21/10/2021

Prehľadávanie stavového priestoru

Umelá inteligencia – zadanie 2

Samuel Hetteš, ID: 110968
STU FIIT 2020/2021

OBSAH

ZADANIE

Toto zadanie sa venuje niekoľkým základným algoritmom prehľadávania stavového priestoru. Základná štruktúra algoritmu by mala byť podľa možností vyjadrená jednou funkciou.

V dokumentácii je potrebné uviesť použitý algoritmus a opísať vlastnosti tohto algoritmu – teoretické aj skutočné – koľko naozaj rozvíja uzlov, koľko mu to trvá a aké riešenie nájde. Použite *aspoň* dva rôzne príklady na hľadanie – vzdialenosť do cieľa napríklad 6 a 10 krokov. Porovnajte dosiahnutý výsledok pri zámene začiatočného a koncového uzla, ak to má zmysel (problém 2).

Základom hodnotenia je funkčnosť programu podľa špecifikácie a osobitná dokumentácia k programu. Dokumentácia musí obsahovať:

- riešený problém – názov špecifického zadania (a meno autora!)
- stručný opis riešenia a jeho podstatných častí
  - reprezentáciu údajov problému, použitý ***konkrétny*** algoritmus
- spôsob testovania a zhodnotenie riešenia
  - možnosti rozšírenia, prípadné optimalizácie, výhody a nevýhody konkrétnej implementácie (aj či sú závislé alebo nezávislé na programovacom prostredí)
  - porovnanie vlastností použitých metód pre rôznu dĺžku riešenia (u eulerovho koňa rôzne veľkosti šachovnice)

Do systému AIS je treba odovzdať elektronickú verziu dokumentácie plus okomentované zdrojové kódy. Najlepšie zabalené do jedného zip súboru.

Ďalej sa hodnotí:

- efektívna a prehľadná implementácia algoritmu (včítane komentárov)
- vhodné otestovanie činnosti algoritmu
- možnosť spracovania hlavolamu m*n (iný než 3x3, problém 2)

Nezabudnite, že ***hlavným výstupom programu*** je postupnosť krokov od začiatku k cieľu! (má byť reprezentovaná postupnosťou operátorov)

K programu je potrebné dodať sadu testovacích príkladov! Ak je program interpretovaný, môžu byť v tom istom súbore ako zdrojový kód.

Problém 2 – 8 hlavolam

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)
  - **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 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.

Špecifikácia zadania

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

OPIS RIEŠENIA

Daný problém som sa rozhodol riešiť v programovacom jazyku Python. To najmä z toho dôvodu, že sa nemusím starať o pamäť a taktiež poskytuje jednoduchú syntax.
Samotná stavba programu pozostáva z viacerých tried, ktoré bližšie opíšem. Získanie vstupných informácií a výpis výstupných informácií je zabezpečený mimo týchto tried. Začnem vysvetlením vedľajších tried a potom sa vrátim k hlavnej triede, ktorá predstavuje logiku algoritmu.

Triedy ManhattanHeuristic a MisplacedHeuristic

Obe tieto triedy obsahujú jednu statickú metódu, ktorej vstupnými argumentmi sú tabuľka a koncové pozície jednotlivých hodnôt v tabuľke (zistenie koncových pozícií bude vysvetlené v hlavnej časti algoritmu). Metódy iba vykonajú výpočet heuristiky a vrátia jej hodnotu.

Trieda Node

Táto trieda predstavuje jeden uzol. Pri inicializácii nového uzla sa vždy ukladá informácia o konfigurácii tabuľky, ktorú predstavuje, odkaz na rodiča, operátor, ktorý bol použitý pre vytvorenie tohoto uzla, hĺbka uzla (koreň má hĺbku 0) a vypočítaná celková kombinovaná heuristika.
Metódy:

- *possible_actions(self)* – táto metóda zistí všetky akcie, ktoré sú nad daným uzlom možné vykonať a vráti ich ako pole, kde sú uložené pätice tuple vo formáte - (operátor, x, y index aktuálnej pozície medzery, x, y index medzery po premiestnení)
- *perform_action(self, action)* – táto metóda vykoná danú konkrétnu akciu – vytvorí kópiu tabuľky a premiestni medzeru, túto vytvorenú tabuľku vráti ako návratovú hodnotu
- *steps_sequence(self)* – táto metóda rekurzívne pozbiera všetky použité operátory až ku koreňu a vytvorí postupnosť, ktorú vráti ako návratovú hodnotu

Teraz už poznáte vedľajšie triedy a môžeme pokračovať opisom získania vstupných informácií a hlavnou časťou algoritmu.

Vstupné informácie

Pri zapnutí programu sa používateľovi poskytne možnosť vybrať si jednu z nasledujúcich heuristík:

- Manhattan – súčet vzdialeností jednotlivých políčok od ich cieľovej pozície

- Misplaced Tiles – počet políčok, ktoré nie sú na svojom mieste
- Mixed – kombinácia predchádzajúcich dvoch

Ďalej si od neho program vypýta rozmery tabuľky – počet stĺpcov a počet riadkov. Po zadaní týchto údajov si zvolí počiatočnú a konečnú konfiguráciu tabuľky, samotná medzera je reprezentovaná '0'. Vstupy do tabuľky musia obsahovať unikátne čísla od 0 – (stĺpce x riadky). Pri veľkosti tabuľky 3x3 by čísla museli byť 0-8, inak program nebude korektne fungovať.

Po zadaní týchto údajov sa používateľovi ďalej poskytne možnosť zvolenia si limitov – počet operácií (rozvinutých uzlov) alebo hĺbka, do ktorej sa má prehladávať. Tieto informácie sú voliteľné.

Ďalej nasleduje inicializácia triedy Solver a spustenie samotného riešenia.

Príklad zadávania celého vstupu:

```
******************************
>>>      PUZZLE SOLVER      <<<
******************************

--- Heuristic configuration ---
'1' - Misplaced tiles
'2' - Manhattan
'3' - Mix of both
Enter the heuristic: 1

--- Table configuration ---
Enter the number of rows: 3
Enter the number of columns: 3

--- Starting table configuration ---
Enter the entries separated by spaces ('0' for space
):
Row #1: 1 2 3
Row #2: 4 5 6
Row #3: 8 7 0

--- Ending table configuration ---
Enter the entries separated by spaces ('0' for space
):
Row #1: 1 2 3
Row #2: 4 5 6
Row #3: 7 8 0

--- Solver limits configuration ---
For no limit enter '0'
Enter the iterations limit: 0
Enter the node depth limit: 0
```

Trieda Solver

Táto trieda obsahuje základ algoritmu, logiku vykonávania. Vstupnými informáciami pre túto triedu sú počiatočná a koncová konfigurácia tabuľky, a heuristika, ktorá má byť použitá. Pri inicializácii tejto triedy sa spustí funkcia *end_positions(self)*, ktorá vytvorí pole *end_positions* o veľkosti stĺpce x riadky. Funkcia prejde celou koncovou konfiguráciou tabuľky a na základe čísla uloží na príslušný index v poli *end_positions* tuple dvojicu, ktorá hovorí o indexoch (pozícii) daného čísla v koncovej konfigurácii tabuľky. Tieto informácie sú potom využívané pri heuristikách.

Samotný algoritmus obsahuje funkcia *solve(iter_limit, depth_limit)*. Na začiatku sa vytvorí prioritný rad do ktorého sa uloží inicializovaný štartovací uzol. Okrem toho sa taktiež vloží prvá unikátna konfigurácia tabuľky do setu. Začne sa vykonávať cyklus nad prioritným radom (kým nie je prázdny).

V tomto cykle sa najprv preusporiada prioritný rad na základe zvolenej heuristiky, z radu sa vyberie uzol s najväčšou prioritou a spustí sa funkcia *solved(table)*, ktorá iba porovná tabuľku uzla s koncovou tabuľkou a vráti hodnotu. Ak je táto hodnota 1, vráti sa postupnosť operátorov, ktoré k tejto konfigurácii viedli, hĺbka do ktorej sa prehľadávalo a počet rozvinutých uzlov. V opačnom prípade sa overí či nebol dosiahnutý limit operácií alebo limitná hĺbka (ak existuje), ak hej končí sa s nenájdeným výsledkom. Inak sa pre daný uzol zavolá funkcia *possible_actions* a vytvoria sa nové uzly. Pre každý uzol sa skontroluje či sa už nenachádza v unikátnom sete konfigurácii tabuliek, ak nie tak sa do setu konfigurácia pridá a rovnako sa pridá uzol do radu. Cyklus sa opakuje.

Výstupné informácie

Po skončení hľadania riešenia môže program dospieť k dvom fázam. Buď sa riešenie nenájde alebo nájde. V prípade nenájdenia riešenia je používateľ o tejto skutočnosti oboznámeny – buď boli prehľadané všetky možné konfigurácie alebo sa dosiahol jeden z limitov. V prípade nájdenia riešenia sú používateľovi poskytnuté informácie o počte rozvinutých uzlov, hĺbke prehľadávania, vykonávanom čase, postupnosti operátorov a počte krokov.

Príklad výstupu riešenia:

```
--- Summary ---
Heuristic used: Manhattan
Operators: L-Left, R-Right, U-Up, D-Down
Time taken: 0.0002 sec
Depth: 3
Expanded nodes: 3
Solution: L U L
Number of steps: 3
```

TESTOVANIE

Spôsob testovania je založený na porovnávaní počtu rozvinutých uzlov, hĺbky do ktorej sa prehľadávalo a čas, ktorý bol potrebný na vykonanie. Testy boli vykonané pri malej, strednej a veľkej vzdialenosti do cieľa pre tabuľky o veľkosti 3x3. Nasledujúce tabuľky ukazujú výsledky pri použitý jednotlivých heuristík.

- Výsledky pri použití malého počtu krokov do cieľa:

	Malý počet krokov do cieľa – 6/9		
Heuristika	*Rozvinuté uzly*	*Hĺbka prehľadávania*	*Čas vykonávania*
Misplaced Tiles	6/22	6/9	0.0002/0.0006 sec
Manhattan	6/11	6/9	0.0002/0.0003 sec
Mixed	6/10	6/9	0.0002/0.0004 sec

Ako môžeme vidieť pri počte krokov 6 sú výsledky identické. Naopak pri počte krokov 9 je heuristika Misplaced Tiles už menej efektívna, s dvojnásobne väčším počtom rozvinutých uzlov. V tomto prípade sa ako najlepšie heuristiky javia Manhattan a Mixed, ktorých výsledky sú takmer identické.

- Výsledky pri použití stredného počtu krokov do cieľa:

	Stredný počet krokov do cieľa – 12/16		
Heuristika	*Rozvinuté uzly*	*Hĺbka prehľadávania*	*Čas vykonávania*
Misplaced Tiles	102/506	12/16	0.0022/0.0145 sec
Manhattan	41/110	12/16	0.001/0.0026 sec
Mixed	44/126	12/18	0.0013/0.0038 sec

Z tejto tabuľky môžeme vidieť, že pri počte krokov 12 a 16 sa najefektívnejšia javí Manhattan heuristika, pričom tesne za ňou je Mixed heuristika. Mixed heuristika v tomto prípade dokonca pri druhom probléme našla dlhšiu cestu k výsledku. Čas potrebný na vykonanie ako aj počet rozvinutých uzlov sa pri Misplaced Tiles heuristike mnohonásobne zväčšuje.

- Výsledky pri použití veľkého počtu krokov do cieľa:

	Veľký počet krokov do cieľa – 20/24		
Heuristika	*Rozvinuté uzly*	*Hĺbka prehľadávania*	*Čas vykonávania*
Misplaced Tiles	4142/18996	20/24	0.4569/8.1538 sec
Manhattan	750/3284	20/24	0.0249/0.2942 sec
Mixed	792/1060	20/24	0.0331/0.0518 sec

Táto tabuľka opäť prináša veľmi zaujímavé informácie. Heuristika Misplaced Tiles je už takmer nepoužiteľná, rozvíja neskutočné množstvo uzlov a taktiež

potrebuje veľmi dlhý čas na vykonanie. Naopak zaujímavá zmena nastala pri heuristikách Manhattan a Mixed. Pri počte krokov 20 síce ešte víťazí Manhattan, ale pri počte krokov 24 už jasne zvíťazila Mixed heuristika ako aj počtom rozvinutých uzlov, tak aj časovo.

Zhodnotenie testovania

Na základe testovania a toho, čo nám ukázali dáta v tabuľke sme dospeli k nasledujúcim skutočnostiam:

- Pri malom počte krokov okolo 6 na výbere heuristiky nezáleží, všetky ponúkajú dobré výsledky. Avšak pri zvýšení počtu krokov na 9 sa už heuristika Misplaced Tiles javí menej efektívna.
- Pri strednom počte krokov okolo 12 – 16 jasne víťazí Manhattan. Čas vykonávania sa mnohonásobne zvyšuje pri Misplaced Tiles heuristike a veľmi dobré výsledky ponúka aj Mixed heuristika.
- Pri veľkom počte krokov okolo 20 si Manhattan stále drží prvenstvo, ale zvýšenie počtu krokov o 4 už zapríčinilo veľký pokles v efektivite oproti Mixed heuristike, ktorá poskytla relatívne malé množstvo rozvinutých uzlov a veľmi dobrý čas. Mixed heuristika je v týchto končinách už takmer nepoužiteľná.

ZHODNOTENIE RIEŠENIA

Myslím, si že riešenie, ktoré som použil sa javí ako efektívne. Využitie pamäte som sa snažil zredukovať čo najmenej tým, že neukladám v uzloch zbytočné informácie, ale len tie naozaj potrebné. Moje riešenie je možné taktiež použiť na ľuboľný hlavolam rozmeru M x N, čo taktiež vidím ako plus. Na druhú stranu za nevýhodu by sa dal považovať použitý samotný programovací jazyk. Python síce ponúka jednoduchú syntax, množstvo funkcií uľahčujúcich život a taktiež rieši pamäť za nás, no to sa odráža na jeho rýchlosti, ktorá nie je úplne dokonalá a taktiež využíva veľké množstvo pamäte.

ZÁVER

Myslím, že som pokryl všetky veci, ktoré sa od nás v tomto zadaní očakávali. Používateľovi som poskytol jednoduché prostredie a taktiež viacero možností nastavenia daného výpočtu. V poslednom rade by som rád ponúkol svoje zhodnotenie tohto zadania. Zadanie sa mi veľmi páčilo, implementácia nebola až tak časovo náročná a taktiež poskytlo veľmi zaujímavé výsledky. Ako úvod do fungovania stavového priestoru považujem toto zadanie za ideálne.