Fakulta informatiky a informačných technológií STU v Bratislave

Ilkovičova 2, 842 16 Bratislava 4

Umelá inteligencia

8-hlavolam - A\* algoritmus

Ondrej Špánik  
ID: 103151

Meno cvičiaceho: Ing. Boris Slíž

Časy cvičení: Utorok 14:00

Akademický rok: 2021/22 ZS

Obsah

[1. Stručný opis zadania 3](#_Toc86097893)

[2. Opis riešenia a reprezentácia údajov problému 4](#_Toc86097894)

[3. Spôsob testovania 5](#_Toc86097895)

[Obsah spustiteľných .py súborov 5](#_Toc86097896)

[Obsah testovania / Legenda 5](#_Toc86097897)

[4. Výsledky experimentov/testovania 6](#_Toc86097898)

[5. Zhodnotenie riešenia a výsledkov 7](#_Toc86097899)

[Návrhy na zlepšenie 7](#_Toc86097900)

[6. Dodatok: Príručka na skompilovanie 8](#_Toc86097901)

[Minimálne požiadavky 8](#_Toc86097902)

[Spustenie bez kompilácie 8](#_Toc86097903)

[Kompilácia do .exe 8](#_Toc86097904)

[7. Dodatok: Sada testovacích príkladov 9](#_Toc86097905)

# Stručný opis zadania

Úlohou zadania **2 f** je naprogramovať A\* algoritmus pre 8-hlavolam, resp. n-hlavolam.

**A\* algoritmus** spočíva v rozšírení lačného hľadania o heuristickú funkciu – Manhattanovú vzdialenosť, ktorá bližšie cieli prehľadávanie smerom s väčším predpokladom úspechu.

Pre potrebu algoritmu je definovaná štruktúra stavu, operácie nad stavmi (ktoré definujú nové stavy), generovanie nových stavov a ich ukladanie do stromu stavov, ochrana proti zacykleniu (vo forme množiny prejdených stavov) a pomocné funkcie na výpočet cien pre A\* (Manhattanová vzdialenosť).

**Na vstupe algoritmu** je začiatočný a cieľový stav definovaný používateľom (prípadne vybraný z testovacej sady).

**Výstupom algoritmu** je sekvencia operácií potrebných na prechod zo začiatočného do cieľového stavu, alternatívne v prípade neschopnosti problém riešiť (viď. Sam Loyd 15-hlavolam) informačný výpis neúspechu.

Samotný **stav** je definovaný ako pole v rámci štruktúry stavu obohatenú o dodatočné identifikačné informácie.

Ďalej existujú **operátory,** konkrétne 4 na základe možných smerov ktorými sa môže meniť definovať nový stav: LEFT, RIGHT, UP, DOWN

Na základe operátorov je možné vykonávať **operácie**, výsledkom operácie je definícia nového stavu.

Existuje **heuristická funkcia** c = g + h, na základe ktorej sa vypočíta cena z novo-rozvetveného stavu do cieľového stavu a určuje postup algoritmu cez stavy.

# Opis riešenia a reprezentácia údajov problému

Prvá vôbec bola definovaná štruktúra stavu **\_state.py/State**, na ktorej vstupe je Python list reprezentujúci pole prvkov stavu. Spočiatku som premýšľal nad použitím array alebo tuple, čo ale komplikuje manipuláciu dát neskôr a nie je teda ideálne. Prvky sú uložené v atribúte **State.elms**.

Ako prvú funkciu som implementoval výpočet Manhattanovej vzdialenosti **\_alg.py/manhattanSum()** z momentálnej do cieľovej pozície pre všetky prvky začiatočného (resp. akéhokoľvek na vstupe) stavu.

Ďalej som si uvedomil, že daná sumárna funkcia je neefektívna pre ďalšie volania, nakoľko každý nový stav je vlastne delta posun jedného prvku (nepočítajúc prázdne políčko) predchádzajúceho stavu. Nie je teda vôbec nutné počítať pre všetky prvky Manhattanovú vzdialenosť odznova. Týmto spôsobom bola definovaná delta funkcia **\_alg.py/manhattanDelta(),** ktorej výsledok sa len pripočíta k existujúcej sume z predchádzajúceho stavu. Ďalej pre potreby tejto funkcie bola rozšírená štruktúra stavu o atribúty **State.mdelta, State.mpos**.

Na rade bola implementácia operátorov, operácií a generovanie nového stavu, resp. rozvetvenie predchádzajúceho.

* **Stromovú štruktúru** pre stavy som sa rozhodol implementovať priamo v rámci samotnej štruktúry stavu, keďže mi to prišlo ako jednoduchšie a optimálnejšie riešenie.
* **Operátory** sú implementované ako enum **\_state.py/StateOperator**, ktorý zároveň vďaka schopnosti uložiť dodatočnú hodnotu nie len popisuje ale aj definuje posun štyrmi smermi (LEFT, RIGHT, UP, DOWN).
* **Operácia** nad stavom, v ktorej sú definované všetky spôsoby generovania nových stavov pre všetky operátory je uložená v rámci triedy **\_state.py/State** ako funkcia **operation().** Operácia na vstupe berie jeden špecifický operátor z StateOperator, na základe ktorého je definovaný, generovaný a priamo do rodiča uložený ako špecifická vetva nový stav. Napr. current.operation(LEFT) zaručí relevantný obsah atribútu current.left, kde current je momentálny stav a left jeho potomok po vykonaní operácie s operátorom LEFT.
* **Delta** pre Manhattanovu vzdialenosť je vypočítaná pre budúcu novú vetvu stavu ako súčasť implementácie operácie nad stavom.
  + A\* algoritmus hneď po definovaní a generovaní stavu na základe vyššie spomenutých funkcií túto delta hodnotu zohľadňuje, takže sa jednalo o ideálnu pozíciu k jej výpočtu, kedy sú ešte priamo známe všetky rozdiely medzi rodičom a jeho novým potomkom.

Následne po implementovaní funkcií pre vytvorenie stromovej štruktúry a overení ich správnej funkčnosti prišiel na rad **samotný algoritmus.** Jadro je obsiahnuté v rámci funkcie **\_alg.py/explore()**, ktorá má za úlohu definovať prednastavené hodnoty, volať manhattanSum nad začiatočným stavom a manhattanDelta nad každým ďalším, počítať cenu c = g + h pre každý stav, stavy zoradiť a v poradí zohľadňujúcom najlacnejšiu cenu ďalej rekurzívne prehľadávať.

* Pre zoradenie od najlacnejšieho stavu je v algoritme použitá pomocná štruktúra obsiahnutá v **\_direction.py**
* Pre návrat výslednej sekvencie z rekurzie, jej reverz a výpis existuje ďalšia pomocná štruktúra **\_seq.py**

# Spôsob testovania

## Obsah spustiteľných .py súborov

Testovanie je separované od hlavného programu **\_\_main\_\_.py**, ktorý pýta vstup používateľa a je teda interaktívny.

Testovanie interaktívne nie je, nakoľko hlavnou úlohou mojej implementácie testera je spustiť viacero rôznorodých vstupov rovnaký počet krát normálne a reverzne. Z toho vychádzajú dva súbory **tester\_n3.py** a **tester\_n3\_reverse.py**.

## Obsah testovania / Legenda

Pod maximálnou rekurziou možno rozumieť maximálnu hĺbku, ktorú môže algoritmus dosiahnuť.

**Hlavný vstup:**  
Z [3,2,8,4,5,6,7,1,0] DO [1,2,3,4,5,6,7,8,0]:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Normálna hĺbka cieľa | Reverzná hĺbka cieľa |
| 1.1 | Predvolená max. rekurzia 1000 úrovní  (**prakticky bez limitu**, keďže v žiadnom teste nebola dosiahnutá) | 446 | 458 |
| 1.2 | Max. rekurzia **500 úrovní.** | 446 | 458 |
| 1.3.1 | Max. rekurzia **200 úrovní.** Výrazné náznaky zmeny správania, namiesto vertikálneho rastu bol väčší rast horizontálny. | 200 | 198 |
| 1.3.2 | Rovnaké ako 1.3.1, až na limitovanie výpisu na obrazovku.  Výpis sa ukázal za neproblematický (nemení výsledný čas). | nevypísané | nevypísané |
| 1.4.1 | Max. rekurzia **50 úrovní**. | 48 | 48 |
| 1.4.2 | Rovnako ako 1.4.1 bez výpisu. | nevypísané | nevypísané |

**Iné vstupy:**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Normálna hĺbka cieľa | Reverzná hĺbka cieľa |
| 2 | Z [5,0,8,4,2,1,7,3,6] DO [1,2,3,4,5,6,7,8,0] | 43 | 47 |
| 3 | Z [1,2,0,4,5,3,7,8,6] DO [1,2,3,4,5,6,7,8,0] | 2 (up, up) | 2 (down, down) |

Merania boli vykonané rovnakým štýlom aj reverzne, kde sa vymenil počiatočný a cieľový stav. Tie sú označené v grafoch písmenom „R“.

Testovanie bolo spustené 100x normálne aj reverzne. Hodnoty sú automaticky ukladané do CSV súborov, ktoré boli následne spracované manuálne v Exceli do grafov.

# Výsledky experimentov/testovania

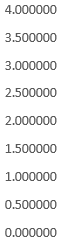
Merané výsledky sú závislé od výkonu PC, aplikácií, ktoré bežia na pozadí a pod. Samotná obťažnosť, ktorá sa za výsledkami skrýva je čiastočne zľahčená zo strany funkcie manhattanDepth. ktorá je na rozdiel od exponenciálnej manhattanSum lineárna.

Reverzné meranie ukázalo, že kým najťažší prípad v normálnom meraní (1.3) sa stal výrazne jednoduchším v reverze, tak opak je pravdou pre druhý najťažší prípad v normálnom meraní (1.4).

Výsledkom najpodstatnejšieho experimentu – úpravy hĺbky rekurzie medzi 1.1-4 je návrh na medziúrovňové zlepšenie v časti 5 tejto dokumentácie.

Experimenty taktiež ukázali, že print funkcie nemali žiaden merateľný vplyv na výsledky.

**Surové výsledky** v sekundách (ľavé normálne, pravé reverzné)  
(ľavý výkyv v strede spôsobil prehliadač otvorený na pozadí)



**Priemer** v sekundách (ľavé tmavozelené normálne, pravé bledozelené reverzné)

# Zhodnotenie riešenia a výsledkov

**Výsledok pozorovania hĺbkového limitu**

To, že výsledky s viac limitovanou rekurziou, resp. hĺbkou môže trvať výrazne dlhšie prepočítať ukázalo, že hlavným faktorom rýchleho nájdenia cieľového stavu nie je limitovanie hĺbky alebo šírky, ale skôr všeobecné limitovanie stavového priestoru. Keď algoritmus totiž narazil na hĺbkový limit, tak namiesto vertikálneho rastu začalo dochádzať k rastu horizontálnemu, ktorý môže byť rovnako ľubovoľne obťažný. Rovnakou ľubovoľnou obťažnosťou mám na mysli, že pre vertikálny aj horizontálny rast platí <0, nekonečno) a ktorýkoľvek môže byť viac zradný v rôznych prípadoch.

## Návrhy na zlepšenie

Ako býva zvykom, nie všetky implementácie sú najideálnejšie a priestor na zlepšenie som si všimol najmä v nasledovných skutočnostiach:

**Cena medzi úrovňami**

Efektívnosť algoritmu by mohla byť ďalej výrazne zlepšená doprogramovaním medzi-úrovňovej komunikácie pri prehľadávaní z hľadiska vyhodnotenia nie len najlepšej ceny na momentálnej úrovni, ale aj najlepšej ceny o celú úroveň, či dve nižšie. Túto skutočnosť som objavil až príliš neskoro, takže som sa nedostal k implementácií. Rovnako to potvrdilo aj testovanie s nízkym limitom rekurzie, kde pokiaľ sa v rámci limitu rekurzie nachádzal hľadaný stav vo vetve s vyššou cenou, tak k jeho objaveniu došlo výrazne plytšie: 3x3 [3,2,8,4,5,6,7,1,0] -> [1,2,3,4,5,6,7,8,0] hĺbka bez limitu 446, s limitom 50 v hĺbke 48. Zároveň avšak trpela rýchlosť, pretože sa jednalo až o príliš cenovo vzdialenú vetvu od ideálnej, takže finálny počet prejdených stavov musel byť vyšší.

**Rekurzia**

Python má voči C o dosť viac predvolene limitovaný stack (~ 1500 v Pythone, ~ 47000 v C na mojom PC). Táto hodnota sa dá upraviť, avšak treba byť opatrnejší ako pri C vzhľadom na to, že Python je len interpretovaná nadstavba nad C. Ideálnejšie riešenie by nezáležalo na rekurzií, teda bolo iteratívneho charakteru.

# Dodatok: Príručka na skompilovanie

## Minimálne požiadavky

* Vhodné mať nainštalovaný Python aspoň verzie 7.3.7.
* Keďže sa jedná o interpretovaný jazyk, kompilácia nie je nutná.
* Program sám o sebe nevyžaduje žiadne dodatočné balíky

## Spustenie bez kompilácie

K programu sa stačí presunúť pomocou „cd“ a spustiť pomocou príkazu „python .“ (čo spustí \_\_main\_\_.py), prípadne „python tester\_main.py“ pre tester.

## Kompilácia do .exe

V prípade nutnosti kompilácie sa mi osvedčil balík **pyinstaller**, ktorý treba nainštalovať pomocou „pip install pyinstaller“ a následne vygenerovať .exe súbor použitím „**pyinstaller -F \_\_main\_\_.py**“ v adresári programu. Vygenerovaný súbor bude uložený do pod-adresára **dist** ako jeden .exe súbor. Dodatočné pomocné súbory, ktoré možno zmazať budú v pod-adresári build

# Dodatok: Sada testovacích príkladov

Viď časť 3 dokumentácie: Obsah testovania, resp. priložené testery tester\_n3.py a tester\_n3\_reverse.py. Všetky priložené testovacie príklady sú pre rozmer hlavolamu 3x3 (teda 8-hlavolam).