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

3. septembar 2019

Sažetak

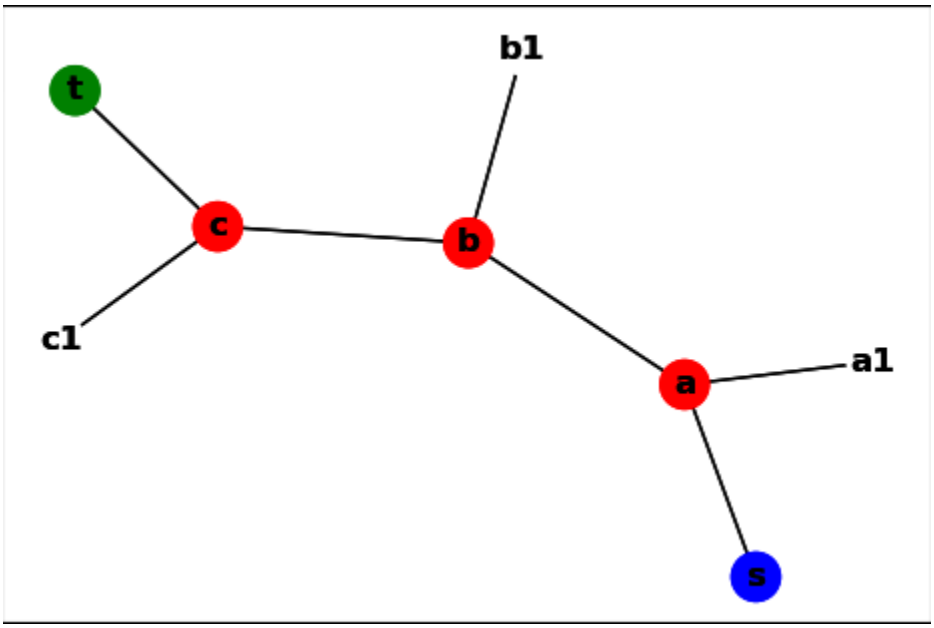
U radu će biti predstavljen gentski algoritam prilagođen problemu optimalnog planiranja kretanja u grafu. ...

Sadržaj

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 sam problem i kako genetski da ga optimizujemo.

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, a svetlo plavom ako je zauzet.



Slika 1: Primer postavke problema planiranja kretanja u grafu

2 Opis algoritma

U radu je dat predlog rešenja korišćenjem genetskog algoritma. Prvi, i jedan od najvažnijih koraka 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.

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 je u obliku četvorke ('A', 'B', 1, ['A', 'B']). U slučaju pomeranja prepreka, dozvoljeni su potezi težine i veće 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 slobodan polazni čvor, 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. Najpre 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. Zatim se provera 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. Povrh toga, proverava se i da li jedinka sadrži pomenute nevalidne vrednosti koje se javljaju kada su ispunjena sva moguća stanja, i od skora se oduzima određena vrednost. Na kraju, od skora se oduzima i težina cele putanje jedinke pomnožena nekim faktorom. U nastavku je dat kod funkcije koja računa taj skor.

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

1002     weight = 0
         obstacles = o[:]
1004
         if r == t:
1006         score += SOLVED_AWARD

1008     for node in path:
             if r == node:
1010             score += ROBOT_ON_PATH_AWARD

1012     count_obs = 0
         for obstacle in obstacles:
1014         if obstacle in path:
                 count_obs += 1
1016             score = score - OBSTACLE_PENALTY

1018     if ssolver.is_hole(o, r, t) and count_obs == 0:
             score += CLOSE_AWARD
1020
         for move in chromosome:
1022         if move == ('0', '0', 0, []):
                 score -= NO_MORE_STATES_PENALTY
1024             break

1026     for i in range(len(chromosome)):
             weight += chromosome[i][2]
1028     score = - weight * WEIGHT_PENALTY

1030     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):
    winner = None
    tournament = random.sample(population, tournament_size)

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

    return winner

def selection(population, reproduction_size, tournament_size):
    selected = []

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

```
def create_new_generation(population, selected, population_size,
                          elite_size, o, r, graph, t, path):
    new_generation = sorted(population, key = lambda item : item[0],
                            reverse = True)[:elite_size]

    while len(new_generation) < population_size:

        valid_parents = False
        while(valid_parents == False):
```

```
            parent1, parent2 = random.sample(selected, 2)

            if len(parent1[1]) > MIN_PARENT_LEN and
               len(parent2[1]) > MIN_PARENT_LEN:

                child = crossover(parent1, parent2, o, r, graph,
                                  t, population_size, path)
                valid_parents = True

        if random.randrange(0, 100) < MUTATION_RATE:
            mutated_child = mutation(child, o, r, graph,
                                     population_size, path, t)
            new_generation.append(mutated_child)
        else:
            new_generation.append(child)

    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. U velikom broju slučajeva potezi iz drugog roditelja neće biti mogući u datom stanju. Tada se dete dopunjava dokle god može. Tako dete koje nastaje često bude lošijeg kvaliteta od samog roditelja. Upravo iz tog razloga i koristimo elitizam sa tako velikim procentom. Ukrštanje mnogo koplikuje situaciju, jer za svaki novi potez iz drugog roditelja mora da se proverava 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. Paremetri 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 možda unaprediti tako što će nastaviti kretanje, i možda u sledećem ukrštanju dozvoliti da se put produži.

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

    moves = chromosome[1]
```

```
    obstacles = copy.deepcopy(o)
    robot = r
    new_o, new_r = ssolver.make_moves(obstacles, robot,
                                      graph, moves)

    pm = ssolver.possible_moves(new_o, new_r, graph)

    for p in pm:
        if different_moves(p, moves[-1]):
            moves.append(p)
            break

    new_o, new_r = ssolver.make_moves(copy.deepcopy(o), r,
                                      graph, moves)

    mutated = fit_chromosome(moves, copy.deepcopy(o), r, graph,
                             population_size, path, t)

    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. Ako se naiđe na dovoljno dobro rešenje, ono koje ima dovoljno visok skor iz pomenute fitnes funkcije pre nego što se generišu sve generacije, izlazi se iz petlje i vraća se to dovoljno dobro rešenje. 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

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

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


    current_pop = copy.deepcopy(scored_population)

    for i in range(MAX_ITERATIONS):
```

```
1020            selected = selection(current_pop, REPRODUCTION_SIZE,
                                     TOURNAMENT_SIZE)
1022            current_pop = create_new_generation(current_pop, selected,
                                                    population_size, e
1024                                                lite_size, o, r,
                                                    graph, t, path)
1026
                best = max(current_pop, key = lambda item:item[0])
1028            if best >= GOOD_ENOUGH:
                    break
1030


1032        return best[1]
```

Listing 6: Genetski algoritam

3 Rezultati

4 Zaključak

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.