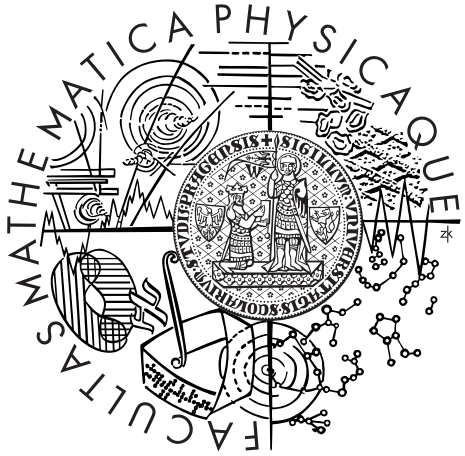
Univerzita Karlova v Praze

BAKALÁSKÁŘSKÁ PRÁCE



Tuan Hiep Tran

Rozpoznávání textu na fotografiích exteriéru

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce:	RNDr. Pavel Surynek Ph.D.
Studijní program:	Informatika
Studijní obor:	Obecná informatika

Praha 2015

Chtěl bych hlavně poděkovat RNDr. Pavlu Surynkovi PhD. bez jehož pomoci by tato práce nebyla možná.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Rozpoznávání textu na fotografiích exteriéru

Autor: Tuan Hiep Tran

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Jméno a příjmení s tituly, pracoviště RNDr. Pavel Surynek, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Ústředním tématem naší práce je rozpoznávání textu v exteriéru. Tato technologie nachází v současnosti mnohá uplatnění v praxi a do budoucnosti skrývá ohromný potenciál. Na téma práce se zaměřujeme jak z teoretického tak i z experimentálního hlediska. Nejdříve v teoretické části práce rozeberáme detailněji existující metody. Na základě tohoto rozboru navrhujeme integraci algoritmu *ER* [17] pro detekci písmen s námi vylepšeným algoritmem od Phana kol. [22] pro detekci slov. Takto integrovaný algoritmus jsme exprimentálně otestovali na sadě *ICDAR 2013* s výsledky *precision* 0.6 a *recallu* 0.73. Součástí práce je rovněž snadno rozšiřitelná C++ knihovna, konzolová a gui aplikace pro rozpoznávání textu využívající navrženou metodu.

Klíčová slova: Počítačové vidění, Rozeznávání textu na fotografiích, MSER, Rozpoznávání slov se slovníkem

Title: Text Recognition in Natural Scenes

Author: Tuan Hiep Tran

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Pavel Surynek, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The main topic of our thesis is text recognition in natural scenes. This technology has finds many useful applications in the present world and has a great potential in the future one. We are analyzing the topic from theoretical but also from experimental perspective. Detailed analysis is done in theoretical part of this work. Based on this analysis, we propose integrating algoritm ER[17] for letter detection with modified algorithm from Phan and co. for word recognition. Proposed method had been experimentally evaluated on testset *ICDAR 2013* and archieved *recall 0.6* and *precision 0.72*. Our work also includes C++ library for text recognition, console and gui application for text recognition using the proposed method.

Keywords: Computer vision, Text recognition, MSER, Word recognition with dictionary

Obsah

1 Úvod **4**
- 1.1 Motivace . . . 4
  - 1.1.1 Aplikace . . . 4
- 1.2 Obsah práce . . . 5
  - 1.2.1 Cíl . . . 5
- 1.3 Selhání *OCR* . . . 5

2 Přehled existujících metod **7**
- 2.1 Úvod do problematiky . . . 7
- 2.2 Detekce písmen . . . 7
  - 2.2.1 Sliding window přístup . . . 7
  - 2.2.2 Component based přístup . . . 9
  - 2.2.3 Shrnutí . . . 10
- 2.3 Detekce slov . . . 11
  - 2.3.1 Detekce na zakladě podobnosti písmen . . . 11
  - 2.3.2 Detekce na základě slovníku . . . 11
- 2.4 Extrakce lexikografické informace . . . 12
  - 2.4.1 Řešení . . . 12
- 2.5 Závěr rozboru . . . 12

3 Vymezení obsahu práce **14**
- 3.1 Vybrané oblasti . . . 14
- 3.2 Experimenty . . . 14

4 Navrhovaná metoda **15**
- 4.1 Motivace . . . 15
- 4.2 Přehled navržené metody . . . 15
- 4.3 Extrakce potencionálních písmen . . . 15
  - 4.3.1 MSER . . . 16
  - 4.3.2 Extremal regions . . . 23
- 4.4 Generování slov . . . 28
  - 4.4.1 Vyčíslení $P(l \mid c)$ . . . 28
  - 4.4.2 Původní algoritmus . . . 30
  - 4.4.3 Navržené modifikace algoritmu . . . 34

5 Experimentální část **40**
- 5.1 Testovací data . . . 40
- 5.2 Metrika testování . . . 40
- 5.3 Detekce písmen . . . 40
  - 5.3.1 Popis testovaných metod . . . 41
  - 5.3.2 Non max suprese s *OCR* . . . 41
  - 5.3.3 Trénovací data . . . 41
  - 5.3.4 Vyhodnocovací protokol . . . 41
  - 5.3.5 Předpoklad . . . 42
  - 5.3.6 Výsledky . . . 42

5.3.7 Zajímavé výsledky . . . 44
5.3.8 Závěr . . . 45
5.3.9 Znovu spustitelnost experimentu . . . 46
5.4 Detekce slov . . . 46
5.4.1 Ground truth . . . 46
5.4.2 True positive . . . 47
5.4.3 Kompatibilita *ER* a *MSER* . . . 47
5.4.4 Optimalizace pomoci slovnikové trie . . . 50
5.4.5 Vliv našich modifikací . . . 51
5.4.6 Závěr . . . 53

6 Závěr **54**
6.1 Alternativní směrování práce . . . 54
6.1.1 Detekce písmen . . . 54
6.1.2 Detekce slov . . . 54
6.1.3 OCR fáze . . . 54
6.1.4 Ostatní trendy . . . 54
6.2 Shrnutí práce . . . 55

Přílohy **56**

7 Teoretická příloha **57**
7.1 Stroke Width Transform . . . 57
7.2 SVM s fast intersection kernel . . . 60

8 Uživatelská dokumentace **63**
8.1 Úvod . . . 63
8.1.1 Rozpoznávání textu na fotografii . . . 63
8.2 NOCR Software . . . 63
8.2.1 Systemové požadavky . . . 63
8.2.2 Obsah instalačního balíčku . . . 63
8.2.3 Obsah multimedia . . . 64
8.2.4 Instalace . . . 64
8.2.5 Návod k použití . . . 65

9 Implementace **70**
9.1 Využité knihovny . . . 70
9.2 Návrh knihovny . . . 70
9.3 Segment . . . 70
9.3.1 Letter . . . 71
9.3.2 Policy class . . . 71
9.3.3 Integrace OCR . . . 73
9.3.4 Rozšiřitelnost o nový typ extrakce . . . 74
9.3.5 Shrnutí . . . 74
9.4 ERTextDetection . . . 75
9.4.1 ComponentTreeBuilder<T> . . . 75
9.4.2 ERTree . . . 76
9.4.3 Stavba komponentového stromu . . . 77
9.4.4 Extrakce komponent z komponetového stromu . . . 77

9.4.5 Implementace MSERU 77
9.5 WordGenerator 79
9.5.1 Dictionary 79
9.5.2 TranslatedWord 79
9.6 Příznaky 79
9.6.1 Composite 79
9.6.2 Factory Method 80
9.7 Trenování klasifikátorů 80
9.7.1 FeatureTraits 81
9.7.2 TrainExtractionPolicy 81
9.7.3 Rozšiřitelnost 81
9.7.4 Trenování klasifikátoru 82
9.8 Klasifikátor 82
9.8.1 Wrapovaní klasifikátoři 82
9.8.2 SVM Fast Intersection Kernel 82
9.9 Shrnutí 82

1. Úvod

Úkolem rozpoznávání textu na fotografiích exteriéru je naučit počítač lokalizovat a rozpoznat text na fotografiích exteriéru, který bude následně převeden do podoby vhodné k dalšímu strojovému zpracování.

1.1 Motivace

Slova na digitálních snímcích jsou užitečným zdrojem informací o daném snímku. Tyto informace nám usnadní například určit místo pořízení snímku, čas, relevantní informaci o daném snímku. Ale pro strojové zpracování textové informace je ale třeba text nejdříve na fotografii rozeznat.

1.1.1 Aplikace

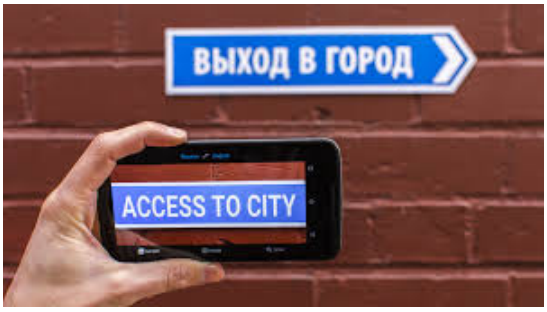
V současnosti si technologie pro rozpoznávání textů na fotkách našla mnohá uplatnění v praxi. Jmenujme alespoň nějaké příklady:

- Rozpoznavání SPZ automobilů, jedná se o integraci rozpoznávání textu s kamerovým systémem. Tato integrace následně může sloužit k hledání odcizených automobilů, evidenci automobilů jenž porušili dopravní předpisy, například překročení maximální rychlosti ve městě atd.
- Robotika, rozpoznané text robotovi přináší důležité informace o okolí.
- Aplikace překladající text z cizích jazyků, pořizuje digitální snímky, na kterých by rozpoznala text, ten následně přeložila do libovolného jazyka. Možné využití například pro turistiku do cizí země.
- Pomůcka pro nevidomé, rozponání textu může být opět integrováno s nějakou mobilní technologií, google glass například, který bude snímat okolí uživatele. Rozpoznaná textové informace, směrové tabule například, následně poskytují řadu užitečných technologií uživateli pomocí hlasového modulu.

Obrázek 1.1: Ukázky aplikací rozponávání textu

(a) rozponávání SPZ

(b) mobilní překladač



1.2 Obsah práce

V práci nejprve provedeme detailní rozbor existujících metod z teoretického hlediska, na základě tohoto rozboru navrhneme experimentálně ověříme úspěšnost navrhované metody.

1.2.1 Cíl

Naším cílem je navrhnout metodu pro rozpoznávání textů ve fotografiích exteriéru. Nejdřív proto problém dekomponujeme do 3 menších podproblému, v každém tomto podproblému si uvede rozbor existujích řešení z teoretického hlediska. Na základě tohoto rozboru vymezíme obsah práce, kterému budeme věnovat detailnější pozornost. Ze získaných poznatků poté navrhneme vlastní metodu, kterou naimplementujeme v softwarovém prototypu jehož úspěšnost následně ověříme na testovací sadě *ICDAR 2013* [12].

Softwarová specifikace Softwarový prototyp bude naimplementován v jazyce C++ s využitím nové normy C++11 pod unixovým překladačem GNU GCC. Dále budeme také využívat knihovnu OpenCV pro počítačové vidění. Naimplementovaný software bude volně distribuovatelný pod opensourcovou licencí.

Software se bude skládat ze 3 hlavních částí.

- Knihovna pro rozpoznávání textu v digitálních snímcích
- Konzolová aplikace
- Interaktivní GUI aplikace

Obě aplikace budou umět interpretovat výsledky rozpoznávání ve vhodné podobě pro další zpracování.

1.3 Selhání *OCR*

Jako možné řešení rozpoznávání textu na digitálních snímcích se nabízí využití *OCR*, úspěšné technologie pro rozpoznávání textu v dokumentech. Toto řešení ale dosahuje velice špatných výsledků, které jsou zapříčeněny mnoha faktory, popišme si ty nejdůležitější:

- složité pozadí, narozdíl od dokumentů kde máme jasně separovatelné písmo od pozadí, mají písmena na fotkách složité pozadí což by pro *OCR* stěžovalo detekci písmen
- různá kvalita snímku, rozlišení má výrazný vliv na rozpoznávaní, nasvícení a stínování také stěžují detekci písmen a šum vytváří nechtěné artefakty, stěžuje detekci písmen
- geometrické deformace, písmena na fotkách mají různé geometrické vlastnosti, je potřeba se vypořádat se s geometrickýma deformacema, má následky v překladu do textové podoby

- různé fonty, je potřeba se vypořádat s různými fonty písmen vyskytujících se na snímku
- struktura textu, text na fotce narozdíl od dokumentu, kde máme řádky rozdělené do slov, nemá jasnou strukturu, což stěžuje shlukování písmen do slov
- různé perspektivy snímků, text na fotce nemusí být vždy frontální narozdíl od dokumentů,

Tedy přímé využití *OCR* není řešení dosahujících slušných výsledků.

2. Přehled existujících metod

2.1 Úvod do problematiky

Problém rozpoznávání textu na digitálních snímcích, se dá podrozdělit do 3 dílčích podoblastí:

- detekce písmen
- detekce slov z extrahovaných písmen
- extrakce lexikografické informace

Ke každému z těchto 3 problému existují různé přístupy řešení. Poznamenejme, že existují přístupy, které vyřeší víc těchto podproblému najednou.

V následujících sekcích této kapitoly popíšeme podrobněji dílčí problémy a uvedeme již existující přístupy k řešení a vzájemné porovanání.

2.2 Detekce písmen

Cílem je detekce a lokalizace písmen na snímku. Často se jedná o výpočetně nejnáročnější část jejíž úspěšnost přímo ovlivňuje úspěšnost celého procesu rozpoznávání.

Pro extrakci existují dva odlišné způsoby. První z nich je *sliding window* přístup, velice úspěšná metoda v počítačovím vidění pro detekci objektů, a druhým je přístup založený na hledání komponent souvislosti ve vstupní bitmapě, takzvaný *component based* přístup.

(a) vstup (b) detekovana pismena na vstupu

Obrázek 2.1: Detekce písmen ukázka

2.2.1 Sliding window přístup

Jedná se o úspěšně aplikováný framework v oblasti rozpoznávání objektů. Obvykle funguje na základě kombinace skenování gaussovské pyramidy vstupního snímku obdelníkovým oknem v kterém se extrahuje deskriptor a platnost kritérií, které

většinou vrací odezvu pro dané okno. Skenováním na gaussovské pyramidě vstupu, si zajišťujeme scale invarianci.

Uveďme algoritmus popsaný v Computer Vision, A Modern Approach od Forsytha a Ponce [7], poznamenejme že se jedná jen o základní kostru, konkretní metody se většinou liší.

Algoritmus 1: Sliding window přístup

vstup : vstupní obrázek
$\Delta x \ldots$ posun po ose x
$\Delta y \ldots$ posun po ose y,
skenovací okno o rozměrech $m \times n$
$t \ldots$ hodnota pro prahování

výstup: list L detekovaných objektů sestupně setříděných podle odezvy

```
1  postav gaussovu pyramidu vstupu
2  foreach vrstvu v pyramidě do
3  |  foreach pozici skenovacího okno při posunu o Δx, Δy do
4  |  |  Aplikuj kriterium, vrať odezvu c
5  |  |  if c > t then
6  |  |  |  Vlož skenovacího okna do listu L, dle jeho odezvy
7  |  |  end
8  |  end
9  end
10 foreach W ∈ L do
11 |  W_prekryv ← {W_1 ∈ L | W_1 ≠ W ∧ W_1 má výrazný překryv s W}
12 |  Při překryvu je nutné jedno okno vyscalovat na uroveň druhého
13 |  foreach W_1 ∈ W_prekryv do
14 |  |  vymaž W_1 z L
15 |  end
16 end
```

Z popisu algoritmu vidíme, že metoda se aplikuje na gaussově pyramidě vstupního obrázku, kde vydetekujeme objekty na různých vrstvách, generujeme jakési hypotézy a ty následně ohodnotíme nějakou odezvou. Následně probíha tzv. non-max suprese, kde se zamítnou okna, čili hypotézy, s menší odezvou.

Metody této kategorie v poslední době hojně využívají algoritmů strojového učení. Mezi zvlášť důležité práce bychom zařadili framework pro detekci obličejů navržený od Violy a Jonase [25]. Tato práce přináší do sliding windows přístupu, ideu kaskádovitého klasifikátoru, díky které algoritmus od Violy a Jonese dosahuje jako první algoritmus pro detekci obličejů real-time výkonu. Kaskádovitý klasifikátor je struktura klasifikátorů, kde na první urovni kaskády počítáme nějaký rychle vypočítatelný deskriptor a provadíme klasifikaci, na druhé úrovni zpočítáme o něco více informativní deskriptory, ale pomaleji vypočitatně náročnější ze všech oblastí, které prošli první fází a opět klasifikujeme, na třetí úrovni volíme informativnější deskriptory a opět klasifikujeme jen obdelníky, které prošli 2. fází, takto pokračujem dále až na poslední uroveň kde již počítáme doopravdy velice informativní příznaky, jejichž vyčíslení je ale časově i výpočetně náročné. Takto zkonstruktovaný klasifikátor umí rychle vyfiltrovat oblasti, které nejsou určitě detekovaná místa v prvních úrovních kaskády. A informativnější příznaky počítá jenom u obdelníku, které jsou pravděpodnější hledané obličeje, to jest ty

které projdem více úrovněmi kaskády. Jako klasifikační algoritmu využívá Viola a Jones *Real AdaBoost* [8] a využívá *Haar* příznaky. Jejich práce navíc navrhuje metodu, jak *Haar* příznaky rychle vyčíslit.

Další příkladem aplikace je využití sliding window přístupu pro detekci chodců. Zde byl přístup kombinován z *HoG (Histogram of oriented gradient)* deskriptorem [4] a *SVM(Support Vector Machine)* s lineárním jádrem.

Framework funguje dobře pro objekty s jasnou strukturou, navíc je poměrně robustní vůči šumu, rozmazání, nasvícení a stínování na vstupu, díky využití sofistikovanějšího deskriptoru (například již zmíněné haar deskriptory a HoG nebo SIFT deskriptor). Výhodou je také jednodušší implementace.

Aplikace sliding window přístupu v rozpoznávání písmen

Popsaný přístup lze aplikovat v rozpoznávání písmen, ale narozdíl od detekce obličejů, chodců se nedá jednoznačně určit rozměr skenovacího okna. Pro obličeje použijeme obdelník, který bude něco málo vyšší než širší, v případě chodců využijeme skenovací okno které bude mít výrazně větší výšku než šířku, tak abychom do něj mohl být schopni zabrat lidskou postavu, ale v případě rozpoznávání písmen je potřeba volit více skenovacích oken. Uvažme například písmena w a l, kdybychom chtěli detekovat jenom písmeno písmeno w, využijeme okno čtvercových rozměrů, ale takovýto čtverec nebude pořádně skenovat písmeno l, jelikož l má podlouhlý obdelníkový charakter. Problémem aplikováním tohoto přístupu je tedy nutnost skenovat gaussovu pyramidu vstupu pomocí více oken což zapříčíní větší výpočetní a časovou náročnost. Nutnost většího počtu skenovacích oken způsobena variací různých tvarů jednotlivých písmen.

Na druhou stranu při vhodném výběru deskriptoru je tento způsob robustní vůči šumu, rozmazání, zhoršené kvalitě vstupu. Sliding window přístup byl například úspěšně aplikován Wangem v [26], nebo od Tiana v [23].

Obrázek 2.2: Písmena nedetekovatelná jedním rozměrem skenovacího okna

2.2.2 Component based přístup

Princip component based metod spočívá ve nalezení komponent souvisloti ze vstupní bitmapy, a následné filtraci na nepísmenové a písmenové komponenty pomocí geometrických vlastností extrahovaných komponent.

Hlavní nevýhodou těchto metod je předpoklad že každé písmeno na vstupu je právě jedna komponenta, dále také nedosahují robustnosti *sliding window* metod vůči šumu, rozmazání a nasvícení světlem. Tyto faktory mohou porušovat tvar

písmenkové komponenty, čímž se zkreslí její geometrické údaje což může vyustít ve špatné rozhodnutí při filtraci.

Na druhou stranu tyto algoritmy jsou výpočetně nenáročné a narozdíl od předchozí kategorie detekce písmen jako výstup dostáváme přímo tvary komponent, kdežto ve sliding window metodách dostáváme pouze údaj o lokaci písmene. Tvar komponenty nám může usnadnit překlad do textové podoby, jelikož známe tvar písmene. V případě component based metod lze tedy pokladát detekci písmen zároveň za segmentaci písmen od pozadí. Ve sliding window metodách lze také segmentovat tvar komponenty, třeba pomocí otsu binarizace z již vydetekovaných oblastí, ale jedná se dodatečnou operaci ne o přímý výsledek běhu algoritmu.

Extrahování komponent

V detekci písmen byly úspěšně aplikovány dva druhy extrakce komponent z bitmapy, a to jsou :

1. hledání souvislých komponent v nějaké transformaci vstupu nebo i ve vstupu samotném, na základě nějaké podobnostní metriky hodnot pixelů.

2. algoritmy typu *MSER* [14]

Pro 1. možnost jako vstup pro libovolný algoritmus hledání komponent, lze buď použít původní vstup a pixely shlukovat do jedné komponenty například na hodnotě intenzity v pixelech, ale tento způsob je zjevně nevhodný. Lepší je použít vhodnou transformaci vstupu, lze například aplikovat lokální otsu binarizaci, nebo komponenty hledat v *SWT* transformaci [5].

Druhou možností je využití algoritmů *MSER*, což je *blob detektor* hledající komponenty souvisloti se specifickou vlastností.

Filtrace

Účelem filtrace je vyfiltrovat z množiny extrahovaných komponent ty písmenkové.

Pro filtraci lze použít jednoduché geometrické testy vlastností jako je výška, šířka, úhlopříčka a podobných, kde se dá práh pro rozdělení intuitivně odhadnout. Tento způsob byl použit Epshteinem na komponentách vyextrahovaných ze *SWT (Stroke width transformace)* [5] vstupu. Velkou nevýhodou je intuitivní volba prahů pro jednotlivé testy.

Druhá, rozšiřenější možnost je využití klasifikačních algoritmů s učitelem, jako jsou *SVM*, *Boosting* nebo *rozhodovací stromy*. Vstupy do těchto algoritmu je feature vectory komponenty, jehož jednotlivé složky tvoří vybrané geometrické vlastnosti dotyčné komponenty. Lze také využít sofistikovaňejší deskriptory jako v případě sliding window přístupu, ale geometrické vlastnosti nalezených komponent jsou dostatečně deskriptivní.

2.2.3 Shrnutí

Představili jsme si základní koncepty používané při detekci písmen v rozponávání textu. V prvním případě jsme si popsali rozšířený framework pro detekci objektů sliding window přístup a jeho použití na detekci písmen. Uvedli jsme, že je výhodný zejména díky možnosti dosáhnout relativní robustnosti vůči horším

kvalitě. Nevýhodou tohoto přístupu je větší výpočetní náročnost způsobená více rozměry skenovacích oken. Druhým konceptem, který jsme uvedli je component based přístup. Tento přístup na rozdíl od toho prvního není robustní vůči horší kvalitě vstupy, obvzláště vůči odleskům a nasvícení snímku a neumí si poradit s detekcí písmen, které se skládají z více komponent, ale za to je poměrně výpočetně nenáročný a seperuje písmena od pozadí snímku.

2.3 Detekce slov

Detekce slov je fáze rozpoznávání textu, kde již pracujeme s detekovanými písmeny. Úkolem je z těchto písmen sestavit slova. Zatím existují dva různé přístupy buď lze detekovat slova dle geometrické podobnosti písmen nebo na základě slovníku.

(a) detekovana pismena na vstupu (b) detekovana slovo na vstupu

Obrázek 2.3: Detekce slov ukázka

2.3.1 Detekce na zakladě podobnosti písmen

V první kategorii se využivá předpoklad podobnosti písmen ve slově, například lze předpokládat že písmena mají relativně podobnou barvu, výšku ..., dalé se předpokládá že vzdálenost mezi písmeny ve slově není moc velká. Na základě těchto informací jsou písmena shlukována do slov například s využitím technik strojového učení. Výstupech těchto metod je seznam skupin písmen tvořecích slovo.

Tento přístup lze dobře kombinovat s detekcí písmen pomocí *component based* metod, protože již máme vysegmentovanou komponentu ze vstupu.

Mezi články, které navrhují metody této kategorie pro hledání slov bychom mohli zařadit práci od Gomeze a Karatzase [9], implementovanou v OpenCV, dále práci od Neumanna a Matase [16].

2.3.2 Detekce na základě slovníku

Druhou kategorii přístupu ke shlukování písmen, je generování slov dle slovníku, tento přístup byl poprvé publikován v článku od Wanga [26], zmiňme ještě práce od Novikové a spol. [20] a od Trunga a spol[22]. Narozdíl od první kategorie

se tento přístup nespoléhá na předpoklad similarity písmen ve slově, ale využíva lexikografickou informaci obsaženou ve slovníku a jistý jazykový model. Na tomto základě generuje slova ze slovníku, které se nejpravděpodobněji nachází na vstupní bitmapě. Zásadním rozdílem oproti předchozímu typu metod je, že výstupem není seznam skupin písmen tvořících slovo, ale slova ze slovníku, které lze nejoptimálněji nějakým způsobem poskládat z detekovaných písmen na základě jistého jazykového modelu. Jistou nevýhodou může být také výpočetní náročnost přímo závisející na velikosti slovníku, na druhou stranu ale metoda v případě volby relevantního slovníku bude dosahovat lepších výsledků detekování.

2.4 Extrakce lexikografické informace

Při extrakci lexikografické informace zjišťuje pro detekované písmeno jeho charakter v abecedě nebo pravděpodobnosti, toho že detekované písmeno je nějaký charakter z abecedy. To co chceme zjistit se liší volbou přístupu detekce slov.

V případě detekce slov na základě podobnosti písmen potřebujeme jenom vědět charakter daného písmene, jelikož výstupem je skupina písmen, tvořících slovo. Textovou podobu slova následně dostaneme spravným seskupením extrahovaných charakterů písmen detekovaného slova. Extrakce lexikografické informace tedy může proběhnout po i před detekovaním slov.

Naopak při detekci slov pomocí slovníku, kde je výstupem nějaké slovo obsažené v daném slovníku, potřebujeme pravděpodobnosti abychom mohli uplatnit daný jazykový model. Proto počítání pravděpodobností musí proběhnout před generování slov.

2.4.1 Řešení

Pro zjištění charakteru nebo vyčíslení pravděpodobnostího výstupu jednotlivých labelů v abecedě daného detekovaného písmena lze použít existující technologie *OCR*

2.5 Závěr rozboru

V rozboru existujících metod pro rozpoznávaní textu, jsme si nejdříve problém rozdělili na 3 menší části. Tyto části jsme detailněji popsali a vždy si zmínili koncepty existujících řešení a nějaké příklady konkrétních metod.

Z rozboru usuzujeme, že nelze vybrat obecně nejlepší přístup. Místo toho jsme dospěli k názoru, že vhodnost jednotlivých přístupů se liší v situacích, kde je rozpoznávaní textu využito.

Pro ilustraci si porovnejme vhodnost jednotlivýh přístupu pro detekci slov při rozpoznávání SPZ automobilů. Volba detekce na základě podobnosti písmen je zde výhodnější než na základě slovníku a to hned z několika důvodů. Prvním je velikost slovníku, v případě detekce SPZ automobilů, by slovník mohl dosahovat poměrně značné velikosti, což se samozřejmě odrazí na výpočetní náročnosti. Dalším faktorem při rozhodování hrala značná podobnost písmen na SPZ značkách, ty mají společný font, stejnou výšku, podobnou barvu a jsou blízko u sebe.

Naopak při nasazení rozpoznávání textu v pomůcce ve vizuálním překladači, může být výhodnější využití detekování na základě slovníku a jazykového modelu, protože nelze předpokládat podobnost mezi jednotlivými rozpoznanými písmeny, dále většinou detekujeme specifikcý okruh slovíček, v případě aplikace pomáhající turistům jde o okruh slovíček použitých na orientačních tabulích.

V následující kapitole přesně specifikujeme obsah této práce, tj. vybereme okruh algoritmů jemiž se budeme zabývat, uvedeme přesnou specifikaci vytvořeného softwaru a popíšeme si experimenty, jenž budeme provádět.

3. Vymezení obsahu práce

3.1 Vybrané oblasti

V naší práci zaměříme se zaměříme zaměříme hlavně na řešení podproblémů detekce písmen a slov.

V detekci písmen budeme věnovat pozornost component based přístupům. Speciálně si detailně představíme přístup detekce pomocí *MSERU* [14], bude uvedena varianta dosahující lineání časové a prostorové složitosti. Na základě popisu *MSER* dále popíšeme algoritmus pracující s konceptem *ER* [17], což je zobecnění *MSERU*.

Následně se zaměříme na algoritmus od Phana a spol. [22]. Tento algoritmus rozeznává slova na základě jazykového modelu a slovníku. Jeho součástí je také *OCR* fáze.

Nejdříve rozebereme návrh původního *OCR*, poté představíme náš návrh na alternativní *OCR*. Poté si zadefinujeme jazykový model v práci od [22] a na základě něho popíšeme původní algoritmus. Nakonec si představíme námi navrhované modifikace.

Všechny naše navrhy budou ověřeny implementací v softwarovém prototypu a experimentálním vyhodnocením.

Poznamenejme, že softwerový prototyp bude plně kompatibilní s knihovnou OpenCV. Tato kompatibilita je žadoucí, protože knihovna OpenCV je běžně využívána v počítačovém vidění navíc implementuje velké množství algoritmů, které využijeme v našem prototypu.

Podrobný popis softwaru, jeho instalaci a ovladaní najdete v kapitole 8.

3.2 Experimenty

V experimentech budeme zkoumat atributy precision a recall uvedených algoritmů na testovací sadě *ICDAR 2013* [12]. V případě trenování klasifikátoru budeme používat vždy stejné vzorky.

V prvním experimentu se zaměříme na detekci písmen. Budeme testovat následující způsoby detekce písmen algoritmus *ER* [17], *MSER* [14] s filtrací na základě geometrických vlastností pomocí *SVM*, lokální *otsu binarizace* [21] se *SWT* a filtrací. Detekovat slova budeme pokaždé pomocí námi vylepšeného algoritmu od Phana a spol. [22].

V druhém experimentu se zaměříme na detekci slov. Budeme zkoumat vliv námi navržených zlepšení oproti původního algoritmu a kompatibilitu algoritmu s metodami pro detekci písmen.

4. Navrhovaná metoda

4.1 Motivace

V předhozí kapitole jsme si vymezili rozsah práce. Při detekci písmen jsme rozhodli se zabývat hlavně komponentovými metodami, z důvodů uvedených v rozboru komponentových metod 2.2.2. Jedná se výpočetní nenáročnost a přímé seperaci písmena od pozadí. Na druhou stranu tyto metody nedosahují takové robustnosti jako *sliding window* metody. Tento nedostatek se pokusíme odstranit využitím metody *ER* od Neumanna a Matase [17], jedná se o specializovanou variantu algoritmu *MSER* pro detekci písmen. Narozdíl od autorů tohoto přístupu nebudeme, ale algoritmus spouštět na každý kanál vstupu(*BGR, intensity, hue*), ale jenom na *intensity* kanál.

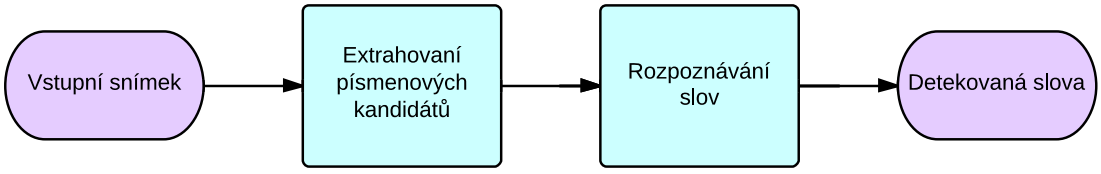
Pro tuto změnu jsme se rozhodli kvůli volbě algoritmu od [22] pro detekci slov. Jedná se o metodu detekce slov se slovníkem, tedy metoda pracuje na základě nějakého jazykového modelu, jak jsme již zmínili 2.3.2. Speciálně Phan a kol. navrhují model, který připouští chyby v detekci písmen, tedy pro detekci slova nepotřebujeme mít vydetekovaná všechna písmena. Nevýhodou tohoto přístupu je větší výpočetní náročnost, zapříčeněná velikostí využitého slovníku. Proto navrhujeme zrychlit algoritmus využitím slovníkové trie, dále také navrhujeme využití vizuálních vlastností informacích o písmenech pro detekci slov.

4.2 Přehled navržené metody

Námi navržená metoda se bude skládat z 2 kroků:

1. **Extrakce potencionálních písmen**
   Lokalizujeme a vysegmentujeme potencionální písmena ze vstupní bitmapy

2. **Rozpoznávání slov**
   Z písmenových kandidátu generujeme slova ze slovníku



Obrázek 4.1: Proces rozponávání textu

4.3 Extrakce potencionálních písmen

Jak již bylo zmíněno v předchozí kapitole, pro extrakci potencionálních písmen budeme využívat algoritmus typu *MSER* [14], jmenovitě použijeme algoritmus Extremal regions publikovaný v článku od Neumanna a Matase [17].

4.3.1 MSER

MSER(Maximally stable extremal regions) je *blob detektor*, algoritmus pro detekování oblastí v bitmapě odlišných od okolí nějakou vlastností, navržený od Matase a kol.[14]. *MSER* detekuje tzv. maximalně stabilní extremní oblasti. což je komponenta souvislosti, která se při prahování pomocí různých θ z nějakého intervalu $P \subset \{0, 1, \dots 255\}$ mění co nejméně, tedy je stabilní.

Pro úplnost si uveďme definice uvedené ve článku od Matase a spol. [14].

Definice

Pro zadefinování pojmu maximally stable extremal region, je nejdříve potřeba zadefinovat pojem extremal region.

Definice 1. *Nechť* ***bitmapa*** I *je zobrazení* $I : D \subset \mathbb{Z}^2 \to S$*, potom můžeme definovat extrémní oblasti, právě tehdy když*

1. *S má uplné uspořádání, tj. existuje binarni relace* $\leq$ *na* S*, které je reflexivní, antisymetrické a tranzitivní*
2. *Máme zadefinovanou relaci* $A \subset D \times D$*, jsou sousedé. V [14] používá 4-adjencency sousedství, tedy* $\forall p, q \in D : pAq \iff \sum_{i=1}^{d} |p_i - q_i| \leq 1$*;*

Definice 2. ***Region*** Q *definujme jako souvislou podmnožinu* D*, kde platí* $\forall p, q \in Q$ *: existuje posloupnost* $p, a_1, a_2, \dots a_n, q$ *a platí že* $pAa_1, a_1Aa_2, \dots a_iAa_{i+1}, a_nAq$*, čili* Q *je komponenta souvislosti*

Definice 3. ***Vnější okruh oblasti*** $\partial Q = \{p \in D \setminus Q \mid \exists q \in Q : qAp\}$*, tj.* ∂Q *je vnější okraj oblasti* Q*, tak že každý jeho pixel má alespoň jednoho souseda v* Q

Definice 4. ***Extrémní oblast*** *je oblast* $Q \in D$*, taková že* $\forall p \in Q, \forall q \in \partial Q$*:* $I(p) > I(q)$ ***(maximální intensity oblast)*** *anebo* $I(p) < I(q)$ ***(minimální intensity oblast)****.*

Definice 5. ***Maximally stable extremal region*** *Nechť* $Q_1, Q_2 \dots Q_n$ *je posloupnost do sebe vnořených Extremal regions, tj.* $Q_i \subset Q_{i+1}$*, potom extremal region* Q_j *je maximally stable extremal region, právě tehdy když funkce stability* $q(i) = |Q_{i+\Delta} - Q_{i-\Delta}|/|Q_i|$ *má lokální minimum v* j*.*

K uvedeným definicím dodejme, že v našem případě pracujeme s grayscale bitmapu, tedy $S = \{0, 1, \dots 255\}$, kde máme jasně definované úplné uspořádání za pomocí binární relace $\leq$.

Nyní si popišme strukturu nazvanou komponentový strom a její využití v detekování *MSERŮ*.

Komponentový strom

Komponentový strom je struktura, která nám kóduje evoluci komponent souvislosti při prahování od 0 do 255, objevením nových komponent souvislosti při prahování a sloučením několika komponent do nové.

Na každém patře takovéhoto stromu se vyskytují buď komponenty, jejíchž pixely mají všechny stejnou hodnotu v bitmapě, v tomto případě mluvíme o listu stromu nebo komponenta, která vznikla sloučením komponent z předchozích pater stromu a nějakých pixelů, jenž se zpřístupnili zvednutím hladiny práhu, taková to komponenta ma za syny komponenty jejichž sloučenim vznikla. Nyní si uveďme formální definici.

Definice 6. *Nechť I je vstupní bitmapa, A je relace sousedství jako v definici 1 a pro $y \in S$ definujme graf $H(y) = (Z_y, F_y)$, kde $Z_y = \{p \in D \mid I(p) \leq y\}$ a $f = (k, l) \in F_y \iff kAl$.*

*Potom **komponentový strom** je orientovaný strom $T = (V, E)$, kde*

- *V je množina všech komponent souvislosti všech grafů $H(\theta)$, kde $0 \leq \theta \leq 255$*
- *$e = (u, v) \in E \iff v \subseteq u \wedge \forall t \in V : v \subseteq t \subseteq u \implies t = v \vee t = u$.*

Formální definice nám říká, že komponentový strom je orientovaný strom, kde vrcholy jsou komponenty souvislosti vzniklé prahováním z bitmapy I. Hrany stromu jsou vždy orientované od větší komponenty k menší. Menší komponenta je zároveň podkomponentou té větší, navíc menší komponenta je vždy přímou podkomponentou té větší, tzn. že mezi nimi neexistuje žádná další komponenta.

Při prahování I pro různá $\theta \in \{0, 1 \ldots 255\}$ dostáváme různé $H(\theta)$, na kterých lze provádět analýzu souvislých komponent, dle relace A uvedené v definicích, navíc při postupném prahování od $0, 1, \ldots 255$ můžeme pozorovat evoluci komponent, evolucí rozumíme vznik nových komponent díky zvýšení práhu nebo sloučením 2 a více komponent do jedné díky zvýšení práhu. Tuto evoluci lze zachytit tzv. komponentovým stromem a to následujícím způsobem, mějmě situaci kdy zvedáme práh od θ k $\theta + 1$, potom v případě že se v H$(\theta + 1)$ objeví nová komponenta C, která je *minimum intensity region*, stává se novým listem stromu. V případě, že vznikla nová komponenta C, sloučením dvou a víc komponent $C_1, C_2 \ldots C_n$ z $H(\theta)$, vytvoříme pro novou komponentu C nový uzel ve stromě a připojíme ho jako rodiče k uzlům komponent $C_1, C_2 \ldots C_n$. Kořenem komponentového stromu je uzel vzniklý z komponenty při prahování na urovni 255, protože $H(255) = D$ a tedy $H(255)$ má jenom jednu komponentu souvisloti a to bitmapu celou.

Nechť $T = (V, E)$ je komponentový strom, splňující výše uvedenou definici potom dokažme následující tvrzení:

Pozorování 1. *Nechť $C = (V_c, E_c)$ je komponenta souvislosti vzniklá prahováním a v komponentovém stromě se jedná o list, pak platí že $\forall p \in V_c(p) : I(p) = \theta$ kde $\theta \in \{0 \ldots 255\}$*

Důkaz. Pro spor nechť existuje list $C = (V_c, E_c)$ pro který platí že $\nexists \theta : \forall p \in V_c(p) : I(p) = \theta$, pak nechť $\gamma = \min\{I(p) \mid p \in V_c\}$, pak ale existuje komponeta $B \in V \wedge B \subseteq C$, tedy dle definice komponentového stromu C nemůže být listem. SPOR □

Pozorování 2. *Pro každou komponentu souvislou C vzniklou prahováním na hladině θ, která je vnitřním vrholem komponentového stromu, platí že $C = \bigcup_i C_i \cup \{p \mid I(p) = \theta\}$, kde C_i jsou synové komponenty C ve stromovém stromě.*

Důkaz. $\bigcup_i C_i \subseteq C$ je zřejmé z definice. Stačí dokázat $\forall p \in C \setminus \bigcup_i C_i : I(p) = \theta$.

Pro spor předpokládejme že existuje $K = \{p \mid p \in C \wedge \forall C_i : p \notin C_i \wedge I(p) < theta\}$, potom ale existuje komponenta D s pixely $V_d = K \cup \{C_i \mid \exists p, q : p \in K \wedge q \in C_i \wedge pAq\}$, která vznikla vznikla prahováním na hladině $\gamma = \max\{p \mid p \in K\}$. čili někteří synové C_i nejsou přímými následníky. SPOR s definicí komponentového stromu.

□

Toto tvrzení nám dokazujeme, že formální definice splňuje požadavek na vnitřní uzly stromu zmíněný na začátku.

Pozorování 3. *Pixely každé komponenty v komponentovém stromě jsou minimální intensity oblastí.*

Důkaz. $\forall C = (U_c, F_c) \in V : C$ je komponenta souvislosti $H(\theta)$, kde $\theta \in \{0 \dots 255\}$. Pak platí že $\forall q \notin U_c : qAp$, kde $p \in U_c$, platí že $I(q) > \theta$, jinak by q muselo patřit do komponenty C.

Nyní položme oblast $Q = U_c$ a $\partial Q = \{q | q \notin U_c \wedge qAp\}$. Pro $Q, \partial Q$ plyne, že $\forall p \in Q, q \in \partial Q : I(p) \leq \theta < I(q)a$, což je definice minimální intensity oblasti. □

Díky pozorování 3. víme, že z komponentového stromu $T = (V, E)$ lze detekovat *MSERY*, mezi minimálnímy intensity oblastmi. Protože pro každou cestu $p = (C_1, C_2, \dots C_n)$ v T platí že $C_1 \supseteq C_2 \cdots \supseteq C_n$, a následně lze s cesty p vybrat dle definice *MSER* dle jeho výše uvedené definice. Dále si všimněme, ještě v případě z invertování bitmapy I, se maximální intensity oblasti, stávájí minimálníma oblastma. Z tohoto plyne, že pro detekci *MSERŮ* mezi maximálníma intensity oblastma, stačí zinvertovat vstupní bitmapu a následně z ní postavit komponentový strom a vydetekovat v něm *MSERY*.

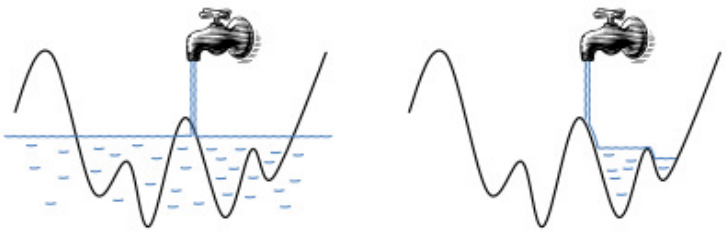
Využití komponentového stromu V této sekci jsme vysvětlili způsob využití komponentového stromu v algoritmu pro detekci *MSER*. Algoritmus pro detekci *MSERU* však není jedinou aplikací komponentového stromu, tato struktura se dá dále využít při registraci obrázků, kompresi snímků a visualizaci dat [15].

Zavedli jsme si pojem komponentový strom a ukázali jeho aplikaci v detekci *MSERU*. Nyní již zbývá jenom popsat algoritmus, který takovýto strom postaví z bitmapy.

Algoritmus

V předchozí části textu, byl popsán komponentový strom, z kterého lze detekovat *MSERY*. Tento strom je využit ve většině algoritmů pro detekci *MSERU*, které probíhají ve dvou fázích:

1. stavba komponentového stromu
2. traversování komponentového stromu a výběr komponent, splňující požadavek stability viz. definice maximally stable extremal region



Obrázek 4.2: Rozdíly mezi klasikým watershed a algoritmem od Nisteriuse a kol

Výběr *MSERU* z již postaveného stromu je ve všech algoritmech podobný, stačí projít strom, pro každý uzel spočítat hodnotu funkce stability v závislosti na daném Δ. Následně ze stromu vybere ty uzly, které reprezenují komponenty mající lokální minimum funkce stability. Okolí uzlu U mohou například tvořit všechny uzly vzdálené méně než Δ od U.

Čím se jednotlivé algoritmy tedy liší je způsob stavby stromu. Pro vybudování komponentového stromu lze použít zaplavování bitmapy, jedná se o tzv. *watershed* algoritmy [24],[15]. Nechť máme bitmapu jako dvourozměrnou funkci $I : D \subset \mathbb{Z}^2 \rightarrow \{0, 1, \ldots 255\}$. Graf této funkce má tvar kopcovité krajiny, kde jednotlivé kopce mají výšku nejvýše 255. *Watershed* algoritmy pak tuto krajinu postupně zaplavují, tzn. že nejdříve máme vodu ve výšce 0, potom zvedneme hladinu na výšku 1 a takto pokračují dokud voda není ve výšce 255. Pomocí *union findu* pozorujeme vývoj komponent souvislosti zaplavených částí, vznik nových zaplavených části a spojení dvou nebo více zaplavených částí do jedné, který zaznamenáváme v komponentové stromě. Tyto algoritmy mají skoro lineární čas. V roce 2008 byl ale Nistériem a Stewéniusem v publikaci *Linear MSER* [19] představen první lineární algoritmus pro budování komponentového stromu. Představme si, že opět máme bitmapu jako 2D funkci, jejímž grafem je kopcovitá krajina. V libovolném bodě krajiny začneme rozlévat vodu a opět zaznamenáváme vývoj zaplavených částí do komponentového stromu, dokud nemáme zaplavenou celou bitmapu. Narozdíl od *watershed* algoritmů, kde se krajina zaplavuje postupným zvyšováním hladiny, Nistériova a Steweniusova metoda nejdřív zaplaví nejbližší okolí bodu rozlevání, voda se nejdříve dostane na nejnižší bod okolí, tedy teče po svahu dolů. Jakmile doteče do nejnižšího bodu okolí, začne ho postupně naplňovat, dokud nedosáhne výšky, kde se bude opět moci rozlít dolů po svahu. Takto algoritmus pokračuje dokud nedosahne nejvyšší hladiny.

Pro budování komponentového stromu to znamená, že v případě watershed algoritmů stavíme komponentový strom od spodních pater a při použití *Linearního MSERU* stavíme stromem jakoby průchodem do hloubky.

Linearní MSER Vstupem algoritmu je bitmapu $I : D \subset \mathbb{Z}^2 \rightarrow \{0, 1, \ldots 255\}$, na niž je definována relaci sousedství relaci A jako v *definici 1.*, dále řekněme, že jestliže pAq, kde $p, q \in D$ potom pixely p a q spojuje hrana.

Linearní MSER potřebuje následující datové struktůry:

- Binarní masku již zaplavených pixelů
- Prioritní frontu pro okrajové pixely, které již jsou zaplavené ale ještě nebyly přidány do žádné komponenty, spolu s pixelem si pamatujeme hranu, která vede k prvnímu nenavštívenému sousedovi. Priorita je $-$intenzita pixelu.

- Zásobník pro uzly stromu reprezentující komponenty, pro běh algoritmu je nutné aby si uzel pamatoval, grayscale level ve kterém byl přidán na zásobník, svého otce a syny. Další infomace jsou volitelné. V našem případě potřebujeme pixely patřící komponentám, ale někdy si stačí pamatovat jen různé geometrické vlastnosti komponent.

Dále uveďme pseudokód algoritmu.

Algoritmus 2: Linear MSER

vstup : Vstupní bitmapa $I : D \subset \mathbb{Z}^2 \to S = \{0, 1, \ldots 255\}$
výstup: Komponentový strom bitmapy

```
1  initializuj binární masku accesibleMask;
2  initializuj priritní frontu okrajových pixelů boundaryPointsQueue;
3  initializuj zásobník pro komponenty boundaryPointsQueue;
4  vlož do componentStack novou komponentu s levelem, který bude vyšší než
   všechny ostatní, 256 například; ;
5  zvol sourcePixel, lze libovolně;
6  currentPixel ← sourcePixel, currentEdge ← první hrana sourcePixel,
   currentLevel ← I(sourcePixel),
7  označ currentPixel jako navštívený v accesibleMask;
8  vlož do componentStack prázdnou komponentu s currentLevel ;
9  for hrana in currentEdge to last edge do
10 |  neighbour ← pixel do kterého vede hrana;
11 |  if označen neighbour v accesibleMask then
12 |  |  continue
13 |  end
14 |  označ neighbour v accesibleMask;
15 |  level ← I(neighbour) ;
16 |  if level ≥ currentLevel then
17 |  |  vlož neighbour, a jeho prvni hranu do boundaryPointsQueue;
18 |  else
19 |  |  vlož currentPixel a hranu currentEdge + 1 do boundaryPointsQueue;
20 |  |  currentPixel ← Neighbour ;
21 |  |  currentEdge ← první hrana ;
22 |  |  currentLevel ← level;
23 |  |  goto řádek 8;
24 |  end
25 end
26 přidej currentPixel je komponentě na vrcholu componentStack;
27 if boundaryPointsQueue je prázdná then
28 |  root ← vrchol componentStack;
29 |  componentStack.pop();
30 |  return root
31 else
32 |  pixel,edge ← boundaryPointsQueue.top();
33 |  level ← I(pixel);
34 |  if level = urovni componentStack.top() then
35 |  |  goto řádek 9 ;
36 |  else
37 |  |  ProcessStackSubRoutine(level);
38 |  |  goto řádek 9 ;
39 |  end
40 end
```

Algoritmus 3: Process stack sub-routine

```
1  Function ProcessStackSubRoutine(newPixelLevel)
   repeat
2      child = componentStack.top();
3      componentStack.pop();
4      if newPixelLevel < topStackLevel then
5          vlož do componentStack komponentu s newPixelLevel urovní;
6          připoj child ke componentStack.top() jako dítě;
7          return
8      end
9      připoj child ke componentStack.top() jako dítě;
10 until newPixelLevel > topStackLevel;
```

Jak vidíme z pseudokodu algoritmus se inicializuje a začne rozlévat vodu s libovolného pixelu p, ten se vloží do prioritní fronty dále se vloží do zásobníku komponentu obsahující pixel p, která by zároveň vznikla při prahování hladiny hodnotou $I(p)$.

Následně vždy vybereme vrchol této haldy koukneme se na jeho neprozkoumaného souseda, v případě že tento soused ma nižší hodnotu než je hodnota vrcholu fronty, tak se nastaví nový vrchol prioritní fronty a vloží se nová komponenta do zásobník, následně se začne se zpracovávat jeho neprozkoumaní sousedé, jinak jestliže má sousední pixel stejnou nebo vyšší hodnotu, potom jej vložíme do prioritní fronty, kde bude čekat na zpracování. Tato operace v naší analogii znamená že voda teče dolů z kopce nebo po hladině.

Algoritmus tímto způsobem doteče až na úplné dno, v tomtu dnu se vytvoří komponenta, která bude list v komponentovém stromu a přidají se do ní všechny její pixely, tzn. že jsme prozkoumali všechny jejich sousedy. Následně ve chvíli kdy na vrholu haldy bude pixel nepatřící do dna, tak je potřeba se zvednou na jeho hladinu. K tomuto slouží procedura *ProcessStackSubRoutine*. V této proceduře se staví komponentový strom ze zdola z komponent na zásobníku. Strom se zde staví dokud se nedostaneme na úroveň hladiny pixelu na vrcholu haldy, nebo se zjistí, že pro tento pixel ještě nebyla přidána komponenta na zásobník, hladina komponenty na vrcholu zásobníku je větší než hladina pixelu na vrcholu haldy, poté tedy přidáme na zásobník novou komponenty hladiny pixelu v haldě a algoritmus se vrátí z *ProcessStackSubRoutine*. V obou případech se pokračuje opět hledáním nového dna. Tato část algoritmus v naší anologii představuje plnění rokliny vodou dokud nedosáhne opět výšky, ve které bude moci téct dolů po svahu.

Implementační detaily Jak již bylo zmíněno v našem případě si chceme v uzlech pamatovat pixely, patřící komponentě daného uzlu. Jelikož otcovská komponenta vznikla spojením několika dětí, tak k otcovské komponentě patří všechny pixely synů. Proto je potřeba vyřešit rychlé sdílení pixelů mezi otcem a synem. Jako vhodné řešení se jeví použití spojových seznamů pixelů, Při přepojení k rodiči stačí jeden spojový seznam pixelů syna připojit za seznam otce.

Vstupní bitmapu obalíme jednou vrstvou pixelu navíc a všechny nově přidané pixely označíme v masce jako zaplavené, abychom se vyhnuly testu jestli soused není mimo bitmapu při procházení sousedních pixelů.

Autoři *Lineárního MSERU* dále navrhují místo implementace využití klasických hald, navrhují haldu implementovat pomocí pole 256 zásobníků. Kde prioritu kóduje index zásobníku v poli. Následně tedy minimum haldy je vždy neprázdný zásobník s nejmenším indexem. Haldu lze následně postavit pouze nad těmito zásobníky, kde priorita je index zásobníků.

V praxi většinou ještě mezi detekovanými MSERY, zamítájí ty které jsou moc velké nebo malé a ty které mají nízkou funkci stability.

Pro seznámení s naší implementací si prohlédněte 9.4.1.

Analýza Uveďme ještě rozbor časové a paměťové složitosti algoritmu uvedený v publikaci od Nistéria a Stewénia [19] v sekci 3.

Tvrzení 1. *Algoritmus postaví komponentový strom v $O(n)$, kde n je počet pixelů bitmapy.*

Důkaz. V prioritní frontě máme vždy přesně jednu kopii pixelu a tato kopie může být nejvýše k krát vložena a vrácena do haldy, kde k je počet sousedů pixelu.

Počet komponent je lineární vůči počtu pixelů, protože každá komponenta musí obsahovat aspoň jeden pixel kterým se liší od svých synů v komponentovém stromě, tedy zpracování všech komponent zabere $O(n)$ času.

Protože dominantní operací je insert a deleteMin na prioritní frontě, která zaberou $log(m)$, kde m je velikost rozsahu priorit, tedy 256. Poté lze odhadnout časovou složitost pomocí

$$O((n+e) * log(m))$$

kde e je počet hran mezi sousedy. Jelikož počet hran $e \approx 2n$ platí :

$$O((n+e) * log(m)) = O(3n * log(256)) = O(n)$$

□

Tvrzení 2. *Paměťová složitost algoritmu je $O(n)$.*

Důkaz. Binární maska již zaplavených pixelů má velikost n, kde n je počet pixelů. Z předchozího tvrzení víme, že počet komponent je lineární a v prioritní frontě máme vždy právě jednu kopii daného pixelu, takže její velikost je také lineární. □

4.3.2 Extremal regions

V předchozí části jsme popsali algoritmus *MSER*, který detekuje oblasti bitmapy, které se mění co nejmíň během postupného prahování bitmapy.

Písmena většinou tuto podmínku splňují, ale nemusí. Například písmeno které je odstíněno nebo má více barev není detekováno jako celek algoritmem *MSER*. *MSER* detekuje i velké množství objektů, které písmeno nejsou potom je zapotřebí provést dodatečného filtrování již detekovaných oblastí, což zapříčíní zvýšení výpočetní náročnosti.

Z těchto důvodů byl Neumannem a Matasem navržen algoritmus [17], který z komponentového stromu detekuje přímo písmena místo *MSERU*. Výběr písmenových komponent, čili extremal regionu dle pozorování, probíhá 2 fázovou klasifikací. V první fázi se pro každou extremní oblast r v komponentovém stromě spočítají $P(r|pismeno)$, následně se z něho extrahují lokální maxima a ty

jsou předány do 2 fáze, kde probíha klasická klasifikace na písmena a nepísmena. Neumann a Matas [17] zvolili 2 fázovou klasifikaci z důvodů ušetření výpočetní složistosti. V první fázi se totiž použijí pouze deskriptory, které lze incrementálně vypočítat již za stavby stromu. Ve druhé fázi se k těmto inkrementálním deskriptorům přidají descriptory, které již nelze vypočítat incrementálně ale za to jsou silnější. Dvou fázový klasifikátor připomíná princip kaskádovitého klasifikátoru navrženého v práci Violy a Davise [25]. V následující části si popíšeme jednotlivé fáze a descriptory, použité deskriptory se až na detaily neliší od původní práce Matase a Neumanna [17], na odlišnosti upozorníme.

Definice

Převezměme veškeré definice uvedené v předchozí části o *MSERU*, akorát nyní budeme považovat extrémní oblast pouze minimální intensity oblast.

Dále si zaveďme notaci použitou v původní práci [17], *extrémní oblasti* budeme zkráceně říkat *ER*.

Definice 7. *Nechť R_θ je množina všech ER na hladině θ. Řekněme, že ER r* ***vznikla na hladině*** *θ, právě tehdy když $r = (\bigcup u \in r_{\theta-1}) \cup (\bigcup p \in d : i(p) = \theta)$.*

Dále definujme pro ER r funkci $\phi(r)$ jako její ***vypočítané deskriptory****, pro pixel p* ***inicializační deskriptor*** *$\varphi(p)$ a operaci $\oplus$ zajišťující* ***kombinaci deskriptorů ER nebo skupiny pixelů****.*

Potom nechť máme ER $r \in R_\theta$ a $r = (\bigcup u \in R_{\theta-1}) \cup (\bigcup p \in D : I(p) = \theta)$ máme deskriptor ϕ a vypočítáné $\phi(u)$, kde $u \in R_{\theta-1}$ potom deskriptor ϕ nazveme ***inkrementálně vypočítatelný*** *právě tehdy, když existuje dobře definována φ pro pixely, operace $\oplus$ a $\phi(r) = (\oplus\phi(u)) \oplus (\oplus\varphi(p))$.*

Nyní si uveďme incrementální deskriptory použité v algoritmu [17]

- **Plocha** *a*, udává počet pixelů daného *ER*. Pro pixel je $\varphi(p) = 1$ a $\oplus$ je realizována pomocí sčítání.
- **Bounding Box** uspořádáná čtveřice $(x_{min}, y_{min}, x_{max}, y_{max})$. Pro pixel p platí $\varphi(p) = (x, y, x + 1, y + 1)$, kde x a y jsou souřadnice pixelu. Operace $\oplus$ je realizována pomocí kvarteta funkcí (min, min, max, max) aplikovány po dvojici čtveřic po odpovídajících složkách.
- **Eulerovo číslo** η, topologická vlastnost binární obrazu BI komponenty, $\eta = \#\text{komponent BI} - \#\text{děr BI}$ Pro výpočet η z BI se dá použít algoritmus v od Graye [11]. Skenujeme obrázek oknem 2×2 a počítáme počty výskytu následujících quadů.

$$Q_1 = \left\{\begin{matrix}1 & 0\\ 0 & 0\end{matrix}\right\}, \left\{\begin{matrix}0 & 1\\ 0 & 0\end{matrix}\right\}, \left\{\begin{matrix}0 & 0\\ 1 & 0\end{matrix}\right\}, \left\{\begin{matrix}0 & 0\\ 0 & 1\end{matrix}\right\}$$

$$Q_3 = \left\{\begin{matrix}1 & 0\\ 1 & 1\end{matrix}\right\}, \left\{\begin{matrix}0 & 1\\ 1 & 1\end{matrix}\right\}, \left\{\begin{matrix}1 & 1\\ 1 & 0\end{matrix}\right\}, \left\{\begin{matrix}1 & 1\\ 0 & 1\end{matrix}\right\}$$

$$Q_D = \left\{\begin{matrix}1 & 0\\ 0 & 1\end{matrix}\right\}, \left\{\begin{matrix}0 & 1\\ 1 & 0\end{matrix}\right\}$$

Nechť C_i udává počet výskytu quadu Q_i. Potom při 4-adjencency sousedství je $\eta = \frac{1}{4}(C_1 - C_3 + 2C_D)$. Potom při přidání pixelu, stačí detekovat změnu počtu quadu při přidání pixelu ke komponentě, tedy $\varphi(p) = \frac{1}{4}(\Delta C_1 - \Delta C_3 + 2\Delta C_D)$, operace $\oplus$ je sčítání.

- **Perimetr komponenty** p, zde se lišíme od algoritmu, místo navrhované metody v [17], používáme algoritmus uvedený opět v Prattovi na též stránce. Algoritmus opět funguje na základě počítání výskytu quadu. Protentokrát ale navíc detekujeme quad:

$$Q_2 = \begin{Bmatrix} 1 & 1 \\ 0 & 0 \end{Bmatrix}, \begin{Bmatrix} 0 & 1 \\ 0 & 1 \end{Bmatrix}, \begin{Bmatrix} 0 & 1 \\ 1 & 1 \end{Bmatrix}, \begin{Bmatrix} 1 & 0 \\ 1 & 0 \end{Bmatrix}$$

  Nechť C_i a ΔC_i jsou definováno jako u eulerova čísla. Pro pixel p je potom $\varphi(p) = \Delta C_2 + \frac{1}{\sqrt{2}}(\Delta C_1 + \Delta C_3 + \Delta 2C_D)$ a kombinace descriptorů je opět sčítání.

- **Horizontal crossing** c_i, udává počet přechodů mezi komponentovým pixelem a nekomponentovým pixelem ve výšce i *ER*. Pro pixel p je inicializační funkce $\varphi(p)$ rovna 0, jestliže pixely napravo a nalevo jsou komponentové, rovna 1 jestli jenom jeden horizontální soused je komponentový a 2 jestliže oba horizontální sousedi jsou nekomponentové. Kombinace descriptorů $\oplus$ je sčítání pro společné řádky a nespolečné řádky se jenom přidají. Pro uchovování c_i je potřeba zvolit vhodnout datovou strukturu, která umožnuje rychlé přidávání hodnot na konec a začátek. Vhodná volba je dle autorů algoritmu deque implementovaný pomocí pole.

Tyto deskriptory počítáme při stavbě stromu za použití Nistériova a Stewéniusova algoritmu. Jelikož hodnoty pro jednotlivé komponenty updatujeme při spojování komponent do jedné a přidávání bodů ke komponentě, jsou tyto deskriptory dopočítány pro všechny komponenty v $O(n)$ čase, kde n je počet pixelů bitmapy.

1. fáze klasifikace Po vybudování stromu a vypočítání incrementálních deskriptorů pro každý *ER* v komponentovém stromě, budeme extrahovat potencionální *ER*, která mohou být písmena. Pro výběr budeme nejdříve vypočítáme pro každý *ER* r ve stromě $P(r \mid charakter)$, kde za charakter považujeme malá, velká písmena a čísla. Na základě těchto pravděpodobností extrahujeme ze stromu ty *ER* r, která mají lokální maximum pravděpodnosti $P(r \mid charakter)$, velikost okolí je opět určena uživatelem v závislosti na parametru Δ stejně jako v klasickém *MSERU*. Navíc dále zamítáme *ER* r pro které platí $P(r \mid charakter) \leq p_{min}$ a ty *ER*, kde rozdíl lokálního maxima a minima v okolí je menší než Δ_{min}. Oba parametry p_{min}, Δ_{min} jsou dána algoritmu na vstupu. Všimněme si že extrakce kandidátu, je téměř identický s výběrem *MSERŮ* z komponentového stromu, až na rozdílné funkce a parametry.

Pro vyčíslení $P(r \mid charakter)$ pro *ER* r použijeme klasifikátor *Real AdaBoost* s rozhodovacími stromy, nyní ukažme schéma klasifikátoru.

Použitý klasifikátor: ***Real AdaBoost***

název přiznaku	vzorec
velikost uhlopříčky	$width/height$
kompaktnost	$\sqrt{a}/p$
počet děr	$1-\eta$
horizontal crossing medián	$med\{c_{\frac{1}{6}}, c_{\frac{3}{6}}, c_{\frac{5}{6}}\}$

Poznamenejme že *width* je šířka hraničního obdelníku, *height* je jeho výška, a je plocha komponenty(počet jejích pixelů), p je délka perimetru, η je eulerovo číslo komponenty a c_i je horizontal crossing ve výšce i.

Výstup klasifikátoru kalibrujeme na pravděpodobnostní výstup dle logistické korekce [18].

Poznamenejme, že Neumann a Matas experimentálně vyčíslili hodnoty $p_{min} = 0.2$ a $\Delta_{min} = 0.1$.

2. fáze klasifikace Protože deskriptor použitý v první fázi není dost informativní, je potřeba přidat v druhé fázi k descriptoru d další přiznaky, které jsou více informativní. Protože byl výrazně redukován počet *ER* z komponentového stromu, lze přidat descriptory, které již nejsou incrementálně vypočítatelné.

poměr plochy děr (a_h/a), kde a_h je plocha děr.

poměr plochy konvexního obalu komponenty a komponenty (a/a_c), kde a_c je plocha konvexního obalu.

počet inflexí na okraji komponenty písmena mají malý počet inflexí, narozdíl od jiných objektů.

Pro každý *ER* vypočítáme uvedené deskroptory a přidáme je k těm z první fáze. Vzniknutý deskriptor je následně použit při klasifikaci *ER* na písmena a nepísmena. Pro klasifikaci využíváme SVM *(Support Vector Machine)* s RBF jádrem. Opět uveďme schéma klasifikace.

Použitý klasifikátor: ***SVM s RBF jádrem***

název přiznaku	vzorec
velikost uhlopříčky	$width/height$
kompaktnost	$\sqrt{a}/p$
počet děr	$1-\eta$
horizontal crossing medián	$med\{c_{\frac{1}{6}}, c_{\frac{3}{6}}, c_{\frac{5}{6}}\}$
poměr plochy děr	a_h/a
poměr plochy konvexního obalu ke komponenty	a/a_c
počet inflexí na okraji komponenty	$inflections$

Odlišnosti od původního algoritmu V publikaci od Neumanna a Matase je výše popsaný algoritmus současně spuštěn nejen na grayscale formě obrázku, ale i na všech barevných kanálech, hue a saturation kanálech vstupu, kvůli zvýšení úspěšnosti detekce. My se ale rozhodli že díky námi zvolené metodě pro detekci slov, která umí tolerovat nedokonalou detekci písmen, současné spouštění na více kanálech vstupu nepotřebujeme.

(a) vstupní bitmapa (b) 1. fáze ER filtrace (c) 2. fáze ER filtrace

Obrázek 4.3: Ukázky jednotlivých fází ER filtrace

Non Max Suppression Obvykle při výstupu z algoritmů typu *MSER*, dostaváme ER, které se navzájem překrývají, a zároveň reprezentují jedno písmeno na obrázku. Ze všech takových to množin je potřeba, vybrat jeden *ER*, který bude reprezentovat dané písmeno. Tento problém je velice podobný max suppresion ve Sliding window přístupu.

Pro Non Max suppresion využijeme metodu navrhovanou ve článku [22], non max supprese je součástí metody pro detekování slov.

Implementace Pro seznámení s naší implementací doporučejeme si prohlédnout sekci 9.4, zvláště pak části 9.4.2.

4.4 Generování slov

Pro generování ze slov budeme používat slovník. Slovník nám výrazně pomahá zlepšovat výsledky metod. Slovník nám pomáhá tyto chyby redukovat. Poskytuje totiž jakýsi kontext pro detekci.

Většina metod se slovníkem přijíma na vstupu kandidáty na písmena $C = \{c_1, c_2 \dots c_n\}$, potom provede ocr fázi a non max suppresi v nějakém pořadí. Úkolem ocr fáze je vyčíslit $P(l_u \mid c_i)$, pravděpodobnost znaku l_u z abecedy slovníku za podmínek kandidáta c_i. Non max suppresion redukuje navzájem překrývají se komponenty reprezentující stekné písmeno. Tato operace je nezbytná, jelikož překrývající komponenty mohou negativně ovlivnit výstup metody. Následně metoda pracuje s jazykovým modelem na základě vyčíslených pravděpodobností $P(l_u \mid c_i)$.

Pro generování slov se slovníkem použijeme námi modifikovaný algoritmus od Phana a kol. [22]. Nejdříve si popíšeme původní algoritmus a následně provedené modifikace.

4.4.1 Vyčíslení $P(l \mid c)$

Pro vyčíslení $P(l \mid c)$ se využívá *OCR* s pravděpodobnostními výstupy pro každé písmeno.

V originální práci od Phana a kol. [22] se pro získání lexikografické informace využívá *Bag of Words(BoW)* framework [3] v kombinaci s deskriptorem *DSIFT*, což je periodicky extrahovaný *SIFT*.

Bag of Words přístup *BoW* model využívá vizuálního slovníku. Ten se vytvoří tak, že ze všech trénovacíh vzorků vypočítáme *DSIFT* deskriptory v jednotlivých key pointech, v případě *DSIFT*, se jedná o všechny body na mřížce. Z vypočítané množiny deskriptorů sestavíme vizuální slovník velikosti n, pomocí clustrovacího algoritmu *KMeans*, kde počet klastrů je n. Následně se z každého trénovacího vzorku vypočítá deskriptor jako histogram frekvence výskytu vizualních slov v jednotlivých vzorcích, tedy pro každý vzorek se opět vezmou jeho extrahovaného keypoint dekriptory, které zařadíme do jednotlivých klustorů (vizuálních slov), z frekvence počtu zařazených keypointů dekriptorů do jednotlivých klasterů získáme výsledný deskriptor, tkz. *BoW* deskriptor. Extrahované *BoW* deskriptory se používají k vytrenování více třídní klasifikaci pomocí *SVM*. Při klasifikaci se využívá stejný způsob. Nejdříve se pomocí vizuálního slovníku vypočte *bag of words* deskriptor a poté se provede klasifikace pomocí *SVM*. Poznamenejme ještě, že tento způsob je inspirován *Bag of words* algoritmem v oblasti klasifikace dokumentu a že popsaná varianta využivaná v počítačovém vidění lze také nalézt pod názvem *Bag of Features*.

Původní OCR Jak již bylo řečeno Phan a kol. využili pro *OCR* fázi kombinaci *DSIFT* s *BoW* Z výkonnostních důvodů se autoři rovněž použít rychlejší způsob algoritmus klasifikace než klasické SVM a známý jako *Fast Intersection Kernel* publikovaný od Malika a spol. v roce 2008 [13].

Phan a kol. tento deskriptor dopodrobna zkoumali, zjistili že se jedná o velice robustní deskriptor dosahující rotační invariance bez využití rotovaných tréno-

vacích dat, dále experimentálně došli k závěru, že optimální velikost vizuálního slovníku a tedy i deskriptoru je 3000.

Dle našeho názoru tento deskriptor není ale příliš praktický. Takto veliká dimenze deskriptoru zapříčíní vysokou časovou i pamětovou náročnost klasifikace. V našich pokusech měl vygenerovaný konfigurátor *SVM* velikost zhruba 2 GB. Zdůrazněme, že velikosti konfiguračního souboru pro *SVM* se zřejmě dala snížit, například využitím jiné formulace *SVM* pro více tříd.

Místo toho navrhujeme použít jednodušší *OCR* s *SVM* a *direction histogramem(DH)* navrženým v práci A. Gonzáleze a kol. [10].

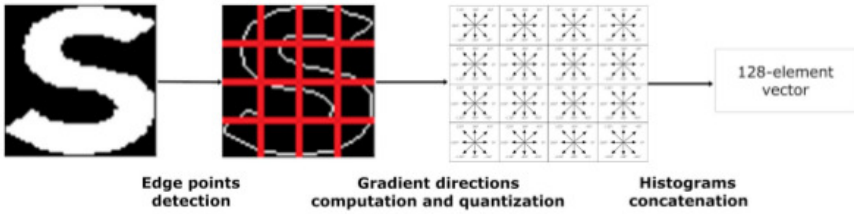
OCR s direction histogramem Námi navržené *OCR* bude využívat *SVM* s klasickým *rbf* jádrem a jako deskriptor využijume již zmíněný *DH*. Jedná se o velice jednoduchý deskriptor, využívající binární obrázek kandidáta a 8-bin histogramy.

Direction histogram Nejdříve si uveďme pseudokód výpočtu *DH*. Vstupem je kandidáta c.

Algoritmus 4: Direction Histogram, vstup kandidát c

```
1  binaryBitmap ← c.binaryMask().resize(128, 128)
2  edgeBitmap ← Canny(binaryBitmap)
3  gradientBitmap ← Sobel(binaryBitmap)
4  directionHistogram ← []
5  for i = 0; i < 128; i += 32 do
6  |   for j = 0; j < 128; j += 32 do
7  |   |   eightBinHistogram ← eightBin(edgeBitmap,gradientBitmap,i,j)
8  |   |   directionHistogram.concatenate(eightBinHistogram)
9  |   end
10 end
11 return directionHistogram
12
1  Function eightBin(edgeBitmap, gradientBitmap, i, j)
   eightBinHistogram ← {0, 0, 0, 0, 0, 0, 0, 0}
2  for k = i; k < i + 32; ++k do
3  |   for l = j; l < j + 32; ++l do
4  |   |   if edgeBitmap(k, l) ≠ 0 then
5  |   |   |   bin ← gradientBitmap(k, l)/45
6  |   |   |   eightBinHistogram(bin) += 1
7  |   |   end
8  |   end
9  end
10 return eightBinHistogram
```

Jak jsme již zmínili *DH* se počítá s binárního snímku kandidáta, který je přímým produktem segmentace pomocí metody *ER* nebo *MSER*. Na tomto binárním snímků nejprve provede výpočet hran pomocí Cannyho Edge detektoru a výpočet směru gradientu pomocí Sobelova filtru. Následně vidíme na řádku 5, že se binární snímek rozdělí na bloky o rozměrech 32×32. Na každém takovém



Obrázek 4.4: Direction histogram

to bloku se počítá tkz. *8 bin* histogram. Výsledný histogram dostaneme spojením všech *8 bin* histogramů jednotlivých aloků.

8 bin histogram je histogram frekvencí směru gradientu v 8 intervalech. Jedná se o intervaly od 0 do 360 stupňů po 45 stupních. *8 bin* pro *DH* počítá tak, že v každé hraně, spočítáme do jakého intervalu patří gradient hrany a následně zaktualizujeme výsledný histogram, více viz kód funkce *eightBin*.

Poznamenejme akorát, že Sobel počítá směr gradientu v souřadnici x a souřadnici y. Směr gradientu lze následně vypočítat pomocí funkce $atan2$.

Délka výsledného histogramu je 128, protože vstupní bitmapa byla resizovaná na rozměry 128×128 a na této resizované bitmapě jsme počítali *8 bin* z bloků velikosti 32×32, tedy máme 16 histogramů po 8 hodnotách, dohromady jeden deskriptor dimenze 128.

Uveďme ještě pro kompletnost pseudokód *OCR* fáze.

Algoritmus 5: Vyčíslení $P(l \mid c)$

```
1 foreach kandidát c do
2 |   directionHistogram ← computeDirectionHistogram(c);
3 |   lexikographicInformation ←
  |   svm.computeProbabilities(directionHistogram);
4 end
```

Nerozlišitelná písmena Pro OCR fázi jsme se rozhodli nerozlišovat třídy klasifikaci mezi následujícími skupinkami písmen a číslic *oO0, il1I, cC, jJ, pP, sS, uU, vV, wW, xX, zZ*. Tedy pro trénování ocr je například písmeno z a Z ve stejné třídě.

Tato změna byla zakomponentována, protože ve vyjmenovaných skupinách lze těžko rozlišit mezi charaktery ve skupinách. Například c a C jsou nerozlišitelná pro naše *ocr*. Pro rozlišení mezi nima bychom museli využít další informaci, například výšky ostatních písmen ve slově.

Nyní když jsme si představili námi navrhované *OCR* je na čase detailněji rozebrat algoritmus pro detekování slov od Phana a kol. [22].

4.4.2 Původní algoritmus

Vstupem algoritmu navrženém Phanem a spol. [22] jsou kandidáti $C = \{c_1, c_2, \ldots c_n\}$. Algoritmus pracuje s abecedou $L = \{a - z, A - Z, 0 - 9\}$, každému znaku $l \in L$

budeme říkat label. Výstupem jsou slova, která lze optimálně poskládat z kandidátů. Konfigurací budeme rozumět složení slova w ze slovníku pomocí kandidátů z C.

V původní práci [22] byl pro detekci písmen použit *MSER* [14]. My ale využíváme lepší algoritmus *Extremal Regions* [17], takže kandidáti C jsou množinou extrahovaných ER.

NonMax supprese na základě confidence komponenty

Nonmax suppresion provedeme na základě confidence pro komponentu $c \in C$

Definice 1. ***Věrohodnost***

$$confidence(c) = \max_{l \in L} P(l \mid c), c \in C$$

Věrohodnost nám vyjadřuje jak moc je pravděpodobné je daný kandidát písmeno. Pozorujme, že jestliže je kandidát artefakt, potom nemá zpravidal confidence moc vysoké, protože není jasně zařaditelný k žádnému labelu abecedy.

V předešlé sekci jsme uvedli, že vybereme rotaci, která bude nejoptimálnější. Za takovou rotaci považujeme tu, při které má kandidát největší *věrohodnost*.

Následně z každé množiny navzájem překrývajících se kandidátů vybereme jen tu s maximální věrohodností.

Jazykový model

Nyní si popíšeme jazykový model od Phana a kol. [22], pomocí kterého budeme detekovat slova. Tento model nám definuje způsob jakým lze ohodnotit nějaké složení slova z podmnožiny detekovaných písmen. Na základě tohoto ohodnocení, následně lze vybrat to nejvhodnější podmnožinu písmen pro složení slov, speciálně v případě modelu od Phana a kol. lze mezi sebou porovnávat i ohodnocení složení různých slov pomocí různých podmnožin detekovaných písmen.

Mějme množinu kandidátu C, resp. komponent představujících písmeno v bitmapě. Potom konfigurace slova je poskládání slova w z kandidátů z C. Phana a spol. [22] na rozdíl od Wanga [26] považují za validní i konfigurace, které nepřídělí každému písmenu slova w kandidáta.

Definice 2. *Pro komponentu $c_i \in C$ a label $l_u \in L \cup \{\epsilon\}$ definujme* ***skóre*** *jako*

$$score(c_i, l_u) = \begin{cases} l_u \in L & P(l_u \mid c_i) \\ \epsilon & 1 - confidence(c_i) \end{cases}$$

Skóre nám vyjadřuje pravděpodobnost, toho že kandidát c_i je písmeno l_u. Všimněme si, že v případě že detekovaný kandidát je nějaký artefakt, potom je jeho *věrohodnost* poměrně nízká a tedy *skóre* pro prázdný label bude vysoké. Což zapříčinuje větší pravděpodobost, že kandidát bude považován za prázdný label.

Definice 3. *Nechť w je slovo délky k a $C = \{c_1, c_2 \dots c_n\}$ je množina všech kandidátů, potom definujme* ***zarovnání*** *pro w jako funkci:*

$$a_w : \{1 \dots n\} \to \{0 \dots k\}$$

Pro které, platí že $\forall i, j \in 1 \dots n : a_w(i) = a_w(j) \iff a_w(i) = 0 \vee i = j$.

Takto definovaná funkce a_w je formalizací kterékóliv konfigurace slova w pomocí kandidátů $\{c_1 \dots c_n\}$. V případě že v konfiguraci kandidát c_i představuje písmeno na j-té pozici, potom $a_w(i) = j$, jestliže kandidát c_i nemá přiřazený žadné písmeno, bude mu přiřazena pozice prazdného labelu a $a_w(i) = 0$.

Uveďme příklad. Mějme $C = \{c_1 \dots c_6\}$ a slovo ahoj a $a_{ahoj} = \{(1,0),(2,1),(3,2),(4,0),(5,4),(6,0)\}$. Tedy a_{ahoj} nám dává konfiguraci kodující konfiguraci, která c_1, c_4, c_6 berou prázdný label a c_2 přiřadí a, c_3 přiřadí h, c_5 přiřadí j, písmenu o není přidělen žádný kandidát.

Definice 4. *Nechť w(i) je funkce, která vrací charakter na i-té pozici slova w pro $i > 0$, pro $i = 0$ vrací prázdný label ϵ, potom zaveďme* ***skoré zarovnání*** *a_w jako*

$$AligmentScore(a_w) = \sum_{i=0}^{n} score(c_i, w(a_w(i))$$

Skóre zarovnání je ohodnocení libovolného zarovnání. Tato funkce nám vyjadřuje jak dobře zarovnání slova w, pasuje ke slovu w. Navíc si všimněme že navíc lze pomocí něj porovnávat i zarovnání libovolných slov.

Definice 5. *Nechť máme slovník W a množinu všech zarovnání A_w pro každé slovo $w \in W$ potom optimálně složitelné slovo w^* ze slovníku W, složitelné z C je:*

$$w^* = \arg\max_{w \in W} MaxWordScore(w)$$

kde MaxWordScore je

$$MaxWordScore(w) = \max_{a_w \in A_w} AligmentScore(a_w)$$

Nyní vidíme, že pro detekci slov, čili výběr optimálně složitelných, za pomocí jazykového modelu od Phana a kol. [22], je zapotřebí vypočítat *MaxWordScore*, což je skóre nejlepšího *zarovnání* slova $w \in W$ pomocí kandidátů z C. Posléze díky definici *skóre zarovnání* tyto zarovnání můžeme porovnávat a tedy vybrat slovo w ze slovníku W, které má nejlepší zarovnání. Nyní si uvedeme algoritmus, který pro dané slovo w najde *MaxWordScore* a příslušné zarovnání.

Algoritmus pro detekování slov ze slovníku

Nejdříve si popíšeme algoritmus pro detekci slov se slovníkem podle popsaného modelu a následně algoritmus pro nalezení konfigurace daného slova ze slovníku s největším skóre.

Algoritmus pro detekování slov je poměrně přímočařý, pro každé slovo najdeme jeho optimální konfiguraci, to jest ty které mají nejvyšší skóre a nepřiřadily písmenům slova žádného kandidáta, který by byl využit zarovnáním s vyšším

skóre.

Algoritmus 6: Detekovani slov na obrazku

```
1  foreach w in Dictionary do
2  |  maxAligmentScore ← maxAligmentScore(w);
3  |  wordConfigurations.add(score, w);
4  end
5  wordConfigurations.sortByScore();
6  foreach c in wordConfigurations do
7  |  configurationLetter ← reconstructLetters(c);
8  |  w ← c.word;
9  |  if letters are not used then
10 |  |  detectedWords.add(w);
11 |  |  mark letters as used;
12 |  end
13 end
```

Takže jak vidíme pro každé slovo ve slovníku najdeme optimální konfiguraci, následně tyto konfigurace setřídíme dle skóre zalodění. Následně prochazíme setříděný seznam od toho s největším skóre k tomu s tim nejmenším, pro každou konfiguraci se vždy kouknem jestli kandidáti jimž nebyl přiřazen prázdný label jestli již nebyl použit konfigurací s vyšším skóre. V případě že v konfiguraci takový kandidát neexistuje slovo a jeho konfiguraci prohlasíme jako detekované slovo a všechny kandidáty konfigurace s neprázdným labelem označíme jako použité. Nyní si uvedeme algoritmus, který pro dané slovo najde jeho optimální zarovnání.

Algoritmus pro nalezení optimalní konfigurace

Setřídění vstupu

Algoritmus od Trunga a spol. [22], předpokládá že na vstupu jsou písmena seřazená zleva doprava, pro detekci horizontálních slov, nebo seřazení ze shora dolů při detekování vertikálních slov.

Předpokládejme že vstup $C = \{c_1, c_2 \dots c_n\}$ je seřazený zleva doprava. Algoritmus počítá $MaxWordScore(w)$ od posledního písmene tj. od c_n. Postupně budeme vyplňovat tabulku F od zadních sloupců. F má velikost $n \times length(w)$ a v $F(c_i, p)$ je hodnota uložena hodnota maximálního skóre zarovnání a_w zužené jen na kandidáty $\{c_i, c_{i+1} \dots c_n\}$ navíc platí že kandidátu c_i je přiřazeno písmeno na p-té pozici slova w.

Pro zaplnění F využijeme dynamické programování a nakonec vybereme z F maximální skóre a zrekonstrujeme příslušnou konfiguraci.

Inicializace Hodnota $F(c_i, p)$ je inicializovana na skóre zarovnání, kde c_i je p-tý charakter slova w a $c_{i+1}, \dots c_n$ jsou nastaveny na prazdné labely, tj. nebyla jim přiřazena žádná pozice ve slově w.

Doplňovaní F Následně tabulku doplníme od spodních po horní řádky, V případě že se jedná o detekci horizontalních slov tabulku vyplňujeme od nejpravějších písmen až po ty nejlevější. Algoritmus staví konfigurace od nejpravějších písmen k těm nalevo.

Algoritmus 7: Inicializace F

```
for i = n downto 1 do
    for p = length(w) downto 1 do
        F(c_i, p) = score(c_i, w(p)) + Σ_{j=p+1}^{n} score(c_j, ϵ)
    end
end
```

Maximální konfiguraci, kde kandidát c_i je na p-té pozici a navíc se jedná o první přiřazeného kandidáta počítáme tak, že najdeme konfiguraci kde c_j je přiřazeno q-té pozici slova w, je první přiřazené písmeno a zároveň platí, že $i < j$ a $p < q$, v případě že taková to konfigurace neexistuje nebo není větší než původní inicializovaná hodnota $F(c_i, p$ tak konfiguraci neměníme. Protože jsme tabulku vyplňovaly od spodních řádků, tak máme vždy hodnoty $F(c_j, q)$ vypočítáné.

Algoritmus 8: Doplnění F pozpátku

```
l ← length(w);
for i = n downto 1 do
    for p = l downto 1 do
        maxSuccesorScore ←
          max_{j∈[i+1,n], q∈[p+1,l]} {F(c_j, q) + Σ_{k=i+1}^{j-1} score(c_k, ϵ)) + score(c_i, w(p))};
        if F(c_i, p) < maxSuccesorScore then
            F(c_i, p) ← maxSuccesorScore;
        end
    end
end
```

$F(c_i, p)$ je buď $F(c_j, q) + \sum_{k=i+1}^{j-1} score(c_k, \epsilon))$, kde suma $\sum_{k=i+1}^{j-1} score(c_k, \epsilon))$ vyjadřuje to že všechny $\{c_{i+1} \dots c_{j-1}\}$ berou prázdné labely nebo $F(c_i, p)$ zůstane nezměněné a tedy c_i je poslední písmeno kterému je přiřazen nějaký label pro optimální konfiguraci $F(c_i, p)$.

Výběr optimální konfigurace z F Následně musíme vybrat $MaxWordScore(w)$ z tabulky F jako:

$$MaxWordScore(w) = \max_{i,p} F(c_i, p) + \sum_{k=0}^{i-1} score(c_k, \epsilon)$$

Opět suma $\sum_{k=0}^{i-1} score(c_k, \epsilon)$ vyjadřuje že všechny kandidáti $c_1, \dots c_{i-1}$ berou prázdné labely.

Jak již bylo zmíněno pro detekci optimálních slov ze slovníku, stačí vypočítat $MaxWordScore(w)$ pro každé slovo ve slovníku a pak vybrat to s nějvětším skóre.

4.4.3 Navržené modifikace algoritmu

Modifikace jazykového modelu V uvedeném popisu jazykového modelu, se pro každé slovo vždy hledá jedna konfigurace, což ale znamená že model předpokládá že na vstupním snímku je každé slovo ze slovníku jenom jednom, tento

odhad je však nerealistický, proto v praxi navrhujeme nehledat pouze jednu konfiguraci pro dané slovo, ale vždy jednu až konstantu konfigurací.

Dále uvedený model neřeší situaci kdy máme dvě slova pro které mají identickou konfiguraci s největším skórem. Tato situace může nastat velice rychle, z nichž jedno je podslovem toho druhého. Představme si že máme vstupní fotografii a na ní je zobrazeno slovo is. Na této fotografii budeme detekovat slova za pomocí slovníku, který obsahuje slova is a miss. Pro obě dvě slova bude jako vhodná konfigurace nalezena konfiguraci s písmeny i, s. Jazykový model nepopisuje, které slovo vybrat. Řešení je poměrně triviální stačí vybrat konfiguraci, která se nejméně liší od slova ve slovníku.

Toho dosáhnem pomocí levenshteinovy editační vzdálenosti a to následujícím způsobem. Vytvoříme si slovo stvořené s *OCR* překladů jednotlivých kandidátů, tzn. že si vezmeme všechny konfiguraci přiřazené kandidáty a z nich stvoříme slovo pomocí písmen kandidátu, která jim byla určena v *OCR* fázi. Následně vypočítáme levehnsteinovu editační vzdálenost mezi slovem ve slovníku a stvořeným slovem z konfigurace. Potom při setřízení konfigurací (řádka 5 algoritmus pro detekování slov 6), budeme třídit primárně dle skóre konfigurace a sekundárně dle editační vzdálenosti od slova ve slovníku.

Optimalizace pomocí Trie Další velkou nevýhodou uvedéného algoritmu pro výběr optimálního slova ze slovníku je výpočetní náročnost. Protože pro výběr optimálního slova ze slovníku musíme pokaždé najít nejoptimálnější konfiguraci zvlášt.

Všimněme si, že pro slova sdílící postfix jako slova svátek a svátek má tabulka vyplňovaná tabuka F 4 spodní řádky stejné. Tohoto pozorování lze využít pro detekci tak abychom si ušetřili práci a stejné řádky od konce počítali vždy jen jednou. Nejdříve potřebujeme shlukovat slova sdílející stejný postfix. Toto se dá velice dobře vyřešit slovníkovou trií, která navíc snižuje paměťové nároky. Při použití trie si potom stačí naalovat tabulku F na rozměry počet písmen $\times$ nejdelšího slova ve slovníku a počet kandidátů. Následně začneme procházet slovníkovou trii do hloubky a přítom vyplňovat tabulku od konce.

Při traversování trie si pamatujeme, hloubku v které jsme a podle ní vyplňujeme tabulku F, vždy se vyplňujeme sloupec odpovídající hloubce průchodu.

Pro úplnost si ještě uveďme pseudokód procedury, která se rekurzivně volá na

uzlech stromů.

Algoritmus 9: Doplnění F pomocí trie

```
1  Function traverseTreeNode(node, currentDepth)
   for i = n downto 1 do
2  |  F(c_i, p) = score(c_i, w(p)) + Σ_{j=currentDepth+1}^{n} score(c_j, ε)
3  end
4  for i = n downto 1 do
5  |  maxSuccesorScore ←
   |        max_{j∈[i+1,n]}  {F(c_j, q) + Σ_{k=i+1}^{j-1} score(c_k, ε)) +
   |        q∈[currentDepth+1,l]
   |        score(c_i, w(currentDepth)};
6  |  if F(c_i, currentDepth) < maxSuccesorScore then
7  |  |  F(c_i, currentDepth) ← maxSuccesorScore;
8  |  end
9  end
10 if Node is WordStart then
11 |  wordConfigurations.add(word, score);
12 end
13 foreach child in node.children do
14 |  traverseTreeNode(child, currentDepth- 1);
15 end
```

Jak vidíme procedura má jako parametry uzel trie a proměnnou, která nám říká v jaké hloubce stromu trie se právě nacházíme, tato proměnná ale není nastavena na vzdálenost od kořene trie ale od jejího nejspodnějšího patra. V pseudokódu vidíme, že algoritmus vždy počítá sloupec tabulky F. Který sloupec vyplňujeme nám právě určuje proměnná *currentDepth*. Důležité je, že pokaždé kdy na řádce 14 voláme rekurzivně proceduru $TraverseNode$ a v tomto rekurzivním voláním hledáme proměnnou $maxSuccesorScore$ v identické části tabulky F, a tedy pro každý postfix jsme tabulku F vyplňovali pouze jednou.

Tento způsob optimalizace byl navržen v práci od Wanga [26].

Výběr užitečných konfigurací V algoritmu prod detekování slov, pro každé slovo ve slovníku nalezneme jeho optimální konfiguraci tu následně uložíme do seznamu detekovaných slov a pokračujeme dalším slovem ve slovníku(řádka 11 u verze s trií 9 a řádka 10 u verze bez trie 6). Jestliže ale budeme ukládat pro každé slovo ukládat jeho konfiguraci dosáhneme poměrně velkého plýtvání pamětí i času. Proto navrhujeme ukládat jen konfigurace slov ve slovníku, které považujeme za užitečné.

Konfiguraci slova ve slovníku budeme pokládat za užitečnou, právě tehdy když bude levenshteinova editační vzdálenost od slova ve slovníku menší než nějaká konstanta γ závislá na délce slova ve slovníku.

Restrikce výběru následníku Phan a spol. [22] dále při výběru maximalního následníka při doplňování tabulky, omezují výběr následníka pro kandidáta c_i, kde následník je od c_{i+1} až do posledního kandidáta, jenom na několik nejbližších následujích. My navrhujeme další restrikci, při které budeme hledat následníka jenom mezi těmi, s kterými c_i může být ve slově. Zadefinujme si relaci $\sim_w$ být

spolu ve slově. Potom pro kandidáta c_i zužíme výběr následníku z $\{c_{i+1} \dots c_n\}$, kde n je počet všech kandidátů na množinu

$$C_R^i = \{c \mid dist(c, c_i) < \gamma \wedge 0.3 \leq height(c_i)/height(c) \leq 3.5\}$$

kde první podmínka nám říká že od sebe nemohou být vzdáleněji než γ, to bylo experimentálně vyčísleno na na minimum z trojnásobku velikost digonály c_i a čtvrtiny šířky vstupního obrázku, druhá podmínka nám říká že písmen poměr výšek písmen nesmí být menší než 0.3 a větší než 3.5.

Deformační cena Naším dalším vylepšením je takzvaná deformační cena. Zavedení deformační ceny jsme se inspirovali v práci od Wanga a kol. [27], Wang a kol. tuto deformační cenu také používají v jejich jazykovém modelu [26]. Tato změna dále specifikuje jakou konfiguraci vybrat pro výpočet $F(c_i, p)$ když hledáme v pseudokodu algoritmu 8 $F(c_j, q)$, kde $i < j$ a $p < q$ pro výpočet $F(c_i, p)$.

Zavedení deformační ceny nám pomůže vyřešit problématickou situaci na obrázku 4.5. Představme si, že detekujeme slovo *NATURAL*, pro jednoduchost předpokládejme, že se nám podařilo vydetekovat všechna písmena a navíc jsme ve fázi kdy pro písmeno A(ve světle modrém rámečku) hledáme následníka.

Obrázek 4.5: Nejasné zvolení nástupce

Jak vidíme z obrázku 4.5 jako následníci se jeví potencionálně 2 konfigurace(v červeném a zeleném rámci). Správný následník je samozřejmě zelená konfigurace. Ale zkoumejme jak na takovou to situaci bude reagovat algoritmus od Phana a kol [22]. V jeho původním návrh se řeší pouze 8 scóre na základě skóre konfigurací, tedy jenom na pravděpodobnostních výstupů z *OCR*. V takovéto situaci ale může původní algoritmus selhat a místo zelené konfigurace vybrat červenou konfigurace. Jediné co stačí je, aby pro kandidát t_{red}, písmeno T v červeném rámci, $P(t_{red} \mid T)$ bylo vyšší než $P(t_{green} \mid T)$, kde t_{green} je kandidát písmena T v zeleném rámci. V tu chvíli bude mít dle definice zelená konfigurace nižší skóre než červená.

Zavedení deformační ceny tuto situaci vyřeší.

Definice 6. *Definujme hraničním obdelník b jako nejmenší obdelník, který lze opsat komponentu daného kandidáta c, Tento obdelník je daný čtveřící* (x, y, w, h),

kde x je x-ová souřadníce horního rohu b, y je jeho y-nová souřadnice, w je šířka obdelníku a h je jeho výška. Nechť $b_i = (x_i, y_i, w_i, h_i)$ je hraniční obdelník kandidáta c_i a $b_j = (x_j, y_j w_j, h_j)$ je hraniční obdelník kandidáta c_j. Dále definujme $x^* = x + w$ a $y^* = y + h$ a $\Sigma = \begin{pmatrix} w_i & 0 \\ 0 & h_i \end{pmatrix}$ Potom deformationCost je daná dvěma vztahy:

1. *Při detekci horizontálních slov*

$$deformationCost(c_i, c_j) = \sqrt{(x_i^* - x_j)\Sigma^{-1}(x_i^* - x_j)}$$

2. *Při detekci vertikálních slov*

$$deformationCost(c_i, c_j) = \sqrt{(y_i^* - y_j)\Sigma^{-1}(y_i^* - y_j)}$$

Při horizontální detekci slov si všimněne že čím blíže jsou k sobě pravý horní roh kandidáta c_i a levý horní roh c_j tím menší je deformační cena. To samé platí při vertikální detekci pro dolní levý roh c_i a horní levý roh c_j.

Uveďme dále změněný pseudokód algoritmu pro doplnění F 8, který zároveň již počítá s restrikcí následníků.

Algoritmus 10: Doplnění F pozpátku s deformační cenou

```
1 for i = n downto 1 do
2     for p = length(w) downto 1 do
3         (j*, q*) ←
          arg max_{j∈C_R^i, q∈[p+1,l]} {F(c_j, q) + Σ_{k=i+1}^{j-1} score(c_k, ε)) + score(c_i, w(p)) −
          deformationCost(i, j)}
4         maxSuccesorScore ←
          {F(c_{j*}, q*) + Σ_{k=i+1}^{j*-1} score(c_k, ε)) + score(c_i, w(p))}
5         if F(c_i, p) < maxSuccesorScore then
6             F(c_i, p) ← maxSuccesorScore
7         end
8     end
9 end
```

Oproti původní verzi 8 se ve vylepšené verzi snažíme vybrat nejen písmeno, které splňuje lexikografickou podmínku ale i písmeno které bude nejlépe vytvářet slovo po grafické podobě s kandidátem c_i. Tototo následníka vybereme tak, že při jeho hledání ještě odečteme deformační cenu c_i a c_j. Jak víme z definice deformační cena dosahuje nejnižších hodnot, právě tehdy když jsou písmena blízko sebe. Analogickým způsobem budeme modifikovat algoritmus využívající slovníkovou trii 9.

Ukažme si, jak zavedení deformační ceny vyřeší situaci na obr. 4.5. Je jasně vidět, že deformační cena mezi kandidátem a_{blue} ve světle modrém rámci A a kandidátem t_{green} je menší deformační cena než mezi a_{blue} a t_{red}. Jelikož se dá předpokládat, že skóre červené i zelené konfigurace budou poměrně podobná, tak po odečtením deformační ceny bude skóre červené konfigurace menší než zelené.

Dodejme, že v praxi je potřeba vyvážit deformační cenou nějakou konstantou θ, tak aby neměla větší význam než informace z jazykového modelu. Experimentálně jsme hodnotu θ vyčíslili na 0.25.

Výhody oproti ostatním algoritmům

Jak je v článku uvedeno [22], největší výhododou tohoto algoritmu je umožňení rozeznávání slov na obrázku, pro nějž jsme nedetekovali všechna písmena, ale jenom většinu. Většina ostatních jazykových modelů toto ale nedovoluje. Detekovaní je umožněno díky uznání konfigurací, které nepřidělují všem písmenům slova nějaké kandidáty, tedy existenci prázdného labelu ϵ.

Ve zbývajících částech této kapitole si popíšeme ještě *Stroke Width transformaci* [5] a *SVM* s *fast intersectio* jádrem [13]. Nejedná se o důležité součásti naší práci, ale přesto pokládame za vhodné se o nich zmínit, jelikož jsme obě metody využili a sami implementovali v softwarovém prototypu.

Implementace Seznámení s třidou implementující uvedený algoritmus najdeme v 9.5.

5. Experimentální část

5.1 Testovací data

Všechny experimenty budeme provádět na testovacích sadě *ICDAR 2013*. V následující kapitole bude provedeny 2 experimenty. Z nichž jeden se zaměří na porovnání metod pro detekci a segmentaci písmen. V druhém experimentu budemem experimentálně ověřovat vliv námi přidanými zlepšením při detekci slov pro algoritmus od Phana [22].

Nejprve si ale představíme základní 2 metriky při testování detekce objektů v oblasti počítačového vidění.

5.2 Metrika testování

Při experimentech v detekci objektů v počítačovém vidění měříme obvykle dvě metriky, tj. recall a precision, definované následujícími vztahy.

$$recall = \frac{|true\ positive|}{|ground\ truth|}$$

$$precision = \frac{|true\ positive|}{|detected\ result|}$$

Vysvětleme si nejdříve co znamenají jednotlivé pojmy:

Ground truth
: Data s informacemi o detekovaných objektech, které budeme hledat v testovací sadě. Jsou dodávány od autorů testovacích sad.

True positive
: Všechny námi nalezené objekty, které jsou zároveň v ground truth. Tedy jedná se o všechny úspěšně detekce.

Detected results
: Všechny výsledky naší detekce na testovacích sadě.

Definujme $|true\ positive|$ jako velikost množiny všech true positive výsledků. Analogicky dodefinujme $|ground\ truth|$ a $|detected\ result|$.

Z uvedených vzorců výplývá, že veličina *recall* vyjadřuje jaký zlomek *ground truth* jsme schopni vydetekovat, a hodnota *precision* udává počet relevatních správně vydetekovaných objektů vůči počtu všech detekovaných objektů.

5.3 Detekce písmen

V prvním experimentu budeme porovnat úspěšnost v detekci písmen pomocí *component based* přístupů využívajících algoritmů *SWT* [5], *MSER* [14] a *ER* [17] v kombinaci s nonmax suppresí pomocí OCR.

5.3.1 Popis testovaných metod

Přístup *ER* Tento postup pro detekci písmen využívá metodu *ER*, která byla detailně rozebrána v 4.3.2.

Přístup *MSER* Druhý přístup kombinuje popsaný algoritmus *MSER*, více viz 4.3.1 s filtrací, která je identická s klasifikací v 2 fázi algoritmu *ER*, která je popsana v 4.3.2.

Přístup *SWT* Tato metoda spočívá v kombinaci lokální otsu binarizace [21] a *SWT* od Epshtaina a kol. [5], použita ale nebyla verze od Epshteina, ale verze od Chena [2], více viz 7.1. V algoritmu se nejprve provede lokální otsu binarizace, tzn. že vstupní obraz skenujeme pod čtvercovým oknem. Část pod čtvercovým skenerem binarizujeme podle otsuva algoritmu. Výslednou binarizovanou bitmapu předáme na vstup algoritmu *SWT*, který tuto bitmapu transformuje do její *SWT* podoby. V *SWT* bitmapě hledáme komponenty souvislosti, kde 2 pixely považujeme za sousední, jestliže spolu sousedí v bitmapě a poměr *SWT* hodnot je menší než 2. Nalezené komponenty následně klasifikujeme na písmenkové a nepísmenkové komponenty, pomocí filtru ve druhé fázi *ER* algoritmu, ke kterému přidáme ještě variační koeficient všech hodnot komponentových pixelů v *SWT* transformaci.

5.3.2 Non max suprese s *OCR*

Nad výstupem všech tří metod je provedena nonmax supresse pomocí *OCR*. Výstup nonmax suprese je následně evaluována vyhodnocovacím protokolem, jenž hned představíme.

5.3.3 Trénovací data

Pro veškeré trénovaní klasifikátorů byly použity trénovací vzorky z trénovací sady *ICDAR 2013*. Pro trénování obou fází ER klasifikace jsme použili zhruba 900 ručně extrahovaných pozitivních vzorků a 1400 negativních pozitivních vzorků. Pro vytrénování *OCR* jsme vygenerovali syntetické vzorky za pomocí klasických windows fontů.

5.3.4 Vyhodnocovací protokol

Úkolem vyhodnovacího protokolu je rozhodnout o jednom detekovaném písmenu jestli je obsaženo v *ground truth*. Nejprve ale popišme podobu *ground truth* pro tento experiment.

Podoba *Ground truth* Nejprve definujeme obdelník jako uspořadanou čtveřici (x, y, w, h), kde x, y jsou souřadnice levého horního rohu a w je šířka obdelníku a h je výška obdelníku. Dále definujme ohraničující obdelník písmena jako nejmenší čtverec (x, y, w, h), kterým lze ohraničit písmeno na bitmapě. Písmeno lze ohraničit obdelníkem jestliže pro každý pixel (a, b) písmene, platí že $x \leq a \leq x + w$ a $y \leq b \leq y + h$.

Potom *ground truth* data pro jeden testovací snímek v testovací sadě tvoří seznam ohraničujících obdelníku všech písmen na snímku.

True positive Písmeno l pokládáme za *true positive*, právě tehdy když existuje alespoň jeden obdelník gt v *ground truth* testovacího snímku, tak že průnik ohraničujícího obdelníku písmena l a gt je neprázdný a dosahuje plochy alespoň 0.7 plochy obdelníku gt a 0.5 plochy obdelníku písmene l. Jestliže žádný takový obdelník v *ground truth* testovacího snímku neexistuje, je písmeno l považováno za falešnou detekci.

Vyhodnocování pro celou testovací sadu probíha ve 2 fázích. Nejprve jsou detekovány písmena pomocí příslušné metody, následně jsou validovány pomocí uvedeného protokolu. Při vyhodnocování si pamatujume relevantní údaje pro výpočet *recall* a *precision*.

5.3.5 Předpoklad

Dle naší analýzi předpokládáme, že algoritmus *ER* dosáhne nejvyšší hodnoty atributu *recall*, algoritmus *MSER* skončí jako druhý a *SWT* přístup skončí jako poslední. Co se týče hodnoty *precision* zde očekáváme, že metoda *MSER* bude mít mírně navrh, jelikož *ER* i *SWT* jsou méně robustnější vůči drobným artefaktům.

5.3.6 Výsledky

Nyní uveďme výsledky našeho experimentu.

Testovací sada ***ICDAR 2013***		
metoda	*recall*	*precision*
ER	0.72	0.074
MSER	0.65	0.35
SWT	0.58	0.089

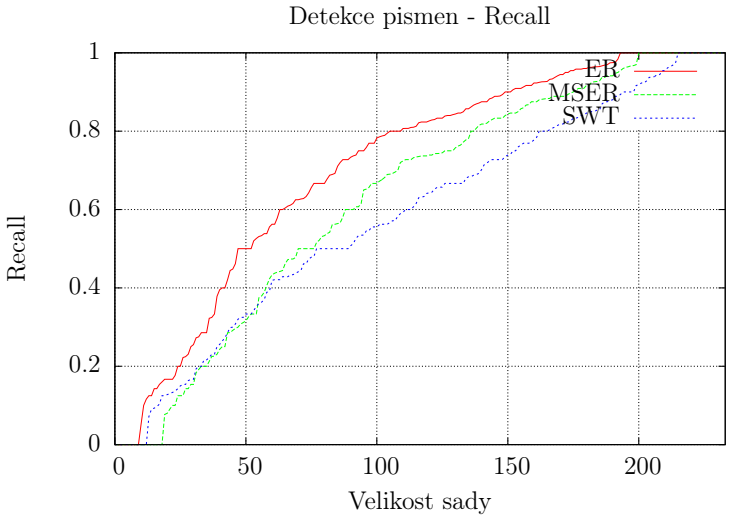
Tabulka 5.1: Výsledky - detekce písmen

Jak vidíme v tabulce výsledků experimentu 5.1, splnil se náš předpoklad že algoritmus *ER* dosáhne nejvýšší hodnoty *recall* a že algoritmus *MSER* bude mít nejlepši *precision*.

Naopak se nepotvrdilo, že algoritmus *MSER* bude mít jenom mírně navrh co se týče atributu *precision* před zbylými dvěmi metodami. Naopak rozdíl je vůči zbylým přístupům je poměrně markantní. Tento rozdíl si vysvětlujeme již zmíněnou menší robustností vůči drobným artefaktům. U algoritmu *ER* se domníváme, že spousta drobných artefaktů je za využití navržených deskriptrorů při 2 filtračním procesu je lehce zaměnitelná s písmeny, tudíž jsou detekovány jako falešné detekce. U algoritmu *SWT* se domníváme že nízký *precision* je způsoben ze stejných důvodů. Malé artefakty mohou mít poměrně stabilní stroke a navíc ve filtraci používáme stejné deskriptory jako ve dvou zbylých přístupech. Naproti tomu algoritmus *MSER*, k filtracím přidává požadavek stability viz 4.3.1 . Tento požadavek výrazně zvyšuje *precision* ale na druhou stranu vidíme, že také snižuje *recall*.

Nyní si uveďme graf setříděné posloupnosti hodnot *recall* vůči velikosti sady a jeho interpretaci.

Interpretace grafu Nechť $r = 0.4$. Potom pro r na ose *recall* jsme schopni vyčíst z grafu 5.1, že metoda *ER* není schopna dosáhnout recallu r zhruba 40 snímcích, metoda *MSER* na zhruba 60 snímcích a metoda *SWT* tohoto recallu r není také schopna dosáhnout na zhruba 60 snímcích.



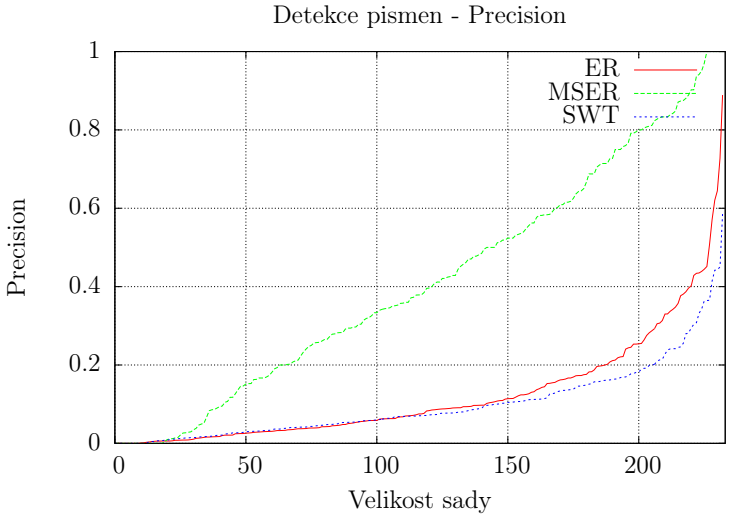
Obrázek 5.1: Růst recall vuči velikosti sady

Z grafu vidíme, že tvar všec tří křivek se blíží logaritmickému, z nichž největší similarity dosahuje křivka ER. Logaritmický tvar je způsoben větším množstvím snímků, kde byla dosažen slušný *recall*(například větší než 0.6), vůči počtu snímků kde byl dosažen menší *recall*.

Dále vidíme, že metoda *ER* jakkýkoliv hodnoty *recall* od 0 do 1 nejdříve, proto jak již bylo řečeno nejvýrazněji připomína logaritmickou křivku a například v počtu snímků kde je $recall \geq 0.8$ jasně předčil oba zbývající přístupy.

Při porovnání křivek *MSER* a *SWT*, lze vidět že do počet snímků, kde byl dosažen $recall \leq 0.4$ je podobný, ale poté *recall MSERU* roste daleko rychleji než *recall SWT* a tedy platí, že pro $\forall r \leq 0.4$ je počet snímků kde byl dosažen $recall = r$ algoritmem *MSER* je větší než za pomocí *SWT*.

Nyní uveďme podobný graf pro *precision*. Interpretace je analogická jako v případě předchozího grafu 5.3.6 akorát místo *recall* řešíme *precision*.



Obrázek 5.2: Růst precision vuči velikosti sady

V tomto grafu lze pozorovat jasnou dominanci algoritmu *MSER*, jak jsme již zmínili zapřičiněnou požadavkem stability. Dodejme, že křivka *MSERU* má lineární růst, kdežto růst zbylých křivek je exponenciální.

Vidíme také, že algoritmus *MSER* dosáhne $precision = 1$ narozdíl od dvou zbývajících přístupů. Algoritmus *ER* je schopen zhruba dosáhnou maximálního $precision = 0.9$ a algoritmus *SWT* je dosáhne přibližně maximální hodnoty 0.6. Poznamenejme ještě, že algoritmus křivky *ER* a *SWT* mají velice podobný tvar do hodnoty 0.1 a od této hodnoty začne algorimus *ER* rychleji růst.

5.3.7 Zajímavé výsledky

Nejdříve si ukažme také příklady vstupu kde selháva algoritmus *ER* a kde naopak funguje velice dobře.

Problematické vstupy

V 5.3 vidíme ukázky, které jsou problematické pro všechny 3 metody.

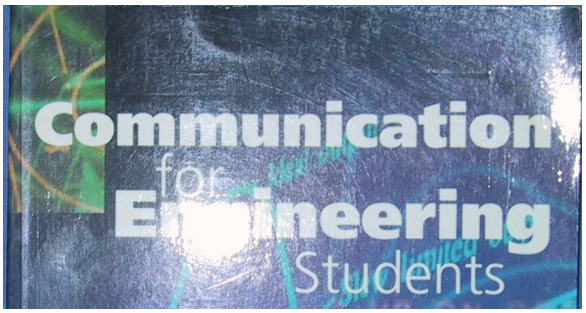
Na vstupu 5.3a vidíme písmena, která jsou velice špatně separovatelná od pozadí. Na tomto vstupu algoritmus selhává, protože neexistuje práh θ, pro který je komponenta tvaru písmene, uzlem v komponentovém stromě, díky tomu že lze velice špatně rozlišit písmeno od pozadí. V druhé ukázce 5.3b se díky nasvícení deformuje tvar komponenty písmene, a proto ho nenalezneme v komponentové stromě. U 5.3c je problém s písmenama které jsou slité do jedné komponenty, jako *ST* na ukázce.

Ukázky segmentace

Uveďme také příklad ze sady *ICDAR 2013* a porovnejme výsledky segmentace písmen pomocí našich tří navrhovaných metod.

V 5.4 je uveden testovací vzorek z sady *ICDAR 2013* a tři různé výsledky na něm provedené segmentace písmen. Vidíme, že segmentace *ER* 5.4b je schopna

(a) špatně seperovatelná písmena



(b) nasvícená písmena

(c) slitá písmena

Obrázek 5.3: Selhání ER

vydetekovat největší množství písmen, ale zároveň také detekuje největší množství falešných detekcí. Oproti tomu segmentace *MSER* je poměrně robustní vůči falešným detekcím, ale nedosahuje schopnosti detekce *ER*, viz 5.4c. Segmentace *SWT* dle ukázky 5.4d je schopna detekovat nejmenší počet písmen a zároveň detekuje poměrně velké množství falešných detekcí.

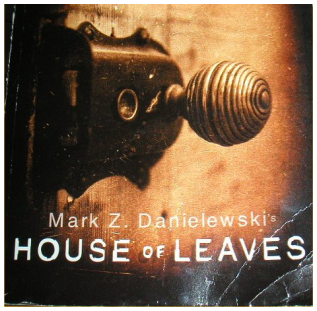
5.3.8 Závěr

V experimentu jsme zkoumali vlastnosti algoritmů *ER*, *MSER* a *MSER* v detekci písmen. Nejdříve jsme si představili protokol vyhodnocení a potom jsme analyzovali výsledky jednotlivých metod.

Z analýzy jasně vyplývá, že algoritmus *ER* dosahuje nejlepšího výsledku co se týče *recall* ale také nejhoršího v *precision*. Naopak algoritmus díky požadovku stability dosahuje výrazně vyššího *precision*, ale také má o 0.1 nižší *recall*. Algoritmus *SWT* naopak dosahuje nejhoršího výsledku co se týče *recall* ale jeho výsledek v *precision* je jen o něco málo lepší než *precision* algoritmu *ER*, z tohoto vyvozujeme že přístup *SWT* je pro detekci nejméně vhodný, ze všech tří testovaných alternativ.

Dle experimentů dále považujeme za nejvíce vhodný algoritmus *ER*, protože dosahuje největší hodnoty *recall*, pravda je že sice také v experimentech dosáhl nejnižší *precision*, tedy je nejmíň robustní vůči artefaktům, ale nižší *precision* nehraje takovou rolu při správně zvoleném algoritmu při detekci slov. Naproti tomu algoritmus *MSER* má sice vyšší *precision*, ale má nižší recall což bude zapříčíní horší detekci slov, protože bez detekovaných písmen lze těžko detekovat slova.

V následujícím experimentu se zaměříme na algoritmus detekce slov a jeho



(a) Vstup

(b) ER segmentace

(c) MSER segmentace

(d) SWT segmentace

Obrázek 5.4: Ukázky segmentace

kompatibilitu s algoritmem *ER* a *MSER*.

5.3.9 Znovu spustitelnost experimentu

Pro spuštění experitmentu používáme program *letter-segmentation*, který má povinné 2 argumenty, ten první je xml soubor s udáji k testovací sadě icdar 2013 (*letter-segmentation-icdar2013.xml*) a ten druhý je cesta ke složka kam se uloží xml záznamy o segmentaci. Vstupy pro segmentaci čte program ze standartního vstupu, stačí do něj přesměrovat vstupní soubor *letter-segment-testset*.

5.4 Detekce slov

V kapitole o námi navrhované metodě, jsme modifikovali algoritmus od Phana a co. [22]. V první části experimentu se budeme zabývat kombinací algoritmu *MSER* a *ER* s algoritmem pro detekování slov popsaným v předchozí kapitole.

Ve druhé části budeme zkoumat vliv naších modifikací algoritmu od [22]. Nejdříve se zaměříme na změnu výkonnostní rozdíl algoritmu bez a s optimalizací pomocí slovníkové trie. Dále se zaměříme na dopad vůči *precision* a *recall* zapříčeněné navrženým zlepšením výběru následníka.

Nejdříve si ale opět představíme podobu ground truth a vyhodnocovací protokol.

5.4.1 Ground truth

Opět definujme ohraničujicí obdelník pro slovo w jako nejmenší ohraničující obdelník, do kterého se lze vtěstnat všechny ohraničující obdelníky písmena slova w.

Následně pro jeden snímek z testovací sady považujeme za *ground truth* seznam dvojic slovo a ohraničující obdelník slova.

5.4.2 True positive

Detekované slovo považujem za *true positive*, právě tehdy když existuje slovo v ground truth, které má stejné písmenové labely v našem *ocr* systemu jako detekované slovo. Dále musí platit že průnik ohraničujícího obdelníku *gt* slova v *ground truth* s ohraničujícím obdelníkem w_{rect} detekovaného slova, má plochu alespoň 0.6 plochy *gt* a 0.4 plochy w_{rect}.

5.4.3 Kompatibilita *ER* a *MSER*

V tomto experimentu budeme zkoumat rozdílnost detekci slov, v případě že byly slova detekovány za pomocí metody *ER* nebo *MSER*. Experiment bude proveden opět na testovací sadě *ICDAR 2013*.

Použitý slovník pro detekování V tomto experimentu jsme jako slovník pro rozpoznávání slov použili seznam všech slov ze všech testovacích vzorků, který je dodaný k testovací sadě.

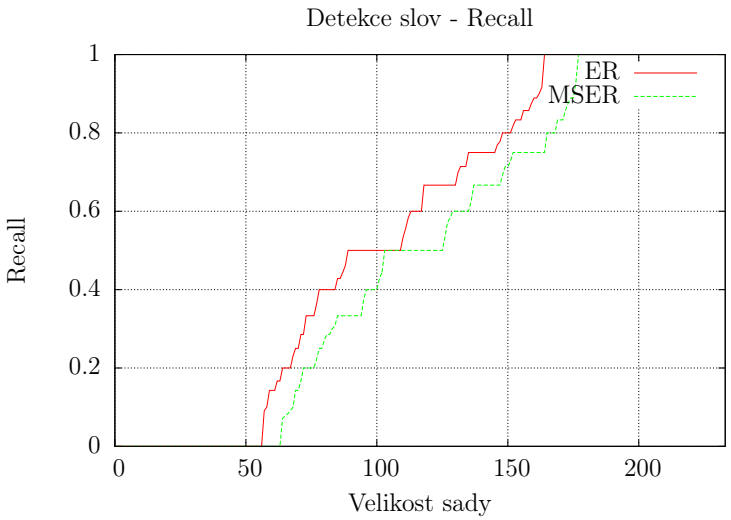
Předpoklad Předpokládáme, že dosáhne algoritmus ER dosáhne vyššího hodnoty *recall*. Také očekáváme zmírnění rozdílu v *precision* obou algoritmů oproti prvnímu experimentu, protože námi zvolený algoritmus pro detekování slov by měl umět eliminovat artefakty díky non max suppresi popsané v 4.4.2.

Výsledky Uveďme výsledky obou metod. Testování opět proběhlo na testovací sadě *ICDAR 2013*.

Testovací sada ***ICDAR 2013***		
metoda	*recall*	*precision*
ER	0.59	0.73
MSER	0.51	0.85

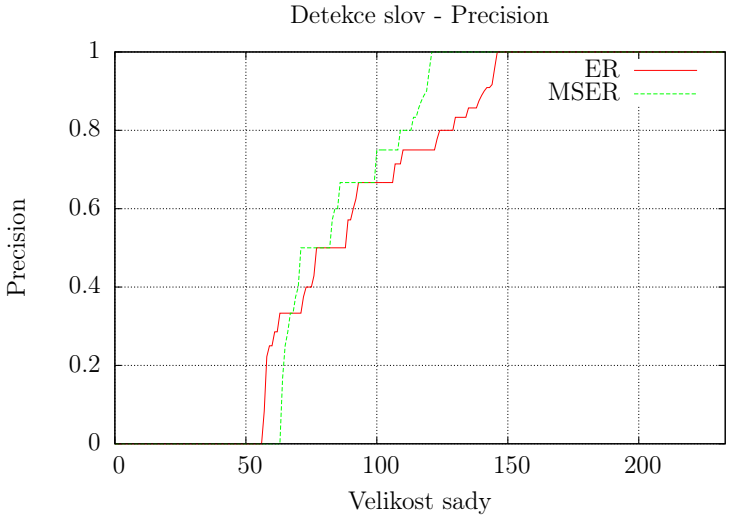
Tabulka 5.2: Výsledky - detekce slov

Z výsledků experimentu vyvozujeme, že předpoklad v naší analýze se splnil. Tedy algoritmus *ER* dosáhl oproti algoritmu *MSER* lepšího výsledku, a zároveň se snížil rozdíl v *precision* o půlku. Ukažme opět grafy růstu *precision* a *recall* vůči velikosti sady. Interpretace je opět analogická v obou grafech jako pro graf v předchozím experimentu 5.3.6.



Obrázek 5.5: Růst recall vuči velikosti sady

Z grafu 5.5 vidíme, že kompatibilita algoritmus od Phana a kol. [22] je lepší s algoritmem *ER* než s algoritmem *MSER*. Platí, že $\forall r \in [0, 1]$ počet snímků na nichž bylo dosaženo hodnoty recallu aspoň r je za pomoci *ER* je větší než *MSERU*. Z grafů lze také vyčíst, že nejsme schopni vydetekovat žádné slovo pomocí *ER* na přibližně 60 snímcích a u *MSERU* se jedná o větší počet.



Obrázek 5.6: Růst precision vuči velikosti sady

V následujícím grafu vidíme, že obě křivky mají poměrně podobný tvar. Tato podobnost experimentálně potvrzuje náš předpoklad schopnosti algoritmu navrženého Phanem a kol. [22] eliminovat falešné detekce, tj. různé drobné artefakty, které byly algoritmem *ER* detekovány oproti algoritmu *MSER*.

Také lze vypozorovat, že každé hodnoty *precision* menší, než zhruba 0.35, dosáhne algoritmus *ER* rychleji, potom ale každou hodnotu *precision* větší než 0.35, dosáhne algoritmus *MSER* rychleji (v menším počtu vzorků). Tento jev je způsoben větším počtem testovacích vzorků, kde díky algoritmu *ER* jsme schopni detekovat alespoň jedno slovo na vstupu oproti algorimu *MSER*, tedy algoritmus

ER dosáhne hodnoty menší než 0.35 na méně vstupech díky lepšímu *recallu* při detekci písmen. Od hodnoty *precision* 0.35 roste ale algoritmus *MSER* rychleji, protože má větší *precision* v detekci písmen než algoritus *ER* a tedy je menší šance na detekování falešně detekovaných slov.

Ukázky jednotlivých příkladu z testovací sady Prohlédněme si ukázky dvou výsledků s testovací sady. Poznamenejme, že všechny true positive detekce jsou ohraničené modře a všechny falešné detekce jsou ohraničeny červeně.

V první ukázce 5.7 je uvedena ukázka vstupu, kde je *ER* varianta sice schopna detekovat jedno slovo narozdíl od *MSER* varianty, ale také varianta *ER* byla schopna detekovat mnohem více falešných detekcí. Tedy na výsledku detekce slov tohoto vstupu se odraží jak větší *recall* algoritmu *ER* tak i jeho nižší *precision*.

(a) ER varianta (b) MSER varianta

Obrázek 5.7: Detekovaná slova pomocí Phana a kol [22] a kompatibilita ER a MSER

V další ukázce 5.8 vidíme jaký vliv má větší *recall* algoritmus *ER* vůči algoritmu *MSER* v detekci písmen.

(a) ER varianta (b) MSER varianta

Obrázek 5.8: Detekovaná slova pomocí Phana a kol [22] a kompatibilita ER a MSER

Znovu spustitelnost experimentu

Pro spuštění experitmentu používáme program *icdar2013-evalution*, pro seznámení s parametry příkazové řádky použijte přepínač "-h". Vstup k tomuto programu je soubor s cestami k jednotlivým vstupům k testovací sadě. Kde na každém řádku je právě jedna ceska k testovacímu vzorku. Testovací sada se programu zadá pomocí přepínače "-t".

5.4.4 Optimalizace pomoci slovnikové trie

V tomto experimentu budeme zkoumat změnu doby výpočtu(detekování slov pomocí slovníku) dle algoritmu od Phana a kol. [22] *ms* za použití optimalizace pomocí slovníkové trie. Čas bude měřený dle wallclock procesoru.

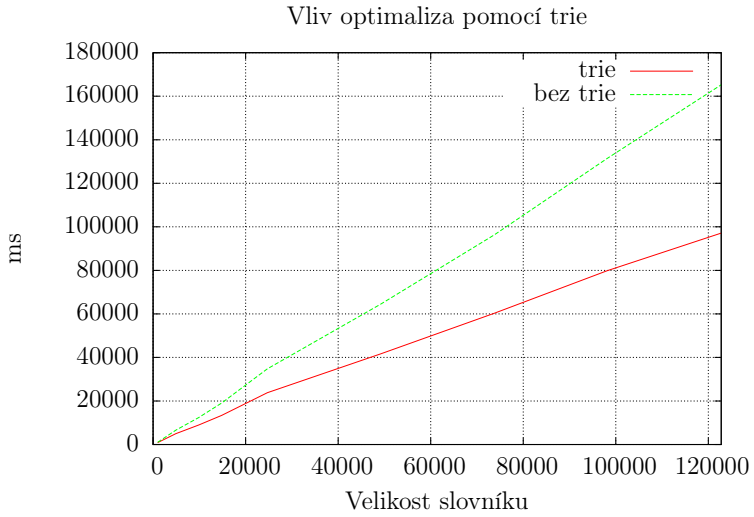
Předpokládáme zkrácení doby výpočtu díky optimalizaci pomocí slovníkové trie.

Podmínky výpočtu a měření délky výpočtu Experiment bude proveden na sestavě s procesorem *Intel Core i7 920 (4x 2.66 GHz + HyperThreading)*. Doba výpočtu budeme měřit pomocí wall clock.

Budeme postupně čas detekování slov na slovnících o velikosti 1000, 5000, 10000, 15000, 25000, 50000, až 125000.

Experiment budeme spouštět na vstupní obrázku *img_115.jpg* s testovací sady *ICDAR 2013*. Nejdříve jsou z tohoto vstupu detekována písmena algoritmem ER a poté je provedena non max suprese. Potom pro každou velikost slovníku jsme detekování slov měření pouštěli u obou variant 4 krát a výsledný čas je aritmetický průměr ze všech měření.

Nyní uveďme graf udáje z měření.



Obrázek 5.9: Doba rozponávání slov vůči různým velikostem slovníku

Výsledky Z výsledku měření vidíme, že obě křivky mají lineární tvar. Z grafu vidíme, že křivka zobrazující čas rozpoznávání slov bez trie roste daleko rychleji

než křivka času dosaženého pomocí optimaliza trie, což experimentálně potvrzuje naš předpoklad o zrychlení za dosaženého za pomocí slovníkové trie.

Dodejme že, při zrychlení dosažené pomocí slovníkové trie by mohlo být větší za využití memory poolu, tak aby se zlepšila cache lokalita jednotlivých uzlů stromu, zabránilo se zbytečné fragmentaci paměti dále by bylo potřeba zvolit vhodnější strukturu stromu v implementaci. Na provedení těchto optimalizaci nám nezbyl čas, i tak jak vidíme z grafu 5.9 se nám podařilo dosáhnou poměrně netriviálního zrychlení.

Znovuspustitelnost experimentu Pro spuštění algoritmus stačí spustit shell script $word - generatin.sh$ s jednim parametrem představující soubor kam se bude zaznamenávat výsledky měření. Ten bude mít formát v podobě 3 sloupců, kde první je velikost slovníku, druhý čas v ms za použití trie optimalizace a poslední je čas v ms bez trie optimalizace.

5.4.5 Vliv našich modifikací

V této části budeme spouštět algoritmus od Phana a spol. bez námi přidaných změn a s našema modifikacema na testovací sadě *ICDAR 2013*. Budeme zkoumat hlavně vliv restrikce následníka a využití deformační ceny.

Zajímat nás bude především zlepšení na atributech *recall* a *precision*.

Co se týče podoby ground truth a vyhodnocovací protokolu, je vše stejné jako při experimentu kompatibility algoritmů *ER* a *MSER* s detekováním slov. Pro detekci kandidátu využijeme algoritmus *ER*.

Výsledky Uveďme výsledky.

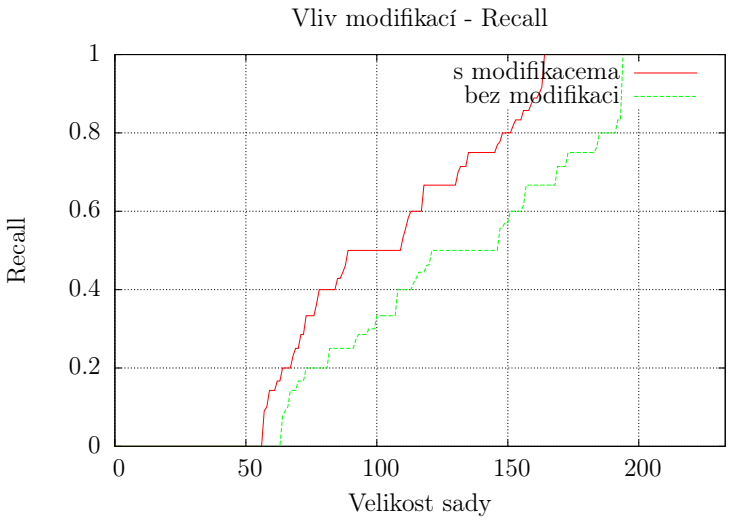
Testovací sada ***ICDAR 2013***		
metoda	*recall*	*precision*
bez vylepšení	0.43	0.62
s vylepšením	0.59	0.73

Tabulka 5.3: Výsledky bez modifikací a s modifikací

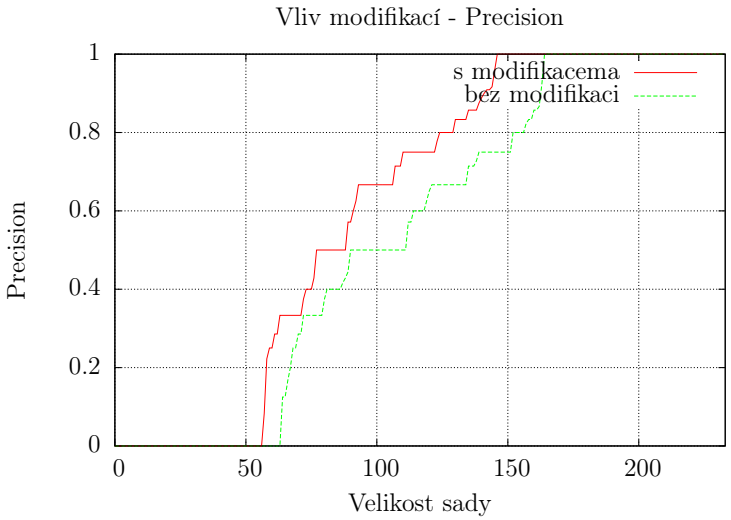
Jak vidíme v 5.3 námi navrhované modifikace přinesly netriviální zlepšení. Díky zavedení deformační ceny při výběru následovníka, jsme byly schopni dosáhnou zlepšení 0.16 v *recall* a zlepšení 0.11 v *precision*.

Uveďme opět vývoj růstu *precision* a *recall* vůči velikosti sady. Interpretace obou grafů je opět analogická jako v případě grafu růstu recall pro detekci písmen 5.3.6.

Z grafu 5.10 jasně vidíme, že zavedené modifikace mají velký vliv na *recall* detekce slov. Díky zavedení deformační ceny se nám například povedlo výrazně zvýšit počet obrázků, kde je $recall = 1$.



Obrázek 5.10: Vliv modifikací na detekci - recall

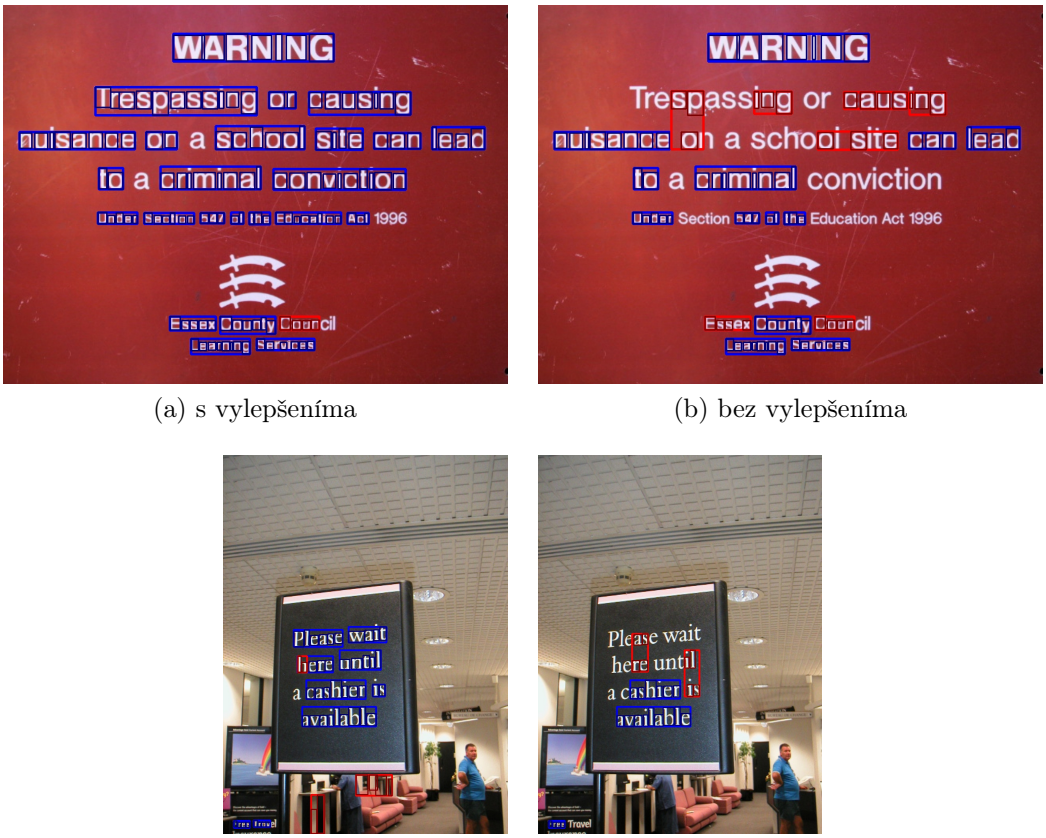


Obrázek 5.11: Vliv modifikací na detekci - precision

I v grafu 5.11 je vidět zlepšení přinesené námi navrhovanymá modifikacema. Opět platí, že pro každé $r \in [0, 1]$ bylo dosaženo dříve algoritmem od Phana a kol. s našima vylepšeníma.

Ukázky zlepšení pomocí našich modifikací Pro úplnost ještě ukažme v 5.12 vybrané výsledky z testovací sady *ICDAR 2013*. Už jenom z těchto ukázek je patrný pozitivní efekt našich modifikací.

Dodejme, že modře orámovaná slova jsou považována za *true positive* a červená slova jsou falešné detekce.



(a) s vylepšeníma (b) bez vylepšeníma

(c) s vylepšeníma (d) bez vylepšeníma

Obrázek 5.12: Ukázky zlepšení dosažených našima vylepšeníma

Znovuspustitelnost experimentu Pro znovuspuštění algoritmu slouží program *modification-evaluation*, který se používa úplně stejně jako program *icdar2013-evaluation*, více viz 5.4.3.

5.4.6 Závěr

Experimentálně jsme dokázali, že námi navrhované modifikace mají pozitivní efekt na schopnost algoritmu detekovat slova.

Nejdříve jsme zkoumali kompatibilitu algoritmu *MSER* [14] a *ER* [17] s algoritmem od Phana a kol. [22] viz. 5.4.3. Tento experiment prokázal lepší kompatibilitu algoritmu *ER* s algoritmem od Phana, čímž jsme experimentálně dokázali vhodnost volby našeho řešení navrhovaného v kapitole 4. Poté se nám povedlo v 5.4.4 experimentálně prokázat zrychlení za použití slovníkové trie. V poslední části experimentu 5.4.5 jsme zkoumali vliv námi navržených modifikací algoritmu od Phana a kol. došli jsme k závěru, že námi zakomponované změny přinesly netriviální zlepšení.

6. Závěr

V poslední části naší bakalářské práce mimo přílohy se budeme věnovat závěrečnému shrnutí. Nejdříve si rozebereme co jsme nestihli. Poté si naznačíme současné trendy a na konec nám zbýva jedině závěrečný povzdych.

6.1 Alternativní směrování práce

6.1.1 Detekce písmen

V oblasti detekce písmen jsme větší část pozornosti věnovali *component based* přístupům a přitom zanedbali *sliding window* metody. Bylo by možné tuto kategorii podrobněji popsat, detailněji rozebrat jednotlivé typy deskriptorů jako je *HoG* [4] a *SIFT*. Dále bychom mohli prozkoumat využití neuronových sítí. V oblasti experimentů by jistě bylo velmi zajímavé porovnat výkon zástupce *component based* přístupu a *sliding window* přístupu.

6.1.2 Detekce slov

V detekci slov jsme se hlavně soustředili na algoritmus od Phana a kol. [22]. Možná alternativní přístup ke prozkoumání, je ten který nepracuje s žádným slovníkem a spoléhá se pouze na geometrické vlastnosti textu a podobnosti jednotlivých písmen. Zajimavá by také mohla být analýza jiné algoritmu, který také využívá pro rozpoznávání slov slovník.

6.1.3 OCR fáze

V této práci jsme se rozhodli využívat OCR, které používá binární bitmapu kandidáta pro výpočet deskriptoru tkz. *Direction histogram* a následnou klasifikaci. Alternativní způsob je počítat deskriptor z přímo ze vstupní bitmapy, tj. přímo z části vstupu na obrázku, kde je zobrazené písmeno. Tento způsob zřejmě ještě sníží *confidence* pro falešné detekce po detekci písmen, ale na druhou stranu bude potřeba zvolit sofistikovanější deskriptor, velice pravděpodobně vyšší dimenze, což se odrazí na složitosti a výpočetní délce klasifikace. Každopádně takovýto přístup byl navržen v původní práci od Phana a kol. [22], zde se využívá pro klasifikaci deskriptor velikosti 3000 a bylo dále dosaženo rotační invariance. My tento způsob ale odmítli z výpočetních důvodů.

6.1.4 Ostatní trendy

V práci jsme věnovali poměrně moderním metodám v detekci textu v digitálních exteriérech. Ale záměrně jsme se nevěnovali nejnovějšímu trendu v oblasti počítačového vidění a tj. neuronové sítě. V posledních letech bylo v oblasti počítačového vidění úspěšně využity neuronové sítě, dokonce existují již práce, které je využívají pro detekci textu. Osobně bychom doporučili prozkoumat tento trend. Dále pro seznámení s existujícími metodami bychom určitě odkazovali na souhrn konference ICDAR.

6.2 Shrnutí práce

V této bakálářské práci jsme se postupně věnovali různým konceptům v oblasti detekce textu v digitálních exteriérech v obecnější rovině. Například jsme si popsali algoritmus využívající framework *sliding window*, který se hojně využívá v oblasti počítačového vidění. Poté jsme zaměřili veškerou pozornost algoritmům *MSER* a *ER*. Nejdříve byl představena struktura komponentový strom pro bitmapu. Poté jsme si popsali lineární verzi algoritmu, který tento komponentový strom sestavil v lineárním čase. Dále jsme probrali jakým způsobem lze vybrat z komponentového stromu *MSERY* a na základě tohoto popisu jsme popsali algoritmus *ER*.

V další části jsme se věnovali primárně algoritmu pro detekci slov z kandidátu(detekovaných písmen) se slovníkem [22]. Nejdříve jsme se věnovali navržšní vlastního *OCR* a následně jsme popsali původní algoritmus. V tomto popisu jsme nejprve zadefinovali tkz. jazykový model na základě, kterého detekujeme slova z kandidátů. Následně jsme popsali algoritmus, který využívá pro detekování slov tento jazykový model a funguje na principu dynamického programování. Tento algoritmus jsme dále modifikovali. Nejdříve jsme dodefinovali uvedený jazykový model v okrajových situacích, dále jsme ho rozšířili tak aby uměl zvládat více instancí jednoho slova na jednom vstupu. Dále jsme algoritmus netriviálním způsobem optimalizovali pomocí slovníkové trie a levenshteinovy vzdálenosti. Naše poslední modifikace se týkala změny běhu algoritmu, tato změna vedla k netriviálnímu zlepšení výkonnosti algoritmu. Pozitivní vliv námi navržených modifikací jsme navíc experimentálně ověřili.

V experimentální části jsme se ještě kromě zkoumání vlivu námi navržených modifikací, zkoumali vlastnosti zmíněných algoritmů *MSER*, *ER* a *SWT* pro detekci písmen, v tomto experimentu jsme zjistili, že algoritmus *ER* dosahuje největšího *recallu*, ale zároveň nejmenšího *precision*, více viz. 5.1. Následně jsme ověřovali kompatibilitu algoritmu *ER* a *MSER* s algoritmem pro detekováni slov s [22]. Jednim z našich hlavních předpokladů byla schopnost algoritmu detekování slov redukovat nizký precision algoritmu *ER*. Tento předpoklad jsme experimentálně dokázali, více viz. 5.2. V dalším experimentu jsme zkoumali zrychlení dosažené při detekci slov v připadě že použijeme optimalizaci pomocí trie. Z grafu 5.9 je jasně vidět, že se nám povedlo dosáhnout netriviálního zrychlení. V posledním experimentu jsme zkoumali vliv našich modifikací zejména zavedení deformační ceny na *precision* a *recall* při detekci slov v testovací sadě *ICDAR 2013*. V tabulce výsledků zlepšení 5.3 je jasně vidět že naše modifikace přináší netriviální zrychlení.

Další součástí této práce je příloha. V příloze nalezneme uživatelskou a softwerovou dokumentaci. V uživatelské dokumentaci naleznete manuál pro použití vytvořeného softwaru a znovu spuštění experimentů. V softwarové specifikaci jsme se zaměřili hlavně na popis námi naimplementované knihovny NOCRLib.

Přílohy

7. Teoretická příloha

V této kapitole si popíšeme algoritmy, které nejsou součástí navrhované metody 4, ale buď se využívají v experimentech a nebo jsou součástí námi vytvořené knihovny.

7.1 Stroke Width Transform

Jedná se o transformaci vstupní bitmapy původně navrženou od Epshteina a spol [5]. Výstupem této transformace je bitmapa stejných rozměrů, kde v každém pixelu je uložena šířka tkz. stroke, do jenž patří. Stroke jsou dle Epshteina a spol souvislá části bitmapy, jenž spolu tvoří nějaký pás pixelů v bitmapě relativně konstantní délky.

Algoritmus

V naší práci nebudeme uvádět algoritmus od Epshteina a spol. [5], ale algoritmus od Chena a spol. [2] Tento algoritmus se liší od původního uvedeného v v již zmíněné práci pd Ephsteina způsobem výpočtu. Zásadní změnou oproti původního algoritmu je že metoda od Chena, předpokládá jako vstup binární vstup narozdíl od původního algoritmu, kdežto algoritmus od Ephsteina očekává na vstupu bitmapu ve stupni šedi. Změna je zapříčeněna odlišným principem výpočtu. Epstein a spol využívají pro výpočet střílení paprsků dle gradientu z hranových pixelů, viz detaily [5], kdežto metoda navržená od Chena, ale využívá distance transformaci, což je transformace binarního bitmapy, do grayscale bitmapy kde, každý nenulový pixel v původní bitmapě má v transformaci hodnotu rovnou vzdálenosti k nejbližšímu nulovému pixelu.

Detailní popis algoritmu Předtim než si uvedeme algoritmus. Definujme bitmapu jako zobrazení $I : D \subset \mathbb{Z}^2 \to \{0, 1, \dots 255\}$ a pro $p = (x, y) \in D$ definujme množinu

$$N_8(p) = \{(x-1, y-1), (x-1, y), (x-1, y+1), (x, y-1), \\ (x, y+1), (x+1, y-1), (x+1, y), (x+1, y+1)\}$$

Poté algoritmus pro SWT od Chena a kol. [2] vypadá takto.

Algoritmus 11: Stroke Width Transformace

Vstup: Binární bitmapa B
Výstup: Stroke width transformace SW

```
1  D ← DistanceTransformation(B);
2  D ← Round(D);
3  foreach p nenulový pixel in D do
4  |  pVal ← D(p);
5  |  LookUp(p) ← {q | q ∈ N_8(p) ∧ D(q) < pVal};
6  end
7  MaxStroke ← max(D);
8  for Stroke = MaxStroke downto 1 do
9  |  StrokePixel ← find(D==index);
10 |  StrokeNeighbourPixels ← LookUp(StrokeIndex);
11 |  while StrokeNeighbourPixels není prázdný do
12 |  |  D(StrokeNeighbourPixels) ← Stroke;
13 |  |  StrokeNeighbourPixels ← LookUp(StrokeNeighbourPixels);
14 |  end
15 end
16 SW ← D;
17 return SW;
```

Z uvedeného popisu vidíme, že nejdříve algoritmus vypočítá distance transformaci vstupu, nazveme ji D, následně hodnoty D zaokrouhlíme. Z vlastností distance transform plyne, že pro každý pás přibližně konstatní šířky platí, že pixely uvnitř ve středu pásu mají největší D(p) s ohledem na ostatní pixely pásu. Dále lze také nahlédnout, že hodnoty pixelů uprostřed pásů jsou zhruba polovina šířky stroke. Pro výpočet hodnoty stroke width pixelů, tedy stačí propagovat hodnoty pixelů uprostřed pásu až k pixelům na okrajích pásů. Tohoto algoritmus dosahuje pomocí struktury LookUp a následném průchodu do šířky od pixelů ve středu pásu.

Aplikace v rozponávání písmen

Všimněme si, že pro každou písmenovou komponentu na bitmapě je charakteriscké, že všechny její pixely budou mít podobnou hodnotu ve stroke width transformaci vstupu. Toto je důležitý poznatek pro detekci písmen.

Detekce písmen pomocí SWT Dejme tomu, že máme vstup a chceme na něm detekovat písmena pomocí stroke width transform. Nejprve provedeme SWT vstupu, v něm nalezneme komponenty souvislosti na základě podobnosti hodnot pixelů. Následně vyfiltrujeme komponenty na základě geometrických vlastností a rozptylu hodnot pixelů komponety v SWT vstupu. V případě většího rozptyly komponentu zamítáme.

Dodejme, že Ephstein a spol. [5] vyčíslili práh na testování rozptylu MSERU, jako $2{*}swt\_mean$, kde swt_mean je průměr hodnot pixelů patřících komponentě.

V případě Chenovy verze je třeba vstup nejprve binarizovat, toto se dá provést třeba pomocí binarizace nebo algoritmu MSER. V případě využití algoritmu

MSERU, stačí pro binarizaci vytvořit bitmapu, kde nenulovou hodnotu budou mít pouze pixely patřící nějakému extrahovanému MSERU.

Využití v naší práci

V práci byl tento algoritmus využit v experimentech pro porovnání s alternativníma metodama pro detekci písmen 5.3.1.

7.2 SVM s fast intersection kernel

Úvod do SVM SVM je klasifikátor s učitelem. Pro jednoduchost předpokládejme, že máme binární SVM. Takovýto klasifikátor separuje pozitivní trénovací vzorky od negativních pomocí jedné nebo více nadrovin.

Avšak ve většině situací v klasifikaci nedostáváme lineárně separovatelná trénovací data. Tento problém řeší SVM za pomocí takzvaného kernel triku, který namapuje trénovací data, do prostoru vyšší dimenze, kde je již lze lineárně separovat. Mezi oblíbená kernely patří například radial basis funkce a lineární kernel. Pro úplnost si pojmy ještě zadefinujme matematicky.

Nechť máme trénovací data $\{(x_i, y_i)\}_{i=1}^{N}$, kde $x_i \in \mathbb{R}^n$ je příznakový vektor a $y_i \in \{-1, +1\}$ je label daného příznaku. Dále máme kernel $k(x, z)$, což je skalární součin $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$. Potom při využití C-formulace svm problému algoritmus najde separátor v namapovaném prostoru pomocí maximalizace

$$\tau(w, \xi) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} \xi_i$$

za podmínek $y_i(w \cdot x_i) \geq 1 - \xi_i$ a $\xi_i \geq 0$.

V duální formulaci, platí že minimalizujeme

$$W(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j k(x_i, x_j)$$

za podmínek $0 \leq \alpha_i \leq C$ a $\sum \alpha_i y_i = 0$. Za decision funkci považujeme $sgn(h(x))$, kde

$$h(x) = \sum_{l=1}^{m} \alpha_l y_l k(x, x_l) + b$$

kde vektory $\{x_l\}_{l=1}^{m}$ říkáme support vectory. Ze vzorce vidíme že pro klasifikaci jednoho bodu, je zapotřebí čas $O(nm)$, jelikož potřebujeme vypočítaní $k(x, x_l)$ trvá $O(n)$ a počítá se m krát.

SVM pro více tříd SVM pro klasifikaci s více třídami lze využít print one agains all. Kde vytrénujeme SVM pro každou dvojici tříd. A při klasifikaci si počítáme pro každou třídu kolikrát byl daný vzorek do ní zařazen. Výsledná třída je ta s největším počtem zařazení. Tento princip je využit v knihovně LibSVM [1].

Intersection kernel Uvedené definice a vzorce jsme převzali z práce [13]. Nechť máme $x, z \in \mathbb{R}^n$, potom definujme intersection kernel jako

$$k(x, z) = \sum_{i=1}^{n} min(x(i), z(i))$$

kde $x(i)$ je i-tá položka vektoru x, analogicky definujme $z(i)$.

Dosaďme tento vzorec do vzorce pro $h(x)$,

$$h(x) = \sum_{l=1}^{m} \alpha_l y_l \sum_{i=1}^{n} min(x_l(i), x(i)) + b$$

Přímé vyhodnocení $h(x)$ bude trvat již zmíněných $O(nm)$, kde n je dimenze prostory příznaků a m je počet support vectorů. Zkusme ještě $h(x)$ poupravit. Nejdříve se zbavíme závorky roznásobením a následně prohodíme sumy.

$$h(x) = \sum_{l=1}^{n} \Big(\sum_{i=1}^{m} \alpha_l y_l min(x_l(i), x(i)) \Big) + b$$

$$h(x) = \sum_{i=1}^{n} h_i(x) + b$$

kde $h_i(s) = \sum_{l=1}^{m} \alpha_l y_l min(x_l(i), s)$. Nyní ukažme, že funkci h_i lze vyčíslit v $O(logm)$.

Předpokládejme, že máme funkci $h_i(s)$ pro pevně dané i. Nechť $\bar{x}_l(x)$ jsou vzestupně vzestupně hodnoty $x_l(x)$ a $\bar{\alpha}_l$ indexy korespondujícími alpha, analogicky definujme $\bar{y}_l$. V případě že $s \leq \bar{x}_l(i)$, platí že $h_i(s) = 0$, protože

$$h_i(s) = \sum_{l=1}^{m} \bar{\alpha}_l \bar{y}_l s$$

$$h_i(s) = s \sum_{l=1}^{m} \bar{\alpha}_l \bar{y}_l$$

$$h_i(s) = s \cdot 0 = 0$$

Jinak nechť platí, že r je největší takové že $\bar{x}_r(i) \leq s$. Potom lze h_i přepsat jako

$$h_i = \sum_{l=1}^{m} \bar{\alpha}_l \bar{y}_l min(\bar{x}_l(i), s)$$

$$h_i = \sum_{l=1}^{r} \bar{\alpha}_l \bar{y}_l \bar{x}_l(i) + s \sum_{j=r+1}^{m} \bar{\alpha}_l \bar{y}_l$$

$$h_i = A_i(r) + sB_i(r)$$

kde

$$A_i(r) = \sum_{l=1}^{r} \bar{\alpha}_l \bar{y}_l \bar{x}_l(i)$$

$$B_i(r) = \sum_{j=r+1}^{m} \bar{\alpha}_l \bar{y}_l$$

Z čehož plyne že funkce h_i je po částech lineární. Navíc se také jedná o spojitou funkci, protože:

$$h_i(\bar{x}_{r+1}) = A_i(r) + \bar{x}_{r+1} B_i(r) = \sum_{l=1}^{r} \bar{\alpha}_l \bar{y}_l \bar{x}_l(i) + \bar{x}_{r+1} \sum_{l=r+1}^{m} \bar{\alpha}_l \bar{y}_l$$

$$= \sum_{l=1}^{r} \bar{\alpha}_l \bar{y}_l \bar{x}_l(i) + \bar{x}_{r+1} \bar{\alpha}_{r+1} \bar{y}_{r+1} + \bar{x}_{r+1} \sum_{l=r+2}^{m} \bar{\alpha}_l \bar{y}_l$$

$$= \sum_{l=1}^{r+1} \bar{\alpha}_l \bar{y}_l \bar{x}_l(i) + \bar{x}_{r+1} \sum_{l=r+2}^{m} \bar{\alpha}_l \bar{y}_l = A_i(r+1) + \bar{x}_{r+1} B_i(r+1)$$

Všímněme si že funkce A_i a B_i nejsou závislé na vstupu s, ale jenom na support vektorech a α. Poté lze předpočítáním hodnot $h_i(\bar{x}_l)$, pro každé $l \in \{1 \dots m\}$ vyčíslit hodnoty $h_i(s)$ následujícím způsobem nejdříve najdeme pozici r dle výše uvedené definice pomocí binárního vyhledávání, následně $h_i(s)$ vypočítám pomocí lineární interpolace mezi $h_i(\bar{x}_r)$ a $h_i(\bar{x}_{r+1})$. Hodnotu $h_i(s)$ lze vypočítat tímto způsobem díky tomu, že $h_i(s)$ je po částech lineární funkce a navíc je spojitá. Tímto trikem jsme dosáhli složitosti vyčíslení $h(x)$ za $O(n \ \log m)$, kde $O(\log m)$ trvá binární vyhledávání.

Aproximace h(x) v $O(n)$ Malik a spol. v [13] dále navrhují aproximovat funkci $h(x)$. K tomuto využívají zjištění, že $h_i(s)$ jsou po částech lineární spojité funkce. Autoři [13] pozorují v jejich experimentech, že tyto funkce jsou poměrně hladké a tedy navrhují $h_i(s)$ aproximovat pomocí $k+1$ úseček. V jejich experimentech pozorují, že volba $k \in \{30 \dots 50\}$, je dostatečná pro přesnost klasifikace.

Pro naše OCR, jsme vypozorovali že volba $k = 10$ je dostačující.

Využití v práci Tento algoritmus sice nebyl nikde využit přímo v této bakalářské práci, ale je součástí naší knihovny, ale byl vyzkoušen v rámci ladění experimentů jako klasifikátor pro *OCR*. V nichž jsme, zjistili, že kromě 2 násobného zrychlení nic nepřinesl.

Implementace Informaci o naši implementaci jsme uvedli 9.8.2.

8. Uživatelská dokumentace

8.1 Úvod

8.1.1 Rozpoznávání textu na fotografii

Cílem rozpoznávání textu na obrázku pořízeném digitalním přístrojem je detekovat slova na vstupu a převést je do ASCII kodu pro další zpracování.

8.2 NOCR Software

NOCR Software je softwarový balík pro detekci a převod slov na obrázku do textové podoby se slovníkem. To znamená, že je schopen detekovat slova, která jsou jen obsažena ve slovníku. Balík obsahuje gui aplikaci NOCRGui, aplikaci spouštěnou v příkazovém řádce NOCR a knihovnu pro detekci slov na digitálních snímcích se slovníkem NOCRLib.

8.2.1 Systemové požadavky

Celý balík je vyvíjen v C++ s využitím standartu C++11 pod unixovou platformou s využitím kompilátoru g++ verze 4.8.3.

NOCRLib využívá kompilaci knihovny OpenCV knihovnu pro počítačové vidění, LibSVM a LibLinear knihovny implementující algoritmy SVM. Poznamenejme, že používáme modifikovanou verzi LibSVM od Malika and co. s Intersection jádrem, samy jsme ale také provedli malé změny.

NOCR využívá knihovnu NOCRLib a tedy všechny knihovny, které jsou vyžadovány od NOCRLib, dále využívá knihovnu pugixml a modul knihovny Boost pro zpracování příkazového řádku a modul s memory poolem.

NOCRGui je aplikace, která opět využívá služeb knihovny NOCRLib, a tudíž potřebuje všechny knihovny vyžadované knihovnou NOCRLib, aplikace je navíc vyvíjena za pomocí QT5 knihovny v prostředí Qt Creator. Obě tyto položky je potřeba mít nainstalované.

Pro kompilaci knihovny je dále potřeba mít nainstalovaný CMake.

8.2.2 Obsah instalačního balíčku

Instalační balíček obsahuje složku NOCRSoftware. Ta obsahuje kromě veškerého naprogramovaného softwaru také modifikovanou verzi LibSVM, pugixml, různé konfigurátory pro NOCRGui a NOCR, testovací sadu ICDAR 2013 a konfigurační soubor pro program CMake CMakeLists.txt.

Knihovny Qt, Boost, OpenCV a program CMake nedodáváme, proto je potřeba je před použitím nejdříve nainstalovat.

Balíček dodáváme jako zkomprimovaný soubor nocr.tar.bz2. Pro úplnost ještě ukažme stromovou strukturu balíčku.

```
NOCR
├── NOCRLib ..... zdrojové soubory knihovny NOCR
└── NOCRMain ..... zdrojové soubory programu nocr
```

- NOCRGui ..... zdrojové soubory programu nocr-gui
- ICDAR 2013 ..... testovací sada icdar 2013
- Experiments-programs ..... zdrojové soubory k experimentům
- External-libraries ..... externí knihovny
- CMakeLists.txt ..... CMake configurator
- bin ..... binární soubory
  - conf ..... různé konfigurátory
  - experiment input ..... vstupy pro experimenty

8.2.3 Obsah multimedia

Součástí práce je také DVD multimedium z dodatečnýma soubory. Nyní uveďme strukturu obsahu balíčku. To již obsahuje knihovny Boost a OpenCV.

- NOCR ..... NOCR balíček
- OpenCV 2.4.11 ..... knihovna opencv 2.4.11
- Boost 1.55.0 ..... knihovna boost 1.55.0
- Výsledky experimentů
  - icdar2013 výsledky ..... obrázky z vydetekovanými slovy
  - letter-segmentation ..... xml soubory s informacema
  - word-detection ..... xml soubory s informacema
  - performace ..... soubor s vysledky měření
  - modification ..... xml soubory s informacema

8.2.4 Instalace

Návod pro instalaci externího softwaru

Qt knihovnu a Qt IDE lze stáhnout na adrese `http://www.qt.io/download/`, stačí community verze. Pro instalaci Boost C++ library doporučejeme postupovat dle uvedených instrukcí na adrese `http://www.boost.org/doc/libs/1_55_0/more/getting_started/unix-variants.html`. Postup a odkaz ke stažení knihovny OpenCV lze najít na `http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html`. Postup pro instalaci CMaku lze nalézt na adrese `http://www.cmake.org/install/`.

Kompilace NOCRSoftware

Předpokládejme, že už byly nainstalovaný všechny externí knihovny. NOCRLib a NOCR rozbalíme a zkompilujeme následujícím způsobem.

```
1    $ tar -xvjf nocr.tar.bz2
2    $ cd NOCRSoftware
3    $ Konfigurace softwaru
4    $ mkdir release && cd release
5    $ cmake -D CMAKE_BUILD_TYPE=RELEASE ..
6    $ make
```

Následně by se vám ve složce NOCRSoftware měly vytvořit složky bin a lib. Kde ve složka bin bude obsahovat program NOCR a lib bude obsahovat knihovnu libNOCRLib.so a další externí knihovny.

Konfigurace kompilování Jestliže chceme zkompilovat pouze knihovnu NOCRLib, stačí v souboru NOCRSoftware/CMakeLists.txt zakomentovat(na začátek řádky napsat #) následující řádky:

```
1    add_subdirectory(NOCRMain)
2    add_subdirectory(NOCRGui)
```

V případě že navíc zkompilovat ještě program NOCR ale ne program NOCRGui, stačí zakomentovat řádek

```
1    add_subdirectory(NOCRGui)
```

Analogicky pro kompilaci NOCRLib a NOCRGui bez NOCR zakomentujeme řádek

```
1    add_subdirectory(NOCRMain)
```

Konfigurace kompilování NOCRGui Před tim než, je možná kompilace programu NOCRGui je potřeba v souboru NOCRSoftware/NOCRGui/CMakeLists.txt nastavit proměnnou na Qt5Path na cestu ke knihovně Qt5. Stačí modifikovat příkaz na řádce 16

```
1    set( Qt5Path <cesta ke Qt5 knihovne>)
```

Cesta ke knihovně Qt5, může vypadat jako /opt/Qt5.x.x/5.x.x/<kompilátor>.

8.2.5 Návod k použití

NOCRLib

Pro použití NOCRLib v externím projektu, stačí nalinkovat statickou knihovnu libNOCRLib.a ve složce lib a includovat headery ve složce NOCRSoftware/NOCRLib/include k vašemu projektu.

Pro důkladnější seznámení s knihovnou NOCRLib doporučujeme prostudovat referenční příručku a prográmatorskou dokumentaci. Rozpoznávání textu na fotce se slovníkem obstará třída TextRecognition definovaná v headeru nocrlib/text_recognition.h. Příklad použití lze vidět v souboru NOCRMain/main.cpp.

NOCR

Vstupem programu je seznam obrázků, ze kterých budeme rozpoznávat text za pomocí slovníku. Všechny vstupní obrázky musí být ve formátech jpg, png, bmp a tiff. Výstupem jsou udáje o detekovaných slovech a to buď v podobě xml souboru nebo textového souboru s údaji o detekovaném vstupu.

Vstup Nejdříve si popišme formát parametrů příkazové řádky programu NOCR. Zde je ukázka nápovědy k programu, ta se zobrazí za pomocí parametrů -h, –help nebo když je program volán bez argumentů.

```
1    -h [ --help ]         display help message
2    -i [ --input ] arg    list of paths to images
```

```
3     -o [ --output ] arg   specifies output file
4     -d [ --dictionary ] arg specifies dictionary
5     --xml                 enable xml output
6     --display-words       enable displaying of detected words
7     --display-letters     enable displaying of detected letters
```

Vidíme, že formát parametrů příkazové řádky dodržuje pravidla posixové konvence. Poznamenejme, že argument na příkazové řádce, který není spjat s žádným parametrem, program automaticky považuje za vstup, na němž budeme rozpoznávat text.

Popišme jednotlivé parametry:

- -h [- -help], zobrazí nápovědu
- -i [- -input], vstupní soubor, kde je na každé řádce jeden vstup pro rozpoznání textu
- -o [- -output], výstupní soubor, kam se budou zapisovat údaje o detekovaných vstupech, defaultně vypisuje na stdout.
- -d [- -dictionary], specifikuje slovník s kterým budeme provádět rozpoznávání vstupu, defaultně pracuje se slovníkem pro testovací sadu ICDAR 2013
- - -xml, místo klasického textového výstupu, bude výstupem xml
- - -display-words zapne zobrazení detekovaných slov
- - -display-letters zapne zobrazení detekovaných písmen

Uveďme příklad

```
1 $ ./NOCR test.jpg --input="listtest" test1.jpg \
2               -o output --xml --display-words \
3               -d dictionary
```

Tento příkaz, bude rozpoznávat text na snímcích *test.jpg* a *test1.jpg* a na snímcích uvedených v souboru *listtest*. Program považuje každou řádku souboru *listtest* za jeden vstupní snímek pro rozpoznávání textu. Rozpoznávaní textu proběhne za pomocí slovníku *dictionary*, kde jsou slova odděleny od sebou libovolným množstvím whitespaců. Během rozpoznávání nám program bude průběžně ukazovat detekovaná slova. Výsledky rozpoznávání budou zapsány do souboru *output* a to v xml formatu.

Výstup Jak již bylo NOCR umí produkova dva různé formáty výstupu. A to jest defaultní a výstup ve formátu XML.

Defaultním typ výstupu má následující formát. Pro každý vstupní snímek se vypíše cesta ke vstupu a následně seznam všech detekovaných slova a pozic na snímku. Kde každá řádka v tomto seznamu je informace o jednom detekovaném slově. Řádka má tvar *text:souřadnice x:souřadnice y:šířka:výška*. Záznamy pro jednotlivé snímky jsou od sebe odděleny separátorem.

Druhým typem výstupu je formát XML. Kořen XML dokumentu je tag *text-detection*. Ten následně obsahuje tagy *image*. Tag *image* obsahuje dva tagy *path-to-image*, ten nám říká kde byl vstupní snímek uložen a list tagů *word*, který uchovává informace o detekovaném slově na vstupním snímků.

Prohlédněme si ukázky obou příkladů výstupů.

Listing 8.1: Ukázka defaulního výstupu

```
test.jpg
HURRY:303:255:221:50
WHY:98:256:179:55
================================================
test1.jpg
NATURAL:231:199:332:55
================================================
```

Listing 8.2: XML výstup

```
<?xml version="1.0"?>
<text-detection>
    <image>
        <path-to-image>test.jpg</path-to-image>
        <word>
            <text>HURRY</text>
            <bounding-box x="303" y="255"
                        width="221" height="50"/>
        </word>
        <word>
            <text>WHY</text>
            <bounding-box x="98" y="256"
                        width="179" height="55"/>
        </word>
    </image>
    <image>
    <path-to-image>test1.jpg</path-to-image>
    <word>
        <text>NATURAL</text>
        <bounding-box x="231" y="199"
                        width="332" height="55" />
    </word>
    </image>
</text-detection>
```

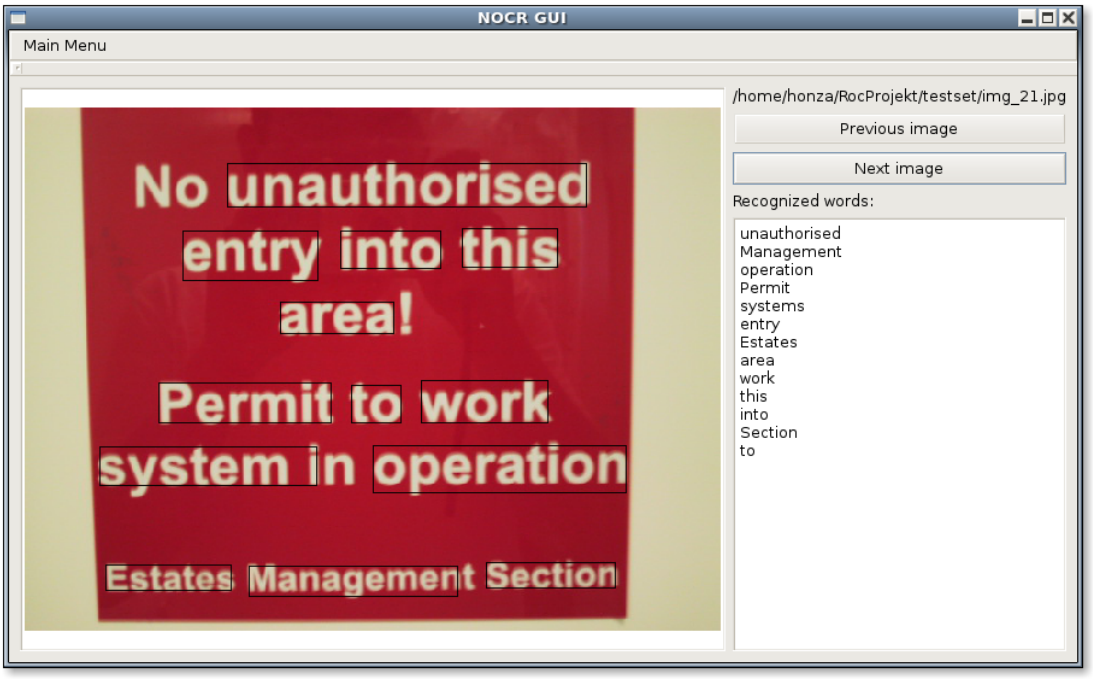
Z obou výstupu plyne, že jsme na snímku test.jpg rozpoznali slova WHY, HURRY a na snímku test1.jpg slovo NATURAL.

NOCRGui

NOCRGui je GUI varianta programu NOCR. Program umí rozpoznávat text z digitáních snímku formátu jpg, png, bmp se slovníkem. Snímky s rozpoznanými slovy program následně vykresluje ve svém GUI. Program dále umí během běhu načítat nová slova ze soubory nebo přímo od uživatele, načítat nový slovník a uložit současný slovník. NOCRGui dále umí exportovat výsledky rozpoznávání textu do xml souboru, formát je stejný jako v případě NOCR.

Vidíme, že aplikace zobrazuje vstupní snímek, dále napravo od snímku se nachází tlačítka Next image a Previous image sloužící pro pohyb mezi zpracovanými vstupy. Pod těmito tlačítky se nám vždy zobrazují rozpoznaná slova na daném vstupu. Nad tlačítky je vždy údaj o cestě k danému vstupu.

Obrázek 8.1: Ukázka aplikace NOCRGui



Menu V menu aplikace NOCRGui lze otevirát nové obrázky pomocí položky *Open image*, dále položka *Clear results* smaže všechny záznamy provedených rozpoznávání textu. Za pomocí *Export to XML* exportujeme výsledky rozpoznáváná do XML souboru.

Menu také obsahuje podmenu *Dictionary*, v tom lze pomocí položek v tomto podmenu lze již dělat zmíněné operace se slovníkem dle příslušných tlačítek. Poznamenejme že po stisknutí tlačítka *Add words to dictionary* se objeví dialog s textovým polem, do kterého zapisujeme nová slova. Slova jsou od sebe oddělena libovolným počtem whitespaců.

9. Implementace

Program byl naprogramován v jazyce C++ za použití standartu c++11 pod linuxovou platformou. Kód byl kompilován GCC verze 4.8.3. Pro naprogramování GUI aplikace byly použity knihovny Qt.

9.1 Využité knihovny

OpenCV 2.4.11
: C++ knihovna pro zpracování obrazu, obsahuje jak velké množství algoritmů ze zpracování obrazu a počítačového vidění tak i modul pro strojové učení.

LibSVM
: Rozsáhlá knihovna [1] pro práci s SVM implementovaná v C++. Obsahuje různé kernel funkce, SVM pro klasifikaci do více než dvou tříd.

9.2 Návrh knihovny

Knihovna NOCRLib se dá rozdělit do dvou části. Ta první se stará o extrakci písmenových kandidátů ze vstupu, ta druhá řeší generování slov dle slovníku z písmenových kandidátů.

První část reprezentuje generická třída Segment<T, OCR>. Na té se voláním metody segment provede detekce písmenových kandidátů a non-max suppresion. Metoda *segment* očekává na vstupu instanci cv::Mat, což je třida reprezentující obrázek v OpenCV. Výstupem metody *segment* je std::vector<Letter>, kde Letter je třída reprezentující písmenového kandidáta.

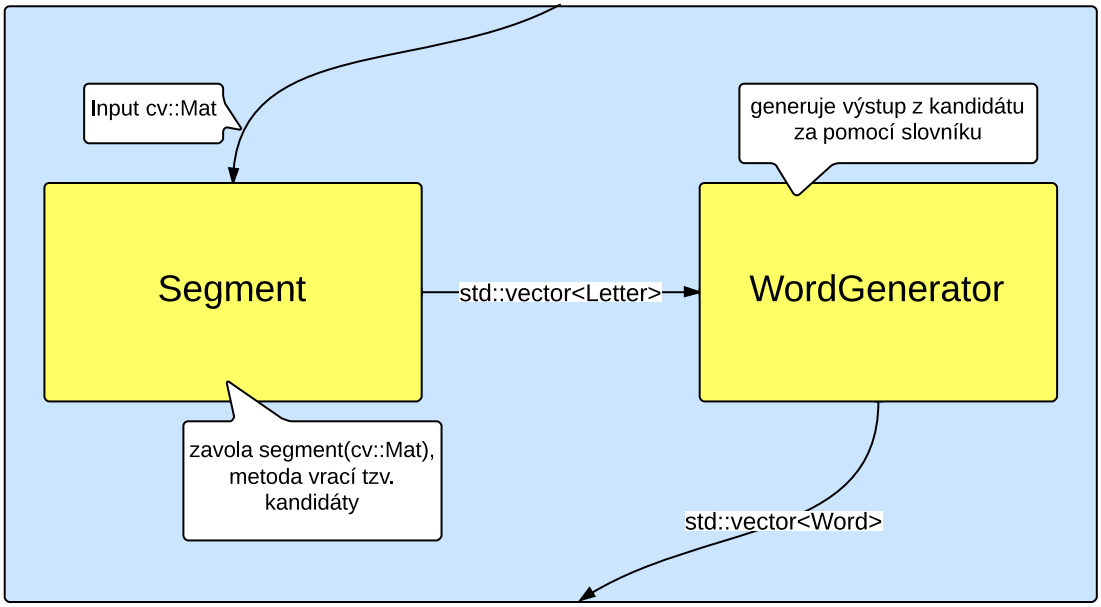
Druhou část reprezentuje třída *WordGenerator*, která implementuje algoritmus popsaný v 13 bez non-max suppresion. Na vstupu očekává vektor objektů typu Letter . Jejím výstupem je std::vector<TranslatedWord>, kde TranslatedWord je třída představující jedno slovo na vstupu.

V následujících sekcích si popíšeme popíšeme obě části knihovny.

9.3 Segment

Jedná se o generickou třídu parametrizovanou třídou zajišťující extrakci písmen a typem OCR. Výstupem metody pro vyextrahování kandidátů segment je vector instancí třídy Letter. Třída Segment<T, OCR> neimplementuje extrakci písmen samotnou, ale pouze zajišťuje integraci detekce písmen s non-max suppresion a unifikací výstupu různých typů extrakce, tak aby byly kompatibilní se třídou WordGenerator.

Jednotlivé kroky integrace libovolného typu extrakce a non-max suppresion pomocí OCR jsou podobné a mohou se lišit jenom v malých detailech. Proto byl v implementaci využit pomocí policy class design, což je specializovaná C++ obdoba návrhového vzoru template method, využívající generického programování.



Obrázek 9.1: Integrace Segment a WordGenerator

Využití policy class umožnuje velice snadnou rozšiřitelnost kódu a jednoduchou integraci libovolným typem extrakce s non-max suppresion.

9.3.1 Letter

Letter je třída vzniklá kompozicí 2 různých tříd. Z nichž každá reprezentuje jinou informaci o detekovaném kandidátu. První třída Component zachycuje informaci o geometrických vlastnostech vysegmentované komponenty, to jest její pozice, rozměry a seznam všech komponentových pixelů. Druhá třída TranslationInfo reprezentuje lexikografické údaje o kandidátovi, jeho confidence a $P(l \mid c)$ pro každý label l v abecedě.

9.3.2 Policy class

Parametr T generické třídy Segment<T, OCR> definuje typ použité extrakce a OCR typ *ocr* použité pro non max supressy. Jednotlivé kroky algoritmu se upřesňují dle politiky SegmentPolicy<T>, která musí obsahovat požadované typedefy a statické funkce a konstanty, které využívá Segment<T, OCR>.

Proto abychom mohli využít typ extrakce T, musíme vždy třídu SegmentPolicy<T> specializovat pro daný parametr T. Ukázka nespecializované třídy SegmentPolicy<T> obsahuje zakomentované všechny nutné typedefy, statické konstanty a funkce. Různýma politikama můžeme snadno měnit chování třídy Segment<T>.

Listing 9.1: Politika pro segment

```
1 // NOCRLib/segment.h
2 template <typename S> class SegmentationPolicy
3 {
4     // nutne typydefy
5     /*
```

```
 6      * required typedefs
 7      * MethodOutput type of method output after letter segmentation
 8      * VisualConvertor type of class that creates VisualInformation
 9      * from MethodOutput
10      */
11
12     // staticke konstanty
13     /*
14      * required static constant
15      * static const bool k_perform_nm_suppresion
16      */
17
18     // staticke funkce
19     /*
20      * following functions are neccesery to implement for integration
21      * with Segment<S>
22      *
23      * static void initialize( VisualConvertor &visual_convertor,
24      *                         const cv::Mat &image );
25      *
26      * static std::vector<MethodOutput> extract
27      *                       ( const std::unique_ptr<S> &method_ptr,
28      *                         const cv::Mat &image );
29      *
30      * static bool haveSignificantOverlap
31      *               ( const MethodOutput &a,
32      *                 const MethodOutput &b );
33      *
34      * static Letter convert
35      *                  ( const MethodOutput &a,
36      *                    const TranslationInfo &translation );
37      */
38 };
```

Popis částí SegmentPolicy<T> V následující části popíšeme jednotlivé součásti politik, a jejich účel.

Typedefy

- *MethodOutput*
  typ výstupu metody

Statické konstanty

- *k_perform_nm_suppresion*
  Statická konstanta typu bool, která nám říka jestli se má provést non-max supprese, v některých případech extrakce může být non-max supprese zbytečná

Statické funkce

- *initialize*
  nastaví instanci VisualConvertoru na vstup
- *extract*
  vyextrahuje vector všech potencionálních kandidátu, pomocí instance T
- *haveSignificantOverlap*
  určí jestli se dvě instance MethodOutput navzájem překrývají
- *convert*
  vytvoří instanci třídy Letter

Vidíme že na řádce 6 využijeme statickou funkci extract politiky, pomocí níchž vyextrahujeme vektor<MethodOutput>, kde MethodOutput, potom následuje inicializace OCR a visual converteru pomocí statické funkce initialize. Následně jestli je flag k_perform_nm_suppresion, politiky nastaven na true provede se non-max supprese. Jinak se non-max supprese neprovádí a všechny kandidáty convertujeme na instance Letter, které vracíme z metody segment.

9.3.3 Integrace OCR

Pro integraci různých OCR se opět dá musí použít policy class pattern. Kdy je potřeba specializovat třídu SegmentOcrPolicy<OCR, MethodOutput> dle daného OCR a výstupu typu extrakce. Povinná metoda pro naimplementování je statická metoda translate, která vezme pointer na ocr a vektor instancí třídy MethodOutput a pro každou tuto instanci zkonstruje za pomocí OCR třídu TranslationInfo.

Ukázka kódu metody segmentu
Nyní ukážeme kód metody segment třídy Segment<T>, na němž budeme demonstrovat jakým způsobem policy class SegmentPolicy<T> ovlivňuje chování třídy Segment<T>.

Listing 9.2: Metoda segment třídy Segment<T>

```
1   template <typename T>
2   std::vector<Letter> Segment<T>::segment(const cv::Mat &image)
3   {
4       typedef typename Segment<T>::MethodOutput MethodOutput;
5
6       std::vector<MethodOutput> letter_candidates =
7              SegmentationPolicy<T>::extract( method_ptr_, image );
8       ocr_->setImage( image );
9
10      SegmentationPolicy<T>::initialize( visual_convertor_, image );
11
12      if ( SegmentationPolicy<T>::k_perform_nm_suppresion )
13      {
14          return nonMaxSuppresion( letter_candidates );
15      }
16
17      std::vector<TranslationInfo> translations =
            SegmentOcrPolicy<OCR,
            MethodOutpu>::translate(letter_candidates);
18      // every letter candidate is extracted because
19      // mask is set true for all
20      // of them, this means that we consider
21      // them all to be maximal.
22      std::vector<bool> mask( letter_candidates.size(), true );
23      return extractMaximal( letter_candidates, translations, mask );
24  }
```

9.3.4 Rozšiřitelnost o nový typ extrakce

Řekněme, že jsme si naprogramovali nový typ extrakce Example písmenových kandidátů ze vstupu. Výstupem této extrakční metody bude vector komponentových bodů. Tuto extrakci budeme chtít zaintegrovat do třídy Segment<T>.

Jediné co musíme udělat je specializovat politiku SegmentPolicy<T>, pro náš nový typ extrakce. Jako MethodOutput budeme definovat vector bodů. Jako VisualConvertor třídu, která vypočítá všechny nutné atributy. Jestliže budeme potřebovat non-max suppresi tak bool k_perform_nm_suppresion nastavíme na true jinak na false. Poté doimplementujeme zbývající požadované statické funkce.

Specializováním této politiky jsme zaintegrovali novou extrakci do naší knihovny a tedy jsme ji udělali kompatibilní se třídami Segment a WordGenerator.

9.3.5 Shrnutí

Generická třída Segment<T> nám poskytuje velice jednoduchou integraci vlastního typu detekce písmenových kandidátu a non-max suppresion popsané v 4.4.2. Navíc výstupem metody segment je vector instancí třídy Letter, se kterou pracuje třída WordGenerator při generování slov.

V naší knihovně je implementován algoritmus 4.3.2 popsaný v předchozí kapitole na extrakci písmenových kandidátů, tento algoritmus je zapouzdřený ve třídě ERTextDetection.

Třída Segment<T> byla v naší knihovně parametrizována třídou ERTextDetection. A tedy pro ERTextDetection byla specializována politika SegmentPolicy<T>. V následující sekci se budeme zabývat primárně implementací ERTextDetection.

9.4 ERTextDetection

Třída ERTextDetection zapouzdřuje algoritmus 4.3.2 pro detekci písmenových kandidátů. V popisu algoritmu je uvedeno, že algoritmus běží ve dvou fázích, postavení komponentového stromu bitmapy a výběru vhodných uzlů(komponent) z konstruktovaného stromu. Stavba komponentového stromu je společný krok pro více algoritmů typu $MSER$ [14], proto jsme se rozhodli oddělit kódu pro stavbu a extrakci. Stavbu komponentového stromu řeší generická třída ComponentTreeBuilder<T> a extrakci, kterou bude řešit třída ERTree.

9.4.1 ComponentTreeBuilder<T>

ComponentTreeBuilder<T>, která je parametrizovaná typem T. Tato generická třída implementuje pouze algorimus *Lineární MSER* [19] popsaný v předešlé části 4.3.1, aníž by věděl cokoliv o interní struktuře stromu nebo i typu uzlu konstruovaného stromu. O interní strukturu stromu zodpovídá třída T samotná a na typ uzlu parametrizujeme pomocí politik. Poznamenejme, že interní struktůrou stromu rozumíme implementací dané datové struktury a manipulaci s ní.

Politika pro ComponentTreeBuilder<T> je třída ComponentTreePolicy<T>, ta tedy obsahuje typedef pro typ uzlu stromu, dále inicializaci algoritmu a statickou funkci getLevel, která vrací při kterém grayscalu byl uzel přidán na zásobník viz. algoritmus. Kromě volání statických metod politik a využívání jejích typedefů ComponentTreeBuilder<T> také volá metody instance třídy T, na níž má pointer. Jedná se o metody merge, accumulate a getDomain, kde první připojí synovskou komponentu ke komponentě otce a ta druhá updatuje uzel ve stromě a poslední vrací bitmapu, nad kterou se bude stavět komponentový strom. Třída T si tedy sama implementuje jednotlivé kroky algoritmu.

Výsledný návrh je kombinace policy class a návrhového vzoru Builder.

ComponentTreeNode V knihovně NOCRLib poskytujeme generickou třídu ComponentTreeNode<T>, která může být použita jako uzel componentového stromu. Parametr T je typ třídy, která bude reprezovantovat detekovanou componentu ve stromě. K této třídě se z ComponentTreeNode<T> dostaneme voláním metody getVal.

Další detaily jsou uvedeny ve zdrojovém kódu v souboru NOCRLib/component_tree_node.h Tato třída se stará o všechny operace spojené se strukturou stromu.

9.4.2 ERTree

ERTree je třída zapouzdřující postavený komponentový strom, extrahuje komponenty (uzly stromu), které jsou nejpravděpodboněji písmena pomocí metody getLetters.

V naší knihovně ji rovněž využíváme jako třídu při stavbě komponentového stromu. Je tedy používána jako parametr T generické třídy ComponentTreeBuilder<T>. Nejdříve ukážme její integraci s ComponentTreeBuilder<T>.

Listing 9.3: integrace ERTree s ComponenentTreeBuilder<T>

```
1  class ERTextDetection
2  {
3      public:
4          //kod
5
6          // metoda pro extrahovani kodu
7          std::vector< Storage > getLetters( const cv::Mat &image) ;
8
9      private:
10         // trida zastavajici roli extraktora
11         ERTree er_tree_;
12         //kod
13 };
14
15 auto ERTextDetection::getLetters( const cv::Mat &image )
16     -> vector<Storage>
17 {
18     // nastavime obrazek extractorovi
19     extremal_region_.setImage( image );
20
21     // inicializujeme ComponentTreeBuilder a predame mu pointer na
22     // extractora
23     ComponentTreeBuilder<ERTree> builder( &extremal_region_ );
24
25     // z vyuzitim extractora postavime strom
26     ComponentTreeNode<Region> *root = builder.buildTree();
27
28     // vyextrahujeme komponenty
29     auto letters_storages = extremal_region_.getLetters(root);
30
31     // flipneme vstup aby detekoval svetla pismena na tmavem pozadi
32     // viz algoritmus
33     extremal_region_.invertDomain();
34
35     // opakuje stavbu stromu a extrakci
36     root = builder.buildTree();
37     auto tmp = extremal_region_.getLetters(root);
38
39     letters_storages.reserve( letters_storages.size() + tmp.size() );
40     letters_storages.insert( letters_storages.end(), tmp.begin(),
           tmp.end() );
```

```
41
42      return letters_storages;
43  }
```

9.4.3 Stavba komponentového stromu

Jak již bylo napsáno ERTree se podílí na stavbě komponentového stromu jako parametr T generické třídy ComponentTreeBuilder<T>. Tedy musí implementovat metody accumulate, merge a getDomain. Jako typ uzlu komponentového stromu je v politice ComponentTreePolicy<ERTree> definován typ ComponentTreeNode<ERRegion>.

Při stavbě komponentového stromu třída se vypočítá pravděpodobnost $P(r \mid charakter)$ dle 4.3.2 pro každý uzlu r v komponentovém stromě.

Pro implementační detaily doporučujeme prohlédnout si zdrojový kód v souboru NOCRLib/extremal_region.h.

ERRegion Třída, která nese informace o komponentě v komponentovém stromě, hodnoty její incrementálně počítaných descriptorů a barevného složení. Pixely patřící komponentě jsou uloženy ve formě obousměrných spojových seznamů. Region dále udržuje informaci o lokálních extrémech synovského okolí od daného uzlu, ta se počítá již při stavbě.

9.4.4 Extrakce komponent z komponetového stromu

Pro nalezení lokálních extrému $P(r \mid charakter)$, stačí traversovat strom do hloubky a zkoumat jednotlivé uzly jestli splňují požadavky uvedené v algoritmu 4.3.2. Druhá fáze algoritmu probíha standartně, vypočítáme descriptory a ty předložíme klasifikátoru.

9.4.5 Implementace MSERU

Ukažme si jak využít třídu ComponentTreeBuilder<T> využít pro implementaci klasického MSERU 4.3.1. Řekněme že o interní strukturu stromu se bude starat třída MSER. Pro tuto třídu bude potřeba specializovat politiku ComponentTreePolicy<T>. Definujme také třídu Region ta v sobě bude držet informace o komponentě, jenž reprezentuje uzel v komponentovém stromě, a hodnotu grayscale, při které byla dána na zásobník.

Nejdříve ukážme jakým způsobem by byla implementována politika ComponentTreePolicy<MSER>.

Listing 9.4: ukazka specializace ComponentTreePolicy

```
1  template <> class ComponentTreePolicy<MSER>
2  {
3      public:
4          // typedef pro typ uzlu ve strome
5          typedef ComponentTreeNode<Region> NodeType;
6
7          // inicializace algoritmu, nastaveni accesible masky,
```

```
8         // inizializace zasobniku pro beh algoritmu
9         // zvoleni bodu z ktereho budeme rozlevat
10        static void init( const cv::Mat &bitmap, std::vector<bool>
              &accesible_mask,
11                std::stack<NodeType*> &stack, int * init_pixel_code )
12        {
13            // kod
14        }
15
16        // vytvoreni noveho uzlu ve strome.
17        static NodeType* createNode( int level, const cv::Point &p )
18        {
19            Region r( level, p );
20            return new NodeType(r);
21        }
22
23        // vraceni levelu daneho uzlu ve strome
24        static int getLevel( NodeType* node )
25        {
26            return node->getVal().getLevel();
27        }
28    };
```

Jako typ uzlu ve stromě je definován ComponentTreeNode<Region>. Pro zjištění informací o komponentě, tedy přístupu k instanci Region stačí volat metodu getVal() viz metoda int getLevel v ComponentTreePolicy<MSER>. Teď si ukažme deklaraci třídy MSER.

Listing 9.5: MSER

```
1     class MSER
2     {
3         public:
4             //kod
5
6             // vrati bitmapu nad kterou stavime komponentovy strom
7             cv::Mat getDomain();
8
9             // prida uzlu reg novy pixel code
10            void accumulate( NodeType *reg, int code );
11
12            // pripoji uzel child jako syna k uzlu parent
13            void merge( NodeType *child, NodeType *parent );
14        private:
15            // kod
16    };
```

Nasledným naimplementováním metod accumulate, merge a getDomain() jsme zintegrovali třídu MSER a ComponentTreeBuilder<MSER>. Poté lze postavit komponentový strom, z kterého budeme extrahovat příslušné komponenty dle 4.3.1 pomocí příslušné třídy.

9.5 WordGenerator

Třída WordGenerator implementuje algoritmus popsaný v [22], se zlepšeními, jenž byli popsány v předchozí sekci. V konstruktor třídy očekává na vstupu vektor instancí třídy Letter. Konstruktor pro každého kandidáta, najde seznam všech kandidátů, kteří s ním mohou být ve slově viz algoritmus.

Metoda process očekává za argument instanci třídy Dictionary, což je naše implementace slovníkové trie. Process následně tuto trii projde a generuje slova dle 13. Výstupeme této metody je třída TranslatedWord.

9.5.1 Dictionary

Jedná se o třídu, která zapouzdřuje práci s slovníkovou trií. Dictionary se může inicializovat za pomocí textového souboru obsahujího slovník. Tento soubor má na každé řádce právě jedno slovo. Dále je možné přidávat slova do již zinicializované třídy. Slovník pracuje s trií, jejíž uzel má typ TrieNode.

9.5.2 TranslatedWord

TranslatedWord je třída, která obsahuje geometrickou i lexikografickou informaci o detekovaném slově na vstupu. Lexikografická informace obsahuje slovo ze slovníku, které bylo zvoleno jako optimální v metodě process třídy WordGenerator. Geometrická informace poskytuje pozici slova, jeho výšku a šířku a seznam jeho písmen.

Následující sekce se již zabývají třídami, které jsou nepřímo využity při implementacích uvedených algoritmů, jedná se hlavně o descriptory a klasifikátory.

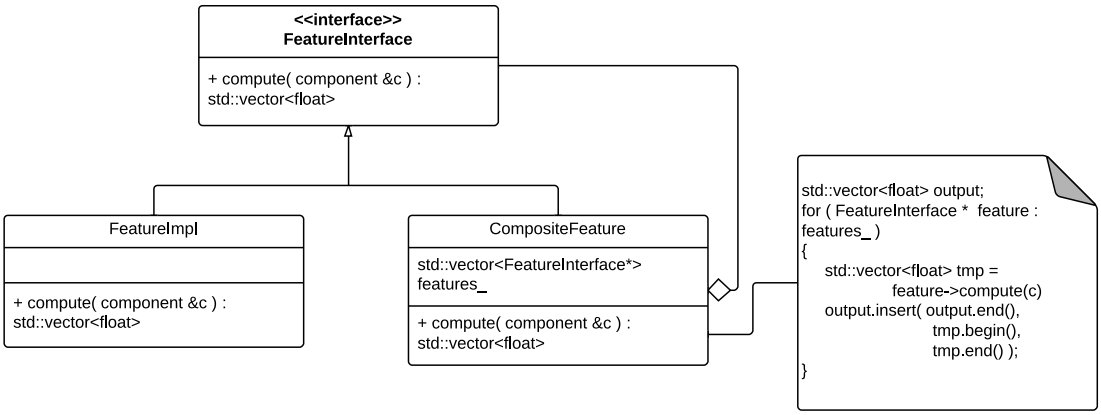
9.6 Příznaky

V algoritmu uvedeném v 4.3.2 jsme v 2.fázi jsme počítali descriptor, složený z několika příznaků na základě vyextahovaných komponent. To samé děláme v OCR, kde počítáme HoG descriptor a ve třídě VisualInfo, která nám dává informaci o SWT a barevném složení komponenty a jejího okolí také vznikla vypočítáním nějakých příznaků. Ve všech případech počítáme příznaky na základě předložení komponenty.

9.6.1 Composite

Příznaky jsou počítané pomocí nějakých tříd. Pro snadné experimentování a rozšiřitelnost by bylo dobré umět tyto přiznaky libovolným způsobem kombinovat a lehce integrovat nové příznaky. Proto jsme rozhodli aplikovat návrhový vzor Composite.

Jak vidíme na uml diagramu 9.2, máme zde interface FeatureInterface pro všechny příznaky, v diagramu je také třída CompositeFeature, která umožňuje kombinovat libovolné implementace rozhraní FeatureInterface a zároveň rovněž dědí od FeatureInterface.



Obrázek 9.2: Composit aplikován na příznaký

9.6.2 Factory Method

Jednou z nevýhod compositu je že při používání stejně složeného compositu na více místech ve zdrojovém kódu, musíme vždy compositu inicializovat přidáváním příslušných implementace FeatureInterface. Tím zbytečně vzniká duplicitní kód inicializace. Abychom tomuto zamezili, rozhodli jsme se využít Factory Method. Tento návrhový vzor zapouzdřuje konstrukci compositu na jedno místě v kódu. Poznamenejme že všechny Factory dědí od rozhraní FeatureFactoryInterface.

9.7 Trenování klasifikátorů

Použité algoritmy často používají klasifikační algoritmy s učitelem. Proto naše knihovna také řeší trenování klasifikátorů, V NOCRLib se kromě trénování samotného, řeší také extrakce descriptoru z trénovacích dat(sada obrázků).

Zabýváme se pouze extrakcí descriptorů z komponent a pracujeme jenom s klasifikátory ze třídy supervised learning(učení s učitelem). Trénování jsme rozdělili na 2 fáze:

1. extrakce descriptorů z trénovacích dat
2. trénování samotné

Tyto 2 fáze probíhají samostatně. V té první se obvykle vyextrahují descriptory za pomocí příznaků a ty se zapíší do nějakého souboru, dle námi určeného formátu. Tento formát je společný pro jakkýkoliv klasifikátor, fungující v NOCRLib. Následně v druhé fázi si příslušný klasifikátor načte trénovací data ze souboru a spustí svůj trenovací algoritmus.

O extrakci se stará třída TrainDataCreater< feature F, extraction E >, kde F je enum feature který reprezentuje daný Composite nebo Descriptor pomocí traits a E je enum extraction, který specifikuje jakým způsobem je ze vstupního obrázku detekována komponenta pomocí politiky, tj. vrací vektor komponent extrahovaných ze vstupu vstupu. TrainDataCreator vytváří trénovací data pomocí metody loadAndProcessSamples, tato metoda má za parametry konfigurační souboru pro trénování a výstupní soubor.

Konfigurační soubor obsahuje udáje o jednolivých obrázcích testovací sady a kterému třídě patří. Každá řádka má následující formát <cesta k trénovacímu

obrázku>:<label trénovací třídy>. Takže jestliže má trénovací sada obsahovat obrázek ve složce *train* a názvem *a.png* patřící třídě s labelem 1. Potom konfigurační soubor obsahuje řádku *./train/a.png:1*. Výstup obsahuje řádek ve formátu <deskriptor>:<label třídy>, kde deskriptor je posloupnost čísel oddělených dvojtečkou pro každý řádek z konfiguračního souboru.

9.7.1 FeatureTraits

Jedná se o generický struct parametrizovaný enumem feature. Obsahuje délku descriptoru a typ Factory pro vytvoření příslušného Compositu nebo implementace FeatureInterface. Uveďme ukázku kódu pro nespecializovaný struct.

Listing 9.6: FeatureTraits<T>

```
1    // NOCRLib/feature_traits.h
2    template < feature F >
3    struct FeatureTraits
4    {
5        // delka descriptoru
6        static const int feature_length = 0;
7        // typ Factory
8        typedef FeatureFactoryInterface FactoryType;
9    };
```

Pro použití je potřeba vždy specializovat tuto traits pomocí specifického člena enumu feature.

9.7.2 TrainExtractionPolicy

Jedná se o policy class pro TrainDataCreator, která je parametrizovaná enumem Extraction. Pro použití je potřeba vždy specializovat třídu TrainExtractionPolicy podle parametru. Policy class obsahuje jenom statickou metodu extract, která vrací vektor instancí třídy Component.

9.7.3 Rozšiřitelnost

Nový příznak Řekněme, že potřebujeme použít externí třídu počítající příznak a kterou chceme zkombinovat s existujícími příznaky v naší knihovně. Následně máme trénovací sadu z kterých chceme extrahovat příznaky.

Jako první pomocí adapteru přizpůsobíme rozhraní externí třídy na rozhraní FeatureInterface. Následně je potřeba přidat nový člen do enumu Feature, nazvěmeho ukázka. Pak je potřeba naimplementovat novou Factory, která nám bude vytvářet požadovaný composite. Teď už stačí jenom specializovat FeatureTraits pro ukázku. A následně inicializovat instaci třídy TrainDataCreater<feature::ukazka,E> a zavolat metodu loadAndProcessSamples, kde E je nějaký typ extrakce.

Nový způsob extrakce Pro nový způsob extrakce stačí přidat nového člena do enum extraction a specializovat TrainExtractionPolicy pro nově přidaný člen do enumu extraction.

9.7.4 Trenování klasifikátoru

Pro spuštění trénovacího algoritmu, stačí předat soubor s deskriptorama, ve formátu používaném třídou TrainDataCreator. Tyto data si každý klasifikátor v NOCRLib, načte a spustí na nich daný učící algoritmus.

9.8 Klasifikátor

V NOCRLib jsme používali několik klasifikátorů. Většinou se jedná o přizpůsobení již existujících klasifikátorů, tak aby pracovali s našim rozhraním. Jedinou výjimku tvoří naimplementování SVM s Fast Intersection Kernel od citace, ručně jsme naimplementovali velkou část tohoto algoritmu pro klasifikaci.

9.8.1 Wrapovaní klasifikátoři

Jedná se o generické třídy, které jsou parametrizované enumem feature, díky znalosti enumu lze pomocí FeatureTraits dostat příslušnou factory pro daný composite nebo feature.

Třídy umí spouštet svůj trénovací algoritmus, uložit a načíst konfigurační soubor klasifikátoru a samozřejmě klasifikovat.

NOCRLib obsahuje následující owrapované klasifikátory z OpenCV desion tree, svm, boosting s desion tree a knn. A dále knihovna obsahuje adapter nad svm z LibSVM [1] a LibLinear [6], tyto svmka narozdíl od svm implementované v OpenCV umí vypočítat i pravděpodobnostní výstupy.

9.8.2 SVM Fast Intersection Kernel

Jedná se o implementaci [13] algoritmu pro rychlejší klasifikaci SVM s intersection kernelem. Pro trénování stačí použít wrap nad svm z libsvm z intersection kernelem. Následně tuto třídu, předložíme třídě IKSVMConverter, ta zkonvertuje svm s intersection kernelem na instanci třídy IKSVM. IKSVM jvyužívá algoritmus navržený v [13] pro rychlejší klasifikaci.

IKSVM umí uložit i načíst svůj vlastní konfigurační soubor.

9.9 Shrnutí

V této kapitole jsme popsali základní koncepty použité v knihovně NOCRLib. Zabývali jsme se především popsáním integrace různých modulů a pohledem z vyšší urovně. Pro seznámení se s jednotlovými rozhraním a jejich implementací doporučujeme prohlídnout soubory se zdrojovým kódem a referenční příručku.

Literatura

[1] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[2] Huizhong Chen, Sam S. Tsai, Georg Schroth, David M. Chen, Radek Grzeszczuk, and Bernd Girod. Robust text detection in natural images with edge-enhanced maximally stable extremal regions. In *2011 IEEE International Conference on Image Processing*, Brussels, September 2011.

[3] Gabriella Csurka, Christopher R. Dance, Lixin Fan, Jutta Willamowski, and Cédric Bray. Visual categorization with bags of keypoints. In *In Workshop on Statistical Learning in Computer Vision, ECCV*, pages 1–22, 2004.

[4] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *In CVPR*, pages 886–893, 2005.

[5] Boris Epshtein, Eyal Ofek, and Yonatan Wexler. Detecting text in natural scenes with stroke width transform, 2010.

[6] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[7] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*, chapter 17. Prentice Hall Professional Technical Reference, 2002.

[8] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28:2000, 1998.

[9] Lluis Gomez and Dimosthenis Karatzas. Multi-script text extraction from natural scenes. In *12th International Conference on Document Analysis and Recognition*, pages 467–471, 2013.

[10] A. Gonzalez, L.M. Bergasa, J.J. Yebes, and S. Bronte. A character recognition method in natural scene images. In *Pattern Recognition (ICPR), 2012 21st International Conference on*, pages 621–624, Nov 2012.

[11] S. B. Gray. Local properties of binary images in two dimensions. *IEEE Trans. Comput.*, 20(5):551–561, May 1971.

[12] Dimosthenis Karatzas, Faisal Shafait, Seiichi Uchida, Masakazu Iwamura, Lluis Gomez i. Bigorda, Sergi Robles Mestre, Joan Mas, David Fernandez Mota, Jon Almazàn Almazàn, and Lluís Pere de las Heras. Icdar 2013 robust reading competition. In *Proceedings of the 2013 12th International Conference on Document Analysis and Recognition*, ICDAR '13, pages 1484–1493, Washington, DC, USA, 2013. IEEE Computer Society.

[13] Subhransu Maji, Alexander C. Berg, and Jitendra Malik. Classification using intersection kernel support vector machines is efficient. In *CVPR*, 2008.

[14] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust wide baseline stereo from maximally stable extremal regions. In *Proc. BMVC*, pages 36.1–36.10, 2002. doi:10.5244/C.16.36.

[15] L. Najman and M. Couprie. Building the component tree in quasi-linear time. *Trans. Img. Proc.*, 15(11):3531–3539, November 2006.

[16] Lukas Neumann and Jiri Matas. Text localization in real-world images using efficiently pruned exhaustive search. In *ICDAR*, pages 687–691, 2011.

[17] Lukas Neumann and Jiri Matas. Real-time scene text localization and recognition. In *CVPR*, pages 3538–3545, 2012.

[18] Alexandru Niculescu-mizil and Rich Caruana. Obtaining calibrated probabilities from boosting. In *In: Proc. 21st Conference on Uncertainty in Artificial Intelligence (UAI '05), AUAI Press.* AUAI Press, 2005.

[19] David Nistér and Henrik Stewénius. Linear time maximally stable extremal regions. In *Proceedings of the 10th European Conference on Computer Vision: Part II*, ECCV '08, pages 183–196, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] Tatiana Novikova, Olga Barinova, Pushmeet Kohli, and Victor Lempitsky. Large-lexicon attribute-consistent text recognition in natural images. In *Proceedings of the 12th European Conference on Computer Vision - Volume Part VI*, ECCV'12, pages 752–765, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] N. Otsu. A threshold selection method from gray level histograms. *IEEE Trans. Systems, Man and Cybernetics*, 9:62–66, March 1979. minimize inter class variance.

[22] Trung Quy Phan, Palaiahnakote Shivakumara, Shangxuan Tian, and Chew Lim Tan. Recognizing text with perspective distortion in natural scenes. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2013.

[23] Shangxuan Tian, Shijian Lu, Bolan Su, and Chew Lim Tan. Scene text recognition using co-occurrence of histogram of oriented gradients.

[24] Luc Vincent and Pierre Soille. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(6):583–598, June 1991.

[25] Paul Viola and Michael Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.

[26] Kai Wang, Boris Babenko, and Serge Belongie. End-to-end scene text recognition. In *ICCV*, pages 1457–1464, 2011.

[27] Kai Wang and Serge Belongie. Word spotting in the wild, 2010.