VYHĽADÁVANIE PODOBNÝCH OBRÁZKOV

DIPLOMOVÁ PRÁCA

Rastislav Vaško

UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

KATEDRA INFORMATIKY

Študijný odbor: Informatika 9.2.1

Vedúci diplomovej práce

RNDr. Elena Šikudová, PhD.

BRATISLAVA 2010

Čestne prehlasujem, že som túto diplomovú prácu
vypracoval samostatne pod vedením svojej diplomovej
vedúcej s použitím citovaných zdrojov
a vedomostí nadobudnutých počas štúdia.

. . . . . .

Touto cestou sa chcem poďakovať vedúcej mojej diplomovej práce RNDr. Elene Šikudovej, PhD. za ochotu aj odborné rady, ktoré mi poskytovala počas vytvárania práce.

Moja vďaka patrí taktiež kolegovi Matúšovi Petruľákovi, ktorý mi pomohol s počítačovým vybavením na testovanie.

Abstrakt

V práci navrhujeme vylepšenia pre existujúcu CBIR schému a spôsob jej zovšeobecnenia. Schéma využíva indexovanie náhodných lokálnych príznakov do náhodných stromov. Predstavujeme nové metódy generovania týchto náhodných stromov a popis ich vlastností. Ďalej navrhujeme spôsob generalizácie vrchola stromu tak, aby bolo možné využiť viaceré príznaky obrázkov a nielen odtlačky, ako tomu je pri pôvodnej schéme. Zavádzame pravdepodobnostnú distribúciu vrcholov ako jemnozrnný nástroj na kombinovanie rôznych typov vrcholov v rámci stromov, pričom sa pridŕžame randomizovanej nátury schémy. Ďalej predstavujeme niekoľko nových typov vrcholov, využívajúcich ďalšie používané príznaky, ktorými sú hlavne histogramy. Implementujeme túto novú schému spolu s webovým rozhraním na pohodlné vyhľadávanie. Podrobujeme testovaniu nové metódy a nové typy vrcholov.

Naša schéma dosahuje oproti pôvodnej zlepšenie o viac než 9%, pri porovnateľnej zložitosti. Týmito výsledkami sa dostáva v rámci testovanej sady na úroveň jednej z najlepších CBIR schém súčasnosti.

Kľúčové slová: cbir, image retrieval, random subwindows, randomized trees, image similarity, histogram, color histogram, query by image

Predhovor

Problematika vyhľadávania podobných obrázkov je študovanou už niekoľko dekád. Avšak až pokrok výpočtovej techniky a rozvoj Internetu v ostatných rokoch prispeli veľkou mierou k zvýšenému záujmu o tento problém.

Vyhľadávanie podobných obrázkov, označované CBIR, v sebe spája viaceré oblasti informatiky, ktorými sú počítačová grafika, vyhľadávanie informácii, strojové učenie a tak ďalej. Praktické využitie predurčuje CBIR k ďalšiemu rozvoju.

V práci staviame na existujúcej schéme na riešenie CBIR, ktorú ďalej rozvíjame a vylepšujeme. Nové nápady implementujeme a testovaním zisťujeme ich vplyv.

Obsah

Zoznam obrázkov

Zoznam použitých skratiek

CBIR Content-based Image Retrieval

IR Image Retrieval

HSV Hue Saturation Value

MAP Mean Average Precision

ML Machine Learning

OOP Objektovo-orientované Programovanie

RF Relevance Feedback

RGB Red Green Blue

STL Standard Template Library

Kapitola 1

Úvod

Pokiaľ by sme uskutočnili prieskum s otázkou, ktorý technologický výdobytok za poslednú dekádu najvýraznejšie poznačil náš život, som presvedčený, že na poprednom mieste by sa umiestnil Internet. Medzi najznámejšie a najpoužívanejšie služby na Internete patria vyhľadávače. Tieto nám umožňujú rýchlo a pohodlne nájsť stránky, ktorých obsah by mal najlepšie zodpovedať nami zadaným kľúčovým slovám. Veľmi prirodzeným rozšírením takýchto vyhľadávačov je hľadanie obrázkov. V súčasnosti však tradičný spôsob hľadania obrázkov začína zvolením kľúčových slov, obdobne ako pri hľadaní stránok. Čo ak chceme hľadať obrázky na základe iného obrázka? Práve táto otázka bola inšpiráciou pre túto diplomovú prácu. Jej cieľom je vytvoriť webovú službu, ktorá pre zvolený vstupný obrázok vyberie z databázy obrázkov množinu tých, ktoré sú mu najpodobnejšie.

Základom práce je publikácia [1] z roku 2007, ktorá opisuje schému pre CBIR a prezentuje experimentálne výsledky. Dôvod prečo sme si vybrali túto prácu ako základ je ten, že táto schéma je v mnohých ohľadoch zaujímavá aj praktická. Dali som si za cieľ implementovať ju, navrhnúť spôsoby jej vylepšenia, otestovať tieto na skutočných dátach a analyzovať výsledky a vytvoriť webové rozhranie na používanie.

Práca stojí na prieniku niekoľkých oblastí informatiky. Využíva metódy z počítačového videnia, vyhľadávania obrázkov, algoritmov, dátových štruktúr, programovania či webových štandardov. V nasledujúcej kapitole je v nutnom rozsahu popísaná problematika a pojmy, o ktoré sa práca ďalej opiera.

Tretia kapitola pojednáva o schéme z publikácie [1], na ktorej práca stavia. Je popísaný princíp fungovania, jej výhody a nevýhody a dosiahnuté experimentálne výsledky. Táto kapitola vychádza zo spomínanej publikácie a neobsahuje žiaden autorov prínos. Je však nutná z pohľadu ucelenosti práce, keďže základný princíp fungovania pôvodnej schémy je zachovaný aj v novej.

Návrhy vylepšení pôvodnej schémy sú prezentované vo štvrtej kapitole. Každý obsahuje teoretický popis, motiváciu, ktorá za ním stojí a očakávaný prínos.

Piata kapitola popisuje spôsob testovania schémy a obsahuje rôzne experimentálne výsledky. Venuje sa vplyvom rôznych parametrov na úspešnosť vyhľadávania. Taktiež sa

merajú výsledky návrhov zo štvrtej kapitoly v kontraste s očakávaným prínosom.

Posledná šiesta kapitola uzatvára prácu a prezentuje stručný záver s ohľadom na dosiahnuté výsledky. Taktiež uvádza možnosti ďalšieho rozvoja práce či neformálny pohľad do budúcnosti CBIR.

Kapitola 2

Problematika

V úvode tejto kapitoly si popíšeme základné spôsoby a metódy pre vyhľadávanie podobných obrázkov. Uvedieme si najbežnejšie spôsoby využitia, aktuálne výsledky a dostupné projekty, ale aj klasické problémy. Ďalej si zadefinujeme pojmy a postupy nutné pre ďalší text.

2.1 Vyhľadávanie obrázkov

Vyhľadávanie obrázkov (*Image Retrieval*) je problém prehľadávania databázy obrázkov. Metódy riešenia tohoto problému sa rozdeľujú na dve skupiny:

Vyhľadávanie podľa textu Vstupom od užívateľa je textový popis obrázka, ktorý hľadá. Databáza obsahuje pri každom obrázku aj meta informácie o ňom, podľa ktorých následne vyhľadáva.

Vyhľadávanie podľa obrázka Vstupom od užívateľa je obrázok podobný tomu, ktorý hľadá. Môže to byť náčrt, fotografia, iný obrázok obsahujúci hľadaný element, atď. Systém následne porovnáva priamo podľa vizuálnych prvkov obrázka.

Druhá metóda sa nazýva CBIR (z ang. *Content-Based Image Retrieval*, čo v preklade znamená *hľadanie obrázkov na základe obsahu*) a je to aplikácia počítačového videnia na vyhľadávanie obrázkov. Táto práca sa venuje výhradne tomuto problému.

2.1.1 Sémantická medzera

Pojem *sémantická medzera* (z ang. *semantic gap*) sa v informatike vyskytuje často. Všeobecne povedané, popisuje rozdiel medzi nejasnou formuláciou poznatku v prirodzenom jazyku a jeho výpočtovou reprezentáciou vo formálnom jazyku. Pri analýze a vyhľadávaní obrázkov je tento problém veľmi výrazný, pretože chceme dosiahnuť vysokú úroveň abstrakcie za pomoci nízkoúrovňových metód. Inými slovami, chceme porozumieť obrázku a k dispozícii máme len jeho farebné body.

2.2 Popis a využitie CBIR

Ako bolo spomenuté, problém hľadania podobných obrázkov pre zvolený vstupný obrázok sa nazýva *CBIR*. V súčasnosti je dostupných niekoľko komerčných aj vedeckých projektov zameraných na CBIR, pričom časť z nich ponúka aj rozhranie cez web. Patria medzi ne:

- ALIPR[1] - `http://www.alipr.com/`
- Bing Image Search[2] - `http://images.bing.com/`
- FIRE[1] - `http://thomas.deselaers.de/fire/`
- Gazopa - `http://www.gazopa.com/`
- Google Image Search[2,3] - `http://images.google.com/`
- Incogna[2] - `http://www.incogna.com/`
- RETIN[1,2] - `http://retin.ensea.fr/`
- SIMBA[1] - `http://simba.informatik.uni-freiburg.de/`
- TinEye - `http://www.tineye.com/`
- Tiltomo[2] - `http://www.tiltomo.com/`

Tomuto problému sa akadémia venuje už niekoľko dekád, avšak k najväčšiemu rozvoju došlo v posledných desiatich rokoch. Len nedávno predstavili svoje CBIR vyhľadávače spoločnosti Microsoft[4] a Google[5]. Hlavným dôvodom rozvoja bola jednoduchšia dostupnosť veľkého množstva rôznorodých obrázkov, výrazné zníženie nákladov spojených s uložením dát a výpočtovej techniky celkovo a rast dopytu po tejto službe.

Dopyt tkvie v rôznorodom využití CBIR v praxi. Patrí sem:

- Vyhľadanie podobných obrázkov za účelom rozšírenia vlastnej zásoby obrázkov. Napríklad pre účely spestrenia prezentácií, dokumentov či webových stránok.
- Ako spôsob hľadania použitia vlastných grafických diel v iných prípadoch. S vážnejšími dôsledkami ako spôsob vyhľadávania prípadov zneužitia duševného vlastníctva a neoprávneného použitia diela.

[1] Vedecký projekt.
[2] Nie je možné vyhľadávanie externých obrázkov.
[3] Len časť databázy je indexovaná na vyhľadávanie podľa podobnosti.
[4] December 2008 - http://www.bing.com/community/blogs/search/archive/2008/12/01/using-images-to-find-other-images.aspx
[5] Apríl 2009 - http://googleblog.blogspot.com/2009/04/hard-at-play-in-google-labs-with.html

- Možný spôsob klasifikácie resp. automatickej anotácie obrázkov. Keď veľa podobných obrázkov spadá do jednej známej kategórie, napovedá to o možnom obsahu nového obrázka.
- Množstvo špecifických odvetví. Patrí sem oblasť kriminalistiky, v ktorej sa porovnávajú odtlačky a tváre, medicíny a hľadanie neobvyklých prípadov, alebo biológie pri určovaní druhov.

2.2.1 Základná schéma

Obr. 2.1: Základná schéma CBIR vyhľadávača.

Pokiaľ abstrahujeme od detailov jednotlivých schém pre CBIR, dostaneme postup niekoľkých krokov, ktorý veľmi dobre vystihuje podstatu problému. Sú nasledovné:

1. Predspracujeme trénovaciu databázu obrázkov na databázu príznakov.
2. Predspracujeme vstupný obrázok od užívateľa.
3. Prehľadávaním vyberieme kandidátov na podobné obrázky.
4. Zvolíme najpodobnejšie obrázky spomedzi kandidátov.

Vo všebecnosti môže byť prvý krok výpočtovo pomerne náročný, pretože zvyčajne predpokladáme, že naň máme dostatok času aj výpočtovej sily. Prirodzene, praktické algoritmy si nemôžu ani na predspracovanie databázy dovoliť nerealistickú zložitosť. Bližší popis schémy, o ktorú sa v práci opierame z pohľadu tohoto postupu, je v sekcii 3.1.

2.2.2 Súčasné výsledky

Bolo už spomenuté, že CBIR ako problém stojí na pomedzí niekoľkých informatických odvetví. V posledných rokoch ešte odvetví vplývajúcich na riešenie CBIR pribúda. Výborný a komplexný prehľad o problematike podáva [2]. Nebudeme preto bližšie popisovať jednotlivé techniky ani práce, ale len v skratke zhrnieme najdôležitejšie smery.

Množstvo iných zaujímavých problémov príbuzných CBIR existuje a vzniká. Patria medzi ne automatická anotácia obrázkov, detekcia duplikátov, obrázková CAPTCHA na rozpoznanie ľudí a počítačov, klasifikácia obrázkov a tak ďalej. Prirodzene, práve postupy pri riešení týchto problémov sa veľmi často využívajú pri riešení CBIR a naopak.

Medzi najvyužívanejšie sféry pre riešenie CBIR už od začiatkov patrí Strojové učenie (*Machine learning*). Aplikujú sa postupy pre zhlukovanie a klasifikáciu. Patria sem algoritmy ako K-Means využívané v [3] či D2-zhlukovanie, ďalej skryté Markovovské modely (*Hidden Markov models*) napr. v [4], Bayesovské klasifikátory, podporné vektory (*Support vector machines*) využívané v [5], analýza hlavných komponentov (*Principal component analysis*) a iné. Skúmajú sa rôzne metriky na meranie podobnosti. Využívajú sa rôzne druhy príznakov ako farby, tvary, textúry, histogramy, významné body a oblasti. Medzi najpokročilejšie algoritmy na extrakciu príznakov patria algoritmy ako SIFT, SURF či MSER, detekujúce invariantné príznaky. Tieto využíva [6]. Taktiež sa používajú príznaky definované v MPEG-7 štandarde, napríklad v [7] a [8].

Hľadajú sa pravdepodobnostné a aproximačné algoritmy, ktoré by pomohli pri škálovaní problému prehľadávania mnohorozmerných priestorov a veľkých databáz obrázkov.

V ostatnom čase sa intenzívne vyvíja v súvislosti s CBIR relevancia spätnej väzby (*Relevance feedback*, RF) z oblasti IR. Prehľad je v [9]. Veľa smerov RF sa skúma, medzi inými odozva od užívateľa, pravdepodobnostná spätná väzba, implicitná spätná väzba, atď. Je to jeden zo spôsobov ako čiastočne prekonať spomínanú sémantickú medzeru, pretože interakciou od užívateľa môžeme získať doplňujúce informácie o jeho úmysloch.

2.2.3 Problémy

Základným problémom vyhľadávania obrázkov je už spomenutá sémantická medzera. Tento problém zjavne postihuje aj CBIR, pretože užívateľ očakáva *sémanticky* podobné obrázky vstupnému obrázku, nie len *vizuálne* podobné, ktoré dostaneme ak porovnávame obrázky jednoducho podľa pixelov.

Ďalším výrazným problémom je subjektívnosť hodnotenia podobnosti. Tento súvisí s predchádzajúcim problémom, ale rozdiel je v tom, že aj keby sme obrázku celkom porozumeli, nemusíme byť schopní správne interpretovať dôvody užívateľa pre voľbu vstupného obrázka. Napríklad, nech je vstupom obrázok červeného Ferrari pri západe slnka. Aj keby sme boli schopní scéne porozumieť, nie je zjavné, či užívateľ očakáva obrázky červených Ferrari, červených áut pri západe slnka, Ferrari pri západe slnka alebo ešte niečoho

odlišného.

Prirodzene, problematické je samotné hľadanie *vizuálne* podobných obrázkov aj bez toho, aby sme sa zaoberali porozumením obrázku či úmyslami užívateľa. Tento problém je bližšie popísaný v sekcii 2.4.

Medzi cieľmi práce bolo aj vytvorenie webového rozhrania pre CBIR vyhľadávač. Táto požiadavka kladie ďalšie nároky na rýchlosť vyhľadávania, pretože štúdie ukazujú, že väčšina užívateľov nie je ochotná čakať viac ako niekoľko sekúnd na zobrazenie stránky[6].

2.3 Definície

Pre ďalší text potrebujeme definovať niekoľko pojmov:

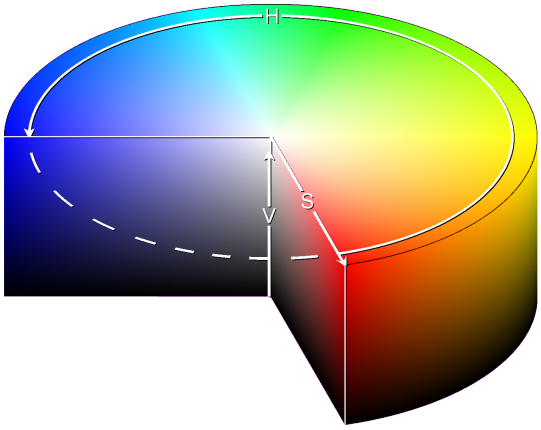
Farebný bod je usporiadaná n-tica čísel z vopred stanovenej množiny. Označenie farebný bod budeme používať striedavo s výrazmi bod a pixel, pričom všetky popisujú to isté. *Obrázok* je matica $m \times n$ farebných bodov. *Okno* je usporiadaná trojica (I, P_1, P_2), kde I je obrázok rozmerov $m \times n$ a $P_i (i \in \{1, 2\})$ je dvojica (x_i, y_i) pre $0 < x_i \leq n, 0 < y_i \leq m$. Intuitívne, okno je obdĺžnikový výrez z obrázka.

Dôvod, prečo sme pixel definovali tak všeobecne, je ten, že konkrétne hodnoty závisia od zvoleného farebného modelu. *Farebný model* je abstraktný model, ktorý definuje základné zložky a spôsob ich miešania na vypočítanie farieb. V prírode vznikajú farby miešaním svetla rôznych vlnových dĺžok. Základným cieľom matematického farebného modelu je čo najvierohodnejšie reprodukovať farby. Medzi ďalšie dôležité parametre modelu patria zložitosť výpočtov či podobnosť so spracovaním farby v ľudskom oku. V práci využívame farebné modely RGB a HSV.

RGB je najbežnejší a najznámejší farebný model. Skratku tvoria tri základné farby modelu - červená (Red), zelená (Green) a modrá (Blue). Spôsobov ako reprezentovať tento model je viacero - aj numerických aj geometrických. Definujme farebný bod v RGB modeli ako usporiadanú trojicu celých čísel (R, G, B), kde $0 \leq R, G, B \leq 255$. Dosiahnuteľných farieb je tým pádom $256^3 = 16777216$. Jeden farebný bod reprezentujeme 24 bitmi, kde 8 bitov reprezentuje jednu základnú farebnú zložku farby. Počet bitov reprezentujúcich jeden pixel sa označuje ako *farebná (prip. bitová) hĺbka*.

Druhým farebným modelom, ktorý v práci využívame, je *HSV*. Obdobne ako pri prvom modeli je skratka zložená z prvých písmen názvov základných zložiek, ktorými sú odtieň (Hue), sýtosť (Saturation) a hodnota jasu (Value). Odtieň, nazývaný aj farba či farebný tón, sa typicky meria ako uhol od 0° po 360° na farebnom kruhu. Na obrázku 2.2 je zobrazený typický spôsob vizualizácie HSV farebného priestoru ako valca. Sýtosť predstavuje množstvo šedej vo farbe; čím nižšia sýtosť, tým šedšia farba. Jas predstavuje tmavosť farby; čím nižší jas, tým tmavšia farba. Pre ilustráciu, nech sýtosť a jas reprezentujeme intervalom reálnych čísel $\langle 0, 1 \rangle$. V takom prípade čierna v HSV je reprezentovaná jasom 0, bez ohľadu

[6] http://www.useit.com/papers/responsetime.html



Obr. 2.2: Vizualizácia HSV. Zdroj: http://en.wikipedia.org/wiki/File:HSV_cylinder.png

na ostatné zložky. Bielu dostaneme, keď sýtosť je 0 a jas je 1, bez ohľadu na odtieň. Model HSV sa označuje aj HSB, kde B pochádza z anglického Brightness, čo v preklade znamená priamo jas. Existujú ešte minimálne dva ďalšie podobné farebné modely, ktorými sú HSL (L od Lightness, žiarivosť) a HSI (I od Intensity, intenzita). Tieto modely sú veľmi podobné HSV a nie je náročné prevádzať jeden na druhý. Z tohoto dôvodu sa im nebudeme bližšie venovať. Digitálne budeme pixely v HSV farebnom modeli reprezentovať obdobne ako RGB. Každú zložku zakódujeme pomocou 8 bitov - (H, S, V), pre $0 \leq H \leq 180$[7] a $0 \leq S, V \leq 255$. Zjavne, takto môžeme popísať 11796480 farieb, čo je menej ako pri zvolenej reprezentácii RGB. Dôležité z pohľadu HSV pre nás je rozloženie obrázka na kanály, čo nám poskytne lepšie výsledky pri vyhľadávaní.

Farebný priestor definujeme ako farebný model s mapovacou funkciou do absolútneho farebného priestoru. *Absolútny farebný priestor* je farebný priestor, v ktorom je interpretácia farieb jednoznačná, teda je definovaná bez externých faktorov. Medzi najznámejšie absolútne farebné priestory patria CIE 1976 (L*, a*, b*) a sRGB farebné priestory. Často sa pojmy farebný model a farebný priestor zamieňajú. To je nesprávne. Farebný priestor bez danej referencie na absolútny farebný priestor môže byť rôzne interpretovaný. Neexistuje preto *jeden* RGB farebný priestor. Avšak pre naše účely môžeme 24 bitovú reprezentáciu RGB farebného modelu chápať a nazývať farebným priestorom, keďže nebudeme závislí od konkrétneho referenčného absolútneho farebného priestoru. Bude nás zaujímať jedine veľkosť priestoru a množstvo dát potrebných na uskladnenie informácií o farbe. V ďalšom texte preto budeme označením RGB (HSV) farebným priestorom mať na mysli RGB (HSV) farebný model s 24-bitovým kódovaním.

Pokiaľ zvolíme jednu farebnú zložku z obrázka a vnímame pri každom bode jedine túto, tak takýto nový obrázok s rozmermi pôvodného nazývame *kanál*.

Schéma pre CBIR je konkrétny algoritmus na riešenie CBIR. *Vstupom* budeme nazývať

[7] Do 8 bitov môžeme zakódovať najviac 256 hodnôt a preto mapujeme odtieň z 0° až 360° na čísla 0 až 180 (vrátane).

vstupný obrázok zadaný užívateľom. *Pýtať sa* schémy predstavuje proces od zadania vstupu, cez vyhľadávanie až po výber najpodobnejších obrázkov. *Trénovacím obrázkom* nazývame obrázok, ktorý je dostupný schéme na predspracovanie. *Trénovacia sada* je množina všetkých trénovacích obrázkov, z ktorých schéma vyberá riešenia.

2.4 Príznaky

Ako bolo spomenuté v úvodnej sekcii o CBIR, základom drvivej väčšiny schém je extrakcia príznakov z obrázkov. *Príznak* je v tomto zmysle všeobecný názov pre informáciu získanú z obrázka, ktorá môže byť relevantná pri riešení nášho problému. V praxi sa za príznak volí významný bod, hrana, tvar, histogram, región obrázka, atď. Často sa volia rôzne druhy príznakov pre väčšiu robustnosť schémy. Všetky príznaky extrahované z jedného obrázka tvoria *vektor príznakov*. *Priestorom príznakov* voláme množinu všetkých možných vektorov príznakov. Problém nájdenia podobných obrázkov je následne redukovaný na problém nájdenia najbližších susedov vektora príznakov vstupného obrázka.

Táto zovšeobecnená schéma zachytáva tú zaujímavú časť väčšiny metód riešenia CBIR. Líšia sa v tom:

- ako a aké príznaky extrahujú z obrázkov
- akým spôsobom hľadajú susedov vstupného obrázka v N-rozmernom priestore príznakov, kde N môže byť veľmi veľké

Ako tieto dve otázky rieši naša schéma, popisuje kapitola 3 a 4.

Vhodný výber príznakov je kritický pre úspešnosť CBIR. Intuitívne, príznak by mal byť niečo, čo vystihuje to podstatné v obrázku. Existuje veľa algoritmov na extrakciu rôznorodých príznakov z obrázka. Tie najsofistikovanejšie hľadajú príznaky, ktoré sú invariantné voči niektorým obrázkovým transformáciám. Invariantné znamená, že sa snažia ignorovať potenciálne príznaky, ktoré vzniknú len ako artefakty konkrétnych vonkajších podmienok pri fotení či nepresnostiach fotoaparátu a nesúvisia so samotnou scénou a objektami v nej. Medzi spomínané transformácie patrí translácia, škálovanie, rotácia, šum, zmeny osvetlenia a iné. Nevýhodou použitia takýchto algoritmov je fakt, že sú vo všeobecnosti výrazne výpočtovo náročnejšie. Voľba algoritmov na extrakciu príznakov je teda zvyčajne kompromisom medzi robustnosťou a výkonom, navyše v kontexte domény, z ktorej obrázky pochádzajú. Táto môže určovať, o akých deformáciách medzi podobnými obrázkami treba uvažovať. Napríklad, pokiaľ porovnávame lekárske röntgenové snímky, musíme rátať so šumom, ale menej už so zmenou osvetlenia či zmenou polohy pozorovateľa.

2.4.1 Odtlačky

Základným prostriedkom, ktorý pôvodná schéma využíva je odtlačok okna. *Odtlačok okna* (ďalej len odtlačok) je štvorcový obrázok fixnej veľkosti, ktorý vznikne preškálovaním štvorcového okna obrázka. V schéme zmenšujeme okná pomocou bilineárnej interpolácie. Zjednodušene, bilineárna interpolácia zohľadňuje pri hľadaní farby nového bodu jeho najbližšie okolie veľkosti 2×2 bodov. Následne vypočíta vážený priemer týchto štyroch bodov a výsledná farba je farbou nového bodu. Je to bežný algoritmus využívaný na škálovanie obrázkov.

Odtlačok má tú vlastnosť, že stráca detaily a zachováva len význačné črty či farby. Zachováva taktiež proporcie, keďže škálovanie sa deje zo štvorca na štvorec. Zjavne, pokiaľ sú dve okná rovnaké, budú aj ich odtlačky rovnaké. Pre účely príkladu definujme vzdialenosť dvoch obrázkov veľkosti $N \times N$ I_1 a I_2 ako $d(I_1, I_2) = \sum_{i=1}^{N} \sum_{j=1}^{N} [I_{1_{ij}} \neq I_{2_{ij}}]$, $kde I_{k_{ij}}$ je farebný bod v i-tom riadku a j-tom stĺpci obrázka I_k. Takto definovaná vzdialenosť je teda sumou bodov dvoch obrázkov, v ktorých sa nezhodujú. Pre túto vzdialenosť platí $\exists I_1, I_2 : d(I_1, I_2) < d(h(I_1), h(I_2))$, kde h je funkcia vytvárajúca odtlačok z obrázka v zmysle vyššie uvedenej definície. Inými slovami, odtlačky budú mať viac rozdielnych bodov ako obrázky. Dôkaz je jednoduchý. Stačí si predstaviť obrázky veľkosti 3×3, pričom prvý je čisto biely a druhý je rovnaký až na bod v strede, ktorý je čierny. Vytvorme odtlačky veľkosti 2×2. Odtlačok z prvého obrázka ostane čisto biely. Odtlačok z druhého obrázka bude mať všetky štyri body jemne šedé.

Toto tvrdenie je však založené na tom, že definovaná vzdialenosť je z pohľadu bodov binárna – jednotlivé body sú buď rovnaké alebo nie. Pokiaľ by sme vzdialenosť počítali napríklad ako sumu Euklidovských vzdialeností, lepšie by to zodpovedalo ľudskému vnímaniu podobnosti. Pretože intuitívne, keď sú dva obrázky veľmi podobné, sú aj ich odtlačky veľmi podobné.

2.4.2 Histogramy

Histogram je aproximácia distribúcie náhodnej premennej. Nech priestor príznakov S je rozdelený na M regiónov S_i ($1 \leq i \leq M$) tak, že $S_i \subseteq S$, $\bigcup_{i=1}^{M} S_i = S$ a $S_i \cap S_j = \emptyset$ pre $1 \leq i < j \leq M$. Histogram s M košmi získam tak, že pre každý z M košov spočítam C_i, ktoré hovorí, koľko z N bodov patrí do regiónu S_i ($\sum_{i=1}^{M} C_i = N$). Následne pravdepodobnosť, že bod p padne do košíka S_i dostaneme ako $P[p \in S_i] = C_i/N$. Typicky sú regióny zvolené ako pravidelná mriežka priestoru S, teda sú to hyperkocky rovnakej veľkosti, ale nemusí tomu tak byť. Pokiaľ za priestor S zvolíme RGB farebný priestor, dostaneme 2^{24} regiónov, čo je nepraktické a väčšinou aj zbytočné. Preto musíme vopred farebný priestor kvantifikovať. Ako konkrétne ho kvantifikujeme závisí od situácie, ale je to zväčša kompromis medzi pamäťovými nárokmi a presnosťou (či cielenou nepresnosťou). Ak pracujeme

so šedotónovým obrázkom alebo jedným kanálom, dostávame 256 regiónov, čo už nie je problém zvládnuť.

Výhodou histogramov je, že nám poskytujú pohľad na distribúciu farieb (resp. intenzity pre jeden kanál) v obrázku. Toto je užitočné, pretože podľa tvaru histogramu sa v určitých prípadoch dá o obrázku (či skôr fotografii) niečo povedať. Má to využitie v automatických algoritmoch na zlepšenie vlastností fotografie či analýze obrázkov. Nevýhodou histogramov je, že aj malá zmena obrázka môže mať za následok veľkú zmenu histogramu. Zvyčajne majú však veľmi podobné obrázky aj veľmi podobné histogramy. Taktiež treba brať v úvahu, že dva absolútne nesúvisiace obrázky môžu mať totožné histogramy. Zjavne, aj každá permutácia bodov v obrázku má rovnaký histogram.

Pre porovnanie podobnosti dvoch histogramov použijeme vzdialenosť, ktorá sa označuje *prienik histogramov*. Počíta sa ako $d(H_1, H_2) = \sum_{b=1}^{N} min(H_1(b), H_2(b))$, kde H_1 a H_2 sú histogramy s N košmi a $H(b)$ zodpovedá výške b-teho koša histogramu H. Čím väčšie $d(H_1, H_2)$, tým podobnejšie histogramy sú. Výhodou tejto metódy je pomerne ľahká výpočtová zložitosť. Jej nevýhodou je, že pre ľudské oko veľmi podobné obrázky môžu mať túto vzdialenosť blízku nule, teda byť označené za veľmi nepodobné. Presnejšie, táto metóda je veľmi náchylná na posunutia histogramov.

Často sa používajú rozvitejšie pojmy než histogram, ktoré sú buď jeho synonymá, alebo bližšie špecifikujú akým delením farebného priestoru histogram vznikol. Patria sem histogram jasu, histogram intenzity alebo šedotónový histogram, ktoré vznikajú zo šedotónových obrázkov alebo sledovaním len príslušnej zložky pri farebných bodoch.

Pojem farebný histogram sa zvyčajne chápe ako synonymum histogramu, ktorý sme si vyššie definovali. V práci budeme pod tento pojem chápať špecifickejšie. *Farebný histogram* je trojrozmerný histogram. Každý rozmer delí jednu zo základných zložiek príslušného farebného modelu. Intuitívne si môžeme farebný histogram predstaviť ako trojrozmerné pole daných rozmerov. Do ktorého koša konkrétneho rozmeru pixel spadá určujeme podľa tej farebnej zložky, ktorá rozmeru prislúcha.

2.4.3 Hodnotenie úspešnosti

Prirodzenou otázkou pre každý CBIR vyhľadávač je, ako dobre funguje. Pre konkrétneho užívateľa to je pomerne jednoduché. Vyskúša si ho párkrát a názor si rýchlo urobí. O to lepšie, ak už má skúsenosti s inými obdobnými vyhľadávačmi, pretože má s čím porovnávať. Otázkou však je, ako to robiť čo najmenej subjektívne.

Existujú dva prístupy:

- Ľudské testovanie - zaobstaráme si dostatočne veľkú vzorku užívateľov, ktorých pozorujeme pri používaní nástroja, prípadne sa ich po použití spýtame na názor. Tento spôsob je z definície náchylný na subjektívnosť. Vo všeobecnosti je lepšie užívateľov

pozorovať, než sa ich *pýtať*. To, čo si myslia že robia, môže byť odlišné od toho, čo v skutočnosti robia.

- Automatické testovanie - máme testovaciu sadu s dobre definovaným spôsobom hodnotenia. Tento spôsob je preferovaný, pretože ak je sada rozumne navrhnutá, dá sa objektívne opakovať s rôznymi schémami. Problém pri testovacích sadách pre CBIR je, že sú už pripravované s určitými očakávaniami na schopnosti vyhľadávača. Príslušné obrázky v testovacej sade sú podobné či rozdielne do takej miery, ako to autor navrhol. Takto síce môžeme objektívne testovať rôzne schémy, avšak len v kontexte danej sady. Nemusí totiž pokrývať všetky predstavy užívateľov vyhľadávača o podobnosti obrázkov. Môžu preto vznikať situácie, kedy schéma s výbornými testovacími výsledkami bude subjektívne podľa užívateľov dosahovať zlé výsledky. Jediným spôsobom, ako toto vyriešiť je použitie viacerých testovacích sád. A veriť, že pokrývajú dostatočne veľkú časť očakávaní používateľov.

Štandardné pojmy v IR sú precision a recall[8]. Nech pre konkrétnu otázku na IR systém je R množina vrátených odpovedí a C množina všetkých správnych odpovedí. Potom *Precision* $= \frac{|R \cap C|}{|R|}$ a *Recall* $= \frac{|R \cap C|}{|C|}$. Inými slovami, precision je podiel správnych odpovedí medzi vrátenými odpoveďami a všetkých správnych odpovedí. Recall je podiel správnych odpovedí medzi vrátenými odpoveďami a počtom odpovedí.

O týchto dvoch mierach sa nezvykne hovoriť jednotlivo. Zvyčajne totiž medzi nimi existuje nepriama úmera, kedy je možné zvýšiť jednu hodnotu na úkor druhej. Napríklad zvýšením počtu vrátených obrázkov by sme zlepšili recall, pretože by sa viac správnych obrázkov dostalo do výsledkov, ale dostalo by sa tam aj viac nesprávnych obrázkov, čo by malo za následok zníženie precision.

Nedostatkom presnosti je fakt, že nezohľadňuje pozíciu správnych odpovedí vo výsledkoch. Pre CBIR sa preto často používa miera *priemerná presnosť* (*average precision*). Majme odstupňovaný zoznam výsledkov pre daný obrázok. Priemerná presnosť je stredná hodnota presností, ktoré vypočítame v stupni každého správneho výsledku. Vysoko postavené správne výsledky majú väčší vplyv, čím minimalizujeme spomínaný problém. Aritmetický priemer zvoleného počtu priemerných presností nazývame *stredná priemerná presnosť* (*mean average precision*).

Tieto miery sú však relevantné len v kontexte testovacej sady a jej komplexnosti. V konečnom dôsledku však kvalitu CBIR hodnotia ľudia pri používaní. Preto sa výskum v poslednej dobe sústredí skôr na hľadanie mier, ktoré by zohľadňovali ľudské nároky na kvalitný vyhľadávač. Toto je však otvorený problém.

[8]Nie sú nám známe žiadne zaužívané slovenské ekvivalenty týchto pojmov, preto ich neprekladáme.

Kapitola 3

Popis pôvodnej schémy

Ako bolo spomenuté v úvode, táto práca výrazne stavia na publikácii [1]. Preto je vhodné objasniť aspoň v nutnej miere, o čom táto publikácia je a aký systém na CBIR zavádza. Na záver tejto kapitoly si predstavíme aj dosiahnuté výsledky na testovacích sadách v porovnaní s inými modernými CBIR systémami.

3.1 Princíp fungovania

Základným motívom celej schémy je náhodnosť. Vyskytuje sa vo všetkých fázach schémy - pri výbere príznakov, pri generovaní stromov a aj pri hľadaní podobných obrázkov vstupu. Zjednodušený popis tejto schémy pomocou štyroch krokov zo sekcie 2.2 je nasledovný:

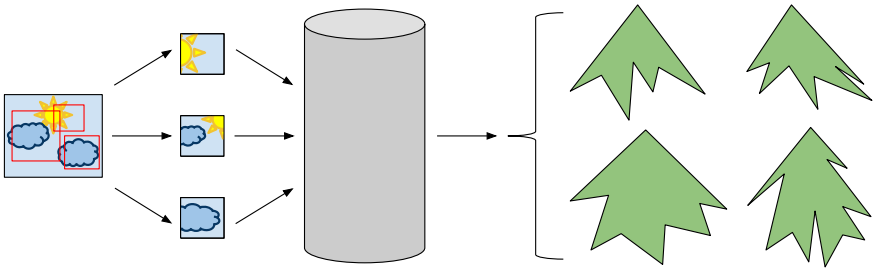
1. Z každého trénovacieho obrázka vyberieme F_I náhodných okien, z ktorých vyrobíme odtlačky. Z týchto vygenerujeme les náhodných binárnych stromov.
2. Zo vstupného obrázka obdobne vyberieme F_Q náhodných odtlačkov.
3. Každým stromov traverzujeme všetkých F_Q odtlačkov vstupu a pamätáme si trénovacie obrázky z listov, v ktorých skončili odtlačky vstupu.
4. Vyberieme najpodobnejšie obrázky podľa stanovenej miery podobnosti.

Zvyšok sekcie sa venuje detailnému popisu fungovania pôvodnej schémy.

3.1.1 Extrakcia príznakov

Práca stavia na nedávnych publikáciách zaoberajúcich sa vzorkovacími schémami na mriežke [10] aj náhodne [11].

Nech testovacia sada má N prvkov. Z každého trénovacieho obrázka sa náhodne vyberie F_I štvorcových okien náhodnej veľkosti na náhodnej pozícii. Každé okno si pamätá, z ktorého obrázka pochádza, avšak ďalej už so samotnými obrázkami pracovať nebudeme.



Obr. 3.1: Kroky schémy: extrakcia odtačkov, vytvorenie zoznamu všetkých odtlačkov, vygenerovanie lesa náhodných binárnych stromov.

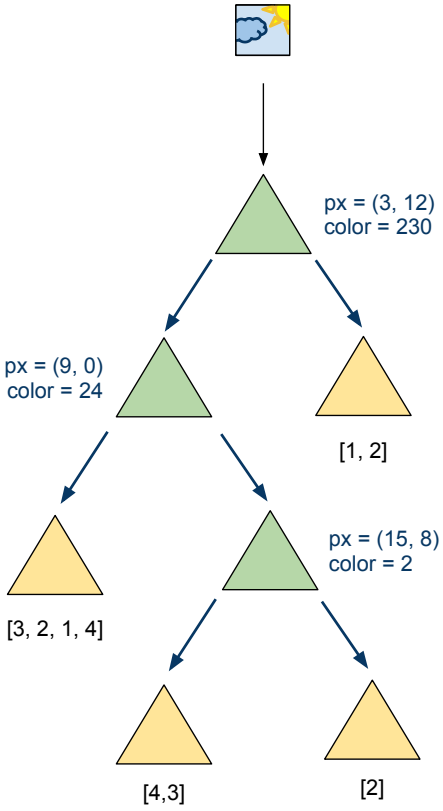
Následne z každého okna vytvoríme odtlačok veľkosti 16×16 pixelov a jednotlivé body reprezentujeme v HSV farebnom priestore. Dostali sme teda NF_I odtlačkov. Každý odtlačok má 256 prvkov v 3 kanáloch. Jeden odtlačok teda predstavuje 768-rozmerný vektor príznakov pre farebné a 256-rozmerný vektor príznakov pre čiernobiele obrázky. Týmto spôsobom dostávame veľký priestor príznakov a bohatú reprezentáciu obrázkov, bez straty potenciálne dôležitých lokálnych informácií v obrázku. Celú trénovaciu sadu môžeme takto predspracovať a pre každý trénovací obrázok si pamätať iba jeho príslušné odtlačky s identifikátorom obrázka.

3.1.2 Generovanie stromov

Hľadanie najbližších susedov (*Nearest-neighbour search*) pre veľký počet príznakov vo vysokorozmernom priestore je výpočtovo náročné. Preto sa objavujú návrhy aproximačných algoritmov na tento problém [12], využívajúce grafové a stromové štruktúry. Práve binárne stromy využíva aj táto práca. Na počítanie podobnosti sa používa les T náhodných binárnych stromov, ktoré sú generované rekurzívne. Každý vrchol stromu predstavuje jednoduchý test. Vrchol obsahuje dve čísla. Prvé určuje náhodný prvok z vektora príznakov, teda číslo od 0 do 767. Druhé číslo určuje prah, ktorý sa vyberá ako náhodné číslo zo všetkých vektorov príznakov prislúchajúcich danému vrcholu na predtým zvolenej náhodnej pozícii. Inak povedané, prvé číslo určuje náhodnú pozíciu na mriežke 16×16 bodov a náhodný kanál. Následne si na chvíľu zapamätáme farebné hodnoty všetkých odtlačkov na danej pozícii v danom kanáli, ktoré vrcholu prislúchajú. Z týchto náhodne vyberieme prah. Takto inicializujeme vrchol. Vrchol si pamätá svoje dve čísla, zoznam svojich odtlačkov a prípadne smerník na ľavého a pravého syna.

Pokiaľ počet odtlačkov prislúchajúcich danému vrcholu prekročí n_{min} a všetky odtlačky nie sú rovnaké, vrchol rozdelíme. Rozdelenie prebieha tak, že najprv si vytvoríme dva zoznamy - pre ľavého a pravého syna. Vrchol sa postupne pýta každého zo svojich odtlačkov, či jeho farebná hodnota v danom kanáli na danej pozícií je menšia ako daný prah[1]. Ak áno,

[1]Tieto parametre sú určené pri inicializácii vrchola a ich hodnotu si vrchol pamätá.



Obr. 3.2: Znázornenie stromu. Vnútorné vrcholy majú náhodné parametre, listy zoznam identifikátorov obrázkov.

pridáme odtlačok do zoznamu pre ľavého syna, inak ho pridáme do zoznamu pre pravého syna. Keď rozdelíme všetky svoje odtlačky, rekurzívne zavoláme incializáciu ľavého syna s ľavým zoznamom a následne incializáciu pravého syna s pravým zoznamom. Zoznam odtlačkov môžeme zabudnúť, keďže vnútorný vrchol stromu už nikdy "svoje" odtlačky mať nebude. Tieto majú len listy stromu.

Takto si vygenerujeme T stromov, pričom koreň každého stromu začína so zoznamom všetkých vektorov príznakov trénovacej sady.

3.1.3 Počítanie podobnosti

Aby sme boli schopní na takýchto štruktúrach vyhľadávať podobné obrázky, musíme si ešte určiť ako počítať miery podobnosti. Nech N_L je počet odtlačkov v liste L stromu $\mathcal{T}$. Potom podobnosť odtlačkov s a s' je

$$k_{\mathcal{T}}(s, s') = \begin{cases} \frac{1}{N_L} & \text{ak } s \text{ aj } s' \text{ patria do } L \\ 0 & \text{inak.} \end{cases} \tag{3.1}$$

Intuitívne, ak dva odtlačky patria do jedného listu, ktorý má málo odtlačkov, sú si podob-

nejšie ako keby patrili do listu s veľa odtlačkami. Podobnosť dvoch odlačkov v lese je

$$k_{ens}(s, s') = \frac{1}{T} \sum_{t=1}^{T} k_{\mathcal{T}_t}(s, s'). \tag{3.2}$$

Dva odtlačky sú si podobné vtedy, ak ich považuje za podobné väčšina stromov. Podobnosť dvoch obrázkov I a I' vypočítame ako

$$k(I, I') = \frac{1}{|S(I)||S(I')|} \sum_{s \in S(I), s' \in S(I')} k_{ens}(s, s'), \tag{3.3}$$

kde $S(I)$ ($S(I')$) je množina odtlačkov všetkých možných okien obrázka I (I'). Keďže toto číslo môže byť veľmi veľké, navrhujú autori aproximovať túto podobnosť počítaním len cez vopred stanovený počet okien. Ďalej upozorňujú, že aj keď tento vzorec vytvára dojem kvadratickej zložitosti, využitím stromovej štruktúry dosiahneme zložitosť lineárnu.

3.1.4 Vyhľadávanie obrázkov

Teraz už máme všetky prostriedky potrebné k vyhľadávaniu. Máme trénovaciu sadu N obrázkov a chceme z nich nájsť R najpodobnejších obrázkov vstupu I_Q. Najprv necháme z trénovacej sady vyrásť les stromov. Následne vypočítame pre každý obrázok $I \in N$ hodnotu $k(I_Q, I)$ a vrátime R obrázkov s najvyššou podobnosťou.

Podobnosť obrázkov I_Q a I môžeme prepísať do nasledovnej podoby:

$$k(I_Q, I) = \frac{1}{T} \sum_{t=1}^{T} \sum_{L \in \mathcal{T}_t} \frac{1}{N_L} \frac{N_{I_Q,L}}{F_Q} \frac{N_{I,L}}{F_I}, \tag{3.4}$$

kde vnútorná suma iteruje cez listy t-teho stromu, $N_{I_Q,L}$ ($N_{I,L}$) je počet odtlačkov I_Q (I) v liste L.

Z tejto formulácie lepšie vidno, ako efektívnejšie túto podobnosť počítať:

- po vygenerovaní stromu si v každom liste pamätáme počet odtlačkov, ktoré patria jednotlivým obrázkom

- vyhľadávanie robíme tak, že postupne každý odtlačok vstupu necháme prejsť každým stromom, až kým padne do niektorého listu. Keď sa odtlačok dostane do listu L, inkrementujeme podobnosť $k(I_Q, I_L)$ pre každé I_L patriace do L o hodnotu F_Q/N_L. Nakoniec predelíme všetky pravdepodobnosti konštantou TF_IF_Q.

Následne môžeme R najpodobnejších obrázkov nájsť so zložitosťou $O(RN)$. Celková zložitosť vyhľadávania je v priemere $O(TF_Q(log(F_I) + N))$.

3.2 Vlastnosti schémy

Pozrime sa bližšie na dobré vlastnosti schémy:

- Rýchle vyhľadávanie - keďže vrcholy vykonávajú len veľmi jednoduché porovnanie dvoch celých čísel, je vyhľadávanie veľmi rýchle. Dlhšie než samotné vyhľadávanie trvá predspracovanie vstupu, teda vytváranie odtlačkov.
- Pamäťové nároky - nutných informácií je v strome veľmi málo. Vrchol potrebuje smerníky na dvoch synov a jedno celé číslo. Listy si okrem toho musia pamätať zoznam identifikátorov obrázkov dlhý maximálne n_{min}.
- Koniec hocikedy - počas vyhľadávania sa nám podobnosť s trénovacími obrázkami počíta okamžite (hneď ako sa dostane odtlačok do listu) a preto môžeme v prípade, že počítame príliš dlho, skončiť a predsa mať čiastočný výsledok.
- Paralelizácia - veľa častí je veľmi dobre paralelizovateľných: vyberanie náhodných okien a vytváranie odtlačkov, generovanie stromov, súčasné vyhľadávanie na viacerých stromoch, atď.

Práve tieto vlastnosti boli dôvodmi výberu tejto schémy ako základu našej práce.

3.3 Testovacie výsledky

Autori vykonali niekoľko experimentov pre rôzne nastavenia parametrov, konkrétne pre počet stromov, maximálny počet odtlačkov v liste a počet odtlačkov z jedného obrázka. Testovanie prebiehalo na štyroch testovacích sadách - UK-Bench, ZuBuD, IRMA-2005 a "META". Posledná menovaná sada vznikla kombináciou rôznych iných dostupných testovacích sád.

Pri UK-Bench [13] dosiahla schéma s $F_I = 1000, T = 10, n_{min} = 4$ hodnotu tesne nad 3 predstavujúcu 75,25% úspešnosť. Na databáze ZuBuD dosiahli s $F_I = 1000, T = 10$ a niekoľkými hodnotami n_{min} menej ako 10 úspešnosť rozpoznávania 95,65%. Hodnota lepšia ako 90% sa podarila dosiahnuť aj pre len jeden strom a 500 odtlačkov z jedného obrázka. Na testovacej databáze IRMA dosiahli 85,4% úspešnosť s $F_I = 1000, T = 10, n_{min} = 2$.

Kapitola 4

Upravená schéma

Táto kapitola sa venuje nášmu prínosu do schémy popísanej v predchádzajúcej kapitole. Opisuje motiváciu rozšírení a zmien, ktoré som vykonal, ale len z teoretického hľadiska a očakávaných prínosov. Či sa v jednotlivých prípadoch očakávania naplnili, popisuje nasledujúca kapitola s testovacími výsledkami.

Spoločným motívom takmer všetkých návrhov v tejto kapitole je, že robia schému všeobecnejšou. A to na rôznych úrovniach. Jednak je to viac metód generovania stromov, každá s odlišnými vlastnosťami a úspešnosťou hľadania. Ďalej zovšeobecnenie pojmu vrchol a jeho rozšírenia o iné typy ako vrcholy na základe odtlačku okna. Rozumná úroveň abstrakcie sa taktiež nachádza v implementácii, o ktorej bližšie hovorí predposledná sekcia tejto kapitoly.

Pripomeňme si, že z jedného obrázka najprv vyberieme F_I náhodných okien. Z nich sme pôvodne robili odtlačky a týchto F_I odtlačkov reprezentovalo obrázok. V práci sme však neuvažovali len o robení odtlačkov z okien, ale taktiež vyberáme iné príznaky. Tým pádom by bolo zavádzajúce ďalej označovať vektor príznakov získaný z okna odtlačkom. V ďalšom texte preto *príznakom* budeme označovať vektor príznakov získaný z jedného náhodného okna konkrétneho obrázka.

Neformálne si popíšme, čo sa stane s dvoma veľmi podobnými trénovacími obrázkami v jednom strome. Z týchto obrázkov vyberieme F_I okien náhodnej veľkosti na náhodných miestach. Ak je F_I dostatočne veľké a máme dobrý generátor náhodných čísel, bude väčšina okien z jedného zoznamu mať veľmi podobné okno v druhom zozname. Podobné v zmysle, že budú podobne veľké a zhruba v rovnakom okolí vybraté. Keď následne z týchto okien extrahujeme príznaky, stratia sa malé rozdiely ešte viac. Traverzovanie je deterministické, teda rovnaké príznaky dôjdu do rovnakého listu. Z tohoto dôvodu sa po vložení spomínaných $2F_I$ príznakov do stromu veľa príznakov z prvého obrázka dostane do listu, kde sa nachádza príznak z druhého obrázka. Podobné príznaky *nemusia* skončiť v rovnakom liste. Čím sú však podobnejšie, tým je pravdepodobnejšie, že sa tak stane. Neskončia v rovnakom liste, ak po

ceste nejaký vnútorný vrchol stromu rozdeľuje podľa toho prvku[1], v ktorom sa spomínané príznaky líšia a súčasne sú parametre tohoto vrchola vygenerované tak, že príznaky pošlú do rôznych synov.

4.1 Metódy generovania stromov

V pôvodnej publikácii sa autori príliš nevenovali možným spôsobom generovania stromov. Pracovali s jednou metódou a tou bola rekurzívna metóda delenia, ktorá vychádza z predpokladu, že v koreni stromu sa nachádza zoznam všetkých príznakov. V jednom odseku bolo spomenuté, že sa dajú stromy rozširovať aj po vytvorení, takpovediac za behu. Konkrétny postup je opísaný neskôr v tomto odseku. Dôležité však je, že autori vyslovili očakávanie, že stromy vygenerované takýmto spôsobom postupného zväčšovania počtu obrázkov v strome budú pravdepodobne dosahovať podobné výsledky, ako stromy vygenerované s novým počtom obrázkov od začiatku. Toto tvrdenie nás zaujalo. Rozhodli sme sa preto implementovať aj túto metódu generovania a testovaním podporiť alebo vyvrátiť ich tvrdenie.

Pamäťové nároky na strom sú pomerne malé. Vo vnútorných vrcholoch si stačí pamätať smerníky na synov a náhodné parametre. V listoch to, čo vo vnútorných vrcholoch a navyše zoznam identifikátorov obrázkov s príslušným počtom príznakov z toho istého okna, ktoré sa do listu dostali. Pre všetky metódy platí, že ak chceme neskôr strom ďalej rozširovať, musíme si v listoch pamätať ešte aj informácie nutné na rekonštruovanie príznakov jeho okien[2].

Aby sme mali ľahšiu predstavu o tom, o akých pamäťových nárokoch budeme hovoriť, odhadnime množstvo pamäte nutnej na zapamätanie si 1000 obrázkov pre $F_I = 1000$. Nech príznak tvorí, tak ako v pôvodnom článku, len odtlačok. Potom veľkosť jedného príznaku v pamäti[3] je aspoň 768 B, pretože obsahuje 256 bodov v HSV farebnom priestore a to ešte neuvažujeme o pamäti nutnej na identifikátor obrázka. Z toho vyplýva, že potrebujeme viac než 730 MB pamäte len na zapamätanie si milióna príznakov.

Pre popis metód v pseudokóde si musíme najprv ukázať niekoľko zjednodušených pomocných funkcií:

Algoritmus 1 vytvorí nový vrchol a prípadne mu nastaví otca či zoznam príznakov. Je zjednodušený o informovanie otca vrcholu o novom synovi a o tom, či ide o ľavého alebo pravého syna. K zoznamu príznakov vrcholu N sa pristupuje cez `list(`N`)`.

Jeden z najdôležitejších algoritmov schémy je vo funkcii `split`, opísanej v Algoritme 2,

[1] Prvkom myslíme príznak z vektora príznakov, v štandardnej terminológii.

[2] Rekonštruovať okná môžeme nasledovným spôsobom: pamätáme si zoznam okien, v ktorom je pri každom zázname identifikátor okna, identifikátor obrázka, ľavý horný roh voči obrázku a rozmer okna. Vo vrcholoch si pamätáme len identifikátory okien do tohoto zoznamu.

[3] Uvažujme, že nepoužívame kompresiu, pretože nárast časovej zložitosti by bol pravdepodobne veľmi výrazny, keďže k bodom musíme pristupovať často.

Algoritmus 1: createNode

Input: voliteľné; zoznam príznakov F, otec P
Output: nový vrchol N
$N \leftarrow$ `new` Node;
if F **then**
  `list(`N`)` $\leftarrow F$;
end
if P **then**
  `parent(`N`)` $\leftarrow P$;
end

Algoritmus 2: split

Input: vrchol N, ktorý má byť rozdelený
Output: -
if `list(`N`)` $< n_{min}$ **then**
  return;
end
$tries \leftarrow 0$;
$leftlist, rightlist \leftarrow$ `new` List;
while $leftlist = \emptyset$ **or** $rightlist = \emptyset$ **do**
  $tries \leftarrow tries + 1$;
  if $tries >$ MAXTRIES **then**
    return;
  end
  if not `regenerate(`N`)` **then**
    continue;
  end
  $leftlist, rightlist \leftarrow \emptyset$;
  foreach $F \in$ `list(`N`)` **do**
    if `chooseChild(`N, F`)` = LEFT_CHILD **then**
      `add(`*leftlist,* F`)`;
    else
      `add(`*rightlist,* F`)`;
    end
  end
end
`clear(features(`N`))`;
$left \leftarrow$ `createNode(`$leftlist$, N`)`;
`split(`$left$`)`;
$right \leftarrow$ `createNode(`$rightlist$, N`)`;
`split(`$right$`)`;

pretože je základom celého generovania stromov. Algoritmus vytvorí dvoch synov a rozdelí medzi ne svoje príznaky, ak ich počet je aspoň n_{min}. Funkcia `chooseChild(`N`, `F`)` zisťuje, či má príznak F vo vrchole N ísť do ľavého alebo pravého syna. Do `while` cyklu vstúpime aspoň raz, kedy sa na začiatku zavolá funkcia `regenerate(`N`)`. Táto funkcia má len dva kroky. Prvým je uvoľnenie synov a zrušenie ich podstromov, ak treba. Druhým je zavolanie funkcie `init(`N`)` a ako návratová hodnota sa použije jej hodnota. Funkcia `init(`N`)` nastavuje vrcholu N náhodné parametre podľa definície 4.3 a je podrobnejšie popísaná v sekcii 4.3 pri popise triedy `Vrchol`.

Parametre sa pokúšame vygenerovať, až kým sa nám pre nejaké nepodarí rozdeliť zoznam príznakov do dvoch skupín alebo kým neprekročíme limit povolených pokusov. Keď sa nám to podarí, vymažeme vlastný zoznam príznakov, keďže si ich už nemusíme pamätať. Následne vytvoríme synov, posunieme každému jeden zoznam a rekurzívne na nich zavoláme `split`. V implementácii má funkcia na vstupe boolovskú premennú $recursive$, ktorá určuje, či sa má na konci rekurzívne volanie na synoch zavolať alebo nie.

Algoritmus 3: traverse

Input: vrchol N z ktorého sa začína a príznak F
Output: list $prev$
cur, $prev \leftarrow N$;
while $cur \neq$ NULL **do**
    $prev \leftarrow cur$;
    $cur \leftarrow$ `getChild(`cur, F`)`;
end

Algoritmus vráti list podstromu s koreňom N, do ktorého sa dostane príznak F. Funkcia `getChild(`N`, `F`)` funguje rovnako ako `chooseChild(`N`, `F`)` s tým rozdielom, že vráti priamo syna do ktorého príznak patrí alebo $NULL$ ak ide o list.

4.1.1 Hromadná metóda

Metódu, ktorú využili aj autori pôvodnej práce, nazveme *hromadná metóda*. Pseudokód generovania stromu touto metódou je jednoduchý a nachádza sa v Algoritme 4.

Algoritmus 4: Hromadná metóda

Input: -
Output: strom T
$F \leftarrow$ `features(`Trainset`)`;
$T \leftarrow$ `new` Tree;
`root` (T) $\leftarrow$ `createNode` (F);
`split` (`root` (T));

$Trainset$ je trénovacia sada, teda množina všetkých obrázkov. Funkcia `features(`$Trainset$`)` vyberie náhodné okná zo všetkých obrázkov v $Trainset$, ex-

trahuje z nich príznaky a vráti ich zoznam. Funkcia `features(`I`)` vráti zoznam príznakov pre konkrétny obrázok I.

Je vhodné spomenúť, že túto metódu implementujeme pomocou nerekurzívnej verzie funkcie `split`. Algoritmus pre takúto hromadnú metódu je taktiež jednoduchý a funguje na princípe prehľadávania do šírky. Udržujeme si FIFO zoznam, v ktorom je na začiatku iba koreň. Následne opakujeme, kým nie je zoznam prázdny: vyber vrchol zo zoznamu, zavolaj metódu `split` a keď bol vrchol rozdelený, tak vlož jeho synov na koniec zoznamu. Nerekurzívne algoritmy sú vo všebecnosti šetrnejšie z pohľadu ako pamäťovej tak časovej zložitosti, preto táto voľba.

Vlastnosti:

\+ Najlepšie výsledky vyhľadávania.

— Keď chceme pridať nový obrázok do stromu, musíme ho celý znovu vygenerovať alebo použiť na rozšírenie inkrementálnu metódu.

— Dlhší čas generovania, hlavne pokiaľ nemáme dosť pamäte na všetky príznaky naraz. V tomto prípade veľmi rýchlo narastá časová zložitosť so zmenšujúcou sa pamäťou, pretože si budeme musieť každý obrázok mnohokrát načítavať a extrahovať z neho príznaky. A to sú práve operácie v praxi najnáročnejšie na čas pri generovaní stromu. Pokiaľ máme dostatok pamäte na všetky príznaky nejakého podstromu, tak tento vieme vygenerovať bez zhoršenia zložitosti. Ak však nemáme, tak delenie vrchola môže byť časovo náročné, pretože sa príznaky môžu rozdeliť práve tak, že si nedržíme v pamäti tie správne. Keby sme totiž vedeli rýchlo určiť, ktoré tie správne sú, nepotrebujeme už generovať stromy a vieme toutou cestou dať dokopy príbuzné príznaky.

4.1.2 Inkrementálna metóda

Druhú metódu spomínanú v úvode sekcie nazveme *inkrementálna metóda*. Jej pseudokód sa nachádza v Algoritme 5.

Vlastnosti má táto metóda opačné ako hromadná:

— Očakávame mierne zhoršenie presnosti vyhľadávania oproti hromadnej metóde.

\+ Ďalšie rozšírenie stromu sa nijako nelíší od jeho generovania.

\+ Rýchlejšia ako hromadná metóda. Ešte výraznejšie, pokiaľ nemám k dispozícii operačnú pamäť na zapamätanie si všetkých príznakov naraz. A ako bolo v úvode sekcie ukázané, pamäťové nároky už pre pomerne malý počet obrázkov sú veľké.

Analyzujme, koľko obrázkov je nutné načítať v najhoršom prípade pri pridávaní jedného obrázka do stromu inkrementálnou metódou. Nech $\mathcal{F}_1, ..., \mathcal{F}_{F_I}$ sú príznaky obrázka I, ktorý chceme pridať do stromu T. Počet obrázkov nutných načítať maximalizujeme, keď:

Algoritmus 5: Inkrementálna metóda

```
Input: -
Output: strom T
T ← new Tree;
root(T) ← createNode();
foreach I ∈ Trainset do
    foreach F ∈ features(I) do
        leaf ← traverse(root(T), F);
        add(list(leaf), F);
        split(leaf);
    end
end
```

- každý príznak $\mathcal{F}_i$ skončí v inom liste
- všetky listy majú n_{min} príznakov, aby ich bolo nutné deliť, keď sa pridá ďalší príznak
- žiadne dva príznaky zo všetkých F_I listov nepatria rovnakému obrázku

Každý list, do ktorého nejaké $\mathcal{F}_i$ padne, treba rozdeliť. List si pamätá len identifikátor príznaku, potrebujeme ho preto znovu z obrázka extrahovať a tým pádom ho načítať. Spolu teda potrebujem načítať $F_I n_{min}$ trénovacích obrázkov. Pre bežné hodnoty $F_I = 1000, n_{min} = 4$ potrebujeme načítať v najhoršom prípade až 4000 obrázkov pri pridávaní jediného, čo je veľké množstvo. Ako vyzerá situácia v priemernom prípade, popisuje na základe experimentov nasledujúca kapitola.

Dôvod, prečo očakávame mierne zhoršenie presnosti, je nasledovný. Bolo spomenuté, že inkrementálna metóda vkladá obrázky do stromov postupne. Pri delení vrchola teda tento ešte nemusí mať v zozname všetky príznaky z trénovacích obrázkov, ktoré by sa do neho dostali, keby už boli vložené do stromu. Toto je v kontraste s hromadnou metódou, kde vrchol má v čase generovania informácie o všetkých príznakoch trénovacej sady, ktoré mu prislúchajú. Intuitívne, vrchol v hromadnej metóde má "väčší prehľad" o hodnotách príznakov a môže túto informáciu lepšie využiť pri generovaní svojich parametrov.

4.1.3 Metóda malých lesov

Obe doteraz spomínané metódy majú spoločnú vlastnosť, že počet vrcholov v strome rastie lineárne s počtom obrázkov. Tretia metóda sa odlišuje v tom, že maximálny počet obrázkov v jednom strome je limitovaný. Presnejšie, pre každých N_{set} obrázkov máme vytvorených T_{set} stromov. Nech trénovacia databáza má N obrázkov. Les potom bude mať $\lceil \frac{N}{N_{set}} \rceil T_{set}$ stromov. Nové trénovacie obrázky pridávame okamžite, inkrementálnou metódou.

Motiváciou pre takýto spôsob generovania stromov je obmedzená pamäťová zložitosť jedného stromu. Toto zabezpečí, že strom nepresiahne určitú veľkosť, čo má prirodzene praktický význam. Očakávame, že táto metóda prinesie zhoršenie presnosti vyhľadávania. Dôvod je obdobný ako pri inkrementálnej metóde, umocnený ešte faktom, že strom musí vybrať podobné obrázky len z podmnožiny trénovacej sady. A to aj v prípade, že jeho podmnožina reálne žiadne podobné obrázky neobsahuje. Na druhú stranu, ak strom nemá v prislúchajúcej trénovacej množine podobné obrázky, zvyčajne sa podobnosť viac-menej rovnomerne rozdeľuje medzi väčšie množstvo obrázkov. Preto by tento problém mohol zmierniť parameter T_{set}, ktorý určuje, koľko stromov obsahuje ten istý balík obrázkov. Vytvárame teda veľa malých lesov spôsobom ako v predchádzajúcich metódach, od čoho je odvodený názov.

Algoritmus 6: Metóda malých lesov

```
Input: -
Output: les F obsahujúci T_set stromov
for t ← 1 to T_set do
    F[t] ← new Tree;
end
while true do
    for i ← 1 to N_set do
        I ← getNewTrainImage();
        for t ← 1 to T_set do
            F[t] ← incremental(F[t], I);
        end
    end
end
```

Algoritmu postupne chodia nové trénovacie obrázky, na ktoré čaká v `getNewTrainImage`. Funkcia `incremental(`T`, `I`)` je obdobou inkrementálneho generovania s tým rozdielom, že na vstupe dostane už existujúci strom a rozšíri ho len o obrázok I.

Vlastnosti:

− Pomalšie vyhľadávanie, pretože rýchlosť vyhľadávania je lineárne závislá od počtu stromov.

\+ Vieme stanoviť maximálnu pamäťovú náročnosť generovania aj prevádzky.

± Rýchlosť generovania závisí od parametru T_{set} aj N_{set}.

− Očakávame zhoršenie presnosti vyhľadávania.

4.1.4 Kombinovaná metóda

Ak by sa ukázalo, že inkrementálna metóda dosahuje zreteľne horšie výsledky ako hromadná metóda, môže byť rozumnou alternatívou pre prax táto tzv. kombinovaná metóda.

Najprv vygenerujeme les z trénovacej sady a môžeme začať vyhľadávať. Časom však budeme chcieť trénovaciu sadu rozšíriť. To vykonáme inkrementálnou metódou. Aby sa však kvalita vyhľadávania opakovaným inkrementálnym rozširovaním príliš nezhoršovala, po $limit$ rozšíreniach vygenerujeme celý les odznova hromadnou metódou. Takto opäť dosiahneme najlepšiu možnú kvalitu vyhľadávania. S tým rozdielom, že sme stále schopní vyhľadávať v celej trénovacej sade, bez nutnosti vytvárať celý les odznova pri každom rozšírení. Princíp metódy je popísaný v algoritme 7.

Algoritmus 7: Kombinovaná metóda

Input: -
Output: Strom T
$T \leftarrow$ `bulk();`
while $true$ **do**
  for $i \leftarrow 1$ **to** limit **do**
    $I \leftarrow$ `getNewTrainImage();`
    $T \leftarrow$ `incremental(`T`,` I`);`
  end
  $T \leftarrow$ `bulk();`
end

Algoritmus beží ako vedľajší proces pre rozširovanie a po úvodnom `bulk` sa na ňom už v inom procese vyhľadáva. Funkcie `getNewTrainImage` a `incremental(`T`,` I`)` sa správajú rovnako ako v Algoritme 6.

Metóda kombinuje dobré vlastnosti hromadnej a inkrementálnej metódy.

4.2 Distribúcia vrcholov

Druhou témou, ktorej sa táto kapitola venuje, je navrhnutie rozšíriteľného systému pre nové typy vrcholov v stromoch. Jediným typom vrchola v pôvodnej práci bol vrchol porovnávajúci na základe náhodného bodu a farby v odtlačku okna. Avšak odtlačok nie je jediný druh príznaku, ktorý sa dá z okna extrahovať. Preto v práci v odseku 4.3 navrhujeme ďalšie typy vrcholov, založené na iných druhoch príznakov, ktoré sa dajú používať v stromoch. Predtým však ešte musíme definovať spôsob výberu typov vrcholov pre konkrétny strom.

Nech M je počet typov vrcholov a $V_1, ..., V_M$ označujú jednotlivé typy. Nech T je strom. Potom $V_i(T)$ predstavuje množinu vrcholov stromu, ktoré sú i-teho typu. $\bigcup_{i=1}^{M} V_i(T) = V(T)$ a $V_i \cap V_j = \emptyset$ pre $1 \leq i < j \leq M$.

Homogénnym stromom budeme nazývať strom, v ktorom $\exists i \;\; V_i(T) = V(T)$. Teda

strom, v ktorom všetky vrcholy sú jedného typu. *Heterogénny strom* je taký, ktorý nie je homogénny. Teda $\exists i, j\ (i \neq j)\ V_i(T) \neq V_j(T) \neq \emptyset$.

Ak by sme chceli len homogénne stromy, tak stačí pri konštrukcii stromu doplniť parameter určujúci požadovaný typ vrcholu. Zaujímavejšie však sú heterogénne stromy. Otázka teda znie, čo určuje akého typu bude ďalší vrchol? Možných spôsobov je veľa. Napríklad zadať permutáciu podmnožiny typov vrcholov. Pri vytváraní vrchola najprv zistíme typ podľa prvého prvku zoznamu a následne tento presunieme na koniec zoznamu. Alebo môžeme určovať typ vrchola podľa jeho hĺbky modulo veľkosť zoznamu. Keďže schéma využíva do veľkej miery náhodnosť, vhodnejšie pôsobí jej využitie aj v tomto prípade. Zavedieme preto ako parameter stromu pravdepodobnostnú distribúciu vrcholov.

Nech M je počet typov vrcholov. Nech $D = (D_1, ..., D_M)$, kde $\forall i\ \ D_i \in \langle 0, 1\rangle$ a $\bigcup_{i=1}^{M} D_i = 1$. Potom D nazveme *pravdepodobnostnou distribúciou vrcholov* (ďalej len distribúcia). Intuitívne, distribúcia hovorí, s akou pravdepodobnosťou sa zvolí daný typ pri vytváraní nového vrcholu. Formálnejšie, nech T je strom s distribúciou D. Potom, $(|V(T)| \to \infty)$ implikuje $(\forall i\ \frac{|V_i(T)|}{|V(T)|} \to D_i)$.

Algoritmus 8: Výber náhodného typu vrchola

Input: Distribúcia D, náhodné $r \in \langle 0, 1)$
Output: Číslo i, ktoré hovorí ktorý typ vrchola bol zvolený

```
f ← 0;
i ← 1;
while r ≥ f ∧ i ≤ |D| do
    f ← f + D[i];
    i ← i + 1;
end
i ← i − 1;
```

Takto definovaná distribúcia má zaujímavé vlastnosti. Jedným spôsobom môžeme generovať homogénne[4] aj heterogénne stromy. Ďalej, nie všetky typy vrcholov pre nás môžu mať rovnaký význam a týmto spôsobom môžeme ľahko určovať váhu jednotlivých typov.

4.3 Rozšírenia vrcholov

Ako bolo uvedené v sekcii 4.2, dôležitou zmenou oproti pôvodnej schéme je pridanie ďalších typov vrcholov. V tomto odseku si uvedieme, ako sme vrchol transformovali na abstraktnú triedu, ktorej sú ostatné typy potomkami. Ďalej si popíšeme pôvodné typy vrcholov a nakoniec typy nové.

Parametrom typu vrcholu (ďalej len parametrom) budeme v tejto sekcii nazývať členskú premennú, ktorá je špecifická pre tento typ, je náhodne vygenerovaná pri inicializácii vrchola

[4] Stačí príslušný typ v distribúcii nastaviť na 1 a ostatné na 0.

a využíva sa pri určovaní syna pre príznak. Parametre určujú náhodné dáta nutné pre vrchol, aby rozhodol o tom, kam má príznak ísť.

4.3.1 Trieda Vrchol

Naša schéma si stanovila za cieľ nielen vytvoriť viacero typov vrcholov, ale zabezpečiť aj ďaľšiu čo najjednoduchšiu rozšíriteľnosť systému. Preto sa ako najrozumnejšia javí cesta definovať vrchol ako abstraktnú triedu. Trieda Vrchol definuje členské premenné a virtuálne metódy, ktoré musí potomok tejto triedy implementovať.

Trieda 9: Zjednodušená schéma triedy `Node`

Členské premenné:
list – zoznam identifikátorov príznakov vrchola; nutný iba pre vnútorné vrcholy
tree – smerník na strom
parent – smerník na otca, prip. na seba ak ide o koreň
left – smerník na ľavého syna, ak existuje
right – smerník na pravého syna, ak existuje

Metódy:
`constructor` – nastaví členské premenné
`init` – nastaví parametre
`chooseChild(`f`)` – vyberie do akého syna vrchola príznak f patrí

Medzi vynechané prvky patrí premenná `id` a metódy `toString` a `fromString`. Tieto sú dôležité hlavne kvôli ladeniu kódu a ukladaniu dát, ktoré bude spomenuté v odseku 4.4.

Členské premenné ani konštruktor nie sú zaujimavé. Detaily závisia od zvoleného jazyka, ale konštruktor potomka sa len odvoláva na konštruktor triedy Vrchol a prípadne môže sám niečo vykonávať. Práve už spomínaný `init` nastaví vrcholu náhodné aj všetky ostatné parametre špecifické pre jeho typ. Metóda `chooseChild` sa využíva, keď sa rozhoduje kam sa má príznak vybrať.

Zjednodušene povedané, to podstatné sa nachádza len v dvoch metódach – `init` a `chooseChild`. Všetky ostatné vznikajú buď z technických obmedzení alebo si ich vyžadujú ďaľšie schopnosti schémy, ktoré považujeme za užitočné. Pri rozumných typoch vrcholov musí v metóde `chooseChild` na konci existovať boolovský výraz, ktorý určuje či sa má vrátiť ľavý alebo pravý syn. Stojí za zmienku, že pokiaľ by sme zvolili negáciu tohoto výrazu a nevymenili poradie voľby synov, nič sa nezmení. Na tomto poradí nezáleži a nemalo by to vplyv na výsledky vyhľadávania. Podstatné je len, aby sme to robili konzistentne v rámci vrchola.

Každý typ vrchola má okrem jeho parametrov uvedenú aj príslušnú podmienku z `chooseChild`. Kvôli spomenutým dôvodom by bolo zbytočné uvádzať, ktorého syna vraciame pri splnení[5]. Obsah metódy `init` je vždy len slovne zhrnutý a jej detaily sa dajú

[5] V implementácii sa pri splnení podmienky vracia vždy ľavý syn.

nájsť v kóde implementácie.

Konkrétna implementácia metódy `init` môže výrazne ovplyvniť schopnosti daného typu vrchola. Štandardne je volaná len z metódy `regenerate`, ktorá je pre zmenu volaná z metódy `split`. Ako je vidno z jej popisu v Algoritme 2, `regenerate` sa môže volať viacnásobne, pokiaľ vygenerované parametre nie sú schopné rozdeliť príznaky vrchola na dve skupiny. Aby nevznikali zacyklenia, je počet volaní tejto metódy obmedzený vopred stanovenou malou konštantou. Pokiaľ sa nám príliš veľa krát nepodarí v jednom volaní `split` vrchol rozdeliť, necháme vrchol nerozdelený, aj keď má priveľa príznakov. Z tohoto dôvodu je pre úspešnosť vyhľadávania dôležité, aby sa `init` snažil parametre voliť rozumne a náhodne súčasne.

Toto je čiastočne spomenuté aj v sekcii 3.1, vychádzajúcej z pôvodnej práce. V práci sa spomína[6], že vrchol sa prestane deliť, keď sú odtlačky identické alebo má vrchol menej ako n_{min} odtlačkov. Nespomína sa však, či sa na začiatku delenia vždy skontroluje, či sú odtlačky identické, alebo sa síce vyberá parameter bodu náhodne, ale postupne sa skúša všetkých 256 možných.

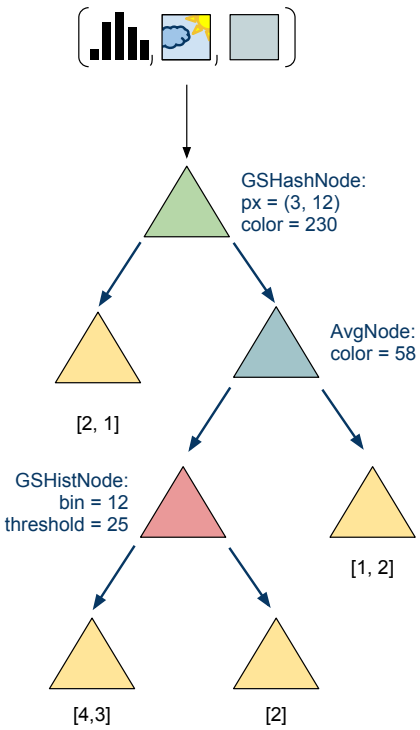
Vo všeobecnosti však kontrola identity príznakov na začiatku trvá dlho a pamätanie si už skúšaných môže byť náročné z viacerých dôvodov, napríklad pri parametroch s pohyblivou rádovou čiarkou. To je dôvod stanovenia maximálneho limitu opakovania volania `init` zo `split` v implementácii. Ďalej, v metódach `init` pre väčšinu typov vrcholov najprv náhodne zvolíme jeden parameter a podľa tohoto prehladávame zoznam príznakov a podľa nich volíme minimálnu a maximálnu hranicu pre ostatné parametre. Robíme tak preto, aby sme zbytočne náhodne nezvolili parametru takú hodnotu, ktoré príznaky v zozname ani neobsahujú. Metódy `init` teda v implementácii nie sú celkom naivné generovaním úplne náhodných hodnôt, ale ani sa nesnažia zistiť, či vôbec rozdeľujúce parametre existujú, za cenu zvýšenej výpočtovej zložitosti. Metóda `init` vracia pravdivostnú hodnotu $false$, ak zvolené parametre určite nerozdelia príznaky na dve skupiny a inak $true$.

4.3.2 Základné typy

V predchádzajúcej kapitole bolo detailne popísané, akým spôsobom sa vyhľadáva a konštruuje strom na základe odtlačkov. V našej schéme sme pre odtlačky vytvorili dva typy vrcholov - `GSHashnode` a `HashNode`. Rozdiel je v tom, že prvý menovaný robí porovnania na základe čiernobieleho odtlačku a druhý z odtlačku v HSV. Parametre sú bod na mriežke, náhodná farba a v prípade `HashNode` ešte náhodný kanál. Metóda `chooseChild` sa pozrie na bod odtlačku príznaku (a príp. kanál) daný parametrom vrchola a porovná ho s parametrom farby.

Kvôli jednoduchosti predpokladajme, že funkcia `hash(`f`, `y`, `x`)` vracia v prvom prípade intenzitu a v druhom bod odtlačku okna z príznaku na pozícii (y, x).

[6]Str. 47, sekcia 2.2, tretí paragraf v [1].



Obr. 4.1: Znázornenie nášho stromu. Príznak obsahuje viac prvkov pre rôzne typy vrcholov.

Trieda 10: GSHashNode

Parametre :

pos $= (x, y) \in \langle 0, 15\rangle \times \langle 0, 15\rangle \cap \mathbb{N}_0^2$

color $\in \langle 0, 255\rangle \cap \mathbb{N}_0$

Podmienka:

`hash` $(f, y, x) \leq$ color

4.3.3 Priemerná farba

Novým jednoduchým typom je `AvgNode`, ktorý porovnáva priemernú farbu pre konkrétny kanál odtlačku. Bol zvolený odtlačok okna namiesto celého okna, kvôli menšej výpočtovej zložitosti. Teda parametre sú kanál a farba. Priemernú farbu si vypočítame pri extrahovaní príznaku ako ďalší prvok. Neočakávame, že by tento typ vrchola mal sám o sebe priliš dobré výsledky. Tento dojem umocňuje fakt, že všetkých navzájom rôznych možných vrcholov tohoto typu je 256 + 256 + 180 = 692[7]. Pozitívne by však mohol pôsobiť napríklad pri malej váhe v distribúcii.

Funkcia `avg` (F) vracia farebný bod v HSV priestore reprezentujúci priemernú farbu okna príznaku F.

[7]Na porovnanie pri `HashNode` je to 256-krát viac, kvôli mriežke 16×16.

Trieda 11: HashNode

Parametre :
pos $= (x, y) \in \langle 0, 15\rangle \times \langle 0, 15\rangle \cap \mathbb{N}_0^2$
chan $\in \{H, S, V\}$
ak chan $= H$ tak color $\in \langle 0, 180\rangle \cap \mathbb{N}_0$ inak color $\in \langle 0, 255\rangle \cap \mathbb{N}_0$
Podmienka:
$pixel \leftarrow$ `hash(`f, y, x`)`
$pixel$[chan] $\leq$ color

Trieda 12: AvgNode

Parametre :
chan $\in \{H, S, V\}$
ak chan $= H$ tak color $\in \langle 0, 180\rangle \cap \mathbb{N}_0$ inak color $\in \langle 0, 255\rangle \cap \mathbb{N}_0$
Podmienka:
$avg \leftarrow$ `avg(`f`)`
avg[chan] $\leq$ color

4.3.4 Celý histogram

Ďalším typom vrchola, ktorý zavádzame, je `FullHistNode`. Tento uzol porovnáva šedotónový histogram okna. Parametrami vrchola sú náhodný histogram a prah. Aj náhodný histogram aj histogram príznaku sú normalizované tak, aby suma veľkostí košov bola rovná 1.

Náhodný šedotónový histogram konštruujeme nasledovne:

1. Náhodne vygenerujeme μ z normálneho rozdelenia.
2. Náhodne vygenerujeme σ^2 z normálneho rozdelenia.
3. Náhodne ťaháme stanovený počet hodnôt z normálneho rozdelenia $\mathcal{N}(\mu, \sigma^2)$.
4. Vypočitame histogram s BINS košmi z týchto hodnôt.
5. Histogram normalizujeme tak, aby suma všetkých BINS košíkov bola 1.

Tento algoritmus nemá žiadne formálne základy a je skôr heuristikou. Ako bolo spomenuté, je možné vytvoriť obrázky pre akýkoľvek histogram. Prečo teda nekonštruovať histogram vrchola úplne náhodne? Pri veľmi neformálnom skúmaní náhodnej vzorky niekoľkých stoviek obrázkov z testovacej sady UK-Bench sme nadobudli dojem, že väčšina z nich mala šedotónový histogram zbežne podobajúci sa na graf hustoty normálneho rozdelenia pravdepodobnosti. Zvyčajne mierne posunutý od stredu a taktiež strmosť bola rôznorodá, z toho dôvodu sú stredná hodnota aj odchýlka tiež znáhodnené. Taktiež sa ťahajú z normálneho rozdelenia, pretože bolo bežnejšie, že maximum bolo v strede histogramu, než na kraji. Obdobne pre odchýlku.

Bolo by vhodné pokúsiť sa nájsť formálnejší prístup a využiť prípadné existujúce výsledky počítačového videnia, avšak to už je nad rozsah tejto práce.

Trieda 13: FullHistNode

Parametre :
hist $\leftarrow$ `randHist()`
threshold $\in \langle 0, 1 \rangle$
Podmienka:
`compareHist(hist(`f`),` hist`)` $\leq$ threshold

Funkcia `randHist` vracia náhodný histogram vygenerovaný postupom spomenutým vyššie. Funkcia `compareHist(`h`,` h'`)` vypočíta nepodobnosť histogramov h a h' na škále $\langle 0, 1 \rangle$ podľa vzdialenosti prienik histogramov definovanej v 2.4.2 – čím sú podobnejšie, tým vyššia hodnota.

Prirodzene, očakávané bolo dosiahnutie lepších výsledkov aspoň pri nejakej váhe v distribúcii v kombinácii s ostatnými typmi. Vplyv má zjavne aj trieda možných histogramov vrchola. V každom prípade je však výpočtovo náročnejší ako pôvodné typy. Musí totiž počítať podobnosť histogramov, čo je náročnejšie ako jednoduché porovnanie farby.

Už prvotné testovanie ukázalo, že tento vrchol bol príliš výpočtovo a pamäťovo náročný a výsledkami neuspokojivý. Preto implementácia nebola úplne dokončená a pozornosť sa ďalej sústredila na ostatné nové typy.

4.3.5 Histogram

Druhým a hneď aj tretím typom porovnávajúcim na základe histogramu je `GSHistNode` a `HistNode`. Parametre pri oboch sú číslo koša a prah. Porovnáva sa veľkosť zvoleného koša histogramu príznaku s prahom.

Rozdiel medzi triedami je podobný ako medzi `GSHashNode` a `HashNode`. Vrchol typu `HistNode` sa pozerá na histogram príznaku v konkrétnom kanáli. Je viacero možností ako to implementovať, my sme použili tú, že horná hranica pre parameter bin je trikrát väčšia ako pri `GSHistNode` a hodnoty košov pre rôzne kanály idú jednoducho zasebou ako prvky poľa. Následne, do ktorého koša histogramu sa pozeráme určuje bin $mod\, 3$ a kanál $\lfloor \text{bin}/3 \rfloor$, pokiaľ koše aj kanály číslujeme od 0.

Trieda 14: GSHistNode

Parametre :
bin $\leftarrow \langle 0, \text{BINS} \rangle \cap \mathbb{N}_0$
threshold $\in \langle \text{TMIN}, \text{TMAX} \rangle \cap \mathbb{N}_0$
Podmienka:
$hist \leftarrow$ `hist(`f`)`
$hist[\text{bin}] \leq$ threshold

Konštanta BINS predstavuje počet košov histogramu. TMin a TMax v najlepšom prípade predstavujú výšku najmenšieho a najväčšieho bin-tého koša histogramu spomedzi všetkých príznakov vo vrchole. Trieda HistNode vyzerá identicky, až na to, že bin vyberáme z intervalu $\langle 0, 3 \times \mathsf{BINS})$, čím sa zároveň zabezpečí voľba náhodného kanála.

Takto formulovaný test vo vrchole je oveľa podobnejší pôvodnému vrcholu s odtlačkom ako `FullHistNode`. Samotné porovnanie je taktiež rýchle a jednoduché. Od tohoto typu očakávame vylepšenie výsledkov.

4.3.6 Farebný histogram

Posledným novým typov vrchola je `ColorHistNode`. Princíp je obdobný ako pre `HistNode`, ale s tým rozdielom, že porovnávame farebný histogram podľa definície 2.4.2 z príznaku na rozdiel od jednorozmerného histogramu. Parametre sú teda číslo koša a prah.

Trieda 15: ColorHistNode

Parametre :
$\mathsf{bin} \leftarrow \langle 0, \mathsf{BINS}^3) \cap \mathbb{N}_0$
$\mathsf{threshold} \in \langle \mathsf{TMIN}, \mathsf{TMAX} \rangle \cap \mathbb{N}_0$
Podmienka:
$hist \leftarrow \texttt{hist}(f)$
$hist[\mathsf{bin}] \leq \mathsf{threshold}$

Konštanta BINS predstavuje počet košov v jednom rozmere, teda jednom kanály.

Od tohoto typu, spolu s `HistNode`, očakávame najlepšie výsledky. TMin a TMax plnia rovnakú funkciu ako v triede 14.

Nebolo zatiaľ nikde spomenuté, akým spôsobom vytvárame histogramy z okien v príznakoch. Je tomu tak preto, že tých možností ako konštruovať histogram je viacero. Napríklad, okrem spočítania histogramu priamo z okna, vypočítať ho až z odtlačku okna. Vhodné je preto podporiť voľbu testovacími dátami. Zvolené možnosti aj testovanie sú popísané v odseku 5.2.1.

Dôležitým faktom, ktorý si treba uvedomiť, je to, že nepotrebujeme pri porovnaniach žiadne komplikované metriky ani spôsoby merania podobnosti. Jediná podobnosť, ktorú potrebujeme vedieť merať, je tá z pôvodnej schémy definovaná vo vzorci 3.4. A tá sa vôbec nepozerá na príznaky alebo iné údaje o oknách.

Trochu s nadhľadom povedané, podobnosť určujeme sériou náhodných jednoduchých testov, ktoré sami o sebe majú len veľmi obmedzenú výpovednú hodnotu. Ilustrujme to na príklade. Povedať o dvoch obrázkoch, že sú podobné, pretože dve okná náhodnej veľkosti z náhodnej pozície majú náhodný kôš histogramov menší ako náhodný prah, je odvážne

tvrdenie. Ale pokiaľ to platí pre veľa dvojíc okien dvoch obrázkov, už to neznie tak nezmyselne.

Práve jednoduchosť testov pri odtlačkoch bola inšpiráciou pre testy histogramov v náhodných košoch. Zjavne `FullHistNode` tento princíp nespĺňa a výsledky tohoto typu sú zlé. Dá sa očakávať, že aj iné príznaky používané v počítačovej grafike, ak sa dajú rozumne diskretizovať, môžu byť dobrým základom pre ďalšie typy vrcholov do takejto schémy.

4.4 Implementácia

Významnou súčasťou práce je implementácia. Tento odsek popisuje jej dôležité prvky, použité technológie a bližšie sa venuje niektorým podstatným častiam.

4.4.1 Technológie

Programovací jazyk, v ktorom je celá implementácia napísaná, je C++. Na tomto jazyku sa začalo pracovať v roku 1979 a dodnes je jedným z najpoužívanejších jazykov na písanie softvéru. Existuje niekoľko voľných aj komerčných kompilátorov pre veľké množstvo platforiem. Vznikol ako rozšírenie jazyka C o prvky objektovo-orientovaného programovania a mnohé ďalšie. Syntakticky nie je C++ priamou nadmnožinou C, ale rozdielov nie je veľa [14]. Silným argumentom pre použitie C aj C++ je veľké množstvo dostupných knižníc na uľahčenie práce, ktoré pokrývajú snáď každú oblasť informatiky. Ďalším typickým dôvodom pre použitie C či C++ je to, že programy v nich napísane sú veľmi rýchle oproti mnohým novším jazykom vyššej úrovne alebo jazykom interpretovaným. Je to dané popularnosťou aj vekom, čoho dôsledkom sú spomínané vysoko kvalitné a odladené kompilátory.

Dôvodom prečo uprednostniť C++ pred C sú už spomínané OOP prvky a taktiež Standard Template Library (STL). STL je sada knižníc pre C++ poskytujúca rôzne programovacie prvky, ktoré sú bežne potrebné. Medzi ne patria kontajnery, iterátory a rôzne algoritmy. Pri ich vhodnom použití sa dajú občas komplikované veci robiť veľmi jednoducho a elegantne.

Ďalším dôvodom pre voľbu C/C++ bola aj knižnica OpenCV. Toto je populárna multiplatformová a voľne dostupná knižnica, vyvíjaná spočiatku firmou Intel, zameraná na počítačové videnie. Poskytuje množstvo užitočných štruktúr a algoritmov pre prácu s obrázkami, ako aj algoritmy strojového učenia.

Implementácia má taktiež integrovaný webový server Mongoose, určený na využívanie webového rozhrania implementácie. Je populárny hlavne kvôli svojej malej veľkosti a rýchlosti. Taktiež je jednoduché ho ľahko integrovať do iného C/C++ programu, čo bol ďalší dôvod, prečo sme ho vybrali.

Všetky použité technológie sú FOSS, alebo existuje ich FOSS implementácia. FOSS je známy pojem zastrešujúci slobodný softvér s voľne dostupným zdrojovým kódom.

4.4.2 Architektúra

Implementácia sa snaží v rozumnej miere využívať princípy OOP aj schopnosti C++. Prehľad tried s krátkym popisom funkcionality je nasledovný:

Window Jednoduchá trieda pre okno. Obsahuje identifikátor okna a obrázka, veľkosť okna a pozíciu ľavého horného bodu okna v obrázku.

Windows Obsahuje mapu[8] pre okná, metódy na pridanie a nájdenie okna a metódy pre náhodne vygenerovanie množiny okien z obrázka aj množiny obrázkov.

Feature Zabezpečuje extraciu všetkých potrebných príznakov z okna.

Cache Táto trieda zabezpečuje jednoduchý a efektívny prístup k príznakom. Udržuje si zoznam príznakov. Inštancii môžeme nastaviť limit, koľko ich má maximálne držať v zozname. Prístup je transparentný, takže pokiaľ si vypýtame príznak nenachádzajúci sa v zozname, inštancia ho načíta, vloží do zoznamu a vráti. Pokiaľ pri tom prekročíme limit, odstránime zo zoznamu nadbytočné príznaky stratégiou FIFO.

Node Ako vyzerajú všetky triedy pre vrchol stromu je popísané v sekcii 4.3.

Tree Najdôležitejšími metódami stromu sú pridanie obrázka do stromu a vyhľadanie podobných obrázkov stromu pre vstup. Jeho parametrom je distribúcia a strom zabezpečuje jej dodržiavanie pri vytváraní nových vrcholov.

Forest Obsahuje zoznam stromov a metódy pre jednoduchšiu prácu s nimi a pre vyhľadávanie.

Storage Táto abstraktná trieda popisuje rozhranie pre perzistentné uloženie nutných dát. Jej bližší popis sa nachádza v ďalšom paragrafe.

FileStorage Je to trieda implementujúca rozhranie Storage, využívajúca textové súbory na ukladanie dát.

Ako bolo spomenuté, Storage predstavuje rozhranie pre perzistentné ukladanie a načítanie nutných dát. Pod týmito sa myslia informácie o oknách, uzloch, stromoch, lese atď. Zmysel má z viacerých dôvodov:

- Potenciálne výrazné zrýchlenie inicializácie v porovnaní s nutnosťou generovať všetky štruktúry pri každom spustení programu.
- Ľahšia manipulácia s vygenerovanými štruktúrami, napríklad pri prenášaní medzi rôznymi počítačmi.

[8] Mapa je v STL štandardne implementovaná ako binárny vyhľadávací strom.

- Výhody pri testovaní, kedy môže byť vhodné niektoré náhodne generované štruktúry mať "zafixované". Napríklad používaním rovnakého zoznamu okien pre všetky stromy znižujeme náchylnosť na ovplyvnenie výsledkov vyhľadávania pri testovaní rôznych distribúcií[9].

4.4.3 Webová služba

Už v úvode bolo spomenuté, že implementácia bude mať webové rozhranie na ľahké a intuitívne ovládanie. Webová stránka pre službu je veľmi jednoduchá a demonštruje možné použitie. Obsahuje textové políčko, kde užívateľ môže zadať číslo obrázku spomedzi trénovacích obrázkov. Pre tento obrázok sa následne vyhľadá najpodobnejších 10 výsledkov a sú užívateľovi prezentované formou tabuľky. Na každý nájdený obrázok sa dá kliknúť a vtedy sa za vstup zoberie on. Ukážka webového rozhrania je na obrázku 4.2.

Ide len o demonštračnú verziu a neobsahuje žiadne pokročilejšie funkcie. Užitočné by bolo napríklad mať možnosť voliť za vstup hocijaký obrázok buď z počítača alebo z webu. Toto však je len otázka doprogramovania načítania súboru cez HTTP a jeho uloženia na disk. Samotná schéma môže mať vstupný obrázok hocijaký (nielen z trénovacej množiny), takže je to len otázka vylepšenia užívateľského rozhrania.

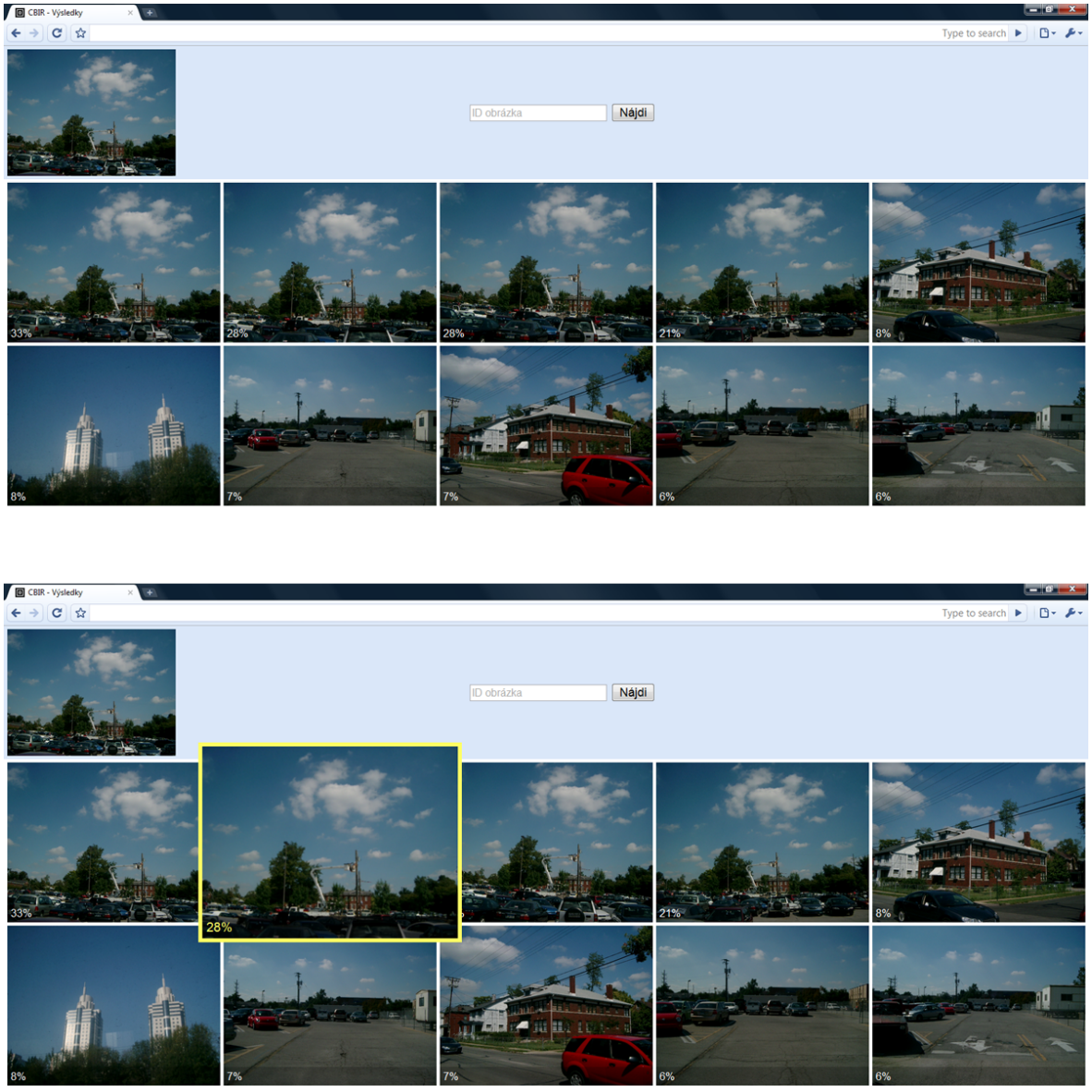
4.4.4 Paralelizácia

Kvôli rýchlejšiemu testovaniu na viacjadrovom počítači sme vytvorili aj funkcie zabezpečujúce viacvláknové predspracovanie trénovacích obrázkov, generovanie viacerých stromov súčasne a paralelné spracovanie testovacej sady.

Tieto priniesli solídne zrýchlenie príslušných procesov. Boli napísané až na záver implementácie a určite by sa dali ďalej vylepšovať.

Takáto dobrá paralelizácia by bola v praxi určite pri veľkom objeme dát veľmi žiadaná.

[9] Za predpokladu testovania počas viacerých spustení programu.



Obr. 4.2: Webové rozhranie. Hore: výsledky vyhľadávania. Dole: zvýraznenie výsledku pri prechádzaní myšou ponad neho.

Kapitola 5

Testovanie

Rovnako dôležité ako mať nápady je overiť, či ich podporujú fakty. Táto kapitola sa sústredí na testovacie výsledky implementácie pri rôznych konfiguráciách zvolených parametrov.

Začiatok kapitoly sa venuje objasneniu spôsobov testovania a popisom použitých testovacích sád. Následne predstavujeme jednotlivé vykonané typy testov.

Testovaniu je podrobená väčšina úprav a rozšírení spomínaných v kapitole 4. Najväčšiu pozornosť ale venujeme porovnaniu rôznych typov vrcholov na báze histogramov. Tieto testy sú základom pre hľadanie najlepšej distribúcie z pohľadu výsledkov vyhľadávania. Pokiaľ nie je uvedené inak, tak v tejto kapitole odkazovaním sa na "výsledky", či už dobré alebo zlé, myslíme výsledky testovania úspešnosti hľadania podobných obrázkov. Ako úspešnosť meriame a o akú testovaciu sadu ide by malo vyplývať z kontextu.

5.1 Spôsob testovania

Za základnú testovaciu sadu sme zvolili *UK-Bench*, čo je sada z Univerzity v Kentucky predstavená v [13], ktorá obsahuje 10200 obrázkov veľkosti 640×480 pixelov. Presnejšie, sada obsahuje 2550 štvoríc obrázkov, kde obrázky v rámci štvorice považujeme za podobné. Sú to obrázky objektov, rastlín, vecí z domácnosti, ľudí, exteriérov, atď.

Skóre počítame tak, že pre každý obrázok zo sady spočítame štyri najpodobnejšie z celej sady a za každý zo štvorice vstupného (vrátane samého seba), ktorý je medzi najpodobnejšími, pripočítame bod. Nakoniec predelíme počet bodov veľkosťou sady, teda 10200. Takto dostaneme skóre medzi 0 a 4.

Autori sady uvádzajú[1], že metódy využívajúce to najlepšie čo sa v súčasnosti používa, kam patrí algoritmus na detekciu zaujímavých regiónov MSER či SIFT na hľadanie a popis lokálnych príznakov, v kombinácií s k-means zhlukovaním a rôznymi hodnotiacimi schémami, dosahujú v priemere skóre na úrovni od 3.07 do 3.29. V [15] z roku 2007 dosiahli skóre 3.45 využitím randomizovaných k-d stromov. Autori pôvodnej práce dosiahli v [1]

[1] Zdroj: http://vis.uky.edu/štewe/ukbench/data/

použitím 10 stromov pre $F_I = 1000 a n_{min} = 4$ skóre 3.01.

Testy boli robené pri fixovanom inicializátore generátora pseudonáhodných čísel a zvyčajne v rámci jedného testu na rovnakých oknách. Kvôli rýchlosti boli testy zväčša robené paralelne na viacerých vláknach, čo prirodzene mohlo mať vplyv na náhodnosť. Avšak viacnásobným skúšaním rôznych testov sme nedostávali väčší rozptyl ako 3 stotiny bodu už pri vzorke 500 obrázkov.

Nie vždy sme použili na testovanie celú sadu. Pokiaľ chceme porovnať dve sady parametrov, stačí nám zobrať rozumne veľkú časť sady a merania spraviť len na nej. Testy by pre väčšie časti sady (resp. pre celú) dosiahli síce rozdielne hodnoty, ale poradie by veľmi pravdepodobne zostalo zachované. Samozrejme, ak sú rozdiely medzi výsledkami meraní minimálne, tak to nemusí platiť.

Všetky testy, ak nie je uvedené inak, boli robené pre $T = 10$, $n_{min} = 4$ a $F_I = F_Q = 1000$.

5.2 Histogramy

Vrcholy založené na testoch histogramov tvorili najvýznamnejšiu časť predchádzajúcej kapitoly a preto im je venovaná najväčšia pozornosť aj pri testovaní.

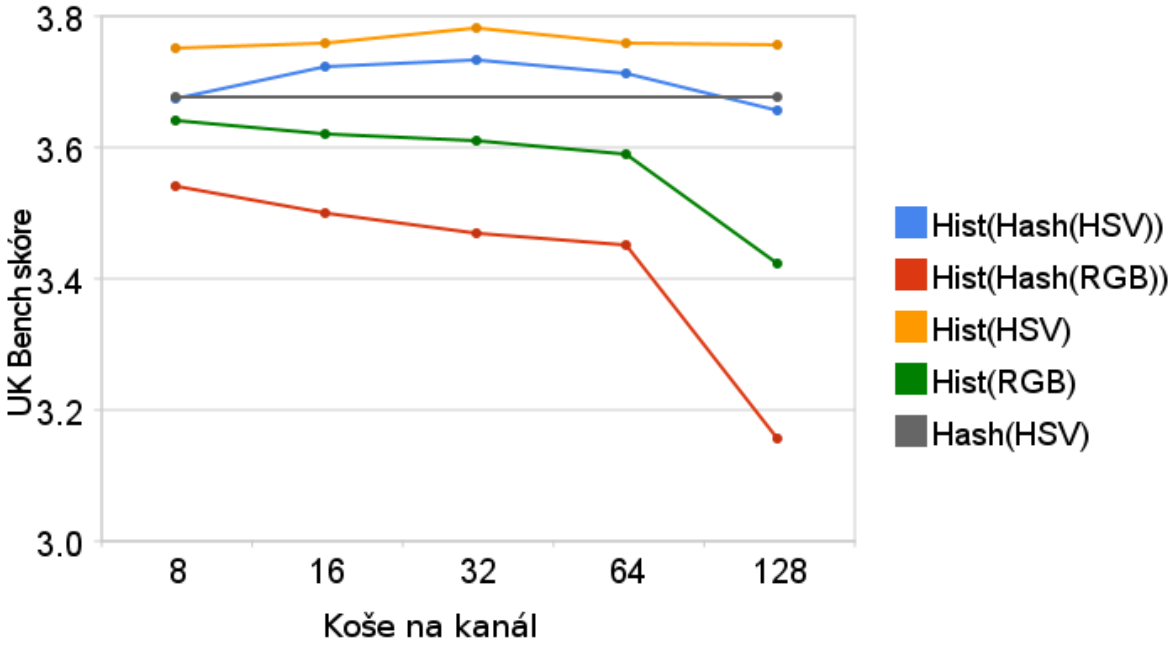
5.2.1 Parametre histogramu

Ako bolo spomenuté, možností ako z okna vytvárať histogramy je viacero. Zvolili sme štyri, ktoré boli podrobené meraniam.

- $Hist(Hash(HSV))$ - Histogram vytvárame z odtlačku okna reprezentovaného v HSV farebnom priestore.
- $Hist(Hash(RGB))$ - Vytvoríme ho z odtlačku okna v RGB.
- $Hist(HSV)$ - Histogram vytvoríme zo všetkých bodov okna v HSV.
- $Hist(RGB)$ - Zo všetkých bodov okna v RGB.

Odtlačok má rozmery 16×16 bodov.

Testovanie sme robili len na podmnožine UK-Bench o veľkosti prvých 500 obrázkov. Lesy sme generovali hromadnou metódou. V meraniach sme sa zaujímali len o výsledné skóre a nie o ďalšie parametre ako čas extrakcie či vyhľadávania. Taktiež sme pre všetky štyri spomenuté spôsoby vytvárania testovali viacero hodnôt počtu košov histogramu. V grafe 5.1 sú výsledky pre 500.000 príznakov a stromy boli vytvárane iba z `HistNode`. Pre referenciu je uvedený aj výsledok pri rovnakých parametroch pre stromy z `HashNode`.



Obr. 5.1: `HistNode`; 500.000 príznakov, T = 10, n_{min} = 4

Výsledky naznačujú niekoľko vecí. V prvom rade, že testy pre obrázky v RGB dosahujú konzistentne horšie výsledky ako tie pre HSV. Ďalej to, že $Hist(Hash(HSV))$ dopadol pre 16, 32 aj 68 košov na kanál lepšie ako $Hash(HSV)$. Najlepšie vo všetkých meraniach dopadol $Hist(HSV)$ a malé odchýlky vo výsledkoch pre rôzne merania naznačujú, že pre túto veľkosť sady dosiahol maximum svojej rozlišovacej schopnosti.

Na základe týchto aj ďalších meraní sme sa rozhodli ďalej používať prvú možnosť, teda $Hist(Hash(HSV))$, s 32 košmi na kanál. Aj keď $Hist(HSV)$ dosahoval v priemere o niekoľko stotín lepšie výsledky, rozdiel v časoch extrakcie bol pre väčšie sady badateľný. Taktiež je dobré, že pre fixovanú veľkosť odtlačku vieme dopredu povedať koľko najviac bodov sa môže ocitnúť v jednom koši, bez ohľadu na veľkosť obrázku resp. okna. Tento nedostatok by sa pre $Hist(HSV)$ dal odstrániť normovaním histogramu.

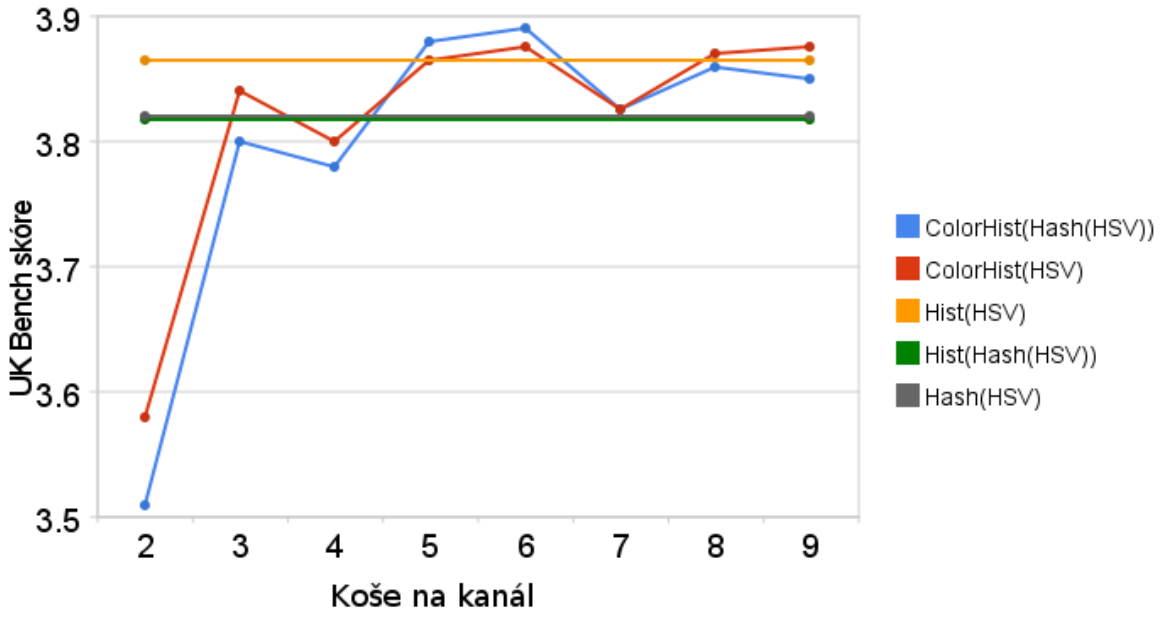
5.2.2 Parametre farebného histogramu

Druhým testom, podobného charakteru ako predchádzajuci, je testovanie spôsobov vytvárania farebného histogramu z okien.

- $ColorHist(Hash(HSV))$ - Farebný histogram vytvárame z odtlačku okna reprezentovaného v HSV farebnom priestore.
- $ColorHist(HSV)$ - Farebný histogram vytvoríme zo všetkých bodov okna v HSV.

Pre niekoľko hodnôt sme testovali aj spôsoby s RGB obrázkami, avšak výsledky boli podobného charakteru ako pri prvom teste a preto sme v tom ďalej nepokračovali. Zvo-

lili sme tentoraz podmnožinu o veľkosti 200 obrázkov. Ostatné parametre ostali zachované. Merania sme robili znova pre viacero hodnôt počtu košov na kanál. Výsledky sú v grafe 5.2.



Obr. 5.2: `ColorHistNode`; 200.000 príznakov, T = 10, n_{min} = 4

Z grafov je zjavné, že výsledky sú veľmi podobné, aj keď $ColorHist(Hash(HSV))$ v najlepších meraniach dosiahol o málo lepšie skóre. Maximum dosiahli oba pre 6 košov na kanál, teda $6^3 = 216$ košov pre celý farebný histogram, avšak len o minimálny rozdiel oproti meraniam pre 5 košov (teda 125 spolu). Za zmienku tiež stojí, že obe metódy dosiahli pre 6 košov lepšie skóre ako $Hist(HSV)$.

Zvyšovanie počtu košov má za následok spomalenie všetkých fáz - predspracovania obrázkov, generovania lesa aj vyhľadávania. Najviac sa to prejavuje na trvaní vyhľadávania.

Uskutočnili sme ešte ďalšie merania zamerané jednak na presnosť, ale taktiež na časové nároky. Skúšali sme porovnávať odtlačky 5 veľkostí, od 16×16 po 64×64 a usúdili sme, že najlepší pomer presnosti a času poskytuje odtlačok rozmerov 32×32 bodov, z ktorého vytvárame farebný histogram s 5 košmi na kanál.

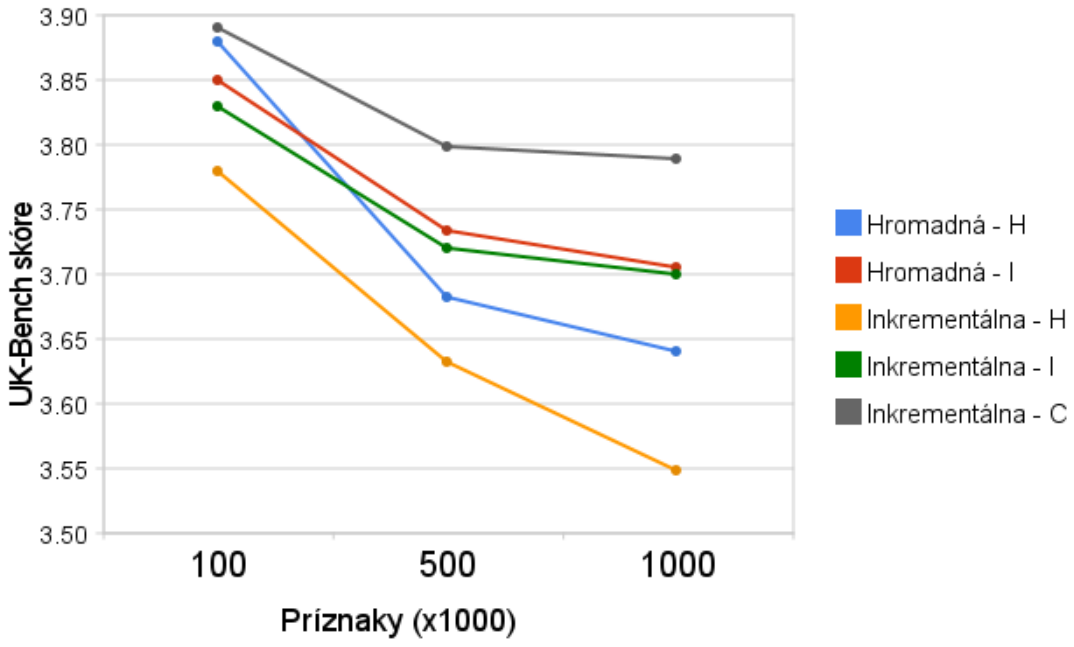
Negatívne je, že typ `ColorHistNode` je celkovo výrazne pomalší než ostatné typy. Zvyšovaním počtu príznakov sa tento rozdiel ďalej prehlbuje, pričom pri použití celej sady UK-Bench pri $F_I = 1000$ (teda 10.2×10^6 príznakov) sa stane tento typ nepoužiteľným pri generovaní hromadnou metódou. Dôvodov, prečo tomu tak je, môže byť viacero – od neefektívnej implementácie vytvárania farebných histogramov až po príliš naivné generovanie náhodných parametrov vo vrchole. Analýza tohoto problému by bola dobrým námetom na ďalší rozvoj práce, pretože ako sa ukazuje v sekcii 5.3, výsledky presnosti pre

`ColorHistNode` sú veľmi povzbudzujúce. Dá sa očakávať, že tento typ vrchola sám alebo v kombinácii s inými typmi môže dosiahnuť na UK-Bench veľmi zaujímavé výsledky. Taktiež by mohlo byť zaujímavé skúsiť rôzne počty košov histogramu pre jednotlivé kanály.

5.3 Porovnanie metód

V sekcii 4.1 sme predstavili niekoľko metód na generovanie lesa, spolu s očakávanými vlastnosťami. Výsledky testovacích meraní porovnávajúcich hromadnú a inkrementálnu metódu sú v grafe 5.3.

Testovali sme viacero variantov: pre hromadnú metódu les z `HashNode` a les z `HistNode`. Pre inkrementálnu metódu les z `HashNode`, `HistNode` aj `ColorHistNode`. Skúšali sme aj hromadnou metódou generovať lesy z `ColorHistNode`, avšak už pri 100.000 príznakoch trvalo generovanie aj vyhľadávanie dvakrát dlhšie, pričom ďalej tento rozdiel ešte rástol. Tým pádom nemalo zmysel ďalej tento variant testovať, keďže pre nás nemal praktické využitie. Pri inkrementálnej metóde sa problémy s `ColorHistNode` nevyskytovali.



Obr. 5.3: Porovnanie hromadnej a inkrementálnej metódy; T = 10, n_{min} = 4, H = HashNode, I = HistNode, C = ColorHistNode

Merania ukazujú niekoľko zaujímavostí. V prvom rade jednoznačne najlepšie dopadli merania pre les z `ColorHistNode` generovaný inkrementálnou metódou, pričom čas generovania bol 430 sekúnd, čo v porovnaní s 320 sekundami pre `HistNode` a 350 sekundami pre `HashNode` (obe inkrementálnou metódou) nie je až taký zlý výsledok.

Druhým záverom je pozorovanie, že inkrementálna metóda generuje stromy rýchlejšie než hromadná. Konkrétny faktor závisí od parametrov, ale pri stromoch z `HistNode` a celej sade ide o 9-násobné zrýchlenie.

Ďalej je zaujímavý fakt, že rozdiel vo výsledkoch v metódach pre `HashNode` bol pomerne výrazný (0.1 bodu), ale rozdiel pre `HistNode` bol takmer zanedbateľný (0.01 bodu). Keď zoberiem do úvahy, že inkrementálna metóda generuje stromy rýchlejšie, ponúka sa otázka, či radšej nevyužiť túto pri generovaní stromov z `HistNode`.

Námetom na ďalšie skúmanie je otestovať metódu malých lesov pre rôzne veľkosti T_{set} a N_{set} a porovnať ju s ostatnými metódami.

5.4 Heterogénne a homogénne stromy

Ďalším testom je porovnanie homogénnych a heterogénnych stromov. Merali sme skóre pre fixovaný pomer vrcholov. Testované štyri varianty generovania sú:

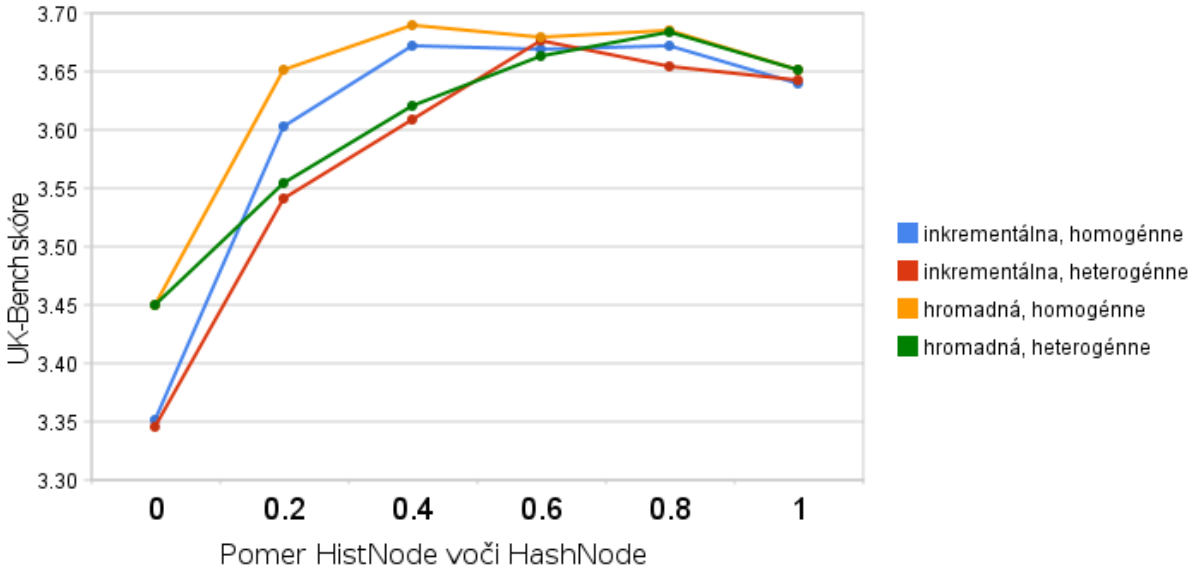
- homogénne stromy inkrementálnou metódou
- heterogénne stromy inkrementálnou metódou
- homogénne stromy hromadnou metódou
- heterogénne stromy hromadnou metódou

Meranie sme robili na vzorke 1000 obrázkov pre $F_I = 200$. Generovali sme 10 stromov. Pri homogénnych variantoch horizontálna os predstavuje pomer stromov z `HistNode`. Zvyšné stromy sú `HashNode`. Pri heterogénnych variantoch predstavuje horizontálna os váhu `HistNode` v distribúcii každého stromu. Zvyšná váha je nastavená pre `HashNode`. Merania sú v grafe 5.4. Homogénne stromy dopadli lepšie pre obe metódy generovania.

5.5 Úspešnosť na UK-Bench

Prirodzene, zaujímavou otázkou je, aké najlepšie skóre na celej sade UK-Bench vieme dosiahnúť. Problém je, že toľko parametrov má vplyv na výsledok, že by bolo treba značné výpočtové aj časové kapacity, aby sme mohli vykonať rigorózne testovanie pre ich rôzne kombinácie.

Na ilustráciu, už pre jeden strom máme množstvo parametrov ovplyvňujúcich jeho rozpoznávacie schopnosti. Treba zvoliť typy vrcholov, ktoré budú v strome. Treba im priradiť váhy v distribúcii. Už pre dve typy vrcholov by bolo vhodné vyskúšať aspoň 11



Obr. 5.4: Porovnanie homogénnych a heterogénnych stromov. T = 10, F_I = 200, 1000 obrázkov

variantov distribúcie (váhu by sme posúvali po 0.1 z jedného na druhý typ), aby sme mali aspoň odhad najlepšej hodnoty. Ďalej sú to vnútorné parametre typov vrcholov na spôsob tých, ktoré určujeme v sekciách 5.2.1 a 5.2.2 pre `HistNode` resp. `ColorHistNode`. Ďalším parametrom je metóda generovania stromu a vplyv má aj voľba n_{min}. Následne môžeme vytvárať lesy z kombinácií ľubovolných stromov.

Kvôli obmedzenej výpočtovej kapacite sme si nemohli dovoliť prehľadávať celý priestor kombinácií parametrov. Voľbu parametrov sme robili na základe predchádzajúcich meraní a snažili sme sa intuitívne odhadnúť ako parametre upraviť, aby sme dostali lepšie výsledky.

Najzaujímavejšie výsledky sa nachádzajú v tabuľke 5.1 a vizualizované sú v grafe 5.5. Číslo pod stĺpcom v grafe určuje riadok v tabuľke. Ak nie je uvedené inak, použité hodnoty pre parametre boli $n_{min} = 4$, $F_I = 1000$ a odtlačky majú také veľkosti, aké uvádzame v príslušných sekciách.

Kvôli čitateľnosti používame v tabuľke skrátený formát zápisu rôznych parametrov. Les môže byť zapísaný pomocou viacerých sád a všeobecná formula pre jednu sadu je `M` − `tV.w`[+`V′.w′`], kde:

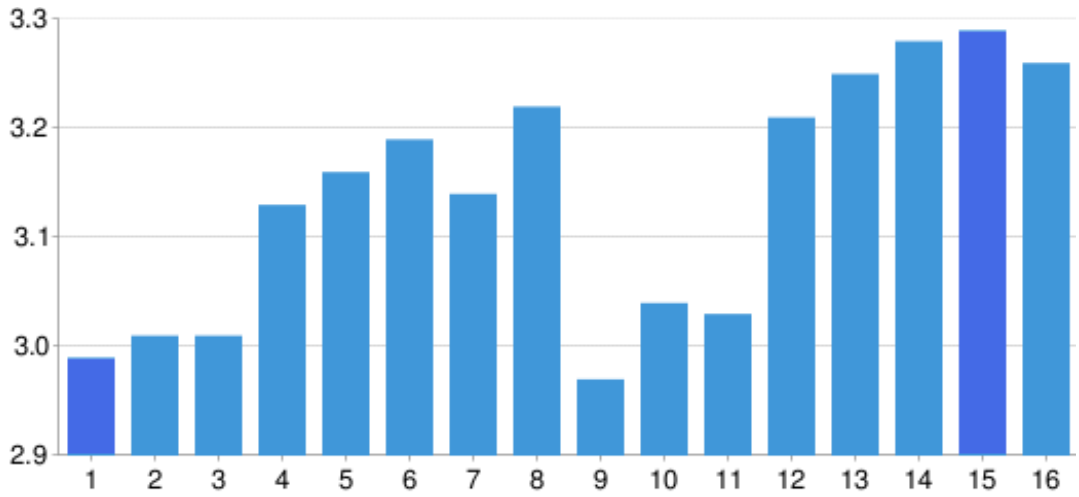
- `M` je použitá metóda pre sadu, buď `b` pre hromadnú (z ang. *bulk*) alebo `i` pre inkrementálnu (z ang. *incremental*).
- `t` je počet stromov v sade.
- `V` je typ vrchola; `H` = `HashNode`, `I` = `HistNode`, `C` = `ColorHistNode`.
- `w` je váha typu `V`, ak nie je uvedená, tak je 1.

- +V′.w′ popisuje váhu ďalšieho typu vrcholu v tej istej sade, ak treba.

Viaceré sady sú oddelené čiarkou. Napríklad zápis `i-4I, b-6H.8+I.2` predstavuje les, v ktorom sú 4 stromy z `HistNode` vytvorené inkrementálnou metódou a 6 stromov z `HashNode` s váhou 0.8 a `HistNode` s váhou 0.2 vytvorených hromadnou metódou. Formát sme nechceli ďalej komplikovať a preto sú v tabuľke ďalšie parametre, ak nemajú pôvodnú hodnotu, uvedené v poznámke.

#	Skóre	Kód	Poznámky
1	2.99	`b-10H`	
2	3.01	`b-1I`	
3	3.01	`b-1I`	odtlačok 32×32
4	3.13	`b-2I`	odtlačok 32×32
5	3.16	`b-3I`	odtlačok 32×32
6	3.19	`b-4I`	odtlačok 32×32
7	3.14	`b-4H.5+I.5`	odtlačok 32×32 pri `I`
8	3.22	`b-10I`	
9	2.97	`i-1I`	$n_{min} = 3$
10	3.04	`i-1I`	
11	3.03	`i-1I`	$n_{min} = 5$
12	3.21	`i-1C`	
13	3.25	`i-1C`	$n_{min} = 5$, odtlačok 24×24
14	3.28	`b-5H, b-5I`	
15	3.29	`b-5H, i-5I`	
16	3.26	`b-3H, i-5I, b-2I`	odtlačky veľkosti 24×24 pri `b-2I`

Tabuľka 5.1: Výsledky rôznych meraní na UK-Bench.



Obr. 5.5: Výsledky rôznych meraní na UK-Bench.

Prvé meranie je referenčné pre porovnanie s výsledkom z [1], kde s `b-10H` (desať stromov a vrcholy porovnávajúce odtlačky, hromadná metóda) dosiahli skóre 3.01. My sme dosiahli 2.99, pričom odchýlka sa dá vysvetliť náhodnosťou spojenou s generovaním.

Zaujímavých výsledkov je niekoľko. Za zmienku stojí hneď druhé meranie, kedy jeden strom z `HistNode` vytvorený hromadnou metódou dosiahol skóre 3.01. Pre $n_{min} = 4$ má strom z `HistNode` aj strom z `HashNode` pre celú sadu zhruba rovnaké množstvo vrcholov a to 11×10^6. Vyhľadávanie v týchto stromoch trvá tiež porovnateľne dlho, keďže oba typy vrcholov len porovnávajú dve čísla. Extrakcia príznakov z obrázku pre stromy z `HistNode` trvá zhruba o dve tretiny dlhšie než pre stromy z `HashNode`, keďže v oboch prípadoch extrahujem odtlačky rovnakej veľkosti, ale v prvom prípade musím extrahovať aj histogram. Z toho vyplýva, že rovnakú presnosť vieme dosiahnúť s 10-násobne menšími pamäťovými aj časovými nárokmi za cenu mierneho predĺženia času extrakcie príznakov.

Desiate meranie dosiahlo oproti druhému zlepšenie o 0.03 bodu. Toto nie je nijak dramatický nárast, ale zaujímavé to je z toho dôvodu, že desiate meranie prebehlo na jednom strome z `HistNode` vygenerovanom inkrementálnou metódou. Teda, inkremetnálna metóda dosiahla v tomto prípade lepšiu presnosť ako hromadná, čo je opak toho čo sme očakávali. Ďalej, inkrementálna metóda generuje stromy z `HistNode` zhruba $9\times$ rýchlejšie než hromadná a stromy majú pri vytváraní z celej sady o 8% menej vrcholov a zaberajú tým pádom menej miesta v pamäti. Pri stromoch z `HashNode` nevznikla situácia, že by inkrementálna metóda dosiahla rovnaké alebo lepšie skóre než hromadná.

Trináste meranie dosiahlo skóre 3.25, čo je už dosť výrazné zlepšenie oproti pôvodnej schéme. Toto meranie pritom prebehlo na jedinom strome vytvorenom z `ColorHistNode` inkrementálnou metódou pre $n_min = 5$ a odtlačky veľkosti 24×24 bodov. Generovanie stromu a extrakcia príznakov trvá spolu zhruba $10\times$ dlhšie než pri `i-1I`. Avšak ešte závažnejší problém je, že vyhľadávania trvá mnohonásobne dlhšie. Spočítanie skóre, teda vyhľadanie pre celú sadu, trvalo takmer 5 hodín v porovnaní so zhruba 5 minútami pre `i-1I`. Ďalej nás to utvrdzuje v presvedčení, že práve `ColorHistNode` je zaujímavým kandidátom na ďalšie skúmanie.

Najlepší výsledok sme dosiahli v meraní č. 15. Dosiahnuté skóre je 3.29, čo je na úrovni výsledkov najpokročilejších schém, s výnimkou [15], kde sa podarilo dosiahnúť skóre 3.45. V každom prípade je to zlepšenie o 0.28 bodu oproti pôvodnej schéme, čo predstavuje nárast o 9% relatívne k ich výsledku. Dá sa očakávať, že vytrvalejšie hľadanie pomerov stromov a nastavení parametrov by viedlo k ďalšiemu zvýšeniu presnosti.

5.6 Výkon

Merania prebiehali na počítači s 24 GB RAM a dvoma procesormi, každý so štyrmi jadrami taktovanými na 2.3 GHz.

Aj keď sú vrcholy stromov veľmi jednoduché a pamäťovo nenáročné objekty, treba si uvedomiť, že pamäte na celú sadu UK-Bench treba dosť a stromy sú veľké. Presnejšie, keďže sme na 64-bitovej architektúre, zaberá každý smerník v pamäti 8 bajtov. V našej implementácii má každý vrchol minimálne smerník na otca, do stromu, na ľavého

a pravého syna. Taktiež má zoznam svojich príznakov, ktorý implementujeme ako objekt triedy std::List a ten, aj keď je prázdny, zaberá veľkosť troch smerníkov. Ďalej má vrchol svoj identifikátor, čo je číslo typu long, ktorý má zvyčajne tiež veľkosť 8 B. Keď to spočítame, zaberá jeden vrchol bez parametrov 64 B.

Strom pri $n_min = 4$ mal pre `HistNode` aj `HashNode` v priemere 11×10^6 vrcholov. Dostávame, že jeden strom zaberá zhruba 670 MB pamäte, čo je veľa. Pre vyhľadávanie nie je nutné mať smerník ani na otca, ani do stromu, ani si držať prázdny zoznam pre vnútorné vrcholy. Po takejto redukcii by strom zaberal zhruba 250 MB, čo už je prijateľnejšie. V implementácii sme si tieto navyše dáta udržiavali z dôvodov testovania a niektorých funkcií ako ukladanie do súborov. Preto sme mali častokrát aj pri 24 GB RAM problém s nedostatkom pamäte pri generovaní väčšieho počtu stromov súčasne. V implementácii pre prax by bolo nutné pamäťové nároky čo najviac skresať.

Kapitola 6

Záver

Problematika CBIR je veľmi zaujímavá z niekoľkých dôvodov. Stojí na prieniku viacerých informatických smerov. Má širšie využitie, než sa na prvý pohľad zdá. A ešte zďaleka nie je vyriešená.

V práci vychádzame zo schémy, ktorá je pomerne jednoduchá a má pekné vlastnosti. Nedosahuje úrovne najlepších existujúcich CBIR, ale aj pri svojej konceptuálnej jednoduchosti dosahuje zaujímavé výsledky.

Po úvode do problematiky a opise spomínanej schémy predstavujeme niekoľko návrhov, ako potenciálne vylepšiť schémy v rôznych oblastiach. Sú to metódy generovania náhodných stromov a popis ich vlastností. Ďalej navrhujeme spôsob generalizácie vrchola stromu tak, aby bolo možné využiť viaceré príznaky obrázkov a nielen odtlačky ako tomu je pri pôvodnej schéme. Zavádzame pravdepodobnostnú distribúciu vrcholov ako jemnozrnný nástroj na kombinovanie rôznych typov vrcholov v rámci stromov, pričom sa pridŕžame randomizovanej nátury schémy. Ďalej predstavujeme niekoľko nových typov vrcholov, využívajúcich ďaľšie používané príznaky, ktorými sú hlavne histogramy.

Takúto novú resp. rozšírenú schému implementujeme spolu s webovým rozhraním na pohodlné vyhľadávanie. Podrobujeme testovaniu nové metódy a nové typy vrcholov. Testovanie uskutočňujeme na sade UK-Bench.

Experimentálne podporujeme alebo vyvraciame očakávania ohľadom metód generovania stromov. Pomocou meraní hľadáme najlepšie nastavenia vnútorných parametrov pre jednotlivé nové typy vrcholov.

Ukazujeme, že naša schéma predčí výsledky pôvodnej pri viac než $10\times$ menšej pamäťovej zložitosti a obdobne rýchlejšom generovaní za cenu mierneho spomalenia extrakcie príznakov.

Dosahujeme na UK-Bench úspešnosť 82.25%, čo je oproti pôvodnej schéme zlepšenie o viac než 9%, pri porovnateľnej zložitosti. Je to výsledok na úrovni najlepších CBIR schém súčasnosti, aj keď existuje ešte lepší výsledok.

Ciele práce sa podarilo naplniť, ale vyskytli sa aj otázky a problémy, ktoré už boli nad rámec práce. Medzi ne určite patrí typ vrcholu pre farebný histogram, ktorý aj pri sľubných

výsledkoch nebolo možné plne otestovať. Prirodzené je taktiež vytvorenie ďalších typov vrcholov využívajúcich iné príznaky. Zaujímavé by mohlo byť aj využitie tejto schémy na automatickú anotáciu obrázku.

Dovolíme si tvrdiť, že generické CBIR vyhľadávače ešte nie sú na takej úrovni, aby sa ich potenciál dal plne využiť a stali sa bežnou pomôckou pre ľudí. Predchádza tomu vyriešenie náročných problémov, ale práve tie sú na informatike najzaujímavejšie a preto bude zaujímavé sledovať, aká budúcnosť čaká CBIR.

Literatúra

[1] Raphaël Marée, Pierre Geurts, and Louis Wehenkel. Content-based image retrieval by indexing random subwindows with randomized trees. *IPSJ Transactions on Computer Vision and Applications (open-access)*, 1(1):46–57, jan 2009. Extended version of ACCV 2007 paper.

[2] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Z. Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys*, 40(2):1–60, 2008.

[3] James Z. Wang, Jia Li, and Gio Wiederhold. Simplicity: Semantics-sensitive integrated matching for picture libraries. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 23(9):947–963, 2001.

[4] A. Georgakis, L. H. Sun, R. Cabral, and H. Li. Www.wawo.net: A cbir for facial similarities. *Proc. of Swedish Society for Automated Image Analysis (SSBA'07)*, pages 125–128, March 2007.

[5] Jing Li, Nigel Allinson, Dacheng Tao, and Xuelong Li. Multitraining support vector machine for image retrieval. *IEEE Transactions on Image Processing*, 15(11):3597–3601, 2006.

[6] Zhong Wu, Qifa Ke, Michael Isard, and Jian Sun. Bundling features for large scale partial-duplicateweb image search. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2009.

[7] Mathias Lux. Caliph & emir: Mpeg-7 photo annotation and retrieval. In *MM '09: Proceedings of the seventeen ACM international conference on Multimedia*, pages 925–926, New York, NY, USA, 2009. ACM.

[8] Jorma Laaksonen, Markus Koskela, and Erkki Oja. Picsom-self-organizing image retrieval with mpeg-7 content descriptors. *IEEE Transactions on Neural Networks*, 13(4):841–853, 2002.

[9] Xiang Sean Zhou and Thomas S. Huang. Relevance feedback in image retrieval: A comprehensive review. 2003.

[10] Thomas Deselaers, Daniel Keysers, and Hermann Ney. Discriminative training for object recognition using image patches. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, pages 157–162, June 2005.

[11] Raphaël Marée, Pierre Geurts, Justus Piater, and Louis Wehenkel. Random subwindows for robust image classification, June 2005.

[12] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[13] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, pages 2161–2168, June 2006. **oral presentation**.

[14] Bjarne Stroustrup. Sibling rivalry: C and c++. Technical report, AT&T Labs, January 2002.

[15] James Philbin, Ondrej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman. Object retrieval with large vocabularies and fast spatial matching, June 2007.

[16] Raphaël Marée, Pierre Geurts, and Louis Wehenkel. Content-based image retrieval by indexing random subwindows with randomized trees, Nov 2007.

[17] Thomas Deselaers. Features for image retrieval. Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen, 2003.