

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](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:** doc. dr. Jurij Mihelič

**Opis:**

V diplomski nalogi [2] 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.

**Title:** Parallel algorithm engineering

**Description:**

opis diplome v angleščini



*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

Povzetek

Abstract

<b>1</b>	<b>Uvod</b>	<b>1</b>
1.1	Motivacija . . . . .	1
1.2	Struktura . . . . .	1
<b>2</b>	<b>Metode in paradigme</b>	<b>3</b>
2.1	Dinamično programiranje . . . . .	3
2.2	LCS . . . . .	3
2.3	Levenshteinova razdalja . . . . .	3
<b>3</b>	<b>Vzporedni algoritmi</b>	<b>5</b>
3.1	Pristop naprej-nazaj . . . . .	6
3.2	Prostorska optimizacija . . . . .	7
3.3	Diagonalni pristop . . . . .	8
<b>4</b>	<b>Rezultati</b>	<b>9</b>
4.1	Generiranje podatkov . . . . .	9
4.2	Delovno okolje . . . . .	10
4.3	Teoretične pohitritve/Časovne zahtevnosti . . . . .	10
4.4	Primerjava sekvenčnih algoritmov . . . . .	10
4.5	Primerjava vzporednih algoritmov . . . . .	10
4.6	Celotna primerjava . . . . .	10

<b>5 Zaključki</b>	<b>11</b>
5.1 Sklepi . . . . .	11
5.2 Nadalnji razvoj . . . . .	11
<b>Literatura</b>	<b>13</b>

# Seznam uporabljenih kratic

kratica	angleško	slovensko
<b>LCS</b>	longest common subsequence	najdaljša skupna podse- kvenca
<b>JSON</b>	JavaScript Object Notation	JavaScript objektna notacija
<b>SVM</b>	support vector machine	metoda podpornih vektorjev



# 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 preverjevalnikov črkovanja do orodij za kontrolo verzij (npr. Git). Diplomsko delo opiše različne pristope optimizacije dveh tovrstnih algoritmov: Levenshteinove razdalje in najdaljše skupne podsekvence (LCS), 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

This sample document presents an approach to typesetting your BSc thesis using L<sup>A</sup>T<sub>E</sub>X. A proper abstract should contain around 100 words which makes this one way too short.

**Keywords:** computer, computer, computer.





# Poglavje 1

## Uvod

V diplomskem delu so opisani različni pristopi za optimizacijo algoritmov za računanje Levenshteinove razdalje in najdaljše skupne podsekvence (??) (LCS - longest common subsequence) z uporabo vzporednega procesiranja.

### 1.1 Motivacija

### 1.2 Struktura

Drugo poglavje opiše algoritma, njuno praktično uporabnost in njuno implementacijo z uporabo dinamičnega programiranja. Tretje poglavje predstavi dva glavna pristopa paralelizacije - diagonalni in naprej-nazaj. V četrtem poglavju so prikazani rezultati meritev na različnih velikostih vhodnih podatkov. V petem poglavju sledijo še sklepi in ideje za nadaljnji razvoj.



## Poglavje 2

# Metode in paradigme

V tem poglavju je predstavljena implementacija algoritma za računanje Levenshteinove razdalje in LCS razdalje z uporabo dinamičnega programiranja. V tem kontekstu je namenjena predvsem za časovno primerjavo med sekvenčnim in vzporednim algoritmom.

### 2.1 Dinamično programiranje

### 2.2 LCS

### 2.3 Levenshteinova razdalja

#### 2.3.1 Primer uporabe

Primer praktične uporabe [1]



## Poglavje 3

# Vzporedni algoritmi

Osnovna struktura vzporednega algoritma za oba problema je enaka kot njuna rešitev iz prejšnjega poglavja - še vedno uporabljamo dvodimenzionalno tabelo za hranjenje vmesnih vrednosti, torej je prostorska zahtevnost enaka -  $O(n*m)$ . Razlika je samo v tem, da vrednosti, ki niso medsebojno odvisne, lahko računamo vzporedno z uporabo koncepta večnitnosti (angl. multithreading). Za to obstajata dva glavna pristopa - diagonalni in naprej-nazaj.

Vsega skupaj smo implementirali 14 različic algoritmov:

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

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

### 3.1 Pristop naprej-nazaj

V prejšnjem poglavju 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  $(i, j)$ . 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  $(0, 0)$ . Izkaže se, da lahko z uporabo dveh niti poženemo oba načina istočasno. Tabelo v abstraktnem smislu prepolovimo na dva dela - ena nit računa zgornjo polovico, druga pa spodnjo. Na ta način, vsaj teoretično, dosežemo dvakratno pohitritev. Na koncu moramo še združiti zadnjo vrstico prve niti z zadnjo vrstico druge niti z združevalni funkcijo [koda], 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.

3.1

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

```
9             arr[i][j] = max(arr[i-1][j], arr[i][j-1]);  
10  
11     return arr[row-1][column-1];  
12 }
```

Izvorna koda 3.1: Algoritem LCS naprej

## 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 overflowa (??), saj lahko z dvema bajtoma hranimo maksimalno vrednost 65535. Vendar ker vemo, da je tako za LCS kot za Levenshteinovo razdaljo zgornja meja rezultata manjša ali enaka dolžini daljšega izmed vhodnih nizov, je ta pristop vseeno smislen za vhodne nize krajše od dolžine 65536 (čeprav nizi te dolžine kljub vsemu presegajo kapaciteto pomnilnika na povprečnem osebнем računalniku, na katerem smo poganjali teste).

Vseeno pa je smiselno najti splošnejšo rešitev. Izkaže se, da pri pristopu naprej-nazaj, oziroma tudi brez uporabe večnitnosti pri pristopih naprej in nazaj, lahko bistveno zmanjšamo porabo prostora tako, da v pomnilniku hranimo samo vrstico, ki jo trenutno računamo in prejšnjo vrstico, ostale so nepotrebne in jih lahko zavržemo. Kot je razvidno iz rezultatov v 4. poglavju, smo s tem nekoliko poslabšali časovno zahtevnost, saj moramo v vsaki iteraciji kreirati dodatno enodimenzionalno tabelo, oziroma prepisovati vrednosti. Vsekakor pa je ta način zelo uporaben za ogromne vhodne podatke.

### 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.

[slikovni prikaz]



# Poglavje 4

## Rezultati

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

### 4.1 Generiranje podatkov

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

## 4.2 Delovno okolje

VSCode G++ -O3 AMD 16GB

## 4.3 Teoretične pohitritve/Časovne zahtevnosti

## 4.4 Primerjava sekvenčnih algoritmov

## 4.5 Primerjava vzporednih algoritmov

## 4.6 Celotna primerjava

## Poglavje 5

# Zaključki

### 5.1 Sklepi

### 5.2 Nadalnji razvoj

Kot je razvidno iz poglavja 4., smo z vzporednimi algoritmi dosegli občutno pohitritev že na običajnih osebnih računalnikih. V nadalnje bi bilo zanimivo preveriti ali je čas izvajanja morda še krajši na zmogljivejših sistemih z ogromno količino jeder, oziroma ali je mogoče algoritme še dodatno prilagoditi za tovrstne sisteme.



# Literatura

- [1] Peter Coates in Frank Breiting. “Identifying document similarity using a fast estimation of the Levenshtein Distance based on compression and signatures”. V: *Proceedings of the Digital Forensics Research Conference Europe (DFRWS EU)*. 2022. DOI: 10.48550/arXiv.2307.11496.
- [2] Leon Premk. *Algoritmi za izračun razdalje med časovnimi vrstami z dinamičnim prilagajanjem časa*. Diplomaska naloga. Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, 2020.