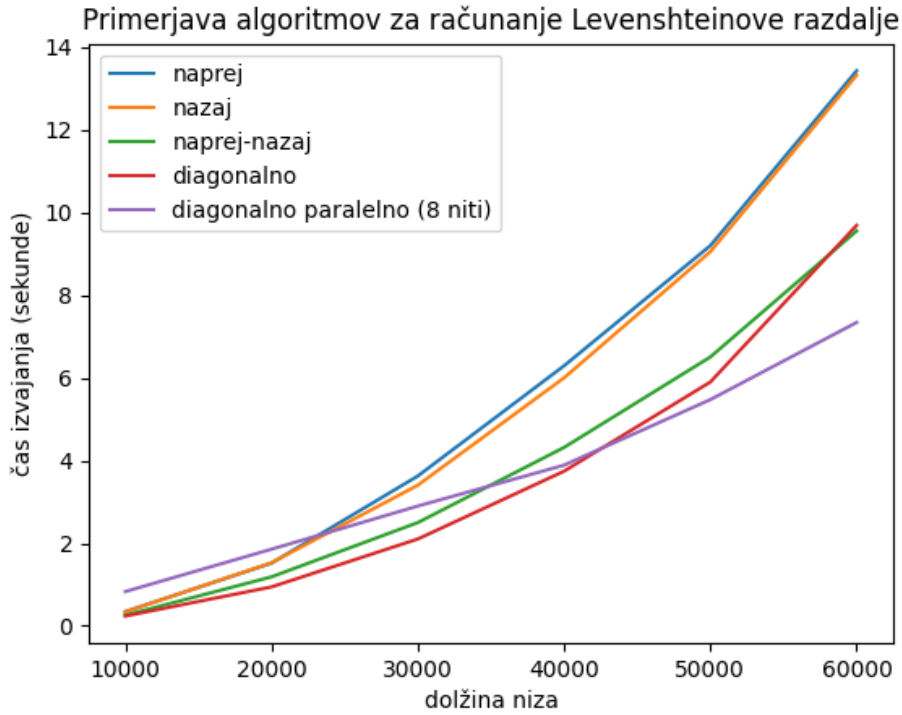


Slika 4.3: Dodatna primerjava časov izvajanja LCS algoritmov s prostorsko optimizacijo

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

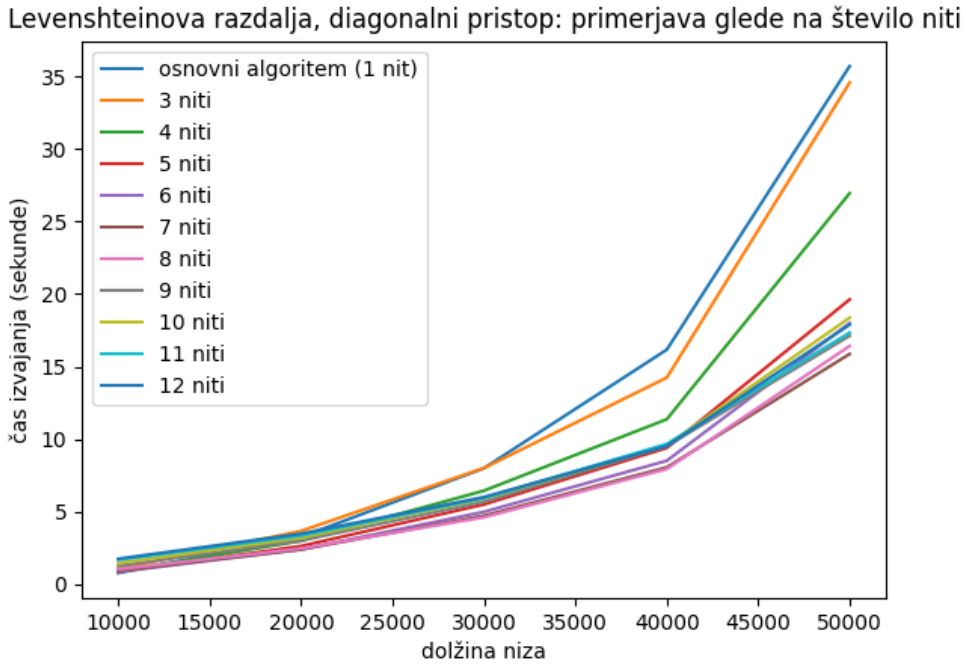
Na tem mestu je smiselno, da si ogledamo še primerjavo časov izvajanja diagonalnega algoritma glede na število niti (4.5). Iz specifikacij sistema vemo, da lahko hkrati poganjamo največ 8 niti, vendar vidimo, da je tudi večje število niti (npr. 12) še vedno hitrejše od osnovnega algoritma. Ker pride v tem primeru do dodatnega čakanja med nitmi, seveda to ni smiselno, zato smo za meritve za Levenshteinovo razdaljo uporabljali 8 niti, za LCS pa se v povprečju bolje izkaže samo 6 niti.

Na sliki (4.6) je prikazana še primerjava časov izvajanja algoritmov s

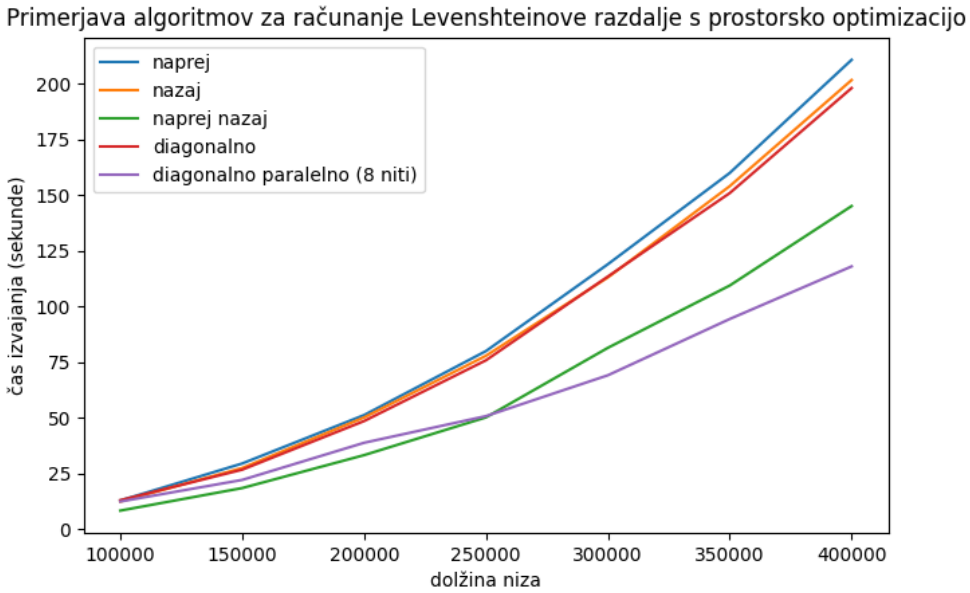


Slika 4.4: Primerjava časov izvajanja algoritmov za iskanje Levenshteinove razdalje brez prostorske optimizacije

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.



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



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

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.

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

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

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

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

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