VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

MOBILNÍ APLIKACE PRO DEMONSTRACI ŘEŠENÍ RUBIKOVY KOSTKY

RUBIK'S CUBE DEMONSTRATION ON MOBILE PLATFORMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DAVID GAJDOŠ**
AUTHOR

VEDOUCÍ PRÁCE **Doc. Ing. FRANTIŠEK ZBOŘIL, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Gajdoš David**
Program: Informační technologie
Název: **Mobilní aplikace pro demonstraci řešení Rubikovy kostky**
Rubix Cube Demonstration on Mobile Platforms
Kategorie: Umělá inteligence
Zadání:

1. Seznamte se s algoritmy řešení problému Rubikovy kostky a s různými variantami tohoto hlavolamu. Dále se seznamte s metodami zpracování obrazu, které by umožnily ze snímků Rubikovy kostky zjistit její aktuální konfiguraci.
2. Navrhněte aplikaci, která na základě snímků Rubikovy dokáže určit typ hlavolamu a následně předvede postup jejího řešení.
3. Implementujte navržené řešení pro systém Android tak, že po nasnímání stavu kostky aplikace navrhne nastavený počet tahů, které vedou k řešení.
4. Vyzkoušejte výslednou aplikaci na dostatečném počtu osob (alespoň deset) a shrňte jejich názory, postřehy a kritiky. Na základně tohoto navrhněte další vylepšení aplikace, pokud bylo požadováno a diskutujte užitečnost systému pro širší veřejnost.

Literatura:

- Russel, S., Norvig, P.: Artificial Intelligence, A Modern Approach, Pearson, 2009

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Zbořil František, doc. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Tato práce se zaměřuje na tři hlavolamy odvozené z Rubikovy kostky. Cílem je vytvořit aplikaci pro operační systém Android, která uživateli umožní nasnímat stěny hlavolamu. Po správném nasnímání barevné konfigurace celého hlavolamu nalezne aplikace sekvenci pohybů vedoucí k jeho vyřešení. Tato sekvence je v textové i vizuální podobě následně uživateli prezentována a aplikace uživatele vede jednotlivými kroky až k vyřešenému stavu.

Abstract

This thesis describes three Rubik's cube like puzzles. The goal is to create Android application which allows users to scan sides of a puzzle. After successfully scanning all sides of a puzzle, the application finds sequence of moves needed to perform to solve the puzzle. This found sequence is then textually and visually presented to the user. The application then guides the user through individual moves until the puzzle is solved.

Klíčová slova

Android, Slimtower, Rubikovo domino, Pyraminx, TensorFlow, OpenCV, Konvoluční neuronové sítě, MobileNetV2

Keywords

Android, Slimtower, Rubik's domino, Pyraminx, TensorFlow, OpenCV, Convolutional neural networks, MobileNetV2

Citace

GAJDOŠ, David. *Mobilní aplikace pro demonstraci řešení Rubikovy kostky.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. František Zbořil, Ph.D.

Mobilní aplikace pro demonstraci řešení Rubikovy kostky

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Doc. Ing. Františka Zbořila, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
David Gajdoš
10. května 2021

Poděkování

Chtěl bych poděkovat Doc. Ing. Františku Zbořilovi, Ph.D. za jeho čas, rady a přípomínky k mé práci. Velké díky také patří rodině a přátelům, kteří mi byli oporou nejen počas psaní práce, ale i v průběhu celého studia.

Obsah

Kapitola 1

Úvod

S Rubikovou kostkou se ve svém životě setkal snad každý. Touto kostkou ovšem u většiny lidí povědomí o hlavolamech končí. Málokdo totiž ví, že existuje celá řada hlavolamů odvozených z Rubikovy kostky. Tyto hlavolamy nabývají nejrůznějších velikostí a tvarů. Pro některé lidi nejsou hlavolamy nic víc než pouhou hračkou. Pro některé tyto "hračky" ovšem znamenají mnohem víc. Existuje spousta nadšenců, kteří mezi sebou pořádají soutěže v rychlosti skládání hlavolamů, poskládání hlavolamu co nejmenším počtem kroků nebo například soutěže o vynalezení nejzajímavěji vypadajícího funkčního hlavolamu.

Tato práce se zaměřuje na tři hlavolamy odvozené z Rubikovy kostky – Pyraminx, Slimtower a Rubikovo domino. Cílem je vytvořit mobilní aplikaci pro operační systém Android, která umožní uživatelům naskenovat hlavolamy prostřednictvím mobilní kamery. Po správném naskenování hlavolamu předvede aplikace pohyby vedoucí k jeho vyřešení.

Čtenář se v práci v kapitole 2 dozví bližší informace o Rubikově kostce a výše zmíněných hlavolamech. Dále je v kapitole 3 popsáno počítačové vidění, knihovna počítačového vidění OpenCV a jak se v práci za pomocí počítačového vidění získávají barvy z obrazových dat. Kapitola 4 pojednává o strojovém učení, konvolučních neuronových sítích, které jsou v práci využity pro klasifikaci skenované stěny, a architektuře MobileNetV2, na níž je založena v práci použitá neuronová síť. Kapitola 5 popisuje operační systém Android, jeho architekturu a knihovny použité v práci. Tyto čtyři kapitoly poskytují teoretické informace o způsobech využitých při implementaci. Implementace je popsána v kapitole 6. Předposlední kapitola práce, kapitola 7, vysvětluje, jak byla aplikace testována v průběhu implementace a následně po dokončení implementace na koncových uživatelích. Závěrečná kapitola 8 obsahuje shrnutí práce a zhodnocení dosažených výsledků.

Kapitola 2

Rubikova kostka a její varianty

Rubikova kostka patří mezi jeden z nejznámějších a nejoblíbenějších mechanických hlavolamů. V druhé polovině 80. let 20. století sklidila velký úspěch mezi lidmi z celého světa a v roce 2012 byla prohlášena za jeden z nejprodávanějších produktů všech dob [22]. Touto magickou kostkou, jak byla prvně nazvána samotným autorem, byla inspirována spousta dalších mechanických hlavolamů fungujících na stejné či podobné bázi. Všechny hlavolamy sdílí stejnou myšlenku — složit hlavolam do stavu, v němž každá stěna obsahuje pouze kostky stejné barvy.

V této kapitole jsou nejprve popsány obecné informace, které sdílí všechny varianty hlavolamů. Dále je zde popsána historie Rubikovy kostky, detailnější popis jednotlivých odvozených variant a postupy a algoritmy potřebné k pochopení a vyřešení těchto hlavolamů.

2.1 Rubikova kostka

Tato sekce seznamuje čtenáře s historií klasické Rubikovy kostky. Ve výsledné aplikaci kostka není implementována, každopádně byla vzorem pro vznik ostatních variant hlavolamu, a proto autor považuje za důležité, aby byla alespoň stručně zmíněna.

Podrobněji popsala Rubikovu kostku v 2. kapitole své bakalářské práce Bc. Liptáková [21]. Z její práce bylo použito několik informací v popisu historie.

Historie

Kostku o rozměrech $3 \times 3 \times 3$ stvořil v roce 1974 maďarský vynálezce, sochař a profesor architektury `Ernő Rubik`. Profesorova motivace pro zkonstruování kostky spočívala v jeho nadšení a obdivu pro geometrii a řešení hlavolamů.

Občas je mylně uváděno, že kostka byla primárně zkonstruována a využívána jako školní pomůcka. Sám Rubik kostku svým studentům předvedl, každopádně ji nevytvořil pro ně. Vytvořil ji proto, že byl designer, který si rád hrál s geometrickými tvary [35].

Při tvoření prototypu sáhl Rubik po nejdostupnějším a nejjednodušeji opracovatelném materiálu – dřevě. Kostka se skládala z vytesaných dřevěných kostiček, některými s dírou uprostřed, spojených gumičkami (viz obrázek 2.1). Tento jednoduchý mechanismus rozpohyboval objekt a umožnil 90° rotace jednotlivých stěn. Rubik si zanedlouho uvědomil, že by se dřevěný prototyp dal velice jednoduše předělat na hlavolam pouhým přilepením barevných nálepek na kostičky [21]. Byl fascinován, že pouze po několika rotacích se kostka nacházela ve stavu absolutního chaosu. Popsal situaci jako "hledět na kus psaní v kódu,

který sám vytvořil, ale nemohl přečíst" [35]. A tak byla na světě úplně první Rubikova kostka, v té době nazývána Magická kostka.

O 3 roky později byl schválen maďarskými úřady patent, o něž Ernő zažádal, a kostka začala být vyráběna v Maďarsku. Pár let později v roce 1979 po domluvě s Rubikem přivezl obchodník Tibor Laczi kostku na veletrh hraček v Norimberku se záměrem hlavolam zviditelnit. To se mu také povedlo. O kostku projevila zájem americká firma Ideal Toys, která nabídla Rubikovi kontrakt o distribuci jeho vynálezu do celého světa [21]. Společnost trvala na změně jména hlavolamu z důvodu ochrany vynálezce. Tímto vznikl roku 1980 v dnešní době již notoricky známý název Rubikova kostka. Ve Spojených státech amerických byl Rubikův patent schválen roku 1983 [30].

První složení kostky zabralo jejímu vynálezci déle než měsíc a dokončení hlavolamu popsal jako velice emocionální zážitek [36].

Obrázek 2.1: První prototyp Rubikovy kostky.[1]

Obrázek 2.2: Nynější verze Rubikovy kostky.

2.2 Obecný popis vlastností všech variant hlavolamů

Rubikova kostka (2.1), Rubikovo domino (2.3), Slimtower (2.4) a Pyraminx (2.5) (dále jen hlavolamy), jimiž se bakalářská práce zabývá, mají několik vlastností společných. Tato sekce slouží jako jejich obecný popis, který bude následně podrobněji popsán u konkrétních variant.

2.2.1 Konstrukce

Každý hlavolam se skládá z určitého počtu menších kostiček. Tyto jsou drženy pohromadě otočným mechanismem nacházejícím se uprostřed tělesa, který umožňuje rotace jednotlivých stěn ve směru i v protisměru hodinových ručiček. Rotace budou blíže popsány v podsekci 2.2.3.

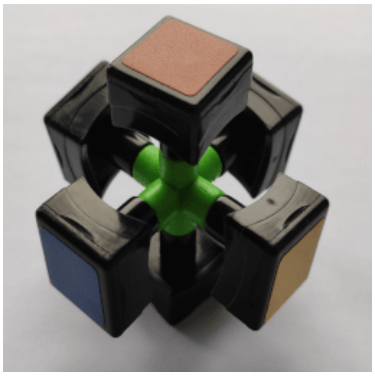
Celý systém lze mechanicky rozložit na jednotlivé kostičky a následně opět složit. Při opětovném skládání je nutné dát si pozor, aby všechny kostičky byly fyzicky na správném místě a správně otočené. Pokud by tomu tak nebylo, hlavolam by nešel vyřešit. Z toho důvodu se doporučuje skládat kostku zpět podle barev.

[1] Obrázek 2.1 byl převzat z https://www.nytimes.com/2020/09/16/books/erno-rubik-rubiks-cube-inventor-cubed.html [navštíveno 8/1/2021]

Malé kostičky se dělí na tři rozdílné typy (popis byl inspirován popisem uvedeným v 2.2 podkapitole Bc. Liptákové [21]):

- **Středové** – Mají jednu barvu. U některých hlavolamů jako například u Rubikovy kostky (2.1) a obecně u všech symetrických N×N×N kostek (kde N je přirozené liché číslo) jsou středy fixní. V případě hlavolamu Pyraminx (2.5) je tomu jinak a středy lze mezi sebou prohazovat. Střed se nachází vždy pouze na jedné stěně tělesa. Středové kostičky Rubikovy kostky jsou zobrazeny na obrázku 2.4.
- **Hranové** – Mají dvě barvy. Barvy jsou od sebe vždy rozdílné a odpovídají barvám dvou sousedních stěn. Hranová kostička se nachází ve středu hrany tělesa. Hranové kostičky Rubikovy kostky jsou zobrazeny na obrázku 2.5.
- **Rohové** – Mají tři barvy. Podobně jako u hranové kostičky platí, že barvy jsou od sebe vždy rozdílné a odpovídají třem sousedním stěnám. Rohové kostičky se nachází ve vrcholech tělesa. Rohové kostičky Rubikovy kostky jsou zobrazeny na obrázku 2.3.

Obrázek 2.3: Rohové kostičky Rubikovy kostky.



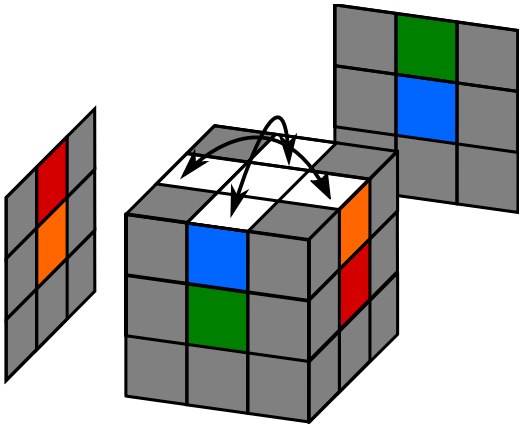
Obrázek 2.4: Středový otočný mechanismus Rubikovy kostky se středovými kostičkami.

Obrázek 2.5: Hranové kostičky Rubikovy kostky.

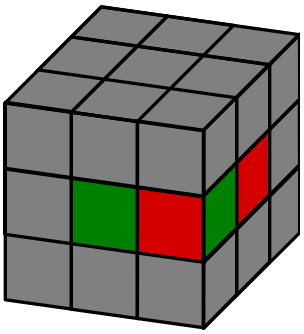
S kostičkami souvisí dva podstatné pojmy, které ve své práci popisuje Bc. Liptáková [21]:

1. **Orientace** – Znamená natočení kostičky. Kostička se může nacházet na správném místě, ale její barvy nemusí odpovídat barvám stěn (viz obrázek 2.7). V tom případě je potřebné kostičku správně natočit.
2. **Permutace** – Vyjadřuje prohození jednotlivých kostiček mezi sebou. Příkladem permutace může být algoritmus H-perm metody PLL zobrazený na obrázku 2.6.

V případě, že se kostička nachází na správném místě a je správně orientovaná, je označována za vyřešenou.



Obrázek 2.6: Vizualizace permutace (prohození) hranových kostiček horní vrstvy Rubikovy kostky.



Obrázek 2.7: Hranová kostička nacházející se na správném místě, ale špatně orientovaná.

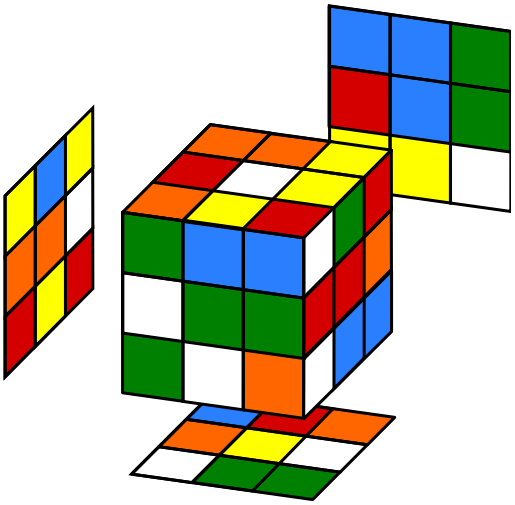
2.2.2 Barva

Jednou z důležitých vlastností hlavolamů je barva, neboť rozložení barev vyjadřuje, v jaké permutaci se hlavolam momentálně nachází a zdali je, či není vyřešitelný (nevyřešitelný hlavolam může být pouze tehdy, když byl špatně složen po mechanickém rozložení).

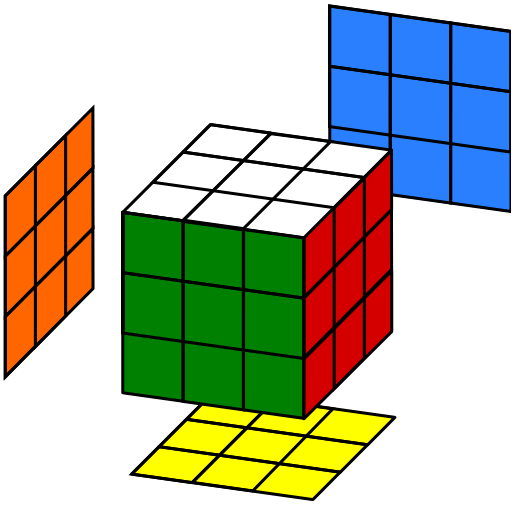
Jak bylo zmíněno v podsekci 2.2.1, každá kostička má přiřazený určitý počet barev. Tyto barvy jsou přiřazeny buď formou nálepky, která je nalepena na jednu ze stěn kostičky, nebo přímým obarvením stěn kostičky.

Všechny tyto nálepky dohromady tvoří stěny hlavolamů. Ve vyřešeném stavu hlavolamu odpovídá každé stěně právě jedna barva (viz obrázek 2.9). Standardní barvy u šestistěnných hlavolamů (Rubikova kostka, Rubikovo domino a Slimtower) jsou – bílá, červená, zelená, modrá, oranžová a žlutá na černé podkladové kostičce. Běžné bývají i bílé podkladové kostičky, k dostání je ale celá řada barevných konfigurací. Dokonce existují i kostky s natisknutými fotografiemi zvířat, přírody, lidí, sudoku kostky apod. Jedná se pouze o designová vylepšení, která na řešení kostky nemají žádný vliv. Obdobně u čtyřstěnných hlavolamů (v práci konkrétně Pyraminx) jsou standardní barvy zelená, žlutá, modrá a červená s černým podkladem. Není-li tedy řečeno jinak, v práci jsou uvažovány standardní barvy s černými podkladovými kostičkami u všech hlavolamů.

Podle rozložení barev se stěna může nacházet v jednom ze dvou stavů – stěna je buď vyřešená, nebo nevyřešená. Obsahuje-li stěna pouze kostičky stejné barvy, je označována za vyřešenou. Podobně je tomu tak i pro celý hlavolam — pokud jsou všechny stěny vyřešeny, je vyřešený i hlavolam.



Obrázek 2.8: Zamíchané barvy Rubikovy kostky.



Obrázek 2.9: Vyřešená Rubikova kostka – každé stěně odpovídá právě jedna barva.

2.2.3 Rotace a jejich značení

Jak již bylo zmíněno v úvodu kapitoly, podstatou je složit hlavolamy do stavu, v němž každá stěna obsahuje pouze kostky stejné barvy. Tohoto stavu nelze docílit jinak, než provedením určité sekvence kroků vedoucích k řešení. Těchto sekvencí bývá zpravidla více. Aby mezi sebou řešitelé hlavolamů mohli sekvence sdílet, bylo nutné stanovit obecný popis rotací. Standardem se mezi řešiteli Rubikovy kostky a jiných hlavolamů na podobné bázi považuje písmenná notace rotování stěn vycházející z poznámek amerického profesora Davida Singmastera (občas nazývána "Singmaster notation") [32].

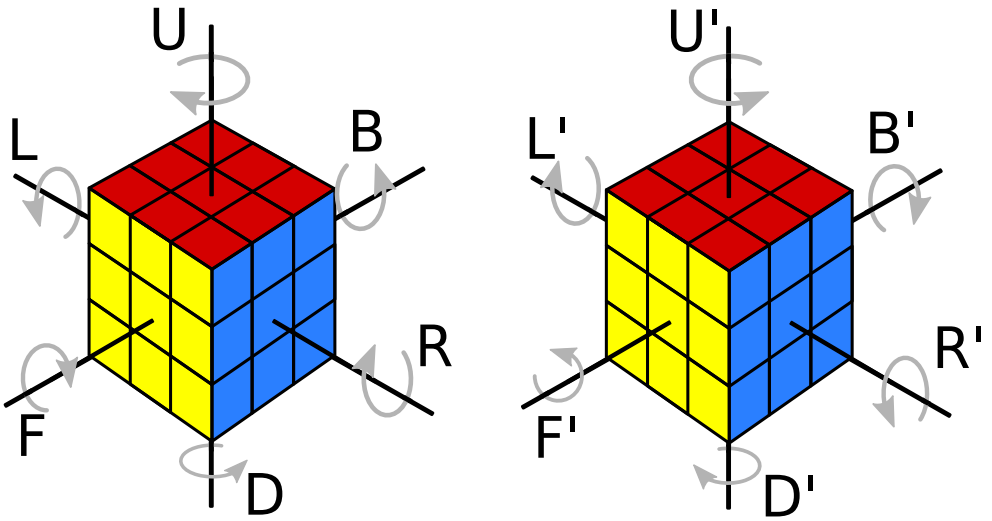
Profesor Singmaster se snažil vymyslet notaci jednotlivých pohybů nezávislou na barvách kostky či stěně, která je přilehlá k řešiteli. Napadlo ho, že by stěny mohl pojmenovat z pohledu držitele kostky. Vzniklo tak písmenné označení každé stěny a její rotace. Přilehlá stěna k řešiteli nese název **F**, levá stěna **L**, pravá **R** atd. (viz označení rotací níže). Postupem času byly některé jeho notace poupraveny nebo rozšířeny, protože nebyly dostatečné pro popis složitějších hlavolamů (jako například kostky větší než $3 \times 3 \times 3$ nebo složitější hlavolamy jako Megaminx). Jedním příkladem malé úpravy je změna z původního označení R^2 na **R2**. Toto označení vyjadřuje 2x otočení pravé stěny ve směru hodinových ručiček.

Vzhledem k tomu, že původní notace popsána profesorem Singmasterem byla zamýšlena pro Rubikovu kostku $3 \times 3 \times 3$, budou jakékoliv příklady uvedené v této podsekci demonstrovány právě na této kostce.

Pro popis rotace se využívá velké počáteční písmeno anglického slova označujícího stěnu. Všechny níže popsané rotace jsou ve směru hodinových ručiček. V případě vyjádření otočení v protisměru hodinových ručiček se za písmenem píše znak apostrof (např. **L'**, **F2'**).
Základních 6 označení popsaných profesorem Singmasterem:

- **F** (front) – Přední stěna
- **B** (back) – Zadní stěna
- **L** (left) – Levá stěna
- **R** (right) – Pravá stěna

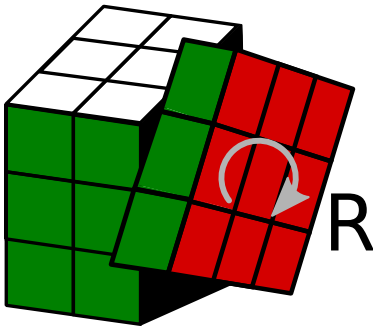
- **U** (up) – Horní stěna
- **D** (down) – Spodní stěna



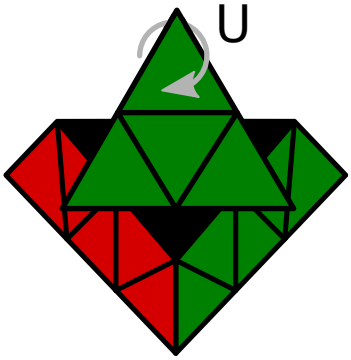
Obrázek 2.10: Levá kostka: Označení rotací jednotlivých stěn ve směru hodinových ručiček Pravá kostka: Označení rotací v protisměru hodinových ručiček.

Mimo těchto 6 základních označení existují ještě i jiná. Jsou jimi například označení pro pohyb středových vrstev ve vertikálním a horizontálním směru, otočení celé kostky, otočení stěny a k ní korespondující střední vrstvy nebo řezné rotace. Detailně jsou tato označení popsána v [38]. V práci je použito pouze základních 6 označení + označení M (rotace střední vrstvy) při řešení Slimtower (2.4) + označení rotací rohů Pyraminx (2.5).

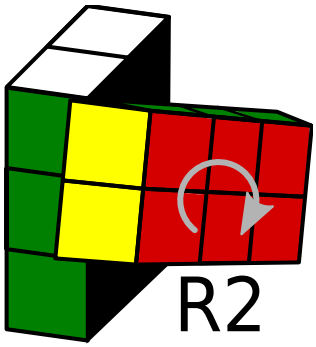
Úhel rotace je závislý na typu hlavolamu a jeho počtu stěn. Existují 90° rotace (hlavolamy ve tvaru krychle), 120° rotace (čtyřstěny – Pyraminx), ale i 180° rotace (hlavolamy tvaru kvádru). Tyto rotace ovšem není nutné mezi sebou jakkoliv rozlišovat podle stupňů otočení, neboť každé písmeno označuje právě jedno otočení. V případě, že se má provést více otočení, je toto vyjádřeno číslicí za písmenem (např. **R2**, **F2'**). Dále se rozlišuje mezi čtvrtinovými a poloviční otočeními. Čtvrtinové otočení odpovídá 90° rotaci, poloviční 180°. Toto rozlišení existuje z důvodu, že některé stěny kvádrových hlavolamů, například Slimtower nebo Rubikovo domino, neumožňují 90° rotace. Z toho důvodu některé zdroje popisující algoritmy uvádí 180° pohyb pravé stěny jako písmeno R (poloviční otočení), jiné zdroje tento stejný pohyb popisují jako R2 (2x čtvrtinové otočení). V práci jsou uvažována pouze čtvrtinová otočení. Otočení pravé stěny Slimtower je potom popsáno jako R2 a pohyb R je zde nevalidní.



Obrázek 2.11: Rotace 90° pravé stěny Rubikovy kostky.



Obrázek 2.12: Rotace 120° horního sub-tetraedru hlavolamu Pyraminx.

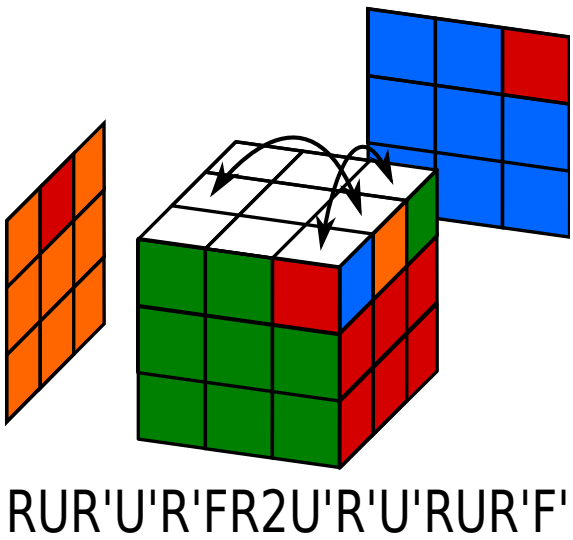


Obrázek 2.13: Rotace 180° pravé stěny hlavolamu Slimtower.

Je nutné také podotknout, že díky cykličnosti rotací mohou rozdílná označení končit v totožném stavu. To znamená, že pohyby R2 a R2' končí ve stejném stavu, neboť u Rubikovy kostky nezáleží, jestli je stěnou otočeno o 180° ve směru nebo v protisměru hodinových ručiček, výsledek je stejný. Podobně končí ve stejném stavu například i pohyby R3 a R', obecně ale platí, že se vždy zapisuje jednodušší varianta s nižším číslem.

Rotace přesahující 360° v notaci neexistují. To znamená, že pro Rubikovu kostku neexistuje pohyb R5, protože je to stejný pohyb jako R. Pohyby končící ve stejném stavu jako na začátku jsou také zbytečné – R4 končí v počátečním stavu, toto není nutné zapisovat.

Za pomocí tohoto písmenného označení jsou popisovány algoritmy k vyřešení hlavolamů. Těchto algoritmů existuje mnoho a liší se pro každý hlavolam. Některé mezi sebou hlavolamy ale sdílí. Algoritmy je možné rozlišovat podle různých aspektů. Existují efektivnější algoritmy, méně efektivní, algoritmy pro začátečníky (jednodušší na zapamatování), pro skládání poslepu atd. Příklad zápisu algoritmu pro Rubikovu kostku lze vidět na obrázku 2.14.



Obrázek 2.14: Příklad písmenného zápisu algoritmu pro prohození naznačených hran a rohů.

2.3 Rubikovo domino

Sekce pojednává o méně známém hlavolamu vynálezce Rubikovy kostky – Rubikovu dominu. Nejprve je zde popsán kousek historie, následně konstrukce kostky a algoritmy potřebné k vyřešení.

Historie

Po obrovském úspěchu svého prvního vynálezu přišel `Ernő Rubik` v roce 1979 s prototypem dalšího mechanického hlavolamu. Tento hlavolam, prvně nazývaný Magické domino, byl modifikací původní Rubikovy kostky. Rubik odebráním jedné vrstvy svého prvního hlavolamu zkonstruoval kvádr o rozměrech $3 \times 3 \times 2$. Původně tento kvádr neobsahoval žádné barvy na jednotlivých kostičkách, ale pouze dvě rozdílně barevné vrstvy (černá a bílá), u nichž na každé kostičce byly puntíky podobné těm, které jsou na dílku domina[2] (proto také hlavolam nese název Rubikovo domino). K vyřešení hlavolamu bylo potřeba seskládat puntíky vzestupně od 1 do 9 počínaje levým horním rohem a konče pravým spodním rohem (viz obrázek 2.15). Hodnota počtu puntíků na kostičce v horní vrstvě musela odpovídat hodnotě počtu puntíků kostičky na stejné pozici ve spodní vrstvě, jako je tomu u klasického domina. Postupem času byl tento design změněn a Rubikovo domino bylo, podobně jako Rubikova kostka, polepeno barevnými nálepkami (viz obrázek 2.16). Toto nezměnilo způsob řešení kostky, pouze její vzhled.

Obrázek 2.15: Původní verze Rubikova domina.[3]

Obrázek 2.16: Nynější verze Rubikova domina s barevnými nálepkami.

Popis hlavolamu

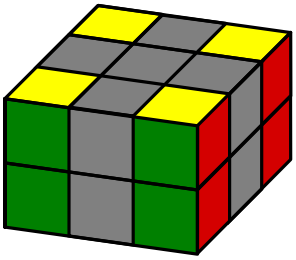
Základní informace o konstrukci, barvách a rotacích hlavolamu jsou v sekci 2.2.

Rubikovo domino se skládá z 18 fyzických kostiček a otočného mechanismu uprostřed. Z těchto 18 kostiček jsou 2 středové, 8 hranových a 8 rohových. Hlavolam má dohromady 6 stěn, 4 z nich jsou o rozměrech 3×2, zbylé 2 mají rozměr 3×3. Stěny o rozměru 3×2 umožňují pouze 180° rotace (poloviční otočení), stěny o rozměru 3×3 umožňují i 90°

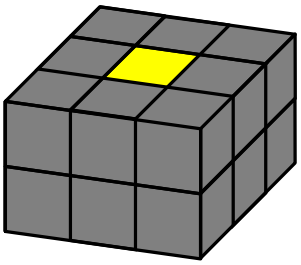
[2] Domino je stolní hra obsahující dřevěné dílky s různým počtem puntíků. Více o dominu – https://cs.wikipedia.org/wiki/Domino

[3] Obrázek 2.15 byl převzat z http://twistypuzzles.com/cgi-bin/puzzle.cgi?pkey=615 [navštíveno 20/12/2020]

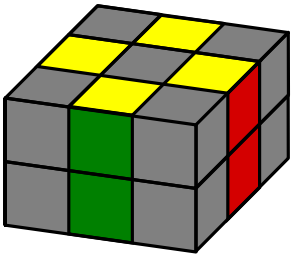
rotace (čtvrtinová otočení). Díky těmto rotacím se hlavolam může nacházet až v 406 425 600 rozdílných stavech.



Obrázek 2.17: Rohové kostičky Rubikova domina.

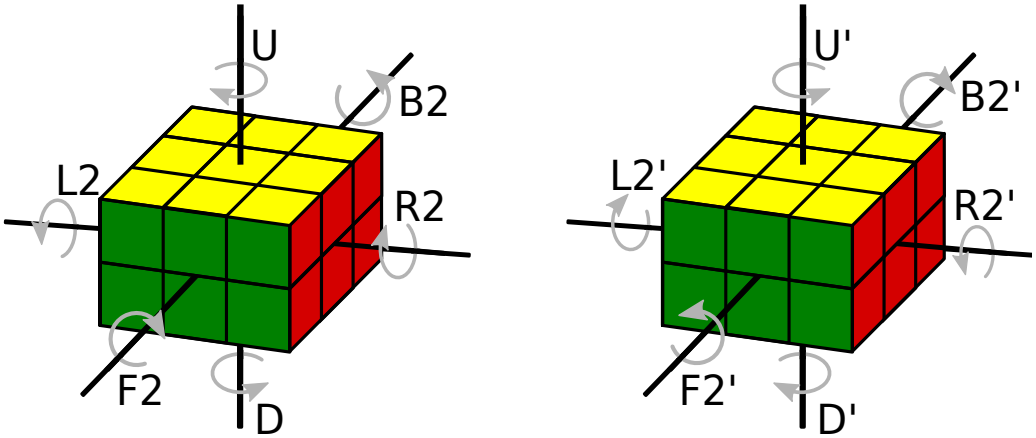


Obrázek 2.18: Středová kostička Rubikova domina.



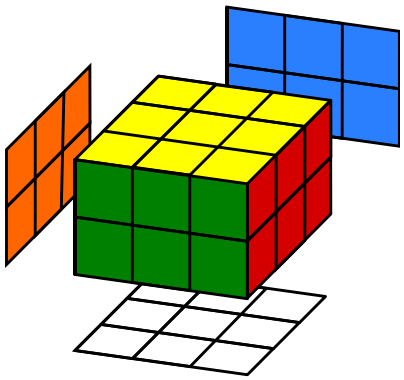
Obrázek 2.19: Hranové kostičky Rubikova domina.

Standardní držení kostky je stěnou 3×3 s bílým středem dole. Tato stěna nese označení **D**. Její protější stěna se žlutým středem nese označení **U**. Zbylé stěny jsou označeny **L, R, F, B** v závislosti na tom, jak je momentálně kostka natočená k řešiteli. Jak již bylo řečeno v sekci 2.2.3, v práci jsou uvažována čtvrtinová otočení, a proto jediné povolené rotace stěn 3×2 jsou **L2, R2, F2, B2**. Alternativně mohou být k vídění i tytéž rotace v opačném směru. Toto ovšem popisuje stejnou věc, proto když se objeví tento popis rotace v protisměru hodinových ručiček, jedná se pouze o symbolické vyjádření, že se strana vrací zpět na své místo. Rotace **L, R, F, B** ani jejich alternativy v případě Rubikova domina nedávají smysl. Rotace a jejich písmenná značení lze vidět na obrázku 2.20.



Obrázek 2.20: Levý kvádr: Označení rotací jednotlivých stěn ve směru hodinových ručiček Pravý kvádr: Označení rotací v protisměru hodinových ručiček.

Hlavolam má standardní barevné schéma. Stěny 3×3 mají žlutou a bílou barvu. Barva těchto stěn je určena barvou středové kostičky. Ostatní stěny mohou mít červenou, modrou, zelenou nebo oranžovou barvu. Při skládání je nutné dát si pozor, aby hranové kostičky byly vůči sobě ve správném rozpoložení tak, aby odpovídaly barvám rohových kostiček. Barvy hlavolamu jsou na obrázku 2.21.



Obrázek 2.21: Barevné schéma vyřešeného Rubikova domina.

Postup řešení

Níže je slovně popsán algoritmus vedoucí k vyřešení hlavolamu. Jednotlivé kroky algoritmu budou odkazovat na sekvence pohybů, které slouží k permutaci (viz definice 2) hlavolamu. Všechny potřebné sekvence jsou uvedeny zde[4]:

R2 U R2 U' R2'	(Algo. 2.3.1)
(R2 U R2 U' R2') (D U') (R2' U' R2 U R2)	(Algo. 2.3.2)
R2 U2 R2 U2 R2 U2	(Algo. 2.3.3)
R2 U R2 U R2 U2 R2 U2 R2 U R2 U' R2	(Algo. 2.3.4)

Algoritmus k vyřešení může být rozdělen do 4 fází:

1. **Bílý kříž** – Základem je udělat bílý kříž na spodní vrstvě. Tato část je jednoduchá a intuitivní, není zde zapotřebí žádných algoritmů. Jediné, co je nutné udělat, je dostat hranovou kostičku nad hranu, kam patří, a poté provést poloviční otočení příslušné stěny. Je nutné dávat pozor, aby vkládané hrany byly ve správném rozpoložení vůči sobě. To znamená, že pokud je například bílo-červená hrana na spodní vrstvě směrem k řešiteli, hrana napravo od ní musí být bílo-zelená, nalevo bílo-modrá a na místě protější hrany bílo-oranžová kostička.

2. **Rohy spodní vrstvy** – Vkládání rohů spodní vrstvy je také jednoduché. Stačí najít na horní vrstvě libovolnou rohovou kostičku obsahující bílou barvu. Zbylé dvě barvy této kostičky určují, na jaké místo patří. Barvy se musí shodovat s barvami sousedních hranových kostiček spodní vrstvy. Po nalezení místa, kam rohová kostička patří, je nutné ji dostat do pravého předního rohu horní vrstvy, pod ni dát roh, za nějž má být vyměněna, a provést algoritmus Algo. 2.3.1. Tímto způsobem budou všechny rohy spodní vrstvy správně vloženy a tím bude i první spodní vyřešena.

3. **Rohy horní vrstvy** – Nyní, když už je spodní vrstva vyřešena, je možné začít s horní vrstvou. Klíčové jsou v této fázi tzv. světla (toto označení nesou kostičky proto, že připomínají světla automobilu). Jedná se o dvě rohové kostičky se stejnou barvou na jedné stěně – to znamená, že tyto rohy jsou již na správném místě. Může nastat situace, v níž se světla na horní vrstvě nacházet nebudou, v tom případě stačí provést

[4] Algoritmy převzaty z https://youtu.be/pbv652cE1AU [navštíveno 12/12/2020]

algoritmus Algo. 2.3.2 na libovolné stěně a světla se objeví. Následně je potřeba světla dostat do pozice, aby se nacházela na levé stěně. Potom již stačí provést algoritmus Algo. 2.3.2, srovnat barvy rohů horní vrstvy s barvami spodní vrstvy a rohy budou vyřešené.

4. **Hrany horní vrstvy** – Jako poslední krok je potřeba permutovat hrany horní vrstvy. K tomu slouží dva algoritmy – jeden zaměňuje dvě protilehlé hrany (Algo. 2.3.3) a druhý zaměňuje pravou hranu s přední hranou (Algo. 2.3.4). Nezáleží, v jakém pořadí jsou hrany vyměněny. Stačí pouze najít místo, na nějž hranová kostička patří, podle toho provést jeden z algoritmů a kostička se bude nacházet na správném místě.

2.4 Slimtower

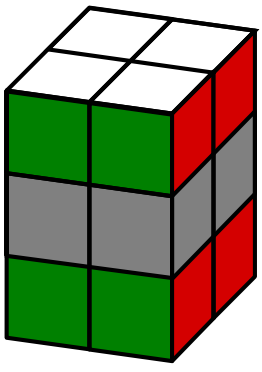
Sekce popisuje jeden z mnoha "tower" hlavolamů o rozměrech $2 \times 2 \times 3$. Je v ní k nalezení obecný popis hlavolamu a postup, jak jej vyřešit.

Popis hlavolamu

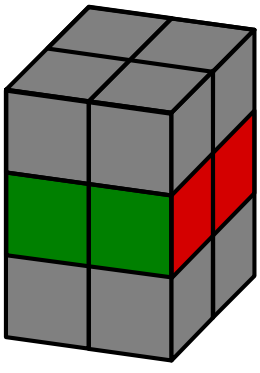
Hlavolam byl vytvořen někdy na přelomu tisíciletí. Není známý přesný datum, kdy byl tento kvádr poprvé zkonstruován, ale údajně podle [37] hlavolam existoval před lednem roku 2002. Autorem hlavolamu je japonský vynálezce Katsuhiko Okamoto, který je mimo jiné i autorem populárního hlavolamu Void Cube nebo čtyřstěnu Mater Pyraminx, který je rozříšenou verzí hlavolamu Pyraminx, o němž pojednává následující sekce 2.5.

Základní informace o konstrukci, barvách a rotacích hlavolamu jsou v sekci 2.2.

Slimtower se skládá dohromady z 12 kostiček, 8 z nich jsou rohové kostičky (viz obrázek 2.22), zbylé 4 jsou hranové kostičky (viz obrázek 2.23). Středové kostičky se na tomto hlavolamu nenacházejí. Obsahuje 6 stěn, 4 z nich jsou o rozměru 3×2, horní a spodní stěna mají rozměr 2×2. Obdelníkové stěny umožňují pouze 180° rotace (poloviční otočení), stěny o rozměrech 2×2 umožňují i 90° rotace (čtvrtinová otočení). Hlavolam se může nacházet v 241 920 pozicích. Z jakékoliv pozice je možné se dostat do konečného stavu za pomocí maximálně 15 tahů [20].



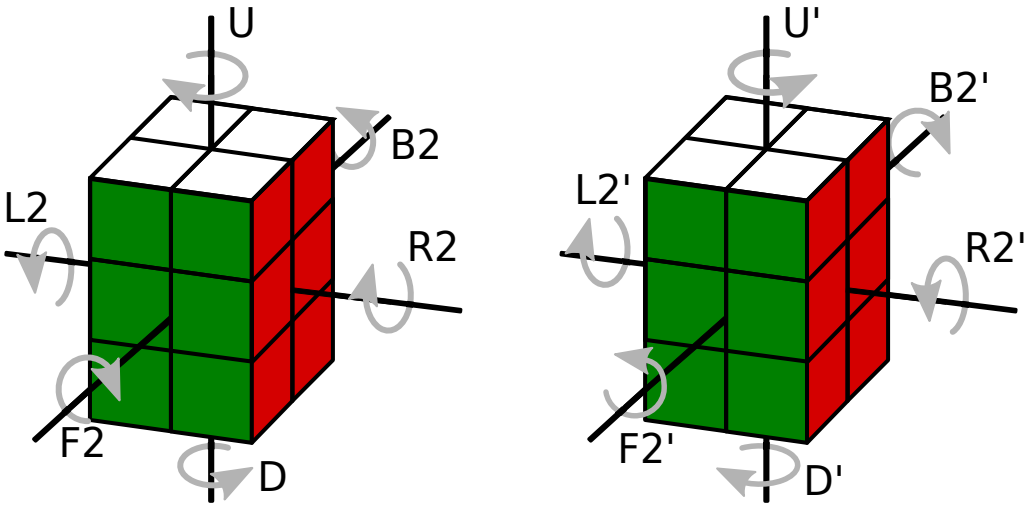
Obrázek 2.22: Rohové kostičky hlavolamu Slimtower.



Obrázek 2.23: Hranové kostičky hlavolamu Slimtower.

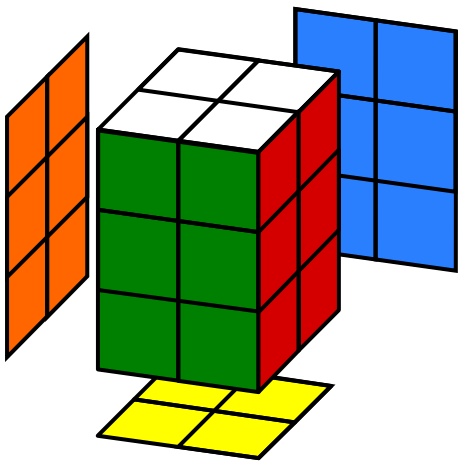
Podobně jako u Rubikova domina jsou zde uvažována čtvrtinová otočení, a proto na stěnách o rozměru 3×2 jsou povoleny pouze pohyby **L2, R2, F2, B2**. Na horní a spodní stěně jsou povoleny všechny typy pohybů pro čtvrtinová otočení.

Standardní držení kostky je stěnami 2×2 nahoře a dole. Barvy jednotlivých stěn nejsou nijak určeny fixními středovými kostičkami, a proto při držení nezáleží, jaké barvy jsou na jednotlivých stěnách. Důležité je pouze, aby čtvercové stěny byly nahoře a dole. Tyto stěny poté nesou písmenná označení **U** pro horní stěnu a **D** pro stěnu spodní. Navíc je zde oproti ostatním hlavolamům ještě i označení **M** pro střední vrstvu a její rotaci ve směru hodinových ručiček. Rotace jsou vyobrazeny na obrázku 2.24.



Obrázek 2.24: Levý kvádr: Označení rotací jednotlivých stěn ve směru hodinových ručiček Pravý kvádr: Označení rotací v protisměru hodinových ručiček.

Barevné schéma je u tohoto hlavolamu standardní. Jedná se tedy o barvy – bílá, červená, zelená, modrá, oranžová a žlutá na černé podkladové kostičce.



Obrázek 2.25: Barevné schéma vyřešeného Slimtower.

Postup řešení

Sekvence pohybů potřebné k vyřešení Slimtower, na něž odkazuje popis níže[5]:

R2 U R2 U' R2 F2 U' F2 U F2 (Algo. 2.4.1)
R2 D R2 D' R2 B2 D' B2 D B2 (Algo. 2.4.2)
R2 U2 R2 U2 R2 U2 (Algo. 2.4.3)
R2 M2 R2 (Algo. 2.4.4)

Řešení hlavolamu může být rozděleno do 3 fází:

1. **Horní a spodní stěna** – Cílem této fáze je vyřešit barvy horní a spodní 2×2 stěny. Barvy celých vrstev nemusí být vyřešeny, důležité jsou pouze barvy stěn. To znamená, že je nutné dostat kostku do stavu, v němž na horní stěně jsou pouze nálepky žluté barvy, a na spodní stěně nálepky barvy bílé. Tohoto se dá docílit jednoduchými kroky a není k tomu zapotřebí žádný algoritmus. Stěny se vzájemně k sobě mohou nacházet v těchto stavech:

   1.1. **Dva pruhy** – Případ, v němž na obou stěnách jsou dvě žluté a dvě bílé kostičky vedle sebe tvořící dva pruhy rozdílných barev. K vyřešení stačí zarovnat bílý pruh na horní stěně nad žlutý pruh na spodní stěně a prohodit tyto rotací příslušné stěny.

   1.2. **Tři a jedna** – Případ, ve kterém jsou tři kostičky stejné barvy na jedné stěně a zbylá se nachází na stěně druhé. Stav lze vyřešit tak, že na jedné stěně je přesunuta osamocená kostička na libovolné místo a osamocená kostička na stěně druhé je umístěna diagonálně k ní (v rámci stěny, ne v rámci celého tělesa). Například je-li horní osamocená kostička vzadu vpravo, spodní osamocená může být vzadu vlevo nebo vepředu vpravo. Nakonec stačí provést rotaci příslušné stěny, aby vznikly dva pruhy, a pokračovat podle předešlého bodu 1.1..

   1.3. **Šachovnice** – V tomto případě barvy na stěně připomínají vzor šachovnice (dvě stejné barvy jsou vůči sobě diagonálně umístěné). Může nastat případ, že budou šachovnice buď na obou stěnách, nebo pouze na jedné ze stěn. V případě, že je šachovnice na obou stěnách, stačí je zarovnat tak, aby byly vzory na obou stěnách pod sebou (bílá pod bílou a žlutá pod žlutou). Poté stačí provést libovolný z pohybů **L2, R2, F2, B2** a hlavolam bude ve stavu 1.1.. Je-li šachovnice na stěně jedné a na stěně druhé jsou dva pruhy rozdílné barvy, přistupuje se k případu jako by na obou stěnách byly pruhy a provede se rotace, jíž se dostane jeden z pruhů na opačnou stěnu. Poté se hlavolam bude nacházet ve stavu 1.2..

2. **Horní a spodní vrstva** – Cílem další fáze je permutovat rohové kostičky horní a spodní vrstvy k jejich složení, tudíž sladění všech barev dané vrstvy. Pro tuto fázi existují dva algoritmy. Rozlišujeme zde tři stavy:

   2.1. **Vrstva má pruh/světla** – Pruhem nebo světly se rozumí dvě stejné barvy sousedních rohových kostiček vrstvy. Rozlišují se dva algoritmy – jeden pro spodní vrstvu a jeden pro horní vrstvu. Pro složení horní (žluté) vrstvy je potřeba dát pruh na levou stěnu a použít algoritmus Algo. 2.4.1. Analogicky pro složení spodní (bílé) vrstvy je potřeba dát pruh na levou stěnu a použít algoritmus Algo. 2.4.2.

[5] Algoritmy převzaty z https://youtu.be/6dYOrUgFCsc [navštíveno 16/12/2020]

2.2. **Vrstva nemá pruh/světla** – V tomto případě pouze stačí provést pro vrstvu bez pruhu odpovídající algoritmus popsaný v předešlém bodě a na hlavolamu se objeví pruh. Poté se postupuje podle předešlého bodu 2.1..

3. **Střední vrstva** – Cílem finální fáze je vyřešit střední vrstvu, čímž bude vyřešen i celý hlavolam. Nejprve je potřeba zarovnat barvy horní a spodní vrstvy, aby se shodovaly na všech stěnách. Následně se zarovná i střední vrstva tak, aby hlavolam byl co nejvíce vyřešen. Nakonec jsou mezi sebou prohozeny jednotlivé středové kostičky za pomocí dvou algoritmů. První z algoritmů Algo. 2.4.3 prohazuje dvě pravé hrany mezi sebou. Druhý Algo. 2.4.4 prohazuje levou a pravou dvojici hran.

2.5 Pyraminx

Sekce pojednává o čtyřstěnném hlavolamu Pyraminx. Je zde popsaná historie, popis hlavolamu a postup vedoucí k jeho vyřešení.

Historie

Hlavolam Pyraminx vytvořil německý vynálezce `Uwe Mèffert`. Stvoření hlavolamu ovšem bylo více méně náhodou než jeho záměrem. Mèffert pracoval na začátku 70. let 20. století na výzkumu ohledně energetických toků v lidském těle, a protože bylo v té době velice populárním tématem energie egyptských pyramid, součástí výzkumu bylo i hraní si s různými geometrickými tvary, mimo jiné i se čtyřstěnem. Uwe Mèffert vytesal ze dřeva několik mnohostěnů, které nasadil na středovou kouli umožňující rotování, a spojil je dohromady gumičkami. Bližší detail prototypu je na obrázku 9 v [25]. Vynálezce chtěl sledovat, jaký efekt má na člověka interakce s tímto předmětem. Po neúspěšném experimentu odložil vytvořený předmět do zásuvky a nadále mu nevěnoval pozornost.

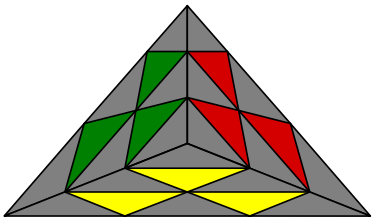
Nebýt Rubikova slavného vynálezu Rubikovy kostky 2.1, zřejmě by na Mèffertův hlavolam sedal prach až do dnes. Právě vydání Rubikovy kostky a její rozšíření mezi veřejnost vyvolalo impulz pro Mèfferta se znovu zajímat o jím vytvořený čtyřstěn. Nedlouho poté německý vynálezce zažádal o vydání patentu pro jeho hlavolam, který byl v prosinci roku 1981 schválen [25]. Dříve téhož roku představil svůj vynález japonské společnosti Tomy Toys, která přistoupila na dohodu, že jeho hlavolam budou vyrábět a prodávat. Uwe Mèffert byl skeptický, že o jeho vynález nebudou mít lidé zájem, každopádně k jeho úžasu se již v období Vánoc roku 1981 prodalo 10 milionů kusů [11].

Mèffert nezůstal pouze u vynálezu Pyraminx. Jeho portfolio se pyšní i jinými hlavolamy jako například Megaminx nebo Skewb.

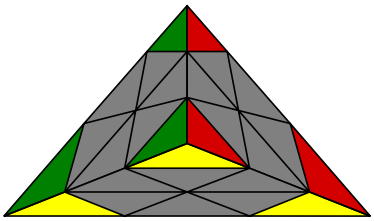
Popis hlavolamu

Základní informace o konstrukci, barvách a rotacích hlavolamu jsou v sekci 2.2.

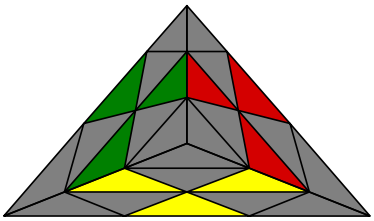
Pyraminx je hlavolam ve tvaru tetraedru (pravidelného čtyřstěnu). Skládá se dohromady z 22 kostiček, 4 z nich jsou rohové (viz obrázek 2.27), 12 kostiček je středových (viz obrázek 2.26) a 6 hranových (viz obrázek 2.28). Oproti předešlým hlavolamům u Pyraminx neplatí, že by středové kostičky byly fixní. Je možné s nimi libovolně hýbat, ovšem platí, že existuje pouze jedna stěna, na níž se mohou nacházet všechny tři středové kostičky stejné barvy najednou – to určuje i barvu celé stěny. Hlavolam se skládá ze tří vrstev – spodní, největší vrstva, střední vrstva a horní vrstva (rohová kostička). Spojením horní a střední vrstvy



Obrázek 2.26: Středové kostičky hlavolamu Pyraminx

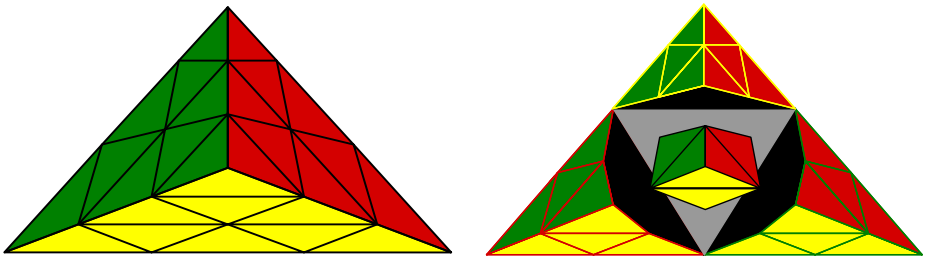


Obrázek 2.27: Rohové kostičky hlavolamu Pyraminx



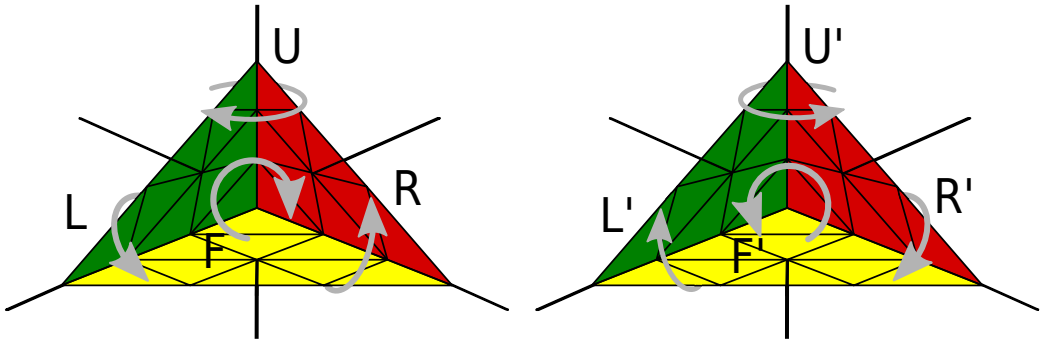
Obrázek 2.28: Hranové kostičky hlavolamu Pyraminx

vznikne pomyslný čtyřstěn, který v práci bude nazýván **sub-tetraedr**. Tento sub-tetraedr je znázorněn na obrázku 2.29.

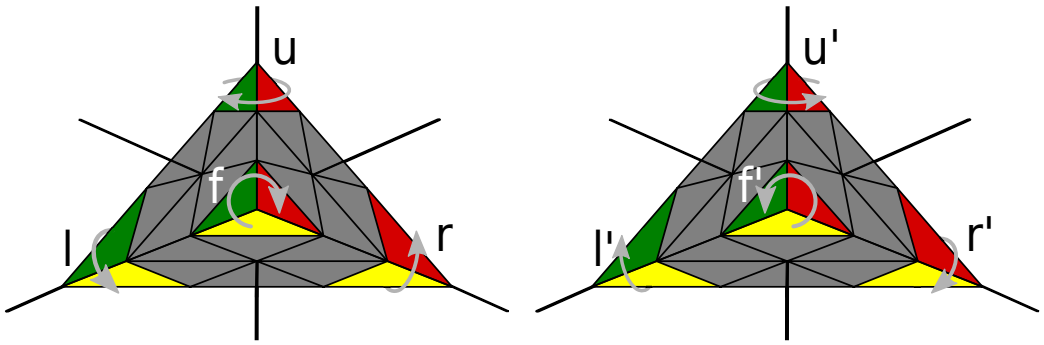


Obrázek 2.29: Na pravém Pyraminx jsou znázorněny 3 oddělené **sub-tetraedry**, díky kterým mohou být na hlavolamu prováděny rotace

Vzhledem k tvaru tělesa jsou zde i trochu rozdílné rotace v porovnání s předešlými hlavolamy. Jak bylo již nastíněno v podsekci 2.2.3, hlavolam umožňuje 120° rotace. Tyto rotace nejsou rotace stěn, ale sub-tetraedrů. Existují dva typy jednoduchých rotací. První typ je podmnožina základních rotací popsaných v podsekci 2.2.3. Jsou jimi rotace **L, R, F, U** a jejich inverzní obdoby. Těmito písmeny jsou označeny rotace jednotlivých sub-tetraedrů z pohledu řešitele. Druhým typem rotací jsou velice jednoduchá otočení rohových kostiček. Tato jsou označena malým písmenem odpovídajícího sub-tetraedru. To znamená, že otočení rohové kostičky sub-tetraedru **U** je značeno jako **u**, tatáž rotace v protisměru hodinových ručiček se značí **u'**. Kvůli 120° rotacím zde není potřeba uvádět u písmenného označení rotace žádnou číslici, neboť jsou dostačující obyčejná označení a jejich inverzní obdoby. Rotaci **U2** lze vyjádřit jako **U'** a pohyby **U3** nebo **U3'** jsou redundantní, protože končí v počátečním stavu. Rotace jsou vidět na obrázcích 2.30 a 2.31.



Obrázek 2.30: Levý Pyraminx: Označení rotací jednotlivých sub-tetraedrů ve směru hodinových ručiček. Pravý Pyraminx: Označení rotací sub-tetraedrů v protisměru hodinových ručiček



Obrázek 2.31: Levý Pyraminx: Označení rotací jednotlivých rohů ve směru hodinových ručiček. Pravý Pyraminx: Označení rotací rohů v protisměru hodinových ručiček

Hlavolam se může nacházet až v 933 120 pozicích nepočítaje triviální rotace rohových kostiček. Maximální počet rotací potřebných k vyřešení Pyraminx je 11 [19]. V tabulce 2.1 je vyobrazený počet pozic a minimální počet kroků k vyřešení hlavolamu z dané pozice[6].

Pyraminx se skládá ze 4 stěn, tudíž obsahuje i odpovídající počet barev. Standardní barvy jsou červená, žlutá, modrá a zelená na černé podkladové kostičce.

Standardní držení hlavolamu je libovolnou stěnou k řešiteli. Ve výsledné aplikaci bude ovšem uvažováno odlišné držení z důvodu vizualizace. Toto držení bude podrobněji popsáno později v kapitole 6. Při popisu algoritmů níže je zamýšleno standardní držení, tedy jednou stěnou k řešiteli.

n	0	1	2	3	4	5	6	7	8	9	10	11
p	1	8	48	288	1 728	9 896	51 808	220 111	480 467	166 276	2 457	32

Tabulka 2.1: Počet pozic (p) a k nim odpovídající minimální počet pohybů (n) nutný k vyřešení Pyraminx. V počtech pozic nejsou zahrnuty triviální rotace rohových kostiček

[6]Tabulka 2.1 převzata z https://www.jaapsch.net/puzzles/pyraminx.htm [navštíveno 8.1.2021]

Postup řešení

Výčet algoritmů potřebných k vyřešení Pyraminx, na něž se odkazuje popis řešení uvedený níže[7]:

R U' R' U	(Algo. 2.5.1)
L' U L U'	(Algo. 2.5.2)
L R' L' R U' R U R'	(Algo. 2.5.3)
R U R' U R U R'	(Algo. 2.5.4)
R U' R' U' R U' R'	(Algo. 2.5.5)
L U R U' R' L'	(Algo. 2.5.6)
R' U' L' U L R	(Algo. 2.5.7)

K vyřešení hlavolamu je využita začátečnická metoda zvaná `layer by layer` (vrstva po vrstvě). Její postup je následující:

1. **Střed libovolné stěny** – Prvním krokem k vyřešení hlavolamu je nalézt barvu spodní stěny. Barvu spodní stěny je možné zjistit tak, že z množiny všech barev hlavolamu jsou odstraněny všechny barvy středových kostiček ze střední vrstvy. Po nalezení takové barvy stačí hledat tuto barvu na středových kostičkách spodní vrstvy, a nenachází-li se na spodní stěně, přesunout je tam rotací příslušného sub-tetraedru.

2. **Zarovnání rohů** – Nachází-li se tři středové kostičky stejné barvy na jedné stěně, dalším krokem k vyřešení je sladění rohových kostiček s barvami středů.

3. **Hrany první vrstvy** - Posledním krokem k vyřešení první vrstvy je vložení správných hranových kostiček. Nyní je potřeba se soustředit pouze na hranové kostičky střední vrstvy a hledat ty, které obsahují barvu spodní stěny. Hranová kostička obsahuje mimo barvu spodní vrstvy ještě i druhou barvu. Provede se otočení horního sub-tetraedru (**U** nebo **U'**) tak, aby se tato druhá barva shodovala s barvou středové a rohové kostičky spodní vrstvy. Nachází-li se vkládaná hranová kostička na levé straně, je potřeba provést algoritmus Algo. 2.5.1 pro vložení do spodní vrstvy, v opačném případě je nutné provést algoritmus Algo. 2.5.2. Analogicky se postupuje i pro zbylé hrany spodní vrstvy.

   V případě, že ani jedna ze 3 hranových kostiček střední vrstvy neobsahuje barvu spodní vrstvy, znamená to, že všechny hledané hranové kostičky se nachází ve spodní vrstvě a jsou pouze špatně orientované nebo prohozené. Proto je potřeba prohodit libovolnou kostičku ze střední vrstvy za spodní. K tomu se využije libovolný z algoritmů Algo. 2.5.1, Algo. 2.5.2. Po provedení se dostane hranová kostička s barvou spodní stěny do střední vrstvy. Vložením všech hranových kostiček správně do spodní vrstvy bude tato vrstva vyřešena.

4. **Střední vrstva a horní roh** – První krok v této fázi je zarovnat horní rohovou kostičku s barvami středů střední vrstvy. Následně proběhne zarovnání středové kostičky střední vrstvy s barvou spodní již vyřešené vrstvy. Nyní se může hlavolam nacházet v jednom z těchto stavů:

[7] Algoritmy převzaty z https://youtu.be/xIQtn2qazvg [navštíveno 1/2/2021]

4.1. **Dvě nevyřešené hrany** – Stav, v němž jsou pouze dvě hrany na celém hlavolamu, které nejsou vyřešeny. Pro vyřešení těchto hran je třeba držet kostku tak, aby obě nevyřešené hrany byly na přední stěně ve střední vrstvě. Následně provedením algoritmu Algo. 2.5.3 bude Pyraminx vyřešen.

4.2. **Dvojice hran** – V tomto stavu není ani jedna hrana střední vrstvy vyřešena, ovšem barvy hran jsou na stěnách sladěny. Rozlišují se zde dva algoritmy v závislosti na barvách hran a barvy stěny přilehlé k držiteli. Shodují-li se barvy hran na levé stěně s barvou přilehlé stěny, provede se algoritmus Algo. 2.5.4, v opačném případě se provede Algo. 2.5.5 a Pyraminx bude ve vyřešeném stavu.

4.3. **Tři nevyřešené hrany** – Poslední případ, v němž není ani jedna hranová kostička střední vrstvy vyřešena. Dvě hrany se ovšem shodují alespoň jednou svojí barvou s barvou stěny. Celý hlavolam je potřeba otočit tak, aby kostička, jejíž barvy se neshodují ani s jednou přilehlou stěnou, byla vzadu. Následně se provede jeden ze dvou algoritmů podle toho, na které straně přední stěny je nevyřešená kostička, která nezapadá barvou. Je-li tato kostička vlevo, provede se algoritmus Algo. 2.5.6, v případě, že je kostička napravo provede se Algo. 2.5.7 a tímto bude Pyraminx vyřešen.

Kapitola 3

Počítačové vidění

Počítačové vidění je odvětví výpočetní technologie, které se zaměřuje na zpracování nasnímaného obrazu s cílem napodobit vidění inteligentních tvorů. K nasnímání obrazu bývají zpravidla využívány videokamery či fotoaparáty. Nasnímaná obrazová data jsou následně transformována buď do rozhodnutí, nebo do nové reprezentace dat. Rozhodnutí mohou být například "Nachází se v obrazu osoba?" nebo "Je na fotografii 6 malých čtverců různých barev?". Novou reprezentací dat může být převedení barevného obrazu do černobílého nebo například odstranění šumu z obrazu, rozostření, korekce jasu apod. [12].

Využití nachází počítačové vidění v nejrůznějších oblastech – od zdravotnictví, průmyslu přes zemědělství až po autonomní řízení vozidel.

Tato kapitola pojednává o knihovně počítačového vidění OpenCV a popisuje způsob prahování, jež je v práci použitý pro extrakci barev hlavolamů.

3.1 OpenCV

Projekt OpenCV byl odstartován roku 1999 firmou Intel, jejíž motivací k vytvoření knihovny pro počítačové vidění bylo poskytnout univerzálně dostupnou a optimalizovanou infrastrukturu pro aplikace počítačového vidění. Tuto infrastrukturu by ke svému účelu mohly využívat výzkumné skupiny a společnosti po celém světě zabývající se zpracováním obrazu a umělou inteligencí. Postupem času se do projektu zapojily ostatní velké společnosti jako Google, Yahoo, Microsoft, IBM, Sony, Honda, Toyota a projekt se pomalu stával soběstačným. V roce 2012 převzala podporu projektu nezisková organizace OpenCV[1], která se stará o správu knihovny a uživatelské dokumentace [12].

Knihovna je šířena jako open-source pod licencí BSD, která umožňuje kód jakkoliv upravovat a dále šířit, ať už komerčně či nekomerčně, s povinností pouze uvést jméno autora a kopii licence. OpenCV je implementováno v programovacím jazyce C/C++. V dnešní době je OpenCV podporováno i jazyky Python, Java a Matlab a běží na platformách Windows, Linux, Android a MacOS. Implementuje více než 2500 optimalizovaných algoritmů pro počítačové vidění a strojové učení. Tyto algoritmy se využívají pro detekci a klasifikaci objektů, rozpoznávání obličejů, sledování pohybujících se objektů ve videu, rozpoznávání lidských gest a pohybů z videa, nalezení podobnosti obrázků, vytvoření 3D modelů z detekovaných objektů a mnoho dalšího [26].

V práci je využita OpenCV verze 4.5.1 pro Javu. K propojení C++ knihovny s Javou je využito rozhraní JNI (Java native interface), které umožňuje volat z Javy funkce knihoven

[1] https://opencv.org/

napsaných v cizích programovacích jazycích jako například C, C++ [2]. OpenCV verze 4.5.1 pro Javu obsahuje 27 modulů poskytujících funkce pro práci s obrazovými daty. V práci jsou využity pouze moduly **Core** a **Imgproc**, proto budou konkrétněji popsány jen tyto dva. Dokumentace všech modulů je dostupná na [27].

3.1.1 OpenCV Core

Tento modul implementuje základní datové typy pro práci s obrazem. Jsou jimi například typy `Point`, `Point3`, které znázorňují bod v rovině nebo v prostoru, typ `Rect` znázorňující obdélník v rovině nebo také typ `Scalar` pro skalární hodnotu. Dále obsahuje nejdůležitější datový typ, v němž jsou uložena obrazová data – matici. V modulu je základní třída `Mat`, z níž jsou odvozené ostatní typy matic uchovávající buď základní datové typy (`MatOfByte`, `MatOfInt`, `MatOfDouble`, ...), nebo odvozené datové typy (`MatOfPoint`, `MatOfRect`). Instance třídy `Mat` musí obsahovat i typ matice, který stanovuje, jak s daty v matici pracovat. Obecný formát tohoto typu je **CV_<bitová_hloubka>{U|S|F}C<počet_kanalů>**, kde `bitová_hloubka` je počet bitů na kanál, potom následuje jeden z datových typů U – unsigned integer, S – signed integer, F – float a `počet_kanálů` je počet barevných kanálů. V tom případě obraz se třemi kanály RGB a hloubkou 8 bitů na kanál by byl uložen jako matice typu **CV_8UC3** a tentýž obraz převeden do černobílé by byl typu **CV_8UC1** nebo zkráceně jen **CV_8U**.

Velice důležitou třídou v tomto modulu je třída `Core` obsahující implementace funkcí pro operace s maticemi. Mezi tyto operace patří základní matematické operace s maticemi – sčítání, odečítání, násobení a dělení, funkce pro kopírování matic, transpozice matice, bitové operace mezi dvěma maticemi, porovnání matic apod.

3.1.2 OpenCV Imgproc

Modul obsahuje algoritmy pro zpracování obrazu. Funkce jsou rozděleny do několika logických celků podle operací, které jsou s obrazem prováděny.

Tyto logické celky jsou[3]:

- **Základní** – Funkce pro převádění obrazu do různých barevných prostorů, vkládání textu a kreslení čar a geometrických tvarů do obrázků, vyhlazování obrázků pomocí lineárních, ale i nelineárních konvolučních filtrů, morfologické operace, funkce pro prahování
- **Transformace** – Afinní transformace, ohraničení obrázků, detektory hran, identifikace přímek a elips pomocí Houghovy transformace
- **Histogramy** – Vytváření, porovnávání a úprava histogramů
- **Kontury (Obrysy)** – Funkce pro nalezení obrysů v binárních obrazech, aproximace polygonů, získávání geometrických tvarů ohraničujících kontury (obdelníky či elipsy)

3.2 Prahování

Prahování je funkce, jež obecně převádí množinu vstupních hodnot na množinu výstupních binárních hodnot. V počítačové grafice se prahování využívá k segmentaci obrazu. Vstupem

[2] https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html
[3] https://docs.opencv.org/master/d7/da8/tutorial_table_of_content_imgproc.html

pro prahovací funkci bývá zpravidla obrázek převedený do odstínů šedi. Pro takový obrázek se stanoví číselná konstanta T, která určuje práh, podle něhož pixely větší než T nabývají maximální hodnotu intenzity a hodnoty menší nebo rovny T nabývají minimální hodnotu intenzity. Matematicky lze prahování zapsat jako:

$$f(x) = \begin{cases} \text{Intensity}_{\text{max}} & \text{pro x} > \text{T} \\ \text{Intensity}_{\text{min}} & \text{pro x} \leq \text{T} \end{cases}$$

Kde x je vstupní hodnota pixelu, $f(x)$ je výstupní hodnota pixelu, T je hodnota prahu a $\text{Intensity}_{\text{max}}$ a $\text{Intensity}_{\text{min}}$ jsou maximální a minimální hodnoty rozsahu intenzity.

Obrázek 3.1: Obrázek hlavolamu Pyraminx převeden do odstínů šedi.

Obrázek 3.2: Obrázek po aplikování prahování s hodnotou prahu 127.

Existuje několik způsobů, jak zvolit správnou hodnotu prahu. Ta nejjednodušší je zobrazena na obrázku 3.2, kde byla zvolena hodnota prahu jako konstanta. Jiný způsob, jak zvolit správnou hodnotu prahu, může být například využít metodu adaptivního prahování, kde je práh proměnný v závislosti na okolí pixelu, nebo vypočítat hodnotu prahu z histogramu použitím Otsu metody popsané jejím autorem Nobuyuki Otsu [29].

Prahování pomocí rozsahu barev

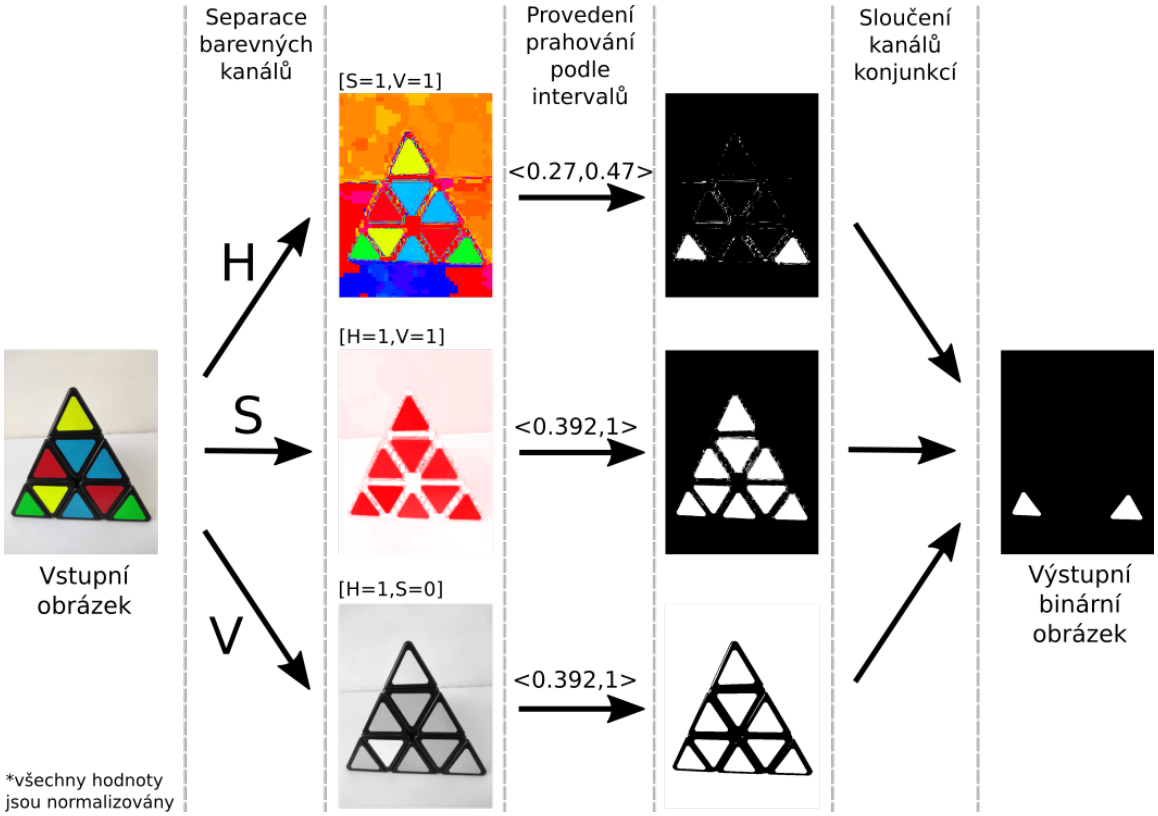
Jak již bylo zmíněno výše, prahování je většinou aplikováno na obrázek převedený do odstínů šedi. Lze je ovšem aplikovat na jakýkoliv obrázek, barevný či černobílý, s rozdílem, že u obrázků s více barevnými kanály je nutné stanovit práh pro každý kanál. V případě vstupního obrázku se třemi kanály RGB je potom potřeba definovat práh pro červený, zelený i modrý kanál zvlášť. Takto definovaná funkce může být ještě upravena tak, že kromě prahu, neboli spodní hranice intervalu přípustných hodnot, může být definovaná i horní hranice intervalu přípustných hodnot. V případě obyčejného prahování by byl rozsah přípustných hodnot pro kanál definován jako interval $<$ T, CHANNEL_MAX $>$, kde CHANNEL_MAX je maximální hodnota intenzity kanálu a T je hodnota prahu. Pro upravenou verzi s horní hranicí by potom interval byl definován jako $<$ T, UPPER_BOUND $>$, kde UPPER_BOUND je horní hranice přípustných hodnot. Výstupem takto upravené funkce je binární obrázek

s jedním kanálem, jehož hodnoty nabývají maximální intenzity, pokud pro všechny kanály platí, že hodnota pixelu spadá do definovaného intervalu, jinak nabývají minimální hodnoty intenzity. Pro 3 barevné kanály toto lze matematicky zapsat jako[4]:

$$f(x_0, x_1, x_2) = \begin{cases} \text{Intensity}_{\text{max}} & \text{pro } \text{L}_0 \leq \text{x}_0 \leq \text{U}_0 \wedge \text{L}_1 \leq \text{x}_1 \leq \text{U}_1 \wedge \text{L}_2 \leq \text{x}_2 \leq \text{U}_2 \\ \text{Intensity}_{\text{min}} & \text{jinak} \end{cases}$$

Kde x_i je hodnota pixelu i-tého kanálu, L_i je spodní hranice intervalu, U_i je horní hranice intervalu a $\text{Intensity}_{\text{min}}$ a $\text{Intensity}_{\text{max}}$ jsou minimální a maximální intenzita kanálu.

Příklad prahování zelené barvy z HSV kanálů:



Obrázek 3.3: Vizualizace prahování zelené barvy z HSV barevného prostoru.

[4]Inspirováno popisem funkce inRange() knihovny OpenCV https://docs.opencv.org/4.5.1/javadoc/org/opencv/core/Core.html

Kapitola 4

Strojové učení

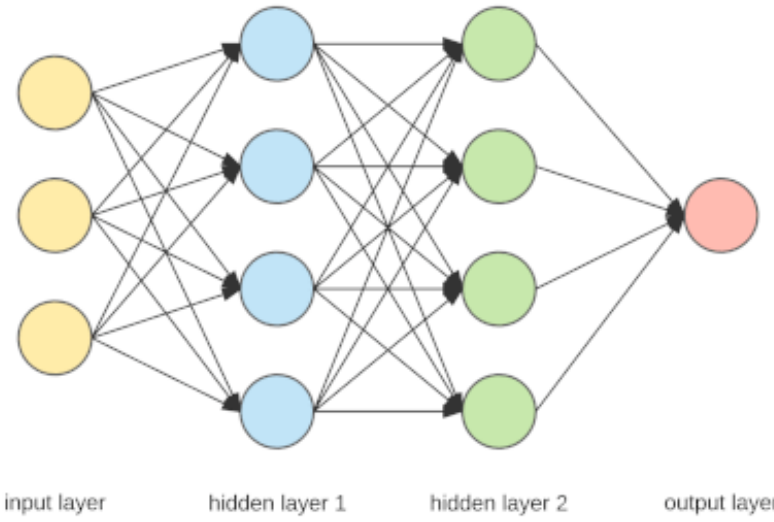
Strojové učení je podoblast oboru umělé inteligence počítače, která studuje algoritmy. Učení systémů probíhá na množině testovacích dat, na základě nichž se zdokonalují parametry systému. Již od vzniku počítačů lidé přemýšleli, zdali stroje mohou být naprogramovány tak, aby se učily a zdokonalovaly se zkušenostmi. Výsledkem by byly systémy, které by byly schopny ze zdravotnických záznamů určit nejefektivnější způsob léčby nových chorob, domy řízené počítači, které by na základě spotřeby optimalizovaly využití energií, nebo například počítače, které by člověku prezentovaly v ranních online zprávách pro něj nejrelevantnější články. Takové systémy by mohly nejen výrazně ulehčit lidský život, ale mohly by také vést k bližšímu porozumění, jak funguje učení lidí samotných [24].

V následující kapitole jsou popsány základy konvolučních neuronových sítí 4.1, architektura konvoluční neuronové sítě *MobileNetV2* 4.2 a knihovna *TensorFlow* 4.3 pro vytváření modelů neuronových sítí.

4.1 Konvoluční neuronové sítě (CNN)

Popis v této sekci vychází z [1, 2, 28].

Konvoluční neuronové sítě se využívají v oblastech rozpoznávání vzorů. Těmito oblastmi jsou převážně zpracování obrazu a mluvené řeči. Zjednodušeně řečeno jsou neuronové sítě zkonstruovány jako sekvence vrstev (viz podsekce **Typická architektura CNN**), kde každá konvoluční vrstva obsahuje sadu konvolučních filtrů (matic) pro extrakci určitých vzorů. V případě zpracování obrazu si lze představit konvoluční vrstvy na začátku neuronové sítě (blíže vstupní vrstvě) jako jednoduché filtry pro detekci vertikálních a horizontálních hran, které na základě vstupního obrázku aktivují neurony v následujících vrstvách, jež obsahují filtry pro rozpoznávání složitějších vzorů jako třeba geometrických tvarů, očí, rysů obličeje, apod.



Obrázek 4.1: Dopředná vrstvená neuronová síť, která se skládá ze vstupní vrstvy, několika skrytých vrstev a výstupní vrstvy.[1]

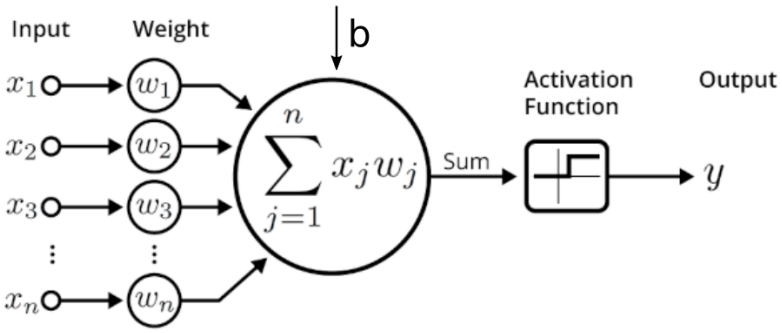
Výstupem neuronové sítě je odhad příslušnosti vstupního vektoru do některé z výstupních tříd, na jejichž rozpoznání byla síť natrénována.

Fungování neuronu

Neuronové sítě jsou inspirovány biologií mozku, jehož základními "stavebními" prvky jsou neurony. Tyto neurony posílají ostatním neuronům elektrické impulzy zaznamenají-li podnět, který je aktivuje. Neurony mozku se skládají z těla (sómy), sta až sta tisíc krátkých vstupů (dendridů) a jednoho dlouhého výstupu (axomu). Rozhraní mezi vstupem (dendridem) jednoho neuronu a výstupem (axomem) druhého neuronu se nazývá synapse, která zprostředkovává vliv jednoho neuronu na druhý. Každé rozhraní (synapse) mezi dvěma neurony má definovanou váhu, která může přenos signálu z jednoho neuronu na druhý buď posílit, či potlačit. Neurony mají definovaný jistý práh hodnoty signálu, který když součet všech vstupních hodnot signálů z dendridů překročí, neuron aktivuje. Neuron touto aktivací zašle impulz na svůj výstup (axom) a stane se na chvíli nečinným. Pokud po uplynutí doby nečinnosti neuron zaznamená na vstupu opět hodnotu součtu signálů vyšší než jeho stanovený práh, proces opakuje [40].

Zkonstruovat umělý neuron s výstupní funkcí podobnou těm biologickým by bylo obtížné, proto se využívají zjednodušené modely, jejichž výstupní signály jsou statické. Každý umělý neuron obsahuje bázovou a aktivační funkci. Bázová funkce může být buď lineární, nebo radiální a aktivační funkce může být buď skoková, nebo spojitá [40]. Model neuronu je na obrázku 4.2.

[1] Obrázek 4.1 byl převzat z https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6 [navštíveno 11/03/2021]

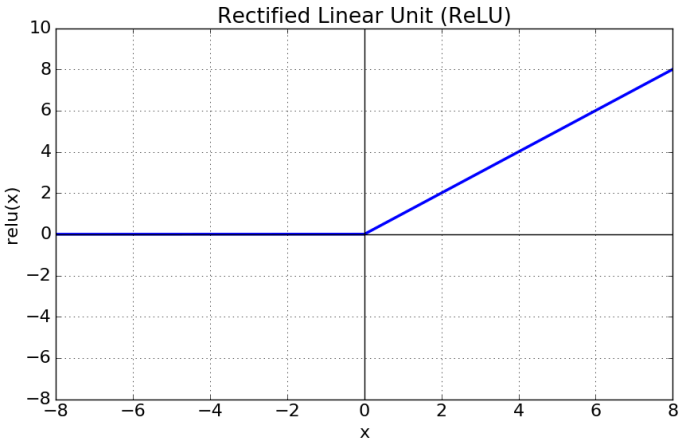


Obrázek 4.2: Model neuronu s lineární bázovou funkcí a skokovou aktivační funkcí.[2]

Aktivační funkce slouží pro aktivaci neuronu, čímž reprezentuje zaslání impulzu ostatním neuronům. Mezi nejpoužívanější aktivační funkce patří funkce **sigmoid**, **tanh** a **ReLU**. Z důvodu, že v práci je použita pouze funkce ReLU (konkrétně v architektuře MobileNetV2 popsané v sekci 4.2), bude popsána pouze tato funkce.

ReLU (rectified linear unit) je funkce, která produkuje vstup na výstup, je-li vstup větší nebo roven 0, jinak produkuje 0 na výstup. Formálně lze toto zapsat:

$$f(x) = \begin{cases} \text{x} & \text{pro x} \geq 0 \\ 0 & \text{jinak} \end{cases}$$



Obrázek 4.3: Graf ReLU funkce.[3]

Existují i obdoby ReLU funkce jako například LeakyReLU, PReLU (Parametric ReLU), ReLU6 apod.

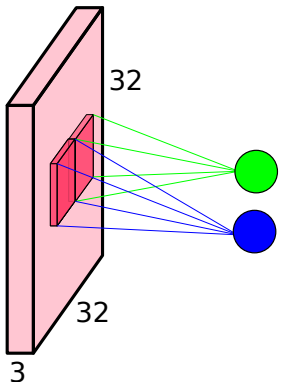
[2]Obrázek 4.2 byl převzat z https://insights.sei.cmu.edu/sei_blog/2018/02/deep-learning-going-deeper-toward-meaningful-patterns-in-complex-data.html [navštíveno 13/03/2021]

[3]Obrázek 4.3 byl převzat z https://mlnotebook.github.io/post/transfer-functions/ [navštíveno 13/03/2021]

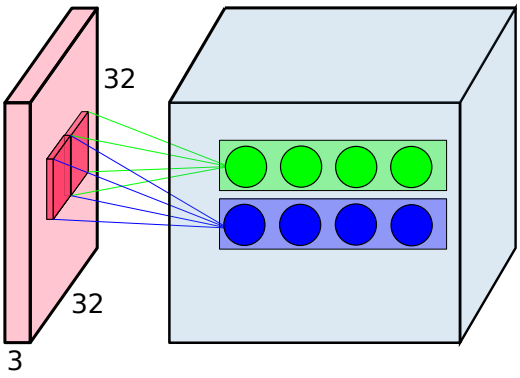
Konvoluční neuronové sítě

Konvoluční neuronové sítě (CNN) fungují na stejné bázi jako Umělé neuronové sítě (ANN) [16]. Oba tyto typy sítí jsou využívány ve strojovém učení a skládají se z neuronů, které se zdokonalují učením. Každý z těchto neuronů obdrží na vstupu vektor, nad nímž vykoná operaci definovanou bázovou funkcí následovanou aktivační funkcí. CNN, jak již bylo zmíněno dříve, jsou díky schopnosti konvoluce využívány v oblasti rozpoznávání vzorů, zejména z obrazových dat. Toto umožňuje do architektury neuronové sítě přidat vlastnosti specifické pro práci s obrázky, což z konvolučních neuronových sítí dělá ideální prostředek pro řešení problémů z oblasti počítačového vidění. I přes to, že jsou ANN schopny vypořádat se například s rozpoznáním ručně napsaných čísel z datové sady MNIST[4] [18], jejíž obrázky jsou o rozměrech $28 \times 28 \times 1$ (šířka $\times$ výška $\times$ hloubka), při vstupních obrázcích větších rozměrů a více barevných kanálech by umělé neuronové sítě musely být mnohem obsáhlejší, což má za důsledek i vyšší výpočetní náročnost.

V případě ANN by při vstupním obrázku o velikosti $32 \times 32 \times 3$ jediný neuron první skryté vrstvy měl $32 \times 32 \times 3 = 3072$ vstupů. Takový počet vstupů by lineárně rostl s každým dalším neuronem ve vrstvě a vrstva vyžaduje mnohem více než pouze jeden neuron. Tento problém velkého počtu vstupů vrstvy řeší právě CNN. Vstupní obrázek je rozdělen do menších oblastí, které jsou nazývané **lokální receptivní pole** (angl. local receptive fields). Každý neuron je potom navázán na každou z těchto oblastí, čímž je rapidně eliminován počet vstupů neuronu a tedy i celkově počet vstupů vrstvy. V případě obrázku $32 \times 32 \times 3$ a konvolučního jádra první skryté vrstvy o veliksoti 5×5 by každý neuron této vrstvy obsahoval pouze $5 \times 5 \times 3 = 75$ (musí být započítána i hloubka) váhových vstupů, což je v porovnání s původními 3072 velké zlepšení. Namapování na lokální receptivní pole je zobrazeno na obrázku 4.4. Sekvence neuronů v jednotlivých vrstvách potom pracuje pouze s lokálním receptivním polem (viz obrázek 4.5). Receptivní pole jednak zmenšují výpočetní náročnost sítě a jednak umožňují, že neuronová síť je schopna najít hledaný objekt nezávisle na umístění v obrázku.



Obrázek 4.4: Neurony první vrstvy neuronové sítě namapované na lokální receptivní pole.



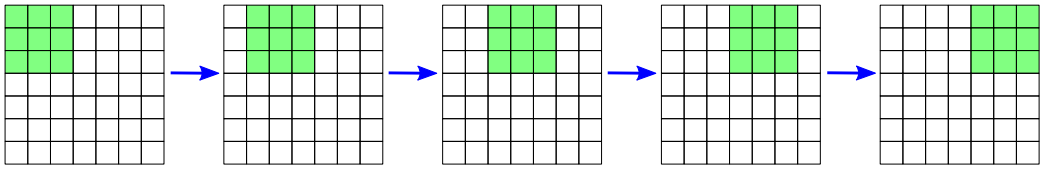
Obrázek 4.5: Sekvence neuronů pracující se stejným receptivním polem, kde každý neuron patří do jiné vrstvy a reprezentuje rozdílný konvoluční filtr.[5]

[4] MNIST je datová sada 70 000 obrázků ručně napsaných číslic http://yann.lecun.com/exdb/mnist/

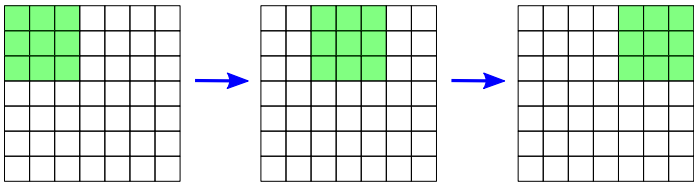
[5] Obrázky 4.4 a 4.5 byly inspirovány [2]

Krok (angl. stride) a nulová výplň (angl. zero padding)

Při dělení vstupního obrázku do jednotlivých receptivních polí je potřeba vědět velikost posuvu filtru vzhledem k předešlé pozici. Velikost posuvu je dána **krokem**. Například krok o velikosti 1 znamená, že se filtr posune o jednu buňku oproti předešlé pozici filtru. Takový případ vyústí ve spoustu překrývajících se receptivních polí, což má za důsledek mnoho aktivací. Na druhou stranu velká hodnota kroku je sice méně paměťově náročná, ovšem může způsobit ztrátu informace.

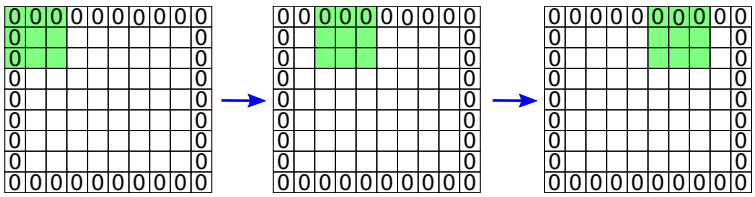


Obrázek 4.6: Posuv filtrového okna s krokem 1. Výstupní obrázek bude o rozměrech 5×5.



Obrázek 4.7: Posuv filtrového okna s krokem 2. Výstupní obrázek bude o rozměrech 3×3.

Nulová výplň je proces, při němž se okraje vstupního obrázku vyplňují nulovými hodnotami. Tento proces zajišťuje větší kontrolu nad rozměry výstupního obrázku. Jak je vidět na obrázku 4.7, ze vstupního obrázku o rozměrech 7×7 by na výstupu byl obrázek pouze 3×3. Přidáním nulové vycpávky k vstupnímu obrázku (viz obrázek 4.8) by byly rozměry výstupního obrázku 4×4.



Obrázek 4.8: Nulová výplň - přidání nul po okraji obrázku. Důsledkem přidání nulové výplně jsou větší výsledné rozměry výstupního obrázku (4×4 oproti obrázku 4.7, jehož výstup by byl pouze 3×3). Nulová výplň zajišťuje větší kontrolu nad rozměry výstupního obrázku.

Pro obrázek o rozměrech N×N a velikosti filtru F×F se velikost výstupního obrázku O dá vypočítat podle rovnice:

$$O = 1 + \frac{(N - F)}{S}$$

Kde O je šířka/výška výstupního obrázku, N je šířka/výška vstupního obrázku, F je velikost konvolučního jádra a S je velikost kroku.

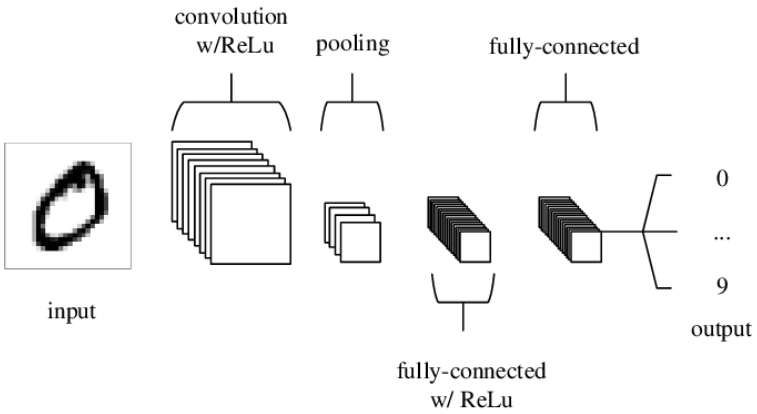
Upravená rovnice pro výpočet výstupní velikosti počítající i s velikostí nulové výplně potom vypadá:

$$O = 1 + \frac{(N + 2P - F)}{S}$$

Kde oproti předešlému rovnici je navíc P, což je počet přidaných vrstev nulové výplně.

Typická architektura CNN

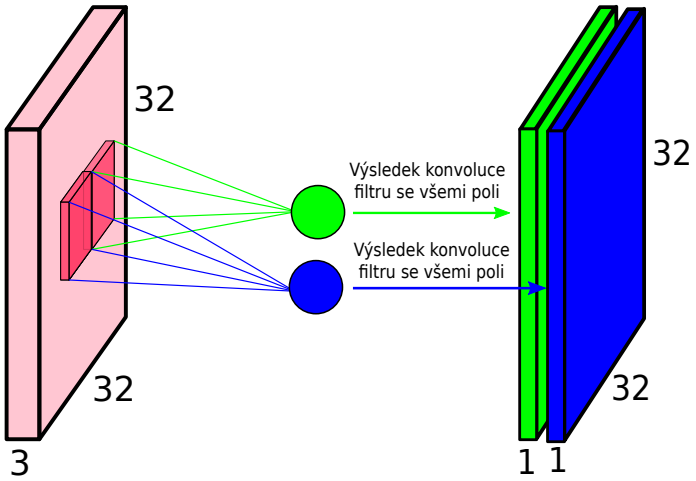
CNN se typicky skládají, nepočítaje vstupní a výstupní vrstvy, ze tří typů vrstev. Tyto jsou **konvoluční**, **pooling** a **plně-propojená** (angl. fully-connected). Skládáním jednotlivých vrstev do série vzniká konvoluční neuronová síť.



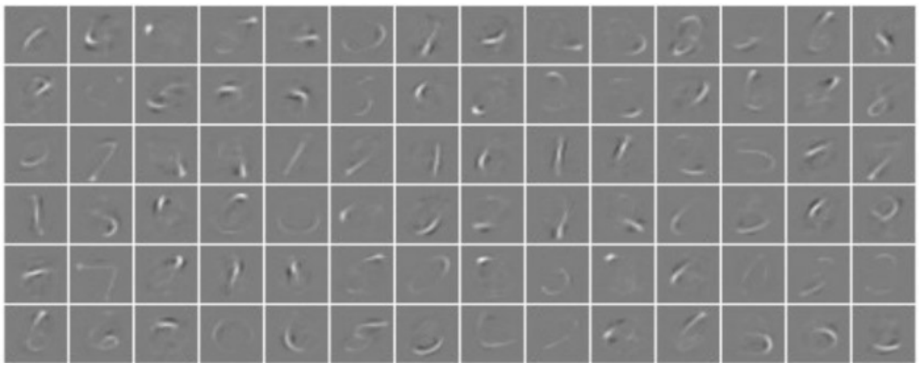
Obrázek 4.9: Jednoduchá architektura konvoluční neuronové sítě složená z 5 vrstev obsahující všechny typy vrstev běžné pro CNN.[6]

- **Konvoluční vrstva** – Konvoluční vrstva se skládá z množiny filtrů, jež je možné zdokonalovat učením. Filtry jsou využity k detekci rysů na vstupním obrázku. Konvolucí filtru s celým vstupním obrázkem vznikne **aktivační mapa**. Vstupem pro další vrstvu bude potom sada aktivačních map, kde každá mapa odpovídá výstupu jednoho neuronu. Toto je vyobrazeno na obrázku 4.10. Příklad reálně vypadajících aktivačních map první vrstvy konvoluční neuronové sítě natrénované na MNIST datové sadě je zobrazeno na obrázku 4.11.

[6] Obrázek byl převzat z [28]



Obrázek 4.10: Zobrazení vytvoření sady aktivačních map jako vstup pro další konvoluční vrstvu. Každý neuron z dané vrstvy produkuje jednu aktivační mapu.[7]

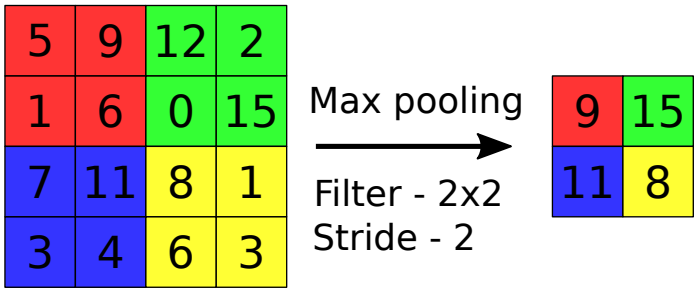


Obrázek 4.11: Aktivační mapy z první konvoluční vrstvy CNN natrénované na MNIST datové sadě pro rozpoznávání ručně napsaných číslic. Při bližším náhledu lze vidět, že mapy obsahují rysy specifické pro jednotlivé číslice 1-9.[8]

- **Pooling vrstva** – Tato vrstva provádí podvzorkování vstupního obrázku na menší výstupní obrázek. Nejpoužívanějšími dvěma typy pooling vrstev jsou max pooling vrstva a average pooling vrstva. Max pooling vybírá z oblasti dané maticí nejvyšší hodnotu, average pooling průměruje všechny hodnoty v oblasti matice. Podvzorkování pomocí max poolingu je zobrazeno na obrázku 4.12.

[7]Obrázek 4.10 byl inspirován http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf [navštíveno 12.3.2021]

[8]Obrázek byl převzat z [28]



Obrázek 4.12: Max pooling s velikostí filtru 2×2 a krokem 2.

- **Plně propojená vrstva** – Poslední vrstva konvoluční neuronové sítě bývá zpravidla plně propojená. Tato vrstva pracuje jako Multilayer perceptron vrstva (MLP), která se učí za pomocí algoritmu zpětného šíření chyby (angl. backpropagation). Na konci plně propojené vrstvy je softmax vrstva, která má za úkol převést vstupní vektor N reálných čísel na výstupní vektor N čísel mezi hodnotami 0 až 1, jejichž součet dává dohromady 1. Tento výstupní vektor N čísel obsahuje pravděpodobnosti příslušnosti vstupního vektoru do výstupních tříd.

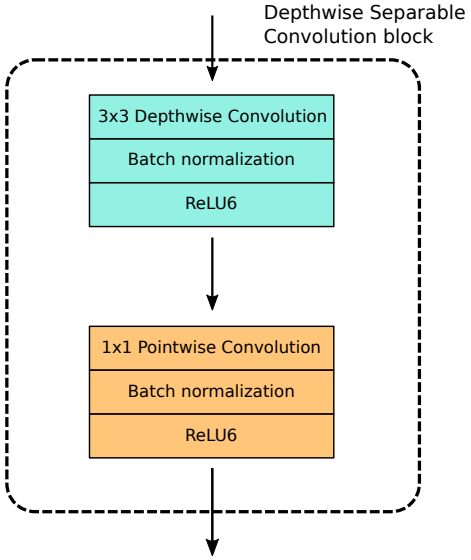
4.2 MobileNetV2

Tato sekce obsahuje stručný popis konvoluční neuronové sítě `MobileNetV2` založený na popisu uvedeném v [15, 31]. Detailnější popis architektury MobileNetV2 je k nalezení v [31].

MobileNetV2 je klasifikační konvoluční neuronová síť zaměřená na mobilní zařízení s limitovaným výpočetním výkonem (chytré telefony, tablety, raspberry pi, apod.). Architektura vychází z předešlé verze stejnojmenné architektury MobileNetV1 [17]. Vývoj obou architektur spadá pod organizaci Google Inc.

MobileNetV1

Architektura MobileNetV1 přinesla efektivnější způsob provádění konvoluce v konvolučních vrstvách [17]. Tato konvoluční vrstva nese název *Depthwise separable convolution* a je rozdělena do dvou podvrstev (obrázek 4.13). První podvrstvou je tzv. *Dephtwise convolution* (hloubková konvoluce) následovaná druhou podvrstvou tzv. 1×1 *Pointwise convolution*. *Depthwise convolution* vrstva provádí konvoluci jádra se všemi vstupními kanály individuálně. Výstupem takové konvoluce je stejný počet matic jako byl počet vstupních kanálů. Tyto výstupní matice jsou následně vstupem pro 1×1 *Pointwise convolution* vrstvu, která slouží pro vytvoření lineární kombinace matic. Takové rozdělení do podvrstev snižuje výpočetní náročnost.

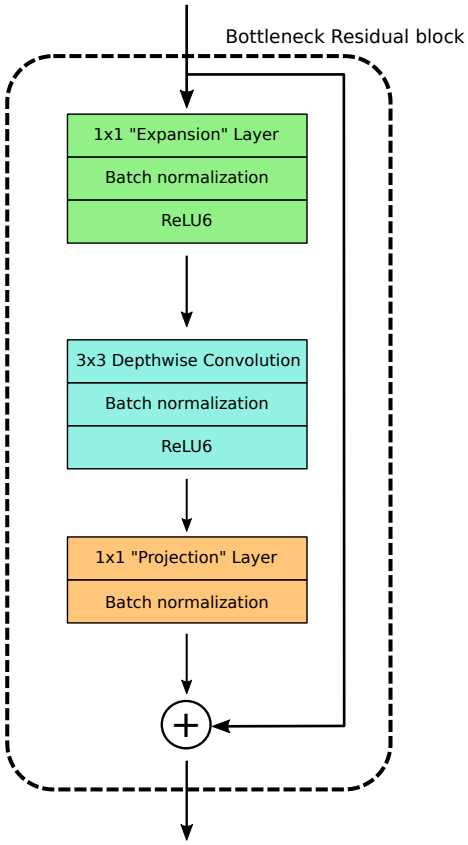


Obrázek 4.13: Depthwise separable convolution – základní stavební blok architektury MobileNetV1. Skládá se ze dvou částí – *Depthwise convolution* a *Pointwise convolution*.[9]

MobileNetV2

Zdokonalená verze architektury využívá, podobně jako verze MobileNetV1, *Depthwise seperable convolution* vrstvu, ovšem v upravené verzi, která u této verze nese název *Bottleneck Residual block*. Tato vrstva se skládá ze dvou podvrstev známých z předešlé verze architektury – *Depthwise convolution* a 1×1 *Pointwise convolution*. Navíc je zde *Expansion layer* (rozšiřující vrstva). V MobileNetV2 je vrstva *Pointwise convolution* pozměněna. Zatímco ve verzi MobileNetV1 zachovávala počet kanálů, ve verzi MobileNetV2 je tomu naopak a tato konvoluce snižuje počet kanálů. Z toho důvodu byl její název změněn na projekční – promítá data s vysokým počtem kanálů na tensor s menším množstvím kanálů. *Expansion layer*, která do tohoto bloku byla přidána, má za cíl navýšit počet kanálů vstupujících do *Depthwise convolution* vrstvy. V podstatě dělá rozšiřující vrstva opak toho, co dělá vrstva projekční. Míra, jakou jsou data rozšířena v rozšiřující vrstvě je dána hodnotou *expansion factor* (faktor rozšíření). Rozšiřující vrstva funguje jako dekomprese dat, nad nimiž je provedena hloubková konvoluce a následně jsou opět zkomprimována projekční vrstvou.

[9] Obrázek 4.13 byl inspirován [15].



Obrázek 4.14: Bottleneck Residual block – základní stavební blok architektury MobileNetV2. Skládá se z rozšiřující vrstvy, která slouží pro dekompresi dat, vrstvy hloubkové konvoluce pro konvoluci všech vstupních kanálů a projekční vrstvy, jež slouží jako zpětná komprese dat.[10]

[10] Obrázek 4.14 byl inspirován [15].

Velikost vstupního obrázku	Typ vrstvy	t	c	n	krok
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1×1	-	1280	1	1
$7^2 \times 1280$	avgpool 7×7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1×1	-	k	-	

Tabulka 4.1: Tělo architektury MobileNetV2 (převzato z [31]). Každý řádek tabulky popisuje sekvenci 1 či více identických konvolučních vrstev opakujících se n-krát. Všechny vrstvy v jedné sekvenci mají stejný počet c výstupních kanálů. Každá první vrstva sekvence má krok o velikosti s, zbylé vrstvy mají krok velikosti 1. Všechny prostorové konvoluce využívají jádro o velikosti 3×3. Faktor rozšíření t je vždy aplikován na vstupní velikost obrázku.

4.3 TensorFlow

TensorFlow je open-source knihovna pro numerické počítání a strojové učení využívající data-flow grafů. Knihovna vznikla v listopadu roku 2015. Jejími autory byli členové týmu Brain z firmy Google a knihovna sloužila jako nástupce systému pro tvorbu modelů neuronových sítí DistBelief [13]. TensorFlow poskytuje high-level API pro tvorbu modelů neuronových sítí. Front-end tohoto API je napsaný v programovacím jazyce Python, ovšem binární soubory pro výpočty jsou naprogramovány v jazyce C++ pro jeho výpočetní efektivitu [39]. Natrénované modely jsou multiplatformní a lze je spustit téměř na všech zařízeních – serverech, počítačích i mobilních zařízeních. Trénování modelu je možné provádět na počítačových architekturách využitím buď CPU, GPU za pomocí technologie Nvidia CUDA[11] nebo speciálně upravených jednotek, které lze využívat pouze při trénování na Google cloudech, TPU (TensorFlow processing unit). Samotná TensorFlow knihovna reprezentuje dostatečně vysokou míru abstrakce pro vytváření grafů a vrstev neuronových sítí. V případě nutnosti vyšší míry abstrakce mohou uživatelé knihovny využít funkcí knihovny Keras[12], kterou TensorFlow podporuje.

4.3.1 TensorFlow Lite Model Maker

Modely natrénované pomocí TensorFlow API jsou ukládány jako soubory ve formátu `pb`. Tyto soubory reprezentují serializované modely s parametry společně i s programem pro trénování. Takto uložené předtrénované modely je možné znovu trénovat na jiných platformách. Pro využití natrénovaného modelu na mobilních zařízeních je potřebné soubor ve formátu `pb` převést na soubor formátu `tflite`, který je podporovaný mobilními knihovnami pro práci s modely neuronových sítí. Jednou z možností, jak z natrénovaného modelu získat

[11] CUDA je sada nástrojů společnosti Nvidia umožňující programátorům tvořit aplikace akcelerované na grafických čipech

[12] https://keras.io/

verzi pro mobilní zařízení, je využít *TensorFlow Lite Model Maker* [34]. Nástroj umožňuje načíst natrénovaný soubor, který pouze převede do souboru formátu `tflite`, nebo je možné pomocí tohoto nástroje model natrénovat a uložit jako `tflite`. Model je možné načíst lokálně, ze sítě, nebo přímo z databáze modelů TensorFlow Hub.

4.3.2 TensorFlow Hub

TensorFlow Hub je databáze modelů, kam nejen Google, ale i uživatelé přidávají předtrénované modely. Jsou tam k nalezení modely předtrénované na rozsáhlých datových sadách ať už pro klasifikaci obrázků, detekci objektů v obrazech, tak i pro rozpoznávání řeči, textu apod. Předtrénované modely umožňují uživatelům přetrénovat model pro jejich specifický případ a nevyžadují tak rozsáhlou datovou sadu jako kdyby uživatel trénoval model od samého začátku [33].

Kapitola 5

Operační systém Android

Android je světově nejrozsáhlejší operační systém pro mobilní platformy. Je založen na linuxovém jádře a šířen jako open-source projekt zaštiťován firmou Google. Operační systém byl původně zamýšlen hlavně pro mobilní zařízení s dotykovými displeji (telefony, tablety), každopádně v dnešní době je možné operační systém Android najít například v hodinkách, televizích, domácích spotřebičích jako ledničkách či vysavačích nebo například i v automobilech.

První verze operačního systému Android 1.0 byla vydána v září roku 2008 [10]. Tato verze nenesla žádné pojmenování, ovšem již v té době obsahovala základní balíček aplikací firmy Google jako například Gmail, Google Contacts, Google Maps apod. První pojmenovanou verzí byla verze Android 1.5 (první verze s jádrem Linuxu), která nesla pojmenování Cupcake. Vývojáři platformy Android se rozhodli, že verze operačního systému budou pojmenovávat abecedně podle názvů sladkostí. Tím pádem existují verze jako například JellyBean (4.1), Kitkat (4.4), Lollipop (5.0) apod. [23]. Momentálně nejaktuálnější stabilní vydanou verzí je verze Android 11.

Každá verze přináší nové funkce do SDK (software development kit), které vývojáři Android aplikací mohou využívat. Tyto verze jsou označovány jako API X, kde X je přirozené číslo. Každá novější verze API musí být kompatibilní s předchozími verzemi, aby i staré Android aplikace fungovaly na nových zařízeních. Nejnovější API verze pro Android 11 je API 30 [9].

5.1 Základní prvky Android aplikací

Android aplikace musí obsahovat některé základní prvky pro jejich správné fungování. Sekce stručně popisuje tyto prvky, na nichž je postaven vývoj aplikace.

5.1.1 Android activity

Aktivita v aplikaci slouží pro načtení grafického obsahu okna a pro celkovou interakci s uživatelem. Aktivita (angl. Activity) je speciální Java třída, která definuje, jaký grafický obsah má být načten a jak má aplikace reagovat na vstupy od uživatele. Například pokud součástí grafického obsahu je tlačítko, v kódu aktivity musí být definována funkčnost tohoto tlačítka [14].

Aktivita se může nacházet ve 4 stavech na základě toho, zdali momentálně aplikace běží na popředí, na pozadí nebo je úplně pozastavena. Toto je nazýváno *životní cyklus aktivity* (angl. Activity lifecycle). *Životní cyklus aktivity* je definovaný několika metodami

v abstraktní třídě Activity. Třídy dědící třídu Acitivity potom mohout (v některých případech musí) tyto metody implementovat. Povinná metoda, která musí být implementována je metoda `onCreate`, která je zavolaná při prvním vytvoření aktivity a uvnitř níž dochází k načtení grafického obsahu. Také je doporučeno uvnitř této metody načíst veškerá statická data a obecně provést úkony, které je potřeba pro funkčnost uvnitř aktivity provést pouze jednou. Další metody jako například `onStart`, `onResume`, `onPause` nebo `onDestroy` není povinné implementovat, ovšem jejich implementace a správné použití zajistí, že aplikace bude fungovat mnohem efektivněji bez zasekávání. Více o životním cyklu aktivity na [4].

5.1.2 Layout

Celá podsekce vychází z [14].

Typická Android aplikace se skládá z jedné či více stránek. Grafické rozvržení těchto jednotlivých stránek je v Androidu popsáno pomocí značek v XML souboru. Tento XML soubor zpracuje Android aktivita a vykreslí grafický obsah na displej mobilního zařízení. Při tvoření grafického rozvržení je možné využít buď předdefinovaných grafických prvků (XML značek), nebo definovat vlastní. Příkladem grafických prvků mohou být tlačítka, textová pole, obrazová pole apod.

5.1.3 AndroidManifest.xml

Každá aplikace musí mít definovaný `AndroidManifest.xml` soubor v kořenovém adresáři aplikace. Soubor obsahuje důležité informace o aplikaci pro sestavovací nástroje Androidu, pro operační systém nebo pro GooglePlay obchod [7].

Uvnitř manifest souboru je definovaná cesta k ikoně aplikace, název aplikace, navigace mezi aktivitami, nebo například periferie, k nimž aplikace vyžaduje přístup (např. GPS, kamera).

5.2 Android architektura

Architektura Androidu sestává z šesti na sebe navazujících vrstev. Každá slouží jako rozhraní pro nadřazenou vrstvu, čímž zároveň zvyšuje míru abstrakce systému. Podkapitola vychází z popisu uvedeném v [6].

1. **Linux kernel** – Základem platformy Android je jádro operačního systému Linux. Například `Android Runtime` vrstva (popsaná později) je závislá na jádru operačního systému Linux, které využívá pro práci s vlákny a správu paměti na nízké úrovni.

   Jádro operačního systému Linux navíc poskytuje dostatečnou míru zabezpečení a dovoluje výrobcům mobilních zařízení vytvářet hardwarové ovladače pro dobře známe jádro.

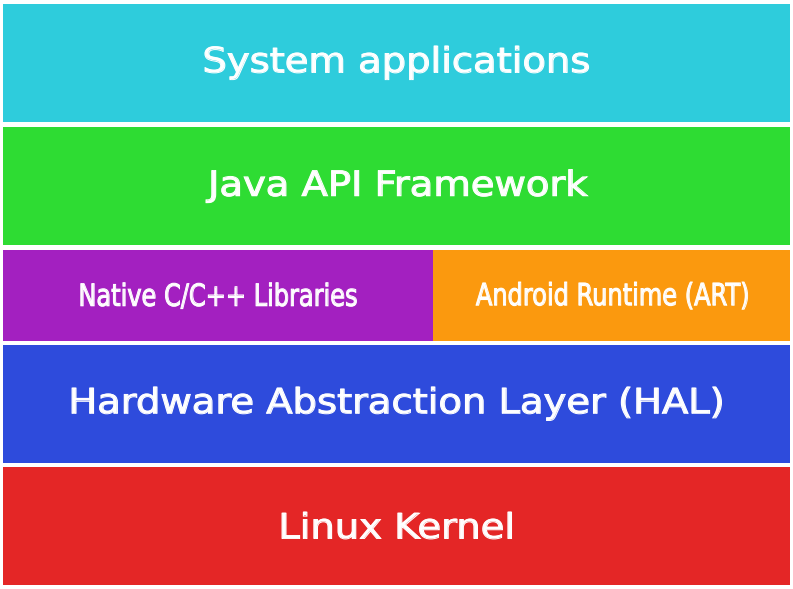
2. **Hawrdware Abstraction Layer (HAL)** – `Hardware Abstraction Layer` (vrstva hardwarové abstrakce) poskytuje standardní rozhraní mezi zařízeními telefonu (např. mikrofon, bluetooth, kamera, snímače tepu apod.) a `Java API` vrstvou. Díky této vrstvě mohou vývojáři aplikací využívat v jejich Java kódu zařízení telefonu bez starostí o jeho správu na nízké úrovni. HAL vrstva obsahuje sadu knihoven, v níž každá knihovna odpovídá jednomu zařízení. Využívá-li aplikace některé z hardware zařízení, operační systém načte knihovnu daného zařízení pro jeho obsluhu.

3. **Android Runtime (ART)** – V zařízeních obsahující Android verzi 5.0 (API 21) a vyšší má každá aplikace svůj vlastní proces a svoji vlastní instanci `Android Runtime` (běhové prostředí). ART je optimalizovaný k tomu, aby byl schopný provozovat několik virtuálních strojů na mobilních zařízeních s nízkou pamětí. Tohoto je dosaženo spouštěním tzv. DEX souborů, které obsahují bytecode speciálně optimalizovaný pro Android zařízení na využití co nejméně paměti. Tyto DEX soubory jsou vytvářeny Android kompilátory zdrojových Java kódů.

4. **Native C/C++ libraries** – Spousta systémových služeb a komponent, jako například ART nebo HAL, vyžaduje k sestavení nativní C/C++ knihovny. Android poskytuje `Java API framework`, aby aplikace napsané v Java kódu mohly využívat funkce těchto C/C++ knihoven. Příkladem mohou být funkce knihovny `OpenGL ES` pro vykreslování 2D či 3D grafiky v mobilním zařízení. Dalším příkladem může být knihovna `OpenCV`, jejíž zdrojové kódy jsou napsané v C/C++, ovšem Android API umožňuje jejich volání prostřednictvím Java frameworku.

5. **Java API Framework** – Veškerá funkcionalita, kterou operační systém Android nabízí, je dostupná skrze API napsané v jazyce Java. Jedná se o stejné API, jež je zmiňováno na začátku kapitoly v souvislosti s SDK (software development kit). Toto API slouží jako základní stavební blok pro vývoj Android aplikací. Umožňuje modulárně využívat systémové komponenty a služby jako například `Activity manager`, který se stará o životní cyklus aplikací, `Resource manager` umožňující přístup ke zdrojům jako například ke grafice, XML souborům popisující rozvržení grafického obsahu, řetězců využívaných v aplikacích, `Package manager`, `Notification manager` a další.

6. **System applications** – Na samém vrcholu zásobníku vrstev stojí vrstva systémových aplikací. Tato vrstva obsahuje jednak systémové aplikace pro správnou funkcionalitu zařízení, jednak uživatelské aplikace, které si uživatel může nainstalovat na své zařízení. Tyto aplikace využívají služeb všech nižších vrstev.



Obrázek 5.1: Architektura operačního systému Android.[1]

5.3 Knihovna CameraX

Knihovna CameraX je součástí kolekce knihoven `Android Jetpack`. Knihovny balíčku Android Jetpack pomáhají vývojářům dodržovat osvědčené postupy při programování, zredukovat množství kódu a zajistit zpětnou kompatibilitu se všemi zařízeními [5].

Knihovna CameraX poskytuje funkce pro přístup ke kameře mobilního zařízení. Využívá komponentu `camera2` z Android API a dbá na správné napojení kamery na životní cyklus aktivity. Také řeší problém kompatibility s různými mobilními zařízeními a redukuje nutnost psát kód specifický pouze pro některá zařízení [8]. Vývojář může pro práci s kamerou využít tři případy užití, které knihovna nabízí.

1. **Preview** – Tento případ užití zobrazuje náhled z kamery. Vyžaduje v grafickém návrhu specifikovat pohled `PreviewView`, do nějž je tok obrázků z kamery přesměrován.
2. **Image analysis** – Případ užití pro analýzu obrázků z kamery. Zpravidla jsou do tohoto případu užití posílány obrázky v menším rozlišení pro zajištění analýzy v reálném čase. Případ užití ovšem umožňuje specifikovat preferované rozlišení, preferovaný poměr stran obrázku nebo zdali analyzovat vždy nejaktuálnější obrázek z kamery nebo obrázky řadit do fronty.
3. **Image capture** – Případ užití pro vyfocení snímku z kamery a uložení do úložiště mobilního zařízení.

Případy užití mohou být mezi sebou kombinovány. Před zaregistrováním případů užití a navázáním na životní cyklus aktivity je nutné vybrat kameru mobilního zařízení (přední nebo zadní) a specifikovat její konfiguraci (může být využita i výchozí konfigurace).

5.4 Knihovna ML kit

ML kit je knihovna společnosti Google umožňující práci s modely neuronových sítí na mobilních zařízeních. Klasifikace modelem probíhá přímo na zařízení, což přináší možnost provádět klasifikaci v reálném čase a bez nutnosti připojení k internetu. Knihovna poskytuje API k práci s různorodými modely jako například pro rozpoznávání textu z obrazu, detekci obličeje, skenování čárových kódů, detekci a sledování objektů v obrazu nebo klasifikaci obrázků.

Jako modely neuronových sítí je možné využít předtrénované modely zveřejněné týmem ML kit na TensorFlow Hub[2] (4.3.2) nebo si natrénovat model vlastní. Knihovna podporuje využití vlastních modelů neuronových sítí pouze pro detekci a sledování objektů v obraze nebo klasifikaci obrazů. Podporovaný formát modelu je `tflite`. Způsob, jak takový model natrénovat, je využít `TensorFlow Lite Model Maker` (4.3.1) nebo využít zpoplatněnou Google službu `AutoML Vision Edge`[3].

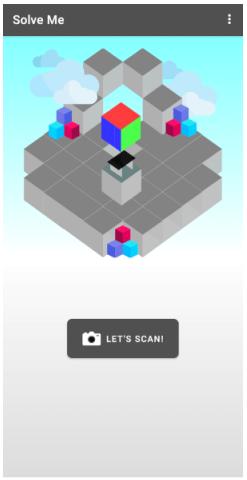
[1] Obrázek 5.1 byl inspirován [6]
[2] https://tfhub.dev/ml-kit
[3] https://firebase.google.com/docs/ml/automl-image-labeling

Kapitola 6

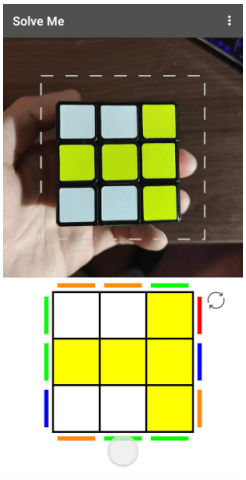
Implementace

Má aplikace nese název **Solve Me**. Cílová platforma, pro níž je aplikace implementována, je Android, který byl blíže popsán v kapitole 5. Minimální podporovaná verze Androidu je Android 7.1 Nougat (verze API 25). Při vytváření implementačního návrhu jsem se snažil aplikaci navrhnout tak, aby nebylo příliš složité přidat do aplikace nový typ hlavolamu nebo i rozšířit aplikaci o nějaké prvky.

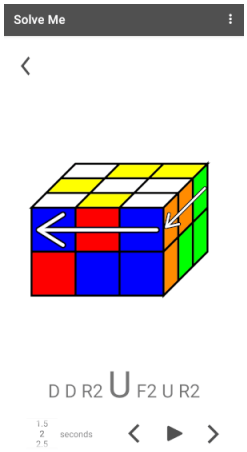
V kapitole je popsán nástroj *Albumentations* využitý pro úpravu obrázků za účelem rozšíření datové sady a dále pak jsou zde popsané tři hlavní moduly, z nichž se aplikace skládá, jejich realizace a funkčnost.



Obrázek 6.1: Úvodní stránka aplikace.



Obrázek 6.2: Skenování hlavolamu.



Obrázek 6.3: Předvedení řešení.

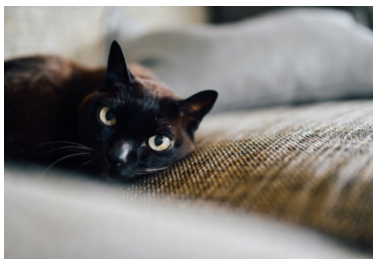
6.1 Knihovna albumentations

Albumentations je Python knihovna poskytující API pro úpravu velkých sad obrázků. Takto upravené datové sady je možné potom využít při trénování konvolučních neuronových sítí. Knihovna implementuje bohatou škálu funkcí pro úpravu obrázků od obyčejných rotací přes změny jasu, rozostřování až po prohazování barevných kanálů a mnoho dalšího. Tyto funkce pro úpravu lze serializovat a přidat jim určitou pravděpodobnost aplikace, což přidává na variabilitě výsledného obrázku a umožňuje vygenerovat z jednoho obrázku několik rozdílných.

Základem knihovny je pipeline, která definuje sérii úprav a jejich pravděpodobnost aplikace. Příkladem takové pipeline může být: (převzato z [3])

```
    transform = A.Compose([
    A.RandomCrop(width=256, height=256),
    A.HorizontalFlip(p=0.5),
    A.RandomBrightnessContrast(p=0.2),
])
```

`RandomCrop`, `HorizontalFlip` a `RandomBrightnessContrast` jsou funkce upravující obrázek a `p=<0;1>` (implicitně je `p=1`) je pravděpodobnost aplikace dané úpravy. Obrázek následně projde celou sérií transformací v pořadí, jak byla definována, každá se na něj aplikuje (v závislosti na pravděpodobnosti) a výsledkem je upravený obrázek. V příkladu ukázaná pipeline je poměrně jednoduchá, lze vytvořit mnohem komplexnější pipeline. Možné výstupy po aplikování výše definované pipeline jsou na obrázcích 6.5, 6.6.



Obrázek 6.4: Originální obrázek.

Obrázek 6.5: Jeden z možných výstupů po aplikování transformací na původní obrázek.

Obrázek 6.6: Další z možných výstupů po aplikování transformací na původní obrázek.

Výše použité obrázky byly převzaty z [3].

6.2 Scanner

Prvním a zřejmě i nejdůležitějším modulem aplikace je **Scanner**, který slouží pro naskenování barevné konfigurace hlavolamu. Scanner má čtyři hlavní úkoly:

1. Obsluhu a náhled z kamery
2. Detekci barev skenované stěny
3. Vizualizaci nadetekovaných barev a vedení uživatele při skenování
4. Rozpoznání hlavolamu z naskenovaných stěn

Obsluha kamery

Kód obsluhy kamery se nachází ve třídě `CameraActivity`. Vzhledem k tomu, že Android aplikace vyžadují oprávnění uživatele pro práci s periferiemi, obsahuje třída `CameraActivity` kód, který kontroluje toto oprávnění přístupu ke kameře, a nemá-li aplikace přístup povolen, vyzve uživatele k udělení oprávnění. Pro obsluhu kamery jsem využil dva případy užití z knihovny CameraX – `Image analysis` a `Preview`. Tyto dva případy užití byly blíže popsány v sekci 5.3. Image analysis případ užití využívám pro analýzu obrázku, z něhož

následně za použití funkcí z knihovny OpenCV (3.1) zjišťuji barevnou konfiguraci stěny. Preview případ užití slouží k náhledu z kamery v reálném čase, aby uživatel mohl správně naskenovat hlavolam. Využití Image analysis případu užití popíšu později v této sekci, nyní popíšu využití Preview případu užití. Pro eliminaci okolního rušení u náhledu z kamery jsem přidal do okna náhledu obdélník, který ohraničuje oblast, v níž je uživatel nucen držet hlavolam pro správné naskenování stěny (viz obrázek 6.7). V aplikaci je pro Preview případ užití nastaven atribut poměru stran obrázku na hodnotu 4:3, tento poměr stran nastavuji i při vytváření Image analysis případu užití, aby se poměry stran vstupních obrázků obou případů užití shodovaly a bylo tak jednodušší tento obrázek v Image analysis ořezat na ohraničující obdélník. V neposlední řadě je pro správnou funkčnost kamery potřeba navázat případy užití na životní cyklus aktivity (životní cyklus byl popsán v podsekci 5.1.1). Toto provádím v metodě `bindPreviewAndAnalysis`, v níž vybírám kameru (přední nebo zadní) a definuji všechny atributy pro oba případy užití a následně případy užití navážu na životní cyklus funkcí `bindToLifecycle` z knihovny CameraX. Pokud by neproběhl poslední krok navázání na životní cyklus aktivity, kamera by při zamčení mobilního zařízení nebo při přechodu do pozadí a následném vrácení do aplikace nemusela fungovat správně.

Obrázek 6.7: Ohraničující obdélník s hlavolamem uvnitř. Obdélník vymezuje uživateli místo, v němž musí držet hlavolam pro správné skenování, a také funguje jako eliminace okolního šumu při zpracovávání obrazu.

Klasifikace stěny z obrazu a datová sada

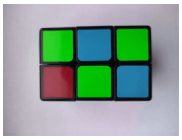
V aplikaci jsem pro rozpoznání stěny zvolil klasifikaci za pomocí konvoluční neuronové sítě. Konvoluční neuronové sítě byly popsány v sekci 4.1. V této práci použitý model neuronové sítě je založen na architektuře **MobileNetV2**, která byla popsána v sekci 4.2. Konkrétně je v aplikaci použitý model **mobilenet_v2_100_192**[1] předtrénovaný na datové sadě ImageNet[2] s hodnotou depth multiplier 1 a vstupním obrázkem o rozměrech 192×192. Tento model jsem přetrénoval pomocí nástroje TensorFlow Lite Model Maker (popsaný v podsekci 4.3.1) na vlastní datové sadě. Datové sady jsem vytvořil dvě verze. První verzi

[1] https://tfhub.dev/google/imagenet/mobilenet_v2_100_192/feature_vector/4

[2] ImageNet je internetová databáze více než 14 mil. obrázků, v níž každý obrázek nese množinu slovních označení věcí nacházejících se na obrázku. Více na https://imagenet.stanford.edu/about.php

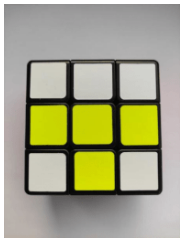
s obrázky jednotlivých stěn hlavolamů na nerušivém jednobarevném pozadí, s případným stínem či odleskem na stěně a různými hodnotami jasu. Druhá verze obsahovala obrázky stěn hlavolamů s rušivým pozadím a jinými objekty v pozadí. V obou verzích bylo ~300 fotek od každého typu stěny. Typy stěn jsem si zvolil jako **2x2**, **3x3**, **3x2_horizontal**, **3x2_vertical** a **pyraminx**. Při prvním natrénování dosahoval lepších výsledků při klasifikaci model natrénovaný na druhé verzi datové sady, ovšem často se dělo, že stěny nepravdivě klasifikoval jako pyraminx.

(a) 2x2



(b) 3x2_horizontal

(c) 3x2_vertical



(d) 3x3

(e) pyraminx

Obrázek 6.8: Ukázka fotografií z první verze mé datové sady. Tato verze obsahuje fotografie stěn hlavolamů na čistém jednobarevném pozadí.

(a) 2x2

(b) 3x2_horizontal

(c) 3x2_vertical

(d) 3x3

(e) pyraminx

Obrázek 6.9: Ukázka fotografií z druhé verze mé datové sady. Tato verze obsahuje fotografie stěn hlavolamů s rušivým pozadím a jinými objekty v pozadí.

Následně jsem zdokonalil obě verze datových sad použitím nástroje **Albumentations**, jež byl blíže popsán v sekci 6.1. Tyto zdokonalené verze obsahovaly ~2000-2500 fotografií typů **2x2** a **3x3** a ~3500-4000 fotografií **3x2_horizontal**, **3x2_vertical**, **pyraminx**. Po zdokonalení datových sad a opětovném natrénování neuronové sítě vykazovala mnohem lepší výsledky síť natrénovaná na první verzi datové sady s čistým pozadím, proto je tento model použitý i ve výsledné aplikaci.
Při trénování pomocí nástroje TensorFlow Lite Model Maker jsem využil 80% obrázků z datové sady pro trénování sítě, 10% pro validaci a 10% pro testování.

Pro práci s modelem využívám Google knihovnu ML kit. Tato knihovna byla blíže popsána v sekci 5.4. ML kit umožňuje využít klasifikační model buď pro klasifikaci obrázku, nebo pro detekci stěny v obrázku. Nejprve jsem zkusil model využít pro detekci stěny v obrázku. V tomto případě bych obdržel souřadnice, na nichž se v obrázku stěna nachází, a o jaký typ stěny se jedná. Výsledky ovšem nebyly uspokojivé, protože souřadnice nebyly vždy přesné a typ stěny také občas nesouhlasil se skutečným typem stěny na obrázku. Proto jsem se model rozhodl použít pro klasifikaci obrázku, kdy neuronová síť pouze informuje o tom, zdali se některý z podporovaných typů stěn na obrázku nachází a s jak velkou

pravděpodobností. Model vytvářím v instanci třídy `PuzzleDetector` jako instanci třídy `ImageLabeler`. `ImageLabeler` umožňuje nastavit počet vrácených objektů nalezených na obrázku a také minimální přípustnou hranici pravděpodobnosti. V mém případě jsem zvolil 1 objekt na obrázku, neboť mě zajímá vždy pouze jedna stěna, a hranici jsem nastavil na 50%, tudíž kdykoliv, kdy model vyhodnotí, že se na obrázku nachází jeden z natrénovaných typů stěny s více než 50% pravděpodobností, zavolá callback funkci registrovanou v naslouchači `addOnSuccessListener`, kde se obrázek dále zpracuje. Pokud by síť vyhodnotila, že se na obrázku nachází více než jeden typ stěny, vrátí tu s největší hodnotou pravděpodobnosti.

Detekce barevné konfigurace stěny

Detekci barev jsem implementoval ve třídě `PuzzleDetector`. Pro analýzu obrázků jsem využil `Image analysis` případ užití z knihovny CameraX zmiňovaný dříve v této sekci. Je realizován implementací rozhraní `ImageAnalysis.Analyzer`, které deklaruje metodu `analyze`. Této metodě jsou předávány obrázky z kamery k dalšímu zpracování a právě v těle této metody za použití funkcí z OpenCV knihovny (viz sekce 3.1) detekuji barevnou konfiguraci. Při registrování `Image analysis` případu užití nastavuji parametr vstupního obrázku na hodnotu 4:3, aby se shodoval s poměrem stran náhledu z `Preview` případu užití zmiňovaného výše v textu a bylo tak jednodušší ořezat pouze ohraničenou část obrázku (viz obrázek 6.7). Další parametr, který využívám při registraci `Image analysis`, je **STRATEGY_KEEP_ONLY_LATEST**, který určuje, že metoda `analyze` obdrží vždy nejaktualnější obrázek z kamery po dokončení analýzy předešlého obrázku.

Proces detekce začíná oříznutím obrázku pouze na velikost ohraničujícího obdélníku zobrazeného na obrázku 6.7. Metodě `analyze` je předáván obrázek z kamery v barevném modelu YUV420 a vzhledem k tomu, že `ImageLabeler`, který obrázek klasifikuje pomocí neuronové sítě, očekává obrázek v RGB barevném modelu, je potřeba tento převést do RGB. Knihovna OpenCV nabízí funkci `cvtColor`, která převádí mezi nejrůznějšími barevnými modely. Bohužel Android YUV420 formát není plně kompatibilní s knihovnou OpenCV, a tak není možné provést konverzi jednoduše pouhým zavoláním funkce `cvtColor`. Je potřeba jednotlivé YUV kanály do matice uložit ve správném pořadí, aby je funkce `cvtColor` převedla korektně. Převod jsem neprogramoval sám, ale využil jsem funkci `rgba`[3] uživatele **Egor Shitov**, kterou jsem našel na jeho github účtu[4].

Z ořízntého RGB obrázku vytvořím instanci třídy `InputImage`, která je vstupní třídou pro klasifikaci objektem `ImageLabeler`. Neuronová síť vyhodnotí obrázek zavoláním funkce `process` nad objektem `ImageLabeler` s `InputImage` objektem jako parametrem. Jak již bylo zmíněno dříve, v případě, že neuronová síť vyhodnotí, že se na obrázku nachází jeden z natrénovaných typů stěn, zavolá callback funkci definovanou v naslouchači `addOnSuccessListener`. Této callback funkci je předána informace o rozpoznaném typu stěny. S informací ve funkci nadále pracuji a snažím se v obrázku nadetekovat právě tento typ rozpoznané stěny. V případě neúspěchu se nic neděje a přecházím na analýzu dalšího nejaktuálnějšího obrázku z kamery.

V objektu `PuzzleDetector` si uchovávám buď pole přípustných typů stěn, které aplikace v momentálním skenu očekává, nebo v případě, že je již jasné, o jaký skenovaný hlavolam se jedná, si uchovávám pouze konkrétní typ momentálně očekávané stěny (sekvence skenování budou blíže popsány v následující podsekci). V callback funkci nejprve kontroluji, shoduje-li

[3] https://gist.github.com/ReDFoX43rus/29311f7dfaeb4565240fd1d45c4d2f77#file-javacamera2frame-java

[4] https://gist.github.com/ReDFoX43rus

se neuronovou sítí rozpoznaný typ stěny s jedním z očekávaných typů. Pokud se typ neshoduje, přecházím na analýzu dalšího obrázku. V případě, že se rozpoznaný typ stěny shoduje s očekávaným, zavolám metodu `detectFaceColourConfig`. Na začátku této metody převedu ořezaný obrázek z RGB do HSL barevného modelu funkcí `cvtColor` z knihovny OpenCV. HSL obrázek následně předám ke zpracování metodě `detectLittlePieces`, v níž pomocí prahování barev, které bylo popsáno v sekci 3.2, detekuji jednotlivé barvy stěny. Pro prahování barev využívám funkci `inRange` z knihovny OpenCV. Funkci předávám ořezaný HSL obrázek a dvě instance třídy `Scalar`, které v sobě uchovávají tři spodní a tři horní hodnoty prahu pro **H**, **S**, **L** barevné kanály. Hodnoty, které používám pro jednotlivé barvy, jsou v tabulce 6.1. Při prahování je občas červená barva detekována jako oranžová a naopak, protože jsou tyto barvy v HSL spektru až příliš blízko sebe a bylo náročné stanovit ideální práh, který by fungoval pro obě barvy za všech podmínek (za sníženého i zvýšeného jasu).

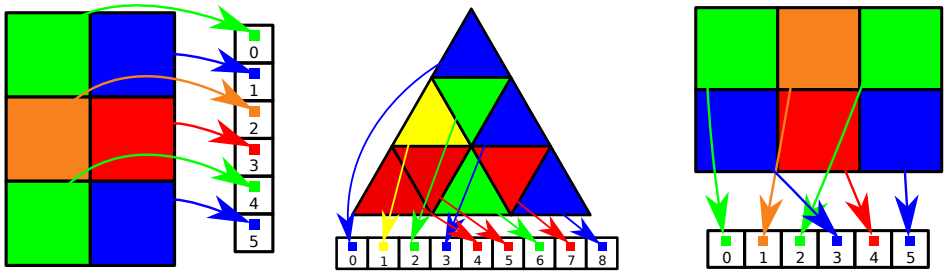
Barva	$\mathbf{H}_{min}$	$\mathbf{H}_{max}$	$\mathbf{S}_{min}$	$\mathbf{S}_{max}$	$\mathbf{L}_{min}$	$\mathbf{L}_{max}$
Červená	0	2	50	255	50	204
Červená	150	180	80	255	70	204
Oranžová	2	30	70	255	70	204
Žlutá	20	44	50	255	50	204
Zelená	45	80	50	255	50	204
Modrá	80	130	50	255	50	204
Bílá	0	180	0	255	170	255

Tabulka 6.1: Tabulka minimálních a maximálních hodnot H (hue) S (saturation) L (lightness) pro prahování jednotlivých barev. Ve standardním HSL modelu nabývají hodnoty S a L hodnot z intervalu <0;100> a hodnota H hodnoty z intervalu <0; 360>. OpenCV funkce `inRange` hodnoty S a L normalizuje hodnoty do 8 bitů, tudíž interval hodnot je pro tyto hodnoty <0;255> a pro hodnotu H používá pouze polovinu kruhu, tedy hodnoty z intervalu <0;180>.

Před samotným prahováním si do pole objektů typu `Scalar` načtu všechny možné barvy, které se mohou objevit na nadetekovaném typu stěny. Přes toto pole potom iteruji barvu po barvě a maskuji každou barvu zvlášť funkcí `inRange` z knihovny OpenCV. Po vymaskování barvy naleznu obrysy v masce OpenCV funkcí `findContours`. Při prahování může nastat, že se vymaskuje i šum v pozadí. Tento šum eliminuji tím, že z pole nalezených obrysů odstraňuji obrysy s malým obsahem a také odstraňuji kontury, jejichž počet vrcholů je rozdílný od 3 nebo 4 (v závislosti na typu očekávané stěny). Počet vrcholů zjišťuji aproximací polygonu OpenCV funkcí `approxPolyDP`, která do pole uloží body reprezentující polygon (počet bodů = počet vrcholů). Nakonec zavolám funkci `boundingRect` z knihovny OpenCV, která vrátí obdélník `Rect` ohraničující nalezený obrys. Pozici obdélníku společně s momentálně maskovanou barvou si uložím do pole a pokračuji na maskování další barvy.

Následně kontroluji, jestli počet nalezených polygonů (barev) odpovídá očekávanému počtu polygonů na stěně. Pokud ano, seřadím barvy podle jejich pozice na obrázku, tedy zleva doprava a následně shora dolů, a uživateli vykreslím na obrazovku nadetekované barvy. Vykreslování a vedení uživatele při skenování bude popsané v následující podsekci. Nadetekované barvy ukládám do pole v pořadí zleva doprava a shora dolů, tudíž barva horního levého rohu je na prvním místě v poli, na druhém místě v poli je barva napravo od tohoto rohu apod. Styl uložení barev do pole je vyobrazen na 6.10.

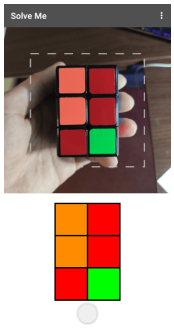
Detekce barev probíhá v reálném čase, tudíž uživatel vždy vidí, jestli aplikace správně rozpoznala barvy hlavolamu, a pokud ano, může stisknout tlačítko pro potvrzení a přejít ke skenování další stěny.



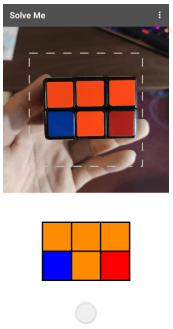
Obrázek 6.10: Vizualizace uložení naskenovaných barev do pole. Barvy jsou vždy uloženy zleva doprava a shora dolů.

Vizualizace kostky a vedení uživatele při skenování

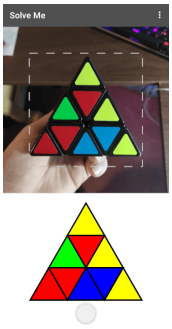
Aby měl uživatel ponětí, zdali aplikace detekuje správně barvy stěny, je mu v aplikaci vykreslena rozpoznaná stěna i s barevnou konfigurací. Vykreslené stěny i s barevnými konfiguracemi jsou na obrázcích 6.11, 6.12, 6.13. Při návrhu jsem se potýkal s problémem, jak v aplikaci znovu zkonstruovat po naskenování hlavolam, pokud bych uživateli dovolil skenovat stěny v jakémkoliv pořadí. Vyplynulo mi z toho, že takto znovu zkonstruovat hlavolam není možné a že musím uživatele při skenování částečně vést, aby stěny skenoval v určité sekvenci, kterou očekávám.



Obrázek 6.11: Vykreslení barevné konfigurace stěny 2x3 hlavolamu Slimtower.



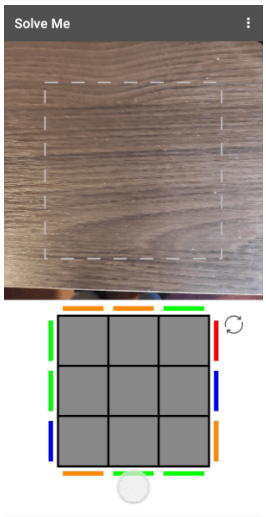
Obrázek 6.12: Vykreslení barevné konfigurace stěny 3x2 hlavolamu Rubikovo domino.



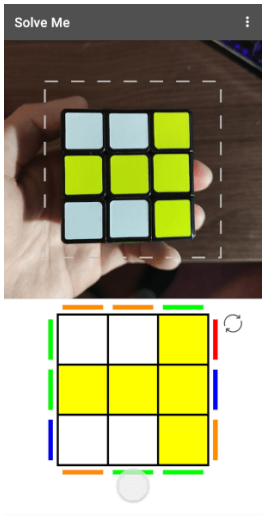
Obrázek 6.13: Vykreslení barevné konfigurace stěny hlavolamu Pyraminx.

Sekvence skenování je reprezentována objekty třídy `ScanSequence`. Objekt `PuzzleDetector` obsahuje objekt typu `ScanSequence`, který je po naskenování první stěny inicializován a uchovává si v sobě pole přípustných stěn, které mohou po naskenované stěně následovat. Tuto sekvenci realizuji jednoduše frontou, z níž je po každé správně naskenované stěně vyjmut první prvek, který je potom v objektu třídy **PuzzleDetector** uchováván jako momentálně očekávaný typ stěny, popřípadě momentálně očekávané typy stěn, jedná-li se o více než pouze jeden typ. To zaručuje, že uživatel nebude moci naskenovat například stěnu hlavolamu Pyraminx a následně na to stěnu 2×3 hlavolamu Slimtower. Aplikace na neočekávaný typ nijak nereaguje, toto lze vidět na obrázku 6.16. Je ovšem potřeba uživateli ukázat, jaký

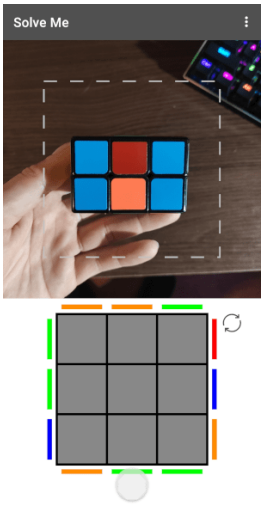
následující typ stěny Scanner očekává. Toto je zobrazeno jako stěna s šedou výplní v oblasti, kde se normálně vykresluje momentálně skenovaná stěna (viz obrázek 6.14). Navíc je potřeba uživateli ukázat, jak správně stěnu držet, aby ji náhodou nedržel naopak. To je provedeno tak, že uživateli kolem stěny ukazuji barvy předchozích naskenovaných stěn. Očekávaná stěna i s barvami v okolí je ukázána na obrázku 6.15.



Obrázek 6.14: V průběhu skenovacího procesu je uživatel aplikací veden, aby věděl, jakou stěnu naskenovat jako další. Tato další očekávaná stěna je vykreslena s šedou barvou.



Obrázek 6.15: Po umístění očekávané stěny se vyplní kostičky barvami. Pro větší přehlednost, jak kostku při skenování držet, jsou po bocích stěny vykresleny barvy sousedních stěn.



Obrázek 6.16: Aplikace reaguje pouze na očekávanou stěnu a na žádnou jinou.

Vykreslování hlavolamu implementuji ve třídě `FacePreview`, která dědí Android třídu `View`. FacePreview přepisuje metodu `onDraw`, která dovoluje malovat na plátno. Třída obsahuje dvě pole vykreslitelných objektů – pole trojúhelníků `Triangle` a pole obdélníků `Rectangle`. Třídy `Triangle` a `Rectangle` jsou mnou definované třídy, které využívám pro zapouzdření geometrického tvaru s barvou. `Triangle` sestává ze 3 bodů typu `Point` a barvy trojúhelníku a `Rectangle` se skládá z již existující Android třídy `Rect`, reprezentující obdélník, k níž je navíc přidaná barva. Do zmíněných polí jsou uloženy buď trojúhelníky, nebo obdélníky, které jsou následně vykresleny na plátno.

Proces vykreslování začíná tím, že instance třídy `FacePreview` přijme od objektu třídy `PuzzleDetector` typ stěny a pole barev stěny (způsob uložení barev v poli je na obrázku 6.10). V metodě `drawShape` zkontroluji, jestli je již na plátně něco vykresleného. Pokud je na plátně již něco vykresleného a typ i pole barev se shoduje s momentálně vykreslenou stěnou na plátně, není potřeba nic překreslovat. Liší-li se ovšem typy či barvy nebo je-li plátno prázdné, nadetekovanou stěnu je potřeba vykreslit na plátno. Prvním krokem při vykreslování je vyprázdnit obě pole. Následně zjistím, jaký typ stěny byl detekován a podle toho zavolám buď metodu `drawTriangles`, nebo `drawRectangles`. Metody naplňují buď pole trojúhelníku, nebo pole obdélníků objekty, které budou následně vykresleny na plátno. Obě metody jsou parametrizované. `DrawTriangles` přijímá počet vrstev trojúhelníku, jež má vykreslit, a indikátor, jestli vykreslit trojúhelník vzhůru nohama či normálně. Není tudíž

omezena pouze na vykreslení stěny Pyraminx, která obsahuje 3 vrstvy, ale může vykreslit i jinak velké trojúhelníkové stěny. Obdobně je parametrizovaná i metoda `drawRectangles`, která přijímá počet čtverců stěny na šířku a počet čtverců na výšku.

Jakmile je naplněno jedno z polí, zavolám metodu `postInvalidate`, která vyvolá překreslení plátna. Metody `drawRectanglesOnCanvas` nebo `drawTrianglesOnCanvas` volám při kreslení. V rámci těchto metod iteruji skrz naplněné pole. Při každé iteraci nejprve nakreslím černou barvou obrys tělesa a následně barevnou výplň tělesa.

Náhled na vykreslené typy stěn je na obrázcích 6.11, 6.12, 6.13.

Pro případ, že by uživatel omylem naskenoval špatnou stěnu, implementoval jsem do části s vizualizovanou stěnou resetovací tlačítko. Toto tlačítko se zobrazí po prvním naskenování stěny. Po stisknutí tlačítka je vyprázdněno pole se všemi doposud naskenovanými barevnými konfiguracemi stěn a Scanner je uveden do počátečního stavu. Toto umožňuje buď začít skenovat tentýž hlavolam znovu, nebo naskenovat jeden z ostatních podporovaných hlavolamů. Resetovací tlačítko lze vidět na obrázku 6.17 v horním pravém rohu.

K tomu, aby měl uživatel lepší ponětí, jak s hlavolamem otočit pro naskenování následující stěny, implementoval jsem v aplikaci animaci šipek, které ukazují, jak by měl uživatel hlavolam otočit. Tyto šipky jsou ukázány na obrázcích 6.17, 6.18 a 6.19.

Obrázek 6.17: Animovaná šipka zespodu nahoru, která ukazuje uživateli, že pro následující sken má otočit hlavolam na spodní stěnu.

Obrázek 6.18: Animovaná šipka shora dolů, která ukazuje uživateli, že pro následující sken má otočit hlavolam na horní stěnu.

Obrázek 6.19: Animovaná šipka zleva doprava, která ukazuje uživateli, že pro následující sken má otočit hlavolam na levou stěnu.

6.3 Solver

Hlavní úlohou **Solveru** je nalézt postup vedoucí k vyřešení naskenovaného hlavolamu. Tyto pohyby byly obecně popsány v sekci 2.2.3.

Solver očekává na vstupu správně naskenovaný hlavolam. Bohužel jsem z časových důvodů nestihl implementovat kontrolu naskenovaného hlavolamu, tudíž Solver funguje pouze pro korektně naskenovanou barevnou konfiguraci a neumí detekovat špatně naskenovanou konfiguraci.

Abstraktní třída Puzzle

Puzzle je abstraktní třída, kterou rozšiřují všechny hlavolamy – **Pyraminx**, **RubiksDomino**, **Slimtower**. Tato abstraktní třída definuje několik abstraktních metod, které musí jednotlivé hlavolamy implementovat. Nejdůležitější abstraktní metodou je `findSolution`. Jakmile proběhne korektní naskenování celého hlavolamu, vytvořím hlavolam statickou tovární metodou `createPuzzle`, která vrací objekt typu `Puzzle`, a nad tímto objektem zavo-

lám metodu `findSolution`. Úkolem této metody je zjistit řetězec pohybů vedoucích k vyřešení hlavolamu. Tento řetězec, nazývaný `solution`, je členská proměnná třídy `Puzzle`. Pro práci s řetězcem jsem implementoval v abstraktní třídě funkci `doMoves`, která přijímá jako parametr řetězec (pohyb/rotaci) a konkatenuje ho k již existujícím pohybům v řetězci `solution`. Řetězec je po nalezení řešení zobrazen uživateli společně s modelem hlavolamu a animací. Zobrazení hlavolamu bude popsáno v následující sekci 6.4. Implementaci `findSolution` popíšu později u popisu jednotlivých hlavolamů, neboť je až na konkrétním hlavolamu, aby metodu implementoval. Poslední důležitou abstraktní metodou je `decodeMoveFromString`, která provede pohyb, jež přijme jako parametr. Provedením pohybu se rozumí zaměnit barvy v barevné konfiguraci hlavolamu tak, aby konfigurace odpovídala stavu po pohybu. Tuto metodu také musí implementovat každá třída, která dědí třídu `Puzzle`, protože změna barev při konkrétním pohybu je závislá na typu hlavolamu.

Postup hledání řešení jednotlivých hlavolamů

Při hledání řešení jsem u všech hlavolamů postupoval tak, jak bylo popsáno v podsekcích **Postup řešení** jednotlivých hlavolamů **Rubikovo domino** (2.3), **Slimtower** (2.4) a **Pyraminx** (2.5).

V kostruktorech jednotlivých tříd hlavolamů naplňuji barevnou konfiguraci do pole barev. Vytvářím si i kopii této konfigurace, protože v průběhu hledání řešení aktualizuji pole s barevnou konfigurací a potřebuji si uchovat i počáteční stav konfigurace, abych mohl barvy po nalezení řešení uvést zpět do stavu, v jakém je uživatel naskenoval.

Aktualizace barev probíhá v metodě `decodeMoveFromString`. Tato metoda obsahuje velký `switch`, který na základě předaného parametru volá metodu reprezentující konkrétní pohyb. Definice metod reprezentující pohyby jsem se snažil seskupit do rozhraní, která následně konkrétní hlavolamy implementují. Rozhraní jsem vytvořil čtyři:

1. **IBasicMoves** – Metody základních pohybů, které sdílí všechny hlavolamy, toto rozhraní implementuje již samotná abstraktní třída `Puzzle`. Jsou jimi pohyby **L, R, F, B, U a D a jejich inverzní obdoby**.

2. **IMiddleMoves** – Metody pohybů středních vrstev. Těmito pohyby jsou **M, E, S a jejich inverzní obdoby**. Toto rozhraní implementuje pouze hlavolam **Slimtower**, neboť využívá pohyby střední vrstvy **M** a **M'**. Našly by se ale i jiné hlavolamy, které by toto rozhraní v budoucnu mohly implementovat. Mohla by to být například Rubikova kostka ($3 \times 3 \times 3$), Rubikova pomsta ($4 \times 4 \times 4$), Profesorova kostka ($5 \times 5 \times 5$) apod.

3. **ITipsMoves** – Metody pohybů vrcholových kostiček. Myslí se jimi pohyby **l, r, f, u a jejich inverzní obdoby**. Toto rozhraní implementuje hlavolam **Pyraminx**, protože jako jediný z implementovaných hlavolamů má pohyblivé vrcholy. Jinými hlavolamy implementující toto rozhraní by mohly být obdoby hlavolamu Pyraminx jako například Master Pyraminx (4×4).

4. **IViewMoves** – Metody otáčení celých hlavolamů. Existují pohyby **X, Y a Z**. Toto rozhraní neimplementuje žádný z hlavolamů, je pouze připraveno pro další rozšíření aplikace.

Je potom úkolem jednotlivých tříd implementovat metody rozhraní tak, aby prováděly pohyb, který mají.

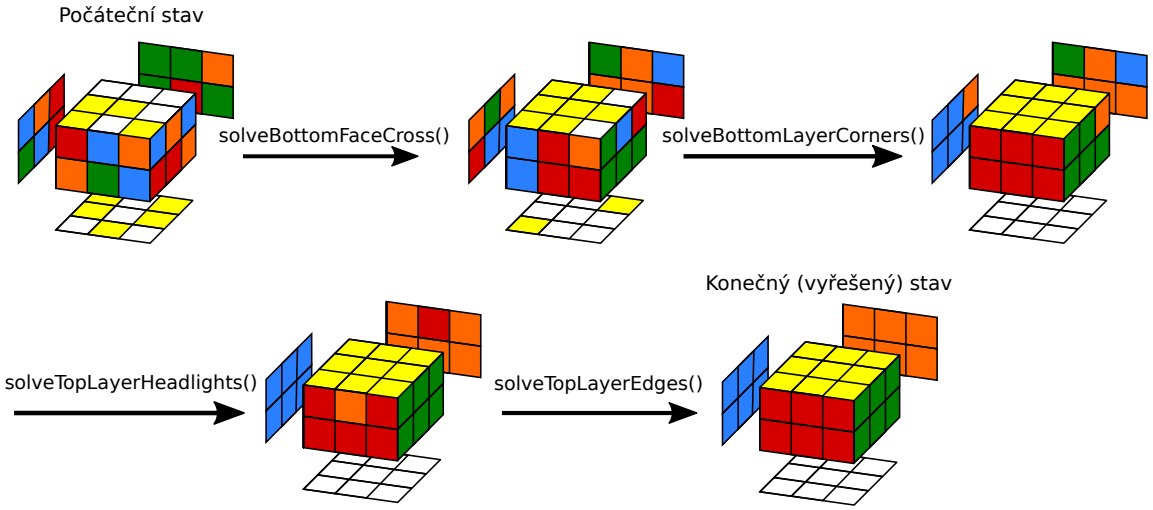
Pro jednoduchost popíšu implementaci metody `findSolution` obrázky a stručným popisem každé metody volané ve `findSolution`. Ve všech případech tříd hlavolamů sestává metoda `findSolution` ze sekvence volání metod, které se starají o to, aby po dokončení metody byl hlavolam v určitém stavu blíž k vyřešení (například aby byly vyřešeny rohové kostičky hlavolamu nebo aby byla vyřešena spodní vrstva hlavolamu). Při tomto procesu se přidávají do řetězce `solution` jednotlivé pohyby vedoucí do dalšího stavu.

Rubikovo domino

Postup přechodů mezi jednotlivými stavy tak, jak byl popsán v podsekci **Postup řešení** hlavolamu **Rubikovo domino** (2.3), je vizualizován na obrázku 6.20.

Stručný popis jednotlivých metod z obrázku 6.20:

- **solveBottomFaceCross()** – Zjistí barvu spodní vrstvy (bílá nebo žlutá) a poskládá kříž této barvy na spodní vrstvě tak, aby jednotlivé barvy byly vůči sobě ve správném rozestavění.
- **solveBottomLayerCorners()** – Vloží všechny rohy spodní vrstvy na jejich správné místo tak, aby se shodovaly s barvami spodních hranových kostiček, čímž je vyřešena i celá spodní vrstva.
- **solveTopLayerHeadlights()** – Vyřeší rohy horní vrstvy použitím algoritmu Algo. 2.3.2.
- **solveTopLayerEdges()** – Prohodí mezi sebou hrany horní vrstvy, čímž vyřeší hlavolam.



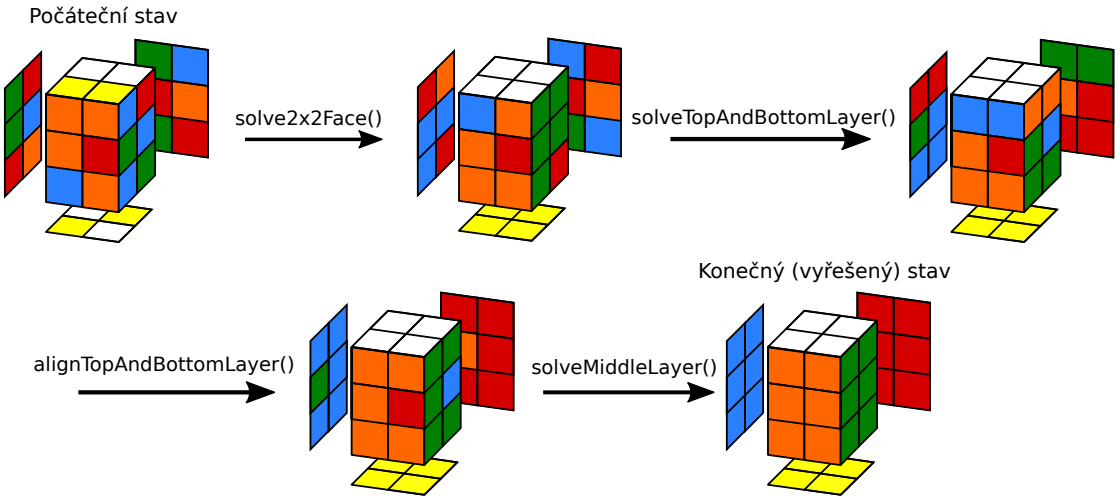
Obrázek 6.20: Přechod mezi jednotlivými stavy hlavolamu **Rubikovo domino**. Názvy nad šipkami jsou názvy metod, které jsou volány ve funkci `findSolution`. Každá metoda přidá sekvenci pohybů do řetězce `solution`, který ve výsledku tvoří sekvenci pohybů vedoucí k vyřešení hlavolamu.

Slimtower

Postup přechodů mezi jednotlivými stavy tak, jak byl popsán v podsekci **Postup řešení** hlavolamu **Slimtower** (2.4), je vizualizován na obrázku 6.21.

Stručný popis jednotlivých metod z obrázku 6.21:

- **solver2x2Face()** – Vyřeší barvu horní a spodní 2x2 stěny.
- **solveTopAndBottomLayer()** – Vyřeší barvu horní a spodní vrstvy tak, že jsou dvě sousední kostičky horní/spodní vrstvy na jedné stěně totožné.
- **alignTopAndBottomLayer()** – Zarovná barvu horní vrstvy s barvou spodní vrstvy tak, aby se shodovaly na všech stěnách.
- **solveMiddleLayer()** – Vyřeší barvu střední vrstvy.



Obrázek 6.21: Přechod mezi jednotlivými stavy hlavolamu **Slimtower**. Názvy nad šipkami jsou názvy metod, které jsou volány ve funkci `findSolution`. Každá metoda přidá sekvenci pohybů do řetězce `solution`, který ve výsledku tvoří sekvenci pohybů vedoucí k vyřešení hlavolamu.

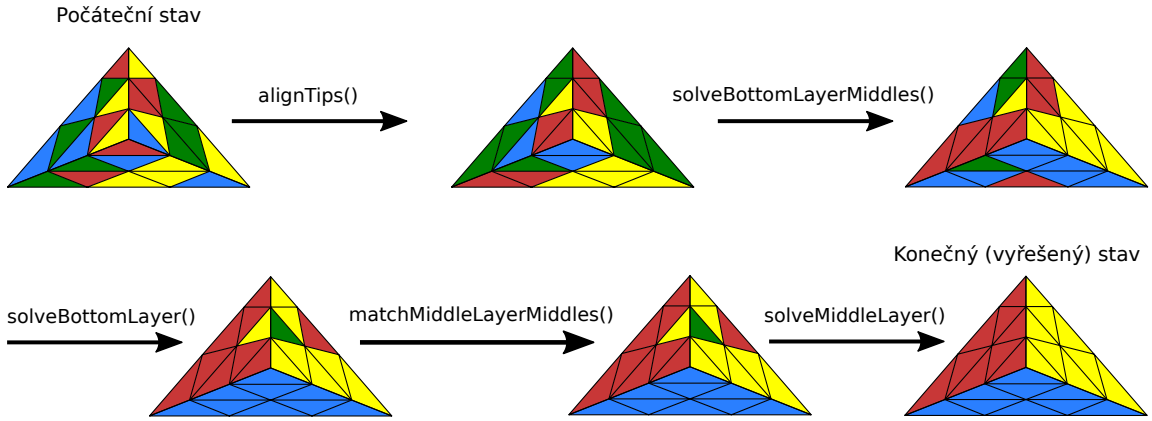
Pyraminx

Postup přechodů mezi jednotlivými stavy tak, jak byl popsán v podsekci **postup řešení** hlavolamu **Pyraminx** (2.5), je vizualizován na obrázku 6.22.

Stručný popis jednotlivých metod z obrázku 6.22:

- **alignTips()** – Zarovná barvy rohových kostiček s barvami středových kostiček
- **solveBottomLayerMiddles()** – Zjistí barvu spodní vrstvy tak, že z pole všech barev hlavolamu vyřadí barvy středových kostiček ve střední vrstvě. Zbylá barva potom určuje barvu spodní vrstvy. Následně tuto barvu hledá na středových kostičkách spodní vrstvy a provádí odpovídající rotace, aby se tyto dostaly na spodní stěnu.

- **solveBottomLayer()** – Vyřeší spodní vrstvu vložením hranových kostiček na správná místa.
- **matchMiddleLayerMiddles()** – Zarovná barvy středových kostiček ve střední vrstvě se spodní vrstvou.
- **solveMiddleLayer()** – Prohodí mezi sebou hranové kostičky střední vrstvy, čímž vyřeší hlavolam.



Obrázek 6.22: Přechod mezi jednotlivými stavy hlavolamu **Pyraminx**. Názvy nad šipkami jsou názvy metod, které jsou volány ve funkci `findSolution`. Každá metoda přidá sekvenci pohybů do řetězce `solution`, který ve výsledku tvoří sekvenci pohybů vedoucí k vyřešení hlavolamu.

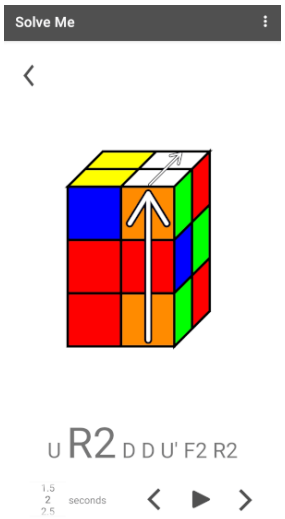
Po nalezení řešení se obnoví barevná konfigurace hlavolamu na původní naskenovanou a aplikace přejde do modulu **Solution visualizer**, v němž je řešení naskenovaného hlavolamu prezentováno uživateli.

6.4 Solution visualizer

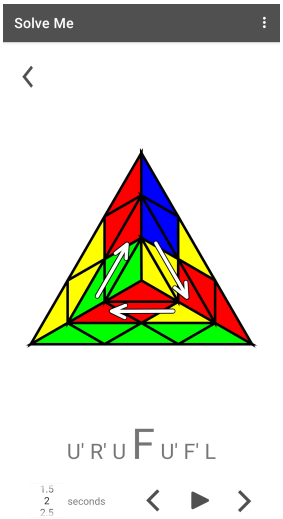
Posledním modulem aplikace je **Solution visualizer**. Tento modul má za úkol prezentovat uživateli řešení (sekvenci pohybů) v textové i vizuální podobě s postupným krokováním až do vyřešeného stavu.

Do modulu jsem naprogramoval šipky, kterými může uživatel krokovat jednotlivé pohyby buď dopředu, či dozadu. Mimo jiné jsem také implementoval možnost automatického přehrávání krokování, neboť se domnívám, že je pohodlnější se dívat na obrazovku a při tom se soustředit pouze na skládání hlavolamu, než každý krok ručně přepínat. Toto automatické přehrávání lze kdykoliv pozastavit a je možné si vybrat i dobu přepnutí na další krok při přehrávání. Možnost výběru jsem nastavil na hodnoty 1-4 sekundy s krokem 0,5 sekundy. Implicitní hodnotu jsem nastavil na 2 sekundy.

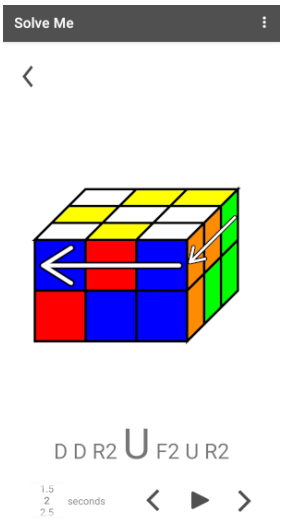
V každém kroku se přepne ve spodním panelu písmeno na aktuálně prováděný pohyb. Toto písmeno má dvakrát větší font než okolní písmena, aby bylo jasné, který pohyb se provádí. Společně s písmenem se změní i barvy na hlavolamu a zobrazí se šipky na stěně, jež má být otočena.



Obrázek 6.23: Slimtower s vyznačeným pohybem pravé stěny **R2**.



Obrázek 6.24: Pyraminx s vyznačeným pohybem předního sub-tetraedru **F**.



Obrázek 6.25: Rubikovo domino s vyznačeným pohybem horní vrstvy **U**.

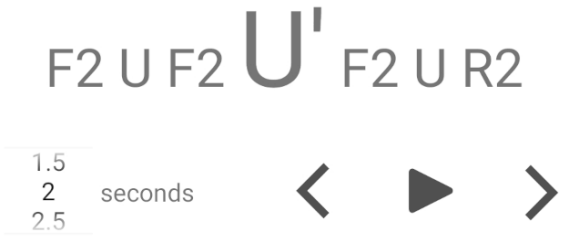
Model hlavolamu

Pro vykreslení hlavolamu jsem zvolil podobný přístup jako při vykreslování stěny v modulu **Scanneru** (6.2). Toto vykreslování jsem implementoval ve třídě `PuzzlePreview`. Třída obsahuje, podobně jako třída `FacePreview` pro vykreslení náhledu stěny, pole trojúhelníků `Triangle`, pole obdélníků `Rectangle` a navíc ještě pole obdélníků `Rectangle4Points`. Pole objektů `Triangle` a `Rectangle` fungují stejně jako ve **Scanneru**. Pole naplňuji trojúhelníky/obdelníky s příslušnou barvou, které mají být vykresleny na plátno. Ostatně pro toto naplnění využívám i stejné metody `drawRectangles` a `drawTriangles`, které pole plní geometrickými tvary. Pole obdélníků `Rectangle` využívám pro vykreslení přední stěny hlavolamů **Rubikovo domino** a **Slimtower** a pole trojúhelníků `Triangle` využívám pro vykreslení všech stěn hlavolamu **Pyraminx**. Navíc jsem přidal pole obdélníků typu `Rectangle4Points` pro vykreslení kosodélníků, neboli boční a horní stěny kvádrových hlavolamů. Využívám metody `drawRectanglesTopFace` a `drawRectanglesRightFace` pro naplnění pole. Obě metody přijímají jako parametr velikost stěny, která má být vykreslena. V případě vykreslování hlavolamu **Slimtower** předávám metodě `drawRectanglesTopFace` velikost stěny 2×2 a metodě `drawRectanglesRightFace` velikost stěny 2×3. Obě tyto funkce obdrží počáteční bod, z něhož mají začít kreslit. V případě horní stěny je tento bod horní levý roh přední stěny a v případě vykreslení pravé stěny je to horní pravý roh přední stěny. Následně si v metodách vypočítám posun po osách X a Y, abych mohl nakreslit obdélník zkoseně a naplním pole `Rectangle4Points` body pro pozdější vykreslení. Hodnota posunu po osách X a Y je třetina velikosti malého čtverce přední stěny. Vzhledem k tomu, že jsou obě funkce parametrizované, je možné vykreslit celou řadu hlavolamů tvaru kvádru.

Přehrávač a krokování

Důležitou částí při vizualizaci je možnost přepínat mezi jednotlivými pohyby. Přepínání, jak již bylo zmíněno dříve v této sekci, může být buď dopředu, nebo dozadu. Uživatel má možnost zapnout automatické přehrávání kroků a tím nechat aplikaci přepínat pohyby až

do vyřešeného stavu. Celé toto přehrávání a krokování řídí přepínání barev na vykresleném hlavolamu, přidávání šipek na hlavolam a zvýrazňování momentálně prováděného kroku v textu.



Obrázek 6.26: Detail spodní části vizualizace řešení. Text v horní částí ukazuje následující a předešlé pohyby. Zvýrazněné písmeno je momentálně prováděný pohyb. V pravé spodní částí jsou tlačítka na krokování vpřed či vzad a tlačítko na spuštění automatického krokování vpřed. V levé spodní části je možnost změnit dobu provedení kroku.

Z výsledného řetězce obsahujícího kroky řešení zobrazuji v aplikaci vždy maximálně 7 kroků. K této části řetězce přistupuji pomocí klouzavého okna. Klouzavá okna využívám dvě – jedno na zvýraznění písmene a druhé na část zobrazeného textu. Obě okna začínají na začátku textu (viz obrázek 6.27). Okna umí klouzat vpřed i vzad. Pro jednodušší vysvětlení momentálně uvažujme pouze okna, která se pohybují vpřed. Okno zvýrazňující písmeno se pohybuje maximálně do poloviny zvýrazněného textu (viz obrázek 6.28), potom se začne pohybovat druhé okno zobrazující část řešení. Jakmile druhé okno narazí na konec řetězce, rozpohybuje se opět první okno a pohybuje se až na konec řetězce (viz obrázek 6.29). Stejný postup je aplikován i při pohybu v opačném směru (vzad).

Obrázek 6.27: Počáteční stav řetězce. Klouzavé okno řetězce i zvýrazněného písmene jsou na úplném začátku řetězce.

Obrázek 6.28: Postupem času se klouzavá okna pohybují. Okno zvýrazněného písmene se pohybuje maximálně do středu řetězce, následně se pohybuje celé okno řetězce.

Obrázek 6.29: Narazí-li okno řetězce na konec řetězce, pohybuje se opět okno zvýrazněného písmene až po konec řetězce.

Klouzavá okna jsou řízena buď přehrávačem nebo šipkami pro pohyb vpřed/vzad. Přehrávač jsem implementoval jako objekt třídy `CountDownTimer` z Android API. Konstruktor této třídy přijímá jako parametry celkový čas v milisekundách a délku kroku v milisekundách, po jehož uplynutí je cyklicky volána funkce `onTick`. Právě v této funkci provádím posun klouzavých oken vpřed a zároveň měním barvu vykresleného hlavolamu a šipky na hlavolamu. Čas časovače počítám jako rozdíl celkového počtu kroků a pozice momentálního kroku. Toto číslo vynásobím 1000 a počtem vteřin, které uživatel na obrazovce vybral, a spustím časovač. Jakákoliv akce od uživatele časovač zastaví. Ať už se jedná o manuální

přepnutí kroku, zastavení přehrávače či změnu vybraného času. Následně při opětovném spuštění přehrávače vytvářím novou instanci třídy `CountDownTimer`.

Při provádění konkrétního kroku volám metodu `decodeMoveFromString` objektu `Puzzle`, kterou jsem zmiňoval v předešlé sekci 6.3. Metodě předám momentálně prováděný krok, ta tento krok provede, změní barevnou konfiguraci hlavolamu a následně je tato barevná konfigurace aktualizována na vykresleném hlavolamu.

Šipky

V průběhu vizualizace se na hlavolamu zobrazují šipky, které ukazují směr, v němž má být stěna otočena (viz obrázek 6.23). Šipky jsem vytvořil v open-source vektorovém editoru **Inkscape**. Vytvořil jsem tři šipky pro **Slimtower** a **Rubikovo domino** – horizontální, vertikální a šipku pod úhlem 45° – a čtyři sady šipek pro **Pyraminx**. Sadou šipek se rozumí několik šipek poskládaných do určitého vzoru (např. jako na obrázku 6.24). Tyto šipky nastavuji jako pozadí Android elementu `ImageView`. Dále v kódu nastavuji velikost `ImageView` a jeho odsazení zleva a shora tak, aby se nacházel na správné stěně.

Kód ovládající šipky jsem implementoval jako součást třídy `PuzzlePreview`. Vytvořil jsem několik metod pro manipulaci s šipkami, nejdůležitější tři jsou – `setArrowRight`, `setArrowFront` a `setArrowTop`. Obecně před voláním těchto metod nastavím správné šipky jako pozadí `ImageView` a ve zmíněných metodách nastavím jejich velikost a odsazení. Velikost a odsazení u hlavolamů tvaru kvádru nastavuji tak, že metodám předávám souřadnice řádku nebo sloupce, do nějž má být šipka vložena. V metodě následně vypočítám šířku a výšku šipky, nastavím odsazení shora a zleva a tyto parametry aplikuji na `ImageView` s již nastavenou šipkou jako pozadí. V případě hlavolamu **Pyraminx** přistupuji k sadě šipek jako ke kružnici. Vypočítám si průměr kružnice, který určuje výšku i šířku `ImageView` a odsazení shora a zleva počítám jako souřadnici středu poníženou o poloměr. Tyto parametry taktéž aplikuji na `ImageView` s nastavenou sadou šipek, aby se šipky nacházely na místě, kde potřebuji.

Kapitola 7

Testování

Testování je nedílnou součástí vývoje jakéhokoliv software. Testování může odhalit nedostatky systému, poskytnout pohled nestranného účastníka a zároveň poskytnout zpětnou vazbu a návrhy na vylepšení od koncových uživatelů.

Tato kapitola popisuje, jak byla aplikace testována v průběhu vývoje a po dokončení implementace na koncových uživatelích. Na základě zpětné vazby od uživatelů navrhuji na konci kapitoly vylepšení aplikace.

7.1 Aplikační testování

V průběhu implementace aplikace jsem důkladně testoval každou z jejích částí. Důraz byl kladen na správnou funkčnost Scanneru, neboť ten je hlavním prostředkem aplikace pro interakci s uživatelem a je nutné, aby fungoval korektně.

Pro testování jsem využil jednak mé vlastní hlavolamy, které jsem využíval v průběhu celého procesu implementace, jednak jsem využil hlavolamy mého kamaráda, který mi je zapůjčil. Jeho hlavolamy se od mých liší hlavně sytostí barev, protože jsou již trochu starší. Při testování jsem tedy využil dva rozdílné hlavolamy Pyraminx, dvě Rubikova domina a dva hlavolamy Slimtower. Každý z těchto hlavolamů jsem zkusil 5 krát naskenovat, pokaždé s rozdílnou barevnou konfigurací, a sledoval jsem, jestli se zobrazují správně barvy na obrazovce. Barvy se zobrazovaly v pořádku až na červenou a oranžovou. V tmavším prostředí byla oranžová občas detekována jako červená. Jako řešení jsem se pokusil změnit HSL hodnoty oranžové a červené tak, aby aplikace co nejpřesněji rozeznávala tyto dvě barvy v jakémkoliv prostředí, ale občas se stane, že i tak jsou ve Scanneru barvy zaměněny a je potřeba hlavolam přesunout na trochu lépe osvětlené místo.

Scanner jsem také po dokončení implementace vyzkoušel v různých světelných podmínkách. K tomu jsem využil opět všechny hlavolamy, které jsem měl k dispozici (2x Pyraminx, 2x Rubikovo domino, 2x Slimtower). Každý hlavolam jsem zkusil naskenovat se stejnou konfigurací na místech s různým osvětlením. Níže jsou světelné podmínky, v nichž jsem konfiguraci skenoval, a výsledek skenování:

1. Dobře osvětlená místnost s přirozeným světlem – V takových podmínkách fungovalo skenování bez problému.

2. Dobře osvětlená místnost s umělým světlem – Problém s červenou barvou. Žluté umělé světlo celkově zesvětlilo červenou barvu a tu Scanner detekoval jako oranžovou.

3. Špatně osvětlená místnost (jediný zdroj světla byla lampa v rohu místnosti) – Z blízké vzdálenosti od lampy Scanner reagoval a zobrazoval barvy jakž takž v pořádku. Z větší vzdálenosti barvy nebyly rozeznány.

4. Noční místnost, jejímž jediným zdroje světla byla pouliční lampa – Za takových podmínek nebyl schopen Scanner barvy rozpoznat.

5. Skenování pod lampou s bílým světlem – V případě, že se na hlavolamu objevil odlesk z lampy, barevná konfigurace stěny nebyla rozpoznána (prahování barev je citlivé na odlesky). Když jsem hlavolam posunul, abych odstranil odlesk, Scanner stěnu detekoval v pořádku.

Obecně Scanner detekuje barvy korektně při dostatečném externím světle. Je citlivý na odlesky a žluté světlo (pouze v případě červené a oranžové barvy). Jako řešení bych navrhoval nepoužívat prahování, ale zobrazit uživateli v náhledu z kamery síť očekávané stěny. Barvy jednotlivých kostiček by se potom počítaly jako průměrná barva buňky v síti. K zobrazení sítě by ovšem bylo ideální, kdyby si uživatel předem vybral, jaký hlavolam bude skenovat. Toto je bohužel v rozporu se zadáním a z toho důvodu to takto není v práci provedeno.

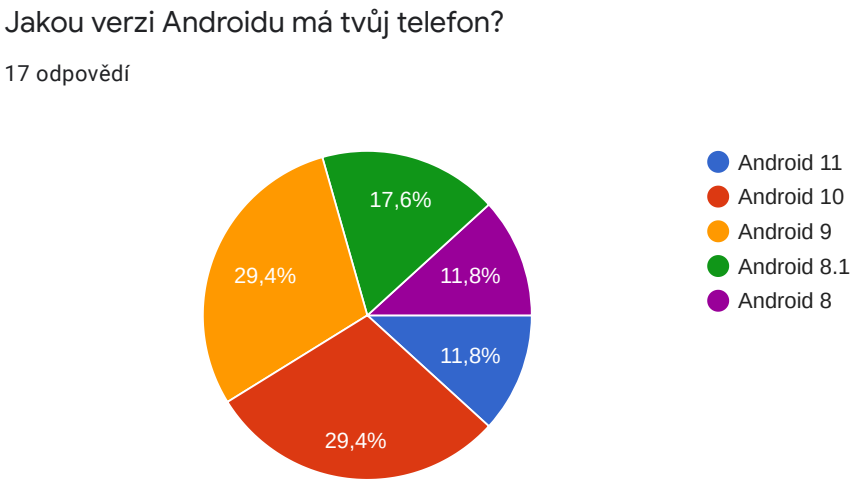
7.2 Uživatelské testování

Aplikaci jsem po dokončení otestoval na 17 uživatelích z mého okolí. Formulář použitý pro testování je na paměťovém médiu, jež se nachází na poslední straně výtisku práce. Na tomto paměťovém médiu je k nalezení v adresáři `dotaznik`.

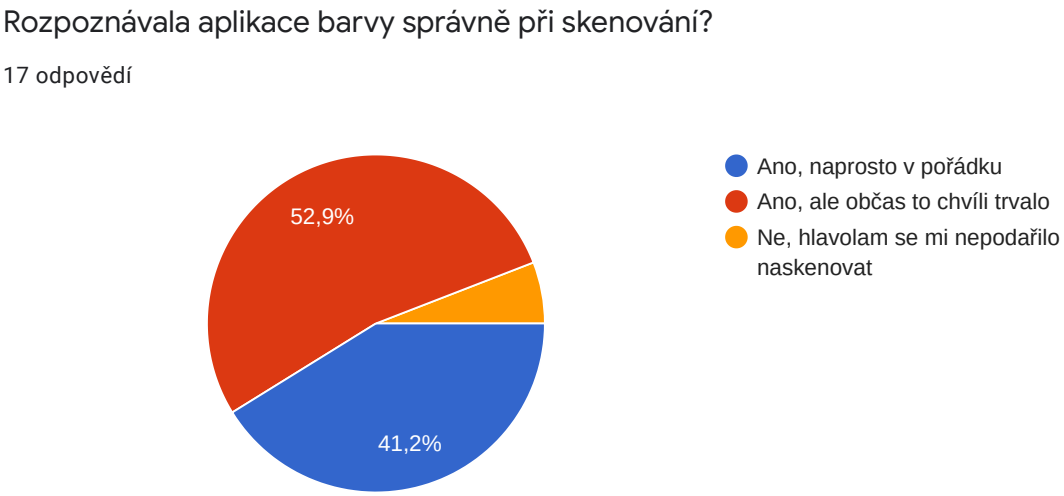
Testování jsem provedl dvěma způsoby. Při práci s aplikací jsem pozoroval uživatele, jak skenují hlavolamy, a všímal si, jestli s aplikací pracují tak, jak bylo zamýšleno. Dále jsem je požádal o vyplnění dotazníku jako způsob zpětné vazby a popřípadě i návrhů na vylepšení.

Při skenování jsem uživatele upozornil na skutečnost, aby si dávali pozor na správně vykreslené barvy v aplikaci, neboť, jak jsem již zmiňoval dříve, aplikace neumí detekovat špatně naskenovanou konfiguraci. Sledováním jsem vypozoroval, že při skenování 17.6% uživatelů otáčelo hlavolam špatným směrem, protože se domnívali, že animovaná šipka ukazuje na stěnu, kterou mají dále naskenovat. Po vysvětlení ovšem uznali, že šipka reprezentující způsob otočení hlavolamu dává smysl, jen je to při prvním skenování nenapadlo. Všem respondentům bylo jasné, jak potvrdit naskenování hlavolamu a i funkčnost tlačítka pro reset. Pár respondentů mylně pochopilo úvodní text, v němž ukazuji, jakou stěnou začít skenování. Tito uživatelé mysleli, že průvodní text slouží pro výběr hlavolamu, který skenují, a ne pouze pro informaci, jak začít skenovat. Jednomu uživateli se nepodařilo hlavolam vůbec naskenovat. Toto ovšem bylo způsobeno tím, že hlavolam skenoval v tmavém prostředí a po přesunutí do lépe osvětlené místnosti Scanner detekoval barvy správně.

Všem uživatelům, jež se podíleli na testování, fungovala aplikace na jejich mobilním zařízení bez problémů. Zastoupení verzí Android mobilních zařízení je zobrazeno na obrázku 7.1. Všichni uživatelé byli schopni podle aplikace hlavolam složit do vyřešeného stavu. Některým ovšem přišla doba kroku příliš krátká, proto jsem přidal ještě další dvě hodnoty (4.5 a 5 vteřin). Uživatelé v tomto případě ocenili možnost kroku zpět, protože pokud nestihli udělat pohyb, mohli v aplikaci skládání zastavit a vrátit se do konfigurace, ve které se jejich hlavolam reálně nacházel. Několik uživatelů navrhlo, že by mohlo být v rohu obrazovky vysvětleno, co znamenají jednotlivá písmena, která se zobrazují ve spodní části obrazovky. Pár uživatelů navrhlo, že by v průběhu skenování i skládání mohl být zvukový doprovod.



Obrázek 7.1: Četnost Android verzí mobilních zařízení uživatelů, jež testovali aplikaci.



Obrázek 7.2: Graf výsledků dotazu o správném rozpoznání skenovaných barev.

7.3 Shrnutí a návrhy na vylepšení

Celkově uživatelé hodnotili zkušenost s aplikací pozitivně, hlavně část vizualizace hlavolamu. Scanner by vyžadoval úpravu vedení uživatele pro otáčení hlavolamu, protože některým uživatelům nebylo z animované šipky jasné, jak hlavolam otočit. Ve fázi vizualizace by v horní části obrazovky mohl být vysvětlen význam jednotlivých písmen a krokování by mohlo být doprovázeno zvukovými efekty.

Kapitola 8

Závěr

Cílem práce bylo vytvořit Android aplikaci, která umožní uživatelům naskenovat jeden z podporovaných hlavolamů a následně předvede postup vedoucí k jeho vyřešení. Aplikace vede uživatele procesem skenování barevné konfigurace hlavolamu. Po nasnímání všech stěn nalezne aplikace sekvenci pohybů vedoucích k vyřešení naskenovaného hlavolamu a tuto sekvenci společně s modelem hlavolamu zobrazí uživateli. Po zobrazení hlavolamu má uživatel možnost spustit automatické přehrávání, při němž aplikace zobrazuje jednotlivé pohyby, které mají být provedeny.

Před začátkem implementace bylo k dosažení cíle nutné nastudovat potřebnou teorii. Nejprve bylo potřeba seznámit se s algoritmy pro vyřešení jednotlivých hlavolamů a označení rotací využívaných při zápisu těchto algoritmů. Potom bylo nutné nastudovat, jak z obrazu získat barevnou konfiguraci stěny. K tomu byla zvolena metoda prahování barev. Další z problémů, které bylo potřeba vyřešit, byla klasifikace stěny v obraze. Pro klasifikaci stěny je v práci využitý model konvoluční neuronové sítě přetrénovaný pomocí nástroje TensorFlow na vlastní datové sadě. Jelikož je cílovou platformou operační systém Android, bylo potřeba se obeznámit se způsoby vývoje aplikací na této platformě. Při implementaci byla využita Android knihovna CameraX pro obsluhu kamery, knihovna počítačového vidění OpenCV, jejíž funkce jsou použité pro extrakci barevné konfigurace z obrazu, a knihovna ML kit pro práci s modelem konvoluční neuronové sítě.

V závěru implementace byla aplikace otestována na 17 uživatelích. Hodnocení aplikace bylo především pozitivní a všichni testovaní uživatelé s výjimkou jednoho, jemuž se nepodařilo hlavolam naskenovat, byli schopni hlavolam podle aplikace složit. Problémy se vyskytovaly ve fázi skenování hlavolamu, při němž aplikace občas zaměňovala červenou a oranžovou barvu. Další problém při skenování nastával za špatných světelných podmínek, kdy aplikace nebyla schopna rozeznat jednotlivé barvy.

V budoucnosti by mohla být aplikace rozšířena i o další hlavolamy. Pro vykreslování kvádrových hlavolamů (Tower hlavolamy, Rubikova kostka, Rubikova pomsta apod.) a hlavolamů tvaru jehlanu v aplikaci již existují funkce. Proto při přidávání dalších hlavolamů by bylo potřebné pouze přetrénovat model neuronové sítě, aby podporoval stěny přidávaného hlavolamu, a naprogramovat algoritmy pro nalezení sekvence pohybů vedoucích k vyřešení. Dalším vylepšením by mohla být možnost manuálně zadávat barvy hlavolamu. Mimo tato vylepšení by aplikace vyžadovala kontrolu, zdali je naskenovaná barevná konfigurace možná. Tohoto by mohlo být dosaženo kontrolou počtu naskenovaných barev nebo kontrolou jednotlivých kostiček hlavolamu.

Literatura

[1] Alake, R. *Understand Local Receptive Fields In Convolutional Neural Networks* [online]. Medium, červen 2020 [cit. 2020-03-11]. Dostupné z: https://towardsdatascience.com/understand-local-receptive-fields-in-convolutional-neural-networks-f26d700be16c.

[2] Albawi, S., Mohammed, T. A. a Al-Zawi, S. Understanding of a convolutional neural network. In: *2017 International Conference on Engineering and Technology (ICET)*. 2017, s. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[3] *Albumentations documentation* [online]. 2021 [cit. 2020-04-15]. Dostupné z: https://albumentations.ai/docs/getting_started/image_augmentation/.

[4] *Activitiy Lifecycle* [online]. 2021 [cit. 2020-04-17]. Dostupné z: https://developer.android.com/reference/android/app/Activity#ActivityLifecycle.

[5] *Android Jetpack Library* [online]. 2021 [cit. 2020-04-18]. Dostupné z: https://developer.android.com/jetpack.

[6] *Android Platform Architecture* [online]. 2021 [cit. 2020-04-17]. Dostupné z: https://developer.android.com/guide/platform/.

[7] *App Manifest Overview* [online]. 2021 [cit. 2020-04-17]. Dostupné z: https://developer.android.com/guide/topics/manifest/manifest-intro.

[8] *CameraX Overview* [online]. 2021 [cit. 2020-04-18]. Dostupné z: https://developer.android.com/training/camerax.

[9] *SDK Platform release notes* [online]. 2021 [cit. 2020-04-17]. Dostupné z: https://developer.android.com/studio/releases/platforms.

[10] Callaham, J. *The history of Android: The evolution of the biggest mobile OS in the world* [online]. Android Authority, listopad 2020 [cit. 2020-04-17]. Dostupné z: https://www.androidauthority.com/history-android-os-name-789433/.

[11] Gardner, M. *Introduction to Uwe Meffert* [online]. 2011 [cit. 2020-12-18]. Dostupné z: http://www.mefferts.com/page.php?lang=en&theme=about_uwe.

[12] Gary Bradski, A. K. *Learning OpenCV*. 1. O'Reilly Media, 2008. ISBN 978-0-596-51613-0.

[13] Goldsborough, P. *A Tour of TensorFlow*. 2016.

[14] Griffiths, D. a Griffiths, D. *Head First Android Development: A Brain-Friendly Guide 2nd Edition*. 2. O'Reilly Media, 2017. Head first Android development. ISBN 9781449362164.

[15] Hollemans, M. *MobileNet version 2* [online]. Duben 2018 [cit. 2020-03-15]. Dostupné z: https://machinethink.net/blog/mobilenet-v2/.

[16] Hopfield, J. J. Artificial neural networks. *IEEE Circuits and Devices Magazine*. 1988, sv. 4, č. 5, s. 3–10. DOI: 10.1109/101.8118.

[17] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W. et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017.

[18] Islam, K. T., Mujtaba, G., Raj, R. G. a Nweke, H. F. Handwritten digits recognition with artificial neural network. In: *2017 International Conference on Engineering Technology and Technopreneurship (ICE2T)*. 2017, s. 1–4. DOI: 10.1109/ICE2T.2017.8215993.

[19] *Pyraminx* [online]. Červenec 2018 [cit. 2020-12-18]. Dostupné z: https://www.jaapsch.net/puzzles/pyraminx.htm.

[20] *Tower Cube, 2×2×3* [online]. Červen 2018 [cit. 2020-12-18]. Dostupné z: https://www.jaapsch.net/puzzles/cube223.htm.

[21] Liptáková, D. *Mobilní aplikace pro demonstraci řešení Rubikovy kostky*. Brno, CZ, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/23032/.

[22] McIntyre, D. A. *The Best-Selling Products of All Time* [online]. 24/7 Wall Street, červen 2012 [cit. 2020-12-05]. Dostupné z: https://247wallst.com/special-report/2012/06/29/the-best-selling-products-of-all-time/.

[23] McLaughlin, M. *Android Versions Guide: Everything You Need to Know* [online]. LifeWire, březen 2021 [cit. 2020-04-17]. Dostupné z: https://www.lifewire.com/android-versions-4173277.

[24] Mitchell, T. M. *Machine Learning*. 1. vyd. USA: McGraw-Hill, Inc., 1997. 1-3 s. ISBN 978-0070428072.

[25] Mèffert, U. *Spatial puzzle toy*. EU Patent 0042965, červen 198. Dostupné z https://www.jaapsch.net/puzzles/patents/ep42695.pdf.

[26] *About OpenCV* [online]. 2021 [cit. 2020-03-05]. Dostupné z: https://opencv.org/about/.

[27] *OpenCV Java documentation (4.5.1)* [online]. 2021 [cit. 2020-03-05]. Dostupné z: https://docs.opencv.org/4.5.1/javadoc/index.html.

[28] O'Shea, K. a Nash, R. *An Introduction to Convolutional Neural Networks*. 2015.

[29] Otsu, N. A Threshold Selection Method from Gray-Level Histograms. *IEEE Transactions on Systems, Man, and Cybernetics*. 1979, sv. 9, č. 1, s. 62–66. DOI: 10.1109/TSMC.1979.4310076.

[30] Rubik, E. *Spatial logical toy.* U.S. Patent 4378116, říjen 1980. Dostupné z https://patents.google.com/patent/US4378116A/.

[31] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. a Chen, L.-C. *MobileNetV2: Inverted Residuals and Linear Bottlenecks.* 2019.

[32] Singmaster, D. *Notes on Rubik's Magic Cube.* 1. Leden 1981. 3-4 s. ISBN 0-89490-043-9. Dostupné z https://maths-people.anu.edu.au/~burkej/cube/singmaster.pdf.

[33] *TensorFlow Hub* [online]. 2021 [cit. 2020-04-15]. Dostupné z: https://tfhub.dev/.

[34] *TensorFlow Lite Model Maker* [online]. 2021 [cit. 2020-04-15]. Dostupné z: https://www.tensorflow.org/lite/guide/model_maker.

[35] Tierney, J. *The perplexing life of Erno Rubik* [online]. Family Media Inc., březen 1986 [cit. 2020-12-08]. Dostupné z: https://fei.edu.br/~rbianchi/daia/celula/cubo/rubik_magic_cube.txt.

[36] Webster, G. *The little cube that changed the world* [online]. CNN Business, říjen 2012 [cit. 2020-12-05]. Dostupné z: https://edition.cnn.com/2012/10/10/tech/rubiks-cube-inventor/index.html.

[37] Wiki, T. P. *Tower Cubes (2x2xN)* [online]. Ruwix, 2017 [cit. 2020-12-16]. Dostupné z: https://ruwix.com/twisty-puzzles/2x2xn-cuboid-puzzles/.

[38] WikiCube. *Notation* [online]. Fandom, 2021 [cit. 2020-12-11]. Dostupné z: https://rubiks.fandom.com/wiki/Notation.

[39] Yegulalp, S. *What is Tensorflow?* [online]. InfoWorld, červen 2019 [cit. 2020-04-15]. Dostupné z: https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html.

[40] ZBOŘIL, F. V. *Opora předmětu Základy umělé inteligence (IZU).* 2021 [cit. 2021-05-01]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IZU/private/2021-opora-IZU.pdf.