

SLOVENSKÁ TECHNICKÁ UNIVERZITA

Fakulta informatiky a informačných technológií

v Bratislave

Umelá inteligencia

Zadanie 2

Adam Tomčala

Prednášajúci: Ing. Lukáš Kohútka, PhD.

Cvičiaci: Ing. Boris Slíž

Čas cvičení: Streda 15:00

Definovanie problému:

Mojou úlohou je nájsť riešenie pre 8-hlavolam. 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.

Príkladom môže byť nasledovná začiatočná a koncová pozícia:

Začiatok:			Koniec:		
1	2	3	1	2	3
4	5	6	4	6	8
7	8		7	5	

Im zodpovedajúca postupnosť krokov je: **VPRAVO, DOLE, VĽAVO, HORE.**

Stav:

Stav predstavuje aktuálne rozloženie políčok. Počiatočný stav môžeme zapísať napríklad ((1 2 3)(4 5 6)(7 8 m))

alebo

(1 2 3 4 5 6 7 8 m)

Každý zápis má svoje výhody a nevýhody. Prvý umožňuje (všeobecnejšie) spracovať ľubovoľný hlavolam rozmerov $m \times n$, druhý má jednoduchšiu realizáciu operátorov.

Vstupom algoritmov sú práve dva stavy: začiatočný a cieľový. Vstupom programu však môže byť aj ďalšia informácia, napríklad výber heuristiky.

Operátory:

Operátory sú len štyri:
VPRAVO, DOLE, VĽAVO a HORE

Operátor má jednoduchú úlohu – dostane nejaký stav a ak je to možné, vráti nový stav. Ak operátor na vstupný stav nie je možné použiť, výstup nie je definovaný. V konkrétnej implementácii je potrebné výstup buď vhodne dodefinovať, alebo zabrániť volaniu nepoužiteľného operátora. **Všetky operátory pre tento problém majú rovnakú váhu.**

Heuristická funkcia:

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

Tieto odhady majú navyše mierne odlišné vlastnosti podľa toho, či medzi políčka počítame alebo nepočítame aj medzeru. Započítavať medzeru však nie je vhodné, lebo taká heuristika nadhodnocuje počet krokov do cieľa.

Uzol:

Stav predstavuje nejaký bod v stavovom priestore. My však od algoritmov požadujeme, aby nám ukázali cestu. Preto musíme zo stavového priestoru vytvoriť graf, najlepšie priamo strom. Našťastie to nie je zložitá úloha. Stavý jednoducho nahradíme uzlami.

Čo obsahuje typický uzol?

Musí minimálne obsahovať

- **STAV** (to, čo uzol reprezentuje) a
- **ODKAZ NA PREDCHODCU** (pre nás zaujímavá hrana grafu, reprezentovaná čo najefektívnejšie).

Okrem toho môže obsahovať ďalšie informácie, ako

- **POSLEDNE POUŽITÝ OPERÁTOR**
- **PREDCHÁDZAJÚCE OPERÁTORY**
- **HĺBK A UZLA**
- **CENA PREJDENEJ CESTY**
- **ODHAD CENY CESTY DO CIEĽA**
- Iné vhodné informácie o uzle

Uzol by však nemal obsahovať údaje, ktoré sú nadbytočné a príslušný algoritmus ich nepotrebuje. Pri zložitých úlohách sa generuje veľké množstvo uzlov a každý zbytočný bajt v uzle dokáže spotrebovať množstvo pamäti a znížiť rozsah prehľadávania algoritmu. Nedostatok informácií môže zase extrémne zvýšiť časové nároky algoritmu. **Použité údaje zdôvodnite.**

Algoritmus:

Každé zadanie používa svoj algoritmus, ale algoritmy majú mnohé spoločné črty. Každý z nich potrebuje udržiavať informácie o uzloch, ktoré už kompletne spracoval a aj o uzloch, ktoré už vygeneroval, ale zatiaľ sa nedostali na spracovanie. Algoritmy majú tendenciu generovať množstvo stavov, ktoré už boli raz vygenerované. S týmto problémom je tiež potrebné sa vhodne vysporiadať, zvlášť u algoritmov, kde rovnaký stav neznamena rovnako dobrý uzol.

Zadanie d) :

- Použite algoritmus lačného hľadania, porovnajite výsledky heuristik 1. a 2.
- Programovací jazyk, ktorý som použil: Python 3.9

Popis algoritmu:

- Pre moje riešenie som použil algoritmus lačného hľadania (greedy algoritmus). Algoritmus vytvára strom z jednotlivých uzlov na základe heuristiky 1, heuristiky 2 a vždy rozvíja taký uzol, ktorý má danú heuristiku najmenšiu (najmenšia manhattanská vzdialenosť všetkých políček v aktuálnom stave od políček v koncovom stave alebo najmenší počet nesprávne uložených políček).

Reprezentácia uzla:

- Jednotlivé uzly som si vytváral ako objekty, ktoré obsahovali nasledujúce atribúty:

```
class Uzol:
    """
    Trieda uzol sluzi na vytvaranie novych stavov pri rieseni hlavolamu.
    Obsahuje tabulku (aktuálny stav hlavolamu), rodica a smer pohybu
    """
    def __init__(self, pole, sirka, vyska, rodic=None, smer=""):
        self.tabulka = []
        self.sirka = sirka
        self.vyska = vyska
        for i in range(0, len(pole)):
            self.tabulka.append(pole[i])
        self.heuristika = 0
        self.rodic = rodic
        self.pohyb = smer
```

- Každý uzol obsahuje svoje rozmiestnenie políček – to som reprezentoval pomocou 1D listu - **tabuľka**
- Keďže som riešil aj hlavolamy MxN, je potrebné si udržiavať informácie aj o rozmere hlavolamu – **šírka, výška**
- Atribút **rodíč** slúži na spojenie potomka s rodičom.
- Atribút **smer** slúži na to, aby som spätne mohol zistiť, aký pohyb sa vykonal, aby som sa dostal k danému uzlu
- **Heuristika** – hodnota buď heuristiky 1 alebo 2

Funkcie v triede Uzol:

- Pre výpočet heuristiky 1 v danom uzle som použil funkciu:

```
def pocet_zlych(self, konecny):  
    """  
    Funkcia zisti, kolko policok v danom stave nie je na rovnakych miestach ako v koncovom stave.  
    Heuristika 1.  
    :param konecny:      Uzol, s ktorym porovnavam.  
    :return:             Nic.  
    """  
    for i in range(0, len(konecny.tabulka)):  
        if self.tabulka[i] != konecny.tabulka[i]:  
            self.heuristika += 1
```

- Pre výpočet heuristiky 2 v danom uzle som použil funkciu:

```
def manhattan(self, konecny):  
    """  
    Funkcia zisti manhattanskú vzdialenosť každého políčka s políčkom v koncovom stave.  
    Heuristika 2.  
    :param konecny:      Uzol, s ktorym porovnavam.  
    :return:             Nic.  
    """  
    for i in range(0, len(konecny.tabulka)):  
        if konecny.tabulka[i] == 0:  
            continue  
        index = self.tabulka.index(konecny.tabulka[i])  
        self.heuristika += (abs(i // konecny.sirka - index // konecny.sirka)) + (  
            abs(i % konecny.sirka - index % konecny.sirka))
```

- Funkcia zisťuje vzdialenosť každého políčka v aktuálnom uzle (okrem 0) od políčka v koncovom uzle.
- Príklad:

1	2	3
4	5	6
7	8	

7	8	6
5	4	3
2		1

Políčko 1 : 4, Políčko 2 : 3, Políčko 3 : 1, ...

- Na vykonávanie jednotlivých pohybov som používal **funkcie**:

```
def hore(self):  
    """  
    Funkcia overuje ci sa moze uskutočnit pohyb hore.  
    :return: Tabuľku (ak sa moze pohnut hore), inak None.  
    """  
    index = self.tabulka.index(0)  
    if (index // self.sirka) - 1 < 0:  
        return None  
    return self.vytvor_novy_vstup([], self.tabulka[index - self.sirka], index, index - self.sirka)
```

```
def dole(self):  
    """  
    Funkcia overuje ci sa moze uskutočnit pohyb dole.  
    :return: Tabuľku (ak sa moze pohnut dole), inak None.  
    """  
    index = self.tabulka.index(0)  
    if (index // self.sirka) + 1 >= self.vyska:  
        return None  
    return self.vytvor_novy_vstup([], self.tabulka[index + self.sirka], index, index + self.sirka)
```

```
def vpravo(self):  
    """  
    Funkcia overuje ci sa moze uskutočnit pohyb vpravo.  
    :return: Tabuľku (ak sa moze pohnut vpravo), inak None.  
    """  
    index = self.tabulka.index(0)  
    if (index % self.sirka) + 1 >= self.sirka:  
        return None  
    return self.vytvor_novy_vstup([], self.tabulka[index + 1], index, index + 1)
```

```
def vlavo(self):  
    """  
    Funkcia overuje ci sa moze uskutočnit pohyb vlavo.  
    :return: Tabuľku (ak sa moze pohnut vlavo), inak None.  
    """  
    index = self.tabulka.index(0)  
    if (index % self.sirka) - 1 < 0:  
        return None  
    return self.vytvor_novy_vstup([], self.tabulka[index - 1], index, index - 1)
```

- Každá z týchto funkcií vykonáva pohyb určitým smerom
- Najskôr sa v každej funkcii skontroluje, či sa daný pohyb môže vykonať, ak áno zavolá sa funkcia **vytvor_novy_vstup**, do ktorej sa posielajú nasledovné argumenty:
 - prázdny list -> do neho sa uloží novovzniknutá tabuľka
 - hodnota -> číslo, s ktorým sa vymení 0
 - Index1 -> index 0
 - Index2 -> index hodnoty, ktorá sa ide vymeniť s 0

- Funkcia **vytvor_novy_vstup**:

```
def vytvor_novy_vstup(self, vysledok, hodnota, index_nuly, index_cisla):  
    """  
    Funkcia vytvara novu tabulku pre zvolenu smer.  
    :param vysledok:      Novy stav.  
    :param hodnota:      Hodnota cisla, s ktorym sa vymenila 0.  
    :param index_nuly:   Index, na ktorom sa nachadzala 0.  
    :param index_cisla:  Index, na ktorom sa nachadzala hodnota.  
    :return:             Pole (novovytvorenu tabulku).  
    """  
    for i in range(0, len(self.tabulka)):  
        if i == index_nuly:  
            vysledok.append(hodnota)  
        elif i == index_cisla:  
            vysledok.append(0)  
        else:  
            vysledok.append(self.tabulka[i])  
    return vysledok
```

- Funkcia vytvorí novú tabuľku, tak že vymení 0 s hodnotou, podľa pohybu
- Na volanie funkcií pre jednotlivé pohyby slúži funkcia:

```
def pohyby(self, smer):  
    """  
    Funkcia zavola funkcia na konkretny pohyb.  
    :param smer:      Smer, ktory sa chceme skusit pohnut.  
    :return:          Vystupnu tabulku  
    """  
    if smer == 'H':  
        return self.hore()  
    elif smer == 'D':  
        return self.dole()  
    elif smer == 'L':  
        return self.vlavo()  
    elif smer == 'R':  
        return self.vpravo()
```

Riešenie:

- Jednotlivé vytvorené uzly (tie, ktoré boli jedinečné) som ukladal do dátovej štruktúry **Priority Queue**, ktorú som si importoval z knižnice queue.

```
from queue import PriorityQueue
```

- Ako **priority key** som používal heuristiku pre daný uzol.
- Ako ďalší **priority key** som používal číslo, ktoré reprezentovalo poradie vytvoreného uzla (kvôli tomu, ak by mali 2 uzly rovnaké heuristiky, vyberie sa ten, ktorý bol vytvorený skôr).

```
priorita = PriorityQueue() # vytvorenie priority queue
```

```
priorita.put((zac.heuristika, pocet_uzlov, zac))
```

Heuristika uzla	Poradové číslo uzla	Uzol
-----------------	---------------------	------

- Ďalšou dôležitou súčasťou implementácie môjho riešenia je vytvorenie si prázdneho listu na začiatku riešenia, kde si ukladám všetky jedinečné vygenerované tabuľky.
- Je to dôležité kvôli tomu, aby sa zbytočne nevytvárali duplicitné uzly.
- Pomocou funkcie priority.get() si vyberiem z priority queue uzol s najmenšou heuristikou a idem skúšať, ktorými smermi sa môžem vybrať.
- Po zistení možných smerov kontrolujem či sa daná tabuľka už nenachádza v liste pre už vygenerované tabuľky.
- Ak áno, ignorujem ju, ak nie vypočítam heuristiku a poradové číslo uzla a uzol vložím do priority queue.
- Tento algoritmus opakujem až kým nedôjdem k riešeniu (kým sa nebude tabuľka aktuálneho uzla rovnáť s koncovým) alebo kým sa neprejdú všetky možnosti.

Testovanie:

- Na testovanie algoritmu som si vybral nasledujúce rozmery hlavolamu:
 - 3x2
 - 4x2
 - 3x3
 - 5x2
 - 4x3
 - 6x2
- Pre každý jeden test som porovnával rozdiel medzi nájdením cesty pri heuristike 1 a 2
- Analyzoval som nasledujúce údaje:
 - Čas riešenia
 - Počet vytvorených uzlov
 - Celkový počet iterácií
 - Cena (resp. dĺžka cesty)
- Vstupné a výstupné uzly som mal pre každý jeden test dané v programe (Vstupné a výstupné uzly som vyberal so stránky v zadaní).

Výsledky testovania:

- Hlavlom 3x2:

```
Zaciatocny stav:  
1 2 0  
3 4 5
```

```
Konecny stav:  
3 4 5  
0 1 2
```


- **Výsledok pre 3x2 hlavolam:**

```
----- Heuristika 1: -----   ----- Heuristika 2: -----  
Cas riesenia: 0.0080           Cas riesenia: 0.0030  
Pocet uzlov: 173               Pocet uzlov: 105  
Pocet iteracii: 139            Pocet iteracii: 78  
Cena cesty: 23                 Cena cesty: 23
```

- Heuristika čísla 2 bola rýchlejšia, vytvorilo sa menej uzlov a vykonalo sa menej iterácií

- **Hlavolam 4x2:**

```
Zaciatocny stav:  
0 1 2 3  
4 5 6 7
```

```
Konecny stav:  
3 2 5 4  
7 6 1 0
```

- **Výsledok pre 4x2 hlavolam:**

```
----- Heuristika 1: -----   ----- Heuristika 2: -----  
Cas riesenia: 0.0569           Cas riesenia: 0.0200  
Pocet uzlov: 933               Pocet uzlov: 474  
Pocet iteracii: 648            Pocet iteracii: 313  
Cena cesty: 54                 Cena cesty: 70
```

- Heuristika čísla 2 bola rýchlejšia, vytvorilo sa menej uzlov a vykonalo sa menej iterácií

- **Hlavolam 3x3 :**

```
Zaciatocny stav:  
1 0 2  
3 4 5  
6 7 8
```

```
Konecny stav:  
8 0 6  
5 4 7  
2 3 1
```

- **Výsledok pre 3x3 hlavolam:**

```
----- Heuristika 1: -----   ----- Heuristika 2: -----  
Cas riesenia: 0.0747           Cas riesenia: 0.0060  
Pocet uzlov: 1199              Pocet uzlov: 172  
Pocet iteracii: 737            Pocet iteracii: 102  
Cena cesty: 62                 Cena cesty: 44
```

- Heuristika čísla 2 bola rýchlejšia, vytvorilo sa menej uzlov a vykonalo sa menej iterácií

- **Hlavlom 5x2 :**

```
Zaciatocny stav:  
1 2 3 4 9  
5 6 0 7 8
```

```
Konecny stav:  
5 1 9 8 7  
2 4 3 6 0
```

- **Výsledok pre 5x2 hlavlom:**

```
----- Heuristika 1: -----   ----- Heuristika 2: -----  
Cas riesenia: 3.5258             Cas riesenia: 0.0080  
Pocet uzlov: 9809               Pocet uzlov: 167  
Pocet iteracii: 6838            Pocet iteracii: 97  
Cena cesty: 168                 Cena cesty: 38
```

- Heuristika čísla 2 bola v tomto prípade oveľa rýchlejšia, vytvorilo sa oveľa menej uzlov a vykonalo sa menej iterácií

- **Hlavlom 4x3 :**

```
Zaciatocny stav:  
1 2 0 3  
4 5 6 7  
8 9 10 11
```

```
Konecny stav:  
2 7 4 6  
1 9 3 11  
10 8 0 5
```

- **Výsledok pre 4x3 hlavlom:**

```
----- Heuristika 1: -----   ----- Heuristika 2: -----  
Cas riesenia: 1.6625            Cas riesenia: 0.1037  
Pocet uzlov: 6686               Pocet uzlov: 1312  
Pocet iteracii: 3600            Pocet iteracii: 704  
Cena cesty: 120                 Cena cesty: 46
```

- Heuristika čísla 2 bola v tomto prípade oveľa rýchlejšia, vytvorilo sa oveľa menej uzlov a vykonalo sa menej iterácií

- **Hlavlom 6x2 :**

```
Zaciatocny stav:  
1 2 3 4 5 0  
6 7 8 9 10 11
```

```
Konecny stav:  
4 5 9 11 10 8  
3 2 1 6 7 0
```

- **Výsledok pre 6x2 hlavolam:**

----- Heuristika 1: -----	----- Heuristika 2: -----
Cas riesenia: 6.1844	Cas riesenia: 0.0877
Pocet uzlov: 12897	Pocet uzlov: 1263
Pocet iteracii: 8294	Pocet iteracii: 762
Cena cesty: 153	Cena cesty: 107

- Heuristika čísla 2 bola v tomto prípade oveľa rýchlejšia, vytvorilo sa oveľa menej uzlov a vykonalo sa menej iterácií

Zhodnotenie testovania:

Z mojich testov som zistil, že heuristika 2 je oveľa efektívnejšia ako heuristika 1. Aj keď je spracovanie heuristika 2 časovo náročnejšie ako heuristika 1, tak celkový čas pre nájdenie cesty z bodu A do bodu B je o mnoho lepší, pretože heuristika 2 dokáže presnejšie určiť uzol, ktorý sa bude rozvíjať.

Používateľská príručka:

Po spustení programu si používateľ bude môcť vybrať, ktorý test chce spustiť. Po vykonaní testu sa vypíšu porovnania heuristík a vypíšu sa aj jednotlivé cesty pre obe heuristiky od začiatočného až po koncový uzol.

```
----- TESTY -----  
TEST 1: Hlavolam s rozmermi 3x2.  
TEST 2: Hlavolam s rozmermi 4x2.  
TEST 3: Hlavolam s rozmermi 3x3.  
TEST 4: Hlavolam s rozmermi 5x2.  
TEST 5: Hlavolam s rozmermi 4x3.  
TEST 6: Hlavolam s rozmermi 6x2.  
Zadajte cislo TESTU:
```

Po vykonaní vybraného testu sa program ukončí.