Oliver Leontiev, 103027
UI, Zadanie 2 e)
2020

Prehľadávanie stavového priestoru

Problém 2: 8-hlavolam

Úloha e) A* algoritmus

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.

Príkladom môže byť nasledovná začiatočná a koncová pozícia (0 = medzera):

1 2 3 1 2 3

4 5 6 4 6 8

7 8 0 7 5 0

Problém máme riešiť využitím **A* algoritmus** s dvoma rôznymi heuristikami:

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

Riešenie

Program je konzolová aplikácia, ktorá bola vytvorená v programovacom jazyku Python 3. Spúšťa sa súbor Main.py a to štandardne:

```
$ python Main.py
```

Štruktúra

Program sa skladá zo štyroch súborov:

Main.py : Spustiteľná časť programu, riadi vstup a výstup

Solver.py: Obsahuje triedu *Solver*, ktorá overuje riešiteľnosť a rieši hlavolam (obsahuje aj funkciu s A* algoritmom)

Node.py: Obsahuje triedu *Node*, ktorá predstavuje jeden uzol v stavovom priestore

Generator.py: Obsahuje triedu *Generator*, ktorá slúži na generovanie vstupov pre účely testovania

V programe sú využité knižnice:

Heapq: poskytuje minimálnu haldu na reprezentáciu prioritného radu uzlov v stavovom priestore.

Numpy: poskytuje vhodné polia (najmä 2D) na reprezentáciu stavu hlavolamu.

Oliver Leontiev, 103027
UI, Zadanie 2 e)
2020

Timeit: umožňuje merať čas trvania programu.

Random: umožňuje generovanie náhodných stavov pre vstup.

Reprezentácia uzla:

```
# One node in the graph of states
class Node:
    def __init__(self, state, parent, op):
        self.state = state  # Puzzle state - represents arrangement of the puzzle for this node
        self.parent = parent  # Parent Node
        self.op = op  # Operator used on parent node to get to this node - l|r|u|d = LEFT|RIGHT|UP|DOWN
        if parent is None:
            self.depth = 0
        else:
            self.depth = parent.depth + 1
        self.value = None  # Value used by the priority queue
```

Uzol predstavuje objekt triedy *Node* a okrem konkrétneho stavu hlavolamu ukladá aj **odkaz na rodiča** (predošlý uzol, aby sme mohli vytvoriť cestu), **operátor** ktorým vznikol (je súčasťou výstupu a potrebujeme ho vypísať), **hĺbku** v ktorej sa nachádza v strome stavov (potrebujeme pri výpočte hodnoty dieťaťa) a svoju **hodnotu** (heuristická funkcia + hĺbka, potrebujeme ju na určenie priority uzla)

Stav uzla je rozloženie políčok (dlaždíc) hlavolamu. Je reprezentovaný 2D poľom a medzera je reprezentovaná nulou.

Operátori sú interne reprezentované ako pohyb prázdneho políčka (teda opačne), a sú preložené až pred výpisom.

Algoritmus

Na nájdenie najkratšej cesty medzi štartovacím stavom a cieľovým stavom naprieč stavovým priestorom využívame A* algoritmus, ktorý má konkrétne takúto podobu v našom programe:

Vstup: štartovací stav, cieľový stav

Výstup: cieľový uzol – cez referencie na rodičovský uzol vieme vystavať najkratšiu cestu

Krok 1: Vytvor* počiatočný uzol a umiestni ho do prioritného radu a medzi vytvorené uzly.

Krok 2: Vytiahni z prioritného radu uzol s najmenšou hodnotou.

Krok 3: Over či stav uzlu je rovnaký ako cieľový. Ak áno vráť uzol a UKONČI.

Krok 4: Rozviň uzol – vytvor* všetky legitímne** detské uzly. Pre každý detský uzol skontroluj, či už nie je medzi vytvorenými uzlami. Ak nie pridaj ho medzi vytvorené uzly a do prioritného radu. Choď na krok 2.

Oliver Leontiev, 103027
UI, Zadanie 2 e)
2020

```
def a_star(self, start_state, h_function):
    heap = []
    generated_count = 0
    opened_count = 0
    max_depth = 0
    start_node = Node(start_state, None, None)
    start_node.value = h_function(start_state)                    Krok 1
    generated_states = {start_node.state.copy().tostring()}
    heappush(heap, start_node)
    while not np.array_equal(heap[0].state, self.goal_state):     Krok 2  Krok 3
        opened_count += 1
        new_children = heappop(heap).make_children(self.rows_am, self.cols_am)   Krok 4
        for child in new_children:
            if child.state.copy().tostring() not in generated_states:
                generated_states.add(child.state.copy().tostring())
                child.value = h_function(child.state) + child.depth
                generated_count += 1
                max_depth = max(max_depth, child.depth)
                heappush(heap, child)
    print(' '.join(["Depth:", str(max_depth), "Generated nodes:",
                    str(generated_count), "Opened nodes:", str(opened_count)]))
    return heappop(heap)                                          Vrátenie cieľového uzla a ukončenie
```

* vytvorenie uzla zahŕňa výpočet jeho hodnoty pomocou heuristickej funkcie a hĺbky.

** legitímny detský uzol je taký, ktorý nemá rovnaký stav ako jeho rodič - nepovoľujeme spätný krok.

Heuristické funkcie

Heuristické funkcie vypočítajú hodnotu uzla, ktorá určuje jeho prioritu (čím menšia hodnota, tým vyššia priorita. K heuristickej hodnote sa ešte pripočítava hĺbka uzla. Používame vždy jednu z dvoch heuristík:

1. Počet políčok, ktoré nie sú na svojom mieste

```
# First heuristic function, returns amount of misplaced numbers
def get_hvalue1(self, node_state):
    hvalue = 0
    for i, row in enumerate(node_state):
        for j, num in enumerate(row):
            if num != 0 and num != self.goal_state[i][j]:
                hvalue += 1
    return hvalue
```

Jedná sa o vcelku jednoduchý výpočet - pre každé políčko v stave uzla (okrem prázdneho) zvýšime hodnotu o 1, ak na jeho pozícii v cieľovom stave je iné políčko.

Oliver Leontiev, 103027
UI, Zadanie 2 e)
2020

2. Súčet vzdialeností jednotlivých políčok od ich cieľovej pozície

```python
# Second heuristic function, returns the sum of distances of number positions from their goal positions.
def get_hvalue2(self, node_state):
    hvalue = 0
    for i, row_node in enumerate(node_state):
        for j, num in enumerate(row_node):
            if num != 0:
                hvalue += abs(i - self.goal_coord[num][0]) + abs(j - self.goal_coord[num][1])
    return hvalue
```

Pri zvolení druhej heuristiky sa najprv vytvorí Dictionary z políčok v cieľovom stave a ich súradníc ([číslo na políčku, (y, x)]). Ten slúži na rýchlejší prístup k súradniciam pri výpočte heuristiky pre každý nový vygenerovaný stav. Vzhľadom na to, že hýbať sa vieme iba po osiach x a y, tak vzdialenosť dvoch políčok dostaneme ako súčet absolútnych hodnôt rozdielov súradníc.

Overenie riešiteľnosti

Pri vytváraní funkcie na overenie riešiteľnosti sa čerpalo z postupov popísaných tu:

https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/

Podmienky riešiteľnosti sú tu uvedené voči **základnému stavu** napr. pre 4 x 4 je základný stav:

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 0

Podmienky riešiteľnosti voči základnému stavu závisia od počtu stĺpcov a sú nasledovné:

1. Ak je počet stĺpcov nepárny, tak je hlavolam riešiteľný ak je počet inverzií párny.

2. Ak je počet stĺpcov párny, tak je hlavolam riešiteľný ak:

2 a). Prázdne políčko je na párnom riadku zdola a počet inverzií je nepárny.

2 b). Prázdne políčko je na nepárnom riadku zdola a počet inverzií je párny.

3. V iných prípadoch je hlavolam neriešiteľný.

Inverzia je, ak sa políčko nachádza pred políčkom s nižším číslom.

Z uvedených podmienok riešiteľnosti a ich odôvodnenia vyplýva, že **platia pre ľubovoľné M x N hlavolamy** a **rozdeľujú potenciálny stavový priestor na dve disjunktné polovice**, pre ktoré platí:

Pre každý stav platí, že pre všetky z neho dosiahnuteľné stavy sú podmienky riešiteľnosti voči základnému stavu splnené alebo nesplnené.

Oliver Leontiev, 103027
UI, Zadanie 2 e)
2020

To znamená, že pre účely nášho programu treba overiť, či podmienky sú splnené aj pre cieľový aj pre štartovací stav alebo nesplnené pre obidva. Teda zistiť, či patria do rovnakej polovice možných stavov. To dosiahneme nasledovnou funkciou:

```
def is_solvable(self, start_state):
    if self.cols_am & 1:
        if (count_inversions(start_state) & 1) == (count_inversions(self.goal_state) & 1):
            return True
    else:
        mover_y = find_mover(start_state)[0]
        star_mover = (self.rows_am - mover_y) & 1
        mover_y = find_mover(self.goal_state)[0]
        goal_mover = (self.rows_am - mover_y) & 1
        if (((count_inversions(start_state) & 1) == star_mover) ==
                ((count_inversions(self.goal_state) & 1) == goal_mover)):
            return True
    return False
```

Testovanie

Na testovanie slúžili:

1. Vytvorené konkrétne hlavolamy, ktoré sme sa snažili vyriešiť postupnosťou krokov vygenerovaných programom.

Na toto okrem vlastnej hlavy poslúžila stránka: https://murhafsousli.github.io/8puzzle/#/ , ktorá generuje 8-Hlavolamy a umožňuje ich riešiť.

Zároveň boli použité aj riešené príklady zo stránky cvičení:
http://www2.fiit.stuba.sk/~kapustik/MN%20hlavolam.html , na overenie základnej funkčnosti programu (či nájde riešenie v rovnakej hĺbke)

2. Generátor hlavolamov priložený k programu.

Generátor generuje aj neriešiteľné stavy, čo umožňuje otestovať aj túto funkcionalitu programu. Vieme, že pravdepodobnosť vygenerovania neriešiteľného hlavolamu je 50%, takže stačí vygenerovať väčšie množstvo naraz a mali by sme dostať aj riešiteľné aj neriešiteľné.

Vzorka 20 príkladov je v priloženom súbore *puzzle_examples.txt* a je použiteľná priamo z programu (pri voľbe vstupu stačí zadať 'f' ako "file").

Program zvláda hlavolamy s rôznym počtom políčok a hĺbkou riešenia (počet krokov do cieľového stavu). Pri hlavolamoch s veľkosťou 4 x 4 už začína trvať riešenie dlhšie až príliš dlho. To isté platí pre hlavolamy 4 x 3 a väčšie. Záleží najmä od **počtu uzlov, ktoré je potrebné rozvinúť** aby sme sa dostali k riešeniu, nepriamo teda závisí primárne od **hĺbky cieľového uzla**. Taktiež je dôležitý výber heuristiky, všetky nižšie uvedené príklady sú s použitím **2. heuristickej funkcie**.

"Generated nodes" je počet uzlov, ktoré boli počas behu programu vytvorené (inicializované), **"Opened nodes"** je počet uzlov, ktoré boli rozvité (boli vytiahnuté z prioritného radu a vytvorili sa všetky ich deti). Môžeme si všimnúť, ako rastie dĺžka trvania programu v závislosti najmä od počtu

otvorených uzlov (v niektorých prípadoch skoro úmerne) a ako toto číslo rastie v závislosti od hĺbky a samozrejme veľkosti hlavolamu.

```
START STATE:
6 0 4 7
3 2 5 1

GOAL STATE:
7 5 3 2
6 1 0 4
Depth: 22 Generated nodes: 375 Opened nodes: 257
Duration: 0.016545699999999997 seconds
```

```
START STATE:
1 2 3 8
6 0 12 11
5 15 14 7
9 13 10 4

GOAL STATE:
6 1 8 4
2 15 11 7
9 5 14 3
13 0 12 10
Depth: 30 Generated nodes: 9183 Opened nodes: 4708
Duration: 0.5549770999999986 seconds
```

Oliver Leontiev, 103027
UI, Zadanie 2 e)
2020

```
START STATE:
2 4 3 5
15 14 10 11
8 7 9 6
0 12 1 13

GOAL STATE:
2 11 5 6
12 4 10 3
14 13 15 7
0 8 9 1
Depth: 40 Generated nodes: 282933 Opened nodes: 146310
Duration: 9.747044500000001 seconds
```

```
START STATE:
1 5
3 2
7 6
8 4
9 0

GOAL STATE:
9 5
1 0
8 6
2 7
4 3
Depth: 35 Generated nodes: 17531 Opened nodes: 12270
Duration: 0.8547548000000003 seconds
```

Oliver Leontiev, 103027
UI, Zadanie 2 e)
2020

```
START STATE:
9 0
7 8
5 1
2 3
6 4

GOAL STATE:
7 5
6 4
1 9
2 0
3 8
Depth: 41 Generated nodes: 64092 Opened nodes: 46648
Duration: 3.4271606000000006 seconds
```

Porovnanie heuristík

Prvá heuristika je jednoznačne výrazne pomalšia ako druhá. Generuje a otvára oveľa viac uzlov. Taktiež výpočet oboch funkcií má obdobnú zložitosť (m x n), takže samotný výpočet do tohto rozdielu prispieva iba minimálne.

Príklady:

1. 3x3

Všimnime si najmä obrovský rozdiel v počte rozvitých uzlov.

```
START STATE:
7 2 6
5 3 8
1 0 4

GOAL STATE:
0 6 4
8 3 7
1 5 2
```

Oliver Leontiev, 103027
UI, Zadanie 2 e)
2020

1. heuristika:

```
Depth: 27 Generated nodes: 61813 Opened nodes: 44289
Duration: 2.8778854000000003 seconds
```

2. heuristika:

```
Depth: 27 Generated nodes: 5068 Opened nodes: 3300
Duration: 0.21066479999999999 seconds
```

2. 5x2

```
START STATE:
0 5
8 2
1 6
7 4
3 9

GOAL STATE:
9 2
3 5
1 0
4 7
8 6
```

1. heuristika

```
Depth: 41 Generated nodes: 718683 Opened nodes: 595729
Duration: 41.1160476 seconds
```

2. heuristika

```
Depth: 41 Generated nodes: 78542 Opened nodes: 58105
Duration: 4.1729899 seconds
```

Oliver Leontiev, 103027
UI, Zadanie 2 e)
2020

3. 2x3

```
START STATE:
3 4 1
2 5 0

GOAL STATE:
0 3 1
4 5 2
```

1. heuristika

```
Depth: 15 Generated nodes: 127 Opened nodes: 97
Duration: 0.0050848000000005555 seconds
```

2. heuristika

```
Depth: 15 Generated nodes: 86 Opened nodes: 66
Duration: 0.0036061000000007226 seconds
```

Vidíme, že pri komplexnejších hlavolamoch je druhá heuristika 10-krát rýchlejšia a vytvára 10-krát menej uzlov a aj rozvíja 10-krát menej uzlov. Zároveň vidíme, že medzi všetkými týmito veličinami je úzky vzťah. Pri menej zložitých hlavolamoch je rozdiel menej výrazný.

Výmena stavov

Čo ak vymeníme cieľový a štartovací stav?

Príklady:

1. druhá heuristika

```
START STATE:
1 8 4
0 2 5
3 7 6

GOAL STATE:
2 8 0
6 3 7
1 5 4
Depth: 21 Generated nodes: 597 Opened nodes: 362
Duration: 0.027481499999996828 seconds
```

Oliver Leontiev, 103027
UI, Zadanie 2 e)
2020

```
START STATE:
2 8 0
6 3 7
1 5 4

GOAL STATE:
1 8 4
0 2 5
3 7 6
Depth: 21 Generated nodes: 391 Opened nodes: 236
Duration: 0.016755199999998638 seconds
```

2. prvá heuristika

```
START STATE:
3 4 2
7 1 8
6 0 5

GOAL STATE:
0 8 2
1 6 5
3 4 7
Depth: 23 Generated nodes: 19445 Opened nodes: 12691
Duration: 0.8330419000000013 seconds
```

```
START STATE:
0 8 2
1 6 5
3 4 7

GOAL STATE:
3 4 2
7 1 8
6 0 5
Depth: 23 Generated nodes: 17908 Opened nodes: 11561
Duration: 0.779551399999999 seconds
```

Oliver Leontiev, 103027
UI, Zadanie 2 e)
2020

Program pri výmene uzlov môže cestu nájsť v inom čase a s iným počtom prejdených uzlov (rozdiely sú ale malé). Taktiež môže nájsť inú cestu, ale ciest je často viac a aj heuristiky zvyknú nájsť cesty odlišné.

Zhodnotenie

Riešenie bolo vytvorené v programovacom jazyku Python 3, ktorý uprednostňuje prehľadnosť a jednoduchosť pred efektivitou. Pre zrýchlenie programu a zvýšenie jeho efektivity, najmä pri väčších hlavolamoch, by bolo prvým krokom zmeniť implementačné prostredie (napríklad na jazyk C). Taktiež by sme ušetrili čas na konvertovaní polí do stringu, aby sa dali zahashovať do hashsetu, čo robíme pri každom vytvorení uzla.

Zistili sme, že prvá heuristická funkcia, počet zle umiestených políčok, je neporovnateľne menej efektívna ako druhá (súčet vzdialeností políčok od ich cieľovej pozície), a že výmena štartovacieho a cieľového stavu vie ovplyvniť efektivitu nájdenia riešenia.

A* algoritmus sa ukázal ako silný nástroj, ktorý nám umožňuje riešiť aj väčšie hlavolamy, vďaka výraznému zníženiu prejdených uzlov - pokiaľ máme dobrú heuristiku (v našom prípade druhú).