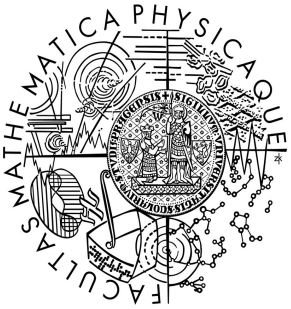
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Josef Pihera

Archivace dat užitím nejdelší společné podposloupnosti

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Pavel Surynek, Ph.D.
Studijní program: Informatika, programování

2010

Tímto bych chtěl poděkovat svému vedoucímu bakalářské práce za všechny cenné rady a připomínky.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 24.5.2010 Josef Pihera

Obsah

1 Úvod **6**

1.1 Archivace redundantních dat....7

1.2 Cíl této práce....7

1.3 Struktura textu....8

2 Nejdelší společná podposloupnost **9**

2.1 Značení....9

2.2 Dynamické programování – základní algoritmus....9

2.3 Páry shody....10

2.4 Dominantní páry shody....12

2.5 Algoritmus Hunta a Szymanskiho....13

2.6 Algoritmus Kua a Crosse....16

2.7 Možnosti implementace, sestavení LCS....18

3 LCS a návrh nové metody **19**

3.1 Společné podřetězce a podposloupnosti....19

3.2 Sufixové pole....20

3.3 Užití nejdelšího společného podřetězce....22

3.4 Návrh prvního algoritmu....23

3.5 Návrh druhého algoritmu....26

4 Implementace algoritmů **32**

4.1 Použitá platforma....32

4.2 Heuristika BestNext....33

4.3 Heuristika založená na Rabin-Karpově algoritmu....33

4.4 Základní dynamický algoritmus – Simple....34

4.5 Vylepšený dynamický algoritmus – Enhanced....34

4.6 HuntSzymanski....35

4.7 KuoCross....35

4.8 První návrh – SA....36

4.9 První návrh a zásobník – ESA....37

4.10 Druhý návrh algoritmu – SAD....37

5 Architektura programu **38**

5.1 První část – LCS-FTP....38
5.2 Druhá část – LCS-FTP-Server....40
6 Uživatelská dokumentace programu 43
6.1 Serverová část s GUI....43
6.2 Ovládání serveru přes rozšířené FTP....44
7 Testování 47
7.1 Testovací vstupy....47
7.2 Naměřené výsledky....49
7.3 Srovnání....51
8 Závěr 55
9 Příloha 56
Literatura 57

Název práce: Archivace dat užitím nejdelší společné podposloupnosti
Autor: Josef Pihera
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: RNDr. Pavel Surynek, Ph.D.
E-mail vedoucího: surynek@ktiml.mff.cuni.cz

Abstrakt: Práce zkoumá existující algoritmy pro řešení problému nejdelší společné podposloupnosti. Snaží se je aplikovat při řešení problému efektivního ukládání dat, která se velmi málo liší. Například se jedná o verze jednoho souboru. Rovněž jsou zde navrženy nové metody vycházející ze zkoumaných algoritmů. Algoritmy jsou pak srovnávány podle dosažených výsledků a rychlosti běhu.

Klíčová slova: nejdelší společná podposloupnost, LCS, archivace, algoritmy, srovnání

Title: Data archiving using longest common subsequence
Author: Josef Pihera
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: RNDr. Pavel Surynek, Ph.D.
Supervisor's e-mail: surynek@ktiml.mff.cuni.cz

Abstract: This thesis examines existing algorithms for solving the problem of the longest common subsequence. It tries to apply them to the problem of effective archiving of data which differ a little, because they are, for example, just versions of one file. There are also new methods suggested. Those are based on the examined algorithms. Then the algorithms are compared according to the reached results and their running time.

Keywords: longest common subsequence, archiving, algorithm, comparison

1 Úvod

Vyspělost hardwarových prostředků se s postupem času podstatně zlepšuje. To umožňuje zpracovávat větší objemy dat, ale zároveň přináší důležitou otázku – jak tato data efektivně ukládat a zálohovat. I v tomto směru se hardware snaží nabídnout různá řešení, která zpravidla nabízejí úložnou kapacitu mnohonásobně převyšující prostředky dostupné před několika lety. Běžně se tak setkáváme s údaji v řádech terabytů. Ale ačkoliv jsou taková zařízení již relativně dostupná, je nutné se ptát, zda neukládáme data v přílišně redundantní podobě, což přináší zbytečné náklady.

Často potřebujeme vědět, jak vypadal soubor, se kterým pracujeme, před či po nějakých úpravách. Je ale nevýhodné si každou takovou verzi souboru ukládat zvlášť. Nezřídka vyžadujeme uchovávat data, která se s časem jen velmi málo mění. V tomto směru jsou typické podnikové databáze a postupně upravované a doplňované dokumenty. Obecně sem pak náleží i obsah uchovávaný v systémech pro správu verzí jako jsou například CVS, GIT a další.

Výkon moderních procesorů dovoluje implementovat řešení, o nichž se před časem soudilo, že jsou příliš pomalá a v praxi nepoužitelná. Velikost operační paměti umožňuje používat datové struktury, jejichž potenciál býval spíše teoretický. Je tedy nasnadě hledat softwarový nástroj umožnující zbavit se zbytečné redundance.

Pokud se omezíme jen na úroveň jednotlivých souborů, měnících se od verze k verzi řádově v jen relativně malém zlomku svého obsahu, je zřejmé, že ukládat každou z těchto verzí odděleně není efektivní. Mnoho identických dat se tak ukládá vícenásobně a nejspíše zbytečně. Ovšem ani komprimace těchto dat v jednom archivu se nezdá být hledanou odpovědí. Doplňování komprimovaných archivů o nové soubory vyžaduje ještě stále nepřiměřeně mnoho času. To platí za předpokladu, že má být vhodně zohledněna právě ona problematická redundance. Je totiž obvykle nutné znovu komprimovat celý obsah archivu.

1.1 Archivace redundantních dat

Zaměřme se na případ, kdy chceme ukládat posloupnost verzí jednoho souboru, přesně jak se vyvíjel v čase. Lze předpokládat, že nejvíce podobné (ač termín podobnosti zatím použijeme jen v běžném slova smyslu) si budou vždy dvě přímo po sobě následující verze. Můžeme tedy redukovat problém na zjištění a využití podobnosti právě takových dvojic verzí souboru. Na každou verzi souboru budeme nazírat jako na nějakou vhodnou výchozí verzi, na kterou jsou postupně aplikovány změny.

Problém podobnosti dvou posloupností dat informatika již jistý čas poměrně intenzivně studuje. Jedním ze základních používaných měřítek je takzvaná editační vzdálenost. Tento pojem je poněkud širší, ale dobře postačí omezit se například jen na Levenshteinovu editační vzdálenost. Určení editační vzdálenosti v tomto smyslu je velmi těsně spojeno s problémem hledání nejdelší společné podposloupnosti, obvykle označované anglickou zkratkou LCS (Longest Common Subsequence).

Pro nalezení nejdelší společné podposloupnosti existuje řada algoritmů, z nichž některé jsou známé již bezmála třicet let. Jejich rychlost a paměťová náročnost se často asymptoticky liší jen málo, ale v praxi jsou rozdíly značné. Nicméně stále nelze tyto algoritmy přímočaře použít pro vyhodnocení podobnosti dvou souborů s větším objemem, ač v dnešní době je takový objem již běžný.

Abychom tedy nalezli nějaké řešení pro ukládání redundantních dat, je důležité najít způsob jak využít algoritmy a postupy známé pro LCS. Další součástí řešení je ovšem i způsob popisování změn mezi verzemi souborů, protože i tuto informaci je nutné ukládat. Její velikost a vlastnosti rovněž vytvářejí požadavky na způsob, jakým mají algoritmy fungovat.

1.2 Cíl této práce

Cílem této práce je srovnat vybrané existující algoritmy a pokusit se navrhnout vlastní algoritmus, a to v podmínkách, které by se blížily reálnému prostředí. Tam je nakonec hlavní smysl jejich používání. Použité algoritmy nepředpokládají žádnou speciální znalost formátů souborů, ale pracují s nimi pouze na úrovni jednotlivých bytů.

Algoritmy jsou implementovány s jistými úpravami, které dovolují jejich aplikaci i na větší soubory. Celý projekt je realizován podobně jako FTP server, ale je rozšířen o potřebnou funkčnost. Soubory jsou ukládány ve virtuálním souborovém systému a lze je na server nahrávat, mazat je a stahovat. Název tohoto serverového rozhraní je LCS-FTP-Server. Rozšířený protokol dovoluje změnit používané algoritmy a pro zpracování souborů nabízí dvě funkce: jednak samotné efektivní uložení, ale i odhad podobnosti a vhodnosti souborů pro takové ukládání.

Všechny implementované algoritmy jsou testovány na předem vybrané sadě vstupních dvojic souborů, u kterých lze rozumně očekávat významnou míru podobnosti. Zohledněn je pak čas potřebný na zpracování, ale i dosažená efektivita uložení.

LCS-FTP-Server disponuje možností ukládání celé posloupnosti jednotlivých verzí souborů, nikoliv jenom dvojic stará – nová verze. Jsou tak lépe zohledněny faktory reálného prostředí, ačkoliv samotné srovnání algoritmů pak této možnosti nepoužívá.

1.3 Struktura textu

V následujícím textu se budeme nejprve věnovat obecně problému nejdelší společné podposloupnosti, dále pak algoritmům a postupům používaných pro jeho řešení. Na tuto část navážeme návrhem nové vlastní metody. Následuje popis implementace algoritmů a v další kapitole popis celého programu. Dále je objasněno ovládání aplikace LCS-FTP-Server uživateli. Poslední kapitolou jsou pak získané výsledky srovnání algoritmů na testovacích datech, vyvození závěrů a návrhy dalších možných postupů pro tento problém.

2 Nejdelší společná podposloupnost

Problém hledání nejdelší společné podposloupnosti je již dlouhou dobu známý a studovaný. Existuje několik různých článků, které se touto problematikou zabývají a v zásadě se setkáváme se dvěma základními přístupy k řešení. Prvním způsobem je využití dynamického programování vzhledem k rekurzivnímu chování problému, druhým způsobem je hledání párů (v originále tzv. matches), případně dominantních párů (dominant matches).

2.1 Značení

Nejprve si zaveďme vhodné značení a problém pomocí něho formulujme. Omezíme se na problém společné podposloupnosti pro dvě dané konečné posloupnosti obsahující symboly z konečné abecedy. Mějme tedy konečnou abecedu Σ. Posloupnosti budeme nadále označovat i jako řetězce. Řetězcem A tedy budeme rozumět A: $a_1,a_2,\ldots a_{|A|}$, kde pro každé $a_i \in \Sigma$, a $|A|$ značí délku posloupnosti A. Vybranou posloupností (resp. podposloupností) X z A pak označme $x_1 = a_{i1}$, $x_2 = a_{i2}$, ... $x_{|X|} = a_{i|X|}$, kde $0 < i_1 < i_2 < i_3 < \ldots < i_{|X|} \leq |A|$. Mějme dva řetězce A a B. Říkáme, že X je jejich společná podposloupnost, pokud X je podposloupností A i B zároveň. Za LCS A B (tedy nejdelší společnou podposloupnost A a B) pak považujeme společnou podposloupnost takovou, že neexistuje žádná delší s touto vlastností. Všimněme si, že tato definice nevylučuje existenci více různých LCS. Za LLCS A B (Length of Longest Common Subsequence) pak označujeme |LCS A B|. Ještě zaveďme pro pohodlí značení $A_{i..j} = a_i, a_{i+1}, \ldots a_j$ a říkejme, že $A_{i..j}$ je podřetězec A. Pokud je $j < i$, považujme podřetězec za prázdný. Dále ještě definujme zřetězení $A + B = a_1, .. a_{|A|}, b_1, .. b_{|B|}$.

2.2 Dynamické programování – základní algoritmus

V jistém směru se LCS chová rekurzivně. Podívejme se, co to ve skutečnosti znamená. Mějme řetězce A a B. Pak zřejmě platí, že je-li $A_{|A|} = B_{|B|}$, pak LCS A B = (LCS $A_{1..|A|-1}$ $B_{1..|B-1|}$) + $a_{|A|}$. Pokud se však řetězce v posledním znaku neshodují, je LCS A B delší z LCS $A_{1..|A|-1}$ $B_{1..|B|}$, LCS $A_{1..|A|}$ $B_{1..|B|-1}$. Tato tvrzení lze snadno vypozorovat, zdůvodnění správnosti však lze najít v Hirschbergově článku [1].

Zde nalezneme popsaný asi nejznámější algoritmus pro hledání LCS v čase O(|*A*| * |*B*|) a s paměťovou složitostí O(|*A*| * |*B*|).

Jeho podstatou je vytvoření matice *C* s políčky 0..|*A*| × 0..|*B*|, přičemž prvek na pozici c_{ij} obsahuje výsledek LLCS $A_{1..i}$ $B_{1..j}$. Prvky v nultém sloupci a řádku jsou zřejmě nulové a ostatní získáme postupnou aplikací rekurzivního pravidla. Zřejmě při výpočtu můžeme postupovat po řádcích od nejkratšího prefixu *A* po nejdelší prefix *A*. Pokud nám jako výsledek postačí LLCS, nemusíme si ukládat matici *C* celou, ale pouze poslední dva řádky, se kterými pracujeme.

Hirschberg ve svém článku uvádí myšlenku, která dovoluje problém vyřešit s lineární paměťovou složitostí (LCS, nikoliv jen LLCS, kde jsme to triviálně nahlédli) za cenu konstantního zhoršení časové složitosti [1]. Toto vylepšení však v praxi vykazovalo nezanedbatelné zpomalení běhu algoritmu, měřeno v reálném čase. V pilotních implementacích se nevyplatilo oproti vyšším paměťovým nárokům. Proto není tato metoda zahrnuta v implementovaných algoritmech.

Pseudokód algoritmu 2.2.:

```
procedure DynamicLCS(A, B : Sequence; C : Matrix)
Begin
  C[0, *] = C[*, 0] = 0

  For i in 1..|A| Do
    For j in 1..|B| Do
      val := Max(C[i - 1, j], C[i, j - 1])
      If A[i] = B[j] And C[i - 1, j - 1] + 1 > val Then
        val := C[i - 1, j - 1] + 1
      C[i, j] := val
End
```

2.3 Páry shody

Při bližším zkoumání použití techniky dynamického programování zjistíme, že algoritmus provede spoustu zbytečných výpočtů, a sice na těch políčkách matice *C*, kde se LCS neprodlouží, tj. těch, kde se neshodnou uvažované podřetězce ve svém posledním prvku. Takový výpočet pak pouze kopíruje hodnoty problému, kdy byl nějaký z podřetězců o jeden znak kratší na svém konci. Pokud bychom si detailně vypsali obsah matice, zjistili bychom, že se hodnoty LLCS pro prefixy řetězců shlukují

do jakýchsi vrstev. Nejedná se o náhodné chování, ale o pečlivě studovanou vlastnost, která dovolila vzniknout dalším algoritmům.

	⊥	a	b	c	b	a	b
⊥	0	0	0	0	0	0	0
a	0	**1**	1	1	1	1	1
b	0	1	**2**	2	2	2	2
a	0	1	2	2	2	**3**	3
a	0	1	2	2	2	3	3
c	0	1	2	**3**	3	3	3
b	0	1	2	3	**4**	4	4

Tabulka 2.3: Matice *C* pro LLCS „abaacb“ „abcbab“. Tučně jsou pak vyznačeny dominantní páry.

Přidejme si ještě označení (i, j) pro pár shody na posloupnostech A, B - tj. tvrdíme, že $A_i = B_j$.

Pozastavme se nad tím, jak moc „vrstevnatě“ se hodnoty chovají. Za nejdůležitější vlastnost lze považovat to, že spolu sousedí vždy pouze vrstvy, jejichž hodnoty se liší právě o jedna. Dalším významným pozorováním je, že ve směru, kam se prodlužují prefixy řetězců, hodnoty neklesají, což si snadno ověříme. Bez újmy na obecnosti, vyberme dva sousedící prvky $c_{i,j}$ a $c_{i+1,j}$ a označme rozdíl jejich hodnot k ($k = c_{i+1,j} - c_{i,j}$). Nejprve si všimněme, že k je určitě nezáporné (s přidáním písmenka za posloupnost se nám nemůže LCS zkrátit). Buď tedy k alespoň 2. To pak znamená, že přidáním jednoho písmene a_{i+1} se LCS $A_{1..i+1}$ $B_{1..j}$ prodlouží o dva znaky. Znak a_{i+1} je buď použit v nějaké X = (LCS $A_{1..i+1}$ $B_{1..j}$) nebo není. Pokud není, pak X je zároveň i (LCS $A_{1..i}$ $B_{1..j}$) a k musí být 0. V opačném případě se podíváme na prefix X, který vznikne oddělením posledního písmene. To ale musí být společná podposloupnost $A_{1..i}$ a $B_{1..j}$, což je ovšem ve sporu s tím, že (LCS $A_{1..i}$ $B_{1..j}$) je alespoň o dva znaky kratší (tedy, že k je alespoň 2).

Rozdělme tedy políčka matice na třídy podle ekvivalence jejich hodnot (definujeme, co zatím popisoval vágní pojem „vrstvy“). Tj. $c_{i,j}$ leží ve stejné třídě jako $c_{p,q}$ pokud $c_{i,j} = c_{p,q}$.

Další zvláštností shodujících se párů je, že některé přispívají ke zvětšení LLCS a jiné

ne. Myslí se tím jev, kdy by bylo možno nějaké hodnoty LLCS dosáhnout již předtím (čteme-li matici C shora a je-li matice orientována jako v tabulce 2.3), i kdybychom daný shodující se pár neměli k dispozici: Příklad nalezneme v tabulce 2.3 v řádku s prvním „b“, kde hodnoty 2 se dosáhne již pro prefixy „ab“ „ab“, a tedy prefixy „ab“ „abcb“ tuto hodnotu nezvětší, ačkoliv A_2 a B_4 tvoří pár shody.

2.4 Dominantní páry shody

Odlišíme tedy ty páry shody, které přispívají k prodloužení LCS, a označíme je jako dominantní páry, na které – jak brzy ukážeme – se při hledání řešení stačí zaměřit. Dominantní pár je takový pár shody (i, j), že s ním ve stejné třídě neleží žádný takový, který by byl v obou složkách menší nebo roven (i, j). De facto jsme vybrali ty páry shody, které leží na rozhraní tříd a to pouze v „konvexních rozích“, tj. žádný pár stejné hodnoty není vlevo nahoře od nich, pokud si matici nakreslíme a zorientujeme tak, jak je ukázáno v tabulce 2.3.

Je dobře známo, že k vyřešení LCS nám stačí hledat takové podposloupnosti, jež prochází pouze přes dominantní páry. Naznačíme, proč tomu tak je. Vezměme si nějakou X = LCS a nyní se pokusme zarovnat indexy, ze kterých vybírá X písmenka tak, aby odpovídala dominantním párům. Budeme postupovat indukcí od kratších prefixů $A\ B$ po delší. Buď nejprve LCS prázdná, to pak vidíme pravdivost tvrzení triviálně. Nechť umíme nalézt LCS jdoucí výlučně přes dominantní páry pro všechny prefixy $A_{1..i}$, $B_{1..j}$ $i \leq m, j \leq n$. Bez újmy na obecnosti pak zvětšujme m o jedna (v dalším kroku bychom analogicky udělali to samé pro n). Mějme tedy nějaké $p = m + 1$, $q \leq n$ a hledejme LCS $A_{1..p}$, $B_{1..q}$, která bude využívat pouze dominantních párů. Nechť poslední symbol této LCS využívá nějaký pár $(m + 1, d)$, kde d je libovolné. Pokud je $(m + 1, d)$ dominantní pár, pak máme téměř hotovo, protože zbytek LCS už využívá pouze tu část matice s prefixy $A_{1..m}$ $B_{1..n}$, kde můžeme odpovídající úsek LCS vyměnit za nějaký, který je složen pouze z dominantních párů, a to díky indukčnímu předpokladu. Pokud
$(m + 1, d)$ není dominantní pár, jistě k němu existuje nějaký dominantní pár ve stejné třídě, který můžeme použít. Zaručuje to definice, podle níž existuje pár shody ve stejné třídě s oběma složkami menšími nebo rovnými $(m + 1, d)$. V tento moment použijeme podobných úvah jako v předchozím případě, a tím máme LCS zarovnanou a tvrzení dokázané.

2.5 Algoritmus Hunta a Szymanskiho

Hledání LCS pomocí párů a dominantních párů je celkem aktivně používáno a jedním z algoritmů, který tento přístup využívá je algoritmus Hunta a Szymanskiho (značme dále jako HS) [2]. Představme si, že nyní nebudeme hledat hodnoty v matici C, ale pro každý řádek si udržujme číslo sloupce, kde začíná která vrstva (můžeme se na to dívat také takto: jaký nejkratší prefix B potřebujeme k tomu, abychom nalezli LCS dané délky s daným prefixem A). Předem uveďme, že pokud známe tyto hodnoty pro řádek i, umíme je spočítat pro řádek $i + 1$. Tedy hodnoty si nemusíme pamatovat pro každý řádek, pokud budeme chtít spočítat pouze LLCS, protože nám bude stačit udržovat hodnoty pro ten poslední zpracovávaný a postupovat shora. Zřejmě jsou hodnoty v matici omezeny délkou kratšího z řetězců, takže si ve skutečnosti pamatujeme nanejvýš tolik hodnot jako v matici C.

Tyto hranice se posouvají směrem z nekonečna (kdy ještě žádná hranice není) k nule. Hranice třídy k se však neposune blíže k nule než do sloupce k (toto jsme nahlédli vzhledem k tomu, že spolu sousedí vždy pouze hodnoty o 1 se lišící a v nejlevějším sloupci matice C je vždy 0). Z technických důvodů místo nekonečna budeme používat pouze nějakou dostatečně velkou hodnotu. Položme ji třeba jako $|B| + 1$, protože nejvyšší index sloupce, který zřejmě můžeme v LCS použít je $|B|$.

Zdůrazněme, že hranice se zřejmě posouvají jenom doleva, protože LCS se nemůže se zvětšující délkou prefixů A zkracovat.

Uvažujeme-li, že uložené hranice jsou jen jakousi komprimovanou verzí původní matice C, snadno si všimneme jedné věci: Protože se LLCS v případě shody zvětšuje o jedna jen vzhledem k hodnotě vlevo nahoře nad ní (podle pravidla v kapitole 2.2), nikdy se žádná hranice neposune více doleva, než do sloupce napravo přímo od hranice pro třídu o jedna menší. Tj. pokud třída j na řádku i začíná sloupcem $T_{i,j}$, bude třída $j + 1$ na řádku $i + 1$ začínat na pozici $T_{i+1,j+1} > T_{i,j}$.

Nyní se zaměříme na to, jak algoritmus přímo funguje. Mějme nějaké pole prahů $T_{i,j}$ (z anglického threshold), kde $T_{i,j}$ říkají, v jakém sloupci začíná třída j na řádku i ($0 \leq i \leq |A|$, $0 \leq j \leq |B|$). Pro třídu 0 jsou zřejmě všechny $T_{i,0}$ nulové, dále $T_{0,j} = |B| + 1$ (pro $j > 0$). Popíšeme, jak se z T_i spočítá T_{i+1}. Mějme na paměti, že hledáme dominantní

páry, které posouvají hranice. Postupně tedy pro všechna j proveďme to, že vyhledáme, zda existuje pár shody v rozmezí $(i + 1, T_{i,j} + 1)$ … $(i + 1, T_{i,j+1} - 1)$. Těchto párů ovšem může existovat více. Důvod, proč jsme se omezili na toto rozmezí sloupců, uvádíme v předchozím odstavci.

Hunt a Szymanski počítali polohy hranic tříd směrem od tříd s větší hodnotou po třídy s hodnotou menší. Algoritmus stavěl na jednoduchém faktu, že s každým nalezeným párem shody postačí pouze vzít hranici třídy, do které pár patří, a hranici posunout k nalezenému páru. Díky sestupnému pořadí zkoumaných párů shody v rámci sloupce poslední změna nějaké hranice třídy odpovídá nalezení dominantního páru. Algoritmus se tedy přímo nesnaží nacházet dominantní páry, ale využívá jejich vlastností. Jakmile je dominantní pár na řádku pro danou třídu j nalezen, nemůže se již poloha hranice změnit, protože další pár shody bude posouvat třídu s hodnotou nejvýše j - 1.

Pokud bychom nebrali páry shody v sestupném pořadí, algoritmus by nefungoval, což je fakt, na který již ve svém článku Hunt a Szymanski upozorňují [2].

Pseudokód algoritmu 2.5.:

```
procedure HuntSzymanski(A, B: Sequence; T : Matrix)
Begin

  {T[i, j] = hranice třídy j na řádku i}
  T[0, *] := |B| + 1
  T[*, 0] := 0

  For j in 1 .. |B| Do
   Matches[ B[j] ].Append(j) {Seznamy výskytů symbolů}

  For X in Alphabet Do
    Matches[ X ].reverse() {Otočit všechny seznamy}

  MaxClass := 1

  For i in 1 .. |A| Do

    LastClass := MaxClass

    T[i, *] = T[i - 1, *] {Tento řádek není nutný - ilustruje vznik pole}
                        {T, které není celé potřeba, stačí poslední řádek}

    For j in Matches[ A[i] ] Do {Pro kazdy index vyskytu znaku A[i] v B}

       {Najdi třídu, kterou posuneme}
       While T[i - 1, LastClass - 1] >= j Do
        LastClass--

       {Změníme maximální dosažitelnou hranici}
       If LastClass = MaxClass Then
        MaxClass := LastClass + 1

       {Posuneme hranici}
       T[i, LastClass] := j

       SAVE_MATCH(i, j) {Nalezli jsme pár shody}
    {EndFor}
  {EndFor}
End
```

2.6 Algoritmus Kua a Crosse

Vraťme se k momentu, kdy hledáme páry shody v rozmezí $(i + 1, T_{i,j} + 1) \ldots (i + 1, T_{i,j+1} - 1)$. Pokud je párů více, vezmeme nejmenší. Označme jej $(i + 1, p)$. Tento pár prodlužuje nějakou LCS $A_{1..i}\ B_{1..T[i,\ j]}$ délky j, a zároveň je dominantní. Tento fakt jsme zmínili výše a rovněž, pokud si uvědomíme význam hodnot v T, je snadno vidět, že žádný pár shody nemůže ležet ve stejné třídě jako právě nalezený a být buď nad ním, nebo nalevo od něho. Zároveň tedy tento pár jistě posunul hranici třídy $j + 1$, tj. $T_{i+1,j+1} = p$.

Kuo a Cross přišli se zlepšením původní myšlenky Hunta a Szymanskiho [3]. Tento algoritmus bude v tomto textu nadále značen jako KC.

Zřejmě není nutné posouvat hranici k nedominantním párům shody, protože tato hodnota se následně změní. Algoritmus nakonec nepostupuje od posledního sloupce k prvnímu. Místo toho se vypořádává s problémem, který je v HS zmíněn pro hledání párů shody v rostoucím pořadí.

Řešení lze považovat za triviální, když si uvědomíme následující fakt: Byla-li původní hodnota $T_{i,j}$ zmenšena na $T'_{i,j}$, pak všechny následující páry shody, které leží v intervalu $<T'_{i,j}, T_{i,j}>$, určitě nejsou dominantní. V HS by byl jejich efekt zřejmě přepsán.

Tento algoritmus přes svoji jednoduchost funguje překvapivě rychleji, ačkoliv nijak teoreticky významně [4]. Tento výsledek je také motivací, proč je algoritmus Kua a Crosse v práci použit. Výchozí algoritmus Hunta a Szymanskiho je pak pro srovnání rovněž implementován.

Pseudokód algoritmu 2.6.:

```
procedure KuoCross(A, B: Sequence; T : Matrix)
Begin

  {T[i, j] = hranice třídy j na řádku i}
  T[0, *] := |B| + 1
  T[*, 0] := 0

  For j in 1 .. |B| Do
   Matches[ B[j] ].Append(j) {Seznamy výskytů symbolu}

  MinIndex := 1 {Index, kdy můžeme opět posouvat hranice třídy}

  NowClass := 1

  For i in 1 .. |A| Do

    T[i, *] = T[i - 1, *] {Tento řádek není nutný - ilustruje vznik pole}
                          {T, které není celé potřeba, stačí poslední řádek}

    For j in Matches[ A[i] ] Do {Pro každý index výskytu znaku A[i] v B}

       If j > MinIndex Then

         {Najdi třídu, kterou posuneme}
         While T[i, NowClass] < j Do
          NowClass++

         If T[i, NowClass] > j Then {Jedná se o dominantní pár shody?}
           SAVE_MATCH(i, j)

         MinIndex := T[i, NowClass] + 1

         {Posuneme hranici}
         T[i, NowClass] := j
    {EndFor}
  {EndFor}
End
```

2.7 Možnosti implementace, sestavení LCS

Abychom dosáhli optimální časové složitosti algoritmů z 2.5 a 2.6, je vhodné si předem vypočítat pro každý symbol abecedy, kde je jeho první další výskyt od pozice i v $B_{i..|B|}$. V případě 2.5 však hledáme výskyty v obráceném pořadí. Pokud je $|\Sigma|$ příliš veliká, postačí uspořádaný seznam výskytů jednotlivých znaků. Protože při počítání T pro řádek i hledáme shody stále s jedním symbolem a sice A_i, jsme schopni jeden řádek vypočítat v lineárním počtu kroků vzhledem k množství tříd na něm. Případně, pokud máme jen uspořádaný seznam výskytů, musíme připočítat i pohyb v něm.

T nám ve svém posledním řádku odhaluje LLCS, nikoliv však LCS samotnou. K tomu je zapotřebí, podobně jako v případě algoritmů používajících celou matici C, vracet se po T zpět, respektive uložit si všechny nalezené dominantní páry a třídy, do kterých tyto páry patří. Z toho pak již LCS snadno dopočítáme. Problém je, že jich může být asymptoticky kvadraticky mnoho – přesněji $O(|A| \cdot |B|)$, ač povětšinou méně než všech párů shody obecně.

3 LCS a návrh nové metody

Protože si klademe za cíl využívat nejdelší společné podposloupnosti k efektivnímu ukládání rozdílů mezi jednotlivými soubory, je nutné zohlednit nároky na ukládání informace o jejich shodě a rozdílu. V této práci je použita nabízející se volba, kdy se zapisují bloky dat, které jsou nové, a informace o blocích dat, které jsou stejné. Každý blok je souvislý a ve skutečnosti mluvíme o podřetězcích. Zřejmě čím více nalezneme takových shodných podřetězců, tím větší bude režie na uchování této informace.

LCS se může ve skutečnosti hodně nespojitě rozprostírat přes A i přes B. Společné podřetězce, kterými taková LCS prochází, mohou mít nepříjemně často délku jen pár znaků, nebo dokonce jen jeden znak. Nejdelší společná podposloupnost pak přeskakuje mezi těmito podřetězci. Uložení velkého počtu skoků pak bude vyžadovat nezanedbatelnou prostorovou režii a je na místě se obávat nepříliš velkého užitku takové nalezené LCS. Nabízí se tedy otázka, zda nebude vhodnější obětovat ideální LCS. Když budeme upřednostňovat použití dlouhých společných podřetězců a třeba tím i LCS definitivně ztratíme, můžeme ještě i v takovém případě potenciálně získat dostatečné procento shodných částí obou posloupností, avšak v mnohem menším počtu bloků.

3.1 Společné podřetězce a podposloupnosti

Nyní se odvážíme tvrdit, že každou LCS lze pozměnit tak, že pokud si z ní vybereme libovolný znak daný nějakým párem shody, můžeme vzít celý nejdelší souvislý podřetězec, který tímto párem shody prochází, a ten v takové LCS použít na místo jiných znaků, aniž bychom zmenšili délku nalezené společné podposloupnosti, tedy najdeme opět LCS.

Nechť máme nějaký pár (i, j) a ten používáme v LCS. Nechť následuje pár shody $(i + 1, j + 1)$, který v LCS nepoužíváme. Pokud (i, j) odpovídá k-tému znaku LCS, patří do k-té třídy. $(i + 1, j + 1)$ tedy určitě musí patřit do třídy $k + 1$. My pouze ukážeme, že ho můžeme požít pro $(k + 1)$-ní znak LCS. Ten jistě musí také patřit do třídy $k + 1$. Protože použité páry shody musí mít zřejmě rostoucí obě složky, je tento pár určitě na nějakých pozicích $(p \geq i + 1, q \geq j + 1)$. To ale znamená, že pokud (p, q) nahradíme $(i + 1, j + 1)$, požadované uspořádání párů podle složek jistě nepokazíme a stále

hovoříme o společné podposloupnosti. Ukazuje se tedy, že pokud lze pokračovat hned za nějakým párem shody, můžeme to provést. To analogicky platí i pro (i - 1, j - 1), tedy pokud se rozhodneme použít pár shody bezprostředně předcházející ve smyslu pořadí v posloupnostech A a B. Chceme-li, použitím indukce snadno ukážeme, že původní část LCS lze vyměnit za nějaký společný podřetězec A B.

Takový společný podřetězec se hledá podstatně lépe než samotná společná podposloupnost, ale bohužel úplně přímočaře toho využít nelze.

Pokud bychom si modelovali průběh KC nad nějakými A B, které mají významně dlouhý společný podřetězec, všimneme si, že se ho KC v podstatě drží, což zřejmě plyne z povahy jeho běhu.

3.2 Sufixové pole

Protože v dalších částech budeme využívat hledání společných podřetězců, je nutné se o něm podrobněji zmínit. Nalézt nejdelší společný podřetězec je známá úloha a lze říci, že je dobře prostudovaná.

Už v roce 1973 Donald Knuth navrhl strukturu zvanou sufixový strom [5]. Jedná se o komprimovaný sufixový trie, který lze zkonstruovat v lineárním čase s délkou řetězce. Díky této vlastnosti dovoluje asymptoticky optimálně řešit celou řadu problémů spojených s řetězci. Jeho hlavní nevýhodou je však prostorová náročnost a čas, který je reálně nutný k jeho vytvoření.

Z tohoto důvodu se stalo úspěšné sufixové pole popsané Manberem a Myersem [6]. Tato struktura nenabízí tak jednoduchou a přímočarou realizaci řešení problémů, které jsou triviální, pokud máme již zkonstruovaný sufixový strom. Časová složitost postavení sufixového pole je asymptoticky stejná jako u sufixového stromu, ovšem měřeno reálným časem, je mnohem rychlejší. Také je nutné poznamenat, že existují metody, jak převést sufixové pole na sufixový strom v lineárním čase.

Sufixové pole obsahuje indexy jednotlivých sufixů daného řetězce uspořádané lexikograficky podle sufixů, které označují. Důležitou informací o jednotlivých sufixech je maximální délka jejich společného prefixu, protože zřejmě každý podřetězec je prefixem nějakého sufixu, což bude důležité pro hledání společných podřetězců.

Předpokládejme, že pro každé dva sufixy sousedící v sufixovém poli známe délku jejich nejdelšího společného prefixu (dále označováno jako LCP – z anglického longest common prefix). Vezměme si nějaké tři sufixy řetězce A, které po sobě následují v sufixovém poli. Označme je po řadě indexy a, b, c. Platí tedy, že $A_{a..|A|} \leq A_{b..|A|} \leq A_{c..|C|}$. Buď L_{ab} = LCP($A_{a..|A|}$ $A_{b..|A|}$), L_{bc} = LCP($A_{b..|A|}$ $A_{c..|A|}$). Triviálně platí, že L_{ac} = LCP($A_{a..|A|}$ $A_{c..|A|}$) = min(L_{ab} , L_{bc}). Z toho okamžitě plyne zobecnění, že délkou nejdelšího společného prefixu libovolných dvou sufixů $A_{x..|A|}$, $A_{y..|A|}$ je minimum z LCP jednotlivých sousedících sufixů ležících v sufixovém poli mezi x a y. Je-li tedy $A_{x..|A|} \leq A_{y..|A|}$, sufix x leží v sufixovém poli S na indexu i a sufix y leží na indexu j, pak platí, že $i \leq j$ a především LCP($A_{x..|A|}$ $A_{y..|A|}$) = min $_{k \in i+1..j}$ (LCP $A_{S[k-1]..|A|}$ $A_{S[k]..|A|}$).

Umíme-li efektivně spočítat LCP a minimum z nějakého intervalu hodnot, lze rychle odpovědět na otázku: Jaký je nejdelší společný podřetězec A a B? Nechť jsou oba řetězce nad abecedou $\sum$ a tato abeceda je uspořádaná. Vezměme nové dva symboly \$ a #, které se v $\sum$ nevyskytují, a platí: \$ < # < δ pro všechna $\delta \in \sum$. Vytvořme nový řetězec $A\#B\$$ a zkonstruujme pro něho sufixové pole a LCP pro sousedící sufixy v něm. Pokud vezmeme dva po sobě následující sufixy v sufixovém poli (jeden sufix X, který začíná v části před #, a druhý sufix Y začínající mezi # a \$), platí, že jejich nejdelší společný prefix je stejný jako nejdelší společný prefix odpovídajících sufixů původních řetězců A a B. Pomocné symboly # a \$ nám pomáhají zajistit, že společný prefix zřetězení A a B nepřesáhne konec řetězce A. V sufixovém poli budou přebývat dva sufixy, začínající právě těmito symboly. Ovšem z definice sufixového pole nutně platí, že to budou první dva sufixy v něm. Lze je tedy snadno vyřadit z dalších úvah. K nalezení nejdelšího společného podřetězce pak stačí lineárně projít sufixové pole a pro každé dva sousedící sufixy různých řetězců (A a B – resp. $A\#B\$$ a $B\$$) se podívat na jejich LCP. Z těchto hodnot pak stačí triviálně vzít maximum.

Zůstává zde samozřejmě otázka, jak rychle sestavit informace o LCP. Existují algoritmy, které LCP vyprodukují přímo spolu se sufixovým polem. Je však známa i velmi jednoduchá metoda, jak délku nejdelších společných prefixů dopočítat přímo ze sufixového pole. Tu popsali Kasai et al. [7].

My se však nechceme omezit čistě na rychlé hledání nejdelšího společného podřetězce, ale umět například efektivně najít nejdelší společný podřetězec řetězců $A_{i..j}$ a $B_{p..q}$, pokud

máme sestavené LCP a sufixové pole pro *A* a *B*. K tomu je nutné umět spočítat LCP libovolných dvou sufixů *A* a *B*. Je-li zadáno pole obsahující LCP pro sufixy sousedící v sufixovém poli, snažíme se řešit další známý problém – range minimum query (RMQ) – kdy chceme umět odpovědět na otázku, jaké je minimum z čísel z daného úseku pole.

Problém RMQ lze řešit asymptoticky optimálně. Existuje velmi sofistikovaná struktura zkonstruovatelná v lineárním čase a odpovídající na tento typ dotazu v konstantním čase. Z praktického hlediska se však příliš nevyplácí. Druhou možností je takzvaná sparse tabulka.

Jedná se o poměrně jednoduchou strukturu, která je pouze dvourozměrným polem. Lze ji vytvořit v čase O(N log(N)), kde N je délka pole, pro které ji vytváříme. Její idea je velmi triviální – i-tý řádek tabulky obsahuje na j-té pozici minimum z prvků na pozicích $j..j+2^i-1$. Každý dotaz pro nějaký rozsah lze snadno rozdělit na dva segmenty, jejichž délka je mocnina dvojky. Vůbec nám totiž nevadí, že se segmenty překrývají, takže se de facto nejedná o rozdělení. Stačí pak vzít minimum z těchto dvou minim pro segmenty.

3.3 Užití nejdelšího společného podřetězce

V části 3.1 je ukázáno, že každá nejdelší společná podposloupnost může být upravena následujícím způsobem: Pokud prochází nějakým společným podřetězcem, pak ho použije celý. Tato vlastnost vede k domněnce, že nejdelší společná podposloupnost bude zahrnovat i nejdelší společný podřetězec. Tato domněnka není samozřejmě korektní a snadno lze vymyslet protipříklad. I přesto má svůj význam.

Prvním argumentem je, že většina takových protipříkladů moc dobře neodpovídá reálným datům, která bychom si přáli archivovat. Změny v souboru mají často poměrně lokální charakter a nejdelší společný podřetězec povětšinou odpovídá tomu, co bychom intuitivně označili za nezměněnou část souboru.

Druhým argumentem pro použití nejdelšího společného podřetězce je jeho výhodnost z hlediska režie nutné na popis shodujících se a lišících se bloků dat. Představme si případ, kdy máme falešnou LCS sestávající pouze z nejdelšího společného řetězce a skutečnou LCS, která je však roztroušena do mnoha malých bloků. Pokud budou

takové bloky dostatečně malé, může být skutečná LCS ve srovnání s falešnou nepoužitelná.

Třetím argumentem je pak skutečnost, že si nebudeme nárokovat nalezení optimálního výběru shodných částí takových, že neexistuje žádný, který by dovolil uložit data úsporněji. Budeme klást důraz především na použitelnost s ohledem na čas potřebný k reálnému výpočtu a na fakt, že reálná data dávají v mnoha případech důvod k upřednostnění teoreticky méně efektivních postupů.

3.4 Návrh prvního algoritmu

Nyní máme již všechny potřebné nástroje k tomu, abychom mohli přikročit k prvnímu z návrhů algoritmů určených pro efektivní ukládání v čase jen málo se měnících dat.

Algoritmus vezme vstupní posloupnosti A a B a najde jejich nejdelší společný podřetězec. Nechť tento podřetězec začíná v posloupnosti A na pozici M_A a v posloupnosti B na M_B. Nechť jeho délka je L. Pokud tímto podřetězcem skutečně procházela nejdelší společná podposloupnost, používala mimo párů shody odpovídajících tomuto řetězci pouze páry shody $(i, j) \in \{1 .. (M_A - 1) \times 1 .. (M_B - 1)\}$ a páry shody $(k, l) \in \{ (M_A + L) .. |A| \times (M_B + L) .. |B|\}$. Aplikujeme tedy rekurzivně toto pravidlo na obě části, které nám vzniknou v důsledku tohoto rozdělení nejdelším společným podřetězcem. Budeme tedy nejprve hledat společné části $A[1 .. (M_A - 1)]$ a $B[1 .. (M_B - 1)]$, v druhém kroku pak $A[(M_A + L) .. |A|]$ a $B[(M_B + L) .. |B|]$.

Bezesporu je důležité tento postup realizovat co nejefektivněji. Samotné nalezení nejdelšího společného podřetězce jsme již popsali. Jak ovšem vyřešit hledání nejdelšího společného podřetězce v jednotlivých částech?

Jistě nechceme zkoumat více sufixů, než kolik je nutné. Přinejmenším se chceme vyhnout všem, které do dané oblasti nepatří. Sufixové pole dané k dispozici však sufixy rozmíchává mezi sebe. Budeme si tedy předávat seznam indexů do sufixového pole uspořádaný ve vzrůstajícím pořadí. Po nalezení nejdelšího společného podřetězce vytvoříme dva seznamy – pro každou část jeden. Při rekurzivním volání si kromě

sufixového pole předáme i seznam indexů ukazujících do něho, které odpovídají zkoumaným podřetězcům.

Zde zdůrazněme, že při hledání nejdelšího společného podřetězce už nepočítáme LCP dvou sousedících sufixů, ale prakticky libovolných dvou sufixů. Právě tento fakt nás nutí využívat vhodnou strukturu pro zodpovězení dotazů na minima.

Časová složitost uvedeného algoritmu je asymptoticky poměrně nepříznivá. V nejhorším případě se dvě posloupnosti vždy budou shodovat jen v jednom znaku. Může tak být nutné vykonat až $O(|A| + |B|)$ rekurzivních volání, což vyústí v časovou složitost $O((|A| + |B|)^2)$.

Povšimněme si alespoň drobného možného zlepšení. Délka nejdelšího společného podřetězce se nezvětšuje při rekurzivním volání. Pokud tedy označíme nějakou délku za minimální (kratší je již nevýhodná, kvůli režii na popis shod a rozdílů), pak v momentě, kdy narazíme na podřetězec kratší, nemusíme provádět další dělení a rekurzi.

Pseudokód algoritmu 3.4.:

```
procedure SA(A, B : Sequence;
             S : SuffixArray; I : SuffixArrayIndices;
             Border : Integer; {Pozice znaku '#' v "A#B$" }
             R : LCPSparseTable)
Begin

  {Najdi nejdelsi spolecny podretezec}
  PosA := 0, PosB := 0, Length := 0 {Jeho pozice a délka}
  For p in 1 .. |I| - 1 Do
    If ( S[I[p]] < Border AND S[I[p + 1]] > Border) Then
    {I[p] je sufix A}
       If R.FindMin(I[p], I[p + 1]) > Length Then
        Length := R.FindMin(I[p], I[p + 1])
        PosA := S[I[p]]
        PosB := S[I[p + 1]] - Border

    If ( S[I[p]] > Border AND S[I[p + 1]] < Border) Then
    {I[p] je sufix B}
       If R.FindMin(I[p + 1], I[p]) > Length Then
        Length := R.FindMin(I[p + 1], I[p])
        PosA := S[I[p + 1]]
        PosB := S[I[p]] - Border
  {EndFor}

  If Length > MinLength Then

    I1 := Index From I Where
            Index in 1 .. PosA - 1 OR
            Index in Border + 1 .. Border + PosB - 1

    SA(A[1 .. PosA - 1],  B[1 .. PosB - 1], S, I1, Border, R)

    FOUND_BLOCK(PosA, PosB, Length)

    I2 := Index From I Where
            Index in PosA + Length .. |A| OR
            Index in Border + PosB + Length .. Border + |B|

    SA(A[PosA + Length .. |A|], B[PosB + Length .. |B|],
       S, I2, Border, R)
  {EndIf}
End
```

3.5 Návrh druhého algoritmu

Předchozí algoritmus má zřejmé potenciální nevýhody. Především pokud je délka nejdelšího společného podřetězce malá, pak lze očekávat větší množství rekurzivních kroků. Zároveň je daleko menší pravděpodobnost, že skutečná nejdelší společná podposloupnost opravdu přes tento podřetězec prochází.

Tento druhý algoritmus se pak více zaměřuje na použití technik typických pro hledání nejdelší společné podposloupnosti. Algoritmus v 3.4 de facto pouze využívá tvrzení o LCS pouze jako teoretické východisko pro další úvahy.

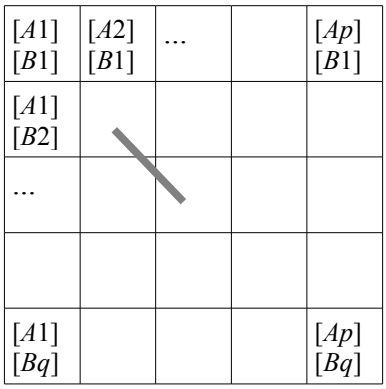
Označme X jako nejdelší společný podřetězec řetězců A a B. Zvolme přirozené číslo M shora omezené délkou kratšího z řetězců A a B. Toto číslo bude jakousi hranicí pro délku nejdelšího společného podřetězce. Můžeme ještě uvažovat nějakou konstantu K, která bude udávat, jaký nejkratší společný podřetězec má ještě vůbec smysl uvažovat (viz poslední odstavec 3.4). Bude-li tedy $|X| \geq M$, budeme postupovat identicky jako v prvním algoritmu. Bude-li $|X| < K$, rovnou ukončíme zpracovávání dané části. Zbývá popsat řešení případu $K \leq |X| < M$.

Nejdelší společný podřetězec je momentálně příliš krátký, je však ještě dost dlouhý, aby mělo smysl se jím zabývat. Vezměme všechny maximální společné podřetězce (maximálním nerozumíme nejdelší, ale každý, který již nelze prodloužit – např. V „ABCDXYZ“ a „ABCDUYZ“ je „YZ“ maximálním, podobně jako „ABCD“, který je však i nejdelším). Podobnou úvahou, kterou jsme použili při argumentaci ve prospěch nejdelšího společného podřetězce, lze přijít k domněnce, že nejdelší společná podposloupnost bude využívat některých maximálních společných podřetězců. Narážíme však na otázku: Které takové maximální podřetězce to jsou?

V části 2.2 je popsán algoritmus založený na dynamickém programování, který si uzpůsobíme pro tyto účely. Podobnost problému nalezení nejdelší společné podposloupnosti a výběru vhodných maximálních podřetězců je vcelku nápadná. Následující postup je tedy takový: Rozdělíme oba zkoumané řetězce na bloky délky M. Potom můžeme označit A=[$A1$][$A2$][$A3$]..[Ap], B=[$B1$][$B2$][$B3$]..[Bq], kde [Ai] (resp. [Bj]) označuje jeden blok délky M. Poslední bloky posloupností mohou být kratší. Uvažujme množinu, která odpovídá kartézskému součinu bloků A s bloky B. Vznikne

nám tak jakási tabulka, jejíž každá buňka odpovídá dvojici bloků řetězců. Každá taková buňka jistě obsahuje nějaký nejdelší maximální společný podřetězec pro danou dvojici – tento podřetězec omezíme pouze na rámec buňky. Délky těchto maximálních společných podřetězců použijeme jako ohodnocení každé z buněk.

Předpokládejme, že nejdelší společná podposloupnost prochází přes nějaké maximální společné podřetězce. Vyznačme si ve vybudované tabulce ty buňky, ve kterých takové podřetězce leží. Zanedbejme případy, kdy podřetězec přechází přes hranici buňky způsobem, kdy protne buňku, kde začíná, buňku přímo pod sebou a třetí buňku napravo pod ní (jak je ukázáno v obrázku 3.5).



Obr. 3.5: Maximální podřetězec, který protíná více buněk. Ve skutečnosti je omezen, aby zůstal uvnitř buňky.

Za takového předpokladu lze tvrdit následující: Je-li označena buňka odpovídající blokům [*Ai*] a [*Bj*] tak, že patří do naší přibližné LCS, pak následující buňka, kterou LCS prochází, odpovídá blokům [*Au*] a [*Bv*], kde $u > i$ a $v > j$. Naším cílem bude vybrat buňky, kde očekáváme průchod nejlepší LCS. Zároveň se spokojíme s tím, že LCS prodlužujeme pouze po buňkách s indexy bloků ostře většími.

Dobrovolně se tedy ochudíme o tyto případy, kdy podřetězec nešikovně přesahuje buňku. Zřejmě však platí, že řetězce nejsou nikdy delší než M – vždyť délka nejdelšího společného podřetězce je takto omezena shora. Nejspíš proto k velkému zanedbání nebude příliš docházet.

Máme tedy ohodnocené buňky v tabulce pro dvojice bloků a chceme si z nich vybrat. Podobně jako v algoritmu 2.2 zde užijeme dynamické programování a dokonce i velmi podobné pravidlo. Ke každé buňce budeme počítat celkové ohodnocení: Sumu ohodnocení nejlepšího výběru buněk pro nejdelší společnou podposloupnost

procházející v tabulce od buňky ([A1], [B1]) doprava dolů po aktuální buňku.

Každá buňka do výběru buď patří nebo naopak nepatří. Pokud do výběru nepatří, přejímá lepší z celkových ohodnocení buňky nad sebou a vlevo od sebe (orientováno ve smyslu obrázku 3.5). Pokud naopak do výběru patří, pak její celkové ohodnocení odpovídá součtu vlastního ohodnocení buňky a celkového ohodnocení buňky vlevo nahoře nad sebou. Uveďme to tedy formálním způsobem: Je-li C[i, j] vlastním ohodnocením buňky (vychází z délky maximálního společného podřetězce uvnitř) a S[i, j] je celkovým ohodnocením pak platí:

1. Položme $S[1, 1] = C[1, 1]$.

2. Pro $i = 1$ pak $S[1, j] = \max(S[1, j - 1], C[1, j])$. Podobně pro $j = 1$ potom $S[i, 1] = \max(S[i - 1, 1], C[i, 1])$.

3. V ostatních případech $S[i, j] = \max(S[i - 1, j], S[i, j - 1], C[i, j] + S[i - 1, j - 1])$.

Na konci výpočtu nalezneme v buňce pro poslední dvojici bloků informaci o celkovém ohodnocení vybraných buněk. Ta ovšem není tak zajímavá, protože je ve skutečnosti jen technickým nástrojem. Skutečně důležité jsou až ony vybrané buňky. Ty nalezneme tak, že se budeme postupně od konce vracet – vždy podle toho, zda jsme navázali na buňku diagonálně vlevo nahoře nad aktuální, nebo jsme pouze převzali ohodnocení zleva či shora.

V algoritmu z 3.4 jsme používali nejdelší společný podřetězec k rozdělení zkoumaných posloupností na dvě části. Nad každou z těchto částí jsme rekurzivně aplikovali stejný postup. Zde jsme však přímo nalezli jednotlivé úseky, pro které budeme rekurzivní postup opakovat. Vyhýbáme se tak nepříjemnému případu, kdy je nejdelší společný podřetězec příliš krátký a my vícenásobně provádíme drahé výpočty nad stejnými daty.

Algoritmus tedy buď řetězce přestane zkoumat, nebo je rozdělí na dvě části, či až případně na min(p, q) částí (p – počet bloků A; q – počet bloků B). Zřejmě bude nezbytné opět rozdělit seznam indexů do sufixového pole na seznamy pro jednotlivé buňky.

Zbývá se však ptát po vhodné volbě M. Tabulka s ohodnocením buněk má asymptoticky

velikost $O((|A| / M) \cdot (|B| / M))$. Příliš malé M tak vyústí v neefektivní výpočet tabulky, naopak velké M nám nijak výrazně neulehčí práci v dalších krocích. Za předpokladu, že $|A| \approx |B|$ budeme M volit jako $\sqrt{M}$.

Pseudokód algoritmu 3.5.:

```
procedure SAD(A, B : Sequence;
              S : SuffixArray; I : SuffixArrayIndices;
              Border : Integer; {Pozice '#' v "A#B$" }
              R : LCPSparseTable)
Begin
  {Najdi nejdelsi spolecny podretezec}
  PosA := 0, PosB := 0, Length := 0 {Jeho pozice a délka}
  For p in 1 .. |I| - 1 Do
    If ( S[I[p]] < Border AND S[I[p + 1]] > Border) Then
    {I[p] je sufix A}
       If R.FindMin(I[p], I[p + 1]) > Length Then
        Length := R.FindMin(I[p], I[p + 1])
        PosA := S[I[p]]
        PosB := S[I[p + 1]] - Border

    If ( S[I[p]] > Border AND S[I[p + 1]] < Border) Then
    {I[p] je sufix B}
       If R.FindMin(I[p + 1], I[p]) > Length Then
        Length := R.FindMin(I[p + 1], I[p])
        PosA := S[I[p + 1]]
        PosB := S[I[p]] - Border
  {EndFor}

  If Length > MinLength Then

    {Je nejdelší společný podřetězec dost dlouhý?}
    If Length > RequiredLength Then {RequiredLength si volíme}

      I1 := Index From I Where
              Index in 1 .. PosA - 1 OR
              Index in Border + 1 .. Border + PosB - 1

      SA(A[1 .. PosA - 1],  B[1 .. PosB - 1], S, I1, Border, R)

      FOUND_BLOCK(PosA, PosB, Length)

      I2 := Index From I Where
              Index in PosA + Length .. |A| OR
              Index in Border + PosB + Length .. Border + |B|

      SA(A[PosA + Length .. |A|], B[PosB + Length .. |B|],
        S, I2, Border, R)

    Else
```

```
      {Pokud je nejdelší společný podřetězec dlouhý, ale ne dostatečně}

      {C - ocenění buněk 1 .. Sqrt |A|, 1 .. Sqrt |B| }
      {Buňka má tedy rozměry (Sqrt |A|) * (Sqrt |B|)}
      C[*, *] := 0;

      Substrings := CommonSubstrings(A, B, S, I) Where
                    SubstrLength > MinLength

      {Ohodnocení buněk}
      For SubstrPosA, SubstrPosB, SubstrLength in Substrings Do
        Cell := FindCell(SubstrPosA, SubStrPosB - Border)
        L := LimitToCell(Cell, SubstrPosA, SubstrPosB, SubstrLength)
        C[Cell] += Value(SubstrLength)
      {EndFor}

      {Celkové ohodnocení}
      Sums[*, *] = 0 {Pro zjednodušení uvažujeme pole velikosti C,
                      ale s okrajem pro indexy [-1, *] a [*, -1] }
      For i in 0 .. Sqrt |A| Do
        For j in 0 .. Sqrt |B| Do
          Sums[i, j] := Max(
                 Sums[i - 1, j] - Penalty, {převezmeme shora}
                 Sums[i, j - 1] - Penalty, {převezmeme zleva}
                 Sums[i - 1, j - 1] + C[i , j]) {buňka [i, j] se použije}

      BestCell := FindMax (Sums)
      CellsOnPath := FindCellsToStart(BestCell)

      {Rozdělíme indexy do sufixového pole k jednotlivým buňkám}
      CellIndices[k in 1 .. |CellsOnPath|] := Index From I Where
                              Index in CellsOnPath[k].RangeInA OR
                              Index in CellsOnPath[k].RangeInB

      {Zpracuj vybrané buňky}
      For k In 1 .. |CellsOnPath| Do
        SAD(A[ CellsOnPath[k].RangeInA ], B[ CellsOnPath[k].RangeInB ],
               S, CellIndices[k], Border, R);
    {EndElse}
  {EndIf}
End
```

4 Implementace algoritmů

Doposud jsme popisovali teoretická východiska a algoritmy určené pro zkoumání a srovnání. Jejich testovací implementace však musí řešit problémy, které úplně nesouvisí s teoretickou stránkou věci, případně nepřinášejí nic myšlenkově zásadního. Přesto je však nutné se o nich zmínit. Pro srovnání jsou mezi srovnávané algoritmy ještě zařazeny jednoduché heuristické techniky, které nevyužívají nic z problematiky spojené s nejdelšími společnými podposloupnostmi, ale je vhodné je popsat.

Zásadním problémem, který je víceméně jednotně řešen v každém z algoritmů, je fakt, že nelze naráz pracovat s celou posloupností. Datové soubory mohou mít velikost v řádech stovek megabytů a v takový moment by pokus o pouhé jejich načtení do paměti byl nemyslitelný. Natožpak budovat nějaké pomocné tabulky. Proto je vytvořeno posuvné okno pro každou posloupnost a algoritmy pracují pouze s oknem specifikovaným úsekem dat.

Posuvné okno zavádí nový prvek do celého problému, jeho užití však bylo nezbytné. Metody, kterými se snažíme omezit dopad tohoto faktoru budou popsány v dalším textu.

Každý algoritmus je pak implementován pro dvě použití: primárně pro samotné nalezení bloků shody, sekundárně také pro odhad podobnosti dvou posloupností. Sekundární implementace je zjednodušenou verzí algoritmu, která má za úkol pouze určit míru shody, nikoliv přímo shodná data. Toto může být užitečné, když se chceme nejprve rozhodnout, zda daná data má smysl ukládat jako rozdíl oproti předchozím.

4.1 Použitá platforma

Veškerý programový kód byl vytvořen pro platformu Microsoft .NET Framework 3.5. Vše je zahrnuto uvnitř jedné Solution pro Microsoft Visual Studio 2008. Jako programovací jazyk byl použit C# ve verzi 3.

Použitá platforma nedovoluje stoprocentně přesně měřit rychlostní výkon programu, neboť se jedná o prostředí s řízenou správou paměti, které navíc kompiluje mezikód až v době běhu – takzvaný Just In Time. Tato práce si však klade za cíl otázku použitelnosti a vhodnosti jednotlivých algoritmů v reálných aplikacích, u kterých nelze očekávat optimální, lze říci až laboratorní, podmínky.

Přesto jsou v některých momentech použity postupy, které upravují dopad chování této platformy. Pro vyloučení zkreslení měření v důsledku Just In Time kompilace jsou všechny používané algoritmy vynuceně kompilovány předem. Jindy je zase použit neřízený přístup k polím. Jedná se zejména o místa, kde víme, že kontrola přístupů ze strany prostředí je zbytečná. Zároveň je však zohledněno, že vynucení si nechráněného přístupu spotřebuje nějaký čas, ačkoliv jinak přináší zrychlení. Je tedy použito tam, kde velmi často potřebujeme pracovat s určitým úsekem pole.

4.2 Heuristika BestNext

Tento postup je běžně používanou heuristikou pro hledání nejdelší společné podposloupnosti. Její zásadní nevýhodou však je, že negarantuje o kolik moc horší může její výsledek být oproti optimu.

Heuristika postupně prodlužuje svůj odhad o nové a nové znaky, staré při tom už nikdy nezmění. Další znak se hledá vždy od posledního použitého páru shody. Ze všech dalších párů shody, které by mohly po aktuálním následovat, je vybrán ten, za nímž zbývá „nejvíce“ znaků. Myslí se tím minimum ze vzdáleností znaku do konce posloupnosti: Máme-li posloupnosti A a B a pár shody (i, j), pak vzdáleností do konce posloupnosti značíme $\min(|A| - i, |B| - j)$.

Heuristika se běžně implementuje tak, že známe další nejbližší pozici každého znaku abecedy a nám stačí pro jednotlivé znaky takto prozkoumat pouze jeden pár shody.

Implementace této heuristiky používá většího posuvného okna, protože neklade tak velké paměťové nároky na jeho zpracování. Okno se pak posouvá vždy, jakmile dosáhneme jeho hranice v nějaké z posloupností.

4.3 Heuristika založená na Rabin-Karpově algoritmu

Tato metoda byla původně určena pouze pro rychlé zpracování odhadu podobnosti, ukázalo se však, že nabízí rozumné srovnání s ostatními sofistikovanějšími algoritmy. Implementaci označujeme jako RabinKarp.

Z posuvného okna první posloupnosti jsou pseudo-náhodně vybrány krátké vzorky, které se pak snažíme hledat v druhé posloupnosti. Rabin-Karpův algoritmus je známý

pro svou rychlost při hledání výskytů vícero vzorků v textu. Každý nalezený výskyt vzorku je okamžitě hladově akceptován. Algoritmus dále jen použije nalezený vzorek jako začátek nějakého shodného podřetězce, který se pak snaží prodloužit.

Podobně jako v předchozí heuristice, i v tomto případě si můžeme dovolit větší posuvné okno vzhledem k nízkým nárokům na paměť.

4.4 Základní dynamický algoritmus – Simple

Jedná se o přímočarou realizaci základního algoritmu z části 2.2. Kvadratická paměťová složitost vzhledem k součtu délek vstupů si však vynucuje menší posuvné okno.

Nejprve se vypočítá matice *C*, tak jak je popsáno v 2.2. Algoritmus pak hledá nejlepší hodnotu co nejvzdálenější od pravého spodního rohu. Ačkoliv je jisté, že v pravém dolním rohu je maximum, snažíme se preferovat políčka, která nám umožní obě vstupní posloupnosti k sobě lépe zarovnávat. Jistě nemá smysl zahodit kus posloupnosti, pokud to ve skutečnosti není vůbec zapotřebí.

Algoritmus však pouze naivně nepočítá matici, ale nejprve zkontroluje, zda se obsahy posuvných oken neshodují. Pokud ano, vezme maximální společný podřetězec, jehož počátek jsme právě nalezli. Tento postup je použit i v dalších algoritmech. Zřejmě totiž platí, že v rámci posuvného okna musí tento podřetězec vždy ležet v nějaké nejdelší společné podposloupnosti.

Ve variantě pro odhad odpadá nutnost konstruovat zpětně z matice bloky shody.

4.5 Vylepšený dynamický algoritmus – Enhanced

Tato varianta implementace algoritmu 2.2, založeného na dynamickém programování, vychází z implementace předchozí, uvedené v části 4.4. Blíže se však zaměřujeme na problematiku zarovnávání posloupností k sobě, respektive řešení problému spojeného s používáním posuvného okna.

Metoda, která bude popsána, je použita i v dalších algoritmech. Vycházíme z předpokladu, že pokud se nepodařilo najít dostatečně mnoho shodných dat, nejspíš

nemáme pozice oken ve vstupních datech správně zarovnané. Toto samozřejmě nelze tvrdit hned s prvním neúspěchem, ale pokud se podobná situace opakuje, má smysl nějak zakročit. Konkrétně se snažíme, aby oběma oknům zbývalo do konce vstupní posloupnosti stejné množství znaků. Budeme tedy pohybovat s tím, které je od konce vzdálenější, a hledat v něm shodu s daty fixovaného okna.

4.6 HuntSzymanski

Zde implementujeme algoritmus popsaný v části 2.5, implementaci označujeme HuntSzymanski. Opět je paměťová složitost asymptoticky zřejmě kvadratická, což nutí k užití malého posuvného okna. Vycházíme z již zmíněných metod v části 4.4 a 4.5 – myslí se tím využití shodného podřetězce, který se nalézá okamžitě na počátku obou posuvných oken, a snaha zarovnávat okna při nízké úspěšnosti hledání shodných dat.

Podobně jako v předchozím případě je při odhadování podobnosti ušetřen čas díky faktu, že nemusíme zpětně konstruovat bloky shody dat.

4.7 KuoCross

Jedná se o implementaci metody uvedené v 2.6, označujeme ji jako KuoCross. Programový kód je prakticky totožný s 4.6. Je to důsledek faktu, že tato metoda se dá považovat za pouhou modifikaci algoritmu Hunta a Szymanskiho. Používá i stejnou volbu velikosti posuvného okna.

Modifikace se tak v kódu projeví výlučně během posouvání hranic tříd shody, kdy nepostupujeme od posledního páru shody k prvnímu a vždy upravujeme patřičnou hranici. Naopak bereme páry shody postupně od nejmenšího a posun hranice se uskuteční jen pro dominantní páry. Můžeme tak nezřídka mnoho párů zahodit.

Je velmi zajímavé, že tato nekomplikovaná úprava přináší takové rychlostní zlepšení. Toto zlepšení jasně vyplývá i ze srovnání a jedná se o důvod, proč byl tento algoritmus vybrán pro účely této práce [8].

4.8 První návrh – SA

Zde je realizován návrh prvního algoritmu vycházejícího z použití sufixového pole (odtud název SA = Suffix Array), tak jak je popsán v části 3.4. Kód pro odhad podobnosti i pro samotné zpracování jsou velmi podobné, odhad však opět šetří čas vynecháním konstrukce informací o shodných blocích.

Tento a oba následující algoritmy používají ke své činnosti sufixové pole. Pro konstrukci sufixového pole je použit Sandersův a Kärkkäinenův algoritmus [9]. Protože konstruujeme sufixové pole pro zřetězení vstupních posloupností, dochází uvnitř k posunům symbolů v abecedě (kvůli $ a #), které jsou nezbytné například i pro další konstrukci LCP.

Stejně jako ostatní algoritmy i tento pracuje s posuvnými okny nad vstupní posloupností. Povaha algoritmu však dovoluje pracovat s oknem významně větším než jaké například používáme v 4.7. Sparse tabulka i sufixové pole tak vždy pracují s daty s předem známou a shora omezenou velikostí. Toho je využito, aby se zabránilo zbytečným alokacím během práce algoritmu. Potřebný paměťový prostor si totiž algoritmus, respektive tyto struktury, vyhradí již na počátku.

Při realizaci rekurzivního kroku si předáváme seznam indexů v sufixovém poli. Pokud bychom ho vždy znovu alokovali, bylo by to zbytečné plýtvání pamětí. Totiž po nalezení nejdelšího společného řetězce v jedné úrovni už nepotřebujeme s tímto seznamem nikdy více pracovat. Chceme ho pouze rozdělit na dva pro úrovně nižší, respektive pro jednotlivé části vstupních posloupností.

Používá se tedy postup, kdy se stále zaměňují pole s aktuálním seznamem indexů a pole s dočasným seznamem indexů, přičemž každá úroveň volání má přesně specifikováno, v které části pole má svůj seznam hledat. Žádné volání pak nemění data mimo takto přesně daný rozsah indexů. Jelikož seznamy pro obě nižší úrovně jsou ve skutečnosti disjunktními podmnožinami původního seznamu, naprosto jednoznačně jim prostor vyhrazený pro nadřazenou úroveň stačí.

4.9 První návrh a zásobník – ESA

Tato implementace plně vychází z předchozí, popsané v 4.8. Oproti ní však slouží pouze jako srovnání. Je tomu tak, protože v algoritmu 4.10 je použita vlastní realizace zásobníku oproti standardnímu, který se používá při rekurzi. Můžeme tak posoudit dopad, který tato změna má, a je tak možné zhodnotit přínosy či nevýhody nového algoritmu a nespojovat je s touto změnou.

4.10 Druhý návrh algoritmu – SAD

Název je zkratkou za Suffix Array Dynamic – protože algoritmus dynamicky přizpůsobuje svůj průběh nalezené míře shody, respektive podle délky nejdelších společných podřetězců jednotlivých částí. Jedná se o implementaci postupu popsaného v části 3.5. Implementace vychází z předchozí, zmíněné v části 4.9.

Kód pro odhad podobnosti a její samotné zpracování se opět nepříliš liší. I tentokrát se konstruují informace popisující jednotlivé bloky shodných a rozdílných dat pouze v části pro zpracování.

Algoritmus nepoužívá při své činnosti rekurzi. Je tomu tak především z jednoho zásadního důvodu: Během rozdělení částí vstupních posloupností do menších bloků, kdy se používá dynamické programování, získáme informace o jednotlivých vyznačených blocích v obráceném pořadí, než v jakém potřebujeme tyto bloky zpracovat. Jednou z možností je si vytvořit jejich seznam a pak postupovat podle něho. Druhou možností je ukládat volání pro tyto bloky na vlastní zásobník, což jednak zaručí správné pořadí, ale především se nemusí tyto informace nikde duplikovat.

Ještě poznamenejme, že podobně jako v 4.8 a 4.9 se používá postup se střídáním dočasného a aktuálního pole pro seznam indexů do sufixového pole. Pouze v kroku s rozkladem na bloky je situace komplikovanější – odpovídající rozsah indexů již nedělíme na dvě, ale na tolik částí, kolik je na cestě pro LCS vyznačeno bloků. To zřejmě vyžaduje jejich správné roztřídění. Aby byla tato část algoritmu realizována rychle, využívá se faktu, že v každém řádku i sloupci tabulky smí být označena jen jedna buňka. To dovoluje sestavit pole, které ke každému indexu v sufixovém poli odpoví, do kterého seznamu patří. Pak již jen využijeme omezenosti hodnot, se kterými pracujeme, a pomocí radix-sortu je rozdělíme.

5 Architektura programu

5.1 První část – LCS-FTP

V této části je jeden hlavní namespace Versioning a ten dále obsahuje dva důležité namespacy: LCS a Storage. Nejprve se věnujme členům namespacu Versioning.

Základní nástroje

Základním prvkem je zde rozhraní IDataSequence. Idea je taková, že všechny algoritmy pracují s dvěma posloupnostmi bytů. Sekvence se chová jako proud bytů, který je indexovatelný od začátku – pozice 0. Nejjednodušší implementací IDataSequence je potom třída FileDataSequence, která přímo zaobaluje FileStream z .NET.

Nutně je třeba nějak popisovat shodné a rozdílné části dvou datových sekvencí. K tomu právě slouží struktura DifferentialBlock, která ukazuje do obou sekvencí a zároveň si nese informaci o počtu bytů, jež souvisle pokrývá v obou sekvencích od daného indexu. Může představovat buď blok shodných dat, či naopak dat rozdílných. Závisí na interpretaci toho, s jakou instancí třídy DifferentialData je spojen. Tato třída sdružuje jednotlivé bloky shody resp. rozdílu pro nějakou jednu sekvenci dat. Ne všechny bloky, které si DifferentialData pamatuje však musí patřit do jedné datové sekvence. Implementace je použita taková, že jedna sekvence dat může být složena z bloků vlastních dat a bloků dat ostatních sekvencí. Tak je právě realizováno ukládání jednotlivých verzí souborů.

V základním tvaru DifferentialData nese pouze bloky své datové sekvence a bloky odkazující se na data předchůdce. Pokud předáme této třídě instanci DifferentialData předchůdce, umí (pomocí metody MergeWithOlder) proložit bloky dat, které se odkazují na předchůdce, daty předchůdce a odkazy na data předchůdce svého předchůdce. Pokud bychom si data aktuální verze představili jako hmotné bloky a naopak odkazy na předchůdce jako mezery, tak spojení verze s předchůdcem odpovídá položení dvou datových sekvencí na sebe.

BlockDataSequence pak vytváří datovou sekvenci ze složení datové sekvence své a svých předchůdců, a to podle jednotlivých bloků své verze a verzí předchůdců, na kterých aktuální verze závisí.

MultiVersionFile a FileVersion pak představují nástroje pro práci na úrovni souborů a verzí souborů, nikoliv již jen datových sekvencí. Ke své práci pak využívají nástroje z namespaců LCS i Storage.

Logický file-system

Namespace Storage představuje vrstvu logického filesystému. Očekává se, že pracuje nad fyzickým diskem, není však vyloučena možnost jeho rozšíření na jiný způsob ukládání dat.

Tak, jak je logický filesystém v Storage používán v tomto projektu, předpokládá, že má k dispozici nějakou fyzickou složku, která reprezentuje logický kořenový adresář. V této složce jsou pak uloženy soubory a podadresáře. Každý soubor může mít své „podsoubory“. Ty právě slouží k ukládání dat spojených s verzemi jednoho souboru. Realizace na fyzickém souborovém sytému je takováto: Ve složce, která odpovídá složce nějakého logického adresáře, jsou jednak přímo uloženy soubory (resp. jejich hlavičky), ve fyzické podsložce „data“ jsou jednotlivé podsoubory a v podsložce „subdirs“ jsou složky logických podadresářů. Tato realizace byla zvolena, aby nedocházelo ke kolizím názvů. Je implementován i jednoduchý systém zamykání přístupů k souborům. Zámek funguje na základě fyzické cesty k souboru, takže pokud se pokouší dva logické filesystémy přistoupit k jednomu souboru, je tomu zamezeno. Použití zámků se však nevynucuje.

Algoritmické jádro projektu

Namespace LCS je z algoritmického hlediska nejzajímavější částí projektu a lze říci, že tvoří logické jádro při řešení hlavního cíle projektu. Základním členem je třída LCSProvider, která je předkem pro všechny ostatní členy tohoto namespacu. LCSProvider poskytuje dvě důležité metody. Odhadnutí podobnosti dvou datových sekvencí a samotné zpracování pro vyhledání shodných bloků. Každý algoritmus je potom vytvořen jako potomek této třídy a tyto dvě metody implementuje. Odhadování podobnosti by mělo fungovat rychleji, než samotné zpracování. Nevyžaduje se od něj naopak taková přesnost a detailní informace.

Nejtriviálnějším implementovaným postupem je heuristika BestNext z 4.2 ve třídě BestNextLCSProviderWindowed. Heuristika s Rabin-Karpovým algoritmem z 4.3 nalezneme v RabinKarpLCSProvider. Pracuje nad posuvnými okny v obou datových sekvencích, přičemž z jedné si vezme náhodně rozmístěných N vzorků (používá se N = 8192) a v druhé se je snaží najít. Vzorky se používají 16-bytové.

Algoritmus z 4.4 nalezneme v SimpleLCSProvider. Jeho vylepšení uvedené v části 4.5 je potom v EnhancedLCSProvider. Algoritmus Hunta a Szymanskiho z 4.6 je implementován ve třídě HuntSzymanskiLCSProvider. Zlepšením této metody, algoritmus Kua a Crosse popsaný v 4.7, implementuje třída KuoCrossLCSProvider.

Navrhované nové algoritmy nalezneme ve třídách SALCSProvider, EnhancedSALCSProvider a SaDynLCSProvider. Implementují přesně v tomto pořadí algoritmy z 4.8, 4.9 a 4.10.

5.2 Druhá část – LCS-FTP-Server

Zatímco první část se starala o algoritmickou funkčnost a správu ukládání dat, druhá část se tyto nástroje zpřístupňuje přes rozšířenou verzi FTP protokolu a dovoluje uživateli řídit běh aplikace pomocí GUI.

GUI část aplikace

V této části je opět jen jeden kořenový namespace, je jím LCS-FTP-Server. Jeho přímí členové se starají o GUI rozhraní a kontrolu spouštění a běhu ostatních částí. V tomto směru je zde základem třída MainForm, která zobrazuje ovládací tlačítka a spouští instanci serveru. Dále zobrazuje informace o uživatelích připojených přes FTP a informace o událostech zpracování požadavků. Třída používá standardní WinForms, nikoliv novější technologie XAML. Z tohoto formuláře lze upravovat nastavení serveru. Dále lze otevírat jiná okna. Jsou jimi SettingsForm, který zpřístupňuje nastavení IP adresy a portu, kde server naslouchá, dále potom UsersForm, kde se spravují FTP uživatelé serveru.

Namespace FTP

Většina aplikační logiky této části je však uložena v namespacu FTP. V něm nalezneme pomocné třídy jako FTPUsers – ta se stará o správu informací o uživatelích –, či třídu FTPServerConfig – jejím stará se o nastavení serveru a ukládání a načítání těchto nastavení.

Nejdůležitější třídou je ovšem FTPServer. Skrze tuto třídu se spouští celá instance serveru a FTPServer se stará o spouštění ostatních závislých programových komponent. Jeho podstatnou metodou je Start, která spustí síťové naslouchání příchozím klientům v samostatném vlákně. Každé nové spojení je zpracováno novou instancí třídy FTPServerAcceptedClient. FTPServer také registruje všechny přístupné příkazy.

FTPServerAcceptedClient zodpovídá za zpracování komunikace na úrovni TCP. K realizaci požadavků používá FTPConnection, která již řeší samotné spouštění příkazů. Zároveň si FTPConnection sebou nese informace o stavu spojení a to pomocí FTPConnectionState. Jedinným dosud nezmíněným přímým členem namespacu FTP je FTPReplyChannel, který spravuje datové spojení s klientem, což obnáší pasivní a aktivní FTP komunikaci – podle toho, co klient zvolil.

V namespacu FTP jsou ještě dva namespacy: Commands a Extension. Extension obsahuje pouze pomocné nástroje, které rozšiřují standardní třídy .NET o novou funkčnost. Zde je nutno poznamenat, že možnost rozšiřování původních tříd využívá speciální syntaxe C#, která se nazývá extension methods a není společná všem jazykům. Na toto je nutno brát zřetel, pokud by vznikl požadavek přepisovat projekt do jiného programovacího prostředí.

Namespace FTP.Commands

Namespace Commands obsahuje implementace všech FTP příkazů. Dva členové zde figurují jako základní. Jedním je FTPCommandRegistry, který eviduje všechny ostatní členy a propojuje instanci třídy nějakého příkazu s řetězcovým příkazem tohoto příkazu (takže například „USER“ a FTPCommandUser). Všechny třídy, které implementují nějaký FTP příkaz jsou odvozeny od druhého základního člena – FTPCommandAbstract. Ten definuje rozhraní, který se od každé třídy pro příkaz

vyžaduje a provádí základní zpracování. Většina příkazů je přímo odvozených, výjimkou jsou například FTPCommandList a FTPCommandNlst, kteří se odvozují od FTPCommandAbstractList, protože jejich funkčnost je velmi podobná.

Rozšiřující příkazy nalezneme ve třídách FTPCommandHist, FTPCommandAsRel, FTPCommandAsNew, FTPCommandAsAuto, FTPCommandMeth, FTPCommandDate a mohli bychom poznamenat i standardní rozšíření FTPCommandMlsd. Jejich implementace je v celku přímočará a není nutno ji nijak komentovat.

Za zmínku však jistě stojí úprava příkazů FTPCommandRetr a FTPCommandStor. FTPCommandRetr je ještě vcelku triviálně modifikovaná, protože jediná věc navíc, kterou zohledňuje, je nastavené datum pro spojení. V případě FTPCommandStor je však problematika složitější.

Návrh protokolu FTP nepočítal s tím, že by po uploadu souboru na server docházelo k nějakému dalšímu zpracování, které by trvalo nezanedbatelný čas. FTP klienti při přenosu souboru ukazují průběh uploadu, vycházejí ovšem jenom z toho, kolik bytů již bylo odesláno. Je zřejmé, že v případě tohoto projektu bude doba zpracování přeneseného souboru významná. Zde narážíme na netriviální problém, jak udržet komunikaci s klientem po dostatečně dlouhou dobu a zároveň rozumně předávat zprávy o zpracování. Jako nejvhodnější metoda se jevilo použití víceřádkové syntaxe přenosu FTP stavových zpráv. Ukazuje se, že například FileZilla FTP klient je ochoten udržovat spojení tak dlouho, dokud se tato víceřádková zpráva nedokončí. Někteří další FTP klienti vykazovali podobné chování. Není však cílem zachovat úplnou kompatibilitu s existujícím FTP, podařilo se však najít cestu, jak toto relativně uspokojivě umožnit.

FTPCommandStor pro zpracování používá další vlákno, které realizuje samotný výpočet a z hlavního vlákna se pouze dotazuje na jeho průběh a v pravidelném časovém intervalu odesílá informace klientovi. Jakmile je zpracování hotové, ukončí se i tato víceřádková stavová zpráva. Toto je důvod, proč se doporučuje při komunikaci s LCS-FTP-Server používat FTP klienta, který zobrazuje stavové zprávy.

6 Uživatelská dokumentace programu

Uživatel se setká s aplikací ve dvou momentech. Jednak při samotné správě běžícího procesu, která je plně dostupná pomocí standardního okenního GUI. A potom při síťové komunikaci s touto aplikací přes FTP protokol (resp. jeho rozšíření pro tento projekt).

Aplikace se spouští hlavním programovým modulem LCS-FTP-Server.exe, který zobrazí uživateli hlavní ovládací okno serveru.

6.1 Serverová část s GUI

Hlavní okno je rozděleno do tří logických částí: v horní části jsou ovládací tlačítka, ve střední části je okno se zprávami o běhu serveru a nakonec ve spodní části jsou informace o připojených uživatelích. Mějme na paměti, že jeden uživatel se může naráz připojit vícekrát a tolikrát také bude ve spodní části zobrazen.

V ovládací části okna jsou nabízeny uživateli čtyři možnosti:

1. spustit instanci serveru
2. zastavit spuštěnou instanci serveru
3. spravovat uživatele, kteří smějí se serverem pracovat
4. měnit nastavení serveru

V rámci procesu nelze spustit více instancí serveru. Pokud chcete změnit na jaké IP adrese a portu bude server naslouchat příchozím připojením, učiňte tak po stisknutí tlačítka Nastavení.

Samozřejmostí je, podobně jako u standardního FTP serveru, že jsou obsluhováni pouze uživatelé, kteří mají vytvořený svůj účet. Každý uživatel má své přihlašovací jméno, heslo a složku, která mu slouží jako úložiště pro jeho data. Důrazně se doporučuje, aby žádný uživatel neměl jako svoji kořenovou složku zvolenu fyzickou podsložku úložiště jiného uživatele. Je to samozřejmě možné, ale nic nebrání uživateli s přístupem k nadřazené složce ve smazání libovolné podsložky, a to včetně té, která slouží jako úložiště jiného uživatele.

Uživatelské účty

Správu uživatelských účtů otevřeme stisknutím tlačítka Uživatelé. V otevřeném okně se zobrazí seznam uživatelů s účtem na serveru. V pravé části okna pak uvidíte podrobnosti vztahující se k vybranému uživateli ze seznamu. Pokud provedeme změny v pravé části, je nutné je uložit pomocí tlačítka Nastavit, které se objeví jakmile k nějakým změnám dojde. Pokud změny neuložíme a vybereme v seznamu jiného uživatele, či pokud zavřeme okno se správou uživatelů, změny se ztratí. Pro vytvoření nového uživatelského účtu stiskněme tlačítko Nový uživatel.

Pro odstranění uživatele proveďme toto: Nejprve uživatele vyberme v seznamu a pak stiskněme tlačítko Odstranit uživatele.

Pokud je server nakonfigurován podle přání uživatele, stačí ho již pouze spustit a používat. Učiníme tak stiskem tlačítka Start. Pro jeho zastavení stiskněme tlačítko Stop. Pokud uzavřeme hlavní ovládací okno serveru, bude ukončena i činnost serveru.

6.2 Ovládání serveru přes rozšířené FTP

Uživatel se asi se serverem nejčastěji setká právě přes FTP protokol. V roli klienta pak oproti běžným příkazům FTP bude mít možnost použít i některé rozšiřující příkazy. Standardní příkazy FTP zde nebudeme popisovat, jednak se s nimi uživatel obvykle ani nesetká, protože mu je zprostředkovává jeho FTP klient, jednak jsou již dokumentovány jinde. Při používání LCS-FTP-Serveru se doporučuje používat klienta, který zobrazuje stavové zprávy ze síťové komunikace. Takovým klientem je například FileZilla. Není to nutné, ale je to jediný způsob jak uživatel může vidět informace o dodatečném zpracování souboru na straně serveru.

Proměnné v průběhu relace

K porozumění funkce některých příkazů je vhodné si ujasnit, jaké proměnné charakterizují dané síťové sezení se serverem. Samozřejmě je to přihlášený uživatel, od čehož se odvíjí používané úložiště. Je to pak ale i používané *datum spojení* – to hraje významnou roli při vypisování obsahu adresáře a dalších příkazech. Při výpisu adresáře se vždy vypíší poslední verze souborů do data spojení. Naopak při mazání souborů se mažou tyto vypisované verze a všechny verze na nich závislé. Při stahování souboru

se vždy stahuje poslední verze do data spojení, při ukládání souborů se však vždy navazuje na reálně poslední verzi, nová verze se vždy ukládá s reálným datem, tj. bez ohledu na proměnnou datum spojení.

Další proměnná spojení je způsob ukládání nových verzí. Buďto se může ukládat každá verze jako originální, tedy nezávislá na předchozích, nebo vynuceně jako závislá na reálně poslední verzi, či je možno nechat rozpoznání vhodné volby na serveru – toto je pak nazýváno odhad podobnosti.

Poslední dvě proměnné pak dovolují uživateli zvolit algoritmus pro odhad podobnosti verzí a pro způsob zpracování těchto podobností a uložení nové verze v podobě, která využívá obsahu verzí předchozích. Jsou to tedy dva algoritmy, které si může uživatel vybrat – jeden pro odhad, druhý pro zpracování.

Nutno poznamenat, že při každém přihlášení jsou všechny proměnné inicializovány na výchozí hodnoty. Pokud používáme FTP klienta, který je schopen přenášet soubory ve více vláknech, je nutné si uvědomit, že tato nová vlákna nebudou mít proměnné prostředí stejné, pokud je v nich explicitně nenastavíme. Doporučuje se tedy vynutit si používání jediného vlákna při komunikaci s LCS-FTP-Serverem.

Rozšíření protokolu FTP

Nyní se podívejme na jednotlivé příkazy.

Seznam rozšiřujících příkazů se uživateli zobrazí standardním způsobem po odeslání příkazu FEAT.

Prvním příkazem je HIST. Jeho syntaxe je HIST *název souboru* a tento příkaz uživateli zobrazí seznam všech verzí dostupných k danému souboru. Každá řádka seznamu má tento formát: [ID verze] [ID rodičovské verze] [velikost verze] [datum a čas vzniku]. Rodičovskou verzí se zřejmě myslí ta verze, od které se odvozuje obsah. Pokud je verze uložena jako originální, bude jako ID rodiče uvedena -1.

Dalšími třemi jsou ASNEW, ASREL a ASAUTO. ASNEW vynutí ukládání nových verzí jako originálních, ASREL naopak vynucuje vždy odvození od verze předchozí a ASAUTO se rozhodne na základě odhadu podobnosti a očekávaného ušetření prostoru.

Příkaz DATE nastaví proměnnou datum spojení. Buď lze použít příkaz DATE NOW, který znamená, že datum spojení bude vždy aktuální okamžik (tedy reálný čas serveru), nebo DATE *den.měsíc.rok hodina:minuta:sekunda*, čímž nastavíte jeden konstantní okamžik.

Poslední rozšiřující metodou je METH, která volí příslušný algoritmus jednak pro odhad podobnosti a jednak pro zpracování souborů. Pro každý z úkonů můžete nastavit jiný algoritmus. Každý algoritmus implementovaný v LCS-FTP-Server byl naprogramován ve dvou variantách: rychlejší a méně přesná varianta pro odhad, přesnější a náročnější varianta pro zpracování. Seznam algoritmů se ukáže při zadání příkazu METH INFO. METH E *název algoritmu* pak nastaví daný algoritmus pro odhad. METH P *název algoritmu* nastaví daný algoritmus pro zpracování. Příklad použití je např takovýto: METH E RK. Toto nastaví upravenou verzi Rabin-Karpova algoritmu pro odhad podobnosti verzí. Algoritmus Rabin-Karpa obvykle slouží pro vyhledávání vzorků v textu a v tomto projektu byl přizpůsoben právě pro hledání podobností dvou posloupností dat. Je výchozím algoritmem pro odhad. Naopak algoritmus Kua a Crosse, je výchozím algoritmem pro zpracování.

V seznamu příkazů uvedených v odpovědi na FEAT je uveden i příkaz MLSD, toto je však standardní rozšíření FTP podle RFC.

7 Testování

Každá z implementací algoritmů, které jsme zmínili v části 4, byla několikrát testována na stejné sadě vstupů a na stejném hardware.

Pro účely testování byl použit osobní notebook HP Compaq 6710s s dvoujádrovým procesorem Intel Core 2 Duo T7300. Operační paměť stroje je 2GB. Použitý operační systém jsou Microsoft Windows Vista Business s nainstalovaným .NET Framework 3.5. Během testování běžely v pozadí pouze obvyklé systémové služby. Jedinými spuštěnými programy byly FileZilla FTP Client ve verzi 3.2.2.1 a aplikace LCS-FTP-Server.

7.1 Testovací vstupy

Bylo použito 11 vstupů, přičemž každý sestával z dvojice souborů – z původního souboru a z dalšího, u kterého se předpokládá rozumná míra podobnosti s prvním. Soubory měly různý charakter a byly buď uměle vytvořeny, nebo pocházely z volně přístupných zdrojů.

Vstup „adr“

Tento vstup odpovídá vývoji obsahu databáze Microsoft SQL Server 2008. Uvnitř obou souborů je pouze jedna datová tabulka, obsahující osobní údaje osob. Ačkoliv se jedná o uměle vygenerovaný vstup, byla uvnitř databáze použita různá existující česká jména a poštovní směrovací čísla. Kromě nich obsahuje databázová tabulka i jiné sloupce. Druhý soubor vznikl z prvního sadou vkládání a úprav řádků.

Vstup „bibleen“

Jedná se o data z volně přístupných zdrojů. Oba soubory jsou anglickým překladem bible v elektronické podobě, a sice jako čistý neformátovaný text. Pravděpodobně je mezi nimi silná obsahová podoba, soubory však od sebe nebyly nijak přímo odvozeny.

Vstup „bswiki“

Tento vstup odpovídá xml výpisu databáze článků bosenské wikipedie. Jedná se o data z veřejně přístupného zdroje. První soubor je stav databáze k 6.3.2010, druhý k 2.4.2010.

Vstup „cswikiquote“

Jedná se o xml výpis databáze citátů české wikipedie. Data pochází z veřejně přístupného zdroje. První soubor je stav databáze k 5.3.2010, druhý k 2.4.2010.

Vstup „dmoz“

Toto je výpis databáze kategorizace stránek v internetovém katalogu Dmoz. Data jsou veřejně přístupná a první soubor odpovídá stavu k 6.9.2005, druhý pak k 19.1.2010.

Vstup „go“

Tento vstup je SQL výpisem databázové tabulky projektu Human Genome Project. První soubor je k lednu 2009, druhý k dubnu 2010. Data jsou volně dostupným zdrojem.

Vstup „hewiki“

Jedná se o výpis databáze nadpisů článků hebrejské wikipedie. Data pochází z volně dostupných zdrojů. První soubor je k 6.3.2010, druhý k 28.4.2010.

Vstup „pic“

Vstupem je obrázek ve formátu PNG. První soubor je fotografie z digitálního fotoaparátu. Druhý je potom první soubor upravený zkopírováním výřezu obrázku.

Vstup „regvoz“

Tento vstup je výpisem databáze registru vozidel v České Republice. Jedná se o volně dostupný zdroj. První soubor odpovídá stavu registru k červenci 2007, druhý k září 2009.

Vstup „zfdoc1“

Tato dvojice souborů jsou různé verze dokumentace projektu Zend Framework. První je dokumentací verze 1.7.1, druhý je pro verzi 1.9.0. Jedná se o data z volně dostupného zdroje. Data jsou ve formátu HTML, ovšem spojeny v archiv v programu tar.

Vstup „zfdoc2“

Tento vstup rovněž tvoří různé verze dokumentace projektu Zend Framework. První soubor je dokumentací verze 1.10.0, druhý je pro verzi 1.10.2. Jedná se o data z volně dostupného zdroje. Data jsou ve formátu HTML, ovšem spojeny v archiv v programu tar.

7.2 Naměřené výsledky

Prvním z výstupů z testů jsou průměrné časy potřebné k odhadu shody dvou verzí souborů v rámci jednoho testu. Průměrným časem zde označujeme střední hodnotu z naměřených údajů. Poznamenejme, že do naměřených časů se nepočítal samotný FTP přenos. Měření bylo realizováno na straně serveru a bylo spuštěno teprve v momentě, kdy byl daný soubor přenesen.

Údaje uvedené v tabulkách jako „-“ znamenají, že tyto hodnoty jsou příliš vysoké, než aby se jimi ještě mělo smysl zabývat v porovnání s ostatními algoritmy. Samotný výsledek odhadu nemá smysl přímo uvádět. Odhad podobnosti vydaný algoritmem velmi těsně koresponduje s efektivitou samotného ukládání. Výsledek odhadu by pak nedával významnější informaci.

	adr	bibleen	bswiki	cswiki	dmoz	go	hewiki	pic	regvoz	zfdoc1	zfdoc2
BestnextWindowed	34,79	7,96	423,83	131,24	90,32	128,08	12,37	21,17	46,53	13,86	43,89
Simple	626,59	330,24	-	-	-	-	355,71	138,35	-	500,26	-
Enhanced	703,93	364,76	-	-	-	-	353,62	143,14	-	531,04	-
HuntSzymanski	315,82	82,12	396,01	563,02	729,52	545,04	129,76	3,23	931,85	76,19	253,12
KuoCross	41,32	7,82	137,59	76,48	63,21	49,14	21,96	0,69	104,95	36,91	25,63
RabinKarp	0,36	0,32	6,40	44,40	2,29	1,75	0,14	15,90	0,65	0,33	1,05
SA	17,91	13,83	299,50	127,71	153,35	118,09	9,28	6,69	35,39	17,27	40,84
ESA	18,64	13,53	300,35	130,44	153,47	118,58	9,26	6,74	38,30	17,28	40,84
SAD	24,29	10,48	328,42	132,92	169,91	119,38	11,35	5,60	124,84	30,18	49,86

Tabulka 7.2.1.: Časy potřebné pro odhad podobnosti, údaje jsou uvedeny v sekundách

Za zajímavější údaj lze považovat průměrný čas potřebný ke zpracování podobnosti dvou souborů. Většina algoritmů nevykazovala žádné podstatné rychlostní výkyvy od své střední hodnoty. Výjimkou byli algoritmy BestnextWindowed a RabinKarp, které při svém běhu využívají generátor pseudonáhodných čísel. Odhlédneme-li tedy od faktu, že se nejedná o skutečná náhodná čísla, nelze pak považovat tyto algoritmy za deterministické. Tím se právě vysvětlují i odchylky od střední hodnoty, které však nikdy nepřesáhly řád několika málo procent střední hodnoty.

	adr	bibleen	bswiki	cswiki	dmoz	go	hewiki	pic	regvoz	zfdoc1	zfdoc2
BestnextWindowed	37,31	9,69	307,91	134,70	97,37	131,18	12,70	21,26	47,00	15,51	44,20
Simple	637,77	337,74	-	-	-	-	360,38	138,91	-	503,56	-
Enhanced	723,07	398,38	-	-	-	-	365,17	143,12	-	542,85	-
HuntSzymanski	409,55	92,46	1230,51	566,83	738,07	562,61	140,38	3,42	1006,17	83,45	260,00
KuoCross	46,88	7,91	163,18	92,33	76,66	56,35	23,29	1,34	113,48	9,27	29,30
RabinKarp	1,36	0,62	12,99	48,74	5,56	3,82	0,62	18,34	1,64	1,02	2,53
SA	23,28	15,62	310,33	143,04	156,15	118,93	10,66	6,85	43,89	20,87	47,09
ESA	23,28	15,72	309,99	139,03	158,61	119,68	10,58	6,84	43,86	21,15	39,51
SAD	25,48	12,34	355,16	139,44	175,10	122,54	12,41	5,78	59,87	21,00	53,55

Tabulka 7.2.2.: Časy potřebné ke zpracování podobných souborů, údaje jsou uvedeny v sekundách

U algoritmů nás však nezajímá pouze rychlost jejich běhu, ale především dosažené výsledky. Jediným rozumným a zároveň nabízejícím se měřítkem je toto: Kolik procent dat jsme ušetřili oproti stavu, kdy bychom uložili oba dva soubory tak jak jsou, bez ohledu na redundanci? Od součtu velikostí obou souborů testu tedy odečteme součet velikostí všech dat, které jsou nutné pro jejich uložení ve zpracovaném stavu pro server. Počítají se sem jak uložené bloky shody a rozdílu, tak jejich popis. Co naopak nezahrnujeme, je XML soubor, který popisuje posloupnost uložených verzí. Jednak je velmi malý, a navíc obsahuje spíše informační údaje než-li údaje nutné k ukládání v aplikaci LCS-FTP-Server.

Většina zkoumaných algoritmů se chová deterministicky, ale opět byly výjimkou BestNext a RabinKarp, kvůli kterým budeme opět používat střední hodnotu z naměřených údajů.

	adr	bibleen	bswiki	cswiki	dmoz	go	hewiki	pic	regvoz	zfdoc1	zfdoc2
BestnextWindowed	11,81	-0,97	-1,60	1,56	-1,13	-1,06	-0,88	38,06	-4,18	-1,26	0,55
Simple	45,51	0,07	-	-	-	-	1,24	38,06	-	0,25	-
Enhanced	45,51	0,06	-	-	-	-	26,64	38,06	-	0,45	-
HuntSzymanski	**45,61**	0,03	17,15	26,36	0,42	-0,62	**47,81**	38,06	**17,88**	0,67	10,54
KuoCross	44,55	0,05	0,24	5,31	-0,02	-0,05	9,50	38,06	0,25	0,17	2,24
RabinKarp	0,21	0,09	0,06	7,94	0,00	0,01	0,30	36,47	0,02	0,01	0,38
SA	38,91	0,57	**40,04**	32,88	0,06	**1,38**	35,80	**38,07**	6,59	**2,12**	**37,71**
ESA	38,91	0,57	**40,04**	32,88	0,06	**1,38**	35,80	**38,07**	6,59	**2,12**	**37,71**
SAD	38,91	**0,91**	**40,04**	**33,03**	0,08	1,22	35,80	38,06	6,33	1,53	36,67

Tabulka 7.2.3.: Úspora dat v procentech oproti nezpracovanému stavu

7.3 Srovnání

Analyzujme tedy uvedené výsledky. Na první pohled je zřejmé, že algoritmus BestnextWindowed postupuje vcelku nešetrně. V důsledku tak dokonce navyšuje množství dat potřebných k uložení souboru. V algoritmu je implementován požadavek na délku ukládaného bloku, ale je pravděpodobně příliš malý, než aby měl algoritmus takto šanci uspět.

Poměrně překvapivý je i fakt, že algoritmus Enhanced často pracoval pomaleji než algoritmus Simple. Lze předpokládat, že samotné zarovnávání posloupností k sobě není tak časově náročné, aby toto způsobilo. Zřejmě se v něm totiž provádí méně operací, než během kvadratického výpočtu matice. Algoritmus však v důsledku pracuje s jinými dvojicemi úseků posloupnosti, a je tak nucen provádět odlišný výpočet (ač algoritmicky téměř totožný). Toto tvrzení se opírá i o fakt, že algoritmy dosahovali různých výsledků, co se týče úspory dat. Zřetelné je to zejména v případě testu hewiki. Zde velmi naléhavě figuruje potřeba okna k sobě lépe zarovnávat, což pak dává algoritmu Enhanced jasnou převahu.

Neočekávaný úspěch zaznamenal algoritmus HuntSzymanski na vstupu regvoz. Úspěch je však vyvážen takřka nepoužitelnou dobou zpracování. Je tedy nutné se ptát, proč tak jasně převážil například nad algoritmem KuoCross, když ten je jeho vylepšením. Odpověď není nijak komplikovaná. Uvědomme si, že algoritmus KuoCross nezkoumá tolik párů shody, ani si je neukládá. Tím si zřejmě oproti HuntSzymanski ušetřil hodně času. Naopak však není schopen tak dobře rekonstruovat bloky shody, protože nemá uložené potřebné páry shody. Dominantní páry shody v tomto případě neleželi dostatečně vhodně na společných podřetězcích, aby algoritmus KuoCross byl ve výhodě.

Pozastavme se ještě u silně heuristicky založeného algoritmu RabinKarp. Ten vykazuje často až extrémně krátké časy, což by ho stavělo do výsadní pozice oproti ostatním. Dosažené úspory jsou však poměrně nevalné. Dokonce se algoritmus chová tak, že je tím pomalejší, čím podobnější si soubory jsou. To je viditelné především v případě testů cswiki a pic, kdy algoritmus běžel nejpomaleji, zároveň však už dosáhl nezanedbatelné úspory.

Uvedené údaje tedy zatím nabídli zajímavý materiál ke zkoumání, klaďme si však otázku, jak bychom nejlépe algoritmy porovnali. Máme zde dva faktory, které chceme zohlednit: Jednak je to čas potřebný ke zpracování, jednak je to dosažená úspora dat. Podívejme se tedy právě na poměr úspory dat ku spotřebovanému času.

	adr	bibleen	bswiki	cswiki	dmoz	go	hewiki	pic	regvoz	zfdoc1	zfdoc2
BestnextWindowed	0,19	-0,06	0,00	0,01	-0,01	0,00	-0,04	1,07	-0,05	-0,05	0,01
Simple	0,04	0,00	-	-	-	-	0,00	0,16	-	0,00	-
Enhanced	0,04	0,00	-	-	-	-	0,04	0,16	-	0,00	-
HuntSzymanski	0,07	0,00	0,01	0,03	0,00	0,00	0,20	6,67	0,01	0,00	0,02
KuoCross	0,57	0,00	0,00	0,03	0,00	0,00	0,24	**17,09**	0,00	0,01	0,05
RabinKarp	0,09	**0,08**	0,00	0,10	0,00	0,00	0,29	1,19	0,01	0,00	0,09
SA	**1,00**	0,02	**0,08**	**0,14**	0,00	0,01	2,01	3,33	**0,09**	0,06	0,48
ESA	**1,00**	0,02	**0,08**	**0,14**	0,00	0,01	**2,03**	3,34	**0,09**	0,06	**0,57**
SAD	0,92	0,04	0,07	**0,14**	0,00	0,01	1,73	3,95	0,06	0,04	0,41

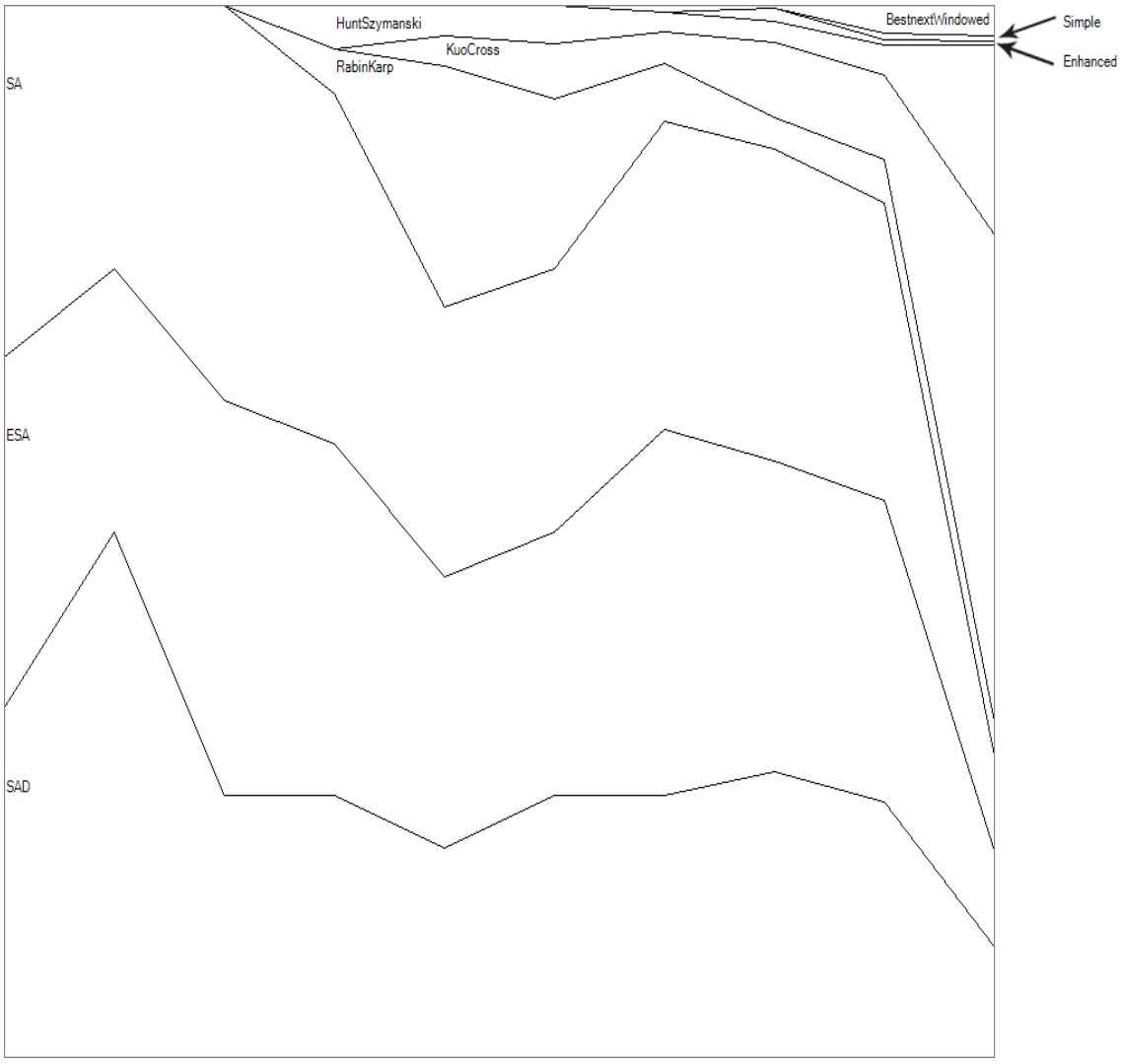
Tabulka 7.3.1.: Poměr promile úspory dat ku počtu minut nutných ke zpracování.

Použití tohoto hlediska zásadně ovlivňuje pomyslné pořadí úspěšnosti algoritmů. Všimněme si toho například v případě testu pic, kdy všechny algoritmy daly velmi podobný výsledek, ale KuoCross dokončil výpočet nejrychleji. Z intuitivního hlediska je toto srovnání pak mnohem důležitější než samotná úspora dat.

V tomto srovnání je pak algoritmus SAD oproti SA a ESA ještě méně úspěšný, než byl předtím. Pouze ve dvou případech dokázal uložit data úsporněji než všechny ostatní algoritmy. Nikdy se však nejednalo o nijak zásadní přínos. Teoreticky lze namítat, že žádný z testů nedovolil vyniknout přednostem tohoto algoritmu. Od reálných dat však rovněž nelze takové vlastnosti očekávat. Dalším možným faktorem jsou možná nevhodně zvolené velikosti buněk a případně i požadavky na délky společných podřetězců. Voleb těchto parametrů je však mnoho a nelze předem nic odhadnout o jejich dopadu.

Pro člověka však bývá jednodušší vnímat větší množství informací z jejich grafického vyjádření. Předložená data ovšem dost dobře neumožňují přímočaré umístění do typicky používaných grafů. Z těchto důvodů je tedy v této práci použito poněkud atypické grafické znázornění. Protože poměr promile úspory dat ku spotřebovanému množství minut se různě skokově mění, nelze triviálně použít testy jako označení jedné z os.

Následný graf vznikl takto: Pro každý algoritmus jsme vzali vzestupně uspořádané údaje o poměru úspory dat k času, čímž jsme zahodili informaci o náležitosti údaje ke konkrétnímu testu. Přesto však každý test jistým způsobem na ose figuruje. Graf, coby obdélník, je protnut křivkami, které jsou v každém bodu zlomu od sebe vzdáleny tak, jak odpovídá vzájemný poměr naměřených hodnot vzhledem k celku. Pokud by tedy každý algoritmus dosáhl stejných hodnot, byl by obdélník horizontálně rozdělen na tolik částí, kolik je algoritmů.



Obrázek 7.3.2.: Poměrové srovnání jednotlivých algoritmů.

Plocha přiřazená algoritmům hrubě odpovídá intuitivnímu měřítku, podle kterého bychom je mohli chtít srovnávat. Ovšem zvláště v pravé části grafu je toto mírně zavádějící. Nezaměřujme se tedy v obrázku ani tolik na plochu přiřazenou jednomu algoritmu, ale spíše na svislý rozestup v místech, kde se křivky lomí. Zřetelně pak vidíme úspěšnost algoritmu KuoCross úplně ke konci, kde je mu přiřazena zhruba polovina z celé výšky grafu. Dobře také vidíme místo, kde algoritmus RabinKarp nebyl možná příliš efektivní v ohledu úspor, ale za to zvládl svoji práci odvést rychle. Obzvláště názorné jsou nelichotivé výsledky pro algoritmy BestnextWindowed, Simple a Enhanced.

Obrázek nesrovnává údaje vzhledem k jednomu testu, ale nejlepší údaj s nejlepšími, druhý nejlepší s druhými nejlepšími a tak dále. V tomto směru tedy dovoluje lépe nahlížet na výsledky algoritmů, které závisejí na různých a těžko graficky popsatelných podmínkách. Toto srovnání jistě není ideální, ale nelze klást takový požadavek na snadno pochopitelné dvojrozměrné grafické vyjádření vzhledem k množství dat.

8 Závěr

V této bakalářské práci jsme tedy srovnali několik známých algoritmů, leč mírně modifikovaných pro naše účely. Rovněž jsme se pokusili nabídnout vlastní variantu, která může zužitkovat poznatky o těch již existujících. Ve světle výsledků porovnání lze konstatovat, že tato vlastní varianta si vedla dobře v porovnání s ostatními.

Přesto jsme nedosáhli použitelnosti v reálném čase, ačkoliv takto silný požadavek jsem si za cíl nekladli. Naměřené časy běhů algoritmů naznačují, že by bylo nutné buď počítat se značným zdržením během ukládání, či se naopak spokojit s menší efektivitou. Jediný algoritmus, který by pravděpodobně byl schopen svůj úkol plnit v reálném čase, je implementován jako metoda označená RabinKarp. Ta však nedosahuje požadovaných výsledku vzhledem k množství uspořených dat.

Téma zůstává otevřené pro řadu dalších možností a zlepšení. Například algoritmus SAD se v jistém množství případů ukázal být buď o něco rychlejší, či o něco efektivnější než jeho přímá konkurence, algoritmus SA. V součtu však vždy druhé hledisko převážilo první, jak je naznačeno ve srovnání v části 7.3. Mohli bychom se například pokusit modifikovat jeho funkci buď v jeho parametrech, nebo odvážněji zkusit zapojit do jeho činnosti i jiný algoritmus.

V tomto směru algoritmus RabinKarp vykazoval úspěšnost odhalovat málo podobné posloupnosti, což by mohlo dovolit dřívější ukončení nerentabilních výpočtů. Je zde však značné riziko, že by takto označil i posloupnosti, které si naopak podobné jsou. Podobně bychom mohli zkusit obětovat část času na důkladnější zpracování kratších úseků posloupností. Pro tyto účely by pak mohl připadat v úvahu i algoritmus HuntSzymanski či KuoCross.

Celkově vzato nabízí problém nejdelší společné podposloupnosti zajímavé hledisko v otázce efektivní archivace dat. Ačkoliv vlastní navržené algoritmy nedosáhli aplikovatelnosti v reálném čase, rozumně obstáli proti konkurenci. Dosažené výsledky tedy vedou k přesvědčení, že si toto téma zaslouží další zkoumání, protože možných směrů pro pokračování je mnoho.

9 Příloha

K této bakalářské práci je přiloženo doprovodné CD. Obsahuje zdrojový kód programu LCS-FTP-Server a jemu příslušnou referenční dokumentaci. Dále jsou na CD jednotlivé vstupní testy a text této bakalářské práce.

Na CD jsou tyto adresáře:

- doc – referenční manuál k LCS-FTP-Server
- source – zdrojový kód LCS-FTP-Server
- tests – vstupní testy
- text – zde je uložen text této bakalářské práce

Literatura

1. Hirschberg, D.S. (1975): A Linear Space Algorithm for Computing Maximal Common Subsequence; Communications of the ACM June 1975 Volume 18 Number 6 Princeton University 1977.

2. Hunt, J. W., Szymanski, T. G. (1977): A Fast Algorithm for Computing Longest Common Subsequences, Communications of the ACM, Vol. 20, nr 5, pages 350-353, may 1977.

3. Kuo, S., Cross, G. R. (1989).: An Improved Algorithm to Find the Length of the Longest Common Subsequence of Two Strings, ACM SIGIR Forum, Spring / Summer 1989, Vol. 23, No. 3-4, str. 89-99.

4. Bergroth, L. (2005): Utilizing Dynamically Updatet Estimates in Solving Longest Common Sequence Problem; Springer Berlin / Heidelberg 2005.

5. Weiner, P. (1973).: Linear pattern matching algorithm, 14th Annual IEEE Symposium on Switching and Automata Theory, str. 1–11.

6. Manber, U., Myers, G. (1991).: Suffix arrays: a new method for on-line string searches, SIAM Journal on Computing, Volume 22, Issue 5 (October 1993), str. 935–948.

7. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K. (2001).: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications, proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (July 01 - 04, 2001). A. Amir and G. M. Landau, Eds. Lecture Notes In Computer Science, vol. 2089, Springer-Verlag, London, str. 181-192.

8. Bergroth, L., Hakonen, H., Raita, T. (2000).: A Survey of Longest Common Subsequence Algorithms, proceedings of the Seventh international Symposium on String Processing information Retrieval (Spire'00) (September 27 - 29, 2000), IEEE Computer Society, Washington, DC, 39.

9. Kärkkäinen, J., Sanders, P. (2003).: Simple linear work suffix array construction, proceedings of the 30th international Conference on Automata, Languages and Programming (Eindhoven, Nizozemí, Červen 30 - Červenec 04, 2003), J. C. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, Eds. Lecture Notes In Computer Science. Springer-Verlag, Berlin, Heidelberg, 943-955.