Slovenská technická univerzita

Fakulta informatiky a informačných technológií

Ilkovičova 3, 842 16 Bratislava 4

Umelá Inteligencia

Zadanie 2

Prehľadávanie stavového priestoru

8-hlavolam - algoritmus obojsmerného hľadania

Matsveyeva Lada-Ivanna

ZS 2021/22

Obsah:

Zadanie úlohy

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.

Implementácia 2

Keď chceme túto úlohu riešiť algoritmami prehľadávania stavového priestoru, musíme si konkretizovať niektoré pojmy:

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*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, VLAVO 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.

Príklad použitia operátora DOLE:

Vstup:

((1 2 3) (4 5 6) (7 8 m))

Výstup:

((1 2 3) (4 5 m) (7 8 6))

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. Stavy 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
- HĹBKA 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 neznamená rovnako dobrý uzol.

Činnosť nasledujúcich algoritmov sa dá z implementačného hľadiska opísať nasledujúcimi všeobecnými krokmi:

1. Vytvor počiatočný uzol a umiestni ho medzi vytvorené a zatiaľ nespracované uzly
2. Ak neexistuje žiadny vytvorený a zatiaľ nespracovaný uzol, skonči s neúspechom – riešenie neexistuje
3. Vyber najvhodnejší uzol z vytvorených a zatiaľ nespracovaných, označ ho aktuálny
4. Ak tento uzol predstavuje cieľový stav, skonči s úspechom – vypíš riesenie
5. Vytvor nasledovníkov aktuálneho uzla a zaraď ho medzi spracované uzly
6. Vytrieď nasledovníkov a ulož ich medzi vytvorené a zatiaľ nespracované
7. Choď na krok 2.

Uvedené kroky sú len všeobecné a pre jednotlivé algoritmy ich treba ešte vždy rôzne upravovať a optimalizovať.

Opis riešenia

Reprezentácia údajov:
Uzol je reprezentovaný triedou *Node*.

```
class Node:
    def __init__(self, p_state, previous, depth, number, opr):
        self.state = p_state
        self.previous = previous
        self.depth = depth
        self.number = number
        self.opr = opr

    def __lt__(self, other):
        return self.number < other.number
```

- state – aktuálny stav uzlu
- previous – rodič uzla
- depth – hĺbka uzla
- number – číslo vytvoreného uzla v danom strome
- opr – operátor vďaka ktorému vznukol tento uzol

Na základe číslu uzla pridávam do haldy.

Na ukladanie prehľadaných stavov pre štartovací stav používam list *src_visited*, pre koncové stavy *dst_visited*.

Všetky vytvorené stavy ukladám do *src_created* a *dst_created*, ich používam pri kontrole či je daný stav existuje.

Do haldy *src_heap* a *dst_heap* ukladám všetky nové stavy, ktoré je potrebné prehľadať.

Do listu *created* predávam stavy, ktoré boli prehľadané a sú unikátne.

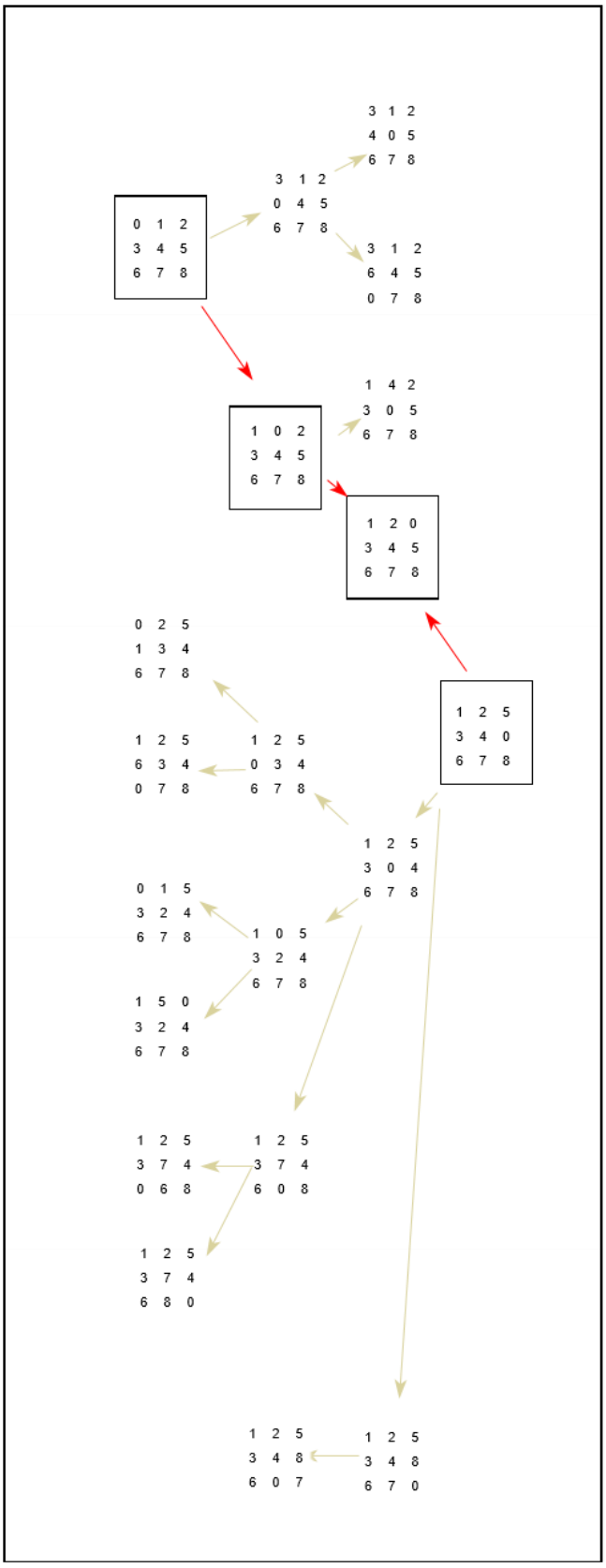
Pri riešení problému používam prehľadávanie do šírky, striedavanim od počiatočného a koncového stavov.

Počiatočný a koncové stavy sa nacitavuju zo súborov. Je možnosť načítavať m*n.

Opis algoritmu:

1. Pre začiatočný stav vytvoriť všetky možné uzly, používaním operátorov dole, hore, vľavo, vpravo.
2. Hýbem sa prázdnym políčkom, ktoré je reprezentované 0.
3. Nové stavy pridám do zoznamu, aby sa necyclil, tiež do haldy.
4. Spracujem 1, 2, 3 kroky aj pre koncový stav.
5. Ak stavy od začiatočného a koncového stavov sú unikátne pridám ich do zoznamu spracovaných uzlov
6. Pokračujem v 1-5 krokoch, kým nenájdem v 5 kroku spoločný stav. Vtedy viem, že v tomto stave počiatočný a konečný stavy sa pretínajú.
7. Vypíšem cestu od počiatočného do koncového stavu.

Ilustrácia riešenia problému 3*3



Na základe týchto stavov je vidieť ako rýchlo sa dá nájsť spoločne stavy, oproti tomu koľko je potrebné prihladat odnosmernym vyhľadávaním.

Spôsob testovania a zhodnotenie riešenia

Pri testovaní boli vyskúšané prípady 2*3, 3*3, 4*4.

Boli vyskúšané štartovacie a koncové stavy, aby počet operácií na spracovanie bol viac ako 10.

	3 * 2	3 * 3	4 * 3
Čas (s)	0.003	0.051	2.404
Počet vytvorených uzlov	43	83	204
Počet operátorov	13	16	20

Podľa výsledkov vidieť ako rastie čas prehľadávania pri zväčšení veľkosti hlavolamu, ale je stále oveľa menší ako pri jednosmernom BFS prehľadávaní. Nevyhodov tohto algoritmu je, že musí pamätať všetky unikátne stavy po prehľadávaní.

Vďaka tomu že sú známy počiatočný a koncový stavy to časová zložitosť je $O(b^{d/2} + b^{d/2})$, je to menšie pri BFS prehľadávaní - $O(b^d)$.

Program bol implementovaný v jazyke Python, ktorý podporuje požívanie heap-u a list-ov, ktoré boli vhodné na riešenie tohto problému. Program dokáže riešiť hlavolamy rôznej veľkosti, napríklad 2*3, 3*3, 4*3. Vstupné dáta sa načítavajú zo súboru.