VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VYUŽITÍ METOD UMĚLÉ INTELIGENCE PRO ŘEŠENÍ HLAVOLAMU RUBIKOVA KOSTKA

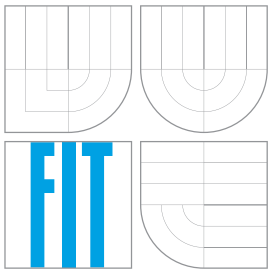
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAKUB KOČVARA

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VYUŽITÍ METOD UMĚLÉ INTELIGENCE PRO ŘEŠENÍ HLAVOLAMU RUBIKOVA KOSTKA

USAGE OF METHODS OF ARTIFICIAL INTELLIGENCE FOR SOLVING OF BRAIN TEASER RUBIK'S CUBE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAKUB KOČVARA
AUTHOR

VEDOUCÍ PRÁCE — Ing. RADIM LUŽA
SUPERVISOR

BRNO 2014

Abstrakt

Cílem této práce je přiblížit čtenáři principy rychlostního skládání Rubikovy kostky a využití umělé inteligence při řešení pomocí těchto metod. Aplikace by měla pomoci řešiteli s logickým myšlením i s učením nových algoritmů.

Abstract

The goal of this paper is to explain the principles of Rubik's Cube speedsolving and use of artificial intelligence with these methods. Application should help solvers with logical thinking and learning new algorithms.

Klíčová slova

Rubikova kostka, řešení hlavolamu, umělá inteligence, algoritmus

Keywords

Rubik's Cube, puzzle solving, artificial intelligence, algorithm

Citace

Jakub Kočvara: Využití metod umělé inteligence pro řešení hlavolamu Rubikova kostka, bakalářská práce, Brno, FIT VUT v Brně, 2014

Využití metod umělé inteligence pro řešení hlavolamu Rubikova kostka

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radima Luži

.....
Jakub Kočvara
19. května 2014

Poděkování

Děkuji Ing. Radimu Lužovi za poskytnuté rady. Také tímto chci poděkovat Ing. Werneru Randelshoferovi a Ing. Stefanovi Pochmannovi za jejich odbornou pomoc a poskytnuté materiály.

Obsah

Kapitola 1

Úvod

Rubikova kostka je jeden z nejpopulárnějších hlavolamů, který pro mnoho lidí pořád představuje nepřekonatelnou překážku. Tato práce si dává za úkol pomoci pokročilejším řešitelům i zaujmout úplné nováčky. Samozřejmě existuje mnoho metod, pomocí niž se dá kostka složit, ale každá z nich je vhodná pro něco jiného. Některé jsou vhodné pro začátečníky, jiné se soustředí na minimální počet pohybů, ale pro nás je důležitá rychlost. Zde se zaměříme zejména na pokročilé metody v čele s Fridrichovou metodou neboli CFOP, kterou dnes využívá většina rychlostních řešitelů, tzv. *speedcuberů*.

1.1 O hlavolamu

V roce 1974 si maďarský profesor architektury Ernő Rubik vyrobil první prototyp, jako pomůcku při vysvětlování třídimenzionálních struktur svým studentům. Záhy si uvědomil potenciál a zajímavé matematické vlastnosti svého vynálezu a o rok později zažádal o patent na nový hlavolam, který nazval *Kouzelná kostka*. Několik let se prodávala pouze v Maďarsku, ale v 80. letech získala celosvětovou popularitu. K dnešnímu dni se prodalo přes 350 milionů kusů, a jedná se tak o nejprodávanější hračku všech dob[2].

K úspěšnému vyřešení Rubikovy kostky je nutné pochopit jak se vlastně chová a funguje. Většina lidí, kteří se s Rubikovou kostkou někdy setkali se ji snaží řešit tzv. *po stranách*. Tento způsob je vhodný pouze na procvičení logických schopností, ale bohužel není vhodný k úspěšnému složení celé kostky. K pochopení principů nám pomůže, pokud se zamyslíme nad tím jak kostka funguje mechanicky. Barevné strany jsou u originální kostky uspořádány takto: bílá naproti žluté, červená naproti oranžové a modrá naproti zelené[1]. Znamená to tedy například, že neexistuje hranová kostička bílá-žlutá. Kostka se skládá z následujících částí:

- 1 jádro, na které je napojeno 6 středů, které nesou pouze jednu barvu a nemění vůči sobě pozici
- 12 hran, které nesou dvě barvy
- 8 rohů, které nesou 3 barvy

[1] Na kostkách z Japonska a některých dalších zemích spolu bílá a žlutá sousedí. Jedná se ale o neoficiální barevné uspořádání a v této práci se jím nebudeme zabývat.

Rubikova kostka by mohla tedy teoreticky nabývat

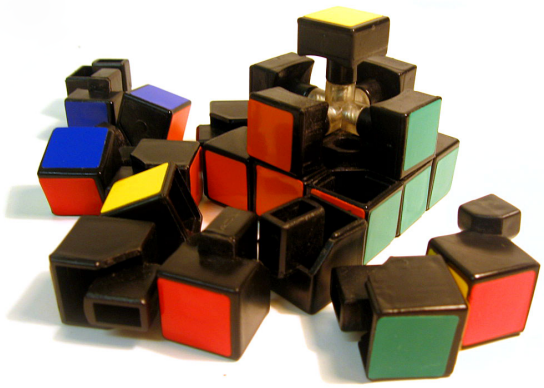
$$8! \times 3^8 \times 12! \times 2^{12} \approx 5.19 \times 10^{20} \tag{1.1}$$

kombinací. Ovšem ne každá z těchto kombinací je řešitelná, neboli se do ní můžeme dostat rozložením kostky z vyřešeného stavu. Například pokud bychom ve složeném stavu pouze vyjmuli hranovou kostku, otočili ji a vrátili nazpět, bude hlavolam bez dalšího vnějšího zásahu nevyřešitelný. Poupravením kombinatorických výpočtů z předchozí rovnice (1.1) získáme tuto rovnici:

$$8! \times 3^7 \times (12!/2) \times 2^{11} \approx 4.32 \times 10^{19} \tag{1.2}$$

Výpočet se zakládá na tom, že máme 8 rohových kostiček, které mohou být v jakémkoli rohu, a každá může mít jednu ze tří orientací. Lze dokázat, že orientaci prvních sedmi můžeme zvolit libovolně a orientace osmého rohu vyplyne z předchozích sedmi (proto jen 3^7). Hranových kostiček je 12, ale mohou být pouze v polovině všech možných pozicí, kvůli sudému znaménku permutace rohových kostiček $(12!/2)$. Prvních 11 hran můžeme opět orientovat libovolně, ale vyplyne z toho orientace poslední hrany. Do výpočtu nejsou zahrnuty středy stran, protože na některých kostkách je důležitá jejich orientace (např. kostky, kde strany tvoří obrázky), na některých starších kostkách dokonce nejsou středy vůbec označeny barevně. Pokud bychom uvažovali standardní Rubikovu kostku, středy mohou být v 24 různých permutacích. Výsledné číslo bychom tedy poté vynásobili touto hodnotou.

Obrázek 1.1: Rubikova kostka. odkaz 1



Obrázek 1.2: Části viditelné na rozložené kostce. odkaz 2

Kapitola 2

Metody

Od úplných začátků se lidé snažili v kostce najít řád a vymyslet univerzální algoritmy, které by jakkoli rozloženou kostku dokázali vyřešit. Dříve se tyto metody šířily pouze v odborné literatuře nebo lidovou slovesností. S příchodem internetu začali lidé své postupy šířit, diskutovat mezi sebou o problémech a úskalích, a tím se začaly metody zdokonalovat. Přestože je kostka stará již přes 30 let, i dnes se objevují nové metody, které přinášejí něco nového. Většina metod je založena na *algoritmech*[1], což jsou sekvence otočení kostkou, která provádíme, abychom dosáhli změny pozice určitých kostiček na hlavolamu, bez porušení toho, co již máme složeno. Metody pro skládání Rubikovy kostky by se daly rozdělit na 3 následující kategorie:

- metody zaměřené na jednoduchost (pro výukové účely a začátečníky)
- metody zaměřené na řešení co nejméně pohyby (nevhodné pro lidské řešitele)
- metody zaměřené na rychlost (pro pokročilé řešitele a speedcubery)

2.1 Metody zaměřené na jednoduchost

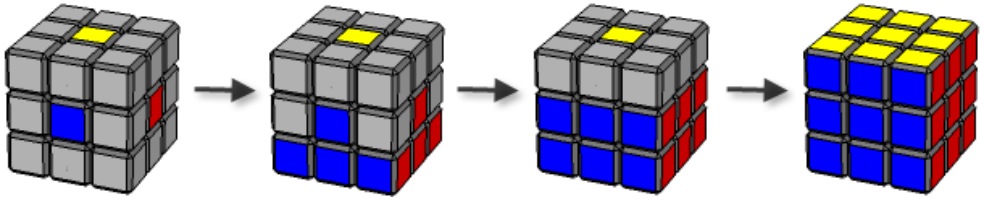
Tyto postupy se snaží najít odpověď na to, jak nejrychleji a s pomocí co nejméně algoritmů naučit řešitele nováčka samostatně složit Rubikovu kostku. Algoritmy jsou sice lehké na zapamatování a jednoduché na provedení, ale nejsou optimalizované, nebo je nutné je několikrát opakovat, abychom se dopracovali ke kýženému výsledku. Proto je skládání pomalé a vyžaduje velké množství pohybů. Časy složení se pohybují v řádech minut a často je potřeba více než 100 pohybů.

Nejrozšířenější metodou je LBL (layer by layer method), z které zpravidla vycházejí i principy ostatních metod (i těch pokročilých). Kostka se skládá po vrstvách (dolní, prostřední a horní) a s každou dokončenou vrstvou se se zvyšuje náročnost použitých algoritmů.

2.2 Metody zaměřené na řešení co nejméně pohyby

Matematikové viděli v Rubikově kostce jinou výzvu. Vytvořit postup, kterým by se dala kostka složit co nejméně pohyby. Za pomocí teorie grup se snažili přijít na algoritmus, který vypočítá optimální postup pro složení kostky nejméně možnými pohyby. Nevýhodou těchto metod je, že jsou závislé na složitých algoritmech na procházení stavového prostoru, a proto

[1] V práci je dále slovo *algoritmus* používáno v obou variantách, význam se liší podle kontextu

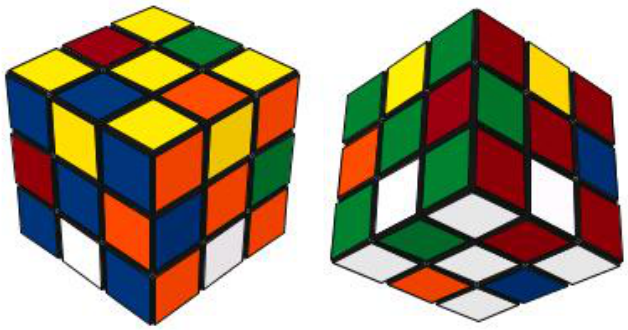


Obrázek 2.1: Znázornění postupu po vrstvách. odkaz 3

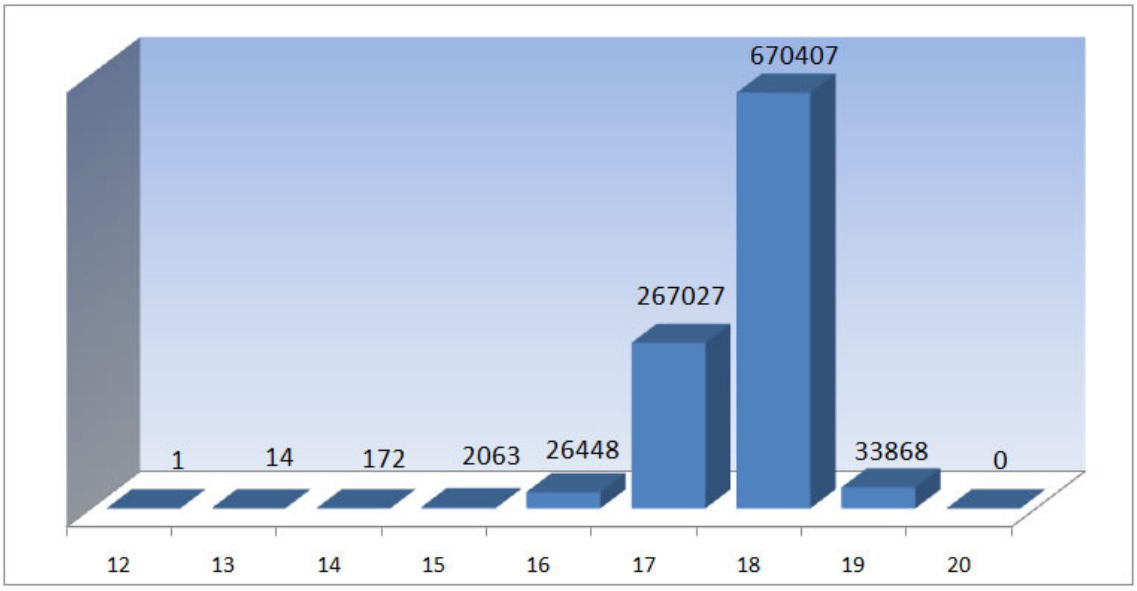
nejsou vhodné pro lidské řešitele. V roce 1981 bylo vynikajících výsledků dosaženo pomocí *Thistlethwaitova algoritmu*[14], který si všiml, že pokud s kostkou provádíme pohyby pouze z určité množiny, může se změnit pouze část kostky. Principem metody je, že na začátku se může při řešení využívat všech pohybů kostkou, cílem bylo dostat se do podgrupy, ve které se k vyřešené kostce můžeme dopracovat množinou méně pohybů. Takto se s přechodem do každé podgrupy zmenšoval počet validních pohybů, až nakonec nezbývaly žádné a kostka byla složena. Thistlethwaite byl schopen rozdělit celý postup do 4 podgrup a jeho algoritmus dokázal vyřešit jakoukoli kostku maximálně 45 pohyby. Na Thistlethwaita navázal v roce 1992 Němec Herbert Kociemba, který ve svém algoritmu používá pouze 2 podgrupy a na procházení možných pohybů používá poupravený A* vyhledávací algoritmus zvaný IDA* (Iterative Deepening A* Search Algorithm), který využívá vyšší informovasti díky vhodným heuristikám a odstraňování neoptimálních podstromů pomocí *pruningu*[4]. Tato metoda potřebuje v průměru ke složení kostky zhruba 21 pohybů.

2.2.1 Maximální nutný počet pohybů (God's number)

Cílem této oblasti bylo také přijít na to, kolik pohybů je nutno maximálně provést, abychom složili jakoukoli kostku. Tato cifra se označuje jako tzv. *God's number*. Ještě na začátku 80. let matematici odhadovali, že God's number bude v řádech stovek. S příchodem grupových metod se očekávané číslo výrazně zmenšilo. V roce 1995 bylo dokázáno, že pozici kostky, které se přezdívá *superflip* je možno složit nejméně 20 pohyby[11]. Superflip je pozice, ve které jsou hrany i rohy na správném místě, ale hrany jsou špatně naorientovány. Několik let se polemizovalo o tom, že God's number je skutečně 20, ale důkaz byl proveden až v roce 2009 týmem, ve kterém figuroval právě Herbert Kociemba[5].



Obrázek 2.2: Pozice *superflip*. odkaz 4



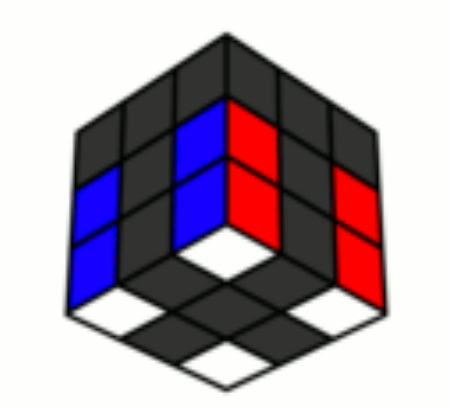
Obrázek 2.3: Znázornění maximálních nutných pohybů pro 1 000 000 náhodně rozložených kostek.[4] odkaz 5

2.3 Metody zaměřené na rychlost

Rychlost složení Rubikovy kostky závisí nejen na vhodnosti metody, ale zejména na zkušenostech a schopnostech řešitele. Rychlostní metody se zakládají na specializovaných algoritmech. Znamená to, že řešitel reaguje na určité situace a vzory na kostce příslušným algoritmem, který ho dostane na další krok. Zpravidla pro pokročilé rychlostní metody platí, že čím více se blížíme složené Rubikově kostce, tím složitější algoritmy musíme aplikovat. V prvních krocích má řešitel často větší volnost a musí se spoléhat zejména na svou logiku při skládání určitých útvarů na kostce (např. *cross*). V dalších krocích už není potřeba tolik přemýšlet a výhodu získavají řešitelé, kteří umí nejrychleji aplikovat naučené algoritmy. Proto můžeme o těchto algoritmech říci, že se skládají z *intuitivní* a *algoritmické* části. Přestože je Rubikova kostka na trhu již dlouho, po roce 2000 zažila druhou vlnu popularity, během které se objevily nové rychlostní metody a *speedcubing* nabral na popularitě. Nejznámější rychlostní metody jsou:

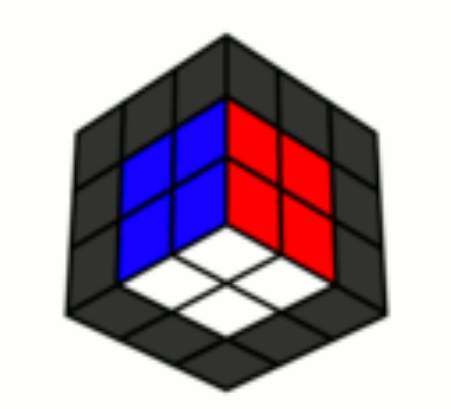
- Fridrichova metoda (CFOP)[3] - nejrozšířenější, nejrychlejší, náročná algoritmická část, více pohybů
- Rouxova metoda[13] - velmi málo pohybů, používá těžko proveditelné pohyby prostřední vrstvou, náročná intuitivní část
- Petrusova metoda[9] - jedna z prvních rychlostních metod, první dvě vrstvy se sestavují intuitivně
- ZZ metoda[12] - nová metoda, efektivní, lehčí na naučení, náročná intuitivní část

Na následujících obrázcích můžete vidět charakteristické oblasti kostky, které se stavějí v jednotlivých metodách.

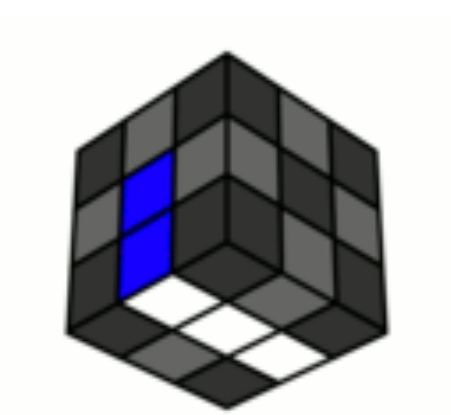


Obrázek 2.4: Fridrichova metoda. odkaz 6

Obrázek 2.5: Rouxova metoda. odkaz 7



Obrázek 2.6: Petrusova metoda. odkaz 8



Obrázek 2.7: ZZ metoda. odkaz 9

Kapitola 3

Fridrichova metoda (CFOP)

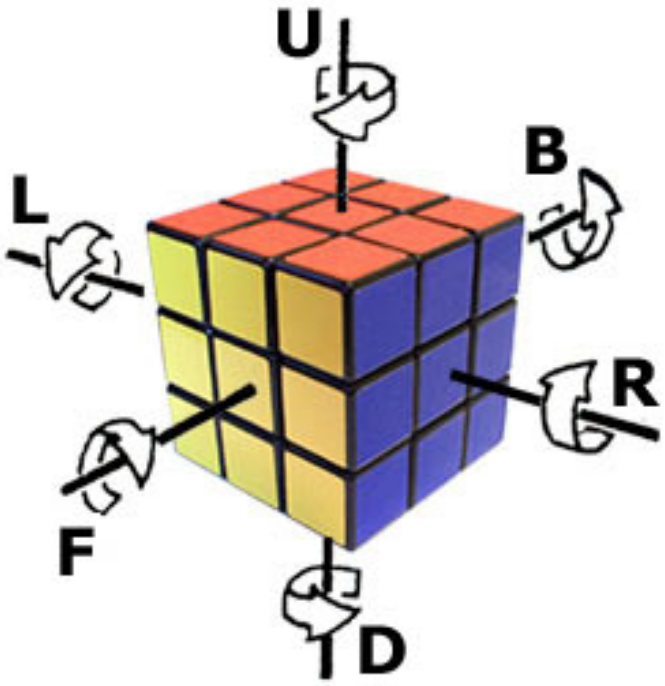
Tato metoda je momentálně nejpopulárnější v kruzích rychlostních řešitelů. Její základy vznikly již v 80. letech, ale zformulována a zpopularizována byla až v roce 1997 českou doktorkou Jessicou Fridrich[3]. Od konce 90. let bylo všech rychlostních rekordů docíleno právě touto metodou[1]. Plnohodnotná Fridrichova metoda se skládá se 4 fází:

- Cross - sestavení kříže na spodní vrstvě, plně intuitivní fáze
- F2L (first two layers) - dokončení prvních dvou vrstev v jednom kroku, částečně algoritmická fáze
- OLL (orienting last layer) - orientace kostiček poslední vrstvy, na konci této fáze je celá vrstva správně orientována, plně algoritmická fáze
- PLL (permutating last layer) - dosazení kostiček poslední vrstvy na správná místa, na konci této fáze je kostka složena, plně algoritmická fáze

3.1 Oficiální notace

Abychom mohli rozebrat algoritmy pro řešení Rubikovy kostky, je nutné znát notaci, v které jsou popsány. Pohyby jednotlivými stěnami se dají popsat písmeny z množiny $A = \{R, L, U, D, F, B\}$. Každé písmeno určuje jednu stěnu kostky označenou pozicí vůči pozorovateli. Jedná se o otočení ve směru hodinových ručiček (představíme si, že by daná strana byla otočena k nám). Pokud je za písmenem „'“, jedná se o otočení proti směru hodinových ručiček. Pokud za písmeno přidáme „2“ (např.: $R2$), znamená to, že danou stěnou otočíme dvakrát, neboli provedeme půlotáčku.

[1] Nejrychlejší řešitelé dosahují průměrného času okolo 8 sekund. Světový rekord k roku 2013 je 5,55 s.[16]



Obrázek 3.1: Oficiální notace. odkaz 10

3.1.1 Speciální pohyby

Přestože můžeme pomocí předchozí notace provést jakýkoli algoritmus, často se používají i další pohyby pro jednodušší provádění a zapamatování postupů.

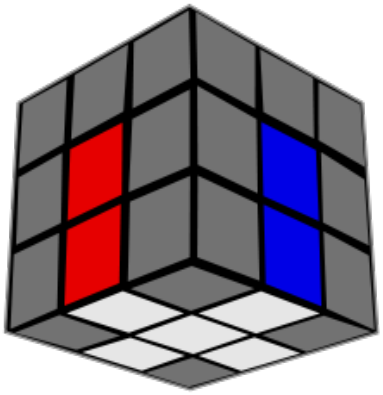
- x - otočení celou kostkou, jako bychom prováděli R
- y - otočení celou kostkou, jako bychom prováděli U
- z - otočení celou kostkou, jako bychom prováděli F
- M - otočení vrstvou M (mezi R a L), jako bychom prováděli L
- E - otočení vrstvou E (mezi U a D), jako bychom prováděli D
- S - otočení vrstvou S (mezi F a B), jako bychom prováděli F

Existuje také zápis pohybu s malým písmenem, což znamená otočení dané strany se sousedící vnitřní vrstvou. Například r, je ekvivalentní zápisu Lx.

3.2 Cross (Kříž)

Sestavení základního kříže spočívá ve správném umístění a orientaci čtyř hran spodní vrstvy. Sice existují algoritmy pro sestavení kříže, ale nejsou optimální, protože umísťují hrany postupně. Zkušení řešitelé provádějí tuto fázi pouze intuitivně. Po dostatečném tréninku by

mělo k naplánování umístění všech hran stačit pouze několikasekundová inspekce kostky. Kříž na libovolné straně se dá vždy složit maximálně 8 tahy[15]. Přestože je tato fáze velmi logická a pro pokročilé řešitele automatická, z hlediska umělé inteligence představuje nejtěžší část.



Obrázek 3.2: Sestavený *cross*.

3.2.1 Řešení kříže pomocí metod umělé inteligence

Chceme dosadit všechny hrany naší spodní vrstvy na jejich správná místa a správně je naorientovat. Přestože existují i naivní metody, budeme chtít efektivnější metodu, která stejně jako lidský řešitel bude umět předvídat tahy a kombinovat více podkroků do jednoho. Budeme proto potřebovat algoritmy, procházející stavový prostor. První technika, které speedcubeři využívají spočívá v tom, že hrany nemusíme dosazovat na předem daná místa. Stačí abychom dodrželi jejich relativní polohu vůči sobě. Pokud bychom uvažovali jako spodní stranu bílou, musí být barvy hran po směru hodinových ručiček takto: *modrá, červená, zelená, oranžová*. Pokud máme takto sestavený kříž, stačí pootočit spodní vrstvu tak, aby barvy hran korespondovaly s bočními stranami.

Algoritmus

Základem našeho algoritmu je cenová funkce, která se vypočítává pro každou hranu. Tou se po každém kroku spočítá počet tahů potřebných k dosazení na správné místo. Na začátku není důležité kam první hranu umístíme. Cenová funkce může být maximálně $F = 2$. Spočítáme cenu dosazení všech hran. Simulujeme postupně dosazení každé hrany a vždy znovu spočítáme cenovou funkci. Po provedení sledujeme změnu v ceně dosazení ostatních hran. Pokud se součet cenové funkce hran ve výsledku zvýšil, můžeme tuto variantu vyloučit. Za nejvhodnější považujeme pohyb, který má nejnižší $F_{tah} + \sum F_{po}$.

Například pokud je na začátku $F_{modra} = 2, F_{cervena} = 2, F_{zelena} = 1, F_{oranzova} = 2$, tak nemusí být vždy nejvýhodnější začít zelenou hranou. Názorně to můžeme předvést na následující situaci:

$$\sum F_{pred} = 2 + 2 + 2 + 1 = 7$$

1. pokud provedeme pohyb zelenou ($F_{tah} = 1$), ceny ostatních hran se nezměnily $\rightarrow$ $F_{po} = 6$
2. pokud provedeme pohyb modrou ($F_{tah} = 2$), cena červené hrany se zmenší na 1 a cena zelené na 0 $\rightarrow$ $F_{po} = 4$

Provedeme tedy zdánlivě méně výhodný pohyb modrou hranou, protože ve výsledku nám přiblíží červenou hranu ke své pozici o jeden tah a zasadí zelenou hranu na správné místo. Podobně dále procházíme stavovým prostorem a hledáme nejvýhodnější pohyb hranou, dokud není kříž složen.

3.3 F2L

Narozdíl od začátečnických metod, kde se nejprve řeší rohy spodní vrstvy a v dalším kroku hrany prostřední vrstvy, F2L spojuje tyto dvě fáze do jedné. Chceme pomocí jednoho algoritmu správně naorientovat hranu s rohem, spojit je a „zasunout“ na správné místo do kříže. Existuje 42 různých pozic, ve kterých se mohou vůči sobě nacházet, a tak existuje také stejný počet algoritmů[2], kterými je lze dosadit na požadované místo[1]. Některé algoritmy jsou však vůči sobě zrcadlové nebo velmi logické, a proto je reálné číslo algoritmů na zapamatování výrazně nižší. Narozdíl od algoritmů pro pozdější fáze, jsou F2L algoritmy pouze takovou pomůckou k intuitivnímu řešení. Zkušenější řešitelé pochopí jejich principy, a poté je dokáží kombinovat a obměňovat je. Proto F2L označujeme jako částečně intuitivní fázi.

Obrázek 3.3: Jedna z F2L situací. odkaz 11

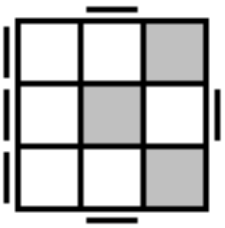
Obrázek 3.4: F2L situace vyřešena pomocí algoritmu $U'RUR'URUR'$. **??**

[2] ve skutečnosti jen 41, protože v jedné z pozic se nacházejí již na požadovaném místě

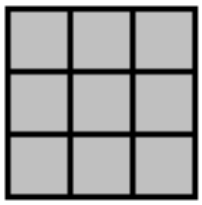
3.4 OLL

Zbývá složit horní vrstvu. V této fázi chceme, aby všechny kostičky horní vrstvy byly orientovány tak, aby měly nahoře barvu středu horní vrstvy. Celkem existuje 57 různých uspořádání a stejně algoritmů[6]. Tyto algoritmy sice nejsou tak složité a dlouhé jako PLL, ale je jich značně více. OLL algoritmy se řešitelé učí až na konec.

Obrázek 3.5: Kostka po fázi OLL. odkaz 13



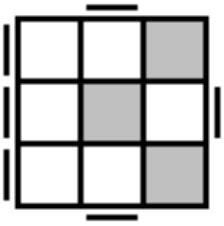
Obrázek 3.6: Jedna z OLL situací.



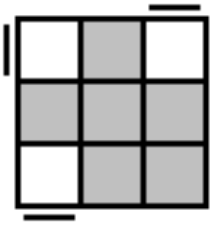
Obrázek 3.7: OLL vyřešeno pomocí algoritmu $y'rUR'URU2r2U'RU'R'U2r$

3.4.1 2-look OLL

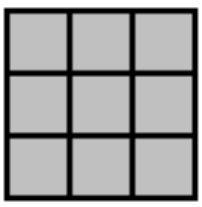
Pro řešitele začínající s CFOP existuje ještě zjednodušená varianta zvaná *2-look OLL*. Znamená to, že se OLL rozdělí na dvě podfáze. V první naorientujeme pouze stěny horní vrstvy a v druhé rohy. Použijeme tedy 2 algoritmy, které jsou ale značně jednodušší. K orientaci hran nám stačí znát pouze 2 algoritmy a k orientaci rohů jen 7. Tato varianta vyžaduje sice více pohybů než plné OLL, ale za to je velmi jednoduchá.



Obrázek 3.8: Stejná situace jako na obr. 3.6



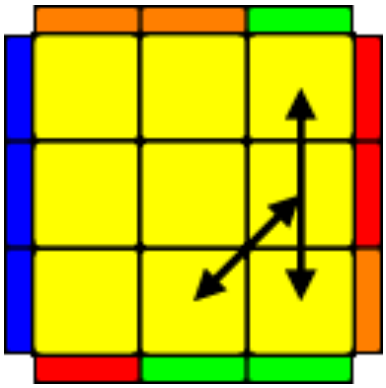
Obrázek 3.9: Hrany jsou orientovány pomocí $FURU'R'F'yFURU'R'F'$



Obrázek 3.10: Rohy jsou orientovány pomocí $y2R'U'RU'R'U2R$

3.5 PLL

Ve finální fázi CFOP je nutné správně orientované kostičky horní vrstvy dosadit na správná místa bez porušení toho, co již máme složeno. Tato část je sice náročná na naučení, ale extrémně zrychluje proces skládání horní vrstvy. Nyní se již kostka může nacházet pouze v jedné z 21 situací[7]. PLL algoritmy jsou nejdelší a nejsložitější. Jsou pojmenovány po písmenech, které připomínají způsob, jakým musíme horní vrstvu permutovat. Z hlediska umělé inteligence je zdaleka nejjednodušší. Stačí na jednu z předem známých variant kostky aplikovat příslušný algoritmus.



Obrázek 3.11: *J-permutace* lze vyřešit např. algoritmem $RU2R'U'RU2L'UR'U'L$. odkaz 14

Kapitola 4

Implementace

Práce je zpracována jako webová aplikace. Část, která slouží k řešení hlavolamu je napsána v čistém JavaScriptu, uživatelské prostředí využívá i frameworku JQuery. Pro vizualizaci kostky a jako model slouží implementace v nové technologii WebGL, kterou podporují moderní browsery. Tato implementace je dílem švýcarského inženýra Wernera Randelshofera vydána v roce 2011 pod licencí CreativeCommons 3.0. Části jeho kódu jsou z WebGL demo repository, tento kód patří Apple Inc. a Google Inc. a je použit v souladu s jeho licencemi.

Technologie WebGL je z dílny Mozilla Foundation a je plně podporována prohlížeči Mozilla Firefox a Google Chrome. Od verze 11 ji podporuje také Internet Explorer. Ostatní prohlížeče WebGL nepodporují, nebo je nutné WebGL manuálně zapnout (např. Safari).

4.1 Datový model kostky

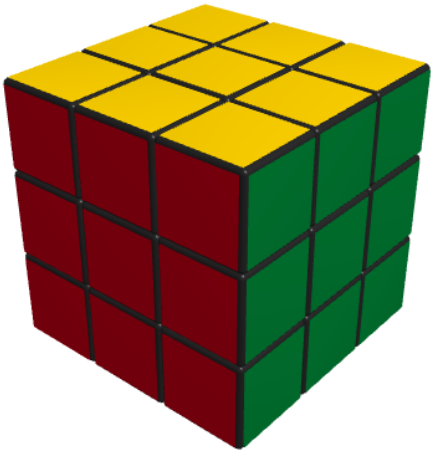
Informace o stavu kostky jsou uloženy v polích a přistupuje se k nim pomocí metod. Aby bylo jasnější, jak je kostka popsána, je nutné definovat následující pojmy:

- strana (face) - počet : 6 - nikdy nemění svoji pozici vůči „pozorovateli“, např. *strana 2* bude vždy vpředu
- stěna (side) - počet : 6 - jednotlivé stěny odpovídají barvám středů kostky a ty mohou měnit svoji pozici, např. *stěna 1 (žlutá)* leží na *straně 4*
- vrstva (layer) - počet : 3 - rozdělují kostku na 3 horizontální části ležící na sobě
- střed (center) - počet : 6 - část kostky nesoucí jednu barvu
- hrana (edge) - počet : 8 - část kostky nesoucí dvě barvy
- roh (corner) - počet : 12 - část kostky nesoucí tři barvy
- nálepka (sticker or facet) - počet : 54 - zde se oprostíme od mechanických principů kostky a zajímá nás pouze dvojrozměrný prvek strany, který nese jednu barvu

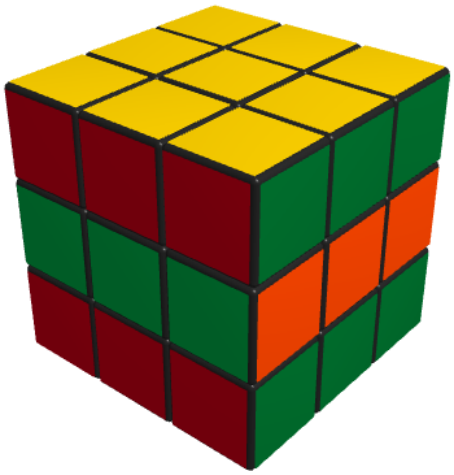
4.1.1 Rozdíl mezi stranou a stěnou

Na obrázku 4.1 vidíme složenou kostku, na které čísla stran odpovídají číslům stěn. Pokud provedeme pohyb prostřední vrstvou podle obrázku 4.2, změní se barva středové části přední strany na oranžovou a čísla stran se již nerovnají číslům stěn.

Pokud by například byly stěny uloženy v poli [červená, zelená, oranžová, modrá], kde na prvním indexu je stěna ležící na přední straně, bude po otočení pole následující: [červená, zelená, oranžová, modrá].

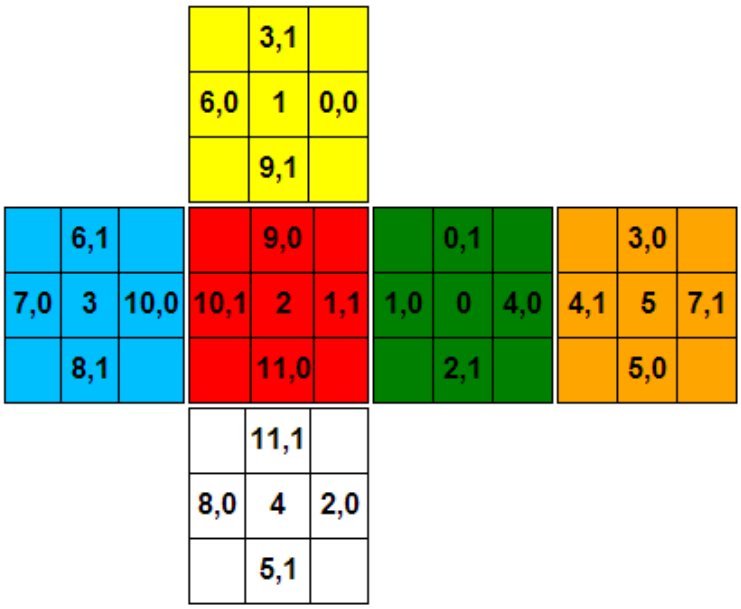


Obrázek 4.1: Strany odpovídají stěnám.



Obrázek 4.2: Stěny jsou vůči stranám o jednu pootočeny

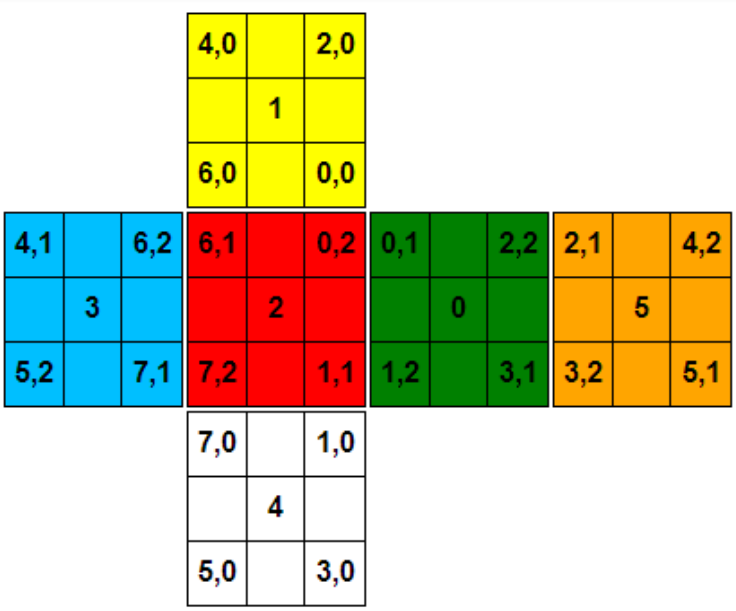
4.1.2 Rozmístění hran



Obrázek 4.3: Označení jednotlivých hran na modelu.

Hrana na kostce může nabývat jedné z 8 pozic a jedné ze dvou orientací. Pokud je tedy hrana na správném místě, musí být navíc správně orientovaná, aby se dala považovat za vyřešenou. Pokud je hrana na správné pozici, ale špatně orientována, je často potřeba velké množství pohybů na její změnu. Proto je při skládání nutné brát ohled na obě kritéria. Vzhledem k vlastnostem použitého Rubikovy kostky je správná orientace nekonzistentní. Některé hrany jsou složeny při orientaci 0, některé při orientaci 1. Na obrázku 4.3 můžete vidět rozloženou síť kostky s vyznačenými pozicemi a orientacemi hran.

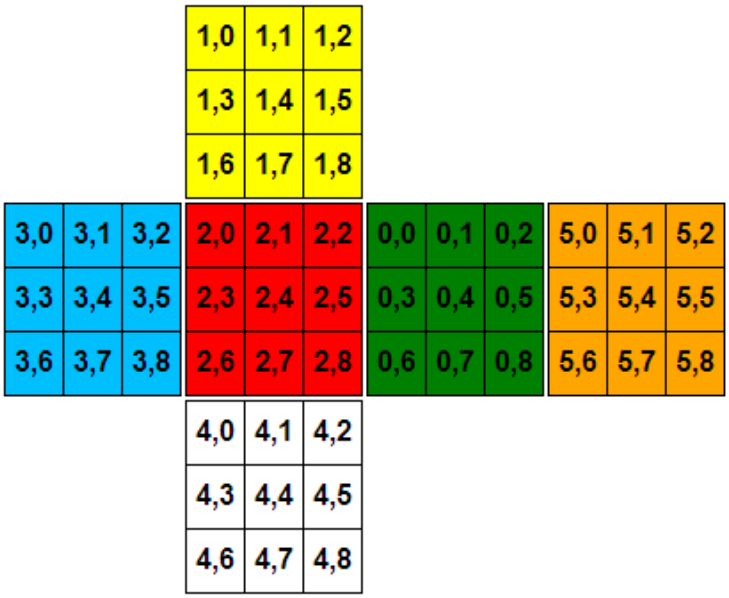
4.1.3 Rozmístění rohů



Obrázek 4.4: Označení jednotlivých rohů na modelu.

Rohy mohou nabývat jedné z 12 pozic a tří orientací. Pokud je roh orientován tak, že barva jeho nálepky souhlasí s barvou stěny, na které leží, můžeme říct, že má orientaci 0. Z toho vyplývá, že na složené kostce mají všechny rohy orientaci 0. Na obrázku 4.4 můžete vidět rozloženou síť kostky s vyznačenými pozicemi a orientacemi rohů.

4.1.4 Rozmístění nálepek



Obrázek 4.5: Označení jednotlivých rohů na modelu.

Nálepka je nejmenší jednotka na kostce. Její pozice se nemění a nabývá vždy jen jedné barvy. Barva nálepky na určité pozici nás zajímá zejména v pozdějších fázích řešení kostky, kdy pozice rohů i hran mohou být pro dva algoritmy stejné a rozhoduje se tak až na úrovni nálepek. Ty jsou uloženy v dvourozměrném poli, v kterém první index označuje číslo stěny a druhý je její pozice na této stěně. Indexy nálepek můžete vidět na obrázku 4.5

4.2 Použité algoritmy

Tato kapitola v detailu popisuje algoritmy implementované pro jednotlivé fáze řešení kostky. První 2 fáze sledují pohyb hran a rohů po kostce a řeší v jednu chvíli vždy jen jeden pohyb. Fáze OLL a PLL rozpoznávají na kostce vždy jednu z předem definovaných situací a na ni reagují příslušnou posloupností pohybů z tabulky. Algoritmy Cross a F2L jsou výpočetně značně náročnější.

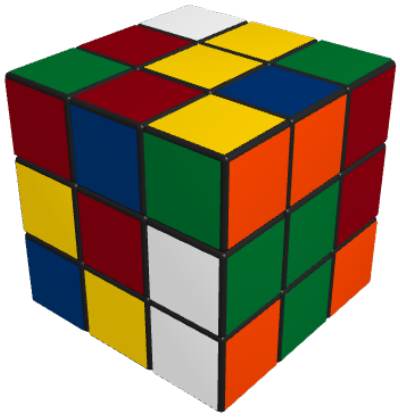
4.2.1 Cross

Aby byl algoritmus schopný konkurovat lidským řešitelům, je nutné, abychom co nejvěrněji algoritmizovali myšlenkový pochod lidských řešitelů. Algoritmus funguje tak, že se snaží dosadit hrany spodní vrstvy na jejich správná místa ve správne orientaci. Složením hran vznikne požadovaný *cross*. Pro zjednodušení a zpřehlednění řešení vždy nastavíme na spodní stranu bílou stěnu. Čísla hran jsou tedy podle nákresu 4.3 tyto: [2,5,8,11].

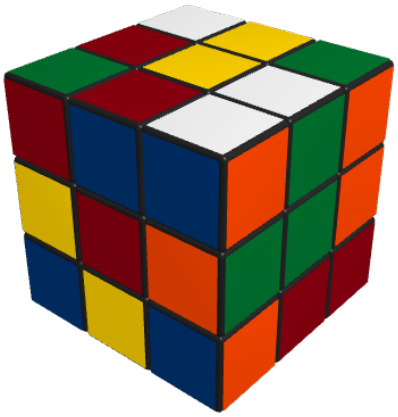
Uvažujeme, že hrana se nemusí zasunout přímo na odpovídající pozici. Při dosazení do spodní vrstvy tedy stačí, aby byly vůči sobě hrany správně napozicovány (např. zelená je naproti modré). Tomuto postupu se říká *relativní kříž (relative cross)*. Relativní kříž snižuje počet pohybů potřebných pro vyřešení této fáze, protože po jeho dostavění můžeme všechny hrany dostat na správná místa pouze jedním pohybem spodní vrstvou.

Hrana může být vůči své správné pozici v několika různých situacích. Počet možných situací lze pomocí nastavení relativního kříže snížit až na pět. Každou lze složit 0-2 pohyby. Počet pohybů nutných ke složení hrany určuje cenová funkce $f(e)$, kde e je skládaná hrana.

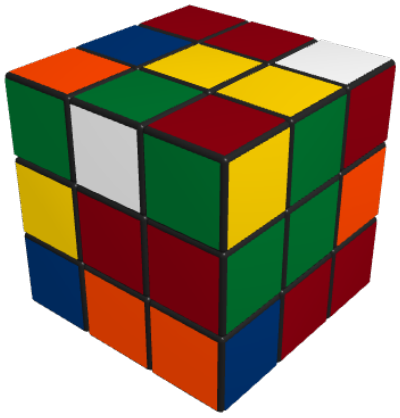
- situace 1 (4.6) - hrana se nachází v prostřední vrstvě, $f(e) = 1$
- situace 2 (4.7) - hrana se nachází v horní vrstvě, bílá nálepka leží na žluté stěně, $f(e) = 1$
- situace 3 (4.8) - hrana se nachází v horní vrstvě, bílá nálepka neleží na žluté stěně, $f(e) = 2$
- situace 4 (4.9) - hrana se nachází ve spodní vrstvě, bílá nálepka neleží na bílé stěně, $f(e) = 2$
- situace 5 - hrana se nachází ve spodní vrstvě, složeno, $f(e) = 0$



Obrázek 4.6: Zelená hrana v situaci 1



Obrázek 4.7: Zelená hrana v situaci 2



Obrázek 4.8: Zelená hrana v situaci 3

Obrázek 4.9: Zelená hrana v situaci 4

Před složením situací 3 a 4 je navíc nutné zkontrolovat, zda 1. pohyb, kterým si hranu připravíme na zasunutí na správnou pozici, neposune již složenou hranu ze své pozice. Pokud by to nastalo, musíme si zapamatovat, že ji tam poté opět musíme vrátit. Tím přidáváme pohyb, takže je možné, že k vyřešení této situace bude potřeba až 3 pohybů. Těmto případům se chceme samozřejmě vyhnout.

Na složení kříže je nutné vyřešit 4 hrany. Pohyby při řešení jedné hrany mohou ovlivnit ostatní hrany negativně, ale i pozitivně. Proto je nevýhodné skládat hrany vždy ve stejném pořadí. Existuje 4! = 24 možných pořadí. Ukázalo se, že postup s algoritmem na výpočet cenovou funkcí a backtrackingem (viz. 3.2.1) nebyl optimální. Může se stát, že pohyb hranou, který na začátku zvýší jiné hraně cenovou funkci, může při řešení dalších hran ušetřit více pohybů a je tím pádem nejoptimálnější. Proto je využito malého počtu permutací a je vyzkoušeno každé pořadí. Ve výsledku je použito to pořadí, které lze složit nejméně pohyby.

Pseudokód pro funkci pro řešení kříže vracející pole optimálních pohybů

```
// získáme všechny permutace možných pořadí
permutace = získat_permutace(červená, zelená, modrá, oranžová);

// minimální počet pohybů
minimum = 0;

// uložíme stav rozložené kostky
ulož_stav_kostky();

// procházení všech pořadí (je jich 6! = 24)
for (pořadí in permutace){

        // nastavení správné stěny dopředu
        načti_stav_kostky();

        // pole s pohyby bude prázdné
        smaž(pohyby);

        // procházení všech hran v daném pořadí
        for (hrana in pořadí) {

                // složí danou hranu, vrátí pohyby, potřebné k řešení
                pohyby += vyřeš(hrana);
        }

        // finální pootočení relativním křížem na správnou polohu
        pootoč_křížem();

        // smazání přebytečných pohybů
        uprav_pohyby(pohyby);

        // pokud je počet pohybů nové minimum -> uložíme ho
        if (první_průchod || délka(pohyby) < minimum) {

                optimální_pohyby = pohyby;
        }
}
return optimální_pohyby;
```

Pseudokód pro funkci na generování pohybů pro složení jedné hrany

```
function vyřeš(hrana) {
        //získání posunutí mezi pozicí hrany a cílovou pozicí na relativním
            kříži
        posun = získej_posun(hrana) - získej_posun_kříže();

        if (situace == 1) {

                // nachystá kříž pro hranu
                otoč_kříž_o(posun);

                // získá stranu, na které hrana se nachází
                strana = získej_stranu(hrana);

                // přidá pohyb stranou (1. parametr: strana, 2. parametr: o
                    kolik)
                přidej_pohyb(strana,2);
        }
        else (situace == 2) {
                otoč_kříž_o(posun);
                strana = získej_stranu(hrana);

                // zjistí o kolik bude nutné stranu otočit
                úhel = získej_úhel(hrana);
                přidej_pohyb(strana,úhel);
        }
        else (situace == 3 || situace == 4) {
                otoč_kříž_o(posun - 1);
                strana = získej_stranu(hrana);

                // pokud změníme znaménko otáčecího úhlu, situace 3 a 4 jsou
                    stejné
                úhel = situace == 3 ? získej_úhel(hrana) : -získej_úhel(hrana
                    );
                přidej_pohyb(strana, úhel);

                // dostáváme se do situace 2, rekurzivně složíme
                vyřeš(hrana);

                // pokud jsme při otáčení narušili složenou hranu, vrátíme ji
                     do původního stavu
                if (narušená_hrana) {
                        přidej_pohyb(strana, -úhel);
                }
        }
}
```

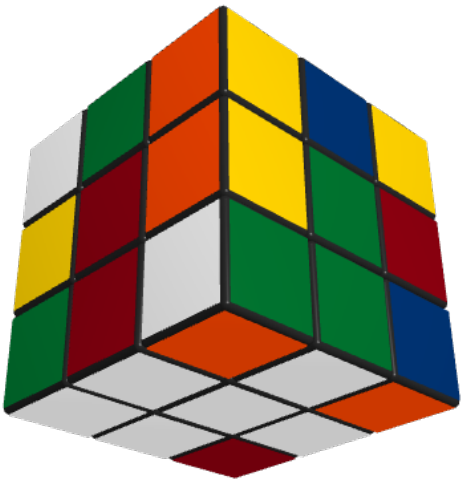
Funkce na sloučení a mazání nadbytečných pohybů

Při řešení se může stát, že nám algoritmus vygeneruje posloupnost pohybů, které jsou buď *nadbytečné* nebo *duplikátní*. Nadbytečný pohyb vznikne, pokud bychom provedli otočení stranou v jednom směru a hned poté v opačném směru. Pokud například provedeme R a hned poté R', ve výsledku se kostka nezmění. Stejně tak dva po sobě jdoucí pohyby R můžeme převést na $R2$. K tomuto dochází pouze na hranicích mezi fázemi (nebo podfázemi), pokud fáze končí na určitý pohyb a další fáze začíná opačným pohybem nebo znovu stejným. Proto po každé fázi vygenerovanou posloupnost pohybů „vyčistíme“ a až poté je aplikujeme.

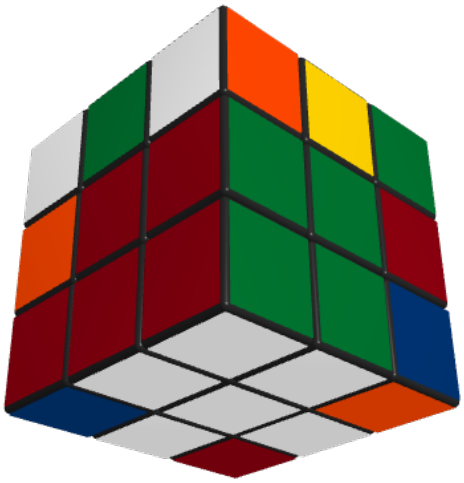
4.2.2 F2L

Fáze F2L (First 2 Layers) je pravděpodobně pro lidské řešitele nejtěžší na naučení. Kombinuje algoritmický s tzv. *intuitivním* přístupem. Zároveň je to také nejdelší fáze při řešení a je na ni potřeba nejvíce pohybů. Před začátkem této fáze máme složený kříž na dolní vrstvě a cílem této fáze je mít kompletně složenou spodní a prostřední vrstvu.

Složení prvních dvou vrstev spočívá ve spárování rohové části ze spodní vrstvy s korespondující hranou a jejich zasunutí na správnou pozici vůči kříži. Seskupení rohu s hranou v této fázi se říká *blok* a místo, na které patří se přezdívá *slot*. V kříži jsou 4 sloty pro 4 bloky, které je do nich potřeba zasunout. Jak lidští řešitelé, tak tato implementace funguje tak, že neskládá bloky v předem určeném pořadí, ale snaží se dávat přednost výhodnějším blokům. Zde již pořadí bloků nemusí být nutně určeno jen pohyby, které jsou nutné na jejich vyřešení, ale i mechanické obtížnosti skládání. Stejně jako většina lidských řešitelů hodnotí i tato implementace bloky jednoduše podle toho, kolik částí tvořících blok je spolu v horní vrstvě.



Obrázek 4.10: Prázdný slot pro blok červená-zelená



Obrázek 4.11: Zaplněný slot červená-zelená

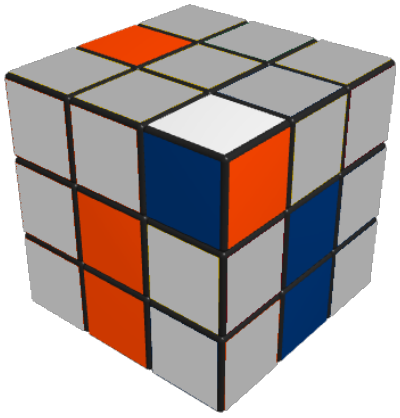
Existuje 42 různých F2L situací, pro které se v tabulce vyhledá příslušná posloupnost pohybů. Abychom mohli tyto postupy aplikovat, je nutné roh s hranou připravit pro jednu z těchto pozic. Tato část je opět intuitivní a v implementaci se tak snažíme napodobit chování lidských řešitelů. Rohová část se musí nacházet buď v horní vrstvě přímo nad odpovídajícím slotem, nebo v dolní vrstvě přímo ve slotu. Hrana může být kdekoli v horní vrstvě, nebo opět ve slotu. Je nutné obě části „vytáhnout“ do horní vrstvy. Předtím je také vhodné nastavit část, co již v horní vrstvě je tak, abychom získali co nejjednodušeji řešitelnou F2L situaci. Používame i speciální algoritmy[1] na výjimečné situace kdy jsou roh s hranou napozicovány ve správném slotu, ale špatně naorientovány.

[1] Zde myšleno jako posloupnost pohybů na kostce.

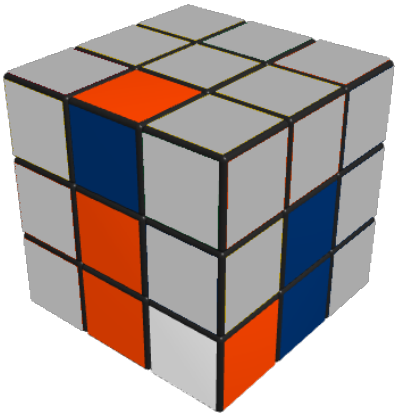
Pořadí, v kterém se sloty budou naplňovat je určeno tak, že se každému potencionálnímu bloku přidělí číslo od 0 do 2, které určuje, jak složité je ho sestavit. Jednotlivé situace, v kterých se části bloku moho nacházet jsou ohodnoceny cenovou funkcí $f(e, c)$, kde e je hrana bloku a c je roh bloku. Pravidla pro hodnocení bloků jsou popsána následovně:

- Situace, kdy žádná část bloku není ve svém slotu
  - Roh ani hrana neleží v horní vrstvě - bylo by nutné je oba připravit, $f(e, c) = 2$
  - Roh leží v horní vrstvě - je nutné připravit hranu, $f(e, c) = 1$
  - Hrana leží v horní vrstvě - je nutné připravit roh, $f(e, c) = 1$
  - Roh i hrana leží v horní vrstvě 4.12 - vše je připraveno, $f(e, c) = 0$
- Situace, kdy alespoň jedna část bloku není ve svém slotu
  - Hrana leží v horní vrstvě (roh ve svém slotu) 4.13 - v některých situacích je navíc nutné upravit pozici hrany v horní vrstvě , $f(e, c) = 0$ nebo $f(e, c) = 1$
  - Roh leží v horní vrstvě (hrana ve svém slotu) 4.14 - vše je připraveno, $f(e, c) = 0$
  - Roh i hrana leží ve svém slotu 4.15 - vše je připraveno, $f(e, c) = 0$

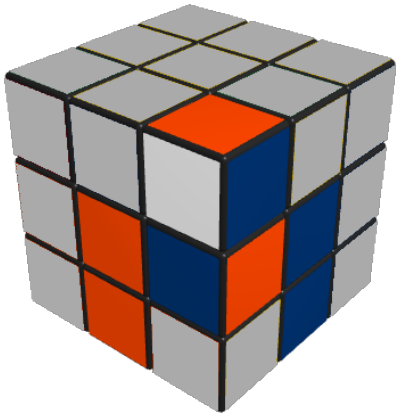
Před řešením této fáze vybereme blok, který je nejvíce připraven pro F2L algoritmus, tedy ten, který má nejnižší cenovou funkci. Po jeho dokončení vypočítáme cenovou funkci pro zbývající bloky a proces opakujeme. Po nachystání částí se dostaneme do jedné z těchto čtyř typů situací, pro které již v tabulce algoritmů existuje záznam. Na obrázcích můžeme vidět tyto situace.



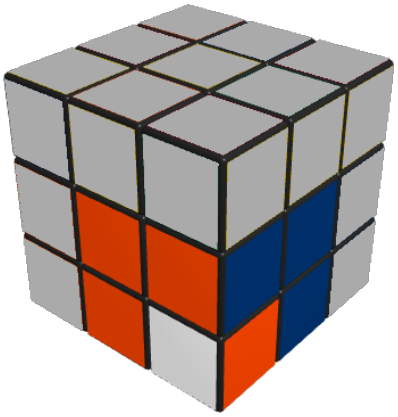
Obrázek 4.12: Roh i hrana leží v horní vrstvě. 24 různých situací.



Obrázek 4.13: Roh je v dolní vrstvě, hrana v horní. 6 různých situací.



Obrázek 4.14: Roh je v horní vrstvě, hrana v dolní. 6 různých situací.



Obrázek 4.15: Roh i hrana jsou ve slotu, ale špatně naorientovány. 6 různých situací.

Pseudokód pro celé F2L

```
//opakujeme pro každý blok
while (zbývají_bloky) {

        // připravíme proměnnou pro nejlepší blok v tento moment
        nejlepší_blok = 0;
        nejlepší_cena = 0;

        // procházíme všechny zbýbající bloky
        for (blok in bloky) {

                // výsledek cenové funkce uložíme
                cena = cenová_funkce(blok);

                // pokud získáme novou nejvýhodnější blok, uložíme si ho
                if (nejlepší_cena < cena) {

                        nejlepší_cena = cena;
                        nejlepší_blok = blok;

                }
        }

        //blok po vyřešení můžeme vyjmout z pole bloků
        vyřeš(nejlepší_blok);
        vyjmi(nejlepší_blok);
}
```

Pseudokód pro řešení bloku

```
// pokud je blok již složen, nemusíme jít dál
if (je_složeno) return;

// vyhledáme, zda pro momentální situaci neexistuje záznam v tabulce
číslo_situace = zjisti_situaci();

// pokud ne, bude nutné připravit části bloku do horní vrstvy
if (není_v_tabulce(číslo_situace)) {

        // pokud není v horní vrstvě ani jedna část, nastavíme tam nejprve
            hranu a poté roh
        if (hrana_dole && roh_dole){
                vytáhni(hrana);
                if (roh_dole) {
                        nastav(hrana);
                        vytáhni(roh);
                }
        }
        // pokud není nahoře hrana, vytáhneme ji
        else if (hrana_dole && roh_nahore){
                nastav(roh);
                vytáhni(hrana);
        }
        // pokud není nahoře roh, vytáhneme ho
        else if (hrana_nahore && roh_dole){
                nastav(hrana);
                vytáhni(roh);
        }

        // v tuhle chvíli jsou všechny části bloku v horní vrstvě, tzn. tato
            F2L situace je uložena v tabulce
        číslo_situace = zjisti_situaci();
}

// použijeme algoritmus, který je uložen pod číslem situace v tabulce
    algoritmů
proveď_algoritmus(číslo_situace);
```

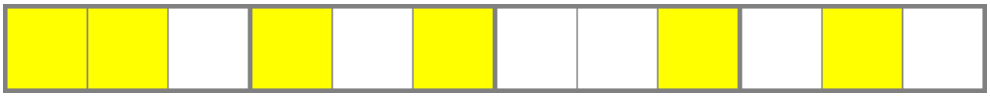
4.2.3 OLL

OLL (Orientation of the Last Layer) spočívá v naorientování všech nálepek horní vrstvy (v našem případě žlutých) na horní stranu. Tato fáze je velmi málo intuitivní, a proto se zde řešení pomocí naší implementace bude lišit jen minimálně od postupu, který by aplikoval lidský řešitel. OLL situací existuje 57 a každá se řeší svým odpovídajícím *algoritmem*, který je uložen v tabulce. Vzhledem k tomu, že v tuto chvíli máme již vyřešené první dvě vrstvy a nechceme si je rozložit, jsou nutné dlouhé a složité posloupnosti pohybů pro požadovaný výstup. Stačí nám tedy jen správně natočit horní vrstvu a aplikovat příslušný algoritmus.

Obrázek 4.16: OLL situace č. 47, pomocí zrcadel můžeme vidět stav na odvrácených stranách.

Princip implementace OLL

Každá OLL situace je uložena v poli jako posloupnost nul a jedniček. Pokud se podíváme na vrchní stranu kostky, obrazec tvořený žlutými nálepkami nám napoví o jakou se bude jednat situaci. Tuto metodu používají lidští řešitele, avšak její nevýhoda je v tom, že podle vzoru na horní straně nelze v mnoha případech poznat exaktní OLL situaci. Řešitelé proto musí navíc zkontrolovat i uskupení žlutých nálepek na bočních stranách. Tato implementace rozpoznává OLL pouze pomocí posloupnosti žlutých nálepek na bocích horní vrstvy. Můžeme si to představit tak, že bychom postupně odlepili všechny nálepky z boků horní vrstvy (začíname přední stranou zleva) a nalepili je za sebe. Dohromady jich bude $3 \times 4 = 12$. Pokud bychom například vzali situaci z obrázku 4.16, můžeme vidět, jak by vypadal pás nálepek na další straně 4.17.



Obrázek 4.17: OLL situace č. 47, pás žlutých nálepek.

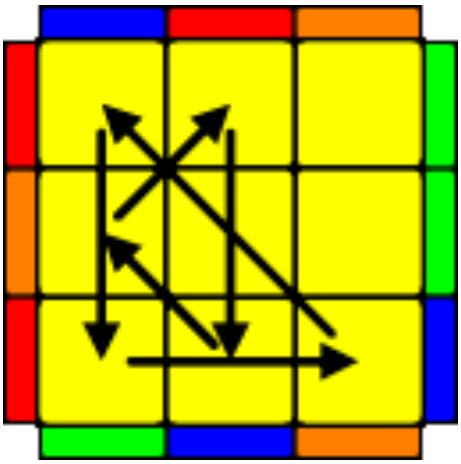
Tento pás namapujeme do pole tak, že žlutá barva odpovídá číslu 1 a vše ostatní číslu 0. Získáme tedy pole [1,1,0,1,0,1,0,0,1,0,1,0]. Touto posloupnosti již lze přesně popsat konkrétní OLL situaci. Pořád je ale pravděpodobné, že nebudeme schopni situaci rozpoznat hned po dokončení F2L, protože bude nutné horní vrstvu natočit tak, aby posloupnost žlutých nálepek souhlasila s polem uloženým v paměti. Využíváme toho, že si tuto posloupnost můžeme představit jako cyklický buffer, v kterém pohyb U horní vrstvou znamená rotaci doprava o 3 a pohyb U' znamená rotaci doleva o 3. Lépe se dá tato technika pochopit z následujícího pseudokódu.

Pseudokód pro OLL

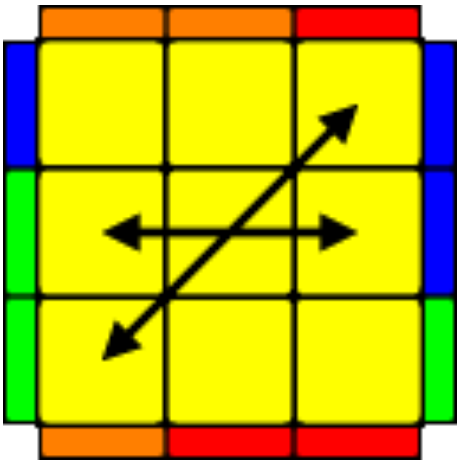
```
// získáme pole v kterém jsou uloženy všechny boční nálepky horní vrstvy
pole_nálepek = nálepky_do_pole();

// převedeme ho na pole, kde žlutá je 1 a vše ostatní 0
vzor = převeď_na_vzor(pole_nálepek);

// procházíme všechny OLL situace (posloupnosti 0 a 1)
for (oll_situace in pole_oll_situací) {

        // 4 možné pootočení horní vrstvou
        for (i = 0; i < 4; i++) {

                // simulujeme otočení horní vrstvou
                upravený_vzor = rotovat(vzor, 3*i);

                // porovnání situací
                if (upravený_vzor == oll_situace) {

                        // pootočíme horní vrstvu
                        otoč_horní_vrstvou_o(i);

                        // aplikujeme algoritmus
                        proveď_algoritmus(číslo_oll_situace);

                        // složeno, můžeme vyskočit z cyklu
                        return;
                }
        }
}
```

4.2.4 PLL

Poslední fáze PLL (Permutation of the Last Layer) vyžaduje opět z velké části exaktní postup, v kterém se lidští řešitelé s naší implementací nebudou velmi lišit. Žluté nálepky jsou všechny správně naorientované na horní straně a zbývá upravit pozice rohů a hran horní vrstvy k tomu, abychom se dostali do složeného stavu. Zkušení lidští řešitelé v okamžiku rozpoznají, které hrany a rohy je mezi sebou nutné vyměnit. Následující obrázky ukazují dvě z 26 možných situací.



Obrázek 4.18: Gb permutace. Je nutné protočit 3 hrany a 3 rohy.



Obrázek 4.19: Na permutace. Je nutné mezi sebou vyměnit 2 protější hrany a 2 protější rohy.

Rozpoznání toho, které části je nutné zaměnit by bylo zbytečně složité naimplementovat, proto použijeme podobný systém jako při OLL. Tentokrát bereme celé části z horní vrstvy a zjišťujeme zda jsou na správné pozici. Je jich celkem 8, protože žlutý střed nás nebude zajímat (víme, že je na správné pozici). Vytvoříme pole, kam uložíme jedničku, pokud část leží na správné pozici, v opačném případě nulu. Začínáme levým předním rohem a postupujeme proti směru hodinových ručiček. Například pole popisující variantu 4.19 bude vypadat takto: [0,1,1,0,0,1,1,0].

Příprava k použití PLL algoritmů není tak jednoduchá jako v případě OLL. Protože všechny žluté nálepky leží na horní straně, nemáme se podle čeho orientovat. Musíme zjístit správné natočení horní vrstvy tak, aby bylo co nejvíce rohů a hran na svém místě. Poté můžeme teprve stav horní vrstvy porovnávat s uloženými situacemi. Tento postup bohužel nefunguje úplně jednoznačně. Například se může stát, že existuje více variant pootočení horní vrstvou se stejným počtem částí na správné pozici, ale jen pro jednu z nich máme záznam v tabulce. Tento jev je naštěstí výjimečný, a tak lze v implementaci ošetřit pouze několika podmínkami.

Pseudokód pro PLL

```
// proměnné na držení informace o nejlepším stavu
nejlepší_součet = 0;
nejlepší_index = 0;

ulož_stav_kostky();

// zkusíme pro všechna pootočení
for (i = 0; i < 4; i++) {

        // pootočíme horní vrstvu
        otoč_horní_vrstvou_o(i);

        // do pole uložíme posloupnost 0 a 1, určujících správnou nebo
            špatnou pozici částí
        pole_pozic = získat_pole_pozic();

        // získáme počet částí na správné pozici
        součet = sečíst_správné(pole_pozic);

        // pokud je součet zatím nejvyšší, uloží se
        if (součet > nejlepší_součet) {
                nejlepší_součet = součet;
                nejlepší_index = i;
        }
}

obnov_stav_kostky();

// pootočíme horní vrstvu
otoč_horní_vrstvou_o(nejlepší_index);

// získáme číslo odpovídající PLL situace
číslo_situace = získat_PLL_situaci();

// aplikujeme příslušný algoritmus
proveď_algoritmus(číslo_situace);
```

Kapitola 5

Porovnání s lidskými řešiteli a statistiky

Cílem práce bylo implementovat algoritmy pro skládání Rubikovy kostky, které pracují podobně jako lidští řešitelé a dokáží jim konkurovat. Nejjednodušší metrikou, kterou lze popsat rozdíl oproti lidem je počet pohybů nutných pro vyřešení jednotlivých fází. Vzhledem k tomu, že ve fázích OLL a PLL skládá i člověk velmi mechanicky, nemá smysl v nich hledat rozdíly proti naší implementaci. Zaměříme se proto jen na intuitivní fáze Cross a F2L. Data, potřebná na srovnání s naší implementací, jsem získal od 10 různých řešitelů, kteří kostku skládají na velmi vysoké úrovni [1]. Od každého jsem získal údaje z 10 pokusů.

5.1 Cross

Pokud chceme kříž postavit vždy jen na bílé stěně, existuje 190 080 možných variací, v kterých se mohou hrany kříže nacházet. Matematicky dokázaný maximální počet pohybů na vyřešení jakékoli z těchto situací je 8. Následující tabulka 5.1 ukazuje rozložení pohybů nutných na složení všech situací. Uvedený počet pohybů je vždy optimální a byl získán pomocí algoritmu s vyčerpávajícím průchodem stavového prostoru. Nejzkušenější řešitelé se moho těmto číslům přiblížit, ale optimálního počtu pohybů nedosáhnou vždy.

Pohyby	**Počet případů**	**Distribuce**	**Kumulativní distribuce**
0	1	0.00%	<0.01%
1	15	0.01%	0.01%
2	158	0.08%	0.09%
3	1 394	0.73%	0.82%
4	9 809	5.16%	5.99%
5	46 381	24.40%	30.39%
6	97 254	51.16%	81.55%
7	34 966	18.40%	99.95%
8	102	0.05%	100%

Tabulka 5.1: Nutné pohyby pro složení kříže. Studie Larse Vandenbergha [15]

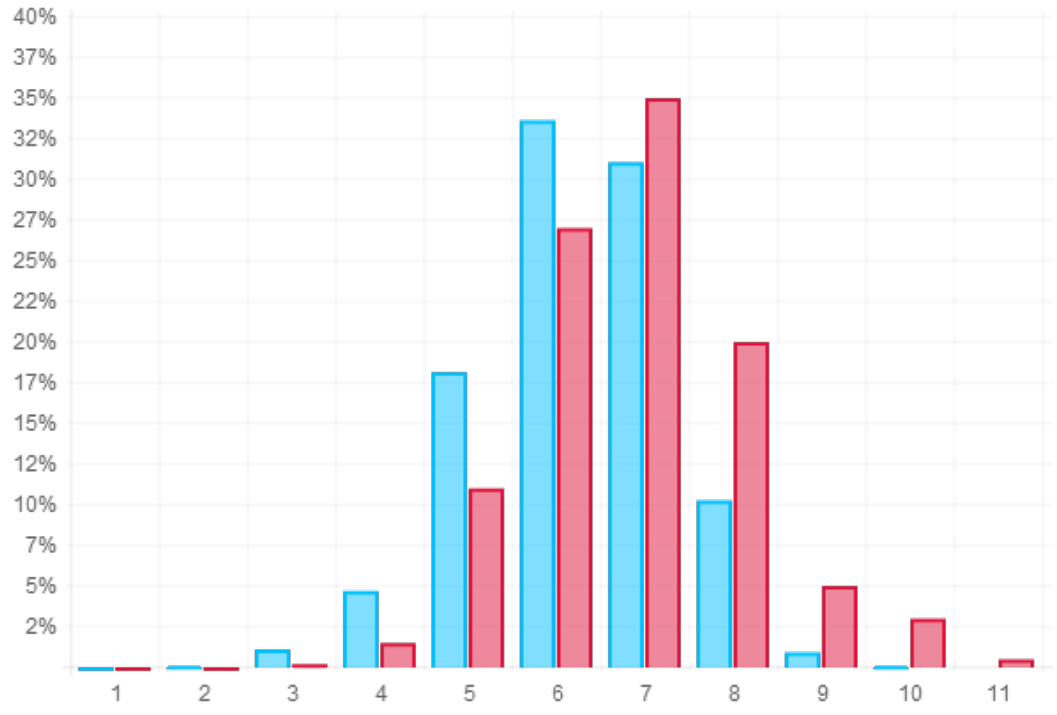
[1] Jejich průměrné časy se pohybují kolem 15-25 sekund.

5.1.1 Srovnání s implementací

Jako statistický vzorek našeho algoritmu jsme použili 10 000 složení kříže z náhodného stavu. Od lidských řešitelů máme informace pouze o 100 složeních (také z náhodného stavu). Pokud však převedeme údaje z histogramu na procenta, jsme schopni je relativně objektivně porovnat.

Na grafu 5.1 vidíme, že náš algoritmus, vyznačený modře, lehce zaostává za optimálním počtem pohybů. Výjimečně potřebuje na složení kříže i 9 nebo dokonce 10 pohybů, přičemž víme, že i tyto situace lze vyřešit maximálně 8 pohyby. Optimální algoritmus, který odpovídá tabulce 5.1 skládá kříž průměrně 5,81 pohyby, náš algoritmus má průměr zhruba 6,23 pohybů. Můžeme tedy vidět, že algoritmus, který imituje intuitivní myšlení lidského řešitele je jen o 7% horší než optimální algoritmus spočívající v mechanickém procházení stavového prostoru.

Údaje od lidských řešitelů můžeme na grafu vidět červeně. Průměrný počet pohybů na složení u nich byl 7,09. Distribuce jednotlivých pohybů je velmi podobná jako u naší impelmentace. Přestože jsou tato data od relativně zkušených řešitelů, jejich výsledky nedosahují optimálních hodnot. Několikrát složili kříž až na 10 - 11 pohybů. Je pravděpodobné, že u nejlepších světových řešitelů bychom se přiblížili optimálním počtům, nebo alespoň našemu algoritmu. Za zmínku stojí také to, že lidští řešitelé se soustředí zejména na rychlost, a proto mohou schválně vybrat delší postup, který je jednodušší na provedení. Tak mohou docílit lepšího času než kdyby použili optimální způsob.



Obrázek 5.1: Histogram pohybů nutných na složení kříže, modrá - naše implementace, červená - lidští řešitelé.

5.2 F2L

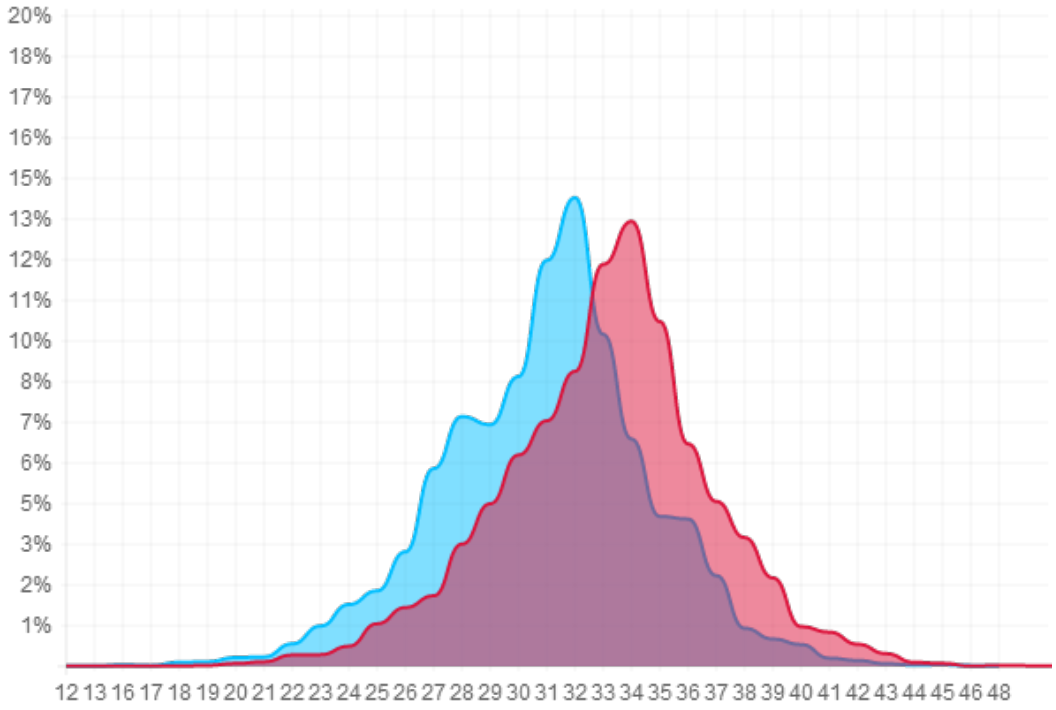
U řešení prvních dvou vrstev budeme opět porovnávat počet pohybů nutných na dokončení fáze. Rychlostní řešitelé ale jednají impulzivně a rychlé dokončení bloku je pro ně důležitější než zdlouhavé hledání optimálního postupu. Tuto část můžeme rozdělit na 4 podfáze, v každé se snažíme složit jen jeden blok. S každým složeným blokem se zvedá průměrný počet pohybů nutný na složení dalšího. Děje se to, protože pořadí, v kterém bloky skládáme není fixní, ale vždy vybíráme ten, který bude nejjednoduší na vyřešení. Pokud jsou již např. tři bloky složené, musíme složit ten poslední, i když je v nevýhodné situaci. Na tabulce 5.2 můžeme vidět, jak vypadají optimální počty pohybů při skládání jednotlivých bloků, když používáme tzv. *greedy order* (vybíráme vždy nejjednodušší blok).

F2L (greedy order)	
Pořadí	**Počet pohybů**
Blok 1	5.03
Blok 2	5.40
Blok 3	5.80
Blok 4	6.81
Dohromady	28.85

Tabulka 5.2: Optimální počet pohybů nutných pro složení bloků. [10]

5.2.1 Srovnání s implementací

Opět porovnáváme percentuální distribuci nutných pohybů naší implementace s lidskými řešiteli. Je nutné podoknout, že ani naše implementace nevyhledává absolutně optimální blok, ale pracuje na bázi jednoduchých pravidel (například „obě části leží v horní vrstvě“), které imitují lidský způsob řešení. Nekontrolujeme v tabulce, kolik je pro danou situaci nutných pohybů. Stává se tak, že dáme přednost situaci, která na první pohled vypadá jednoduše, ale její řešení není počtem pohybů optimální. Náš algoritmus dosáhl při 10 000 složeních průměru 30,99 pohybů. To je o 7,4% více než při optimálním řešení. Lidští řešitelé potřebovali při 100 složeních v průměru 33,42 pohybů. To je nárůst zhruba 8% oproti našemu řešení a skoro 16% oproti optimálnímu. Přestože jsme se snažili algortimicky napodobit lidské přemýšlení při řešení F2L, z výsledků vyplývá, že průměr je zvýšen zejména díky lidské chybě. Na následujícím grafu 5.2 můžeme vidět rozdíl mezi naší implementaci a řešiteli. Zajímavý je pokles v distribuci zhruba u 29 pohybů. To by mohlo být způsobeno tím, že počty pohybů, potřebných pro složení jednotlivých bloků zřídka dávají v součtu číslo 29. Tento pokles se u řešitelů nevyskytuje a graf se více přibližuje normálnímu rozložení. Graf obsahuje pouze diskrétní hodnoty, které jsou spojeny křivkami pouze kvůli přehlednosti.



Obrázek 5.2: Histogram pohybů nutných na složení F2L, modrá - naše implementace, červená - lidští řešitelé.

Kapitola 6

Závěr

Cílem práce bylo implementovat aplikaci, která by měla skládat Rubikovu kostku lidskými metodami, alespoň na jejich úrovni. Ze statistik vidíme, že velmi pokročilé řešitele dokonce v obou kategoriích předčila. Můžeme tedy říct, že náš algoritmus pracuje na úrovni nejlepších lidských řešitelů.

Neznamená to však, že neexistuje prostor pro zlepšení. Na sestavení kříže je výjimečně potřeba více než matematicky dokázaný maximální počet nutných pohybů. To se stává ve specifických situacích, na které není algoritmus připraven. Člověk by dokázal v této situaci improvizovat a zkrátit počet pohybů nestandardním způsobem. Implementace této funkcionality by dále zoptimalizovala řešení kříže. Podobně by šlo vylepšit fázi F2L. Zde by bylo možné značně zkrátit řešení, ale zase by to neodpovídalo rychlostnímu řešiteli, což je hlavní náplní této práce. Fáze OLL a PLL pracují podle očekávání optimálně.

Výstupem bakalářské práce je webová aplikace, která dokáže velmi věrohodně imitovat lidské postupy při řešení kostky. Na 3D modelu kostky si mohou uživatelé vyzkoušet různé situace v praxi. Pro začátečníky může sloužit jako výukový software a pokročilí řešitele mohou porovnat svá řešení s naším algoritmem a trénovat s ním.

Při zpracovávání tohoto úkolu jsem prohloubil své znalosti jak v oblastech týkajících se Rubikovy kostky, tak zejména v odvětví umělé inteligence. Vzhledem k tomu, že aplikace předčila původní očekávání, dá se výsledek práce považovat za úspěch.

Literatura

[1] Akkersdijk, E.: F2L cases. [online] 2006-11-06 [cit. 2014-04-11]. Dostupné z: http://erikku.er.funpic.org/rubik/F2L.html.

[2] Bellis, M.: The History of Rubik's Cube. [online] [cit. 2014-03-27]. Dostupné z: http://inventors.about.com/od/rstartinventions/a/Rubik_Cube.htm.

[3] Fridrich, J.: My system for solving Rubik's cube. [online] 1996 [cit. 2014-04-11]. Dostupné z: http://www.ws.binghamton.edu/fridrich/system.html.

[4] Kociemba, H.: The Two-Phase-Algorithm. [online] 2010-07-01 [cit. 2014-04-11]. Dostupné z: http://kociemba.org/cube.htm.

[5] Kociemba, H.; Rokicki, T.; Davidson, M.; aj.: The Diameter of the Rubik's Cube Group Is Twenty. In: *SIAM Journal on Discrete Mathematics (Volume 27, Issue 2)*. [online] 2013-06-19 [cit. 2014-04-11]. Dostupné z: http://tomas.rokicki.com/rubik20.pdf.

[6] Speedsolving Wiki: OLL. [online] 2014-02-23 [cit. 2014-04-22]. Dostupné z: http://www.speedsolving.com/wiki/index.php/OLL.

[7] Speedsolving Wiki: PLL. [online] 2014-05-16 [cit. 2014-05-16]. Dostupné z: http://www.speedsolving.com/wiki/index.php/PLL.

[8] Obítko, M.: Prohledávání stavového prostoru. [online] 2004 [cit. 2014-03-27]. Dostupné z: http://labe.felk.cvut.cz/~obitko/xkui/materialy/prostor.pdf.

[9] Petrus, L.: Solving Rubik's Cube for speed. [online] 1997-03-09 [cit. 2014-04-11]. Dostupné z: http://lar5.com/cube/.

[10] Pochmann, S.: Analyzing Human Solving Methods for Rubiks Cube and similar Puzzles. [online] 2008-03-29 [cit. 2014-05-15]. Dostupné z: http://www.stefan-pochmann.info/hume/hume_diploma_thesis.pdf.

[11] Reid, M.: Michael Reid's Rubik's Cube page M-symmetric positions. [online] 1995 [cit. 2014-03-29]. Dostupné z: http://math.cos.ucf.edu//~reid/Rubik/m_symmetric.html.

[12] Rider, C.: ZZ method tutorial. [online] 2013-11-27 [cit. 2014-04-04]. Dostupné z: http://cube.crider.co.uk/.

[13] Roux, G.: Roux method introduction. [online] [cit. 2014-04-11]. Dostupné z: http://grrroux.free.fr/method/Intro.html.

[14] Scherphuis, J.: Thistlethwaite's 52-move algorithm. [online] [cit. 2014-03-27]. Dostupné z: http://www.jaapsch.net/puzzles/thistle.htm.

[15] Vanderbergh, L.: Cross study. [online] 2008-03-29 [cit. 2014-05-05]. Dostupné z: http://www.cubezone.be/crossstudy.html.

[16] WCA: World Cube Association Official Results. [online] 2014-05-15 [cit. 2014-05-15]. Dostupné z: https://www.worldcubeassociation.org/results/events.php?eventId=333&regionId=&years=&show=100%2BPersons&single=Single.

Seznam převzatých obrázků

odkaz 1 http://www.onereason.org/wp-content/uploads/2012/04/Rubiks_cube_blind.jpg

odkaz 2 http://upload.wikimedia.org/wikipedia/commons/f/fa/Disassembled-rubix-1.jpg

odkaz 3 http://rubikscubesolution.org/wp-content/uploads/2013/10/rubiks_cube_solution_layer_by_layer_0.png

odkaz 4 http://www.chowkafat.net/Picture/rubik51.jpg

odkaz 5 http://kociemba.org/pics/opt1000000.gif

odkaz 6 http://www.speedsolving.com/wiki/images/f/f9/Fridrich_method.gif

odkaz 7 http://www.speedsolving.com/wiki/images/2/2a/Roux_method.gif

odkaz 8 http://www.speedsolving.com/wiki/images/e/e7/Petrus_method.gif

odkaz 9 http://www.speedsolving.com/wiki/images/4/45/Eoline.gif

odkaz 10 http://cubercritic.com/wp-content/uploads/2013/05/3x3-Notation.jpg

odkaz 11 http://www.speedsolving.com/wiki/images/c/c0/F2L10.png

odkaz 12 http://www.speedsolving.com/wiki/images/4/4a/F2L37.png

odkaz 13 http://cube.crider.co.uk/visualcube.php?fmt=gif&size=150&co=100&fo=100&r=y60x330&fd=uuuuuuuuuooorrrrrrooofffffddddddddooollllllooobbbbbb

odkaz 14 http://www.speedsolving.com/wiki/images/f/fb/J1.gif

Dodatek A

Obsah CD

bp-xkocva02.pdf - Písemná zpráva práce (tento dokument).

doc - Zdrojový kód dokumentace v LaTeX.

lib - Všechny knihovny potřebné k vytvoření kostky.

lib/solver - Soubory určené pro řešení kostky.

lib/plugins - Obsahuje knihovny JQuery a Chart.js, které byly v kódu použity.

script.js - Chování samotné aplikace.

style.css - Vzhled aplikace.

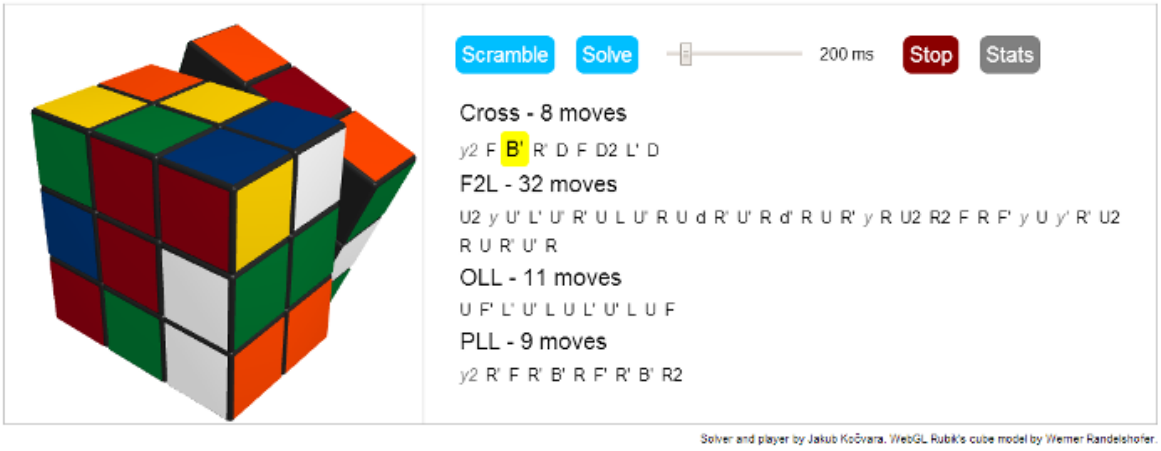
index.html - Hlavní HTML soubor.

Dodatek B

Manuál

Pro instalaci je nutné soubory umístit na běžící HTTP server. Je to nutné kvůli načítání souborů pomocí AJAX (vznikla by chyba Cross-Origin Request). Klient musí mít prohlížeč Chrome nebo Firefox ve verzi, která podporuje WebGL a zapnutý JavaScript.

B.1 Popis funkcionality módu řešení



Obrázek B.1: Aplikace v módu řešení.

Scramble Rozloží kostku do náhodného stavu.

Solve Vypočítá pohyby potřebné ke složení a spustí jejich přehrávání na virtuální kostce.

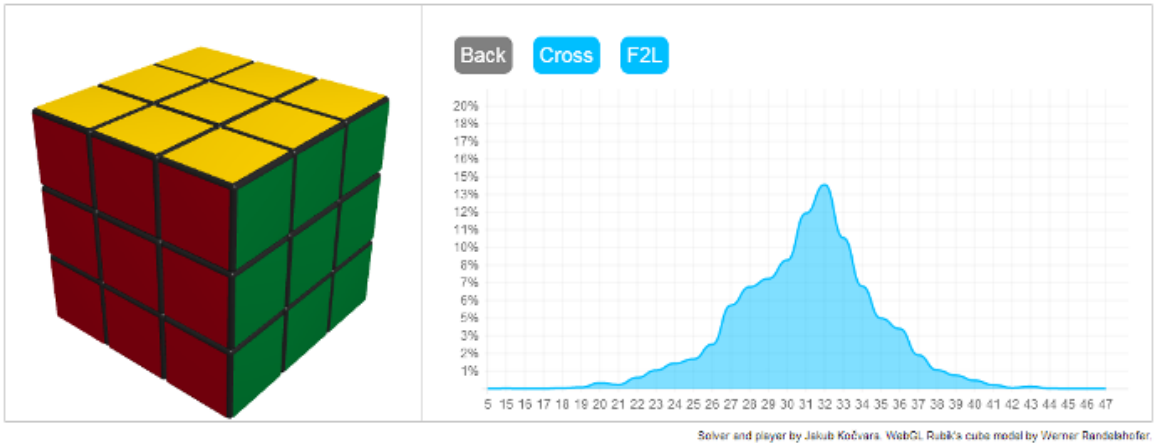
Slider Určuje, jak dlouho bude trvat jedno otočení kostkou v milisekundách. Pokud chceme zpomalit a pozorovat průběh, můžeme nastavit až na 1 pohyb/sekundu.

Play/Stop Spouští/zastavuje přehrávání.

Stats Přechod do módu statistik.

Virtuální kostka je plně interaktivní. V pravé části můžeme vidět pohyby nutné ke složení kostky zapsané oficiální notací. Právě prováděný pohyb je zvýrazněn žlutě. Pokud klikneme na nadpis fáze, můžeme se vrátit na její začátek a znovu si ji spustit.

B.2 Popis funkcionality módu statistika



Obrázek B.2: Aplikace v módu statistiky.

Cross Spustí test algoritmu pro fázi cross. Zobrazí výsledná data jako sloupcový graf.

F2L Spustí test algoritmu pro fázi F2L. Zobrazí výsledná data jako liniový graf.

Pro každý test se vždy spustí 10 000 složení a zobrazí se histogram, na kterém je na ose x zobrazen počet pohybů a na ose y percentuální výskyt tohoto počtu.