Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

Паралелизација проблема Левенштајновог растојања

Аутор:
Владимир Јовин

Индекс:
Е2 120/2022

18. јануар 2023.

Сажетак

Динамичко програмирање предтсавља оптимизацију на обичну рекурзију. Своди се на решавање оптимизационог проблема који је подељен на мање потпроблеме, односно формирана је нека рекурентна форма, чији резултати израчунавања се чувају у некој структури, да би би били израчунати највише једном, иако више елеменатат може зависити од истог потпроблема. Са друге стране, паралелно програмирање има за циљ смањење времена извршавања. Овај рад се бави разматрањем коришћења ове две методе за потребе решавања проблема Левенштајновог растојања.

Садржај

Списак изворних кодова

Списак слика

Списак табела

1 Увод

Проблем Левенштајнове дистанце може се неформално дефинисати као растојање између две секвенце карактера [2], у смислу, који је минималан број потребних операција из скупа $\{I, R, D\}$, где је I операција убацивања карактера, R операција измене карактера, D операција брисања карактера.

Овај проблем се типично решава методама базираним на динамичком програмирању. Наивно решење, односно решење коришћењем чисте рекурзије је донекле погодније за паралелизацију, међутим тада је временска ефикасност таквог алгоритма $O(3^n)$, где је n дужина дуже секвенце. *Wagner-Fischer* алгоритам, базиран на динамичком програмирању, као и његове модификације, због рекурентне једначине коју користе, нису директно погодне за паралелизацију. Међутим, могуће је одређеним техникама превазићи ове проблеме.

У овом раду биће анализирани различити начини решавања проблема Левенштајновог растојања, као и њихове имплементације.

2 Проблем Левенштајновог растојања

У овом поглављу биће дата формална дефиниција проблема Левенштајновог растојања. Поред тога, биће представљени алгоритми за решавање овог проблема, уз дискутовање импликација у оквиру истих алгоритама у оквиру контекста паралелизације.

2.1 Дефиниција проблема

Левенштајново растојање је метрика за секвенце карактера, која је један од начина да се одреди растојање уређивања (енгл. *edit distance*) две секвенце карактера. Левенштајново растојање две секвенце карактера је одређено минималним бројем операција неопходним да се једна секвенца трансформише у другу, а операције су уметање, брисање или замена једног карактера другим. Добило је име по Владимиру Левенштајну, који га је развио 1965.

Математичка формулација Левенштајновог растојања за две секвенце a, b је:

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + [a_i \neq b_j] \end{cases} \end{cases}$$

Први случај представља почетно, тривијално стање где се пореди секвенца дужине $max(i.j)$ карактера, са секвенцом дужине 0, тада је друга секвенца очигледно удаљена од прве $max(i,j)$ уметања. У другом случају посматра се минимум између операције брисања карактера ($lev_{a,b}(i-1,j)+1$), операције уметања карактера ($lev_{a,b}(i,j-1)+1$), и операције измене карактера ($lev_{a,b}(i-1,j-1)$), где се додаје 1 ако су карактери различити. Овим је овај проблем представљен рекурентном једначином погодном за решавање методама рекурзије, односно динамичког програмирања.

2.1.1 Наивно решење

Наивно решење подразумева коришћење чисте рекурзије која директно имплементира формалну дефиницију проблема. Ова имплементација је поприлично неефикасна превасходно из два разлога. Први разлог јесте то што се врло често дешава поновно израчунавање вредности које су већ претходно израчунате. Други разлог јесте што постоје три рекурзивна позива, што за последицу има $O(3^n)$ временску комплексност алгоритма.

2.1.2 *Wagner-Fischer* алгоритам

Wagner-Fischer алгоритам представља оптимизацију наивног решења техником мемоизације [1]. У овом случају на почетку извршавања алгоритма иницијализује се матрица димензија $(|s1| + 1) \times (|s2| + 1)$ у којој ће се чувати вредности израчунате функције. Овим се избегава потреба за поновним израчунавањем већ израчунатих вредности.

Даље, ову матрицу је могуће секвенцијално попуњавати јер тренутно поље зависи само од поља лево и изнад од тренутног поља. Овај алгоритам има знатно бољу временску комплексност у односу на наивно решење, пошто се извршава у квадратном времену, за разлику од наивног решења које се извршава у експоненцијалном времену.

Меморијска комплексност овог алгоритма је далеко лошија него код наивног решења, јер је потребно алоцирати матрицу димензија $(|s1|+1)\times(|s2|+1)$. Овај проблем је могуће релативно лако превазићи ако није потребно конструисати листу операција које доводе до изједначавања секвенци. Наиме, за израчунавање свих вредности у тренутном реду, потребне су само вредности из претходног реда. Тада, довољно је чувати само два реду у једном циклусу у петљи, што значајно умањује овај проблем.

Овај алгоритам није погодан за паралелизацију. Наиме, израчунавање тренутног поља захтева да су поља лево и изнад од тренутног посматраног поља већ израчуната. Начин на који овај проблем може да се превазиђе јесте уместо обиласка матрице по редовима, да се уради обилазак по дијагоналама. Тада су сва поља од којих тренутно поље зависи већ израчуната у претходне две дијагонале. Ово омогућава паралелизацију израчунавања елемената матрице који се налазе на истој дијагонали. За два елемента $m_{i,j}$ і $m_{p,q}$ важи да се налазе на истој дијагонали ако и само ако важи $i + j = p + q$. Проблем са овим начином паралелизације јесте што је за општу матрицу немогуће добити затворену форму овакве итерације без кондиционих скокова за потребе провере да ли је елемент у границама димензије матрице. Други, и већи проблем, јесте нарушњавање принципа локалности због шаблона приступа меморији. Пошто су матрице секвенцијално чуване у меморији као низ низова, дијагонално приступање има за резултат да се стално приступа различитим блоковима. Ово додатно деградира перформансе.

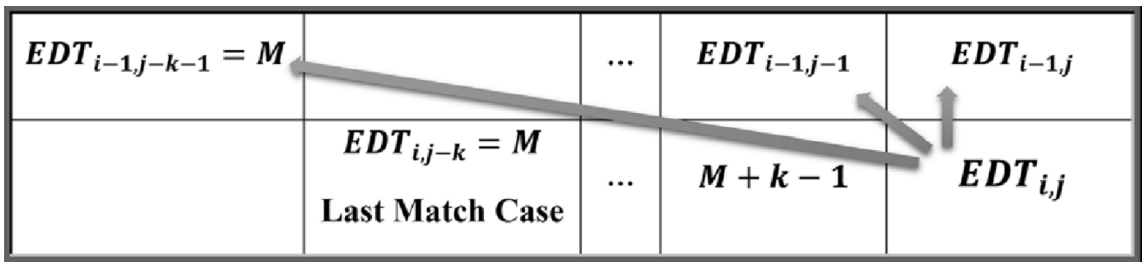
```
function distance(char s[1..m], char t[1..n]):
  declare int d[0..m, 0..n]

  set each element in d to zero

  for i from 1 to m:
      d[i, 0] := i

  for j from 1 to n:
      d[0, j] := j

  for j from 1 to n:
      for i from 1 to m:
          if s[i] = t[j]:
            substitutionCost := 0
          else:
            substitutionCost := 1

          d[i, j] := minimum(d[i-1, j] + 1,
                             d[i, j-1] + 1,
                             d[i-1, j-1] + substitutionCost)

  return d[m, n]
```

Изворни код 1: Псеудокод

2.1.3 Модификација *Wagner-Fischer* алгоритам

Могуће је превазићи претходне поменуте проблеме везане за паралелизацију *Wagner-Fischer* алгоритма. Алгоритам је детаљније описан у [6]. Највећи проблем код оригиналног алгоритма јесте зависност тренутног поља и од тренутног реда, и од тренутне колоне. Суштина модификације алгоритма јесте померање завиност од тренутног реда на искључиво претходни (Слика 1). Ово се постиже конструисањем помоћне матрице, односно табеле, која служи да се одреди удаљеност тренутног од последњег поља где је дошло до поклапања карактера. Ова табела додаје додатну меморијску комплексност, међутим с обзиром да се израчунава само једном, и да је једна димензија ограничена кардиналношћу алфабета краће речи, не долази до превеликог повећања комплексности. Са друге стране, ово омогућава паралелизацију у оквиру једног реда, што је значајно боље од претходно описаног алгоритма где

се паралелицазија манифестовала у оквиру једне дијагонале, јер не нарушава принцип локалности. Такође, расподела је боље балансирана у овом случају, у односу на претходни алгоритам.



Слика 1: Измена зависности на претходни ред

3 Имплементација

У овом поглављу биће описане и дискутоване имплементације претходно представњених алгоритама. Такође, биће упоређени и резултати тестирања ових имплементација.

3.1 Серијска имплементација

Серијска имплементација је урађено у потпуности по дефиницији проблема (Изворни код 3). Имплементација би се могла проширити меморијском оптимизацијом, где би уместо да се алоцира и користи цела матрица, користила само два реда.

```
1  int serial_levenshtein(char *s1, char *s2) {
2    unsigned int i, x, y, s1len, s2len, j;
3    s1len = strlen(s1);
4    s2len = strlen(s2);
5
6    unsigned int **matrix = malloc((s2len + 1) * sizeof(unsigned int *));
7    for (i = 0; i < s2len + 1; i++)
8      matrix[i] = malloc((s1len + 1) * sizeof(unsigned int));
9
10   matrix[0][0] = 0;
11   for (x = 1; x <= s2len; x++)
12     matrix[x][0] = matrix[x - 1][0] + 1;
13   for (y = 1; y <= s1len; y++)
14     matrix[0][y] = matrix[0][y - 1] + 1;
15   for (x = 1; x <= s2len; x++)
16     for (y = 1; y <= s1len; y++)
17       matrix[x][y] =
18           MIN(matrix[x - 1][y] + 1, matrix[x][y - 1] + 1,
19               matrix[x - 1][y - 1] + (s1[y - 1] == s2[x - 1] ? 0 : 1));
20
21   unsigned int return_val = matrix[s2len][s1len];
22   deallocate_matrix(matrix, s2len + 1);
23   return return_val;
24 }
```

Изворни код 2: Серијска имплементација

3.2 Имплементација паралелизације по дијагонали

Кључни део имплементације овог алгоритма дат је у коду испод (Изворни код 3). Спољашња петља представља итерацију по вредностима које су мање од збира вредности димензија матрице. Као што је раније поменуто претходном поглављу, за два елемента матрице важи да припадају истој дијагонали акко им је збир индекса реда и колоне међусобно једнак. Даље, унутрашња петља итерира по свим могућим дво-елементним партицијама вредности итератора спољашње петље. У линији 9. (Изворни код 3) се проверавају ограничења димензија, и петља се наставља у случају да су тренутни вредности ван опсега.

Приметимо да у општем случају није могуће ову итерацију записати у затвореној форми без кондиционог дела. У случају да можемо да претпоставимо да је једна димензија матрице увек мања од друге, нпр. да је број редова увек мањи од броја колона, ова итерација би могла другачије да се запише.

```
1  int parallel_diagonal_levenshtein(char *s1, char *s2) {
2    for (i_x = 2; i_x <= s1len + s2len; i_x++) {
3  #pragma omp parallel for firstprivate(i_x, x, y) shared(matrix)
4
5      num_threads(NUM_THREADS)
6      for (i_y = i_x - 1; i_y >= 1; i_y--) {
7        y = i_x - i_y;
8        x = i_y;
9        if ((x > s2len) || (y > s1len)) {
10         continue;
11       }
12       matrix[x][y] =
13           MIN(matrix[x - 1][y] + 1, matrix[x][y - 1] + 1,
14               matrix[x - 1][y - 1] + ((s1[y - 1] == s2[x - 1]) ? 0 : 1)
15     }
16   }
17
18   unsigned int return_val = matrix[s2len][s1len];
19   deallocate_matrix(matrix, s2len + 1);
20   return return_val;
21 }
```

Изворни код 3: Имплементација паралелизације по дијагонали

3.3 Паралелна имплементација модификованог алгоритма са помоћном табелом

Основу ове имплементације чини помоћна матрица којом се уклања зависност од тренутног реда. Алфабет речи за коју се конструише је претходно евалуиран, и након тога се попуњава помоћна матрица. Приметимо да се овде може паралелизовати спољашња петља из разлога што је за потребе рачунања тренутног елемента потребна само претходна колона.

```
1  int parallelised_friendly_algorithm(char *s1, char *s2) {
2  #pragma omp parallel for private(j) shared(s2len, mi, s2, alphabet)
3    for (i = 0; i < u; i++) {
4      for (j = 0; j < s2len; j++) {
5        if (j == 0) {
6          mi[i][j] = (s2[j] == alphabet[i]) ? 0 : -1;
7        } else if (s2[j] == alphabet[i]) {
8          mi[i][j] = j;
9        } else {
10         mi[i][j] = mi[i][j - 1];
11       }
12     }
13   }
```

Изворни код 4: Конструисање помоћне матрице

Тривијални случајеви, односно индекс колоне или реда имају вредност 0 се могу паралелизовати, што је и урађено у линијама 1. и 5. (Изворни код 5) респективно. Након тога у главном делу алгортима може се урадити паралелизација унутрашње петље, тј. један ред се може паралелно израчунавати. У линији 13 (Изворни код 5). се преко помоћне матрице добија вредности индекс где је последњи пут дошло до поклапања карактера [5]. Потом, у линији 19. (Изворни код 5) се та вредност употребљава да би се зависност померила на претходни ред, овде би у оригиналној верзији алгоритма била коришћена и i-та колона, односно тренутна колона, што би спречавало паралелизацију. У односу на алгоритам који врши итерације по дијагонали и тиме нарушава принцип локалности, овај алгоритам користи принцип локалности.

```
1  #pragma omp parallel for
2    for (j = 0; j < s2len + 1; j++) {
3      matrix[0][j] = j;
4    }
5  #pragma omp parallel for
6    for (i = 0; i <= s1len; i++) {
7      matrix[i][0] = i;
8    }
9    for (i = 1; i <= s1len; i++) {
10     int a = index_of_letter(s1[i - 1], alphabet);
11 #pragma omp parallel for shared(i, s2len, mi) private(lmi)
12     for (j = 1; j <= s2len; j++) {
13       lmi = mi[a][j - 1] + 1;
14       if (j == lmi) {
15         matrix[i][j] = matrix[i - 1][j - 1];
16       } else if ((lmi == -1) || (lmi == 0)) {
17         matrix[i][j] = MIN(matrix[i - 1][j - 1] + 1, matrix[i - 1][j] +
18       } else if (j > lmi) {
19         matrix[i][j] = MIN(matrix[i - 1][j - 1] + 1,
20                            matrix[i - 1][j] + 1,
21                            matrix[i - 1][lmi - 1] + (j - lmi));
22       }
23     }
24   }
25 }
```

Изворни код 5: Паралелна имплементација модификованог алгоритма са помоћном табелом

3.4 Резултати

Спецификације система на ком је вршено тестирање су:

- AMD Ryzen 7 3700x 8C16T.
- 32 GB RAM DDR4 3600MHz.
- Intel 660p
- Arch Linux x86_64
- gcc 12.2

Тестирани алгоритми су:

- Серијски - С.
- Итерација по дијагоналама, серијска имплементација - ДС
- Итерација по дијагоналама, паралелна имплементација - ДП
- Алгоритам са помоћном матрицом, серијска имплементација - МС
- Алгоритам са помоћном матрицом, паралелна имплементација - МП

Колоне представљају број нити, док редови представљају евалуирани алгоритам. Као што можемо приметити на основу представљених резултата, до знатног побољшања перформани долази тек са матрицама великих димензија (Табела 1). Такође, алгоритам који користи итерацију по дијагоналама константно даје лошије резултате, који имају тенденацију деградирања са повећањем броја нити. Потом, примећује се да је перфоранса алгоритма са помоћном матрицом далеко бољи када матрица има много већи број колона него редова, упоредимо Табелу 3. и Табелу 4. Са мањим димензијама матрице не долази до побољшања, шта више серијска имплементација оригиналног алгоритма даје најбоље резултате (Табела 5).

	С	ДС	ДП	МС	МП
2	0.030697	485.045237	588.540534	0.002287	0.000538
4	0.027455	484.818786	607.215918	0.002236	0.000287
8	0.026827	485.185366	652.106079	0.002295	0.000170
16	0.027921	499.975804	1144.832312	0.002313	0.000162

Табела 1: Резултати за димензије матрице 800000×800000

	С	ДС	ДП	МС	МП
2	0.026039	475.972798	545.148893	0.001863	0.000539
4	0.028138	511.520019	593.131504	0.002254	0.000306
8	0.033595	514.150049	804.537528	0.002290	0.000388
16	0.030363	508.214492	3898.244563	0.002365	0.000187

Табела 2: Резултати за димензије матрице 80000×800000

	С	ДС	ДП	МС	МП
2	0.062948	8.258610	9.162211	0.151563	0.363156
4	0.065733	8.745489	10.333694	0.167686	0.708155
8	0.064834	8.565872	11.045861	0.188561	1.505345
16	0.063208	8.506860	19.556950	0.218408	4.722902

Табела 3: Резултати за димензије матрице 100000×100

	С	ДС	ДП	МС	МП
2	0.003189	8.282298	9.626102	0.000278	0.000057
4	0.003211	6.889069	9.208704	0.010274	0.000033
8	0.003323	7.699393	10.988810	0.000290	0.000022
16	0.005310	7.075573	18.172686	0.002310	0.000023

Табела 4: Резултати за димензије матрице 100×100000

	С	ДС	ДП	МС	МП
2	0.000385	0.069242	0.091012	0.000029	0.000009
4	0.000702	0.069531	0.138217	0.000029	0.000008
8	0.000363	0.068156	0.209607	0.000029	0.000008
16	0.000369	0.068393	0.577592	0.000029	0.000013

Табела 5: Резултати за димензије матрице 10000×10000

4 Закључак

У овом раду представљен је проблем Левенштајновог растојања, као и алгоритми, односно технике и приступи за решавање истог. Фокус рада био је на употреби паралелизације и динамичког програмирања, са одређеним помоћним техникама за потребе решавања овог проблема.

Иако сам проблем није превише погодан за паралелизацију, представљен је начин на који то може да се превазиђе уз минимално додавање комплексности. Представљени су резултати тестирања имплементације и урађена је компаративна анализа истих. Даљи рад би свакако могао да укључи имплементирање додатних оптимизација, које су и биле споменте у оквиру анализа алгоритама и кодова у претходним поглављима. Осим тога, постоји још приступа који би могли да се употребе, попут нпр. коришћења SIMD [4].

Комплетна имплементација која садржи претходно анализирни код, са помоћним скриптама за тестирање и покретање је јавно достуна на GitHub-у [3].

Библиографија

[1] Dynamic programming. https://en.wikipedia.org/wiki/Dynamic_programming. Accessed: 2023-01-14.

[2] Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance. Accessed: 2023-01-13.

[3] Parallel edit distance. https://github.com/dovvla/parallel-edit-distance. Accessed: 2023-01-17.

[4] Parallel edit distance simd. https://turnerj.com/blog/levenshtein-distance-with-simd. Accessed: 2023-01-17.

[5] Muhammad Umair Sadiq, Muhammad Murtaza Yousaf, Laeeq Aslam, Muhammad Aleem, Shahzad Sarwar, and Syed Waqar Jaffry. Nvpd: novel parallel edit distance algorithm, correctness, and performance evaluation. *Cluster Computing*, 23(2):879--894, 2020.

[6] Muhammad Murtaza Yousaf, Muhammad Athar Sadiq, Laeeq Aslam, Waqar Ul Qounain, and Shahzad Sarwar. A novel parallel algorithm for edit distance computation. *Mehran University Research Journal of Engineering & Technology*, 37(1):223--232, 2018.