SLOVENSKÁ TECHNICKÁ
UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY
A INFORMAČNÝCH TECHNOLÓGIÍ

Prehľadávanie stavového priestoru

Matúš Krajčovič

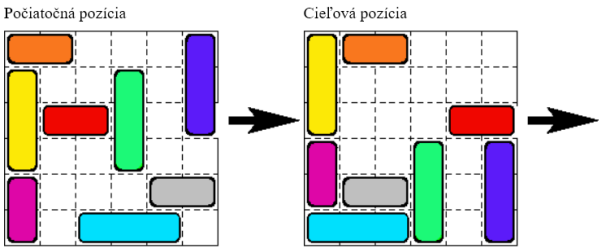
Kód predmetu: **UI_B**
Názov predmetu: **Umelá inteligencia**
Študijný program: Informatika
Študijný odbor: Informatika
Garant predmetu: : **Ing. Lukáš Kohútka, PhD.**
Vyučujúci: Ing. Lukáš Kohútka, PhD.
Cvičiaci: Ing. Ivan Kapustík
Úloha: **Zadanie 2 - problém 1 (Bláznivá križovatka), bod a)**
AIS ID: 103003
Dátum: 27. 10. 2020

Zadanie úlohy

Problém 1: Úlohou je nájsť riešenie hlavolamu Bláznivá križovatka. Hlavolam je reprezentovaný mriežkou, ktorá má rozmery 6 krát 6 políčok a obsahuje niekoľko vozidiel (áut a nákladiakov) rozložených na mriežke 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. Príklad možnej počiatočnej a cieľovej pozície je na obrázku.

bod a): Použite algoritmus prehľadávania do šírky a do hĺbky. Porovnajte ich výsledky.



1 Stručný opis riešenia

Pre jednoduchosť v programe používame globálne premenné pre stack, queue, hash mapu, začiatočný stav, počet áut, rozmery plochy a tiež načítané farby.

Program začína vo funkcii **main()**, kde na začiatku načíta externý súbor, ktorý obsahuje rozmery hracej plochy, počet áut spolu s ich súradnicami, dĺžkou, smerom a farbou - všetko funkciou **load_file()**. Potom skontrolujeme, či je stav validný funkciou **is_valid()**. Pri správnom načítaní hodnôt (do niektorých globálnych premenných) pokračujeme spustením funkcie **search_solution()** pre oba typy vyhľadávania (DFS, BFS), pričom pomocou knižnice <chrono> meriame čas, za aký sa tieto funkcie vykonajú.

Funkcia **search_solution()** vykonáva samotné hľadanie a ako argument berie daný typ (BFS, DFS). Podrobne funkciu opisujeme v časti 3.

Funkcia **get_moves()** dostane ako parameter stav a auto, pričom vracia std::vector všetkých rôznych možností pohybu pre dané auto.

Funkcia **create_state()** dostane ako parameter stav, auto a vzdialenosť. Vráti nový stav, v ktorom je auto o danú vzdialenosť posunuté.

Funkcia **get_str_array()** dostane stav, pričom vráti pole celej hracej plochy s autami a prázdnymi miestami.

Napokon funkcia **is_final_state()** zistí, či je daný stav konečný.

Zvyšné funkcie sú funkcie výpisu a funkcie konvertovania rôznych typov. Všetky dôležité funkcie sú opísané v časti 3. Program sa dá spustiť v command line prostredí. Bez parametrov automaticky načítava zo súboru input.txt a vypisuje do output.txt. Výpis do konzoly sa dá zabezpečiť parametrom std::cout vo funkcii print_solution(). Ak program spustíme s jedným parametrom, otvorí sa namiesto input.txt zadaný vstupný súbor. Pri dvoch parametroch sa otvorí zadaný vstupný a vypisuje sa na zadaný výstupný súbor (druhý parameter, namiesto output.txt).

2 Reprezentácia údajov

2.1 Uzly

Samotný **stav** je reprezentovaný premennou typu std::string, v ňom je každé auto reprezentované štvoricou číslic, napr. 1120, pričom prvé dve čísla sú súradnice na ploche (počnúc číslom 0, pričom v textovom súbore sa začínajú číslom 1). Tretia číslica je dĺžka auta a posledná 0 znamená horizontálny smer (1 pre vertikálny). Prvé auto je vždy to, ktoré sa musí dostať na pravý, resp. dolný okraj (červené).

Toto tiež znamená, že maximálna veľkosť plochy je 10x10 (v stave zápis 99), rovnako tak dĺžka auta je maximálne 9. Počet áut je teda teoreticky neobmedzený (pokiaľ to plocha dovolí).

Ukážkový začiatočný stav na obrázku v zadaní vyjadríme:

```
2120 0020 1031 4021 1331 5230 4420 0531
```

Rodičovský uzol vieme spätne zistiť pomocou hash mapy, do ktorej stavy ukladáme, pričom kľúč je potomok a hodnota je rodič.

Aplikovaný **operátor** zistíme podľa rozdielnej číslice v aktuálnom a rodičovskom stave. Týmto ušetríme nejakú pamäť, no o niečo strácame na výpočtovej zložitosti, no iba pri samotnom výpise, nie hľadaní.

Počet uzlov, resp. počet krokov na vyriešenie úlohy počítame pri spätnom rekonštruovaní riešenia. Tým znova šetríme nejakú pamäť.

Informácie o cene cesty ani o potomkoch neukladáme.

2.2 Strom

Stromovú štruktúru jednotlivých stavov reprezentujem pomocou hash mapy, v ktorej ako kľúč používam stav, ktorého hodnota je jeho rodič. Ako som už spomínal, informácie o potomkovi neukladáme.

2.3 Externé súbory

Využívame externý súbor input.txt (poprípade prvý argument pri spustení programu), v ktorom je začiatočný stav v tvare:

```
6 6                 //rozmery plochy
8                   //pocet aut
3 2 2 h cervene     //samotne auta
1 1 2 h oranzove
2 1 3 v zlte
```

```
5 1 2 v fialove
2 4 3 v zelene
6 3 3 h svetlomodre
5 5 2 h sive
1 6 3 v tmavomodre
```

Veľkosť plochy je max. 10 x 10. Prvé dve čísla sú súradnice auta, počnúc 1. Maximálna možná súradnica je teda 10. Tretie číslo je dĺžka vozidla, maximálne 9. Posledné je písmeno v alebo h pre vertikálny alebo horizontálny smer. Nasleduje farba vozidla.

Výstupný súbor output.txt (prípadne druhý parameter pri spustení programu) bude pri vyriešení úlohy obsahovať všetky informácie vypísané funkciou print_solution().

3 Použitý algoritmus

3.1 search_solution()

Do funkcie **search_solution()** vkladáme argument DFS alebo BFS, teda táto funkcia vykonáva hľadanie do šírky aj do hĺbky (breadth-first search a depth-first search). Funkcia vracia konečný stav, resp. prázdny reťazec pri neúspechu. Na začiatku vždy vložíme začiatočný stav do hash mapy. Zistíme, či náhodou nie je konečný, ak nie, podľa typu hľadania ho vložíme do QUEUE pri BFS, resp. STACK-u pri DFS. Nasledovný cyklus sa vykonáva pokým nenájdeme riešenie (ktoré vrátime) alebo pokým sa nevyprázdni daná dátová štruktúra (QUEUE, STACK). Vždy odstránime stav z vrchu štruktúry, pre každé auto nájdeme všetky možnosti pohybu, pre každú túto možnosť vytvoríme nový stav ktorý vložíme hasp mapy - ak sa v nej už nenachádza. Ak nie, tak ho vložíme aj do štruktúry, pričom skontrolujeme, či náhodou nie je konečný.

Vyhľadávanie do šírky spočíva v použití štruktúry **queue**, ktorá zabezpečí, že po rozvití uzla sa postupne za sebou rozvijú všetkci jeho potomkovia, následne potomkovia týchto potomkov. Pri vyhľadávají do hĺbky sa používa **stack**, resp. LIFO štruktúra. Rozvijú sa všetci potomkovia daného uzla, no pri rozvíjaní samotných potomkov sa tieto "vnúčatá" rozvinú pred ostatnými potomkami.

Algoritmus BFS by teda mal trvať dlhšie, a má predpoklad byť aj priestorovo náročnejší, no nájde vždy optimálne riešenie (s minimálnym počtom krokov). Presne naopak je to pri DFS, pričom tento môže hľadať do nekonečna, ak by sme neoverovali duplcitné stavy (čo ale robíme).

Priestorová aj časová **zložitosť** pre hľadanie do **šírky** je a^b, kde "a" je faktor vetvenia a "b" je hĺbka.

Pri hľadaní do **hĺbky** je síce horný odhad časovej zložitosti rovnaký, no v praxi sa riešenie nájde skôr. Pri priestorovej zložitosti je možné pamätať si iba cestu ku aktuálnemu uzlu (to by bola zložitosť $a * b$), no my si pamätáme všetky uzly, čiže zložitosť je rovnaká.

3.2 get_moves()

Funkcia berie ako argument aktuálny stav a poradové číslo auta. Postupne prehľadáva všetky možnosti jeho pohybu, najskôr v zápornom smere (smer doľava alebo nahor) a potom v kladnom smere (doprava, nadol). Skontroluje, či je daný presun možný, ak áno, pridá toto číslo do výstupného vektora. Ten by teda mohol vyzerať napríklad ako {-2, -1, 1, 2, 3} alebo {-2}.

3.3 create_state()

Funkcia dostane aktuálny stav, poradové číslo auta a vzdialenosť, o ktorú sa posúva (jedna z hodnôt funkcie get_moves()). Podľa auta a vzdialenosti jednoducho zmení jeden znak v stave na novú hodnotu a vráti tento nový stav.

3.4 get_str_array()

Funkcia dostane aktuálny stav, pričom ho preformátuje na inú formu - jeden znak pre každé políčko na ploche. Ak je prázdne, bude na ňom 0, inak je písmeno auta (priradené funkciou car_to_char()). Táto forma stavu je tiež vo forme stringu.

3.5 is_final_state()

Funkcia z aktuálneho stavu zistí, či sa červené (prvé) auto nachádza na pravej, resp. dolnej stene hracej plochy.

3.6 print_solution()

Funkcia v hash mape hľadá rodičov až po začiatok a priebežne ich vkladá do std::vector, odkiaľ ich potom spätne vypisuje. Vypíše aj počet krokov a rôzne ďalšie informácie (dĺžka hľadania, počty stavov, typ hľadania) spolu s jednotlivými operátormi, ktorými sa dopracujeme k riešeniu. Pri použití parametra VERBOSE a operátory vypíšu spolu s aktuálnym stavom. Pri parametri COMPACT sa vypíše iba operátor. Ďalšími parametrami sú daný konečný stav, spôsob výpisu a hľadania a tiež aj čas meraný vo funkcii main().

Pôvodne bola funkcia jednoduchšia, bez osobitého zoznamu, a bola riešená rekurzívne, no pri mapách 10x10 som narážal na stack overflow.

4 Testovanie

4.1 Funkčnosť

Ako prvé som testoval správne fungovanie programu. Do vstupného súboru som zadal invalidné vstupy, ktoré sú priložené v súbore examples.txt. Testoval som, či program rozpozná nepovolenú veľkosť hracej plochy, autá vychádzajúce z hracej plochy a tiež prekrývanie áut. Program úspešne označil dané vstupy ako nesprávne.

Rovnako tak som sa zameral aj na vstupy, pri ktorých neexistuje riešenie. Ak funkcia search nevráti stav, znamená to, že sme preskúmali celý priestor, no riešenie neexistuje.

4.2 BFS vs. DFS

Pri testovaní som sa zameriaval najmä na porovnávanie oboch implementácií, resp. algoritmov. Tabuľka všetkých mojich pokusov aj s priloženými číslami hracích plôch, ktoré sú v súbore examples.txt, sa nachádza na konci dokumentácie. Priložené sú aj grafy.

Hlavné testovanie prebiehalo na plochách o veľkosti 6x6, pričom počet áut sa pohyboval od 8 do 14. Vyskúšal som tiež plochy o veľkosti 8x8 a 10x10, aj s väčšími veľkosťami áut.

4.3 Výstup

Pri ukážkovom vstupe je pri BFS výstup (skrátený) do súboru nasledovný:

```
Search type: BFS
1058 unique states found.
979 explored states.
79 states still in QUEUE.
Duration: 0.0093106 seconds.

Solution: 8 steps.

Initial state.
BB...H
C..E.H
CAAE.H
C..E..
D...GG
D.FFF.

RIGHT(oranzove(B), 1)
.BB..H
C..E.H
CAAE.H
C..E..
D...GG
D.FFF.

UP(zlte(C), 1)
CBB..H
C..E.H
CAAE.H
...E..
D...GG
D.FFF.

...

DOWN(tmavomodre(H), 3)
CBB...
C.....
CAA...
D..E.H
DGGE.H
FFFE.H

RIGHT(cervene(A), 3)
CBB...
C.....
C...AA
D..E.H
DGGE.H
FFFE.H
```

Pri nevalidnom vstupe sa chybová hláška vypíše iba do konzoly.

5 Zhodnotenie

5.1 Testovanie BVS a DFS

Z výsledkov uvedených v tabuľke možno usúdiť, že pri využití BFS je cesta optimálna, čiže najkratšia možná. V kontraste, pri DFS má cesta k optimálnosti veľmi ďaleko, podľa grafu vidíme relatívne pomery. Čo ale zvýhodňuje DFS oproti BFS je čas, za ktorý dané riešenie dokázal nájsť. Priemerne je to 5 až 10 krát rýchlejšie, aj keď výsledná cesta je často mnohonásobne dlhšia.

Vidíme to tiež z množstva stavov, pri BFS je ich oveľa viac, pričom väčšina z nich je aj preskúmaná. Pri DFS sa nájde menej stavov, a iba časť z nich sa aj preskúma, pričom táto hodnota je často veľmi blízka dĺžke nájdenej cesty. Slpec stavy obsahuje počet stavov, ktoré sme objavili, druhý stĺpec je počet stavov, ktoré sme aj skontrolovali (prípade rozvili).

Pri testovaní plôch s veľkosťou 10x10 som pri väčšom počte áut narážal na problémy s pamäťou (a aj časom). Pri ploche 17 som musel odobrať niektoré autá, lebo program padal na chybe nepodarenej alokácie pamäte, hlavne pri BFS, kde sa pamäť programu pohybovala až okolo 2 GB. V tabuľke možno vidieť, koľko stavov sme pri skúmaní objavili, a koľko z nich sme pred ukončením hľadania stihli preskúmať.

5.2 Rozšírenia

Program by sa inou implementáciou stavov (napr. pomocou samostatnej triedy) dal rozšíriť, aby dokázal spracovať plochy väčšie ako 10x10, tým pádom aj autá s dĺžkou väčšou ako 9. Ďalším rozšírením by mohlo byť iba čiastočné pamätanie si uzlov pri DFS - zahadzovanie už prehľadaných vetiev, čím by sme ušetrili pamäť.

5.3 Implementácia

Keďže obe implementácie môžu byť časovo aj priestorovo náročné, zvolil som programovací jazyk C++, ktorý by si s danými úlohami mal poradiť rýchlejšie, ako napríklad Python.

č.	velk.	autá	BFS				DFS			
			kroky	čas [s]	stavy	preskúmané	kroky	čas [s]	stavy	preskúmané
1	6x6	8	8	0,0066154	1058	979	158	0,0011136	770	159
2	6x6	10	10	0,0580821	7171	6906	1202	0,0130780	6722	1342
3	6x6	10	13	0,0156889	2009	1631	134	0,0014852	677	137
4	6x6	10	32	0,0074177	622	602	1125	0,0015217	436	184
5	6x6	13	37	0,0540536	6150	5821	854	0,0097734	4657	1022
6	6x6	11	36	0,0145025	2464	2177	431	0,0045886	2327	796
7	6x6	12	34	0,0183867	2334	2110	411	0,0052779	1742	426
8	6x6	12	35	0,0107445	1291	1244	256	0,0025025	965	345
9	6x6	11	21	0,0182977	2160	1816	223	0,0018112	970	232
10	6x6	8	15	0,0121817	1643	1398	160	0,0014070	702	179
11	6x6	8	17	0,0106090	1290	1142	268	0,0025488	1007	302
12	6x6	8	15	0,0049446	536	517	96	0,0011201	372	99
13	6x6	14	26	0,0259787	2672	2637	456	0,0087921	2479	988
14	6x6	10	21	0,0549807	5977	5913	736	0,0076106	3845	838
15	6x6	11	23	0,0395741	4587	4354	836	0,0090988	3661	912
16	10x10	11	10	57,4961000	4760488	2571879	28917	0,773132	403408	28918
17	10x10	12	12	153,528	7560956	5789075	137965	4,83484	2414259	137965
18	10x10	12	5	0,996093	219398	32047	23214	0,855517	478805	23214
19	10x10	12	8	26,2041	3100528	1200294	229731	6,66323	3919405	229731
20	8x8	12	8	5,8066	849111	302782	9328	0,184453	158073	9328
21	8x8	13	10	11,8742	1515178	774440	36289	0,632282	457884	36289
22	8x8	15	13	84,4863	6800199	4770294	89933	1,88947	1256017	89940