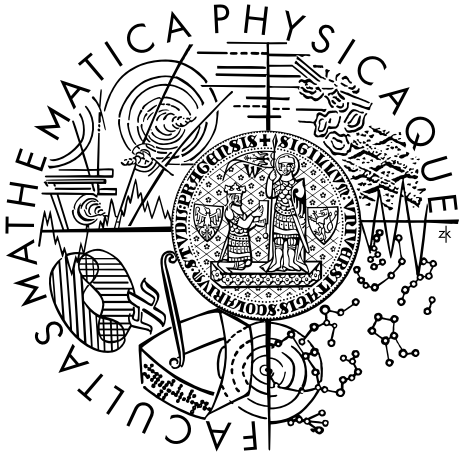
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Jaroslav Kubát

Databáze otisků prstů

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Martin Babka

Studijní program: Informatika (N1801)
Studijní obor: Softwarové systémy

Praha 2015

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

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

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

Název práce: Databáze otisků prstů

Autor: Bc. Jaroslav Kubát

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Martin Babka, Katedra teoretické informatiky a matematické logiky

Abstrakt: Popis implementace aplikace, která slouží pro správu databáze otisků prstů, je hlavní náplní této diplomové práce. Text je členěn podle funkčních bloků programu. Jednotlivými fázemi jsou zpracování vstupního otisku, jeho analýza k získání rozdílových znaků, převod znaků do specifické datové reprezentace, popis databáze, indexu a jejich operací. Práce na několika místech obsahuje dvojí způsob provedení dané operace, každý z nich je popsán a porovnán se svým protějškem. Na konci práce je také sepsán uživatelský manuál popisující základní scénáře použití programu.

Klíčová slova: otisk prstu, databáze, index, analýza otisku prstu

Title: A fingerpring database

Author: Bc. Jaroslav Kubát

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Martin Babka, Department of Theoretical Computer Science and Mathematical Logic

Abstract: The main goal of this thesis is to provide full description of an implementation of an application that is supposed to be a complex dactyloscopic database tool. The text is divided into parts according to the functional blocks of the application. The parts are namely processing of input fingerprint, the minutiae point analysis, the description of minutiae representation, the database and index structure described with its operations. Usually we describe multiple algorithms for a single task. In this case we provide their comparison. At the end of this thesis, there is a user manual describing the main usage scenarios.

Keywords: fingerprint, database, index, fingerprint analysis

Děkuji svému vedoucímu RNDr. Martinu Babkovi za ochotu a trpělivost a také organizátorům projektu FVC-onGoing za poskytnutí testovacích dat.

Jaroslav Kubát

Obsah

Úvod

Tato diplomová práce představuje průřez problematikou jedné z disciplín biometrie - daktyloskopie - z pohledu informačních technologií.

Daktyloskopie je obor zabývající se identifikací člověka podle otisku jeho prstů. Neomezuje se sice jen na prsty, zabývá se i kresbou na dlaních či ploskách nohou, ale právě prsty jsou nejčastějším nástrojem ověření identity. Využití tato disciplína našla zejména v kriminalistice, kde zaujala neochvějné místo při vyšetřování trestných činů. Avšak i komerční sféra otisky prstů využívá. Čtečky nalezneme na mnoha osobních počítačích, mohou se vyskytovat u kontroly na letištích nebo v soukromých firmách jako zámky oddělení s omezeným přístupem. Využití poznatků daktyloskopie je tedy široké. Z pohledu informatiky ale přináší mnoho nelehkých výzev - od vývoje snímání otisků přes efektivní analýzu rozdílových znaků až po jejich uchovávání a porovnávání.

Důležitým aspektem této práce je také skutečnost, že se snaží přinést zpět do popředí problematiku, která se stále hlouběji ztrácí v komerčním prostředí. Dostupné open-source řešení, které by se alespoň zčásti podobalo naší aplikaci není. V akademickém poli se pohybuje jen málo ústavů orientovaných na daktyloskopii. Navíc si své poznatky pečlivě chrání a zveřejňují jen základní návrhy postupů. Proto jsme se snažili vytvořit přehled, který by čtenáři pomohl se v této oblasti zorientovat.

Aplikace *DaktylDB*, kterou práce detailně popisuje, je pokusem o vytvoření komplexního programu umožňujícího zpracovávat otisky prstů poskytnuté ve formě obrazových dat, tyto otisky analyzovat, uložit, porovnat a to vše efektivně. Hlavními body tedy jsou zpracování vstupu za účelem získání informací o bodech sloužících k odlišení dvou otisků, analýza těchto bodů a jejich převod do snadno porovnatelné formy a nakonec uložení otisků do databáze s primárním úkolem poskytnout v krátkém čase spolehlivý výsledek při vyhledávání pomocí podobného otisku. Popisy všech algoritmů a postupů jsou v textu doprovázeny ilustračními obrázky.

Text práce je, vyjma první a poslední kapitoly, rozdělen do kapitol podle jednotlivých funkčních bloků - zpracování vstupního otisku, detekce významných bodů, jejich datová reprezentace, organizace databáze a databázový index. První kapitola se snaží uvést čtenáře do problematiky daktyloskopie a poslední je podrobnějším popisem implementace programu.

Závěrem práce uvádíme zhodnocení dosažených výsledků, které jsou zpracovány do přehledných tabulek, v porovnání s výsledky uvedenými v odborné literatuře.

1. Daktyloskopie

Daktyloskopie (nebo také dermatoglyfika) je jednou z částí oboru biometrie. Ta se zabývá rozlišováním osob pomocí specifických znaků lidského těla. Může zkoumat například tvar a rozložení částí obličeje, mapy žil na rukou, charakteristiky oční duhovky či sítnice a mnoho dalšího. Samotná daktyloskopie se zabývá identifikací, respektive rozlišením, osob podle otisku jejich prstu, dlaně či plosek nohou. [WIKIa]

Zásadní způsob využití našla daktyloskopie v kriminalistice a řadí se mezi její nejstarší techniky. Pro využívání daktyloskopických stop jako důkazního materiálu však bylo zapotřebí exaktně potvrdit jejich individuálnost, neměnnost a neodstranitelnost – dále popsáno v kapitole 1.2. [SUCH96]

1.1 Charakteristika kůže lidské ruky

Kůže je největší orgán lidského těla a jako takový má mnoho funkcí. Jednou z nich je percepce vzruchů z okolí na těch částech těla, kde je hmatu zapotřebí – ruka a chodidlo. Schopnost vnímat hmatem je založena na tom, že vrchní vrstva kůže je zformovaná do takzvaných hmatových lišt a při styku kůže s nějakým objektem je vytvořen tlak na tyto lišty. Ty přenesou tlak do nižší vrstvy kůže, kde na něj mohou reagovat hmatové buňky.

Hmatové lišty se soustřeďují do delších linií – lineárních vyvýšenin oddělených rýhami. Vznik těchto vyvýšených linií způsobují výběžky škáry (nižší vrstvy kůže), jež jsou vždy dva pod každou linií, resp. hmatovou lištou. Tyto výběžky škáry se nazývají papily, proto nazýváme vyvýšené linie vrchní části kůže „papilární linie“.

Hmatovou funkci papilárních linií vypozoroval již J. E. Purkyně v roce 1823, který také objevil papilární linie na končetinách a chápavém ocasu opic. Pozdější výzkum zjistil přítomnost papilárních linií, jako součásti hmatového ústrojí, napříč celou třídou savců. Ovšem je důležité podotknout, že jsou rozdíly v daktyloskopických stopách mezi člověkem a jiným živočichem na Zemi, který je schopen tuto stopu zanechat, zásadní a tedy jsou takové stopy nezaměnitelné. [STRA05] [SUCH96]

1.2 Zákony daktyloskopie (postuláty)

Mezi rozdílové znaky obrazců, které tvoří papilární linie, zahrnujeme všechny změny v průběhu těchto linií - tedy ukončení, křížení (linie tvaru písmene „X“) a rozdvojení (linie tvaru písmene „Y“). Pro klasifikaci se v daktyloskopii používá také globální vzor, který linie tvoří (výr, smyčka, oblouk a další). Těmto znakům se však v naší práci nevěnujeme.

Aby mohla být daktyloskopie uznána jako exaktní a spolehlivý způsob identifikace osob, musely být zformovány a prokázány níže uvedená tvrzení. Jejich znění a dokazování uvádějí ve svých knihách Suchánek a Straus.

1.2.1 Individuálnost

První základní zákon daktyloskopie zní: „Na světě nejsou dva lidští jedinci, kteří by měli shodné obrazce papilárních linií“. Ke zformování toho zákona dopomohly metody matematické statistiky, díky kterým bylo prokázáno, že na jednom článku prstu může být 64 miliard variant obrazců [STRA05]. Takto vysoký počet variant v součinnosti s náhodností výskytu rozdílových znaků (prokázáno lékařskými výzkumy) a počtem lidí na planetě Zemi umožňuje vyslovení právě zákona o individuálnosti.

Pro úplnost bylo také prokázáno mnohými výzkumy, že pomocí otisků prstů lze jednoznačně rozlišit i jednovaječná dvojčata, u nichž je doposud velmi obtížné rozlišení pomocí testu DNA.

1.2.2 Neměnnost

Druhý základní zákon daktyloskopie zní: „Jedince je možné identifikovat pomocí otisku prstu bez ohledu na dobu uplynulou mezi sejmutím testovaného otisku“. V důsledku toto tvrzení znamená, že pokud porovnáme otisk prstu osoby sejmutý v dětství s otiskem sejmutým stejné osobě ve stáří, spolehlivě nalezneme shodu v rozdílových znacích.

Neměnnost otisku prstu se považuje za relativní pojem, protože během růstu se může lidská kůže natáhnout, případně se mohou objevit jizvy či rýhy v návaznosti na zranění nebo ohyb. Tyto anomálie však nedokáží změnit původní kresbu a výskyt původních rozdílových znaků.

1.2.3 Neodstranitelnost

Třetí základní zákon daktyloskopie zní: „Papilární linie jsou neodstranitelné, pokud není odstraněna nebo zničena zárodečná vrstva kůže“. Výzkumy bylo prokázáno, že i po opakovaném odírání povrchových vrstev kůže na prstech se tato kůže zhojila a získala zpět svůj původní tvar.

2. Zpracování vstupního otisku prstu

Základním stavebním kamenem tohoto programu je otisk prstu. Takže typickým vstupem je obraz ve formátu *PNG*, *JPEG*, *BMP* nebo *TIFF* obsahující otisk jednoho článku prstu v odstínech šedi, kde papilární linie jsou vyznačené tmavými barvami a rýhy mezi nimi a okraje obrazu jsou světlé (viz obrázek 2.1).

Zpracování vstupu jsme rozdělili na dvě základní části, které popisují následující podkapitoly. První z nich je binarizace a druhou je detekce kostry. Pro detekci kostry jsme se opřeli o poznatky zdrojů zabývajících se zpracováním obrazu. Binarizaci jsme implementovali jak pomocí prověřené techniky konvexního práhování tak i vlastním postupem, který využívá zejména obrazových transformací. Srovnání výsledků obou technik je uvedeno na konci kapitoly 2.3.

Obrázek 2.1: Typický vstup

Aplikace transformuje vstup grafickými úpravami tak, aby otisk mohl být dále zpracováván. Vytvoří obraz, kde papilární linie mají všude sílu 1 pixel a jsou bílé na černém pozadí. Takovému obrazu budeme říkat kostra, jelikož se jedná o pomyslnou kostru původního obrázku.

Získávání významných bodů otisku typicky probíhá dvěma způsoby - analýzou původního vstupu v odstínech šedi a nebo binarizované[1] verze vstupního obrazu. První metoda se vyznačuje lepšími výsledky nad nekvalitním vstupem, druhá naopak vyniká svou jednoduchostí pro implementaci a rychlostí. My jsme v této práci zvolili postup binarizací, protože se jedná o nejčastěji používaný způsob předzpracování vstupního obrazu. V následujících kapitolách ukážeme dva způsoby binarizace a následnou detekci kostry ze získaného binárního obrazu. [BANS11]

Obrazové transformace jsou implementovány zejména za použití frameworku *AForge.NET*, který se specializuje mimo jiné i na zpracování obrazu. Samotné transformace se provádějí způsobem aplikace filtru – filtrovací sekvenci přidáme požadované filtry a spuštěním sekvence dostaneme na výstupu upravený obraz. Pokud tedy v následující části hovoříme o použití určitého filtru, máme na mysli filtr z této knihovny.

2.1 Příprava na binarizaci

Odstíny šedi

Jako první operace je jednoduché převedení vstupu na verzi v odstínech šedi. Toto je důležité jako příprava pro následující operace. Pokud vstupní obraz splní výše zmíněné parametry, nebude tato transformace mít viditelný výsledek. Jedná se totiž jen o převedení formátu vstupního obrázku. Obraz, který takto vznikl budeme dále nazývat šedý.

Rozmazání

Další transformací je vytvoření kopie šedého obrazu a jeho rozmazání. To provedeme pomocí filtrů uzávěr (*Closing*; zastupuje proces dilatace – rozšíření bílých objektů – následovaný erozí – zúžení bílých objektů) a sjednocení s originálem. Výsledkem je obrázek 2.4. Vytvoření tohoto obrazu je opět jen pomocná operace pro další průběh programu. Tento krok je však zapotřebí jen pro binarizaci pomocí transformací popsanou v kapitole 2.2, nikoliv při postupu z kapitoly 2.3.

[1] Binarizací nazýváme proces, kdy obraz v odstínech šedi upravíme tak, že ho převedeme do černobílé verze (tedy podle tmavosti/světlosti určíme, jestli má být konkrétní pixel bílý nebo černý – vznikne binární tvar obrazu).

2.2 Zdůraznění okrajů a binarizace transformacemi

Nejprve uvádíme vlastní jednoduchý postup pro získání binarizovaného obrazu, který využívá pouze transformace bez adaptace parametrů k aktuálnímu vstupu.

Před samotnou binarizací zvýrazníme okraje objektů (zde rozumějte ve smyslu zkrácení barevného přechodu v okraji linie), abychom co nejvíce omezili vytvoření nových děr mezi liniemi. Toto vylepšení se použije z toho důvodu, že binarizace se dá chápat jako zaokrouhlování. Ačkoliv se může opticky zdát, že v určitých částech by měla po binarizaci být například bílá, zaokrouhlení se provede opačně na černou, protože v tomto místě nedosahuje hodnota dostatečné hranice. Právě možná chybovost je největší slabinou tohoto procesu a proto jsme zavedli ještě proces vylepšení okrajů.

2.2.1 Vylepšení okrajů papilární linie

Vylepšení okrajů je proces skládající se z aplikace několika filtrů. Prvním z nich je takzvaný Cannyho detektor okrajů (*CannyEdgeDetector*). Ten dostane na vstupu obraz (v našem případě šedý obraz, kde jsou linie tmavé a pozadí světlé) a výstupem je černý obraz s okraji objektů vykreslenými v takových barvách, jako by tímto okrajem prosvítal původní okraj (tedy z šedého obrazu vznikne opět šedý, ale jinak vykreslený – viz obrázek 2.5). Pro další práci je třeba převést okraje do jednotné bílé barvy pomocí filtru adaptivního prahování (*BradleyLocalThresholding*), který tuto operaci provede (výsledkem je obrázek 2.6). Pak provedeme inverzi obrazu Cannyho okrajů kvůli následující transformaci – tedy se vymění barvy bílá za černou a naopak.

Poslední operací vylepšující okraje linií před binarizací je průnik (filtr *Intersect*). Výsledkem průniku dvou černobílých obrazů je takový černobílý obraz, kde každý pixel má minimální hodnotu z odpovídajících pixelů obou vstupních obrazů (hodnota pixelu odpovídá světlosti barvy – černá = 0 a bílá = 255). V našem případě je jedním vstupem výsledek předchozího procesu a druhým pak rozmazaný obraz jehož popis je v kapitole 2.1. Vezmeme li do úvahy, že po inverzi jsou Cannyho okraje černé na bílém pozadí a rozmazaný obraz má tmavé linie na světlém podkladu, vrátí nám tento proces takový obraz, kde pozadí je světlé, linie jsou tmavé, ohraničené ostře černou barvou. Tím jsme docílili toho, že okraje linií jsou zvýrazněné a tedy připravené na binarizaci s výrazně sníženou chybovostí, jak ilustruje 2.2.

Obrázek 2.2: Výsledek binarizace transformacemi s vylepšením okrajů (vlevo) a bez vylepšení (vpravo).

2.2.2 Binarizace

V tuto chvíli přijde na řadu samotná binarizace. Ta je oproti předešlému postupu jednoduchým procesem aplikace jediného filtru – lokálně adaptivní prahování (*BradleyLocalThresholding* filtr). Tento proces zaokrouhlí barvy a výsledkem je černobílý obraz, jak ukazuje obrázek 2.7. Jako mez pro prahování se při testech osvědčila hodnota 200 – tedy hodnota blíže k bílé. Protože jsou linie ohraničeny ostře černou, jsou jejich okraje téměř neměnné při prahování s jakoukoliv hodnotou od 10 do 230, ale pro vnitřní prostor linií (myšleno výplň objektu linie) je lepší, když se práhuje podle hodnoty bližší k bílé, protože je potřeba vyplnit tyto linie v maximální možné míře.

Pro další práci je potřeba invertovaného binarizovaného obrazu (linie bílé a pozadí černé). Tato inverze je v programu součástí funkce binarizace. Pokud budeme dále hovořit o binarizovaném obrazu, myslíme tím tuto jeho invertovanou verzi.

Obrázek 2.3: Originál vstupu

Obrázek 2.4: Rozmazaná verze

Obrázek 2.5: Výsledek detekce okrajů Cannyho algoritmem

Obrázek 2.6: Cannyho okraje po adaptivním prahování

Obrázek 2.7: Výsledek z vylepšené Cannyho detekce

2.3 Binarizace konvexním prahováním

Jiným způsobem provedení binarizace je použití konvexního prahování, které doporučují Kim a Park ve svém článku „Fingerprint binarization using convex threshold“[KIM03]. Tento proces se skládá ze čtyř částí: získání lokálních tangentových směrnic linií, jejich kvantifikace, regularizační filtrování a konvexní prahování, které rozhodne o výsledné barvě bodu. V naší aplikaci jsme však od těchto

autorů použili jen základní koncept - kvantifikaci a samotné práhování. Zbylé dva kroky, výpočet lokálních orientací a regularizační filtrování, jsme naimplementovali způsobem, který uvádějí Maio a Maltoni ve své práci „Direct Gray-Scale Minutiae Detection In Fingerprints“[MAIO97] zabývající se metodou detekce významných bodů otisku bez použití binarizace. Současně jsme také pozměnili pořadí kroků celého procesu v návaznosti na implementaci regularizace, protože způsob, který používají Maio a Maltoni, pracuje nad libovolnými směry na rozdíl od původního způsobu Kima a Parka. Mohli jsme tedy prohodit pořadí operací regularizace a kvantifikace, takže regularizace pracuje s přesnějšími údaji bez zaokrouhlování. V tomto pozměněném pořadí následují jednotlivé popisy operací.

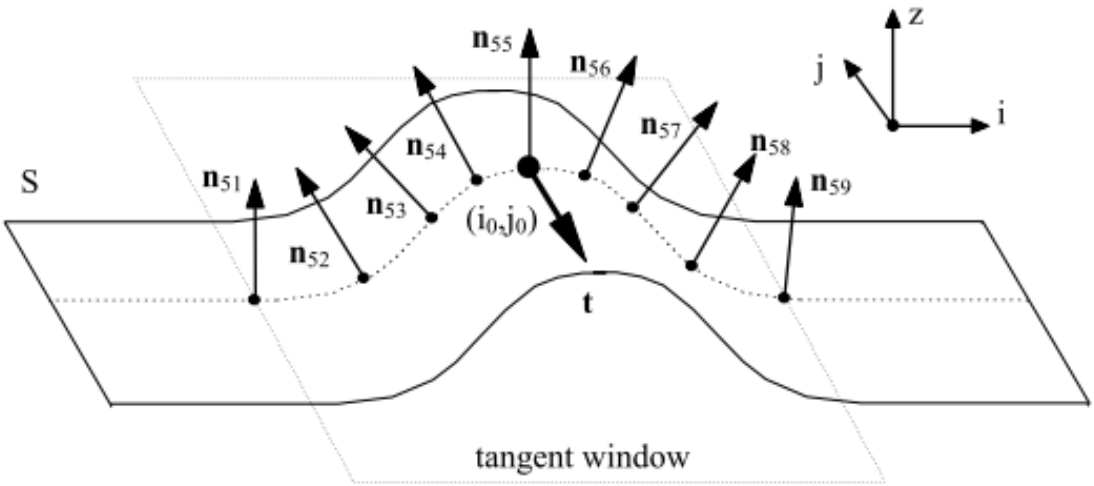
2.3.1 Výpočet lokálních tangentových směrnic linií

V tomto kroku se zaměříme na výpočet lokálních orientací dle [MAIO97]. To znamená, že celý vstupní obrázek rozdělíme na bloky a pak pro každý vypočítáme průměrný směr, kterým směřují linie v tomto bloku.

Mějme pixel (i_0, j_0) obrázku v odstínech šedi, označíme φ_0 úhel - tangentovou směrnici - kterou chceme vypočítat. Tangentovým oknem označíme čtvercovou oblast se středem v bodě (i_0, j_0) s délkou strany α. Pro každý bod (i_h, j_k) tangentového okna vypočítáme normálový vektor n_{hk} kolmý na reliéf povrchu obrázku. Průměrná tangentová směrnice je takový jednotkový vektor, který je „nejvíce kolmý“ ke všem normálovým vektorům n_{hk} v aktuálním tangentovém okně. Celou situaci ilustruje obrázek 2.8.

Definujme $gray\,(i, j)$ jako funkci, která vrací hodnotu obrázku v daném bodě (i, j) z intervalu reálných čísel $\langle 0, 1\rangle$, kde černá je 1 a bílá je 0. Bod (i_h, j_k) je bod tangentového okna. Výpočet tangentové směrnice probíhá dle níže uvedeného postupu.

$$a_1 = gray\,(i_{h+1}, j_{k+1}) \qquad a_2 = gray\,(i_{h-1}, j_{k+1})$$
$$a_3 = gray\,(i_{h-1}, j_{k-1}) \qquad a_4 = gray\,(i_{h+1}, j_{k-1})$$
$$a_{hk} = \frac{-a_1 + a_2 + a_3 - a_4}{4} \qquad b_{hk} = \frac{-a_1 - a_2 + a_3 + a_4}{4}$$
$$n_{hk} = (a_{hk}, b_{hk}, 1)\,, \quad \text{zjednodušeně bez osy } z\text{:} \quad v_{hk} = (a_{hk}, b_{hk})$$
$$A = \sum_{h=1}^{\alpha}\sum_{k=1}^{\alpha} (a_{hk})^2 \qquad B = \sum_{h=1}^{\alpha}\sum_{k=1}^{\alpha} (b_{hk})^2 \qquad C = \sum_{h=1}^{\alpha}\sum_{k=1}^{\alpha} (a_{hk} \cdot b_{hk})$$



Obrázek 2.8: Povrch reliéfu znázorňuje S v místě, kde probíhá papilární linie ve směru j se středem v bodě (i_0, j_0) a s délkou strany 9. Pro jednoduchost jsou zobrazeny normálové vektory jen pro jeden řádek okna. Vektor t je hledaný „nejvíce kolmý“ jednotkový vektor vzhledem k normálovým vektorům.

$$t = \begin{cases} \left(1, \frac{B-A}{2C} - sign(C)\sqrt{\left(\frac{B-A}{2C}\right)^2 + 1}\right) & \text{pokud } C \neq 0 \\ (1,0) & \text{pokud } C = 0 \wedge A \leq B \\ (0,1) & \text{pokud } C = 0 \wedge A > B \end{cases}$$

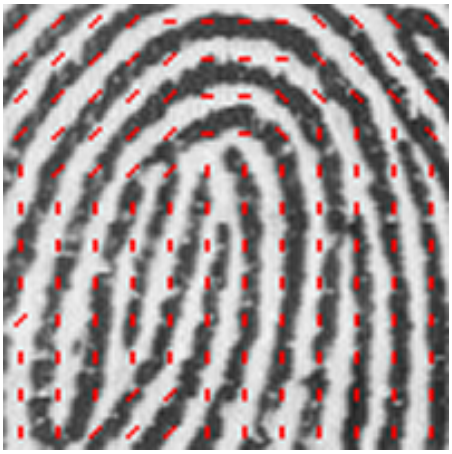
Označme t_1 a t_2 prvky vektoru t, tedy $t = (t_1, t_2)$

$$\varphi_0 = \begin{cases} \arctan\left(\frac{t_2}{t_1}\right) & \text{pokud } t_1 \neq 0 \\ \frac{\pi}{2} & \text{jinak} \end{cases} \tag{2.1}$$

Pomocí výpočtu 2.1 získáme průměrnou směrnici jednoho tangentového okna. Spočteme tedy směrnice pro všechna okna ve vstupním obrázku a získané hodnoty použijeme v následujících krocích. Výsledek ilustruje obrázek 2.9.

2.3.2 Regularizace obrazu

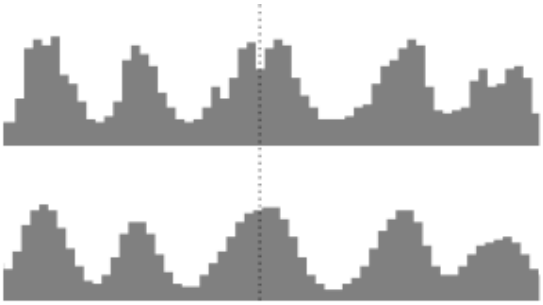
Pomocným krokem celé binarizační operace je takzvaná *regularizace obrazu*. Je to postup, při kterém se odstraní z otisku drobné artefakty (ostrůvky mezi liniemi či díry uvnitř linie), které znesnadňují nalezení lokálního maxima určujícího



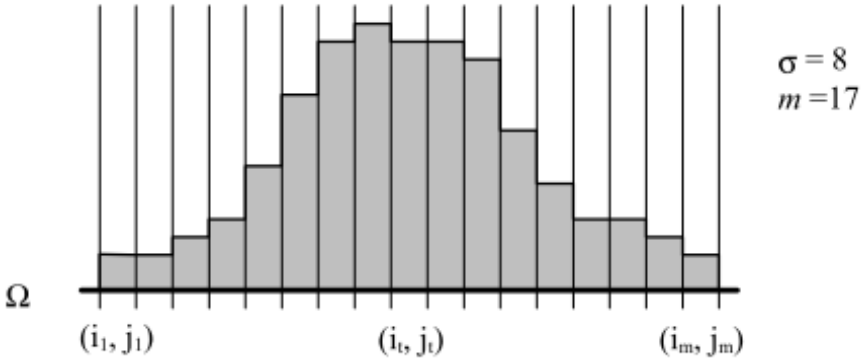
Obrázek 2.9: Zobrazení části výstupu výpočtu tangentových směrnic. Pro zjednodušení vykreslení byly směrnice zaokrouhleny na 4 směry (svislý, vodorovný a dva diagonální)

střed linie. Tyto nepravidelnosti můžeme vidět, pokud provedeme řez linií kolmý k její směrnici a zároveň k ploše obrázku. Na řezu si zobrazíme hodnoty odstínů šedi tak, že světlé odstíny jsou malé a tmavé jsou vysoké. Vidíme tedy siluetu střídání linií a mezer. Nepravidelnosti se zde projevují jako chyby v pravidelném, téměř sinusoidovém, průběhu siluety. Můžeme je také popsat jako tvary podobné sopkám (linie není hladce zaoblená, ale má v sobě „kráter“). Tuto situaci znázorňuje obrázek 2.10, který stejně tak ukazuje, jak by měla vypadat silueta po provedení regularizace. Druhotným efektem procesu je zjemnění přechodů v místě stoupání/klesání siluety, což opět napomáhá přesnějšímu určení lokálního maxima. Plynulejší průběh siluety pak výrazně zlepšuje výsledek detekce kostry otisku popsaný v kapitole 2.4.

Autoři Maio a Maltoni regularizaci dělí na dvě hlavní části, které popisují následující odstavce. Jako vstup této operaci slouží vstupní obrázek programu v odstínech šedi a výsledek detekce lokálních orientací z kapitoly 2.3.1. Základní strukturou, která je využita v obou těchto částech je právě silueta (značíme Ω). Získáme ji tak, že zvolíme její středový bod a pak σ bodů v ortogonálním směru vůči lokální tangentové orientaci linie v tomto středovém bodě a stejný počet bodů ve směru opačném. Celkem tedy $2\sigma+1$ bodů (označme m). Získaná silueta by měla vypadat jako na obrázku 2.11.



Obrázek 2.10: Srovnání siluety v řezu napříč papilární linií před (nahoře) a po regularizaci (dole). Vyznačen je bod, kde by měl být vrchol linie (lokální maximum), ale na původním vstupu je hodnota, která znemožňuje určení středu linie.



Obrázek 2.11: Grafická reprezentace siluety, kde šedé sloupce znázorňují hodnotu konkrétního pixelu v příčném řezu papilární linií. Zvolený středový bod je označen (i_t, j_t).

Průměrování odstínu ve směru linie

První částí procesu regularizace je vyhlazení siluety podélně vzhledem k průběhu papilární linie. Toho docílíme tak, že vedle siluety ve vybraném středovém bodě vezmeme ještě obdobnou siluetu Ω_{-1} se středovým bodem vzdáleným 1 pixel od původního proti směru průběhu linie a jednu siluetu Ω_{+1} se středovým bodem vzdáleným 1 pixel od původního po směru průběhu linie. Takto tedy získáme vždy tři pixely, které jsou po sobě jdoucí, pokud bychom postupovali obrázkem ve směru papilární linie. Pro každou takovou trojici vypočítáme aritmetický průměr jejich hodnot a ten pak použijeme v siluetě Ω' v následujícím kroku regularizace.

$$\Omega'[i] = \frac{\Omega_{-1}[i] + \Omega[i] + \Omega_{+1}[i]}{3} \qquad \text{pro } i = 1, \ldots, m$$

Konvoluce konstantní maskou

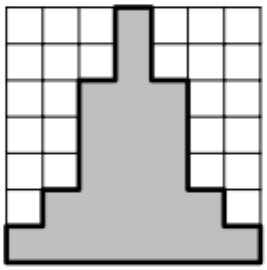
Pro druhý krok potřebujeme masku, kterou budeme aplikovat na průměrnou siluetu Ω'. Masku d graficky znázorňuje obrázek 2.12 a musí pro ni platit:

$$d_k \text{ je prvek masky pro } k = 1, \ldots, 2p+1 \text{ a } p \geq 0$$

$$d_k \geq 0 \wedge \sum_{k=1}^{2p+1} d_k = 1$$

Výpočet konvoluce, která vyhladí pomocí uvedené masky siluetu, proběhne takto:

$$\Omega''_k = \frac{1}{2p+1} \sum_{v=-p}^{p} d_{p+1+v} \cdot \Omega' [k+v] \qquad \text{pro } k = p+1, \ldots, m-p$$



Obrázek 2.12: Grafické znázornění symetrické gaussovské masky použité pro regularizaci. Její číselná reprezentace je $d = \left[\frac{1}{23}, \frac{2}{23}, \frac{5}{23}, \frac{7}{23}, \frac{5}{23}, \frac{2}{23}, \frac{1}{23}\right]$ pro $p = 3$.

Výsledek regularizace promítneme do obrázku a ten použijeme jako vstup konvexního prahování. Srovnání vstupu před a po regularizaci můžeme vidět na obrázku 2.13.

2.3.3 Kvantifikace směrnic

Protože následující krok - konvexní prahování - používá jen malý prostor tří pixelů pro svou analýzu, můžeme kvůli jednoduchosti implementace kvantifikovat každou lokální směrnici $\theta(x, y)$ do 4 základních směrů $0, \frac{\pi}{4}, \frac{\pi}{2}, -\frac{\pi}{4}$ podle vzorce 2.2. [KIM03]

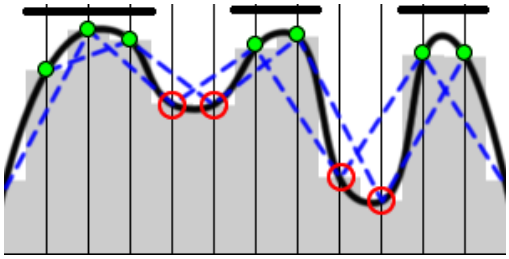
Obrázek 2.13: Efekt regularizace (vpravo) ve srovnání se vstupem (vlevo)

Obrázek 2.14: Binarizovaný vstup bez regularizace

$$\theta_q(x,y) = \begin{cases} 0 & \text{pokud } -\frac{\pi}{8} < \theta(x,y) \leq \frac{\pi}{8} \\ \frac{\pi}{4} & \text{pokud } \frac{\pi}{8} < \theta(x,y) \leq \frac{3\pi}{8} \\ -\frac{\pi}{4} & \text{pokud } -\frac{3\pi}{8} < \theta(x,y) \leq -\frac{\pi}{8} \\ \frac{\pi}{2} & \text{pokud } \theta(x,y) > \frac{3\pi}{8} \vee \theta(x,y) \leq -\frac{3\pi}{8} \end{cases} \tag{2.2}$$

2.3.4 Konvexní prahování

Samotné konvexní prahování je konečným procesem binarizace. Využije jak kvantifikované lokální orientace θ_q, tak i regularizovaný obraz. Pro každý jednotlivý bod obrázku je rozhodnuto o jeho výsledné barvě podle toho, jestli je v konvexní části siluety příčného řezu linií nebo ne (ilustruje obrázek 2.15). Pro určení barvy bodu (x, y) je jeho hodnota porovnána s body (x^+, y^+) a (x^-, y^-), které získáme v závislosti na lokální orientaci (viz obrázek 2.16). Všechny takové body (x, y), které vyhovují podmínce 2.3 budou prohlášeny za černé (tedy body v linii). Výsledkem je obrázek 2.17. [KIM03]



Obrázek 2.15: Šedá oblast je silueta řezu napříč liniemi, černá křivka tuto siluetu opisuje, modře jsou vyznačeny linky představující trojice analyzovaných bodů, zelené body jsou pak konvexní, červené jsou konkávní. Černé linky nahoře zvýrazňují konvexní oblasti, které jsou uznány jako výsledná linie.

$$(x, y) \geq \frac{(x^-, y^-) + (x^+, y^+)}{2} \tag{2.3}$$

Obrázky 2.17 a 2.18 umožňují porovnat výsledky obou způsobů binarizace. Vidíme, že při použití transformací má výstup méně drobných artefaktů, ale na druhou stranu se má větší tendenci zesilovat linie a tak nám vznikají nechtěné spojnice v místech, kde byla mezera mezi liniemi užší. Také se zde projevila výhoda konvexního prahování v podobě velmi dobré adaptovatelnosti. Místa se slabým kontrastem byla korektně přečtena, naopak transformace v těchto místech zanechaly mezeru. V dalších kapitolách tedy budeme používat výsledek binarizace konvexním prahováním.

2.4 Detekce kostry

Detekce kostry z binarizovaného obrazu je nejsložitější proces jeho zpracování, který tato aplikace využívá. Pro tuto detekci se používá hit-and-miss filtr (*HitAndMiss* z knihovny AForge), jehož práce je popsána v následující kapitole 2.4.1.

$\theta_q = \frac{\pi}{2}$	$x-1$	x	$x+1$
$y-1$			
y	(x^-, y^-)	(x, y)	(x^+, y^+)
$y+1$			

$\theta_q = -\frac{\pi}{4}$	$x-1$	x	$x+1$
$y-1$			(x^+, y^+)
y		(x, y)	
$y+1$	(x^-, y^-)		

$\theta_q = \frac{\pi}{4}$	$x-1$	x	$x+1$
$y-1$	(x^-, y^-)		
y		(x, y)	
$y+1$			(x^+, y^+)

$\theta_q = 0$	$x-1$	x	$x+1$
$y-1$		(x^-, y^-)	
y		(x, y)	
$y+1$		(x^+, y^+)	

Obrázek 2.16: Jednotlivé tabulky ukazují, které pixely vzhledem k prostřednímu se používají při určených hodnotách lokálního směru. Body tedy vybíráme ortogonálně ke směru papilární linie.

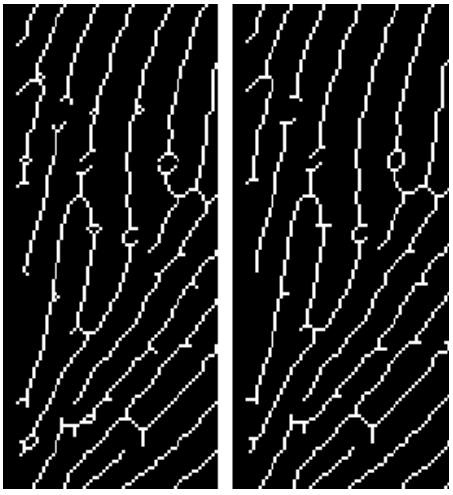
Tento filtr je zastřešen funkcí *skeletonizeImage*, která vrací právě kostru z vloženého binárního obrazu. Aby byla výsledná kostra hladší a umožňovala lepší detekci významných bodů otisku, je tato funkce zavolána dvakrát nad jedním vstupem, přičemž mezi těmito dvěma voláními se provede ještě operace uzávěru. Rozdíl mezi jednoduchým provedením funkce *skeletonizeImage* a dvojným spolu s aplikací uzávěru je vidět na srovnání, které zobrazuje obrázek 2.19.

2.4.1 Hit-and-miss filtr

Filtr s názvem hit-and-miss (někdy také hit-or-miss) je základním nástrojem pro detekci tvaru objektu nad binárním obrazem. Při svém běhu využívá matici pro porovnávání nazývanou strukturální element, přičemž hodnoty v této matici určují, kde se má při srovnání matice s blokem obrazu nacházet objekt (popředí) a kde pozadí. Podle shody matice s obrazem se do testovaného pixelu vyplní hodnota. Do matice se také běžně vkládá třetí hodnota určující nesměrodatný bod. Právě hodnoty matice a způsob jejich aplikace hit-and-miss metodou určují, jakou povahu má vykonávaná operace. [GONZ92]

Obrázek 2.17: Výsledek binarizace konvexním práhováním.

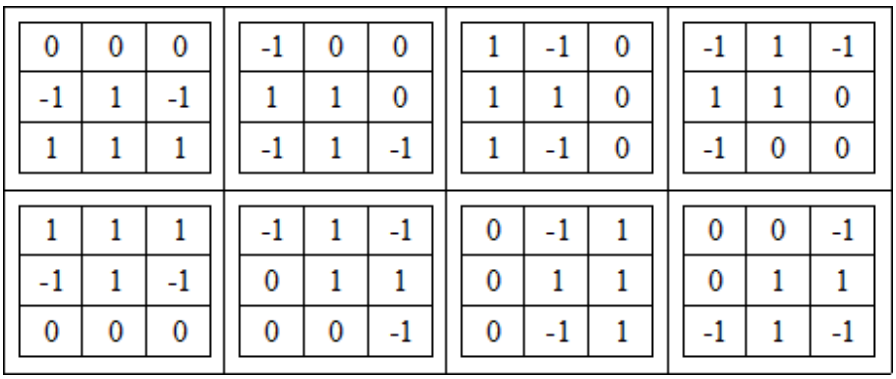
Obrázek 2.18: Výsledek binarizace transformacemi.



Obrázek 2.19: Porovnání jednoho průchodu detekce kostry (vlevo) s dvěma průchody proloženými uzávěrem (vpravo)

V této práci se hit-and-miss transformace používá pro ztenčení objektů. Ztenčením zde myslíme proces, na jehož konci mají objekty tloušťku jednoho pixelu, ale zachová se jejich délka a výsledná linka kopíruje celkový tvar objektu. K tomu

se využívá postupné aplikace filtru s osmi strukturálními elementy. Tyto elementy ukazuje obrázek 2.20. Je možné si všimnout, že se vlastně jedná o dva opakující se elementy, jen postupně rotují o 90°. [FISH04]

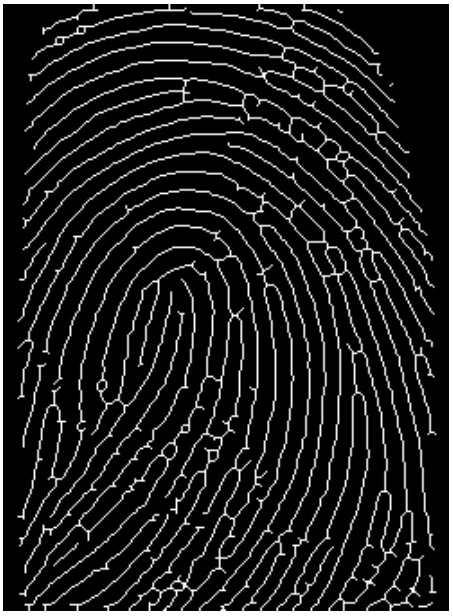


Obrázek 2.20: Strukturální elementy pro ztenčování

Postup filtru je takový, že pro každou matici projde všechny pixely obrázku a o každém z nich určí jestli má být bílý nebo černý. K tomu použije právě hodnoty strukturálního elementu. Matici zarovná s aktuálním pixelem přesně na střed. Pak ověří, jestli je pod buňkami matice s hodnotou 0 bílá barva a pod hodnotou 1 černá. Pokud ano, bude tento pixel bílý, jinak bude černý. Body, které leží pod buňkami s hodnotou -1 mohou mít libovolnou barvu. Aby výsledné linky měly tloušťku jednoho pixelu, musí se proces iterativně opakovat dokud dochází ke změně obrazu.

2.5 Výsledek transformačního procesu

Výsledkem celého procesu transformací je obraz, který obsahuje kostru otisku prstu. Tento obraz je černobílý, linky jsou bílé a pozadí je černé. Dále v textu se na něj budeme odkazovat jako na kostru. Typický výstup tedy vypadá jako obrázek 2.21.



Obrázek 2.21: Výsledek transformačního procesu (kostra otisku prstu)

3. Detekce a zpracování významných bodů

Kvůli jednoduchosti implementace řekneme, že významný bod je takový, kde končí papilární linie (zanedbáme křížení a rozdvojení). Chceme tedy znát souřadnice takových bodů na vstupu a směr, kterým linie směřuje.

Pro detekci významných bodů námi byly navrhnuty a implementovány dva postupy. První z nich je jednoduchý záplavový algoritmus a druhý postup používá upravený spojový seznam (graf). Protože bylo k záplavovému algoritmu nutno implementovat ještě samostatnou detekci směrnic, zůstal tento algoritmus v pozadí a dále nebyl rozvíjen. Poslouží nám pro ilustraci minimálních požadavků na lepší grafový algoritmus.

V dalších kapitolách jsou pak popsány postupy zlepšení výsledků detekce bodů i výpočtu koncových směrnic.

3.1 Detekce významných bodů pomocí záplavového algoritmu

Protože se významné body nacházejí na koncích linií, je třeba nalézt tyto konce. Použití jednoduchého záplavového algoritmu zajistí jejich spolehlivé nalezení ve vstupním obrazu kostry otisku. Pseudokód SimpleFloodfillPointsDetector ukazuje práci tohoto algoritmu.

Hledání probíhá tak, že program prochází všechny pixely obrazu a pokud narazí na bílou barvu (barva linie), spustí záplavové hledání. To je implementováno pomocí fronty a využívá změn barev pixelů pro naznačení stavu, ve kterém se pixel nachází (bílá = nezpracovaný bod; červená = zpracovaný bod nebo bod ve frontě). Pro jednoduchost neukazuje pseudokód přesnou implementaci průchodu sousedních pixelů. Ve skutečnosti se používá funkce, která otestuje stav určeného pixelu. Pokud má být pixel prozkoumán, je vložen do fronty. Navíc tato funkce vrátí kód stavu tohoto pixelu: 1 = pixel byl vložen do fronty; 10 = pixel už je ve frontě či zpracován; 0 = neznámý stav pixelu (neexistující pixel nebo neznámý stav). Návratové kódy všech sousedních pixelů aktuálního bodu se sečtou a získáme tak „sousedský kód“. Tento kód obsahuje informace potřebné k rozhodnutí o tom, jestli je aktuální bod koncovým bodem či ne. Kód totiž obsahuje počet sousedů,

Algoritmus 1 SimpleFloodfillPointsDetector

```
1: function findEndings
2:     resultPoints ← empty list
3:     for all pixel in inputImage do
4:         if pixel.color = whiteColor then
5:             points ← floodFillWithEndings(inputImage, pixel)
6:             resultPoints.add(points)
7:     return resultPoints

8: function floodFillWithEndings(inputImage, pixel)
9:     result ← empty list
10:    firstIteration ← true
11:    pointsQueue ← empty list
12:    pointsQueue.add(pixel)
13:    while not pointsQueue.empty do
14:        p ← pointsQueue.pop()
15:        enqueuedCounter ← 0
16:        unenqueuedCounter ← 0
17:        for all px in p.neighbours do
18:            if px is in pointsQueue then
19:                enqueuedCounter + +
20:            else
21:                unenqueuedCounter + +
22:                if px.color = whiteColor then
23:                    pointsQueue.push(px)
24:                    px.color ← redColor
25:        if enqueuedCounter <= 1 and unenqueuedCounter = 0 then
26:            result.add(p)
27:        else if firstIteration = true then
28:            if enqueuedCounter = 0 and unenqueuedCounter = 1 then
29:                result.add(p)
30:        firstIteration ← false
31:    return result
```

kteří již byli zpracováni v násobcích deseti, a počet sousedů, kteří na zpracování čekají ve zbytku po dělení deseti. Například tedy kód 42 značí, že aktuální bod má 4 již zpracované sousedy a 2 dosud nezpracované.

Díky zjištěným počtům sousedů a skutečnosti, že testujeme kostru otisku, která má vždy tloušťku jednoho pixelu, není těžké si uvědomit, že koncový bod musí mít buď kód 10 (jeden zpracovaný soused, žádný nezpracovaný) nebo kód 1 (žádný zpracovaný, jeden nezpracovaný) za předpokladu, že testujeme první bod celého algoritmu. Tomu také odpovídá dvojice podmínek, které umožňují přidání bodu do výsledného seznamu, uvedených výše v pseudokódu.

Tento algoritmus pracuje rychle – lineárně k počtu pixelů vstupního obrazu – a je jednoduchý na implementaci. Nicméně jeho výstupem je pouze seznam bodů a ani jeho průběh neumožňuje jednoduché odvození dalších potřebných informací o linii – zejména se jedná o směrnice koncových bodů.

3.2 Detekce významných bodů grafem směrnic

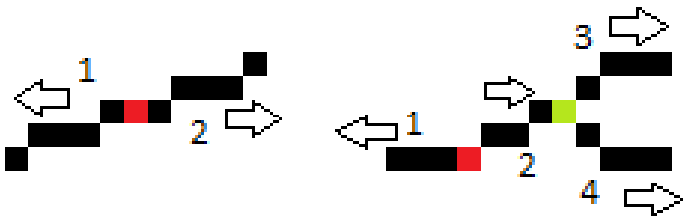
Záplavový algoritmus ve smyslu prohledání vstupního obrazu systematicky v liniích posloužil jako dobrý a dostatečně efektivní základ pro další algoritmus používající pomocnou strukturu – sadu lineárních spojových seznamů, kterou budeme dále nazývat **sada grafů směrnic**. Jedná se o datový objekt, který zapouzdřuje seznam grafů směrnic a implementuje nad ním operace umožňující jeho stavbu v závislosti na vstupním obrazu.

Graf směrnic, který je základem sady grafů směrnic, je speciální typ oboustranného spojového seznamu. Kromě položek hlava (první prvek seznamu) a ocas (poslední prvek seznamu) má také ukazatele na rodičovský seznam a seznam ukazatelů na své potomky. Tyto položky nejsou použity pro vyjádření stavby grafu směrnic, ale slouží k udržení informací o návaznosti jednotlivých grafů (bude detailněji popsáno níže).

Uzel grafu směrnic je reprezentací jednoho bodu, takže obsahuje datové položky pro uchování souřadnic bodu v obraze, vektoru směrnice a ukazatele na následníka a předchůdce.

Samotný běh algoritmu je v zásadě podobný běhu záplavového algoritmu – nalezneme bílý bod, od tohoto bodu hledáme sousední bílé body a ty zpracováváme. Rozdíl je však v tom, že přímo neanalyzujeme koncové body, ale jen stavíme grafy.

Z jedné souvislé linie na vstupu vznikne jedna sada grafů. Samozřejmostí je barvení již prozkoumaných bodů, aby se zamezilo vícenásobnému průchodu jedním bodem.



Obrázek 3.1: Ukázka postupu detekce grafu směrnic (černý bod je bod v linii; červený bod je počáteční bod detekce; zelený bod je společný bod přiléhajících grafů; čísla označují jednotlivé grafy; šipky ukazují směr postupu detekce)

Jak ukazuje obrázek 3.1, algoritmus začne v zadaném bodě a vytvoří dva grafy, jestliže je tento bod uprostřed linie. První z těchto grafů je prohlášen za otce toho druhého. Pokud je bod na konci linie, vytváří se jediný graf. V případě, že algoritmus narazí na rozcestí (aktuální bod má více než jeden bod k dalšímu postupu), ukončí se stavba aktuálního grafu a začnou se tvořit nové grafy v takovém počtu, kolik je cest z aktuálního bodu. Bod, který byl rozcestím je vložen do všech grafů a nové grafy jsou nastaveny jako potomci toho, který k nim směřuje. Na předchozí ilustraci vpravo tedy vznikne sada grafů označených 1, 2, 3 a 4, přičemž červený bod je společným bodem pro grafy 1 a 2, zelený bod je společný pro grafy 2, 3 a 4, základním grafem je 1, jeho potomkem je 2 a potomky grafu 2 jsou 3 a 4.

Vzhledem ke způsobu, jakým se grafy staví, můžeme říci, že papilární linie má koncové body tam, kde jsou v její grafové reprezentaci hlavy a ocasy grafů s jediným výskytem.

3.3 Srovnání detekčních algoritmů

Oba detekční algoritmy jsou založeny na záplavovém algoritmu prohledávání obrazu a není těžké ověřit, že v časové náročnosti jsou srovnatelné, protože oba pracují v lineárním čase vzhledem k počtu pixelů vstupního obrazu, resp. k počtu pixelů linií kostry. Díky obarvování pixelů totiž žádný bod není prozkoumán více než jednou.

Paměťová náročnost je vyšší u algoritmu s grafem směrnic, protože je třeba ukládat body do pomocné datové struktury. Tato složitost je lineární oproti složitosti konstantní, kterou disponuje první jednoduchý algoritmus. Pomocná datová struktura nám ale dál poslouží k výpočtu směrnic a ještě k dalším úpravám, které popisují následující kapitoly.

Protože časová náročnost v tomto případě nerozhoduje a lineární paměťová náročnost je únosná vzhledem ke skutečnosti, že máme připravena data k dalším operacím, budeme dále pracovat s výsledkem detekce významných bodů pomocí algoritmu s grafy směrnic, tedy s množinou sad grafů směrnic (jedna sada odpovídá jedné papilární linii, respektive ploše dosažitelné průchodem přes papilární linii z jednoho jejího bodu).

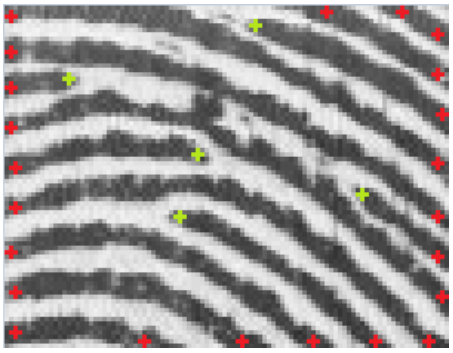
3.4 Výpočet směrnic

V grafu směrnic definujeme směrnici uzlu, jako vektor směřující od jeho předchůdce uzlu k následníkovi (tedy přes jeden uzel mezi nimi). Tento postup výpočtu byl zvolen kvůli jemnějšímu rozlišení směrů oproti směrnici mezi dvěma sousedními body. Vektory jsou dvourozměrné a jsou automaticky při vytvoření převedeny na ekvivalentní vektory o jednotkové délce. Jestliže uzel nemá následovníka nebo předchůdce, použije se aktuální uzel místo chybějícího. Směrový vektor mají všechny uzly grafu směrnic.

Již v binarizačním algoritmu používáme takzvané *pole směrnic*, které uchovává informace o směru linií ve čtvercových oknech. My jsme se rozhodli naimplementovat vlastní způsob dodatečného výpočtu směrnice, protože původní hodnoty nemusí nutně reflektovat přesný směr linie. Počítají se nad celými okny a my se chceme v této části vyhnout zaokrouhlování.

3.5 Rozlišení okrajových bodů

Při pohledu na obraz otisku prstu si musíme uvědomit, že ne všechna zakončení papilárních linií jsou skutečná. Okraje otisku ve skutečnosti jen limitují sledovanou plochu prstu, ale linie, které zde končí pravděpodobně na skutečném prstu pokračují dále. Detekce koncových bodů nám ale tyto body označí za významné, přestože nejsou. Je třeba tyto body rozlišit a vyloučit ze seznamu významných. K tomuto účelu nám poslouží jejich směrnice.



Obrázek 3.2: Zobrazení detekovaných bodů (červené jsou falešné - okrajové; zelené jsou skutečné)

Obrázek 3.2 znázorňuje, jak by mělo vypadat rozdělení detekovaných bodů podle toho, jestli jsou okrajové nebo vnitřní. Pro tento účel je v naší aplikaci implementován postup, který využívá směrnice a kostru.

Detekce okrajových bodů funguje tak, že se linie prodlouží ve směru směrnice koncového bodu (je nutné zajistit, aby směrnice koncového bodu směřovala od linie pryč) a pokud se toto prodloužení protne s jinou linií, jedná se o vnitřní bod, pokud se protne až s okrajem obrazu, jedná se o vnější bod. Tento postup vychází z pozorování obrazů otisků prstů, kdy je zřejmé, že linie jsou v naprosté většině případů ohnuté, tedy nejsou přímé. Pokud máme v otisku přímou linii, tak je velmi často obklopena jinou. Může však nastat situace, kdy označíme vnitřní bod za vnější a pro další operace ho tak zapřeme. Tento případ není příliš častý (viz kapitola 3.6.4) a pokud je otisk dostatečně velký, rozuměj „obsahuje dostatek významných bodů“, můžeme několik z nich zanedbat.

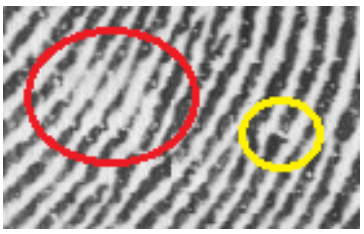
3.6 Optimalizace

Vedle výše zmíněných postupů implementovaných k získání významných bodů v otisku prstu, jsou v aplikaci *DaktylDB* implementovány i některé pomocné akce, které zlepšují mezivýsledky operací a umožňují tak lepší přiblížení k ideálnímu případu.

3.6.1 Redukce falešných spojení linií v grafech směrnic

Protože obraz otisku prstu může být nekvalitní, mohou se na něm vyskytnout chyby způsobené právě kvalitou snímání. Tyto chyby mohou být v zásadě tří typů. Buď může být v otisku „jizva“ - ostré krátké přerušení papilární linie – nebo se

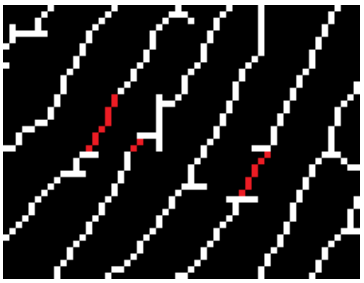
část otisku může naskenovat s nízkým kontrastem a nebo může být rýha mezi liniemi narušená nečistotou nebo rozmazáním při skenování. Všechny zmíněné chyby zobrazují obrázky 3.3 a 3.4.



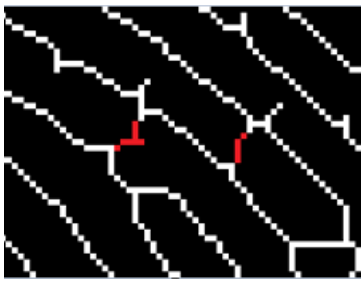
Obrázek 3.3: Chyby snímání (červeně oblast nízkého kontrastu; žlutě falešné přerušení – jizva)

Obrázek 3.4: Chyby snímání (vyznačená oblast obsahuje nečistotu mezi liniemi a rozmazání)

Uvedené chyby snímání se mohou přenést až do kostry otisku a vznikne tak falešná přerušení (obrázek 3.5) nebo naopak spojení linií (obrázek 3.6).



Obrázek 3.5: Falešné přerušení (červené body v kostře chybí)



Obrázek 3.6: Falešné spojení (červené body přebývají)

Omezení následků falešného přerušení se budeme zabývat v kapitole 3.6.3.

Redukce falešného spojení spočívá ve sledování průběhu kostry při stavbě jí odpovídajícího grafu směrnic a porovnávání aktuálního bodu s originálním obrazem, respektive s barvou aktuálního pixelu. Abychom se maximálně vyhnuli zaokrouhlovacím chybám, které vznikají při konverzích obrazu, použijeme pro porovnávání přímo originální vstup (tedy formát pixelu je RGB) a zjistíme hodnotu takzvané pozorované světlosti tohoto pixelu. Jedná se o přepočet hodnot RGB do odstínů

šedi. Tím získáme hodnotu, u které můžeme rozhodnout, jestli je příliš světlá na to, aby byla součástí linie, nebo naopak dost tmavá, aby linii náležela. Snažíme se tak ještě zlepšit výsledky binarizace.

Výpočet pozorované světlosti se provádí podle vzorce 3.1. [W3C]

$$\frac{(R \times 299) + (G \times 587) + (B \times 114)}{1000} \tag{3.1}$$

Takže nejprve projdeme vstupní obraz (pouze jednou při inicializaci analýzy kostry) a zjistíme průměrnou hodnotu světlosti, tuto hodnotu pak používáme jako mez pro určení povahy konkrétního bodu při analýze kostry a stavbě grafu. Pokud je hodnota světlosti bodu nižší než mez, nepřidáme tento bod do grafu.

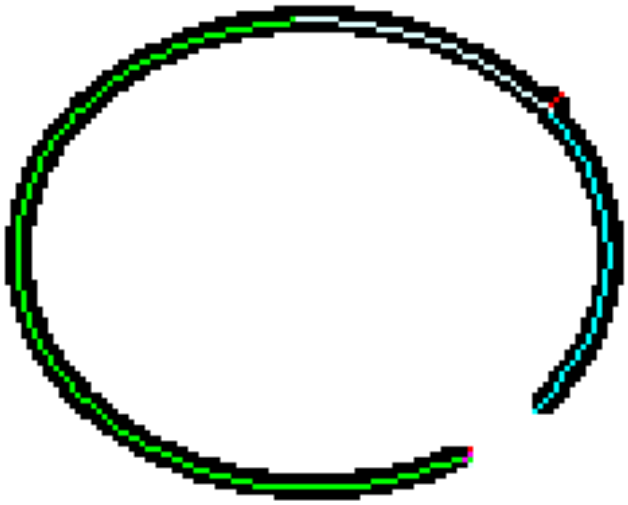
3.6.2 Sloučení grafů se společným bodem

Protože chceme dosáhnout ideálního případu, kdy můžeme prohlásit, že každý okrajový bod grafu (hlava či ocas) je buď okrajovým bodem otisku nebo významným bodem papilární linie, musíme se vrátit ke stavbě grafu a uvědomit si, že stávající způsob stavby dovoluje označit i vnitřní vrchol linie jako koncový. To je způsobeno tím, že se buď graf této linie stavěl od jejího vnitřního bodu a tudíž se vytvořily dva grafy, které se překrývají v onom bodě, a nebo bylo při stavbě nalezeno rozcestí a opět se opakuje situace s několika grafy a společným bodem.

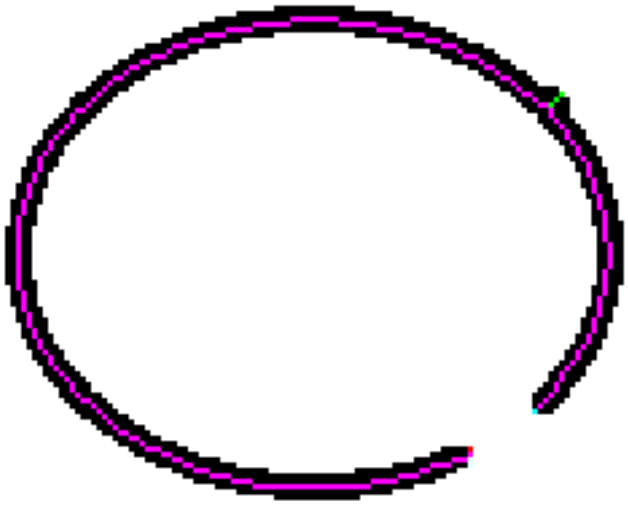
Díky tomu, že datová struktura grafu směrnic udržuje reference na potomky, můžeme jednoduše ověřit, které grafy mohou být spojeny. Tento proces spojování grafů je prováděn rekurzivní funkcí, která se nejprve zavolá nad každým potomkem aktuálního grafu. Potom se prodloužení potomci seřadí podle délky (počtu uzlů) od nejdelšího po nejkratší a postupně se snaží spojit s aktuálním grafem. Pokud se nějaký potomek spojí, nevylučuje se možnost spojení jiného, protože ten může mít společný bod s otcovským grafem na jeho druhém konci. Funkce tedy upravuje objekt, na kterém je volána. Návratovou hodnotou je seznam těch potomků, kteří nebyli spojeni včetně potomků z nižších vrstev. Srovnání grafů před spojením a po spojení ukazuje obrázek 3.7 a 3.8. Pro přehlednost zobrazeno na jednoduchém testovacím vzoru.

3.6.3 Redukce krátkých větví grafů směrnic

Při pohledu na kostru vidíme, že na ní vznikly krátké slepé větve (obrázek 3.9), které jsou způsobené nepravidelností linie vstupního obrazu. Protože jsme v kapitole 3.6.2 provedli spojení grafů s nejdelšími potomky, dostaneme zmíněné

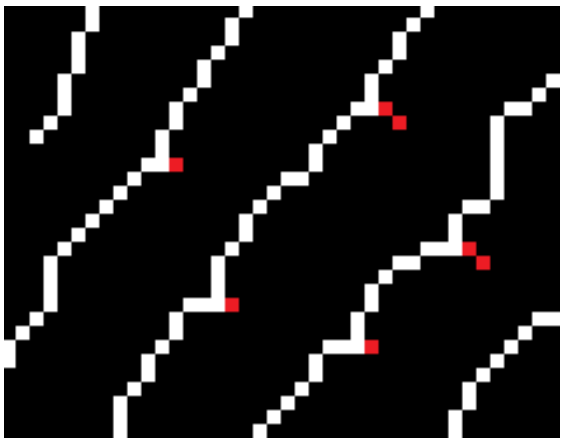


Obrázek 3.7: Výsledek neoptimalizované stavby grafu (každá barva představuje jeden samostatný graf, černá je vstupní linie)



Obrázek 3.8: Výsledek spojení grafů se společným bodem (každá barva představuje jeden samostatný graf, černá je vstupní linie)

krátké větve v seznamu nespojených. Tento seznam tedy vyčistíme od krátkých slepých linií pomocí porovnání délky – příliš krátká větev bude zahozena. Výsledný graf je znázorněn na obrázku 3.10.



Obrázek 3.9: Krátké slepé větve (vyznačeny červeně)

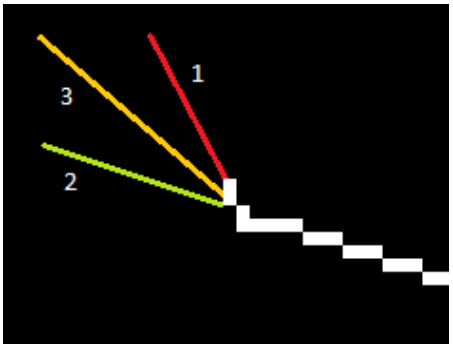
Obrázek 3.10: Výsledek filtrace krátkých slepých větví (výsledkem je jediný graf)

3.6.4 Průměrování koncových směrnic

Tvorba kostry také vytváří nechtěné odchylky od skutečného průběhu linie. Tyto artefakty jsou zejména patrné jako malá rozdvojení (křížení typu Y) na koncích linií. Protože jsme provedli sloučení potomků grafu a pak redukci krátkých větví, z tohoto rozdvojení nám zbude jen ostřeji zahnutý konec jak ukazuje obrázek 3.11.

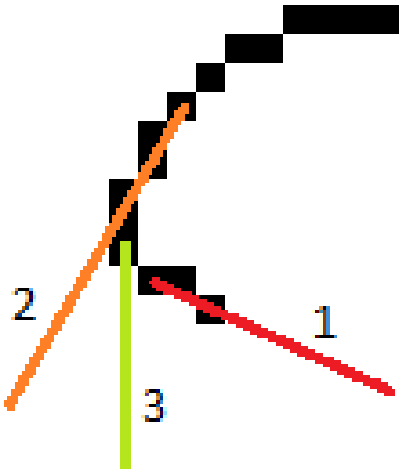
Obrázek 3.11: Zakončení linie před (vlevo) a po redukci krátkých větví (vpravo)

Protože ale používáme směrnice k detekci okrajových bodů a budeme dále v programu používat i k dalším účelům, potřebujeme co nejpřesnější aproximaci tohoto prvku. Následuje obrázek 3.12, který zobrazuje rozdíly mezi směrnicí koncového bodu, skutečnou pozorovatelnou směrnicí linie v části dostatečně blízké konci a průměrnou hodnotou směrnice těchto dvou. Tato ilustrace jasně ukazuje, že rozdíly směrnic způsobené nežádoucím zakřivením na konci linie, jsou velmi výrazné a tedy nezanedbatelné.



Obrázek 3.12: Rozdíly ve směrnicích (1 - směrnice koncového bodu; 2 - očekávaná směrnice vzhledem k průběhu linie; 3 - průměr 1 a 2)

Jednoduchým řešením tohoto problému je použití průměru koncové směrnice s hodnotou z předchozího průběhu linie. Je nutné podotknout, že nemůžeme vytvářet průměry směrnic pro celé grafy, ale jen pro jejich konce. To proto, že se nám do koncového zkreslení promítne i vnitřní průběh linie. Zároveň je ale třeba nedávat konci linie plnou váhu, protože pak nám směrnici může zkreslit drobná, ale častá, nepravidelnost (např. zbytek po rozdvojení).



Obrázek 3.13: Nechtěné zkreslení směrnice ořezem konce linie (1 - směrnice koncového bodu; 2 - směrnice konce linie po zanedbání posledních 4 bodů; 3 - skutečná očekávaná směrnice)

Aplikace *DaktylDB* implementuje výpočet průměru směrnice několika způsoby. Varianty výpočtu se rozlišují podle váhy operandů a podle hloubky zanoření do grafu.

Pseudokód *avgDirectionByNext* ukazuje výpočet průměru směrnice koncového bodu (pro jednoduchost se omezíme na dopředný směr od hlavy grafu k ocasu – použití následující položky; pro opačný konec grafu je výpočet triviální obměnou).

Varianty výpočtu tedy vznikají díky parametrizaci hloubky zanoření rekurze a díky řízení váhy při výpočtu průměru. Průměr se počítá pomocí rekurzivní formule 3.2 pro klouzavý exponenciální průměr. [WIKIb]

$$S_0 = X_0; \forall i > 0 : S_i = S_{i-1} \times (1 - \alpha) + X_i \times \alpha \tag{3.2}$$

Algoritmus 2 avgDirectionByNext

```
1: function avgDirectionByNext(maxDepth)
2:     if maxDepth < 1 then
3:         return this.vector
4:     if this.next! = null then
5:         nextVector ← this.next.avgDirectionByNext(maxDepth − 1)
6:     else
7:         nextVector ← this.vector
8:     vectorAVG ← weightedVectorAVG(this.vector, nextVector)
9:     return vectorAVG

10: function weightedVectorAVG(new, old)
11:     return old ∗ (1 − CONSTANT) + new ∗ CONSTANT
```

Tento výpočet pro hodnotu koeficientu 0,5 dává výsledek běžného průměru dvou hodnot. Koeficient nám umožňuje řídit míru ovlivnění aktuální hodnoty předešlou (to platí rekurzivně pro ostatní předcházející hodnoty).

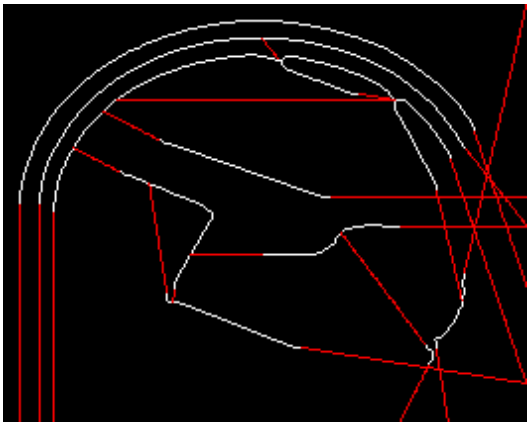
Díky použití klouzavého průměru a řízení hloubky rekurze můžeme ovlivnit, jakou měrou se podílí směr předchozích bodů na koncové směrnici. Výsledky několika konfigurací uvádí tabulka 3.1. Hloubku rekurze značí D a koeficient je hodnota A.

Porovnáním různých výstupů pro různé konfigurace nad testovacím obrazem můžeme vidět, že je výhodné vnořit se hlouběji do rekurze (D=10) a váhu průměru přenést spíše k hodnotám předchozích bodů (A=0,3). Ačkoliv se lehce do směrnic promítne zakřivení předchozího průběhu, tak se na jiných místech dokážeme vyhnout zkreslení kvůli „háčku“ na konci linie.

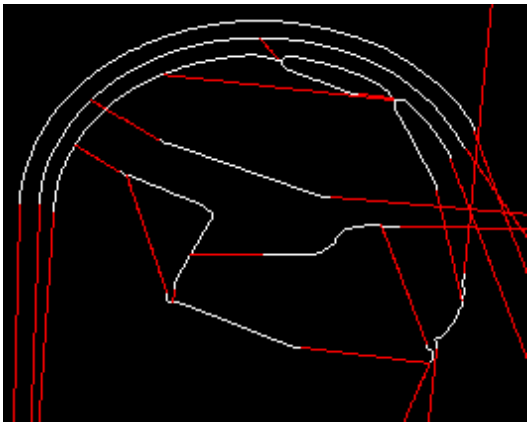
3.6.5 Redukce falešného přerušení linie

Protože při snímání vznikají chyby a dochází tak přerušení linií v místech, kde by neměly být (viz obrázky 3.3 a 3.4 v kapitole 3.6.1), vzniká v otisku mnoho falešných bodů. Pozorováním otisků prstu zjistíme, že umístění dvou konců linií přímo naproti sobě navíc v těsné blízkosti je velmi nepravděpodobné. Vyloučením těchto bodů tedy ztratíme jen velmi malou až nulovou část významných bodů.

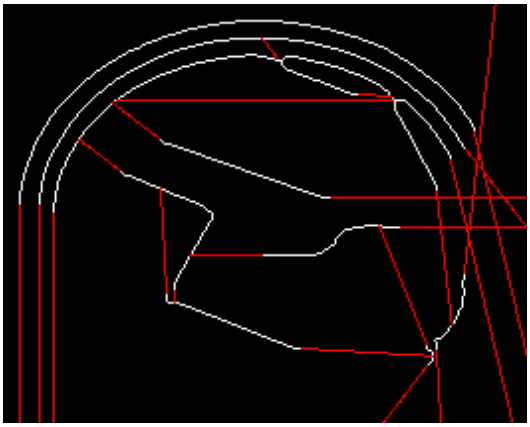
Redukci provedeme tak, že jednoduše projdeme všechny dvojice grafů a otestujeme jejich okrajové body. Jestli jsou dva okrajové body dvou různých grafů dostatečně blízko (maximálně šířka jedné linie a jedné rýhy mezi liniemi), můžeme



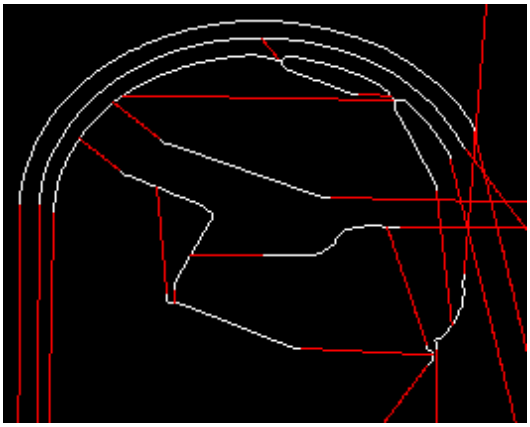
D=3; A=0,3



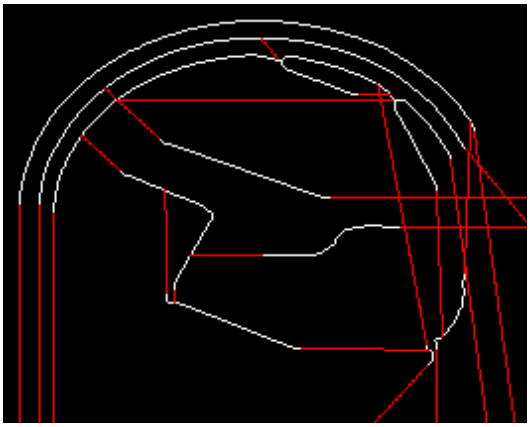
D=10; A=0,3



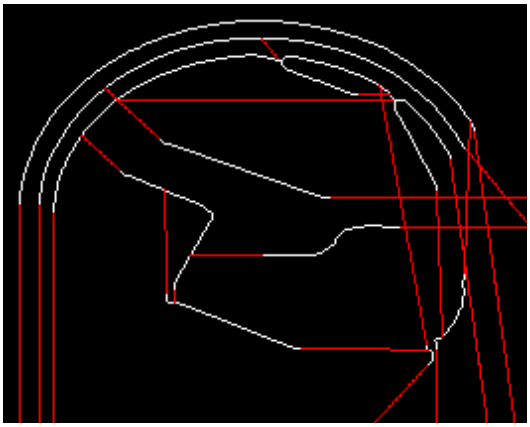
D=3; A=0,5



D=10; A=0,5



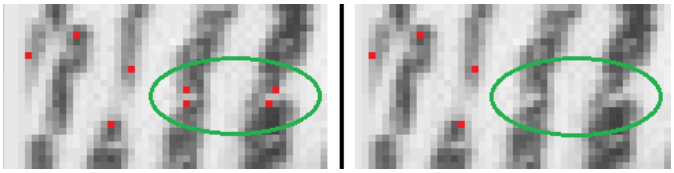
D=3; A=0,7



D=10; A=0,7

Tabulka 3.1: Výsledky různých konfigurací výpočtu průměru koncové směrnice (červeně jsou vyobrazeny směrnice linií)

tyto grafy spojit (zrušit tak blízké okrajové body). Po spojení ale musíme znovu otestovat celou sadu protějšků k upravenému grafu (spojení probíhá tak, že připojíme jeden graf ke druhému) a musíme z testů vyloučit ten graf, který byl připojen (stal se částí druhého grafu). Pro upřesnění: Máme ukazatele $G1$ a $G2$ na dva různé grafy, u nichž platí, že ocas $G1$ je dostatečně blízko hlavě $G2$. Připojíme tedy graf $G2$ přímo za graf $G1$ a přeznačíme prvky $G1$ (nová hlava = hlava $G1$; nový ocas = ocas $G2$). Pak $G2$ je ukazatelem na část grafu $G1$ a tudíž tento ukazatel musí být vyloučen ze seznamu grafů. Následující obrázek 3.14 ukazuje výsledek tohoto postupu.



Obrázek 3.14: Ukázka vstupu (vlevo) a výstupu (vpravo) redukce falešných přerušení linií (červeně jsou koncové body grafů; zeleně je označena opravená oblast)

3.6.6 Vyřazení křížení a rozdvojení

Kvůli jednoduchosti práce jsme se rozhodli, že budeme využívat jen konce linií. Aktuální implementace detekce významných bodů ale umožňuje najít i křížení a rozdvojení linií. Body v těchto částech otisku tedy musíme odstranit z výsledné sady bodů. Pro tento účel použijeme příznak uzlu grafu, který určuje, jestli je uzel bodem křížení nebo ne. Při stavbě grafu jsme rozlišili situaci, kdy jsme narazili na více možností pokračování. V takovém případě označíme společný uzel vytvářených grafů jako křížení. Po dokončení celého procesu detekce významných bodů tyto uzly zanedbáme při přenosu informací z grafů do výsledné sady uzlů.

3.6.7 Práce s bitmapou

Protože se v celé této kapitole věnujeme práci nad bitmapami (kostra i původní vstup), používá aplikace pomocnou třídu *LockBitmap*, která dovoluje úpravu této bitmapy ve zrychleném, takzvaném *unmanaged* režimu. Tento režim je implementován tak, že při začátku práce s bitmapou vytvoříme instanci příslušné třídy uzamykatelné bitmapy a předáme jí naší původní bitmapu. Pak zavoláme funkci *LockBits*, která zkopíruje obsah původní bitmapy do bytového pole a uzamkne ji pro čtení i zápis. Následující operace úprav a čtení pixelů se provádějí nad bytovým

polem a po skončení práce se voláním *UnlockBits* zpět překopírují upravené hodnoty a původní bitmapa se odemkne. Tento postup mnohonásobně urychluje práci s bitmapou, protože odstraní kontroly prováděné .NET frameworkem při přístupu k jednotlivým pixelům.

3.6.8 Shrnutí optimalizací

Všechny výše uvedené optimalizace mají za cíl přiblížit výsledek detekce významných bodů a směrnic ideálu. Ideálním výsledkem rozumíme takový stav, kdy máme označeny všechny vnitřní koncové body papilárních linií a jejich směrnice přesně kopírují konec papilární linie. Je třeba si ale uvědomit jakým způsobem bude probíhat práce s výsledky této detekce dále. Protože máme za cíl vyhledávání podobných otisků v databázi, je nutné minimalizovat počet falešně pozitivních bodů, to znamená, že chceme, aby nalezené body byly skutečně reálné. Když totiž budeme porovnávat otisky s databází, přítomnost některých falešných bodů by mohla vyloučit pravý výsledek. Takže tím, že některé optimalizace mohou z výsledné sady bodů odstranit několik pravých, říkáme, že za cenu větší spolehlivosti můžeme rozšířit množinu „podobných“ výsledků, které mohou být při vyhledávání relevantní.

3.7 Přesnost detekce

Ke zjištění přesnosti detekce byl pro svou spolehlivost použit proces binarizace vstupu konvexním práhováním. Vstupy jsme rozdělili na kvalitní (označeno H) a méně kvalitní (nižší kontrast a spojitost linií; označeno L). U méně kvalitních vstupů byla zároveň zanedbána část, která má pro analýzu příliš nízký kontrast. Výsledky ukazuje tabulka 3.2.

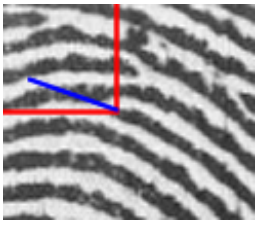
Vstup	Pravé	Falešné	Nenalezené
H 1_1	23	18	1
H 1_3	22	1	1
H 1_3_rot_10cw	23	12	1
L 10_1	14	41	5
L 10_2	17	22	6

Tabulka 3.2: Porovnání výsledků detekce významných bodů nad různými vstupy. Pravé body jsou ty, které byly nalezeny a vizuálně potvrzena jejich pravost. Falešné jsou ty, které byly nalezeny, ale při vizuální kontrole byly vyvráceny. Nenalezené jsou ty body, které algoritmus nenašel, ale vizuálně jsou prokazatelné.

Podle tabulky 3.2 vidíme, že algoritmus je správně nastaven na vyhledání maximálního počtu bodů. Docílili jsme tedy toho, že zejména na kvalitním vstupu nám unikne jen malé procento markantů. Na druhou stranu je zde poměrně velký počet falešných bodů, které zkreslují výsledek.

4. Reprezentace významných bodů

V této části práce analyzujeme fakta, která mají vliv na reprezentaci nalezených bodů otisku. Jedná se zejména o změny, které mohou být pozorovatelné při porovnání dvou rozdílných snímků jednoho otisku. Pro takové případy učiníme několik pozorování a jejich důsledky nám pomohou vytvořit reprezentaci, která bude invariantní vůči takovým změnám.



Obrázek 4.1: Významný bod má vyznačeny souřadnice osy X a Y (počátek soustavy je levý horní roh obrázku) červeně a směrnice je vyobrazena modře (úhel je relativní vzhledem k ose X)

4.1 Problémy při získávání otisků

Otisky prstů se získávají různými způsoby – skenováním na digitální čtečce, vyfotografováním zvýrazněného otisku na předmětu nebo jeho skenováním z papírového formuláře. Při snímání otisku pak může nastat několik různých situací, nebo jejich kombinace, které ztěžují práci automatizovaným porovnávacím systémům. Druhy těchto situací jsou popsány v následujících bodech. [MALT12]

4.1.1 Rotace

Při snímání kontrolního otisku se může stát, že se prst přiloží v jiném úhlu natočení než tomu bylo při předchozím sejmutí. Tedy je otisk pootočen. V takovém případě je buď nutné vytvořit reprezentaci otisku, která je odolná vůči pootočení a nebo vytvořit algoritmus, který otočení detekuje a dále se podle něj zachová (vrátí vstup do očekávané polohy nebo připočítá úhel pootočení při výpočtech). Musíme si však uvědomit, že rotaci je velmi složité detekovat – otisk prstu sice můžeme klasifikovat podle základních tvarů, ale nuance v náklonu tohoto tvaru jsou stále možné.

4.1.2 Posunutí

Posunutí je jev, kdy subjekt položí prst na čtečku v jiné rotaci podle osy opisující kost prstu a nebo podle osy, která je na kost kolmá a rovnoběžná s podložkou. Takže můžeme říct, že výhledové okno nad celou plochou článku prstu je posunuto. Pokud se jedná o malé posunutí – myšleno „takové posunutí, aby na průniku s původním otiskem byl dostatek významných bodů k porovnání“ – není tento jev příliš závažný.

4.1.3 Deformace

Deformace otisku prstu nastane tehdy, působí-li subjekt na svůj prst při snímání otisku nerovnoměrným tlakem či tahem. Kůže je totiž pružná a může se tak stát, že výsledný otisk je deformovaný – nahoře zúžený a dole rozšířený či naopak nebo může být deformace patrná jako pootočení středu otisku vůči okraji. Tyto deformace bývají jen velice jemné, ale je třeba na ně myslet při návrhu porovnávacích algoritmů. Díky deformaci se totiž může stát, že při porovnání dvou dvojic bodů, resp. jejich vzájemné polohy, bude jejich rozdíl vzdáleností nerovnoměrný.

4.1.4 Zvětšení

Obraz otisku prstu může být pořízen v různém rozlišení a tedy můžeme mít různě velký vstupní otisk s různou jemností linií. Tento problém je kompenzovatelný stanovením minimální velikosti vstupu a konverzí všech vstupů na tuto velikost.

4.1.5 Částečný otisk

Zejména v kriminalistice se často setkáme s částečným otiskem, protože otisky prstů sňaté z předmětů mohou být jen na okraji předmětu nebo mohou být z části příliš rozmazané. Jedná se o takový otisk prstu, který zobrazuje jen malou část celé plochy článku prstu. Pro tento případ je nutné testovat, jestli vstupní otisk obsahuje dostatečný počet významných bodů.

4.2 Analýza problémů při získávání otisku

Protože je nutné vzít do úvahy všechny možnosti, které otisk prstu deformují či jinak znehodnocují jeho kvalitu, musíme učinit několik pozorování.

P1 Souřadnice významného bodu na vstupním obraze nemůže být brána jako údaj určující charakteristiku tohoto bodu, protože jiný obraz stejného otisku může

být posunutý, rotovaný či deformovaný a v takovém případě se může poloha bodu významně změnit.

P2 Vzdálenost mezi dvěma body nemůže být brána jako charakteristika těchto bodů, protože posun může zapříčinit zmizení jednoho z bodů a deformace může měnit vzdálenosti nerovnoměrně v jediném otisku.

P3 Vzájemná poloha dvou bodů ve smyslu směru jejich spojnice podléhá rotaci celého vstupu.

P4 Směrnice bodu (směr konce papilární linie) podléhá rotaci celého vstupu.

4.3 Převod významného bodu na kód směrnice

Datová struktura použitá v aplikaci *DaktylDB* k reprezentaci významného bodu otisku prstu byla navržena s ohledem na všechna pozorování uvedená v kapitole 4.2. Jedná se o strukturu obsahující dvě celočíselné položky k uchování souřadnic bodu v otisku (použito později při klastrování pro výpočet další hodnoty) a jednu další celočíselnou položku, kterou je kód směrnice bodu.

Kód směrnice je pevně definovaný index bodu v okolí významného bodu, ke kterému směřuje směrnice. Tyto indexy zobrazuje obrázek 4.2.

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

Obrázek 4.2: Rozpis indexů směrů (černý střed značí aktuální bod; modrá část definuje indexy pro směrnice s nízkým rozlišením; žlutá definuje indexy pro normální směrnice)

Protože jsme výpočty směrnic prováděli pokud možno nad dvěma body, které nebyly přímo přilehlé, ale měly mezi sebou ještě jeden další bod, můžeme tyto směrnice označit relativním indexem podle uvedeného klíče z obrázku 4.2. Směrnice vytvořené nad kratším úsekem mají také svůj index, který by odpovídal delší

směrnici o stejném průběhu. Takže všechny směrnice významných bodů jsou porovnatelné.

Výsledkem tedy je taková reprezentace významného bodu, která má dvě souřadnice (x, y) a třetí číselnou hodnotu jíž je index směru. S touto reprezentací bodu budeme dále pracovat.

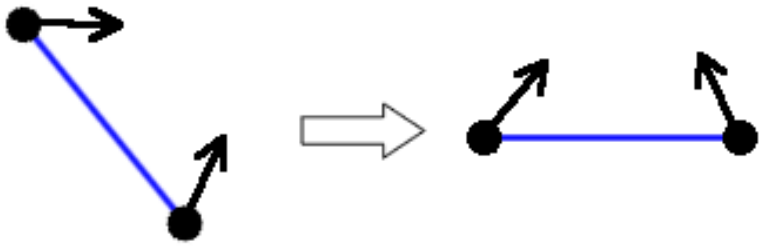
4.4 Klastrování bodů a rozdělení klastrů do tříd ekvivalence

Pozorováním v kapitole 4.2 jsme zjistili, že samostatné body otisku se směrnicemi nejsou příliš vhodné pro charakterizaci otisku. Abychom tedy mohli otisk nějak charakterizovat, vytvoříme malé klastry. Název „klastr“ se sice běžně používá pro označení větší skupiny nějak souvisejících prvků (např. skupina blízkých bodů v rovině), ale my jej použijeme pro označení dvojice významných bodů otisku. Jeden otisk prstu tedy bude mít mnoho klastrů. Vytvoříme je ze všech dvojic bodů. Dvojice vytváříme tak, že nám nezáleží na pořadí prvků ve dvojici, tedy dvojice *(b,a)* je stejným klastrem jako *(a,b)* a tudíž ji nevytvoříme. Do klastru uložíme indexy směrnic obou bodů. Klastr tedy obsahuje dvě celočíselné hodnoty z intervalu $\langle 0; 15 \rangle$. Označme je $d1$ a $d2$.

4.4.1 Třídy ekvivalence klastrů

Protože chceme zabránit, aby rotace otisku měla vliv na uložené hodnoty, musíme klastry rozdělit do tříd. Je nutno si také uvědomit, že rotace otisku mohla způsobit, že jsme klastr vytvořili z bodů v opačném pořadí než bylo očekáváno.

Z celého prostoru 16×16 možných kombinací směrových indexů musíme získat bázi, pro kterou platí, že nezáleží na pořadí indexů v klastru a že platí také ekvivalence pro všechny rotace jednoho typu klastru. Množinu všech rotací omezíme na dvě tím způsobem, že při vytvoření klastru vypočítáme směrnici z prvního do druhého bodu a klastr pootočíme tak, aby tato směrnice byla v horizontálním směru. Toho docílíme tak, že přičteme k $d1$ i $d2$ tolik jednotek indexu, kolik jich bylo potřeba přičíst ke směrnici mezi body, aby se dostala do horizontálního směru (indexy 6 a 14). Tuto operaci ilustruje obrázek 4.3.



Obrázek 4.3: Eliminace rotace klastru - modrá spojnice mezi dvěma body se přesune do horizontální polohy, přičemž úhel, který svírá se směrnicemi bodů zůstane stejný

Pozorování (pravidla bázového klastru)

PK1 Pro klastr $(d1, d2)$ platí: $d1 < 8 \wedge 0 \leq d2 \leq 15$ (všechny klastry $(d1, d2)$, kde $d1$ je větší než 7 jsou ekvivalentní klastrům tvaru $((d2 + 8) \bmod 16, d1 - 8)$ – rotace klastru o 180°)

PK2 Pro klastr $(d1, d2)$ platí: $d1 \leq d2$ (nechceme, aby záleželo na pořadí – vždy indexy v klastru seřadíme podle hodnoty indexu)

PK3 Dle PK1 a PK2 platí, že bázových klastrů je 100 ($\sum_{i=0}^{7} \sum_{j=i}^{15} 1 = 100$ kde první suma reprezentuje hodnoty, které nabývá $d1$ a druhá zase hodnoty pro $d2$)

Díky pozorování PK1 můžeme klastry sjednotit do tříd ekvivalence podle bázových klastrů a těmto třídám přiřadit index třídy klastru. Třídy klastrů setřídíme primárně podle $d1$ a sekundárně podle $d2$ a očíslujeme od nejmenšího po největší čísly $0 - 99$. Nyní můžeme každý klastr vyjádřit jediným číslem. To znamená, že jsme vytvořili prostor o 100 dimenzích. Navíc víme, že v otisku je malý počet náhodných znaků (dle literatury $40 - 100$ [STRA05]), což nám umožní shora omezit počet klastrů na 255 (vyšší počty budou oříznuty na tuto hodnotu). Tím pádem můžeme uložit počty klastrů v otisku do číselného typu *byte* a celkově jsme vytvořili prostor s dimenzí 100 a rozsahem hodnot 256 v každé z nich. To dává dohromady celkem $2^8 00$ různých kombinací, které můžeme pomocí naší reprezentace rozlišit.

4.5 Reprezentace celého otisku

Definovali jsme si klastry významných bodů otisku. Pro jeden otisk získáme množinu významných bodů a z ní vytvoříme množinu klastrů, resp. jejich indexů.

Protože klastrů může být hodně, ale je jich omezený počet druhů, vyplatí se nám vytvořit reprezentaci pomocí pole čítačů. Vytvoříme pole bytů o délce 100. Za klastr s indexem X přičteme jedna na odpovídající pozici X v našem poli. Získáme tak strukturu, kterou budeme nazývat „kód směrnic otisku“.

Kód směrnic otisku je pole bytů o délce menší, než je rozsah jednoho bytu, takže můžeme pro účely uložení v databázi toto pole jednoduše serializovat na posloupnost dvojic bytů – index, hodnota. Kód otisku nám pak bude sloužit k porovnávání otisků, protože díky němu známe, kolik klastrů kterých typů by měl obsahovat hledaný otisk.

4.6 Srovnání s kódem MCC

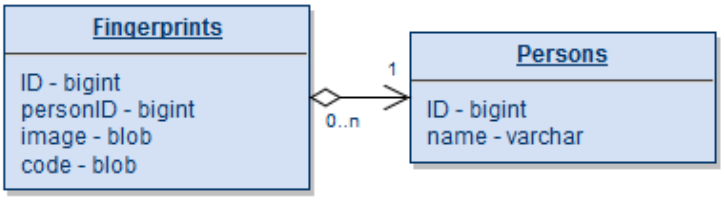
Kód MCC (*Minutia Cylinder-Code*) je způsob uložení signifikantních informací o otisku prstu, který využívá vlastností bodů k vytvoření informace o jednotlivých podmnožinách otisku. Tyto podmnožiny jsou kruhové a vytvářejí se pro každý bod. Každá taková kruhová podmnožina se nazývá *cylindr* a uchovává informace jak o svém středovém bodu, tak i o jeho sousedech, které se v daném cylindru nacházejí. [CAPP10]

Hlavní výhodou MCC oproti ostatním je fakt, že je reprezentovatelný jedním vektorem hodnot, které mohou být binarizovány. Tedy lze s tímto vektorem pracovat jako s bitovým polem. Navíc kód nese dostatek informací k opětovné rekonstrukci otisku. [FERR12] Nevýhodou naopak je, že užívá ke svému výpočtu fixní hodnotu průměru cylindrů (absolutní či relativní vůči síle linie).

Ve srovnání s kódem směrnic je MCC lepší pro porovnávání otisků $1 : 1$, protože porovnává větší podmnožiny otisku a hlavně není omezen nízkou dimenzí, takže poskytuje lepší přesnost a jistotu. Na druhou stranu přímočarost kódu směrnic otisku umožňuje sestavení implementačně jednoduchého indexu, který výrazně usnadní práci s rozsáhlejší databází (viz kapitola o databázovém indexu 6).

5. Organizace databáze

Protože není cílem této práce vytvořit komplexní databázi osob a jejich daktyloskopických stop, je samotná databáze implementována jednoduše a přímočaře. Ukládáme data jen v základním tvaru, aby bylo možné rozeznat výsledky při testování ostatních částí systému. Databáze je relačního typu a je vytvořena pomocí knihovny *SQLite*. Struktura databáze je znázorněna diagramem vykresleném na obrázku 5.1. Obsahuje dvě tabulky - osoby (*Persons*), otisky (*Fingerprints*) - které jsou spojeny relací 1:N vyjadřující: „jedna osoba může mít N otisků“. V rámci naší aplikace je databáze zapouzdřena modulem *Database* a její rozhraní definuje a implementuje třída *DatabaseService*, která tedy vytváří pro databázi jediný přístupový bod a funguje na principu poskytování služeb.



Obrázek 5.1: Diagram zobrazující strukturu databáze

Důležitější částí databáze z pohledu této práce je databázový index. Ten nám slouží k omezení množiny porovnávaných záznamů na minimum, aby se zrychlilo vyhledávání shodných otisků v databázi. Nejprve tedy otisk porovnáme s indexem, tím získáme množinu identifikátorů otisků, které mají podobné vlastnosti jako vstup, a ty pak porovnáváme se vstupem jednotlivě a detailně. Více informací o databázovém indexu a popis jeho implementace je uveden v kapitole 6.

6. Databázový index

Databázový index je vedle analýzy otisku druhou nejdůležitější částí daktyloskopické databáze. Jeho účel se od běžného databázového indexu liší. Nehledá totiž přesnou shodu záznamu s hledaným klíčem, ale měl by fungovat tak, že vylučuje ty záznamy, které se výrazně odlišují od daného klíče. Uvědomme si, že přímočaré vyhledávání nad databází by znamenalo porovnání všech záznamů se vstupem. To by při větším počtu položek byla časově neúnosná operace. Je tedy třeba vytvořit nástroj, který nám záznamy umožní klasifikovat tak, aby bylo možno jich co nejvíce vyloučit bez složitého porovnávání. K tomu by měl sloužit index daktyloskopické databáze. Měl by vytvářet nějakou strukturu, která bude klasifikovat vyhledávací klíče, a při dotazu by měl vrátit všechny podobné. Ačkoliv je podobnost velmi relativní pojem, musíme brát do úvahy, že přesná shoda je u tohoto typu dat velmi nepravděpodobná (viz problémy při získávání otisků v kapitole 4.1).

6.1 Analýza dat

Implementace indexu v této aplikaci těsně navazuje na předchozí kapitoly, obzvláště na kapitolu 4, a využívá jejich výsledky. Tedy pro stavbu indexu, který reprezentuje strukturu ukládající hodnoty podle klíče, použijeme jako hodnotu číselný identifikátor otisku prstu v databázi a jako klíč nám bude sloužit kód otisku.

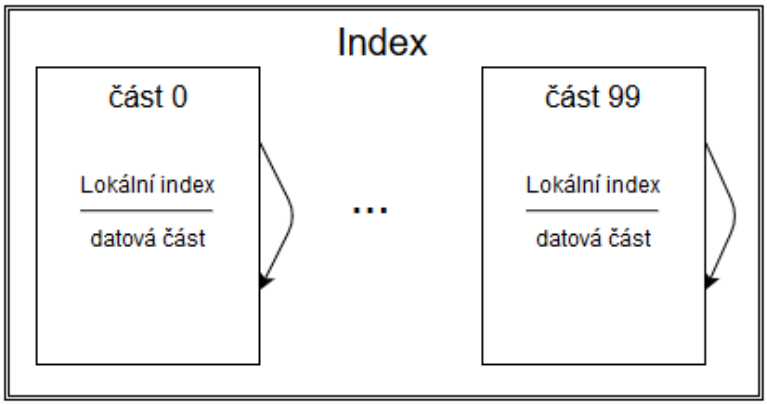
Klíč indexu je tedy podle kapitoly 4.5 definován jako pole délky 100 obsahující hodnoty z intervalu $\langle 0, 255 \rangle$. Můžeme očekávat rovnoměrné rozdělení hodnot mezi klíče. Tím získáme $2^8 00$ různých klíčů reprezentujících otisky prstů. Toto číslo výrazně převyšuje současný počet obyvatel planety země vynásobený deseti (každý člověk má deset prstů). Tedy máme k dispozici dostatečně velký prostor pro rozlišení otisků.

6.2 Dělený dvojitý index

6.2.1 Základní popis struktury

Vzhledem ke konstantní dimenzi našeho prostoru klíčů jsme se rozhodli rozdělit celý index na části, přičemž jedna část reprezentuje právě jednu dimenzi. Fakticky jsme index rozložili do sta souborů - proto název „dělený“. Soubory jsou identifikovány číslem z intervalu $\langle 0, 99 \rangle$ přesně podle rozměrů klíče. V každém souboru je pak lokální index, který slouží k rychlému zjištění adresy bloku uložených hodnot podle konkrétní části klíče - proto „dvojitý“. Pokud tedy chceme vyhledávat podle

klíče (kódu otisku), který má na pozici 21 hodnotu 130, tak použijeme soubor odpovídající číslu 21 a v něm budeme pomocí lokálního indexu hledat bloky vedené pod číslem 130. Podrobněji je tento proces popsán v následujících kapitolách.



Obrázek 6.1: Nákres zobrazující strukturu databázového indexu

Vnitřní strukturu indexového souboru jsme navrhli s ohledem na poznatky, které vycházejí z implementace klíče - kódu směrnic otisku - a z pozorování, jak by měla být databáze používána. Vycházeli jsme tedy vstříc níže uvedeným předpokladům.

- Nejčastější operací je vyhledávání. Počet vyhledávání bude výrazně převyšovat počet vložení či smazání.
- Počet záznamů odpovídající jednomu klíči není omezen.
- Rozsah klíčů je omezen.

Pro splnění těchto podmínek jsme vytvořili strukturu, kterou je možno rozšiřovat pro účely uchovávání narůstajícího počtu záznamů a zároveň je vhodná pro efektivní získávání více položek se stejným klíčem. Přizpůsobena je tomu i samotná implementace.

Jak už bylo výše zmíněno a vyobrazeno na obrázku 6.1, je indexový soubor rozdělen na dvě hlavní části - vnitřní index a datová část. Vnitřní index slouží jako mapa hodnot klíče na pozice odpovídajících záznamů v datové části (nazývejme je *bloky záznamů*). Tato mapa je tedy seznamem dvojic *(klíč, pozice)*. Pro rychlou orientaci jsou tyto dvojice setříděné podle klíče, takže celý vnitřní index je posloupností dvojic *(0, poziceX), (1, poziceY), ..., (255, poziceZ)*. Pozice je relativní, tedy je to číslo vyjadřující počet bytů od začátku datové části k odpovídajícímu bloku záznamů.

Datová část indexu je neuspořádaný seznam bloků záznamů. Tento blok obsahuje klíč, počet záznamů v něm uložených a jejich seznam. Klíč je shodný s tím, který je obsažen v jeho mapování. Počet záznamů slouží hlavně pro kontrolu naplnění bloku. Seznam uložených položek je pole pevné délky, které je setříděné kvůli možnosti binárního hledání.

Indexový soubor obsahuje ještě jednu část, která je ale potřebná jen pro implementaci, a proto ji v základní struktuře neuvádíme. Jedná se o prvních $8 + 8$ bytů v indexovém souboru, které slouží pro uchování dvou 64-bitových čísel - velikosti vnitřního indexu v bytech a počtu bloků záznamů v datové části. Tyto údaje jsou uchovávány pro zjednodušení implementace a jsou aplikací načteny do paměti vždy po otevření souboru. Rozlišujeme tedy tři základní pozice v souboru: *fileStart* (0), *indexStart* (16) a *dataStart* (*indexStart + velikost indexu*). Na tyto bude odkazováno v následujícím textu.

6.2.2 Přeplnění bloku záznamů

Protože máme blok záznamů s pevnou délkou a neomezený počet záznamů pro každý klíč, musíme určit, co se má stát, pokud je blok záznamů naplněn a my chceme vložit další záznam se stejným klíčem. V takovém případě vytvoříme nový blok, umístíme ho na konec souboru a vložíme do něj záznam. Je třeba si také zapamatovat pozici, kde se tento blok nachází a tu zaznamenat do souborového indexu. Protože už ale jeden blok se stejným klíčem existuje, musíme rozšířit souborový index o další položky. To provedeme tak, že celý index zdvojíme s tím rozdílem, že neobsazené položky mají speciální hodnotu umožňující rozlišit konec obsazené části. V tuto chvíli tedy soubor vypadá takto: 16 bytů pro implementační účely, $2 \times 256 \times 12$ bytů ($12B$ je velikost jedné mapy) a datová oblast. Protože máme obě sady mapovacích dvojic setříděné podle klíčů, můžeme přistoupit k položkám jednoho klíče k pomocí formule $\forall i \geq 0 : P = k + (256 * i)$, kde P je pozice i-té mapy pro klíč k. Množina všech takto získaných pozic je seznamem adres na bloky záznamů pro daný klíč. Samozřejmě je nutné při této expanzi indexové části zvětšit soubor a celou datovou část posunout o potřebný počet bytů.

6.2.3 Operace nad indexem

Vkládání prvků

Vložení prvku do indexu je jednou ze základních operací. Jejím vstupem je kód otisku (pole délky 100 s položkami z intervalu $\langle 0, 255 \rangle$) a 64-bitové číslo reprezentující identifikátor otisku prstu v databázi. Díky rozdělení indexu na soubory se ale můžeme zaměřit na vložení záznamu do jednoho souboru. Tedy máme na

vstupu číslo souboru, klíč (resp. jeho hodnotu v položce odpovídající aktuálnímu číslu souboru) a identifikátor otisku, který chceme do indexu vložit. Vložení celého kódu je pak opakování této operace nad různými vstupy.

Nejprve ze souborového indexu získáme pozice všech bloků záznamů odpovídajících danému klíči. Pak postupně testujeme, jestli je v bloku volné místo. Pokud ano, zatřídíme záznam do vnitřního pole a blok v indexovém souboru aktualizujeme. Pokud ne, otestujeme další blok v pořadí podle toho, jak byly načteny pozice bloků z lokálního indexu (snažíme se zaplnit první možný blok). Takto postupujeme dokud máme načteny bloky. Pokud jsou všechny bloky plné, postupujeme tak jak je uvedeno v kapitole 6.2.2.

Vyhledávání podobných záznamů

Vyhledávání probíhá podobně jako hledání vhodného místo pro vložení nového záznamu. Musíme v první řadě nalézt všechny pozice odpovídající klíči, pak je potřeba načíst záznamové bloky na těchto pozicích. Dál už se postupy liší. Při vyhledávání musíme sloučit data ze všech bloků - tím získáme jeden seznam identifikátorů otisků pro konkrétní klíč. Pokud provedeme průnik nad všemi seznamy ze všech indexových souborů, získáme sadu těch identifikátorů, které zastupují otisky velmi podobné vstupu.

Jak už jsme zmínili dříve, vyhledávací index neslouží k nalezení přesně shodného záznamu, ale k získání malé části všech záznamů, které se vstupu podobají. Důvodem je to, že s velkou pravděpodobností nebude vstup programu (druhotně získaný otisk) totožný se svým protějškem, který byl do databáze vložen. Proto by předchozí postup měl vysokou míru chybovosti v tom smyslu, že by mohl vyloučit i správný záznam. Proto musíme tento postup upravit, aby byl vůči takovému případu robustní. Docílíme toho tím, že při vyhledávání budeme brát v potaz rozptyl hodnot klíčů. Při analýze otisků se totiž mohlo stát, že program nenašel některý bod, nebo že omylem detekoval bod falešný. Vyhledávání tak provedeme pro více po sobě jdoucích hodnot klíče. Například pro klíč hodnoty 110 načteme záznamy s klíči $108, 109, 110, 111, 112$. Současně také můžeme filtr rozšířit pomocí neúplného výskytu - to znamená, že neprovedeme průnik identifikátorů nad všemi soubory, ale řekneme, že nám stačí, aby daný identifikátor byl například jen v 80% souborů.

Při hledání je třeba si uvědomit, že se snažíme vyvážit chyby, které mohly nastat v analýze otisku, respektive musíme brát do úvahy možnost nedokonalého vstupu (viz kapitola o problémech při skenování 4.1). Konkrétním nastavením rozptylů se zabývá kapitola 6.2.4.

Mazání prvků

Mazání probíhá jednoduše, bez pomocných operací typu „setřásání“, protože při vkládání se zaplňují prázdná místa a ze způsobu užívání databáze přímo vyplývá, že počet smazání je v dlouhodobém horizontu stejný, nebo spíše menší, oproti počtu vložení. Proces probíhá tak, že vyhledáme bloky, které mají přesnou shodu s klíčem, nalezneme v bloku požadovaný identifikátor a ten odebereme.

6.2.4 Implementační detaily a optimalizace

Přístup k souborům

Protože chceme využívat náhodného přístupu k částem indexových souborů a hlavně potřebujeme těchto přístupů vykonat několik během jedné operace, využili jsme konceptu „memory-mapped files“ (soubory mapované do paměti). Jejich implementaci poskytuje *.NET* framework[1]. S jejich využitím můžeme namapovat část indexového souboru do paměti a přistupovat na náhodné pozice v rámci tohoto mapování. Díky tomu se zrychlí a zjednoduší práce s indexovými soubory.

Před prací s indexem se v aplikaci vytvoří zástupné objekty pro všechny indexové soubory a automaticky si namapují oblast lokálního indexu, protože s ní nejčastěji pracují. Úseky v datové oblasti se mapují dynamicky dle potřeby.

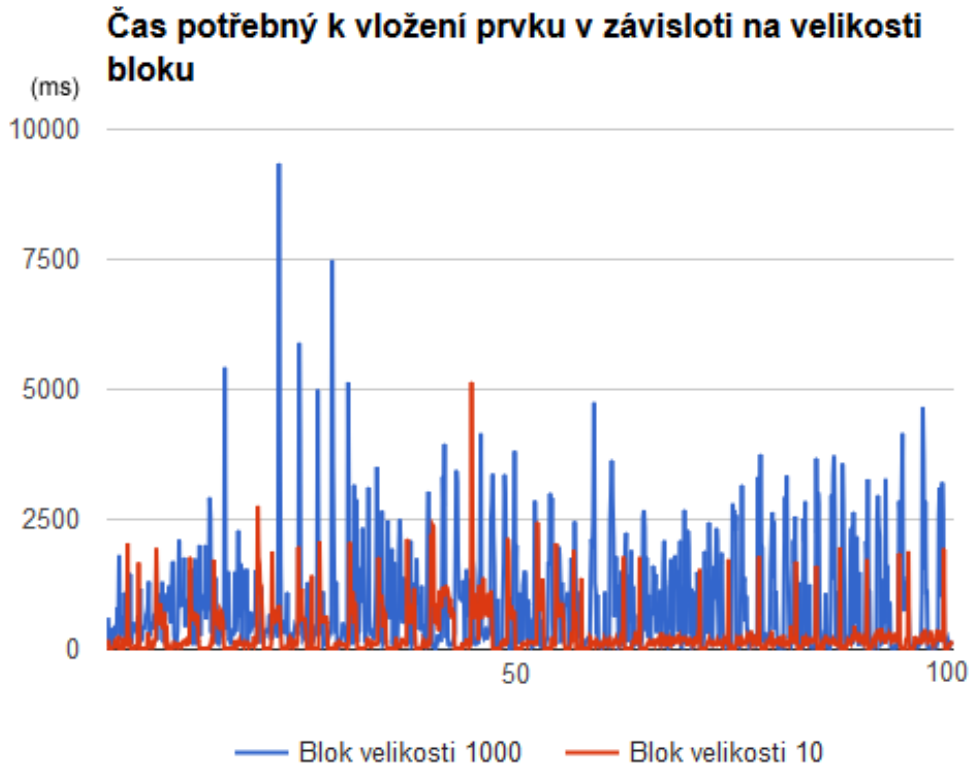
Velikosti bloků

Při implementaci bylo nutné rozhodnout, jaké velikosti se použijí pro rezervované místo lokálního indexu a pro délku pole uvnitř bloku záznamů. Po otestování více variant jsme tyto hodnoty nastavili tak, že velikost prostoru pro lokální index je 10krát větší než jedno jeho základní pole, a počet záznamů v jednom bloku je maximálně 60. Důvodem pro volbu této velikosti indexu je dostatečný prostor pro uchování přiměřeně velkého počtu záznamů (až 150000). Počet záznamů v jednom bloku jsme vybrali tak, aby se celý jeden blok vešel do stránky virtuální paměti systému (v případě systému, na kterém byl program testován, se jako optimální ukázal počet mezi 60-ti a 70-ti). Toto nastavení však může mít jinou optimální hodnotu při běhu na jiném operačním systému.

Jako potvrzení správnosti volby délky bloku záznamů nám slouží měření časů jednotlivých operací vložení do indexu s 1000 náhodnými vstupy (použit generátor pseudonáhodných čísel) nad různými variantami nastavení. Měření jsme nejprve

[1]Dokumentace na: `https://msdn.microsoft.com/en-us/library/dd997372%28v=vs.100%29.aspx`

provedli s bloky velikosti 1000, pak 100 a následně s velikostí 10. Mezi testy s hodnotami 1000 a 100 byly jen malé odchylky a průměrný čas pro vložení byl téměř totožný ($854ms$/jedno vložení s blokem velikosti 1000). Naopak při zmenšení bloků na velkost 10 se průměrný čas razantně snížil ($284ms$/vložení). Následným propočtem jsme zjistili, že v testovaném prostředí je taková velikost jedné virtuální stránky, že se do ní vejde jeden blok záznamů s polem velikosti 67. Dalšími testy jsme si ověřili, že zlom v měření nastává pro hodnoty těsně nad 70. Porovnání měřených časů zobrazuje graf na obrázku 6.2. Je nutné ale zdůraznit, že uvedené nastavení je závislé na použitém běhovém prostředí.



Obrázek 6.2: Graf porovnávající dobu vložení záznamu do indexu (osa Y) vůči velikosti bloku. Osa X zobrazuje pořadová čísla vykonávané operace.

Mazání prvků

Při implementaci mazání prvků z indexu jsme neimplementovali operaci defragmentace, protože vycházíme z předpokladu, že do databáze se bude častěji vkládat než z ní mazat. Tudíž se uvolněná místa zase zpětně zaplní.

Jako optimalizaci procesu mazání jsme implementovali hledání konkrétního záznamu v poli pomocí binárního vyhledávání.

Užití více vláken

Naše aplikace také obsahuje možnost použití více vláken při práci s indexem. Paralelizace probíhá v místě, kde se index dělí na konkrétní soubory - k vláknu se přiřadí soubor a pak toto vlákno vykoná požadovanou operaci nad svým souborem. Důvodem k této implementaci bylo zjednodušení ve formě vyhnutí se synchronizaci při přístupech do jednoho souboru. Implementovali jsme paralelizaci pomocí konceptu *thread pool*.

Při testování efektivity využití více vláken - konkrétně dvou v prostředí s jedním dvou-jádrovým procesorem umožňujícím paralelní práci dvou vláken - jsme však zjistili, že použití tohoto postupu má za následek snížení výkonnosti programu pro operace vkládání a mazání. To je způsobeno pravděpodobně právě formou, jakou jsme paralelizaci implementovali. Každé vlákno totiž ve svém kontextu obsahuje i mapování toho konkrétního souboru, s kterým pracuje. Proto je operace přepnutí kontextu vlákna příliš drahá. Pro operaci hledání bylo dosaženo zrychlení, které ale není dostatečně razantní, aby vyvážilo snížení výkonnosti ostatních operací. Řešení této situace by mohlo přinést buď spuštění aplikace na systému, který disponuje vícero plnými procesory umožňujícími běh dvou vláken bez přepínání kontextu, nebo reimplementace paralelizace do nižší vrstvy práce s indexem.

	1 vlákno	2 vlákna	přírůstek
Vložení	$572ms$	$863ms$	$+51\%$
Hledání	$34ms$	$24ms$	-29%
Mazání	$519ms$	$759ms$	$+46\%$

Tabulka 6.1: Porovnání průměrných časů operací indexu při běhu s jedním či dvěma vlákny. Přírůstek času ukazuje výrazné zpomalení u operací *vlož* a *smaž*. *Pozn:* Hodnoty byly naměřeny při běhu s náhodnými vstupy (stejné vstupy pro všechny operace) v počtu 100 nad indexem obsahujícím 10000 záznamů.

6.3 Jiná existující řešení

V následujících sekcích uvedeme tři běžně užívané postupy pro indexaci daktyloskopické databáze. Tyto metody následně srovnáme s námi implementovaným

dvojitým děleným indexem využívajícím kód směrnic. Jako základní zdroj nám posloužil článek de Boera, Bazena a Gereze uvádějící přehled základních indexačních metod s porovnáním. [BOER01]

6.3.1 Pole směrnic

Nejjednodušším způsobem indexace je použití pole směrnic (v naší aplikaci jej využíváme jako pole lokálních orientací v kapitole 2.3.1). V něm se detekuje *jádro* otisku a podle něj se určí celkový tvar kresby a ten slouží pro základní klasifikaci do tříd otisku. Následně se vytvoří vektor vzdáleností k jednotlivým třídám. Tento vektor pak slouží k porovnávání, respektive k indexaci databáze. [CAPP00]

Výhodou tohoto způsobu je přímočarost vzhledem ke standardní klasifikaci užívanou v daktyloskopii. Nevýhodou je, že při porovnávání neúplného otisku má absence významných bodů z okrajových částí otisků velký vliv na vektor vzdáleností. Druhý problém nastává v případě detekce jádra otisku třídy *oblouk*, protože tento otisk neobsahuje tak výrazné změny směrnic - *singulární body* - jako ostatní třídy (viz obrázek 6.3). Třetí nevýhodou této metody je špatná robustnost vůči nekvalitnímu otisku. Pokud se při extrakci nepoužije sofistikovanější postup, který eliminuje falešné body, bude to mít velký vliv na výsledný vektor.

Obrázek 6.3: Otisk prstu třídy *oblouk*

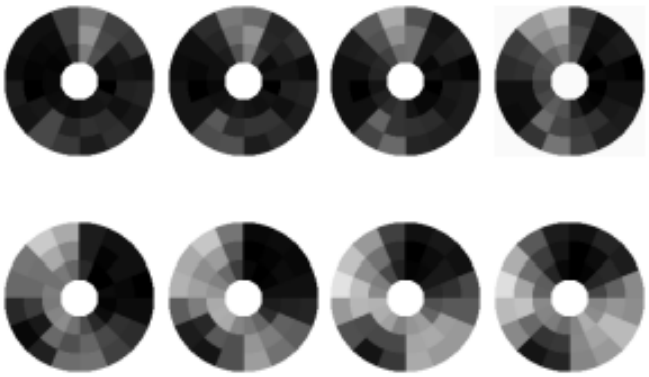
6.3.2 FingerCode

Název FingerCode se používá jako označení vektoru osmi oblastí otisku. Ve skutečnosti se jedná o jednu oblast - konkrétně kruhovou kolem jádra otisku - transformovanou Gaborovým filtrem pokaždé v jiném z osmi směrů. Celá oblast je rozdělena do 48 sekcí. Tímto způsobem se získá z otisku kód o 386 dimenzích s hodnotami reprezentovanými odstíny šedi. Kód je navržen tak, aby zachytil jak lokální

tak i globální charakteristiku otisku. FingerCode se užívá zejména k porovnávání jednotlivých otisků jedna k jedné, ale lze jej využít i k indexaci databáze.

Použití FingerCode je limitováno stejnými problémy s detekcí jádra jako předešlá metoda s polem směrnic (viz 6.3.1). Navíc je výpočetně náročnější než ostatní způsoby, což výrazně ovlivní výkonnost databáze, která by tuto metodu používala. Výhodou však je množství informací, které FingerCode nese - ne pouze informace o konkrétních bodech, ale o celé oblasti otisku.

Obrázek 6.4: FingerCode - ilustrace extrahované oblasti. [BOER01]



Obrázek 6.5: Příklad instance FingerCode. [BOER01]

6.3.3 Trojice markatů

Indexace pomocí trojic markantů (anglicky *minutiae triplets*) je metoda, při níž se získané body otisku shlukují do trojic. Tyto trojice tvoří trojúhelníky a jejich specifika jako jsou úhly nebo délky stran, případně jiné jejich geometrické vlastnosti, slouží jako klíče pro indexaci. [BHAN01]

Výhodou tohoto procesu je malá výpočetní náročnost, protože ze získaných bodů se jednoduše vytvoří trojúhelníky a s nimi se pracuje. Nevýhodou, kterou uvádí de Boer, Bazer a Gerez ve svých testech, je nízká spolehlivost.

6.3.4 Srovnání s kódem směrnic otisku

V této kapitole srovnáme výhody a nevýhody výše uvedených metod indexace s naší.

	impl	comput	pointErr	coreErr
Pole směrnic	ne	ne	ne	ne
FingerCode	ne	ne	ano	ne
Trojice markantů	ano	ano	část	ano
Kód směrnic (náš)	ano	ano	část	ano

Tabulka 6.2: Porovnání výhod jednotlivých návrhů pro indexaci daktyloskopické databáze. *(impl = jednoduchá implementovatelnost; comput = nízká výpočetní náročnost; pointErr = robustnost vůči chybám detekce bodů; coreErr = robustnost vůči chybám detekce jádra)*

Z tabulky 6.2 je patrné, že námi implementovaný způsob indexace pomocí kódu směrnic má obdobné vlastnosti jako použití trojice markantů. Oproti tomu pole směrnic nevyniká žádným zásadním kladem a FingerCode sráží vysoká výpočetní náročnost.

6.4 Měřené vlastnosti děleného dvojitého indexu

Nad děleným dvojitým indexem jsme provedli několik testů a měření, které nám umožní jej objektivně zhodnotit. Měřili jsme výkonnost indexu (časy potřebné pro provedení jednotlivých operací) a přesnost. K testování byla použita databáze FVC2006 [FIER07].

6.4.1 Výkonnost

Pro měření výkonnosti jsme vygenerovali 10000 náhodných kódů směrnic otisku a vložili je do indexu. Pak jsme provedli měření pro každou operaci s dalšími 100 náhodnými vstupy. Průměrná doba potřebná pro vložení záznamu do indexu je $572,23ms$, pro mazání je to $518,93ms$ a pro vyhledávání jsme naměřili průměrně $36,06ms$. Měření probíhalo na počítači s konfigurací: Intel Pentium CPU B970 (2,3 GHz, 64-bit), 3,6 GB RAM.

6.4.2 Přesnost

Měření přesnosti probíhá tak, že se snažíme zjistit, jaká je šance nalezení očekávané shody vůči velikosti části prohledané databáze. Otestovali jsme sadu 536 otisků prstů (8 verzí vždy od jednoho prstu) a testů jsme provedli několik s různým nastavením rozptylu. Ten můžeme nastavovat ve dvou rozměrech - minimální počet shodných typů hodnot (procentuálně) a rozptyl hodnot (číselně). Výsledky uvádí tabulky 6.3 a 6.4. Zobrazují procentuální hodnoty vyjadřující část databáze, která byla prohledána (*Otestováno*), úspěšnost nálezu při aktuálním nastavení (*Úspěch*) a pro srovnání procentuální hodnoty velikosti prohledané databáze naměřené de Boerem (*Otestováno v literatuře*). Kvůli objektivitě testu samotného indexu jsme použili k analýze vstupu a následném porovnání prověřenou aplikaci *NBIS*.

Rozptyly	Otestováno (%)	Úspěch (%)	Otestováno (%) v literatuře
30%, 10	14,18	43,50	-
30%, 15	25,37	64,61	-
30%, 20	31,94	71,00	-
20%, 10	25,52	65,67	-
20%, 15	33,13	72,71	-
20%, 20	38,81	77,40	-
10%, 10	36,12	76,76	-
10%, 15	42,24	82,30	DF: 2, FC: 1, MT: 4
10%, 20	47,16	86,35	DF: 4, FC: 1, MT: 8
0%, 20	66,27	97,23	DF: 15, FC: 13, MT: 75

Tabulka 6.3: Výsledky měření přesnosti indexu (DF = pole směrnic, FC = FingerCode, MT = trojice markantů)

Podle naměřených výsledků v tabulkách 6.3 a 6.4 vidíme, že pro průměrný vstup se přesnost vyhledávání dá srovnávat s úspěšností indexu pomocí trojic markantů.

Rozptyly	Otestováno (%)	Úspěch (%)	Otestováno (%) v literatuře
30%, 10	14, 63	82, 14	DF: 2, FC: 1, MT: 4
30%, 15	19, 55	92, 86	DF: 8, FC: 3, MT: 28
30%, 20	24, 48	97, 32	DF: 15, FC: 13, MT: 75
20%, 10	18, 66	93, 75	DF: 9, FC: 4, MT: 38
20%, 15	23, 88	97, 32	DF: 15, FC: 13, MT: 75
20%, 20	27, 61	99, 11	DF: 25, FC: 45, MT: 95
10%, 10	24, 33	98, 21	DF: 19, FC: 34, MT: 91
10%, 15	29, 10	100, 00	DF: 65, FC: 65, MT: 98
10%, 20	33, 43	100, 00	DF: 65, FC: 65, MT: 98
0%, 20	51, 79	100, 00	DF: 65, FC: 65, MT: 98

Tabulka 6.4: Výsledky měření přesnosti indexu na kvalitních vstupech (DF = pole směrnic, FC = FingerCode, MT = trojice markantů)

Nad kvalitními vstupními obrázky však náš index dosahuje výborných výsledků a to i proti FingerCode a indexaci polem směrnic.

7. Základní schéma práce programu

Základem této diplomové práce je program, který zajišťuje celý proces uchovávání a porovnávání otisků prstů. Tento proces je přirozeně rozdělen na několik částí: transformace vstupního obrazu, detekce rozdílových znaků, relativizace nalezených znaků, klastrování, uložení do databáze a úprava indexu pro vyhledávání podobných otisků.

Přestože se v následujících kapitolách budeme zabývat zmíněnými částmi detailněji, je nutné uvést některé informace, které pomohou vytvořit ucelený obraz o naší aplikaci.

Zdrojové kódy programu byly napsány v jazyce *C#* s *.NET* verze 4.0 a celá aplikace byla vytvořena a spravována v programu *Microsoft Visual Studio 2010*. Program byl vytvářen se snahou o modularitu - je rozčleněn na několik knihoven odpovídajících výše zmíněným podúlohám celého procesu. Tyto knihovny jsou s řídící částí aplikace spojeny prostřednictvím rozhraní. Pro případný pozdější vývoj tedy máme možnost měnit implementaci jednotlivých částí aplikace bez nutnosti rekompilace celého projektu. Například pokud vytvoříme lepší algoritmus pro detekci rozdílových znaků, můžeme ho implementovat do odpovídajícího modulu, ten pak jednoduše samostatně zkompilovat a vyměnit knihovnu.

7.1 Strukturální detaily implementace

Struktura programu, myšleno struktura jmenných prostorů, většinou kopíruje modulární strukturu projektu až na výjimky zejména v pomocných třídách, které bylo nutné oddělit, ale nebylo třeba vytvářet nový projekt (například deklarace vlastních výjimek či rozhraní).

Aplikace obsahuje velké množství konstant, které mohou například určovat, který postup se kde zvolí nebo fungují jako pevný parametr pro určitou funkci. Tyto konstanty jsou rozděleny do modulů vždy podle toho, kde jsou použity. V každém modulu se pak soustřeďují v souboru *Constants.cs*. Konstanty představují základní nastavení aplikace, které bylo otestováno a je doporučeno k bezpečnému používání. Pokud pokročilý uživatel chce tato nastavení změnit, musí pak rekompilovat modul obsahující upravenou konstantu. Záměrně jsme nastavení systému neimplementovali pomocí jednoduše měnitelných parametrů programu (settings),

protože uvedené konstanty reprezentují nastavení, ke kterému se vztahují výsledky této diplomové práce a tedy by mělo být měněno jen za účelem pokročilých testů.

Jako řídící část aplikace byl vytvořen projekt *DaktylDB*, který se kompiluje do spustitelného souboru a je hlavním vstupním bodem pro práci s aplikací. Je rozčleněn na dvě části - uživatelské rozhraní a třídu *ProgramBaseFunctions*, která poskytuje funkce programu. Pro svou práci tento modul využívá rozhraní, která mu umožňují práci s ostatními částmi aplikace. Následující části jsou vždy moduly kompilované jako dynamicky linkované knihovny *DLL*.

Projekt *ImageTransformations* zastřešuje práci se vstupním obrazem otisku prstu. Základním požadavkem je vstup v jednom z formátů *PNG*, *JPEG*, *BMP* nebo *TIFF*. Operace se provádějí skrze rozhraní *IImageTransformator*. Hlavní třídou, která řídí transformace obrazu je *ImageTrasformator* s jedinou veřejnou funkcí *getSkeleton*. Ta v závislosti na nastavení konstant provede binarizaci buď pomocí transformací obrazu (implementuje sama třída funkcí *binarizeImage*) nebo pomocí konvexního práhování (implementuje třída *ConvexThresholdBinarizer*). Z výsledku binarizace pak tato třída pomocí funkce *skeletonizeImage* využívající hit-and-miss filtru vytvoří kostru. Ta je vrácena volajícímu. Pomocnými třídami v tomto projektu jsou *TangentDirectionComputer* počítající lokální orientace linií před binarizací konvexním práhováním a *RegularizationFilter*, který implementuje proces regularizačního filtrování.

Následuje projekt nazvaný *FeatureDetection*, který provádí detekci významných bodů nad výstupem výše zmíněného *ImageTransformator*-u. Práci tohoto modulu definuje rozhraní *IFeatureDetector* a implementuje jej třída *FeatureDetector*. Tato třída deleguje požadavek na detekci bodů kostře otisku dál třídě *PointsDetector*, která podle konstant rozhodne, jestli se má provést detekce metodou záplavového hledání (třída *SimpleFloodfillPointsDetector*) nebo pomocí stavby grafu směrnic (třída *DirectionDetector*). Stavba grafu směrnic se provede voláním funkce *buildGraphSets* na instanci zmíněné třídy a funkcí *findPoints* získáme požadovaný seznam bodů, který je výsledkem celé operace.

Modul *FeatureDetectionNBIS* je nadstavbou, která umožňuje použití funkcí knihovny *NBIS* v našem programu. Tento modul není definován žádným rozhraním, protože slouží jen jako pomocný nástroj a neměl by být upravován. Modul se používá pro porovnání 1:1 pomocí *NBIS* a k detekci bodů při testování spolehlivosti indexu kvůli minimalizaci chyb.

Po detekci významových bodů přichází na řadu příprava dat k práci s databází, tedy klastrování. To provádí modul *PointsClusterization* postavený na rozhraní *IClusterizer*. Tento modul obsahuje jen třídu *Clusterizer*, která z dvojic bodů vytváří klastry.

Práci s databází provádí modul *Database*, který nemá definované rozhraní, ale jeho vstupním bodem je statická třída *DatabaseService*, která zde slouží jako poskytovatel služeb databáze. Tuto implementaci jsme zvolili kvůli její jednoduchosti. Není totiž cílem této práce vytvořit univerzální databázový nástroj, ale ukázat zejména práci navrhovaných algoritmů. Třída *DatabaseService* deleguje požadavky třídě *DatabaseHandler*, která za pomoci reprezentace databázového spojení *DatabaseStorage* implementuje jednotlivé operace. Zde je použito návrhového vzoru návštěvník - základní datové objekty *Person* a *Fingerprint* založené na rozhraní *IStorable* implementují funkce, které mají jako parametr databázové spojení a na něm vykonají potřebnou operaci. Důvodem k použití tohoto konceptu bylo zjednodušení návrhu třídy *DatabaseHandler* - jedna funkce dokáže obsloužit dvě různé entity.

Posledním funkčním modulem je projekt *DatabaseIndex*, který je využíván pouze skrz modul *Database* a to za pomoci rozhraní *IIndexer*. Tento modul spravuje vyhledávací index nad otisky prstů uloženými v databázi. Obsahuje hlavně třídu *DividedFilesDoubleIndex*, která implementuje index popsaný v kapitole 6.

Nakonec ještě celá aplikace obsahuje jeden pomocný projekt nazvaný *Utils*, který shromažďuje ty prvky aplikace, které nepatří přímo do žádného z předešlých projektů, například obsahuje výše uvedená rozhraní (loosely-coupled architektura implementace rozhraní), výjimky a některé další třídy zastřešující nějakou jednodušší funkcionalitu. Patří sem *LockBitmap* pro unmanaged práci s bitmapou, *SimpleThreadPool* jako paralelizační nástroj, *ArrayAggregator* implementující částečný průnik výsledků vyhledávání nad indexem, *ClusterDirectionMap* určující indexy směrnic detekovaných bodů. Zároveň jsou zde i základní datové položky *FPPoint* (nalezený bod otisku), *DirectionVector* (směrový vektor bodu otisku), *FPCluster* (klastr dvou bodů) a *FPPointsCode* (kód směrnic otisku).

7.2 Užité externí knihovny

Aplikace *DaktylDB* používá několik externích knihoven. Jednou z nich je framework *Aforge.NET*[1] použitý pro zpracovávání obrazu. Jeho výhodou proti ostatním je implementace přímo v jazyce C# a hlavně primární určení výzkumníkům v oblasti zpracování obrazu (mimo jiné). Proti tomu například OpenCV je určeno zejména pro práci s grafikou a jeho nástroje se tedy pro náš případ obtížněji používají. Další knihovnou je framework *Json.NET*[2], který je v aplikaci použit pro zpracování vstupního řetězce typu *JSON*, a aplikaci *NBIS*[3] připojenou k tomuto projektu skrze poskytovaný adaptér. Konkrétní způsob použití těchto knihoven je popsán v textu práce.

[1] Volně dostupný open source framework zaměřující se na zpracování obrazu, umělou inteligenci, neuronové sítě, genetické algoritmy, strojové učení a robotiku. Dostupný na: `http://www.aforgenet.com/framework/`

[2] Volně dostupný open source framework společnosti Newtonsoft pro práci s řetězci typu JSON. Dostupný na: `http://www.newtonsoft.com/json`

[3] Volně dostupná aplikace vyvinutá pro agentury FBI a DHS vládní agenturou „The National Institute of Standards and Technology (NIST)“ v USA. Dostupná na: `http://www.nist.gov/itl/iad/ig/nbis.cfm`

8. Uživatelský manuál

Aplikace *DaktylDB* disponuje textovým uživatelským rozhraním, které poskytuje přístup k funkcím programu. Rozhraní se aktivuje spuštěním hlavního programu (jediný spustitelný soubor) a očekává od uživatele číselný vstup v závislosti na požadované funkci.

Vložení, úprava, smazání a vyhledání záznamu jsou základní funkce, které by měl používat běžný uživatel programu. Při volbě vyhledávání se zobrazí rozšířená volba umožňující specifikovat jedno z vyhledávacích kritérií - identifikátor osoby, jméno nebo otisk prstu. Úprava databázového záznamu mění pouze jméno osoby. Pokud chceme změnit otisk přiřazený k osobě, je nutné tuto osobu smazat a pak ji opětovně vytvořit se správnými otisky.

Další funkce - automatické vložení, porovnání 1:1 pomocí kódu směrnic otisku nebo NBIS a hromadné testy indexu - jsou funkce určené pro testování aplikace.

Automatické vložení je operace, která do databáze vloží všechny otisky ve zdrojovém adresáři (více v 8.1) a pro každý otisk vytvoří jednu osobu. Jméno této osoby je název souboru s otiskem. Tato funkce slouží k rychlému vytvoření testovací databáze.

Porovnání 1:1 může být provedeno dvěma způsoby, které se liší použitým algoritmem. Buď můžeme vstupy porovnat pomocí kódů směrnic otisků nebo pomocí funkcí poskytované aplikací NBIS. Proces načte první dva specifikované otisky a ty porovná. Na výstup napíše vypočtenou míru shody.

Hromadné testy indexu jsou čtyř typů. Můžeme měřit rychlost vkládání [1], mazání [2] a nebo hledání [3]. Také můžeme měřit přesnost hledání. Při té aplikace načte všechny otisky ze zdrojového adresáře vyjma prvních variant (varianta otisku jejíž název končí řetězcem „_1“) a snaží se je vyhledat v indexu. Získanou sadu podobných otisku porovná s databází. Detailní výstup je pak v souboru „index_search_match_bench.out“.

[1] Aplikace provede operaci vložení záznamu do indexu stokrát s náhodnými vstupy, průměrný čas vypíše na výstup, detailní výstup do souboru „index_insert_bench.out“.

[2] Aplikace provede operaci smazání záznamu s náhodnými vstupy stokrát, průměrný čas vypíše na výstup, detailní výstup do souboru „index_remove_bench.out“.

[3] Aplikace provede stokrát operaci vyhledávání s náhodnými vstupy, průměrný čas vypíše na výstup, detailní výstup do souboru „index_search_bench.out“.

8.1 Parametry a zdrojová data operací

Aplikace nepodporuje vkládání parametrů funkcím přes uživatelský terminál, ale načítá je ze specifického souboru. Distribuce programu zahrnuje adresář „container“, který slouží právě jako vstupní bod pro uživatelská data a parametry. Tento adresář nazýváme *zdrojový adresář* a musí obsahovat soubor „_params.json“. Nazvěme jej *soubor parametrů*.

V souboru parametrů musí být data ve tvaru řetězce *JSON*[4] obsahující položky *ID*, *name* a *fingerprints*. První zmíněná položka je specifikace identifikátoru osoby, která má být upravena, smazána či vyhledána. Druhá položka je jméno osoby specifikované za účelem uložení. Poslední položka *fingerprints* je seznam otisků, respektive řetězců obsahujících názvy souborů s otisky. Tento seznam se buď používá ke specifikaci všech otisků osoby, která má být vložena, nebo k určení otisku, podle kterého má být vyhledáno (zvolí se první soubor v seznamu), a nebo k určení dvou otisků porovnávaných 1:1 (první dva soubory). Nepoužívané položky souboru parametrů mohou mít hodnotu prázdného řetězce.

Všechny soubory s otisky, na které se odkazujeme v parametrech, nebo které program sám vyhledá při hromadných operacích, musejí být ve zdrojovém adresáři (ne v podadresářích).

8.2 Typické scénáře použití

8.2.1 Vložení

Uživatel vloží do zdrojového adresáře soubory formátu *JPEG*, *BMP*, *PNG* nebo *TIFF*, které obsahují otisky prstu vkládané osoby. Do souboru parametrů vyplní správné údaje o osobě a zapíše názvy souborů s otisky. Pak spustí program a zvolí funkci vložení.

8.2.2 Úprava

Nejprve je třeba zjistit identifikátor upravované osoby vyhledáváním. Do souboru parametrů uživatel vyplní správné údaje o osobě - identifikátor a nové jméno. Pak spustí program a zvolí úpravu.

[4]Syntaxe řetězců *JSON* dostupná na: `http://www.w3schools.com/json/json_syntax.asp`

8.2.3 Smazání

Nejprve je třeba zjistit identifikátor upravované osoby vyhledáváním. Pak do souboru parametrů uživatel vyplní zjištěný identifikátor, spustí program a zvolí funkci mazání.

8.2.4 Hledání

Uživatel vloží do zdrojového adresáře soubor formátu *JPEG*, *BMP*, *PNG* nebo *TIFF* obsahující otisk prstu, podle kterého chce hledat (pokud nevyhledává podle otisku, tento krok přeskočí). Do souboru parametrů vyplní údaje pro vyhledávání - identifikátor nebo jméno či název souboru s otiskem - a spustí program. Zvolí pak operaci dle potřeby.

Závěr

Tato práce si za hlavní cíl kladla vytvoření aplikace, která představuje průřez celou problematikou daktyloskopických aplikací. Poukazuje na omezení jak v oblasti zpracování obrazu a rozpoznávání specifických vzorů, tak i v kontextu uchovávání a správy dat. Ve všech těchto ohledech jsme se snažili přiblížit čtenáři problematiku a seznámit ho s existujícím řešením. Vytvořili jsme také několik postupů, které se pokouší nalézt nové cesty k řešení jednotlivých problémů. Za úspěch můžeme považovat vytvoření postupů a algoritmů, které svou výkonností a efektivitou mohou konkurovat již existujícím řešením.

Neméně důležitým záměrem je také rozšíření povědomí výzkumníků o problematice, která se v současné době uzavírá do čistě komerční sféry. Nových poznatků, které by byly zcela veřejné, je pomálu a jejich implementace je až na několik málo výjimek nedostupná i akademickému světu.

Celou práci jsme rozdělili do logických částí, které představují hlavní bloky procesu od úpravy vstupu do „čitelné podoby“ až po uložení dat do databáze a její správu. V prvních kapitolách jsme tak implementovali dva způsoby zpracování vstupu - obrázku jednoho článku prstu lidské ruky - z nichž jeden postup jsme také sami navrhli. Výsledek obou těchto operací je použitelný pro další krok, kterým je analýza.

Analýza předzpracovaného otisku měla za cíl vyhledat konkrétní body tvořící porovnatelnou sadu dat a tyto body rozšířit o další potřebné informace. To se nám povedlo pomocí našeho vlastního postupu za použití jednoduché datové struktury - spojového seznamu. Také návrh datových struktur použitých jako těžiště následujících částí byl naší autorskou prací.

Třetí, a poslední, hlavní částí byla databáze. Zde jsme se soustředili zejména na návrh a implementaci databázového indexu, jakožto hlavního nástroje pro zajištění efektivity vyhledávání podobných otisků. Tento index jsme otestovali jak na rychlost, tak i na jeho spolehlivost, která je právě v daktyloskopii zásadní. Naměřené výsledky jsou srovnatelné s obdobným měřením v literatuře [BOER01]. Práce našeho indexu je dostatečně rychlá a navíc spolehlivá. Při správném nastavení je dokonce spolehlivější než indexy, které zdroj uvádí.

Protože je daktyloskopie velmi rozšířeným oborem užívaným v mnoha odvětvích, kde je potřeba důsledně zajistit identifikaci osob, můžeme tvrdit, že je výzkum v

této oblasti stále cenný. Zejména proto, že rychlé a přesné určení identity může pomoci v kriminalistické činnosti a zajištění všeobecné bezpečnosti. Nemusí se však jednat jen o oblast kriminality, můžeme najít využití i v jiných odvětvích. Jednoduché a spolehlivé algoritmy na rozeznání otisků prstů nám mohou umožnit přestat používat klíče od auta nebo bytu. Stejně tak bychom mohli využít čtečku otisků prstů osobního počítače ke vzdálenému bezpečnému přihlášení do podnikové sítě nebo banky a mnoho dalšího.

Doporučení pro další vývoj

Před pokračováním vývoje aplikace *DaktylDB*, respektive algoritmů a postupů, které tato práce navrhuje, by bylo vhodné podrobit všechny procesy detailnějším testům, které odhalí případné slabiny. Stejně tak se nabízí možnost implementace algoritmů pro zařízení s vysokou úrovní paralelizace. Jelikož bylo původním úmyslem vytvořit aplikaci, která se soustředí na efektivitu v rámci velkého objemu dat, je využití paralelizace dalším logickým krokem. Pro případné použití programu v komerční sféře je nutností vytvořit grafické uživatelské rozhraní.

Literatura

[WIKIa] Daktyloskopie. Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2015-07-18]. Dostupné z: https://cs.wikipedia.org/wiki/Daktyloskopie

[SUCH96] SUCHÁNEK, Jaroslav. Kriminalistika - kriminalistickotechnické metody a prostředky. Vyd. 1. Praha: Policejní akademie České republiky, 1996. s. 25, 36. ISBN 80-85981-21-1.

[STRA05] STRAUS, Jiří a Viktor PORADA. Kriminalistická daktyloskopie. Vyd. 1. Praha: Vydavatelství PA ČR, 2005. s. 53-54, 56, 57, 59. ISBN 80-7251-192-0.

[GONZ92] GONZALEZ, Rafael C a Richard E WOODS. Digital image processing. Reading: Addison-Wesley Publishing Company, c1992. World student series. s. 532-541. ISBN 0201600781.

[FISH04] FISHER, Robert, Simon PERKINS, Ashley WALKER a Erik WOLFART. Thinning. HIPR2 - Image processing learning resources [online]. 2004 [cit. 2015-07-19]. Dostupné z: http://homepages.inf.ed.ac.uk/rbf/HIPR2/thin.htm

[W3C] W3C Working Draft: Techniques For Accessibility Evaluation And Repair Tools [online]. [cit. 2015-07-20]. Dostupné z: http://www.w3.org/TR/AERT#color-contrast

[WIKIb] Moving average. Wikipedia: the free encyclopedia [online]. Wikimedia Foundation, 2001- [cit. 2015-07-21]. Dostupné z: https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average

[BANS11] BANSAL, Roli, Priti SEHGAL a Punam BEDI. Minutiae extraction from fingerprint images-a review. arXiv preprint arXiv:1201.1422 (2011).

[KIM03] KIM, Doo-Hyun a Rae-Hong PARK. Fingerprint Binarization using Convex Threshold. V: Computer Graphics and Imaging. 2003. s. 224-227.

[MAIO97] MAIO, Dario a Davide MALTONI. Direct gray-scale minutiae detection in fingerprints. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 1997, 19.1: 27-40.

[MALT12] MALTONI, Davide. Fingerprint recognition: Basics and Recent Advance [online]. s. 105 [cit. 2015-12-01]. Dostupné z: http://icb12.iiitd.ac.in/Fingerprint-ICB2012.pdf

[CAPP10] CAPPELLI, Raffaele, Matteo FERRARA a Davide MALTONI. Minutia Cylinder-Code: A New Representation and Matching Technique for Fingerprint Recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2010, 32(12): 2128-2141. DOI: 10.1109/TPAMI.2010.52. ISSN 0162-8828. Dostupné také z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5432197

[FERR12] FERRARA, Matteo, Davide MALTONI a Raffaele CAPPELLI. Noninvertible Minutia Cylinder-Code Representation. IEEE Transactions on Information Forensics and Security [online]. 2012, 7(6): 1727-1737 [cit. 2015-11-30]. DOI: 10.1109/TIFS.2012.2215326. ISSN 1556-6013. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6307852

[CAPP00] CAPPELLI, Raffaele, Dario MAIO a Davide MALTONI. Indexing Fingerprint Databases for Efficient 1:N Matching. Sixth International Conference on Control, Automation, Robotics and Vision (ICARCV2000). Singapore, 2000.

[BOER01] de BOER, Johan, Asker M. BAZEN a Sabih H. GEREZ. Indexing fingerprint databases based on multiple features. Technology Foundation STW. 2001

[BHAN01] BHANU, Bir a Xuejun TAN. A triplet based approach for indexing of fingerprint database for identification. In: Audio-and Video-Based Biometric Person Authentication. Springer Berlin Heidelberg, 2001. p. 205-210.

[FIER07] FIERREZ, J., J. ORTEGA-GARCIA, D. TORRE-TOLEDANO a J. GONZALEZ-RODRIGUEZ. BioSec baseline corpus: A multimodal biometric database, Pattern Recognition, Vol. 40, n. 4, pp. 1389-1392, April 2007.