SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

HEURISTIČKE METODE OPTIMIZACIJA

Raspoređivanje testova

Goran Belinić
Kristijan Bartol

Zagreb, Siječanj, 2018.

Sadržaj

1. Opis problema

U zadanom zadatku potrebno je rasporediti T testova na M računala. Neki se testovi mogu izvesti samo na određenom podskupu računala, a neki zahtjevno određene globalne resurse R pri izvođenju. Za vrijeme izvođenja testa, svi globalni resursi R koje koristi test T su zauzeti i drugi testovi ih ne mogu koristiti, što znači da se u istom trenutku ne mogu izvoditi.

Zadatak zahtijeva korištenje heurističkih metoda jer se radi o NP teškom problemu i nije za takve probleme nije poznato deterministično rješenje (npr. *brute force* rješenje bi ispitalo svaki vremenski trenutak na svakom mogućem računalu za svaki od zadanih testova pa je jasno da se takav postupak nikad nebi završio).

Primjer dijela ulaznih podataka je:

```
test('t1', 424, [], [])
test('t2', 15, [], [])
...
test('t6', 427, ['m8','m3'], ['r2','r3','r1'])
...
```

Svaki test na ulazu redom ima: svoju oznaku, potrebno vrijeme izvršavanja, niz računala na kojima se može izvršiti (u slučaju prazne liste nema uvjeta) te niz globalnih resursa koje mora koristiti prilikom izvršavanja (u slučaju prazne liste ne test ne koristiti nikakve globalne resurse).

2. Korištene heuristike

Kao rješenje opisanog problema koristi se genetski algoritam. Zadatak podsjeća na problem optimalnog raspoređivanja štapova različitih duljina u kutiju zadane visine, gdje je cilj minimizirati dužinu kutije, a zamišljamo da je širina kutije jednaka debljini štapa, odnosno, 1 (*bin-packing problem*) [1]. U ovom zadatku možemo zamisliti da je visina kutije dimenzija koju minimiziramo, a štapove ponekad ne možemo slagati jedan na drugi (neki štapovi su debeli nekoliko jedinica širine).

Genetski algoritam je metaheuristika pa za njega postoje generičke smjernice za implementaciju, a ostalo ovisi o problemu i jednostavnosti izvedbe križanja, mutacije i određivanja dobrote rješenja.

2.1. Reprezentacija jedinke

Jedinka predstavlja raspored testova po računalima i predstavljeno je običnom Python listom koja sadrži listu testova na svakom računalu. Točnije, listu složenih n-torki sljedeće strukture:

```
rac = (test, vrijeme_pocetka, vrijeme_zavrsetka).
```

Element n-torke *test* opet je složena n-torka strukture:

```
test => (test_id, vrijeme_izvođenja, [racunala], [resursi]).
```

U nastavku je još jednom prikazana struktura nekog računala s naznačenim indeksima te struktura glavne komponente jedinke – *raspored,* koja sadrži rasporeda izvršavanja po računalima

```
rac_i = [(test_1, vrijeme_pocetka_1, vrijeme_zavrsetka_i), …]
raspored = [rac_1, rac_2, …, rac_n]    # n ← broj računala.
```

Ova struktura je intuitivno jasna, a uz par pomoćnih varijablu jednostavno se izvode i potrebne operacije genetskog algoritma. Neke od tih pomoćnih varijabli izravno su uvrštene u samu jedinke, opet u obliku n-torke s 3 elementa:

```
rjesenje_i = (raspored_i, dobrota_i, raspored_resursa_i).
```

Varijabla *raspored_i* sadrži navedenu listu rasporeda izvršavanja po računalima i čini najbitniju komponentu u jedinki populacije. Varijabla *raspored_resursa_i* je pomoćna struktura uz pomoć koje se jednostavnije i brže provjerava je li u određenom trenutku zauzet određeni globalni resurs, a strukturirana je opet u obliku liste po resursima. Kompletna hijerhija po glavnim komponentama je, dakle:

```
populacija => rjesenje => raspored => rac => test => test_id.
```

2.2. Inicijalizacija populacije

Vrlo bitan korak u postupku genetskog algoritma je inicijalizacija populacije. Pri inicijalizaciji obraćalo se pozornost na ispravnost rješenja (*engl. Feasibility*) te **nesavršenost.** Konkretno, računala su se namjerno popunjavala neoptimalno u nadi da će to ostaviti prostora genetskom algoritmu za lokalno pretraživanje.

Prvi korak u inicijalizacijskom postupku je raspoređivanje testova koji koriste globalne resurse na način da se svaki test raspoređuje na slučajno odabrano računalo na kojem se test može izvršiti. Točno vrijeme početka izvršavanja određuje se ispitivanjem dostupnosti svakog od globalnih resursa koje test koristi kako bi se osigurala ispravnost rješenja. Test se na tom računalu postavlja na zadnje mjesto.

U drugom koraku raspoređuju se svi ostali poslovi. Svi se postavljaju na zadnje mjesto slučajno odabranog računala.

Postoje mnogo bolje heuristike za raspoređivanje testova na računala, ali ideja je upravo bila stvoriti relativno loša rješenja koja će imati prostora za napredak u postupku genetskog algoritma.

2.3. Eliminacijski genetski algoritam

Eliminacijski genetski algoritam u svakoj iteraciji odabire dvije jedinke za križanje, tim postupkom dobiva dijete koja mutira i zamjenjuje s treće odabranom jedinkom u slučaju da je dijete bolje (uzimajući u obzir njegovu izračunatu dobrotu).

Pseudokod generičkog postupka je:

```
GEN_ALG(npopulation, cross_prob, mut_prob)
     population = sort(init_population())
     for i in range(iterations)
          // elitistička selekcija
          parent1 = population[0]
          parent2 = population[1]

          // križanje roditelja
          child = cross(parent1, parent2)
          // mutacija djeteta
          child = mutate(child)

          // usporedba s treće odabranim rješenjem
          rand_idx = randint(2, npopulation – 1)
          if child.fitness < population[rand_idx].fitness
```

```
            population[rand_idx] = child
            population = sort(population)

        if TLE or no_improv > 2000 or i > max_iter
            return population[0]
```

U ovom slučaju TLE označava uvjet zaustavljanja, odnosno, iteraciju u kojoj je isteklo dodijeljeno vrijeme za izvršavanje dotičnog testa. Varijabla *no_impr* označava broj iteracija u kojima nije došlo ni do kakvog poboljšanja. Križanje i mutacija predstavljaju dvije suprotne sile – jedna koja kreira jedinke slične roditeljima pa time sužava populaciju te jedna koja gura jedinku u (nadajmo se) novom smjeru pa tako diverzificira populaciju.

2.4. Operator selekcije

Koristi se jednostavna uniformna selekcija iz skupa N% najboljih jedinki populacije.

2.5. Operator križanja

Operator križanja zbog jednostavnosti radi samo s testovima koji ne drže globalne resurse. Ideja je da se genetski materijal prvog roditelja s vjerojatnošću *cross_prob* prenosi u kopiju drugog roditelja na način da se za svaki test koji ne drži globalne resurse, taj test prebacuje na isto računalo drugog roditelja, a originalni test se briše s računala (kopije) drugog roditelja. Pseudokod je dan u nastavku:

```
CROSS(parent1, parent2, cross_prob)
    child = deepcopy(parent2)
    for m in parent1
        for t in m
            if rand.uniform(0,1) < cross_prob
                // u dvostrukoj for petlji kao i gore
                // prodi po testovima od child-a
                if t2.id == t.id
                    m2.delete(t2)
                if m2.id == m.id
                    // pronadi optimalnu lokaciju
                    // na racunalu m2
                    m2.insert(t, optimal_idx)
```

```
child.fitness = calc_fitness(child)
return child
```

Može se primijetiti da u ovoj proceduri nije potrebno provjeravati smije li se zadatak *t2* izvršavati na računalu *m* jer ista ograničenja za testove i računala vrijedi za rješenje *parent1* kao i za rješenje *parent2.*

2.6 Operator mutacije

Ideja operatora mutacije je direktno poboljšati trenutno rješenje. Direktno se poboljšanje ostvaruje tako da se, kao i u postupku križanja, prvo u dvostrukoj *for* petlji prolazi po svim raspoređenim testovima kroz *m* računala. Svaki se test potom s vjerojatnošću križanja *mutate_prob* premješta na računalo koje je trenutno najmanje opterećeno, a to se može odrediti tako da dohvati *end_time* zadnjeg testa postavljenog na izvršavanje na tom računalu.

U ovom postupku, za razliku od postupka križanja, premještaju se i testovi koji koriste globalne resurse. Ako takvo premještanje nije direktno moguće, taj test se ostavlja na starom računalu. Pseudokod je dan u nastavku:

```
MUTATION(child, mutate_prob)
     for m in child
          for t in m
               if rand.uniform(0,1) < mutate_prob
                    // pronađi najmanje opterećeno rač
                    // pobriši s originalnog računala
                    // *dodaj na optimalno mjesto u NOČ
     return child
```

Pronalazak najmanje opterećenog računala jednostavno se provodi iterativno ispitivanjem *end_time*-a posljednjeg testa svakog računala i nalazi se ono s najmanjim brojem i taj se indeks pamti. Dodavanje na optimalno mjesto unutar računala podrazumijeva ili pronalazak dovoljno velike praznine unutar rasporeda izvršavanja u koji bi stao potrebni test ili jednostavno postavljanjem izvršavanja na zadnje mjesto u rasporedu tog računala.

Kako bi se svaki test izvršio samo i točno jednom, potrebno ga je pobrisati s originalnog računala. (Može se primijetiti da ovakav postupak ne ugrožava situacija u kojoj brišemo pa dodajemo test opet na isto mjesto – nakon što pobrišemo test nalazimo se na istom mjestu na kojem se nalazi dovoljno velika praznina za isti taj test.)

2.7. Uvjet zaustavljanja

Postupak se zaustavlja nakon one iteracije algoritma u kojoj je trajanje premašilo dopušteno vrijeme izvršavanja. U tom se trenutku vraća najbolje rješenje populacije koje je ono na indeksu 0 trenutne sortirane populacije (prikazano u pseudokodu odjeljka 2.3.). Za neograničeno vrijeme izvršavanja postoje 2 (teoretski 3) moguća uvjeta zaustavljanja. Prvi je uvjet broj iteracija postavljen na 10000 (usporedbe radi, testovi ukupnog trajanja 5 minuta izvrše nešto manje od 1000 iteracija). Postupak se također u prekida u slučaju da više od 1000 iteracija nije bilo apsolutno nikakvog poboljšanja. To znači da se 1000 iteracija nije dogodilo da je dijete bolje od neke slučajno odabrane jedinke unutar populacije.

2.8. Hiperparametri algoritma

Osim broja iteracija i uvjeta zaustavljanja, podešavaju se i drugi hiperparametri. Kroz par eksperimenata zaključeno je da se vjerojatnosti križanja i mutacije najbolje ponašaju u omjeru 1:2 s obzirom na brzinu konvergencije rješenja. To ima smisla jer je mutacija osmišljena da nastoji urediti rješenja i popuniti manje opterećа računala, a križanje takve zahtjeve ne uzima u obzir. Tako se lako može dogoditi premještanje posla s računala koje je izvršavalo samo jedan test pa je tako ostalo prazno. Konkretne vjerojatnosti su 0.3 za operator križanja i 0.6 za operator mutacije.

Populacija je konačno postavljena na veličinu 100 jer se dodatnim povećanjem rezultati nisu poboljšavali. Bolji udio populacije iz kojeg se biraju roditelji postavljen je na 50%, odnosno, za populaciju veličine 100, to je 50 jedinki. Jedinke lošije od 50. nisu imale nikakvu šansu biti odabrane. Ovakav se odabir za uniformni operator selekcije pokazao optimalnim s obzirom na brzinu konvergencije populacije.

Dodatno, u postupku selekcije treće jedinke – jedinke koja će potencijalno biti zamijenja, koristi se elitizam. Najbolje 3 jedinke ne mogu biti evaluirane za zamjenu. Sve ostale jedinke zajedno s odabranim roditeljima mogu biti odabrane za zamjenu.

3. Rezultati i zaključak

Ispostavilo se da genetski algoritam ipak daje neispravna rješenja prilikom izvođenja križanja i mutacije pa se ne može pouzdano govoriti o kvaliteti rješenja. Ipak, greška se s iteracijama smanjivala do konvergencijske točke koja je varirala, a najčešće bi nastupila prije 1000. iteracije. Greška u kodu je dosta komplicirana jer ne nastupa nakon svakog pomaka testa, nego samo u određenim situacijama, a ista je logika i u funkciji križanja i mutacije.

Primjer izlaza programa je:

Solving in ne time...

Iteration #0 | *Fitness: 55468* | *Improvements: 0*

Iteration #100 | *Fitness: 54995* | *Improvements: 72*

Iteration #200 | *Fitness: 54691* | *Improvements: 84*

Iteration #300 | *Fitness: 54691* | *Improvements: 60*

Iteration #400 | *Fitness: 54002* | *Improvements: 58*

Iteration #500 | *Fitness: 53968* | *Improvements: 16*

Iteration #600 | *Fitness: 53661* | *Improvements: 34*

Iteration #700 | *Fitness: 53661* | *Improvements: 52*

Iteration #800 | *Fitness: 53661* | *Improvements: 43*

...

Nakon 600. iteracije nastupa još jedno poboljšanje negdje oko 2000. iteracije. Dio ispisa u kojem je naveden *Improvements* broji koliko je puta križano i mutirano dijete postalo bolje od roditelja. Ako je taj broj jednak 0 barem puta u slučaju ovog ispisa, izvođenje će se zaustaviti jer će biti zadovoljen uvjet za konvergenciju.

2.1. Moguća poboljšanja

Osim toga što je genetski algoritam neispravan, poboljšanja su vjerojatno moguća u inicijalizacijskom mehanizmu, ali u tom slučaju potrebno je eksperimentirati sa svakim mehanizmom pojedinačno.

4. Literatura

[1] Bin-Packing Problem (2006.), https://en.wikipedia.org/wiki/Bin_packing_problem

5. Sažetak

Genetski algoritam korišten kao metaheuristika i inicijalizacijska heuristika korištena za specifični problem raspoređivanja ispita na dostupna računala. Raspoređuje se *T* testovana *M* računala, uz dodatno ograničenje od *R* globalnih resursa koje neki testovi trebaju pri izvođenju. U određenom trenutku samo jedan test može koristiti potrebni resurs. Posebna reprezentacija rješenja koja je u osnovi lista rasporeda po dostupnim računalima kroz vrijeme. Takva reprezentacija izravno se koristi pri križanju i mutaciji genetskog algoritma.