Rešavanje problema optimalnog planiranja kretanja u grafu primenom genetskog algoritma

Seminarski rad u okviru kursa Računarska inteligencija Matematički fakultet

Petrović Ana, Spasojević Đorđe
pana.petrovic@gmail.com, djordje.spasojevic1996@gmail.com

4. septembar 2019

Sažetak

U ovom radu predstavljen je gentski algoritam prilagođen problemu optimalnog planiranja kretanja u povezanom, neusmerenom, acikličnom grafu. Cilj je pronaći rešenje koje u najmanjem broju poteza dovodi robota od početnog čvora do cilja, uz izbegavanje prepreka koje se takođe mogu pomerati. Najpre su prikazani pojedinačni koraci genetskog algoritma, a zatim i testiranje kvaliteta rešenja u zavisnosti od promene različitih parametara. Rezultati pokazuju da genetski algoritam relativno uspešno rešava postojeći problem, kao i da u zavisnosti od tipa grafa, menjanje različitih parametara može uticati na kvalitet rešenja.

Sadržaj

1 Uvod **3**

2 Opis algoritma **4**
2.1 Reprezentacija jedinke 4
2.2 Funkcija prilagođenosti 5
2.3 Selekcija 6
2.4 Kreiranje nove generacije 6
2.4.1 Ukrštanje 7
2.4.2 Mutacija 9
2.5 Prikaz algoritma 9

3 Rezultati **11**

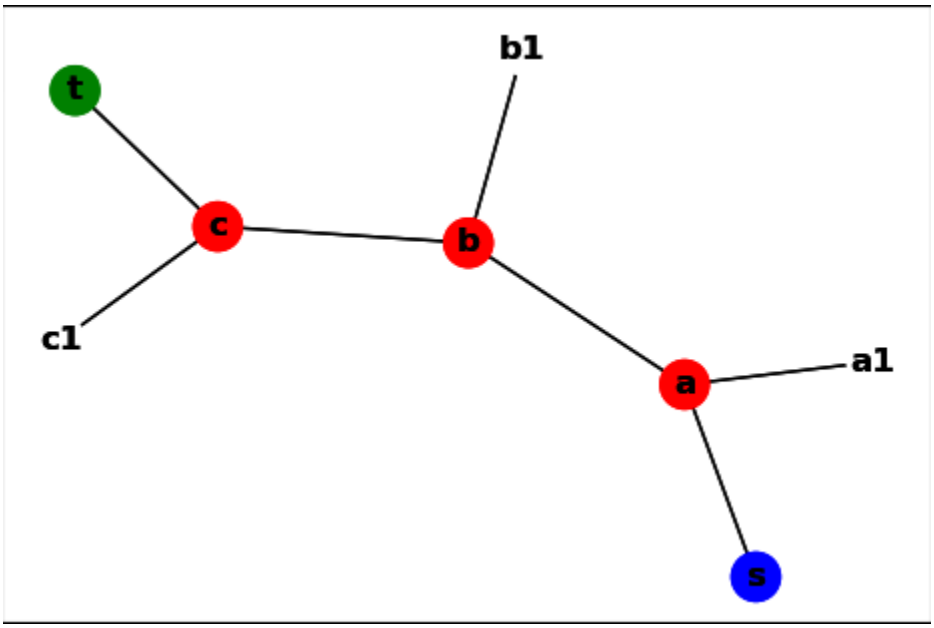
4 Zaključak **15**

Literatura **16**

1 Uvod

Planiranje kretanja (eng. *Motion Planning*) predstavlja jedan od opštih problema u oblasti robotike, čija postavka podrazumeva postojanje robota kojeg treba dovesti od početne tačke do cilja, uz izbegavanje postojećih prepreka [4]. U literaturi je prethodno ponuđeno više rešenja za ovaj problem primenom genetskog algoritma [1, 2, 5]. Iako nijedan od ovih pristupa nije sasvim primenjiv na konceptualizaciju našeg problema, svakako su nam ovi radovi pomogli da bolje razumemo i genetski optimizujemo sam problem.

U ovom radu, dati problem je specifikovan tako da je potrebno naći sekvencu validnih poteza u povezanom, neusmerenom grafu. Radi pojednostavljivanja pronalaženja rešenja, odabrano je da se u primerima koriste samo aciklični grafovi. Graf se sastoji od unapred određenog broja čvorova, od koji svaki može u jednom trenutku da sadrži ili robota, ili prepreku, ili može biti slobodan. Jedan potez podrazumeva premeštanje robota ili prepreke u susedni slobodni čvor. Cilj je pronaći rešenje koje u najmanjem broju poteza dovodi robota od početnog čvora, označenog kao S, do ciljnog čvora, označenog kao T [3]. Na slici 1 je prikazan jednostavan primer postavke jednog ovakvog problema. U svim grafovima koje ćemo koristiti kao primere, čvor je crvene boje ako je na njemu trenutno prepreka, robot se nalazi na plavom čvoru, a ciljni čvor je obojen zelenom bojom dok je slobodan.



Slika 1: Primer postavke problema planiranja kretanja u grafu

2 Opis algoritma

U radu je dat predlog rešenja korišćenjem genetskog algoritma. Prvi korak, koji je ujedno i jedan od najvažnijih u genetskom algoritmu jeste određivanje načina na koji će jedinka (hromozom) biti predstavljena. Od reprezentacije hromozoma značajno zavisi ponašanje algoritma. Nakon toga, najčešće nasumično, generiše se početna populacija koja se sastoji od određenog broja jedinki. U narednom koraku, računa se kvalitet svake jedinke, tj. funkcija prilagođenosti, a na osnovu nje se vrši selekcija najprilagođenijih jedinki iz populacije. Potom se nad selektovanim jedinkama primene genetski operatori ukrštanja i mutacije, kojima se formiraju nove generacije, sve dok se ne dostigne neki kriterijum zaustavljanja i pronađe najbolje rešenje u datom trenutku.

2.1 Reprezentacija jedinke

Pri generisanju početne populacije, razmatrani su različiti načini predstavljanja rešenja. U većini prethodnih radova autori predlažu hromozom u obliku koordinata pozicija robota, ili putanje od početnog do krajnjeg čvora. Takva reprezentacija se nije dobro pokazala u našem slučaju, s obzirom na to da se prepreke mogu kretati na isti način kao i robot, kao i da postoji samo jedan put od čvora S do čvora T, jer je u pitanju aciklički graf.

Zbog toga, početnu populaciju u našem algoritmu čine jedinke koje predstavljaju neku nasumičnu putanju, odnosno niz različitih validnih poteza, koji počinju od početnog stanja grafa. Preciznije, jedinka se sastoji od pokreta robota ili prepreka, koji su predstavljeni preko izvornog čvora, ciljnog čvora, težine puta između ta dva čvora i liste koja predstavlja celu putanju. Na primer, potez od čvora A do čvora B bi bio dat u obliku četvorke ('A', 'B', 1, ['A', 'B']). U slučaju pomeranja prepreka, dozvoljeni su potezi i veće težine od 1, jer je prethodno dokazano da kada je potrebno da se prepreka premesti u neki slobodan čvor (koji joj nije susedan), broj poteza je isti kao i dužina puta između ta dva čvora. Ono što je bitno u tom slučaju je da je na kraju tih poteza stanje grafa takvo da je polazni čvor slobodan, a ciljni zauzet preprekom [3].

Svaka od putanja u populaciji kreće od početnog stanja i zatim se za svaki naredni potez robota ili prepreke najpre razmatra da li je potez validan iz trenutnog stanja grafa, tj. trenutne pozicije prepreka i robota. Generiše se lista svih mogućih poteza iz trenutnog stanja i nasumično bira jedan iz liste. Zatim se izvrši taj potez i dobija se novo stanje grafa. Ako je novo stanje takvo da se robot nalazi u ciljnom čvoru, ne traže se naredni mogući potezi, jer je pronađeno rešenje. Ako je to stanje već ranije bilo razmatrano za tu jedinku, takav potez ne dodajemo u nju, jer ne želimo da se stanja ponavljaju. Može se desiti da se ovakvim nasumičnim dodavanjem istroše sva moguća stanja grafa i da više ne može da se doda nijedan potez. Iz tog razloga, dužina hromozoma neće biti fiksna, jer postoje slučajevi kada više neće biti validnih poteza iz nekog stanja. Maksimalna dužina jednog hromozoma ChromosomeLength određena je tako da bude proizvod dužine putanje od početnog čvora do ciljnog čvora, u oznaci P, i broja prepreka O koje se nalaze na toj putanji.

$$ChromosomeLength = P * O$$

2.2 Funkcija prilagođenosti

Svakoj jedinki dodeljuje se skor, čija visina ukazuje na prilagođenost jedinke. Skor se računa na osnovu poteza koji čine tu jedinku, i stanja grafa nakon tih izvršenih poteza, odnosno mesta na kojima se nalaze prepreke i robot nakon što su potezi izvršeni. Funkcija kao parametar prima i putanju od početnog do ciljnog čvora *path*. Najpre se izračuna ukupna dužina putanje koja čini jedinku *weight*, i skor se na početku postavi na *-weight*. Zatim se proverava da li je trenutno stanje robota jednako ciljnom čvoru T. Ukoliko se robot nalazi na cilju, skoru se dodaje neki određeni visok broj SOLVED_AWARD. Zatim se proverava da li je robot lociran na glavnoj putanji između početnog i ciljnog čvora, za šta se takođe dodaje određena vrednost na postojeći skor. Uz to, ukoliko se na tom putu nalazi neka prepreka, za svaku od njih se skor umanjuje za određenu vrednost. Ukoliko se nijedna prepreka ne nalazi na glavnom putu, a uz to je i ciljni čvor T slobodan, skor se povećava za određenu vrednost. Pored toga, proverava se da li se robot kreće ka cilju kada mu je oslobođena putanja i za svaki potez ka cilju povećava se skor.

```
1000 def fitness_fun(chromosome, o, r, graph, t, path):

1002     weight = 0
         obstacles = copy.copy(o)
1004     for i in range(len(chromosome)):
             weight += chromosome[i][2]
1006
         score = - weight
1008
         if r == t:
1010         score += SOLVED_AWARD

1012     for node in path:
             if node == r:
1014             distance = nx.shortest_path_length(graph, r, t)
                 for obstacle in obstacles:
1016                 if obstacle in path:
                         distance -= 1
1018             score += LENGTH_FACTOR*((len(path)-distance+1))

1020     count_obs = 0
         for obstacle in obstacles:
1022         if obstacle in path:
                 count_obs += 1
1024             obs_distance = nx.shortest_path_length(graph,
                                                        obstacle, t)
1026             score = score - OBS_DIST_PENALTY*count_obs -
                         OBS_DIST*(len(path)-obs_distance+1)
1028
         if ssolver.is_hole(obstacles, r, t) and count_obs == 0:
1030         for i in range(0, len(path)-1):
                 for move in chromosome:
1032                 if move == (path[i], path[i+1]) and move[0] == r:
                         score += ROBOT_MOVE_AWARD
1034         distance = nx.shortest_path_length(graph, r, t)
```

```
            score += R_DIST*(len(path)-distance+1)

    return score
```

Listing 1: Fitnes funkcija

2.3 Selekcija

Pri selekciji, vrši se izbor jedinki iz trenutne populacije za reprodukciju. U našem radu korišćena je turnirska selekcija. Turnirska selekcija podrazumeva da jedinke učestvuju u turnirima, a u svakom od njih, najbolje prilagođena jedinka se proglašava pobednikom. Veličina turnira zadata je parametrom tournament_size. U listu selektovanih jedinki se dodaju pobednici turnira sve dok veličina liste ne dostigne parametar reproduction_size.

```
def tournament_selection(population, tournament_size):
    pop = copy.copy(population)

    winner = None
    tournament = []
    for i in range(tournament_size):
        c = random.choice(pop)
        tournament.append(c)

    winner = max(tournament, key=lambda item:item[0])

    return winner


def selection(population, reproduction_size, tournament_size):

    pop = copy.copy(population)
    selected = []

    while len(selected) < reproduction_size:
        selected.append(tournament_selection(pop, tournament_size))

    return selected
```

Listing 2: Turnirska selekcija

2.4 Kreiranje nove generacije

Pri kreiranju nove generacije, korišćena je strategija elitizma. Njome se obezbeđuje da se kvalitet rešenja ne smanjuje iz generacije u generaciju, tako što se uvek u narednu generaciju direktno prenosi određeni broj najbolje prilagođenih jedinki. U našem algoritmu, broj elitnih jedinki je velik (na primer, polovina veličine populacije) jer se pri generisanju populacije stvaraju veoma nasumične jedinke, s obzirom da se pri svakom potezu u potpunosti menja stanje grafa, odnosno menja se svaki mogući legitimni potez. Na ostale jedinke u populaciji koje ne prelaze direktno u narednu generaciju, primenjuje

se operator ukrštanja, tako što se od prethodno selektovanih jedinki odabiraju dve nasumično, i one postaju roditelji. U ukrštanje će ulaziti samo jedinke koje su veće od neke minimalne dužine, da bi mogle da se iseku na nekoj nasumičnoj poziciji u procesu ukrštanja. Mutaciju primenjujemo i na jednu nasumičnu elitnu jedinku, i na dete koje nastane procesom ukrštanja.

```
1000 def create_new_generation(elite, selected, population_size,
     elite_size, o, r, graph, t, path):

1002     new_generation = copy.copy(elite)

1004     random_elite = random.choice(new_generation)
         while len(new_generation) < population_size:
1006
             valid_parents = False
1008         while(valid_parents == False):
                 parent1, parent2 = random.sample(selected, 2)
1010
                 if len(parent1[1]) > P_LENGTH and
1012                len(parent2[1]) > P_LENGTH:
                     child1 = crossover(parent1, parent2, o, r, graph,
1014                                     t, population_size, path)
                     valid_parents = True
1016

1018         if random.randrange(0, 100) < MUTATION_RATE:
                 mutated_elite = mutation(random_elite, o, r, graph,
1020                                       population_size, path, t)
                 mutated_child = mutation(child1, o, r, graph,
1022                                       population_size, path, t)
                 new_generation.append(mutated_child)
1024             new_generation.append(mutated_elite)
             else:
1026             new_generation.append(child1)

1028     return new_generation
```

Listing 3: Generisanje nove generacije

2.4.1 Ukrštanje

Kako postoji velik broj elitnih jedinki, ukrštanje se vrši nad manjim brojem jedinki iz populacije. U našem algoritmu korišćeno je jednopoziciono ukrštanje, tako što je roditelj čija je putanja kraća presečen na nasumično odabranoj poziciji i. Deo poteza do i iz prvog roditelja mora da se izvrši da se dobije novo stanje grafa. Nakon toga, polazi se od i-*te* pozicije u drugom roditelju i za svaki pojedinačan potez proverava da li je moguć iz prethodno dobijenog stanja. Ovde nastaje problem, jer svaki put mora da se izvrši i doda novi potez na trenutno stanje, čime se dobija potpuno novo stanje. Dete će se dopunjavati dokle god je to moguće, ali u velikom broju slučajeva potezi iz drugog roditelja neće biti mogući u datom stanju. Tako dete koje nastaje često može biti lošijeg kvaliteta od roditelja. Upravo iz tog razloga i koristimo elitizam sa

tako velikim procentom. Za svaki novi potez iz drugog roditelja potrebno je proveriti da li je moguć u odnosu na *sve* prethodne, i samo ako jeste, može da se doda jedan potez, a već za sledeći možda opet neće biti moguće. Parametri funkcije ukrštanja su prvi roditelj, drugi roditelj, stanje prepreka, stanje robota, graf, ciljni čvor, veličina populacije i putanja od prvog do poslednjeg čvora.

```
def crossover(parent1, parent2, o, r, graph,
              t, population_size, path):

    (score1, moves1) = parent1
    (score2, moves2) = parent2
    obstacles = copy.deepcopy(o)
    robot = r

    if len(moves1) <= len(moves2):
        i = random.randrange(1, len(moves1)-1)
        new_moves =  moves1[:i]
    else:
        i = random.randrange(1, len(moves2)-1)
        new_moves = moves2[:i]
    new_o, new_r = ssolver.make_moves(obstacles, robot,
                                      graph, new_moves)

    if len(moves1) <= len(moves2):
        for j in range(i, len(moves2)):
            if moves2[j] in ssolver.possible_moves(new_o, new_r,
                                                   graph) and
                            different_moves(moves2[j],
                                            new_moves[-1]):
                new_moves.append(moves2[j])
                new_o, new_r = ssolver.make_move(new_o, new_r, graph
    ,
                                                 moves2[j][0],
                                                 moves2[j][1])
                if new_r == t:
                    break
    else:
        for j in range(0, len(moves1)):
            if moves1[j] in ssolver.possible_moves(new_o, new_r,
                                                   graph) and
                            different_moves(moves1[j],
                                            new_moves[-1]):
                new_moves.append(moves1[j])
                new_o, new_r = ssolver.make_move(new_o, new_r, graph
                                                 moves1[j][0],
                                                 moves1[j][1])
                if new_r == t:
                    break

    child = fit_chromosome(new_moves, obstacles, robot,
                           graph, population_size, path, t)
    return child
```

Listing 4: Ukrštanje

2.4.2 Mutacija

Mutacija podrazumeva malu promenu genoma koja se dešava sa jako malom verovatnoćom, a koja omogućava različitost u narednoj generaciji i izbegavanje zaglavljivanja u lokalni optimum. U našem slučaju, mutacija će biti jednostavno dodavanje samo jednog poteza iz liste svih mogućih u trenutnom stanju, nakon izvršenih poteza do tog trenutka. Putanje su se nekad skraćivale u ukrštanju, pa će ih dodavanje jednog poteza povećavati s vremena na vreme. Funkcija za parametre ima jedinku *chromosome*, stanje prepreka *o*, stanje robota *r*, graf *graph*, veličinu populacije *population_size*, putanju *path*, i ciljni čvor *t*. Na početku se izvrše svi potezi iz jedinke da bi se dobilo novo stanje. Proveri se da li je novo stanje robota jednako t, i ako jeste izlazi se iz funkcije jer ne želimo da mutiramo jedinku koja je pronašla rešenje. Ako nije, uzima se prvi potez iz liste svih mogućih poteza iz datog stanja koji predstavlja kretanje robota, jer najviše želimo da se robot kreće, a ako ne postoji takav potez onda se uzima bilo koji drugi koji nije vraćanje u prethodan čvor (funkcija *different_moves*). Nakon dodavanja poteza, potrebno je da se ponovo izvrše svi potezi sa novim dodatim, i nakon toga odredi skor novonastale jedinke.

```
1000 def mutation(chromosome, o, r, graph, population_size, path, t):

1002     moves = chromosome[1]

1004     obstacles = copy.copy(o)
         robot = r
1006     new_o, new_r = ssolver.make_moves(obstacles, robot,
                                           graph, moves)
1008     if new_r == t:
             return chromosome
1010
         pm = ssolver.possible_moves(new_o, new_r, graph)
1012
         for p in pm:
1014         if p[0] == new_r and different_moves(p, moves[-1]):
                 moves.append(p)
1016             break
             elif different_moves(p, moves[-1]):
1018             moves.append(p)
                 break
1020
         new_o, new_r = ssolver.make_moves(copy.copy(o), r, graph, moves)
1022     mutated = fit_chromosome(moves, copy.copy(o), r, graph,
                                 population_size, path, t)
1024
         return mutated
```

Listing 5: Mutacija

2.5 Prikaz algoritma

U nastavku je dat kod funkcije koja radi optimizaciju, odnosno objedinjuje sve prethodno opisane korake i izvršava ih sve dok nije postignut kriterijum zaustavljanja,

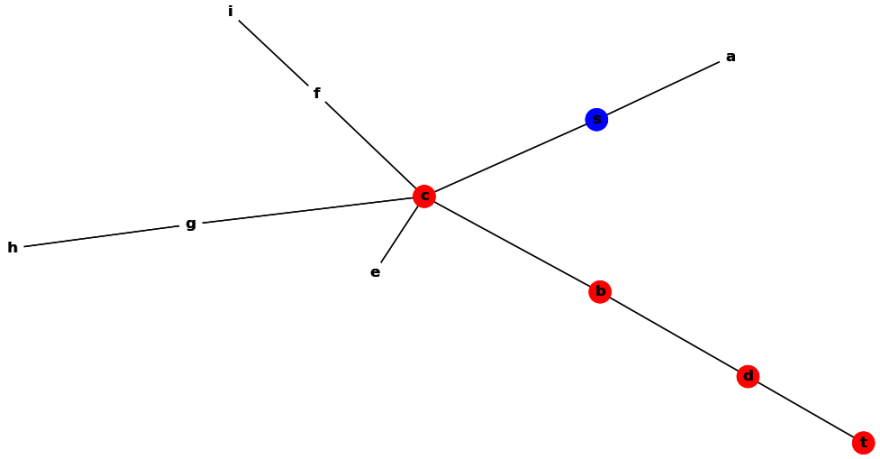
koji je odabran da bude neki maksimalan broj iteracija. Parametri funkcije su stanje prepreka *o*, stanje robota *r*, graf *graph*, ciljni čvor *t*, i putanja od početnog do ciljnog čvora *path*.

```
def solve_genetic(o, r, graph, t, path):

    chromosome_size = len(path) * obstacles_in_path

    population_size = 100
    elite_size = 60
    max_iterations = 200
    reproduction_size = 30
    tournament_size = 10

    initial_population = create_initial_population(o, r, graph, t,
                                                   chromosome_size,
                                                   POPULATION_SIZE)

    scored_population = []
    for i in range(population_size):
        chromosome = initial_population[i]
        scored_population.append(fit_chromosome(chromosome, o, r,
                                                graph,
                                                POPULATION_SIZE,
                                                path, t))

    current_pop = copy.copy(scored_population)

    for i in range(max_iterations):

        elite = []
        for_selection = copy.copy(current_pop)
        for j in range(elite_size):
            largest = max(for_selection, key=lambda item:item[0])
            elite.append(largest)
            for_selection.remove(largest)

        selected = selection(for_selection,
                             reproduction_size,
                             tournament_size)
        current_pop = create_new_generation(elite, selected,
                                            population_size,
                                            elite_size, o, r,
                                            graph, t, path)

        best = max(current_pop, key=lambda item:item[0])

    return best[1]
```

Listing 6: Genetski algoritam

3 Rezultati

U ovom poglavlju biće prikazani rezultati izvršavanja algoritma na različitim primerima. Koristićemo 3 različite postavke problema, i menjati neke od parametara da bismo zaključili kako koji utiče na kvalitet rešenja. Kao ocenu kvaliteta rešenja izdvajamo broj generacije u kojoj je pronađeno to rešenje, broj poteza za koje se izvršava i vreme izvršavanja u sekundama. Primere ćemo nazvati G1, G2 i G3. Primeri problema dati su na slikama 2, 3, 4.



Slika 2: Primer G1

Za sva tri primera podrazumevana je veličina populacije 200, broj elitnih jedinki je 20, maksimalan broj iteracija je 500, veličina za reprodukciju je 30, veličina turnira 5, verovatnoća mutacije 10%.

Kao što se vidi u Tabeli 1, bez obzira na verovatnoću mutacije, do rešenja se uvek stiže u istom broju poteza. Međutim, kada je verovatnoća mutacije 10%, do rešenja se stiže najbrže, odnosno u 12. generaciji, te ovo može sugerisati da je ova visina kriterijuma verovatnoće mutacije najoptimalnija za pronalaženje rešenja.

Tabela 1: Rezultati za G1 sa promenom parametra mutacije

	M = 5%	M = 10%	M = 20%
Generacija	16	12	22
Broj poteza	15	15	15
Vreme izvršavanja	25.871	28.319	20.389

S obzirom da se radi o jednostavnom grafu, rešenje će se pronaći u nekoj od ranijih generacija, te parametar broja iteracija može biti postavljen na 50. Međutim, ovo ne garantuje najoptimalnije rešenje, s obzirom da kasnije generacije mogu unaprediti kvalitet rešenja, što se vidi u slučaju kada je parametar postavljen na 200.

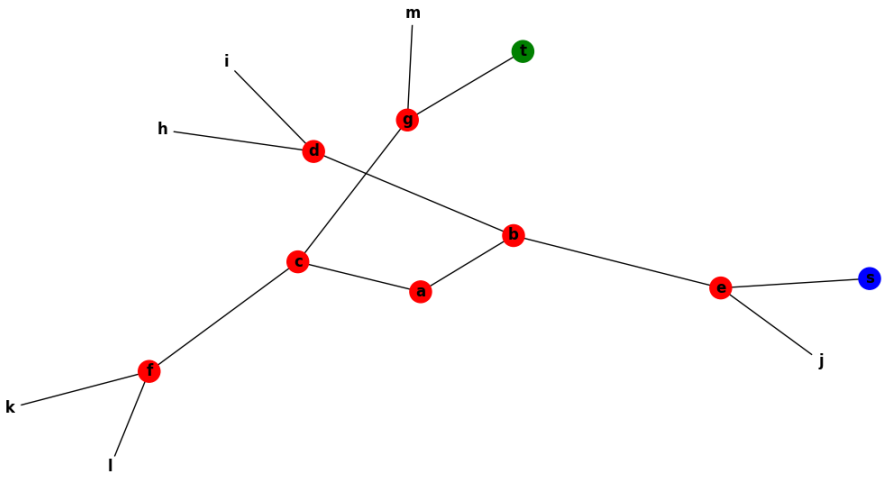
Tabela 2: Rezultati za G1 sa promenom broja iteracija

	maxIt = 50	maxIt = 200	maxIt = 500
Generacija	6	14	22
Broj poteza	17	15	15
Vreme izvršavanja	3.458	8.251	20.389

Kao što Tabela 3 pokazuje, kada je populacija najveća, do rešenja se stiže veoma rano. Za razliku od toga, kada je populacija manja, do rešenja se dolazi kasnije, a uz to je ono za najmanju populaciju i najlošijeg kvaliteta. S obzirom na jednostavnost problema, može se izabrati i manja populacija, kako se ne bi gubilo vreme na kreiranje incijalne populacije.

Tabela 3: Rezultati za G1 sa promenom veličine populacije

	popSize = 50	popSize = 200	popSize = 1000
Generacija	12	78	2
Broj poteza	25	15	17
Vreme izvršavanja	2.39	9.36	66.1



Slika 3: Primer G2

U Tabeli 4 nalaze se rezultati variranja verovatnoće mutacije za primer G2. Kao što tabela pokazuje, kada je verovatnoća mutacije najniža, dobija se i najbolje rešenje. U ovom slučaju, ovaj parametar predstavlja najbolje rešenje jer je priroda grafa takva - kako postoji veliki broj prepreka, ne želimo da se one prečesto nasumično pomeraju, što je ono što mutacija zapravo radi.

Tabela 4: Rezultati za G2 sa promenom parametra mutacije

	M = 5%	M = 10%	M = 20%
Generacija	9	5	20
Broj poteza	15	21	18
Vreme izvršavanja	22.833	28.141	21.351

U Tabeli 5 se nalaze rezultati promene broja iteracija za graf G2. Ova tabela nije mnogo informativna, s obzirom da će se rešenja pronaći relativno rano, te nam nije neophodan veliki broj iteracija.

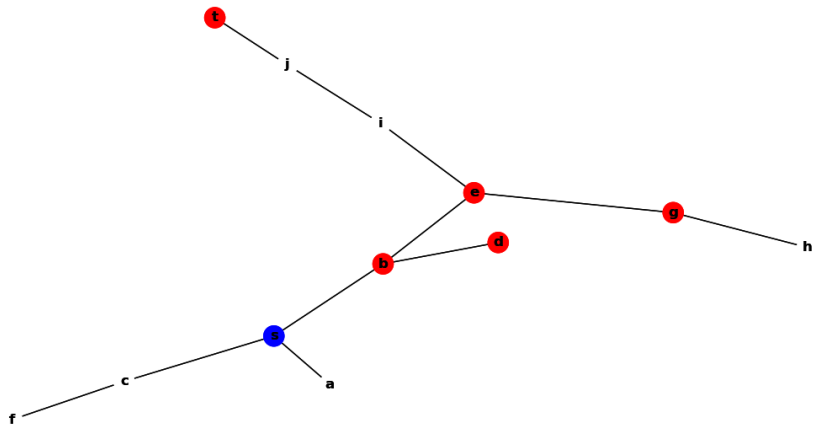
Tabela 5: Rezultati za G2 sa promenom broja iteracija

	maxIt = 50	maxIt = 100	maxIt = 500
Generacija	19	14	7
Broj poteza	16	21	16
Vreme izvršavanja	9.85	17.50	45.87

Tabela 6 pokazuje da kada je populacija manja, do rešenja se stiže u kasnijim generacijama, što sugeriše da bi u ovom slučaju bilo korisno uzeti veću veličinu populacije, slično kao i u prvom primeru.

Tabela 6: Rezultati za G2 sa promenom veličine populacije

	popSize = 200	popSize = 500	popSize = 1000
Generacija	20	9	9
Broj poteza	18	18	17
Vreme izvršavanja	63.7	8.25	129.84



Slika 4: Primer G3

Treći graf je najkompleksniji za rešavanje, odnosno robot može često zapasti u zaglavljeno stanje. Zbog toga je bolje parametar podesiti na veću verovatnoću mutacije, jer se njome obezbeđuje nasumično pomeranje prepreke, što može dovesti do oslobođenja puta robotu, o čemu svedoče rezultati u Tabeli 7.

Tabela 7: Rezultati za G3 sa različitim parametrom mutacije

	M = 5%	M = 10%	M = 20%
Generacija	190	89	17
Broj poteza	41	26	40
Vreme izvršavanja	15.704	24.09	23.999

Rezultati u Tabeli 8 svedoče o tome da ovakva vrsta problema zahteva veći broj iteracija, s obzirom da se problem ne rešava kada je broj iteracija 50. Kada je broj iteracija postavljen na 200, vidimo da se pronalazi rešenje, koje je čak i bolje nego ono koje dobijamo kada je broj iteracija 500.

Tabela 8: Rezultati za G3 sa promenom broja iteracija

	maxIt = 50	maxIt = 200	maxIt = 500
Generacija	nije rešen	89	73
Broj poteza	nije rešen	26	40
Vreme izvršavanja	8.10	24.09	49.87

Tabela 9 pokazuje da je populacija od 50 mala, ali da se sa većim populacijama, od 200 i 1000 jedinki brzo dolazi do rešenja.

Tabela 9: Rezultati za G3 sa promenom veličine populacije

	popSize = 50	popSize = 200	popSize = 1000
Generacija	33	4	6
Broj poteza	40	38	20
Vreme izvršavanja	6.18	32.41	93.064

Na kraju ćemo uporediti vreme izvršavanja našeg algoritma u odnosu na algoritam grube sile, za sva tri primera. Kao što je rečeno, najbolje moguće rešenje obično bude pronađeno već u ranim iteracijama, tako da nije potrebno iterirati do poslednje kao što smo u prethodnim testovima. Dakle, pri poređenju algoritma sa algoritmom grube sile, algoritam se završava čim se pronađe dovoljno dobro rešenje.

Kao što se vidi u Tabeli 10, što graf ima više čvorova, to će algoritam grube sile duže raditi, te u ovom i njemu sličnim primerima, genetski algoritam predstavlja dobru optimizaciju - ne pronalazi najbolje rešenje, ali pronalazi dovoljno dobro rešenje, u značajno kraćem vremenskom intervalu. U druga dva primera, koji imaju manji broj

čvorova, genetski algoritam ne pruža značajno poboljšanje u odnosu na algoritam grube sile, što je i očekivano.

Tabela 10: Vreme izvršavanja u sekundama u odnosu na algoritam grube sile

	Gruba sila	Genetski algoritam
Graf G1	0.146	0.177
Graf G2	29.084	5.068
Graf G3	0.53	1.73

4 Zaključak

U ovom radu prikazan je genetski algoritam za rešavanje problema planiranja pokreta u povezanom, neusmerenom grafu. Kao što je prikazano u radu, algoritam se pokazao kao relativno uspešan u rešavanju ovog problema, i u slučaju primera sa puno čvorova, pruža značajno poboljšanje u odnosu na algoritam grube sile. S obzirom na kompleksnost problema i činjenicu da nije postojala prethodna literatura na koju smo mogli da se oslonimo pri rešavanju problema, ovaj algoritam predstavlja preliminarno rešenje, koje se sigurno može unaprediti, ali se u ovom slučaju pokazao dovoljno dobrim rešenjem.

Literatura

[1] Ismail AL-Taharwa, Alaa Sheta, and Mohammed Al-Weshah. A mobile robot path planning using genetic algorithm in static environment. 2008.

[2] Sarah Alnasser and Hachemi Bennaceur. An efficient genetic algorithm for the global robot path planning problem. *Digital Information and Communication Technology and its Applications (DICTAP) 2016 Sixth International Conference*, 2016.

[3] Christos H. Papadimitriou, Prabhakar Raghavan, Madhu Sudan, and Hisao Tamaki. Motion planning on a graph. *Proc. 35th IEEE Symposium on Foundations of Computer Science (FOCS)*, 09 2001.

[4] Lydia E. Kavraki and Steven M. LaValle. *Motion Planning*. 2008.

[5] Chaymaa Lamini, Said Benhlima, and Ali Elbekri. Genetic algorithm based approach for autonomous mobile robot path planning. 2018.