Sadržaj

[Predgovor 1](#_Toc340959914)

[1. EM algoritam 2](#_Toc340959915)

[1.1. Uvod 2](#_Toc340959916)

[1.2. Motivacija 2](#_Toc340959917)

[1.3. Heuristike zasnovane na elektromagnetizmu 3](#_Toc340959918)

[2. Keširanje 12](#_Toc340959919)

[2.1. Tehnike keširanja 12](#_Toc340959920)

[2.2. LRU primenjen na linearnoj strukturi 12](#_Toc340959921)

[2.3. LRU primenjen na dvostrukoj heš tabeli 13](#_Toc340959922)

[2.4. Implementacija pomoću heš-red strukture 13](#_Toc340959923)

[2.5. Heš funkcija 14](#_Toc340959924)

[2.6. Rešavanje kolizije 14](#_Toc340959925)

[2.7. Dobijanje CRC kodova 15](#_Toc340959926)

[2.8. Dodavanje elemenata u heš tabelu 16](#_Toc340959927)

[2.9. Korišćenje reda 17](#_Toc340959928)

[3. Maksimalni problem srednjih elemenata 18](#_Toc340959929)

[3.1. Uvod 18](#_Toc340959930)

[3.2. Matematička formulacija 19](#_Toc340959931)

[3.3. Reprezentacija jedinki i funkcije cilja 20](#_Toc340959932)

[3.4. Motivacija za ovakvu reprezentaciju 21](#_Toc340959933)

[4. Rezultati 23](#_Toc340959934)

[4.1. Kratka analiza dobijenih rezultata 27](#_Toc340959935)

[Literatura 28](#_Toc340959936)

[Dodatak A 30](#_Toc340959937)

# Predgovor

Metaheuristika označava metod koji optimizuje određene probleme u računarstvu tako što iterativno pokušava da poboljša pronađeno rešenje, čiji je kvalitet određen datom merom. Njihova dobra strana je što su u velikoj meri nezavisne od problema koji optimizuju, jer se oslanjaju ili na osnovne pretpostake o problemu ili im pretpostavke nisu ni potrebne. Takođe, veoma su popularne i zbog mogućnosti pretrage veoma velikih prostora u kome se nalaze moguća rešenja. Međutim, one ne garantuju da će biti pronađeno optimalno rešenje. Jedan od razloga za to je što mnoge metaheuristike koriste neki oblik stohastične optimizacije.

Želja autora ovog rada je da prikaže jedan mogući metod ubrzanja za jednu posebnu vrstu metaheuristika koje su zasnovane na elektromagnetizmu. Ideja za njihovo uvođenje je potekla iz teorije eletromagnetizma koja posmatra zakone po kojima naelektrisane čestice pokušavaju da dostignu optimalni položaj.

Rad je organizovan na sledeći način. U prvoj glavi je opisana heuristika koja je zasnovana na elektromagnetizmu uključujući motivaciju za njeno uvođenje, sličnosti i razlike algoritma i fizičkog zakona, kao i način implementacije na računarima. Fokus druge glave je na implementaciji pogodne strukture podataka za skladištenje keširanih podataka. Maksimalni problem srednjih elemenata je uveden u trećoj glavi. U njoj su opisani i srodni problemi zajedno sa primenama u raznim oblastima, data je precizna matematička definicija problema, kao i pristup njegovom rešavanju koji omogućava da se uspešno primeni EM algoritam. Četvrta glava je rezervisana za rezultate i njihovu analizu. U dodatku A se nalaze komentarisani izabrani delova koda implementacije.

Najveću zahvalnost za nastanak ovog rada dugujem svom mentoru, profesoru dr Vladimiru Filipoviću koji je pokazao izuzetno veliku podršku i pomoć pri odabiru stručne literature, stvaranju rezultata i pisanju samog rada. Takođe, želim da se zahvalim profesorima dr Miroslavu Mariću i dr Dušanu Tošiću koji su sa pažnjom pročitali ceo rad i dali niz korisnih primedbi i sugestija. Osim njih spomenuo bih i asistenta Aleksandra Kartelja sa kojim sam sarađivao oko pisanja programa. Naravno, za sve greške i nedostatke u radu krivicu snosi samo autor.

Beograd Uroš Rajković

Oktobar, 2012.

# EM algoritam

## Uvod

Globalna optimizacija je poslednjih godina postala oblast koja se jako brzo razvija. Mnogi problemi koji se javljaju u stvarnom životu iz oblasti poput fizike, hemije i molekularne biologije uključuju *nelinearne funkcije sa mnogo promenljivih* (multimodalna interakcija, neprekidnost, itd.) koje je veoma teško optimizovati pomoću konvencionalnih matematičkih aparata, kao što su gradijentne metode. Primera ima puno [1, 2].

Da bi prevazišli opisane poteškoće, 1980-tih je počeo ubrzani razvoj stohastičkih metoda pretraživanja. One se uglavnom oslanjaju na snagu savremenih računara. Tako, na primer, algoritam slučajnog pretraživanja može dati korisne rezultate čak i u situacijama kada drugi algoritmi podbace zbog nepravilnosti ili velike dimenzije problema.

Radi jednostavnosti, posmatraćemo specijalnu klasu optimizacionih problema sa ograničenim promenljivim u sledećoj formi:

|  |  |  |
| --- | --- | --- |
|  |  | 1 |

gde je

|  |  |  |
| --- | --- | --- |
|  | . | 2 |

Zbog svoje jednostavnosti ovaj model je obimno korišćen prilikom izučavanja algoritma slučajnog pretraživanja [3], kao i algoritama koji koriste direktne metode [4].

## Motivacija

Kod stohastičnih metoda pretraživanja, algoritmi zasnovani na populaciji počinju izborom nasumičnih tačaka iz dozvoljenog prostora (2). Zavisno od vrednosti funkcije cilja u ovim tačkama identifikuju se privlačniji regioni u prostoru. Zatim se poziva mehanizam koji vrši pretragu u okviru ovih regiona. U *Genetskim algoritmima* ovaj mehanizam odgovara operacijama reprodukcije, ukrštanja i mutacije, dok se kod *dvofaznih metoda* pretraga obećavajućih regiona odvija u dve faze – prva je slučajno pretraživanje, a druga ili vertikalno penjanje (eng. hill-climbing) ili primena gradijentnog pretraživanja.

Slično, konstruišu se mehanizmi koji ohrabruju tačke da konvergiraju prema veoma privlačnim regionima, i obeshrabljuju tačke da idu ka manje privlačnim područjima. Ova ideja omogućava da se napravi analogija sa mehanizmom privlačenja i odbijanja koji se javlja u teoriji elektromegnetizma.

Kao što je slučaj kod elektromagnetizma, svaka tačka se može posmatrati kao naelektrisana čestica koja je puštena u prostor. U ovom pristupu, naelektrisanje posmatrane čestice je vrednost funkcije cilja u toj tački, koju pokušavamo da optimizujemo. Ovo naelektrisanje takođe određuje stepen privlačenja i odbijanja tačaka u datoj populaciji – što je veća vrednost funkcije cilja u pojedinačnoj tački, veći će biti stepen privlačenja i odbijanja od te tačke.

Kada izračunamo ova naelektrisanja, koristićemo ih za računanje smera u kom će se kretati svaka pojedinačna tačka u narednoj iteraciji. Ovaj smer će se računati kombinovanjem sila koju svaka od čestica u fiksnoj generaciji vrši na datu česticu. Kao i u elektromagnetizmu, ova sila se računa vektorskim sabiranjem sila ostalih čestica u generaciji, koje se računaju zasebno.

Ovde treba naglasiti da, iako je motivacija za ovaj pristup došla iz elektromagnetizma, postoje neke značajne razlike, koje ćemo jasno naglasiti kada u narednim poglavljima budemo uveli heuristike.

Konačno, slično kao kod hibridnih algoritama zasnovanih na populaciji [5], primenjivaćemo proceduru za lokalno pretraživanje kako bi popravili uočene vrednosti funkcije cilja u populaciji.

## Heuristike zasnovane na elektromagnetizmu

Posmatrajmo funkciju u obliku 1.1, sa sledećim parametrima:

– dimenzija problema

– donja granica u -toj dimenziji

– gornja granica u -toj dimenziji

– funkcija koja se minimizuje

U ovoj sekciji uvešćemo generalnu shemu heuristika zasnovanih na elektromagnetizmu i predstaviti njene podprocedure. Sekciju završavamo primerom koji vizuelno demonstrira ponašanje algoritma.

#### Generalna shema heuristika zasnovanih na EM

Heuristika se sastoji iz četiri faze. To su *inicijalizacija* algoritma, računanje vektorskog *zbira svih sila* koje deluju na česticu, računanje *pomeraja* u smeru delovanja sile i primena *pretraživanja u okoline* da bi iskoristili lokalni minimum. Generalna shema može biti data kao u Algorimu 1.

ALGORITAM 1. EM(, MAXITER, LOKITER, )

: broj čestica u generaciji

MAXITER: maksimalan broj iteracija (tj. generacija) u glavnom algoritmu

LOKITER: maksimalan broj iteracija lokalnog pretraživanja

: parametar koji se koristi za lokalno pretraživanje,

1. Inicijalizacija()
2. iteracija 1
3. **while** iteracija < MAXITER **do**
4. LokalnoPretraživanje(LOKITER, )
5. **F** IzračunajSile()
6. PomeriČestice(**F**)
7. iteracija iteracija + 1
8. **end while**

#### Inicijalizacija

Procedura **Inicijalizacija** se koristi da se nadje slučajan uzorak od tačaka iz dopuštenog domena 1.2, koji je -dimenzionalna hiper kocka. Za svaku koordinatu tačke se pretpostavlja da je uzeta iz uniformne raspodele između odgovarajućih donjih i gornjih granica. Kada su odabrane tačke iz prostora, izračunava se funkcija cilja u svakoj tački (Algoritam 2, linija 6). Procedura se završava identifikovanjem *trenutno najbolje* tačke čija je vrednost funkcije cilja minimalna.

ALGORITAM 2. Inicijalizacija()

1. **for** **to**  **do**
2. **for**  **to**  **do**

5. **end for**
6. Izračunaj
7. **end for**

#### Lokalno pretraživanje

Procedura **LokalnoPretraživanje** se koristi da bi se skupile lokalne informacije o tački . Parametri LOKITER i koji se prenose proceduri, predstavljaju respektivno broj iteracija odnosno množilac u lokalnom pretraživanju.

U njoj se prvo određuje maksimalna moguća dužina koraka (promenljiva *dužina*) na osnovu parametra (Algoritam 3, linija 2). Zatim se, za svako , popravka za traži koordinata po koordinata (linije 5-14). Za svaku koordinatu, tačka se dodeljuje privremenoj tački da bi čuvala inicijalne podatke. Zatim se uzima slučajan broj za dužinu koraka i tačka se pomera u tom pravcu. Ako tačka dobije vrednost tačke sa boljom vrednošću funkcije cilja u okviru LOKITER iteracija, tačka se zamenjuje sa i lokalno pretraživanje po koordinati se završava (linije 15-18). Na kraju, osvežava se *trenutno najbolja* tačka (linija 23).

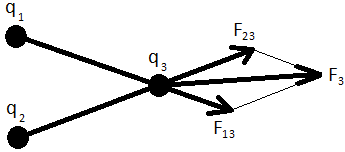
Ovo je jednostavno slučajno pretraživanje po linijama primenjeno koordinatu po koordinatu. Ova procedura ne zahteva informacije o gradijentu da bi izvela lokalno pretraživanje.

ALGORITAM 3. LokalnoPretraživanje(LOKITER, )

1. dužina
2. **for** **to** **do**
3. brojač 1
4. **for** **to** **do**
6. **if** **then**
8. **else**
10. **end if**
11. **while** brojač < LOKITER **do**


15. **if** **then**
17. brojač LOKITER – 1 // kraj za **while** i unutrašnji **for**
18. **end if**
19. brojač brojač + 1
20. **end** **while**
21. **end for**
22. **end for**

#### Računanje zbira sila



Slika 1: Princip superpozicije

Princip superpozicije iz teorije eletromagnetizma (slika 1) kaže da je sila koja deluje na jednu tačku od strane drugih tačaka inverzno proporcionalna distanci između tačaka i direktno proporcionalna proizvodu njihovih naelektrisanja.

Za razliku od stvarnog sveta, naelektrisanja čestica u ovoj heuristici nisu konstantna i menjaju se iz iteracije u iteraciju, tako da će se u svakoj iteraciji računati naelektrisanja tačaka i to na osnovu vrednosti funkcije cilja.

Naelektrisanje tačke se označava sa . Ono predstavlja snagu tačke sa kojom privlači odnosno odbija druge tačke. Ovo naelektrisanje se računa formulom:

|  |  |  |
| --- | --- | --- |
|  |  | 3 |

Na ovaj način, tačke sa boljom (manjom) vrednošću funkcije cilja sadrže veća naelektrisana. U formuli se vrši množenje razlomka sa dimenzijom problema , jer sa većom dimenzijom, broj tačaka u populaciji može postati veliki. Kao rezultat toga, vrednost razlomka može postati veoma mala što dovodi do problema potkoračenja na računarima prilikom računanja eksponenta.

Ovde je definisano naelektrisanje kao relativna efikasnost vrednosti funkcije cilja odgovarajuće tačke u populaciji. Jasno je da ovo nije ni jedinstven ni optimalan izbor računanja. Moguće je koristiti i alternativne načine računanja, na primer rangirati tačke prema vrednosti funkcije cilja. Eksperimenti su pokazali da je računanje naelektrisanja dato u jednačini (3) dovoljno dobro za ovo istraživanje.

Primetimo da je izraz kojim je definisan uvek pozitivan. Prema tome, u ovoj heuristici, za razliku od električnih naelektrisanja, nema pozitivno i negativno naelektrisanih čestica. Umesto toga, razlučujemo smer sile između dve tačke na osnovu vrednosti funkcije cilja u njima. Prema tome, zbir svih sila koje deluju na tačku se računa po formuli:

gde množilac upravo služi da odredi smer sile.

Као što se vidi u Algoritmu 4 (linija 7), tačka koja ima bolju vrednost funkcije cilja će privlačiti onu drugu. Isto tako, tačka koja ima lošiju vrednost funkcije cilja će odbijati drugu (linije 9-10). Pošto je minimum funkcije cilja u populaciji u tački , ona se ponašanja kao tačka apsolutnog privlačenja, tj. ona privlači sve druge tačke u populaciji.

Kada pažljivo posmatramo algoritam, možemo videti da određivanje smera pomoću sume svih sila oponaša statističku procenu gradijenta funkcije . Međutim, procena dobijena primenom heuristike se razlikuje, pošto smer zavisi od euklidskog rastojanja između dve tačke, tj. tačke koje postanu dovoljno bliske se mogu međusobno voditi u smeru koji se razlikuje od statističke procene gradijenta.

ALGORITAM 4. IzračunajSile(): **F**

1. **for** **to** **do**

4. **end** **for**
5. **for** **to** **do**
6. **for** **to** **do**
8. **end** **if**
9. **end** **if**

#### Pomeranje čestica

Posle računanja zbira sila , tačka se pomera u smeru sile koja na nju deluje, sa korakom koji se slučajno određuje. Ovaj korak se određuje kao proizvod maksimalnog dozvoljenog koraka u datom smeru i množioca za koji se pretpostavlja da je uzeta iz uniformne raspodele na intervalu . Naravno, postoji puno drugih raspodela koje su se mogle koristiti i koje bi možda poboljšale algoritam. Uniformna raspodela je korišćena samo zbog lakoće izračunavanja.

Slučajan množilac se koristi kako bi osigurali da čestice imaju ne nula verovatnocu da dođu u neobiđene regione na izračunatom pravcu. U jednačini (4), M je veličina koja predstavlja najveći mogući pomeraj u datom smeru, kojim će tačka ostati u dozvoljenim granicama. Prema tome, za najveće moguće , tačka će se pomeriti na granicu dozvoljenog prostora **Error! Reference source not found.**2 u tačku određenu jednačinom 1.4 u smeru vektora . Prema tome:

|  |  |  |
| --- | --- | --- |
|  |  | 4 |

U Algoritmu 5 je data procedura **PomeriČestice**. Primetimo da u njoj trenutno najbolju tačku ne pomeramo već je prenosimo u naredne generacije (linija 2). Zbog toga je moguće preskočiti računanje zbira svih sila za trenutno najbolju tačku, što bi dovelo do zanemarljivog ubrzanja.

ALGORITAM 5. PomeriČestice(**F**)

1. **for** **to** **do**
2. **if** **then**

5. **for** **to** **do**
6. **if** **then**
8. **else**
10. **end** **if**
11. **end** **for**
12. **end** **if**
13. **end** **for**

#### Kriterijum zaustavljanja

Kako metaheuristike ne garantuju tačnost rešenja, mora se definisati kriterijum zaustavljanja. Postoji puno načina za određivanje trenutka zaustavljanja rada programa. Za implementaciju programa koji primenjuje keširanje jedinki radi ubrzanja rada meteheuristika korišćen je maksimalan broj iteracija (tj. generacija) za koju je uzeta vrednost , tj. izvršavalo se 25 iteracija po dimenziji problema. Ispostavlja se da za to vreme algoritam već počne da konvergira.

Drugi kriterijum zaustavljanja koji bi mogao da se koristi je broj uzastopnih iteracija u kojima se nije menjala najbolja čestica. Drugim rečima, ako se najbolja čestica nije menjala određeni broj iteracija, algoritam se zaustavlja. Ovaj kriterijum može dovesti do zaustavljanja programa pre njegovog konvergiranja, ali sa druge strane može da smanji broj izračunavanja funkcije cilja.

Još jedan, često korišćeni algoritam zaustavljanja je kada vrednost funkcije cilja postane -blizu optimalnom rešenju. Naravno, za ovo je potrebno da unapred znamo globalni optimum. Sličan pristup bi bio da zaustavimo algoritam kada sve čestice budu na rastojanju manjem od od najbolje tačke i može se koristiti uz prikladnu metriku.

#### Primer

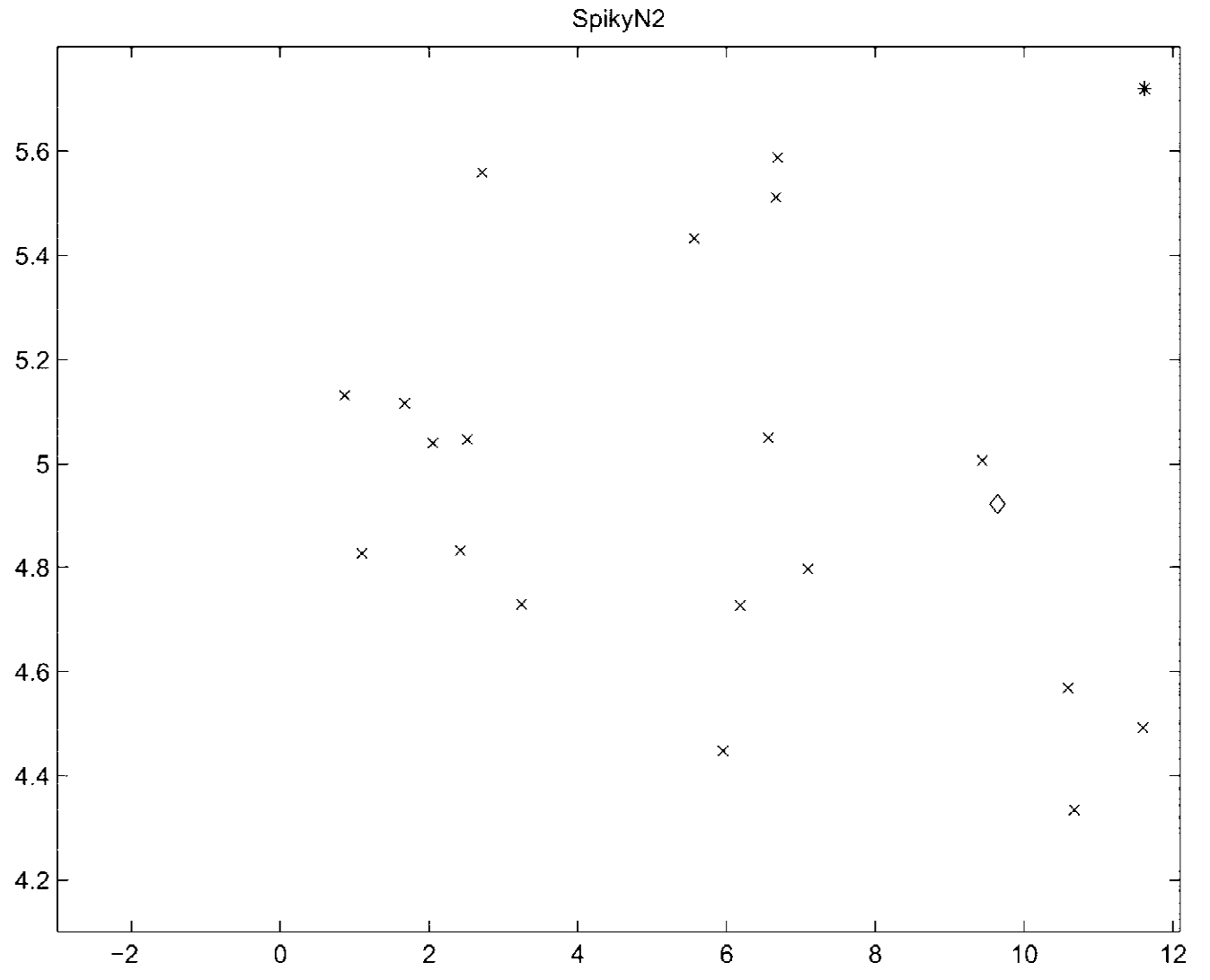
Ovaj primer demonstrira grafički prikaz tipičnog toka izvršavanja algoritma prilikom minimizacije Spajki (eng. Spiky) funkcije. Na slikama 2-5, \* predstavlja lokaciju globalnog optimuma, ◊ predstavlja trenutno najbolju tačku, a × preostale tačke u generaciji.

Slika 2 pokazuje lokaciju tačaka prilikom startovanja algoritma. One su slučajno odabrane. Trenutno najbolja tačka () je u početnom položaju daleko od globalnog optimuma.

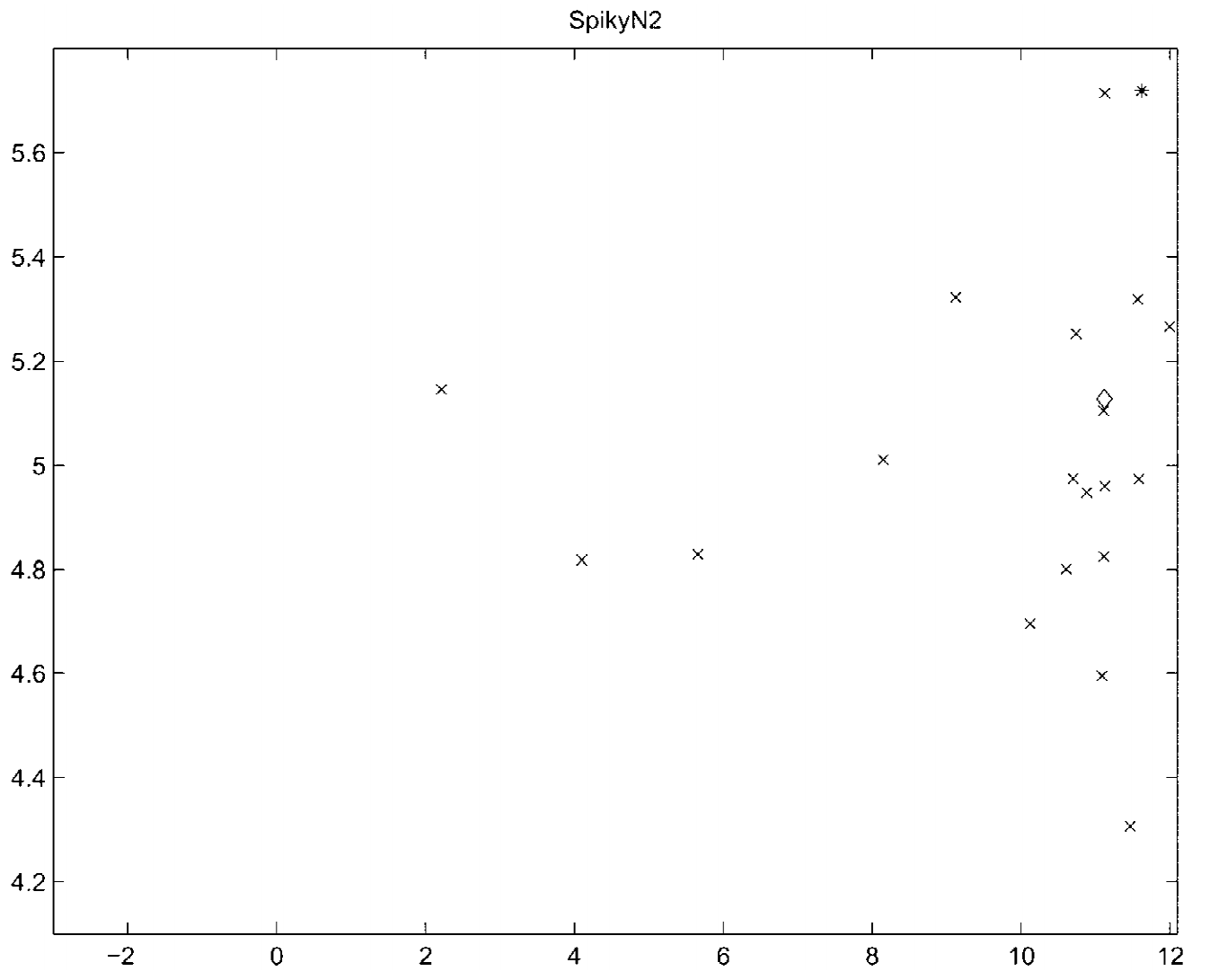
Tačke u populaciji se kreću prema regionu oko trenutno najbolje tačke. Neke od njih su odgurnute bliže globalnom optimumu (slika 3).

Na slici 4 je prikazano da jedna tačka pronalazi položaj čija je vrednost funkcije cilja bolja od vrednosti funkcije cilja trenutno najbolje tačke. Nova tačka postaje trenutno najbolja. Prema tome, tačke sada počinju da konvergiraju ka ovoj novoj tački .

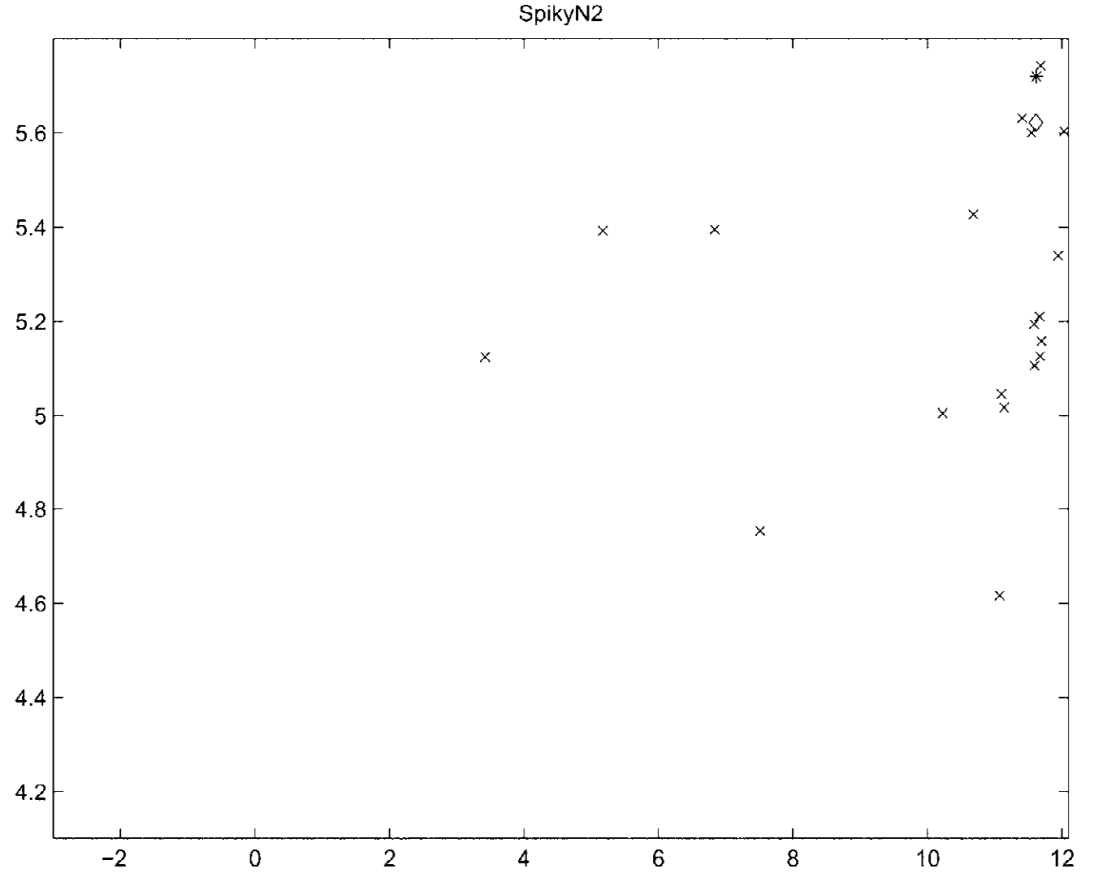
Slika 5 demonstrira da su tačke u populaciji usmerene prema regionu oko trenutno najbolje tačke. Posle ove iteracije, lokalizovan je globalni optimum.



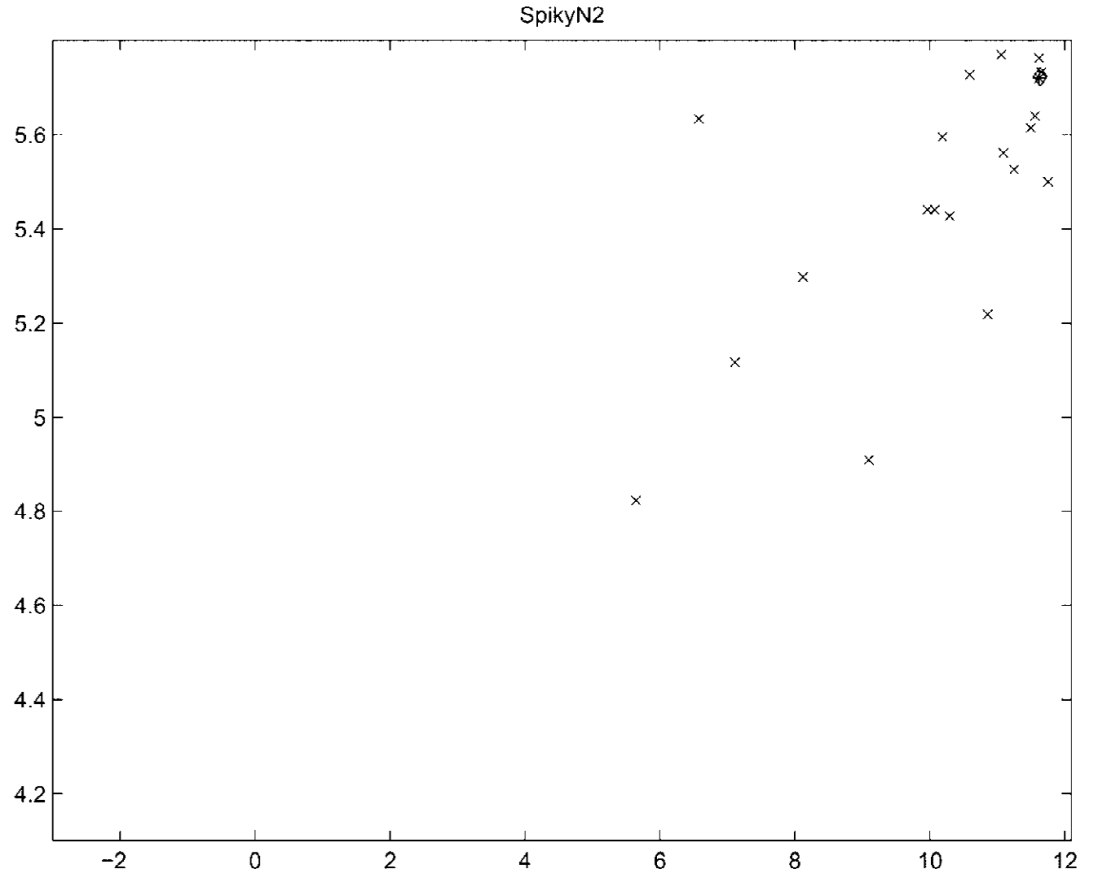
Slika 2: Početna pozicija čestica, odmah posle inicijalizacije



Slika 3: Tačke počinju da se međusobno privlače i odbijaju



Slika 4: Jedna od odgurnutih tačaka pronalazi bolji region i signalizira drugim



Slika 5: Najbolja tačka privlači druge i globalni optimum je pronađen

# Keširanje

Što se tiče heuristika zasnovanih na elektromagnetizmu, ovo je prva implementacija u kojoj je primenjeno keširanje kao način poboljšanja performansi. Keširanje se primenjuje zato što ne postoji nikakva garancija da se jedinke za koje je već izračunata funkcija cilja neće ponovo pojaviti u nekoj od narednih generacija. Zbog toga je ponekad pogodnije zapamtiti jedinke i njihove vrednosti u prvom pojavljivanju, kako bi se pri sledećim pojavljivanjima direktno očitala vrednost, umesto ponovnog računanja. Ovde treba istaći da keširanje ni na koji način ne utiče na tok izvršavanja heuristike, iz generacije u generaciju se dobijaju identične jedinke i na kraju, naravno, isto rešenje. Ono služi samo za smanjenje vremena izvršavanja heuristike.

## Tehnike keširanja

Keširanje se veoma često koristi u programiranju radi ubrzanja performansi. Neke ideje se uzimaju iz tradicionalnog keširanja u operativnim sistemima. Na primer, preuzeta tehnika izbacivanja najduže nekorišćenog člana (eng. Least Recently Used – LRU) se veoma lako implementira i daje dobre rezultate.

Sa druge strane potrebno je izabrati pogodnu strukturu podataka koja će minimizovati vreme osnovnih operacija keša.

## LRU primenjen na linearnoj strukturi

Keširanje kod heuristika za pretraživanje se prvi put javilo u radu [6] radi unapređenja prostog genetskog algoritma i dobijena su značajna ubrzanja izvršavanja (20-40%), ali su dalja istraživanja primene složenijih genetskih operatora i izvršavanja na problemima veće dimenzije dovela do značajno skromnijih rezultata. Pri tome je za keširanje korišćena linearna struktura podataka, koja ima linearno vreme pristupa. Međutim, osnovna shema primene keširanja je od tada ostala ista:

SHEMA 1. LRU shema izbacivanja blokova iz keša

1. **if** (Sadrži(keš\_memorija, jedinstveni\_kod\_jedinke)
2. DodeliVrednost(jedinka, keš\_blok)
3. **else**
4. {
5. IzračunajFitnes(jedinka);
6. **if** (Popunjena(keš\_memorija)
7. ObrišiNajdužeNekorišćeniBlok(keš\_memorija);
8. Dodaj(keš\_memorija, jedinka);
9. }
10. PostaviZaNajnoviji(keš\_blok);

Ukoliko se kešira veći broj jedinki, vreme pretrage u kešu postaje značajno, pa se pribegava komplikovanijim strukturama podataka.

## LRU primenjen na dvostrukoj heš tabeli

Kako bi se ubrzala operacija **Sadrži** koja je u gornjoj implementaciji zahtevala neprihvatljivo linearno vreme, dr Kratica je došao na ideju da za potrebe keširanja koristi posebnu strukturu podataka – dvostruku heš tabelu. Ona se sastoji iz dve heš tabele koje sadrže pokazivače na blokove keš memorije. Svaki blok keš memorije se može naći pretragom po proizvoljnoj heš tabeli.

Kao što se vidi u Shemi 1, najvažnije komponente programa za keširanje u oblasti optimizacije zasnovanih na LRU strategiji su sledeće procedure:

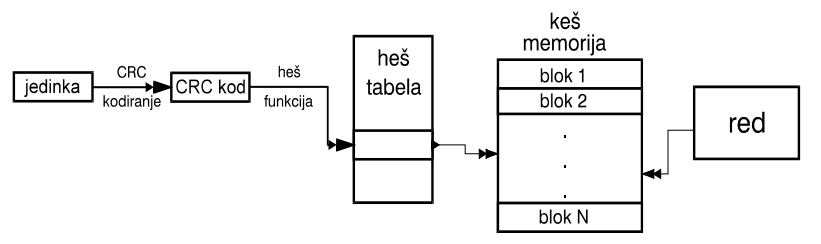
* **Sadrži** – pretražuje keš memoriju i ispituje da li ona sadrži dati genetski kod jedinke, u slučaju da sadrži, vraća vrednost funkcije cilja za datu jedinku
* **Dodaj** – gde se tekuća jedinka smešta u slobodan blok keš memorije zajedno sa vrednošću funkcije cilja
* **Obriši** – ona izbacuje iz keš memorije jedinku koja najduže vremena nije korišćena i time oslobađa mesto za smeštanje nove jedinke.

Prva heš tabela se koristi za implementaciju procedura **Sadrži** i **Dodaj**, dok je druga keš tabela isključivo potrebna radi implementacije procedure **Obriši**.

Na ovaj način dobijena je vrlo efikasna implementacija keš memorije koja koristi LRU strategiju izbacivanja. U njoj su operacije **Sadrži**, **Dodaj** i **Obriši** implementirane u konstantnom vremenu, što daje dobru osnovu za primenu keširanja u raznim algoritmima.

## Implementacija pomoću heš-red strukture

Heš-red struktura predstavlja dodatni korak ka ubrzanju performansi keširanja. Kod dvostruke heš tabele operacije **Sadrži**, **Dodaj** i **Obriši** implementirane na upravo opisan način zahtevaju konstantno vreme izvršavanja, međutim, ispostavlja se da je moguće dodatno poboljšanje, kojim se ubrzava izvršavanje ovih procedura. Ovo ubrzanje je u praksi izuzetno bitno, jer se keširanje poziva veoma često. To je postignuto zamenom druge heš tabele redom (queue) [7]. Time su procedure **Dodaj** i **Obriši** nešto uprošćenije i imaju manje osnovnih operacija, što dovodi do ubrzanja. Iako poboljšanje vremena izbršavanja (u odnosu na dvostuku heš tabelu) nije spektakularno, razlike se mogu uočiti. Na slici 6 je dat šematski prikaz ove strukture podataka.



Slika 6: Heš-red struktura

## Heš funkcija

Prilikom izbora heš funkcije najbitnije je posmatrati koliko ravnomerno raspoređuje blokove u heš memoriju. Ovde je za heš funkciju izabrana metoda množenja (multiplication method). Iako se u teoriji [8] kao nešto bolja prikazuje univerzalna heš funkcija (universal hashing), u našem slučaju se ne pokazuju nikakve razlike. Stoga je izabrana metoda množenja, koja je jednostavnija za implementaciju.

Postavlja se pitanje zbog čega je i ovaj metod bio dovoljno dobar. Razlog je što argument heš funkcije, u ovom slučaju, nije genetski kod jedinke (koji bi posle nekoliko generacija mogli biti veoma slični), već njegova CRC vrednost. Poznato je da pri CRC preslikavanju slični argumenti daju potpuno različite vrednost. Zbog toga su CRC vrednosti već prilično ravnomerno raspoređene u 32-bitnom opsegu, pa je tako zadatak heš funkcije za što ravnomernijim rasporedom blokova u velikoj meri olakšan.

Kod metode množenja računanje heš funkcije je vrlo jednostavno, dovoljno je izkoristiti formulu:

|  |  |  |
| --- | --- | --- |
|  |  | 5 |

Konstanta je veličina heš tabele, je argument heš funkcije, a je unapred izabrana realna konstanta, pri čemu funkcija računa celobrojni, a razlomljeni deo broja. U ovoj implementaciji je korišćeno .

## Rešavanje kolizije

Prilikom keširanja moguće je da različite jedinke imaju iste vrednosti heš funkcije. Ova pojava se naziva kolizija. Takođe, moguće je napraviti implementaciju u kojoj je gornja granica veličine keša veća od kardinalnog broja kodomena heš funkcije. U takvim situacijama kolizija je neminovna, čim se u keš unese dovoljan broj elemenata.

Ovde je za rešavanje kolizije korišćeno otvoreno heširanje (eng. collision resolution by chaining). Kod njega se kolizija rešava formiranjem povezane liste za svaku poziciju heš tabele i time se može pojaviti više jedinki sa istom vrednošću heš funkcije. Pošto je u implementaciji heš tabela spregnuta za redom, umesto obične (jednostruke) povezane liste, za svaku poziciju heš tabele je neophodno da se koristi dvostruko povezana lista. Na ovaj način je omogućeno brisanje iz liste proizvoljnog elementa.

Kao što se može videti na slici 6, pretraživanje heš tabele se odvija u tri faze. Prvo se za datu jedinku izračunava CRC vrednost. U drugoj fazi se izračunava vrednost heš funkcije za dobijenu CRC vrednost i na osnovu nje se određuje pokazivač na povezanu listu. Na kraju se odgovarajuća povezana lista pretražuje sekvencijalno, do pronalaženja traženog elementa. Pošto se gornja granica veličine keša i kardinalni broj kodomena keš funkcije mogu nezavisno menjati, moguće ih je izabrati tako da njihov količnik uzme povoljnu vrednost i time dobijamo da dvostuko povezane liste budu kratke i brzo se pretražuju.

Kada treba da se unese novi element u dvostuko povezanu listu, dovoljno je da se unese na prvo mesto, a stari elementi pomere za jedno mesto. Ovo se postiže jednostavnim ulančavanjem koje se izvodi samo na početku liste, što je veoma efikasno, a i lako se implementira. Izbacivanja elementa iz heš tabele je iste složenosti kao pretraživanje, uz dodatno postavljanje pokazivača u odgovarajućoj dvostruko povezanoj listi.

Dodajmo još da, osim potrebe za realizacijom preko dvostruko povezane liste, nema nikakvih drugih uticaja činjenica da se razrešava pojava kolizije na heš-red strukturi, umesto na običnoj heš tabeli.

## Dobijanje CRC kodova

Svaka implementacija keša sadrži neku vrstu rešavanja kolizija. Prilikom toga, za datu jedinku, potrebno je vršiti poređenje sa raznim drugim jedinkama koje se nalaze na poziciji u kešu koji odgovara datoj jedinki. Prva jedinka pripada jedinki za koju želimo da saznamo vrednost funkcije cilja, a druga je zapamćena u određenom bloku keš memorije.

Poređenje reči je u većini primena efikasno, jer se one najčešće razlikuju već u nekom od prvih nekoliko slova, pa se poređenje brzo završava. Međutim, genetski kodovi imaju nezgodnu osobinu da se posle nekoliko generacija grupišu u određeni region pretrage pa postanu vrlo slični. O ovom slučaju operacija poređenja genetskih kodova može imati značajan uticaj na performanse.

Situacija je još lošija ukoliko su genetski kodovi relativno veće dužine pa tada direktno poređenje može biti veoma skupo. Tada je pogodnije poreženje genetskih kodova preko njihovih CRC crednosti, Pri tome je neophodno izračunavanje CRC vrednosti za svaku jedinku u populaciji, ali se to izvršava samo jednom i to na početku pretraživanja. Posle toga je poreženje vrlo efikasno, jer se genetski kodovi dve jedinke ne porede direktno, već se prvo uporede njihove CRC vrednosti. Ako su one različite, genetski kodovi se sigurno razlikuju. Direktno poreženje celih gentskih kodova je potrebno samo kada su CRC vrednosti jednake, što se u praksi dešava vrlo retko.

Kada se izračuna vrednost CRC vrednost genetskog koda jedinke, heš funkcija se primenjuje direktno na tu CRC vrednost, a ne na ceo genetski kod. Time, ne samo što je ubrzano u uprošćeno računanje heš funkcije, već se jedinke ravnomernije smeštaju kao blokovi u keš memoriju.

U implementaciji koja se opisuje su korišćene 32-bitne CRC vrednosti, koje su dobijene primenom algoritma zasnovanog na čitanju gotovih kooficijenata (eng. table-driven algorithm), koji je razvijen u radu [9]. Pošto su koeficijenti uvek isti, ima ih relativno malo (256), a njihovo izračunavanje je vremenski relativno zahtevno, oni se računaju unapred pomoćnim programom, koji ih smešta u posebnu datoteku “*CRC.DAT*”. Pri inicijalizaciji se samo dati koeficijenti porčitaju iz datoteke I smeste u niz *crctable*. Na osnovu koeficijenata iz datoh niza se vrlo efikasno računa CRC vrednost, kao što se može videti u Algoritmu 6 koji sadrži odgovarajući programski kod u jeziku C.

Algoritam 6

unsigned long INIT = 0xFFFFFFFF;

unsigned long XOROUT = 0xFFFFFFFF;

unsigned long FindCRC(int len, char \*msg)

{

int l;

unsigned long crc = INIT;

l = len;

**while** (l--)

crc = (crc >> 8) ^ crctable[(crc ^ \*msg++) & 0xFFL];

crc = crc ^ XOROUT;

**return** crc;

}

## Dodavanje elemenata u heš tabelu

Prilikom implementacije procedure **Sadrži**, nametnula se ideja da se u jednoj heš tabeli memorišu pokazivači na jedinke, po heš funkciji koja zavisi od genetskog koda svake jedinke. Kao što je rečeno, heš funkcija se primenjuje na 32-bitni CRC genetskog koda jedinke, čime je smeštanje blokova u heš tabelu ravnomernije. Pri tome je, takođe, ubrzana procedura poređenja dve jedinke, što dodatno poboljšava performanse cele implementacije.

Procedura **Dodaj** je veoma slična proceduri **Sadrži**, jer se na osnovu genetskog koda jedinke računa CRC, a nakon toga heš funkcija računa poziciju gde treba umetnuti datu jedinku.

## Korišćenje reda

Razlog za korišćenje reda je procedura **Obriši**. Jedino ona se ne može implementirati pomoću prve heš tabele. Njena funkcija je da oslobodi memorijski prostor prilikom smeštanja nove jedinke ukoliko je keš memorija potpuno popunjena. U prvim implementacijama je korišćena druga heš tabela, koja numerički vrednuje vreme poslednjeg pojavljivanja date jedinke. Kasnije je druga heš tabela zamenje red strukturom, čime je cela implementacija pojednostavljena.

# Maksimalni problem srednjih elemenata

## Uvod

Problem srednjih elemenata PSE (eng. Betweenness problem) je veoma poznat problem kombinatorne optimizacije. Za dati skup od elemenata i dati skup trojki , problem srednjih veličina je da se utvrdi linearno uređenje elemenata skupa , takvo da trojke skupa zadovoljavaju uslov srednjih elemenata, tj. element mora da bude između elemenata i . Maksimalni problem srednjih elemenata MPSE (Maximum Betweenness Problem), koji se ovde izučava, traži maksimalni broj elemenata skupa koji zadovoljavaju dati uslov.

Pošto je skup konačan, elementi mu se mogu indeksirati sa , gde je . Ovo indeksiranje se može posmatrati i kao bijektivna funkcija između skupa i skupa koje uvodi linearno uređenje elemenata skupa : ako su , onda je “pre” ako i samo ako je . Prema tome, bez gubljenja opštosti, možemo pretpostaviti da je skup od koga polazimo ustvari skup .

Sada je jasno da za rešavanje MPSE možemo posmatrati bijektivne funkcije →, kojima je predstavljeno potencijalno rešenje početnog problema. Naravno ove funkcije su ustvari permutacije skupa . Dakle, MPSE traži pronalaženje permutacije skupa koja maksimizuje broj trojki , takvih da važi ili uslov ili uslov .

**Primer 1.** Neka je , zatim , a kolekcija sadrži 6 trojki i . Optimalno rešenje, koje je dobijeno pomoću CPLEX-a je permutacija . Vrednost koju smo računali je 6, što znači da su svi uslovi u zadovoljeni. Lako se vidi da se ovom permutacijom elementi skupa slikaju u trojke: i .

Odlučiva verzija PSE je da se razluči da li svi uslovi skupa mogu biti istovremeno zadovoljeni primenom neke permutacije. U [10] je dokazano da je ova verzija, a samim tim i MPSE, NP teška.

MPSE i druge varijante PSE pripadaju klasi problema diskretne optimizacije koji imaju puno primena u različitim poljima nauke. Na primer, MPSE je značajan jer se koristi za rešavanje nekih problema fizičkog mapiranja u molekularnoj biologiji [11]. Tokom radijacionih hibridnih eksperimenata, X-zraci se koriste da izdele hromozome. U slučaju kada su markeri na hromozomima međusobno udaljeniji verovatnoća da će data doza X-zraka podeliti hromozom između njih veća. Procenom frekvencije tačaka podele i time distance između markera moguće je odrediti njihov redosled na način sličan meotičnom mapiranju. U ovom kontekstu, poboljšanje radiacionog eksperimenta se postiže pronalaženjem linearnog uređenja markera koje maksimizuje broj zadovoljenih uslova. Softverski paket RHMAPPER [12, 13] koristi ovaj pristup za pronalaženje frejmvork markera, koristeći dva pohlepna algoritma za rešavanje PSE. Detaljna analiza radijacionih hibridnih eksperimenata je van opsega ovog master rada i mogu se naći u [14, 15, 16].

Broj promašaja, tj. broj trojki koje ne zadovoljavaju uslov za datu permutaciju, može biti penalizovan dodavanjem težina na svaki uslov. Time se početni problem svodi na pronalaženje minimuma ove sume i taj problem nazivamo Težinski problem srednjih elemenata TPSE. Ovaj problem se javlja u računarskoj biologiji [17], preciznije kod problema fizičkog mapiranja sa krajnjim sondama. Za svaki element , iz datog skupa od “klonova”, dve takozvane krajnje sonde i su pridružene. Za svaki par klona i sonde , uslov srednjih elemenata ili važi ili ne važi. Ako uslov ne važi, onda mu se dodeljuje težina i izračunava suma koja pokazuje koliko je rešenje dobro. Sada se primenjuje TPSE za pronalaženje linearnog uređenja sondi koji minimizuje izračunatu sumu.

Opisani problemi su NP-teški, što pre svega znači da nisu poznati algoritmi koji bi ih rešavali u polinomnom vremenu. Želja za razvojem ovakvih algoritama je opravdana, jer algoritmi koji imaju eksponencijalnu složenost, pri povećanju dimenzije problema, vrlo brzo zahtevaju neprihvatljivo veliko vreme izvršavanja.

## Matematička formulacija

Neka je broj elemenata u konačnom skupu . Kao što je već pomenuto, bez gubljenja opštosti možemo pretpostaviti da je . Neka je kolekcija od trojki iz skupa i neka je -ta trojka iz kolekcije označena sa , za . Neka je još realan broj iz skupa .

Neka je unapred data bijekcija . Uvedimo četiri skupa promenljivih: promenljive definišimo sa:

|  |  |  |
| --- | --- | --- |
|  |  | 7 |

Za svako , binarne promenljive , i definišimo sa:

Zbog načina na koji su definisane promenljive i , one ne mogu istovremeno biti jednake , za fiksirano . Sada MILP model se formuliše sa:

|  |  |  |
| --- | --- | --- |
|  |  | 8 |

pri uslovima:

|  |  |  |
| --- | --- | --- |
|  |  | 9 |
|  |  | 10 |
|  |  | 11 |
|  |  | 12 |
|  |  | 13 |
|  |  | 14 |
|  |  | 15 |
|  |  | 16 |

Kao što se može videti, javlja se realnih promenljivih (promenljive ) i binarnih promenljivih (promenljive , i ). Imamo uslova za ove promenljive. Parametar je uveden kako bi razlomak bio veći od greške zaokruživanja i prema tome, mora da bude veći od nule. Gornja granica za je 1, sa ciljem da se osigura da će sistem jednačina (9) – (16) uvek imati neprazan skup rešenja. Sada se rešava pri uslovima (9) – (16). Detaljan dokaz se može videti u [18].

## Reprezentacija jedinki i funkcije cilja

Sa ciljem da zadržimo efikasnost heuristike zasnovane na elektromagnetizmu (EM heuristike), izbor pogodne reprezentacije za kandidate rešenja igra ključnu ulogu. U slučaju MPSE, svaka tačka (EM tačka) u skupu rešenja je povezana sa jednom permutacijom skupa . Ova permutacija se koristi za izračunavanje broja zadovoljenih uslova.

Predstavimo svaku EM tačku kao -dimenzionalni vektor čije koordinate uzimaju realne vrednosti iz intervala . Označimo taj vektor sa , , . Za datu takvu tačku , svakom elementu skupa odgovara jedna koordinata te tačke i obrnuto. Tačka će predstavljati uređenje na sledeći način: ako su i dva elementa skupa , onda je ako i samo ako je . Sada možemo definisati i funkciju cilja direktno na tački koja će računati broj zadovoljenih uslova:

U slučaju da su sve koordinate različite, onda će uređenje koje je dobijeno iz tačke indukovati bijektivnu funkciju , što je ustvari permutacija skupa . Fukcija se određuje sortiranjem indeksa skupa po kriterijumu datom linearnim uređenjem: ako su i dva indeksa, onda je ako i samo ako je . Ako je niz indeksa sortiran po pomenutom kriterijumu, onda je permutacija data sa , . Primetimo da za računanje funkcije cilja , nije potrebno računanje permutacije.

Sa druge strane, treba primetiti da je moguće da se tokom izvršavanja algoritma pojavi situacija za neke i . To bi značilo da za odgovarajuću funkciju ne važi ni ni . U radu [18] je dokazano da za svako takvo uređenje postoji “1-1” funkcija takva da je , gde su i funkcije cilja za i . Dokaz se zasniva na činjenici da svi uslovi koji sadrže i indeks i indeks sigurno nisu zadovoljeni. Prema tome, funkcija cilja primenjena na ovo uređenje vraća manju (ili u najgorem slučaju jednaku) vrednost nego u slučaju kada je , zadržavajući redosled ostalih elemenata u uređenju. Kao zaključak, možemo reći da se ovakve situacije mogu pojaviti, ali će ih proces optimizacije, koga vodi funkcija cilja, odbaciti.

**Primer 2.** Neka je i = (0.98, 0.86, 0.37, 0.78). Onda je odgovarajuće uređenje dato sa .

## Motivacija za ovakvu reprezentaciju

Motivacija za ovaj pristup enkodiranja se opravdava sledećim razmišljanjem: za vreme izvršavanja EM algoritma, tačke se neprestalno pomeraju iz jedne tačke u drugu zavisno od sila koje na njih deluju. Pošto postoji “više” tačaka u neprekidnom prostoru nego u diskretnom, jedno uređenje se neće transformisati u drugo prilikom svakog pomeranja. Šta više, vrednost funkcije cilja se neće menjati pri (dovoljno) malim pomeranjima. Predloženo enkodiranje zadovoljava ovaj uslov, jer ako je vektor (ili drugim rečima EM tačka) u neprekidnom prostoru, onda će svaki vektor koji zadovoljava uslov za sve predstavljati isto uređenje i samim tim će im odgovarati ista vrednost funkcije cilja. Predstavimo ovaj pristup slikom 7, gde je prikazana samo dvodimenzionalna restrikcija celog prostora. Bez gubljenja opštosti, pretpostavimo da je vektor već sortiran i da predstavlja neko uređenje. Označena površina (trougao) predstavlja skup tačaka za koje je uslov zadovoljen. I ne samo on, za takve tačke će važiti i što znači da vektoru odgovara ista vrednost funkcije cilja. Prema tome, kao što je ilustrovano na slici 7, sva pomeranja tačke , unutar označenog trougla neće menjati vrednost funkcije cilja.

|  |
| --- |
| C:\Users\a-urosr\Desktop\master\EM tacka.png |
| Slika 7: dvodimenziona restrikcija  pretraživačkog prostora |

Na isti način moguće je zatim pomerati i druge parove uzastopnih koordinata i pri tome će EM tačke koje se tako dobiju imati istu vrednost funkcije cilja kao i početna tačka. Ovaj pristup garantuje da će ceo neprekidni prostor biti pokriven procesom pretraživanja i da će susedne tačke odgovarati istim vrednostima funkcije cilja.

# Rezultati

Sva testiranja su obavljena u sledećem okruženju:

* Računar sa Intel Xeon CPU E5410 @ 2.33 GHz (2 procesora, svaki po 4 jezgra)
* Operativni sistem je Windows Server 2008 R2 Enterprise (64-bitna verzija)
* Program je pisan u programskom jeziku C

Za testiranje je korišćen skup test primera... (qqq ovo ne znam koji su test primeri. Mislim da su to benchmark instance koje imaju ime?)

Opis kolona u tabelama koje slede:

|  |  |
| --- | --- |
| Ime | - ime instance |
| Broj | - broj ponavljanja |
| Min | - minimalna vrednost rezultata za različite vrednosti semena (eng. seed) |
| Max | - maksimalna vrednost rezultata za različite vrednosti semena |
| Sr | - srednja vrednost rezultata za različite vrednosti semena |
|  | - prosečno vreme izvršavanja programa |
|  | - prosečno vreme izvršavanja programa do pronalaska najbolje tačke |
|  | - prosečan broj iteracija (generacija) programa |
|  | - prosečan broj iteracija (generacija) programa do pronalaska najbolje tačke |
|  | - prosečan broj lokalnih iteracija programa |
|  | - prosečan broj lokalnih iteracija programa do pronalaska najbolje tačke |
| Cache\_total | - prosečan broj traženja u kešu |
| Cache\_hit | - prosečan broj pronalaska tražene vrednosti u kešu |
| Cache\_hit\_ratio | - prosečan procenat pronalaska tražene vrednosti u kešu |
| CHR\_Dev | - prosečna standardna devijacija broja pronalazaka vrednosti u kešu |

Pošto se početna populacija jedinki kreira pomoću slučajnih brojeva tok izvršavanja programa zavisi i od tih slučajnih brojeva. Da bi se dobio deterministički program uvodi se dodatni ulazni parametar – seme za generator slučajnih brojeva. Pored toga, za svaku instancu problema se izvodi veći broj eksperimenata sa različitim vrednostima ulaznog semena. U ovom testiranju je korišćeno različitih vrednosti što se može videti i u sledećim tabelama.

Rezultati testiranja bez keširanja:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ime | Broj | Min | Max | Sr |  |  |  |  |  |  |
| mbp-10-20a | 25 | 16 | 16 | 16 | 0.10584 | 0.0012 | 100.2 | 0.2 | 11642.8 | 132.3 |
| mbp-10-50a | 25 | 29 | 29 | 29 | 0.20332 | 0.003 | 100.5 | 0.5 | 12505.1 | 181.6 |
| mbp-10-100a | 25 | 50 | 50 | 50 | 0.40396 | 0.00996 | 101.5 | 1.5 | 13843 | 335.4 |
| mbp-11-20a | 25 | 14 | 14 | 14 | 0.09248 | 0.001 | 100 | 0 | 10697.6 | 105 |
| mbp-11-50a | 25 | 33 | 33 | 33 | 0.25468 | 0.00988 | 103 | 3 | 16960.1 | 648.2 |
| mbp-11-100a | 25 | 55 | 55 | 55 | 0.48292 | 0.00732 | 100.5 | 0.5 | 17151.2 | 255.8 |
| mbp-12-20a | 25 | 17 | 17 | 17 | 0.10504 | 0.00264 | 101.6 | 1.6 | 12788.7 | 332.1 |
| mbp-12-50a | 25 | 34 | 34 | 34 | 0.28484 | 0.02028 | 106.7 | 6.7 | 17105.8 | 1261.8 |
| mbp-12-100a | 25 | 56 | 56 | 56 | 0.53228 | 0.0178 | 102.4 | 2.4 | 18525.2 | 613.3 |
| mbp-15-30a | 25 | 25 | 26 | 25.2 | 0.20168 | 0.02892 | 115.6 | 15.6 | 19804.6 | 2859 |
| mbp-15-70a | 25 | 46 | 46 | 46 | 0.46944 | 0.01812 | 102.9 | 2.9 | 22632.2 | 861.2 |
| mbp-15-200a | 25 | 106 | 106 | 106 | 1.74212 | 0.19832 | 111.6 | 11.6 | 28843.1 | 3289.3 |
| mbp-20-40a | 25 | 36 | 37 | 36.56 | 0.4822 | 0.15624 | 146.7 | 46.7 | 36241.4 | 11699.3 |
| mbp-20-100a | 25 | 66 | 68 | 67.04 | 1.41096 | 0.42208 | 141.2 | 41.2 | 44001.2 | 13124.9 |
| mbp-20-200a | 25 | 115 | 117 | 116.32 | 3.69876 | 1.2066 | 146.8 | 46.8 | 56240.6 | 18382.6 |
| mbp-30-60a | 25 | 52 | 54 | 53.04 | 1.19012 | 0.36296 | 142.6 | 42.6 | 55997.6 | 17072.2 |
| mbp-30-150a | 25 | 105 | 109 | 107.2 | 4.93824 | 2.04272 | 169.5 | 69.5 | 90902.6 | 37730 |
| mbp-30-300a | 25 | 178 | 185 | 180.08 | 12.13372 | 4.24608 | 153.4 | 53.4 | 102617.2 | 36175.8 |
| mbp-50-100 | 25 | 84 | 88 | 85.76 | 1.12908 | 0.40192 | 37.2 | 12.2 | 25975.8 | 9243.9 |
| mbp-50-200 | 25 | 144 | 154 | 147.92 | 3.73792 | 1.70664 | 44.1 | 19.1 | 40424.4 | 18408.2 |
| mbp-50-400 | 25 | 248 | 259 | 252.32 | 9.33396 | 3.44612 | 38.1 | 13.1 | 45416.4 | 16774.8 |
| mbp-50-1000 | 25 | 518 | 527 | 522.72 | 86.2492 | 35.15368 | 40.4 | 15.4 | 148488.8 | 60368.5 |

Prilikom izrade rada, testiranje je vršeno na različitim implementacijama keša. Najbolje vreme izvršavanja je dobijeno korišćenjem heš-red strukture. Rezultati tog testiranja su:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ime | Broj | Min | Max | Sr |  |  |  |  |  |  |
| mbp-10-20a | 25 | 16 | 16 | 16 | 0.12096 | 0.00268 | 100.2 | 0.2 | 11642.8 | 132.3 |
| mbp-10-50a | 25 | 29 | 29 | 29 | 0.21312 | 0.00464 | 100.5 | 0.5 | 12505.1 | 181.6 |
| mbp-10-100a | 25 | 50 | 50 | 50 | 0.41568 | 0.01212 | 101.5 | 1.5 | 13843 | 335.4 |
| mbp-11-20a | 25 | 14 | 14 | 14 | 0.11228 | 0.00216 | 100 | 0 | 10697.6 | 105 |
| mbp-11-50a | 25 | 33 | 33 | 33 | 0.27628 | 0.01256 | 103 | 3 | 16960.1 | 648.2 |
| mbp-11-100a | 25 | 55 | 55 | 55 | 0.49336 | 0.00904 | 100.5 | 0.5 | 17151.2 | 255.8 |
| mbp-12-20a | 25 | 17 | 17 | 17 | 0.13664 | 0.00488 | 101.6 | 1.6 | 12788.7 | 332.1 |
| mbp-12-50a | 25 | 34 | 34 | 34 | 0.29984 | 0.02368 | 106.7 | 6.7 | 17105.8 | 1261.8 |
| mbp-12-100a | 25 | 56 | 56 | 56 | 0.56024 | 0.02044 | 102.4 | 2.4 | 18525.2 | 613.3 |
| mbp-15-30a | 25 | 25 | 26 | 25.2 | 0.27344 | 0.03996 | 115.6 | 15.6 | 19804.6 | 2859 |
| mbp-15-70a | 25 | 46 | 46 | 46 | 0.54508 | 0.0226 | 102.9 | 2.9 | 22632.2 | 861.2 |
| mbp-15-200a | 25 | 106 | 106 | 106 | 1.80712 | 0.20688 | 111.6 | 11.6 | 28843.1 | 3289.3 |
| mbp-20-40a | 25 | 36 | 37 | 36.56 | 0.68732 | 0.21792 | 146.7 | 46.7 | 36241.4 | 11699.3 |
| mbp-20-100a | 25 | 66 | 68 | 67.04 | 1.6462 | 0.48856 | 141.2 | 41.2 | 44001.2 | 13124.9 |
| mbp-20-200a | 25 | 115 | 117 | 116.32 | 3.98528 | 1.28828 | 146.8 | 46.8 | 56240.6 | 18382.6 |
| mbp-30-60a | 25 | 52 | 54 | 53.04 | 1.7752 | 0.52408 | 142.6 | 42.6 | 55997.6 | 17072.2 |
| mbp-30-150a | 25 | 105 | 109 | 107.2 | 6.10632 | 2.44792 | 169.5 | 69.5 | 90902.6 | 37730 |
| mbp-30-300a | 25 | 178 | 185 | 180.08 | 13.57772 | 4.65276 | 153.4 | 53.4 | 102617.2 | 36175.8 |
| mbp-50-100 | 25 | 84 | 88 | 85.76 | 1.7452 | 0.61704 | 37.2 | 12.2 | 25975.8 | 9243.9 |
| mbp-50-200 | 25 | 144 | 154 | 147.92 | 4.80124 | 2.1674 | 44.1 | 19.1 | 40424.4 | 18408.2 |
| mbp-50-400 | 25 | 248 | 259 | 252.32 | 10.6766 | 3.9062 | 38.1 | 13.1 | 45416.4 | 16774.8 |
| mbp-50-1000 | 25 | 518 | 527 | 522.72 | 92.66968 | 37.16492 | 40.4 | 15.4 | 148488.8 | 60368.5 |

Statistika keširanih jedinki:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ime | Broj | Cache\_tot | Cache\_hit | Cache\_hit\_ratio | CHR\_Dev |
| mbp-10-20a | 25 | 196793 | 17321 | 8.8135 | 0.8194 |
| mbp-10-50a | 25 | 208816 | 18358 | 8.7948 | 0.6127 |
| mbp-10-100a | 25 | 226428 | 18283 | 8.0743 | 0.3783 |
| mbp-11-20a | 25 | 205905 | 9109 | 4.4233 | 0.2925 |
| mbp-11-50a | 25 | 327464 | 20591 | 6.2888 | 0.2096 |
| mbp-11-100a | 25 | 322181 | 20778 | 6.4536 | 0.5528 |
| mbp-12-20a | 25 | 288232 | 10301 | 3.575 | 0.1146 |
| mbp-12-50a | 25 | 387940 | 17306 | 4.4671 | 0.386 |
| mbp-12-100a | 25 | 405331 | 18687 | 4.6175 | 0.3195 |
| mbp-15-30a | 25 | 616764 | 14893 | 2.4146 | 0.0532 |
| mbp-15-70a | 25 | 711340 | 17487 | 2.459 | 0.0496 |
| mbp-15-200a | 25 | 910877 | 23092 | 2.535 | 0.0873 |
| mbp-20-40a | 25 | 1813109 | 27403 | 1.5103 | 0.0191 |
| mbp-20-100a | 25 | 2219179 | 33542 | 1.5118 | 0.0273 |
| mbp-20-200a | 25 | 2856755 | 43317 | 1.5163 | 0.0238 |
| mbp-30-60a | 25 | 5239409 | 42943 | 0.8199 | 0.0105 |
| mbp-30-150a | 25 | 8908919 | 69579 | 0.7821 | 0.0093 |
| mbp-30-300a | 25 | 10367540 | 78808 | 0.7592 | 0.0109 |
| mbp-50-100 | 25 | 5637110 | 20155 | 0.3576 | 0.0035 |
| mbp-50-200 | 25 | 9363076 | 30795 | 0.3288 | 0.004 |
| mbp-50-400 | 25 | 10885292 | 34257 | 0.3146 | 0.0029 |
| mbp-50-1000 | 25 | 36257179 | 106833 | 0.2949 | 0.0013 |

Odnos dužine izvršavanja programa bez keša i sa kešom:

# Kratka analiza dobijenih rezultata

Iz poslednjeg grafika[[1]](#footnote-1) se vidi da nije došlo do ubrzanja prilikom korišćenja keša. Postoji dva razloga za to. Prvo, procenat keš pogodaka nije bio dovoljno veliki da se nadomesti dodatno vreme potrebno za traženje jedinke u kešu. Drugi problem leži u činjenici da se keširanjem jedinki pokušava ubrzati već veoma efikasan program. Naime, vreme pronalaženja vrednosti funkcije cilja na novoj jedinki je veoma kratko jer se u većini slučajeva pronalazi jedinka koja je susedna sa prethodnom jedinkom za koju je već poznata vrednost funkcije cilja. Dovoljno je ažurirati ovu vrednost prolaženjem kroz trojke koje sadrže neki od dva permutovana indeksa. Pošto se provera vrši za dva indeksa, složenost izračunavanja vrednosti funkcije cilja je , što je za instance sa ulaznim parametrima koji zadovoljavaju svega 12 operacija.

I pored negativnih rezultata, opisani metod je obećavajući. Najvažnije je primeniti ga na problem kod koga bi procenat keš pogodaka bio veći. Što se tiče drugog problema, u većini primena je vreme izračunavanja vrednosti funkcije cilja značajan faktor. Čak i kod ovog problema su rezultati značajno bolji za instance sa povoljnijim odnosom . To se može videti iz poslednjeg grafika, gde se ubrzanje povećava sa prelaskom sa prve na drugu instancu i sa druge na treću instancu, zatim sa prelaskom sa četvrte na petu i sa pete na šestu instancu i tako dalje. Svaki put kada se broj trojki poveća sa fiksiranom dimenzijom problema , odnos postaje povoljniji i program sa keširanjem daje bolje rezultate. Prilikom primene na problem sa povoljnijim odnosom opisanih faktora, moguće je očekivati još bolje rezultate.

Između ostalog, radi se i simboličko izračunavanje u bilogiji [19].

# Literatura

|  |  |
| --- | --- |
| [1] | F. Schoen, "A wide class of test functions for global optimization," *Proceedings of the 3rd International Conference on Genetic Algorithms,* pp. 51-60, 1989. |
| [2] | B. J. More and Y. Wu, "Global Smoothing and Continuation for Large-scale Molecular Optimization," *Argonne National Laboratory, Illinois,* pp. MCS-P539-1095, 1995. |
| [3] | M. Demirhan, "A geometric partitioning metaheuristic for global optimization," *Journal of Global Optimization,* pp. 14: 415-435, 1999. |
| [4] | W. Huzer and A. Neumaier, "Global optimization bz multilevel coordinate search," *Journal of Global Optimization,* pp. 14: 331-355, 1999. |
| [5] | W. E. Hart, "Adaptive Global Optimization with Local Search," *PhD Thesis, University of California, San Diego,* 1994. |
| [6] | J. Kratica, S. Radojević and V. Šešum, "Jedna metoda poboljšanja vremena izvršavanja prostog genetskog algoritma," *XXIII Jupiter konferencija, Zbornik radova, Mašinski fakultet, Beograd,* pp. 457-462, 1997. |
| [7] | J. J. Kratica, "Paralelizacija genetskih algoritama za rešavanje nekih NP-kompletnih problema," Doktorska disertacija, Matematički fakultet, Univerzitet u Beogradu, 2000. |
| [8] | T. H. Cormen, C. E. Leiserson and D. L. Rivest, Introduction to Algorithms, Cambridge MA: MIT Press, 1990. |
| [9] | R. Williams, "A Painless Guide To CRC Error Detection Algorithms," *ftp://ftp.adelaide.edu.au/pub/rocksoft/crc\_v3.txt,* 1993. |
| [10] | J. Opatrny, "Total ordering problem," *SIAM J. Comput. 8,* pp. 111-114, 1979. |
| [11] | B. Chor and M. Sudan, "A geometric approach to betweenness," *SIAM J. Discrete Math. 11 (4),* pp. 511-523, 1998. |
| [12] | D. Slomin, D. Stein, L. Kruglyak, D. Lander and Rhmapper, "An interactive coputer package for constructing radiation hybrids maps," *URL http://www.genome.wi.mit.edu/ftp/pub/software/rhmapper,* 1996. |
| [13] | D. Slonim, D. Stein, L. Kruglyak and E. Lander, "Building human genome maps with radiation hybrids," *J. Comput. Biol. 4,* pp. 487-504, 1997. |
| [14] | D. Cox, M. Burmeister, E. Price, S. Kim and R. Myers, "Radiation hybrid mapping: A somatic cell genetic method for constructin high resolution maps of mammlian chromosomes," *Science 250,* pp. 245-250, 1990. |
| [15] | T. Christof, M. Junger, J. Kececioglu, P. Mutzel and G. Reinelt, "A branch-and-cut approach to physical mapping with end probes," *Proceedings of the 1st Annual International Conference on Computational Molecular Biology (RECOMB-97),* pp. 84-92, 1997. |
| [16] | G. Goss and H. Harris, "New methods for mapping genes in human chromosomes," *Nature 255,* pp. 680-84, 1975. |
| [17] | T. Christof, M. Oswald and G. Reinelt, "Consecutive ones and a betweenness problem in computational biology," *Proceedings of the 6th Conference on Integer Programming and Combinatorial Optimization, Springer-Verlag Lecture Notes in Computer Science, Vol. 1412,* 1998. |
| [18] | A. Savić, J. Kratica, M. Milanović and D. Dugošija, "A mixed integer linear programming formulation of the maximum betweenness problem," *European Journal of Operational Research 206 (3),* pp. 522-527, 2010. |
| [19] | M. P. Barnett, "Symbolic Calculation in the Life Sciences - Some Trends and Prospects," *Algebraic Biology,* pp. 1-16, 2005. |
| [20] | F. J. Solis and R. J.-B. Wets, "Minimazation by random search techniques," *Mathematics of Operations Research 6,* pp. 19-31, 1981. |

# Dodatak A

U dodatku ćemo opisati najbitnije aspekte implementacije kako bi čitalac mogao ponoviti postupak i napraviti sopstvenu implementaciju. Ono na čemu se ceo kod zasniva su strukture podataka koje osim za skladištenje podataka služe i za prosleđivanje parametara procedurama. Pored toga, razmotrićemo i implementaciju keša pomoću heš-red strukture koja je ključna za ceo rad.

Za početak, potrebno je napraviti pogodnu strukturu podataka za skladištenje jedinke. Podsetimo se da je kod MPSE jedinka predstavljena permutacijom brojeva , gde je broj koji nije unapred poznat. Prva ideja je da se koristi niz dužine , ali se odmah uočava da to zahteva više prostora nego što je neophodno. Ono što se ne uočava na prvi pogled je da bi ovaj pristup doveo do linearne složenosti za proceduru nalaženja CRC koda jedinke koja je esencijalna za heš-red strukturu. Ona se poziva u veoma osetljivom delu koda, čija složenost predstavlja složenost celog programa i neprihvatljivo je na ovakvom mestu koristiti linearno računanje. Zbog toga se koristi jedna vrsta kompresije ove permutacije što dovodi do mnogostrukog ubrzanja:

typedef unsigned long T;

typedef struct Fen

{

T \*\_y;

} Fen;

Tip T je uveden kako bi se na lak način mogao promeniti osnovni tip skladištenja podataka u slučaju prebacivanja (tzv. portovanja) programa na drugi operativni sistem ili kompajler. Struktura Fen sadrži niz elemenata izabranog tipa, čija minimalna dužina mora da se odredi na početku izvršavanja programa, kada se učita veličina :

static int \_n;

static int \_bits; // # of bits for 1 number

static int \_nums; // # of numbers in 1 T

static int \_size; // # of Ts, i.e. size of \_y

void FenInitStatic(int n)

{

\_n = n;

\_bits = (int)ceil(log10(\_n\*1.)/log10(2.));

\_nums = (int)floor(sizeof(T) \* 8 \* 1./\_bits);

\_size = (int)ceil(\_n \* 1. / \_nums);

}

Ova izračunavanja je dovoljno izvršiti samo jednom u toku izvršavanja programa, dok se alokacija i brisanje memorije mora pozivati za svaku jedinku ponaosob:

void FenInit(Fen \*fen)

{

fen->\_y = (T \*)malloc(\_size\*sizeof(T));

}

void FenDeinit(Fen \*fen)

{

free(fen->\_y);

}

Kada je memorija alocirana potrebno ju je popuniti iz dostupnih podataka. Na početku izvršavanja to je niz brojeva (sama permutacija), dok je kasnije potrebno prepisati podatke iz strukture koja sadrži promenljive problema koji se rešava ili iz druge jedinke:

void FenReinit(Fen \*fen, int niz[])

{

    int posi = 0; // index in \_y

    int posj = 0; // index in \_y[posi]

    int i;

    T mask;

    T initMask = 1;

    T reverseMask;

    T shiftedNumber;

    for (i = 0; i < \_size; i++)

    {

        fen->\_y[i] = 0;

    }

    initMask <<= \_bits;

    initMask -= 1; // 0...01..1

    mask = initMask;

    for (i = 0; i < \_n;)

    {

        shiftedNumber = ((T)niz[i]) << posj\*\_bits;

        reverseMask = mask^((T)-1);  //1..10..01..1

        fen->\_y[posi] &= reverseMask;

        fen->\_y[posi] |= shiftedNumber;

        i++;

        posj++;

        if (posj != \_nums)

        {

            mask <<= \_bits;

        }

        else

        {

            posj = 0;

            posi++;

            mask = initMask;

        }

    }

}

void FenReinitPSE(Fen \*fen, struct problem\_struct \*problem)

{

    int posi = 0; // index in \_y

    int posj = 0; // index in \_y[posi]

    int i;

    T mask;

    T initMask = 1;

    T reverseMask;

    T shiftedNumber;

    for (i = 0; i < \_size; i++)

    {

        fen->\_y[i] = 0;

    }

    initMask <<= \_bits;

    initMask -= 1; // 0...01..1

    mask = initMask;

    for (i = 0; i < \_n; )

    {

        shiftedNumber = ((T)problem->p[problem->se[i]]) << posj\*\_bits;

        reverseMask = mask^((T)-1);  //1..10..01..1

        fen->\_y[posi] &= reverseMask;

        fen->\_y[posi] |= shiftedNumber;

        i++;

        posj++;

        if (posj != \_nums)

        {

            mask <<= \_bits;

        }

        else

        {

            posj = 0;

            posi++;

            mask = initMask;

        }

    }

}

void FenReinit2(Fen \*fen1, Fen \*fen2)

{

    int i;

    for (i = 0; i < \_size; i++)

    {

        fen1->\_y[i] = fen2->\_y[i];

    }

}

U prve dve procedure su elementi niza jedan za drugim stavljani na odgovarajuće bitove strukture Fen. U trećoj proceduri je primenjena brza obrada tako da su kopirani celi elementi niza \_y koji odgovaraju većem broju elemenata originalne permutacije. Ona će biti pozivana prilikom skladištenja jedinke u kešu.

Za dobijanje elementa iz permutacije sa izabranim indeksom dovoljno je pozvati proceduru:

int FenGetI(Fen \*fen, int i)

{

    int posi = i/\_nums;

    int posj = i%\_nums;

    int sol = fen->\_y[posi] >> (posj\*\_bits);

    int mask = 1;

    mask <<= \_bits;

    mask -= 1;

    sol &= mask;

    return sol;

}

Prilikom lokalne pretrage po prostoru jedinki, potrebno je kreirati jedinku koja je veoma slična onoj koju već imamo. Pri tome je potrebna operacija, koja za dve izabrane pozicije u permutaciji zamenjuje vrednost permutacije na tim indeksima:

void FenSwap(Fen \*fen, int a, int b)

{

    int posia = a/\_nums;

    int posja = a%\_nums;

    int posib = b/\_nums;

    int posjb = b%\_nums;

    T maska, maskb, broja, brojb;

    T initMask = 1;

    initMask <<= \_bits;

    initMask -= 1; // 0001

    maska = initMask << posja\*\_bits; // 0010

    maskb = initMask << posjb\*\_bits; // 0100

    broja = fen->\_y[posia]&maska; // 00a0

    brojb = fen->\_y[posib]&maskb; // 0b00

    fen->\_y[posia] &= ~maska;

    fen->\_y[posib] &= ~maskb;

    if (posja > posjb)

    {

        broja >>= (posja-posjb)\*\_bits;

        brojb <<= (posja-posjb)\*\_bits;

    }

    if (posja < posjb)

    {

        broja <<= (posjb-posja)\*\_bits;

        brojb >>= (posjb-posja)\*\_bits;

    }

    fen->\_y[posia] |= brojb;

    fen->\_y[posib] |= broja;

}

Najbitnija procedura za rad sa jedinkama pronalazi CRC vrednost. U prvi mah je korišćen originalan metod izračunavanja. Međutim, zbog performansi je ovaj metod promenjen, tako da koristi heuristiku koja se brzo izračunava i daje zavidne rezultate pri “razbacivanju” jedinki na interval , gde je ulazni parametar ili unapred izabrana konstanta ili vrednost koja zavisi od ulaznih parametara. U ovoj implementaciji je izabrana vrednost puta manja od veličine keša, što znači da je za pun keš matematičko očekivanje zastupljenosti svakog od brojeva iz pomenutog intervala kao CRC vrednosti svake od jedinki u kešu jednako .

int FenFindCRC(Fen \*fen, int max)

{

    int i;

    T sol = 0;

    for (i = 0; i < \_size; i++)

    {

        sol ^= (fen->\_y[i]<<i);

    }

    return sol % max;

}

Složenost je linearna, ali ne po , već po veličini niza što je nekoliko puta manje. Isto važi i za sledeću proceduru, koja proverava da li su dve jedinke jednake:

int FenEqual(Fen \*fen1, Fen \*fen2)

{

    int i;

    for (i = 0; i < \_size; i++)

    {

        if (fen1->\_y[i] != fen2->\_y[i])

        {

            return 0;

        }

    }

    return 1;

}

Ona vraća 0 (netačno) ako su jedinke različite i 1 (tačno) ako su jedinke iste.

Posmatrajmo sada strukturu koja će sadržati jedan element za skladištenje u kešu:

typedef struct cacheitem\_struct

{

    Fen \*fen;

    double fv;

unsigned long CRC;

int fhash;

    cp fp;

cp fs;

    cp up;

    cp us;

} cacheitem\_struct;

Pored fenotipa same jedinke tu se još nalazi vrednost funkcije cilja fv i CRC vrednost jedinke. Promenljiva fhash se koristi za čuvanje pozicije u nizu fitem na kojoj se nalazi dvostruko povezana lista sa elementima koji su instance tipa cacheitem\_struct. Pokazivači na prethodni i sledeći element u ovoj listi se čuvaju u pokazivačima fp i fs. Sa druge strane pokazivači up i us se odnose na red, ponovo dvostruko povezanu listu za pronalaženje najduže nekorišćenog člana (eng. Least Recently Used – LRU), politike koja se primenjuje u situaciji kada je keš pun i potrebno je osloboditi prostor za nove elemente.

Struktura koja predstavlja ceo keš:

typedef struct cache\_struct

{

   int n;

  int maxn1, maxn2;

   long count;

  unsigned long CRC;

   cp pfitem;

   cp \*fitem;

   int fhash;

  cp ubegin, uend;

  long int nc;

   long int nu;

  long int nc1;

   long int nu1;

   long int nc2;

   long int nu2;

   double pc;

   void (\*Init)(pem);

  void (\*Search)(pem);

   void (\*AddNew)(pem);

  void (\*SetOld)(pem);

} cache\_struct;

Promenljiva n sadrži trenutni broj elemenata u kešu, dok maxn1 odnosno maxn2 predstavljaju maksimalni broj elemenata u kešu, tj. njegovu veličinu odnosno broj povezanih listi u kešu, tj. broj elemenata niza fitem. Promenljiva count sadrži ukupan broj jedinki koje su bile u kešu. Promenljiva CRC odgovara CRC vrednosti koja je poslednja tražena u kešu i ovde se čuva kako bi se izbeglo njeno dvostruko računanje. Kada se jedinka pronađe u kešu, na nju se pokazivati pfitem, koji će u slučaju nepronalaženja sadržati NULL. Kao što je već pomenuto fitem je niz dvostuko povezanih listi i u njemu će indeks jedinke koja se trenutno dodaje biti fhash, dok su ubegin i uend pokazivači na početak i kraj reda.

Promenljive nc, nu, nc1, nu1, nc2, nu2 i pc su razni brojači i predstavljaju redom: ukupan broj iskorišćenih keširanja, ukupan broj pokušaja keširanja, broj iskorišćenih keširanja dok je , ukupan broj pokušaja keširanja dok je , broj iskorišćenih keširanja do pronalaženja najboljeg rešenja, ukupan broj pokušaja do pronalaženja najboljeg rešenja i procenat iskorišćenih keširanja.

Pored njih ova struktura sadrži i pokazivače na procedure za rad sa kešom. Ovo je posebno zgodno za program sa više ražličitih implementacija keša. Prilikom promene implementacije dovoljno je ove pokazivače postaviti na imena odgovarajućih procedura i ostatak koda može da ostane isti. Ovde je urađen korak više, tako da što je definisana globalna promenljiva cacheInUse koja će sadržati ime izabrane implementacije keša:

enum cacheType

{

    CacheNone,

    CacheFL,

    CacheCDHT,

    CacheCDHT2,

    CacheCHQT,

    CacheCHQT2

};

enum cacheType cacheInUse;

Prikažimo ovde još i centralnu strukturu u aplikaciji:

typedef struct

{

    char wo;

    int n;

    int nitm, iitm;

    item\_struct \*pop;

    int \*ipop;

    int gen, genp, genmax;

    int rep;

    int repmax;

    int nls, nlsp;

    FILE \*finp,\*fsol,\*fout;

    char sinp[MAXS], ssol[MAXS], sout[MAXS];

    char cmdl[MAXS];

    double lb,ub,trs;

    clock\_t runs, runp, rune;

    time\_t ts,tp,te;

    char sts[MAXS], ste[MAXS], stp[MAXS];

    double rt, rtp;

    unsigned long seed;

    double randrate;

    f\_struct f;

    cache\_struct cache;

} em\_struct;

Ukratko, ona sadrži vrstu optimizacije, dimenziju problema, broj jedinki u populaciji kao i trenutnu jedinku, jednu populaciju jedinki, jednu permutaciju čiji elementi predstavljaju indekse jedinki u populaciji sortirane po vrednosti funkcije cilja za tu jedinku. Zatim broj generacija, generacija kada je dobijeno rešenje, maksimalni broj generacija, broj ponavljanja iste jedinke kao najbolje u uzastopnim generacijama, najveći dozvoljeni broj tih ponavljanja, broj lokalnih iteracija i broj lokalnih iteracija kada je dobijeno rešenje. Ovde se čuva i ulazni fajl, dva izlazna fajla i njihova imena, komandna linija, granice i kvant prostora, a tu su i razne promenljive za rad sa vremenom i slučajnim brojevima. Promenljiva f sadrži procedure za rad sa konkretnim problemom, dok je cache već opisani keš za skladištenje jedinki.

Razmotrimo sada implementaciju keša pomoću heš-red strukture. Kao što je već opisano u radu, postoji četiri procedure za rad sa kešom. Prva inicijalizuje keš strukturu:

void CacheInitCHQT(em\_struct \*em)

{

    int k;

    em->cache.n = 0;

    em->cache.count = 0;

    em->cache.nc = 0;

    em->cache.nu = 0;

    hashM = (sqrt(5.)-1.)/2.;

    for(k = 0; k < em->cache.maxn2; ++k)

    {

        em->cache.fitem[k] = NULL;

    }

    em->cache.ubegin = NULL;

    em->cache.uend = NULL;

}

Brojači se anuliraju, a pokazivači na liste kao i na početak i kraj reda se postavljaju na NULL. Promenljiva hashM se postavlja na već pomenutu konstantu. Traženje jedinke u kešu se vrši pozivanjem procedure:

void CacheSearchCDHT(em\_struct \* em)

{

    int j;

    double tmp;

    cp z, yy;

    em->cache.nu++;

    em->cache.CRC = FenFindCRC(em->pop[em->iitm].fen, em->cache.maxn2);

    em->pop[em->iitm].CRC = em->cache.CRC;

    tmp = hashM \* ((double) z->CRC);

    tmp -= (unsigned long) tmp;

    em->cache.fhash = (int) (tmp \* em->cache.maxn2);

    z = em->cache.fitem[em->cache.fhash];

    yy = NULL;

    while(yy==NULL && z!=NULL)

    {

        if(em->cache.CRC == z->CRC)

        {

            if (FenEqual(z->fen, em->pop[em->iitm].fen) == 1)

            {

                yy = z;

            }

        }

        z = z->fs;

    }

    if(yy != NULL)

    {

        em->cache.nc++;

    }

    em->cache.pfitem = yy;

}

Prvo se računa CRC vrednost jedinke i na osnovu nje indeks fhash u nizu fitem svih dvostuko povezanih listi. Pokazivač na odgovarajuću listu se smešta u promenljivu z i zatim se u njoj traži jedinka em->pop[em->iitm].fen. Petlja while se izvršava do god nije pronađen odgovarajući element u listi i dok se nije došlo do kraja liste. Prvo se vrši znatno brža provera – da li dve jedinke imaju isti CRC kod. U slučaju da jeste isti, može se vršiti dodatna provera da li je fenotip dve jedinke zaista jednak. Pronadjena jedinka se smešta u promenljivu yy. U slučaju da se jedinke razlikuju, nastavlja se pretraživanje od sledećeg elementa u listi. Na kraju procedure, em->cache.pfitem se postavlja ili na pronađeni element ili na vrednost NULL.

U slučaju da jedinka nije pronađena u kešu, ona se unosi u njega:

void CacheAddNewCHQT(em\_struct \*em)

{

    int i;

    cp z, item, x;

    double tmp;

    item = em->cache.fitem[em->cache.fhash];

    if(em->cache.n == em->cache.maxn1)

    {

        z = em->cache.uend;

        em->cache.uend = z->up;

        z->up->us = NULL;

        if(z->fp != NULL)

        {

            z->fp->fs = z->fs;

        }

        else

        {

            em->cache.fitem[z->fhash] = z->fs;

        }

        if(z->fs != NULL)

        {

            z->fs->fp = z->fp;

        }

    }

    else

    {

        em->cache.n++;

        z = (cacheitem\_struct \*)malloc(sizeof(cacheitem\_struct));

        z->fen = (Fen \*)malloc(sizeof(Fen));

        FenInit(z->fen);

        if(z == NULL || z->fen == NULL)

        {

            printf("Allocation %d failed\n", em->cache.n);

            exit(0);

        }

    }

    x = em->cache.ubegin;

    em->cache.ubegin = z;

    z->us = x;

    z->up = NULL;

    if(x != NULL) x->up = z;

    else em->cache.uend = z;

    FenReinit2(z->fen, em->pop[em->iitm].fen);

    z->fv = em->pop[em->iitm].fv;

    z->CRC = em->cache.CRC;

    em->cache.fhash = z->CRC;

    x = em->cache.fitem[em->cache.fhash];

    z->fs = x;

    z->fp = NULL;

    if(x!=NULL)

    {

        x->fp = z;

    }

    em->cache.fitem[em->cache.fhash] = z;

    z->fhash = em->cache.fhash;

    em->cache.pfitem = z;

}

Vrednost fhash je već izračunata tako da se u fitem stavlja pokazivač na listu u koju treba ubaciti novi element.

Sada je potrebno razmotriti koliko jedinki se nalazi u kešu. U slučaju da je keš pun, potrebno je osloboditi prostor za novu jedinku. Strategija za izbacivanje je pomunuti LRU, pa se u promenljivu z stavlja poslednji element iz reda na koji pokazuje pokazivač uend. Pored toga potrebno ažuriranje raznih pokazivača, kako bi se jedinka z izbrisala iz keša. Među njima je pokazivač na poslednji element u redu, pokazivači u dvostruko povezanoj listi u kojoj se jedinka nalazila, kao i pokazivač na tu listu, u slučaju da je jedinka bila na početku liste. Kada keš nije pun, potrebno je samo alocirati prostor za novu jedinku.

Sada se jedinka z koja je ili izbačena iz keša ili je novonapravljena unosi u keš. Potrebno je prepisati karakterisike jedinke, kao i izvršiti odgovarajuće ulančavanje. Na kraju procedure se u pfitem stavlja pokazivač na novododatu jedinku.

Za kraj, kada je jedinka pronađena u kešu, potrebno je da se ona nekako označi kako bi se znalo da je skoro korišćena. To se postiže njenim razlančavanjem iz reda i ponovnim ulančavanjem na njegov početak.

void CacheSetOldCHQT(em\_struct \*em)

{

    cp x, z;

    z = em->cache.pfitem;

    em->cache.count++;

    em->pop[em->iitm].fv = z->fv;

    if(z != em->cache.ubegin)

    {

        if(z->us != NULL)

        {

            z->us->up = z->up;

        }

        else

        {

            em->cache.uend = z->up;

        }

        if (z->up != NULL)

        {

            z->up->us = z->us;

        }

        x = em->cache.ubegin;

        em->cache.ubegin = z;

        z->us = x;

        z->up = NULL;

        x->up = z;

    }

}

Ovim su opisane najvažnije strukture za implementaciju rešenja opisanog problema, kao i procedure za rad sa širokoprimenljivom heš-red strukturom.

1. Neposredno iznad [↑](#footnote-ref-1)