Meno: Martin Pažický
ID: 103086

Umelá Inteligencia

Zadanie č.2 – Prehľadávanie stavového priestoru

(Problém 1-a)

FIIT STU

24.10.2020 Martin Pažický

Meno: Martin Pažický
ID: 103086

1) Úvod

Mojou úlohou bolo riešiť hlavolam Bláznivá Križovatka pomocou algoritmov prehľadávania do šírky (BFS) a do hĺbky (DFS) a následne porovnať vlastnosti týchto algoritmov.
Hlavolam bláznivá križovatka pozostáva z mriežky (6x6), na ktorej sú rozmiestnené vozidlá tak, aby sa neprekrývali. Všetky vozidlá majú šírku 1 políčko, autá sú dlhé 2 a nákladiaky sú dlhé 3 políčka. V prípade, že vozidlo nie je blokované iným vozidlom alebo okrajom mriežky, môže sa posúvať dopredu alebo dozadu, nie však do strany, ani sa nemôže otáčať. V jednom kroku sa môže pohybovať len jedno vozidlo. V prípade, že je pred (za) vozidlom voľných *n* políčok, môže sa vozidlo pohnúť o 1 až *n* políčok dopredu (dozadu). Ak sú napríklad pred vozidlom voľné 3 políčka (napr. oranžové vozidlo na počiatočnej pozícii, obr. 1), to sa môže posunúť buď o 1, 2, alebo 3 políčka.
Hlavolam je vyriešený, keď je červené auto (v smere jeho jazdy) na okraji križovatky a môže z nej teda dostať von. Predpokladajte, že červené auto je vždy otočené horizontálne a smeruje doprava. Je potrebné nájsť postupnosť posunov vozidiel (nie pre všetky počiatočné pozície táto postupnosť existuje) tak, aby sa červené auto dostalo von z križovatky alebo vypísať, že úloha nemá riešenie.

2) Vlastnosti zvolených algoritmov

Obomi algoritmami je možné prehľadať celý stavový priestor. Hlavný rozdiel medzi nimi je v akom poradí prehľadávajú jednotlivé stavy, z čoho potom vyplývajú aj ďalšie rozdiely.
Prehľadávanie do šírky stavy prehľadáva v poradí v akom boli objavené, ide teda postupne po vrstvách stromu, dôsledkom čoho je nájdenie optimálneho riešenia (najkratšia cesta), ale aj vysoká pamäťová náročnosť.
Toto poradie prehľadávania je zabezpečené pomocou FIFO dátovej štruktúry – fronty[3].
Pri prehľadávaní do hĺbky sa naopak vždy rozvinie jedna vetva stromu až dokonca. Keď je vetva na konci, algoritmus sa vráti na posledný nerozvinutý stav. Toto poradie prehľadávania je zabezpečené pomocou LIFO dátovej štruktúry – zásobník[3].
Hlavnou nevýhodou tohto postupu je nájdenie neoptimálneho riešenia. Naopak pamäťová náročnosť DFS je nižšia.
DFS by malo byť taktiež rýchlejšie ako BFS, avšak existujú výnimky, napríklad keď je riešenie veľmi blízko pri koreni stromu, môže byť BFS, ktoré strom prehľadáva po jednotlivých vrstvách, efektívnejšie ako DFS[3].

3) Riešenie a reprezentácia údajov

Postup pozostáva z generovania stavov(uzly grafu) a ich následného prehľadávania. Stavy sú reprezentované ako reťazce, s ktorými pracujem ako s dvojrozmerným polom znakov (mapa) pre lepšiu abstrakciu. Na mape sú jednotlivé vozidlá reprezentované unikátnymi písmenami a prázdne miesta ako nuly.
Prvý stav sa vygeneruje zo vstupu, následne sa stavy generujú pohybmi vozidiel po mape.
Vozidlo reprezentujem ako objekt, ktorému ukladám základné údaje (súradnice, písmenová reprezentácia, orientácia, dĺžka) pre lepšiu abstrakciu.
Keďže s reťazcom pracujem ako s dvojrozmerným poľom, potreboval som pomocné funkcie, ktoré vykonávali predovšetkým prevody medzi dvojicou súradníc (riadok, stĺpec) a indexom v reťazci.
Prvým dôležitým mechanizmom riešenia je teda vygenerovať zo zadanej mapy všetky dosiahnuteľné mapy.
Na jednej mape sa vždy vykonajú všetky možné pohyby všetkých vozidiel (mapa sa postupne

prechádza po znakoch, keď sa nájde znak reprezentujúci vozidlo, vykonajú sa s ním všetky možné pohyby).
Mechanizmy pohybov a teda generovanie nových stavov sú totožné pri oboch použitých algoritmoch.
Všetky vygenerované mapy je potom potrebné zaradiť do rady na prehľadávanie (BFS – fronta, DFS – zásobník).

Nosným cyklom riešenia je teda vyťahovanie stavov z rady a následne objavovanie nových dosiahnuteľných stavov. Cyklus beží až po kým sa rada nevyprázdni alebo pokým sa nevytiahne stav, ktorý je riešením problému (hlavné auto je v poslednom stĺpci). Pokiaľ sa rada vyprázdni, zadanie nemá riešenie.

Ďalšou podstatnou časťou riešenia je spätné nájdenie cesty k finálnemu stavu.
Na to využívam hashmapu – každú novú nájdenú mapu uložím do hashmapy ako kľúč a ako jej príslušnú hodnotu uložím mapu, ktorou som novú mapu vygeneroval.
Keď sa pri prehľadávaní nájde finálny stav, využijem tuto hashmapu na vystopovanie cesty až k počiatočnému stavu.
Týmto riešením by som teda zistil všetky mapy ktorými som sa dostal k finálnej, avšak potrebná je postupnosť krokov. K tomu využívam ďalšiu hashmapu, kde ukladám dvojice - mapa(konkrétne hashcode mapy, kvôli efektivite) a krok ktorým bola dosiahnutá.
Prvá spomenutá hashmapa má však ešte jedno využitie – ukladá už objavené stavy, aby sa zabránilo zacykleniu. Pokiaľ sa vygeneruje nová mapa a už sa nachádza v hashmape ako nejaký kľúč, táto mapa sa do radu znova nezaradí, pretože raz už prehľadaná bola.

4) Testovanie

V tejto sekcii sa budem venovať skôr spôsobu testovania funkčnosti programu, konkrétne údaje testov (čas, pamäť) budem rozoberať v sekcii o porovnávaní algoritmov.

a) Vstup a výstup:

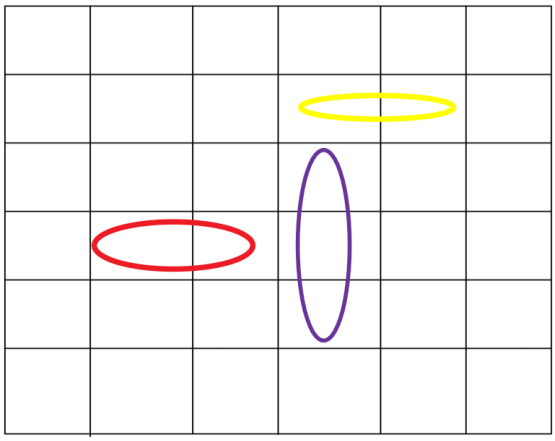
Vstup aj výstup je v rovnakom formáte ako je špecifikované pri zadaní.
Formát vstupu: ((názovauta, dĺžka, riadok, stĺpec, orientácia)(vozidlo2)...)
- auto ktoré treba vyslobodiť je zadávané ako prvé
Formát výstupu: očíslovaná postupnosť pohybov (SMER (názov počet)) jednotlivých vozidiel, ktorou sa dostanem do finálneho stavu.
Pri výstupe je ešte možnosť zapnutia prepínača printMaps na true a okrem postupnosti krokov sa budú vykreslovať aj jednotlivé mapy.

b) Testy:

Najprv som testoval funkčnosť riešenia na jednoduchých mapách, ktoré som vymyslel. Zadal som mapu, nakreslil som si ju (OneNote) a následne som nakreslenú mapu vyriešil pomocou pohybov z výstupu.
Pri jednoduchých mapách sa optimálnosť riešenia dala skontrolovať ručne (BFS). Samozrejme, overil som aj ukážkový vstup zo zadania.

Meno: Martin Pažický
ID: 103086

Náročnejšie mapy som čerpal zo stránky https://www.michaelfogleman.com/rush/[1], kde bol dostupný aj simulátor na otestovanie výstupu a taktiež bol pri vstupoch vypísaný počet krokov optimálneho riešenia. Otestoval som aj vstupy bez možného riešenia.
Jednotlivé testy sú uložené v triede Main ako reťazce a taktiež v samostatnom súbore tests.txt.
Tiež som s pomocou programu skúšal prechádzať levely hry[2].



Jednoduchá mapa - OneNote

```
String simpleInput2 = "((cervene 2 4 2 h)(zelene 3 3 4 v)(zlte 2 2 4 h))";
```

Jednoduchá mapa - vstup

```
1 VLAVO(zlte 2)
2 HORE(zelene 2)
3 VPRAVO(cervene 3)
--- 41 unique states generated ---
Runtime: 0.0067434 seconds
```

Jednoduchá mapa - výstup

5) Porovnanie použitých algoritmov

Použité algoritmy som porovnával z troch hľadísk:

1. optimálnosť riešenia (počet krokov (hĺbka))
2. rýchlosť nájdenia riešenia (meraná v Jave pomocou System.nanoTime())
3. pamäťová náročnosť (množstvo vygenerovaných unikátnych stavov)

Názov	Počet vozidiel	Hĺbka optimálneho riešenia	BFS hĺbka riešenia	DFS hĺbka riešenia	BFS čas (s)	DFS čas (s)	BFS počet vygenerovaných unikátnych stavov	DFS počet vygenerovaných unikátnych stavov
simpleInput1	2	2	2	3	0,0049255	0,0047656	17	11
simpleInput2	3	3	3	5	0,0060484	0,0049813	41	23
simpleInput3	5	5	5	33	0,0171095	0,0074485	376	163
sampleInput	8	8	8	91	0,031102	0,0125944	1079	496
hardInput1	10	23	23	246	0,0556288	0,030768	1970	1577
hardInput2	13	26	26	349	0,0632259	0,0253858	3442	1322
hardInput3	9	38	38	102	0,0199832	0,0122551	615	356
hardInput4	13	49	49	1835	0,2094244	0,080714	16232	9621
hardInput5	13	51	51	349	0,0582758	0,0287706	3202	1530
blockedInput	9	-	-	-	0,0138266	0,0132866	-	-

Porovnanie algoritmov

Meno: Martin Pažický
ID: 103086

Teoretické rozdiely medzi algoritmami sa prejavili aj v praxi. Zatiaľ čo BFS hľadá vždy najoptimálnejšie riešenie, aj za cenu väčšej pamäťovej a časovej náročnosti, DFS nájde obvykle neoptimálne riešenie (dalo by sa upraviť aj tak aby hľadalo optimálne riešenie), za kratší čas a s menším počtom vygenerovaných stavov. Tiež možno sledovať, že čas je priamo závislý od počtu vygenerovaných stavov.
Ďalšia zaujímavá vlastnosť programu je, že čas alebo počet stavov nezávisí vždy len od hĺbky optimálneho riešenia – pri vstupe hardInput4 sa vygenerovalo značne viac stavov aj keď má oproti vstupu hardInput5 menšiu hĺbku optimálneho riešenia. Zapríčinené to môže byť rozdielnymi tvarmi stromov (pri hardInput4 sa musí generovať širší strom).

6) Zhodnotenie riešenia

Oba použité algoritmy dokážu nájsť riešenia aj najťažších hlavolamov[1] rádovo v desatinách sekúnd. Vysokú rýchlosť algoritmov som však dosiahol až postupným optimalizovaním pôvodného riešenia. Najzásadnejší vplyv na časy riešenia malo zavedenie hashmapy na kontrolovanie navštívených stavov, ktoré sa vykonáva s časovou zložitosťou O(1), na rozdiel od pôvodného listu, kde táto kontrola bola O(N). Obrovské časové zlepšenie sa prejavilo najmä preto, že kontrola navštívených stavov sa vykonáva vždy pri vygenerovaní novej mapy, čo je pomerne často.
Okrem zrýchlenia programu, som tým dosiahol aj zlepšenie pamäťovej efektivity, keďže som sa úplne zbavil listu navštívených vrcholov a na kontrolu používam už vytvorenú hashmapu, ktorá slúži aj na vystopovanie finálneho stavu k počiatočnému.
Riešenie je tiež možné použiť aj na hlavolamy inej veľkosti ako 6x6 (treba zmeniť konštanty ROWS a COLS v triede Solver.java). Hlavolamy iných veľkostí som však dostatočne neotestoval, keďže to nebol zámer zadania.
Ďalším možným zlepšením rýchlosti riešenia by mohlo byť využitie viacniťovosti.

7) Zdroje

1. Názov: Solving Rush Hour, the Puzzle
   Autor: Michael Fogleman
   Dátum: Júl 2018
   Odkaz: https://www.michaelfogleman.com/rush/
2. Názov: Parking Panic
   Autor: ?
   Dátum: ?
   Odkaz: https://www.agame.com/game/parking-panic
3. Názov: Difference between BFS and DFS
   Autor: MK$075
   Dátum: Júl 2020
   Odkaz: https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/#:~:text=BFS(Breadth%20First%20Search)%20uses,Search)%20uses%20Stack%20data%20structure.&text=BFS%20can%20be%20used%20to,edges%20from%20a%20source%20vertex.