ZÁPADOČESKÁ
UNIVERZITA
V PLZNI

Dokumentace k semestrální práci z předmětu KIV/PC
Loydova patnáctka

Vypracoval Oldřich Pulkrt
2012/2013

Obsah

1 Zadání

Naprogramujte v ANSI C přenositelnou konzolovou aplikaci, která jako vstup načte z parametru na příkazové řádce výchozí stav hlavolamu „Loydova patnáctka“ a tento hlavolam vyřeší, tj. převede jej posloupností povolených tahů do základní pozice (posloupnost od jedné do nejvyššího čísla zadání a na poslední pozici - vpravo dole - je prázdné pole).

Celé zadání lze nalézt na adrese http://www.kiv.zcu.cz/studies/predmety/pc/doc/work/sw2012-02.pdf

2 Analýza úlohy

Krátký popis Loydovy patnáctky - Jde o hlavolam, kde na čtvercové hrací ploše ($n \times n$) jsou zamíchány hrací kameny (kterých je $n^2 - 1$) a cílem této hry je seřadit tyto kameny postupně od prvního do posledního podle čísel na těchto kamenech. Jedno pole vždy zůstává volné, díky čemuž je možné na jeho místo vždy přesunout jiný kámen - v každé situaci je možno udělat 2-4 různé tahy. Ve vyřešeném hlavolamu se prázdné pole nachází vpravo dole na hrací ploše.

V této sekci se budu postupně zabývat dvěma hlavními problémy u této úlohy. Prvním z nich je ukázat, zda je zadání opravdu řešitelné, a druhým problémem bude samotné vyřešení. Předem předpokládáme, že uživatel nemusí znát správný tvar zadání, takže se na vstupu může objevit cokoli a program sám se musí ujistit o tom, že je zadání v pořádku. Bude-li cokoli v nepořádku, program na to upozorní, viz uživatelská příručka.

2.1 Řešitelnost zadání

Nejprve musíme zkontrolovat, zda je zadání ve správném tvaru. Musíme se ujistit, že:

- se shoduje počet řádek i počet sloupců a že je na každém řádku stejný počet zadaných hodnot
- se žádná hodnota v zadání neopakuje
- zadání obsahuje prázdné místo
- jsou zadány nejméně 3 řádky a sloupce (pro nižší hodnoty je zadání triviální)

Každý z těchto bodů je jednoduché ověřit, v programu dojde k průchodu celého zadaného řetězce a spočtení počtu řádků a počtu hodnot na každém z nich. Poté se vytvoří pole o velikosti n^2, které se inicializuje na samé nuly, a poté se znovu prochází celý zadaný vstupní řetězec, ale na indexu každého nalezeného čísla se nastaví hodnota 1 (dojde vlastně k označení čísel, která jsou zadaná). Pro ověření, že byly zadány všechny nutné hodnoty, stačí projít toto pole a ujistit se, že neobsahuje žádné nuly.

Nadále budu předpokládat, že zadání bylo v pořádku a budu se věnovat zjištění, zda je toto zadání řešitelné.

Počet všech možných zadání je $n^2!$, ale ne každé z nich lze seřadit do vyřešené pozice. Rozdělíme si všechny možné stavy hry na dvě třídy, ty, které lze vyřešit, a ty, které vyřešit nelze. Zde jsou vidět obě možnosti pro zadání 4×4, jak konečná neřešitelná pozice, tak konečné správné řešení:

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

Obrázek 1: Neřešitelné zadání

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

Obrázek 2: Vyřešené zadání

Všimneme si, že ve správném zadání je kterékoli číslo větší než kterékoli číslo před ním, počet inverzí (počet čísel, která jsou menší než některé z čísel před nimi) je tedy 0. Počet inverzí se při pohybu vodorovně nemění, a při svislém pohybu se změní o $n - 1$ (pokud je tedy n sudé číslo, střídá počet inverzí při svislém posuvu sudou a lichou hodnotu, pokud je ale n liché číslo, zůstává počet inverzí sudý). Z těchto poznatků můžeme vyvodit věty, které nám pomohou u každého zadání určit, zda je řešitelné či ne.

1. Pokud je n liché číslo, pak počet inverzí v řešitelném zadání je sudý.
2. Pokud je n sudé číslo a prázdné pole je na sudém řádku od spodu, pak počet inverzí v řešitelném zadání je sudý.
3. Pokud je n sudé číslo a prázdné pole je na lichém řádku od spodu, pak počet inverzí v řešitelném zadání je lichý.

Řešitelnost zadání budeme ověřovat spočtením inverzí a porovnáním s těmito větami (kromě počtu inverzí tedy budeme potřebovat n jako počet řádků nebo sloupců a pozici prázdného pole odspodu).

2.2 Řešení úlohy

Úlohu lze řešit mnoha různými způsoby, v této části představím několik z nich a vyberu optimální řešení.

2.2.1 Řešení využitím Dijkstrova algoritmu

Tento způsob řešení úlohy je založen na nalezení nejkratší cesty mezi zadaným stavem a stavem koncovým průchodem grafu *Dijkstrovým algoritmem*. Program by probíhal tak, že

by z aktuálního vrcholu vytvořil všechny dostupné uzly grafu (všechny možnosti okolních stavů), a propojil vrcholy cestou délky 1. Poté by tímto způsobem prozkoumal každý z těchto nových stavů. Takto by postupoval, dokud by nenalezl nejkratší cestu (první výskyt řešení by mu nestačil, neboť by mohla existovat kratší cesta a tu by se snažil nadále najít.

Na první pohled je zde patrná největší nevýhoda tohoto řešení - paměťová náročnost a průchod uzly i po nalezení řešení, kdy algoritmus hledá kratší cestu.Toto řešení tedy můžeme zavrhnout.

2.2.2 Řešení využitím BFS

BFS (Breadth-First Search) je algoritmus procházení grafu do šířky. Z počáteční pozice bychom určili všechny možné tahy a uložili je do *FIFO fronty*. Poté bychom postupně vybírali prvky z této fronty a určovali další možné tahy, které bychom stále ukládali. Ačkoli je jisté, že tento algoritmus by našel všechna řešení (pokud bychom ho nechali prohledat celý graf), trvalo by to velice dlouho. Podle Wikipedie[1] je u zadání 4×4 potřeba k optimálnímu řešení 0 - 80 tahů. Pokud si řekneme, že průměrně je třeba k řešení 40 tahů, a z každého vrcholu existují 2 - 4 cesty (pro větší jednoduchost budeme předpokládat právě 2 cesty), můžeme určit, že tento algoritmus prohledá 2^{40} stavů (přibližně 10^{12}), než nalezne optimální řešení (a to zdaleka není nejhorší možný případ, zde šlo pouze o průměrný počet tahů a ani jsme nepočítali se všemi možnými tahy.

Je vidět, že tento způsob řešení by byl výhodnější než Dijkstrův algoritmus, protože namísto nejkratší cesty by nám stačil první výskyt řešení. Přesto je pro naše účely stále příliš pomalý, ale v následující části se již podíváme na zvolené řešení, které je vlastně vylepšené *BFS*.

2.2.3 Řešení využitím A* algoritmu

Předchozí algoritmus bychom mohli vylepšit, pokud bychom přibližně věděli, zda je cesta, kterou právě v grafu procházíme, blízká řešení. Toto lze ověřit použitím heuristiky, což je vlastně odhadnutí vzdálenosti řešení od aktuální pozice.

Hammingova vzdálenost Hammingovu vzdálenost je velmi snadné určit a jde o prvního kandidáta na heuristickou funkci. Hammingova vzdálenost se určí jako počet hracích kamemů, které jsou na špatných pozicích. Použití Hammingovy vzdálenosti jako heuristické funkce se přímo nabízí, ale nepoužijeme ji, protože tato hodnota není příliš přesná. Když si totiž uvědomíme, že hrací kameny, které jsou na špatném místě, se musí pohnout více než jednou, aby se dostaly na svou pozici (například pokud jsou vedle sebe dva špatně umístěné kameny, nestačí ani dva pohyby, abychom je posunuli na jejich správné pozice - musíme nejprve uvolnit jedno místo a až poté je prohodit).

[1] Více na http://en.wikipedia.org/wiki/15_puzzle

Manhattanská vzdálenost Tato vzdálenost je již mnohem lepší pro použití jako heuristické funkce, protože zohledňuje nejen to, zda je hrací kámen špatně umístěn, ale i to, jak daleko je od svého správného umístění. Parciální Manhattanskou vzdálenost čísla a tedy spočteme jako součet vzdálenosti od správného umístění v ose x a vzdálenosti od správného umístění v ose y:

$$manhattan_x = (a-1) \mod n \tag{1}$$
$$manhattan_y = (a-1)/n \tag{2}$$
$$manhattan = |manhattan_x| + |manhattan_y| \tag{3}$$

Celková manhattanská vzdálenost je tedy součet těchto „parciálních Manhattanských vzdáleností“. Čím nižší tato vzdálenost bude, tím blíže k řešení se budeme nacházet. Protože Manhattanská vzdálenost bude vždy vyšší nebo rovna Hammingově vzdálenosti, rozhodl jsem se jako heuristickou funkci použít právě Manhattanskou vzdálenost.

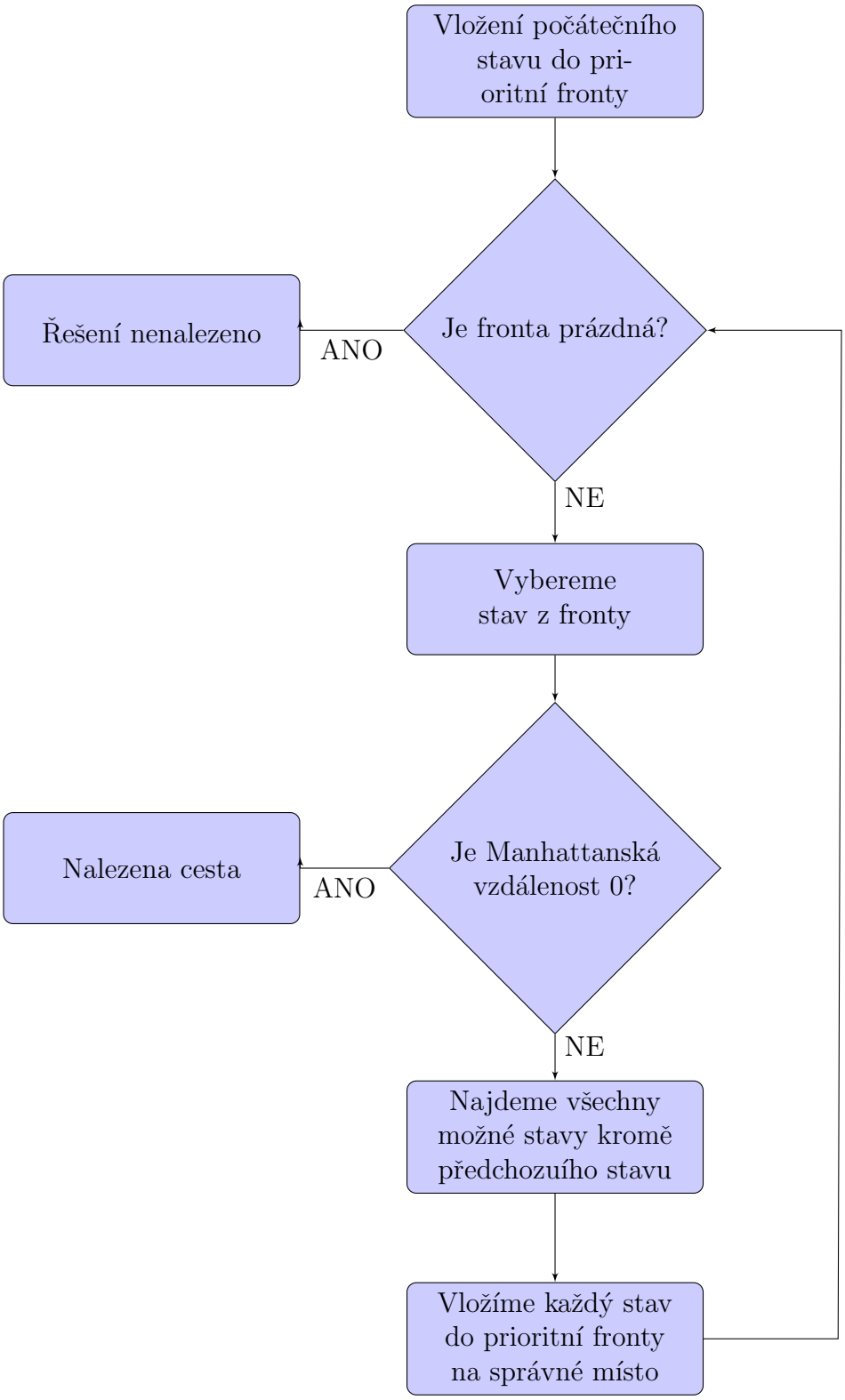
A* algoritmus A* algoritmus je podobný BFS, ale cestu si vybírá podle funkce f(x):

$$f(x) = h(x) + g(x)$$

Funkce *h(x)* je zvolená heuristická funkce a *g(x)* je vzdálenost vrcholu od počátečního stavu. Čím nižší je hodnota funkce *f(x)*, tím lepším kandidátem je pro pokračování. Každý navštívený vrchol se bude ukládat do prioritní fronty, kde na prvním místě bude vždy prvek s nejnižší hodnotou funkce *f(x)*. Díky tomu se budou prohledávat opravdu nejnadějnější uzly. Přesto můžeme počet prohledávaných vrcholů ještě snížit - pokud do fronty nebudeme přidávat vrchol grafu, ze kterého jsme se do aktuální pozice dostali (tím zabráníme zacyklení a zároveň budeme uvažovat pokaždé jen 1 - 3 cesty z daného vrcholu). Na obrázku na následující straně je vidět samotný algoritmus pro nalezení řešení.

Díky použití heuristické funkce prozkoumáváme skutečně pouze ty vrcholy, které vypadají, že by mohly vést k řešení. Do prioritní fronty se ukládá každý následovník (kromě toho, ze kterého jsme se do aktuálního vrcholu dostali), proto se můžeme kdykoli ke kterémukoli vrátit, pokud se ukáže, že je výhodnější než aktuálně prozkoumávaný vrchol.

Toto řešení mi přišlo nejlepší z výše zmiňovaných, proto jsem se rozhodl ho implementovat v jazyce C. Myslím si ale, že by bylo možné řešení ještě vylepšit, možná nějakou lepší heuristickou funkcí (žádná mne ale nenapadla a ani jsem nenašel zmínku o žádné lepší funkci pro tento problém), využitím databáze řešení dílčích problémů (pokud bychom znali posloupnost tahů, kterou bychom část hlavolamu vyřešili, nebylo by nutné tuto část řešení hledat algoritmem, což by určitě ušetřilo některé zdroje) nebo podmínkou pro přidání prioritní fronty (pokud víme, že pro hlavolam o velikosti 4×4 je průměrný počet tahů 40, mohli bychom všechny stavy, pro které by $f(x) \geqq 40$, vypustit a vůbec je do fronty nepřidávat - to by ušetřilo především paměť).


Vložení počátečního stavu do prioritní fronty
Je fronta prázdná?
ANO
Řešení nenalezeno
NE
Vybereme stav z fronty
Je Manhattanská vzdálenost 0?
ANO
Nalezena cesta
NE
Najdeme všechny možné stavy kromě předchozuího stavu
Vložíme každý stav do prioritní fronty na správné místo

3 Popis implementace

3.1 Prioritní fronta

Prioritní frontu jsem implementoval jednoduše, v programu je struktura **GameState**, která obsahuje ukazatel na následující prvek ve frontě. Vkládání probíhá metodou `insertPQ(struct GameState *s)`, která sama najde místo, kam by měl být prvek *s* vložen, podle jeho vzdálenosti od počátečního bodu a Manhattanské vzdálenosti.

Struktura GameState:

```
struct GameState
{
        int** tilesPosition;
        int manhattanDistance;
        int distance;
        struct GameState *next;
        struct GameState *prev;
}
```

int tilesPosition** Dvourozměrné pole obsahující rozmístění jednotlivých hracích kamenů

int manhattanDistance Manhattanská vzdálenost pro toto rozmístění hracích kamenů

int distance Aktuální vzdálenost od počátečního stavu (každým pohybem se zvyšuje o 1

struct GameState *next Ukazatel na další prvek v prioritní frontě

struct GameState *prev Ukazatel na předchozí stav - ten je nutné uchovávat, abychom při nalezení řešení mohli zpětně projít všechny stavy, kterými jsme se k řešení dostali

Ke správné funkčnosti ovšem nestačí mít strukturu, která bude prioritní frontou, jsou třeba i funkce, které s ní budou pracovat. V našem případě jde o funkce:

- notEmptyPQ() - vrací **1**, pokud fronta není prázdná, jinak vrací **0**
- getQueueTop() - Vrací ukazatel na první prvek v prioritní frontě (=prvek s nejnižším součtem vzdálenosti od počátku a Manhattanské vzdálenosti)
- insertPQ() - vloží prvek na správné místo uvnitř prioritní fronty (správné místo se určí součtem vzdálenosti od počátečního stavu a Manhattanské vzdálenosti)
- printPQ() - funkce vypíše od začátku všechny prvky v prioritní frontě, tato funkce slouží pouze pro kontrolu a ve finální verzi programu se nepoužívá

3.2 Ostatní funkce

Ve stručnosti zde popíši i ostatní funkce, nejprve ty skutečně důležité, na kterých stojí samotná funkčnost programu, a poté ostatní.

solveFifteen() Funkce postupuje podle vývojového diagramu, postupně vybírá z prioritní fronty prvky s nejnižším součtem vzdálenosti od počátečního stavu a Manhattanské vzdálenosti, zjišťuje, zda nejde o cílový stav, a pokud ne, vkládá do prioritní fronty další stavy (které se vytvářejí funkci getNewState).

isSolvableState(struct GameState *state) Tato funkce zjišťuje, zda je tento stav skutečně řešitelný, podle pravidel, která jsme určili v analýze úlohy.

getLastMove(struct GameState *state) Vrátí poslední pohyb hracího kamene, každý směr má definovánu konstantu (MOVE_LEFT, MOVE_UP...) a tuto konstantu funkce vrací. Tuto funkci využijeme při výpisu řešení funkcí printSolution().

getManhattanDistance(struct GameState state) Vrací Manhattanskou vzdálenost určenou podle vzorečků z kapitoly o analýze úlohy. Funkce vrací součet všech částečných Manhattanských vzdáleností.

moveLeft() (moveRight(), moveUp(), moveDown()) Tyto funkce provádí samotný pohyb některého hracího kamene na prázdné místo. Před použitím těchto funkcí nejprve zkontrolujeme, zda je tento tah vůbec přípustný, pomocí funkce canMoveLeft() (canMoveRight()...).

getMovedTile(struct GameState *state) Vrací číslo hracího kamene, se kterým bylo v posledním tahu pohnuto. Tuto funkci využijeme při výpisu řešení metodou printSolution().

canMoveLeft(struct GameState *state) (canMoveRight(), canMoveUp(), canMoveDown()) Zjišťuje, zda je možné provést pohyb daným směrem.

getNewState(struct GameState *state) Vytvoří nový vrchol, včetně číselné reprezentace rozmístění hracích kamenů a alokace paměti pro následující prvek v prioritní frontě a nastaví předaný vrchol jako předchozí (kvůli uchování posloupnosti tahů v řešení).

getRowsCount(char *initState) Vrací počet řádků v zadání, je nutný hlavně při kontrole správnosti zadání (musí se shodovat počet prvků ve sloupci i řádku).

printSolution(struct GameState *queue) Vypíše řešení, postupně prochází celý řetězec vrcholů (pomocí ukazatele na předchozí vrchol) a ukládá si jednotlivé tahy. Na výpisu dojde jen k obrácení pořadí tahů, abychom nevypisovali tahy od vyřešeného hlavolamu k zadání.

printMatrix(int **matrix) Vypíše dvourozměrné pole.

4 Uživatelská příručka

4.1 Vstupy

Veškerá práce s programem probíhá ještě před ejho spuštěným, jelikož se data programu předají jako parametr již na příkazové řádce. Program se spouští příkazem

```
solve15.exe <parametr>
```

Parametr by měl být řetězec v uvozovkách, obsahující zadání hlavolamu tak, že mezery oddělují jednotlivé hodnoty a středníky oddělují jednotlivé řádky. Správné zadání by mělo vypadat podle následujících bodů:

- středník na posledním místě být může, ale nemusí
- žádné číslo by se nemělo v zadání opakovat
- v zadání by měla být čísla pouze v rozsahu daných počtem sloupců a řádků
- počet řádků a sloupců by měl být minimálně *3*

4.2 Výstupy

Program má dva druhy výstupů, výstupy chybové a výstup s řešením.

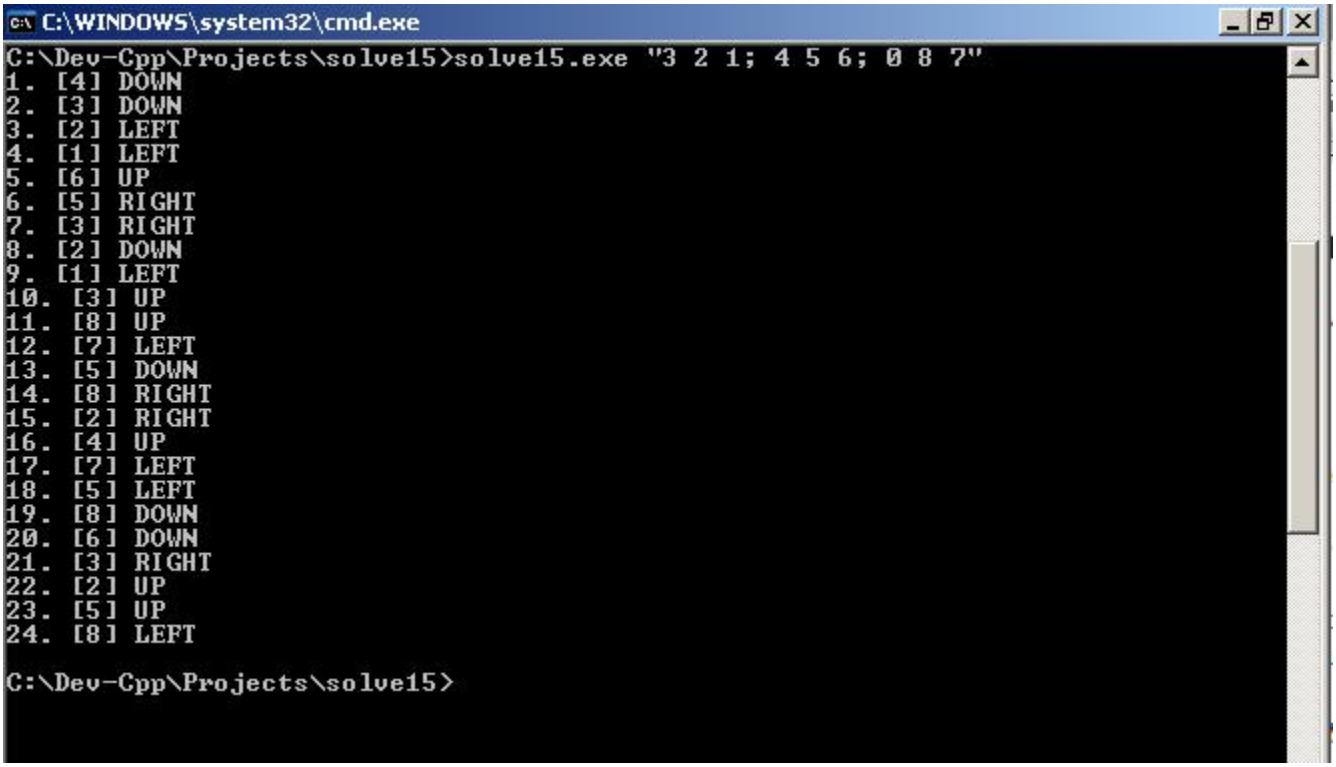
Chybové výstupy Následující tabulka zachycuje chybové výstupy:

Chybový kód	Chybová hláška	Význam
1	ERR#1: Missing argument!	Programu nebyl předám paramter, takže nemá s čím pracovat.
2	ERR#2: Malformed input!	Nejde o platné zadání, některá z hodnot se opakuje, neshoduje se počet prvků v řádcích a sloupcích..
3	ERR#3: Field too small!	Byla zadáno menší pole než 3×3
4	ERR#4: Non-existent solution!	Pro dané zadání neexistuje řešení.
5	ERR#5: Out of memory!	Není k dispozici dostatek operační paměti.
6	ERR#6: Cannot continue!	Nespecifikovaná chyba.

Výstup řešení Pokud bylo zadání v pořádku, program bude hledat řešení a když ho nalezne, vypíše seznam kroků, kterými lze zadání vyřešit. Každý z kroků je ve tvaru:

`<číslo tahu>: [<kámen>] <směr posunu>`

Příklad řešení



5 Závěr

Semestrální práci jsem programoval v programu Dev-C++, což zřejmě není příliš dobré prostředí, a kdybych měl začít od začátku, vybral bych si některé jiné. Nejprve jsem chtěl práci programovat v prostředí NetBeans, ale nepodařilo se mi v něm zprovoznit plugin pro C, proto jsem se rozhodl nainstalovat program Dev-C++, který jsem používal ve škole.

Při řešení tohoto úkolu jsem zaznamenal několik problémů. Prvním byla alokace paměti, kdy při spouštění programu nastávaly chyby, a mně trvalo dlouho, než jsem zjistil, kde ke špatné alokaci dochází. Pro některá zadání také program trvá velmi dlouho, nepodařilo se mi ověřit, zda je to kvůli velkému množství tahů, které je třeba vyzkoušet před nalezením řešení, ale věřím, že tomu tak je, především proto, kolik možných stavů je třeba prozkoumat již pro některá zadání 3×3.

Samostatnou kapitolou byla práce na dokumentaci, která byla hlavně ze začátku pomalá, ale po vyzkoušení si práce s LaTeXem a prohlédnutí několika tutorialů již šla dobře a mnohem rychleji.

Zdroje

http://www.cut-the-knot.org/pythagoras/fifteen.shtml

http://www.cs.cmu.edu/~adamchik/15-121/labs/HW-7%20Slide%20Puzzle/lab.html

http://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html

http://www.cs.princeton.edu/courses/archive/spring09/cos226/assignments/8puzzle.html

http://larc.unt.edu/ian/pubs/saml.pdf