Lovac na blago

Luka Sesartić

PMF Zagreb, Fizički odsjek
lsesartic.phy@pmf.hr

December 23, 2022

I. Uvod

Ovaj dokument namijenjen je da se koristi uz priloženu Jupyter bilježnicu. Napisani kod je relativno modularan i moguće je mijenjati ulazne podatke kako bi se primijenio na rješavanje drugih sličnih problema.

i. 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 proizvoljno odabranom heuristikom. Usporedite složenost upotrijebljenih metoda.

						2		2	
	0	1			3	4		3	
	0		1					4	
				1					2
2							3	2	
		4				2	3		
2			0						
2			2	1		3	4	4	
							3		
	2	1	2			3		2	

Figure 1: *Postava zadatka*

					●	2		2	
	0	1	●		3	4	●	3	●
	0		1		●	●		4	●
				1		●		●	2
2	●	●					3	2	
	●	4				2	3	●	
2	●		0			●	●		●
2			2	1		3	4	4	●
●		●		●		●	3	●	
	2	1	2		●	3	●	2	

Figure 2: *Riješene zadatka*

ii. Interpretacija zadatka

Zadatak od nas zahtjeva da sa danom mapom sa *slike* 1 generiramo mapu prikazanu u *slici* 2. Ovaj zadatak može se interpretirati kao otvoreni minesweeper te mi samo moramo naći lokacije svih mina (u našem slučaju zlatnika). Zadatak nam za to određuje da moramo ko-

ristiti: Depth First Search (DFS), Breath First Search (BFS) i A* algoritam.

iii. Problem zadanog zadatka

Svi navedeni algoritmi su algoritmi na grafovima, DFS i BFS su algoritmi pretraživanja grafa dok je A* algoritam za pronalazak najkraćeg puta između dva čvora grafa. Ako koristimo ove algoritme ne možemo dobiti traženo rješenje pošto za rješavanje našeg problema je potrebno koristi neka logička pravila asocirana uz igru minesweeper koja nisu dio BFS/DFS/A* algoritama, a ujedno su i neprimjenjiva jer se ona oslanjaju na poznati broj mina/zlatnika koji je nama u ovako danom zadatku nepoznat. Za ovako zadan zadatak bi algoritamski pristup backtracking-a bio primjenjiv, algoritam bi stavljao jednu po jednu minu/-zlatnik dok ne bi popunio sve moguće lokacije mina/zlatnika i time došao do konačnog broja i lokacije svih mina/zlatnika.

iv. Prijedlog reinterpretacije zadatka

Dani zadatak možemo shvatit kao da naš lovac (možemo mu i dati ime npr. Joško) uistinu ide kopati zlatnike po polju, te sada problem postaje kako minimalizirati broj kopanja s danom mapom. Pošto nam je i dalje nepoznat ukupan broj zlatnika, moramo pretražiti svako moguće polje kako bi bili sigurni da nema još neiskopanih zlatnika.

Ideja ovakve reinterpretacije problema bi bila pokazati različiti pristup ophodnji grafa koristeći BFS i DFS te uvesti A* kao alternativu u kojoj dobivamo opciju žrtvovanja točnosti za brzinu.

v. Strategija rješavanja

Prva dva algoritamska pristupa DFS i BFS koristiti ćemo standardno ignorirajući brojeve na danoj mapi i tretirati ih kao polja koja ne možemo kopati, te ćemo proći kroz cijelu mapu na dva različita načina (BFS/DFS) i prebrojiti broj kopanja potreban da pronađemo lokacije svih zlatnika te iz toga generirati mapu svih zlatnika.

Sami A* algoritam nije primjenjiv u ovom zadatku pošto on služi za pronalazak najkraćeg puta između dva čvora u grafu. Čak i ako bi mu dali naš zadani graf te označili polje $(1,1)$ kao početno i polje (n,n) kao završno nemamo neku prigodnu heuristiku koja će nam garantirati da prođemo kroz sva bitna polja matrice. Iako nismo u mogućnosti primijeniti sam A* algoritam i tako pokazati naše razumijevanje samoga možemo konstruirati naš vlastiti algoritam koji će koristiti iste ideje kao i A* algoritam te nam dopustiti da žrtvujemo točnost za brzinu samog algoritam.

II. Algoritmi

i. Depth First Search

U ovom algoritamskom pristupu ćemo kako smo i spomenuli prije koristiti DFS da pretražimo sva moguća polja u matrici i tako pronađemo sve zlatnike.

DFS radi koristeći podatkovnu strukturu stack, koja radi na principu LIFO odnosno "last in, first out". Stack u python-u ćemo kao listu unutar koje ćemo spremati liste uređenih parova koje će predstavljati polja koja algoritam mora posjetiti sljedeća. Implementacija je izvedena na sljedeći način:

```
stack = [[1,2], [2,4], [5,6]...]
```

Bitne operacije na stack-u su nam *pop* i *append*, *pop* koristimo da dobijemo zadnji dodani element a *append* koristimo da stavimo novi uređeni par u stack. Implementacija je izvedena na sljedeći način:

```
i,j = stack.pop()
stack.append([i,j])
```

DFS će prvo dobiti početno polje koje smo izabrali da bude (0,0) te će sve moguće susjede (susjedi su polja koja graniče s trenutnim poljem) staviti u stack i označiti ih kao posječena, nakon toga će uzeti sljedeći element iz stack-a i ponoviti proceduru. Prilikom svakog otvaranja polja algoritam će testirati jeli dano polje ima

zlatnik u sebi. Pseudo kod za DFS izgleda ovako:

```
->inicijalizacija polja posjecenih
->inicijalizacija polja zlatnika
->while stack not empty:
    pop stack
    if polje ima zlatnik:
        zapisi lokaciju
        ukupni broj zlatnika++
    dodaj susjede na stack
    oznaci susjede kao posjecene
    ukupni broj kopanja++
```

Možemo i grafički prikazati način obilaženja matrice koristeći DFS, plava polja označavaju polje na kojem se algoritam trenutno nalazi, narančasta polja su polja koja se u tom koraku dodaju na stack (polja se dodaju na stack u smjeru obrnutom kazaljke na satu počevši od polja iznad onoga na kojem se algoritam trenutno nalazi), a siva polja su ona koja je algoritam već posjetio. Grafički prikaz obilaska matrice koristeći DFS je prikazan na slici 3.

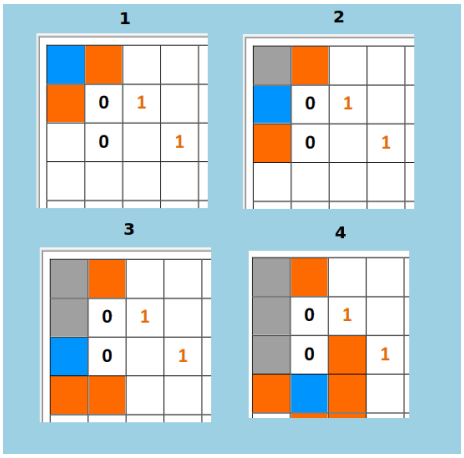

Figure 3: *Grafički prikaz obilaska matrice koristeći DFS*

Implementirani kod može se vidjeti u Jupyter bilježnici. Output tog možemo vidjeti u slici 4.

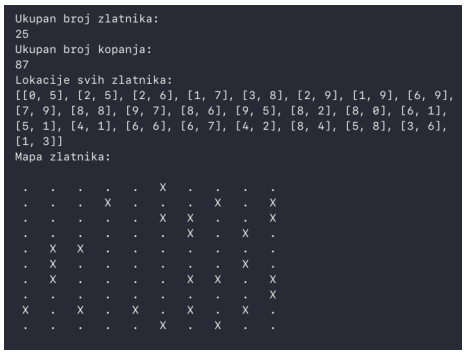

Figure 4: *Output DFS algoritma*

Iz outputa vidimo da je algoritam pronašao 25 zlatnika što je točan broj, da je naš lovac Joško izvršio 87 kopanja, ujedno smo dobili i poziciju svakog zlatnika u grafičkom obliku i u obliku polja uređenih parova. Bitan dio outputa je broj kopanja to u našem slučaju možemo gledati kao ocjenu algoritma odnosno njegovu složenost, naš cilj je dakako da Joško mora što manje puta kopati jer će tada biti najbrži.

ii. Breath First Search

BFS pristup vrlo je sličan gore opisanom DFS pristupu samo umjesto stack-a koristimo queue. Queue je podatkovna struktura tipa FIFO odnosno "first in, first out". Nju isto implementiramo pomoću liste te u tu listu opet spremamo uređene parove koji nam označavaju koordinate određenih polja. Operacije koje su nam potrebne za queue su opet *pop* i *append* samo ovaj put *pop* uzima prvi element queue-a (zadnji stavljeni), a *append* radi isto kao i u stack-u tako da dodaje element na kraj liste. Implementacija queue-a je vrlo slična stack-u kao što je vidljivo iz koda:

```
queue = [[4,6], [1,6], [7,7]...]
```

Dok su *pop* i *append* implementirani na sljedeći način:

```
i,j = queue.pop(0)
queue.append([i,j])
```

BFS će prvo dobiti početno polje (isto kao i DFS) koje smo izabrali da bude (0,0) te će sve

moguće susjede (susjedi su polja koja graniče s trenutnim poljem) staviti u queue i označiti ih kao posječena, nakon toga će uzeti sljedeći element iz queue-a i ponoviti proceduru. Prilikom svakog otvaranja polja algoritam će testirati jeli dano polje ima zlatnik u sebi. Pseudo kod za BFS izgleda ovako:

```
->inicijalizacija polja posjecenih
->inicijalizacija polja zlatnika
->while queue not empty:
    pop queue
    if polje ima zlatnik:
        zapisi lokaciju
        ukupni broj zlatnika++
    dodaj susjede na queue
    oznaci susjede kao posjecene
    ukupni broj kopanja++
```

Punu implementaciju koda možete potražiti u pridruženoj Jupyter bilježnici. Nadalje trebamo prokomentirati put koji će BFS koristiti, pošto koristi queue onda će uvijek prvo istražiti sve susjede pa po redu krenuti dalje na susjede tih susjeda. Grafički prikaz pretraživanja može se vidjeti na slici 5. gdje je trenutno polje označeno plavom bojom, polja koja se u tom prolasku algoritam dodaju u queue su

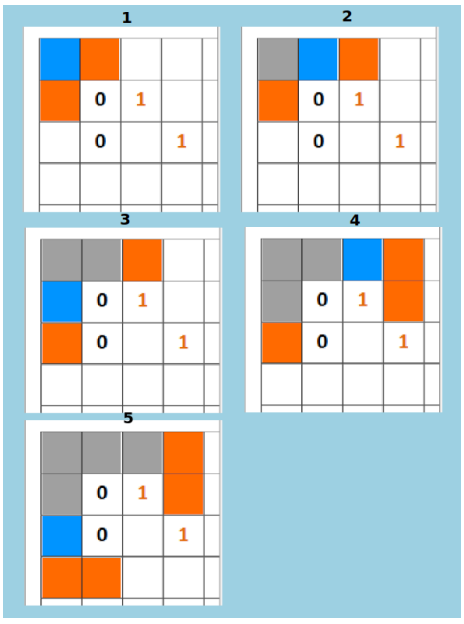

Figure 5: *Grafički prikaz obilaska matrice koristeći BFS*

označena narančastom bojom a polja koja su posječena su označena sivom bojom. Sada možemo preći na output BFS pretraživanja koji se vidi na slici 6.

```
Ukupan broj zlatnika:
25
Ukupan broj kopanja:
87
Lokacije svih zlatnika:
[[1, 3], [4, 1], [4, 2], [5, 1], [2, 5], [0, 5], [6, 1], [3, 6],
[2, 6], [1, 7], [8, 0], [8, 2], [8, 4], [8, 6], [6, 6], [3, 8],
[9, 5], [9, 7], [6, 7], [2, 9], [8, 8], [5, 8], [1, 9], [7, 9],
[6, 9]]
Mapa zlatnika:

 .  .  .  .  .  X  .  .  .  .
 .  .  .  X  .  .  .  X  .  X
 .  .  .  .  .  X  X  .  .  X
 .  .  .  .  .  .  X  .  X  .
 .  X  X  .  .  .  .  .  .  .
 .  X  .  .  .  .  .  .  X  .
 .  X  .  .  .  .  X  X  .  X
 .  .  .  .  .  .  .  .  .  X
 X  .  X  .  X  .  X  .  X  .
 .  .  .  .  .  X  .  X  .  .
```

Figure 6: *Output BFS algoritma*

Vidimo da je broj zlatnika i broj kopanja ostao isti, što smo i očekivali, no sada možemo vidjeti ključnu razliku BFS i DFS algoritama a to jest put kojim obilaze graf/matricu. Da su putevi koje BFS i DFS biraju različiti može nam biti jasno ako usporedimo polje zlatnici koje sadrži koordinate svih zlatnika, polja su jednaka do na poredak, a pošto su algoritmi napisani tako da ćim naiđu na zlatnik da njegovu lokaciju stave u polje zlatnici možemo zaključiti da su algoritmi različitim redoslijedom obišli graf/matricu.

iii. A* algoritam

Kao što smo i diskutirali u uvodnom dijelu, A* algoritam je algoritam koji se služi da nađe najkraći put između dva čvora grafa. To je modifikacija Dijkstrinog algoritama, odnosno Dijkstrin algoritam s dodanom heuristikom. Standardni A* koristi dvije funkcije g i h te pomoću njih zadaje vrijednost f svakom polju gdje je:

$$f(x) = g(x) + h(x)$$

Funkcija g je funkcija koja govori o udaljenosti od polazišta dok funkcija h govori o udaljenosti do odredišta. Grubo govoreći A* algoritam pretražuje čvorove/polja grafa/matrice na "pametniji" način od BFS-a i DFS-a tako da daje prioritet nekim čvorovima/poljima za koja je heuris-

tika odredila da su statistički vjerojatnija da brže vode ka cilju. Tu ideju davanja prioriteta nekim određenim poljima ćemo mi iskoristiti prilikom izgradnje našeg algoritma. Mi ćemo dati veći prioritet susjedima ćelija s većim brojem u sebi. Za primjer možemo uzmemo sliku 7. gdje toplije boje označavaju veći prioritet a hladnije manji prioritet.

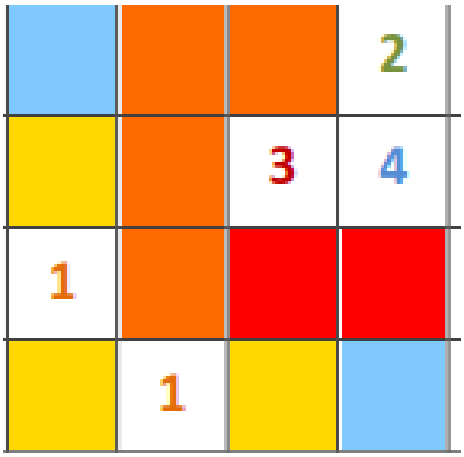

Figure 7: *Primjer odredbe prioriteta u našem algoritmu. Toplije boje su veći prioritet i vice versa*

Ovakvu toplinsku mapu smo dobili tako da smo prvo označili sve susjede najvećeg broja (4) kao visoko prioritetne u crvenoj boji, onda smo za do sada ne označena polja koja su susjedi sljedećeg najvećeg broja (3) označili kao sljedeće na listi prioriteta i tako dok nismo prošli kroz sve preostale brojeve. Prilikom prvog prolaska kroz graf sastavlja liste u koje svrstava polja po zadanom prioritetu. Kasnije prolazi kroz ta polja po njihovom prioritetu i tako brže pronalazi zlatnike. Ako pogledamo output našeg algoritma koji je vidljiv na slici 8. vidimo da je pronađen isti ukupni broj zlatnika u manjem broju kopanja pa možemo zaključiti da naš algoritam brži.

```
Ukupan broj zlatnika:
25
Ukupan broj kopanja
61
Lokacije svih zlatnika:
[[3, 8], [6, 1], [4, 2], [8, 8], [7, 9], [6, 9], [2, 5], [2, 6],
[2, 9], [3, 6], [5, 8], [8, 6], [9, 7], [9, 5], [0, 5], [1, 7],
[1, 9], [5, 1], [4, 1], [6, 6], [6, 7], [8, 0], [8, 2], [8, 4],
[1, 3]]
Mapa zlatnika:

 .  .  .  .  .  X  .  .  .  .
 .  .  .  X  .  .  .  X  .  X
 .  .  .  .  .  X  X  .  .  X
 .  .  .  .  .  .  X  .  X  .
 .  X  X  .  .  .  .  .  .  .
 .  X  .  .  .  .  .  .  X  .
 .  X  .  .  .  .  X  X  .  X
 .  .  .  .  .  .  .  .  .  X
 X  .  X  .  X  .  X  .  X  .
 .  .  .  .  .  X  .  X  .  .
```

Figure 8: *Output našeg algoritma*

III. Žrtvovanje točnosti za brzinu

Ako bi heroj naše priče Joško htio iskopati samo nekoliko zlatnika npr. 5 možemo mjeriti vrijeme "kopanja" za oba algoritma i usporediti ih.

i. Promjene u kodu

U zadnjem dijelu Jupyter bilježnice smo uzeli smo DFS algoritam (mogli smo i BFS nema razlike) i mjerili ga naspram našeg algoritma. Pošto smo rekli da Joško želi iskopati samo 5 zlatnika dodali smo par dodatnih uvjeta u algoritme kako bi oni stali nakon iskopanog 5. zlatnika. Ujedno smo i koristili modul time iz standardne python biblioteke za mjerenje vremena "kopanja" .

ii. Rezultati

Na slikama 9. i 10. su prikazani vremenski intervali potrebni za pronalazak 5 zlatnika koristeći oba algoritma. Iz priloženog vidimo da je naš algoritam otprilike dva puta brži od DFS-a (a samim time i BFS-a) što su sjajne vijesti za heroja naše priče.

```
Vrijeme potrebno za kopanje je:
0.0001838207244873047
```

Figure 9: *Vrijeme potrebno za "kopanje" našem algoritmu*

```
Vrijeme potrebno za kopanje je:
0.0003948211669921875
```

Figure 10: *Vrijeme potrebno za "kopanje" DFS algoritmu*

References

[1] Wikipedia contributors (2022) Breadth-first search. Available at: https://en.wikipedia.org/wiki/Breadth-first_search.

[2] Wikipedia contributors (2022b) Depth-first search. Available at: https://en.wikipedia.org/wiki/Depth-first_search.

[3] Wikipedia contributors (2022a) A* search algorithm. Available at: https://en.wikipedia.org/wiki/A*_search_algorithm.

[4] Levengood, C. (no date) Solving Minesweeper in Python as a Constraint Satisfaction Problem. Available at: https://lvngd.com/blog/solving-minesweeper-python-constraint-satisfaction-p .