SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta informatiky a informačných technológii

Umelá Inteligencia

Zadanie č.2:

Prehľadávanie stavového priestoru pomocou algoritmu lačného hľadania

Tomáš Socha

ID: 110896

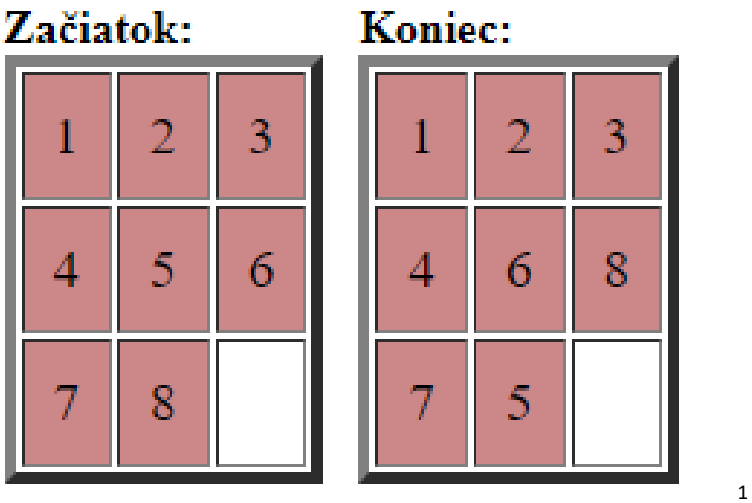
Akademický rok 2021/2022

1. Zadanie

Mojou úlohou bolo nájsť riešenie 8-hlavolamu pomocou použitia lačného hľadania a následnom porovnaní rôznych heuristík.

Hlavolam je zložený z 8,alebo viac očíslovaných políčok a jedného prázdneho miesta. Nad políčkami je možné vykonávať pohyby hore, dole, vľavo alebo vpravo, ale len ak sa tým smerom nachádza medzera. Vždy je zadaná nejaká začiatočná a cieľová pozícia, a úlohou je nájsť postupnosť krokov, ktoré vedú z jednej pozície do druhej.

Príkladom môže byť nasledovná začiatočná a koncová pozícia:



[1]

2. Implementácia

Vo vlastnej implementácii bolo mojou úlohou použiť na vyriešenie hlavolamu **lačné hľadanie**. Na jej realizáciu bol použitý programovací jazyk Python vo verzii 3.9 hlavne z dôvodu jednoduchšej práce a množstvu vstavaných funkcii.

a) Algoritmus

Algoritmus lačného hľadania je kombinácia prehľadávania do šírky s heuristickou funkciou. Jeho princíp spočíva v prehľadávaní uzlov, ktoré sú bližšie ku výsledku. Ohodnotenie jednotlivých uzlov je zabezpečené pomocou heuristík. Algoritmus vždy vygeneruje potomkov uzla v ktorom sa aktuálne nachádza, aplikuje na nich heuristickú funkciu a vloží do zoznamu vygenerovaných uzlov podľa najmenšej hodnoty.

[1] http://www2.fiit.stuba.sk/~kapustik/z2d.html

Následne vyberie uzol s najmenšou hodnotou a zaradí ho medzi navštívené a generuje jeho potomkov. Tento cyklus sa opakuje pokiaľ sa nenájde koncový stav, alebo kým sa nevyprázdni list vygenerovaných uzlov

Bol implementovaný vo funkcii *najdenie_cesty()* ktorej úlohou bolo nájdenie cesty z počiatočného stavu hlavolamu, do koncového.

b) Stav

Je reprezentovaný pomocou listov vložených v jednom liste, kde každý vnorený list predstavuje jeden riadok. Hodnoty v **liste** sú reprezentované ako **znaky**. Výhodou zvoleného zápisu je jednoduchšia a prehľadnejšia reprezentácia hlavolamu ľubovoľných rozmerov m*n na úkor komplikovanejšej realizácie jednotlivých operácii nad stavmi. Prázdne políčko je reprezentované písmenom **m**.

```
[['13', '11', '8', '15'], ['12', '3', '2', '9'], ['4', '1', '5', '7'], ['6', '14', 'm', '10']]
```

c) Operátory

Pohyb jednotlivých políčok je zabezpečený pomocou **operátorov** VPRAVO, VLAVO, DOLE, HORE. Tieto pohyby som implementoval do jednej komplexnejšej funkcie *pohyb(),* ktorej jedným z argumentov je premenná *smer_pohybu,* ktorá zabezpečí pohyb správnym smerom.

d) Heuristiky

Pre porovnanie boli použité 3 **heuristiky**:

I. Počet políčok, ktoré nie sú na svojom mieste
II. Súčet vzdialeností jednotlivých políčok od ich cieľovej pozície
III. Kombinácia predchádzajúcich odhadov v pomere 1/4 pre prvú a druhú heuristiku

Každá heuristika bola reprezentovaná vlastnou funkciou.

e) Uzol

Na reprezentáciu **uzla** bola použitá trieda *Uzol*. Jeho atribútmi sú:

I. Stav(to, čo uzol reprezentuje)
II. Ukazovateľ na rodiča – kvôli vypísaniu cesty z počiatočného stavu do koncového
III. Hodnota – kvôli rýchlejšiemu vkladaniu do listu vygenerovaných listov

```
class Uzol:
    stav = None
    predchodca = None
    hodnota = None
```

3. Opis riešenia

Jadro celého programu je obsiahnuté vo funkcii *najdenie_cesty(),* ktorá obsahuje while cyklus slúžiaci na prechod jednotlivými uzlami a porovnávaním aktuálneho stavu s koncovým na nájdenie **cesty** k nemu.

V každej iterácii sa najskôr inkrementuje premenná *pocet_iteracii*, v prípade dosiahnutia **5mil** iterácii sa proces hľadania cesty zastaví a program vypíše hlášku neúspešného nájdenia cesty.

Následne sa na základe pozície medzery zavolá funkcia *pohyb()*, ktorá **vygeneruje** uzly reprezentujúce možné pohyby z aktuálneho stavu. V tejto funkcii sú generované len uzly, ktorých stavy ešte neboli **navštívené**.

Novo vzniknuté uzly sa vložia do listu vygenerovane_uzly() tak, aby daný list zostal **usporiadaný** vzostupne podľa ohodnotenia jednej z heuristík. Na to slúži funkcia vlozenie_do_listu(). V prípade viacerých uzlov s rovnakou hodnotou sa vkladaný uzol vloží pred nich. Funkcia taktiež prechádza všetkými už vloženými uzlami s rovnakou hodnotou, a v prípade nájdenia rovnakého stavu ukončí svoju činnosť a vkladaný uzol nepridá do listu **vygenerovaných** uzlov.

Po rozšírení listu vygenerovaných uzlov sa program pokúsi z neho vybrať prvok s **najmenšou** hodnotou. V prípade že je daný **list** prázdny, program skončí s neúspechom. V opačnom prípade vloží uzol do slovníka(dict) spracovane_uzly.

Ďalej vo svojej implementácii používam funkciu vyber_heuristiky() ktorá má za úlohu vybrať správnu **heuristiku** na základe zvolenia. Jednotlivé heuristiky sú následne reprezentované vlastnými funkciami, kde sa vykoná potrebný výpočet. Taktiež som spravil funkciu *list_na_string(),* ktorá prevedie vybraný stav reprezentovaný listom na string, aby bolo možné mať jednotlivé spracované stavy uložené v slovníku vďaka čomu je možné pristupovať k jednotlivým **prvkom** v konštantnom čase.

Taktiež som do riešenia implementoval *funkciu riesitelnost_hlavolamu()* na zistenie **riešiteľnosti** hlavolamu. Funkcia je zavolaná pri každom vygenerovaní nových stavov. Jej princíp je založený na zistení počtu **inverzii** počiatočného a koncového stavu pomocou for cyklov a **porovnávaním** hodnôt na jednotlivých pozíciách. Taktiež sa zistí riadok, v ktorom sa nachádza prázdne políčko. Aby bol daný problém **riešiteľný** musí byť splnená niektorá z nasledujúcich podmienok[2]:

I. Ak je počet stĺpcov **neparný** a počet inverzií je v oboch prípadoch buď párny alebo nepárny
II. Ak je počet stĺpcov **párny** a počet inverzii + **číslo riadka** prázdneho políčka v oboch prípadoch buď párny alebo nepárny.

Na testovanie je vytvorená funkcia *generator(),* a *tester()* slúžiace na vygenerovanie začiatočného a koncového stavu podľa **zvolených rozmerov** a následné zavolanie ostatných funkcii na nájdenie cesty.

Časová aj pamäťová zložitosť je v najhoršom prípade O(b^m), kde m je max hĺbka prehľadávaného priestoru a b je faktor vetvenia. Taktiež **nieje prípustný** ani úplný, takže nehľadá vždy najlepšie riešenie.

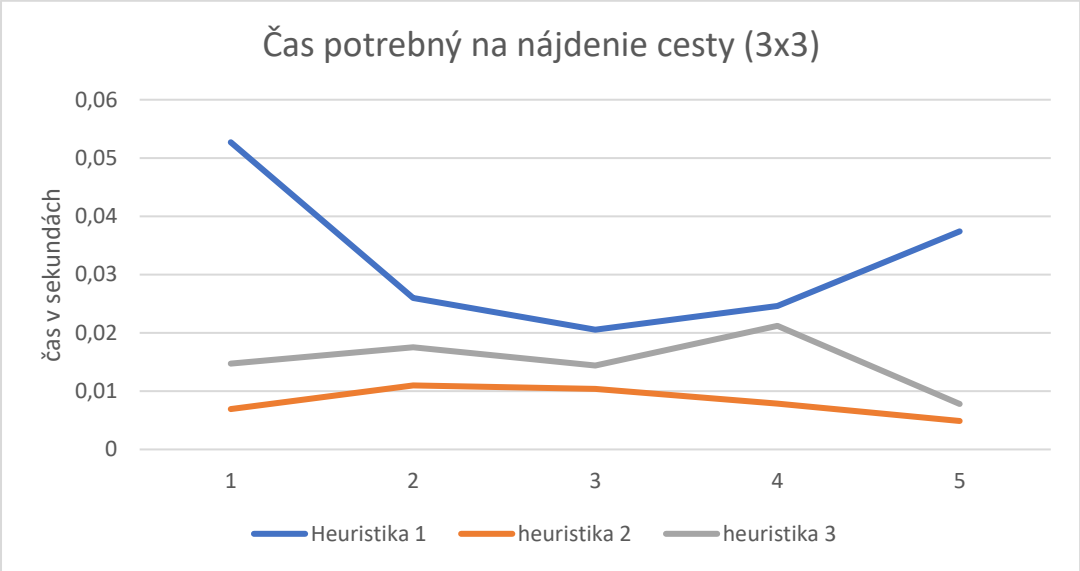
[2] https://www.cs.mcgill.ca/~newborn/nxp_puzzleOct9.htm

4. Testovanie

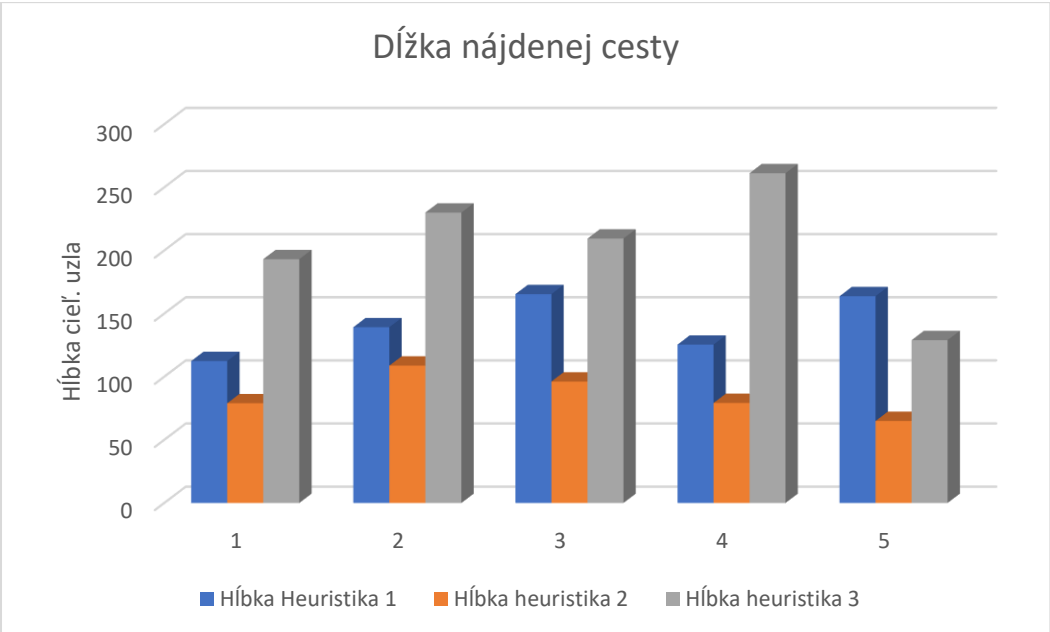
a) Porovnanie časov rôznych heuristík na rozmeroch 3x3

Začal som s testovaním času na štandardných rozmeroch, čiže 3x3. Na grafe nižšie môžete spozorovať priemerné časy jednotlivých meraní po aplikovaní rôznych heuristík. Jedna hodnota na grafe zodpovedá jednému meraniu, počas ktorého bolo vygenerovaných 5 rôznych vstupov a výsledkov.

Najlepšie obstála heuristika_2, čo ma mierne prekvapilo, pretože som očakával lepší výsledok heuristiky_3. Avšak rozdiely týchto dvoch sú minimálne, jasne víťazia nad heuristikou_1.



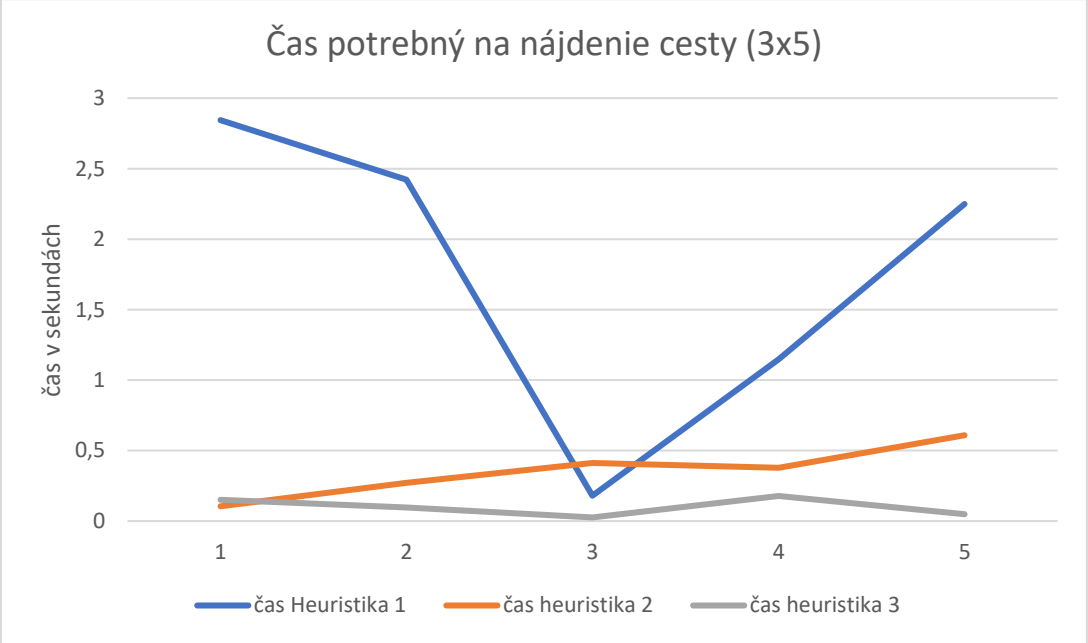
Počas testovania som taktiež zaznamenával dĺžky nájdených ciest, čiže v akej hĺbke sa nachádza cieľový stav. Vo všetkých prípadoch sa heuristike_2 podarilo nájsť najkratšiu cestu v porovnaní so zvyšnými. Vo väčšine prípadov najhoršie dopadla heuristika_3, čo mohlo byť spôsobené že heuristika nadhodnocuje, čiže nie je prípustná.



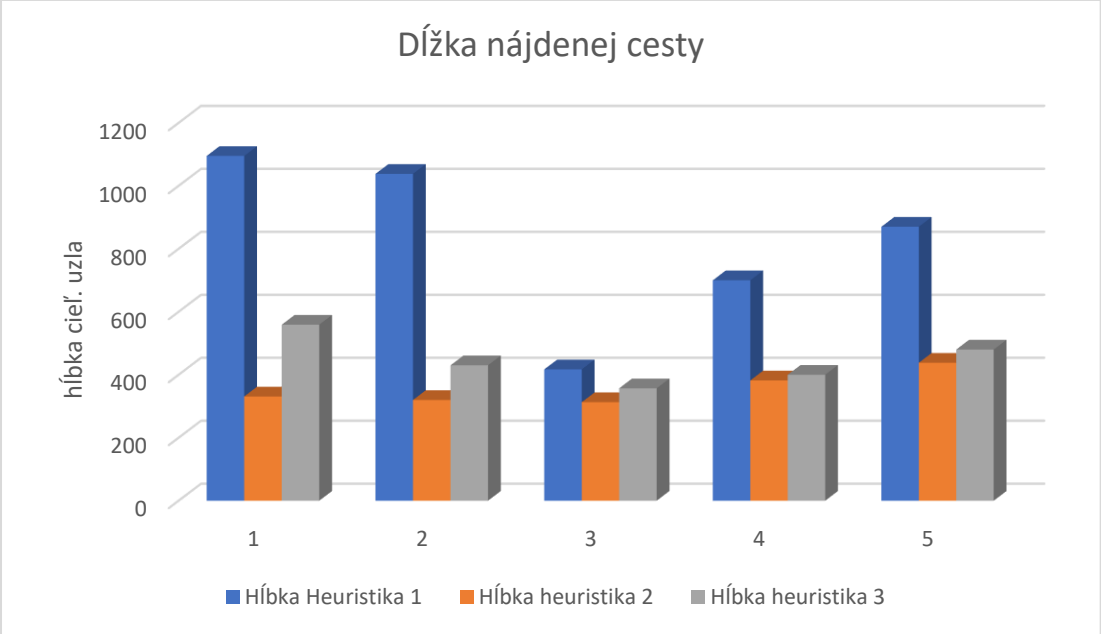
b) Porovnanie časov rôznych heuristík na rozmeroch 3x5

V ďalšom teste som sa snažil zistiť, ako sa zmení porovnanie jednotlivých heuristík pri väčšom počte políčok. V tomto prípade jednému meraniu zodpovedá vygenerovanie jedného riešiteľného vstupu a výstupu, pretože programu sa častejšie stávalo, že sa mu nepodarilo nájsť správne riešenie v prípade heuristiky_1.

V tomto prípade už má mierne navrch heuristika_3 oproti heuristike_2. Heuristika_1 výrazne zaostáva, no v jednom prípade dokázala nájsť cestu skôr ako heuristika_2



Zmeny nastavili aj pri porovnaní hĺbok. Heuristika_2 je na tom stále najlepšie, no heuristika_3 a 1 si vymenili pozície a heuristika_3 sa v niektorých prípadoch úplne približuje hodnotám 2.heuristiky

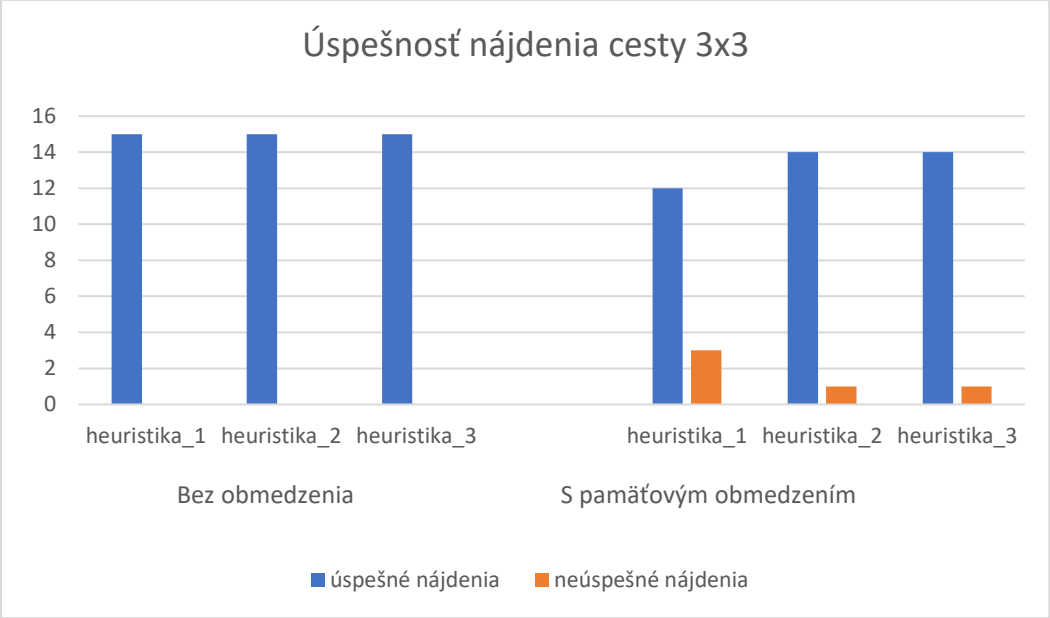


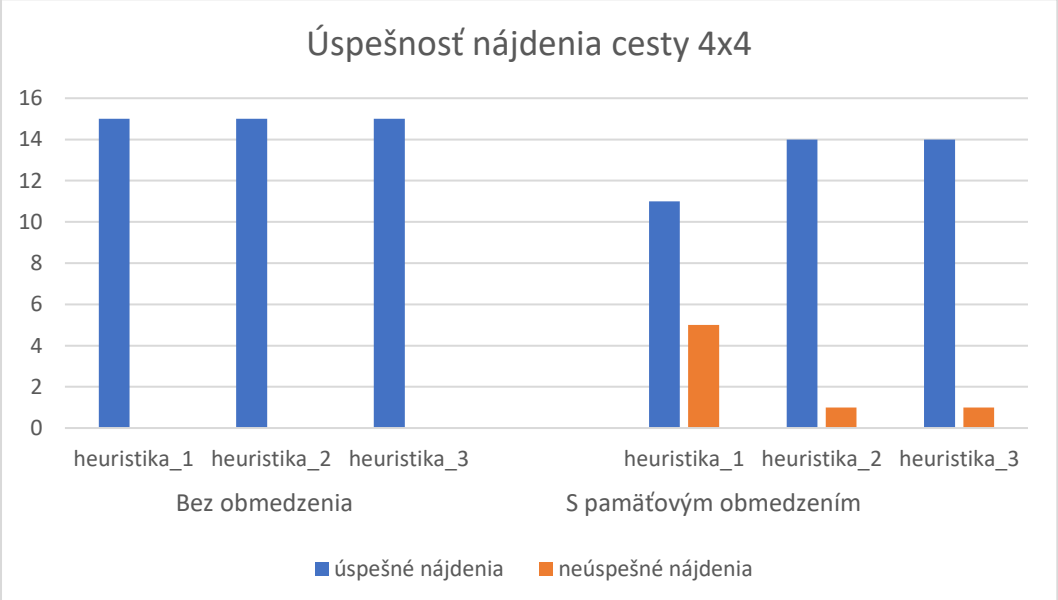
c) Úspešnosť nájdenia cesty

Taktiež ma zaujímalo, aké dopady bude mať zmenšenie maximálnej veľkosti listu vygenerovaných listov ako aj dict pre spracované stavy. Skúšal som rôzne veľkosti, a hľadal nejaký kompromis, aby bolo možné ušetriť nejakú pamäť a zároveň aby bol program schopný nájsť riešenie v rovnakom čase.

Maximálnu veľkosť slovníka som nastavil na veľkosť 16*pocet_riadkov**(pocet_stlpcov-1) a veľkosť listu vygenerovaných listov na hodnotu *15*pocet_riadkov**pocet_stlpcov.*V grafe nižšie si môžete všimnúť dosiahnuté výsledky.

Najväčší problém robilo toto obmedzenie heuristike_1 hlavne pri rozmeroch 4x4 kde ušetrená pamäť nepredstavuje výhodu, pretože sa tým rapídne znížila úspešnosť nájdenia cesty.





5. Používateľské prostredie

Môj program neobsahuje interaktívne prostredie, no je možné meniť jeho správanie na základe globálnych premenných umiestnených na začiatku programu. Je možné si nastaviť rozmery hlavolamu, a taktiež aj maximálne veľkosti listu vygenerovaných stavov a dict už spracovaných.

```
pocet_riadkov = 5
pocet_stlpcov = 5
max_velkost_spracovane = 17*pocet_riadkov**(pocet_stlpcov-1)*100
max_velkost_vygenerovane = 15*pocet_riadkov**pocet_stlpcov*100
```

Po spustení programu a úspešnom nájdení ciest sa zobrazí nasledovný výpis. Cesta sa zapíše do externého súboru *cesta_vypis.txt*

```
Zaciatocny stav:
 6   13  9   2   3
 8   24  5   22  19
 10  12  21  14  16
 11  7   1   17  m
 23  4   18  20  15

Cielovy stav:
 20  7   12  9   22
 14  24  5   1   17
 10  15  13  m   3
 6   8   19  4   2
 18  16  23  11  21

heuristika 3
-----------------------
hodnota heuristiky: 80.5
1.3982 sekund
pocet iteracii: 6041
hlbka: 1250
Cesta bola nájdená!!!

heuristika 2
-----------------------
hodnota heuristiky: 70
3.5126 sekund
pocet iteracii: 7983
hlbka: 570
Cesta bola nájdená!!!
```

```
heuristika 1
-----------------------
hodnota heuristiky: 21
16.3066 sekund
pocet iteracii: 18462
hlbka: 1986
Cesta bola nájdená!!!
```

6. Rozšíriteľnosť a optimalizácia

Môj program sa mi už pár krát podarilo zrýchliť, vďaka čomu sa celkový čas potrebný na vykonanie znížil na menej ako polovicu oproti prvej verzii.

Prvým krokom bolo odstránenie funkcie deepcopy čo zrýchlilo celý program o štvrtinu. Namiesto toho som použil funkciu map(), ktorá sa mi osvedčila ako rýchlejšia ako ručné kopírovanie. Funkcia je použitá pri kopírovaní stavu starého uzla do nového.

Ďalej som zmenil princíp vyhodnocovania možných pohybov. Na začiatku program generoval všetky prípustné pohyby, vytvoril ich uzly a následne sa pozeral do už spracovaných uzlov a vkladal do listu vygenerovaných. Tento spôsob bol neefektívny a list vygenerovaných obsahoval množstvo duplikátov. V súčasnej verzii si už na začiatku, len raz, zistím pozíciu prázdneho políčka a na základe toho zavolám funkciu pre vykonanie pohybu. V tejto funkcii sa ešte overuje, či sa daný stav nenachádza v žiadnom liste, a ak nie tak až potom vytvorí nový uzol a vloží do listu.

Ďalšie možné vylepšenia by sa dali realizovať hlavne na spôsobe ukladania vygenerovaných uzlov, kde by stačilo ukladať len stavy do hash tabuľky a uzly vytvárať až pri vkladaní do dict navštívených. Program by sa tým optimalizoval z časového aj pamäťového hľadiska.

Rýchlosť programu by bolo taktiež možné zrýchliť programovaním ho v jazyku C

Program zvláda aj iné rozmery hlavolamov ako 3x3. Pri rozmere 5x5 dokáže nájsť cestu do 10 sekúnd pri druhej a tretej heuristike. Prvej to trvá dlhšie a nie vždy je schopný nájsť cestu. Taktiež som skúšal aj veľkosť 6x6 no na nájdenie riešenia boli potrebné 4 minúty. A aj to sa zvládlo len pomocou heuristiky_3