Umelá inteligencia, zadanie č.2

Prehľadávanie stavového priestoru (d)

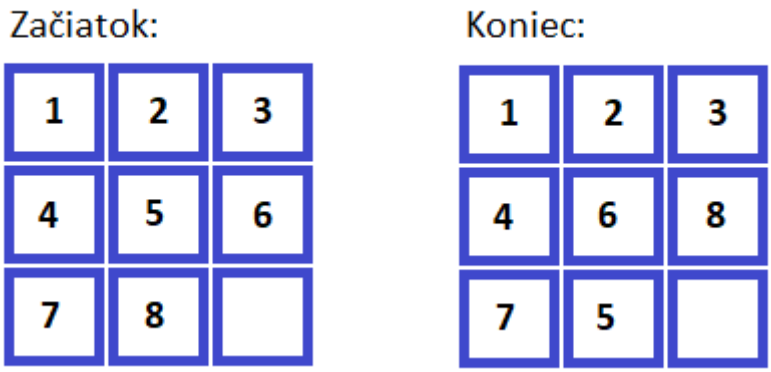
Patrícia Huďanová

Definovanie riešeného problému

Problém 2. Použite algoritmus lačného hľadania, porovnajte výsledky heuristík 1. a 2.**

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:

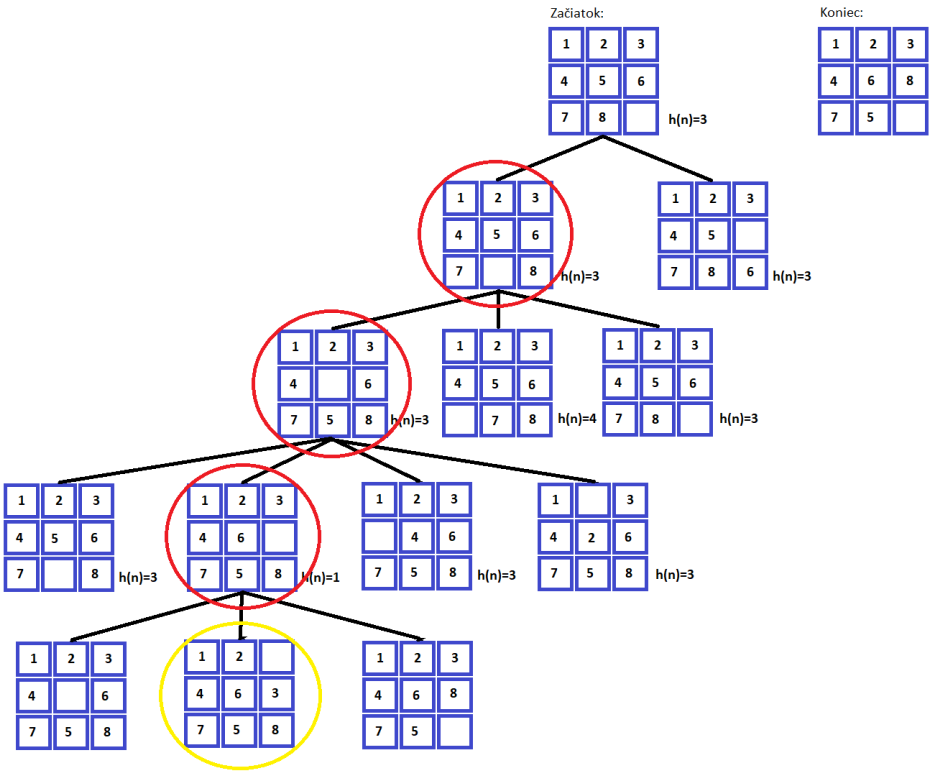


Im zodpovedajúca postupnosť krokov je: **VPRAVO, DOLE, VĽAVO, HORE**.

Opis riešenia

Mojou úlohou bolo implementovať riešenie 8-hlavolamu (MxN hlavolamu) s použitím lačného algoritmu. Tento algoritmus postupne generuje stavy do ktorých sa môže hlavolam dostať s tým, že si ako ďalší stav ktorý bude rozvíjať vždy vyberie ten, ktorý má najmenšiu odhadovanú vzdialenosť od cieľového stavu. Túto vzdialenosť vypočítavam pomocou dvoch heuristík:

- počet políčok na svojom mieste
- súčet vzdialenosti jednotlivých políčok od ich cieľovej pozície



Na obrázku som ilustrovala príklad použitia lačného algoritmu s heuristikou o počte políčok ktoré nie sú na svojom mieste.

Na implementáciu som použila PriorityQueue(), ktorá zabezpečí správne prehľadanie stavov.

Voľba vhodnej paradigmy a programovacieho jazyka

Najideálnejšie sa mi zdalo program riešiť objektovým prístupom a preto som zvolila na implementáciu jazyk Python.

Spracovanie vstupu

Po spustení programu musí používateľ zadať cestu k súboru s hlamolamami, ktoré chceme vyriešiť (moje testovacie súbory sú uložené v priečinku input_puzzles).

Vstupný súbor vyzerá nasledovne:

- prvý riadok: prvé číslo - počet stĺpcov hernej plochy; druhé číslo - počet riadkov hernej plochy
- druhý riadok: postupnosť čísel vyjadrujúca začiatočný stav zapísaný postupne po riadkoch
- tretí riadok: postupnosť čísel vyjadrujúca koncový stav zapísaný postupne po riadkoch
- číslo nula znázorňuje prázdne políčko

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

Voľba vhodnej reprezentácie stavov

Stavy mám reprezentované triedou State()

Atribúty:

```
# heuristic - hodnota vypocitana heuristikou pre daný stav
# parent - predosly stav z ktoreho sme sa dostali do aktualneho
# data - aktualny stav
# depth - hlbka v ktorej sa nachadza stav
# move - pohyb ktorym sme sa dostali do stavu(UP,DOWN,LEFT,RIGHT)

    def __init__(self, heuristic, depth,  parent, init_state,move):
        self.heuristic = heuristic
        self.parent = parent
        self.data = init_state
        self.depth = depth
        self.move = move
```

Hlavná metóda

generate_children: funkcia vráti všetky možné stavy do ktorých sa vieme z pôvodného stavu dostať

Hlavolam mám reprezentovaný triedou Puzzle()

Atribúty:

```
# priority_queue - prioritny rad pouzivany na vyber stavu v greedy_algo
# start - pociatocny stav ako postupnost cisel
# goal - cielovy stav ako postupnost cisel
# initial_state - pociatocny stav v 2D
# goal_state - cielovy stav v 2D

    def __init__(self, start_data, goal_data, init_puzzle_2D, goal_puzzle_2D):
        self.priority_queue = []
        self.start = start_data
        self.goal = goal_data
        self.initial_state = State(0,0,None,init_puzzle_2D,None)
        self.goal_state = State(0,0,None,goal_puzzle_2D,None)
```

Metódy

heuristic_misplaced_tiles: určí počet políčok ktoré nie sú na svojom mieste

heuristic_manhattan: určí súčet vzdialenosti políčok od cieľovej pozície

---

Algoritmus lačného hľadania

Ide o informované hľadanie, ktoré využíva heuristiku. Lačné hľadanie však nie je úplné a riešenie ku ktorému dospeje nie je optimalizované. Za úplné nemôže byť považované preto, že ak začne prehľadávanie smerom kam vedie nekonečná cesta, tak to nemusí nikdy rozpoznať. A riešenie neoptimalizuje pretože na základe heuristickej funkcie môže nájsť najskôr horšie riešenie.

V najhoršom prípade je časová aj pamäťová zložitosť O(m*n), kde m je faktorom vetvenia a n je maximálne hĺbka prehľadaného priestoru.

Implementácia prehľadávacieho algoritmu- lačný algoritmus

Algoritmus vo svoje podstate spočíva z nasledujúcich krokov:

1. vytvorenie objektu so začiatočným stavom (start)
2. pridanie tohto objektu do priority_queue
3. kontrola či priority_queue nie je prázdny - ak áno tak hlavolam nemá riešenie a algoritmus skončí
4. z priority_queue vyberieme stav
5. ak heuristika pre daný stav vypočíta hodnotu 0 tak sme dosiahli konečný stav a algoritmus skončí
6. prehľadanie stavu - vygeneruje všetkých možných susedov a vyráta im na základe heuristiky hodnotu
7. pridá všetky nové (ešte nenavštívené) stavy do priority_queue a opakuje algoritmus

   Nižšie môžeme vidieť implementáciu tohoto algoritmu (je mierne upravený oproti reálnemu kódu iba pre ukážku)

```
def greedy_search_algo(puzzle_board, COLUMN, ROW):
    # vytvorenie objektu so začiatočným stavom (start)
    start=puzzle_board.initial_state
    num=0
    start.heuristic=puzzle_board.heuristic(start.data, COLUMN, ROW)

    # pridanie tohto objektu do priority_queue
    heappush(puzzle_board.priority_queue, (start.heuristic, num, start))

    visited
    # kontrola či priority_queue nie je prázdny
    while len(puzzle_board.priority_queue):
        # z priority_queue vyberieme stav
        current = heappop(puzzle_board.priority_queue)[-1]

        visited.add(current.data)

        # ak heuristika pre stav vypočíta hodnotu 0 tak sme dosiahli konečný
stav
        if puzzle_board.heuristic(current.data, COLUMN, ROW)==0:
            print("Succesfully reached the goal state.")
            break

        # vygeneruje všetkých susedov stavu a vyráta im na základe heuristiky
hodnotu
        children = current.generate_children(puzzle_board, COLUMN, ROW,
heuristic_num)
        for child in children:
            # skontroluje ci uz bol stav navstiveny
```

```
            if tuple(child) not in visited:
                num=num+1
                # prida stav do radu
                heappush(puzzle_board.priority_queue, (child.heuristic, num,
child))
```

####

Testovanie

Testovanie som robila najprv na vzorových vstupoch ktoré sú k dispozícii na stránke predmetu (sú pre iný algoritmus, ale na začiatok stačili). A postupne som testovala zložitejšie vstupy. Následne som si vytvorila niekoľko vstupných testovacích súborov. Takisto som vytvorila súbor s neriešiteľnými vstupmi aby som otestovala či túto situáciu správne ošetrujem.

Pri testovaní som sledovala čas za ktorý sa nájde riešenie, koľko stavov sa prehľadá, koľko sa ich vytvorí a akú dĺžku má výsledná cesta.

Závery z testovania

Zo začiatku som testovala vstupy kde bolo na vyriešenie hlavolamu potrebné posunúť len jedno políčko a vstupy do 10 krokov.

Tabuľka nižšie napríklad ukazuje dva rôzne vstupy, kde som robila meranie dvakrát- druhé meranie je po zamenení koncového a začiatočného stavu. V tabuľke je pekne vidieť, že rozdiely neboli takmer žiadne.

Heuristika	Počet políčok na svojom mieste				Súčet vzdialenosti jednotlivých políčok od cieľovej pozície			
	Čas	Prehľadané stavy	Vygenerované stavy	Dĺžka výslednej cesty	Čas	Prehľadané stavy	Vygenerované stavy	Dĺžka výslednej cesty
3 2 0 1 2 3 4 5 1 0 5 3 2 4	0.000408	7	10	5	0.000368	6	8	5
	0.000452	7	10	5	0.000459	7	10	5
3 2 0 1 2 3 4 5 1 2 5 3 4 0	0.000205	4	5	3	0.000211	4	5	3
	0.000193	4	5	3	0.000209	4	5	3

Mojou úlohou bolo okrem implementácie lačného algoritmu aj porovnať 2 heuristiky:

1. počet políčok na svojom mieste
2. súčet vzdialenosti jednotlivých políčok od ich cieľovej pozície

   V tabuľke nižšie môžeme vidieť niekoľko výsledkov z testovania.

Heuristika	Počet políčok na svojom mieste				Súčet vzdialenosti jednotlivých políčok od cieľovej pozície			
	Čas	Prehľadané stavy	Vygenerované stavy	Dĺžka výslednej cesty	Čas	Prehľadané stavy	Vygenerované stavy	Dĺžka výslednej cesty
3x2	0,0015	56	77	21	0,0022	70	93	21
3x2	0,0041	146	198	19	0.0037	112	150	27
3x3	0,0428	1047	1863	62	0,0185	374	656	58
3x3	0.0121	308	533	45	0,0092	203	352	49
4x3	0.5313	9727	20702	169	0.0764	1357	2582	125
4x3	1.9398	28923	67537	261	0.1672	1295	2432	125
5x2	0.4607	10696	18008	223	0.0437	960	1521	121
6x2	12.9428	156134	333626	322	0.1893	3658	6187	256

Na základe testovanie som dospela k záveru, že použitím druhej heuristiky nájdeme riešenie rýchlejšie a zároveň aj častejšie nájdeme optimálnejšiu cestu než pri použití prvej heuristiky.

Zhodnotenie

Myslím si že moje riešenie spĺňa zadanie. Avšak najideálnejšie by asi bolo vytvoriť aj grafické rozhranie s možnosťou vykreslenia celej cesty od začiatočného stavu po cieľový (pôvodne som mala implementované vykresľovanie do textového súboru, ale pri testovaní viacerých vstupov naraz vykresľovanie trvalo pridlho).

Nájdenie riešenia hlavolamu ak nejaké existuje však funguje spoľahlivo.

Zdroje

Návrat a kol.: Umelá Inteligencia, STU Bratislava, 2002, 2006, 2015. (Malé centrum)