Lovac na blago

1 Zadatak

Lovac na blago metalnim je detektorom pronašao novčiće. Nije ih imao vremena iskopati, pa je napravio kartu nalazišta u nadi da je, čak iako je izgubi, nitko neće razumjeti. U poljima koja sadrže brojeve nema novčića, ali ti brojevi ukazuju na to koliko se novčića nalazi u 8 polja koja ih okružuju, U polju se nalazi točno jedan novčić. Otkrijte točan broj i položaj novčića metodama pretraživanja u dubinu i širinu i algoritmom A* s proizoljno odbaranom heuristikom. Usporedite složenost upotrijebljenih metoda.

2 Pokretanje programa

Program je napisan u c++ jeziku.
Sastoji se od datoteka *main.cpp*, *funkcije.h*, *funkcije.cpp*.
Za pokretanje programa, potrebno je staviti sve tri datoteke u isti projekt, kompajlirati i pokrenuti *main.cpp*. Potrebno je biti na operacijskom sustavu windows.

3 Unos podataka

Program korisnika prvo upita želi li isprobati testni primjer (a), testni primjer (b) ili samostalno upisati lovčevu kartu. U slučaju da korisnik odabere jedan od testnih primjera, program vraća *pair<int,int>* gdje se nalazi broj redaka i stupaca u karti te strukturu *vector<pair<pair<int,int>,int> >* gdje se nalazi vektor popunjen parovima koordinati i brojevima koji se na njima nalaze. U slučaju da korisnik odabere samostalni upis podataka, program prvo upita korisnika koliko redaka i stupaca njegova karta sadrži, te nakon toga traži upis koordinati i brojeva koji se na njima nalaze.

Primjer upisa:
1 2 3 (govori programu da se na poziciji (1,2) nalazi broj 3)
0 0 0 (govori programu da se na poziciji (0,0) nalazi broj 0)
-1 (-1 označava kraj upisa)

Nakon toga, program korisnika upita želi li vizalizaciju algoritma, te koji od 3 algoritma za pronalazak zlata namjerava koristiti.

4 Obrada podataka

1. Program poziva funkciju:
vector<vector<char> > get_mat (pair<int,int> rc,
vector<pair<pair<int,int>, int> > codes),
koja prima par *rc* koji sadrži broj redaka i stupaca, te vektor *codes* koji sadži zapis koordinati sa brojevima i brojeve koji se na njima nalaze.
Funkcija ide iteratorom po dobivenom vektoru, te vadi koordinate i brojeve te stavlja brojeve u matricu oblika *vector<vector<char> >* na zadane pozicije. Svaka pozicija koja nema šifru sadrži točku. Funkcija vraća takvu matricu.

2. Program poziva funkciju:
vector<vector<char> > sort_mat(vector<vector<char> > matrix, pair<int,int> rc),
koja prima popunjenu matricu *matrix* i *rc*, te stavlja '*' na mjesta na kojima zlato ne može biti. Funkcija radi na način da za svako polje provjerava nalazi li se u blizini '0'. Ako se nalazi (i to polje ima '.'), stavlja '*'. Također, ako se polje ne nalazi u blizini nijednoga broja, na njega stavlja '*'. Funckija nakon toga mjenja sva polja s '0' u '*'. Konačan produkt je matrica koja ima brojeve nejednake '0', i točke oko njih, koje označavaju polja u kojima postoji mogučnost za pronalaskom zlata. Funkcija vraća takvu matricu u obliku *vector<vector<char> >*.

Nakon toga, program je spreman za kretanje s algoritmom kojeg je korisnik odabrao.

5 Pretraživanje u širinu (BFS)

Program poziva funkciju:
vector<vector<char> > bfs(vector<vector<char> > matrix_sorted,
pair<int, int> rc,
int koraci_o,*
int koraci_p,*
int visualize).

matrix_sorted - predstavlja početnu matricu nad kojom izvršavamo algoritam
rc - par s brojem redaka i stupaca u matrici
koraci_o - varijabla koja pamti broj otvorenih matrica, tojest onih koje su izašle iz liste
koraci_p - varijabla koja pamti broj spremljenih matrica, tojest onih koje su pohranjene u listu
visualize - 1 ili 2, govori programu treba li usred odvijanja ispisivati matrice koje koristi u postupku

Rad BFS-a:

```
open <− [matrix_sorted]
while ( !open.empty() )
    matrix <− open.front()
    open.pop()
    koraci_o <− koraci_o + 1

    if ( odabrana vizualizacija ) Nacrtaj matricu
    if ( matrica je rješenje ) return matrica

    prvi_broj <− Nađi prvi broj u matrici
    parovi <− Koordinate svih točaka oko prvi_broj
    for every ( par ∈ parovi )
        nova_matrica <− Stavi zlato na matrix[par.i][par.j]
        if ( Postoji broj 'i' oko kojeg ima (< i) točaka ) continue
        koraci_p <− koraci_p + 1
        open.insert ( nova_matrica )
    end for
end while
return fail
```

Vizualizacija:

1. Poziva se funkcija *system("cls");* koja briše prethodni tekst u terminalu.
2. Poziva se funkcija *print_mat* koja ispisuje matricu. Zlata su označena slovom 'G' (Gold), moguće pozicije za zlato su označene s '.', na pozicijama na kojima zlato ne može biti; ostaje praznina.
3. Ispisuje se broj trenutnog koraka algoritma.
4. Program spava na 500ms.

Provjera rješenja:

Pozivaju se funkcije:
int dots_number (vector<vector<char> > matrix), pair<int,int> rc),
int prebroji (vector<vector<char> > matrix),
koje redom vraćaju preostali broj točaka u matrici i zbroj preostalih brojeva u matrici. Ako su oba jednaka 0, onda je tražena matrica rješenje.

Nalaženje prvog broja u matrici:

Prvi broj u matrici traži se prolaskom matrice po retcima. Dakle, u matrici je prvi broj onaj koji ima najamnji indeks retka, ili stupca u slučaju da više brojeva ima isti indeks retka.

Stavljanje zlata na poziciju (i,j):

Poziva se funkcija:
vector<vector<char> > place_gold(vector<vector<char> > matrix,
int i, int j,
pair<int, int> rc),
koja mijenja '.' na poziciji (i,j) u zlato ('G'). Funkcija tada smanjuje sve brojeve oko pozicije (i,j) za 1. U slučaju da se neki broj smanji s '1' na '0', poziva se funkcija *sort_mat*.

Napomena: Radi bržega izvođenja, BFS algoritam se grana po svim mogućim pozicijama za zlato oko prvog pronađenog broja u matrici, a ne po svim mogućim pozicijama za zlato u matrici.

Primjer jedne iteracije algoritma:

queue:

*	*	G	.
.	.	.	2
.	1	.	.
.	.	.	*

*	*	.	G
.	.	.	2
.	1	.	.
.	.	.	*

Prva matrica se vadi iz queue-a i obrađuje.
Stavljaju se zlata oko prvoga broja (2) te se novonastale matrice stavljaju u queue.

queue:

*	*	.	G
.	.	.	2
.	1	.	.
.	.	.	*

*	*	G	G
.	.	.	1
.	1	.	.
.	.	.	*

*	*	G	.
*	*	G	1
*	*	*	.
*	*	*	*

*	*	G	.
*	*	*	1
*	*	G	.
*	*	*	*

*	*	G	.
.	.	.	1
.	1	.	G
.	.	.	*

6 Pretraživanje u dubinu (DFS)

Program poziva funkciju:
vector<vector<char> > dfs(vector<vector<char> > matrix_sorted,
pair<int, int> rc,
int koraci_o,*
int koraci_p,*
int visualize).

s jednakim opisom parametara kao i u BFS. Funkcija *dfs* radi na potpuno isti način kao i funkcija *bfs*, uz jedinu razliku u tome što se koristi struktura *stack* umjesto strukture *queue* za "open" iz pseudokoda.

Primjer jedne iteracije algoritma:

stack:

*	*	G	.
.	.	.	2
.	1	.	.
.	.	.	*

*	*	.	G
.	.	.	2
.	1	.	.
.	.	.	*

Prva matrica se vadi iz stack-a i obrađuje.
Stavljaju se zlata oko prvoga broja (2) te se novonastale matrice stavljaju u stack.

stack:

*	*	G	G
.	.	.	1
.	1	.	.
.	.	.	*

*	*	G	.
*	*	G	1
*	*	*	.
*	*	*	*

*	*	G	.
*	*	*	1
*	*	G	.
*	*	*	*

*	*	G	.
.	.	.	1
.	1	.	G
.	.	.	*

*	*	.	G
.	.	.	2
.	1	.	.
.	.	.	*

7 Stablo pretraživanja

- Svaki dozvoljeni čvor u stablu pretraživanja širi se na maksimalno osmero djece. (Zbog maksimalnih 8 slobodnih pozicija oko prvoga broja u matrici čvora.)
- Budući da su čvorovi prošireni funkcijom *place_gold*, tojest metodom stavljanja zlata, nakon što na pojedinu poziciju stavimo zlato, u toj grani ga više ne možemo maknuti. To osigurava da se nijedno stanje ne može pojaviti više puta u pojedinom prolasku od vrha do dna stabla pretraživanja.
- Maksimalna dubina stabla jednaka je sumi svih brojeva u matrici.
- Put do rješenja uvijek je jednako dugačak i duljina mu je jednaka broju zlata u matrici.

-Zbog navedenih tvrdnji, svaki od naših algoritama za pretraživanje je potpun.
-Zbog jedinstvenosti rješenja, svaki od naših algoritama za pretraživanje je optimalan.

Iz gore navedenog možemo zaključiti da A* algoritam nema prednosti nad greedy best-first search algoritmom uz set *closed* posjećenih stanja.

8 Greedy best-first search

Program poziva funkciju:
vector<vector<char> > greedy(vector<vector<char> > matrix_sorted,
pair<int, int> rc,
int koraci_o,*
int koraci_p,*
int visualize).

s jednakim opisom parametara kao i u BFS i DFS.

Koristi se heuristika $h(S)$ =*zbroj svih brojeva u matrici stanja S*

Uz takvu heuristiku, GreedyBFS algoritam će prioritizirati stavljanje zlata na pozicije kojima može najviše smanjiti sumu brojeva u matrici.

Algoritam koristi strukturu *priority_queue* u koju šalje:
$par < int, vector < vector < char >>>$,
gdje prvi element u paru (int) označava sumu svih brojeva u matrici ($h(S)$), a drugi element ($vector < vector < char >>$) označava matricu. Priority_queue prioritizira elemente kojima je $h(S)$ minimalan.

Rad greedy best-first search-a:

```
open <− [par(zbroj_polja, matrix_sorted)]
closed <− [ ]
while ( !open.empty() )
    matrix <− open.front().second
    open.pop()
    closed.insert ( matrix )
    koraci_o <− koraci_o + 1

    if ( odabrana vizualizacija ) Nacrtaj matricu
    if ( matrica je rješenje ) return matrica

    prvi_broj <− Nađi prvi broj u matrici
    parovi <− Koordinate svih točaka oko prvi_broj
    for every ( par ∈ parovi )
        nova_matrica <− Stavi zlato na matrix[par.i][par.j]
        if ( Postoji broj 'i' oko kojeg ima (< i) točaka ) continue
        if ( nova_matrica ∈ closed ) continue
        koraci_p <− koraci_p + 1
        open.insert ( par(zbroj_polja, nova_matrica) )
    end for
end while
return fail
```

Primjer jedne iteracije algoritma:

priority_queue:

*	*	G	.
.	.	.	2
.	1	.	.
.	.	.	*

*	*	.	G
.	.	.	2
.	1	.	.
.	.	.	*

Prva matrica se vadi iz stack-a i obrađuje.
Stavljaju se zlata oko prvoga broja (2) te se novonastale matrice s njihovim vrijednostima h(S) stavljaju u priority_queue.

priority_queue:

*	*	G	.
*	*	G	1
*	*	*	.
*	*	*	*

*	*	G	.
*	*	*	1
*	*	G	.
*	*	*	*

*	*	G	G
.	.	.	1
.	1	.	.
.	.	.	*

*	*	G	.
.	.	.	1
.	1	.	G
.	.	.	*

*	*	.	G
.	.	.	2
.	1	.	.
.	.	.	*

Brzina algoritama na testnim primjerima:

	testni primjer (a)		testni primjer (b)	
	Broj koraka	Vrijeme izvršenja	Broj koraka	Vrijeme izvršenja
BFS	16339	∼80sek	1560	∼5sek
DFS	3272	∼15sek	55	/
GreedyBFS	146	/	283	/

- Budući da se stablo pretraživanja odsjeca na dubini većoj od tražene, DFS i greedyBFS će samo u najgorem slućaju biti jednako brzi kao i BFS, dok će u svim ostalim slućajevima biti znatno brži.

- Brzina greedyBFS-a u odnosu na DFS ovisi o zadanoj matrici i ne može se osigurati prednost jednog algoritma nad drugim u svakome slučaju, dok će u prosječnom slučaju greedyBFS biti znatno brži zahvaljujući polju *closed* i načinu na kojem vrši pretragu.

Gornja ograda za broj iteracija algoritama:

$$\prod_{t \in T} t!$$

gdje je T skup svih brojeva u matrici.

Napomena:

Ukoliko je odabrana vizualizacija, programu će zbog učestalih pauzi (kako bi korisniku prograrma bilo moguće predočiti postupak) trebati znatno duže da dođe do rješenja.