PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Rozpoznávání ručně psaného textu pomocí fuzzy logiky

2013 Tomáš Urbanec

Anotace

Kvalitní rozpoznávání ručně psaného textu je výzva už pro několik generací informatiků. Existuje zde velké množství metod, myšlenek a principů. Jedním z těch zajímavějších je právě přístup založený na fuzzy logice. V práci je obsaženo základní shrnutí možností fuzzy logiky v této oblasti a hlavně popis několika takových algoritmů, včetně návrhu algoritmu nového. Součástí práce je i demonstrační aplikace pracující s autorovým rukopisem.

Obsah

Seznam obrázků

Seznam tabulek

Seznam pseudokódů

1. Úvod

I přesto, že se dnes většina informací zpracovává pomocí počítačů, je stále mnoho situací, kdy je máme k dispozici pouze na papíře. Samozřejmě si často přejeme uložit je pro další práci do elektronické podoby a to pokud možno jinak, než ručním přepisováním. Právě tímto problémem se zabývá oblast informatiky většinou nazývaná OCR. OCR je zkratka z anglického Optical Charater Recognition, česky optické rozpoznávání znaků, neboli rozpoznávání znaků pouze na základě jejich vzhledu.

OCR se tedy zabývá digitalizováním textové informace, která už byla někdy napsána nebo vytištěna a není dostupná v elektronické formě. Případů, kdy je toho potřeba, je dnes čím dál tím více: skenování dokumentů do počítače, rozpoznání poštovních směrovacích čísel na dopisech, přečtení státní poznávací značky z radarové fotografie rychle jedoucího automobilu, používání dotykových přístrojů pro ruční zapisování poznámek, digitalizace starých výtisků novin a takto by se dalo pokračovat ještě dlouho. Je tedy zřejmé, že OCR se stalo důležitou součástí moderního světa.

Skutečnost je však taková, že lidé se touto myšlenkou zabývají už téměř sto let. Přesněji, už v roce 1929 si G. Tausheck nechal patentovat myšlenku OCR v Německu a o čtyři roky později udělal totéž P. W. Handel v USA. Tenkrát to sice byla záležitost mechanických strojů a pro určení znaku byla nutná přesná shoda, ale základní myšlenku má OCR od té doby stále stejnou.[1] Druhým významným rokem pro tuto práci je rok 1965, kdy L. A. Zadeh publikoval článek „Fuzzy Sets“, neboť tématem této bakalářské práce je právě využití fuzzy logiky v OCR.

Téma jsem si vybral proto, že bych v budoucnu rád vytvořil nástroj pro digitalizaci ručně psaných poznámek, neboť v mém případě jejich množství velmi rychle roste. Díky tomu je obtížné udržovat si přehled o tom, co vše a kde přesně se v nich dá najít. Jejich převod do elektronického textu by to, díky dostupným nástrojům pro organizaci dat na počítači, velice usnadnil. Pro takovýto nástroj je ovšem potřeba použít mnoho různých technologií a přístupů k jednotlivým částem problému. Znalost využití fuzzy logiky v oblasti OCR by při takovém vývoji mohla být velkým přínosem.

1.1. Dělení OCR

Je mnoho různých způsobů, jak dělit OCR na podkategorie. Například podle použitých technik určení znaku: porovnávání se vzorem, neuronové sítě, fuzzy logika, ...; nebo podle způsobu použití: offline - text byl získán někdy v minulosti (např. naskenované noviny), online - text je rozpoznáván ihned při psaní (například dotyková zařízení); či podle druhu textu: ručně psaný, tištěný (speciální font pro OCR nebo libovolný font). Dělení se dá vymyslet mnoho. S přihlédnutím k těm uvedeným zapadá tato práce do offline rozpoznávání ručně psaného textu založeného na fuzzy logice.

1.2. Základní poznatky o fuzzy logice a fuzzy množinách

Následuje stručný úvod do fuzzy logiky. Při jeho psaní jsem čerpal z [2], [3] a [4]. Jsou zde jen základní definice a vztahy tak, aby to stačilo pro pochopení principů použitých v algoritmech určování znaků. Pro hlubší porozumění tématu pak lze použít některý z citovaných zdrojů.

1.2.1. Fuzzy množiny

Pro pochopení pojmu fuzzy množina je nejlepší popsat jeho vztah k pojmu klasické množiny. Zaveďme si tedy nejprve pojmy množina a charakteristická funkce množiny.

Množina *Množina* je soubor přesně určených a rozlišitelných objektů, tyto objekty pak nazýváme prvky množiny. Speciálními množinami jsou pak univerzum (U) - množina všech prvků a prázdná množina ($\emptyset$) - množina neobsahující žádný prvek.

Charakteristická funkce množiny Zobrazení $\mu_A : X \to \{0, 1\}$ nazveme *charakteristickou funkcí množiny* A právě tehdy, když pro všechna x platí:

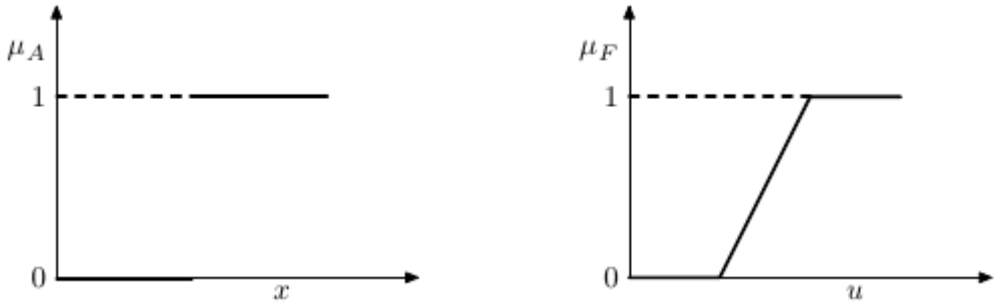
$$\mu_A(x) = \begin{cases} 1 & \text{pro } x \in A \\ 0 & \text{pro } x \notin A \end{cases}$$

Nyní nad těmito pojmy postavíme definici pro fuzzy množinu. Představme si, že existuje prvek, který do množiny patří jen částečně. Například máme-li zpola naplněnou sklenici, pak bychom chtěli říct, že do množiny plných sklenic patří jen napůl. „Charakteristická funkce“ množiny s takovouto sklenicí by pak nenabývala pouze hodnot $\{0, 1\}$ ale i hodnoty 0.5. A nyní si představme, že vodu do sklenice můžeme libovolně dolévat a zase z ní vylévat. Pak by příslušná funkce mohla nabývat hodnot z intervalu $[0, 1]$. A právě takovouto funkci nazýváme *funkce příslušnosti* a množinu, kterou lze popsat pomocí této funkce příslušnosti, nazýváme *fuzzy množina*. Toto doplnění klasických množin zavedl L.A. Zadeh v práci „Fuzzy sets“ v roce 1965. Je potřeba uvést ještě formální definici zmíněných pojmů.

Funkce příslušnosti a fuzzy množina Funkce μ_F se nazývá *funkce příslušnosti fuzzy množiny* F, pokud $\mu_F : U \to [0, 1]$.
Fuzzy množina F je pak jednoznačně určena všemi dvojicemi $(u,\ \mu_F(u))$, kde $u \in U$.

Na obrázku 1. jsou grafy charakteristické funkce množiny A a funkce příslušnosti fuzzy množiny F. Lze si je představit třeba jako znázornění plnosti sklenice dle množství tekutiny pomocí klasické množiny a pomocí fuzzy množiny.

Obrázek 1. Rozdíl mezi klasickou množinou A a fuzzy množinou F znázorněný pomocí charakteristické funkce množiny A, μ_A a funkce příslušnosti fuzzy množiny F, μ_F. Oba obrázky převzaty z [3].



Fuzzy logika je pak rozšířením klasické logiky postaveným na fuzzy množinách. Umožňuje nám formalizovat neurčitá tvrzení, jako například „Sklenice je skoro plná“. Právě díky tomu je velice užitečná v mnoha oblastech lidského vědění. Její využití v OCR je značné a existuje mnoho algoritmů, které se s nepřesností jednotlivých písmen (nejen) při ručním psaní vypořádávají právě pomocí fuzzy logiky.

1.3. Využití fuzzy logiky v OCR

Ze shrnutí doposud popsaných aplikací fuzzy logiky v OCR v práci [5] plyne, že fuzzy logiku lze pro zlepšení výsledků použít mnoha způsoby. Například strukturální způsoby, kdy je znak rozdělen na jednotlivé části a ty jsou následně fuzzy způsobem přiřazeny k některé ze vzorových částí, nebo přímo k některému ze vzorových znaků. Další možností je využít fuzzy logiku k vytvoření pravidel popisujících všechny možnosti změn v rámci jednotlivých znaků. Také je v citované práci zmíněna existence fuzzy algoritmu pro detekci přebytečných inkoustových skvrn v dodaném dokumentu, nebo pro opravu sklonu znaků či řádků. Různých způsobů, jak využít fuzzy logiku k návrhu algoritmu souvisejícího s OCR, je tedy opravdu mnoho.

V této práci jsem se zaměřil na, dle mého názoru, nejvýznamnější z těchto skupin a to využití fuzzy logiky v algoritmech identifikujících jednotlivé znaky. Přesto je zde mnoho různých přístupů, a proto jsem se snažil vybrat takové algoritmy, aby pokryly co největší část tohoto velkého prostoru.

1.4. Fáze rozpoznávání

Pro rozpoznání textu nestačí jen jeden algoritmus. Ve skutečnosti je k tomuto úkolu potřeba spolupráce několika algoritmů, které spolu na první pohled nemusí

souviset. Proto si zde uvedeme základní fáze rozpoznávacího procesu a nastíníme, co se během nich děje. Detailnější popis fází pak obsahují příslušné kapitoly. Fáze jsou tedy následující:

1. Preprocessing, neboli předzpracování vstupního dokumentu
2. Samotné rozpoznávání znaků a úkony s tím související
3. Postprocessing, neboli finální úpravy získaného výstupu

2. Preprocessing

Do kategorie preprocessingu patří všechny operace, které se vstupním dokumentem musíme provést před samotným určováním znaků. Patří zde převod do vyžadovaného barevného modelu, opravy chyb vzniklých při digitalizaci dokumentu, rozdělení textu na písmena a mnoho dalších úprav vstupu. Následuje stručný popis nejdůležitějších úprav.

2.1. Binarizace

Binarizace obrázku je jeho převedení do barevného modelu s pouze dvěma barvami, obvykle černou a bílou. Problémem je určit, kde v původním barevném modelu bude hranice. Tento problém řeší algoritmus, který se, podle svého autora, často označuje jako „Otsu thresholding“. Při jeho implementaci jsem čerpal z [6] a [7]. Tato statistická metoda je založena na postupném procházení všech možných hranic pro binarizaci vstupu (takových hranic je 256), přičemž nás zajímá právě ta hranice, která minimalizuje následující vztah.

$$\sigma_W^2 = W_b\sigma_b^2 + W_f\sigma_f^2$$

kde W_b je počet pixelů menších než hranice (tedy počet pixelů, ze kterých bude při dané hranici pozadí) podělený celkovým počtem pixelů, W_f je počet pixelů větších než hranice (tedy počet pixelů, ze kterých bude při dané hranici popředí) podělený celkovým počtem pixelů a σ_*^2 jsou odpovídající výběrové rozptyly.

Takto získaná minimální hranice se použije pro rozdělení obrázku. Pro samotnou implementaci se používá vztah odvozený od výše uvedeného, neboť není tak výpočetně náročný. Protože princip algoritmu se tím nezmění a naším cílem je popsat hlavně algoritmy pro rozpoznávání znaků založené na fuzzy logice, tak zde tyto úpravy nebudou rozepisovány. Zájemci je mohou najít v obou citovaných zdrojích.

Poznámka V dalším textu budeme předpokládat, že obrázek, o kterém mluvíme, je černobílý, nebude-li uvedeno jinak.

2.2. Rozdělení textu na písmena

Pro použití algoritmů určujících hodnotu znaku je potřeba získat obrázek, který obsahuje pouze tento znak. Než se ovšem dostaneme k samotným znakům, musíme nejdříve rozdělit text na řádky a řádky na slova.

2.2.1. Rozdělení textu na řádky

Text je v této práci dělen na řádky algoritmem jasových součtů, který je pro tento účel hojně používán. Mě k němu přivedla práce [8]. Algoritmus vytvoří pro obrázek pole jasových součtů, které obsahuje tolik prvků, jaká je výška obrázku. I-tý prvek tohoto pole obsahuje součet hodnot pixelů v i-tém řádku. Nyní pole procházíme a hledáme maxima a minima. Tam, kde jsou maxima, je nejméně černých pixelů, a tedy prázdný prostor mezi jednotlivými řádky. Naopak tam, kde jsou minima, je mnoho černých pixelů, a tedy je v daném pixelovém řádku nějaký objekt. Princip nejlépe vystihne pseudokód 1.

Pseudokód 1 Psedudokód pro hledání řádků pomocí jasových součtů. Prefix p_ označuje proměnné a funkce pracující s pixelovými řádky, prefix t_ pak proměnné a funkce pracující s textovými řádky.

```
1.  Radky-jasovymy-soucty(obrazek)
2.     p_soucty <- new pole[obrazek.p_vyska]
3.     foreach p_radek in obrazek
4.        p_soucty[p_radek.pozice] <- p_radek.p_soucet
5.     endforeach
6.     p_maximalniSoucty <- p_soucty.najdiMaxima
7.     p_indexyMaxim <- p_maximalniSoucty.p_urciIndexyRadku
8.     t_radky <- p_indexyMaxim.t_urciRozsahyRadku
       //t_rozsahyRadku jsou skupiny po sobě jdoucích indexů p_řádků
9.     return t_radky
```

2.2.2. Rozdělení řádků na slova a písmena

Vzhledem k podmínkám zadání můžeme na vstupu očekávat pouze velká tiskací písmena, a tedy pro rozdělení řádku na slova a písmena můžeme opět použít princip jasových součtů. Jedinou změnou je, že nyní děláme jasové součty sloupců místo řádků. Kdyby na vstupu mohla být i psací písmena nebo jiné spojité typy písma, pak bychom museli použít jiný algoritmus.

Také je v této fázi nutné odlišit, kde je mezera pouze mezi písmeny a kde je mezera mezi slovy. Řešením je stanovit minimální velikost mezery mezi slovy. Tato hranice může být stanovena jako konstanta, nebo lépe pomocí šířky znaků z daného řádku. Například dvojnásobek průměrné velikosti znaku.

Poznámka Toto je poslední algoritmus, který se vstupním obrázkem pracuje v původním rozdělení barev, tedy popředí je černé a pozadí bílé. Další algoritmy už u vstupu předpokládají převrácené barvy.

2.3. Algoritmus pro získání kostry znaku

Pro některé algoritmy je vhodné nebo dokonce nezbytné, aby dodaný obrázek obsahoval kostru znaku. Protože lze najít více definic toho, co je to kostra znaku, tak zde uvedu definici, která je použitá v této práci.

Kostra znaku Obrázek obsahuje kostru znaku z původního obrázku, pokud všechny čáry v něm mají jednopixelovou tloušťku, pokud je zachováno propojení jednotlivých čar dle původního obrázku a pokud všechny čáry zachovávají pozici středu a délku původních čar. Dále je nezbytné, aby každý bod objektu mimo bodu křížení/spojení dvou čar měl nejvýše dva sousední pixely patřící k nějakému objektu.

Algoritmus Algoritmus, použitý k tvorbě kostry znaku, pochází z práce [9]. Tento algoritmus je iterační a v každé iteraci provádí dvě subiterace přes celý obrázek.

Obrázek 2. Pořadí sousedních bodů pro algoritmus kostry znaku

P_9	P_2	P_3
P_8	P_1	P_4
P_7	P_6	P_5

V první subiteraci je odstraněn každý bod P_1, který splňuje všechny následující podmínky:

1. $2 \leq B(P_1) \leq 6$
2. $A(P_1) = 1$
3. $P_2 \cdot P_4 \cdot P_6 = 0$
4. $P_4 \cdot P_6 \cdot P_8 = 0$

Kde $A(P_1)$ je počet 01 přechodů v posloupnosti $P_2, P_3, P_4, \ldots, P_9$; $B(P_1)$ je počet nenulových sousedů pixelu P_1 a P_* jsou body dle obrázku 2. První krok odstraňuje přebytečné body na spodní a pravé straně a v horních a levých rozích objektu.

Ve druhé subiteraci jsou podmínky trochu jiné (označeno *), je tedy odstraněn každý bod P_1 splňující následující:

1. $2 \leq B(P_1) \leq 6$
2. $A(P_1) = 1$
3. * $P_2 \cdot P_4 \cdot P_8 = 0$
4. * $P_2 \cdot P_6 \cdot P_8 = 0$

Kde jednotlivé symboly mají stejný význam jako u prvního kroku. Druhý krok odstraňuje přebytečné body na horní a levé straně a ve spodních a levých rozích objektu.
Tyto dvě subiterace jsou opakovány, dokud se obrázek mění.

Úpravy algoritmu V aktuálním stavu by kostra nesplňovala poslední část výše uvedené definice kostry, a tedy je nutné celý obrázek ještě jednou projít a odstranit jeden bod z každé trojice bodů, kde spolu každé dva sousedí.

Navíc je do algoritmu přidáno ještě spojování nedotažených čar. Princip je takový, že pokud prodloužením některé čáry o malou část její délky, ve směru určeném poslední dvojicí pixelů, dojde ke spojení s jinou čárou, tak se spojení dokreslí. Toto opraví většinu nedotažených čar. Pokud k takovému dokreslení nedojde, tak se ještě prověří, jestli v blízkém okolí daného konce čáry není konec jiné čáry. Pokud ano, tak se tyto dva konce spojí. Toto pak opraví většinu špatně ukončených cyklů ve znaku (například rozpojené O vlivem těsného minutí jeho počátku).

2.4. Diakritika

Znaménka u znaků působila v algoritmech mnoho špatných identifikací, proto jsem se rozhodl zpracovat diakritiku ihned po rozdělení slov na jednotlivé znaky. Opětovným užitím principu jasových součtů (2.2.) se u každého znaku pokusíme najít vertikální mezeru v horní polovině znaku, a tím jeho obrázek rozdělit na obrázek samotného znaku a obrázek znaménka. Pokud jsme úspěšní, tak část se znaménkem ihned zpracujeme a znaku přiřadíme výsledné znaménko. Pokud úspěšní nejsme, pak předpokládáme, že znak nemá diakritické znaménko. Díky tomu dojde k významnému zmenšení množin se vzorovými znaky u všech algoritmů. K detekci diakritiky lze použít i některý ze sofistikovanějších přístupů (např. [10]), ovšem pro potřeby specifikované zadáním (tiskací abeceda) je uvedený přístup plně dostačující. Jinak by tomu bylo například u arabského písma.

Zpracování znaménka V této práci se berou v potaz pouze tři diakritická znaménka a to háček, čárka a kroužek. Pro určení, o které z nich se v daném obrázku jedná, nám stačí udělat na obrázku znaménka tři vertikální řezy (vlevo, vpravo a uprostřed) a porovnat, kde se na jednotlivých řezech vyskytuje první platný pixel. Pro háček budou pixely na krajních řezech výše než na prostředním

řezu. U čárky bude pozice pixelů postupně stoupat a u kroužku budou pixely na krajních řezech níže než pixel na prostředním řezu (Obrázek 3.).

Obrázek 3. Určení diakritických znamének pomocí řezů. Červeně jsou pixely, jejichž souřadnice porovnáváme.

Úspěšnost nalezení a určení znaménka Na testovacích datech byla zaznamenána 81% úspěšnost nalezení a určení znaménka. Většina chyb byla způsobena už samotnými zdrojovými daty, kdy háček vypadal jako čárka. Jedinou možností, jak takový typ chyby opravit, je dodatečná kontrola textu (4.1.).

3. Algoritmy

Tato kapitola obsahuje nejdůležitější část práce a to tři algoritmy používající k eliminaci nepřesností ve znacích fuzzy logiku. První dva jsou nastudovány a implementovány podle prací jiných autorů, třetí je pak algoritmus dle vlastního návrhu.

Metodika testování Pro testování úspěšnosti algoritmů bylo použito pět různých textů napsaných v různých dnech a za různých podmínek, aby výsledky nebyly závislé na mém momentálním rozpoložení. Jediná úprava provedená na vstupních obrázcích je jejich ořezání na menší velikost. Všechny tyto soubory jsou pak přiloženy u demonstrační aplikace pro možnost ověření výsledků. Dále do úspěšnosti jednotlivých algoritmů nebyla započtena úspěšnost diakritiky, neboť tato je zpracována ještě před samotným spuštěním algoritmu, a tedy už jeho během nemůže být nijak ovlivněna.

3.1. MMS

První z implementovaných algoritmů pochází z práce [11]. Algoritmus nejprve vstupní znak rozdělí na jednotlivé segmenty. Pro ty následně určí charakteristické vlastnosti, získané hodnoty fuzzifikuje a celek pak porovná se všemi znaky v databázi. Výsledná hodnota znaku pak odpovídá nejlepší shodě. Fáze algoritmu jsou tedy následující: získání vstupu, segmentace, fuzzifikace charakteristických vlastností segmentů, porovnání s databází a určení hodnoty znaku.

Jméno algoritmu Algoritmus v citované práci nebyl nijak pojmenován, zde použité označení je zkratka ze jmen autorů: Mahashukhon, Mousavinezhad a Song.

3.1.1. Požadavky na vstup

Vstupem je obrázek s následujícími vlastnostmi:

1. Černobílý - bílé pixely (1) jsou objekty v obrázku, černé (0) jsou pozadí.
2. Obrázek obsahuje kostru právě jednoho znaku k rozpoznání.
3. Obrázek má normovanou velikost
4. Znak na obrázku neobsahuje diakritická znaménka

Příkladem může být obrázek 4.

Obrázek 4. Platný vstup - kostra znaku B

3.1.2. Segmentace

Pro určení hodnoty znaku je nutné jej rozdělit na jednotlivé segmenty. Segment je část znaku, která je oddělená od ostatních částí význačnými body. Každý segment je určen typem, vertikální pozicí, horizontální pozicí a délkou.

Typy segmentů

1. UD - vertikální čára
2. LR - horizontální čára
3. RI - čára stoupající zleva doprava
4. LI - čára klesající zleva doprava
5. D - bod
6. C - cyklický segment

Význačné body

1. Osamocený bod - bod, který nemá žádného souseda
2. Koncový bod - bod, který má právě jednoho souseda
3. Spojení dvou čar - bod, který má právě tři sousedy
4. Křížení dvou čar - bod, který má právě čtyři sousedy
5. Změna směru - bod ve kterém se segment zatočí mimo dosud stanovený směr

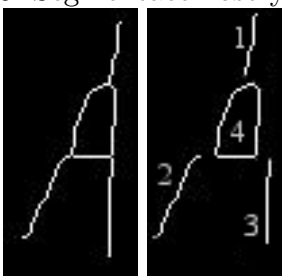
Bod, který má právě dva sousedy je obyčejným bodem segmentu a žádný bod nemůže mít více než čtyři sousedy (z definice kostry znaku).

Postup segmentace: Obrázek procházíme pixel po pixelu, dokud nenarazíme na koncový bod nějakého segmentu, osamocený bod, nebo dokud není dosaženo konce obrázku. Pokud jsme nalezli osamocený bod, tak jej z obrázku odstraníme a vytvoříme nový segment typu bod (D). Pokud jsme nalezli koncový bod segmentu, tak segment extrahujeme, což znamená, že odstraníme bod a přesuneme se na jeho souseda a takto pokračujeme, až dokud nenarazíme na další význačný bod, kde extrakci ukončíme. Výsledný segment uložíme k již nalezeným a pokračujeme hledáním dalšího segmentu (zase od počátku). Pokud dojdeme až na poslední (pravý dolní) pixel obrázku bez nalezení nějakého segmentu, tak přejdeme k hledání cyklických segmentů.

Hledání cyklických segmentů probíhá velmi podobně. Rozdílem je, že nehledáme jen koncové nebo osamocené body, ale stačí nám libovolný bod objektu v popředí. Jakmile na takový narazíme, tak opět přecházíme ze souseda na souseda (již zpracované body mažeme) a to až do okamžiku, kdy se dostaneme k bodu, který je sousedem počátečního bodu segmentu. Nyní segment zpracujeme a opět zkusíme hledat necyklické segmenty. Může se totiž stát, že po odstranění cyklického segmentu uvolníme konec nějakého dalšího segmentu, který by při použití postupu pro cyklické segmenty byl špatně zpracován. Z tohoto důvodu jsou necyklické segmenty extrahovány vždy v maximálním možném množství a cyklické vždy jen po jednom. U cyklických segmentů je navíc nutno ošetřit případ, kdy dva cyklické segmenty mají společnou část.

Celý postup provádíme až do okamžiku, kdy v obrázku není ani jeden pixel patřící k objektu v popředí.

Obrázek 5. Segmentace kostry znaku A



Z důvodu, že segmentace byla v citované práci popsána jen velmi stručně, jsou dále uvedeny i pseudokódy nejdůležitějších funkcí tak, jak jsou implementovány v této práci (segmentace kostry (Pseudokód 2), extrakce necyklického segmentu (Pseudokód 3) a extrakce cyklického segmentu (Pseudokód 4)).

Pseudokód 2 Segmentace znaku

```
Segmentace-Kostry-Znaku(obrazek-znaku)
1.   segmenty <- vytvor-seznam()
2.   while neni-prazdny(obrazek-znaku)
3.      segment-nalezen <- true
4.      while segment-nalezen
5.         segment-nalezen <- false
6.         foreach p in obrazek-znaku.Pixely
7.            if p is osamoceny bod or p is koncovy bod
8.               segmenty.pridej(
                        extrakce-necyklickeho-segmentu(obrazek-znaku, p))
9.               segment-nalezen <-true
10.              break
11.           endif
12.        endforeach
13.     endwhile
14.     segment-nalezen <- false
15.     foreach p in obrazek-znaku.Pixely
16.        if p is bod-popredi
17.           segmenty.pridej(
                       extrakce-cyklickeho-segmentu(obrazek-znaku, p))
18.           segment-nalezen <-true
19.           break
20.        endif
21.     endforeach
22.  endwhile
23.  return segmenty
end
```

Pseudokód 3 Extrakce necyklického segmentu

```
Extrakce-necyklickeho-segmentu(obrazek-znaku, pocatek-segmentu)
1.   segment.body <- vytvor-seznam()
2.   segment.body.pridej(pocatek-segmentu)
3.   odtsran-z-obrazku(obrazek-znaku, pocatek-segmentu)
4.   if je-prazdne(pocatek-segmentu.sousede)
5.      return zpracuj-segment(segment)
6.   endif
7.   aktualni-bod <- vyber-souseda(pocatek-segmentu.sousede)
8.   while neni-vyznacny(aktualni-bod)
9.      segment.body.pridej(aktualni-bod)
10.     odstran-z-obrazku(obrazek-znaku, aktualni-bod)
11.     aktualni-bod <- vyber-souseda(aktualni-bod)
12.  endwhile
13.  return zpracuj-segment(segment)
end
```

Pseudokód 4 Extrakce cyklického segmentu

```
Extrakce-cyklickeho-segmentu(obrazek-znaku, pocatek-segmentu)
1.   segment.body <- vytvor-seznam()
2.   aktualni-bod <- pocatek-segmentu
3.   spolecna-cara <- false
4.   while neni-vyznacny(aktualni-bod) or je-spojeni-car(aktualni-bod)
5.      segment.body.pridej(aktualni-bod)
6.      if je-spojeni-car(atkualni-bod)
7.         spolecna-cara <- not(spolecna-cara)
8.      endif
9.      if not(spolecna-cara)
10.        odstran-z-obrazku(obrazek-znaku, aktualni-bod)
11.     endif
12.     aktualni-bod <- vyber-souseda(aktualni-bod)
13.  endwhile
14.  return zpracuj-segment(segment)
end
```

3.1.3. Zpracování a fuzzifikace vlastností segmentu

Nyní máme k dispozici všechny segmenty znaku jako seznam bodů. Pro přehlednost dále předpokládejme, že máme délku, počáteční, koncový a prostřední

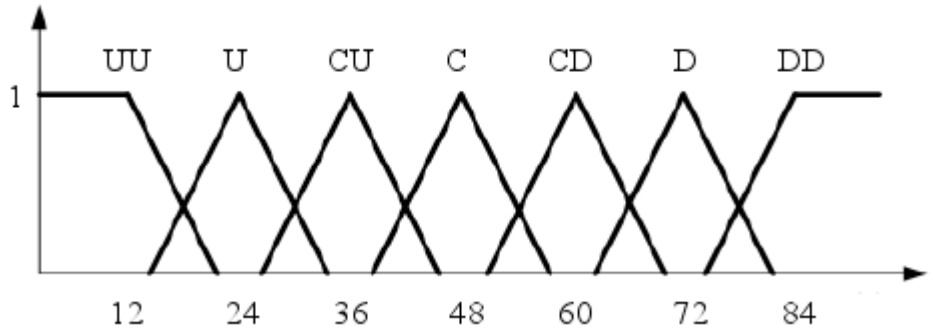
bod segmentu dostupné i mimo tento seznam. Teď musíme určit všechny potřebné vlastnosti segmentu: typ, vertikální pozici, horizontální pozici a délku.

Typ segmentu Typ segmentu určíme dle následujících kritérií.

1. UD - segment, který začíná a končí ve stejném sloupci
2. LR - segment, který začíná a končí ve stejném řádku
3. RI - segment, který začíná (končí) výše a více vpravo než končí (začíná)
4. LI - segment, který začíná (končí) výše a více vlevo než končí (začíná)
5. D - segment s délkou jedna
6. C - segment, který začíná a končí na stejném místě a je delší než jedna

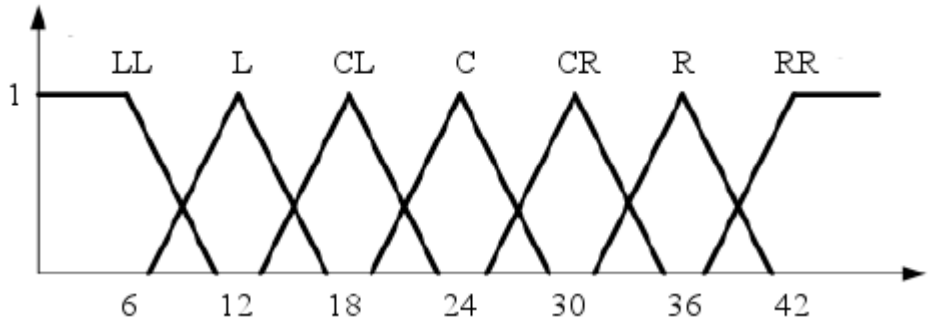
Vertikální pozice segmentu Pro vertikální pozici segmentu je definováno sedm fuzzy množin (Obrázek 6.). Významy jejich názvů jsou následující: UU pro velmi nahoře, U pro nahoře, CU pro spíše nahoře, C pro uprostřed, CD pro spíše dole, D pro dole a DD pro velmi dole. Hodnotu pro fuzzifikaci získáme jako průměr vertikálních souřadnic počátečního, prostředního a koncového bodu segmentu. Výsledná vertikální pozice segmentu bude rovna jménu fuzzy množiny s největším stupněm příslušnosti. V případě rovnosti stupňů příslušnosti u dvou fuzzy množin je vybrána libovolná z nich.

Obrázek 6. Fuzzy množiny pro vertikální pozici segmentu



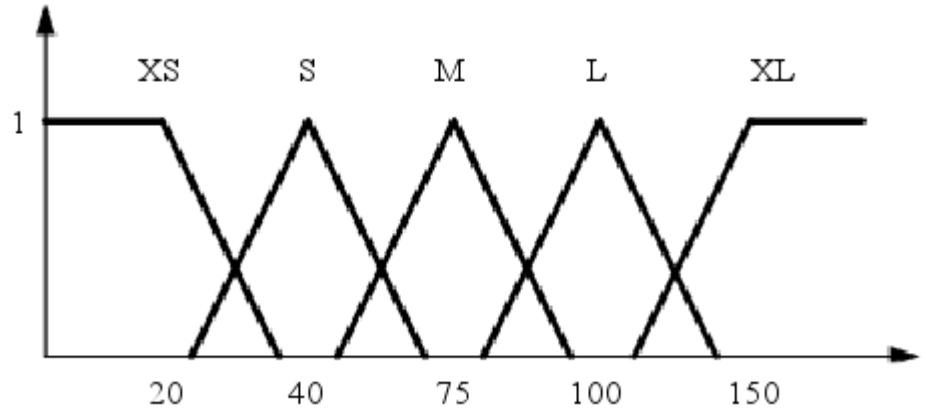
Horizontální pozice segmentu Pro horizontální pozici segmentu je definováno opět sedm fuzzy množin (Obrázek 7.). Významy jejich názvů jsou následující: LL pro velmi vlevo, L pro vlevo, CL pro spíše vlevo, C pro uprostřed, CR pro spíše vpravo, R pro vpravo a RR pro velmi vpravo. Hodnotu pro fuzzifikaci získáme jako průměr horizontálních souřadnic počátečního, prostředního a koncového bodu segmentu. Výsledná horizontální pozice segmentu bude rovna jménu fuzzy množiny s největším stupněm příslušnosti. V případě rovnosti stupňů příslušnosti u dvou fuzzy množin je vybrána libovolná z nich.

Obrázek 7. Fuzzy množiny pro horizontální pozici segmentu



Délka segmentu Pro délku segmentu je definováno pět fuzzy množin (Obrázek 8.). Významy jejich názvů jsou následující: XS pro velmi malá, S pro malá, M pro střední, L pro velká a XL pro velmi velká. Hodnotou pro fuzzifikaci je počet bodů segmentu. Výsledná délka segmentu je jméno fuzzy množiny s největším stupněm příslušnosti. V případě rovnosti stupňů příslušnosti u dvou fuzzy množin je vybrána libovolná z nich.

Obrázek 8. Fuzzy množiny pro délku segmentu



3.1.4. Určení hodnoty znaku

Když máme fuzzifikovány všechny segmenty, tak můžeme přistoupit k samotné identifikaci znaku. Pro porovnání dvou znaků je potřeba porovnat jejich segmenty. Bereme tedy jeden segment neidentifikovaného znaku za druhým a postupně k nim vyhledáváme nejlepší shody mezi segmenty znaku z databáze (každý segment znaku v databázi může být přiřazen nejvýše jednomu segmentu neidentifikovaného znaku). Kvalita shody mezi dvěma segmenty je určena následujícími vztahy:

$$
\begin{aligned}
D_t &= compare(type_{DB}, type_{UN}) \\
D_y &= compare(hPos_{DB}, hPos_{UN}) \\
D_x &= compare(vPos_{DB}, vPos_{UN}) \\
D_l &= compare(length_{DB}, length_{UN}) \\
D &= (D_t + D_y + D_x + D_l)/4
\end{aligned}
$$

Kde D_t je stupeň podobnosti typů segmentů, D_y je stupeň podobnosti horizontálních pozic segmentů, D_x je stupeň podobnosti vertikálních pozic segmentů, D_l je stupeň podobnosti délek segmentů a D je výsledný stupeň podobnosti segmentů. Jednotlivé argumenty jsou pak *type* pro typ, $hPos$ pro horizontální pozici, $vPos$ pro vertikální pozici a *length* pro délku, kde dolní index DB značí segment znaku z databáze vzorů a UN značí segment zatím neidentifikovaného znaku.

Nyní spočítáme stupeň podobnosti znaků jako průměrnou podobnost jejich segmentů:

$$D_{DB}^{UN} = \sum_{i=1}^{n_{UN}} D_i / max(n_{DB}, n_{UN})$$

Kde n_{UN} je počet segmentů určovaného znaku, n_{DB} počet segmentů vzorového znaku a D_i stupeň podobnosti i-tého segmentu určovaného znaku s přiřazeným segmentem vzorového znaku.

Tímto způsobem znak porovnáme se všemi vzory v databázi a jako výslednou hodnotu určíme hodnotu vzoru s nejvyšším stupněm podobnosti.

Poznámka Myšlenka porovnávání pochází z citované práce, ale byly v ní provedeny některé úpravy. Konkrétně bylo přidáno i porovnávání typů segmentů, neboť může nastat případ, kdy si jsou pozicemi a délkou rovny i segmenty různých typů, což by vedlo k možnému spárování i velmi odlišných znaků. Další změnou je porovnávání pozic počátečního a koncového bodu segmentu u všech segmentů kromě osamocených bodů a cyklických segmentů. Toto pomůže odlišit znaky, které v nejhorším případě mohou mít naprosto stejné složení segmentů (například C a G při příliš velkém úhlu u zlomu v G). Porovnávání pozic bodů je stejné jako porovnávání pozic segmentů a do výsledného stupně podobnosti segmentů se započítává opět průměrováním.

3.1.5. Úspěšnost algoritmu

Dle citované práce (část "IV. Simulations") by měl algoritmus mít úspěšnost přes 90% (testováno na 100 vzorcích). Mně se takto vysoké úspěšnosti dosáhnout nepodařilo. Jedním z důvodů by mohl být fakt, že autoři práce pro zjednodušení předpokládali na vstupu bezchybnou kostru znaku a v mé implementaci jsou největším problémem právě různé artefakty vzniklé při digitalizaci naskenovaného textu a následném získávání koster jednotlivých znaků.
Úspěšnost dosažená v implementaci je popsána v tabulce 1..

Tabulka 1. Úspěšnost implementace algoritmu MMS. Ve výsledku není započtená úspěšnost diakritiky, neboť tu už algoritmus nemůže ovlivnit.

Pořadí testu	Počet písmen	Počet rozpoznaných	Počet chyb	%
1	146	110	36	75.3
2	160	122	38	76.3
3	195	143	52	73.3
4	71	52	19	73.2
5	186	131	55	70.5
Celkem	758	558	200	73.6

Výsledky tedy ukazují, že na reálných datech se úspěšnost pohybuje v rozsahu 70 - 77%, což je na hranici čitelnosti.

3.1.6. Vylepšení algoritmu

V rámci snahy o zlepšení úspěšnosti algoritmu jsem se pokusil navrhnout a implementovat různé úpravy. Zde tedy jsou ty nejzajímavější a stručný popis jejich dopadu na kvalitu algoritmu.

Další typy segmentů První změnou bylo dodefinování dalších typů segmentů pro různé oblouky (R/LUD pro vertikální oblouk vyklenutý doprava/doleva, U/DLR pro horizontální oblouk vyklenutý nahoru/dolů a podobně i U/DRI a U/DLI pro oblouk odvozené od šikmých čar). Toto sice vedlo ke snížení omylů u dvojic znaků jako je L a C nebo O a D, ale na druhou stranu to zhoršilo celkovou úspěšnost algoritmu, neboť se velmi zvýšily počty možných složení jednotlivých znaků. Tato úprava tedy byla odstraněna.

Kostra znaku U tohoto algoritmu je určení znaku velmi závislé na dodané kostře znaku, takže při jeho implementaci došlo k četným úpravám algoritmů používaných během předzpracování vstupu, zejména algoritmu pro vytváření kostry znaku. (více v 2.3.) Během testování a úprav těchto algoritmů se ukázalo, že právě získání kvalitní kostry pro libovolný vstup je asi nejnáročnější a zároveň nejkritičtější místo algoritmu MMS. Náročnost je dána velkým množstvím chyb v obrázcích ručně psaných znaků (nejednotnost stopy psacího nástroje, nedotažené segmenty, občasné inkoustové fleky okolo znaku, atd.).

Porovnávání fuzzifikovaných hodnot Další úspěšnou úpravou je přidání stupňů podobnosti pro různé hodnoty jednotlivých fuzzifikovaných vlastností segmentů. I přes to, že tím dojde k dalšímu zvýšení tolerance při porovnávání, se zlepšila celková úspěšnost algoritmu, a tedy bylo tato změna v implementaci ponechána. Více informací k porovnávání jednotlivých hodnot je v poznámkách k implementaci (3.1.7.).

Množina vzorových znaků Jisté zlepšení také přináší zvětšování množiny vzorových znaků, ale s každým novým vzorem se zvýší i čas potřebný k identifikování znaku. Je tedy důležité množinu vzorů udržovat co nejmenší. Zároveň je nutné velice pečlivě vybírat, které znaky budou zařazeny mezi vzory. V extrémním případě, kdy jsou v množině vzorů od každého znaku jen dva nebo tři kusy, může jeden špatný vzor zamezit rozpoznání několika znaků. Příkladem může být M, které během testování v několika případech při špatné detekci hranic znaku nahradilo I a při nevhodně nastaveném předzpracování jeho kostra splývala s kostrou znaku H.

3.1.7. Poznámky k implementaci

Podobnost segmentů Jak již bylo zmíněno v poznámce 3.1.4., bylo upraveno počítání podobnosti segmentů.

Souřadnicový systém V implementaci je při určování pozic použit jako počátek souřadnicového systému levý horní roh obrázku. Pro jiný počátek je nutné změnit definice fuzzy množin pro horizontální a vertikální pozici.

Uchovávání vzorů Pro uložení databáze vzorů je použit XML soubor. Jeho struktura a další informace jsou obsaženy v programátorské dokumentaci (5.3.).

Tolerance sklonu segmentů U segmentů typu UD (vertikální čára) a LR (horizontální čára) je tolerance posunutí několika řádku/sloupců, neboť je nepravděpodobné, že takový segment bude přesně v jednom sloupci (řádku). Důvodem jsou nepřesnosti při ručním psaní, skenování dokumentu a předzpracování obrázku.

Podklady k porovnávání segmentů Následují tabulky se stupni podobnosti mezi hodnotami vlastností segmentů (typ v tabulce 2., pozice v tabulce 3. a délka v tabulce 4.).

Tabulka 2. Stupně podobnosti typů segmentů

	UD	LR	RI	LI	C	D
UD	1	0	0.5	0.5	0	0
LR	0	1	0.5	0.5	0	0
RI	0.5	0.5	1	0	0	0
LI	0.5	0.5	0	1	0	0
C	0	0	0	0	1	0
D	0	0	0	0	0	1

Tabulka 3. Stupně podobnosti fuzzifikovaných horizontálních pozic, pro vertikální pozice jsou hodnoty obdobné.

	LL	L	C	R	RR
LL	1	0.8	0.3	0	0
L	0.8	1	0.8	0.3	0
C	0.3	0.8	1	0.8	0.3
R	0	0.3	0.8	1	0.8
RR	0	0	0.3	0.8	1

Tabulka 4. Stupně podobnosti fuzzifikovaných délek

	XS	S	M	L	XL
XS	1	0.8	0.3	0	0
S	0.8	1	0.8	0.3	0
M	0.3	0.8	1	0.8	0.3
L	0	0.3	0.8	1	0.8
XL	0	0	0.3	0.8	1

3.2. Algoritmus fuzzy skeneru

Druhým zvoleným algoritmem je algoritmus, na jehož principu pracují některé fuzzy skenery[3]. Naneštěstí se po implementování algoritmu ukázalo, že pro účely této práce je naprosto nevhodný, neboť není schopen zpracovat větší odchylky v jednotlivých znacích. Ty jsou ovšem pro ruční psaní typické. Při dohledávání dalších informací pro případné zlepšení jeho úspěšnosti jsem narazil na původní článek s jeho popisem [12], ze kterého plyne, že algoritmus byl navržen pouze pro rozpoznávání tištěných znaků a to hlavně číslic. Fuzzy logika zde hraje roli jen jako prostředek pro vypořádání se s drobným pootočením či posunutím znaku. Algoritmus jsem z práce chtěl úplně odstranit, ale nakonec jsem se rozhodl, že pouze výrazně redukuji prostor, který mu je věnován, neboť i když pro naše účely není použitelný, tak je dobrou ukázkou dalšího možného přístupu k zakomponování fuzzy logiky do rozpoznávání textu. V další části tedy jen velmi stručně popíšu princip algoritmu a způsob využití fuzzy logiky, kvůli kterému zde byl ponechán.

3.2.1. Princip algoritmu

Na vstupu je očekáván obrázek znaku s normovanou velikostí (tištěný znak). Následně jsou provedeny vertikální řezy obrázkem. Kolik sloupců má obrázek, tolik je řezů. V každém z řezů je spočítán počet platných pixelů a výsledky jsou uloženy do tabulky. Nyní je potřeba určit, jak se znak v jednotlivých řezech mění, neboť pravě to je veličina, kterou budeme fuzzifikovat. Nejprve určíme směr změny porovnáním hodnoty v aktuálním řezu s hodnotou z předchozího řezu. Je-li aktuální větší, pak směr je rostoucí, je-li menší, pak směr je klesající a jsou-li si rovny, pak je zachován směr předchozí dvojice. Dale musíme určit lokální minima a maxima v oblastech se souvislým směrem. Nakonec dle lokálních minim a maxim v oblastech před přechody z jednoho směru na druhý určíme výsledné hodnoty pro fuzzifikaci dle vzorců:

$$X_i = l_{max}^{i-1} - l_{min}^{i-1} \quad (1)$$

$$X_i = l_{min}^{i-1} - l_{max}^{i-1} \quad (2)$$

Kde i označuje i-tý řez, vzorec (1) je pro přechod v i-tém řezu z kladného směru na záporný a vzorec (2) pro přechod v i-tém řezu ze záporného směru na kladný. Celý tento průběh pro znak 0 (jeho řezy jsou v tabulce 5.) je ukázán v tabulce 6.

Tabulka 5. Řezy pro znak 0. Celá tabulka převzata z [3].

Řez	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hodnota	0	12	16	4	2	2	2	2	2	4	16	12	0	0	0

Tabulka 6. Počítání hodnot přechodů P pro znak 0.

Řez	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Aktuální	0	12	16	4	2	2	2	2	2	4	16	12	0	0	0
Předchozí	0	0	12	16	4	2	2	2	2	2	4	16	12	0	0
Směr	+	+	+	-	-	-	-	-	-	+	+	-	-	-	-
Přechody				P1						P2		P3			P4
l^i_{min}	0	0	0	4	2	2	2	2	2	2	2	12	0	0	0
l^i_{max}	0	12	16	16	16	16	16	16	16	4	16	16	16	16	0
P hodnota				16						-14		14			-16

Nyní máme hodnoty pro jednu z možných pozic znaku 0 v obrázku. Aplikováním stejného postupu spočítáme hodnoty i pro několik různých pootočených a posunutých verzí téhož znaku. Takto dostaneme například hodnoty $P1 = 16$, $P2 = -14$, $P3 = 14$ a $P4 = 16$ pro znak pouze posunutý (to odpovídá myšlence, že stejný znak se bude vyvíjet stejně na všech pozicích) nebo hodnoty $P1 = 17$, $P2 = -14$, $P3 = 16$ a $P4 = -16$ pro znak pootočený dle svého středu o několik stupňů doleva. Při zpracování dostatečného množství verzí znaku tedy můžeme určit rozsahy jednotlivých přechodů P. Zpracování více různých verzí téhož znaku je zde stěžejní (více v 3.2.2.). Pro znak 0 jsou tyto rozsahy v tabulce 7.

Tabulka 7. Rozsahy přechodů pro znak 0. Celá tabulka převzata z [3].

Znak 0	P_1	P_2	P_3	P_4
min	16	-15	12	-19
max	19	-12	15	-16

Pro zamezení chybného určení u znaků s podobnými hodnotami P přechodů (0 a 8, 1 a 3, ...) určíme ještě celkový počet pixelů znaku (označme SP) a šířku znaku (označme W).

Nyní potřebujeme všechny naměřené hodnoty fuzzifikovat. Protože funkce příslušnosti hodnot přechodů P do jednotlivých fuzzy množin na užitém principu nic nezmění, tak je u tohoto algoritmu z výše popsaných důvodů nebudu popisovat. Všechny je lze najít v primárním zdroji ([12]).

3.2.2. Určování hodnoty znaku

U tohoto algoritmu místo množiny vzorových znaků používáme jen jedno pravidlo pro každý znak. Toto pravidlo je určeno právě na základě fuzzifikace hodnot, které jsme získávali pomocí tabulek 6. a 7. Proto bylo v části 3.2.1. důležité, abychom hodnoty určili pro různě posunuté a pootočené znaky (bez toho bychom nebyli schopni takto znehodnocené znaky přiřadit k danému pravidlu). Následují příklady takových pravidel pro některé znaky.

```
if P1 is VLP and P2 is LN and P3 is VLP
and P4 is LN and  SP is L and W is L
then char is '0';

if P1 is VLP and P2 is LN and SP is M
and W is S
then char is '1';

if P1 is P and P2 is N and P3 is P
and P4 is N and SP is S and W is S
then char is '2';
```

Kde VLN znamená velmi velký negativní přechod, LN velký negativní přechod, N negativní přechod, P pozitivní přechod, LP velký pozitivní přechod, VLP velmi velký pozitivní přechod, S malá velikost, M střední velikost a L velká velikost.

Určení hodnoty znaku se pak tedy sestává z vypočítání hodnot přechodů P a vlastností SP a W dle výše uvedeného principu a jejich následné porovnání s jednotlivými pravidly. Pokud je nalezena shoda, pak je znak identifikován.

3.2.3. Shrnutí poznatků

Fuzzy logiku tedy lze při rozpoznávání využít i značně odlišným způsobem než je použit v algoritmu MMS. Výhoda vytvoření pouze jednoho pravidla na znak, tedy jakéhosi supervzoru, je ve velikosti a údržbě množiny pravidel, neboť ji stačí vytvořit pouze jednou a po dostatečné optimalizaci už v ní nebudou potřeba žádné další úpravy. Nevýhoda je, že tato pravidla jsou schopna přiřadit správné hodnoty jen ke znakům pokrytým rozsahem odlišnosti jednotlivých vzorů použitých při jejich tvorbě. Právě tento fakt se nakonec ukázal jako důvod, proč je uvedený algoritmus nepoužitelný na ručně psaný text.

Dalším omezením algoritmu, které je dáno typem a množstvím zkoumaných vlastností znaku, je velikost množiny rozpoznatelných znaků. Při testování na abecedě docházelo k velkému množství záměn jednoho znaku za jiný, neboť písmena pokrývají jen velmi malou část všech možných kombinací hodnot. Toto

ovšem není chyba algoritmu, ale jeho vlastnost, neboť, jak je uvedeno v primárním zdroji, byl algoritmus navržen pouze pro rozpoznávání tištěných číslic a čtyř dalších znaků.

3.3. VHE

Posledním bodem zadání a zároveň posledním algoritmem je algoritmus dle vlastního návrhu. Po zkušenostech s implementací algoritmu MMS jsem se snažil navrhnout přístup, který by byl založen na jiném principu než rozdělování znaku na jednotlivé části (segmenty, významné body, úhly mezi jednotlivými čárami). Místo toho jsem se pokusil napodobit způsob určování znaků, který používám při čtení já. Zde je potřeba říct, že člověk při čtení textu nevnímá všechna písmena se stejnou váhou[13], a proto jsem při analýze náhodně vybíral jednotlivá písmena a snažil se určit, dle čeho je rozeznávám. Tímto způsobem jsem došel k tomu, že je pro mě nejdůležitější celkový tvar písmene. To znamená, že mnohem důležitější než sklon a délka jednotlivých částí je, jak velké množství prostoru zabírá obrys písmene v jednotlivých částech pomyslného obdélníku kolem písmene.

Nyní bylo potřeba vytvořit obal písmene, který by ale nezakryl důležité rysy. První volbou byl konvexní obal, ale ten jsem záhy zavrhnul, neboť je několik skupin písmen s téměř stejným konvexním obalem. Například M,H,K,X,N,Z a E nebo O,D,C a G. Nakonec jsem se rozhodl udělat obalů více (ne nutně konvexních) a to takových, aby každý z nich obsahoval důležité vlastnosti z jednoho úhlu pohledu na písmeno. V jednotlivých obalech se pak porovnává velikost objektů v předem určených oblastech. Fáze algoritmu tedy budou následující: Získání vstupního obrázku znaku, vytvoření všech potřebných obalů, zjištění a fuzzifikace velikostí objektů v jednotlivých oblastech obalů, porovnání určených hodnot s databází vzorů a určení výsledné hodnoty znaku.

Poznámka Obsahem předchozích odstavců není tvrzení, že lidé právě takto určují písmena. Je to pouhý popis postupů, které jsem při určování písmen vypozoroval sám na sobě.

Název algoritmu Z výše popsaného postupu plyne i význam zkratky v nadpisu kapitoly VHE pro Vertical and Horizontal Envelopes.

3.3.1. Požadavky na vstup

Vstupem je obrázek s následujícími vlastnostmi:

1. Černobílý - bílé pixely (1) jsou objekty v obrázku, černé (0) jsou pozadí.
2. Obrázek obsahuje právě jeden znak k rozpoznání.
3. Znak na obrázku neobsahuje diakritická znaménka.

4. Obrázek by měl být normován na velikost vzorů. (nepovinné)

Normovaná velikost obrázku zde není nezbytně nutná, avšak musí být zajištěno, že znak bude v obrázku vždy na přibližně stejné pozici a jeho velikost bude vůči velikosti obrázku mít vždy přibližně stejný poměr. Toho nejsnáze docílíme ořezáním a následným normováním velikosti. Příkladem vstupu je obrázek 9.

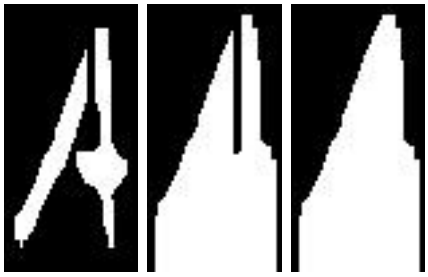
Obrázek 9. Platný vstup - znak A

3.3.2. Vytvoření obalů

Jak již bylo zmíněno v úvodu k tomuto algoritmu, je potřeba vytvořit obaly z několika úhlů pohledu na znak. Obaly budou čtyři a to pro pohled zleva, zprava, zespodu a shora.

Horní obal Horní obal vytváříme postupným procházením jednotlivých sloupců obrázku shora dolů. Dokud jsme nenarazili na pixel nějakého objektu, tak všechny doposud navštívené pixely sloupce zůstanou prázdné. Poté, co se dostaneme na první pixel objektu, zaplníme i zbylé pixely ve sloupci. Po zpracování celého obrázku je vhodné celý obal ještě jednou projít a uzavřít případné rýhy v obalu vzniklé chybějícím pixelem v obrázku znaku. Na obrázcích 10. vidíme zleva doprava: vstupní znak s chybějícími pixely, horní obal, který z takového znaku vznikne včetně rýhy po chybějících pixelech a výsledný horní obal.

Obrázek 10. Vstup s chybějícími pixely, přenesení chyby do obalu a opravený horní obal.



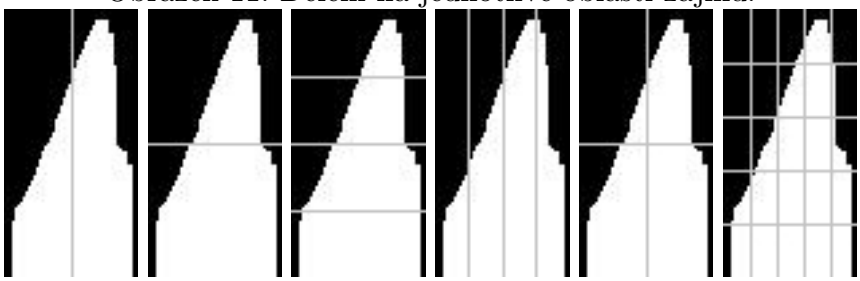
Ostatní obaly Obaly pro ostatní pohledy (zleva, zprava a zespodu) se vytváří obdobným způsobem jako horní obal, jen se prochází řádky zprava doleva (pohled zprava), řádky zleva doprava (pohled zleva) a sloupce zdola nahoru (pohled zespodu).

3.3.3. Získání a fuzzifikace vlastností obalů

Jako charakteristické vlastnosti obalů použijeme procentuální zaplnění 41 různých oblastí. A to:

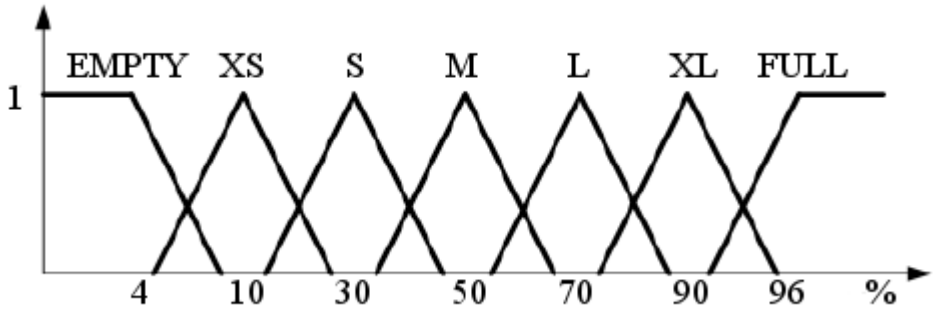
1. Pravá, levá, horní a dolní polovina obrázku (Obrázek 11. první a druhý zleva)
2. Čtvrtiny ve vertikálním směru (Obrázek 11. třetí zleva)
3. Čtvrtiny v horizontálním směru (Obrázek 11. čtvrtý zleva)
4. Čtvrtiny při dělení do mřížky 2x2 (Obrázek 11. předposlední)
5. Pětadvacetiny při dělení do mřížky 5x5 (Obrázek 11. poslední)

Obrázek 11. Dělení na jednotlivé oblasti zájmu.



Fuzzifikace Hodnotou pro fuzzifikaci každé oblasti je procentuální vyjádření jejího zaplnění zaokrouhlené na jednotky. Použité fuzzy množiny jsou na obrázku 12. Výslednou hodnotou je jméno fuzzy množiny, pro kterou má získaná hodnota největší stupeň příslušnosti. Má-li získaná hodnota stejný stupeň příslušnosti pro dvě množiny, pak se vybere libovolná z nich. Právě díky využití procentuálního vyjádření zaplnění, není podmínka normalizované velikosti vstupu stěžejní (více v 3.3.1.).

Obrázek 12. Fuzzy množiny pro zaplnění oblasti



3.3.4. Určení hodnoty znaku

Nyní, když máme k dispozici hodnoty zaplnění pro všechny oblasti, můžeme přistoupit k porovnávání znaku s databází vzorů. Při každém porovnání dvou oblastí jsou použity vztahy z tabulky 8.

Tabulka 8. Stupně podobnosti fuzzifikovaných zaplnění oblastí

	EMPTY	XS	S	M	L	XL	FULL
EMPTY	1	0.8	0.3	0	0	0	0
XS	0.8	1	0.8	0.3	0	0	0
S	0	0.8	1	0.8	0.3	0	0
M	0	0.3	0.8	1	0.8	0.3	0
L	0	0	0.3	0.8	1	0.8	0.3
XL	0	0	0	0.3	0.8	1	0.8
FULL	0	0	0	0	0.3	0.8	1

Vždy porovnáváme dvě odpovídající si oblasti. Nejvyšší váhu má podobnost na úrovni polovin, poloviční váhu poté podobnost jednotlivých čtvrtin a čtvrtinovou podobnost příslušných pětin. Celkovou podobnost dvou obalů tedy spočítáme dle následujícího vzorce:

$$D_{E1}^{E2} = (4 * D_{half}^{avg} + 2 * D_{quarter}^{avg} + D_{fifth}^{avg})/5$$

Kde D_{E1}^{E2} je celková podobnost obalů, D_{half}^{avg} je průměrná podobnost polovin, $D_{quarter}^{avg}$ je průměrná podobnost čtvrtin a D_{fifth}^{avg} je průměrná podobnost pětin.

Nyní zbývá už jen určení hodnoty znaku v závislosti na podobnostech jednotivých obalů. Opět použijeme průměr, tedy:

$$D_{CH1}^{CH2} = (D_{CH1_L}^{CH2_L} + D_{CH1_R}^{CH2_R} + D_{CH1_B}^{CH2_B} + D_{CH1_T}^{CH2_T})/4$$

Kde D_{CH1}^{CH2} je celková podobnost znaků, $D_{CH1_*}^{CH2_*}$ je podobnost levých (L), pravých (R), spodních (B) a horních (T) obalů.

Výslednou hodnotou určovaného znaku je hodnota vzoru s největší celkovou podobností s určovaným znakem.

3.3.5. Úspěšnost algoritmu

Úspěšnost dosažená v implementaci je popsána v tabulce 9.

Tabulka 9. Úspěšnost implementace algoritmu VHE. Ve výsledku není započtená úspěšnost diakritiky, neboť tu už algoritmus nemůže ovlivnit.

Pořadí testu	Počet písmen	Počet rozpoznaných	Počet chyb	%
1	146	133	13	91.1
2	160	140	20	87.5
3	195	168	27	86.1
4	71	64	7	90.1
5	186	154	32	82.8
Celkem	758	659	99	86.9

Výsledky tedy ukazují, že na reálných datech se úspěšnost algoritmu pohybuje v rozsahu 82 - 92 %, kdy celková úspěšnost je 86.9 %. Tedy výstup už je čitelný a při použití některé z metod dodatečných úprav textu (4.1.) je algoritmus reálně použitelný.

3.3.6. Možná vylepšení

Velikost význačných oblastí Během implementace algoritmu bylo provedeno testování i s jinými velikostmi zkoumaných oblastí. Poloviny a čtvrtiny jsou použity vždy, neboť ty rozdělují znaky na skupiny dle nejvýraznějších rysů. Experimentováno tedy bylo s nejmenšími oblastmi, které zachycují detaily odlišující i podobné znaky. Pro velká tiskací písmena se jako nejvhodnější velikost nejmenší oblasti ukázala právě pětadvacetina. Při dalším zmenšování totiž úspěšnost rozpoznání neroste a naopak velmi stoupají nároky na čas a hlavně na prostor. Pro představu jsou v tabulce 10. uvedeny průměrné časy běhu procesu a velikost souboru se vzory pro rozpoznání daného počtu znaků při použití pětadvacetin a setin (v obou případech je použita stejná množina vzorových znaků). Samotná

čísla nelze brát jako ukazatel výkonnosti algoritmu, neboť jsou závislá na prostředí běhu, ale jejich vzájemné vztahy situaci popisují dostatečně. Pro jiné znaky (malá písmena, čísla, jiné abecedy, ...) by změna nejmenších oblastí mohla zvýšit úspěšnost algoritmu.

Tabulka 10. Porovnání úspěšnosti a náročnosti algoritmu na zdroje vzhledem k velikosti nejmenších význačných oblastí

Velikost nejmenší oblasti	Průměrný čas běhu algoritmu [ms]	Velikost souboru se vzory [MB]	Průměrná úspěšnost algoritmu [%]	Počet rozpoznávaných znaků	Počet simulací
1/25	2269.1	3.3	86.9	99	10
1/25	3864.2	3.3	81.0	157	10
1/100	2983.7	8.8	83.8	99	10
1/100	5312.5	8.8	82.2	157	10

Poznámka na závěr Při zpětném pohledu se ukáže, že myšlenka algoritmu není nikterak složitá, takže mě napadlo, že někdo už mohl mít podobný nápad. Věnoval jsem tedy značné úsilí hledání nějakého podobného algoritmu, abych zde netvrdil, že jsem „objevil již objevené“, ale v žádném z mně dostupných zdrojů jsem neuspěl.

4. Postprocessing

Po ukončení identifikace znaků lze výsledný text ještě zpracovat některým z následujících způsobů.

4.1. Korekce výstupu

Celkový výsledek identifikace můžeme velice zlepšit porovnáním se slovníkem možných slov. V této práci je použit jeden z algoritmů pro editační vzdálenosti.

Editační vzdálenost Zde použitý algoritmus pro editační vzdálenost pochází z kurzu [14]. Algoritmus pro dvě slova určí, kolik elementárních kroků je potřeba k přepsání prvního slova na druhé. Elementární kroky jsou následující změny:

1. Vložení znaku do slova (LUK ->KLUK)
2. Smazání znaku ze slova (KLUK ->LUK)
3. Substituce znaku za jiný (KLUK ->HLUK)

Průběh algoritmu včetně pseudokódů lze nalézt na URL ve zdroji.

Pro nás je důležité, že algoritmus nám určí cenu přepisu slova z našeho textu na každé slovo ze slovníku. Jako náhrada je tedy vybráno slovo s nejmenší cenou. Tento typ úpravy má dvě velké nevýhody. První z nich je, že pokud ve slovníku nemáme některá ze slov vstupního textu, tak tato slova budou nahrazena jinými i při správném rozpoznání. Druhá pak nastane při použití velkého slovníku, který obsahuje mnoho možných náhrad pro jednotlivá slova. Bez zásahu uživatele je pak velice obtížné rozhodnout, které z nich je to správné. Aplikace byla testována i se slovníkem obsahujícím téměř osm set tisíc českých slov[15], ale tato možnost byla zavrhnuta, neboť vyžadovala velkou míru spolupráce uživatele a to je u pouze demonstrační aplikace nežádoucí. Ve finální verzi je tedy použit slovník obsahující pouze slova, která jsem nashromáždil během různých testů algoritmů. V aplikaci určené pro běžné používání by pak bylo vhodné, využít ještě před porovnáním se slovníkem některou z možností pro určení slova na základě kontextu (např. [16]).

4.2. Aktualizace vzorů

Další skupinou dokončovacích operací je aktualizování vzorů dle zpracovaného textu. Je ovšem důležité zamezit přidání nekvalitního nebo špatně označeného vzoru, neboť takový stav může výrazně snížit úspěšnost algoritmu při dalším spuštění. Problém je, že pro rozpoznání kvalitního vzoru je většinou potřeba, aby uživatel znal princip používaného algoritmu. Automatizovaná aktualizace vzorů pouze na základě znalosti rozpoznaného a požadovaného výsledku by vydala na samostatnou práci, a tedy zde není řešena[17].

4.3. Vzhled textu

Posledním typem dokončovacích operací, které mohou být v některých aplikacích potřeba, je úprava vzhledu a pozice textu dle vstupního dokumentu. Toto se týká zejména dokumentů obsahujících obrázky, grafy či tabulky, kde si uživatel může přát pro tyto objekty ponechat místo i ve výstupním textu. Pokud je v dané aplikaci kvalitní segmentační algoritmus pro nalezení jednotlivých řádků, slov a znaků, pak lze tato volná místa objevit už v něm a tím z konečné úpravy vzhledu textu udělat snadný úkol.

5. Programátorská příručka

Následuje stručný popis logiky přiloženého programu z programátorského hlediska.

5.1. Použité vývojové nástroje

Program je napsán objektově v jazyce Java verze 7 (Oracle JDK v 7)[18] s využitím vývojového prostředí NetBeans[19]. Původně byl pro algoritmy zpracovávající obrázky použit jazyk GNU Octave[20], opensource ekvivalent známějšího Matlabu, a pro ostatní funkčnost Java. Výhody plynoucí z množství funkcí pro zpracování obrázku, které jsou v Octave dostupné, byly nakonec převáženy celkovým zpomalením aplikace pocházejícím z komunikace mezi procesy a nutností instalace Octave na počítač uživatele. Proto byly všechny algoritmy přepsány do Javy a od Octave upuštěno. Z programátorských zdrojů pak byly nejvíce využívány webové stránky dokumentace k Javě od Oraclu[21], komunitní server StackOverflow[22] a specifikace jazyka Java[23].

5.2. Základní přehled tříd a logiky programu

5.2.1. Nejdůležitější třídy

Program obsahuje celkem pět balíčků: *Data*, *GUI*, *OCR*, *Postprocessor* a *Preprocessor*.

V balíčku *Data* jsou všechny potřebné třídy a výčty pro data používaná v aplikaci. Mezi nejdůležitější z nich patři *Document*, *Row*, *Word* a *Character* pro reprezentaci objektů k rozpoznání, dále třída *FuzzySet* reprezentující fuzzy množinu a práci s jejími prvky a také třídy *FScannerCharacter*, *MSSCharacter* a *VHECharacter* pro specifické zpracování znaku jednotlivými algoritmy.

Balíček *GUI* obsahuje uživatelské rozhraní aplikace.

V balíčku *OCR* jsou třídy a rozhraní pro algoritmy rozdělující dokument na jednotlivé části a pro algoritmy rozpoznávání znaků. Zde jsou důležitá rozhraní *Analyzer* a *Recognizer*, která definují, co vše musí přidávané algoritmy obsahovat, pro bezproblémové začlenění do aplikace. Ostatní třídy v balíčku jsou pak implementace jednotlivých algoritmů použitých v práci.

Balíček *Postrpocessor* je určen pro algoritmy upravující výsledný text. V této práci je zde pouze třída pro použití editační vzdálenosti.

Posledním balíčkem je *Preprocessor*, který obsahuje algoritmy a funkce potřebné před samotným určováním znaku. Tedy binarizace obrázku, hledání kostry znaku, odstraňování chyb z naskenovaného dokumentu, normalizace velikostí atp.

5.2.2. Logika běhu programu

Nejprve je vstupní dokument načten, binarizován a rozdělen na jednotlivé řádky, slova a znaky. Následuje převod na interní reprezentaci obrázku. To znamená, že jsou zaměněny černé a bílé pixely. Tato reprezentace byla zvolena proto, že nás ve všech algoritmech zajímají pouze pixely patřící k objektu na obrázku, a tedy je žádoucí, aby právě tyto pixely měly nenulovou hodnotu.

Poté je u každého znaku identifikováno a odstraněno případné diakritické znaménko. V této fázi tedy získáváme obrázky jednotlivých znaků k identifikaci.

Poslední úpravou znaku před spuštěním OCR algoritmu je normalizace jeho velikosti, oprava případných chyb vzniklých skenováním nebo psaním a zajištění speciálních požadavků jednotlivých algoritmů (např. vytvoření kostry znaku pro algoritmus MMS). V aplikaci jsou použity tyto opravy: uzavření různých malých děr ve znaku[24] (chybějící pixely vlivem skenování a použitého psacího nástroje), spojení případných nedotažených čar, vytvoření kostry znaku, a ztenčení nebo naopak zvýraznění jednotlivých tahů. K nejzajímavějším z těchto úprav jsou informace v kapitole 2.

Nyní následuje samotné určení hodnoty znaku dle zvoleného algoritmu a nakonec, pokud je to požadováno, je provedena korekce výstupního textu pomocí algoritmu editační vzdálenosti.

5.3. Uchovávání vzorových množin

Pro uložení množin vzorových znaků jsou použity XML soubory. Příčinou této volby je jejich přenositelnost a poměrně snadná interpretace obsahu i pro méně zkušené uživatele. V přiložené podobě aplikace to nemá až takový význam, neboť není zpřístupněna žádná možnost přidávání nových vzorových znaků z důvodu uvedených v kapitole 4.2. Avšak pro případné budoucí rozšíření o nějakou polo či plně automatickou správu vzorů je možnost zásahu uživatele velice důležitá.

Každý z algoritmů potřebuje pro svůj běh jiná data, a tedy i XML schémata příslušící k jednotlivým algoritmům se liší. Například pro algoritmus MMS vypadá možný popis znaku takto:

```
<character xsi:type="mmsCharacter" character="E"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <segment type="UD" horizontalPosition="R" verticalPosition="UU"
         length="XS" startHPos="CR" startVPos="DD" endVPos="D"
         endHPos="RR"/>
  <segment type="LR" horizontalPosition="CR" verticalPosition="CU"
         length="XS" startHPos="CL" startVPos="CD" endVPos="CD"
         endHPos="RR"/>
  <segment type="LR" horizontalPosition="C" verticalPosition="DD"
         length="XS" startHPos="RR" startVPos="UU" endVPos="UU"
         endHPos="LL"/>
  <segment type="RI" horizontalPosition="L" verticalPosition="CD"
         length="S" startHPos="CL" startVPos="CD" endVPos="UU"
         endHPos="LL"/>
  <segment type="RI" horizontalPosition="C" verticalPosition="U"
         length="XS" startHPos="CR" startVPos="DD" endVPos="CD"
         endHPos="CL"/>
```

```
</character>
```

Avšak všechny algoritmy mají stejnou první úroveň schématu:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
   <characterCollection>
       znaky
   </characterCollection>
```

O práci se vzorovými množinami se stará generická třída *TrainingCharacterCollection*<*T*>, kde T je třída pro znak daného algoritmu. Samotné ukládání a načítání pak zajišťuje JAXB[25].

5.4. Rozšíření aplikace

Účel přiložené aplikace je pouze ukázka funkčnosti jednotlivých algoritmů na mém rukopisu. Pro jakékoliv jiné účely je aplikace nedostačující hned z několika důvodů. Prvním z nich je použitý algoritmus jasových součtů pro detekci znaků ve slovech. Tento algoritmus byl použit jen proto, že objektem zájmu v práci jsou pouze velká tiskací písmena, která se píší odděleně. Pro libovolné jiné písmo a v podstatě i pro použití na skutečný tiskací text napsaný člověkem bez jakéhokoliv úmyslu usnadnit strojové zpracování je tento algoritmus téměř nepoužitelný, neboť většině lidem se i tiskací text slévá do větších celků (různé sklony sousedních písmen, slitky typu AV nebo FJ, ozdobné styly písma s ocásky, ...). Dalším důvodem je již zmíněná absence možnosti přidání nových vzorů uživatelem nebo automaticky.

Logická vrstva programu ovšem může být solidním základem pro mnohem výkonnější aplikaci, neboť v ní při návrhu bylo počítáno s dalším vývojem, a tedy udržována nejvyšší možná modularita. Pro vytvoření takové aplikace je tedy potřeba nastudovat a naimplementovat některý z výkonnějších algoritmů pro detekci jednotlivých znaků (například [26]) a umožnit nějakou formu přidávání nových vzorů. Ve skutečnosti jsou funkce pro přidávání nových vzorů již naimplementovány u všech OCR algoritmů v této práci, není ovšem vyřešen problém s určením kvality vzoru, a tedy by neřízené přidávání mohlo vést (a při testování vždy vedlo) k degradaci množin vzorů a tedy i daného algoritmu (více v kapitole 4.2.).

Obecně k náhradě libovolného z algoritmů, nebo přidání nového algoritmu pro danou funkčnost stačí, aby tento implementoval příslušná rozhraní.

Další možnosti rozšíření aplikace vidím hlavně v návrhu a implementování dalších algoritmů pro rozpoznávání písmen, algoritmů pro korekci sklonu slov, písmen a řádků, algoritmů pro detekci a vykreslování tabulek, grafů a obrázků

v dokumentu, atd. Mnoho prostoru pro zlepšení je také ve fázi předzpracování dat.

Zajímavou myšlenkou je náhrada výstupu například LaTeXovým kódem, kdy by výstupní dokument mohl přesně dodržet formát vstupu včetně výše zmíněných tabulek a obrázků. S tímto částečně souvisí i myšlenka algoritmů schopných rozpoznat matematický text[27].

6. Uživatelská příručka

Poslední část práce je věnována popisu uživatelského rozhraní přiložené aplikace.

Pro spuštění aplikace je nutné mít na počítači nainstalováno prostředí Java runtime environment verze 7[18]. Java je sice multiplatformní, ale na 32-bitovém Windows 7 bylo při testování zjištěno odlišné chování některých pvků GUI. Všechny nalezené odlišnosti jsem sice opravil, přesto bych pro použití aplikace doporučil spíše linuxový systém. Také je nutné, aby binární soubor *ocr.jar* byl v adresáři spolu s adresářem *data* a v něm dále adresář *vzory* obsahující XML soubory se vzorovými daty a soubor se slovníkem.

Aplikaci spustíme dvojklikem na *ocr.jar*, nebo z příkazové řádky pomocí `java -jar ocr.jar` Po spuštění se objeví hlavní okno aplikace, které obsahuje osm prvků: oblast pro zobrazování vstupního obrázku na levé straně, pod ní indikátor průběhu informující o výpočtu a tlačítko pro zrušení probíhajícího výpočtu, dále oblast pro zobrazování výstupního textu na pravé straně a čtyři ovládací prvky v pravém dolním rohu. Jedinými dalšími prvky, se kterými se uživatel může setkat, jsou dialogy s informačními či chybovými zprávami a standardní dialog pro otevírání souboru.

Používání aplikace Pro spuštění rozpoznávacích algoritmů musíme nejprve načíst obrázek s textem. To provedeme stisknutím tlačítka *Načíst*, objeví se nám dialog pro otevření souboru. Najdeme tedy požadovaný soubor a výběr potvrdíme. Nyní musíme vybrat jeden z dostupných algoritmů v rozbalovacím seznamu označeném nápisem *Algoritmus*. Před samotným spuštěním ještě můžeme zaškrtnout možnost *Kontrola dle slovníku*, čímž zajistíme, že všechna slova výstupního textu budou porovnána se slovníkem. Posledním krokem je kliknout na tlačítko *Rozpoznat* a v závislosti na množství textu v souboru a zvolených možnostech chvíli počkat. Doba čekání se dá přibližně určit dle rychlosti růstu vyplněné části indikátoru průběhu. V případě, že výpočet chceme přerušit, použijeme tlačítko *X*.

Závěr

V práci byl popsán proces strojového rozpoznávání ručně psaného textu od okamžiku jeho získání, až po některé úpravy ve výsledné digitalizované podobě. Největší prostor byl věnován jednotlivým algoritmům pro samotnou identifikaci znaků a zejména metodám založeným na fuzzy logice, které tyto algoritmy využívají k eliminaci odchylek v rukopise.

Nejzajímavější částí pak byl poslední bod zadání, tedy návrh vlastního algoritmu. Výsledkem tohoto snažení je algoritmus VHE, který vykazuje relativně zajímavou úspěšnost rozpoznávání. Považuji jej tedy i za největší přínos této práce.

Součástí práce je i demonstrační aplikace, která v současné podobě nemá příliš velkou možnost využití, neboť, zejména ve fázi segmentace textu na znaky, jsou některé algoritmy nedostatečné pro rozpoznávání psacího, či jinak spojitého písma. Na druhou stranu jsem při její tvorbě počítal s dalším rozšířením, a tedy celé její jádro bude použitelné i s lepšími algoritmy pro pre i postprocessing, které bych v budoucnu rád implementoval.

Reference

[1] K. Yamamoto S. Mori, C. Y. Suen. Historical review of OCR research and development. *Proceedings of IEEE*, 80, 1992.

[2] R. B. Knapp. Fuzzy sets and pattern recognition. http://hci.sapp.org/lectures/knapp/fuzzy/fuzzy.pdf [Cit. 4.8.2013], 1998.

[3] V. Nepožitek. Co a k čemu je fuzzy logika?, 2009.

[4] *A first course in fuzzy logic*. Boca Raton: Chapman & Hall edition, 2000.

[5] G. Al-Talib M. Z. Khedher. Recognition of secondary characters in handwritten arabic using fuzzy logic. *International Conference on Machine Inteligence (ICMI'05)*, 2005.

[6] N. Otsu. A threshold selection method from gray-level histograms. *IEEE Transactionson Systems, Man and Cybernetics*, 1979.

[7] A. Greensted. Otsu thresholding explained. www.labbookpages.co.uk/software/imgproc/otsuthreshold.html [Cit. 29.7.2013].

[8] M. Černý. Rozpoznávání registračních značek motorových vozidel, 2010.

[9] C. Y. Suen T. Y. Zhang. A fast parallel algorithm for thinning digital patterns. *Communications of the ACM*, 27, 1984.

[10] P. Veselý. Ocr analýza, 2005.

[11] P. Mahashukhon, H. Mousavinezhad, J. Y. Song. Hand-printed english character recognition based on fuzzy theory. *IEEE International Conference on Electro/Information Technology (EIT)*, 2012.

[12] W. A. Gowan. Optical character recognition using fuzzy logic. http://www.freescale.com/files/microcontrollers/doc/app_note/an1220_d.pdf [Cit. 2.8.2013].

[13] The significance of letter position in word recognition. http://www.mrc-cbu.cam.ac.uk/people/matt.davis/cmabrigde/rawlinson/ [Cit. 29.7.2013].

[14] P. Osička. Kurz algoritmická matematika 3 na KMI UPOL, zimní semestr 2012. http://phoenix.inf.upol.cz/~osicka/courses/alm3/slides.pdf [Cit. 2.8.2013].

[15] J. Fiala. Slovník českých slov. http://www.pspad.com/cz/download.php [Cit. 2.8.2013].

[16] D. Roth A. R. Golding. A winnow-based approach to context-sensitive spelling correction. *Machine Learning 34*, page 107–130, 1999.

[17] J. Cao, M. Shridhar, F. Kimura, M. Ahmadi. Statistical and neural classification of handwritten numerals: A comparative study. *11th IAPR International Conference on Pattern Recognition*, pages 643 – 646, 1992.

[18] Oracle java development kit (JDK) verze 7. http://www.oracle.com/technetwork/java/javase/downloads [Cit. 2.8.2013].

[19] NetBeans IDE. https://www.netbeans.org/ [Cit. 2.8.2013].

[20] GNU Octave. http://www.gnu.org/software/octave/ [Cit. 2.8.2013].

[21] Java Platform Standard Edition 7 Documentation. http://www.docs.oracle.com/javase/7/docs/ [Cit. 2.8.2013].

[22] StackOverflow. http://www.stackoverflow.com/ [Cit. 2.8.2013].

[23] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley. The java language specification. http://www.docs.oracle.com/javase/specs/jls/se7/jls7.pdf [Cit. 3.8.2013].

[24] P. Tarábek. Morphology Image pre-processing for thinning algorithms. *Journal of Information, Control and Management Systems*, 5, 2007.

[25] Java Architecture for XML Binding (JAXB). http://www.oracle.com/technetwork/articles/javase/index-140168.html [Cit. 2.8.2013].

[26] G. Louloudis, B. Gatos, I. Pratikakis, C. Halatsis. Text line and word segmentation of handwritten documents. *G. Louloudis et al., Pattern Recognition 42*, 2009.

[27] G. Labahn, S. MacLean. A new approach for recognizing handwritten mathematics using relational grammars and fuzzy sets. *International Journal on Document Analysis and Recognition (IJDAR)*, 16, 2013.

A. Obsah přiloženého CD

`bin/`
: Zkompilovaná verze demonstrační aplikace spolu s adresářovou strukturou obsahující vzorové soubory a ukázková data. (Binární soubor musí vždy zachovat relativní pozici vůči souborům se vzory.)

`doc/`
: Text práce ve formátu PDF a ZIP archiv se zdrojovými soubory textové části.

`src/`
: Kompletní zdrojové soubory demonstrační aplikace.