VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# MEDIADIFF – DIFF PRO STATICKÉ OBRÁZKY

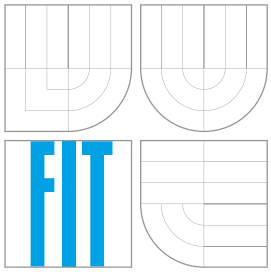
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN BROTHÁNEK

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# MEDIADIFF – DIFF PRO STATICKÉ OBRÁZKY

MEDIADIFF – DIFF FOR STATIC IMAGES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN BROTHÁNEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR CHMELAŘ
SUPERVISOR

BRNO 2011

# Abstrakt

Tato bakalářská práce se zaměřuje na porovnávání dokumentů v digitální podobě, zejména statických obrázků. Představuje algoritmy, které je možno použít k obecnému porovnávání i algoritmy specializované na obrazová data. Jsou uvedeny možnosti, které nyní uživatel má, pokud potřebuje dokumenty porovnávat. Na základě této současné situace je navržen a implementován prototyp modulární aplikace MediaDiff pro porovnávání různých typů dokumentů. Jsou popsány jednotlivé etapy vývoje aplikace, nalezené problémy a jejich řešení. Jsou diskutovány i budoucí možnosti vývoje aplikace.



# Abstract

This bachelor's thesis focuses on comparison of digital documents, especially static images. This paper presents algorithms, which are usable for general data comparison and also specific algorithms major in image data comparison. We introduce possibilities of document comparison which are now available for users. On the basis of this current situation, a prototype of new modular application MediaDiff for various documents comparison is designed and implemented. We describe in detail each stage of the application development, problems which had arisen and their solutions. The possibilities of future development of the application are discussed.



# Klíčová slova

obrázek, porovnávání, podobnost, vzdálenostní metriky, Hammingova vzdálenost, Levenshteinova vzdálenost, LCS, registrace obrazů, sesouhlasení, diff, cmp, ImageMagick, MediaDiff

# Keywords

image, comparison, similarity, affinity, distance metrics, Hamming distance, Levenshtein distance, LCS, image registration, diff, cmp, ImageMagick, MediaDiff


# Citace

Jan Brothánek: Mediadiff – diff pro statické obrázky, bakalářská práce, Brno, FIT VUT v Brně, 2011

# Mediadiff – diff pro statické obrázky

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Petra Chmelaře

.....
Jan Brothánek
17. května 2011

## Poděkování

Děkuji vedoucímu této bakalářské práce, panu Ing. Petru Chmelaři, za osobní konzultace, odborné vedení, cenné rady, náměty a připomínky.

# Obsah

# Kapitola 1

# Úvod

S rozvojem informačních technologií se stále více stávají počítače běžnou součástí životů většiny lidí. Informační technologie již nejsou výsadami v domácnostech, ale jsou vnímány jako její běžné součásti, stejně jako například kuchyňské náčiní. Využívání počítačů pro osobní potřeby do značné míry napomáhá možnost zjednodušit běžné úkony. Mnoho lidí dnes například upřednostňuje emailovou komunikaci před běžnými dopisy nebo práci s dokumenty v elektronické podobě raději než v podobě ručně psané. Digitálně zaznamenaná hudba přehrávaná například z osobního počítače začíná nahrazovat dříve obvyklý poslech z analogových nosičů zvuku nebo z vysílání rádií. Rozvíjí se práce s digitálními fotografiemi nebo jinými obrazovými daty na úkor analogových fotografií s chemickými metodami záznamu i jiná práce s drobnějšími obrázky.

Již před rozvojem osobních počítačů takových, jaké známe dnes, bylo výhodné strojově zpracovávat produkty práce s počítačem, jakými byly v této době prosté textové dokumenty nebo zdrojové kódy. Vznikla tak například potřeba porovnávání takovýchto dat. Bylo vhodné automaticky určovat rozdíly mezi různými verzemi téhož textu, například pro snadné sloučení práce více lidí. Proto vznikl program `diff`, který umožňuje vyhledávání rozdílů mezi textovými soubory tak, že při postupném průchodu oběma dokumenty hledá řádky, které se liší a výstupem jsou následně vyznačené změny. Takto je možno snadno kontrolovat práci na dokumentu v průběhu času. S takovýmto výstupem může také pracovat program `patch`, který provádí aktualizaci původního dokumentu bez nutnosti přenesení nezměněných řádků. Nástroj `diff` se stal součástí systému Unix a dnes je již neodmyslitelným prvkem systémů tohoto typu a závisí na něm mnohé další systémové nástroje.

Vzhledem k velmi odlišné povaze různých typů dat není snadné provádět porovnávání souborů tak, aby měl výsledek dobrou vypovídací hodnotu pro běžného uživatele. Zatím neexistuje standardní porovnávací nástroj podobný programu `diff` s možností použití pro obecné typy dat, který by byl zaměřen na výstižnou prezentaci výsledků uživatelům.

Tato práce se zabývá způsoby, jakými by bylo možno implementovat aplikaci využitelnou pro porovnávání různých, nejen textových dokumentů s ohledem na potřeby běžných uživatelů. Dále nabízí též ukázkovou implementaci vycházející ze závěrů této práce.

Kapitola 2 obsahuje teoretický úvod do problematicky porovnávání. Přibližuje i algoritmus, na jehož základě pracuje program `diff`. V kapitole 3 jsou diskutovány možnosti porovnávání obrazových dat, tedy algoritmy, již existující aplikace a knihovny umožňující porovnávání obrázků. Obsahuje též zhodnocení těchto možností a zamyšlení nad současnou situací. Kapitola 4 seznamuje s postupem vývoje aplikace MediaDiff a dává náhled do problémů, které se během vývoje vyskytovaly, a jejich řešení. V této kapitole nalezneme též popis principů, na kterých je aplikace vystavena.

# Kapitola 2

# Obecné porovnávací algoritmy

V oblasti porovnávání digitálních dat je možno setkat se se třemi základními přístupy.

Nejjednodušším z nich je postupné vyznačení rozdílů znak po znaku, tedy podobný princip, na jakém pracuje program `cmp`. Tento přístup je přiblížen, jelikož porovnávání obrazových dat, kterým se tato práce mimo jiné zabývá, se v některých případech může provádět právě tímto způsobem.

Jiným často využívaným přístupem je hledání rozdílů mezi dvěma soubory dat, obecně řešení problému nalezení nejdelšího společného podřetězce dvou nebo více řetězců. Tento přístup je pro tuto práci stěžejní, protože na jeho principu pracuje program `diff`. Bude proto detailně popsán.

Dalším možným přístupem je snaha o ohodnocení míry podobnosti dvou řetězců, obecně metrika editační vzdálenosti. Tohoto přístupu se využívá, není-li nutné znát výčet a popis změn, ale stačí číselné ocenění podobnosti. Pomocí něho je možno například nalézt nejvíce nebo nejméně změněný řetězec v množině řetězců relativně k referenčnímu řetězci. Tento přístup je též nastíněn, jelikož může být výhodný v některých případech porovnávání multimediálních dat.

Tato kapitola představuje uvedené přístupy a pojednává o možnostech řešení těchto problémů. Z důvodu rozsahu nerozvádí jiné přístupy, které se této práce bezprostředně netýkají.

## 2.1 Základní úvod do problematiky

V této sekci jsou představeny základy probírané problematiky. Pokročilejší poznatky však neposkytne, ty budou předmětem dalších sekcí a kapitol.

### 2.1.1 Metoda postupného vyznačování rozdílů znak po znaku

Tato metoda je velmi snadná a intuitivní, proto se v literatuře nevyskytuje. Přesto ji přiblížíme, protože je základem důležitých porovnávacích nástrojů.

Porovnávání touto metodou lze velmi snadno realizovat, jak je možno ozřejmit na příkladě 2.1.1.

**Příklad 2.1.1.** Mějme následující řetězce symbolů:

- `a b c d e f g h`
- `a x c d e y g z`

Tyto řetězce se na některých místech liší, zatímco většina symbolů zůstala zachována.

Algoritmus pro vyznačování rozdílů poskytuje jako svůj výstup řetězec, na jehož každé pozici je zaznačeno, zda se na příslušném místě v obou vstupních řetězcích symboly liší. Algoritmus tedy prochází oba řetězce tak, že z každého vždy postupně načte další symbol a oba načtené symboly vzájemně porovná. Je zřejmé, že tedy porovnává postupně znaky na shodných místech v obou řetězcích. Pakliže jsou aktuálně porovnávané znaky shodné, do výstupního řetězce je na stejnou pozici zapsána hodnota 0. Jestliže symboly shodné nejsou, je zapsána hodnota 1. Výstupem porovnání uvedeného příkladu tedy bude řetězec:

```
0 1 0 0 0 1 0 1
```

Popis využitelnosti a problémů tohoto algoritmu je představen v 2.2.

### 2.1.2 Problém nejdelšího společného podřetězce

Metoda porovnávání znak po znaku, jak bylo uvedeno výše (2.1.1), může být sice užitečná a velmi rychlá a v mnoha případech postačuje. Pokud bychom však potřebovali preciznější metody porovnávání, například takové, které by braly v úvahu posuny symbolů, vkládání apod., využity mohou být složitější metody. Důležitou skupinou použitelných algoritmů mohou být algoritmy založené na řešení problému nejdelšího společného podřetězce.

S problémem nalezení nejdelšího společného podřetězce, v anglické literatuře běžně „Longest common subsequence problem“ (odtud značeno LCS), se v informatice setkáváme velmi často.

Porovnávání řetězců je stěžejní operací v různých prostředích: jazyková korektura, kde je snaha programově nalézt nejpodobnější řetězec k obdrženému chybnému řetězci. V molekulární biologii je nutno porovnávat DNA nebo proteinové sekvence za účelem zjištění podobnosti. V souborových archivech se vyskytuje potřeba uchování více verzí zdrojového textu pokud možno efektivně, například uložením původní verze vcelku, avšak ostatní verze mohou být zkonstruovány z popisu rozdílů oproti původní verzi. Atd. Zřejmou metrikou blízkosti dvou řetězců je pak nalezení počtu všech shodných symbolů (se zachováním pořadí) v řetězcích. Takto je neformálně definován nejdelší společný podřetězec [3].

Problém a jeho hledané řešení je možno ilustrovat na následujícím příkladě 2.1.2:

**Příklad 2.1.2.** Podle [16].
Předpokládejme dvě sekvence symbolů:

- `a b c d f g h j q z`
- `a b c d e f g i j k r x y z`

V těchto sekvencích budeme chtít nalézt nejdelší sekvenci symbolů, které jsou přítomny v obou původních sekvencích ve stejném pořadí. Chceme tedy nalézt novou sekvenci, kterou je možno získat z první sekvence odstraněním některých symbolů, nebo z druhé sekvence odstraněním jiných symbolů. Též chceme, aby výsledná sekvence byla nejdelší možná. V tomto případě je požadovaný výsledek následující:

```
a b c d f g j z
```

Z tohoto výsledku je již možné snadno generovat výstup, který požadujeme po programu `diff`: pokud symbol není obsažen ve výsledném podřetězci, ale nachází se v originálním řetězci, byl tedy smazán a je značen „`-`“. Pokud se naopak symbol nachází ve druhém řetězci a nikoliv ve společném podřetězci, byl přidán a je značen „`+`“. Ostatní symboly pak zůstaly nezměněny. Výstup bude vypadat následovně:

```
e   h i   q   k r x y
+   - +   -   + + + +
```

Konkrétní algoritmus je více popsán v 2.3.

### 2.1.3 Vzdálenostní metriky

V mnoha případech je nutné znát míru podobnosti řetězců.

Kvantifikace podobnosti řetězců je důležitým vědeckým problémem, protože v mnoha odvětvích, jakými jsou převody textů, zpracování signálů nebo bioinformatika, mohou být klíčové informace zaznamenány jako posloupnost symbolů. Přitom nemusí být zaznamenány vždy shodně, například z důvodu nepřesností digitalizace analogových dat. K porovnávání těchto záznamů je pak vhodné užití vzdálenostní metriky, běžně některé metriky editační vzdálenosti (v angl. literatuře se lze setkat s termínem „edit distance“). Ta obvykle zahrnuje operace odstranění, vložení a náhradu jednotlivých znaků a může být definována jako nejmenší cena transformace jednoho řetězce na řetězec jiný s využitím sekvence vážených editačních operací [20].

Příklad výpočtu podobnosti řetězců s využitím metriky editační vzdálenosti je možno demonstrovat na příkladě 2.1.3

**Příklad 2.1.3.** Podle [18].
Uvažujme následující řetězce:

- `kitten`
- `sitting`

Vzdálenost (zde Levenshteinova – detailně v 2.4.2) řetězců je 3. Výpočet je proveden jako nejmenší počet změn nutných k přepisu jednoho řetězce na druhý. Tyto změny jsou:

1. `kitten` → `sitten` (náhrada `k` znakem `s`)
2. `sitten` → `sittin` (náhrada `e` znakem `i`)
3. `sittin` → `sitting` (vložení `g`)

Vzdálenostní metriky řetězců budou přiblíženy v 2.4.

### 2.1.4 Registrace obrazů

Obecné porovnávací algoritmy jsou však také upraveny do konkrétních metod pro práci s obrazovými daty. V současnosti se používají různé metody, které si kladou za cíl zpracování různých typů dat s různými výstupy. Souhrnně je nazýváme *Registrace obrazů*, někdy též *Sesouhlasování obrazů*.

Registrace obrazů je proces nanášení dvou nebo více obrazů přes sebe, přičemž vstupní obrazy mohou být pořízeny v různých časech z různých úhlů pohledu a různými zařízeními. Cílem procesu je geometricky zarovnat referenční a zpracovávaný obraz. Rozdíly mezi těmito obrazy jsou dány různými podmínkami pořízení obrazů. Registrace obrazů je kritickým bodem všech metod analýzy obrazů, ve kterých je konečná informace získána kombinací různých zdrojů dat. Těmito metodami mohou být např. syntéza obrazů, detekce změn nebo vícekanálová rekonstrukce obrazu. Typickými oblastmi použití registrace obrazů jsou vzdálené snímání (např. monitorování životního prostředí, detekce změn, předpovědi

počasí, vytváření obrazů s velmi vysokým rozlišením nebo integrace do geografických informačních systémů (GIS)), medicínské obory (kombinace dat z počítačové tomografie (CT) k zisku kompletních informací o pacientovi, monitorování růstu nádorů, ověřování výsledků ošetření nebo porovnávání stavu pacientova těla s referenčním anatomickým atlasem), kartografie (modernizace mapových podkladů) a počítačové vidění (rozpoznávání cíle nebo automatická kontrola kvality) [21].

Bližší popis konkrétních metod registrace obrazů uvádíme v 2.5.

## 2.2 Prosté porovnávání znak po znaku

Algoritmus porovnávání řetězců znak po znaku produkuje řetězec, kde je na každé pozici hodnota 0 v případě, že jsou na stejném místě v obou vstupních řetězcích tytéž symboly, v opačném případě je zapsána hodnota 1.

Podobný výstup můžeme obdržet od programu `cmp` s použitím přepínače `-l` (s tím rozdílem, že pozice beze změn nejsou vypisovány).

Sekvenční porovnávání symbolů v řetězcích vedle sebe je velmi rychlé a algoritmus je intuitivní. Je vhodné například v datech, kde vznikají lokální úpravy operací náhrady symbolu, nikoliv však odstranění nebo vložení. Typicky se jedná o kreslené obrázky s jednoduchými úpravami menších částí. Pokud například uživatel upraví obrázek tak, že v jeho části změní barvu, je tento způsob porovnání velmi vhodný.

Jeho zásadním nedostatkem je však právě nepostižení operací odstranění nebo vložení symbolů. Pokud bychom například chtěli porovnávat textové soubory řádek po řádku, potom při vložení řádku na první pozici souboru (ve kterém žádné dva po sobě jdoucí řádky nejsou shodné) bude nový soubor vyhodnocen jako zcela odlišný od původního. Tento přístup je proto nevhodný pro účely porovnávání, pokud požadujeme výstup podobný výstupu programu `diff`.

S problémem nepostižení operací odstranění a vložení symbolů úzce souvisí také to, že algoritmus z definice porovnává pouze řetězce stejné délky. Obvykle ale bývá vhodné dodefinovat chování v případech různých délek vstupních řetězců. Zde již záleží na konkrétním předpokládaném využití tohoto algoritmu. Např. program `cmp` ve výchozím nastavení s přepínačem `-l` při zjištění konce některého souboru vypisuje „EOF“ a název neočekávaně ukončeného souboru. Jeho chování je ale možno měnit použitými přepínači. Jinou možností je pokusit se nalézt pozici menšího řetězce ve větším a poté provést porovnávání. To bývá vhodné zejména při porovnávání obrazových dat, kdy se snažíme lokalizovat výřez většího obrázku, který by nejvíce odpovídal obrázku menšímu.

## 2.3 Nalezení nejdelšího společného podřetězce

Problém *nejdelšího společného podřetězce* a jeho řešení jsou velmi obsáhlá témata. Proto v této práci uvedeme jen některé jejich nejdůležitější aspekty. Více podrobně je problém popsán v příloze této práce (F). Nejprve byl celý text součástí práce, avšak později bylo vyhodnoceno, že není k tématu práce příliš relevantní.

Definice problému nejdelšího společného podřetězce je podle [3] následující:

Mějme dva vstupní řetězce: $X[1..m]$ a $Y[1..n]$, které jsou podmnožinami množiny $\Sigma^*$, kde $\Sigma$ značí vstupní abecedu symbolů $\sigma$. Podřetězec $S[1..s]$ řetězce $X[1..m]$ je získán odebráním $s - m$ symbolů řetězce $X$. Společný podřetězec ($cs$) řetězce $X[1..m]$ a řetězce $Y[1..n]$, značený $cs(X, Y)$, je podřetězec, který se nalézá v obou řetězcích (bez újmy na obecnosti

uvažujeme $m \leq n$). Nejdelší společný podřetězec ($lcs$) řetězce $X[1..m]$ a řetězce $Y[1..n]$, značený $lcs(X, Y)$, je takový společný podřetězec $cs(X, Y)$ v množině všech podřetězců $cs(X, Y)$, který je nejdelší. Délka $lcs(X, Y)$ je značena $r(X, Y)$.

Tradiční technika řešení problému $lcs(X[1..m], Y[1..n])$ spočívá ve vyvození nejdelšího společného podřetězce všech možných kombinací prefixů vstupních řetězců. Rekurentní vztah pro prodloužení délky LCS pro každý pár prefixů $(X[1..i], Y[1..j])$ je následující:

$$R[i,j] = \begin{cases} 0 & \text{pokud } i = 0 \text{ nebo } j = 0 \\ R[i-1, j-1] + 1 & \text{pokud } X[i] = Y[j] \\ max\ \{R[i-1, j], R[i, j-1]\} & \text{pokud } X[i] \neq Y[j] \end{cases} \tag{2.1}$$

kde jsme použili tabulkovou notaci $R[i, j] = r(X[1..i], Y[1..j])$. Podstatou této rekurze je, že jedna $R$-hodnota záleží právě na třech sousedních hodnotách v tabulce, tzn. je velmi závislá na umístění. Po naplnění tabulky je délka zapsána v $R[m, n] = r(X[1..m], Y[1..n])$. Společný podřetězec je nalezen zpětným vyhledáváním od $R[m, n]$, kde na každém kroku buď následujeme ukazatel nastavený během výpočtu hodnot, nebo přepočítáme předchůdce, který ovlivnil hodnotu aktuálního prvku tabulky. Pokaždé, když je nalezena shoda (použito pravidlo $R[i-1, j-1]+1$), nalezli jsme symbol v LCS. Tímto způsobem můžeme spojit cestu tabulkou, dokud není nalezena nulová délka. Obecně může být takových cest více, protože LCS není nutně jedinečný. Pro nalezení všech LCS můžeme systematicky procházet grafem vytvořeným ukazateli (např. použitím prohledávání do hloubky). Jako příklad můžeme ukázat tabulku 2.3, náležící k výpočtu $r(abcdbb, cbacbaaba)$.

Tabulka 2.1: Tabulka výpočtu LCS algoritmem 2.1, podle [3]

| | | | | Y | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | c | b | a | c | b | a | a | b | a |
| | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | a | 1 | 0 | 0 | 0 | [1] | 1 | 1 | (1) | (1) | 1 | (1) |
| | b | 2 | 0 | 0 | [1] | ←1 | 1 | [2] | 2 | 2 | (2) | 2 |
| X | c | 3 | 0 | [1] | 1 | 1 | [2] | 2 | 2 | 2 | 2 | 2 |
| | d | 4 | 0 | [1] | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| | b | 5 | 0 | 1 | [2] | 2 | 2 | [3] | ←3 | ←3 | (3) | 3 |
| | b | 6 | 0 | 1 | (2) | 2 | 2 | (3) | 3 | 3 | [4] | ←4 |

V tomto případě $r(X, Y) = 4$ a nejdelší společný podřetězec náležící k vyobrazené cestě je $bcbb$ (jiným může být $acbb$).

Definice obecného algoritmu je příliš rozsáhlá, a ačkoliv byla součástí této práce, později bylo vyhodnoceno, že není k jejímu tématu příliš relevantní. Proto zde nyní není uvedena. Uvádíme ji však v příloze této práce (F.3).

### 2.3.1 Aplikace problému LCS na obrazová data

Problém nejdelšího společného podřetězce je možno také zobecnit a použité algoritmy upravit tak, aby bylo možno hledat nejdelší společné podřetězce i ve více-dimenzionálních datech, jakými jsou například obrázky. Algoritmy poté pracují v polynomiálním čase. Řešení je však implementačně náročné a výpočet LCS pro obrázky (tzn. dvojrozměrné řetězce)

by byl velice časově a paměťově náročný, jelikož právě polynomiální složitost způsobuje při běžně používaném počtu barev (tedy počtu symbolů vstupní abecedy $\Sigma$) náročnost výpočtu velmi vysokou. Používá se však v některých případech porovnávání malých obrázků se značně redukovaným počtem barev (tzv. ikony) [13].

## 2.4 Vzdálenostní metriky podobnosti řetězců

Vzdálenostní metriky vyhodnocující podobnosti řetězců provádí hledání shod vzorů v řetězcích, kde vzor nebo řetězec (případně oba) obsahují porušení. V některých odvětvích potřebujeme obnovovat původní signály po přenosu přes kanál s rušením, jinde můžeme chtít porovnávat sekvence DNA s možnými mutacemi, nebo například vyhledávání v textu porušeném různými přepisy. Obecným problémem je tedy nalézt řetězec, v němž se nachází vzor s omezeným počtem „chyb“. Každé praktické použití pak potřebuje jiný algoritmus a jako chyby chápe různé druhy porušení, které definují, jak rozdílné jsou vstupní řetězce. Základní myšlenkou je nalézt takovou metriku, která by dávala jako výsledek porovnání tím nižší velikost „vzdálenosti“ řetězců, čím vyšší je pravděpodobnost, že se jedná o tentýž řetězec s vloženými chybami. Taková metrika se nazývá metrikou „editační vzdálenosti“ [15].

Definice této metriky je podle [5] následující:

Abeceda $I, D, R, M$ definuje operace „vložení“, „odstranění“, „náhrada“ a nulovou operaci „ponechání“ symbolu v řetězci. Řetězec vystavený nad touto abecedou, která popisuje transformaci jednoho řetězce na řetězec jiný, je takzvaný „přepis změny“ (anglicky „edit transcript“) těchto dvou řetězců.

Obecně při vstupních řetězcích $S_1$ a $S_2$ a vstupním přepisu změny pro $S_1$ a $S_2$ je transformace provedena úspěšnou aplikací specifikovanou v přepisu změny na následující symbol v příslušném řetězci. Zvláště pak, nechť $next_1$ a $next_2$ jsou ukazatele na $S_1$ a $S_2$. Oba ukazatele mají stejnou počáteční hodnotu – 1. Přepis změny je pak čten a aplikován zleva doprava. Pokud je v něm dosaženo symbolu $I$, symbol $next_2$ je vložen před symbol $next_1$ v $S_1$ a ukazatel $next_2$ bude ukazovat na další symbol. Při obdržení symbolu $D$ v přepisu změny, symbol na pozici $next_1$ je odstraněn z $S_1$ a $next_1$ je zvýšen o jeden symbol. Při nalezení symbolu $R$ nebo $M$ je symbol na pozici $next_1$ v $S_1$ nahrazen znakem na $next_2$ z $S_2$ nebo ponechán původní. Oba ukazatele jsou pak zvýšeny o jednu pozici.

*Editační vzdálenost* mezi dvěma řetězci je definována jako nejmenší počet editačních změn (vložení, odstranění, náhrada) nutných k přeměně prvního řetězce na řetězec druhý.

Běžně používanými funkcemi výpočtu editační vzdálenosti jsou „Hammingova vzdálenost“, „Levenshteinova vzdálenost“ nebo „Epizodní vzdálenost“. Za zvláštní případ problému editační vzdálenosti můžeme považovat i problém LCS (popsán v 2.3) [15].

### 2.4.1 Hammingova vzdálenost

V roce 1950 představil pan Hamming (v publikaci [6]) metriku popisující odlišnost digitálních dat. Metrika sloužila původně pro detekce a opravy chyb v přenášených datech. Pomocí Hammingových kódů (založených na Hammingově vzdálenosti) je možno snadno zjišťovat a případně opravovat chybně přenesené bity. Hammingova vzdálenost se vypočítává jako počet všech odlišností ve dvou řetězcích.

Jako ukázka nám poslouží příklad 2.4.1.

**Příklad 2.4.1.** Podle [17]. Mějme dva řetězce:

- `toned`
- `roses`

Tyto řetězce se liší na třech místech (1., 3. a 5. znak), tedy celkem došlo ke třem změnám. Hammingova vzdálenost je proto 3.

### 2.4.2 Levenshteinova vzdálenost

Levenshteinova vzdálenost je jednou z nejběžněji používaných metrik vzdálenosti řetězců. Její definice je podle [20] tato:

Mějme abecedu $\Sigma$ a množinu všech řetězců $\Sigma^*$ nad $\Sigma$. $\lambda \notin \Sigma$ je řetězec nulové délky. Řetězec $X \in \Sigma$ je značen $X = x_1x_2 \dots x_n$, kde $x_i$ je $i$-tý symbol $X$. $X_{i\dots j}$ označíme jako podřetězec $X$ obsahující symboly $x_i$ až $x_j$ včetně těchto symbolů, $1 \leq i \leq j \leq n$, jeho délka je definována jako $|X_{i\dots j}| = j - i + 1$ a je řetězcem s nulovou délkou $\lambda(|\lambda| = 0)$, pokud $i > j$. Základní editační operace je pár $(a, b) \neq (\lambda, \lambda)$, často psáno $a \to b$, kde $a$ i $b$ jsou řetězce délky 0 nebo 1. Zápisy $\lambda \to a$, $a \to b$ a $b \to \lambda$ v tomto pořadí reprezentují *vložení*, *náhradu* a *odstranění*, což jsou tři základní typy editačních operací. $T_{X,Y} = T_1T_2 \dots T_l$ je značení používané pro transformaci $X$ na $Y$. Pokud váhová funkce $\lambda$ přiřazuje $a \to b$ nezáporné číslo $\lambda(a \to b)$, váha editační vzdálenosti $T_{X,Y}$ může být vypočítána použitím $\gamma(T_{X,Y}) = \sum_{i=1}^{l} \gamma(T_i)$. Při vstupech $X, Y \in \Sigma^*$ je „zobecněná Levenshteinova vzdálenost“ („GLD“) definována jako

$$GLD(X, Y) = min\{\gamma(T_{X,Y})\} \tag{2.2}$$

Ukazuje se, že GLD je metrika nad $\Sigma^*$, pokud jsou splněny podmínky:

$$\forall a, b \in \Sigma \cup \{\lambda\} \tag{2.3}$$

$$\gamma(a \to a) = 0 \tag{2.4}$$

$$\gamma(a \to b) > 0, a \neq b \tag{2.5}$$

$$\gamma(a \to b) = \gamma(b \to a) \tag{2.6}$$

Algoritmus 1 ukazuje výpočet GLD podle [14].

### 2.4.3 Použití pro porovnávání obrázků

V mnoha případech by bylo vhodné znát metodu porovnávání více-dimenzionálních dat založenou na algoritmu pro obecnou vzdálenostní metriku. Vhodnými oblastmi mohou být například databáze obrázků, kde je vhodné obrazová data uspořádávat podle podobnosti. Tento přístup může být výhodný i při rozpoznávání objektů nebo v podobných problémech spojených s počítačovým viděním. Může nabídnout základní měření a porovnávání medicínských struktur podle tvaru, např. porovnání tvaru pacientovy kosti s běžným tvarem a vyčíslení rozdílu [8].

Představení, popis a výsledky takové metody můžeme nalézt v článku [8]. Zde navržený algoritmus je optimalizován na porovnávání dvou-dimenzionálních tvarů za pomoci upraveného principu řešení editační vzdálenosti.

**Algoritmus 1**: Výpočet Levenshteinovy vzdálenosti řetězců $X$ a $Y$, podle [14]

```
1  begin
2      D[0,0,0] := 1;
3      D[0,0,1] := ∞;
4      for j := 1 to |Y| do
5          D[0,j,j−1] := ∞;
6          D[0,j,j] := D[0,j−1,j−1] + γ(λ → Y_j);
7          D[0,j,j+1] := ∞;
8      for i := 1 to |X| do
9          D[i,0,i−1] := ∞;
10         D[i,0,i] := D[i−1,0,i−1] + γ(X_i → λ);
11         D[i,0,i+1] := ∞;
12         for j := 1 to |Y| do
13             D[i,j,max(i,j)−1] := ∞;
14             for k := max(i,j) to i+j do
15                 D[i,j,k] := min(D[i−1,j,k−1] + γ(X_i →
                   λ), D[i,j−1,k−1] + γ(λ → Y_j), D[i−1,j−1,k−1] + γ(X_i → Y_j))
16             D[i,j,i+j+1] := ∞
17     d := ∞
18     for k := |X| to |X| + |Y| do d := min(d, D[|X|,|Y|,k]/k)
19     return d;
20 end
```

## 2.5 Metody registrace obrazů

V této sekci představujeme jednotlivé konkrétní metody registrace obrazů podle [21].

Jak již bylo nastíněno (2.1.4), metody registrace obrazů jsou široce používány zejména ve vzdáleném snímání, medicíně, počítačovém vidění atd. Metody registrace obrazů mohou být obecně rozděleny do čtyř základních oblastí podle typů dat, která zpracovávají:

- *Různé úhly pohledu.* Obrazy jedné scény jsou získány z různých míst. Scéna je tak v každém obrazu jinak natočená a vždy obsahuje jisté společné prvky i prvky, které na jiných obrazech být nemusejí. Cílem zpracování těchto dat má být zisk lepšího 2D pohledu nebo 3D reprezentace snímané scény.

- *Různé časy snímání.* Obrazy jedné scény jsou získány v různých časech, často s pravidelnými prodlevami s možností změny různých podmínek. Výsledkem má být nalezení a zhodnocení změn ve scéně, které vznikly během zisku jednotlivých obrazů.

- *Různé senzory.* Obrazy jedné scény jsou získány různými zařízeními. Snahou o registraci obrazů z více různých senzorů s různými nastaveními chceme dosáhnout více komplexní a detailní reprezentace scény.

- *Registrace scény na model.* Zde vycházíme ze snahy provést registraci (někdy též nazýváno „sesouhlasení“ nebo „sjednocení“) obrazových dat s modelem scény. Modelem může být počítačová reprezentace scény, např. mapa, digitální model vyvýšení („digital elevation model“) v GIS jiná scéna s podobným obsahem (např. jiný pacient se stejným typem nádoru) apod. Cílem je lokalizace obrazu v modelu a jejich srovnání.

Přestože vzhledem k odlišnosti představených oblastí není možné nalézt jedinou metodu, která by byla použitelná zároveň ve všech oblastech, rozeznáváme čtyři základní prvky, které většina registračních metod obsahuje:

- *Detekce význačných příznaků.* Výrazně ohraničené a zřetelné objekty (okraje, obrysy, křížení úseček, rohy atd.) jsou manuálně nebo v lepším případě automaticky rozpoznány. Pro další zpracování jsou tyto příznaky (v angl. literatuře „features“) reprezentovány svými bodovými charakteristikami (těžiště, zakončení úseček, význačnými body), v literatuře nazývanými „kontrolní body“ (angl. „control points“). Větší přehled o tomto kroku popisuje oddíl 2.5.1.

- *Párování význačných bodů.* V tomto kroku je ustaven vztah mezi příznaky detekovanými v pořízeném a referenčním obrazu. K tomuto účelu jsou použity metriky podobnosti mezi různými charakteristikami příznaků a jejich prostorových vztahů. Více o tomto kroku uvádíme v 2.5.2.

- *Ohodnocení transformací.* Typ a parametry tzv. mapovacích funkcí, zarovnávajících získaný snímek se snímkem referenčním, jsou vyhodnoceny. Parametry mapovacích funkcí jsou vypočítány vzhledem ke způsobu ustanovení vztahů mezi příznaky. Větší přehled o této fázi registrace obrazu poskytuje část 2.5.3.

- *Převzorkování obrazu a transformace.* Zachycený obraz scény je transformován vzhledem ke způsobům použití mapovacích funkcí. Neceločíselné souřadnice obrazů jsou vypočítány příslušnou interpolační technikou. Další informace uvádíme v 2.5.4.

Implementace každého z těchto základních kroků má své typické problémy. Nejprve musíme rozhodnout, jaký druh příznaků je příslušný k dané úloze. Rysy by měly být charakteristické objekty, které jsou pravidelně rozmístěny v obraze a jsou snadno detekovatelné. Obvykle je požadována jejich fyzická interpretovatelnost. Detekovaná množina příznaků v porovnávaném a referenčním obraze musí mít dostatek společných prvků i v situacích, kdy obrazy nepokrývají přesně tutéž scénu, obsahují průniky objektů nebo jiné nečekané změny. Detekční metody by měly mít dobrou přesnost lokalizace a neměly by být citlivé na chyby v obrazových datech, např. šumy. V ideálním případě by měl být algoritmus schopný detekovat stejné příznaky ve všech projekcích scény nezávisle na částečných deformacích obrazů.

Při párování nalezených příznaků se mohou projevit problémy způsobené nesprávnou detekcí příznaků nebo poškozením kvality obrazu. Fyzicky příslušející příznaky mohou být v obrazech různé kvůli rozdílným podmínkám pořízení obrazů a odlišné spektrální citlivosti senzorů. Výběr způsobů charakteristiky příznaků a metriky jejich podobnosti musí uvažovat tyto faktory. Charakteristiky by měly být neměnné vzhledem k obdrženým degradacím vstupů. Zároveň by však měly být dostatečně odlišitelné, aby bylo možno vyznačit rozdílné příznaky, a stabilní, aby výsledek nebyl ovlivňován menšími nečekanými změnami příznaků nebo rušením. Párovací algoritmus by měl být s množinou těchto invariantů robustní a efektivní. Ojedinělé příznaky bez příslušných příznaků v párovém obraze by neměly ovlivnit výkon algoritmu.

Mapovací funkce by měly být vybrány s ohledem na předem známé informace o očekávaných degradacích obrazů. Pokud nejsou tyto informace předem dostupné, model by měl být flexibilní a dostatečně obecný, aby mohl zpracovat všechny možné degradace, které mohou nastat. Přesnost metod detekujících příznaky, spolehlivost odhadu vzájemné příslušnosti

příznaků a akceptovatelná míra aproximačních chyb by měly být též zváženy. Navíc musí být učiněna rozhodnutí, které rozdíly mezi obrazy mají být odstraněny registrací. Pochopitelně je vhodné neodstranit takové rozdíly, které chceme hledat, pokud je cílem detekce rozdílů. Tento problém je velmi důležitý a řešení je velice složité.

Výběr požadovaného typu techniky převzorkování závisí na vyvážení mezi požadovanou přesností interpolace a komplexností výpočtu. Vzdálenostní metriky nebo bilineární transformace jsou ve většině případů dostatečné, avšak některé aplikace vyžadují více precizní metody.

### 2.5.1 Krok detekce příznaků

V dřívějších dobách byly za příznaky považovány objekty, které určil expert. Během automatizace detekce příznaků však byly zformovány dva převládající přístupy strojového rozpoznávání příznaků.

#### Metody vlícování ploch

Metody vlícování ploch (v anglické literatuře „area-based matching“) kladou důraz na párování příznaků před jejich detekcí. Nedetekují tedy žádné příznaky, takže je první krok registračního postupu vynechán.

#### Metody založené na příznacích

Tento přístup je založen na extrakci příznaků – význačných struktur v obrazech. Významné rozeznávané oblasti (např. les, jezero nebo pole), křivky (např. hraniční ohraničení objektů, pobřeží, cesty nebo řeky) a body (např. rohy oblastí nebo přerušení křivek) jsou zde považovány za příznaky. Měly by být zřetelné, rozmístěné po celém obrazu a efektivně detekovatelné v obou obrazech. Očekává se, že budou stabilní v čase, tedy na stejné pozici během celého experimentu.

Porovnatelnost množin příznaků v obou obrazech je zajištěna neměnností, přesností detekce příznaků a kritérii překryvu. Počet společných prvků množin by tedy měl být dostatečně velký, nezávisle na změnách rozměrů obrazů, podmínkách jejich pořízení, rušení a změnách ve zkoumané scéně. V porovnání s metodami vlícování ploch tyto metody nepracují přímo s hodnotami intenzity v obrazech. Rysy tak reprezentují informace na vyšších úrovních. Tato vlastnost činí metody založené na příznacích použitelnými v situacích, kde je očekávána změna osvětlení nebo snímky pořizujeme senzory s různou citlivostí na světelnost scény.

Souhrnně tedy můžeme konstatovat, že metody založené na detekci příznaků jsou vhodné, pokud obrazy obsahují dostatek zřetelných a snadno rozpoznatelných objektů. To je obvykle případ aplikací vzdáleného snímání nebo počítačového vidění. Typické snímky zde obsahují množství detailů, jakými jsou města, řeky, cesty nebo lesy. Naopak v medicínských oblastech, kde snímky nejsou tak bohaté na detaily, se častěji používají metody vlícování ploch. Někdy bývá nedostatek zřetelně odlišitelných objektů v medicínských snímcích řešen interaktivním výběrem nebo přidáním dalších vnějších příznaků s ohledem na pacienta (např. označení kůže). Můžeme nalézt i metody, které kombinují oba uvedené přístupy.

### 2.5.2 Krok párování příznaků

Detekované příznaky v referenčním a zkoumaném obrazu mohou být párovány různými způsoby, např. porovnáváním intenzity světla v jejich blízkém okolí, prostorovou distribucí příznaků nebo symbolickým popisem příznaků. Některé metody během hledání odpovídajících příznaků zároveň odhadují parametry mapovacích funkcí, a tak spojují druhý a třetí krok registrace obrazů.

Zde si uvedeme opět práci metod vlícování ploch a metod založených na příznacích.

#### Metody vlícování ploch

Metody vlícování ploch (někdy též „analýza podle šablon“, v anglické literatuře „template matching“) spojují krok detekce příznaků s krokem jejich párování. Tyto metody zpracovávají obraz bez snahy o detekci význačných objektů. Během tohoto druhého kroku jsou k odhadu souhlasících částí použita okna definované velikosti nebo dokonce celé obrazy.

Omezení vlícovacích metod mají původ v základní myšlence tohoto přístupu. Obdélníkové okno, které je nejčastěji použito, nachází sesouhlasení obrazů s lokálními odlišnostmi pouze překládáním obou obrazů přes sebe. Pokud je vstup deformován více komplexní transformací, tento typ okna není schopen pokrýt stejné části scény v obou obrazech.

Další nevýhoda je spojena s vypovídací hodnotou obsahu okna. Je vysoká pravděpodobnost, že okno neobsahující žádné výrazné detaily, bude nesprávně sesouhlaseno s jiným podobným místem v referenčním obraze právě kvůli nevýraznosti obou míst.

Klasické metody vlícování ploch, jako jsou vzájemná korelace (v anglické literatuře „cross-correlation“), jsou využívány k přímému sesouhlasení podle intenzity světla bez jakékoliv strukturní analýzy. V důsledku toho jsou citlivé na změny intenzit, vnesené např. šumem, měnícím se osvětlením scény nebo různými typy senzorů.

V literatuře (např. [21]) můžeme najít různé používané metody vlícování ploch. Mezi nimi můžeme zmínit metody *vzájemných korelací* („cross-correlation“), které jako souhlasící okna uvažují nejlepší korelaci mezi všemi možnými páry, *sekvenční detekce podobností* („sequential similarity detection“), kde je sekvenčně počítána suma absolutních rozdílů intenzity obrazu. Jako rychlé metody vlícování ploch mohou být v některých případech použity *Fourierovy metody*, které jsou vhodné též v proměnlivých podmínkách snímání. *Metody vzájemných informací* („Mutual information“) jsou využívány pro sesouhlasení multimodálních obrazových dat. To bývá obvykle složitý, avšak často nezbytný úkol, zejména v lékařství, kde některé diagnózy nelze získat jiným způsobem.

#### Metody založené na příznacích

Předpokládáme, že jsme obdrželi dvě množiny příznaků – ve zkoumaném snímku a ve snímku referenčním. Jako cíl párování příznaků si klademe ustanovení příslušnosti příznaků mezi jedním a druhým obrazem. Toho můžeme dosáhnout hledáním shodnosti charakteristik příznaků, např. jejich prostorových vztahů nebo popisu blízkého okolí.

Opět nacházejí využití různé metody. *Metody používající prostorové vztahy* („spatial relations“) jsou obvykle aplikovány, pokud detekované příznaky nejsou jednoznačné nebo pokud je jejich okolí lokálně deformováno. Jsou využívány informace o vzdálenostech mezi kontrolními body a jejich prostorové distribuci. *Metody používající popisy invariantů* („invariant descriptors“) jsou alternativou k metodám využívajícím prostorové vztahy. Odpovídající příznaky mohou být nalezeny pomocí svých popisů, nejlépe invariantních k očekávaným deformacím obrazu. *Relaxační metody* („relaxation methods“) jsou velkou skupinou

metod založených na relaxačním přístupu jako řešení optimalizace Markovských náhodných polí („Consistent labeling problem“) k označení příznaků referenčního obrazu tak, aby bylo konzistentní s označením jiného páru příznaků. *Pyramidové* a *vlnkové úpravy metod* se snaží redukovat výpočetní požadavky způsobené velikostí obrazu způsobem pyramidového přístupu. Tyto techniky je možné použít na běžné registrační metody tak, že začínáme s registrací obrazů v malém rozlišení a postupně vylepšujeme odhady příslušnosti příznaků při zvětšování rozlišení.

### 2.5.3 Krok ohodnocení transformací

Po ustanovení odpovídajících příznaků je zkonstruována mapovací funkce. Měla by transformovat obdržený snímek tak, aby mohl být přeložen přes snímek referenční. Přitom používáme nalezené odpovídající příznaky a předpoklad, že by tyto příznaky měly být v transformovaném obrazu nejblíže, jak je to možné. Úloha, kterou řešíme, sestává z výběru typu mapovací funkce a odhadu jejich parametrů.

Modely mapovacích funkcí mohou být rozděleny do dvou hlavních kategorií, vzhledem k množství obrazových dat, která používají. Globální modely používají kontrolní body pro odhad jedné množiny parametrů platných pro celý obraz. Naproti tomu lokální modely uvažují obraz jako sestavu menších částí a funkční parametry závisí na konkrétní lokalitě v obraze.

#### Globální mapovací funkce

Jeden z nejčastěji používaných globálních modelů – *podobnostní transformace* („similarity transform“) – počítá parametry pomocí dvojrozměrných polynomů nižších stupňů. Sestává z rotace, posunu a škálování. Na jeho jednoznačnou definici postačují dva různé páry příznaků. *Afinní transformace* („affine transformation“) je více obecným, avšak stále lineárním modelem. Může transformovat rovnoběžníky na obdélníky. Definuje se nejméně třemi páry příznaků neležících na přímce. Zachovává přímky a rovnoběžky. *Perspektivní projekční model* („perspective projection model“) je použit v podmínkách špatné vzdálenosti snímače od scény. Může mapovat obecný čtyřúhelník na obdélník při zachování přímek. Může být definován čtveřicí párů nezávislých příznaků.

#### Lokální mapovací funkce

Přestože globální mapovací funkce mohou pracovat rychle a může být výhodou, že berou obraz jako celek, nemohou správně postihnout jeho lokální deformace. To může nastat např. v lékařských obrazech a leteckých snímcích.

#### Mapování radiálními bázovými funkcemi

Metody mapování *radiálními bázovými funkcemi* („radial basis functions“) reprezentují skupinu globálních mapovacích metod, ale jsou schopny zpracovat dokonce i lokální proměnlivé pokřivení. Výsledná mapovací funkce má podobu lineární kombinace posunuté radiálně symetrické funkce a polynomu nižšího stupně. Z radiality vyplývá důležitá vlastnost, a sice to, že funkční hodnota každého bodu závisí jen na vzdálenosti bodu od kontrolního bodu, tedy není důležité jeho konkrétní umístění.

#### Elastická registrace

Jiným přístupem k registraci obrazů s velkými lokálními pokřiveními je nepoužití žádné parametrické mapovací funkce, přičemž je odhad geometrické deformace redukován na hledání „nejlepších“ parametrů. Obrazy jsou pokládány za části „gumového listu“, na které působí vnější síly, definované tuhostí nebo jemností tlaku, za účelem jejich zarovnání s minimálním množstvím kroucení a roztahování. Zde je párování a mapování příznaků prováděno zároveň. To je výhoda, protože charakteristiky invariantních příznaků v komplikovaných deformacích nejsou známy a párování příznaků tradičním způsobem je zde složité na provedení. Je zde též často používaný pyramidový přístup.

### 2.5.4 Krok převzorkování a transformace

Zde jsou použity mapovací funkce z předchozího kroku k transformaci zpracovávaného obrazu za účelem sesouhlasení s referenčním vzorem. Transformace může být provedena dopředným nebo zpětným způsobem.

Každý pixel pořízeného snímku může být přímo transformován za použití odhadnutých mapovacích funkcí. Tento přístup, nazývaný *dopředná metoda*, je komplikovaný na implementaci, jelikož může vytvářet díry a překryvy ve výstupním obraze (kvůli diskretizaci a zaokrouhlování). Proto se obvykle používá *zpětný přístup*. Sesouhlasená data obrazu z pořízeného snímku jsou určena použitím souřadnic výsledného pixelu (ve stejném souřadném systému jako referenční obraz) a inverzní funkcí k funkci mapovací. Interpolace je prováděna na pravidelné mřížce ve snímku. Tímto způsobem nemohou vznikat díry ani překryvy ve výsledném obrazu.

Interpolace samotná je provedena obvykle konvolucí obrazu s jádrem interpolace.

Sesouhlasování obrazů je jedna z nejdůležitějších úloh při integraci a analýze informací z různých zdrojů. Je klíčovým stupněm v detekci změn, ve spojování obrazů a v zisku různých informací o scéně, které by jinak jednotlivě nebylo možno získat. Přestože již bylo odvedeno mnoho práce a představeny různé pokročilé metody, registrace obrazů zůstává stále otevřeným problémem.

# Kapitola 3

# Možnosti porovnávání dat uživatelem

V této kapitole budou postupně představeny možnosti, které má uživatel na výběr, pokud potřebuje na počítači porovnávat data. Zvláštní pozornost věnujeme porovnávání obrazových dat, které je pro tuto práci stěžejní.

V části 3.1 je představen přehled běžně používaných aplikací, představení některých známých knihoven, které může využít vývojář, můžeme nalézt v části 3.2. Následně je v oddílu 3.3 více přiblížen balík programů a knihoven ImageMagick.

## 3.1 Existující aplikace

Uživatel, který potřebuje hotovou aplikaci pro porovnávání dat, může s ohledem na povahu porovnávaných souborů a své zkušenosti sáhnout k některé z aplikací, které si zde přiblížíme.

### 3.1.1 Program diff

Dnes již standardní součást distribucí systémů typu Unix, program `diff`, se typicky používá k nalezení rozdílů mezi jednou verzí souboru a jeho předchozí verzí. Program `diff` vypisuje všechny změny v řádcích textových souborů. Výstup, který program `diff` běžně poskytuje, může být použit programem `patch` pro vytvoření jedné verze souboru z verze druhé. Program je nyní součástí balíku Diffutils, stejně jako např. program `cmp`, který je představen dále.

Program byl vytvořen tak, aby efektivně využíval výkon a paměť počítače pro typické vstupy textových souborů, které jsou obsluhovány nebo vygenerovány počítačem. Časová a prostorová náročnost jsou koncipovány tak, aby závisely na délkách souborů. Program `diff` řeší „problém nejdelšího společného podřetězce“ („LCS“, blíže v 2.3 k nalezení řádků, které zůstaly v souborech nezměněny. Efektivita je dosažena soustředěním se pouze na „kritické“ shody v souborech, takové, jejichž porušení by vedlo ke zkrácení LCS na nějaký pár počátečních částí souborů. Program používá různé techniky k hashování, před-seřazování do tříd ekvivalence, spojování binárním vyhledáváním a dynamické alokování paměti k dosažení dobré výkonnosti [7].

### Způsob použití

Program `diff` je možné používat z příkazové řádky (nemá grafické uživatelské rozhraní) běžně spuštěním pomocí příkazu `diff [PŘEPÍNAČ]... SOUBORY`. Místo souboru lze použít i standardní vstup. Program provede porovnání a poskytuje různé formáty výstupu (příklady jsou inspirovány [16]):

**Normální výstup** Ve výchozím nastavení produkuje program `diff` výstup tak, že prvním řádkem změny je označení pozice (např. `8,14c14`), kde jedna resp. dvě číslice oddělené čárkou značí pozici resp. rozsah pozic v prvním souboru (zde 8. až 14. řádek), písmeno `a`, `c` nebo `d` značí operaci (`přidání`, „změna“, `odstranění`) a následuje označení pozice nebo rozsahu pozic v druhém souboru (zde 14. řádek). V dalších řádcích jsou vypsány samotné změny tak, že řádek, který byl ubrán z původního souboru, je značen `<` a přidaný řádek `>`. Změny v řádcích se vypisují jako odebrání starého a přidání nového řádku. Tento výstup můžeme ilustrovat na příkladu 3.1.1.

**Příklad 3.1.1.** Normální výstup programu `diff`

```
1c1
< smazaný řádek
---
> přidaný řádek
```

**Editační skript** Program `diff` může jako svůj výstup poskytnout i sekvenci příkazů a textu, která může být použita jako skript programem `ed` pro vytvoření nového souboru z původního. Takovéto chování je možné iniciovat přepínačem `-e`. Příklad výstupu můžeme vidět v 3.1.2.

**Příklad 3.1.2.** Výstup programu `diff` ve formátu „editační skript“

```
1c
přidaný řádek
.
```

**Kontextový formát** S přepínačem `-c` je možno dosáhnout výpisů, které obsahují změněné řádky stejně jako řádky původní v blízkosti změněných. Nezměněné řádky v okolí modifikovaných jsou nazývány „kontextem“. Pomocí kontextu lze vhodně lokalizovat odchylky a provádět tak spolehlivější aktualizace původních verzí na nové programem `patch`. Tento výstup je také lépe čitelný lidmi. Velikost kontextu je možno nastavit. Pokud se kontexty dvou změn překrývají, jsou sloučeny do jednoho. Ve výstupu před řádky se změnami nalezneme značky označující charakter změny: „!“ značí změněný řádek, „+“ je použito pro označení vložení řádku a „-“ je ve významu operace odebírání. Každý výstup začíná názvy souborů s časovými razítky a každý úsek změn s kontexty je uvozen rozsahem umístění, kde rozsah mezi hvězdičkami označuje umístění v původním souboru, zatímco rozsah mezi pomlčkami je použit pro označení pozice v souboru nové verze. Příkladem výstupu je 3.1.3.

**Příklad 3.1.3.** Kontextový formát výstupu programu `diff`

```
*** s1.txt 2011-04-17 18:47:56.168417002 +0200
```

```
--- s2 2011-04-17 18:47:45.328417002 +0200
***************
*** 1,3 ****
  předchozí nezměněný řádek, který bude použit jako kontext
! původní řádek
  následující nezměněný řádek, který bude opět použit jako kontext
--- 1,3 ----
  předchozí nezměněný řádek, který bude použit jako kontext
! změněný řádek
  následující nezměněný řádek, který bude opět použit jako kontext
```

**Unifikovaný formát výstupu** Tento formát vychází z kontextového formátu, ale redukuje jej tak, aby nedocházelo ke zbytečné redundanci vypisováním kontextu. Oproti kontextovému formátu jsou v něm proto změny vypisovány ihned pod sebou a následně jsou „obaleny“ kontextem. Tento formát se velmi často používá jako formát pro výměnu změn ve zdrojových kódech. Hlavičky kontextů zde mají tvar `@@ -[rozsah1] +[rozsah2] @@`, kde `rozsah1` lokalizuje změnu v původním a `rozsah2` v novém souboru. Tento formát označuje pouze operace vložení („+“) a odebrání („-“) řádku. Změna řádku je zde reprezentována odebráním původního a přidáním nového řádku, jak můžeme vidět na příkladě 3.1.4.

**Příklad 3.1.4.** Normální výstup programu `diff`

```
@@ -1,3 +1,3 @@
předchozí nezměněný řádek, který bude použit jako kontext
-původní řádek
+změněný řádek
následující nezměněný řádek, který bude opět použit jako kontext
```

### 3.1.2 Program cmp

Další součástí balíku Diffutils je program `cmp`. Jedná se o velmi jednoduchý, avšak např. ve skriptech často používaný program, který pracuje na principu prostého porovnávání bajt po bajtu (představeno v 2.2). Program pracuje v příkazové řádce a v základním nastavení nabízí uživateli možnost zjistit, zda mají dva soubory různé obsahy. Navíc vypisuje pozici prvního bajtu, na které dochází k odlišnosti. Program `cmp` můžeme spustit příkazem `cmp [PŘEPÍNAČ]... SOUBOR1 [SOUBOR2 [SKIP1 [SKIP2]]]`, kde můžeme místo souboru využít standardní vstup. Dále je možno pomocí přepínače `-i` deklarovat pozice nebo rozsah bajtů, jejichž změny má `cmp` ignorovat. Příkladem výstupu programu (počeštěná verze programu) je 3.1.5.

**Příklad 3.1.5.** Normální výstup programu `cmp`

```
soubor1 soubor2 se liší: bajt 43, řádek 2
```

Je také možno dosáhnout výpisu všech změn mezi soubory. K tomuto účelu je použit přepínač `-l`. Program v tomto režimu vypisuje všechny rozdílné bajty ve formátu `pozice bajt1 bajt2`, kde `bajt1` je oktalový zápis hodnoty rozdílného bajtu v prvním souboru a `bajt2` reprezentuje novou hodnotu na téže pozici v druhém souboru. Výstup pak může vypadat podobně jako v příkladu 3.1.6.

**Příklad 3.1.6.** Výstup programu `cmp` s přepínačem `-l`

```
26 172 132
27 155 115
28 145 105
```

Pokud před nalezením neshodných bajtů (nebo vždy při porovnávání všech změn přepínačem `-l`) nastane konec některého souboru, program `cmp` vypisuje `EOF` a jméno souboru, ve kterém nastal nečekaný konec.

### 3.1.3 Program compare

Balík open source programů a knihoven ImageMagick nabízí uživateli široké spektrum možností vytváření, upravování, skládání nebo konverzi bitmapových obrázků. V jeho možnostech je číst a zapisovat velké množství (přes 100) formátů. S obrazovými daty lze provádět různé operace, jako změny velikosti, překlápění, zrcadlení, rotace, pokřivování, ořezy a transformace, změny barev, speciální efekty, nebo vykreslování geometrických objektů včetně textů, čar, polygonů, elips a Bézierových křivek [10].

Balík ImageMagick a jeho možnosti budou blíže představeny v 3.3.

Koncovému uživateli je nabídnut soubor programů příkazové řádky s ovládáním pomocí parametrů spouštěného programu, zatímco vývojáři mohou použít aplikační rozhraní ImageMagicku. To je původně určeno pro jazyk C, avšak pomocí vazeb na tyto knihovny může být používáno z různých programovacích jazyků, jakými jsou C++, Java, Lisp, .NET, PHP, Python verze 2 a další.

Jedním z programů balíku ImageMagick je program `compare`. Je přímo určen na porovnávání dvou obrázků způsobem pixel po pixelu, který byl představen v 2.2. Příloha této práce obsahuje část zdrojového kódu aplikace, který názorně ukazuje způsob jeho práce, tedy porovnávání po pixelech. Do výstupního souboru označuje rozdíly tak, že každý rozdílný pixel zvýrazní. Způsob zvýraznění a barvu je možno nastavit, stejně jako práh pro ignorování malých rozdílů. Uživatel tak může snadno vidět, kde se nachází rozdíly dvou obrázků. Běžný výstup programu `compare` z dvou vstupních obrázků (obr. 3.1 a 3.2) můžeme vidět na obrázku 3.3[1]. Program je též možno použít pro hledání podobrazů. Tato schopnost však není optimalizována, a proto je vyhledávání uskutečňováno jako porovnání v každé možné pozici způsobem pixel po pixelu. Je tak zřejmé, že náročnost roste kvadraticky vzhledem k velikosti vstupních obrázků.

### 3.1.4 Matematické programy

Obrazová data je možno chápat také jako matice barevných bodů. Je proto možné provádět různé operace nad obrázky pomocí matematických programů, jakými jsou Matlab, Octave nebo Maple. Tyto programy mívají obvykle dobře optimalizované výpočty operací nad maticemi. Funkce přímo pro porovnávání obrázků však neobsahují a je tak nutné podle potřeby vytvořit vlastní. Matematické programy poskytují vhodné prostředky pro snadnou implementaci algoritmů, je tak možno relativně pohodlně implementovat způsoby porovnávání představené v kapitole 2.

[1] Původní obrázek převzat z http://www.python.org/community/logos/

Obrázek 3.1: Původní obrázek

Obrázek 3.2: Změněný obrázek

### 3.1.5 Perceptual Image Diff

Zajímavým open source nástrojem pro porovnávání obrázků je program Perceptual Image Diff. Aplikace se pokouší výpočetně vytvářet model lidského vidění a pomocí něj obrázky porovnávat. Její snahou je poskytnout informaci o tom, zda může člověk v obrázcích nalézt rozdíl. Projekt je však rozpracován a již delší dobu na něm práce neprobíhají.

### 3.1.6 Další programy

Na porovnávání souborů se specializují i další programy. Vesměs jde o komerční aplikace s uzavřeným zdrojovým kódem. Seznam nejznámějších programů pro porovnávání souborů a jejich vlastnosti můžeme nalézt na http://en.wikipedia.org/wiki/Comparison_of_file_comparison_tools.

## 3.2 Využitelné knihovny

Pokud by chtěl vývojář použít existující knihovnu k implementaci operací s obrázky, má na výběr z různých knihoven podle jazyka, ve kterém se rozhodne pracovat. Zde si představíme některé známé využitelné knihovny vesměs pro použití v různých programovacích jazycích.

Obrázek 3.3: Nalezené změny (program compare)

### 3.2.1 ImageMagick

Balík ImageMagick, který má počátek v roce 1999, je kolekce výkonných nástrojů příkazové řádky a knihoven pro čtení, zápis a manipulaci s obrazovými daty ve více než 100 formátech. Z uživatelského hlediska je výhodné používat jeho aplikace z příkazové řádky, zejména pokud jsou potřeba hromadné operace, například použití ve skriptech. Při použití balíku ImageMagick můžeme velmi snadno měnit rozměry obrázků, rotovat a překlápět je, vkládat texty, upravovat ostrost nebo barvy a mnohé další operace [19].

Aplikacemi příkazové řádky balíku ImageMagick jsou program `animate`, s jehož pomocí lze vytvářet animace, tedy posloupnosti souvisejících obrázků promítaných za sebou, program `compare`, s jehož využitím můžeme porovnávat rozdíly mezi dvěma obrázky (blíže je popsán v 3.1.3), `compozite`, kterým můžeme různými způsoby překrývat dva obrázky přes sebe. Pro spouštění skriptů ve speciálním jazyku Magick Scripting Language můžeme využít program `conjure`, programem `convert` lze převádět formáty obrázků, měnit velikosti, provádět změny barev, kreslení, překlápění a mnohé další operace. S využitím aplikace `display` lze zobrazovat předložené obrázky pomocí X serveru. Program `identify` můžeme použít, pokud potřebujeme informace o uloženém obrázku a jeho formátu, `import` zajišťuje zachycení obrazu z X serveru, například snímky oken na pracovní ploše. Aplikace `mogrify` pracuje podobně jako program `convert`, avšak mění přímo původní obrázek, nevytváří tedy zbytečně nový, pokud to není potřeba. Pomocí programu `montage` můžeme skládat více obrázků na sebe nebo vedle sebe a vytvářet tak například kolekce náhledů. Aplikaci `stream` můžeme využít, pokud potřebujeme přenášet obrazová data textovým způsobem [10].

Pokud bychom chtěli vytvářet vlastní aplikace zpracovávající obrázky, můžeme použít programové rozhraní knihoven ImageMagick. Jádro programů ImageMagicku je implementováno v jazyku C, a protože má ImageMagick otevřený zdrojový kód, můžeme používat jeho knihovny přímo ve svých programech v tomto jazyce. ImageMagick však nabízí také rozhraní pro různé další známé programovací jazyky, jakými jsou například C++, Java, Lisp, .NET, Pascal, Perl, Php, Python verze 2, Ruby a další. V našich programech tak můžeme snadno implementovat stejné funkce, jaké mají představené aplikace příkazové řádky.

### 3.2.2 OpenCV

Knihovny OpenCV jsou kolekcí nástrojů pro práci s obrázky s otevřeným zdrojovým kódem. Zaměřují se na zpracování obrazových dat metodami počítačového vidění v reálném čase.

Jsou často nasazovány v prostředích, kde je nutné rychle zpracovat data a reagovat na ně. Uplatnění nachází nejčastěji v oblasti bezpečnostního monitorování, kombinování mapových podkladů a robotiky. Knihovny OpenCV obsahují více než dva tisíce optimalizovaných algoritmů a je možné používat je z programovacích jazyků C, C++ a Python verze 2 [11].

Jedním z cílů OpenCV je nabídnout snadno použitelnou sadu nástrojů, která uživateli pomůže snadno vybudovat sofistikovanou aplikaci pro zpracování počítačového vidění. OpenCV knihovna obsahuje více než 500 funkcí, které se zaměřují na mnoho oblastí vidění, jakými jsou inspekce průmyslových výrobků, medicína, bezpečnost, uživatelské rozhraní, kalibrace fotoaparátů, prostorové vidění nebo robotika. OpenCV též obsahuje knihovnu Machine Learning Library, která poskytuje možnosti obecného strojového učení, které se s výhodami používá při zpracovávání obrazových dat [4].

### 3.2.3 FreeImage

FreeImage je open source projekt spravující knihovnu pro vývojáře, kteří chtějí mít ve svých programech podporu pro oblíbené grafické formáty, jakými jsou PNG, BMP, JPEG, TIFF a další, které jsou běžnými součástmi multimediálního obsahu. FreeImage je snadno použitelný a multiplatformní.

FreeImage nabízí knihovny pro využití z mnoha běžně používaných programovacích a skriptovacích jazyků, jakými jsou C, C++, VB, C#, Delphi, Java, Perl, Python, PHP a další.

Mezi výhody FreeImage patří jednoduchost použití, možnost využití různých doplňků nebo snadná integrace do OpenGL a DirectX. S využitím knihoven FreeImage můžeme snadno provádět běžné operace jako změnu barevné palety obrázků, rotace, změna velikosti či překlápění. Je možné také na nižší úrovni přímo přistupovat k jednotlivým pixelům [9].

### 3.2.4 Vips

Vips je další multiplatformní open source balík programu a knihoven používaných k práci s obrázky. Je optimalizován pro operace s velice objemnými obrázky (typicky většími než kapacita paměti RAM). Dokáže efektivně rozložit práci mezi více procesorů. Používá se běžně pro práci s barvami a vědecké analýzy na velkých obrazových datech, jakými jsou například satelitní snímky Země. V porovnání s jinými knihovnami pracujícími s obrázky Vips používá málo paměti a operace provádí rychle, zvláště na strojích s více procesory.

Vips sestává ze dvou částí – knihovna *libvips* a grafické uživatelské rozhraní *nip2*.

Knihovny *libvips* lze používat z programovacích jazyků C, C++ a Python. Jeho výhodou je jednoduchost použití optimalizovaných vysokoúrovňových operací.

Grafické uživatelské rozhraní *nip2* se snaží kombinovat výhody tabulkového procesoru a nástroje pro přímou editaci obrázků. Uživatel obrázky neupravuje přímo, ale prací s tabulkovými hodnotami vytváří a mění vztahy mezi objekty v obrázku [12].

## 3.3 Možnosti použití balíku ImageMagick

Jak již bylo nastíněno v oddílu 3.2.1, možností balíku ImageMagick můžeme využít jak z vlastního programu použitím programového rozhraní ImageMagicku, tak použitím aplikací příkazové řádky, které ImageMagick nabízí, například ze skriptů. V této sekci budou blíže představeny přístupy, jak můžeme možností ImageMagicku v praxi využít pro tvorbu našich programů.

### 3.3.1 Programové rozhraní

Zde si představíme použití programového rozhraní ImageMagicku podle [2].

Programové rozhraní knihoven ImageMagick je přizpůsobeno zvyklostem každého jazyka, pro který jsou knihovny poskytovány. Způsob využití poskytovaných funkcí zvolíme v závislosti na tom, jaký programovací jazyk budeme používat. V objektově orientovaných jazycích poskytují programová rozhraní ImageMagicku pro operace s obrázky různé objekty jako Image, Canvas, Geometry, Color nebo Pixel.

Využití v programu si ukážeme na příkladu 3.3.1 v jazyce C++:

**Příklad 3.3.1.** Práce s obrázkem v jazyce C++ pomocí programového rozhraní ImageMagicku.

```
Image source_image("puvodni.png");  // načtení obrázku ze souboru
Image sub_image(source_image);      // vytvoření kopie obrázku

sub_image.chop(Geometry(100,100));  // ořez 100px zleva a 100px shora
sub_image.crop(Geometry(200,200));  // ořez 200px zprava a 200px zdola
sub_image.rotate(45);               // rotace o 45 stupňů doprava

sub_image.write("zmeneny.png");     // uložení změněného obrázku do souboru
```

Jak vidíme, je zde možno intuitivně provádět operace s obrázky na poměrně vysoké úrovni. I v ostatních programovacích a skriptovacích jazycích se programové rozhraní ImageMagicku používá podobně.

### 3.3.2 Aplikace příkazové řádky

Způsob použití aplikací příkazové řádky, které ImageMagick nabízí, si zde blíže představíme podle [10].

Aplikace balíku ImageMagick se používají z příkazové řádky. Operace, které na obrázcích provádějí, specifikujeme pomocí parametrů, se kterými programy spouštíme. Výhodou tohoto přístupu je možnost použití nástrojů bez potřeby programovat operace za použití nějakého programovacího jazyka. Lze je také snadno využít ve skriptech pro případ, že chceme operace provádět hromadně a automatizovaně.

Tento způsob používání si můžeme ilustrovat na následujících příkladech:

Provést konverzi obrázku z formátu JPG do formátu PNG můžeme jako na příkladu 3.3.2.

**Příklad 3.3.2.** Konverze z formátu JPG do formátu PNG pomocí nástroje `convert`

```
convert rose.jpg rose.png
```

Příklad příkazu provádějícího konverzi obrázku z formátu JPG do formátu PNG a zmenšení rozměrů výsledného obrázku na polovinu můžeme nalézt v 3.3.3.

**Příklad 3.3.3.** Konverze z formátu JPG do formátu PNG a zmenšení na polovinu pomocí nástroje `convert`

```
convert rose.jpg -resize 50% rose.png
```

Příkazem 3.3.4 postupně vytvoříme obrázek o rozměru $320 \times 85$ px, vybereme font a jeho velikost, na pozici $(25, 60)$ vysázíme text „Magick“, který rozostříme, poté vybereme

temně červenou jako barvu výplně a purpurovou jako barvu obrysu a takto vysázíme text „Magick“ na pozici $(20, 55)$. Vznikne tak obrázek s temně červeným nápisem „Magick“ a jeho rozostřeným stínem.

**Příklad 3.3.4.** Tvorba obrázku s textem a jeho stínem pomocí nástroje `convert`

```
convert -size 320x85 canvas:none -font Bookman-DemiItalic -pointsize 72 \
-draw "text 25,60 'Magick'" -channel RGBA -blur 0x6 -fill darkred -stroke \
magenta -draw "text 20,55 'Magick'" fuzzy-magick.png
```

Vykreslit obrázek `vrchni.gif` na střed obrázku `spodni.gif` můžeme jako v příkladě 3.3.5.

**Příklad 3.3.5.** Vykreslení obrázků přes sebe pomocí nástroje `composite`

```
composite -gravity center vrchni.gif spodni.gif vysledek.gif
```

Příkazem 3.3.6 porovnáme obrázek `puvodni.jpg` s obrázkem `zmeneny.jpg` způsobem pixel po pixelu (představeno v 2.2). Změněné pixely budou zaznačeny do nového obrázku `rozdil.jpg` červenou barvou.

**Příklad 3.3.6.** Porovnání dvou obrázků pomocí nástroje `compare`

```
compare puvodni.jpg zmeneny.jpg rozdil.jpg
```

Porovnat obrázek `puvodni.jpg` s obrázkem `zmeneny.jpg` a přitom ignorovat rozdíly menší než 5 %, což je vhodné např. pro porovnávání obrázků se ztrátovou kompresí (např. ve formátu JPG), můžeme podobně jako v příkladu 3.3.7.

**Příklad 3.3.7.** Porovnání dvou obrázků se zanedbáním rozdílů menších než 5 % pomocí nástroje `compare`

```
compare -fuzz 5% puvodni.jpg zmeneny.jpg rozdil.jpg
```

Některé konstrukce parametrů příkazové řádky nemusí být tak intuitivní, jako je tomu v případě použití programových rozhraní.Na druhou stranu však poskytují snadný přístup k operacím s obrázky bez nutnosti překladu a spouštění vlastního programu.

## 3.4 Zhodnocení současného stavu

Zde budou představeny konkrétní důvody, proč vznikla tato práce. Nastíníme, jaké nedostatky má současný stav problému, který byl popsán v předchozích částech, a uvedeme, jak by bylo možno stav zlepšit.

### 3.4.1 Porovnávání dokumentů

Doposud vyvíjené aplikace pro porovnávání dokumentů, které byly představeny v 3.1, se vždy zaměřují na určitý typ dat.

Mezi nimi můžeme najít užitečné a pokročilé nástroje pro porovnávání textových souborů (zejména aplikaci *diff*). Porovnávání textových souborů je zřejmě nejčastější prováděnou porovnávací operací, které uživatelé využívají. Program *diff* je vyvíjen dlouhou dobu a ve svém oboru je velmi využívanou aplikací. Z toho důvodu je velmi dobře vyladěn a optimalizován. Uživatelům poskytuje množství možností zpracování vstupů a formátování výstupu. Pracuje na dobře popsaném matematickém modelu (řešení problému „LCS“, více

v 2.3). Proto bychom za něj obtížně nalezli nebo sami implementovali lepší náhradu se stejnou nebo podobnou funkčností.

V oblasti porovnávání obrazových dat můžeme zmínit nástroj *compare* z balíku ImageMagick. Aplikace je přímo určena pro porovnání dvou podobných obrázků (zejména o stejných rozměrech), které se od sebe částečně liší. Uživatelům poskytuje srozumitelné označení provedených změn s různými možnostmi nastavení výstupů. Ačkoliv k tomuto účelu není přímo určena, je možné využít také její funkci pro hledání podobrazu, tedy lokalizaci obrázku s menším rozměrem v obrázku s rozměrem větším. Tato funkčnost však není optimalizována, a je proto vhodné využívat jí pouze pro vstupní obrázky s malými rozměry. Také tato aplikace je dlouhou dobu vyvíjena a ve svém oboru často používána. Lze ocenit zejména její široké možnosti nastavení a uživatelskou přívětivost, tedy snahu o pochopitelnost výstupů již v základním nastavení.

Kromě textových souborů a obrázků můžeme porovnávat také další data. Mezi nimi můžeme zmínit např. binární data, jak bylo přiblíženo v 3.1.2 nebo soubory ve standardních formátech *XML*, *PDF* nebo soubory se zvukovým záznamem. Nástroje k porovnávání těchto dokumentů nebyly z důvodu rozsahu blíže popsány, jelikož se této práce přímo netýkají. Můžeme však krátce zmínit nástroj *diffxml*[2], který na rozdíl od standardního *diffu* prochází strukturu dokumentů nikoliv po řádcích, ale stromovým způsobem, a poskytuje výpis rozdílů i aplikování změn na soubory původní (podobně jako program *patch*). Podobným nástrojem, který poskytuje i trojcestné prohledávání, je *3DM*[3]. Pokud budeme chtít zjistit rozdíly mezi soubory ve formátu *PDF*, můžeme využít open source program *DiffPDF*[4], který uživateli poskytuje přívětivé zobrazování rozdílů stylem „vedle sebe“ s barevným vyznačením změn. K porovnávání zvukových záznamů bude sloužit plug-in *Proposal Audio Diff*[5] ve známém programu Audacity (momentálně ve fázi návrhu).

### 3.4.2 Porovnávání různých typů dokumentů jednou aplikací

Jak jsme si tedy ukázali, aplikací, které provádějí porovnávání různých typů dokumentů, je více a specializují se na odlišná data. Mnohé z nich jsou dlouhou dobu každodenně využívány mnoha uživateli, a proto lze předpokládat, že jejich kvalita je vysoká.

Zatím se však běžně nepoužívá žádná aplikace, která by prováděla porovnávání více různých typů dat. Takový nástroj by byl velmi vhodný např. pokud provádíme spojování dvou verzí většího projektu. Takový projekt může obsahovat právě různá data, například zdrojové texty programů, xml dokumenty s daty a obrázky, které program používá. V tomto případě, pokud na projektu pracuje více vývojářů, je vhodné využít nástroj, který automaticky provádí spojování změn přijatých od vývojářů a pomáhá tak udržovat aktuální verze dokumentů tak, jak je vývojáři upravují.

Tento nástroj by pak nemusel obsahovat samotnou implementaci porovnávání, ale mohl by sloužit jako zastřešení stávajících aplikací. Pro uživatele by bylo výhodné využít stejného programu a jeho rozhraní, tedy stejných přepínačů a dalších možností nastavení, pro libovolné typy porovnávaných dokumentů. Rozhodování o konkrétním způsobu porovnání je pak možno provádět automaticky na základě obdržených vstupních souborů.

Cílem této práce je tvorba právě takové aplikace.

[2] http://diffxml.sourceforge.net/
[3] http://www.cs.hut.fi/~ctl/3dm/
[4] http://www.qtrac.eu/diffpdf.html
[5] http://wiki.audacityteam.org/wiki/Proposal_Audio_Diff

# Kapitola 4

# Vývoj aplikace MediaDiff

Tato kapitola se zabývá procesem tvorby aplikace MediaDiff. Tato aplikace vychází z požadavků na nástroj, které byly uvedeny v sekci 3.4.2. Aplikace se pokouší tyto požadavky splnit.

Pro tvorbu aplikace byl zvolen vodopádový model životního cyklu software. Tento model je jednoduchý na zavedení a jeho výhodou je i přímočarost a jasné vymezení vstupů a výstupů. Je vhodný zejména pro menší softwarové projekty. Tato kapitola popíše jednotlivé etapy vývoje, jak následovaly za sebou podle vodopádového modelu, a uvede vzniklé problémy a jejich řešení.

Vodopádový model je lineárním modelem tvorby software, sestávajícím z pěti etap, které na sebe bezprostředně navazují. Každá etapa využívá výstupů etapy předchozí a začíná po jejím ukončení. Vývojář má přitom možnost vrátit se k předchozí etapě při zjištění nedostatků v jejich výstupech. Chyby, které v etapě vznikly, by tak měly být objeveny nejpozději při následující etapě. Je tak možno předejít větším ztrátám oproti případu, kdy bychom postupovali méně systematickým přístupem, například od definice požadavků přímo k implementaci. Pokud totiž výstupy definice požadavků obsahují chyby, ve vodopádovém modelu je zjistíme, pokud se pokusíme software navrhnout, a můžeme se vrátit zpět. Pokud tyto chyby nalezneme až při vlastní implementaci, bude oprava výrazně nákladnější, protože může být například nutno celé části kódu vypustit.

V jednotlivých sekcích této kapitoly tak budou uvedeny etapy *definice požadavků* (4.1), *návrhu aplikace* (4.2), *implementace* (4.3), *testování* (4.4) a možnostmi dalšího vývoje při *nasazení a údržbě aplikace* (4.5).

## 4.1 Definice požadavků

Jak již bylo nastíněno (3.4.2), prototyp aplikace, který je výsledkem této práce, je určen k porovnávání různých typů dokumentů. Jde o modulární aplikaci, tedy aplikaci zastřešující různé moduly, které jsou určeny pro specifické typy dokumentů. Moduly mohou využívat některých stávajících implementací porovnávání dokumentů nebo implementovat vlastní funkčnost. Aplikace je připravena pro použití i s dalšími moduly, které mohou být případně přidány. Aplikace má jedno rozhraní, tedy způsob ovládání uživatelem je shodný pro všechny případy dokumentů, které porovnává. Dále je však možno jednotlivým modulům přidávat i vlastní možnosti nastavení podle porovnávaných dokumentů. Nastavení činnosti aplikace je ovládáno z příkazové řádky pomocí parametrů při jejím spouštění. Je tak snadno použitelná v různých uživatelských skriptech. Po dohodě s vedoucím práce a dalšími členy

týmu není úplná implementace hlavní aplikace součástí této práce. Proto v jejím rámci bude implementován pouze její prototyp, aby bylo možno prověřovat chování implementace modulu.

Součástí implementace prototypu aplikace je jeden konkrétní modul – modul pro provádějící porovnávání statických obrázků. Práce modulu bude spočívat v porovnávání obrazových dat takovým způsobem, který je pochopitelný koncovému uživateli. Při obdržení obou vstupních obrázků tedy bude výstupem obrázek, kde budou odlišnosti výrazně označeny. Bude též možno uvažovat různé rotace obrázků. Informace o nich bude vypisována na standardní chybový výstup. Při výstupu změn na standardní výstup bude využit snadno čitelný textový formát. Tento modul bude také možno samostatně spustit, a tak bude vykonávat stejnou činnost jako v případě spuštění přes hlavní aplikaci.

Dále je v rámci této práce připraven modul `utils`, který poskytuje hlavnímu programu a modulům důležité informace související se zpracováním parametrů příkazové řádky a konfiguračních souborů.

## 4.2 Návrh

Nyní tedy máme definovány požadavky, víme, jaká by měla výsledná aplikace být. Dalším krokem tvorby software je podle zvoleného vodopádového modelu návrh. Nyní tedy navrhneme aplikaci s ohledem na budoucí implementaci. Implementační detaily však nyní postihovat nebudeme, ty budou řešeny v kroku následujícím.

### 4.2.1 Volba programovacího jazyka, knihoven a platformy

Zásadním výsledkem návrhu, který ovlivní celý budoucí vývoj aplikace, je mimo jiné volba implementačního jazyka. Po zvážení možností přicházejících v úvahu a dohodě s vedoucím práce byl pro implementaci aplikace zvolen programovací jazyk Python ve verzi 3.

Jazyk Python je dynamický objektově orientovaný skriptovací jazyk. Jeho verze 3, která byla představena v roce 2008, se v některých ohledech liší od předchozí verze 2. Není proto možné jednoduše zajistit vzájemnou kompatibilitu programů využívajících jedné z verzí ve verzi druhé. Z důvodů konzervativnosti a závislosti stávajících produktů na starší verzi 2 je Python verze 3 používán stále v menšině aplikací, přestože vývojáři Pythonu se snaží prosazovat novější verzi kvůli její větší systematičnosti. Podle vyjádření vývojářů[1] je Python verze 2.7 poslední významnou verzí druhé řady jazyka Python. Dále již budou poskytovány pouze opravy chyb a vývoj se bude soustředit na verzi 3.

Pro práce s obrazovými daty bude využito možností balíku ImageMagick (představeno v 3.3). Samotný způsob porovnávání obrazových dat programem `compare` je zde vhodný, pokud se chceme přiblížit způsobu činnosti programu `diff`. Aplikace by měla být provozovatelná v operačním systému Linux.

### 4.2.2 Modularita

Jazyk Python přímo poskytuje prostředky pro zajištění modularity aplikací. Moduly jsou zde reprezentovány běžnými soubory s definicemi jazyka Python (obvykle s příponou .py). Moduly mohou být hierarchicky uspořádány, a jsou proto obvykle umísťovány v hierarchické struktuře souborového systému tak, že jednotlivé adresáře obsahují související soubory. Mohou být zanořeny i více. Máme-li např. modul `sound`, ten může obsahovat moduly

[1] http://docs.python.org/py3k/tutorial/modules.html

`formats`, `effects` a `filters`. Tyto moduly dále mohou obsahovat další moduly, např. jednotlivé soubory s definicemi. Jimi mohou být např. `wavread.py`, `wavwrite.py` nebo `equalizer.py` atp. Hierarchické uspořádání v systému souborů pak může být například následující: `sound/formats/wavread.py` nebo `sound/filters/equalizer.py`. Načtení modulu a jeho přípravu k použití v jazyce Python zajistíme použitím klíčového slova `import`, např. `import sound.filters.equalizer`. Poté můžeme používat funkce, třídy a proměnné definované v souboru `equalizer.py`. Je však nutno odkazovat se na ně celou cestou, např. tedy `sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)`. Pokud bychom chtěli tuto funkci používat snáze, můžeme ji přiřadit do nějaké proměnné, např. `echofilter = sound.effects.echo.echofilter()`. Každý modul musí také obsahovat soubor `__init__.py`, ve kterém je umístěna inicializace modulu. Většinou však zůstává prázdný. Je nutný k rozlišení, zda jde o modul, který se má interpret jazyka Python pokusit importovat [1].

### 4.2.3 Problémy

Při bližším pohledu na navržené prostředky vyvstala otázka využitelnosti knihoven balíku ImageMagick. Přestože balík poskytuje rozhraní pro použití knihoven z jazyka Python, tyto knihovny jsou k dispozici pouze pro Python verze 2 a podle vyjádření[2] vývojářů balíku ImageMagick v dohledné době nebudou poskytnuty pro verzi 3. Proto bylo po dohodě s vedoucím práce přistoupeno k náhradnímu řešení – použití spustitelných programů balíku ImageMagick formou systémového volání pro spouštění podprocesů (využití třídy `Popen`).

## 4.3 Implementace

Po návrhu aplikace již můžeme přistoupit k vlastní implementaci podle něj. Návrh je vhodně dekomponován, tedy rozčleněn na menší podproblémy, jak nyní uvedeme a konkrétní implementaci těchto podproblémů přiblížíme.

Základní rozčlenění na prototyp hlavní aplikace a jejího modulu pro porovnávání obrázků se samo nabízí. I tyto podproblémy jsou dále členěny.

Aplikace sestává ze souboru `mdiff`, který se používá ke spuštění, soubory potřebné pro běh hlavní aplikace jsou umístěny v adresáři `mediadiff` a jednotlivé moduly jsou umístěny v adresáři `modules`.

### 4.3.1 Prototyp hlavní aplikace

Hlavní adresář obsahuje modul `utils.py`, kde nalezneme třídu `properties`, využitelnou pro zpracování konfiguračních souborů. K tomuto účelu používá modul `configparser` běžně poskytovaný se základní instalací interpretu jazyka Python. S jeho pomocí je možno zpracovávat konfigurační soubory v běžně užívaném formátu INI. Aplikaci a modulům poskytuje informace o výchozím nastavení, možnosti zpracovávaných typů souborů a priority použití modulů pro konkrétní soubory. Typy souborů jsou zde charakterizovány svými příponami. Tato třída je také implementována tak, aby bylo možno používat ji za stejným účelem i z konkrétních modulů při jejich samostatném spuštění.

Prototyp jádra aplikace samotné je umístěn v souboru `main.py`. Provádí načtení modulů (prototyp načítá pouze modul pro porovnávání statických obrázků), kterým předá argumenty příkazové řádky. Poté sám zpracuje argumenty, protože některé z nich mohou

[2] http://www.imagemagick.org/discourse-server/viewtopic.php?f=1&t=18223

být určeny pro ovládání hlavního programu. Rozhodne, který modul bude spuštěn, a následně jej využije pro zpracování vstupních souborů. Pro zpracování parametrů příkazové řádky, pro hlídání jejich správného zadávání a výpisy nápověd je po dohodě s vedoucím práce použit modul `argparse`, který se stal součástí instalace interpretu Pythonu ve verzi 3.2.

Součástí aplikace je i její konfigurační soubor `mediadiff.conf` obsahující výchozí nastavení aplikace.

### 4.3.2 Modul pro porovnávání statických obrázků

V adresáři `modules` je ve vlastním adresáři `imagemagick` umístěn modul pro porovnávání statických obrázků. Modul sestává ze zdrojového souboru `imagemagick.py`, který obsahuje třídu `imagemagick`. Její metoda `argParser` poskytuje zpracované argumenty příkazové řádky. Metoda `run` provádí vlastní porovnávání. Nejprve zajistí načtení obrázku do dočasného souboru (s využitím modulu Pythonu `tempfile`), pokud byl soubor předán ze standardního vstupu. Dále jsou zjištěny vlastnosti obrázků pomocí nástroje `identify` balíku ImageMagick (voláno jako podproces s využitím třídy `Popen`). Výstup je zpracován za použití regulárních výrazů (modul `re` v Pythonu). Poté jsou podle zvolených parametrů, pokud to dává smysl (z hlediska rozměrů obrázků), provedeny rotace a vybrána nejvhodnější z nich (taková, při níž porovnávání programem `compare` vrací nejnižší číslo rozdílnosti metrikou *Mean absolute error*). Samotné porovnání s výstupem do souboru, v němž jsou změny zaznačeny, se provádí opět programem `compare` do definovaného souboru s typem podle přípony nebo typu vstupních obrázků nebo na standardní výstup ve formátu PBM (textový výstup, kde *1* značící změny a *0* označuje shodné body), který byl zvolen po dohodě s vedoucím práce a dalšími členy týmu.

Byly implementovány různé možnosti nastavení porovnávání. Tato nastavení je možno zvolit při použití parametrů příkazové řádky nebo v konfiguračním souboru. Pokud potřebujeme ignorovat některé rozdíly mezi obrázky, což bývá vhodné zejména při použití ztrátových formátů, můžeme použít přepínač `--fuzz`. Parametrem `--dissimilarity-threshold` nebo zkráceně `-dth` je možno ovládat práh, za kterým bude aplikace považovat rozdíly mezi soubory za příliš velké, a tudíž prohlásí obrázky za nepodobné a porovnání neprovede. Použitím parametru `--rotate` můžeme zvolit možnost provedení porovnání i s rotovanými obrázky, pokud to bude dávat smysl vzhledem k jejich rozměrům. Podobně je možno přepínačem `--match-subimages` zajistit hledání podobrazů, pokud je to možné. Přepínač `--fast` povolí rychlou detekci rozdílů a hledání podobrazů tak, že nejprve oba obrázky patřičně zmenší tak, aby žádný rozměr nepřesáhl 128 px. Stručný popis použitelných parametrů můžeme získat spuštěním s parametrem `--help`, jak je ukázáno v příloze (E).

Možnost samostatného spuštění modulu je zajištěna běžnou zvyklostí jazyka Python – testem na řetězcovou proměnnou `__name__`. Pokud je její hodnota rovna řetězci `__main__`, je modul spouštěn přímo a před jeho vlastní činností jsou tedy provedeny nezbytné kroky jako zpracování argumentů příkazové řádky nebo načtení konfiguračního souboru.

### 4.3.3 Modul `utils` pro konfiguraci

V rámci implementace aplikace MediaDiff byl implementován také modul `utils`, který provádí zpracovávání konfiguračních souborů. Očekávány jsou konfigurační soubory ve formátu INI, kde je možno využít sekce `[parameters]` k definování výchozích parametrů, které by jinak bylo potřeba vždy zadat z příkazové řádky. Příklad takového konfiguračního souboru můžeme vidět v příloze (D).

## 4.4 Testování

Po implementaci aplikace bylo provedeno testování, aby bylo možno ověřit, zda implementované schopnosti programu pracují takovým způsobem, jak jsme požadovali a navrhli. Testování probíhalo nejprve na případech, které očekáváme při běžném použití a následně byla sestavena sada testů případů, které mohou nastat okrajově. Všechny případy byly testovány jak s využitím hlavního programu, tak při samostatném spuštění modulu.

### 4.4.1 Běžné případy předpokládaného použití

Postupně byl testován běh aplikace a ověřovány výstupy pro různé případy vstupních obrázků:

- Dva stejné obrázky ve stejném formátu
- Dva stejné obrázky v různých formátech
- Dva různé obrázky ve stejném formátu
- Dva různé obrázky v různých formátech

Každý tento test byl vždy proveden několikrát v dalších variantách:

- Různé rotace obrázků
- Vstupní obrázky ze standardního vstupu a výsledky na standardní výstup
- Různé kombinace parametrů spouštění (povolení hledání rotací a podobrazů, různá míra tolerance odchylek)

Po některých drobných opravách (zejména přehlédnutí a překlepy) byly všechny testy úspěšně provedeny, přičemž kontrola probíhala u každého jednotlivého případu manuálně.

### 4.4.2 Okrajové případy

Jako zvláštní případy byly definovány některé kombinace nastavení a parametrů spouštění, které nesouvisí přímo s hlavní funkčností aplikace:

- Chybějící nebo chybně zadané parametry příkazové řádky
- Chybějící nebo porušený konfigurační soubor
- Různě zadané cesty k souborům (absolutní, relativní) a různé znaky v cestách a názvech souborů (diakritika, mezery)
- Hledání podobrazů i s rotacemi

Také tyto testy proběhly úspěšně a jejich výsledky odpovídaly očekávání. Opět bylo opraveno několik menších chyb souvisejících s nepozorností.

### 4.4.3 Závěry vyplývající z testování, řešení nalezených problémů

Testováním aplikace byly objeveny některé menší chyby, které se podařilo snadno opravit. Mezi nejvýraznějšími byla chyba zpracování výstupu programu `identify`, kde byl použit regulární výraz, který ovšem nepředpokládal mezery ve jménech vstupních obrázků.

Jiným problémem, který však není chybou v aplikaci samotné, je hledání podobrazů. Jak již bylo naznačeno (3.1.3), toto hledání může trvat velmi dlouhou dobu. V praxi se ukázalo, že je kvůli tomu pro běžné obrázky nepoužitelné. Naměřené časy (běžný počítač, využit systémový program `time`) a srovnání s lokalizací podobrazů knihovnami OpenCV (zdrojový kód v příloze C) uvádí tabulka 4.4.3.

Tabulka 4.1: Porovnání časů hledání podobrazů

| *Obrázek* | *Podobraz* | *Doba* `compare` | *Doba* `OpenCV` |
|---|---|---|---|
| $100px \times 100px$ | $50px \times 50px$ | 0m0.376s | 0m0.437s |
| $300px \times 300px$ | $50px \times 50px$ | 0m18.743s | 0m0.754s |
| $1000px \times 1000px$ | $200px \times 200px$ | 3m39.159s | 0m3.915s |
| $2816px \times 2112px$ (6 Mpx foto) | $400px \times 400px$ | $>$ 8 h (dále neměřeno) | 0m8.964s |

Během testování byla potvrzena správná implementace. Přestože byly opraveny některé menší chyby, nebylo nutno do aplikace nijak výrazně zasahovat. Ukázala se tak výhodnost dobře propracovaného návrhu, který již nebylo třeba měnit a nalezené chyby tak bylo poměrně snadné opravit. V současné době aplikace neobsahuje žádné další známé chyby. Přestože další chyby mohou být ještě nalezeny, zřejmě nebudou zásadního charakteru a návrh aplikace zůstane původní.

## 4.5 Další možnosti vývoje

Prototyp aplikace MediaDiff a jeho modul po porovnávání statických obrázku jsou tedy implementovány a úspěšně otestovány. Další vývoj bude tedy závislý na nasazení a použití aplikace v praxi. Mohou být odhaleny další chyby a nedostatky, které zatím nevynikly, protože žádná sada testů nemůže pokrýt naprosto bezvýhradně všechny možnosti použití a kombinace parametrů a vstupních dat.

Je možno též spatřovat některé možnosti vývoje, které zatím nejsou dosažitelné, ale lze očekávat, že v budoucnu tyto cesty vývoje a zlepšení využitelné budou. Jiná vylepšení, která by mohla být učiněna, přesahují rámec této práce. Oblasti možného zlepšení jsou různé, jak si nyní krátce naznačíme.

### 4.5.1 Aplikace MediaDiff

Vývoj samotné aplikace bude závislý na konkrétním rozpracování prototypů. Již nyní je však možno spatřovat některé možnosti dalšího rozvoje. Zejména jde o oblast začlenění do systému. Bylo by například vhodné vytvořit instalační balíčky pro různé operační systémy nebo jiným způsobem zajistit pro uživatele snadnou integraci aplikace do systému. Vylepšen by mohl být systém používání modulů, pokud by hlavní aplikace zpracovávala MIME typy souborů a podle nich vybírala vhodný modul. To se zatím jeví jako problematické, protože

jazyk Python tuto funkčnost přímo nepodporuje. Bylo by však možné použít systémový nástroj `file`, který je běžnou součástí distribucí systémů Unixového typu.

### 4.5.2 Uživatelské rozhraní

Rozhraní, které nyní aplikace využívá, tedy ovládání z příkazové řádky, je velmi vhodné při strojovém zpracování vstupů, např. při dávkovém zpracování dat, kdy můžeme aplikaci začlenit do různých skriptů. Takovéto rozhraní však není příliš přívětivé pro uživatele, kteří nemají větší zkušenosti s ovládáním počítače pomocí příkazové řádky. Do budoucna by tedy bylo možné implementovat nějaké grafické uživatelské rozhraní, které by mohli intuitivně ovládat i méně zkušení uživatelé.

### 4.5.3 Použití programového rozhraní balíku ImageMagick

Místo přímého použití programu `compare` by bylo vhodné pro samotnou činnost porovnávání obrázků využít knihovny. Balík ImageMagick nyní nabízí programové rozhraní pouze pro verzi 2 jazyku Python. Až budou představeny knihovny i pro verzi 3, mohou být využity místo spouštění programu `compare`. Knihovny by měly poskytnout zejména větší pohodlí při implementaci, kdy bude možno získat informace přímo a nebude potřeba zpracovávat výstup programu.

### 4.5.4 Možnosti optimalizace vyhledávání podobrazů

Jak bylo uvedeno, vyhledávání podobrazů, které je nyní dostupné použitím nástroje `compare`, je relativně k jiným možnostem velmi pomalé. Proto, pokud by tato funkcionalita byla požadována, je možno zaměřit další vývoj tímto směrem. V této oblasti by bylo možno využít např. knihoven OpenCV, které nabízí přímo funkce pro vyhledávání podobrazů (jak ukazujeme v příloze (C pro Python 2.7). Zde můžeme využít některých metod *vlícování ploch* (v rámci metod Registrace obrazů představeno v 2.5). Podle krátkých letmých pokusů je zde vyhledávání podobrazů rychlé. Např. v běžné fotografii (6 Mpx) je podobraz tvořený jednou čtyřicetinou celku vyhledán do deseti sekund (blíže v tabulce 4.4.3), a to i při mírných porušeních způsobených ztrátovostí formátu souborů.

# Kapitola 5

# Závěr

Tato práce poskytla bližší pohled na problematiku porovnávání souborů. Nejprve byly představeny algoritmy vyvinuté a používané za účelem porovnávání digitálních dat, jejich vlastnosti, přednosti i protiklady a oblasti jejich využití. V rámci uživatelského pohledu na porovnávání dat byly představeny běžně používané programy, které mají uživatelé k dispozici, pokud chtějí získat informace o rozdílech mezi různými soubory, a knihovny, které mohou využít ve svých programech vývojáři. Bližší pohled byl věnován balíku ImageMagick, který hraje klíčovou roli v implementační části této práce. Poslední část práce pojednává o vývoji aplikace MediaDiff, která vznikla v jejím rámci. Jsou zde rozebrány jednotlivé etapy vývoje, tak jak následovaly za sebou.

Zadání této práce bylo splněno ve všech svých bodech. Programy `diff` a `patch` byly představeny mezi dalšími programy využitelnými uživateli, kteří chtějí provádět porovnávání dat. Různé metriky podobnosti multimediálních dat, zejména obrázků, byly představeny při popisu algoritmů, kde bylo vždy diskutováno využití daného algoritmu pro obrazová data. Modulární programování bylo uváženo a využito při návrhu a implementaci aplikace MediaDiff.

Prototyp aplikace MediaDiff a jeho modul pro porovnávání statických obrázků implementované v rámci této práce jsou nyní plně funkční a využitelné ke svému účelu podle požadavků, které byly před vlastním návrhem definovány. Aplikace se tak může pokoušet o vyniknutí na místech, kde právě takováto funkcionalita chybí. Doposud byly běžně používány různé aplikace, které provádí porovnání jednoho typu dat. Avšak aplikace, která by zpracovávala různé typy dat současně s předností použitelnosti rozhraní jedním způsobem společným pro všechny typy vstupních dat, se běžně nepoužívá. Právě taková se aplikace MediaDiff snaží být. Její následující vývoj bude záviset na praktickém používání. Již nyní jsou však znatelné jisté oblasti, do kterých by mohlo být v budoucnu při dalším vývoji aplikace vkládáno úsilí. I tyto možnosti jsou v této práci diskutovány.

# Literatura


[1] The Python Tutorial: Modules [online]. `http://docs.python.org/tutorial/modules.html`, [cit. 29. 4. 2011].

[2] Avasilcutei, A.; Paslariu, C.; Ungureanu, L.; aj.: The ImageMagick graphics library: A gentle introduction to Magick++ [online]. `http://www.imagemagick.org/Magick++/tutorial/Magick++_tutorial.pdf`, [cit. 14. 4. 2011].

[3] Bergroth, L.; Hakonen, H.; Raita, T.: A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, 9 2000, s. 39–48.

[4] Bradski, G.; Kaehler, A.: *Learning opencv*. O'Reilly, 2008, ISBN 0596516134.

[5] Gusfield, D.: *Algorithms on strings, trees, and sequences: computer science and computational biology*. New York, NY, USA: Cambridge University Press, 1997, ISBN 0-521-58519-8.

[6] Hamming, R. W.: Error detecting and error correcting codes. *Bell System Tech. J.*, ročník 29, 1950: s. 147–160, ISSN 0005-8580.

[7] Hunt, J.; McIlroy, M.: An Algorithm for Differential File Comparison. *Computing Science Technical Report 41*, 1976.

[8] Klein, P. N.; Sebastian, T. B.; Kimia, B. B.: Shape matching using edit-distance: an implementation. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, SODA '01, Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2001, ISBN 0-89871-490-7, s. 781–790.

[9] Kolektiv autorů: FreeImage [online]. `http://freeimage.sourceforge.net/`, [cit. 20. 4. 2011].

[10] Kolektiv autorů: ImageMagick [online]. `http://www.imagemagick.org/`, [cit. 29. 4. 2011].

[11] Kolektiv autorů: OpenCV [online]. `http://opencv.willowgarage.com/`, [cit. 15. 4. 2011].

[12] Kolektiv autorů: VIPS [online]. `http://www.vips.ecs.soton.ac.uk/`, [cit. 23. 4. 2011].

[13] Lee, S.-Y.; Shan, M.-K.; Yang, W.-P.: Similarity retrieval of iconic image database. *Pattern Recognition*, ročník 22, č. 6, 1989: s. 675–682, ISSN 0031-3203.

[14] Marzal, A.; Vidal, E.: Computation of Normalized Edit Distance and Applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, ročník 15, 1993: s. 926–932, ISSN 0162-8828.

[15] Navarro, G.: A guided tour to approximate string matching. *ACM Comput. Surv.*, ročník 33, March 2001: s. 31–88, ISSN 0360-0300.

[16] Přispěvatelé encyklopedie Wikipedia: Diff — Wikipedia, The Free Encyclopedia [online]. `http://en.wikipedia.org/wiki/Diff`, 2011, [cit. 16. 3. 2011].

[17] Přispěvatelé encyklopedie Wikipedia: Hamming distance — Wikipedia, The Free Encyclopedia [online]. `http://en.wikipedia.org/wiki/Hamming_distance`, 2011, [cit. 11. 4. 2011].

[18] Přispěvatelé encyklopedie Wikipedia: Levenshtein distance — Wikipedia, The Free Encyclopedia [online]. `http://en.wikipedia.org/wiki/Levenshtein_distance`, 2011, [cit. 28. 4. 2011].

[19] Salehi, S.: *Imagemagick Tricks: Web Image Effects from the Command Line And Php*. Packt publishing ltd, 2006, ISBN 1904811868.

[20] Yujian, L.; Bo, L.: A Normalized Levenshtein Distance Metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, ročník 29, 2007: s. 1091–1095, ISSN 0162-8828.

[21] Zitová, B.; Flusser, J.: Image registration methods: a survey. *Image and vision computing*, ročník 21, č. 11, 2003: s. 977–1000.

# Příloha A

# Obsah CD

- Adresář `mediadiff` – aplikace MediaDiff a její moduly.
  - Soubor `mdiff` – zdrojový kód v jazyce Python zajišťující spuštění aplikace MediaDiff.
  - Adresář `mediadiff` – jádro aplikace MediaDiff (soubor `main.py`), pomocný modul (soubor `utils.py`), konfigrační soubor (soubor `mediadiff.conf`).
  - Adresář `modules` – moduly aplikace MediaDiff.
  - Adresář `modules/imagemagick` – moduly pro porovnávání statických obrázků (soubor `imagemagick.py`) s možností samostatného spuštění, konfigrační soubor (soubor `imagemagick.conf`).
- Adresář `thesis` – tato práce ve formátu PDF a její zdrojové texty pro program LaTeX.

# Příloha B

# Úsek zdrojového kódu programu `compare`

Zde uvádíme část zdrojového textu programu `compare` zejména pro názornost jeho práce. Z uvedeného výňatku (soubor `compare.c` je zřejmé, jak program zpvacovává obrázky po jednotlivých pixelech. Zdrojový kód je šířen pod podmínkami open-source license Apache 2.0[1]. Znaky `...` označují vypuštění části kódu, který je zde nepodstatný.

```
for (y=0; y < (ssize_t) image->rows; y++)
{
...
  p=GetCacheViewVirtualPixels(image_view,0,y,image->columns,1,exception);
  q=GetCacheViewVirtualPixels(reconstruct_view,0,y,
    reconstruct_image->columns,1,exception);
  r=QueueCacheViewAuthenticPixels(highlight_view,0,y,
    highlight_image->columns,1,exception);
...
  for (x=0; x < (ssize_t) image->columns; x++)
  {
    MagickStatusType
      difference;
...
if (((channel & RedChannel) != 0) &&
    (GetRedPixelComponent(p) != GetRedPixelComponent(q)))
  difference=MagickTrue;
if (((channel & GreenChannel) != 0) &&
    (GetGreenPixelComponent(p) != GetGreenPixelComponent(q)))
  difference=MagickTrue;
if (((channel & BlueChannel) != 0) &&
    (GetBluePixelComponent(p) != GetBluePixelComponent(q)))
  difference=MagickTrue;
if (((channel & OpacityChannel) != 0) &&
    (image->matte != MagickFalse) &&
    (GetOpacityPixelComponent(p) != GetOpacityPixelComponent(q)))
```

[1] http://www.imagemagick.org/script/license.php

```
  difference=MagickTrue;
...
    if (difference != MagickFalse)
      SetPixelPacket(highlight_image,&highlight,r,highlight_indexes+x);
    else
      SetPixelPacket(highlight_image,&lowlight,r,highlight_indexes+x);
    p++;
    q++;
    r++;
  }
}
```

# Příloha C

# Ukázka zdrojového kódu hledání podobrazu knihovnami OpenCV

Zde ukazujeme způsob, jakým mohou být použity knihovny OpenCV pro lokalizaci podobrazů. Zdrojový kód je spustitelný interpretem jazyka Python ve verzi 2.7:

```
import cv

# načtení obrázků
src = cv.LoadImage("1.JPG", 1)
dest = cv.LoadImage("2.JPG", 1)

# ořez většího obrázku o rozměry menšího, aby nebylo hledáno za jeho hranicí
res = cv.CreateImage((2816-340+1, 2112-300+1), cv.IPL_DEPTH_32F, 1)

# hledání
cv.MatchTemplate(dest, src, res, cv.CV_TM_SQDIFF)

# výpis výsledku
print(cv.MinMaxLoc(res))
```

# Příloha D

# Ukázka konfigračního souboru

Zde je pro názornost konfigurace aplikace pomocí konfiguračního soubrou uveden jeho příklad.

```
  # general parameters of this module to be processed in mediadiff
# author: Jan Brothanek, xbroth01@stud.fit.vutbr.cz
[mediadiff]
extensions=bmp,jpg,png,gif

[priorities]
# priority of each extension, default is the lowest possible
bmp=100000
gif=100000

# default values of parameters
[parameters]
--dissimilarity-threshold=0.2
--fuzz=5%
--rotate=True
--match-subimages=True
```

# Příloha E

# Nápověda modulu pro obrázky

Zde uvádíme nápovědu, kterou vypisuje modul `imagemagick.py` a můžeme ji obdržet spuštěním programu s parametrem `--help` s využitím modulu `argparser`.

```
usage: imagemagick.py [-h] [--fuzz FUZZ] [-dth DISSIMILARITY]
                      [--rotate] [--match-subimages] [--fast] [-o OUTPUT]
                      file1 [file2]

positional arguments:
  file1                 A reference file
  file2                 A file to compare

optional arguments:
  -h, --help            show this help message and exit
  --fuzz FUZZ           Don't consider differences smaller than value
  -dth DISSIMILARITY, --dissimilarity-threshold DISSIMILARITY
                        Maximum RMSE for (sub)image match
  --rotate              Choose best match by rotating image (where sensible)
  --match-subimages     Try to find subimages
  --fast                First resize images to be 128px or smaller in each
                        axis
  -o OUTPUT, --output OUTPUT
                        An output file
```

# Příloha F

# Bližší popis problému nejdelšího společného podřetězce a jeho řešení

## F.1 Další způsoby řešení

LCS problém může být redukován i na dva jiné známé problémy. $lcs(X, Y)$ je běžně řešen technikami dynamického programování a plněním tabulky velikosti $m \times n$. Prvky tabulky mohou být považovány za vrcholy grafu a jednoduché závislosti mezi hodnotami tabulky definují hrany. Úloha spočívá v nalezení nejdelší cesty mezi vrcholy a levým horním a pravým spodním rohem tabulky. Jinou redukcí může být LCS problém převeden na problém nejdelšího rostoucího podřetězce ($lis$).

## F.2 Teoretická složitost řešení

Studie teoretické složitosti problému LCS (odkazy na další literaturu můžeme nalézt v [3]) uvádí jako spodní hranici $\Omega(n^2)$ za předpokladu, že základní porovnávací operací je „rovnost/nerovnost“ a vstupní abeceda je neomezená. Nicméně pokud je vstupní abeceda omezená, dosáhneme spodní hranici $\Omega(\sigma n)$. V praxi pak podkladové schéma kódování symbolů vstupní abecedy implikuje jejich topologické uspořádání a relace porovnání $\leq$ nám dává více informací, čímž redukuje spodní hranici složitosti na $\Omega(n \log m)$. Ze současných metod je teoreticky nejrychlejší metoda pánů Maseka a Petersona o složitosti $O(n^2/\log n)$.

## F.3 Obecný algoritmus řešící problém nejdelšího společného podřetězce

Tato sekce poskytuje obecný algorimus řešící problém nejdelšího společného podřetězce podle [3].

Nejprve představme některé myšlenky, které jsou potřebné k algoritmickému popisu. Pár $(i, j)$ definuje shodu, jestliže $X[i] = Y[j]$. Potom množina všech shod bude

$$M = \{(i, j) \mid X[i] = Y[j],\ 1 \leq i \leq m,\ 1 \leq j \leq n\}. \tag{F.1}$$

Každá shoda náleží třídě $C_k = \{(i,j) \mid (i,j) \in M \text{ a } R[i,j] = k\}$, $1 \leq k \leq r$. Může být též vhodné definovat pseudotřídu $C_0 = \{(0,0)\}$. Shoda, která náleží $C_k$ je nazývána *k-shoda*. V představené tabulce 2.3 shody v kroužcích a obdélnících mající hodnotu $k$ definují třídu $C_k$. Evidentně každá shoda náleží právě jedné třídě. Takto třídy rozdělují všechny shody v $M$. Některé z $k$-shod jsou více důležité (v obdélnících) než jiné (v kroužcích). Pro objasnění významu uvažujeme shody $(i,j)$ a $(i',j')$ z $C_k$, pro které buď $i = i'$ a zároveň $j \leq j'$, nebo $i \leq i'$ a zároveň $j = j'$. Každý prvek $C_{k+1}$, který může následovat za $(i',j')$, může následovat také za $(i,j)$ v LCS. Proto postačuje zabývat se jen *dominantními shodami* $(i,j)$ při řešení problému. Nechť $D_k$ značí množinu všech dominantních $k$-shod. Dále nechť $D = \bigcup_{k>0} D_k$ a $d = |D|$.

V tabulce 2.3 oblasti, kde $R[i,j]$-hodnoty jsou stejné, byly ohraničeny lomenými čarami, tzv. *konturami*. Bezprostředně pod $k$-tou konturou leží všechny $k$-shody (1-shody jsou pod nejvyšší čárou, 2-shody pod další atd.). Rohy lomených čar jsou definovány polohou dominantních hodnot, které se řídí vlastností uspořádanosti: pokud $D_k = (i_1,j_1), (i_2,j_2), \ldots, (i_\ell,j_\ell)$, shody mohou být očíslovány tak, že $i_1 < i_2 < \cdots < i_\ell$ a $j_1 > j_2 > \cdots > j_\ell$.

Běžně jsou užívány tři strategie nalezení dominantních shod:

1. Je možno zpracovat tabulku *řádkově* hledáním všech dominantních shod, jedna pro každou jednu konturu. Použitím pomocných datových struktur je toho možno dosáhnout efektivně, protože je možno soustředit se pouze na dominantní shody.

2. Jinou možností je postupovat od *kontury ke kontuře*. V tomto přístupu je však někdy nutno zpracovávat velkou oblast (více řádků) tabulky v každém kroku a oblasti dvou po sobě následujících kroků se mohou značně překrývat. Pro vyhnutí se tomuto problému je možno užít sofistikovaného obratu k omezení oblasti zájmu na malé nepřekrývající se oblasti.

3. Poslední užívanou možností je snaha o *diagonální plnění tabulky* se začátkem v $R[0,0]$ a dosažení spodního pravého rohu cestou nejvíce blízkou diagonále, jak je to možné.

Z těchto přístupů bude z důvodů rozsahu představen první – Zpracování tabulky po řádcích. Přehled o ostatních strategiích je možno získat v literatuře, např. v [3].

### F.3.1 Zpracování tabulky po řádcích

Řádkové zpracování je metoda odvozená od tradičního postupu pro plnění dynamické programovací tabulky. Nicméně v tomto případě se soustřeďujeme jen na prvky tabulky, které korespondují ke shodám. Každá dominantní shoda definuje nový zlom v kontuře. K udržování sloupců, kde všechny kontury prochází aktuálním řádkem, používáme pole $MinYPrefix[1..p]$, kde $MinYPrefix[\ell]$ dává $Y$-index, na kterém se $\ell$-tá kontura nachází. Jak je naznačeno jménem pole, hodnota $MinYPrefix[\ell]$ může být pokládána za kurzor značící minimální délku prefixu $Y$ potřebného k vytvoření společného podřetězce délky $\ell$ s prvními $i$ prvky $X$. Hodnota $p$ značí $r(X[1..i], Y[1..n])$, tedy počet kontur procházejících řádkem $i$. Ve výchozím stavu jsou hodnoty $MinYPrefix$ nastaveny na „nedefinováno“.

Tabulka F.3.1 ukazuje, jak budou hodnoty pole změněny při uvážení demonstračních řetězů $X = abcdbb$ a $Y = cbacbaaba$ (nedefinované hodnoty jsou reprezentovány $n+1$; nejlevější vstup je v roli příznaku a je nastaven na hodnotu 0).

K udržování $MinYPrefix$ hodnot při pohybu z řádku na řádek potřebujeme následující pravidlo:

Tabulka F.1: Pole $MinYPrefix[1..p]$ při zpracování řádek po řádku, podle [3]

| | $MinYPrefix$ | | | | | | |
|---|---|---|---|---|---|---|---|
| Row | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | 3 | 10 | 10 | 10 | 10 | 10 |
| 2 | 0 | 2 | 5 | 10 | 10 | 10 | 10 |
| 3 | 0 | 1 | 4 | 10 | 10 | 10 | 10 |
| 4 | 0 | 1 | 4 | 10 | 10 | 10 | 10 |
| 5 | 0 | 1 | 2 | 5 | 10 | 10 | 10 |
| 6 | 0 | 1 | 2 | 5 | 8 | 10 | 10 |

**Pravidlo přepisu** Předpokládejme, že zpracováváme řádek $i$. Pro každý otevřený interval $MinYPrefix[\ell]..MinYPrefix[\ell+1], (\ell = 0, .., r)$ nalezneme shody $(i, j)$, které do něj padnou (např. shody, pro které hodnota $j$ je v tomto intervalu). Ohraničení intervalu zprava je ponecháno, pokud žádná taková shoda neexistuje. V opačném případě je přepsán na nejmenší takovou hodnotu $j$ (nejlevější shoda v intervalu). Povšimněme si, že změny probíhají současně.

Takže například při přechodu z řádku 2 na řádek 3 (v demonstrační tabulce F.3.1 můžeme vidět, že $X[3] = Y[4]$ a $MinYPrefix[1] < 4 \leq MinYPrefix[2]$, proto je proveden přepis $MinYPrefix[2]$ na 4. Obecné schéma pro postupování v dynamické programovací tabulce je představeno v algoritmu 2.

**Algoritmus 2**: Plnění tabulky dynamického programování pro LCS, podle [3]

```
1  begin
2      for i := 1 to m do MinYPrefix[i] := n + 1;
3      MinYPrefix[0] := 0; r := 0;
4      for i = 1 to m do
5          /* Přepis hodnot pro řádek i */
6          for j = 0 to r pardo
7              /* Paralelně pro každé j */
8              if rozsah [MinYPrefix[j] + 1..MinYPrefix[j + 1] − 1] obsahuje shody
               then
9                  begin
10                     MinYPrefix[j + 1] := min { ℓ|(i, ℓ) je shoda v tomto rozsahu };
11                     if j = r then
12                         r := r + 1;
13                 end
14     return r;
15 end
```

Byly navrženy různé způsoby dosažení serializace smyčky (řádek 6) a minimalizace (řádek 9). [3] odkazuje na další literaturu:

V práci pánů Hunta a Szymanského je každý řádek zpracován zprava doleva prohledáváním každé shody a přepisu pole $MinYPrefix$ pro každý z nich (nejlevější hodnota

z rozsahu zůstává jako konečná $MinYPrefix$ hodnota, jak vyplývá z přepisovacího pravidla). Pro každou shodu může být nalezen příslušný rozsah použitím binárního vyhledávání, protože hodnoty v poli $MinYPrefix$ jsou vždy ve vzestupném pořadí. Časová náročnost je potom $O(|M| \log n)$, kde $|M|$ značí počet všech shod.

Nezávisle pan Mukhopadhyay zkonstruoval velmi podobnou implementaci. Jediný základní rozdíl je pořadí zpracování: od spodního řádku nahoru, každý řádek zleva doprava. Později si pánové Hsu a Du všimli, že zde ve skutečnosti provádíme zvláštní druh sloučení mezi $MinYPrefix$ a seznamem shod na aktuálním řádku. Kvůli této symetrii můžeme provádět binární vyhledávání také v seznamu shod. Výsledkem je časová náročnost $O(rm \log(n/r) + rm)$, pokud je slučování prováděno sofistikovaně. Pánové Kuo a Cross zpracovávají shody zleva doprava, takže je možno vyhnout se nepotřebným přepisům $MinYPrefix$. Stejná myšlenka byla představena také pány Apostolico a Guerra. V téže práci také ukazují, jak se můžeme soustředit pouze na dominantní shody při zpracování každého řádku. Pan Rick rozvíjí další implementaci tohoto algoritmu a dosahuje časové složitosti $O(\sigma n + rm)$.

Pokud je $X$ dlouhé nebo $\sigma$ malé, hodnoty prvních vstupů v $MinYPrefix$ vytváří poměrně rychle sekvenci po sobě jdoucích přirozených čísel $1, 2, \ldots, k$. Důvodem je, že brzy dosahujeme stavu, kdy prvních $k$ kontur prochází sloupci $1, 2, \ldots, k$. Pokud se toto stane, nemusíme se již zabývat pozicemi indexů během přepisování, jde pouze o omezení rozsahu zájmu z předchozího pole. Je zřejmé, že počet kontur procházejících aktuálním řádkem značí horní hranici rozsahu pole. Sofistikovaným způsobem udržování rozsahu relevantních indexů je studie řádků a sloupců zároveň. Tzn. v kroku $i$ přepíšeme pole $MinYPrefix$ jak pro $i$-tý řádek, tak pro $i$-tý sloupec. V zásadě procházíme po diagonále tabulky a když zjistíme během vyšetřování řádků dominantní shodu nalezenou sloupcovým výpočtem (nebo opačně), víme, že všechny dominantní shody kontury byly nalezeny. Přestože implementace může být nesnadná, ukazuje se v praxi jako velmi rychlá. Její teoretická časová náročnost je $O(\sigma n + r(n - r))$.