SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Umelá inteligencia

Zadanie 2

Prehľadávanie stavového priestoru

Martin Melišek

Cvičenie: Streda 16:00

Cvičiaci: Ing. Ivan Kapustík

Prehľadávanie stavového priestoru

Úvod

Zadaním tohto projektu je implementovať funkčný **algoritmus lačného prehľadávania** stavového priestoru na nájdenie riešenia NxM-hlavolamu. Hlavolam je reprezentovaný stavmi, kde jednotlivé prvky sú čísla, ktoré predstavujú pozície políčok v hlavolame. Cieľom tejto úlohy je *porovnať* výsledky z rôznych **heuristík**, ktoré urýchľujú vyhľadávanie a výstupom programu bude následnosť krokov, akými sa dostaneme z počiatočného usporiadania políčok do cieľového.

Hlavnou podstatou tejto práce je implementácia funkcie:

```
lacne_hladanie(problem, heuristika)
            returns riesenie or None
```

Funkcia dostane ako vstupný parameter problém vo forme počiatočného stavu, cieľového stavu a heuristickú funkciu. Hlavolam môže byť ľubovoľnej veľkosti od *2x2* do *7x7*, vrátane obdĺžnikových hlavolamov (väčšie už neboli testované). Funkčnosť riešenia si overíme otestovaním viacerých hlavolamov a porovnáme namerané hodnoty s očakávanými výsledkami.

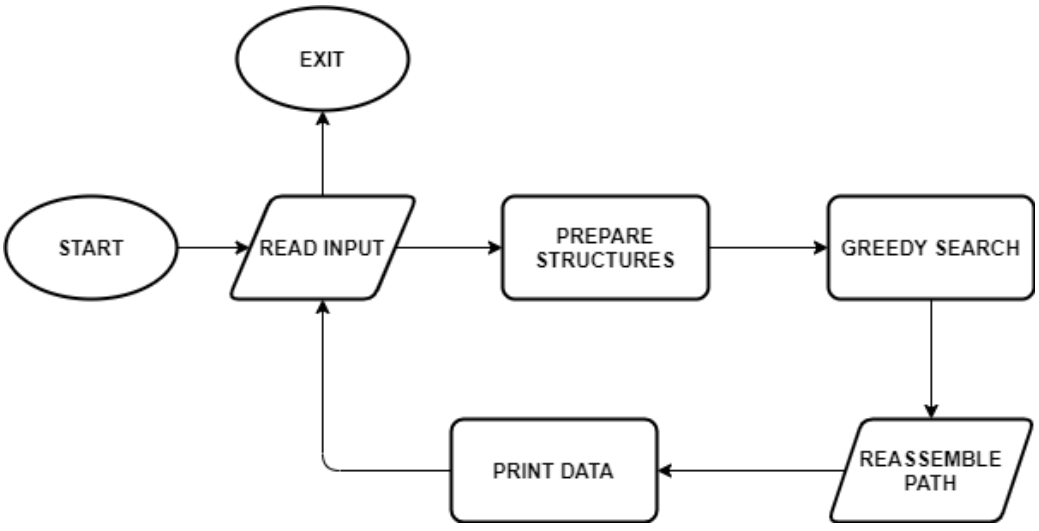
Testované heuristiky:

- Počet políčok, ktoré nie sú na svojom mieste
- Súčet vzdialeností jednotlivých políčok od ich cieľovej pozície

Blokový návrh – koncepcia fungovania riešenia

Program je rozdelených do troch častí:

1. Príprava štruktúr a načítanie vstupu
2. Lačné hľadanie
3. Výpis riešenia



Obr. 1: Základný blokový návrh programu

Implementačné prostredie

Program predstavuje konzolovú aplikáciu, ktorá bola vytvorená v programovacom jazyku **Python v3.8.5 64bit**. Spúšťa sa cez Command Line, PowerShell alebo iný CLI nástroj. Počas celého vykonania programu používame nasledujúce importované moduly:

- heapq
  - Reprezentácia binárnej haldy ako prioritného radu na vkladanie a vyberanie prvkov s najlepším výsledkom heuristickej funkcie[1]
- timeit
  - Meranie času vykonania funkcie lacne_hladanie

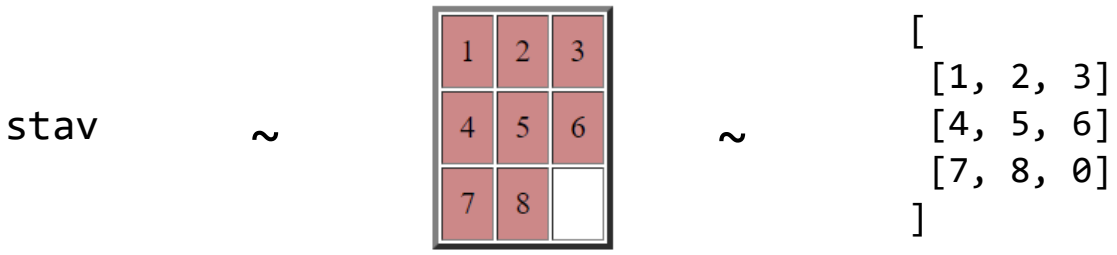
Následné zachytávanie časovej a pamäťovej efektívnosti(nie sú potrebné pri spustení):

- cProfile
  - Poskytuje možnosť zobrazenia času stráveného v jednotlivých funkciách a dobu vykonávania príkazov v programe
- memory-profiler
  - Zaznamenáva využitú pamäť spusteného procesu python skriptu[2]

Reprezentácia údajov

Rozloženie políčok v hlavolame je reprezentované ako stav. Na to aby bolo možné prehľadávať a kontrolovať tieto stavy je potrebné reprezentovať tento priestor ako vrcholy a hrany grafu. Každý jeden stav predstavuje v programe jeden uzol – reprezentovaný ako klasická štruktúra z jazyka C, pomocou objektu, teda inštancie triedy: `Node`, ktorá obsahuje nasledujúce atribúty:

- Stav
  - Stavy sú reprezentované ako 2D pole – list of lists, kde riadok a stĺpec sú konfigurácia v akom sa hlavolam v danom uzle nachádza



Obr. 2: Vizualizácia stavu[3]

  - Môžeme si všimnúť, že medzera je reprezentovaná ako 0
- Odkaz na rodičovský uzol, z ktorého bol uzol vytvorený, pre záverečné zostavenie cesty

Tieto atribúty sú povinné a bez nich by sme nikdy nedokázali nájsť riešenie. Pre nasledujúce informácie, ktoré si o každom uzle uložíme musíme byť veľmi opatrný a preto sa snažíme nájsť kompromis pre časovú a pamäťovú efektivitu. Odstránenie niektorej z nich buď zaopatrí potrebu výpočtu pri hľadaní, čím predĺži vyhľadávanie alebo na druhú stranu skráti, ale zvýši množstvo potrebnej pamäti.

- Posledný použitý operátor, ktorý vygeneroval tento stav
    - Pri vytváraní nasledovníkov a aplikovaní operátorov nemusíme počítať aký bol rodičovský operátor na to aby sme predišli aplikovaniu opačného smeru.
    - Rovnako pri zostavovaní výslednej cesty stačí prejsť v grafe smerom k rodičovi a získať hodnotu z tohto operátora bez potreby robiť ďalšie výpočty.
- Výsledok z heuristiky
    - Nakoľko používame informované hľadanie, využijeme túto hodnotu na vybratie najlepšieho stavu na spracovanie

Na ušetrenie pamäte neukladáme pozíciu medzery, ale počítame ju pre každý uzol zvlášť, vnútorné testovanie, ešte pred dokončením práce ukázalo, že tento prechod stavom a vrátenie dvojice `i,j` nezefektívnilo ani nespomalilo riešenie a program iba nabral na potrebnej pamäti.

Metódy a opis použitých algoritmov

Aplikovaný algoritmus na prehľadanie stavového priestoru a nájdenie riešenia hlavolamu je **Lačné hľadanie**, ako prioritný rad na výber ďalšieho spracovávaného stavu je použitá **binárna halda (Min Heap)**. Zoznam spracovaných (rozvitých) uzlov je uložený v **hashSete.**

Prioritný Rad – Binárna Halda – Min Heap

Binárna halda je špeciálny typ binárneho stromu. Pravidlá na udržiavanie tejto dátovej štruktúry sú menej striktné voči klasickým binárnym stromom. Implementácia haldy je prevzatá z python built-in module ***heapq***. Na zabezpečenie jednoduchšieho fungovania v programe používame tzv. wrapper triedu `MinHeap`, ktorá obsahuje rovnaké, všetky potrebné funkcie a má len jeden atribút a to samotnú haldu – klasické pole teda `List`

- `init(self, start)`
    - Vytvorí pole a vloží doňho počiatočný stav
- `insert(self, uzol)`
    - Vloženie prvku do haldy a zachovanie haldovej vlastnosti
- `pop(self)`
    - Odobratie prvku na vrchu haldy
- `isEmpty(self)`
    - stav, či je halda prázdna

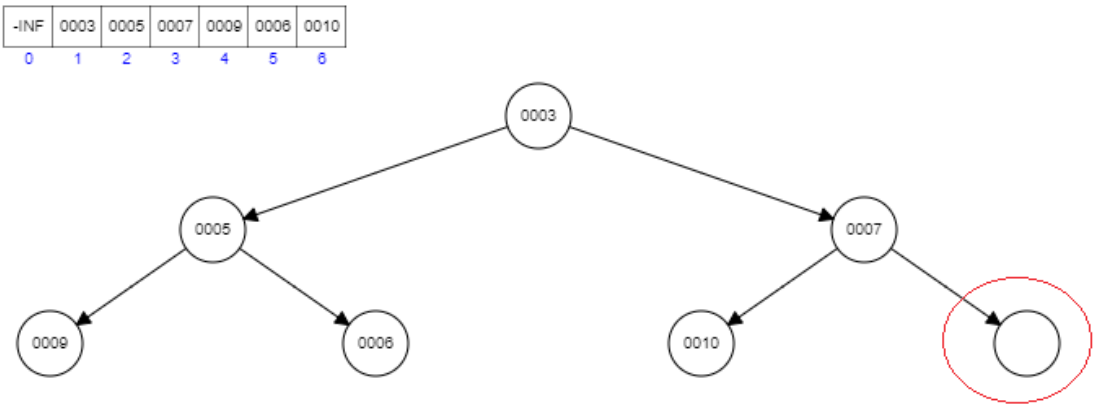
Rozbor zložitosti

Všimnime si, že ľubovoľný posun v halde je vykonávaný násobením resp. delením 2 indexu, na ktorom sa nachádzame. To znamená, že posun v tomto našom pseudo binárnom strome sa stáva byť **logaritmický**.
Nové prvky sú pridané vždy na koniec a aj keď nerobíme žiadne vyvažovanie ako pri stromoch vznikne nám **vždy** úplný binárny strom s *n* prvkami a má hĺbku ***O(log n)***. Je pravda, že ľavý podstrom neobsahuje čísla len menšie ako pravý podstrom, ale to nás nebude zaujímať, pretože podstatné je jedno hlavné pravidlo, tzv. haldová vlastnosť, ktorá je dodržaná vždy:

$$data(\boldsymbol{parent}(\boldsymbol{i})) \leq data(\boldsymbol{i})$$

Podobne pri vyberaní prvku, vždy logaritmicky opravujeme vzniknuté chyby a bonus: nájdenie min hodnoty je vždy ***O(1)***. Táto skutočnosť je dôsledkom haldovej vlastnosti a teda môžeme tvrdiť, že koreň stromu (binárnej haldy) má vždy najmenšiu hodnotu kľúča (≤ ako ostatné vrcholy).

Insert vizualizácia



Obr. 3: Vizualizácia binárnej haldy pri pridávaní prvku

Ako dáta v jednotlivých uzloch haldy si ukladáme vygenerované stavy. Na to aby sme mohli vykonávať porovnania a presúvať prvky s nižšou hodnotou do koreňa potrebujeme zaviesť *custom comparator* pre triedu `Node`.

```
def __lt__(self, other):  # custom comparator pre minHeap
    return self.cenaCiel < other.cenaCiel
```

HashSet

Spracované uzly ukladáme do štruktúry `set`, ktorý predstavuje hashovaciu tabuľku, kde neexistuje kľúč je zahashovaná value. Nie je to klasická dvojica key value ako pri slovníkoch(`dict` - hash tabuľkách) a preto do neho len pridávame pomocou `set.add()` a hľadáme cez kľúčové slovo *in*.

Pri hashovaní prvku sa snažíme zobraziť daný kľúč do rozsahu indexu hashovacej tabuľky. Po použití hashovacej funkcie a následnej kompresie môže nastať **kolízia,** ktorá vypovedá o tom, že na danej pozícií sa už nachádza iný prvok. V takomto prípade je nutné riešiť situáciu. Na riešenie kolízií sa používajú rôzne metódy napr. otvorené adresovanie alebo pomocou spájaných zoznamov. Využíva sa tzv. *Alpha faktor*, ktorý stanovuje maximálne zaplnenie tabuľky predtým ako je potrebné ju zväčšiť – najbežnejšie je to 0.5, teda polovičná veľkosť zaplnenia. Pri rozšírení je potrebné prepočítať *hash hodnotu* všetkých prvkov a znovu ich vložiť do tabuľky, ale dôležité je, že nám to postupne pomôže zabezpečiť lepší rozptyl pri väčšej vzorke dát.

Teoreticky je dosť ťažké pozerať sa na priemerné alebo najhoršie časy vkladania, resp. vyhľadávania, pretože na efektivitu algoritmu vplýva viacero faktorov; alpha faktor, faktor naplnenia, zväčšovania, hashovania, riešenia kolízií atď. Implementácia v programovacom jazyku python dosahuje priemerné vkladanie a hľadanie v čase ***O(1)***.

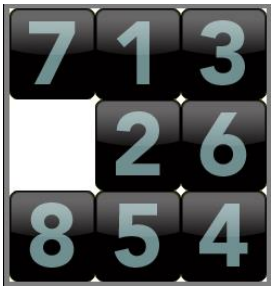
Heuristické funkcie

Na to aby sme mohli aplikovať informovaný algoritmus potrebujeme opísať použité heuristické funkcie, ktoré vedú náš algoritmus k cieľovému stavu. Je vhodné si uvedomiť, že výsledok z vybratej funkcie potrebujeme vypočítať pre každý vygenerovaný stav a toto číslo predstavuje určitú *lokálnu* hodnotu bez ohľadu na stavy, cez ktoré sme prešli predtým alebo stavy, ktoré navštívime neskôr.
Fakty o heuristických funkciách:

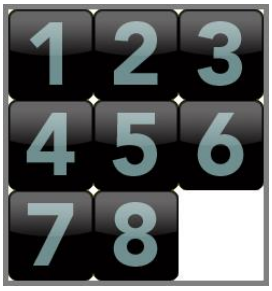
- Jedná sa len o odhady.
- Obe funkcie dostanú na vstup terajší skúmaný stav a cieľový.
- Do výpočtu nikdy neberieme ohľad na pozíciu medzery.

1. Počet políčok, ktoré nie sú na svojom mieste

Podstata heuristiky spočíva v tom, že porovnáva každé jedno políčko s cieľovým a ak sa nezhodujú inkrementuje výsledok.



Obr. 4: Počiatočný stav pre heuristiku 1[4]



Obr. 5: Cieľový stav pre heuristiku 2[4]

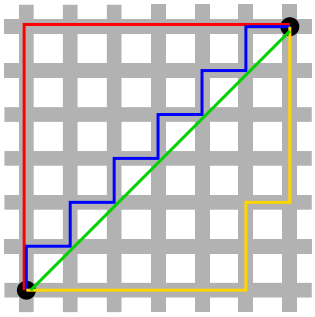
Začíname na pozícii [0][0] (políčko s hodnotou 7) nenachádza sa na správnom mieste, je za neho + 1..

$$1 + 1 + 0 + 1 + 0 + 1 + 1 + 1 \;=\; 6$$

2 políčka sú na správnom mieste 6 nie je. Takto vidíme, že rozsah odhadu bude od 1 po 8 resp. 1 $až$ $(n \cdot m) - 1$

2. Súčet vzdialeností jednotlivých políčok od ich cieľovej pozície

Táto mierne zložitejšia funkcia dokáže podať presnejší odhad nakoľko výsledky sú z väčšieho rozsahu. Na výpočet sa používa tzv. Manhattanskú vzdialenosť, ktorú počítame na základe najkratšej vzdialenosti k cieľu. Nakoľko máme povolené len 4 možné pohyby, nemôžeme počítať priamočiaro, ale pohybovať sa len po blokoch.



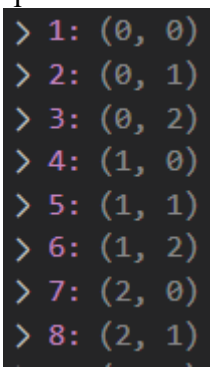
Obr. 6: Porovnanie vzdialeností Zelená – najkratšia červená, modrá, žltá - rovnaká manhattanská[5]

Ak uvažujeme o rovnakej počiatočnej a koncovej pozícií políčok z príkladu vyššie môžeme počítať nasledovne:

Políčko [0][0] má hodnotu 2, pretože vo výsledku sa nachádza na [2][0]; $|(x - i)| + |(y - j)|$
kde x a y sú hodnoty z cieľového stavu a i, j z terajšieho.

$$2 + 1 + 0 + 1 + 0 + 1 + 1 + 3 = 9$$

Tu si môžeme všimnúť, že pre túto heuristiku sa raz pred spustením hľadania oplatí vypočítať hodnoty x, y pre každé políčko.



Obr. 7: Cieľové pozície pre políčka

Definovanie operátorov

Počas tohto problému sa môžeme v hlavolame pohybovať iba do štyroch smerov;

`"VPRAVO"`, `"DOLE"`, `"VLAVO"`, `"HORE"`. Každý z operátorov je možné vykonať len za podmienky, že po zmene nevyjdeme za hranice hlavolamu.
Ak pozícia [y][x] predstavuje medzeru nie je možné vykonať ťahy v nasledujúcich prípadoch:

Operátor	Neplatná pozícia
VPRAVO	x = (šírka hlavolamu – 1)
DOLE	y = (dĺžka hlavolamu – 1)
VLAVO	x = 0
HORE	y = 0

Tab.1: Neplatné akcie pre operátory

Vo všetkých ostatných je to možné.
Vykonanie operátora vymení medzeru s políčkom, ktoré je v smere daného operátora.

Lačné hľadanie

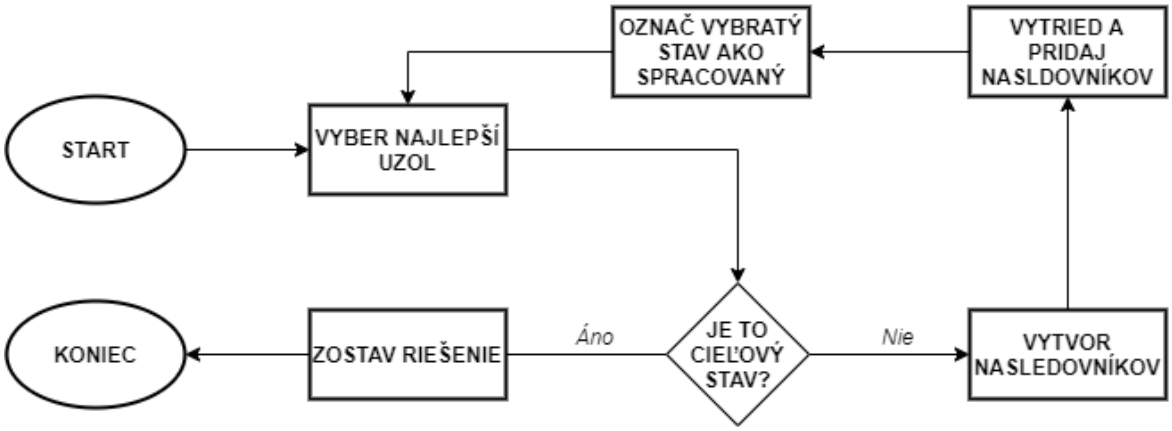
Akonáhle máme pripravené všetky dátové štruktúry a heuristické funkcie potrebné na prevedenie hľadania, môžeme sa vrátiť k cieľu nášho problému a to nájdenie riešenia k hlavolamu. Lačné hľadanie rozvíja uzly podľa výsledku heuristickej funkcie, ktorá je pre cieľ rovná 0.

Inicializácia

Pokiaľ používame na hľadanie heuristiku 2, najprv predvypočítame cieľové pozície políčok. Je to z toho dôvodu aby sme nemuseli pri každom novom stave robiť to isté – opätovne hľadať číslo v cieľovom stave. Takto si držíme jednu globálnu tabuľku (pozri obr.7) s pozíciami a iba sa do nej pozrieme a dopočítame hodnotu.

Prvým krokom je vytvorenie `haldy` `minHeap` a `hashSetu` `spracovaneStavy`. Obe štruktúry sú zatiaľ prázdne. Do haldy vložíme počiatočný uzol – s názvom `start`.

Hlavný cyklus prehľadávania



Obr. 8: Blokový návrh pre lačné hľadávanie

V halde máme iba jeden prvok – start, ktorý z nej vyberieme pomocou funkcie `pop()`. Pozrieme sa, či sa nejedná o cieľový prvok a potom ho začneme rozvíjať. Rozvinutie jedného stavu znamená vytvorenie nasledovníkov pomocou operátorov. Funkcia dostane stav a vráti zoznam vytvorených nasledovníkov:

```
vytvorNasledovnikov(parent, heuristika, ciel, operatory)
                    returns nasledovnici : List
```

Nasledovníkov vytvárame cez operátory. Každý z operátorov, ak je to možné zamieňa políčko v jeho smere s medzerou. Pozíciu medzere musíme vždy nájsť, nakoľko si ju neukladáme. Tu si musíme ešte uvedomiť, že je vhodné nevytvárať stav, z ktorého sme práve prišli – teda rodičovský uzol. Tomuto neželanému správaniu predídeme, tak, že zavoláme funkciu: `opacnySmer(operator, parent.lastOperator) returns True or False`

Táto funkcia vráti *True,* ak by sa jednalo o prechod naspäť do opačného smeru. Opačný smer pre každý operátor je zadefinovaný nasledovne (pozri tab. 2): →

Tab.2: Opačný smer pre operátory

Operátor	Opačný smer
VPRAVO	VLAVO
VLAVO	VPRAVO
HORE	DOLE
DOLE	HORE

Následne je potrebné skontrolovať, či je možné aplikovať operátor (pozri tab. 1). Ak nie je, tak ideme okamžite na ďalší operátor, ale ak je, môžeme ho vykonať. Skopírujeme rodičovský stav, vymeníme potrebné políčka a vytvoríme nový uzol s novým stavom. Pre tento nový stav vypočítame heuristiku a zaradíme ho do zoznamu nasledovníkov. Maximálny počet vygenerovaných nasledovníkov je *3,* pokiaľ sa políčko nenachádza pri stene, vtedy len *2* a ak je v rohu tak *1*. Počítame s opačným stavom(okrem prvého).

Potom ako sme vygenerovali nasledovníkov môžeme považovať rodičovský stav za **spracovaný**, čiže rozvinutý a zaradíme ho do `hashsetu` spracovaných uzlov.

Nasledovníci ešte nie sú v našom priority queue, pretože ich musíme odfiltrovať, nakoľko nechceme pridať do haldy stavy, ktoré sme už spracovali. Koniec koncov, ak by sme to neurobili, mohli by sme sa točiť donekonečna.

`vytriedNasledovnikov(nasledovnici, spracovaneStavy, minHeap)`

Na to aby sme overili, či sa nový stav nachádza v `hashsete` musíme z neho spraviť štruktúru, ktorá sa dá *zahashovať*. Nakoľko polia sú v pythone mutable(a teda unhashable) použijeme `Tuple`. Tuple predstavuje usporiadanú nemennú n-ticu. Prekonvertujeme list of lists na tuple of tuples a hľadáme. Každý stav , ktorý sme už spracovali odstránime zo zoznamu a ostatných **insertneme** do `haldy`.

Tento jednoduchý proces opakujeme dokým, z haldy nevyberieme **stav**, **ktorý predstavuje cieľ**. Pokiaľ halda ostane prázdna a už neexistujú uzly, ktoré by sme mohli spracovať cyklus skončíme a môžeme prehlásiť, že sme skontrolovali všetky možné stavy $\frac{(n \cdot m)!}{2}$. Keďže ani jeden z nich neviedol k cieľovému stavu, **hlavolam nemá riešenie.**

Zostavenie riešenia

V prípade, že sme našli riešenie, na záver je potrebné zostaviť postupnosť krokov, ktorými sa sme sa dopracovali až k cieľovému stavu. Toto by nemalo predstavovať problém, nakoľko v každom uzle máme uložený naposledy použitý operátor a jeho rodičovský uzol. Postupne sa posúvame smerom až k koreňovému počiatočnému stavu a pridávame do zoznamu naposledy použité operátory.

`zostavRiesenie(poslednyUzol) returns riesenie: List`

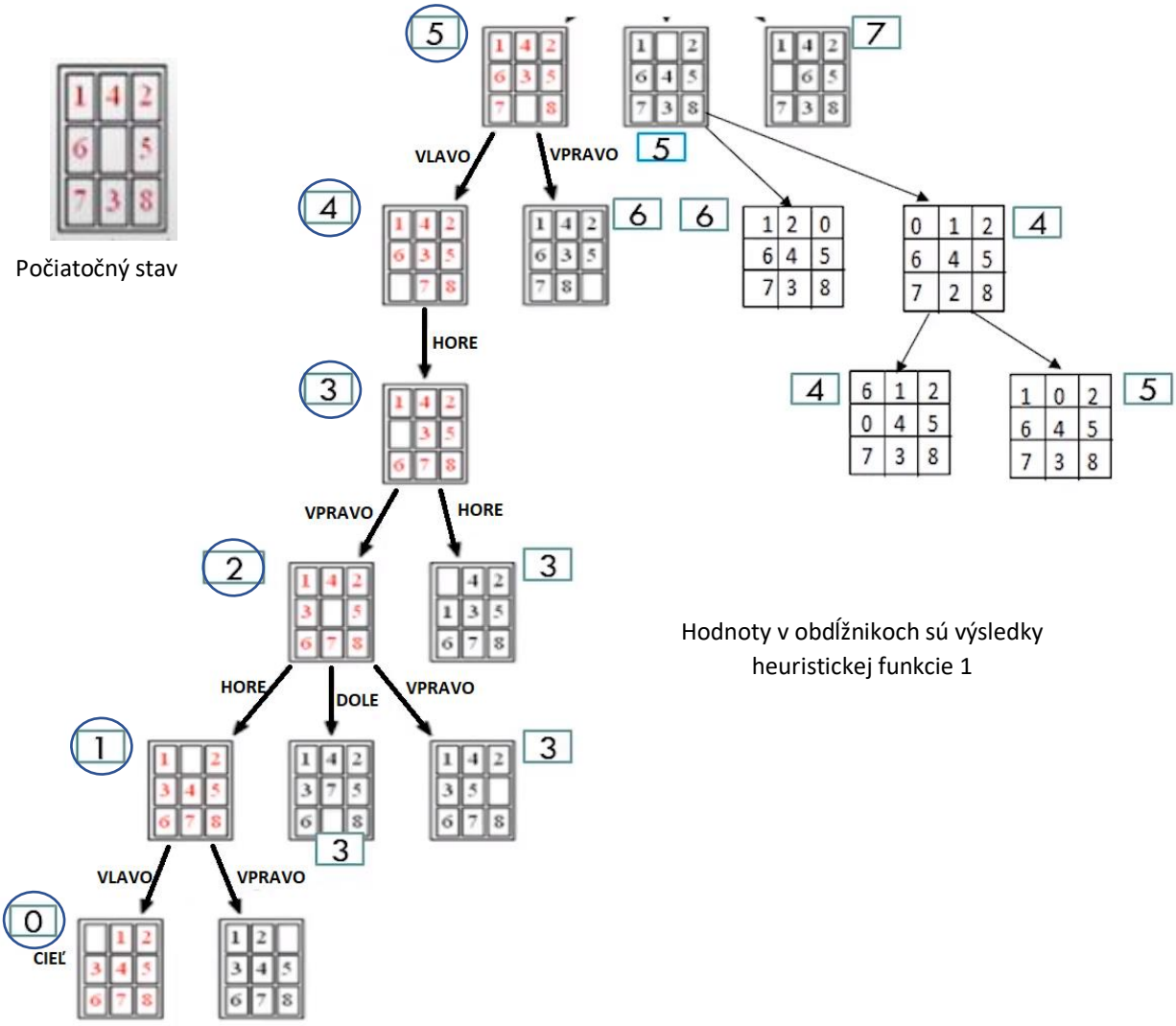
Rozbor zložitosti riešenia

Pri diskusií o zložitosti lačného riešenia musíme brať do úvahy **faktor vetvenia**, ktorý označíme b. Tento faktor vetvenia predstavuje počet vygenerovaných nasledovníkov (pozri funkciu vytvorNasledovnikov) – nových stavov každým jedným spracovaným stavom. Ďalej potrebujeme brať do úvahy **maximálnu hĺbku priestoru hľadania** m. Pokiaľ by sme potrebovali spracovať všetkých nasledovníkov bola by zložitosť riešenia $\boldsymbol{O(b^m)}$[6]. Pamäťová zložitosť je rovnaká ako časová, nakoľko si potrebujeme uložiť všetky vygenerované stavy.

Celkovo je ťažko hovoriť o priemernej zložitosti, pretože algoritmus môže **náhodou** nájsť riešenie relatívne skoro alebo môže zablúdiť a hľadať vo väčšej hĺbke ako sa nachádza cieľ. Použitie dobrej heuristickej funkcie zlepšuje priemerný čas, ale nájdeme len *nejaké* riešenie a nie najkratšie, nakoľko sa orientujeme podľa **lokálneho minima**.

Diagram vývoja riešenia

Toto je príklad, toho ako algoritmus vyberá z haldy nasledujúce uzly, ktoré potom rozvíja. Môžeme si všimnúť, že v tomto prípade algoritmus šiel za lokálnym minimom 4, ktoré sa mohlo nachádzať aj v rôznych stavoch. Eventuálne v hĺbke ~6 z haldy vybral stav, ktorý sa zhodoval s cieľovým riešením, čiže podľa heuristickej funkcie mal hodnotu **0**.



Obr. 9: Diagram vývoja riešenia[7]

Používateľské rozhranie

Ovládanie programu spočíva v interaktívnom režime, konkrétne odpovedáme na výzvy programu, pomocou klávesnice píšeme vstup a na obrazovke sa objavuje výstup, poprípade môžeme presmerovať výstup do súboru.
Hlavný cyklus interaktivity, ktorý je možné vypnúť pomocou príkazu „q“:

1. Voľba hlavolamu
    a. Výber predvygenerovaného; veľkosti:
    2x7, 3x2, 3x3, 4x3, 5x2, 5x5, 6x6, 7x7, 9x3 a 3x3 neriešiteľné
    b. Vlastný vstup zo súboru start.txt a ciel.txt
2. Výber heuristiky – [1] alebo [2]
3. Výpis riešenia [y] alebo [n]

Predvygenerované hlavolamy sa nachádzajú v priečinku tests\\{veľkosť}
Vlastný hlavolam je možné doplniť do start.txt v tests resp. ciel.txt

Možné optimalizácie

Pri optimalizácií riešenia by sme dokázali nájsť niekoľko krokov, ktoré by vylepšili časovú efektivitu alebo pamäťovú, jednoduché možnosti sme opísali vyššie (ukladanie pozície medzery, neukladanie posledného operátoru a pod.). Existujú aj ďalšie a to napríklad:

- Custom comparator pre heap
    - Teoreticky by sme ho mohli nastaviť na $\leq$, pretože je nám predsa jedno, či v halde bude stav s výsledkom heuristickej funkcie vyššie ako stav s rovnakou hodnotou, ale v poradí novšie pridaný
    - Výsledky testovania napovedali, že predsa len to **jedno nie je** a aj keď väčšina testov sa zrýchlila, nakoľko sme ušetrili väčší počet porovnaní a mohli breaknut loop heapify skôr. Niektoré testy sa nečakane spomalili, presný dôvod prečo implementácia heapq používa $<$ sa dočítame tu.
- Opätovné generovanie ešte nespracovaných nasledovníkov
    - Častokrát sa stane, že stav vygeneruje nasledovníka, ktorý ešte nebol spracovaný, ale je rovnaký ako stav, ktorý už raz bol vygenerovaný a nachádza sa v **priority queue**.
    - Tento novo vygenerovaný stav nemá najnižší výsledok heuristiky, takže nie je na vrchu haldy a teda zatiaľ len čaká na spracovanie. V tomto momente sa tam nachádza rovnaký stav minimálne dva krát.
    - Eventuálne sa stane, že jeden z týchto stavov sa vyberie, spracuje a potom sa toto isté zopakuje, nakoľko sa vyberie ten druhý(rovnaký) stav. Pri vytváraní nasledovníkov sa nemôžeme rozhodnúť, či sa už nachádza v halde, lebo by ju bolo nutné celú prechádzať.
    - Skontrolovať by sme to mohli jednoducho; po každom vybratí z haldy prehľadáme `hashSet` spracovaných stavov a ak tam už je, tak len `continue;` a vyberie sa ďalší uzol.
    - Táto optimalizácia zefektívnila všetky menšie testy niekedy až dvojnásobne.
    - Väčšie testy (7x7) boli prekvapivo spomalené a trvali dlhšie. Môže sa zdať, že keď sme preskočili tieto rovnaké stavy prišli sme o nejakú informáciu, ktorú

nám dávali. Prečo sa tento úkaz ukázal až pri väčších hlavolamoch nie je celkom známe, ale eventuálne sa podarilo nájsť kompromis riešenia.
    - Akonáhle sme prestali triediť nasledovníkov a slepo ich pridávali do haldy, aj keď už boli raz spracovaný, stým, že ak sme z nej vybrali už spracovaného tak sme ho už nerozvíjali znovu, tak toto riešenie sa ukázalo ako najefektívnejšie približne vo všetkých prípadoch.
    - Nakoľko toto porušuje základnú myšlienku algoritmu pre túto úlohu a celkovo nie som si istý o fungovaní toho, prečo sa tento úkaz vyskytol pri testovaní až väčších hlavolamov záver je ten, že:
      Táto optimalizácia NIE JE v programe zohľadnená!
- Ukladanie posledných použitých operátorov, len ako číslo a nie ako reťazec. Preloženie na postupnosť až pri zostavení riešenia
- Ukladanie stavu do menšej dátovej štruktúry, nakoľko objekt `List` je v Pythone dosť nákladnou štruktúrou, pričom my potrebujeme n listov.

Testovanie

Na úvod je vhodné povedať, že aj testy rovnakej veľkosti nemusia byť rovnako zložité, nakoľko cieľová pozícia sa môže nachádzať vo väčšej hĺbke ako pri iných rovnako veľkých hlavolamoch. Náš algoritmus nenájde najkratšie, ale iba nejaké riešenie, na ktorého hľadanie vplýva mnoho faktorov, ktorým sa budeme v tejto sekcii venovať. Konfigurácia programu je nastavená na kompromis metód, ktoré by sa mali ukázať ako najkonzistnejšie podľa hypotéz vysvetlených vyššie a pri opise algoritmov.

Testovací hardware: Windows 10 Home 64bit, i7-7700K @ 4.20GHz, 16GB RAM.
Python 3.8.5 64bit.

Testy boli vykonané ~3x pre každý problém a sú tu uvedené priemerné výsledky.

Neriešiteľné hlavolamy (3x3)

V tomto hlavolame nie je možné sa dostať do cieľového stavu, takže algoritmus musí prehľadať celý stavový priestor a dá sa povedať, že heuristika ho dokonca spomaľuje pretože nevedie nikam.

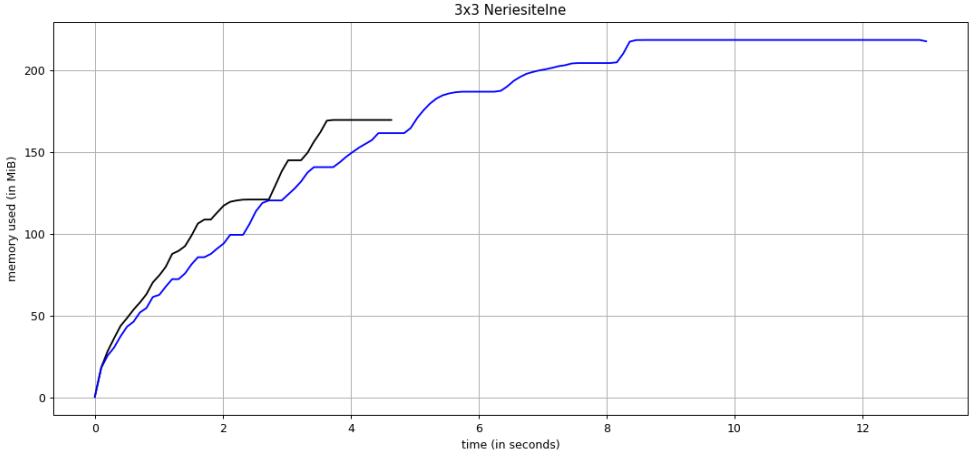
7	8	6
5	4	3
2	0	1

1	2	3
5	5	6
7	8	0

Tab.3 a 4: Počiatočný a cieľový stav pre neriešiteľný 3x3 hlavolam

Heuristika	Čas	Spracované uzly	Vygenerovane uzly
1	4.803	181440	181440
2	14.556	181440	181440

Tab.5: Výsledky pre 3x3 neriešiteľný hlavolam



Obr. 10: Využitie pamäte

Toto je jeden z mála prípadov, kedy bolo **lepšie použiť heuristiku 1** a vidíme aj to, že bolo potrebné preskúmať naozaj všetky dostupné stavy. $3x3 = 9! / 2$,
polovička dosiahnuteľných – 181440 stavov.
Môžeme si na konci všimnúť, že rovná čiara odzrkadľuje, len vyberanie z haldy pretože, všetky stavy sú už vygenerované. Oba grafy stúpajú rovnako.

Malé hlavolamy
Pri týchto veľkostiach je využitie pamäte minimálne.

Tab.3: 3x3

0	1	2
3	4	5
6	7	8

8	0	6
5	4	7
2	3	1

Heuristika	Čas	Spracované uzly	Vygenerované uzly	Dĺžka Riešenia
1	0.0081	648	1 085	119
2	0.0020	78	131	45

Tab.4: 5x2 - široký

0	1	2	3	4
5	6	7	8	9

4	3	2	6	1
9	8	7	5	0

Heuristika	Čas	Spracované uzly	Vygenerované uzly	Dĺžka Riešenia
1	0.110	8 428	12 837	401
2	0.007	554	882	147

Tab.5: 2x7 – vysoký

13	11
8	4
3	2
1	9
5	0
6	10
12	7

1	2
3	4
5	6
7	8
9	10
11	12
13	0

Heuristika	Čas	Spracované uzly	Vygenerované uzly	Dĺžka Riešenia
1	1.250	42 022	71 789	1 500
2	0.060	3 073	5 332	274

Aj napriek tomu, že je dĺžka riešenia až 1500 krokov, čas sa odvíja hlavne od počtu spracovaných uzlov.
Rôzne veľkosti môžu mať podobný počet krokov.

Stredné hlavolamy

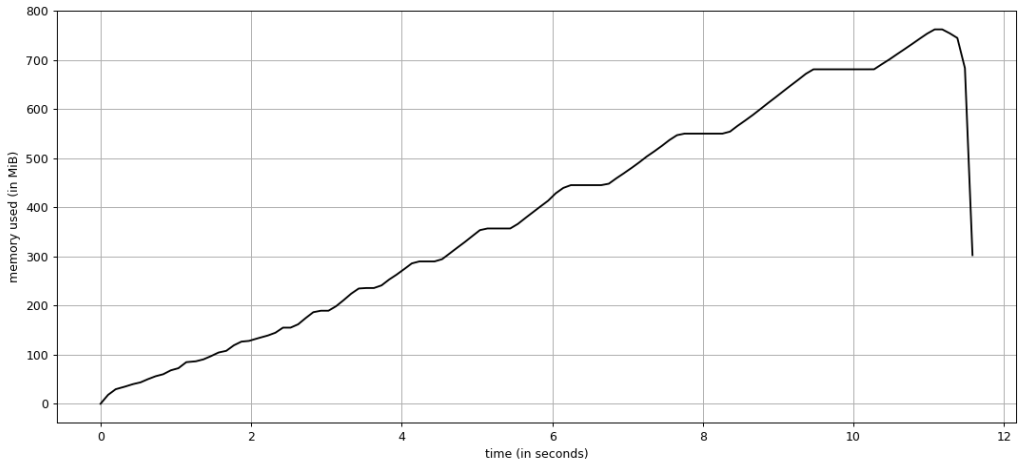
Tab.6: 5x5

6	7	16	12	5
15	20	3	10	2
11	1	13	9	18
17	0	22	14	8
21	23	19	4	24

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	0

Heuristika	Čas	Spracované uzly	Vygenerované uzly	Dĺžka Riešenia
1	11.600	293 193	698 839	3 342
2	0.250	9 449	22 367	490

Už si začíname všímať, že heuristike 1 dochádza dych a generuje *mierne* zložitejšie riešenie
Výsledok heurestík pre počiatočný stav
H1 – 20
H2 – 50

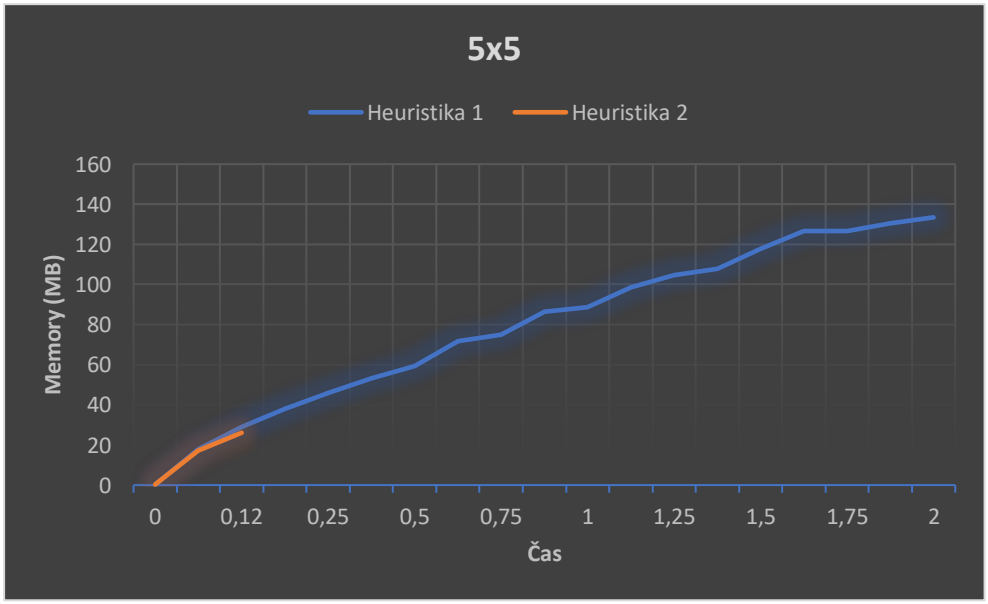


Obr. 11: Využitie pamäte
heuristika 1 5x5

Rovnaká konfigurácia, ale zameníme počiatočný a cieľový stav:

Heuristika	Čas	Spracované uzly	Vygenerované uzly	Dĺžka Riešenia
1	1.750	45 493	112 952	1144
2	0.135	5 354	12 317	526

Pozorujeme výraznú zmenu v oboch výsledkoch, najmä pri prvej heuristike. Do grafu sme pre porovnanie pridali aj druhú heuristiku.



Obr. 12: Využitie pamäte
heuristika 1 a 2 5x5 opačné

Veľké hlavolamy

8	19	1	3	5	6	7
10	26	2	21	4	12	14
9	29	39	37	11	18	13
15	16	17	32	25	41	34
30	23	22	38	47	40	27
36	31	46	0	33	35	42
43	44	24	45	28	20	48

Tab.8: 7x7 – zložitejší hlavolam

1	9	2	5	4	13	12
8	3	46	26	18	7	14
23	17	31	16	39	21	20
15	29	24	11	35	27	28
25	36	30	32	10	34	42
43	22	37	41	38	0	19
44	40	45	33	47	6	48

Tab.9: 7x7 – jednoduchší hlavolam

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42
43	44	45	46	47	48	0

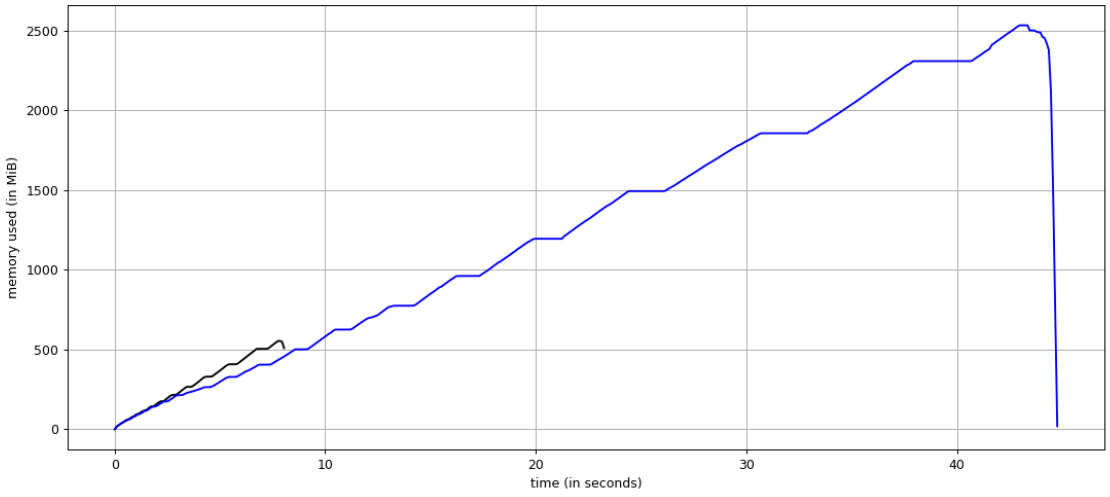
Tab.10: 7x7 – cieľový stav

Pri väčších hlavolamoch už **heuristika 1 nie je dostačujúca** (max 6x6 ~35 sekúnd) a dĺžka hľadania riešenia sa pohybuje v minútach, preto nie je vôbec prípustná, nakoľko musí vygenerovať veľmi veľké kvantum uzlov, čím zaplní celý dostupný adresný priestor. Tu môžeme porovnávať ako zložito je zadaná počiatočná pozícia ak cieľová zostane rovnaká.

Hoci tieto konfigurácie majú podobné výsledky heuristiky riešenie hlavolamu označeného ako jednoduchší nájdeme omnoho skôr ako zložitejšieho. Je to jednoducho kvôli tomu, že sa môže stať, že nás heuristika zavedie ďalej od finálneho riešenia a pri jednoduchšom trafíme hneď cestu ktorá vedie k cieľu.

Heuristika	Čas	Spracované uzly	Vygenerované uzly	Dĺžka Riešenia
Jednoduchší	8.030	144 888	363 985	1 164
Zložitejší	40.670	662 110	1 699 471	1 574

Riešenia nie sú ani o toľko dlhšie alebo kratšie, ale pri prvom hlavolame sme našli výsledok o dosť skôr a nemuseli generovať niekoľko násobne viac uzlov. Počet vygenerovaných uzlov **korešponduje** s potrebným časom – 5x viac vygenerovaných uzlov znamená 5x dlhší časový výsledok. Taktiež si všimnime, že dĺžka riešenia pri zložitejšom hlavolame je približne 1500, tento výsledok je podobný ako pri 2x7 pre heuristiku 1, ale samozrejme dôležité sú vygenerované uzly, ktorých je niekoľko násobne viac.



Obr. 13: Využitie pamäte heuristika 2 7x7

Záver a zhodnotenie

V závere tejto práce môžeme potvrdiť, že sa nám z pod-úloh kladených v úvode podarilo naprogramovať a spojiť rôzne implementácie **do jedného fungujúceho celku**. Využili sme poznatky z teórie o grafoch, ozrejmili si efektívne algoritmy, ktoré vyhľadávajú v dynamických množinách a zúročili teoretické znalosti o správnom použití **lačného hľadania**.

Pomocou testovania sa nám podarilo porovnať dve heuristické funkcie a zistili sme, kedy je lepšie použiť, ktorú funkciu. **Heuristická funkcia č.1** sa veľmi rýchlo ukázala ako dosť **nevýhodná**, kvôli nízkemu rozsahu výsledku pre konkrétny stav. Pre veľké množstvo vygenerovaných stavov sa **osvedčila heuristika č.2** a v prípade, keď sme museli spracovať všetky stavy bola lepšia heuristika č.1.

Mimo iné sa počas riešenia úlohy zužitkovali doposiaľ nadobudnuté princípy **manažovania pamäte** so snahou ušetriť, tam kde je to možné. V projekte sme museli myslieť ako aj priestorovú, tak aj časovú efektivitu a nájsť **kompromis** pre korektné riešenie. Popri vytváraní a zhotovení riešenia sa objavili aj možné optimalizácie, ktoré bolo potrebné zvážiť. Projekt vytvorený v jazyku **Python** uľahčuje používanie základných dátových štruktúr a zjednodušuje implementáciu oproti iným low-level programovacím jazykom. Kód je prehľadnejší a počas tvorby sme nestrácali čas na veciach, ktoré by v inom prostredí mohli zabrať dlhší čas. Na druhú stranu máme len veľmi malú možnosť spravovať alokáciu a uvoľňovanie systémových prostriedkov - pamäte.

Záver mimo iné priniesol aj poznatky vo forme kreslenia vývojových diagramov, grafov v programe Excel, či využitie modulov na profilovanie pamäte. Cesta k zložitejším riešeniam v ďalšom pokračovaní tohto alebo iného predmetu, či v praxi je otvorená.

Zdroje a prevzaté implementácie

[1] - O'Connor, Kevin. Peters, Tim. Hettinger, Raymond. August 2002. Heap Queue Algorithm
https://github.com/python/cpython/blob/master/Lib/heapq.py
https://docs.python.org/3.8/library/heapq.html
[2] - Pedregosa, Fabian. Gervais, Philippe. 2007. Memory Profiler
https://github.com/pythonprofilers/memory_profiler
Obrázky puzzle
[3] - http://www2.fiit.stuba.sk/~kapustik/z2d.html
[4] - http://www.artbylogic.com/puzzles/numSlider/numberShuffle.htm
[5] - Manhattan distance
https://en.wikipedia.org/wiki/Taxicab_geometry
[6] - Návrat a kol.: Umelá Inteligencia, STU Bratislava.
[7] - Rohilla, Mansi. Lal, Shikha. Khan, Rashad. August 2018. Eight Puzzle Problem
https://www.youtube.com/watch?v=41nDCiNcEbs
Kreslenie diagramov
http://draw.io/

Martin Melišek 27.10.2020