Matej Delinčák, ID: 102895

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava 4

Riešiteľ 8-hlavolamu

2020/21, ZS

Matej Delinčák

Matej Delinčák, ID: 102895

Znenie zadania

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.

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

STAV

Stav predstavuje aktuálne rozloženie políčok.

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.**

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 políčok, ktoré nie sú na svojom mieste
2. Súčet vzdialeností jednotlivých políčok 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. 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 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 riešenie
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.

ÚLOHA

Použite A^* algoritmus, porovnajte výsledky heuristík 1. a 2.

Matej Delinčák, ID: 102895

Opis riešenia

Mojou úlohou bolo použiť A* algoritmus na nájdenie riešenia pre 8-hlavolam. Moje pomocné funkcie tu opisovať nebudem, opíšem len moju hlavnú funkciu *find_final()*, kde sa vykonáva všetko podstatné. V tomto algoritme som použil minimálnu haldu, pre výber stavu s najnižším súčtom heuristiky a vzdialenosti od začiatku.

Algoritmus začína vytvorením prvého uzla, ktorý vložím do haldy (táto halda ukladá na prvé miesto najmenší súčet heuristiky a hĺbky v strome a ak sa dva prvky rovnajú, tak ešte porovná samostatne ich heuristiku). V každom uzle mám uložené hodnotu heuristiky (aktuálne používanú heuristiku si vyberá užívateľ na začiatku programu), hĺbku stavu, smerník na predchádzajúci stav a aktuálny stav hracej plochy v podobe matice.

```
class Node:
    heuristic = -1
    distance = -1
    state = []
    predecessor = None
```

Algoritmus vkročí do cyklu, ktorý sa zastaví, až keď nájde uzol, ktorého stav ma hodnotu heuristiky 0. To znamená, že sa jedná o koncový stav.
V cykle sa uskutočňujú tieto veci:

- V danom stave sa nájde prázdne miesto.
- Pozriem sa, či je možné operátor VPRAVO uskutočniť. Ak áno, tak si vytvorím kópiu daného stavu a vymením v ňom hodnoty (operácia VPRAVO). Následne sa pozriem, či sa tento stav, už nenachádza v tabuľke už vytvorených stavov (toto je implementované pomocou *dictionary*). Ak sa nenachádza, vložím ho tam a vytvorím nový uzol stromu. Tento uzol pridám do haldy.
- Takto prejdem zvyšné tri operátory VĽAVO, HORE a DOLE.

Algoritmus vráti uzol finálneho stavu a pomocou funkcie *print_process()*, vypíšem postupnosť operátorov potrebných na prechod zo štartovacieho stavu do finálneho. V tejto funkcií používam smerník na predchodcu, ktorý sa nachádza v uzle.

Reprezentácia údajov a použitý algoritmus

Moje uzly mám reprezentované pomocou objektov. Skúšal som môj algoritmus prerobiť, aby nepoužíval objekty, ale nemenné množiny (tuple()). Čakal som, že môj program sa zrýchli, ale nepozoroval som žiadne zrýchlenie. Áno, takýmto prístupom by som ušetril na pamäti, ale rapídne by som nesprehľadnil program. Čiže som sa vrátil naspäť k objektom.

Ako som už opísal vyššie, použil som A* algoritmus. Ide v podstate o dijkstrov algoritmus, ale s tým, že je použitá nejaká heuristika. Heuristika je iným slovom odhad ceny cesty do cieľa. Mojou úlohou bolo porovnať dve dané heuristiky. K nim som pridal ešte tretiu, ktorá je v podstate len ich kombináciou. Použité heuristiky:

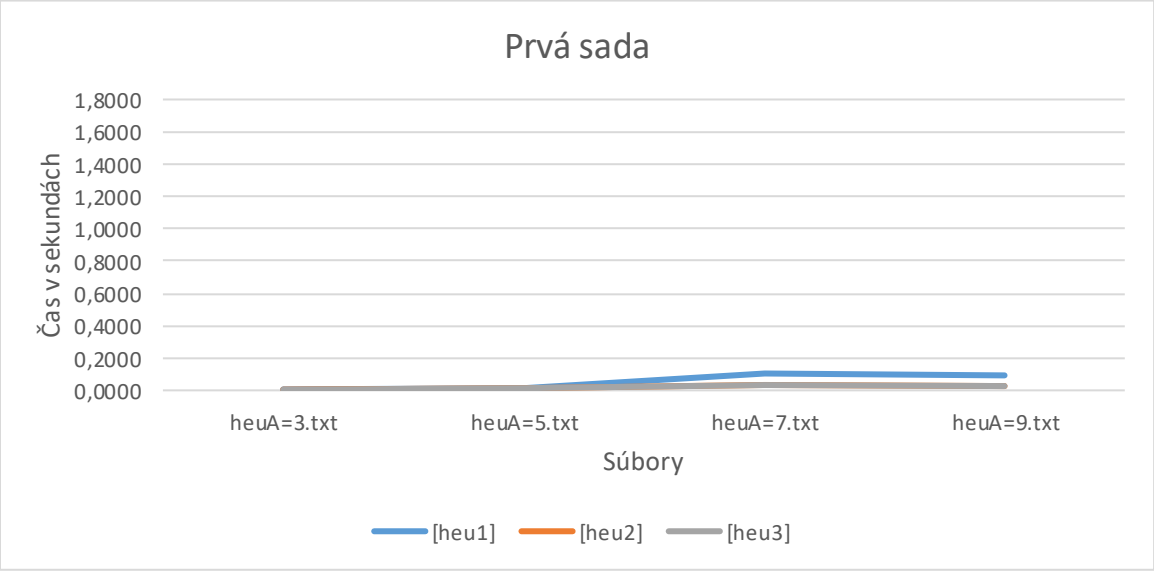
- Počet políčok, ktoré nie sú na svojom mieste (Ďalej označovaná ako [heu1])
- Súčet vzdialeností jednotlivých políčok od ich cieľovej pozície [heu2]
- Súčet prvej a druhej [heu3]

Ak heuristika vyráta menej krokov do cieľa ako v skutočnosti existuje, vyšlo mi, že je horšia. Naopak ak heuristika vyráta viacej krokov, ako by to to reálne trvalo, je rýchlejšia. Samozrejme, ak je odchýlka príliš veľká, táto výhoda sa stráca.

Spôsob testovania

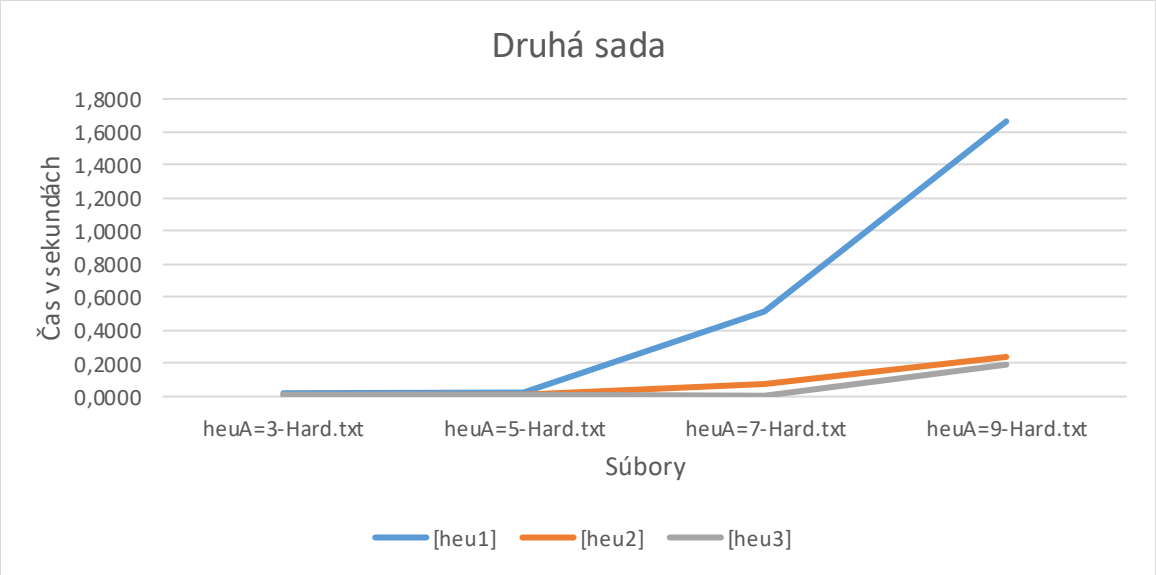
Testoval som dokopy tri heuristiky. Najskôr som otestoval, či dokáže každá z nich priviesť algoritmus k správnemu riešeniu. Na pár náhodných testoch, ktoré som si vygeneroval, som ručne podľa poradia operátorov na výstupe, zistil, či program prebehol v poriadku. Príslušné súbory sú *vstup1.txt, vstup2.txt* a *vstup3.txt*. Následne som prešiel k testovaniu efektivity.

V prvej sade som testoval najmä rýchlosti na ľahkých vstupoch. Išlo o vstupy, ktoré boli veľmi jednoduché, t.j. algoritmus mal poposúvať len pár políčok. Vo vstupných súboroch som zvyšoval postupne počet políčok, ktoré nie sú na svojich miestach. Súbor *heuA=3.txt* mal práve tri políčka na inom mieste. *HeuA=5.txt* ich mal päť, atď. Nižšie uvediem graf s časmi jednotlivých heuristík.



Ako môžeme vidieť, ani jedna z heuristík moc nezaostáva za druhými. Heuristika [heu1] mierne zaostáva, ale rozdiel v sekundách je pre človeka nebadateľný.

V druhej sade vstupných dát, som sa zameral na prípady, ktoré útočia na heuristiku [heu1]. Tu je vidno, že táto heuristika je veľmi zlá, a teda bude razantný rozdiel medzi [heu1] a ostatnými heuristikami. Súbory (*HeuA=3-Hard.txt*, ...) mám zase zoradené podľa počtu zle uložených políčok, ale už nie sú jednoduché ako v prvej sade.



Vidíme, že heuristika [heu1] výrazne zaostáva za druhými dvoma. Je to spôsobené tým, že táto heuristika odhaduje cestu veľmi zle a jej obor hodnôt je v podstate len od 0 po 9. Kdežto v ďalších je obor hodnôt väčší a zároveň sú aj presnejšie.

Najrýchlejšia heuristika však vyšla tá tretia. Táto bola o chlp lepšia ako [heu2]. Ide o veľmi malý rozdiel, takže zase si to ani poriadne nevšimneme.

Používateľské prostredie

Pre môj program som zvolil interaktívne prostredie. Užívateľ ma na výber:

- Načítať vstup zo súboru v tvare:
  - „počet riadkov“
  - „počet stĺpcov“
  - „začiatočný stav“
  - „koncový stav“
- Zvoliť rôzny počet riadkov a stĺpcov – nie len 8-hlavolam
- Vybrať si z dostupných heuristík
- Vybrať si počet zopakovaní výpočtu úlohy. POZOR! Postupnosť operandov sa vypíše iba pri možnosti 1, ak užívateľ napíše väčšie číslo, žiadne operandy sa nevypíšu.

Ako môže vyzerať vstup pre 3x3 hlavolam – medzi číslami a „m“ sú medzery:

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

Ako nemôže vyzerať:

1 2 3 m
m a 2 3 4 5 6 7 8 9 m 11 12

Pred každým začatím programu sa ešte vyráta, či je daný vstup vôbec vykonateľný.

Ak si teda užívateľ vybral počet opakovaní jedna, vypíše sa postupnosť operandov ako sa dostane z počiatočného stavu do finálneho a na konci sa vypíše čas trvania. Ak si zvolí možnosť opakovať program viac krát, operandy sa nevypisujú. A na konci sa vypíše priemerný čas trvania (vhodné na testovanie rýchlosti).

Zhodnotenie riešenia a rozšíriteľnosť

Moje riešenie by sa dalo určite o pár sekúnd zrýchliť, ale mňa už nenapadá ako. Ako prvé vylepšenie bolo, že som použil hash tabuľku namiesto obyčajného poľa pre uchovávanie vytvorených vrcholov. Ako druhý krok bolo zmenenie uzlov, ktoré som si ukladal do tejto tabuľky. Najskôr sa tu nachádzali uzly, ktoré som už rozbalil. Toto vytváralo zbytočné duplikáty. Tak som to prerobil na tabuľku vytvorených uzlov. Toto zase zrýchlilo trošku program. A na koniec som odstránil *deepcopy* objektu. Namiesto toho som použil len vytvorenie nového objektu s ručným nakopírovaním informácií, ktoré som potreboval. Tento počin program zrýchlil len nepatrne. Na koniec som ešte skúsil pozmeniť funkciu *choose_heuristic(),* ktorá sa volá pri každom rátaní heuristiky. Táto funkcia len rozhodne na základe globálnej premennej, na akú heuristiku sa program má presmerovať. Čakal som, že keď volám túto funkciu a vždy musím prejsť podmienkou, tak bude program pomalší a tak som to zmenil. Na začiatku programu som vybral jeden krát, akú heuristiku budem používať a už som volal len tú. No ale na môj podiv, sa program spomalil. Tak som túto úpravu vrátil späť.

A* algoritmus je určite výhodný, lebo pri porovnaní s kolegami, ktorí mali iné zadania, bol A* jeden z tých rýchlejších. Bohužiaľ, nemôžme to brať ako umelú inteligenciu, lebo stále je to len algoritmus, ktorý nájde riešenie hrubou silou. Rýchlejší algoritmus bol, ale greedy. Kde algoritmus berie len najmenšiu hodnotu heuristiky, žiadnu cestu zo štartu. Môj program by som dokázal zmeniť na greedy algoritmus, zmenou len v tom, podľa čoho bude halda usporiadávať. A to len podľa hodnoty heuristiky. Avšak tento algoritmus nemusí nájsť najoptimálnejšiu cestu, pretože hľadá prvú cestu, ktorá dokáže prísť do cieľa.

Ako jazyk som použil Python, už pre rýchlosť v akom som bol schopný nakódiť zadanie, ako aj pre množstvo vstavaných funkcií. Nevýhodou je samozrejme to, že tento jazyk je interpretovaný, a je samozrejme pomalší ako také C, ktoré by bolo výhodnejšie pre rýchlosť programu.

Čo sa týka rozšíriteľnosti programu, tak som ho programoval od začiatku tak, aby som mohol riešiť aj iné hlavolamy ako len 3x3. Všetko pracuje všeobecne na základe vstupu od užívateľa. Môj program dokáže riešiť aj 4x4 hlavolamy, len mu to v niektorých prípadoch dlho trvá. Takže program sa javí akoby sa zacyklil, aj keď tomu tak nie je.