Univerzit Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Michal Penkala

Neoptimální řešení permutačních hlavolamů rozkladem na podproblémy

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Vladan Majerech, Dr.

Studijní program: informatika, programování

2006

Poděkování

Děkuji svému vedoucímu Mgr. Vladanu Majerechovi za vedení a podnětné připomínky, a to zejména při hledání a implementaci vhodných algoritmů pro řešení permutačních hlavolamů a při psaní dokumentace a bakalářské práce.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 11.8.2006 Michal Penkala

Obsah

Název práce: *Neoptimální řešení permutačních hlavolamů rozkladem na podproblémy*
Autor: *Michal Penkala*
Katedra (ústav): *Katedra teoretické informatiky a matematické logiky*
Vedoucí bakalářské práce: *Mgr. Vladan Majerech, Dr.*
E-mail vedoucího: *maj@ktilinux.ms.mff.cuni.cz*

Abstrakt: V předložené práci studuji vlastnosti permutačních hlavolamů a hledám algoritmy použitelné k řešení těchto hlavolamů. Úlohou práce je implementovat algoritmus neoptimálního řešení permutačních hlavolamů rozkladem na podproblémy a navrhnout vhodný formát definice hlavolamu. Výsledkem práce je program, který umožní uživateli pomocí grafického návrhu vytvořit libovolný permutační hlavolam. Na tomto hlavolamu potom bude moci uživatel provádět nadefinované tahy hlavolamu, zadávat libovolnou aktuální pozici a také hledat řešení aktuální pozice.

Klíčová slova: hlavolam, permutace, neoptimální řešení

Title: *Non-optimal solution of permutation puzzles by decomposition to sub problems*
Author: *Michal Penkala*
Department: *Department of Theoretical Computer Science and Mathematical Logic*
Supervisor: *Mgr. Vladan Majerech, Dr.*
Supervisor's e-mail address: *maj@ktilinux.ms.mff.cuni.cz*

Abstract: In the present work I study the attributes of permutation puzzles and try to find the algorithms usable for solving these puzzles. The task of this project is to implement the algorithm for non-optimal solution of permutation puzzles by decomposition to sub problems and invent a suitable form of puzzle definition. The result of this project is a program with graphic interface, which allows the user to create custom permutation puzzle. With this puzzle, the user will be able to do the predefined moves, make custom positions and search the result of the position.

Keywords: puzzle, permutation, non-optimal solution

Kapitola 1 - Úvod

1.1 Zadání

Cílem práce je program, který z definice permutačního hlavolamu vytvoří algoritmus, řešící daný hlavolam z libovolné počáteční pozice (případně rozhodne, že hlavolam nemá řešení). Program by si měl poradit i s hlavolamy s nerozlišitelnými díly, i s blokačními hlavolamy (hlavolamy, kde je možnost provést pohyb vázaná na aktuální permutaci dílů).

Vzhledem k počtu konfigurací netriviálních hlavolamů není šance řešit úlohu na grafu všech konfigurací. Typicky je ale možno díly permutačního hlavolamu rozdělit na vzájemně se neovlivňující skupiny.

Práce má navrhnout vhodný formát definice hlavolamu.

1.2 Cíl

Cílem mojí práce bude vytvořit aplikaci, která bude umožňovat uživateli zadat nebo vytvořit libovolný permutační hlavolam a umožňovat uživateli pracovat s tímto hlavolamem. Prací s hlavolamem se rozumí možnost provádět zadané tahy hlavolamu, zadávat libovolnou možnou pozici hlavolamu, možnost hledat řešení dané pozice hlavolamu a také schopnost zjišťovat určitých charakteristických vlastností, které nemusí být na první pohled zřejmé.

Celá aplikace by měla být v podobě uživatelsky příjemného grafického prostředí. Tvorba hlavolamu by měla probíhat způsobem grafického návrhu hlavolamu, ne pouze definováním charakteristických permutací hlavolamu, které mohou být pro obyčejného uživatele (který nemusí o permutacích nic vědět) překážkou. Taktéž celá následná práce s hlavolamem by měla být formou interaktivního grafického zobrazení hlavolamu.

Stěžejním cílem programu bude samozřejmě implementace algoritmu řešení, pokud možno co nejobecnějších hlavolamů. Bude ovšem také snaha vytvořit program, který se

bude zabývat také grafickou stránkou hlavolamu a způsobem konverze této grafické podoby do podoby vhodné pro řešení našeho hlavního cíle (tedy do podoby permutací), z důvodů usnadnění a zpříjemnění uživateli práci s programem.

1.3 Motivace

Motivace ke vzniku této bakalářské práce vychází ze spojení potřeb obyčejného uživatele (permutačních) hlavolamů a z potřeby zabývat se implementací matematického problému řešení obecných permutačních hlavolamů.

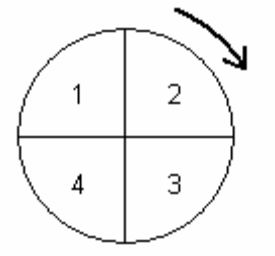
Téměř každý z nás už měl v ruce nějaký permutační hlavolam. Zamíchat hlavolam je velice jednoduché, ovšem složit ho už tak jednoduché být nemusí. Často se stává, že se uživateli podaří složit určitou část hlavolamu, ale ne a ne poskládat celý hlavolam. Proto je potřeba uživateli pomoct. Na internetu lze nalézt mnoho návodů na složení mnoha hlavolamů, ale ne vždy se musí podařit nalézt návod na konkrétní uživatelův hlavolam, a navíc hledá-li reprezentaci hlavolamu v podobě programu. Vzniká tedy myšlenka vytvořit nějaký jeden, pokud možno univerzální program, ve kterém by se dal daný hlavolam vytvořit a následně vyřešit.

Z hlediska řešení obecných permutačních hlavolamů existuje několik způsobů jak daný problém řešit (viz. další kapitola), ale v programech, které se zabývají řešením určitých hlavolamů, se objevují algoritmy pro řešení jednoho konkrétního hlavolamu. Kdybychom si však vzali hlavolam, který se liší třeba jen nějakou malou drobností, nedal by se již vytvořený algoritmus použít a musel by se vytvořit nový, opět pro řešení jednoho konkrétního hlavolamu. Myšlenkou je tedy zase program, ve kterém se implementují algoritmy obecného řešení, na které nebude mít změna hlavolamu vliv.

Kapitola 2 - Permutační hlavolamy

2.1 Co je permutační hlavolam

Permutační hlavolam je takový hlavolam, kdy každý jeho stav a tahy na něm je možné vyjádřit jako permutaci – tedy je-li možné jeho grafickou podobu rozložit na jednotlivá políčka, která jsou v určitém pořadí očíslována. Vezměme například „hlavolam“ na *Obrázku 1*. Jeho základním stavem je permutace 1 2 3 4 a tah otočení o 90° doprava můžeme vyjádřit jako permutaci 4 1 2 3.

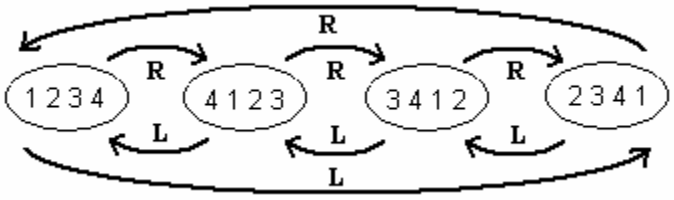


Obrázek 1 – Jednoduchý permutační „hlavolam“

Mezi nejznámější permutační hlavolamy patří například Rashkey nebo Rubikova kostka, které ve svém programu používám jako testovací příklady.

2.2 Obecné vs. konkrétní řešení

Obecně se dá řešení permutačního hlavolamu vyjádřit jako problém hledání nejkratší cesty v grafu (viz. *Obrázek 2* vycházející z hlavolamu na *Obrázku 1*, kde R je otočení doprava a L otočení doleva).



Obrázek 2 – Graf stavů permutačního hlavolamu

Použití obecných algoritmů pro řešení našeho problému však není možné. Už i méně složitý hlavolam má množinu vrcholů tak velkou, že při použití obecných algoritmů by byl díky časové složitosti program prakticky nepoužitelný.

U řešení konkrétních hlavolamů zase využíváme znalostí určitých vlastností, díky kterým si můžeme dovolit provést určité optimalizace, které hledání značně urychlí. Těchto optimalizací ale použít také nemůžeme, protože fungují třeba jen na tom jednom konkrétním hlavolamu.

2.3 Vlastnosti permutačních hlavolamů

Vezměme si typické přestavitele permutačních hlavolamů jako Rashkey nebo Rubikova kostka (viz. *Obrázek 3*). Už na první pohled je použití algoritmů pro hledání cesty v celém grafu permutací nemyslitelné. Například Rubikova kostka se svými 54 poli má asi $4{,}3 \cdot 10^{19}$ kombinací postavení. Je tedy potřeba hledat určité specifické vlastnosti permutačních hlavolamů, které by nám pomohly k efektivnějšímu hledání cesty.

Obrázek 3 – Rubikova kostka (vlevo), Rashkey (vpravo)

2.3.1 Obarvení

Za jednu z nejočividnějších vlastností hlavolamů se dá považovat jejich obarvení. Když se podíváme na naše hlavolamy, všimneme si skupin polí se stejnou barvou. Toho by se dalo využít, pokud nebudeme poskládaný hlavolam brát jako permutačně poskládaný, ale jako barevně poskládaný. Permutačně je hlavolam poskládaný, když $perm[i] = i$. Barevně je hlavolam poskládaný, když $barva(perm[i]) = barva(i)$. Budeme-li tedy brát jako poskládaný barevně poskládaný hlavolam, mohli bychom permutace vyjadřující stav hlavolamu upravit tak, že číslice v permutaci reprezentující políčka nahradíme číslem barvy daného políčka. Tím by se nám (na první pohled) mohla podstatně zmenšit množina stavů hlavolamu tak, že by bylo možné použít obecných algoritmů. Problém však nastává, kdybychom dostali hlavolam, kde každé políčko má svoji vlastní barvu. Množina stavů by zůstala stejná a byli bychom opět tam, kde jsme byli. Další problém vzniká třeba u Rubikovy kostky, kde kdybychom chtěli vyměnit třeba jedno rohové

políčko na libovolné stěně s jiným rohovým na stejné stěně, znamenalo by to výměnu celých rohových kostiček, a tedy i výměnu políček, které už stejnou barvu nemají. To znamená, že i když má Rubikova kostka více políček stejné barvy, pozice jednotlivých políček je určena jednoznačně, nezávisle na stejném obarvení, což pro člověka nemusí být hned samozřejmé.

Další vlastností obarvení, která může být pro člověka handicapem, může být malá rozlišovací schopnost barev. Při hledání „krátkých tahů“ není jednoduché vysledovat, jakou permutaci tah skutečně provádí, zatímco při řešení počítačem tento problém odpadá.

2.3.2 Soupolí

Z předchozího problému s Rubikovou kostkou nám vzniká další vlastnost permutačních hlavolamů a tou jsou „soupolí“. Soupolí je skupina políček, které tvoří jakýsi celek (např. u Rubikovy kostky roh), který se dá brát jako jedno pole. Políčka z této skupiny se pohybují závisle na sobě. Neexistuje tedy posloupnost základních tahů, která by dokázala tato políčka rozdělit. Kdybychom tuto skupinu políček nahradili jedním polem a považovali ho za obyčejné políčko, počet stavů by se opět zmenšil, ale to udělat nejde, protože je potřeba rozlišovat i stavy jednotlivých políček v soupolí. Tím dostaneme zase tu samou množinu stavů.

Soupolí se dá ale využít pro definování orientace u polí, které žádnou orientaci nemají. Pokud bychom chtěli například zaznamenat orientaci středových kostiček Rubikovy kostky, mohli bychom tento střed rozdělit na 4 políčka, které by vytvořili soupolí. Naopak, pokud bychom nechtěli rozlišovat orientace rohů Rubikovy kostky, mohli bychom každý z nich reprezentovat jedním políčkem.

2.3.3 Skupiny

Další vlastností, kterou bychom se měli zabývat jsou „skupiny polí“. Skupina polí je množina políček hlavolamu, které mohou v této skupině „cestovat“. Například u Rubikovy kostky tvoří jednu skupinu rohová políčka. Skupiny polí nám tedy tvoři disjunktní množiny polí, jejichž stav by se dal vyjádřit samostatnou permutací. Z toho se nám přímo nabízí řešit jednotlivé skupiny zvlášť. Problém vzniká, když máme poskládanou jednu skupinu a chceme skládat další skupinu. Skládáním druhé skupiny

bychom totiž mohli zase tu první rozhodit. Využití této vlastnosti si rozebereme podrobněji v kapitole 2.4.

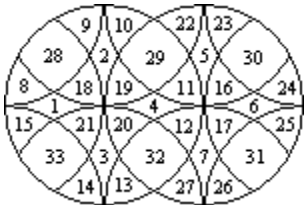
2.3.4 Inverzní tah

Každý tah permutačního hlavolamu má tah inverzní. Inverzní tah je tah, kdy složením libovolného tahu a inverzního tahu tohoto tahu dostáváme identitu. Každý inverzní tah k základnímu tahu hlavolamu lze také vyjádřit pomocí základních tahů. Vezměme si libovolný základní tah A permutačního hlavolamu. Inverzní tah A^{-1} pak můžeme vyjádřit jako tah B, který je také základním tahem hlavolamu (např. otočení stěnou Rubikovy kostky v protisměru tahu A), nebo jako násobek tahu A (např. 3x otočíme stěnou Rubikovy kostky v tom samém směru). Existence tahu B závisí na definici základních tahů. Násobek tahu A však existuje vždy. Tah A je totiž složen z jednoho nebo více cyklů. Zjištěním společného násobku délek těchto cyklů, dostaneme násobek, kterým je potřeba vynásobit tah A, abychom dostali identitu.

2.3.5 Blokace

Blokace je skupina políček, které nelze rozdělit tahem, i když by to normálně šlo. Když si vezmeme definici soupolí, uvidíme, že blokace je vlastně takové explicitní soupolí. Naopak soupolí by se dalo definovat jako blokace, akorát by neexistoval tah, který by mohl případně jednotlivé políčka soupolí rozdělit.

Blokační hlavolam je permutační hlavolam, který obsahuje blokace. Zkusme si blokační hlavolam vytvořit z hlavolamu Rashkey (viz. *Obrázek 4*).



Obrázek 4 – Rashkey (očíslovaná pole)

Tahy Rashkey jsou otáčení jeho kružnicemi o 90° libovolným směrem. Když na Rashkey zavedeme blokaci – sloučíme políčka 29 a 22 jako jedno políčko, nelze v tomto postavení otáčet levou kružnicí hlavolamu. Získali jsme tedy blokační hlavolam. Nejznámějším příkladem blokačního hlavolamu je Cube21 (viz. *Obrázek 5*).

Obrázek 5 – Cube21

2.3.6 Parita

Poslední takovou vlastností, která stojí za zmínku je parita hlavolamu. Parita hlavolamu je vlastně seznam parit permutací jednotlivých skupin. Parita hlavolamu nám k hledání řešení asi moc nepomůže, ale dá se použít ke kontrole řešitelnosti dané pozice, což se nám bude hodit při zadávání libovolné počáteční pozice hlavolamu. Hlavolam v základním postavení má nulovou paritu (všechny parity permutací skupin jsou nulové). Každý základní tah má svoji paritu, a ty se mohou navzájem lišit. Zapíšeme-li tyto parity jako binární rozvoj, můžeme operací XOR zjišťovat výsledné parity více tahů, tedy můžeme snadno vygenerovat množinu možných parit hlavolamu. Když potom budeme zadávat nové postavení hlavolamu, stačí zjistit paritu nového postavení a zkontrolovat jestli se nachází v množině možných parit hlavolamu. Pokud ne, nové postavení nemá řešení.

2.4 Metody „Rozděl a panuj“

Jak již bylo v předchozích kapitolách řečeno, díky velké množině stavů permutačních hlavolamů není možné řešit problém hledáním v celém grafu všech stavů. Z toho vyplývá, že problém je potřeba určitým způsobem rozdělit na podúlohy a ty řešit jednotlivě.

Algoritmus 1

V minulé kapitole byla zmiňována vlastnost permutačních hlavolamů, dělení polí do skupin. Tato vlastnost se jeví jako nejpřirozenějším způsobem, jak dosáhnout našeho cíle. Jak již také bylo zmíněno, nastává problém při řešení jedné skupiny, kde hledané řešení může narušit ostatní, již poskládané skupiny. Při řešení první skupiny (ostatní skupiny jsou zamíchané) lze bez problémů použít libovolných základních tahů hlavolamu. Řešení druhé skupiny je však potřeba hledat tak, aby zachovalo první skupinu stále složenou. Trochu obecněji, budeme hledat řešení i-té skupiny takové, aby neovlivnilo skupiny s menšími indexy. Pro toto řešení budeme muset tedy pomocí základních tahů najít tahy neovlivňující předchozí skupiny a přitom budou na i-té skupině provádět nějakou (pokud možno malou) permutaci.

Tady vznikají dva způsoby řešení i-té skupiny. Jedním ze způsobů je vygenerovat více tahů takových, že budou generovat stejnou množinu permutací i-té skupiny jako základní tahy. Nastává ale problém, jak kontrolovat, zda je množina stejná. Ještě bychom se mohli pokusit vygenerovat tolik tahů, že by pokryly každou permutaci i-té skupiny. To by ale nejspíš vedlo k velké paměťové i časové složitosti. Dalším problémem by mohlo být prohledávání pomocí velikého počtu vygenerovaných tahů, což by bylo docela neefektivní.

Dalším způsobem je vygenerovat jednu nebo více trojpermutací (popř. dvojici transpozic) na i-té skupině. Potom bychom byli schopni provádět tahy typu ABA^{-1}, kde A je posloupnost základních tahů hlavolamu, B je naše vygenerovaná trojpermutace ovlivňující pouze i-tou skupinu nebo skupiny s vyšším indexem a A^{-1} je inverzní tah k tahu A. Tímto tahem bychom byli totiž schopni vygenerovat libovolnou trojpermutaci na i-té skupině. Když máme možnost provádět libovolnou trojpermutaci na i-té skupině, tak stačí jenom určit pořadí a způsob výběru trojic políček, na kterých tyto trojpermutace provedeme tak, abychom se dopracovali řešení skupiny.

Algoritmus 2

Další metoda, která funguje na principu „rozděl a panuj“ (ThistlethWait), využívá vlastností obarvení hlavolamu. Pracuje tak, že definuje posloupnost zjemnění obarvení plošek hlavolamu v závislosti na množině základních pohybů. Při prvním obarvení mají plošky všechny plošky v jednotlivých skupinách stejnou barvu. Základní tahy tedy nemají na hlavolam žádný vliv a hlavolam se stále jeví jako poskládaný. Další zjemnění

obarvení získáváme omezením rozsahu jednoho nebo více základních tahů. Například, jsou-li délky všech cyklů permutace tahu sudé, můžeme vytvořit jeho modifikaci dvojnásobnou aplikací základního tahu. Na ukázku si můžeme představit otočení stěny na Rubikově kostce o 90°. Aplikací příkladu, který jsem uváděl, by se dalo se stěnou pohybovat už pouze 180°. Toto další obarvení bude tedy takové, že hlavolam bude vypadat stále složený i při aplikaci modifikovaných pohybů základních tahů. Postupným zhrubováním základních pohybů získáváme jemnější a jemnější obarvení. Ve chvíli, kdy již nemáme jak základní tahy zhrubovat (tady dalším omezením tahů bychom už vlastně neměli čím pohybovat), máme definované předposlední obarvení hlavolamu. Posledním obarvením hlavolamu je standardní počáteční obarvení hlavolamu. Otázkou je jenom, výběr zhrubování jednotlivých základních tahů.

Když máme definovanou posloupnost zjemnění obarvení U_1 až U_n, můžeme začít řešit. Nejdříve najdeme obarvení U_i, na kterém je hlavolam složený (většinou U_1). Hlavolam pak vnímáme jako hlavolam U_i, jehož cílem je složit pomocí tahů úrovně U_i hlavolam podle barev U_{i+1}. Vyřešením úrovně U_i jsme zajistili správný rozklad polí do skupin. Dále je ještě potřeba zkontrolovat správnost parit a rozkladu polí do soupolí vůči úrovni U_{i+1}, aby byl hlavolam řešitelný i v dalších úrovních.

Kapitola 3 - Návrh programu

3.1 Struktura programu

Ze zadání a z cíle, který jsem definoval v 1. kapitole vyplývá, že program by se měl skládat z hlavní části, ve které bude probíhat řešení hlavolamu, a z části, ve které se bude daný hlavolam definovat (čili vytvářet). První část si pojmenujeme jako **část řešení** a druhou část jako **část návrhu**.

Část návrhu je trochu obsáhlejší. Abychom mohli definovat hlavolam jak po grafické stránce, tak po funkční, budeme muset tuto část ještě dělit. Rozhodl jsem se návrh rozdělit na 5 podčásti:

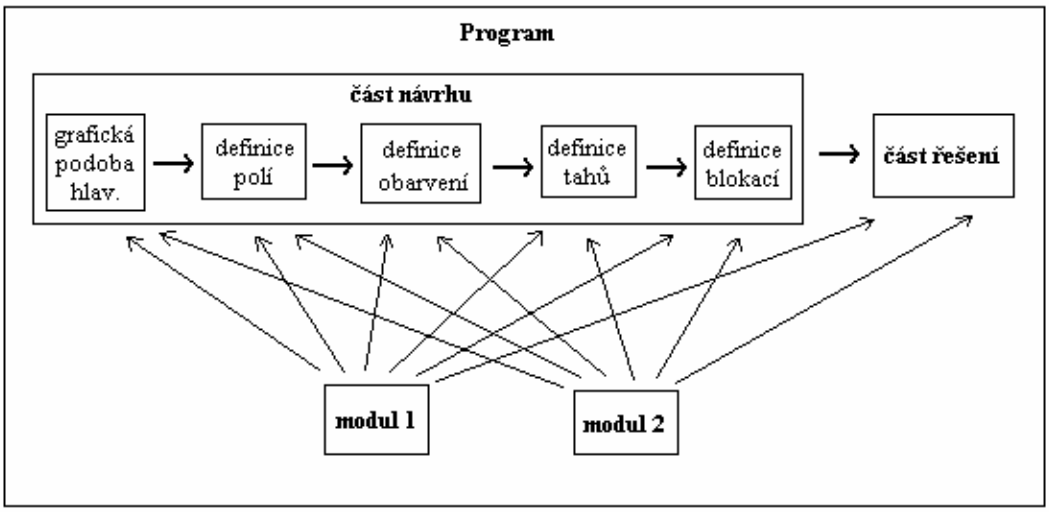
- Návrh grafické podoby hlavolamu
- Definování polí
- Definování obarvení
- Definování tahů
- Definování blokací

Uživatel by měl tedy nejdříve pomocí geometrických obrazců či jiných nástrojů vytvořit grafickou podobu hlavolamu. Dále by měl na této grafické podobě nadefinovat, co budou pole hlavolamu (např. jednotlivé čtvercové plošky na Rubikově kostce). Na těchto polích se nadefinují barvy plošek v základním postavení hlavolamu. Dále je potřeba definovat tahy, čili definovat pohyby jednotlivých plošek pro každý tah. Pak už jen zbývá definice blokací, tedy určit seznam plošek pro každou blokaci.

Část řešení se zdá být už nedělitelná. Na vstupu dostane hlavolam definovaný v části návrhu, na kterém bude hledat řešení či provádět jiné požadované operace. Přitom by měl používat grafickou podobu hlavolamu pro interaktivní zobrazování probíhaných prací s hlavolamem. Část řešení si představuji tak, že uprostřed obrazovky bude vykreslován daný hlavolam. Na boku by se měl nacházet panel, který bude nabízet funkce, kterými bude možné zobrazovaný hlavolam ovládat. Určitě by tam měly být nabízeny tahy hlavolamu (třeba v podobě tlačítek), možnost řešit hlavolam a zobrazit toto řešení (popřípadě možnost provést toto řešení), a také možnost zadat aktuální pozici (buď pomocí permutace nebo pomocí definování obarvení, pokud to bude

realizovatelné). Další funkce nebyly u návrhu známy a jejich případné přidání bylo realizováno až při implementaci, dle případných potřeb.

Toto dělení ovšem není úplné. Když si vezmeme ohromné množství známých hlavolamů a budeme si všímat jejich geometrických rozmanitostí, je v této práci těžko realizovatelné vytvořit nástroj, kterým by se dalo univerzálně definovat takové množství různých tvarů (leda, že bychom chtěli vytvořit druhý AutoCad). Proto jsem se rozhodl dělit program ještě podle typu hlavolamů. Tedy pokusit se rozdělit obrovskou množinu hlavolamů na skupiny, ve kterých by se vyskytovaly hlavolamy se stejnými znaky (např. různé druhy Rubikových kostek jako 2x2, 3x3, 4x4,…). Je samozřejmé, že nebudu moci vytvořit tolik skupin, abych opět pokryl celou množinu možných hlavolamů. Toto dělení provádím proto, abych si usnadnil práci při návrhu grafické podoby hlavolamu, a potom i následně ve všech ostatních částech, které tohoto grafického návrhu budou používat. Jednotlivé části, na které budu tedy hlavolamy dělit nazvu **moduly**. Navrhovaná struktura programu by měla vypadat jako na *Obrázku 6*.



Obrázek 6 – Struktura navrhovaného programu

Kapitola 4 - Implementace

4.1 Vývojové nástroje

Jelikož jsem si zadání rozšířil tím, že aplikace by měla být formou uživatelsky přívětivého grafického prostředí, vhodné vývojové prostředí by mělo poskytovat rychlý grafický nástroj pro návrh oken a dialogů pro komunikaci s uživatelem. Jelikož nebyly požadavky, na kterých operačních systémech by měl program fungovat a výsledná aplikace bude typu oken, zvolím si tedy operační systém Microsoft Windows.

Z programovacích jazyků, které bych mohl použít k tvorbě aplikace, jsem si vybral C++ díky stabilně dobré programátorské základně a výkonu na Windows.

Z výběru operačního systému a zvoleného programovacího jazyku dostávám na výběr mezi dvěma vývojovými nástroji:

- Borland C++ Builder
- Microsoft Visual Studio 2003

Jelikož zadavatel neměl žádné požadavky na vývojové prostředí, tak jsem si zvolil prostředí, které mi nejvíc vyhovovalo a se kterým jsem již měl praktické zkušenosti, a tím je Borland C++ Builder.

4.2 Moduly

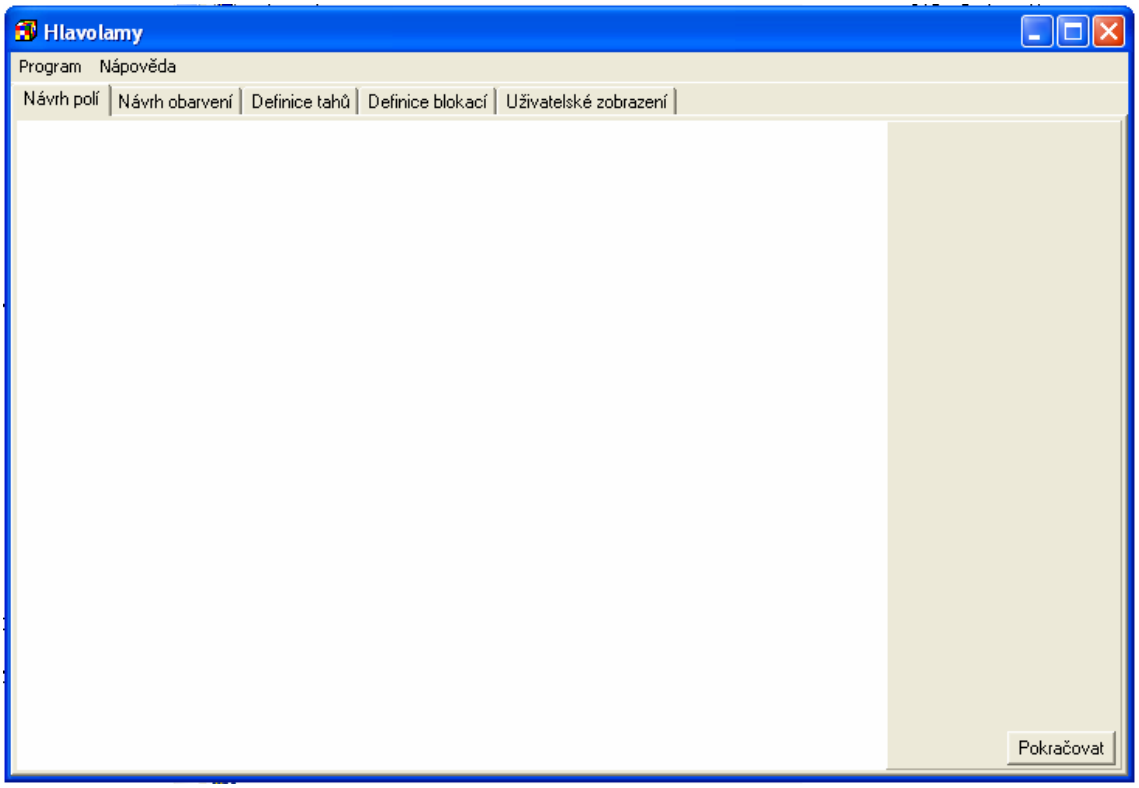
Při implementaci navržené struktury programu jsem se rozhodl, že jednotlivé moduly nebudou přímou součástí aplikace, ale budou ve formě DLL (dynamicky připojovaných knihoven). Kdykoli bude uživatel pracovat s nějakým hlavolamem, bude k programu připojena příslušná knihovna (modul) k danému typu hlavolamu, která se bude starat o grafickou reprezentaci hlavolamu a převod specifických dat daného hlavolamu na data pro jednotlivé části programu. Pro tyto knihovny je v kořenovém adresáři programu vytvořen adresář „Moduly“, ve kterém musí být uloženy. Nebude-li knihovna uložena v tomto adresáři, program o ní nebude vědět. V rámci vývoje programu jsem vytvořil 2 funkční moduly, jejichž funkce budou popsány v kapitolách 4.7.1 a 4.7.2. Implementace modulů jako knihovny má tu výhodu, že je možné zvlášť vytvořit nový modul, který bude zahrnovat skupinu hlavolamů, které nebylo doposud možné vytvářet pomocí

dosavadních modulů, bez jakéhokoli zásahu do hlavního programu, a stačí tuto vytvořenou knihovnu pouze nahrát do příslušného adresáře programu a program už si ji sám načte. Více o komunikaci modulu s programem bude probráno v kapitole 4.4.

4.3 Implementace struktury programu

V kapitole 4.2 jsem se zmínil, že moduly jsou implementovány mimo hlavní program, takže zbývá implementovat část návrhu a část řešení, dle navrhované struktury. Rozmýšlel jsem se zda nemám část návrhu vytvořit jako samostatný celek, ale celý projekt by pak byl moc rozkouskovaný. Nakonec jsem se rozhodl obě části sloučit do jedné aplikace a jednotlivé podčásti z části návrhu a část řešení jsem implementoval jako záložky (viz. *Obrázek 7*).

Aplikaci jsem pojmenoval jednoduše a příznačně: **Hlavolamy**.



Obrázek 7 – Program Hlavolamy

Při implementaci jsem zjistil, že podčásti *Grafická podoba hlavolamu* a *Definice polí* z návrhu struktury programu se dají díky funkčním vlastnostem modulů sloučit, popřípadě se dají sloučit podčásti *Definice polí* a *Definice obarvení*. Každý modul totiž

využívá určitých geometrických vlastností hlavolamu, takže už při samotném návrhu grafické podoby může modul rozpoznat jednotlivá pole hlavolamu.

Vzniklé části programu tedy jsou *Návrh polí*, *Návrh obarvení*, *Definice tahů*, *Definice blokací* a *Uživatelské zobrazení*. Každá část je tedy reprezentována svojí záložkou, kde se v pravé části každé záložky nachází jakýsi panel nástrojů specifický pro každou záložku.

Návrh polí

Záložka návrh polí odpovídá podčásti *Grafická podoba hlavolamu* z návrhu struktury programu. Tato část umožňuje navrhnout (nakreslit) grafickou podobu hlavolamu pomocí nástrojů na panelu nástrojů, které jsou závislé na použitém modulu hlavolamu.

Návrh obarvení

Záložka návrh obarvení umožňuje definovat obarvení jednotlivých polí hlavolamu. Pokud nejsou pole definovány v předchozí záložce (Návrh polí), budou pole definovány zároveň s obarvením daného pole. Na panelu nástrojů lze přidávat libovolné barvy, které je potom možno použít.

Definice tahů

Záložka definice tahů umožňuje na doposud vytvořeném hlavolamu definovat tahy hlavolamu použitím myši a tlačítek na panelu nástrojů.

Definice blokací

Záložka definice blokací by měla umožňovat definovat blokace hlavolamu. Bohužel v doposud vytvořené verzi programu není řešení hlavolamů s blokacemi implementováno, proto tato záložka není aktivní.

4.3.1 Uživatelské zobrazení

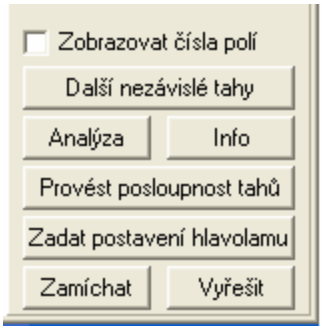
Uživatelské zobrazení je poslední záložkou, která reprezentuje celou část řešení z návrhu struktury programu. V této části se odehrávají hlavní pochody programu, proto si ji rozebereme trochu podrobněji.

V okně uživatelského zobrazení je již vykreslován plně navrhnutý hlavolam v závislosti na jeho aktuální permutaci (na počátku je samozřejmě v základním postavení). O kreslení hlavolamu se stará modul, program pouze volá příslušné funkce modulu (více o komunikaci programu s modulem proberu v kapitole 4.4).

V horní části panelu nástrojů jsou zobrazovány tlačítka reprezentující jednotlivé tahy hlavolamu (viz. *Obrázek 8*). V názvu tlačítek je obsaženo identifikační číslo tahu a název tahu, který byl nadefinován v záložce definice tahů. Stisknutím tlačítek se na hlavolamu provádí příslušné tahy a aktuální stav hlavolamu je automaticky překreslován.

Obrázek 8 – Tlačítka tahů hlavolamu

V dolní části panelu nástrojů se nachází skupina tlačítek (viz. *Obrázek 9*), jež umožňují práci s hlavolamem.



Obrázek 9 – Tlačítka pro práci s hlavolamem

Analýza

Tlačítko *Analýza* spouští dialog pro analýzu vstupních dat pro řešení hlavolamu. Dialog analýzy se spouští také automaticky, pokud nebyl hlavolam doposud analyzován. Bez analýzy vstupních dat není možné provádět na hlavolamu složitější operace, jako řešení aktuální pozice hlavolamu nebo zadávání libovolné aktuální pozice. V analýze jsou ze vstupních dat (z permutací, tvořících základní tahy hlavolamu) zjišťovány charakteristické vlastnosti hlavolamu jako skupiny, soupolí, a jsou také hledány trojpermutace, které jsou použity k řešení hlavolamu z aktuální pozice (více o těchto algoritmech je popsáno v kapitole 4.6).

Info

Tlačítko *Info* spouští dialog pro zobrazení charakteristických vlastností hlavolamu, které byly zjištěny v analýze. Tato funkce byla přidána kvůli umožnění uživateli dozvědět se nějaké informace o hlavolamu, které třeba nevěděl a mohly by mu pomoci pochopit, jak hlavolam funguje. Další funkcí je zobrazování výsledků některých algoritmů z důvodu detekce chyb.

Zamíchat

Tlačítko zamíchat otevírá dialog pro míchání hlavolamu. Jeho funkce je probrána v uživatelské dokumentaci. Tato funkce byla přidána, protože se ukázala jako uživatelsky nutná. Představa, že by si měl uživatel míchat kostku jednotlivými tahy, by ho odrazovala od používání programu.

Vyřešit

Tlačítko *Vyřešit* otevírá dialog pro řešení hlavolamu z aktuální pozice (algoritmus probrán v kapitole 4.6.6). Řešení je poté vypisováno do textového okna dialogu a je umožněno i jeho provedení na aktuální pozici hlavolamu.

Zadat postavení hlavolamu

Tlačítko *Zadat postavení hlavolamu* otevírá dialog pro zadávání libovolné pozice hlavolamu (algoritmus probrán v kapitole 4.6.5), buď pomocí permutace nebo graficky.

Provést posloupnost tahů

Tlačítko *Provést posloupnost tahů* otevírá dialog pro zadávání více tahů najednou a umožňuje několikanásobné provedení této posloupnosti. Tato funkce se ukázala jako dost užitečná, protože výsledky řešení a téměř všechny nezávislé tahy jsou zobrazovány pomocí těchto násobných posloupností. Kdybych chtěl takový tah provést ručně, musel bych třeba 15x provést tah 0 2 4 – což není zrovna uživatelsky příjemné a ani jednoduché neudělat chybu.

Další nezávislé tahy

Tlačítko *Další nezávislé tahy* otevírá dialog pro hledání nezávislých tahů pro jednotlivé skupiny (tahů, které neovlivňují ostatní skupiny). Dialog zobrazuje seznam

vyhledaných tahů a umožňuje jejich okamžité provedení na hlavolamu. Tato funkce byla přidána pro potřebu zkoušet hledat i jiné řešení, než jaké nabízí automatické řešení, přičemž znalost nezávislých tahů pro jednotlivé skupiny je velkou pomocí.

Zobrazovat čísla polí

Zaškrtávací pole *Zobrazovat čísla polí* umožňuje zapnutí a vypnutí zobrazování očíslování polí hlavolamu v základním postavení. Usnadňuje to pozorování a kontrolu pohybu jednotlivých tahů pomocí permutací.

4.4 Komunikace aplikace s moduly

Jak při návrhu hlavolamu, tak při samotné práci s hlavolamem je využíván modul příslušný k danému hlavolamu. Modul je tedy nahráván při otevírání již vytvořeného nebo vytváření nového hlavolamu. Pokud byl už před tím otevřen jiný modul, bude před otevřením nového uzavřen. Jelikož je modul formou DLL, komunikace probíhá prostřednictvím sdílených funkcí modulu, na které jsou po otevření modulu vytvořeny deskriptory. Existuje tedy pevný seznam funkcí, které by měla každá knihovna sdílet (tento seznam je vyjmenován v programátorské dokumentaci). Ale ne každý modul musí nabízet všechny z této sady funkcí. Každý modul totiž nese informaci o tom, které části programu jsou potřeba k práci s daným typem hlavolamu. Může se tedy stát, že některý modul nebude používat třeba část *Definice tahů*, jak je tomu u mnou vytvořeném modulu pro tvorbu Rubikových kostek (popsán v kapitole 4.7.2). Program tedy načte informace, které části programu se budou používat, nepoužívané části skryje a u používaných částí inicializuje deskriptory na příslušné funkce.

Každá část obsahuje sadu funkcí, volaných při událostech vyvolaných myší při práci na ploše dané části (funkčnost těchto funkcí se však v jednotlivých částech liší) a potom každá část obsahuje ještě svoje speciální funkce, které jsou potřebné pouze v dané části (např. při stisku nástroje v *Návrhu polí* se posílá modulu index stisknutého tlačítka, aby věděl, jak má reagovat třeba při pohybu myši na ploše).

Vezmu-li to z hlediska práce, kterou provádí modul a kterou program v jednotlivých částech programu, tak při návrhu hlavolamu je téměř veškerá činnost schována v modulech a v *Uživatelském zobrazení* zase pracuje spíš program. Při návrhu jsou všechny informace o hlavolamu uchovávány a zpracovávány v modulu. Program

pouze poskytuje takový univerzální interface mezi uživatelem a modulem. V *Uživatelském zobrazení* je tomu přesně naopak. Program si od modulu vyžádá základní charakteristické informace o hlavolamu, jako permutace tahů a obarvení, na těchto datech provede analýzu a výsledné informace už si ukládá k vlastnímu použití. Modul je volán pouze pro vykreslování aktuální pozice hlavolamu. Ještě akorát při zadávání aktuální pozice hlavolamu pomocí obarvení jsou průběžné informace uchovávány v modulu a při definování obarvení celého hlavolamu jsou data předány programu, který si je potom zpracuje.

4.5 Použité datové struktury

V celém projektu, už od grafického návrhu až po hledání řešení, jsou nejpoužívanější datovou strukturou seznamy. Jako nejvhodnějším řešením pro reprezentaci dat se ukázalo použití STL (Standard Template Library) kontejnerů, obzvláště datového kontejneru *Vector* a asociativního kontejneru *Map*. Knihovna bývá součástí každého C++ překladače, má slušnou programátorskou základnu a pro reprezentaci našich dat je jako šitá, tak proč si znepříjemňovat život tvorbou vlastních a nejspíš ne tak dokonalých struktur.

Vector je šablona jednorozměrného pole. Narozdíl od „klasického“ pole má mnoho užitečných vlastností a služeb. Lze do něj například pomocí různých metod vkládat prvek a tím i zvětšovat jeho velikost. Hodí se jak pro reprezentaci samotných permutací, tak i pro reprezentaci seznamů skupin, barev, atd.

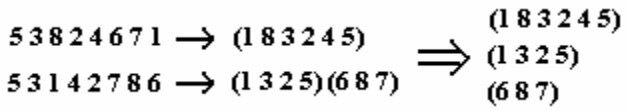
Map je asociativní pole, které nemusí být indexováno celočíselným typem, ale čímkoli. Hodí se pro rychlé vyhledávání podle neceločíselného klíče, například při hledání permutace reprezentované *Vectorem*.

4.6 Stěžejní algoritmy

4.6.1 Hledání skupin

Pro nalezení skupin je potřeba vytvořit množiny políček, které mohou mezi sebou cestovat. Vstupem pro můj algoritmus je seznam základních tahů hlavolamu ve formě

permutací. Nejdřív se tyto tahy převedou na cykly, a potom vytvořím seznam cyklů (viz. *Obrázek 10*).



Obrázek 10 – Převod tahů na seznam cyklů

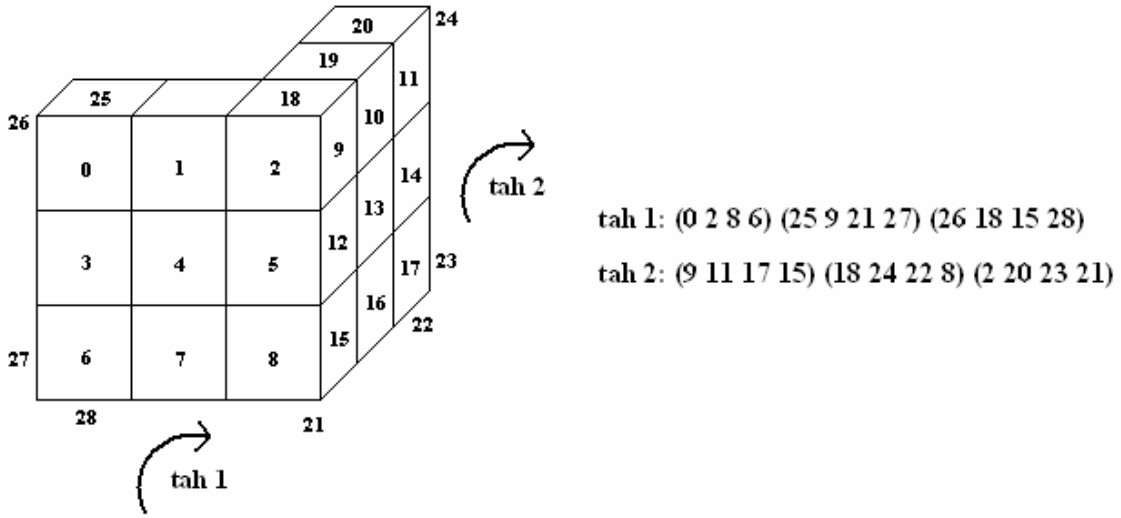
Mám tedy seznam cyklů C a vytvořím si ještě seznam skupin S (na počátku je prázdný). Postupně budu brát jednotlivé cykly C[i] a začleňovat je do seznamu skupin S. Začleňování cyklu C[i] bude probíhat tak, že jej budu porovnávat se všemi skupinami S[j]. Když bude existovat průnik C[i] a S[j], tak C[i] = sjednocení C[i] a S[j], a skupina S[j] bude ze seznamu vyjmuta. Až C[i] porovnám se všemi skupinami S[j], výsledné C[i] vložím do seznamu skupin S. Když tohle začlenění provedu se všemi C[i] ze seznamu cyklů C, výsledný seznam skupin S bude ten, který hledám. Algoritmus by se dal ještě optimalizovat tak, že při existenci průniku C[i] a S[j] porovnám, zda C[i] není podmnožinou S[j]. Pokud ano, je cyklus ve skupině obsažen a nemusíme pokračovat v porovnávání a přejdeme k dalšímu cyklu C[i+1]. Výsledný algoritmus bude vypadat asi takto:

```
S – seznam skupin (prázdná), C – seznam cyklů
for(int i = 0; i < C.size();i++)
{
  int j;
  for(j = 0; j < S.size();)
  {
    if(prunik(C[i], S[j]))
    {
      if(podmnozina(C[i], S[j]))
        break;
      else
      {
        C[i] = sjednoceni(C[i], S[j]);
        S.erase(j);
      }
    }
  }
  if(j == S.size())
    S.insert(C[i]);
}
```

4.6.2 Hledání soupolí

Najít soupolí znamená najít největší n-tice políček, se kterými budou tahy vždy hýbat jako s celky, ne pouze s částí této n-tice. Vstupem pro algoritmus jsou opět základní tahy hlavolamu a ještě seznam skupin hlavolamu. Při prvním vývoji algoritmu jsem uvažoval, že soupolí tvoří skupiny, tedy že soupolí mohou být tvořeny políčky z nějaké skupiny hlavolamu (tak, jak je tomu u Rubikovy kostky 3x3, z které jsem vycházel). Vyvíjel jsem tedy algoritmus pro hledání soupolí v určité skupině, a ten jsem pak prováděl na všech skupinách hlavolamu. Později jsem zjistil, že soupolí může výt tvořeno i dvěma (či více) skupinami (jako u Rubikovy kostky 4x4), kde v každé n-tici soupolí je obsaženo políčko z každé skupiny, na kterých jsou soupolí tvořena. Musel jsem tedy algoritmus upravit i na prohledávání soupolí ve více skupinách najednou.

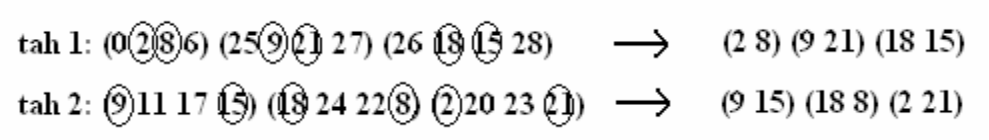
Nejdříve ukážu jak funguje hledání soupolí v jedné skupině. Pro představu můžeme použít Rubikovu kostku 3x3, kde hledáme rohy. Na počátku je potřeba převést základní tahy na cykly, které ale musí obsahovat pouze cykly skupiny, ve které hledáme. V první fázi algoritmu se kontroluje, zda je možné, aby soupolí na skupině existovaly. Je-li soupolí n-tice políček, tak všechny tahy, které se skupinou pohybují, musí být tvořeny n-ticemi cyklů o stejné velikosti (viz. *obrázek 11*).



Obrázek 11 – Cykly tahů ve skupině, tvořené rohovými políčky

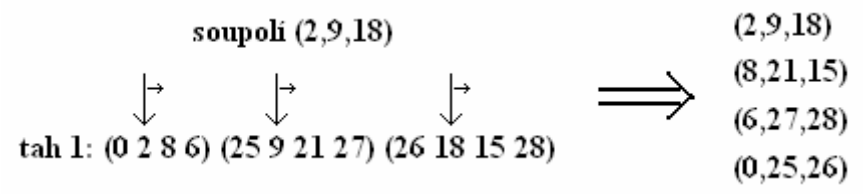
Pokud tedy najdeme $n > 1$ takové, že všechny tahy, které se skupinou pohybují, jsou tvořeny n-ticemi cyklů stejné velikosti, může skupina obsahovat soupolí a budeme tedy pokračovat druhou částí, která má za úkol zjistit jednotlivé n-tice polí. Nyní budeme brát dvojice tahů a pomocí jejich průniků zjišťovat hledané n-tice (v našem případě z *Obrázku 11*, trojice). Když narazíme na 2 tahy, které mají průniky polí a přitom tyto

průniky neobsahují celou množinu polí tahu, jsou to vhodní kandidáti pro nalezení soupolí. Nejdříve provedeme jejich průniky (viz. *Obrázek 12*).



Obrázek 12 – Průniky cyklů tahů

Když si vezmeme jeden z průniků (třeba ten první), tak možnými kandidáty na soupolí jsou skupiny polí (2,9,18) nebo (2,9,15) nebo (2,21,18) nebo (2,21,15). Když vezmeme druhý průnik a taky vybereme vhodné kandidáty (kteří budou obsahovat pole 2), dostaneme seznam (2,9,18), (2,9,8), (2,15,18), (2,15,8). Když průnikem těchto dvou množin možných soupolí bude pouze jediné soupolí, nalezli jsme skutečné soupolí hlavolamu, pomocí kterého nyní můžeme vygenerovat zbývající soupolí. A to tak, že vezmeme tah a zjistíme pozice políček soupolí v jednotlivých cyklech tahu. Tyto pozice budeme posouvat v cyklech, a čísla, na které budou pozice ukazovat nám budou tvořit další soupolí (viz. *Obrázek 13*).



Obrázek 13 – Generování ostatních soupolí

Tímto způsobem generuji soupolí i pro ostatní dvojice tahů, které mají průniky polí a zároveň kontroluji, zda jsem nevygeneroval soupolí, které by obsahovalo políčka z více již nalezených soupolí. To by byla chyba a hledání by bylo ukončeno s tím, že jsem nenalezl soupolí.

Pro hledání soupolí ve více skupinách akorát je potřeba převést tahy na cykly, tak aby byly obsaženy cykly ze všech hledaných skupin. Ještě před tím se ale kontroluje, zda mají skupiny stejný počet polí. Pokud ne, nemohly by obsahovat soupolí, které hledáme. Postup hledání soupolí je jinak stejný.

Při nalezení soupolí na více skupinách vyvstávají pro tyto skupiny zajímavé vlastnosti. Například, když složíme jednu ze skupin, ostatní skupiny budou také složeny (aniž bychom se o to snažili), čehož se dá využít při řešení hlavolamu. Takové to skupiny nazveme jako úplně závislé. Na druhou stranu ale díky těmto vlastnostem nastávají problémy při hledání nezávislých tahů.

4.6.3 Nezávislé tahy

Hledání nezávislých tahů probíhá tak, že procházím graf všech kombinací hlavolamu prohledáváním do hloubky po vrstvách. Jinak řečeno provádím všechny kombinace základních tahů hlavolamu. Kdykoli se dostanu do nějaké pozice, kontroluji, zda násobným provedením aktuálního tahu nevznikne nezávislý tah.

Kontrolu provádím tak, že si vezmu aktuální permutaci, do které jsem se prohledáváním zrovna dostal, převedu ji na cykly a zjistím společné násobky velikostí cyklů v jednotlivých skupinách. Například, budu mít 4 skupiny a společné násobky velikostí cyklů v jednotlivých skupinách mi vyjdou 2, 5, 3, 7. Chci-li vědět, jestli by se z aktuálního tahu dal vytvořit nezávislý tah pro skupinu X (pro náš příklad vezměme třeba $X = 1$, kde skupiny jsou číslované od 0), spočítám společný násobek společných násobků všech ostatních skupin N (tedy společný násobek 2, 3, 7 a $N = 42$), a zeptám se, zda je N dělitelné společným násobkem cyklů skupiny X (zda $(N \bmod 5) = 0$). Pokud není dělitelné, tak nezávislý tah pro skupinu X získám N-násobnou aplikací aktuálního tahu. Tuto kontrolu provádím pro všechny skupiny hlavolamu. Tento způsob vytváření nezávislých tahů se ukázal jako velice efektivní.

V minulé kapitole jsem se zmínil o problému, při hledání nezávislých tahů pro úplně závislé skupiny. Úplně závislé skupiny mají totiž vždy stejné společné násobky cyklů, což by v popsaném algoritmu způsobilo, že $(N \bmod 5)$ se bude vždy rovnat 0. Při výpočtu N je tedy potřeba vyjmout násobky úplně závislých skupin ke skupině X. Získáme tak nezávislé tahy, které budou řešit tyto závislé skupiny zároveň.

4.6.4 Hledání trojpermutací

Trojpermutace je permutace, která obsahuje jediný cyklus délky 3. Při hledání trojpermutací používám algoritmus pro nalezení nezávislých tahů v jednotlivých skupinách. Naleznu-li nějaký nezávislý tah pro skupinu, pro kterou jsem ještě žádnou trojpermutaci nenalezl, otestuji, zda je nalezený tah trojcyklus. Pokud není, pokusím se jej z tohoto tahu vygenerovat.

Pro generování trojcyklu požívám takový trik, kdy provedu tah typu $ABA^{-1}B^{-1}$, kde A je libovolný základní tah hlavolamu, B je nalezený nezávislý tah, A^{-1} je inverzní tah k tahu A a B^{-1} je inverzní tah k tahu B. Výsledným tahem je opět nezávislý tah, který bývá docela často menší, než nezávislý tah B. Porovnání menší tady znamená, že

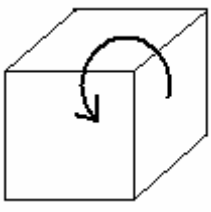
porovnáváme společné násobky velikostí cyklů permutací. Tento trik tedy provádím s naším původně nalezeným nezávislým tahem a postupně se všemi základními tahy hlavolamu, a testuji, jestli jsem náhodou nevygeneroval trojpermutaci. Tento trik by se dal samozřejmě použít rekurzivně i víckrát, ale při implementaci se ukázalo, že efektivnější je provádět ho na nezávislý tah pouze jednou.

Celé toto hledání trvá tak dlouho, dokud nejsou nalezeny trojpermutace pro všechny skupiny nebo pokud není hledání zastaveno uživatelem. Pro naše potřeby se dá místo trojcyklu také použít dvojice transpozic, proto při hledání trojpermutace kontroluji také dvojice transpozic. Ukázalo se však, že hledání řešení pomocí dvojic transpozic není tak efektivní jako pomocí trojcyklů, proto i když už jsem při hledání trojpermutací našel dvojici transpozic, snažím se najít i trojcyklus a transpozice jím nahradit. Dále je ještě potřeba kontrolovat, zda nehledám pro skupinu tvořenou soupolími. V tom případě nehledám trojcyklus políček, ale trojcyklus soupolí.

4.6.5 Zadávání postavení

Zadávání libovolné aktuální pozice hlavolamu je v programu implementováno dvěma způsoby. Buď přímo zadáním permutace hlavolamu nebo definováním obarvení hlavolamu.

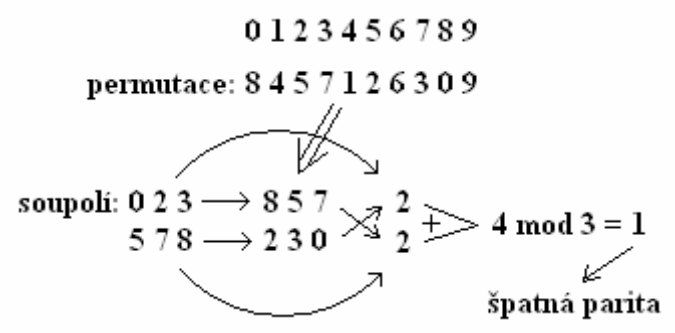
První způsob je pro kontrolu jednodušší. Nejdříve zkontroluji, zda je permutace korektně zapsaná a zkontroluji, zda jsou jednotlivá políčka ve správných skupinách. Potom použiji paritní kontrolu z kapitoly 2.3.6. Vygeneruji množinu možných parit hlavolamu a zkontroluji, zda se v ní nachází parita zadávaného postavení. U skupin které jsou tvořené soupolími však nepoužíváme paritu jednotlivých políček, ale paritu soupolí. To ale není nijak složité. U těchto skupin je však ještě potřeba kontrolovat paritu otočení jednotlivých soupolí (viz. *Obrázek 14*).



Obrázek 14 – Otočení soupolí

Z analýzy máme zjištěný seznam soupolí. Každé soupolí v tomto seznamu je tvořeno seznamem indexů políček hlavolamu, zapsaných v určitém pořadí. My si vždy

vezmeme jedno soupolí ze seznamu a indexy v tomto soupolí nahradíme indexy polí, které se v zadávané permutaci nacházejí na pozici našeho soupolí (viz. *Obrázek 15*).



Obrázek 15 - Zjištění parity soupolí

Tím dostaneme opět soupolí, jehož zápis však může být oproti zápisu v seznamu soupolí posunutý. Nás zajímá o kolik je tento seznam posunutý. Spočítáme-li tento rozdíl pro každé soupolí a ty to rozdíly sečteme, tak když na toto číslo provedeme modulo počet políček v jednom soupolí, získáme jakousi paritu otočení všech soupolí ve skupině. Pokud je tato parita nulová, je postavení soupolí možné (v případě na *Obrázku 15* je parita špatná).

U druhého případu může nastat problém nejednoznačnosti. Uživatel zadá pouze obarvení hlavolamu a naším úkolem je zjistit vhodnou permutaci k tomuto obarvení (pokud taková existuje). Problém může nastat, když k jednotlivým políčkům přiřazuji index políčka podle barvy. Takových políček se stejnou barvou totiž může být víc, a při konečné kontrole parity permutace, která vznikla tímto skládáním, může být tato parita nevyhovující. Nejdříve jsem tento problém řešil rekurzivním přidělováním indexů, ale při velikém počtu polí mohlo nalezení správné kombinace trvat příliš dlouho (díky exponenciální složitosti).

Jako docela jednoduché, ale efektivní se ukázalo řešit případnou neshodu parity vyměněním indexů dvou stejně barevných polí ve správné skupině. Při přidělování indexů jednotlivým políčkům si stačí vytvořit seznam dvojic stejně barevných políček v každé skupině. Pokud ve skupině neexistují stejně barevná políčka, jejich indexy jsou tedy určeny jednoznačně a nemusíme se o nic starat. Při kontrole parity sestavené permutace pak stačí zjistit, u kterých skupin je potřeba změnit paritu, aby odpovídala, a v těchto skupinách vyměnit indexy dvojic, které jsme si připravili.

4.6.6 Řešení hlavolamu

U řešení hlavolamu z aktuální pozice dávám uživateli na výběr, zda chce řešit celý hlavolam najednou, nebo po skupinách. U řešení po skupinách je ještě umožněno zvolit řešení bez narušení ostatních skupin nebo nezávisle na ostatních skupinách.

K řešení hlavolamu nezávisle na ostatních skupinách jsem použil obecný algoritmus prohledávání v grafu do hloubky po vrstvách s menší modifikací. Tou modifikací je to, že na počátku si nastavím nějakou proměnnou $I = 1$ a prohledávám graf stavů, dokud nenarazím na stav, kde prvních I políček skupiny, ve které hledám, je na svém místě. Zvýším I o jedna a hledám znovu, dokud není celá skupina vyřešená. Tento algoritmus není nijak moc efektivní, obzvlášť když řeší pouze jednu skupinu, ale je efektivnější než prohledávání bez modifikace. Implementoval jsem ho do programu pouze z pokusných důvodů.

U řešení celého hlavolamu a řešení skupin bez narušení ostatních jsem implementoval 1.algoritmus z kapitoly 2.4 s několika úpravami. Rozhodl jsem se tak proto, že to mi to jevilo jako přirozenější a pochopitelnější řešení (oproti 2.algoritmu z kapitoly 2.4). V prvních verzích programu jsem se pokoušel o implementaci verze algoritmu, kdy pro jednotlivé skupiny vyhledám nezávislé tahy a pomocí nich se pokouším danou skupinu řešit. V průběhu vývoje se však ukázalo, že to není zas tak dobré řešení, protože tím skupinu základních tahů nahrazuji zase jinou množinou tahů. Tato množina se většinou skládá z velkého počtu tahů, což není dobré pro prohledávání a kontrola, zda množina tahů generuje stejnou množinu stavů jako tahy základní není triviální. Proto jsem nakonec zvolil verzi, u které hledám trojpermutace v jednotlivých skupinách.

Jednou ze změn oproti původně definovanému algoritmu je to, že trojpermutace, které potřebuji pro řešení jednotlivých skupin, generuji nezávislé vůči všem ostatním skupinám, ne jenom skupinám s menším indexem. Zaprvé, nemusím určovat, v jakém pořadí budu skupiny řešit, a zadruhé, když umožňuji řešení jednotlivých skupin zvlášť, tyto trojpermutace bych musel hledat stejně, tak už je tedy použiji.

Další úpravou je použití hashovací tabulky, kterou používám pro kontrolu, abych se nedostával do stavů, které už jsem prohledával. V tabulce uchovávám stav a hloubku, kterou mám z daného stavu ještě prohledat. Když se dostanu do nějakého stavu, permutaci vhodně zahashuji a zkontroluji jestli nedošlo ke konfliktu. Pokud došlo, hodnoty v tabulce akorát přepíši aktuálním stavem, jinak konflikt neřeším. I

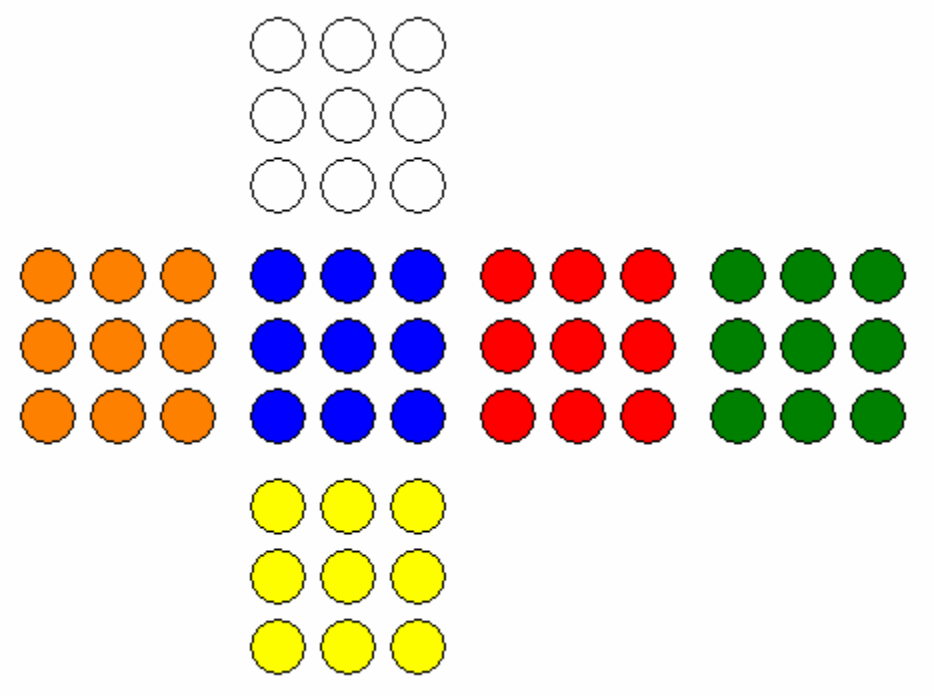
kdyby to byl stav, ve kterém už jsem byl, s velkou pravděpodobností se prohledávání zastaví na okolních vrcholech. Pokud to konflikt nebyl tak musím zkontrolovat jestli je hloubka v tabulce větší než zbývající aktuální. Pokud ano, hledání z aktuálního stavu nepokračuje. Pokud ne, přepíše se v tabulce hloubka na aktuální.

Dále jsem algoritmus obohatil o kontrolu vzdáleností políček. Jedná se o to, že při řešení skupiny potřebuji přesunout políčko A do místa B. Prohledáváním do hloubky po vrstvách tedy hledám stav, ve kterém bude toto políčko přesunuté. V každém stavu kontroluji, zda zbývající hloubka, do které hledám, není menší než nejmenší vzdálenost mezi aktuální pozicí políčka A a místa B. Pokud ano, hledání z dané pozice nepokračuje. Abych mohl vzdálenosti porovnávat, musel jsem si nejdřív připravit tabulku nejmenších vzdáleností mezi každou dvojicí políček ve skupině. Pro zjištění nejmenších vzdáleností jsem použil Floyd-Marshalův algoritmus pro nalezení nejkratších cest v grafu. Díky této kontrole se celkový algoritmus podstatně zrychlil.

4.7 Popis jednotlivých modulů

4.7.1 Modul „Kružnice“

Modul *Kružnice* je určen tvorbu hlavolamů, jejichž geometrickým základem jsou kružnice. Nechal jsem se inspirovat hlavolamem Rashkey. Tento modul je ovšem také univerzálním prostředkem pro reprezentaci libovolného hlavolamu. Výsledný grafický vzhled sice nebude odpovídat reprezentovanému hlavolamu, ale z hlediska funkce se dá bez problému použít. Vezměme si například Rubikovu kostku, ta by v reprezentaci modulem kružnic vypadala např. jako na *Obrázku 16*.



Obrázek 16 – Reprezentace Rubikovy kostky pomocí kružnic

Díky této skutečnosti jsem se nemusel moc zabývat tvorbou více funkčních modulů a mohl jsem se věnovat implementaci hlavní části programu.

Modul *Kružnice* nabízí v návrhu polí dva nástroje. Jeden pro vytvoření kružnice a druhý pro smazání. Vytvářené kružnice mohou mít střed pouze na jakési mřížce, to kvůli jednoduššímu skládání kružnic do roviny. V návrhu polí se ještě nedefinují samotné políčka hlavolamu, ale je zřejmé, že políčka budou tvořeny jednotlivými částmi, které vznikly rozsekáním kružnicemi, a které se dají vyjádřit jako seznam průniků jednotlivých kružnic. Indexovat políčka pomocí seznamu průniků kružnic sice není zas tak úplně jednoznačné, ale jelikož při testování nevznikaly problémy, tak jsem tento problém neřešil. Navíc mojí prioritou nebylo tvořit dokonalý nástroj pro grafickou tvorbu hlavolamu.

Až v návrhu obarvení, při definování barev jednotlivých polí, jsou tyto obarvené pole zároveň definovány jako pole hlavolamu. Každé pole je definováno x-ovou a y-ovou souřadnicí a barvou pole. Pro vykreslení je potřeba vykreslit síť kružnic a na dané souřadnice pole se spustí FloodFill s příslušnou barvou. Síť kružnic se potom smaže. Vykreslování polí by se sice dalo provést inteligentněji, kdybych si pole reprezentoval jako seznam kruhových úseků. Ale jak jsem již napsal, dosavadní řešení se ukázalo jako dostačující a jeho vylepšování nebylo prioritou.

4.7.2 Modul „Kostky“

Modul *Kostky* jsem se rozhodl implementovat ze dvou důvodů. Zaprvé, ukázat funkčnost více než jednoho modulu, a zadruhé, implementovat nástroj pro tvorbu nejznámějších permutačních hlavolamů, Rubikových kostek. Modul tedy umožňuje vytvořit Rubikovy kostky 2x2 až 5x5. Jelikož je funkce modulu specializovaná přímo na Rubikovy kostky, hodně věcí provádí modul automaticky. Například při vytvoření kostky v návrhu polí (postup je vysvětlen v uživatelské dokumentaci) je zároveň definováno obarvení hlavolamu a jsou také definovány jeho tahy. Uživatel má možnost měnit pouze obarvení v návrhu obarvení, tahy jsou definovány pevně a záložka pro definici tahů se při použití tohoto modulu nezobrazuje.

U Rubikových kostek, jako trojrozměrných hlavolamů, je potřeba umožnit zobrazování hlavolamu z více pohledů pro shlédnutí stavu celé kostky. Proto má modul možnost na ploše dynamicky vytvářet svoje objekty, jako například v našem případě tlačítka pro otáčení kostky. Při odpojování modulu pak program automaticky dealokuje objekty, které tam modul vytvořil, aby měl jiný modul čistou plochu.

4.8 Ukládání a načítání hlavolamu

Součástí implementace programu je také možnost uložení vytvořeného nebo rozpracovaného hlavolamu do souboru a jeho následné otevření. Data potřebná pro uložení se nacházejí jak v samotném programu, tak v modulu. Při ukládání se do souboru zapíší informace o tom, jaký modul používá, potom data z programu (např. informace zjištěné analýzou) a potom pomocí příslušné funkce data z modulu (informace o grafické podobě, barvy, pole, tahy). Při otevírání souboru se přečte, jaký modul hlavolam potřebuje, a ten je poté k programu připojen. Dál se načtou data pro program a data pro modul se pošlou modulu ve formě inicializační funkce.

K projektu byly vytvořeny 3 hlavolamy, které jsou jeho součástí: Rashkey, Rashkey 4x, Rubik.

Kapitola 5 - Dokumentace

5.1 Uživatelská dokumentace

Uživatelská dokumentace obsahuje podrobný návod k používání vytvořeného programu (Hlavolamy, verze 2.4). Součástí uživatelské dokumentace je také návod k instalaci. Dokumentace je přepracována také do podoby Nápovědy v samotném programu. Na CD, které je přílohou k Bakalářské práci, se nachází v adresáři „Dokumentace“ pod názvem „Penkala_uziv_dokument“ jako *doc* nebo *pdf* .

5.2 Programátorská dokumentace

Programátorská dokumentace je rozdělena na více částí. Popis algoritmů, který by měl být v programátorské dokumentaci obsažen je popsán přímo v Bakalářské práci, proto je nebudu psát do dokumentace. Dále je součástí dokumentace vygenerovaná dokumentace ze zdrojových souborů programu pomocí programu DoxyGen, která obsahuje stručný popis jednotlivých knihoven a funkcí v nich obsažených. Samotná programátorská dokumentace obsahuje trochu výstižnější popis jednotlivých knihoven programu, popis modulů a návod k jejich vytvoření. Jak samotná dokumentace, tak generovaná dokumentace jsou také přiloženy na CD v adresáři „Dokumentace“.

Kapitola 6 - Závěr

6.1 Zhodnocení vývoje

Po čas celého vývoje aplikace „Hlavolamy“ jsem se potýkal se spoustou implementačních zádrhelů, jako reprezentace permutace od jedničky či od nuly, vymyšlení způsobu zjištění informací o hlavolamu z jeho tahů, nestandardní inicializace funkcí DLL v prostředí Borland C++ Builder, až po implementaci 3 různých algoritmů, než jsem nalezl ten „zatím“ nejefektivnější. Bylo potřeba nastudovat několik způsobů řešení různých známých hlavolamů, pro vysledování specifických vlastností permutačních hlavolamů a odvození efektivního, co nejuniverzálnějšího řešení. Přitom většina nalezených zdrojů, ze kterých jsem čerpal, se týkala pouze konkrétního řešení konkrétního hlavolamu. Když už se mi povedlo na internetu najít nějaké publikace obecnějších řešení, byly převážně v angličtině a někdy se stávalo, že stránky byly zrušeny.

Při vývoji aplikace jsem získal cenné praktické zkušenosti při práci s dynamicky připojovanými knihovnami a s použitím vláken v prostředí Borland C++ Builder. Dále jsem získal komplexnější teoretické znalosti o chování permutací a permutačních hlavolamů (v oboru permutačních hlavolamů jsem získal i mnoho praktických zkušeností).

Program byl testován na řešení hlavolamů Rashkey, Rashkey 4x (moje upravená verze hlavolamu Rashkey) a Rubikových kostek 2x2 až 5x5 s uspokojivými výsledky. Hlavolamy Rashkey, Rashkey 4x a Rubikova kostka 3x3 jsou uloženy jako hlavolamy, které jsou už součástí programu.

6.2 Budoucí vývoj

Implementace samotného řešení hlavolamů, které mi bylo zadáno se ukázalo jako obtížnější, než se na první pohled zdálo. Při stanovení cíle, návrhu a implementaci programu jsem si k řešení přidal ještě grafický návrh, implementaci modulů a pár funkcí uživatelského zobrazení. Proto některé funkce nebyly v projektu vypracovány a některé by se daly ještě vylepšit:

- Jako první věcí, která by se měla v programu dodělat, je řešení blokačních hlavolamů. Program s řešením blokačních hlavolamů počítá pouze tím, že

má v návrhu nachystanou záložku pro definice blokací a má definované základní funkce pro komunikaci s knihovnou, které jsou zatím používány pouze pro vypnutí záložky.

    - U blokačních hlavolamů nastává problém, že tahy je nutné provádět pouze za určitého postavení blokací. Takovým návrhem, jak řešit blokační hlavolam je, že dostaneme blokace na pozice, kam patří. A z této pozice budeme provádět řešení pro ostatní políčka pomocí tahů, které je možné provést, když jsou blokace na pozicích, kam patří. V této fázi je to ovšem pouze nápad, který nebyl implementován.

- V průběhu testování programu se ukázalo, že algoritmus pro detekci soupolí není zas tak univerzální, jak by bylo potřeba. U základních hlavolamů, jako Rubikovy kostky, Rashkey i modifikovaný Rashkey, funguje tak jak má, ale při určitých modifikacích tahů se může stát, že nenalezne soupolí a tím ovlivní i řešení celého hlavolamu. Proto je potřeba implementace obecnějšího řešení.
- Dále by se dal vylepšit dosavadní hlavní algoritmus pro hledání řešení. Kdybychom při hledání trojpermutací nepoužívali pouze jednu, ale víc trojpermutací ve skupině, mohli bychom se dopracovat rychlejšímu nalezení přesunutí jednotlivých 2 políček, protože by bylo jedno, kterou trojpermutaci použijeme.
- Další věcí pro vylepšení je vytvořit různé kvalitnější moduly. Dosavadní moduly jsou tvořeny pouze pro splnění funkčních požadavků, ale moc se nestarají o uživatelské pohodlí. Dalo by se vylepšit použití různých nástrojů, reprezentace dat, grafická příjemnější podoba a ovládání například u otáčení Rubikovou kostkou, kde jsou pohledy dost omezené.
- Dále by se mohl vylepšit celkový vzhled aplikace nebo aspoň jeho součástí (např. přehlednost zadávaných permutací v části *Definice tahů*).
- Při testování také vznikly připomínky a požadavky na rozšíření funkcí v *Uživatelském zobrazení*, jako nabízet při vyřešení hlavolamu provádět tahy po krocích, zobrazovat hlavolam zároveň ve složené podobě (kdyby byl hlavolam nějak složitě obarven, bylo by těžké se orientovat, která pole hlavolamu už jsou poskládaná) a také nabízet rovnou inverzní tah. Poslední

dva požadavky nebyly doposud realizovány, protože u testovaných hlavolamů nabyly potřeba.

6.3 Splnění cíle

Cíl a samotné zadání práce se podařilo splnit ve velké míře, až na požadavek řešení blokačních hlavolamů. Ten nebylo, díky složitosti implementace celé práce a implementačním zádrhelů, možné z časových důvodů implementovat. Jinak se povedlo splnit všechny požadavky, které byly na program kladeny.

- Program řeší hlavolamy se slušnou efektivitou.
- Uživatel má možnost zadat libovolné postavení hlavolamu buď pomocí permutace nebo definováním obarvení polí.
- Díky modulům je každý typ hlavolamu optimálně definován a datově reprezentován.
- Vytvořeno univerzální grafické prostředí pro návrh hlavolamu.
- Interaktivní zobrazování a práce s hlavolamem v „uživatelském režimu“.
- Možnost přidávání modulů bez zásahu do programu.

Byl tedy vytvořen funkční nástroj se snahou o řešení co největší množiny permutačních hlavolamů, s možností vytváření libovolného hlavolamu a práce na něm, a možností pokračování ve vývoji.

Literatura

[1] Standart Template Library Programmer's Guide,
http://www.sgi.com/tech/stl/

[2] Charles Petzold: Programming Windows by Charles Petzold,
Microsoft Press, 1988

[3] Borland C++ Builder
http://www.functionx.com/bcb/index.htm

[4] Solving Rubik's Cube for speed by Lars Petrus
http://lar5.com/cube/

[5] RUBIK's CUBE-LIKE PUZZLES
http://cadigweb.ew.usna.edu/~wdj/rubik.html