Univerzitet u Novom Sadu

Fakultet tehničkih nauka

# Dokumentacija za projektni zadatak


| | |
|---|---|
| Studenti: | Stojkov Sara, SV38/2023<br>Milutin Marko, SV40/2023 |
| Predmet: | Nelinearno programiranje i evolutivni algoritmi |
| Broj projektnog zadatka: | 3 |
| Tema projektnog zadatka: | Genetski algoritam, problem pravljenja rasporeda |

# Opis problema

Problem pravljenja rasporeda nastave je čest izazov različitih organizacija i institucija zbog ograničenog vremena i prostora koje treba što bolje iskoristiti. Ovaj problem svodi se na raspoređivanje predavanja i vežbi po učionicama i radnim danima, u okviru definisanog radnog vremena. Ulazni podaci su imena i trajanja događaja koje treba rasporediti. Raspored podrazumeva dodeljivanje tačne satnice, dana i učionice svakom događaju, uz poštovanje pravila – između aktivnosti u istoj učionici mora biti najmanje 15 minuta pauze, a radni su dani od ponedeljka do petka. Takođe, podrazumeva se da se ne mogu odvijati 2 aktivnosti u istoj učionici u isto vreme.

S obzirom na to da ovaj problem spada u NP-teške probleme, odnosno one za koje ne postoje efikasni polinomijalni algoritmi koji mogu da obrade veliki broj ulaznih podataka, ovo je dobra prilika za korišćenje genetskog algoritma. Takav pristup predstavlja aproksimacionu metodu koja omogućava brzo pronalaženje dovoljno dobrog rešenja, bez napornog ispitivanja svih mogućnosti.

# Uvod

Evolutivno programiranje koristi algoritme koji imaju fiksnu strukturu, dok dozvoljavaju numeričke parametre da *evoluiraju* odnosno menjaju se na osnovu nekih pravila i time se približavaju rešenju. Jedan od glavnih predstavnika ovakvih algoritama je upravo genetski algoritam. Genetski algoritmi su algoritmi za pretragu i optimizaciju koji se baziraju na principima prirodne evolucije. Problemi koji se rešavaju ovim algoritmom bivaju apstrakovani na jedinke i populacije pri čemu je jedinka jedno od potencijalnih rešenja. Ovaj tip algoritama primenjuje strategije optimizacije tako što forsira prirodnu selekciju odnosno opstanak najboljih jedinki. Generalno, ovaj proces se sastoji iz dve faze: prva podrazumeva selekciju jedinki koje će učestvovati u pravljenju sledeće generacije a druga podrazumeva primenu *crossover* operacije odnosno ukrštanja jedinki i mutiranje tih jedinki. Na početku algoritma je uvek potrebno generisati nasumično prvu generaciju koja će ući u pomenuti ciklus.

# Implementacija

Fajl struktura projekta je sledeća: u folderu *genetic_algorithm* se nalaze fajlovi *individual.py* i *generation.py*. U modulu *individual.py* se nalazi klasa jedinke u okviru koje je njena randomizacija, računanje fitnesa, mutacije i van klase je funkcija za ukrštanje dve jedinke. U *generation.py* se nalaze funkcije za selekciju, odabir roditelja pri ukrštanju, kreiranje početne generacije i sam „životni ciklus“ odnosno petlja genetskog algoritma. U folderu *structures* nalazi se klasa koja predstavlja jedan predmet (jednu aktivnost) odnosno pomenuta *Subject* klasa. U folderu *schedules* se nalaze rasporedi koje je algoritam generisao. Takođe postoji i *Python* modul *loader.py* koji je zadužen za učitavanje ulaznih podataka iz *txt* fajla (ili alternativno iz stringa kako bi se izbegle greške sa putanjom do fajla).

Pri pokretanju programa (*main.py*), podaci se učitavaju iz zadatog ulaznog fajla. Svako predavanje ili vežbe su predstavljeni klasom *Subject* koja za atribute ima naziv aktivnosti i *duration* odnosno trajanje (koliko uzastopnih *timeslot* mesta od po 15 minuta će da zauzme). Predavanja i vežbe se čuvaju kao *Subject* objekti i učitavaju u listu, pri čemu je indeks svakog elementa u listi ujedno indeks tog predmeta. Sem aktivnosti, iz fajla se učitava i broj učionica koje su na raspolaganju. Učionice se čuvaju u rečniku, gde je ključ indeks učionice a vrednost je naziv te učionice ili prostorije.

Genetski algoritam je u početku implementiran kroz definisanje njegovih koraka i ulaznih parametara. Osnova ovog algoritma je jedinka – u našem slučaju jedinka jeste jedan raspored časova odnosno način da se ulazni časovi rasporede u *n* učionica i *m* radnih dana. Radi pojednostavljenja logike, vremenski intervali odnosno satnice podeljene su na intervale od po 15 minuta – to je jedinica vremena odnosno trajanje jednog *timeslot* intervala u rasporedima (jedinkama). Takođe, jedan raspored logički je podeljen u blokove, pri čemu jedan blok predstavlja jednu učionicu i jedan dan odnosno jednu celinu u kojoj neko predavanje ili vežbe mogu da se odigraju. U ovom slučaju, blok će imati 4 (četvrtine sata) * 12 (sati) = 48 timeslotova. Prelamanje aktivnosti između blokova je rešeno proverama u kodu, dok je sama jedinka implementirana kao lista pomenutih *timeslot*-ova (timeslot je lista – dakle lista listi). Smeštanje nekog časa u raspored je zapravo upisivanje njegovog indeksa u onoliko uzastopnih timeslotova koliki je *duration* tog časa (*Subject).*

Jedinka je predstavljena klasom *Schedule* koja kao atribute sadrži:

- *class_list* – lista koja predstavlja sve timeslotove koji će imati upisane indekse aktivnosti, dužina velike liste je:

  $$12\ (sati)\ *\ 4\ (\check{c}etvrtine\ sata)\ *\ 5\ (dana)\ *\ 5\ (u\check{c}ionica)\ =\ 1200\ timeslot-ova$$

  kao što je spomenuto, timeslot je lista u kojej može biti 0 ili više indeksa aktivnosti koje se tada dešavaju. Preklapanja aktivnosti su dozvoljena, algoritam to rešava penalom u *fitness* funkciji.

- *mapping* – rečnik u kom ključevi predstavljaju indekse časova, a vrednost je indeks početnog *timeslot*-a te aktivnosti u listi *class_list* koja služi za bržu pretragu
- *fitness*_score – *fitness* jedinke kako bi se izbeglo redudantno računanje (i skupi prolasci kroz listu)
- *num_blocks* – ukupan broj blokova, proizvod broja dana i broja prostorija
- *block_size* – veličina bloka potrebna za kasnija računanja

Populacija je implementirana kao lista jedinki  lista objekata *Schedule*.

**Formiranje prve generacije** vrši se tako što se funkcija *set_random_classes* poziva nad praznim inicijalizovanim rasporedima. Ova funkcija prolazi kroz sve časove zadate ulaznim fajlom i smešta svaku od aktivnosti u nasumični blok. Pokušava da smesti časove u istom bloku tako da nema preklapanja odnosno *overlap*-a ukoliko je to moguće, i garantuje da nema prelaska aktivnosti iz jednog bloka u drugi. Svakoj jedinki se pri kreiranju računa i *fitness score*.

**Kriterijum optimalnosti** ove metode optimizacije je već unapred zadat u specifikaciji problema. Cilj je da prosečno vreme početka nastave za svaku učionicu i svaki dan bude što kasnije, a prosečno vreme završetka što kasnije. Ukoliko obeležimo broj učionica sa $n$, a broj radnih dana sa $m$ tada će broj kombinacija učionica i dana biti $n \cdot m$ i to možemo nazvati broj blokova. Takođe, za svaki od ovih blokova (sa rednim brojem *i*) može se definisati $p_i$ odnosno vreme od 07:00 do početka prvog časa, kao i $k_i$ vreme od kraja poslednjeg časa do 19:00. Ako ovaj kriterijum formalizujemo to je:

$$f(p,k) = \sum_{i=1}^{n \cdot m} p_i \cdot k_i$$

***Fitness* funkcija** služi da odredimo koje jedinke su bolje od drugih. U našoj implementaciji, *calculate_fitness* funkcija se sastoji od 2 dela: prvi deo proverava validnost rasporeda (preklapanje aktivnosti, pauze pre i posle) i ukoliko raspored ne poštuje pravila povećava mu se penal, a drugi deo služi da vrednuje učionice u kojima časovi počinju kasnije i završavaju se ranije kao bolje. Konačan rezultat jeste skor drugog dela podeljen sa penalom.

$$fitness\_score\ =\ total\_score\ /\ penalty$$

**Selekcija** odnosno izbor jedinki koje čine sledeću generaciju implementirana je kroz elitizam. Elitizam podrazumeva da postoji podela između roditeljske populacije i populacije dece. Iz roditeljske populacije u sledeću prelazi samo *n* najboljih jedinki po fitnesu (elita), dok se ostatak nove populacije popunjava najboljim jedinkama dece.

**Ukrštanje (*crossover*)** implementirano je tako da dva roditelja daju dva potomka. Izbor roditelja vrši se ruletskom selekcijom gde se favorizuju jedinke sa boljim fitnesom ali ipak se unosi i stepen nasumičnosti. Za rangiranje jedinki se ne koristi njihov fitnes već njihov rang u populaciji radi skaliranja, smanjenja razlika i efikasnije randomizacije roditelja. Ovaj način odabira je bitan kako se ne bi došlo do jednolikosti populacije, a opet se smanjuje šansa da manje adaptirane jedinke daju potomstvo. Samo ukrštanje je implementirano preko *three-way crossover* tehnike – proizvoljno se odaberu 2 tačke odnosno indeksa u listi časova. Jedna tačka se bira u prvoj polovini, a druga u drugoj polovini liste časova. Te tačke dele liste oba roditelja na po 3 segmenta časova. Ta 3 segmenta se kombinuju kao 1-2-1 i 2-1-2 gde 1 predstavlja reprezentativni segment prvog roditelja, a 2 segment drugog, čime se dobijaju 2 nove jedinke. Navedeni raspored podrazumeva primarne roditelje za te segmente, odnosno definiše od kojeg roditelja će prvo pokušati da prepiše sekvencu časova. Ukoliko dođe do preklapanja kod nekog časa, tada se prepisivanjem sekvence od ne-primarnog roditelja pokušava korekcija. Nakon što se formiraju nove jedinke, one potencijalno prolaze kroz mutacije.

**Mutacije** služe da unesu varijabilnost u populaciju i pomognu da se prođe neki lokalni ekstrem. Zato je i u ovoj implementaciji funkcija mutacije u formi randomizacije odnosno *shuffle*-a časova – časovi se sa trenutne pozicije nasumično premeštaju na druge pozicije u okviru istog bloka uz pokušaj sprečavanja *overlap*-a aktivnosti. Broj časova koji se ovako premešta je takođe varijabilan i to nasumičan između $\frac{1}{4}$ i $\frac{1}{16}$ ukupnog broja časova. Šansa da se mutacija desi je varijabilna tokom izvršavanja algoritma i sa brojem generacija opada.

Algoritam se ponavlja dok se ne dođe do maksimalnog broja generacija ili do toga da je vrednost *fitness score*-a najbolje jedinke u populacji dovoljno blizu vrednosti za koju se smatra da je optimalna.

# Zaključak

Sem implementacija funkcionalnosti, parametri zadati pre pokretanja programa (u *const.py* fajlu) takođe utiču na rezultate algoritma. Parametri obuhvataju: broj radnih dana u nedelji *DAYS_NUM* (unapred zadato), putanju do fajla sa ulaznim podacima *FILE_PATH*, veličinu populacije (broj jedinki) *POPULATION_SIZE*, šansu da jedinka mutira *MUTATION_CHANCE*, deo jedinki iz prethodne generacije koji se čuva pri selekciji *KEEP_PERCENT*, maksimalan broj generacija *MAX_GENERATIONS*. Ovde ćemo diskutovati koje vrednosti ovih parametara su izabrane kao optimalne i zašto.

Parametar *POPULATION_SIZE* postavljen je na 100 jedinki jer se pri povećavanju populacije ne dolazi do boljeg rešenja, već algoritam samo postaje značajno sporiji. Manje vrednosti ovog parametra ne pružaju dovoljnu varijabilnost jedinki.

Parametar *KEEP_PERCENT* koji se koristi pri selekciji (elitizmu) postavljen je na 0.2 jer se pri manjim procentima dešava da algoritam ne uspeva da nađe dovoljno dobro rešenje, a ukoliko se čuva više jedinki iz roditeljke populacije, sam fitnes brzo konvergira i nestane varijabilnost u populacijama.

Parametar *MAX_GENERATIONS* je testiran više puta, ali pokazalo se da nakon 2000 generacija algoritam stagnira, tako da uglavnom nema potrebe za daljim izvršavanjem.

Što se tiče *MUTATIONS_CHANCE* parametra, on sadrži 3 vrednosti i to [0.4, 0.3, 0.2]. Prva vrednost 0.4 se primenjuje na prvoj polovini maksimalnog broja generacija, druga vrednost 0.3 na sledećoj polovini (ukupno četvrtina) generacija, a 0.2 do kraja. Za varijabilnu šansu mutacije odlučili smo se jer se pokazalo da je u prvim iteracijama mnogo bitnije da mutacije unesu diverzitet u populaciju, dok kasnije mogu odvratiti od pravog rešenja.

Takođe, nešto što je možda i najviše uticalo na rešenja koja smo dobijali jeste implementacija *calculate_fitness* funkcije. Bazirano samo na kriterijumu optimlanosti, dobijaju se rešenja odnosno rasporedi u kojima je većina blokova (odnosno učionica) sasvim prazna, dok je nekoliko njih puno od početka do kraja radnog vremena. Zdravorazumski, to rešenje ne može biti optimalno jer se na taj način nepotrebno troše resursi, odnosno ne koriste se sve učionice koje su na raspolaganju. Zbog toga smo uveli penale odnosno „kazne“ za prazne učionice. Prema zadatom kriterijumu optimalnosti gde se gleda samo proizvod vremena pre početka i nakon kraja časova, bolje se rangira primer u kome je jedna učionica gusto zauzeta od ujutru do uveče, a druga ima jedno kratko predavanje ili vežbe, nego primer u kome su te aktivnosti ravnomerno podeljenje između ove dve učionice (2 bloka). Iz sličnih razloga kao i ocenjivanje praznih učionca, odlučili smo da tražimo rešenje koje će pokušati da ravnomerno rasporedi časove na sve dostupne učionice i dane. Te uravnoteđene slučajeve smo pospešili zatvorenim mutacijama (u okviru istog bloka). Iako će takve jedinke imati manji *fitness score*, iz navedenih razloga takve jedinke smatramo boljim i bližim rešenju u realnom svetu.

Rezultat algoritma ogleda se u fitnesu najbolje jedinke u poslednjoj generaciji odnosno pri izlasku iz petlje i uglavnom iznosi između 1 000 000 i 1 200 000 u skladu sa našom *fitness* funkcijom, uz samo nekoliko rezultata čiji je fitnes van tog opsega (1 202 850.00, 1 147 725.00, 1 107 450.00, 1 052 100.00, 1 170 675.00, 1 137 150.00, 1 152 225.00, 1 183 500.00). Najbolje rešenje do kog smo došli korišćenjem gore navedenih parametara je 1 210 950.00 i taj raspored se nalazi u folderu *schedules* u fajlu ***najveciFitness.html***. Ipak, kao što je rečeno, jedinke nismo vrednovali samo na osnovu fitnesa, već i po tome koliko su praktični u realnom svetu. Pomenuta jedinka sa najvećim fitnesom ne izgleda kao da su svi dani i učionice približno „opterećeni“, odnosno nisu svi dani ravnomerno popunjeni. U skladu sa tim izdvojili bismo rasporede ***uravnotezen1.html*** i ***uravnotezen2.html*** kao takođe reprezentativne. Odabir ovih realnijih rasporeda možda jeste subjektivan, ali i *fitness* ovih rasporeda je dovoljno velik da se smatraju odgovarajućim.

U skladu sa tim da su rešenja međusobno prilično slična i uravnotežena, nameće se zaključak da je implementacija genetskog algoritma na ovaj problem dala prihvatljivo i brzo rešenje.