Slovenská technická univerzita

Fakulta informatiky a informačných technológii

Ilkovičova 3, 812 19 Bratislava

Umelá inteligencia

Zadanie č. 2

Peter Berta

Cvičiaci: Ing. Ivan Kapustík

Študijný odbor: Informatika

Ročník: 2. Bc

Akademický rok: 2015/2016

1. Riešený problém

Úlohou je nájsť riešenie 8-hlavolamu. Hlavolam je zložený z ôsmich očíslovaných políčok a jedného prázdneho miesta. Očíslované políčka je možné posúvať hore, dole, vľavo a vpravo len vtedy, ak je tým smerom medzera. Sú zadané dve konkrétne rozpoloženia všetkých políčok. Tieto rozpoloženia predstavujú začiatočnú a cieľovú pozíciu. Cieľom je dostať sa zo začiatočného stavu do cieľového stavu, a pritom urobiť čo najmenej ťahov.

Na riešenie tejto úlohy použijeme lačný algoritmus. Na konci porovnajte výsledky a efektivitu dvoch rôznych heuristík.

2. Opis riešenia

Pri riešení tohto problému použijeme lačný algoritmus, ktorý sa zakladá na prehľadávaní vygenerovaných stavov podľa toho, ako blízko sa zdajú byť k cieľovému stavu. Lačný algoritmus prehľadáva najprv tie stavy, ktoré sa zdajú byť bližšie k cieľovému stavu, ako tie ostatné stavy. Na vypočítanie tejto hodnoty sme použili heuristickú funkciu. [1][2][3]

Prvýkrát použijeme heuristickú funkciu, ktorá spočíta počet políčok, ktoré nie sú na správnom mieste. Prázdne miesto medzi tieto políčka nepočítame. Čím je toto číslo väčšie, tým je daný stav viac vzdialený od cieľového stavu a musíme vykonať viac pohybov, aby sme sa do cieľového stavu dostali. Stav, ktorého hodnota heuristickej funkcie je nula, predstavuje cieľový stav.

Druhýkrát použijeme heuristickú funkciu, ktorá spočíta vzdialenosti políčok od svojich správnych miest v cieľovom stave. Prázdne miesto medzi tieto políčka nepočítame. Rovnako, ako pri prvej heuristickej funkcii, aj v tomto prípade sú žiadané nízke hodnoty heuristickej funkcie daného stavu.

3. Reprezentácia údajov problému, použitý algoritmus

Generované stavy sú ukladané do spájaného zoznamu, ktorý sa udržiava zoradený podľa heuristickej funkcie daného stavu. Nové stavy sa generujú vždy, keď sa stav z najnižšou hodnotou heuristickej funkcie vyberie z tohto zoznamu. V prípade, že majú dva stavy túto hodnotu rovnakú, práve vygenerovaný stav sa pridá až za stav, ktorý sa už v zozname nachádza.

```
struct list {
	struct node *node;
	struct list *next;
};
```

Jednotlivé stavy sú ukladané v štruktúre, ktorá okrem daného stavu obsahuje aj hodnotu heuristickej funkcie tohto stavu, hĺbku, v ktorej sa tento stav nachádza, posledný pohyb, ktorým sme sa do tohto stavu dostali a odkaz na posledný stav. Pomocou tohto odkazu jednoducho nájdeme všetky stavy od začiatočného až po cieľový, vrátane použitých operátorov. Operátory sú definované číslom, teda číslo jedna znamená pohyb hore, číslo dva pohyb dole, číslo tri pohyb vľavo a číslo štyri pohyb vpravo.

```
struct node {
	char *state;
	int heur, depth, move;
	struct node *last;
};
```

Počas hľadania správneho riešenia a generovania nových stavov sa môže stať, že vygenerujeme stav, ktorý sme už v minulosti vygenerovali. Takáto situácia je nežiadúca. Na zamedzenie generovania už v minulosti vygenerovaných stavov použijeme hashovaciu funkciu.

```
struct tableNode *table[100000];
```

(pseudokód hashovacej funkcie)

```
Hash(state) = ((int(state) / 100000) + (int(state) % 100000)) % 100000;
```

Jednotlivé prvky v tabuľke sú reprezentované štruktúrou, ktorá okrem daného stavu obsahuje aj odkaz na nasledujúci prvok rovnakej štruktúry v prípade, že je miesto v tabuľke už obsadené. Vznikajú tu teda spájané zoznamy.

```
struct tableNode {
	char *state;
	struct tableNode *next;
};
```

4. Spôsob testovania a výsledky experimentov

Na testovanie správnosti riešenia a funkčnosti algoritmu som používal rôzne funkcie.

Funkcia `print_list(list);` vypíše všetky stavy, ktoré boli doposiaľ vygenerované, a zároveň ešte neboli prehľadané.

```
void print_list(struct list *list){..};
```

Funkciu `print_state(state);` som používal na vypísanie niektorého stavu.

```
void print_state(char *state){..};
```

Na kontrolovanie, či algoritmus vôbec nájde správne riešenie, som používal taký vstup, ktorý mal určite jednoduché správne riešenie. Nepoužíval som žiaden externý kód, či program.

5. Zhodnotenie riešenia

Lačný algoritmus často nenájde najrýchlejšie, ani najefektívnejšie riešenie. Ak ale riešenie existuje, je veľmi pravdepodobné, že mu nebude trvať príliš dlho, kým ho nájde. Mierny problém nastáva vtedy, keď riešenie neexistuje. V tom prípade algoritmus musí prejsť všetkými stavmi, aby mohol zhodnotiť, že riešenie neexistuje. Kód by mohol byť upravený tak, aby sa v určitom momente rozhodol, že k nájdeniu riešenia sa neblížime. Príliš dlhý čas behu programu, alebo vysoké hodnoty heuristickej funkcie by mohli byť použité pri tomto rozhodovaní.

Do budúcnosti by bolo možné zjednodušiť program tým, že zlúčime štruktúru zoznamu generovaných stavov a štruktúru jednotlivých stavov. Takáto zmena by znížila počet riadkov kódu a znížila pamäťové nároky. Ďalej by bolo možné zefektívniť funkčnosť hash tabuľky. Rýchlejšia, či lepšia hashovacia funkcia a efektívnejšie reťazenie prvkov v tabuľke by mohli znížiť čas behu programu.

Pri implementácii tohto programu som použil programovací jazyk C. Nevýhodou použitia tohto jazyka je hlavne potreba implementácie vlastnej hash tabuľky a vlastného spájaného zoznamu. Za výhodu použitia programovacieho jazyka C považujem jeho jednoduchosť a prehľadnosť.

6. Používateľská príručka

Program je ovládaný pomocou konzoly. Najprv je potrebné zadať začiatočný stav, a potom konečný stav. Príklad vzorového vstupu:

((1 2 3)(4 5 6)(7 8 0))

((3 2 1)(7 5 6)(4 8 0))

‘0‘ predstavuje voľné miesto.

Premenná ‘typZobrazenia’ modifikuje formu výpisu správneho riešenia, ak je nájdené. Pre dlhé zobrazenie nastavte hodnotu tejto premennej na 1. Pre jednoduché zobrazenie nastavte hodnotu tejto funkcie na 2.

7. Vzorové riešenie

Input:

```
((1 2 3)(4 5 6)(7 8 0))

((1 2 3)(4 8 5)(7 0 6))
```

Output:

```
Zaciatocny stav:
 1 | 2 | 3
 4 | 5 | 6
 7 | 8 | 0

Konecny stav:
 1 | 2 | 3
 4 | 8 | 5
 7 | 0 | 6

Zaciname hladanie s pouzitim PRVEJ heuristickej funkcie...
START  1 2 3 4 5 6 7 8 0
DOLE   1 2 3 4 5 0 7 8 6
VLAVO  1 2 3 4 0 5 7 8 6
HORE   1 2 3 4 8 5 7 0 6
```

```
Dlzka hladania riesenia bola 0 sec a 0 msec
Konecny stav najdeny v hlbke 3...
Bolo generovanych 13 stavov a z toho pouzitych 4 stavov...
Zaciname hladanie s pouzitim DRUHEJ heuristickej funkcie...
START  1 2 3 4 5 6 7 8 0
DOLE   1 2 3 4 5 0 7 8 6
VLAVO  1 2 3 4 0 5 7 8 6
HORE   1 2 3 4 8 5 7 0 6
Dlzka hladania riesenia bola 0 sec a 0 msec
Konecny stav najdeny v hlbke 3...
Bolo generovanych 10 stavov a z toho pouzitych 3 stavov...

Rychlejsi bol algoritmus, ktory pouzival heuristicku funkciu c. 2...
```

Zdroje:

[1] Black, Paul E. (2. 2. 2005). Dictionary of Algorithms and Data Structures. U.S. National Institute of Standards and Technology (NIST).

[2] Audiopedia (31. 10. 2014). Greedy algorithm. https://www.youtube.com/watch?v=A8CEvPmNpKQ

[3] prof. Ing. Pavol Návrat, PhD. 2016. Prednášky z predmetu Umelá inteligencia.