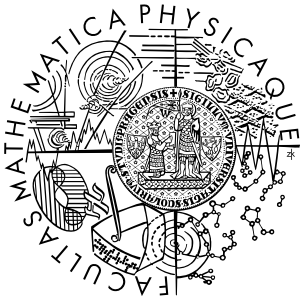
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Petr Paščenko

Verzovaná komprese textových dokumentů

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Michal Žemlička, Ph.D.
Studijní program: Informatika, Obecná informatika

2007

Rád bych poděkoval doktoru Michalu Žemličkovi, vedoucímu této práce, za všestrannou radu a pomoc při jejím vytváření.

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

v Praze dne 29. května 2007 Petr Paščenko

Obsah

1 Úvod **5**
1.1 Zadání práce 5
1.2 Úkoly a cíle práce 5

2 Verze textového souboru a práce s nimi **7**
2.1 Úvod 7
2.2 Formulace problému 8
2.3 Algoritmus Diff 9
2.4 Diff trochu jinak 12
2.5 Víceúrovňový diff 16
2.6 Shrnutí 21

3 Správa Archivu **22**
3.1 Úvod 22
3.2 Požadavky na archiv 22
3.3 Formát souboru s archivem 24
3.4 Vnitřní uspořádání archivu 26
3.5 Operace s archivem 28
3.6 Testování archivu 31
3.7 Shrnutí 33

4 Aplikace DiffArchive **34**
4.1 Vrstevnatý model 34
4.2 Knihovny projektu DiffArchiv 35
4.3 Shrnutí 39

5 Závěr **41**

Literatura **43**

Název práce: Verzovaná komprese textových dokumentů
Autor: Petr Paščenko
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Michal Žemlička, Ph.D.
e-mail vedoucího: michal.zemlicka@mff.cuni.cz

Abstrakt: v předložené práci studujeme efektivní správu různých verzí textových dokumentů. Výsledkem práce je aplikace pro řízení takového archivu. Aplikace umožňuje maximální využití podobnosti verzí souborů pro dosažení minimální redundance uložené informace. Aplikace využívá metody rozdílového porovnávání souborů (Diff), obohacenou o rozlišování menších změn — oproti původnímu Diffu může efektivně zachytit i změny na úrovni slov či znaků. Citlivost diffu se dynamicky mění na různých úsecích textového dokumentu v závislosti na hustotě a charakteru provedených změn. To zajišťuje dostatečnou přesnost pro úspornou velikost archivu v kombinaci se stále velmi dobrou rychlostí aplikace. Pro další zmenšení velikosti archivu je využita komprese metodou zip pomocí volně šířitelné knihovny Info-ZIP.
Klíčová slova: diff, text, dokument, verze, archiv

Title: Versioning compression of text documents
Author: Petr Paščenko
Department: Department of Software Engineering
Supervisor: RNDr. Michal Žemlička, Ph.D.
Supervisor's e-mail address: michal.zemlicka@mff.cuni.cz

Abstract: In the work we study effective management of different versions of text files. Outcome of the work is an application managing such archives. The application is able to maximally exploit a similarity between versions of files to avoid redundancy of stored information. Application uses the method of Differential File Comparision (diff), improved by additional smaller dividing entities. Contrary to the the classical diff it recogizes also word and character differences. Precision of the diff vary along a text file according to density and character of differences. Due to that a smaller archive is made and moreover in shorter time. Extra reduction of the archive is achieved by the use of the zip compression method implemented by freeware library Info-ZIP.
Keywords: diff, text, document, version, archive

Kapitola 1

Úvod

1.1 Zadání práce

Cílem této práce je vytvořit aplikaci umožňující efektivně ukládat různé verze textových dokumentů do společného archivu.

Vytvořte aplikaci efektivně komprimující sady textových dokumentů. Zaměřte se zejména na efektivní ukládání různých verzí týchž dokumentů (využijte možnosti zaznamenávat pouze rozdíly mezi jednotlivými verzemi).

Výsledné řešení by se mělo skládat z aplikace pracující v příkazové řádce a z uživatelsky přívětivější aplikace (celoobrazovkové v textovém režimu, či grafické).

1.2 Úkoly a cíle práce

Takto formulované zadání klade před řešitele několik víceméně nezávislých úkolů, které se liší svým charakterem i obtížností.

Prvním z nich je obecné zvládnutí práce s archivem textových souborů: volba formátu archivního souboru, jeho vnitřního uspořádání a použitých kompresních metod. V úvahu je třeba vzít předpokládanou velikost archivu a počet uskladněných souborů. Klíčové pro správný návrh řízení archivu je vyvážení protichůdných požadavků na velikost archivu a rychlost

práce s archivem. Při hledání rozumného kompromisu je třeba brát ohled na předpokládané vyžití archivu a očekávaný charakter a četnost dotazů na jeho obsah.

Druhým úkolem je vytvořit metodu pro správu různých verzí téhož textového souboru. Předpokládáme obecný textový soubor, jehož autor provádí postupnou úpravu jeho obsahu. Často jde o drobné místně ohraničené změny, jejichž úhrnná informace je podstatně menší v porovnání s celým souborem. Lokalizace této informace a její efektivní uložení představuje netriviální cíl této práce, jehož dosažení umožňuje velmi podstatnou úsporu ve velikosti spravovaného archivu. Cílem této práce je nalezení takové metody a podrobné prozkoumání jejích vlastností.

Třetím a posledním úkolem této práce je vytvořit užitečnou funkční a uživatelsky přívětivou aplikaci realizující zadání práce v duchu výsledků předchozích dvou úkolů. Aplikace by měla využívat přiměřenou množinu nástrojů a hotových řešení s ohledem na rozumnou úroveň efektivity, modularity a budoucí přenositelnosti aplikace.

Kapitola 2

Verze textového souboru a práce s nimi

2.1 Úvod

Mějme textový soubor: ať již jde o text v přirozeném jazyce jako například dopis či knihu, zápis kódu nějakého programu, logovací soubor nějaké aplikace nebo třeba zdrojový soubor internetové stránky. Jde-li o alespoň trochu rozsáhlejší text, lze předpokládat, že nevznikl naráz, ale byl svým autorem či autory postupně vytvářen, rozšiřován, upravován, krácen, přepracováván a aktualizován — zkrátka, že jeho obsah se postupně měnil až dospěl ke své finální verzi.

Často bývá užitečné mít přehled o jednotlivých přechodných verzích dokumentu a moci se kdykoli ke kterékoli z nich vrátit, zjistíme-li, že cesta po níž jsme se vydali, je slepá. Leckdy také může mít sama informace o vývoji větší cenu, než jeho samotný výsledek.

Z tohoto a mnoha jiných důvodů stojí za to mít k dispozici archiv, do něhož by bylo možné jednotlivé verze ukládat a opět je získávat zpět. Překážkou může být velikost takového archivu. Pro rozsáhlý soubor může být velmi nesnadné udržovat velký počet jeho verzí, zvláště uvážíme-li, že jednotlivé verze se nemusí vzájemně příliš lišit. Cílem této kapitoly je představit metodu, která umožňuje efektivně uchovávat pouze informaci o změně souborů a vyhnout se tak zbytečnému ukládání stále týchž dat.

2.2 Formulace problému

Vezměme textový soubor o řádově stovkách až tisících řádků. Předpokládejme, že jednotlivé verze se od sebe liší ve spíše lokálních úpravách, což odpovídá běžnému způsobu práce s textovými soubory. Hledáme algoritmus splňující následující vlastnosti:

1. Ze dvou verzí dokáže extrahovat informaci o změnách provedených v nové verzi v porovnání s verzí starší.

2. Takto získanou informaci dokážeme uchovat ve formátu umožňujícím jeho ukládání a načítání z a do archivu. Preferován je textový formát pro svou vhodnost ke kompresi v archivu tvořeném textovými soubory.

3. Ze starší verze a informace o změně dokáže zpětně zrekonstruovat verzi novou.

4. Čas potřebný pro běh algoritmu bude umožňovat jeho pohodlné použití na rozumně velkých souborech odpovídajících běžné velikosti lidmi upravovaných textů (max. tisíce řádků) na obvyklém pracovním počítači.

5. Velikost informace o změnách bude v rozumné relaci s množstvím a rozsahem provedených změn.

Poslední bod seznamu vyžaduje rozvedení. Následující seznam klasifikuje nejobvyklejší druhy změn v textových souborech co do velikosti a míry změny souboru od nejrozsáhlejších po nejméně rozsáhlé.

- **Třída I: Kompletní přestavba souboru.**
  Nový soubor si s předchozím není vůbec podobný; došlo ke kompletní změně struktury souboru přidáním a odstraněním podstatné části obsahu nebo změně znakové sady. Změna zasahuje více než 50% řádků a na průměrném řádku bylo pozměněno více než 50% znaků.

- **Třída II: Úprava na úrovni řádků**
  Běžná větší úprava textu. Na několika místech souboru přibyly sekvence nových řádků, na jiných místech byly řádky vypuštěny. Občas se změnil text některých řádků a změna zasáhla více než 50% znaků na těchto řádcích. Celkově je změnami zasaženo méně než 25% textu a převažují spíše větší změny zahrnující více po sobě následujících řádků.

- **Třída III: Stylistické úpravy textu**
  Došlo převážně k většímu počtu menších změn. Převažujícím typem je změna řádku přidáním, vypuštěním či nahrazením slova nebo několika slov. Zřídka došlo ke změně více řádků po sobě a na jednotlivých řádcích převažují změny zasahující méně než 50% znaků.

- **Třída IV: Drobné znakové úpravy**
  Jde o nejjemnější rozpoznávanou úpravu. Text obou verzí se na první pohled neliší, pouze občas přibyla čárka, první písmeno se změnilo z malého na velké, místo proměnné i je ve volání funkce proměnná j. Počet zasažených řádků není rozhodující. Nezřídka se na celém řádku změnil jen jediný znak.

Třída I nedává mnoho možností pro uplatnění jakéhokoli algoritmu rozpoznávajícího změny a zjevně nejefektivnějším řešením je uložit soubor jako celek. Pro potřeby dalších úvah v této kapitole ji tedy vypustíme. Soustředíme se na hledání algoritmu úspěšně zvádajícího třídy II až IV v duchu vlastností 1 – 5 z prvního seznamu.

2.3 Algoritmus Diff

Po zvážení podmínek a vlastností popsaných v předchozí sekci se jako vhodný kandidát pro základ algoritmu rozpoznávajícího změny mezi textovými soubory jevil algoritmus *Diff: algoritmus rozdílového porovnávání souborů* [2].

Vstupem algoritmu je dvojice textových souborů. Diff dokáže vytvářet seznam změn řádků mezi soubory. Rozeznává tři druhy změn:

- **a**(p,text) – přidání řádků s *textem* na danou pozici p
- **d**(p,d) – smazání řádků z dané pozice p a v dané délce d
- **c**(p,d,text) – náhrada obsahu řádků na nadné pozici p a v dané délce d *textem*. Tato varianta je vlastně zkratkou pro současné použití obou předcházejících metod.

Výstupem algoritmu diff je například zpráva ve tvaru:

```
7 a 2
první přidaný řádek
druhý přidaný řádek
```

```
12 d 6
18 c 1
změněný obsah osmnáctého řádku
```

Analýza splňování podmínek

Algoritmus Diff z definice své základní funkce splňuje podmínku číslo jedna seznamu z minulé sekce.

Výstup algoritmu Diff lze linearizovat a uložit ve standardním textovém formátu. Tento formát lze načíst a snadno a rychle zpracovat pro získání seznamu změn nutného pro rekonstrukci původního souboru. Ze znalosti staré verze a diffu lze v čase úměrném součtu jejich délek zrekonstruovat verzi novou. Podmínky dvě a tři jsou tedy rovněž splněny.

Jak bude rozebráno později, čas běhu algoritmu Diff nutný pro vytvoření seznamu změn mezi dvěma verzemi textového souboru je v průměrném případě asymptoticky lineární vůči součtu počtu řádků obou těchto souborů a v nejhorším případě asymptoticky lineární vůči jejich součinu. Tento nejhorší případ nastává pouze u degenerovaných případů souborů s dlouhými sekvencemi totožných řádků. Není pravděpodobné, že by se při rozumném použití vyskytovaly takové soubory. Pro tuto chvíli považujme podmínku číslo čtyři taktéž za splněnou.

Poslední podmínka se týká vztahu mezi množstvím a rozsahem změn a velikostí jejich seznamu. Tu je třeba detailně prostudovat ve vztahu k jednotlivým typům změn rozeznávaných diffem a třídám změn souborů definovaným v minulé sekci.

Podmínka č. 5 vzhledem ke třídám změn

Nejsnadnější situace nastává v případě mazání celých řádků. Zde je délka uložené informace velmi malá (do deseti znaků i u značně velkých souborů) a dokonce nezávisí ani na velikosti smazaného textu (pomineme-li zcela marginální dopad v podobě počtu cifer informace o množství smazaných řádků). Z hlediska podmínky číslo pět žádný problém.

Jen o málo složitějším případem je přidávání řádků. I zde (až na malou servisní informaci) počet a délka řádků výpisu odpovídá počtu a délce přidaných řádků (jsou to ony).

Podívejme se nyní na případ, kdy došlo ke změnám současných řádků, respektive k záměně jejich obsahu za jiný obsah. Pro jednoduchost budeme hovořit o jednom řádku. Situace se značně mění v závislosti na třídě

převažujících změn. Třída I je pro analýzu irelevantní, věnujme se tedy třídám II, III a IV.

V případě třídy II jde o rozsáhlejší změny, jež zpravidla zasáhly více než 50% znaků na řádku. Vypsání celého řádku, jak jej provádí diff, pomineme-li servisní informace, znamená řádově maximálně dvojnásobné zvětšení informace ve výpisu oproti skutečnému množství informace reprezentující velikost změny (měřeno v počtu znaků). To lze považovat za uspokojivé.

Třída III: stylistické úpravy v textu, jak byla definována v sekci 1.2, znamená úpravy převážně na úrovni slov zasahujících méně než 50% znaků na řádku. Odtud lze odvodit, že počet znaků ve výpisu bude nejméně dvakrát větší než počet skutečně v textu změněných znaků. Předpokládáme-li nahrazení jednoho osmiznakového slova na řádku o osmdesáti znacích, dostáváme dokonce poměr 10:1, což již jen ztěží můžeme považovat za uspokojivý výsledek.

V případě třídy IV se jedná o občasné úpravy na úrovni jednotlivých znaků, rozprostřených po textu. Uvážíme-li nejjemnější změnu, tedy úpravu jediného znaku, pak poměr navýšené informace se může rozrůst na běžném počítačovém řádku až na zdrcujících 80:1. Za zmínku stojí i fakt, že v mnoha formátech textových souborů a mnoha editorech se používá umělého zalamování řádek a délka jednoho řádku může být mnohonásobně delší než uvedených 80 znaků. V takových případech může dosahovat poměr navýšené informace hodnot ještě mnohonásobně vyšších.

Závěrem těchto úvah je konstatování, že v případě tříd III a IV jsou výsledky algoritmu diff v jeho standardní podobě neuspokojivé. Dosud jsme se v analýze věnovali algoritmu Diff zvenčí. Abychom dosáhli lepších výsledků, je třeba prozkoumat fungování algoritmu Diff zevnitř.

Algoritmus Diff zevnitř

Základní princip algoritmu diff se podstatně liší od jeho přirozeného vnímání jeho uživateli. Algoritmus diff ve skutečnosti nehledá rozdíly mezi oběma vstupními soubory, jak hlásá jeho název: Diff – algoritmus rozdílového porovnávání souborů, ale naopak se snaží nalézt maximum shodného mezi nimi. Tento nikoli triviální fakt má za následek, že algoritmus diff nutně potřebuje nějakou přirozenou strukturu uvnitř obou vstupních souborů, jejíž prvky by mohl mezi oběma soubory párovat. Takovou strukturou je v klasické implementaci posloupnost řádků.

Volba řádku jako základní entity diffu má několik důsledků: Prvním

z nich je relativní rychlost. Počet řádků v běžném textovém souboru nebývá velký, algoritmus tedy pracuje zpravidla se stovkami, maximálně s tisíci položek na obou stranách.

Dalším důsledkem je fakt, že řádky v sobě zahrnují relativně velkou informaci, řádově přibližně 80 bytů, ve skutečnosti samozřejmě méně (ne všechny hodnoty a kombinace bytů mají smysl čitelného textu). To v porovnání s jejich počtem významně snižuje výskyt kolizí, tedy shodných prvků na více místech posloupnosti, jež podstatně snižují rychlost běhu algoritmu [2].

Řádky také vykazují velkou stabilitu při přechodu od verze k verzi, alespoň u člověkem psaného textu, což je důležité pro nalezení podobnosti mezi oběma soubory a nepřímo tedy i pro velikost diffu.

Negativním důsledkem volby řádku jako přirozené dělící jednotky je omezení přesnosti diffu právě na hodnotu jednoho řádku, což může být dělení velmi hrubé, jak jsme již ukázali.

2.4 Diff trochu jinak

Logická otázka, která se nabízí po prozkoumání silných a slabých stránek algoritmu Diff založeného na řádcích, zní, zda existují nějaké alternativy pro řádek jako dělící entitu, jež pokud možno zachovávají dobré vlastnosti řádku jak byly popsány výše a přitom představují jemnější jednotku členění textu.

Alternativní dělící jednotky

Prozkoumejme nyní, jaká entita uvnitř přirozeného textu by mohla hrát roli dělící jednotky pro algoritmus diff. Vzhledem k tomu, že hledáme jednotky menší než řádek, patrně se nepodaří zachovat jejich nízký počet, srovnatelný s počtem řádků. Soustřeďme se tedy na stabilitu, přirozenost, očekávané množství kolizí a univerzálnost jednotky vůči různým formátům textových souborů.

Z hlediska přirozenosti a univerzálnosti jsem do úvah zahrnul dvě potenciální jednotky: slova a znaky. Je tomu tak proto, že jak slovo, tak samozřejmě i znak, představují entity, jež je možno nalézt v drtivé většině lidmi psaných textů. Tyto jednotky nejsou závislé na volbě formátu, vyskytují se stejně v přirozeném textu, zdrojovém textu většiny programovacích jazyků i nejrůznějších na textu založených formátech přenosu dat. Existují

samozřejmě i výjimky, ale není jich mnoho a pro nápravu zpravidla stačí zadefinovat jiný oddělovač slov (např. čárku u CSV apod.). Slova a znaky jsou tedy jednotkami univerzálními i přirozenými.

Stabilitou pro účely této analýzy rozumím schopnost jednotky nepropagovat změnu na další sousední jednotky. Tato vlastnost je zcela nezbytná pro použitelnost jednotky. V tomto smyslu lze považovat obě zmíněné alternativní dělící jednotky za stabilní.

Zbývá prozkoumat jejich náchylnost ke kolizím. K tomu poslouží malá statistika shrnutá v tabulce 2.1.

Tabulka 2.1: STATISTIKA VYBRANÝCH TEXTŮ

Text	**řádků**	**slov**	**znaků**	**%10ř**	**%10s**	**%10z**
K. Čapek: Válka s mloky	6 789	67 066	439 053	5,32	14,67	54,96
diffcz.html	514	4 665	30 724	24,32	28,78	58,87
difftree.cpp	594	1 731	16 218	33,16	39,77	73,28

Jde o trojici textů: prvním z nich je Čapkova Válka s mloky – textový soubor bez jakýchkoli formátovacích značek. Čapek reprezentuje autora s maximálním smyslem pro bohatost textu, neupadajícího v sebeopakování a používání ustálených klišé. Představuje jakési minimum šablonovitosti, jaké si lze ve smysluplném textu představit. Třetím textem je část kódu knihovny difflib psaná v jazyce C++. Lze ji považovat za jakýsi protipól k Čapkovi: značnou část textu tvoří formální programátorské obraty a několik standardních konstrukcí. Mezi těmito dvěma extrémy se pohybuje soubor diffcz.html, což je můj překlad článku Hunta a McIlroye o algoritmu Diff [2], reprezentující přirozený text doplněný o formátovací značky jazyka html.

Zkoumanými parametry jsou kromě informativních délek v řádcích, slovech a znacích zejména položky % 10 slov a % 10 znaků. Udávají procento, které v textu zaujímá 10 nejčastějších řádků, slov a znaků.

Z tabulky lze vyčíst několik zajímavých informací. Na jeden řádek připadá asi 10 slov a 80 znaků. Procento shodných řádků je u formálních textů velmi výrazně vyšší než u přirozeného textu, v našem případě více než pětinásobně. Pokud jde o slova, jejich podobnost je obecně mírně vyšší než podobnost řádků ale rozdíl mezi formálními a přirozenými texty je již menší – jde zhruba o 2,5 násobek. Podobnost znaků je nejvyšší, 50% – 75% a liší se zhruba o polovinu mezi formálními a přirozenými texty.

Obsah tabulky nás svádí k formulování hypotéz o účinnosti a rychlosti diffu nad slovy a znaky. Lze předpokládat jisté zpomalení způsobené

větším počtem jednotek v témže textu. Zpomalení způsobené prodloužením posloupností bude minimálně desetinásobné u slov a osmdesátinásobné u znaků, nepočítáme-li vliv kolizí. Ten by mohl mít značný dopad zvláště u znaků. Abychom dodali těmto úvahám pevnější základy, bude lépe provést podrobná měření.

Testování diffu postaveného na slovech a znacích

Pro potřeby detailního prozkoumání jsem implementoval tři varianty algoritmu diff postavené na článku Hunt a McIlroye [2]. První je založena na řádcích, označme ji LineDiff, druhá na slovech – WordDiff a poslední na znacích – CharDiff.

Testovací data se skládají ze tří výše popsaných souborů. Pro každý z nich jsem zpracoval varianty změn reprezentující třídy II až IV:

Ve třídě II bylo přibližně 10% řádků náhodně vybraných řádků změněno tak, že polovina byla vypuštěna a za druhou polovinu přibyl jeden náhodný řádek z téhož souboru. Třída III vznikla tak, že 1% slov bylo buď smazáno nebo za něj bylo vloženo náhodné slovo z téhož souboru. Třída IV byla vytvořena obdobně pouze změny se týkaly 0,1% znaků. Pro každý soubor jsem připravil ještě čtvrtou variantu kombinující všechny předchozí změny.

Cílem testování bylo zjistit čas běhu a velikost normovaného diffovacího výstupu pro každý algoritmus, každý soubor a každou třídu změn. Výsledky jsou shrnuty v tabulkách 2.2 a 2.3.

Podívejme se nyní podrobně na výsledky testu. Předně lze říci, že testy nemají jasného vítěze co do obou hodnotících kritérií – tedy účinnosti a rychlosti na všech variantách zadání.

Účinnost diffů měřená v počtu znaků rozdílové zprávy v normovaném formátu se velmi liší mezi jednotlivými zadáními. Každý algoritmus vítězí ve své skupině. V celkové kombinaci změn, jež má nejlepší vypovídací hodnotu, dosahuje nejlepších výsledků CharDiff. WordDiff je jen nepatrně horší, zatímco LineDiff zaostává troj až čtyřnásobně. Je tedy zřejmé, že větší granularita může velmi významně zlepšit účinnost diffu.

Pokud jde o rychlost, jsou výsledky CharDiffu naprosto tristní. I na nejmenším souboru trvá výpočet 12s a téměř minuta na souboru střední délky odsouvá CharDiff mimo praktickou použitelnost. Na Čapkovi algoritmus CharDiff nedoběhl vůbec: téměř po hodině a půl vyčerpal veškerou dostupnou systémovou paměť a byl operačním systémem ukončen.

Výsledky WordDiffu jsou sice lepší, ale pro využití v aplikaci vy-

Tab. 2.2: TESTOVÁNÍ VARIANT ALGORITMU DIFF – ČAS

difftree.cpp

Algoritmus	**Třída II**	**Třída III**	**Třída IV**	**kombinace**
LineDiff	0,01	0,02	0,02	0,02
WordDiff	3	3	3	3
CharDiff	12	12	9	13

diffcz.html

Algoritmus	**Třída II**	**Třída III**	**Třída IV**	**kombinace**
LineDiff	0,01	0,02	0,02	0,02
WordDiff	3	2	3	2
CharDiff	40	32	39	61

valka s mloky.txt

Algoritmus	**Třída II**	**Třída III**	**Třída IV**	**kombinace**
LineDiff	0,23	0,28	0,25	0,31
WordDiff	81	71	61	78
CharDiff	—	—	—	—

žadující více po sobě jdoucích běhů algoritmu jsou výpočtové časy nad 1s taktéž nevyhovující. Zajímavé je povšimnout si, že délka souboru není jediným činitelem ovlivňujícím rychlost diffovacích algoritmů. Zatímco počet řádků války s mloky je 6 789 a počet slov souboru diffcz.html je pouze 4 665, čas běhu LineDiffu na prvním z nich činil pouze 0,23s, zatímco algoritmu WordDiff trvalo zpracování o třetinu kratšího souboru celé 2s. V souladu s článkem Hunta a McIlroye [2] přikládám tento vliv většímu počtu kolizí během běhu algoritmu.

Závěr

Testování odhalilo značné naděje pro zmenšení velikosti diffu nahrazením klasické dělící jednotky algoritmu Diff, jíž je řádek, jednotkami menšími jako slovo a znak. I značně velká úspora (až 75%), které může být takto dosaženo na souborech kombinujících změny různých tříd však není vyvážena podstatným prodloužením času výpočtu (až 200x). Hlavními rizikovými faktory tohoto prodloužení je nepochybně délka souboru a velký

Tab. 2.3: TESTOVÁNÍ VARIANT ALGORITMU DIFF – Velikost

difftree.cpp

Algoritmus	**Třída II**	**Třída III**	**Třída IV**	**kombinace**
LineDiff	836	4 791	1 962	6 418
WordDiff	1 296	1 019	736	2 722
CharDiff	1 078	1 134	340	2 504

diffcz.html

Algoritmus	**Třída II**	**Třída III**	**Třída IV**	**kombinace**
LineDiff	1 479	12 437	7 220	16 131
WordDiff	1 655	1 435	1 296	4 063
CharDiff	1 936	1 797	558	4 291

valka s mloky.txt

Algoritmus	**Třída II**	**Třída III**	**Třída IV**	**kombinace**
LineDiff	25 911	124 423	72 397	190 793
WordDiff	27 417	18 058	14 660	57 738
CharDiff	—	—	—	—

počet shodných entit zabraňujících algoritmu Diff ve správné synchronizaci.

Ideálním řešením by bylo nějak zkombinovat rychlost řádkového diffu s přesností Diffu založeného na slovech a znacích. Pokus o takové řešení bude tématem další sekce.

2.5 Víceúrovňový diff

Cílem této sekce je zmapovat možnosti kombinace více dělících jednotek v rámci jednoho algoritmu Diff. Výsledkem by měl být algoritmus spojující rychlost klasického Diffu s přesností diffu založeného na slovech a znacích.

Pro jednodušší pochopení mechanizmu práce algoritmů založených na diffu s námi použitými třemi dělícími jednotkami si rozdělíme změny provedené v souboru na několik kategorií. Budeme používat dvě kritéria: typ změny a velikost změny.

První kritérium dělí změny na tři typy: změny vedoucí k přidání (add), smazání (delete) a změně (change). Jde o tři základní typy změn

algoritmu diff.

Druhým kritériem bude velikost změny dle použité jednotky s rozdělením na řádkové, slovní a znakové.

Kombinací těchto dvou kritérií dostáváme devět kategorií změn. Prozkoumejme nyní jak trojice algoritmů, popsaných v minulé sekci, s jednotlivými kategoriemi změn nakládá a pokusme se tak odhalit silné a slabé stránky každého algoritmu.

- **LineDiff**
  Algoritmus LineDiff si dokáže rychle a elegantně poradit se změnami na úrovni řádků všech tří typů. Režie na jejich zpracování je velmi malá. Všechny drobnější změny jsou zahrnuty do kategorie řádkového change a spolu se zbytkem svého řádku celé vypsány do výstupní zprávy. To je sice rychlé, ale může to být velmi neefektivní u malých změn jak bylo prokázáno měřením v předchozí sekci.

- **WordDiff**
  Algoritmus WordDiff přistupuje přímo ke slovům. Řádkové změny dokáže bez problémů zpracovat jako součást vícenásobného slovního add, delete a change. Vyžaduje to sice vyšší režii pokud jde o čas, ale výsledná velikost diffu je téměř (rozdíly procent) stejná jako u LineDiffu. Slovní změny řeší nejlépe a nejefektivněji. Se změnami na úrovni znaků se vypořádává tak, že uloží celé slovo jako záznam slovního change. Pro krátká slova vyskytující se převážně v přirozeném jazyce to není žádný problém. Komplikace v podobě nižší efektivity přicházejí pouze u formálních textů, v nichž se často vyskytují velmi dlouhá slova:
  MyMainForm.TopBlackPanel.MyMostImportantTitleLabel.Color:= LoadColorFormMyFavouriteColorBuffer(12); změnit na 13).

- **CharDiff**
  Algoritmus CharDiff se dokáže účinně nebo téměř účinně vypořádat se všemi druhy změn. Pro svou pomalost se však nehodí na nasazení nikde jinde než u velmi krátkých textů.

Konstrukce algoritmu

Jako základ výsledného algoritmu vezměme LineDiff. Nic jiného nám vzhledem k jeho rychlosti a rychlosti ostatních alternativ ani nezbývá. Ponechme z něj to nejlepší, tedy způsob jakým se LineDiff vyrovnává s velkým souborem a způsob nalézání a ukládání změn typu řádkový add a řád-

kový delete. Tím získáváme velmi efektivní jádro algoritmu. Zatím dokáže řešit dvě z devíti kategorií změn, jak byly definovány v úvodu této kapitoly.

Všechny ostatní změny nám tímto sítem propadly do řádkového change. Co tedy vlastně máme? Máme seznam změn, na němž figurují změny typu add, delete a change. O prvních dvou víme, že byly rozpoznány správně, a nebudeme se jimi proto už nadále zabývat.

Zbývá posloupnost změn typu řádkový change. Některé z těchto změn skutečně představují změnu celých řádků, ale značná část z nich jsou ve skutečnosti změny patřící do některé z menších kategorií, které propadly příliš hrubým sítem řádkového diffu. Důležité je, že ke každé změně typu řádkového change máme k dispozici dvojici řetězců. Jsou jimi řetězec s jedním či několika řádky před touto změnou a po ní.

Na dva řetězce řádkové změny použijeme algoritmus WordDiff. Výsledkem je opět zpráva obsahující tentokrát slovní změny typu add, delete a change. Mezi těmito změnami je i několik nesprávně zařazených změn typu řádkových change. Těm se budeme věnovat později. Pro tuto chvíli předpokládáme, že máme co do činění pouze se slovními a znakovými změnami. Slovní změny typu add a delete jsou ve zprávě odděleny zvlášť zatímco slovní change a znakové změny jsou společně zařazeny mezi slovní change.

Na jednotlivé změny kategorie slovní change spustíme stejným způsobem algoritmus CharDiff. Výsledkem je posloupnost znakových změn pro každou slovní změnu typu change.

Celý dosavadní výstup takto konstruovaného algoritmu lze uložit do stromové struktury o třech patrech. Změny typu add, delete a znakový change představují listy stromu. Ostatní změny leží v nelistových uzlech.

Nyní je třeba strom linearizovat a převést na textový výstup. Linearizace stromu není za normálních okolností těžký úkol. V linearizaci tohoto stromu a jeho zakódování se však skrývá ještě jeden úkol: je třeba odhalit a odstranit ty změny typu change, jež byly nesprávně rozloženy na jemnější změny, než jakými ve skutečnosti jsou. Jejich odhalení a oprava nepředstavuje obtížný ani složitý (ve smyslu výpočtové složitosti) úkol.

V případě, že linearizovaná podoba daného uzlu typu slovní change včetně všech jeho poduzlů je větší než prostý výpis textu který tento výpis poduzlů nahrazuje, pak jde o nesprávně rozpoznanou změnu a do výpisu bude uložen text, v opačném případě budou uloženy poduzly. Lépe to osvětlí následující zápis:

Mějme řádkovou změnu $C(A \rightarrow B)$ od řádku A k řádku B, která

byla rozložena v posloupnost slovních změn $c_1, c_2, c_3, ...$ Pokud platí:

$$L(C, B) < L(C, \sum_{\forall i} c_i)$$

pak se uloží $L(C, B)$, jinak se uloží $L(C, c_1, c_2, ...)$, kde $L(X)$ je linearizovaná podoba uzlu X.

Je zřejmé, že tento postup úspěšně napravuje všechny předchozí omyly, protože byl-li řádkový change zapsán jako posloupnost slovních změn, pak tyto změny nemohly popisovat skutečnost úsporněji než jak by byla popsána prostým vypsáním změn například v nějakém velkém slovním change. Pokud by mohli zachycovat změnu úsporněji, pak to patrně ve skutečnosti byla změna slovní a nikoli řádková. Totéž platí i pro rozhraní mezi slovními a znakovými změnami.

Analýza

Vlastnosti výše popsaného algoritmu vedou k domněnce, že algoritmus by mohl splňovat vytyčené cíle. Za předpokladu alespoň trochu rozumného zadání vytvoří LineDiff posloupnost změn. Přibližně třetina z nich budou řádkové change. Každý z řádkových change bude podstatně kratší, než jak dlouhý je celý soubor. Zpravidla půjde o jeden či několik málo řádků. Na ně bude aplikován WordDiff.

Každý algoritmus WordDiff spuštěný nad jednou řádkovou změnou vytvoří seznam slovních změn. Opět zhruba třetina z nich budou slovní change. Přibližnou délku jedné slovní změny lze očekávat v řádu jednotek až desítek znaků. Takto krátký úsek textu lze bez obav z přílišného zdržení postoupit algoritmu CharDiff.

Algoritmus CharDiff rozmělní slovní změny na nejnižší úroveň rozpoznatelných podobností a rozdílů textu.

Ve fázi linearizace stromu změn se v každé větvi vždy ukládá optimální hloubka, což zaručuje nejmenší velikost výsledného výpisu.

Testy

Provedl jsem dvanáct testů víceúrovňového diffu na používané trojici souborů s testovacími daty, každý ve čtyřech verzích podle charakteru změn. Výsledky shrnuje tabulka 2.4.

Velikost výsledných verzí není nikdy větší než u LineDiffu, což bylo možné díky mechanismu linearizace stromu změn očekávat.

Tabulka 2.4: TESTOVÁNÍ VÍCEÚROVŇOVÉHO DIFFU

	Třída II		**Třída III**		**Třída IV**		**kombinace**	
soubor	**vel**	**čas**	**vel**	**čas**	**vel**	**čas**	**vel**	**čas**
difftree.cpp	836	0,02	1 532	0,03	717	0,02	2 666	0,03
diffcz.html	1 479	0,01	1 880	0,04	1 126	0,03	4 025	0,06
Válka s mloky	25 911	0,23	23 338	0,46	16 652	0,42	59 183	0,66

U třídy II nejlépe vyhovující algoritmu WordDiff dosáhl Víceúrovňový diff v průměru o pětinu až třetinu horšího výsledku než WordDiff – ovšem výměnou za 100 až 200 násobné zrychlení. Naopak nejrychlejší LineDiff byl přibližně dvakrát rychlejší ale velikost jeho výstupu je 3 až 6 krát větší.

Změny třídy III nezvýšily časovou složitost víceúrovňového diffu, opět nebyl více než dvakrát pomalejší než LineDiff, zato byl opět 3,5 - 6,5 krát účinnější. V porovnání s CharDiffem byl Víceúrovňový diff přibližně dvakrát větší a 1200 krát rychlejší.

V nejdůležitější kategorii kombinovaných změn se Víceúrovňový diff držel na druhém a v jednom případě dokonce prvním místě. Byl v průměru 3 - 4 krát lepší než LineDiff a zhruba 2 krát pomalejší. V porovnání s WordDiffem a CharDiffem nebyl nikdy hůř než o několik procent horší a stále o dva řády rychlejší.

Tyto výsledky dělají z Víceúrovňového diffu velmi úspěšný algoritmus.

2.6 Shrnutí

Při práci s textovými dokumenty na počítači vzniká potřeba nějakým způsobem uchovávat více časově sousledných verzí postupně vznikajícího textu. Pro tuto správu je vhodné mít možnost ukládat pouze rozdíly mezi verzemi a nikoli celé verze. Cílem této kapitoly bylo nalézt rychlý a účinný algoritmus, který to umožňuje.

Zevrubně byl prozkoumán charakter změn, jimiž se od sebe liší různé verze textových dokumentů a byly vytyčeny podmínky, jež by měl algoritmus splňovat pro úspěšné použití ve vytvářené aplikaci.

Za základ algoritmu byl zvolen Diff – algoritmus rozdílového porovnávání souborů [2]. Diff byl konfrontován s podmínkami a byl shledán vyhovujícím ve všech s výjimkou jedné. Problém se nalézal v nedostatečné jemnosti diffu vedoucí k tomu, že nepatrné změny způsobovaly velký nárůst výstupu algoritmu Diff.

Analýza algoritmu Diff ukázala, že zásadním omezením je velikost základní jednotky algoritmu Diff, jíž je řádek. Hledání alternativní jednotky na základě kriterií univerzálnosti, jemnosti, přirozenosti a stability přineslo dvě nové jednotky: slovo a znak. Provedené pokusy s modifikacemi klasického algoritmu Diff na WordDiff a CharDiff, jejichž základními jednotkami jsou slovo a znak, přinesly rozporuplné výsledky: bylo dosaženo slušného zmenšení diffu (i na šestinu původní velikosti), zvláště u souborů s převládajícími malými změnami, ale stalo se tak na úkor neúměrného prodloužení běhu algoritmu diff (o tři řády i více).

Ve snaze zkombinovat přednosti obou řešení byl zkonstruován víceúrovňový algoritmus postavený na klasickém Diffu a doplňený o WordDiff i CharDiff používané na vybrané menší úseky textu.

Testování ukázalo, že Víceúrovňový diff je pouze zhruba dvakrát pomalejší než klasický LineDiff, za to ale dosahuje na souborech kombinujících různé typy změn účinnosti srovnatelné s CharDiffem a WordDiffem.

Takový výsledek lze jednoznačně považovat za splnění jednoho ze tří cílů stanovených v úvodu této práce.

Kapitola 3

Správa Archivu

3.1 Úvod

Podstatnou částí aplikace pro správu verzí textových souborů je její archiv. Jde o soubor, do něhož aplikace ukládá jednotlivé spravované texty a jejich verze. V lepším případě je možné je posléze z archivu opět vybírat a rekonstruovat jejich pokud možno původní podobu. Tohoto cíle se pokusím dosáhnout i v aplikaci DiffArchive, vytvářené v rámci této práce.

Cílem třetí kapitoly je pohovořit o principech a implementaci správy archivu v aplikaci DiffArchiv. Nejdříve prozkoumám požadavky na práci s archivem, předpokládané operace a očekávanou relativní četnost jednotlivých operací. V závislosti na nich byl navržen formát souboru s archivem, jeho vnitřní uspořádání, mechanismy přístupu k datům a použitá metoda komprese archivu.

Popsány budou také knihovna ZipFile obstarávající práci s komprimovaným souborem a knihovna Archive sloužící pro vnitřní správu archivu, dvě důležité součásti programu DiffArchive, jehož ostatní součásti budou rozebrány ve čtvrté kapitole této práce.

3.2 Požadavky na archiv

Předpokládejme archiv textových souborů a jejich verzí tvořících zdrojové a doplňkové texty nějakého projektu. Cílem této sekce je stanovení požadavků kladených na takový archiv.

Operace s archivem

Zvažme nyní potřeby práce se soubory ve výše popsaném archivu, předpokládané dotazy vůči archivu, jejich relativní četnost a požadavky na rychlost. Je pravděpodobné, že ačkoli rychlost je potřebná vždy, u různých dotazů je tato potřeba různě akutní. Výsledkem, nikoli nutně jediným, těchto úvah je následující seznam:

1. **Vložení nového souboru**
   Rozumí se tím vložení první verze souboru, který se v archivu nevyskytuje. Lze předpokládat, že tato operace se nebude provádět příliš často.

2. **Vložení nové verze souboru v archivu**
   Zde se naopak předpokládá, že soubor a řada jeho starších verzí se již v archivu vyskytuje a je přidávána nová verze. Tato operace bude patrně činěna často a je tedy třeba provádět ji co nejrychleji.

3. **Načtení poslední verze souboru**
   Opět jde o frekventovanou operaci, u níž lze očekávat nejvyšší nároky na rychlost provedení: zatímco u vkládání uživatel zadá příkaz a může jít dělat něco jiného, zde na svá data aktivně čeká a je tedy třeba optimalizovat archiv zejména právě pro tuto operaci.

4. **Načtení obecné verze souboru**
   Zatímco u poslední verze lze očekávat velký počet dotazů například kvůli další editaci, obecná verze ležící uprostřed souboru se nevyhledává příliš často. Je tedy možné snížit požadavky na dobu zpracování, ačkoli uživatel archivu by měl mít možnost tuto dobu nějak shora omezit výměnou za o něco větší velikost archivu.

5. **Vypsání seznamu souborů v archivu**
   Dotaz lze očekávat často, například jako kontrolu provedené změny. Rychlost je vyžadována.

6. **Vypsání seznamu verzí souboru uloženého v archivu**
   Platí totéž jako u bodu 6.

7. **Vymazání souboru z archivu**
   Tato operace zpravidla nespěchá. Není třeba pro ni archiv optimalizovat.

8. **Vymazání verze souboru z archivu**
   Platí totéž jako u bodu 7.

Vůči vnějšímu světu by se k souborům uloženým v archivu mělo přistupovat na základě jejich vnitřního jména, které se určí při jejich vkládání do archivu. Přístup k verzím skýtá více možností. Verzi lze při vkládání také pojmenovat, respektive jí přiřadit verzovací označení. Pro specifikaci verze je možné použít i její pořadové číslo ve výpisu, a dokonce údaj o času. V takovém případě se za vybranou verzi považuje verze aktuální k danému datu a času.

Komprese archivu

Cílem aplikace DiffArchiv, jak byl stanoven v úvodu této práce, je co nejvíce zmenšit velikost archivu. Hlavním nástrojem pro dosahování tohoto cíle je komprese dat uložených v souboru. Existují dvě hlavní cesty komprese dat a aplikace DiffArchive využívá obě z nich.

První z nich operuje nad obsahy verzí jednoho souboru. Když aplikace obdrží příkaz na vložení nové verze souboru, jenž se již v archivu vyskytuje, nedojde ve skutečnosti k přidání celého souboru, ale pouze k vložení rozdílu (diffu) oproti poslední vložené verzi. K tomuto účelu aplikace DiffArchive používá algoritmus Diff v modifikaci popsané ve druhé kapitole této práce. Ukládání diffu namísto celého souboru znamená velmi značnou úsporu místa. Úspora je tím větší, čím větší jsou vkládané soubory a čím menší jsou prováděné změny.

Druhá metoda komprese operuje nad archivem jako celkem a bude podrobněji rozebrána v sekci věnované formátu souboru s archivem.

3.3 Formát souboru s archivem

Již od počátku práce na programu DiffArchive bylo jasné, že volba vhodného formátu archivačního souboru bude klíčová pro efektivitu aplikace. Bylo zřejmé, že by se mělo jednat o formát komprimovaný, neboť textová data se komprimují velmi dobře a formalizované výpisy diffovacího algoritmu jsou pro další kompresi téměř ideální.

Otázkou, jíž bylo třeba rozřešit, bylo, jakou část implementovat vlastními silami a pro jakou část zvolit veřejně přístupné knihovny (kompresní, databázové…). Výhodou vlastního řešení by byla větší možnost přizpůsobit si prostředky vlastnímu účelu. Nevýhodou pak zejména vyšší časová

náročnost implementace a nižší dosažené výsledky v obecných částech úkolů (např. textová komprese…).

Rozhodl jsem se předpokládanou funkčnost rozdělit na dvě skupiny. Pro úkoly pro něž existují efektivní nástroje převzít veřejně přístupnou knihovnu a ostatní implementovat vlastními silami. Zejména jsem se rozhodl neimplementovat vlastní algoritmus textové komprese. Důvod je zřejmý: existuje mnoho veřejně přístupných nástrojů, jejichž autoři věnovali řádově více času jejich vytváření (a zejména pečlivé optimalizaci) než kolik času si vyžádala celá tato práce.

Nyní bylo třeba zvolit kompresní algoritmus z množiny volně dostupných algoritmů. Pro tento výběr jsem si stanovil několik kritérií, byly jimi:

- **Rychlost**
  Uvádím ji na prvním místě, protože ji považuji za nejdůležitější vlastnost pro toto použití. Důvodem je zejména fakt, že jde pouze o sekundární kompresi, která může zmenšit velikost archivu, ale neměla by příliš brzdit aplikaci.

- **Přenositelnost**
  Knihovna by neměla být platformě závislá, nebo by alespoň měly existovat komprimační i dekomprimační nástroje pod všemi hlavními platformami.

- **Kompresní poměr**
  Jedná se především o schopnost dobře komprimovat texty.

- **Další bonusy**
  Zde mám na mysli zejména snadnou použitelnost, spolehlivost a zavedenost formátu, škálovatelnost: míra komprese × čas, přítomnost podpory souborů…

Po zvážení možností jsem se rozhodl pro algoritmus ZIP ve volně dostupné implementaci Info-ZIP [3]. K tomuto výběru mne vedla především již zmíněná rychlost. Zip patří k „lehčím“ kompresním algoritmům. Jeho výsledky na textu jsou velmi dobré. Jde o jeden z nejčastěji používaných formátů. Komprimační i dekomprimační utility existují úplně na všech platformách, neomezují tedy přenositelnost aplikace. Formát v provedení Info-ZIPu navíc implementuje i souborovou databázi, což rovnou řeší i problém s implementací vnitřního formátu souboru.

Info-ZIP

Knihovna Info-ZIP vyvíjená stejnojmenným sdružením je volně šířitelná a použitelná knihovna utilit pro kompresi a dekompresi souborů ve formátu zip. Obsahuje dvojici programů: zip.exe a unzip.exe, které pokrývají veškerou standardní funkčnost komprimačních knihoven. Jejich ekvivalenty existují pro 29 známých i zcela obskurních platforem. Ve vývoji je též dynamicky linkovaná knihovna zlib.dll. Ta však nepodporuje souborovou databázi a nedisponuje takto širokou přenositelností.

Z těchto důvodů jsem se rozhodl pro obě utility příkazového řádku. Komunikaci s nimi jsem umístil do jedné třídy ZipFile zapouzdřující práci se souborem formátu zip a volání utilit do jedné vnitřní funkce implementující práce s rourami (anglicky pipe – oficiální překlad neexistuje), abych maximálně usnadnil budoucí přenositelnost aplikace nebo případnou změnu knihovny. Podrobněji se budu rozčlenění aplikace věnovat ve čtvrté kapitole.

3.4 Vnitřní uspořádání archivu

Jak bylo rozebráno v předchozí sekci, archiv má formát zip souboru včetně podpory vnitřní souborové databáze. To předznamenává i vnitřní uspořádání archivu. Všechna data jsou uložena ve formě vnitřních textových souborů. Základním modelem uspořádání dat je jakási odlehčená verze indexového souboru[1].

Index

Index použitý v archivu má dvě úrovně. První (nejvyšší) úroveň indexu tvoří soubor s názvem *Index*. Tento soubor se vyskytuje v každém archivu právě jednou. Slouží k propojení jmen uložených souborů a vnitřního identifikačních čísel souborů. Jedná se o jména, pod nimiž byly soubory do archivu uloženy a pod nimiž jsou vůči archivu zadávány dotazy na ně. Identifikační čísla souborů (FID) se vybírají z množiny přirozených čísel jako číslo o jedna vyšší, než je nejvyšší doposud použité číslo. Mapování mezi názvy souborů a identifikačními čísly je vzájemně jednoznačné. Následuje malý příklad souboru Index:

```
1,Karel Čapek: Válka s mloky.txt
2,diffcz.html
7,difftree.cpp
```

Druhou úroveň indexu tvoří soubory se seznamem verzí. Pro každý soubor vložený do archivu je vytvořen jeden vnitřní soubor s názvem *I[FID]*. Uvnitř tohoto souboru jsou uloženy potřebné informace o jednotlivých verzích uložených v archivu. Soubor se seznamem verzí má formát CSV[4] Každý řádek souboru, jehož formát následuje, odpovídá jedné verzi souboru:

```
Typ,VID,Velikost,ČasVytvoření,ČasVložení,Verze,Komentář
```

Jednotlivá pole budou rozebrána podrobněji:

- **Typ**
  Určuje způsob, jakým je verze v archivu uložena.

  0 jde o první vloženou verzi a verze je uložena v plném znění

  1 verze je uložena jako diff vůči předchozí verzi

  2 jde o poslední verzi uloženou jako plná verze

  3 jde o plnou verzi uvnitř řady verzí, sloužící jako záchytný bod urychlující práci s archivem

- **VID**
  Jde o jednoznačný identifikátor verze v seznamu verzí. Jde o obdobu FID pouze omezenou na jeden soubor. Verze v rámci archivu je identifikována dvojicí (FID, VID).

- **Velikost**
  Označuje délku textového řetězce dané verze. Do délky je započten pouze jeden z ukončovačů každého řádku (CR,LF). Informace o délce slouží zejména pro sériové čtení.

- **Čas vytvoření**
  Udržuje informaci o souborovém času vkládané verze, což může být užitečné pro orientaci uživatele v seznamu verzí.

- **Čas vložení**
  Je čas, kdy byla daná verze vložena do archivu. Podle tohoto času je možné později obnovit verzi k danému datu.

- **Verze a Komentář**
  Označují položky, do nichž může uživatel při vkládání verze zadat jednoduchý textový řetězec sloužící k pozdější orientaci.

Opět následuje malý příklad souboru *I2*:

```
0,0,625,1.1.2005 14:00:00,1.1.2005 14:05:03,0.5,první verze
2,1,847,1.1.2005 14:00:00,1.1.2005 14:05:03,,!last
1,2,115,2.1.2005 15:47:14,2.1.2005 16:02:25,0.6,drobné změny
1,3,206,5.1.2005 11:35:28,5.1.2005 11:37:16,0.7,další drobné změny
3,4,847,9.1.2005 18:58:44,9.1.2005 19:01:34,1.0,konečná verze
```

Soubory s verzemi

Kromě indexových informací jsou součástí archivu také vnitřní soubory s obsahy jednotlivých verzí. Každá verze je uložena do samostatného vnitřního souboru. Jména těchto vnitřních souborů mají formát F[FID]_[VID] tedy např. vnitřní soubor: „F2_4“ obsahuje konečnou verzi souboru „diffcz.html“ z předchozích příkladů.

Obsahem těchto vnitřních souborů je buď přímo text dané verze. To platí pro verze typy 0,2 a 3. Druhou možností je, že obsah vnitřního souboru tvoří rozdílová zpráva oproti předchozí verze, což se týká typu 1.

3.5 Operace s archivem

V sekci 3.2 zabývající se požadavky na archiv byly podrobně stanoveny priority pro časovou náročnost operací pracujících s archivem. Cílem této sekce bude prozkoumat, zda se podařilo pomocí vnitřního uspořádání archivu v sekci 3.4 dosáhnout požadovaných vlastností.

Měřítkem pro odhadnutí časové náročnosti operací bude zejména počet vkládání/vybírání souborů měřené počtem samostatných volání utilit zip.exe a unzip.exe. Doplňující informací bude i počet vkládaných/vybíraných souborů při jednom volání.

Funkčnost popsaná v následujících odstavcích je implementována v knihovně Archive. Knihovna Archive tvoří vyšší vrstvu nad knihovnou ZipFile a narozdíl od ní již není závislá na formátu archivního souboru. Více o uspořádání aplikace lze najít ve čtvrté kapitole.

První praktické výsledky nedosahovaly požadovaných vlastností, a proto byla implementace upravena přidáním několika podstatných vylepšení. Prvním z nich bylo využití možnosti hromadného čtení a zápisu. Je-li najednou do archivu zapisováno větší množství souborů, tedy utilita

zip.exe dostane více souborů pro vložení do archivu, podstatně to zrychlí celou operaci, jedná-li se o malé soubory pak téměř tolikrát, kolik souborů je vkládáno.

U hromadného čtení je situace poněkud odlišná. Zrychlení způsobené hromadným zpracováním více souborů není tak výrazné. Navíc unzip.exe posílá všechny soubory za sebou do výstupního proudu a je třeba si tento proud rozsekat na základě vlastní znalosti o velikosti souborů uvnitř archivu. Proto je třeba tuto informaci ukládat navíc do souboru s informacemi o verzích. Důvodem proč se tento postup přece jen vyplatí je fakt, že počet načítaných verzí bývá řádově vyšší než počet zapisovaných souborů, jak bude později rozebráno.Z těchto důvodů disponuje knihovna ZipFile podporou pro hromadný zápis i čtení souborů.

Druhým vylepšením je uchovávání poslední verze každého souboru zvlášť v plném znění. Jak vyplyne z odstavce o načítání poslední verze, je to jediný způsob jak garantovat rychlý přístup k poslední verzi každého souboru.

Třetím vylepšením je možnost uložit některou verzi do posloupnosti verzí v plném znění namísto standardního uložení ve formě diffu. To přináší možnost svrchu omezit čas potřebný pro načtení obecné verze z archivu.

Nyní je již možné probrat časovou náročnost jednotlivých operací:

1. Vložení nového souboru

První věcí, kterou je třeba zjistit je, zda se daný soubor již v archivu nalézá, či nikoli. Proto se z archivu načte Index a zpracuje se jeho obsah. V případě, že v archivu už je uložen soubor daného jména, postupuje se podle odstavce o vkládání nové verze. Nyní předpokládejme, že v archivu zatím soubor daného jména není. V tom případě se do archivu vloží upravený soubor Index a dvojice nových vnitřních souborů se seznamem verzí a s obsahem první verze. Celkem tedy 1 čtení a 3 vložení.

2. Vložení nové verze

I zde se nejdříve načte index a zjistí FID daného souboru. To umožní načíst seznam verzí. Nyní se využije fakt, že poslední verze je uložena jak ve formě diffu, tak i v plném znění. Načte se plné znění a vyrobí se diff nové verze vůči plnému znění poslední staré verze. Nyní se do archivu najednou vloží diff, nová verze v plném znění nahradí obsah poslední staré verze a dojde k ukončení upraveného seznamu verzí. Celkem tedy 3 čtení a 1 větší vložení.

3. Načtení poslední verze

Jako u všech změn se nejdříve načte index, následně příslušný seznam verzí, poté se načte poslední verze v plném znění. Zde se uplatňuje hlavní výhoda a účel uložení poslední verze v plném znění. Celkem tedy 3 čtení.

4. Načtení obecné verze

Po načtení indexu a příslušného seznamu verzí, je třeba zrekonstruovat příslušnou verzi. Na základě zadané podmínky (číslo, čas nebo označení verze) se zjistí VID načítané verze. Nyní se načtou všechny verze od posledního předcházejícího záchytného bodu, je-li v archivu umístěn. Pokud není, čtou se všechny verze od první až po hledanou. Zde se uplatní hromadné čtení. Může se jednat i o desítky verzí. Postupně se jedna po druhé odzadu aplikují na první verzi a výsledkem je zrekonstruovaná požadovaná verze. Počet čtení se může velmi lišit, ale uživatel má možnost jejich maximální počet omezit vhodným umístěním záchytných bodů. Jejich počet závisí na jeho potřebách. Celkem je tedy třeba 3 čtení z nichž jedno zahrnuje hromadné čtení jednoho či více souborů.

5. a 6. Výpis seznamu souborů nebo verzí konkrétního souboru

Vyžaduje načtení indexu a popřípadě jednoho soupisu verzí. Zde stačí 1 až 2 čtení.

7. Vymazání souboru z archivu

Pro vymazání souboru z archivu je třeba vyjmout všechny jeho verze, seznam jeho verzí a odstranit zmínku o něm z indexu. Každá smazaná verze odpovídá jedné operaci zápisu. Má-li soubor N verzí, pak celkový počet operací je 2 čtení a N + 3 „vložení“.

8. Vymazání verze souboru z archivu

Jde o poměrně netriviální operaci. Záleží na tom, kde se daná verze nalézá. Pokud jde o poslední verzi (patrně nejčastější případ), je třeba obnovit verzi předposlední a jejím obsahem nahradit obsah souboru s poslední verzí v plném znění, pak je třeba smazat diff s poslední verzí a nahradit obsah souboru se seznamem verzí.

Pokud mažeme verzi z prostředka seznamu je situace ještě komplikovanější. Nejdříve je třeba zrekonstruovat verzi předcházející mazané verzi a verzi po mazané verzi následující, vyrobit jejich diff, smazat mazanou verzi

a diffem nahradit obsah verze následující po smazané verzi. Pokud šlo o verzi poslední pak je nutné navíc upravit obsah plného znění poslední verze.

Situace se poněkud zjednoduší, pokud verze následující po smazané verzi je verzí v plném znění, pak není třeba nahrazovat její obsah. Mažeme-li verzi ze začátku seznamu, stačí zrekonstruovat verzi následující a jejím plným zněním nahradit její diff a samozřejmě patřičně upravit poslední verzi v plném znění pokud by druhá verze zároveň byla i verzí poslední. Celkem tedy v nejhorším případě N + 2 čtení a 4 vložení.

Závěr

Počet čtení a vložení se díky vhodnému návrhu a provedeným vylepšením podařilo u klíčových operací snížit na několik málo jednotek, což plně odpovídá cílům stanoveným v úvodu této kapitoly.

3.6 Testování archivu

Cílem této sekce je otestovat účinnost komprese poskytované archivem v porovnání s alterantivními metodami. Jako porovnávací vzorek jsem zvolil text této práce. Jde o celkem deset zdrojových souborů systému TeX. Jejich velikost postupně narůstá, jak se práce zvětšovala, od zhruba 7kB po přibližně 75kB.

Jako srovnávací programy pro DiffArchive jsem zvolil algoritmy zip, rar, tar/gzip a tar/bzip2. Všechny algoritmy jsou použity v defaultním nastavení. Výsledky shrnuje tabulka 3.1.

Tabulka 3.1: TEST KOMPRESE

Algoritmus	**Velikost archivu**
DiffArchive B	40 769
TAr/bzip2	44 504
DiffArchive A	68 384
TAr/GZip	171 927
Rar	193 105
Zip	197 604
Tar	536 576
Soubory celkem	528 228

Je třeba vysvětlit, proč se v tabulce DiffArchive vyskytuje dvakrát.

Zatímco DiffArchive A je varianta archivu s uchováváním poslední verze v plném znění, DiffArchive B, která zaujala první místo v tabulce je variantou, která neuchovává poslední verzi v plném znění nýbrž pouze její rozdílovou verzi.

Soudím, že vítězství nad nejlepšími komerčními algoritmy, přestože na datech, pro něž byl DiffArchive navržen je cenný úspěch.

3.7 Shrnutí

V této kapitole byla představena architektura archivu budovaného knihovnou DiffArchiv. V úvodu byly stanoveny dva cíle, jež musí tato architektura splňovat pro úspěšnou správu archivu: jsou jimi vysoká úroveň komprese dat a dostatečná rychlost práce s archivem. Cílem kapitoly bylo nalezení co nejlepšího kompromisu mezi nimi vzhledem k předpokládaným potřebám uživatele aplikace.

Pro maximální kompresi uložených souborů a jejich verzí byly použity dvě poměrně odlišné kompresní metody. První z nich je založena na rozpoznání rozdílů mezi jednotlivými uloženými verzemi týchž souborů a ukládání pouze informací o změnách namísto celých souborů. Tato metoda byla podrobně rozebrána ve druhé kapitole.

Druhou metodou je komprese na úrovni souborů s využitím standardního kompresního algoritmu. Byla stanovena kritéria výběru a jako vhodný byl zvolen algoritmus zip v implementaci volně dostupné knihovny Info-ZIP [3]. Tento výběr přináší výhody vysoké kompresní rychlosti, dobré komprese textu, snadné přenositelnosti a správu vnitřní souborové databáze.

V sekci 3.2 byly zmapovány jednotlivé operace pro správu archivu a byla stanovena jejich důležitost a očekávaná četnost. Výsledkem je seznam podmínek a priorit pro vnitřní uspořádání archivu a mechanismus přístupu k datům.

S přihlédnutím k metodám komprese, formátu souboru a podmínkám přístupu k datům bylo navrženo vnitřní uspořádání archivu založené na dvouúrovňovém indexování. Společně s několika dalšími vylepšeními toto uspořádání plní všechny stanovené podmínky, což bylo potvrzeno analýzou vstupně výstupních operací.

Testování archivu provedené v sekci 3.6 prokázalo velmi dobré výsledky algoritmu DiffArchive v porovnání s ostatními komprimačními algoritmy. Závěrem lze konstatovat, že bylo dosaženo všech cílů stanovených v úvodu této kapitoly.

Kapitola 4

Aplikace DiffArchive

Cílem této kapitoly je shrnout hlavní myšlenky použité při tvorbě aplikace DiffArchive. Podrobný popis technické implementace se nalézá v programátorské dokumentaci na doprovodném CD. V této kapitole bych se rád soustředil především na zásadní problémy a jejich řešení.

Aplikace DiffArchive má ve skutečnosti dvě hlavy, jsou jimi konzolová a grafická aplikace. Technicky jde vlastně o dva samostatné programy, ačkoli oba implementují podobnou funkčnost a sdílejí značnou část kódu. Abych udržel pořádek v názvosloví, budu se důsledně držet následujících tří označení:

Konzolová aplikace zahrnuje aplikaci pracující v prostředí příkazové řádky.

Grafická aplikace je aplikace s grafickým uživatelským rozhraním.

Projekt DiffArchiv v sobě zahrnuje sjednocení obou předchozích aplikací.

4.1 Vrstevnatý model

Vzhledem k rozsahu a množství nesourodých úkolů kladených na projekt DiffArchive jsem se rozhodl rozčlenit jej do většího počtu víceméně samostatných částí. Tyto části spolu nesdílejí žádná data a vystupují vůči sobě jako knihovny tříd a funkcí. Celou podobu projektu přehledně shrnuje obrázek 4.1.

Jak je patrné z diagramu, Grafická aplikace, představující tvář aplikace, komunkuje s dynamicky linkovanou knihovnou, která implementuje veškeré výkonné části kódu. Funkční části aplikace jsou dále rozděleny do

jednotlivých knihoven, na diagramu představovaných obdélníky. Komunikace mezi nimi se odehrává po šipkách vždy ukazujících ve směru od volajícího k volanému.

Hlavní výhodou použitého vrstevnatého modelu je přehlednost a možnost vytvářet, upravovat a testovat jednotlivé části nezávisle na sobě. Díky vrstevnatému modelu bylo možné odsunout nepřenositelné části kódu do dvou knihoven a usnadnit tak případnou budoucí přenositelnost projektu.

4.2 Knihovny projektu DiffArchiv

Obsahem této sekce je popis jednotlivých knihoven projektu DiffArchiv, jejich rozhraní i náznak vnitřní implementace.

ZipFile

Jazyk: C++
Zdroje: zipfile.h, zipfile.cpp

Knihovna ZipFile slouží jako rozhraní pro práci s archivy ve formátu zip. K tomu jí slouží dvojice programů pro příkazovou řádku *Zip.exe* a *UnZip.exe* z volně šířitelné knihovny Info-ZIP [3]. Více informací o těchto programech lze nalézt ve třetí kapitole této práce.

S oběma programy zajišťujícími kompresi a dekompresi archivu zip komunikuje knihovna ZipFile prostřednictvím příkazové řádky a systémových rour (angl. pipe). Využívá systémové funkce API (např.: CreateProcess nebo CreatePipe). Kód knihovny ZipFile tedy není přenositelný na jinou platformu než MS Win32.

S ohledem na snadnou přenositelnost představuje implementovaná funkčnost knihovny ZipFile minimalistickou množinu funkcí. Obsahuje zejména nepřenositelné části aplikace a základní operace pro práci s archivem formátu zip: přidávání a mazání souborů a poskytování základních informací o archivu. Knihovna ZipFile obsahuje hlavní třídu ZipFile, která se vůči uživateli knihovny chová jako archiv formátu zip a několik malých tříd pro předávání a vracení parametrů.

Archive

Jazyk: C++
Zdroje: archive.h, archive.cpp

Knihovna Archive stojí nad knihovnou ZipFile. Zapouzdřuje chování Archivu souborů a jejich verzí (podle principů obsažených v kapitole číslo tři) nezávisle na volbě komprimačního algoritmu. Její hlavní třída Archive implementuje funkce umožňující číst z archivu jednotlivé inexovací informace a vyhledávat podle nich požadovaná data, vkládat a mazat do a z archivu soubory i jejich verze. Třída Archive také podporuje hromadné čtení a zápis souborů, což urychluje práci s archivem. Součástí knihovny jsou i menší třídy určené pro udržování a předávání dat o archivu, obsažených souborech a jejich verzích.

Kód knihovny Archive využívá pouze standardních knihoven jazyka C++ a je tedy plně přenositelný.

DiffLib

Jazyk: C++
Zdroje: abstractdiff.h, linediff.h, worddiff.h, chardiff.h, difftree.h, abstractdiff.cpp, linediff.cpp, worddiff.cpp, chardiff.cpp, difftree.cpp

Knihovna DiffLib je pravým jádrem projektu DiffArchiv. Obsahuje třídy pro vytváření a používání rozdílových zpráv mezi verzemi textových souborů. Algoritmus Vícevrstvý Diff a jeho varianty jsou podrobně popsány ve druhé kapitole, zde se budu věnovat toliko jeho implementaci.

Součástí knihovny DiffLib je třída AbstractDiff. Jde o abstraktní třídu implementující jádro algoritmu Diff [2] nad obecnou dělící entitou. Od této třídy je zděděna trojice tříd LineDiff, WordDiff a CharDiff, jež implementují algoritmus Diff nad řádky, slovy a znaky.

Druhou část knihovny DiffLib představují třídy zachycující výstup diffovacích tříd. Výstupem algoritmu LineDiff je třída DiffTree. Třída DiffTree je potomkem třídy `std::vector<LineNode>`. Umožňuje uchovávat výstup řádkového diffu tak, že každý záznam je prvkem typu LineNode. DiffTree umí svůj obsah uložit do textového řetězce a opět jej načíst. Umí také svůj obsah aplikovat na původní soubor a vyrobit tak novou verzi. K tomu se využívá netriviálního rekurentního algoritmu běžícím nad objekty stromu změn.

Úkolem třídy LineNode je uchovávat řádkové změny. V případě, že jde o změny typu přidání (add) nebo smazání (delete) řádku, je LineNode koncovým uzlem. Jde-li naopak o změnu typu change pak objekt typu LineNode obsahuje ukazatel na objekt typu `std::vector<WordNode>`. Ten zachycuje výstup algoritmu WordNode. Obdobně objekt typu WordNode

obsahuje ukazatel na vektor s objekty typu CharNode obsahující výstup algoritmu CharDiff.

Kód knihovny DiffLib využívá pouze standardních knihoven jazyka C++ a je tedy taktéž plně přenositelný.

StringList

Jazyk: C++
Zdroje: stringlist.h, stringlist.cpp

Knihovna StringList odvozená od `std::vector<std::string>` obsahuje jedninou třídu StringList sloužící k práci s kolekcí textových řetězců. Umožňuje jejich načtení z textu (rozdělené podle konců řádků) nebo textového souboru, či uložení tamtéž nebo jejich konverzi z a na CSV [4]. Jde o mimořádně praktický způsob práce s textem. Tato třída je široce používaná ve všech ostatních částech projektu.

I kód knihovny DiffLib využívá pouze standardních knihoven jazyka C++ a je tedy také plně přenositelný.

Utils

Jazyk: C++
Zdroje: utils.h, utils.cpp

Tato knihovnička užitečných funkcí obsahuje drobné funkce použitelné na více místech projektu tematicky nespadající do žádné z nich. Jde zejména o funkce konverzní, formátovací nebo systémově závislé.

Konzolová aplikace / DLL

Jazyk: C++
Zdroje: main.h, main.cpp, add.cpp, cmdline.cpp, delete.cpp, diff.cpp, extract.cpp, help.cpp, list.cpp

Konzolová aplikace zpřístupňuje veškerou výše popsanou funkčnost formou aplikace pro příkazový řádek. Alternativní možností je zkompilovat ji jako dynamicky linkovanou knihovnu a začlenit ji jako součást do grafické aplikace.

Grafická aplikace

Jazyk: Object Pascal / Delphi
Zdroje: UArchive.pas, Unit1.pas, Unit2.pas, Unit3.pas, ...

Grafická aplikace DiffArchive umožňuje uživateli pohodlný přístup k archivu typu DiffArchive. Obsahuje podporu pro funkce srovnatelné s konzolovou aplikací obohacené o komfortní ovládání na úrovni standardní grafické aplikace typu MDI (multi document interface).

Jako jazyk pro tvorbu Grafické aplikace byl zvolen Object Pascal v prostředí pro rychlý vývoj aplikací Delphi. To umožnilo zvládnutí vytvoření aplikace v rozumném rozsahu i čase.

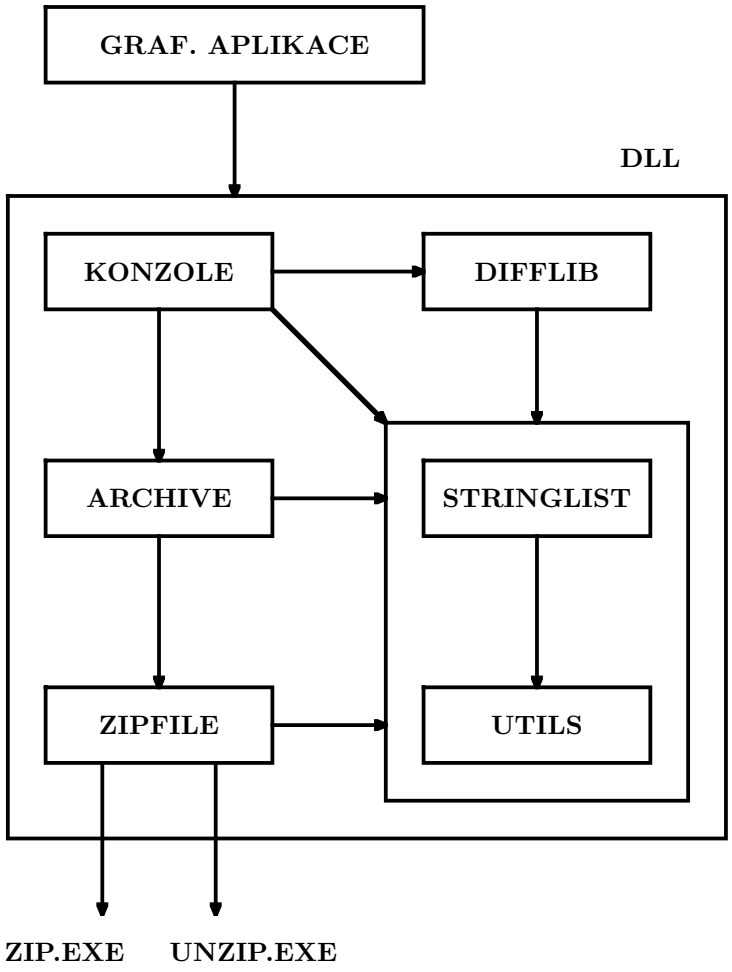
Grafická část tvoří pouze lehkou nadstavbu nad výkonnou dynamicky linkovanou knihovnou obsahující veškerou faktickou funkčnost. Pro přenos strukturovaných dat mezi grafickou aplikací a dynamickou knihovnou je použit univerzální textový formát CSV [4].

4.3 Shrnutí

V této kapitole byl představen Projekt DiffArchiv z programátorského hlediska. Jsou zde popsány obě aplikace, v projektu obsažené, tedy konzolová i grafická. Byl představen vrstevnatý model sloužící k usnadnění orientace v relativně nesourodých úkolech, z nichž se projekt skládá.

Jak bylo detailně popsáno, projekt DiffArchive využívá dvou programovacích jazyků, tedy C++ a Delphi. Jazyk C++ byl pro svou rychlost a univerzální přenositelnost použit pro tvorbu výkonných částí a konzolové aplikace. Prostředí Delphi usnadnilo vytvoření přívětivého uživatelského prostředí.

V sekci 4.2 byly zběžně popsány všechny knihovny obsažené v projektu, podstatné rysy jejich implementace, funkční závislosti mezi nimi i možnosti jejich přenositelnosti.



Obrázek 4.1: Diagram knihovních částí projektu DiffArchive

Kapitola 5

Závěr

V této práci byl podrobně prozkoumán způsob, jak efektivně udržovat verzovaný archiv textových dokumentů – tedy archiv zachycující různé verze postupně vznikajících textových souborů.

V úvodní kapitole obsahující zadání práce byly stanoveny tři hlavní cíle práce: prozkoumání možnosti neukládat celé verze, ale pouze tu část jejich obsahu, která se změnila oproti verzi předešlé, dále navržení příhodného vnitřního uspořádání archivu verzí s ohledem na jeho očekávané používání a konečně vytvoření aplikace účinně realizující výsledky předchozích dvou kapitol.

V druhé kapitole byl zmapován charakter změn textových dokumentů. Jednotlivé změny byly rozčleněny do čtyř tříd podle četnosti a mohutnosti změn. Dále byly stanoveny požadavky na algoritmus rozpoznávající původní a změněné části souboru. Jako vhodný kandidát na takový algoritmus byl zvolen Diff, algoritmus rozdílového porovnávání souborů [2].

Algoritmus Diff vyhověl podmínkám na rychlost i funkční vlastnosti, ale tváří v tvář drobným a rozprostřeným změnám byl shledán příliš hrubým. Příčinou se ukázala být velikost základní dělící jednotky, tedy jednoho řádku. Byly provedeny pokusy s alternativními jednotkami: slovem a znakem. Jejich výsledky ukázaly, že Diff postavený na slovech či znacích má podstatně lepší schopnost popisovat drobné změny (až několikanásobně) avšak za cenu velmi značného zpomalení běhu algoritmu.

Jako ideální řešení se jevila kombinace obou algoritmů, tedy takový algoritmus, který by si dokázal poradit s velkými úseky textu s rychlostí klasického Diffu, ale v případě nutnosti by dokázal dostatečně zjemnit na úroveň slov a znaků. Tak se zrodil Víceúrovňový Diff. Detailní testování

ukázalo, že Víceúrovňový Diff úspěšně spojuje přednosti obou vzorů, tedy rychlost i přesnost. Tento výsledek lze považovat za jednoznačný úspěch.

Třetí kapitola je věnována konstrukci archivu. Nejdříve byly stanoveny cíle, které by měl archiv splňovat: kompresi dat a rychlost použití. Pro dosažení vysoké úrovně komprese byly použity dvě metody: komprese pomocí ukládání rozdílových verzí popsaných v druhé kapitole a komprese archivu jako celku konvenčním binárním algoritmem. Zejména s ohledem na dostatečnou rychlost komprese byl zvolen algoritmus zip ve veřejně dostupné implementaci knihovny Info-ZIP [3].

Vnitřní upořádání archivu bylo navrženo s ohledem na analýzu předpokládaných dotazů. U každého dotazu byla vyhodnocena očekávaná četnost a citlivost na rychlost provedení. Podle toho bylo navrženo uspořádání archivu pomocí dvouúrovňového indexování [1].

Obsahem čtvrté kapitoly je popis dvojice aplikací vyvíjených v rámci projektu DiffArchive. První z nich je konzolová aplikace vytvořená v jazyce C++ a určená pro snadnou přenositelnost mezi platformami. Druhou je potom aplikace s grafickým uživatelským rozhraním určená pro platformu MS Win32. Ta je vytvořena v prostředí Delphi. Výkonná část kódu je sdílena oběma aplikacemi formou statického linkování v případě konzolové aplikace a dynamického v případě grafické aplikace.

Při vývoji obou aplikací byl kladen důraz na precizní objektový návrh a rozčlenění kódu do autonomních částí – knihoven. To zajišťuje dobrou udržovatelnost a budoucí rozvoj kódu.

Domnívám se, že se podařilo naplnit jak zadání práce, tak i všechny cíle vytyčené v jejím úvodu. Jak prokázaly praktické testy, algoritmus Víceúrovňového Diffu i obě aplikace DiffArchive představují účinný nástroj pro správu verzovaného archivu textových dokumentů.

Literatura

[1] Pokorný, J., Žemlička M.: *Základy implementace souborů a databází*, Karolinum 2004,

[2] Hunt, J. W., McIlroy, M. D.: *An Algorithm for Differential File Comparison*, Bell Telephone Labs, Murray Hill, NJ, USA, Aug. 1976, (http://www.cs.dartmouth.edu/~doug/diff.ps)

[3] Sdružení osob Info-ZIP: *http://www.info-zip.org/*

[4] Comma-separated values, hodnoty oddělené čárkami *http://cs.wikipedia.org/wiki/CSV*