SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta informatiky a informačných technológií
Ilkovičova 2, 842 16 Bratislava 4

Zadanie č. 2 – 8 hlavolam

D) Použite algoritmus lačného hľadania, porovnajte výsledky heuristík 1. a 2

UMELÁ INTELIGENCIA

Róbert Junas

FIIT STU

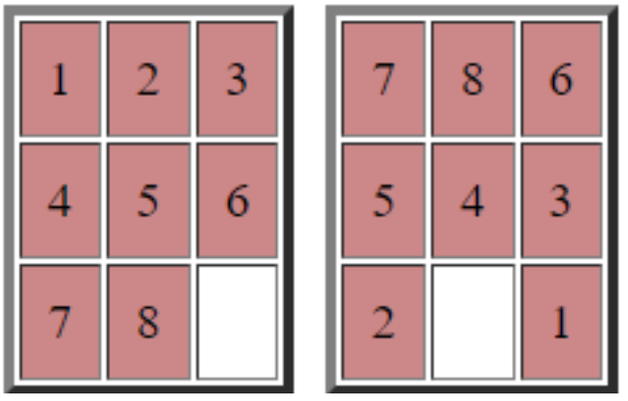
Cvičenie: Utorok 16:00

25.10.2021

Róbert Junas, 102970

1. Zadanie

Našou úlohou je nájsť riešenie 8-hlavolamu. Hlavolam je zložený z 8 očíslovaných políčok a jedného prázdneho miesta. Políčka je možné presúvať hore, dole, vľavo alebo vpravo, ale len ak je tým smerom medzera. Je vždy daná nejaká východisková a nejaká cieľová pozícia a je potrebné nájsť postupnosť krokov, ktoré vedú z jednej pozície do druhej. Použite algoritmus lačného hľadania, porovnajte výsledky heuristík 1. a 2.



Niektoré z algoritmov potrebujú k svojej činnosti dodatočnú informáciu o riešenom probléme, presnejšie odhad vzdialenosti od cieľového stavu. Pre náš problém ich existuje niekoľko, môžeme použiť napríklad:

1. Počet políčok, ktoré nie sú na svojom mieste
2. Súčet vzdialeností jednotlivých políčok od ich cieľovej pozície
3. Kombinácia predchádzajúcich odhadov

D) Použite algoritmus lačného hľadania, porovnajte výsledky heuristík 1. a 2

2. Implementácia

Naše riešenie podporuje všetky možné platné veľkosti, resp. ak hracia plocha má aspoň dve políčka. Ďalej jedno z týchto políčok musí byť prázdne, resp. vo vstupe musí byť označené znakom 0. Samotný vstup pozostáva z veľkosti hracej plochy, počiatočného a cieľového stavu. Každé políčko stavu je na vstupe rozdelené čiarkou a obsahuje hodnoty od 0 po Veľkosť-1. Jeden zadaný hlavolam by vyzeral nasledovne: <počet riadkov>x<počet stĺpcov>=<počiatočný stav>=<cieľový stav>. Napr.:

- 3x3=1,5,2,7,0,4,6,3,8=1,2,3,4,5,6,7,8,0
- 2x6=0,1,2,3,4,5,6,7,8,9,10,11=2,3,4,5,9,11,1,6,0,7,8,10

2.1. Spustenie

Program bol vytvorený v programovacom jazyku **Python 3.10.0** za pomoci knižníc:

- Queue

- Time
- Tkinter
- Colorama
- Gc
- sys

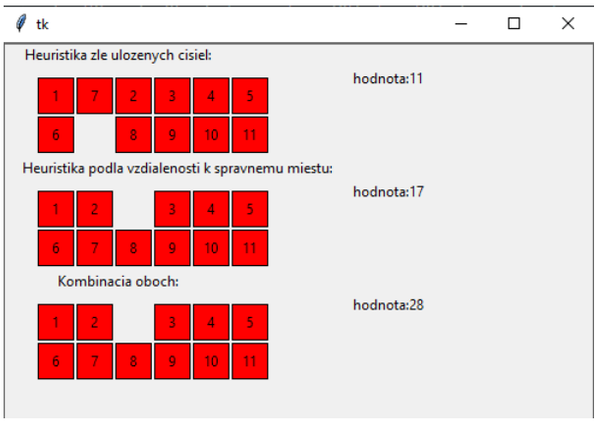
Z týchto knižníc je potrebné nainštalovať **Colorama**. Navyše potrebujeme mať vstupný súbor s hlavolamami **stavy.txt.** Program sa skladá z dvoch súborov **main.py**, kde sa rieši zadanie a v priečinku src/ je **gui.py s** kódom pre vizualizáciu. Program spustíme pomocou príkazu:

Py main.py

Program vypisuje výsledky do konzole a ak bolo pre niektorý rámec špecifikované, že má byť vykreslený, tak ho vykresľujeme pomocou tkinter. Ďalej ak bolo špecifikované rozmedzie v aké hlavolamy chceme porovnať, tak vytvoríme aj graf a uložíme ho do priečinku grafy. Tieto funkcie a ich použitie je vysvetľované v ďalšej kapitole.

2.2. Formát a funkcie vo vstupnom súbore

Formát v akom sú zadané hlavolamy sme si už prebrali, ale vo vstupnom súbore môžu byť komentáre začínajúce znakom # a tieto na vstupe vynechávame ako aj prázdne riadky. Ďalej je v súboroch aj vstavaná funkcionalita začínajúca sa znakom !. Máme funkciu !block, ktorá zastaví vykonávanie hlavolamov dokým nestlačíme enter. Ďalšou funkciami je !average, ktorý končí !endaverage a medzi nimi sú hlavolamy. Táto funkcia spraví priemer nad informáciami (dĺžka cesty, počet vytvorených a spracovaných uzlov a času hľadania) na hlavolamoch, ktoré pokrýva nie je však možné vnárať tieto funkcie do seba. Potom máme funkciu !skip a !endskip, ktorá vymedzuje hlavolamy, ktoré chceme preskočiť. Medzi ďalšie funkcie patrí !show, ktorý vizuálne zobrazí pomocou knižnice tkinter nájdené riešenie. V zobrazení si pomocou klávesy medzera preklikávame jednotlivé kroky.



2.3.Príklad výpisu

Na prvom riadku sa zobrazia rozmery hracej plochy a aj počiatočný stav. Ďalej nasledujú výsledky vyhľadávania. Vypisuje sa čas za aký bolo riešenie nájdene ako aj kroky, ktoré musíme vykonať, aby sme sa dostali ku výsledku. Potom sú 3 riadky, kde je počet vytvorených a spracovaných uzlov ako aj hĺbka do cieľového stavu.

```
==== 2 x 3 = [0, 1, 2, 3, 4, 5] ====
Hueristika 1. podla poctu zle ulozenych cisiel:
vysledok najdeny za: 0.002991 s
---- ('vysledok =', [3, 4, 5, 0, 1, 2]) -----
 0 -> hore -> vlavo -> dole -> vlavo -> hore -> vpravo -> vpravo -> dole -> vlavo -> hore -> vlavo -> dc


pocet vytvorenych uzlov:  58
pocet spracovanych uzlov:  42
pocet uzlov ku vysledku:  22 ( 21 - hlbka)

Hueristika 2. podla vzdialenosti k spravnemu miestu:
vysledok najdeny za: 0.003989 s
---- ('vysledok =', [3, 4, 5, 0, 1, 2]) -----
 0 -> hore -> vlavo -> dole -> vlavo -> hore -> vpravo -> dole -> vpravo -> hore -> vlavo -> dole -> vpr


pocet vytvorenych uzlov:  58
pocet spracovanych uzlov:  43
pocet uzlov ku vysledku:  26 ( 25 - hlbka)

Hueristika 3. kombinacia oboch:
vysledok najdeny za: 0.003988 s
---- ('vysledok =', [3, 4, 5, 0, 1, 2]) -----
 0 -> hore -> vlavo -> dole -> vlavo -> hore -> vpravo -> dole -> vpravo -> hore -> vlavo -> dole -> vpr


pocet vytvorenych uzlov:  56
pocet spracovanych uzlov:  42
pocet uzlov ku vysledku:  26 ( 25 - hlbka)
```

```
================================================
Priemery:
Heuristika 1 .
dlzka cesty: 58.833
pocet vytvorenych: 729.833
pocet spracovanych: 445.333
cas: 0.402
------------
Heuristika 2 .
dlzka cesty: 52.167
pocet vytvorenych: 210.667
pocet spracovanych: 125.167
cas: 0.029
------------
Heuristika 3 .
dlzka cesty: 54.167
pocet vytvorenych: 284.333
pocet spracovanych: 167.833
cas: 0.061
------------
================================================
```

2.4.Štruktúry

V programe máme dve štruktúry. Jednu na reprezentovanie uzla a jednu na reprezentovanie samotného stavu hlavolamu.

```python
def __init__(self,stav : Stav ,parent, operacia,mode, final : Stav) -> None:
    self.stav = stav
    self.parent = parent
    self.operacia = operacia    #cislom reprezentovana vykonana operacia
    self.heuristika = 0
    if mode == 0:
        self.heuristika = self.vypocitatVzdialenost(final)  # heuristika 1
    elif mode == 1:
        self.heuristika =  self.vypocitatZleUlozene(final)# heuristika 2
    else:
        self.heuristika =  self.vypocitatZleUlozene(final) + self.vypocitatVzdialenost(final)# heuristika 3
```

V štruktúre Uzol si pamätáme štruktúru stavu, rodiča uzla, vykonanú operáciu, aby sme vedeli rýchlo vykonštruovať cestu (ako sa pohybovať), aby sme dostali výsledok. Ďalej si pamätáme hodnotu vybratej heuristiky, aby nemuseli byť neustále počítané. Heuristiku vyberáme podľa parametru mode. Hodnoty parametru mode sú:

- 0 – súčet od správnych pozícií
- 1 – počet zle uložených
- 2 – súčet oboch heuristík

Uzol má metódy `vypocitatZleUlozene(self,finalnyStav)`, `vypocitatVzdialenost(self,finalnyStav)` na výpočet heuristík a funkciu `getHeuristika(self)` na získanie tejto hodnoty pre doplňujúcu funkciu tkinteru. Ďalej má metódu `comp(self, finalnyStav : Stav)` na porovnanie svojho stavu s iným stavom (použité iba pre porovnanie s cieľovým stavom). Poslednou funkciou je `__lt__(self,other)`, ktorá preťažuje operátor < aby sme mohli použiť priorityQueue.

```
class Stav:
    def __init__(self,cols : int, rows : int, data):
        self.rows = rows
        self.cols = cols
        self.matrix = data
```

Štruktúra Stav sa skladá z troch atribútov: počtu riadkov a stĺpcov a samotného stavu, ktorý sa reprezentuje ako jednoduché jednorozmerné pole. Stav má nastavených niekoľko getter-ov, ktoré vracajú počet riadkov, stĺpcov, veľkosť hracej plochy, kópiu hracieho poľa ako aj pole samotné. Ďalej má funkcie na porovnávanie veľkostí hracej plochy

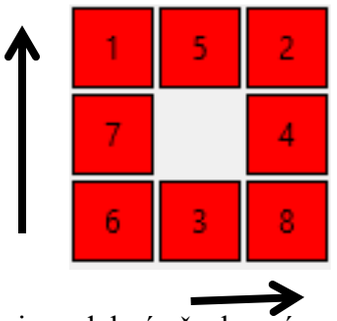
2.5. Riešenie

Najprv sa načítajú vstupy zo súboru stavy.txt, spracujú sa aby s nimi vedel program pracovať. Každý spracovaný riadok sa uloží do zoznamu, z ktorého v cykle vyberáme stavy a funkcie.

Po ukončení načítavania vstupov sa spustí hľadanie riešenie všetkými 3 heuristikami. Spustí sa hľadanie riešenie najprv pre heuristiku „počet zlých pozícií“. Ďalej sa pustí hľadanie cez vzdialenosti ku správnym pozíciám a nakoniec sa pustí hľadanie kombináciou predchádzajúcich dvoch.

Výpočet zlých pozícii (heuristika 1.) je jednoduchý, nakoľko sa prechádza stavom a iba sa spočíta počet políčok, ktoré sú na inom mieste ako majú byť v cieľovom stave. Za to výpočet vzdialeností (heuristika 2.) je komplikovanejší, nakoľko si stav je uložený ako jednorozmerné pole, tak musíme vykonať operácie na zistenie riadka a stĺpca, v ktorom sa číslo nachádza a kde sa má v cieľovom stave nachádzať. Následne si urobíme manhattanskú vzdialenosť. Výsledkom heuristiky je súčet manhattanskej vzdialenosti všetkých políčok. Pri obidvoch

heuristikách vynechávame medzeru. Príklad výpočtu manhattanskej vzdialenosti pre políčko 3., ktorého správna pozícia je v pravom hornom rohu (0,2) a aktuálna pozícia je (2,1) teda vzdialenosť bude $|2-0|+|1-2|=3$. Heuristika 3. je súčet oboch heuristík.



Hlavná funkcia hľadania je podobná všeobecnému riešeniu problému, resp. má takéto kroky:

1. Vytvor počiatočný uzol a vlož ho do prioritného radu,
2. Ak je prioritný rad prázdny tak sme prešli všetky možnosti a teda hlavolam nemá riešenie – vráť hodnotu -1
3. Vyber uzol z prioritného radu, taký aby mal najnižšiu hodnotu podľa heuristiky, ktorú práve robíme,
4. Ak tento uzol predstavuje cieľový stav, pridaj ho do spracovaných, skonči s úspechom – vráť uzol, počet vytvorených a spracovaných uzlov,
5. Ak ho nepredstavuje, tak vygeneruj všetky možné pohyby,
6. Skontroluj či uzol s rovnakým stavom neexistuje buď v spracovaných alebo ešte nespracovaných,
7. Ak sa nenachádza tak vypočítaj heuristiky a pridaj ho do prioritného radu, (kroky 6 a 7 vykonaj pre všetky nové stavy)
8. Pridaj vybratý uzol do spracovaných a vráť sa na druhý krok.

Riešenie má štyri operátory: hore, dole, vpravo, vľavo. Každý pohyb vytvorí nový objekt Stavu, na ktorom boli vymenené políčka podľa vykonaného pohybu. Operátory pohybu majú ako parametre stav, ktorý upravujú a index na ktorom sa nachádza prázdne miesto aby ho nemusela hľadať vždy každá operácia.

```
def hore(stav : Stav, indexPraz : int):
    x,y = stav.getXY()
    nStav = stav.getMatrixCopy()
    #ak na hracej ploche je pod prazdnym miestom este policko
    #resp. ak index prazdenho miesta + dlzka riadka je mensia ako velkost plochy
    if indexPraz + x  < x*y:
        #vymena prazdneho miesta s polickom
        nStav[indexPraz],nStav[indexPraz + x] = nStav[indexPraz + x], nStav[indexPraz]
        novy = Stav(x,y,nStav)
        return novy
    return -1
```

Figure 1 pohyb políčkom hore

2.6. Zložitosť

Výpočet heuristík sa vždy robí nad konštantným počtom elementov, resp. veľkosť hracej plochy nikdy nebude taká obrovská aby nejako spomalila hľadanie riešenia. Teda môžeme predpokladať, že výpočet bude O(1).

Lačné hľadanie svojim správaním sa podoba hľadanie do hĺbky, pretože pokračuje po jednej ceste pokým má najvýhodnejší výsledok z heuristickej funkcie. Ak sa ukáže, že uzol nemá potomkov, ktorý sú výhodnejší ako potomkovia iných uzlov, tak začne prehľadávať ich. Preto aj časová zložitosť je $O(b^d)$, kde *b* je faktor vetvenia a *d* je maximálna hĺbka vetvenia. Priestorová zložitosť je rovnaká ako časová, nakoľko si pamätáme všetky možné stavy [1]. Každý stav môže mať maximálne 4 potomkov, teda pre náš problém je o zložitosti $O(4^d)$.

Prioritný rad nespracovaných uzlov rovno vyberá podľa nami zvolenej heuristiky najmenšiu hodnotu. Implementácia je priamo vstavaná do štandardných knižníc jazyka python a jeho zložitosť je vkladania a vyberania je O(log(n))

3. Testovanie

V prvom bloku testov sme sa rozhodli porovnať výsledky nášho riešenia (výsledky dvoch heuristík) s riešeniami prehľadávania do šírky [2]. Rozhodli sme sa neporovnávať časy, nakoľko riešenie do šírky bolo implementované v rýchlejších jazykoch ako je nami zvolený. V tabuľke 1. je počet vytvorených a spracovaných uzlov.

Veľ.	Heuristika 1.			Heuristika 2.			Do šírky		
	Hĺbka	Vyt.	Sprac.	Hĺbka	Vyt.	Sprac.	Hĺbka	Vyt	Sprac.
2x3	21	58	43	25	58	44	21	360	359
2x4	104	850	602	82	304	200	36	20160	20159
3x3	137	929	562	45	116	70	31	181440	181438
2x5	505	12481	8451	209	1456	937	55	1814400	1814398
3x4	327	16649	10371	137	3443	1861	315	239500800	239500782
2x6	1060	70132	54282	301	3929	2408	80	239500800	239500798

Tabuľka 1

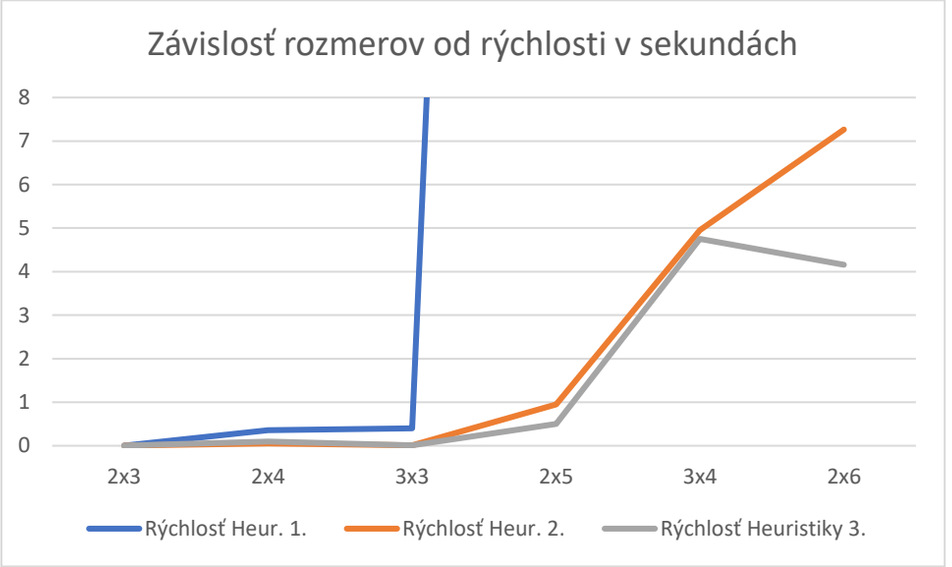
Ako je vidieť tak prehľadávanie do šírky nájde optimálnejšie výsledok, ale pamäťovo naše riešenie lačným hľadaním bolo optimálnejšie, nakoľko sa nerozvíjajú všetky uzly, ale iba tie, ktoré majú najväčšiu pravdepodobnosť, že vedú ku správnemu výsledku. Nad rovnakými hlavolamami sme vykonali aj porovnanie so všetkými troma heuristikami (Tabuľka 2.) Rýchlosti hľadania sú uvedené v tabuľka 3.

Veľ.	Heuristika 1.			Heuristika 2.			H1 + H2		
	HĺbkaVyt.		Sprac.	HĺbkaVyt.		Sprac.	HĺbkaVyt.		Sprac.
2x3	21	58	43	25	58	44	25	56	43
2x4	104	850	602	82	304	200	110	451	301
3x3	137	929	562	45	116	70	49	112	68
2x5	505	12481	8451	209	1456	937	185	1072	692
3x4	327	16649	10371	137	3443	1861	315	3405	1839
2x6	1060	70132	54282	301	3929	2408	252	2890	1824

Tabuľka 2

Veľ.	Rýchlosť Heur. 1. [s]	Rýchlosť Heur. 2. [s]	Rýchlosť Heuristiky 3. [s]
2x3	0.0019	0.0029	0.0029
2x4	0.3590	0.0548	0.0997
3x3	0.3979	0.0079	0.0079
2x5	75.502	0.9474	0.5026
3x4	145.3843	4.9507	4.7522
2x6	3027.37	7.2645	4.1588

Tabuľka 3



Ako môžeme vidieť, tak heuristika 2. je rýchlejšia a celkovo rozširuje menej uzlov ako heuristika 1. Kombinácia oboch mala negatívny vplyv na rýchlosť heuristiky 2. Dôvodom môže byť ten fakt, že mnohokrát čísla, ktoré sú uložené na správnych miestach, je ich potrebné presunúť najprv na zlú pozíciu, aby sme iné číslo priblížili ku správnej pozícií a heuristika 1. sa snaží takým krokom vyhnúť. Preto aj heuristika 1. má taký problém s riešením hlavolamov, ktoré sú pohybovo obmedzené (resp. nedá sa vyhnúť pohybom už dobre umiestených čísel), napr. hlavolamy 2x5, kde našlo riešenie až v hĺbke 505 a pri riešení 2x6 už heuristike trvalo

skoro hodinu s porovnaním s inými výsledkami. Heuristika 3. našla riešenia častejšie hlbšie ako Heuristika 2., ale na druhú stranu bola o niečo rýchlejšia pri väčších hlavolamoch.

Ďalej sme testovali iba s hracou plochou veľkosti 3x3, kde sme si pomocou stránky vygenerovali 8-hlavolamy [3]. Pri týchto hlavolamoch je výsledným stavom pole [1,2,3,4,5,6,7,8,0]. Celkovo sme ich otestovali 6 a potom sme ešte raz prešli, ale vymenili sme počiatočné uzly s cieľovými hlavolamy sú v tabuľke 4. (v tabuľke 5. je čas hľadania)v poradí ako v súbore teda:

1. 1,5,2,7,0,4,6,3,8
2. 1,2,3,4,0,6,7,5,8
3. 1,6,2,4,3,8,7,0,5
4. 2,1,3,4,7,6,5,8,0
5. 5,0,2,8,4,7,6,3,1
6. 8,6,7,2,5,4,3,0,1

č.	Heuristika 1.			Heuristika 2.			H1 + H2		
	HÍbka	Vyt.	Sprac.	HÍbka	Vyt.	Sprac.	HÍbka	Vyt.	Sprac.
1	29	85	50	73	319	189	51	264	159
2	85	1284	775	43	138	78	71	254	150
3	10	134	80	10	23	12	10	28	15
4	31	773	470	51	234	140	45	366	220
5	74	361	212	76	307	184	102	680	394
6	124	1742	1085	60	243	148	46	114	69
Priem.	58	729	445	52	210	125	54	284	167

Tabuľka 4

Č.	Rýchlosť Heur. 1. [s]	Rýchlosť Heur. 2. [s]	Rýchlosť Heuristiky 3. [s]
1	0.0039	0.0498	0.0349
2	0.6891	0.0129	0.4388
3	0.0159	0.0009	0.0009
4	0.3	0.0299	0.0678
5	0.0668	0.0478	0.2084
6	1.3374	0.0319	0.0089
Priem.	0.402	0.029	0.061

Tabuľka 5

Ako je vidieť z dát, tak použitie jednej alebo druhej heuristiky nemalo obrovský vplyv na hĺbku výsledku, nakoľko na niektorých hlavolamoch excelovala heuristika 1. a na iných heuristika 2.. Všetky 3 heuristiky našli v priemere riešenia približne v rovnakej hĺbke. Na druhú stranu. Heuristika 2. bola oproti iným priemere, a aj vo väčšine prípadov rýchlejšia. Celkovo Heuristika 2. rozširovala menej uzlov ako ostatné, preto bola aj rýchlejšia.

Tabuľku pre opačne smery tu nezahrnujeme, ale prikladáme výstup priemerov pre tento test:

```
============================================
Priemery:
Heuristika 1 .
dlzka cesty: 73.5
pocet vytvorenych: 904.833
pocet spracovanych: 547.5
cas: 0.916
------------
Heuristika 2 .
dlzka cesty: 58.5
pocet vytvorenych: 336.5
pocet spracovanych: 200.333
cas: 0.131
------------
Heuristika 3 .
dlzka cesty: 54.5
pocet vytvorenych: 364.333
pocet spracovanych: 217.167
cas: 0.143
------------
============================================
```

4. Záver

Projekt sa nám úspešne podarilo implementovať a je dostatočne rýchle pre všetky hlavolamy a aj sme spozorovali rozdiel medzi heuristikami. Ako rozšírenie navrhujeme použitie inej kombinácie heuristík, namiesto sčítavania, ktoré hľadalo vo väčšine prípadov rýchlejšie, ale hlbšie ako druhá heuristika, použitie dvoch kľúčov, podľa ktorých sa vyberá z prioritného radu, napr. najprv sa zoradí podľa heuristiky 2. a ak nastane, že viac uzlov má rovnakú hodnotu tak sa vyberie uzol, ktorý má viac uzlov uložených na správnych miestach. Výhodou implementácie je, že vyhľadáva riešenie pre všetky nami zadefinované heuristiky, vypisuje postupnosť krokov aké pri nájdení riešenia použilo hľadanie pomocou heuristík. Ďalej naše riešenie podporuje aj vizualizáciu riešení , ktoré heuristiky našli, čo je výborná pomôcka na overenie korektnosti riešenia. Na optimalizáciu pamäte by sme mohli upraviť štruktúru stav aby si nemusela pamätať rozmery hracej plochy, nakoľko pre jeden hlavolam je veľkosť konštantná. Čo sa týka heuristík tak heuristika 2. bola v každom ohľade lepšia ako heuristika 1. Prvá heuristika mala problémy s riešením hlavolamov s väčšou hracou plochou. Na druhú stranu všetky heuristiky sa ukázali pamäťovo lepšie ako prehľadávanie do šírky. Na celkovú rýchlosť hľadania mal vplyv aj jazyk Python, ktorý je pomalší ako jazyky, ktoré sa kompilujú a vykonávajú sa nad nimi rôzne optimalizácie.

5. Použitá literatúra

1. NÁVRAT, Peter a kol. 2002. *UMELÁ INTELIGENCIA*. Bratislava : Slovenská technická univerzita v Bratislave, 2015. 220 s. ISBN 978-80-227-4344-0.
2. http://www2.fiit.stuba.sk/~kapustik/MN%20hlavolam.html
3. https://deniz.co/8-puzzle-solver/