Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5208-49439

Bc. Róbert Cuprik

Hľadanie motívov v DNA sekvenciách

Diplomová práca

Vedúci práce: prof. Ing. Pavol Návrat, PhD.

Máj 2017

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5208-49439

Bc. Róbert Cuprik

Hľadanie motívov v DNA sekvenciách

Diplomová práca

Študijný program:	Informačné systémy
Študijný odbor:	9.2.6 Informačné systémy
Miesto vypracovania:	Ústav informatiky, informačných systémov a softvérového inžinierstva, FIIT STU v Bratislave
Vedúci práce:	prof. Ing. Pavol Návrat, PhD.

Máj 2017

ČESTNÉ PREHLÁSENIE

Čestne prehlasujem, že záverečnú prácu som vypracoval samostatne s použitím uvedenej literatúry a na základe svojich vedomostí a znalostí.

.....

Bc. Róbert Cuprik

POĎAKOVANIE

Touto cestou by som sa chcel poďakovať svojmu vedúcemu práce prof. Ing. Pavlovi Návratovi PhD. za odborné vedenie a rady pri písaní tejto práce.

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Študijný program: Informačné systémy
Diplomová práca: Hľadanie motívov v DNA sekvenciách
Autor: Bc. Róbert Cuprik
Vedúci práce: prof. Ing. Pavol Návrat, PhD.
Máj 2017

V súčasnom svete sa často stretávame s problémami, ktorých riešenie si vyžaduje časovo náročné prehľadávanie doménového priestoru riešení. Jedným z takýchto problémov je aj problém hľadania motívov v DNA sekvenciách. Motívy sú krátke úseky DNA, ktoré nesú biologickú informáciu. Tento problém sa dá opísať ako problém optimalizácie, kde sa snažíme nájsť taký motív, ktorý dosiahne čo najlepšie ohodnotenie.

Jedným z možných prístupov, ako riešiť problémy optimalizácie, sú metaheuristiké algoritmy. Táto skupina algoritmov sa snaží iteratívnym procesom vylepšovať nájdené riešenia až kým nedosiahne globálne optimum, ktoré ale negarantuje.

Vylepšenie týchto algoritmov paralelizáciou nemusí predstavovať iba ich zrýchlenie, ale ponúka aj možnosť zlepšenia kvality výsledkov. Príkladom môžu byť genetické algoritmy, kde viaceré populácie paralelne hľadajú riešenie, a ich jedinci následne migrujú, aby dosiahli vyššiu kvalitu výsledkov.

V tejto práci sa budeme zaoberať takýmito prístupmi a opíšeme paralelnú kooperatívnu metaheuristiku, ktorá sa dá aplikovať nie len na problémy hľadania motívov.

Annotation

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Degree Course: Information systems
Master thesis: Motif Finding In DNA Sequences
Author: Bc. Róbert Cuprik
Supervisor: prof. Ing. Pavol Návrat, PhD.
2017, May

Currently, it is common to be challenged by a problem whose solution requires exhaustive search in domain space. One example of such can be motif finding problem in DNA sequences. Motifs are short sequences of DNA that contain biologically relevant information. This problem can be seen as problem of optimization, where our goal is to find motif that has the highest score in some kind of evaluation metric.

One way to solve optimization problems is to use metaheuristic algorithms. This group of algorithms tries to improve found solutions in iterative process until it reaches global optima, which is not guaranteed.

Parallelization in this group of algorithms may improve not only time needed for execution but quality of found solutions as well. In parallel genetic algorithm, multiple concurrent populations exchange individuals to improve overall quality of solutions.

In this work we analyze possibilities of parallelization in metaheuristics and describe parallel cooperative algorithm that can be used to solve problems of optimization.

Obsah

1 Úvod

Každý organizmus na tejto zemi sa formuje pomocou DNA. Snaha ju pochopiť a rozlúštiť motivovala biológov z celého sveta, aby vyvíjali stále nové a nové metódy na jej čítanie. Proces tohto čítania nie je dokonalý a preto biológovia potrebovali pomoc informatikov, aby vyvíjali algoritmy na jej skladanie. Medzi takéto bioinformatické problémy patrí aj problém hľadania motívov, ktorý pracuje už s poskladanými úsekmi DNA.

Takéto motívy sú krátke úseky DNA nazývané aj „binding sites“, ktoré sa vyskytujú pred ďalšími dlhšími úsekmi, ktoré používa organizmus na tvorbu bielkovín. Tieto proteíny, nazývané aj transkripčné faktory, sa upínajú na motívy a pomáhajú následnému čítaniu informácie z DNA [11]. Aby sme vedeli lepšie pochopiť obsah DNA, je užitočné vedieť, kde sa takéto motívy vyskytujú. Za nimi sa totiž nachádza prepis pre proteín, ktorý môže organizmus použiť napríklad na boj proti infekcii.

Hľadanie týchto motívov nie je jednoduché. Vieme o nich iba to, že sú pomerne krátke (rádovo v desiatkách nukleotidov) a že sa zvyknú opakovať. Takže pre danú množinu reťazcov hľadáme taký podreťazec, ktorý sa vyskytuje viackrát. Za naivné riešenie by sa dalo považovať skúšanie všetkých možných podreťazcov. To bohužiaľ nie je možné nie len kvôli časovej zložitosti (tento problém je NP-úplný), ale aj pre to, že tieto motívy podliehajú mutáciám. Môže sa stať, že v danom motíve je nejaký nukleotid navyše, chýba, alebo je zmenený [8].

Metaheuristické algoritmy sa na problém hľadania motívov pozerajú ako na problém optimalizácie. Predmetom optimalizácie je samotný motív, resp. jeho ohodnotenie. To, ako tento motív ohodnotiť, sa stále skúma, no v [17] použili na jeho ohodnotenie tri metriky. Dĺžku motívu – čím dlhší tím lepší, podporu motívu – v koľkých sekvenciách sme daný motív našli, a podobnosť motívu – ako veľmi sa tento motív medzi sekvenciami podobá. Na toto ohodnocovanie sa dá použiť aj tzv. entropia, ktorá presnejšie opisuje podobnosť medzi motívmi. Pri normálnej podobnosti sa sčíta frekvencia najčastejšie sa vyskytujúcich nukleotidov na daných pozíciách. Pri entropii sa berú do úvahy aj ostatné frekvencie a teda dokážeme rozlíšiť motívy, ktoré možno majú rovnaké frekvencie najdominantnejších nukle-

otidov, ale môžu sa líšiť v ostatných frekvenciách. Pomocou entropie dokážeme určiť, ktorý z týchto motívov je zakonzervovanejší, to znamená, že menej podlieha mutáciám.

Pri metaheuristických algoritmoch sú ohodnocovacie funkcie rovnako dôležité, ako samotný spôsob hľadania riešenia. Metaheuristiky sa snažia rôznymi iteratívnymi metódami posúvať riešenie v priestore tak, aby zlepšovali – optimalizovali – jeho ohodnotenia. Existuje veľké množstvo spôsobov tohto prehľadávania, ale vo všeobecnosti každý zachováva dva princípy. Intenzifikácia, kde sa algoritmus snaží usmerniť nájdené riešenie k lepšiemu, a diverzifikácia, kde sa algoritmus snaží nájsť nové riešenia. Pri dôraze na intenzifikáciu algoritmus nájde riešenie skôr, no nemusí dosiahnuť globálne optimum.

Jedným z druhov metaheuristík je tzv. rojová inteligencia. Je to koncept umelej inteligencie, kde sa práve na prehľadávanie doménového priestoru využívajú algoritmy inšpirované tým, ako zvieratá a hmyz hľadajú potravu. V [17] využili metaforu včiel a implementovali algoritmus MOABC (Multi Objective Artificial Bee Colony). Tento algoritmus je založený na tom, že doménový priestor, v ktorom sa pohybujú včely, určuje polohu a veľkosť motívu v sekvenciách. Keď sa včela pohne, zmení sa aj príslušný motív.

V súčasnosti sa trend v metaheuristikách používaných pri hľadaní motívov presúva od hľadania najlepšej metafory alebo ohodnotenia k preskúmavaniu možnosti využitia viacerých metaheuristík a prístupov počas prehľadávania. Následne sa kombinujú výsledky, či už na konci, alebo počas samotného procesu hľadania.

V tejto práci sa budeme venovať problému hľadania motívov použitím metaheuristík ako jedného z možných prístupov. Pokúsime sa navrhnúť riešenie, ktoré bude využívať paralelné prístupy, nie len na zrýchlenie behu algoritmu, ale aj na zlepšenie kvality dosiahnutých riešení.

1.1 Štruktúra práce

Táto práca je členená nasledovne. V kapitole 2 opisujeme, čo sú to motívy a definíciu problému ich hľadania. V kapitole 3 a 4 opisujeme metaheuristiky, paretové rady a klasifikáciu metaheuristík z paralelizačného hľadiska. V kapitole 5 následne opíšeme perzistentné dátové štruktúry ako jeden z možných prístupov

zdieľanej pamäte medzi paralelne bežiacimi procesmi. V 6. kapitole opíšeme náš návrh algoritmu, ktorý je postavený na predošlých kapitolách. V kapitole 7 opíšeme bližšie implementáciu a niektoré návrhové rozhodnutia. V kapitolách 8 a 9 následne opíšeme spôsob testovania a vyhodnotíme celú prácu.

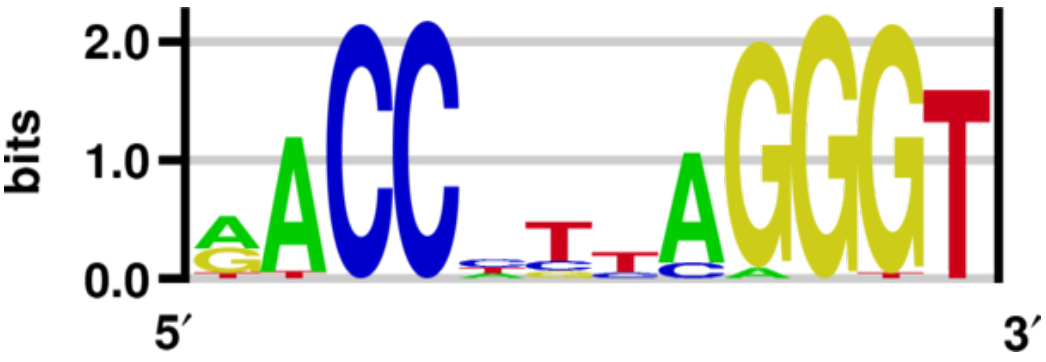
2 Problém hľadania motívov v DNA sekvenciách

DNA motívy sú krátke opakujúce sa vzory v DNA, o ktorých sa predpokladá, že majú biologickú funkciu [11]. Častokrát indikujú väzobné miesto pre proteíny zvané transkripčné faktory alebo sú zahrnuté v procesoch s mRNA.

Aj keď sú tieto motívy krátke (okolo 30 nukleotidov), miesta, na ktorých sa môžu vyskytovať v DNA, môžu mať niekoľko sto až vyše tisíc nukleotidov [21]. Tieto motívy navyše nemusia byť rovnaké. Zatiaľ čo enzým EcoRI sa napája iba na sekvenciu GAATTC, enzým HindII sa napája na sekvenciu GTYRAC, kde písmeno Y môže predstavovať nukleotid C alebo T a písmeno R môže byť nukleotid A alebo G [11]. Motívy tvoria nukleotidy (A,T,G,C), ktoré nemusia byť v každom výskyte motívu rovnaké. Zvyknú sa preto reprezentovať cez tzv. *position frequency matrix* (PFM). Takúto maticu je možno vidieť v tabuľke 1 spolu s logom, ktoré je znázornené na obrázku č. 1.

Tabuľka 1: *Časť PFM pre motív YJL056C v géne ZAP1*[1].

A	0.502	0.9243	0.0137	0.0029	0.259	...
T	0.0912	0.0485	0.006	0.0129	0.2702	...
G	0.3905	0.0168	0.0022	0.003	0.0795	...
C	0.0163	0.0104	0.9781	0.9812	0.3913	...



Obr. 1: *Logo motívu YJL056C v géne ZAP1*[1].

[1] http://yetfasco.ccbr.utoronto.ca/showPFM.php?mot=YJL056C_2097.0

PFM matica je tvorená frekvenciami výskytov určitého nukleotidu na danom mieste. Logo motívu, ktoré navrhol Schneider a Stephens [34], zobrazuje tieto frekvencie pomermi veľkosti písmen. Narozdiel od PFM uvádza aj zakonzervovanosť motívu znázornenú výškou nukleotidu v bitoch. Tá súvisí s tým, ako veľmi podlieha dané miesto mutáciám.

2.1 Definícia problému

V [21] definujú problém hľadania motívov ako problém lokálneho zarovnania viacerých sekvencií, za predpokladu, že model motívu dosiahne optimálne skóre pre nejakú ohodnocovaciu funkciu. Tento problém je NP-úplný, no dá sa na neho pozerať ako na problém optimalizácie. V [21] sú optimalizačné problémy o M ohodnocovacích funkcíach matematicky opísané takto:

$$\text{Minimalizuj} f(x) = [f_i(x), i = 1, \ldots, M]$$

zatiaľ čo platí:

$$g_j(x) \leq 0 \qquad j = 1, 2, ...J$$

$$h_i(x) = 0 \qquad i = 1, 2, ...K$$

2.1.1 Ohodnotenie motívov

Takýto opis problému hľadania motívov využívame ďalej v tejto práci. Preto uvádzame ohodnocovacie funkcie a ohraničenia, ktoré boli opísané v [17] a [21] vzhľadom na túto problematiku.

Maximalizuj:

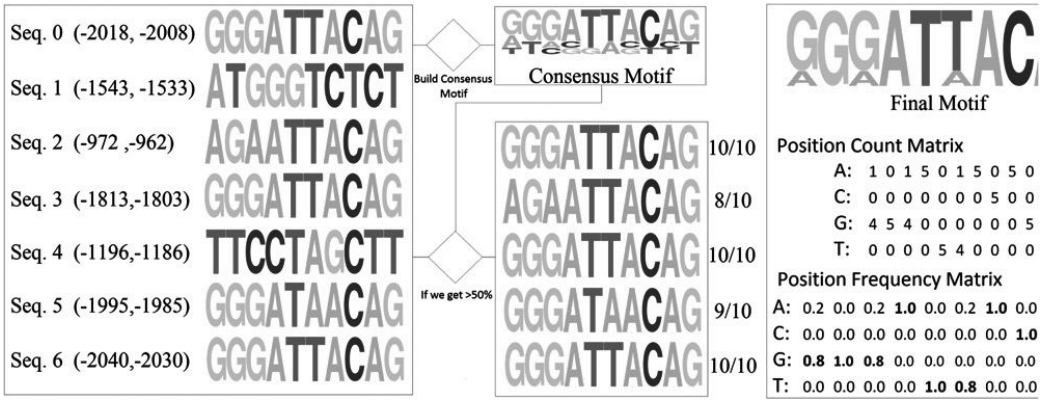
- dĺžku motívu l,
- podporu motívu – táto metrika opisuje, v koľkých sekvenciách sme boli schopní daný motív identifikovať,
- podobnosť motívu medzi sekvenciami na základe PFM,

- entropiu motívu – táto metrika opisuje mieru neistoty v rozložení nukleotidov v PFM, čím je entropia nižšia, tým lepšie. Keďže maximalizujeme ohodnocovacie funkcie, musíme túto metriku upraviť napríklad prenásobením -1. Vo všeobecnosti ju ale upravujeme tak, aby mala rozsah od 0 po 1,

zatiaľ čo je splnené:

- ohraničenie dĺžky – v [17] je to medzi 6 a 64 nukleotidov,
- minimálna podpora motívu medzi sekvenciami,
- minimálna komplexita motívu.

Na to, aby sme vedeli vyrátať tieto metriky, potrebujeme získať PFM reprezentáciu daného motívu. Prehľadávací algoritmus (metaheuristika) nám poskytne pozíciu v doménovom priestore, ktorá predstavuje jedno možné riešenie. Toto riešenie obsahuje pozície motívov v každej sekvencii a jeho dĺžku. PFM avšak nerátame z každého motívu [17]. Najprv si z týchto motívov vytvoríme tzv. konsenzus motív, čo je motív, ktorý obsahuje pre každú pozíciu nukleotid vyskytujúci sa najviac. Proces skladania matice PFM je znázornený na obrázku č. 2.



Obr. 2: *Skladanie matice PFM [17].*

Do matice PFM akceptujeme iba motívy, ktoré majú s konsenzus motívom spoločných aspoň 50% nutkleotidov. Počet akceptovaných motívov tvorí výsledok

jednej z ohodnocovacích funkcií, konkrétne podpory motívu. PFM následne tvoria frekvencie nukleotidov z akceptovaných motívov.

V rovnici 1 uvádzame vzorec na výpočet podobnosti motívu ako jednej z ohodnocovacích funkcií. $f(b,i)$ je frekvencia nukleotidu $b \in \{A,T,G,C\}$ na i-tej pozícii a l je dĺžka motívu.

$$Podobnost = \frac{\sum_{i=1}^{l} max_b f(b,i)}{l} \tag{1}$$

V rovnici 2 je uvedený výpočet komplexity. n_i je počet nukleotidov $i \in \{A,C,G,T\}$ vo výslednej sekvencii motívu.

$$Komplexita = \log_N \frac{l!}{\prod(n_i)!} \tag{2}$$

Posledná ohodnocovacia funkcia je uvedená v rovnici 3. Entropia ohodnocuje frekvencie výskytov podobne ako podobnosť. Odlišujú sa v tom, že entropia berie do úvahy celé rozloženie frekvencií a nie iba najdominantnejší nukleotid.

$$Entropia = -\frac{\sum_{i=1}^{l} \sum_b f(b,i) \log_2(f(b,i))}{l} \tag{3}$$

2.2 Existujúce riešenia

Aj keď existujú algoritmy, ktoré exhaustívne prehľadávajú doménový priestor ako napr. mediánový reťazec, v praxi sa väčšinou využívajú iba algoritmy, ktoré prehľadávajú iba časť doménového priestoru. Tieto algoritmy sú založené na štatistických prístupoch [21]. Medzi ne patrí napríklad Consensus [18] a MEME [3].

Okrem nich existujú aj metaheuristické algoritmy ako MOGAMOD [21], čo je genetický algoritmus alebo MOABC [17], kde využívajú na hľadanie motívov metaforu včiel. Predmetom výskumu sú aj hybridné prístupy ako v [16], kde využívajú viacero metafor naraz.

2.2.1 Weeder

Algoritmus Weeder [31] je založený na prehľadávaní sufixového stromu vytvoreného zo vstupných DNA sekvencií. Sufixový strom [36] pre n znakový reťazec S je stromová dátová štruktúra s n listami. Každá hrana je označená neprázdnym

podreťazcom z S a každá cesta od koreňa po list predstavuje jeden sufix reťazca S.

Algoritmus Weeder hľadá všetky (m, e) vzory, teda motívy o dĺžke m, ktoré sa vyskytujú aspoň v q sekvenciách a podliehajú maximálne e mutáciám. Rekurzívne prehľadáva cesty v sufixovom strome a zahadzuje tie cesty, ktoré prekročili maximálny počet mutácií.

Ohodnotenie motívov

Nad testovacím datasetom od Tompa a spol. [35], kde je úlohou nájsť vo viacerých vstupných sekvenciách motívy s neznámou dĺžkou, museli spustiť vyhľadávanie pre každý vstup viackrát. Použili viaceré hodnoty m od 6 po 12 a e od 1 až 4. q mohlo byť pri k vstupoch rovné k alebo $\lceil k/2 \rceil$ [32]. Takto nájdené motívy dodatočne ohodnocovali, aby nemali príliš veľa výsledkov a vyhli sa tak falošným pozitívam.

Rovnica 4 opisuje tzv. očakávanú frekvenciu[1]. $\mathcal{H}(p, e)$ je množina motívov, ktoré majú Hammingovú vzdialenosť[2]od p menšiu alebo rovnú e. $obs(p')$ je počet výskytov p' v sekvenciách. $total_m$ je počet známych motívov danej dĺžky pre daný organizmus.

$$E(p, e) = \sum_{p' \in \mathcal{H}(p,e)} \frac{obs(p')}{total_m} \tag{4}$$

Skóre špecifické pre sekvenciu je dané nasledovne:

$$Seq(p) = \sum_i \log \frac{1}{E(p, e_i) \cdot l_i} \tag{5}$$

kde p je motív, ktorý sa vyskytuje aspoň v q sekvenciách s maximálnym počtom mutácií, e. l_i je dĺžka i-tej sekvencie a e_i je minimálny počet mutácií, s ktorým sa p vyskytuje v i-tej sekvencii. Ďalej definovali aj globálne skóre, kde $L = \sum l_i$:

$$Glo(p) = \log \frac{obs(p, e)}{E(p, e) \cdot L} \tag{6}$$

[1]Pre krátke motívy používali verziu, ktorá nedovoľovala mutácie.

[2]Hammingová vzdialenosť medzi dvoma reťazcami je rovná počtu miest, kde sa nezhodujú znaky.

Výsledné skóre pre daný motív p je definované ako:

$$Score(p) = Seq(p) + Glo(p) \tag{7}$$

Následne si odložili päť najlepšie ohodnotených motívov a tie sa snažili ešte dodatočne vylepšiť vzájomným spájaním alebo rozširovaním.

Označenie výskytu motívu

Testovací dataset [35] vyžaduje, aby program na výstupe presne uviedol reťazec nukleotidov, ktorý popisuje motív spolu s jeho pozíciou v konkrétnej sekvencii. Motív definovaný ako výsledok prehľadávania (m, e) preto nestačí. Na ohodnotenie jednotlivých reťazcov bolo preto definované skóre výskytov. Toto skóre využíva maticu PFM vytvorenú z reťazcov $\mathcal{H}(p, e)$. Potom pre každý výskyt reťazca p' v sekvenciách vyráta:

$$Occ(p') = \frac{M(p') - Min(PFM)}{Max(PFM) - Min(PFM)} \tag{8}$$

kde $M(p')$ je

$$M(p') = \sum_{j=1}^{m} \log f(b, i) \tag{9}$$

a $f(b, i)$ je hodnota frekvencie[3](z PFM) nukleotidu b na pozícii i daného hodnoteným výskytom p'. $Min(PFM)$ a $Max(PFM)$ sú definované ako súčet minimálnych resp. maximálnych hodnôt jednotlivých stĺpcov v matici PFM. Algoritmus Weeder následne uvádzal na výstupe iba reťazce, ktoré mali toto skóre väčšie ako 0,9.

[3]Nulové frekvencie boli nahradené hodnotou 0,001.

3 Metaheuristiky

Metaheuristiky sú jedným z možných prístupov na riešenie problému optimalizácie. Sean Luke ich vo svojej publikácii *Essentials of Metaheuristics* [26] opísal ako algoritmy, ktoré sú aplikované na tzv. *I know it when I see it* problémy. Tieto algoritmy zvyčajne iteratívnym spôsobom prehľadávajú doménovú oblasť, pričom sa pri každej iterácii snažia zlepšiť nájdené riešenia. Takto generované riešenie po istom čase (keď je splnená ukončovacia podmienka) prehlásia za výsledné riešenie problému. Toto riešenie nemusí byť optimálne, avšak dobrá metaheuristika nájde dostatočne „dobré" riešenie.

Pojmy diverzifikácia a intenzifikácia sú spojené metaheuristickými algoritmami. Intenzifikáciou ovplyvňujeme algoritmus tak, aby riešenie konvergovalo do lokálneho optima. Ak dbáme na intezifikáciu, nájdeme riešenie rýchlejšie, no nemusíme nájsť globálne optimálne riešenie. Diverzifikácia kladie naopak dôraz na rôznorodosť riešení pri spomalení konvergencie.

3.1 Rozdelenie metaheuristík

Christian Blum a Andrea Roli vo svojej práci [5] rozdelili metaheuristiky na dve skupiny – metaheuristiky založené na trajektórii a metaheuristiky založené na populáciách.

3.1.1 Metaheuristiky založené na trajektórii

Metaheuristiky založené na trajektóriách získali svoje meno pretože algoritmus charakterizuje trajektória, ktorú riešenie prejde počas vyhľadávacieho procesu. Úplne najzákladnejší algoritmus je znázornený v 1 [5].

Algorithm 1 Lokálne vylepšovanie - iteratívne prehľadávanie

1: $s \leftarrow$ *GenerujPociatocneRiesenie*()
2: **repeat**
3: $s \leftarrow$ *Zlepsi*(*Susedia*(s))
4: **until** s *sa neda vylepsit*

Funkcia $Zlepsi(Susedia(s))$ vyberá lepšieho suseda riešenia s. Výber zlepšenia môže byť založený na tom, že vyberieme prvého lepšieho suseda, alebo napríklad na tom, že vyberieme najlepšieho zo všetkých.

Obidva tieto prístupy však uviaznú v lokálnom optime a neobsahujú žiaden mechanizmus, ako sa z neho dostať. Tento problém sa snaží riešiť algoritmus simulovaného žíhania, ktorý formuloval Kirkpatrick a spol. v [22] ale aj Vladimír Černý v [6]. Tento algoritmus je znázornený v ukážke 2.

Algorithm 2 Simulované žíhanie

```
1:  s ← GenerujPociatocneRiesenie()
2:  T ← T_0
3:  while nie je splena ukoncovacia podmienka do
4:      s' ← VyberNahodne(Susedia(s))
5:      if f(s') < f(s) then
6:          s ← s'
7:      else if p(f(s'), f(s), T) ≥ random(0, 1) then
8:          s ← s'
9:      end if
10:     T ← Zmensi(T)
11: end while
```

Algoritmus simulovaného žíhania je založený na metafore chladnutia rozžeraveného železa, ktoré, ak sa správne schladzuje, zaujme energeticky najvýhodnejší stav.

Po tom, čo vygenerujeme počiatočné riešenie a určíme počiatočnú teplotu, prebieha iterácia tak, ako je znázornená na riadkoch od 3 po 11. Nové riešenie s' sa zvolí náhodne z okolia s a porovná sa ich ohodnotenie f. Nové riešenie je automaticky akceptované, ak je vyhodnotené ako lepšie. Aby nedošlo k uviaznutiu v lokálnom minime, riešenie s' je akceptované ako nové s pravdepodobnosťou p, aj keď je horšie. Táto pravdepodobnosť závisí od ohodnotení daných riešení ako aj od času T, ktorý uplynul. p je z Boltzmannovej distribúcie $\exp(-((f(s') - f(s))/T)$. Pravdepodobnosť, že bude akceptované ako nové horšie riešenie, klesá jednak s klesajúcim T, ale aj ak vzdialenosť medzi $f(s')$ a $f(s)$ rastie.

3.1.2 Metaheuristiky založené na populáciách

Na rozdiel od trajektoriálnych metaheuristík, populačné metaheuristiky generujú viacero riešení v rôznych oblastiach domény naraz. Sú založené na populáciách jedincov, kde každý predstavuje jedno možné riešenie problému. Nové riešenia sa medzi iteráciami získavajú rekombinovaním elementov z už existujúcich riešení [15]. Medzi tieto algoritmy patria metaheuristiky rojovej inteligencie ale aj rôzne evolučné algoritmy. Príklad jednoduchého evolučného algoritmu je znázornený v ukážke č. 3 [2][5].

Algorithm 3 Evolučný algoritmus

```
1:  P ← InicializujPociatocnuPopulaciu()
2:  Ohodnot(P)
3:  while nie je splena ukoncovacia podmienka do
4:      P' ← Rekombinacia(Selekcia(P))
5:      P'' ← Mutacia(P')
6:      Ohodnot(P'')
7:      VlozDoPopulacie(P, P'')
8:  end while
```

Na začiatku algoritmu 3 vidíme inicializovanie a ohodnotenie populácie. Populáciu tvoria potencionálne riešenia problému. V iteratívom procese (riadok 3 až 8) najprv vystupuje selekcia a rekombinácia riešení. Pod týmito funkciami rozumieme výber rodičov a následnú tvorbu potomkov krížením. Selekcia vo všeobecnosti využíva prvky náhody a ohodnotenie jedinca. Mutácia s určitou pravdepodobnosťou mení potomka. Nakoniec sa nová generácia kombinuje s tou starou. Tento proces sa opakuje, až kým nie je splnená ukončovacia podmienka. Tou môže byť uplynutie času alebo aj nájdenie dostatočne dobrého riešenia.

3.2 Multi-objective metaheuristiky

Existuje veľa metaheuristických algoritmov. Okrem evolučného algoritmu opísaného vyššie, poznáme aj algoritmy rojovej inteligencie ako napríklad Ant Colony Optimization [12], ktorý je inšpirovaný metaforou kolónie mravcov hľadajúcich potravu. Tie vypúšťajú feromóny na ceste, ktorú prešli. V Artificial Bee Colony [19] zasa využívajú metaforu včiel a ich tancovania podľa toho, aký dobrý našli

zdroj potravy. V Glowworm Swarm [23] využívajú svetielkovanie hmyzu, založené na látke luciferín. Jedinec svetielkuje viac vtedy, ak našiel dobré riešenie, a tým láka k sebe ostatných jedincov.

Všetky tieto algoritmy sú ale vo svojom základe definované tak, že využívajú jednu ohodnocovaciu funkciu. V kapitole 2.1.1 sme na druhej strane zadefinovali problém hľadania motívov ako optimalizáciu, ktorá využíva ohodnocovacie funkcie štyri.

Jednou z možností, ako riešiť tento problém, je použiť prístup, ktorý si zvolili v [39]. Ich ohodnotenie jedinca predstavovalo agregáciu všetkých ohodnocovacích funkcií:

$$ohodnotenie = \sum_{i=1}^{m} w_i f_i, \sum_{i=1}^{m} w_i = 1, w_i > 0 \tag{10}$$

kde f_i sú odohodnocovacie funkcie, m je ich počet a w_i sú náhodne zvolené váhy. Existujú ale aj prístupy, ktoré riešia túto problematiku inak. Tie opíšeme v nasledujúcich kapitolách.

Zadefinovanie problému cez viaceré ohodnocovacie funkcie prináša vyššiu zložitosť vo viacerých fázach metaheuristických algoritmov. Vo všeobecnosti je problém porovnať dve riešenia a najlepšie riešenie nemusí byť iba jedno.

3.2.1 Pareto optimálne riešenia

Pareto optimálne riešenia sú také, pre ktoré na to, aby sme zlepšili jednu ohodnocovaciu funkciu, musíme zhoršiť inú. O takýchto riešeniach vieme, že nie sú dominované žiadnym iným riešením. Riešenie a dominuje riešenie b ak platí [25]:

$$\forall i \in \{1, \ldots, m\} : f_i(a) \geq f_i(b) \wedge \exists j \in \{1, \ldots, m\} : f_j(a) > f_j(b) \tag{11}$$

kde f_i je ohodnocovacia funkcia a m je ich počet. Teda jedno riešenie dominuje iné, ak nie je horšie v žiadnom ohodnotení a aspoň v jednom je lepšie. Takéto riešenia tvoria paretový rad. Cieľom metaheuristických algoritmov je tento rad aproximovať.

V [24] sa venovali problematike udržovania paretových radov. V ukážke 4 opísali metaheuristiky, ktoré pracujú s týmito radmi ako s archívom riešení.

V tomto pohľade na metaheuristiky $A^{(t)}$ predstavuje archív a $f^{(t)}$ predstavuje

Algorithm 4 Iterácie metaheuristík s archívom

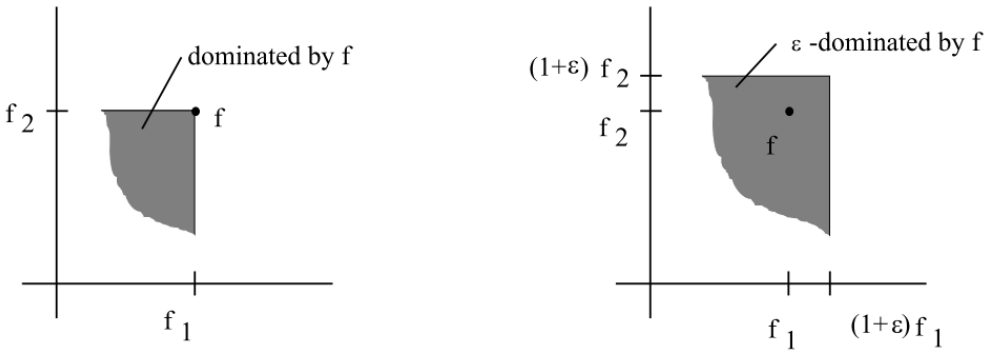
1: $t \leftarrow 0$
2: $A^{(0)} \leftarrow \emptyset$
3: **while** *ukoncit?*$(A^{(t)}, t) = false$ **do**
4: $t \leftarrow t + 1$
5: $f^{(t)} \leftarrow$ *generuj*()
6: $A^{(t)} \leftarrow$ *aktualizuj*$(A^{(t-1)}, f^{(t)})$
7: **end while**

množinu jedincov vygenerovaných v iterácií t. Metóda $generuj$ predstavuje proces tvorby nových riešení a $aktualizuj$ sa venuje udržiavaniu archívu. Na základe stratégie archívovania vieme potom metaheuristiky rozdeliť na:

- agloritmy, ktoré sa zameriavajú na konvergenciu – tieto algoritmy vo všeobecnosti akceptujú riešenia do archívu na základe toho, či sú dominované riešeniami v archíve alebo samy nejaké nedominujú,
- agloritmy, ktoré sa zameriavajú na distribúciu – tieto algoritmy sa rozhodujú nie len na základe dominancie, ale aj na základe rozloženia riešení. Napríklad v NSGA-II [9] využívajú vzdialenosť riešení na dodatočné ohodnotenie jedinca.

Obidva tieto prístupy majú svoje opodstatnenia. Zatiaľ čo druhé riešenie dbá na to, aby boli výsledky distribuované a teda paretový rad bol dobre aproximovaný, podlieha chybe zhoršenia riešení. Prvý prístup garantuje, že sa nemôže stať, aby sa riešenie na paretovej krivke zhoršilo [24], pričom druhý prístup môže hodnotenie najlepšieho lokálneho riešenia znížiť kvôli distribúcii a potom ho stratiť.

V [24] a [25] sa následne venovali práci s paretovými radmi a dominancii. Využívajú koncept ϵ-dominancie, ktorý je znázornený na obr. č. 3, na zlepšenie vlastností paretových radov.



Obr. 3: *Rozdiel medzi dominanciou a ϵ-dominanciou [24].*

3.2.2 Multi-objective Artificial Bee Colony

Artificial Bee Colony (ABC) [19] je algoritmus rojovej inteligencie, ktorý na prehľadávanie doménového priestoru využíva metaforu včiel. Definuje tri tipy:

- pracovníčky, ktoré sa snažia zmeniť svoje riešenie k lepšiemu,
- prizeračov, ktorí si vyberú riešenie nejakej pracovníčky a to sa snažia následne zlepšiť,
- prieskumníčky, ktoré prinášajú do kolónie úplne nové riešenia.

Tieto typy sa v rámci iterácií striedajú v troch fázach (fáza pracovníčiek, fáza prizeračov a fáza prieskumníčok). Poloha týchto včiel predstavuje možné riešenie problému.

V MOABC [17], rozšírili tento algoritmus pomocou zoraďovania z NSGA-II [9], ktoré využíva paretové rady, aby mohol ABC riešiť problém hľadania motívov tak, ako sme ho opísali v kapitole 2.1.1. Ich prístup je príkladom, ako sa dá metaheuristika postavená na jednom ohodnotení rozšíriť na tzv. *multi-objective* metaheuristiku.

4 Paralelizmus v metaheuristikách

Základnou motiváciou na paralelizovanie metaheuristík môže byť zrýchlenie ich behu. V obidvoch typoch spomínaných v kapitole 3.1.1 a 3.1.2 nájdeme vnútorné cykly, ktoré sa dajú paralelizovať. Hľadanie najlepšieho suseda v trajektoriálnych metaheuristikách vieme rozdeliť medzi procesy na základe prehľadávaného priestoru. Pri populačných algoritmoch vieme na druhej strane paralelizovať ohodnocovanie jedincov. Takéto optimalizácie na nízkej úrovni ale nemusia byť dostačujúce. V našej implementácii MOABC [17] algoritmus trávi najviac času (viac ako polovicu[4]) pri zoraďovaní populácie[5]. Kedže paralelizácia tohto zoraďovania je netriviálna, algoritmus sme paralelizovali na tej najvyššej úrovni, teda spúšťali sme viaceré inštancie vyhľadávania paralelne nezávisle od seba.

Nami zvolený prístup ale nijako nezlepšil výsledky vyhľadávania a práve to môže byť druhá, silnejšia motivácia pre paralelizáciu. V [16] napríklad použili pri hľadaní motívov paralelne bežiace metaheuristiky, ktoré v určitých bodoch synchronizujú svoje paretové rady riešení na to, aby zlepšili výsledky vyhľadávania. Takýto prístup zlepšuje výsledky, pretože rôzne algoritmy spolu interagujú a môžu si vymieňať elitných jedincov.

V nasledujúcich kapitolách opíšeme rôzne prístupy pri paralelizácií metaheuristických algoritmov.

4.1 Zdroje paralelizmu

V [15] opisujú hlavne zdroje paralelizmu a ich aplikovateľnosť v metaheuristikách. Paralelizmus ako súbežné vykonávanie viacerých procesov je založený na rozdelení záťaže medzi samostatné jednotky. Takáto dekompozícia sa da docieliť na viacerých úrovniach:

- algoritmus – *functional paralelism* – viaceré úlohy, ktoré môžu pracovať nad rovnakou množinou dát a vymieňať si údaje, sú rozdelené medzi procesy tak, aby bežali paralelne,

[4] Pri niektorých behoch až 80% v závislosti od konfigurácie.
[5] Tento čas sme merali pomocou IDE Netbeans a jeho funkcie Profiler.

- dáta problémovej oblasti – *data parallelism / domain decomposition* – delená je problémová doména alebo prehľadávaný priestor,

- štruktúra problému – *mathematical / attribute-based decomposition* – rozdelenie problému na základe atribútov, kde niektoré úlohy riešia podproblémy (podmnožinu atribútov) a ostatné kombinujú tieto riešenia na riešenie celého problému.

V metaheuristikách je paralelizácia na úrovni vnútorných cyklov častokrát najjednoduchšia. Na druhej strane paralelizované kroky bývajú na sebe časovo závislé a nároky na synchronizáciu jej prínos znižujú.

Zaujímavé je paralelizovať prehľadávanie na úrovni problémovej oblasti, kde jedným zo spôsobov je explicitne rozdeliť prehľadávaný priestor medzi jednotlivé procesy. Druhou možnosťou je deliť priestor implicitne pomocou metódy tzv. *multi-search*, kde využívame možnosť aplikovať na doménu viaceré algoritmy s rôznymi spôsobmi prehľadávania. S obidvoma prístupmi prichádza problém riadenia celého vyhľadávacieho procesu, pričom pri prvom prístupe existuje ešte možnosť, že proces bude prehľadávať oblasť s malou kvalitou riešení. Na druhej strane, pri implicitnom delení neexistuje garancia, že bude problémová oblasť rozdelená tak, aby sa jednotlivé časti neprekrývali.

4.2 Klasifikácia paralelných stratégií

V [15] použili klasifikáciu z [7], ktorá delí paralelné metaheuristiky na základe troch dimenzií:

1. Kardinalita ovládania prehľadávania (angl. Search Control Cardinality) – táto dimenzia opisuje koľko procesov riadi priebeh vyhľadávania.

   (a) *1-control* (1C) – prehľadávanie riadi jeden proces,

   (b) *p-control* (pC) – prehľadávanie je riadené viacerými procesmi, ktoré spolu môžu ale nemusia spolupracovať.

2. Riadenie a komunikovanie (angl. Search Control and Communications) – zvyčajne sa v tejto dimenzii delia algoritmy na synchrónne – procesy

musia zastaviť v dohodnutom čase a potom komunikovať – a asynchrónne, kde si každý proces riadi vyhľadávanie sám a rozhoduje sa, kedy chce komunikovať. V [15] tieto triedy rozšírili a rozdelili nasledovne:

(a) *Rigid* (RS) – procesy medzi sebou počas vyhľadávania nekomunikujú,

(b) *Knowledge Synchronization* (KS) – procesy si na dohodnutých miestach zosynchronizujú vedomosti o doménovom priestore napr. zjednotia populácie ponechaním iba najlepších,

(c) *Collegial* (C) – asynchrónne komunikujú so susedmi a vymieňajú si riešenia,

(d) *Knowledge Collegial* (KC) – snažia sa získať vedomosti z komunikácie, napríklad tvoriť nové riešenia z častí komunikovaných riešení.

3. Delenie prehľadávania (angl. Search Differentiation) – táto dimenzia delí algoritmy podľa toho, či paralelné procesy začínajú z rovnakého miesta alebo populácie a či používajú rovnaký algoritmus na prehľadávanie doménového priestoru:

   (a) *Same initial Point/Population, Same search Strategy* (SPSS),

   (b) *Same initial Point/Population, Different search Strategies* (SPDS),

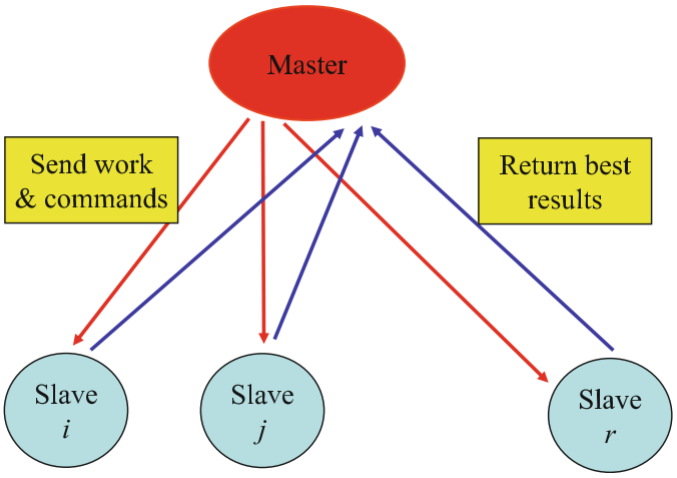
   (c) *Multiple initial Points/Populations, Same search Strategies* (MPSS),

   (d) *Multiple initial Points/Populations, Different search Strategies* (MPDS).

V nasledujúcich kapitolách opíšeme vybrané prístupy, klasifikované v rámci týchto dimenzií.

4.2.1 1C paralelizácia na nízkej úrovni

Tento druh paralelizácie je založený na modeli *master-slave*, ktorý je znázornený na obrázku č. 4. Jeden proces kontroluje celý priebeh vyhľadávania, zatiaľ čo ostatné procesy spracúvajú nízko úrovňové výpočty. V populačných algoritmoch *master* proces rieši procesy ako výber rodičov a pohyb jedincov. *Slave* procesy riešia ohodnotenia a posielajú výsledky *master*-ovi. V trajektoriálnych metaheuristikách môžu zasa procesy paralelne hľadať a ohodnocovať suseda daného riešenia. Takáto paralelizácia patrí do triedy 1C/RS/SPSS.



Obr. 4: *Konfigurácia master-slave [15].*

4.2.2 Nezávislé multi-vyhľadávanie

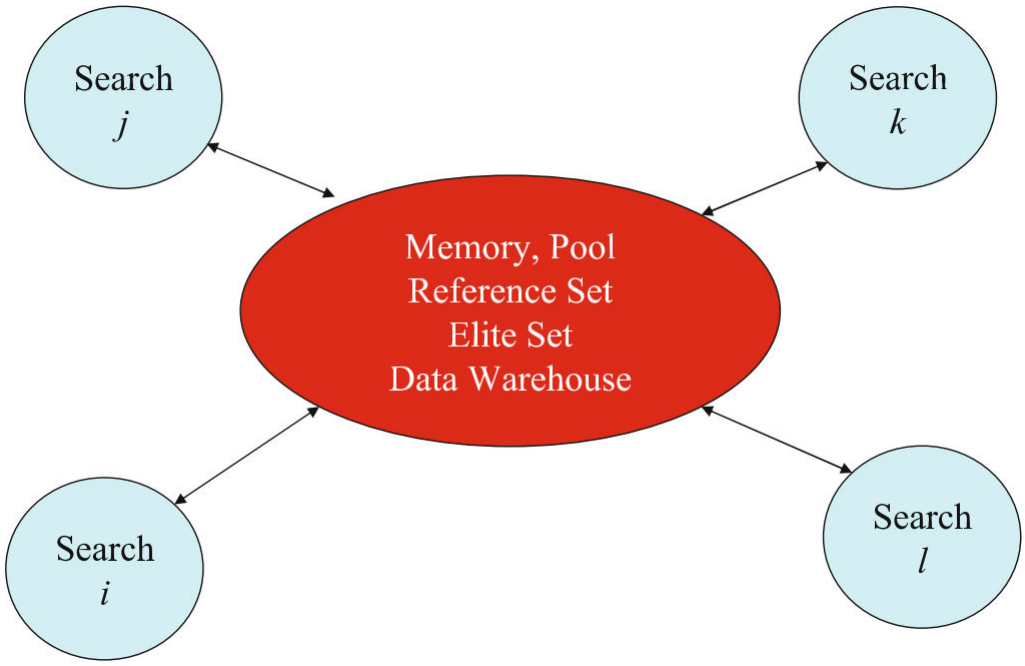
V [15] definujú toto vyhľadávanie ako stratégiu, kde prebieha nezávisle od seba niekoľko vyhľadávaní v rámci celého doménového priestoru. Po skončení prehľadávania sa vyberie najlepšie riešenie zo všetkých prehľadávaní. Procesy spolu nijako inak nekomunikujú. Algoritmy, ktoré sa riadia touto stratégiou, sú v triede pC/RS a môžu byť v hociktorej triede delenia prehľadávania.

4.2.3 Kooperátívne stratégie

Tieto stratégie majú za cieľ pomocou interakcií medzi nezávislými procesmi vyhľadávania zlepšiť kvalitu výsledkov. Napríklad si môžu vymieňať doteraz najlepšie získané výsledky. Jedným zo spôsobov, ako docieliť takúto výmenu, je pomocou tabúľ (angl. blackboard), teda nejakej formy zdieľanej pamäte ako je znázornené na obrázku č. 5. Takáto tabuľa môže byť jedna, alebo ich môže byť viacero.

Jedným z príkladov kooperatívnych algoritmov pC/C je paralelný evolučný algoritmus založený na ostrovoch, ktorý bol v [28] opísaný tak, ako je uvedený v ukážke 5.

Tento algoritmus obsahuje E hlavných iterácii zvaných epochy. Na riadku 6 až 8 vidíme, ako v rámci tejto epochy prebieha paralelne v procesoch sekvenčný evolučný algoritmus nad podpopuláciou P_i o G_i iteráciách. Následne na riadkoch 9



Obr. 5: *Model zdieľanej pamäte[15].*

Algorithm 5 Model ostrovov v EA

```
1:  procedure MODELOSTROVOV(E,N,μ)
2:      paralelne for i ← 1 to N do
3:          Inicializuj(P_i, μ)
4:      end for
5:      for epocha ← 1 to E do
6:          paralelne for i ← 1 to N do
7:              Sekvencna_EA(P_i, G_i)
8:          end for
9:          for i ← 1 to N do
10:             for each j ∈ Susedia(i) do
11:                 Migracia(P_i, P_j)
12:             end for
13:             Asimiluj(P_i)
14:         end for
15:     end for
16: end procedure
```

až 12 prebieha proces migrácie a následnej asimilácie jedincov. Tieto dva procesy na sebe závisia a sú špecifické pre implementáciu. Napríklad pri migrácii jedinca z P_i do P_j ho môžeme z P_i vymazať alebo nie. Ak to tak neurobíme, asimilácia musí po skončení migrácií populáciu redukovať, aby sa zachovala jej veľkosť.

4.3 Asynchrónny model ostrovov

Asynchrónny model ostrovov je modifikácia modelu ostrovov načrtnutého v kapitole 4.2.3. Tento model využívali napríklad v [27], kde ho použili spolu s algoritmom NSGA-II [9] na návrh trajektórie medziplanetárneho letu. V [30] ho využívali takým spôsobom, že každý ostrov vylepšoval podmnožinu rozhodovacích premenných a kompletné riešenia boli vyhodnotené pomocou výmeny jedincov medzi ostrovmi. Ich asynchrónny model je opísaný v algoritme 6.

Algorithm 6 Asynchrónne CCMOEA

```
1:  t ← 0
2:  paralelne for i ← 1 to N do
3:      Inicializuj(P^0, i)
4:  end for
5:  for i ← 1 to N do
6:      Zdielaj(P^0, i)
7:  end for
8:  paralelne for i ← 1 to N do
9:      Ohodnot(P^0, i)
10: end for
11: synchronizuj()
12: paralelne while not ukoncovaciaPodmienka()
13:     Generuj(P^t, i)
14:     Zdielaj(P^t, i)
15:     t ← t + 1
16: end while
17: spojParetoveRady()
```

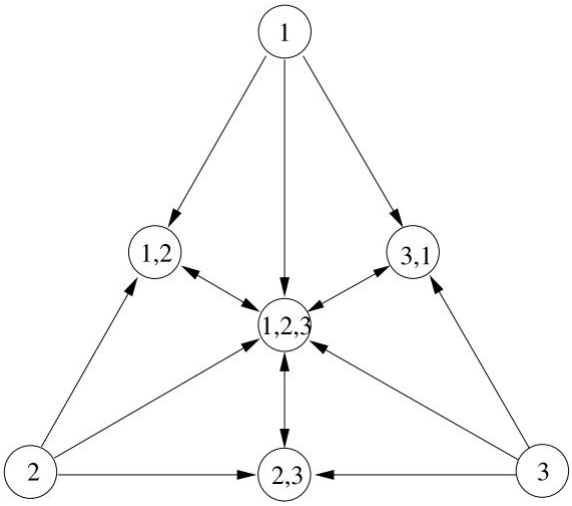
Na riadkoch 1 až 11 prebieha inicializácia algoritmu. Najprv na riadku 3 algoritmus vytvorí podpopulácie. Potom sa na riadku 6 prešíria čiastočné riešenia

a následne sa ohodnotia riešenia v podpopuláciach. Podstatné pre opis asynchrónneho modelu ostrovov sú riadky 12 až 17, kde sa paralelne generujú nové populácie a následne každý ostrov asynchrónne zdieľa svoje riešenia. Po ukončení iterácií algoritmus spojí paretové rady všetkých ostrovov.

Výhodou asynchrónneho prístupu je jeho zrýchlenie oproti normálnemu prístupu, pri ktorom sa musia ostrovy čakať. Nevýhodou je nižšia kontrola nad vyhľadávaním a riziko, že niektorý z ostrovov iteruje rýchlejšie ako ostatné.

4.4 Špecializovaný model ostrovov

Špecializovaný model ostrovov (SIM) [38] je taký model, kde každý ostrov optimalizuje inú podmnožinu ohodnocovacích funkcií s cieľom zlepšiť diverzifikáciu vyhľadávania. V [38] testovali viacero možností usporiadania ostrovov a ich topológie a zistili, že usporiadanie na obrázku č. 6 bolo pre nimi testované problémy najoptimálnejšie.



Obr. 6: *Príklad SIM pre 3 ohodnocovacie funkcie[38].*

5 Perzistentné dátové štruktúry

Rich Hickey[6] v jednej svojej prezentácii[7] na tému stavu a času uviedol zaujímavý príklad s kontrolovaním dodržiavania pravidiel v pretekoch v atletickej chôdzi. Pretekári porušia pravidlá, ak sa počas celého behu nedotýkajú zeme aspoň jednou nohou, teda nemôžu mať vo vzduchu obidve nohy naraz. Ak by sme programovali kontrolovač behu pretekára, proces by zrejme vyzeral nasledovne – spýtali by sme sa prvej premennej, či je pravá noha vo vzduchu a dostali by sme kladnú odpoveď. Následne by sme sa spýtali druhej premennej, ľavej nohy, či je vo vzduchu a dostali by sme opäť kladnú odpoveď. Na otázku, či pretekár porušil pravidlá, ale nevieme odpovedať. Museli by sme poznať, akú mali hodnotu tieto premenné naraz v jednom čase.

Toto je paralela s viacniťovým programovaním, kde viaceré paralelné algoritmy neustále menia stav nejakých premenných. Počas pretekov nechceme súťažiaceho zastaviť len preto, aby sme skontrolovali, či dodržuje pravidlá tak, ako častokrát nechceme zastaviť bežiace procesy. Jedným z riešení je odfotiť si pretekára, teda priradiť k hodnotám premenných aj čas, kedy ich nadobudli. S programovacieho hľadiska nám takéto fotky umožňujú robiť perzistentné dátové štruktúry. Sila týchto štruktúr spočíva v tom, že pri každej zmene dostaneme úplne novú štruktúru. Pri paralelnom programovaní to napríklad znamená, že pri niektorých operáciach nepotrebujeme zámky a iné synchronizačné nástroje. Každý proces alebo vlákno si vypýta aktuálny stav (fotku premenných) a s nimi následne pracuje, aj keď ich ostatné procesy počas toho znova menia.

5.1 Path copying

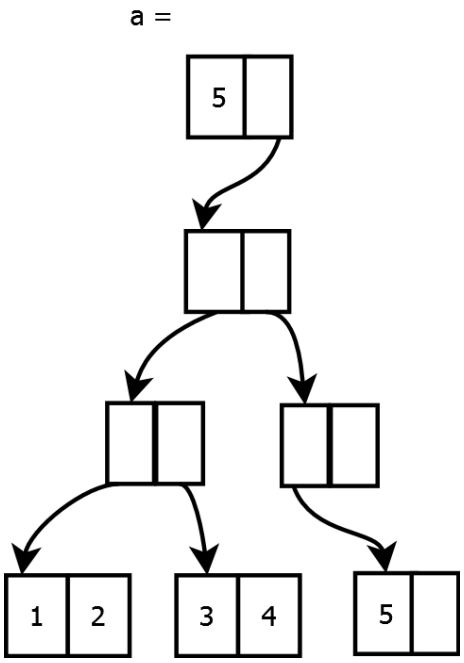
Predstavme si, že máme jednorozmerné pole prvkov. Na to, aby sme vedeli vytvoriť jeho fotografiu za každým, keď ho zmeníme, by sme museli vytvárať jeho kópie. Kopírovanie n-prvkového poľa po každej zmene nie je časovo optimálne. Našťastie, nemusíme kopírovať celé pole, ale iba zmenené časti. Jedným zo spôsobov, ako docieliť zdieľanie prvkov medzi verziami poľa, je použitie metódy tzv.

[6] Autor programovacieho jazyka Clojure

[7] http://www.infoq.com/presentations/Valu-Identity-State-Rich-Hickey

Path copying [20]. Základnou myšlienkou tohto prístupu je udržiavanie si vektora v stromovej dátovej štruktúre.

Majme pole prvkov $a = [1, 2, 3, 4, 5]$. Jeho perzistentná reprezentácia použitím binárnych stromov je znázornená na obrázku č. 7.

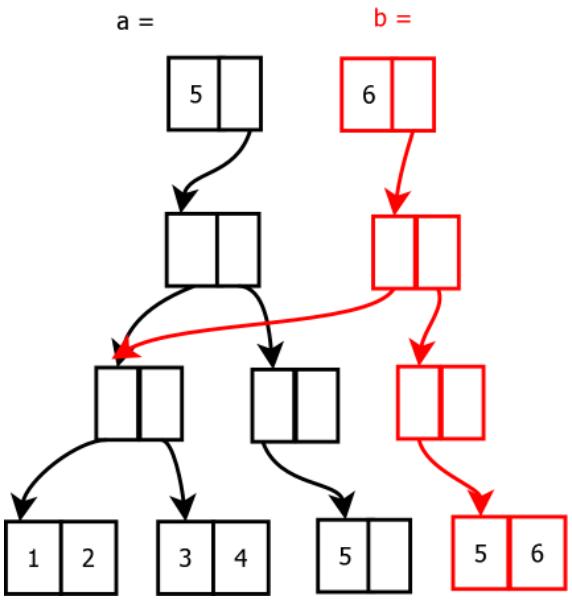


Obr. 7: *Vektor a reprezentovaný binárnym stromom*[8].

Vrchol stromu obsahuje dĺžku poľa a smerníky na ostatné časti. Dáta sú uložené v listoch stromu. Ak by sme toto pole zmenili napríklad pridaním ďalšieho prvku $b = a.push(6)$, dostali by sme nové pole, tak ako je znázornené na obrázku č. 8. Každá zmena poľa a vytvorí úplne nové pole, v tomto prípade je to b znázornené červenou farbou.

Pri takejto práci so štruktúrami nevytvárame skutočné kópie celého poľa ale iba kópie cesty. Analogicky fungujú aj ďalšie operácie ako zmena prvku alebo jeho odobratie.

[8] http://hypirion.com/musings/understanding-persistent-vector-pt-1



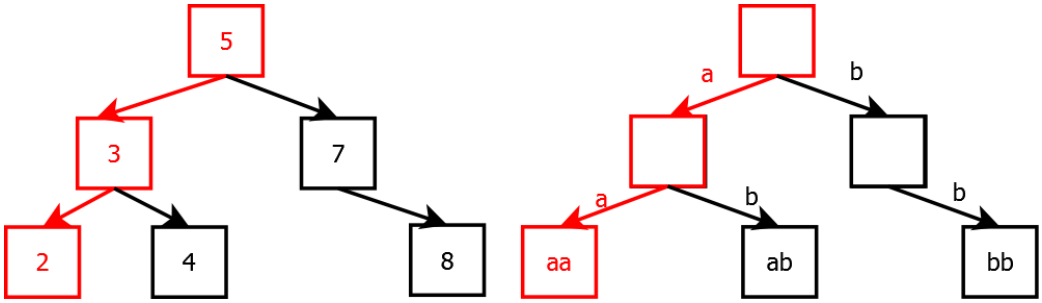
Obr. 8: *Pole a a b po pridaní prvku.*

5.2 Trie

Vyhľadávanie prvkov v takomto strome je založené na stromoch zvaných *Trie* [14]. Pri klasických binárnych stromoch je prehľadávanie postavené na tom, že hodnoty menšie ako hodnota vrchola sa nachádzajú v ľavom podstrome a hodnoty väčšie naopak v pravom. V trie stromoch je nájdenie prvku založené na kľúči – napríklad na postupnosti písmen, ako je znázornené na obrázku č. 9.

Vyhľadávanie vrchola v binárnom strome, ktorý je na obrázku č. 9 vľavo, predstavuje rekurzívne vyberanie správneho podstromu na základe hodnoty vrchola, až kým sa nedostaneme po hľadaný vrchol. Na obrázku je znázornené hľadanie vrchola 2, kde na základe hodnôt ideme stále do ľavého podstromu.

V pravej časti obrázka je znázornený strom trie a kľúč „aa“. Pri hľadaní prvku podľa kľúča sa najprv vyberieme po vetve „a“ a následne znovu po vetve „a“.



Obr. 9: *Nájdenie prvku v binárnom strome a strome trie.*

5.3 Persistent bit-partitioned vector trie

V Clojure implementácií takýchto perzistentných štruktúr využívajú stromy, ktoré sa volajú *Persistent bit-partitioned vector trie*. Tieto stromy su veľmi podobné stromom opísaným vyššie, líšia sa však v niektorých vlastnostiach:

- počet hodnôt uložených v jednom vrchole je 32 – takáto reprezentácia spôsobuje, že stromy sú plytké a tučné,
- kľúčom k prvku je 32-bitové číslo – na adresovanie jednej vetvy potrebujeme 5 bitov z tohto kľúča, teda maximálna hĺbka takéhoto stromu je 6. Počet prvkov, ktoré sa dajú v takomto strome uložiť, je dostatočne veľký vzhľadom na hardvérové obmedzenia a zložitosť prístupu je maximálne $O(6)$,
- *tail optimisation*[9] – tieto stromy využívajú optimalizáciu, kde vrchol ukazuje na koncový vrchol priamo. Pri takomto usporiadaní pridávanie prvku do poľa znamená, že vytvárame iba dva vrcholy – koreň a koncový vrchol. Novú cestu treba vytvárať, iba ak sa koncový vrchol zaplní celý. To sa pri 32-prvkových vektoroch stáva pri každom 32. pridaní.

[9] http://hypirion.com/musings/understanding-persistent-vector-pt-3

6 Návrh algoritmu

V [15] spomínajú 6 problémov, ktorým sa treba venovať pri návrhu paralelných kooperatívnych metaheuristík:

- aké informácie sa prenášajú medzi procesmi,
- medzi ktorými procesmi sa tieto informácie prenášajú,
- kedy sa tieto informácie prenášajú,
- akým spôsobom,
- ako proces narába so získanými informáciami,
- sú informácie menené v rámci prenosu alebo sú vytvárane úplne nové informácie.

V nasledujúcich riadkoch opíšeme návrh algoritmu a budeme sa snažiť tieto otázky zodpovedať.

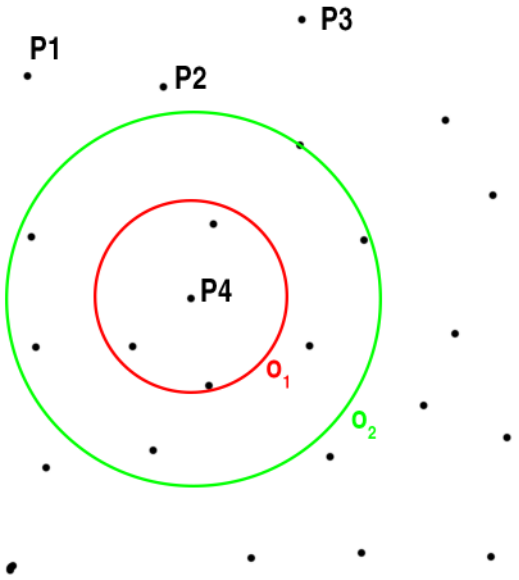
Prvotnou ideou, s ktorou sme pracovali, bolo rozdelenie ohodnocovacích funkcií. Rozhodli sme sa ich rozdeliť do rôzne veľkých skupín. Týchto skupín je $2^n - n - 1$, kde n je počet ohodnocovacích funkcií a dokopy tvoria každú podmnožinu množiny ohodnocovacích skupín, okrem prázdnej množiny a množín, ktoré obsahujú iba jednu funkciu. Ak teda O je množina ohodnocovacích funkcií, potom definujeme množinu F, takú, že $F = \{F_i, F_i \in P(O) \wedge |F_i| > 1\}$, kde $P(x)$ je potenčná množina.

Pri čítaní článkov sme našli paralelu tohto prístupu v dekompozícií multi-objective ohodnotení pomocou vážených vektorov. V SDMOPSO [1] rozšírili Particle Swarm Optimisation na multi-objective takým spôsobom, že každému jedincovi priradili iný vektor váh, pomocou ktorého sčítaval ohodnocovacie funkcie a agregát následne optimalizoval. Každý jedinec teda optimalizoval single-objective problém len s inými náhodne zvolenými váhami, pričom dva jedince nemali rovnaké váhy. Ich motiváciou bola aproximácia paretového radu, kde každý jedinec inklinoval k nejakej jeho časti na základe svojho vektora váh. Je vhodné spomenúť, že v [1] používali aj iné mechanizmy, avšak našu paralelu tvorí

fakt, že potláčame niektoré ohodnocovacie funkcie na to, aby sme dosiahli čo najlepšiu diverzifikáciu a aproximáciu paretového radu. Tento prístup je podobný modelu SIM, ktorý sme opísali v kapitole 4.4.

Máme teda $2^n - n - 1$ procesov P_i kde každý pracuje s nejakou množinou ohodnocovacích funkcií $F_i \in F$. Algoritmus, ktorý navrhujeme, je asynchrónny a decentralizovaný, teda v klasifikácii opísanej v kapitole 4.2 zapadá do pC a do niektorej z kolegiálnych kategórií.

Komunikácia medzi procesmi prebieha asynchrónne a s pravdepodobnosťou p_c po každej iterácii algoritmu. Komunikujeme difúziou so susednými procesmi. Tieto procesy budu náhodne rozložené v 2D priestore tak, ako je znázornené na obrázku č. 10. Táto komunikácia sa dá opísať nasledovnými bodmi:



Obr. 10: *Procesy P_i v 2D priestore a ich dosah.*

- Proces P_i má dosah iba na susedov vo svojom okolí o_t. Toto okolie sa časom mení tak, ako je znázornené na obrázku č. 10, kde proces P_4 na začiatku ovplyvňuje procesy v červenom kruhu (o_1) a po istom čase sa jeho okolie zväčší na zelený kruh (o_2).

- Procesy ponúkajú ostatným procesom celý paretový rad. Prijímajúci proces

sa teda môže rozhodnúť, ktorého jedinca si vyberie. V našej implementácii to je prvý jedinec v rade, no napríklad v [27] využívajú pri výbere jedinca metriku *crowding distance* z NSGA-II [9].

- Veľkosť okolia, s ktorým komunikujú o_t aj pravdepodobnosť p_c ako často komunikujú, časom rastie. Toto je paralela so simulovaným žíhaním, kde na začiatku je podporovaná diverzita a neskôr chceme, aby algoritmus konvergoval. Takto procesy na začiatku môžu konvergovať podľa svojich ohodnocovacích funkcií, ale neskôr budú inklinovať k implicitnému globálnemu optimu.

Komunikácia prebieha pomocou tabúľ, teda existuje centralizovaná pamäť, kde sú pomocou perzistentných dátových štruktúr uložené ponúkané riešenia spolu s pozíciou, kde boli vytvorené. Proces P_i po skončení iterácie ponúkne ostatným procesom riešenie x_i s pravdepodobnosťou p_c. Ostatné procesy toto riešenie vidia keď komunikujú a vyberú si ho s pravdepodobnosťou, ktorá závisí od vzdialenosti medzi P_j a P_i. Proces P_j si môže vybrať jedno alebo viacero riešení. Pravdepodobnosť výberu x_i procesom P_j rastie lineárne no existuje možnosť použitia gausovej distribúcie.

V prehľadávacích procesoch beží sekvenčná multi-objective metaheuristika. Krok jedinca resp. jeho zmena musí obsahovať prvky, ktoré využívajú najlepšieho doteraz nájdeného jedinca, alebo rôzne pamäťové prvky, kde si jedince pamätajú doteraz najlepšie riešenie. V algoritme Gbest-guided ABC [40] využívali na pohyb jedinca rovnicu 12.

$$v_{ij} = x_{ij} + \phi_{ij}(x_{ij} - x_{kj}) + \Psi_{ij}(y_j - x_{ij}) \tag{12}$$

Táto rovnica popisuje pohyb včely, kde v je nová pozícia a x je stará. Index i a j popisuje, o ktorú včelu sa jedná a o aký prvok vektora. ϕ_{ij} je náhodné číslo medzi [-1,1] a Ψ_{ij} je náhodné číslo medzi [0,C]. Okrem prvej časti rovnice pridali aj člen, v ktorom sa vyskytuje y, čo je najlepšie dosiahnuté riešenie. To teda smeruje riešenia k nejakému určenému bodu. V našom prípade to je k bodom, ktoré proces prijal od susedov. Pravdepodobnosť, s akou sú jedince „ťahané“, sa môže časom meniť. Alternatíva k tomuto riešeniu je, aby proces začlenil všetky jedince, ktoré

prijme do svojej populácie.

Z globálneho pohľadu na prehľadávanie existuje proces, ktorý spravuje tabuľu. Keďže procesy ponúkajú svoje najlepšie riešenia, dá sa na ne pozerať ako na riešenia, ktoré aproximujú paretový rad celého prehľadávania. Výsledný rad, teda bude zložený z riešení, ktoré boli použité pri komunikácii.

7 Implementácia

Algoritmus sme sa rozhodli implementovať v programovacom jazyku Clojure[10]. Ide o funkcionálny jazyk bežiaci v Java JVM. Tento jazyk natívne pracuje s perzistentnými dátovými štruktúrami opísanými v kapitole č. 5 a obsahuje aj iné mechanizmy na zjednodušenie paralelizácie[11].

Pri implementácii sme dbali na to, aby sme využívali prevažne iba čisté funkcie. Funkcia je čistá, ak jej výsledok závisí iba od jej vstupu a počas svojho vykonávania nemá žiadne vedľajšie účinky. Takéto funkcie sa ľahšie testujú, no primárnym dôvodom pre ich používanie bolo zjednodušenie paralelizácie. Čisté funkcie môžeme bezpečne používať v paralelnom prostredí, kedže nijako nezmenia svoj vstup a ani nepristupujú do žiadnej kritickej oblasti.

Tento pohľad na funkcie ovplyvňuje aj návrh celého systému. Entity ako jedinec (riešenie) alebo ostrov by boli v OOP klasickými triedami, ktoré by mali svoj stav a metódy. Volanie týchto metód by menilo stav danej entity. Pri funkcionálnom programovaní sa ale na tieto entity pozeráme ako na dáta. Jedinec alebo aj ostrov sú dáta, ktoré nejaká funkcia pretvorí na iné dáta.

Funkcionálna paradigma, ktorá využíva perzistentné dátové štruktúry opísané v kapitole č. 5 a čisté funkcie, nám umožňuje paralelné programovanie pri minimálnom použití zámkov.

V nasledujúcich kapitolách opíšeme návrh a implementáciu ostrova opísaného v kapitole č. 6. Ostrov musí vedieť fungovať v paralelnom prostredí, kde si budú ostrovy medzi sebou vymieňať jedincov asynchrónne. Musí vedieť pracovať s ľubovolným optimalizačným problémom a ľubovoľnou metaheuristikou.

7.1 Jedinec

Jedinec predstavuje jedno možné riešenie problému, ktorý sa snažíme optimalizovať. V ukážke č. 1 je znázornená naša implementácia jedinca ako záznamu dát, bez vlastných metód alebo funkcionality.

[10] https://clojure.org/

[11] Futures, Promises, Agents a iné

S týmto jedincom musí vedieť pracovať čo najviac metaheuristík a problémov, preto sme sa ho snažili navrhnúť čo najvšeobecnejšie. Skladá sa z atribútov, ktoré opisujú jeho ohodnotenie, či už normalizované alebo nie, pozíciu v doménovom priestore alebo to, či je validný. Jedinec je validný, ak spĺňa ohraničenia dané problémom a zároveň je jeho pozícia v hraniciach doménového priestoru. Každá metaheuristika si môže navyše ukladať pre ňu špecifické dáta, napríklad počet pokusov o zlepšenie pozície a pod.

```
{:objectives {:original {}
              :normalized {}}
 :position []
 :data {}
 :valid false}
```

Ukážka 1: *Jedinec*[12].

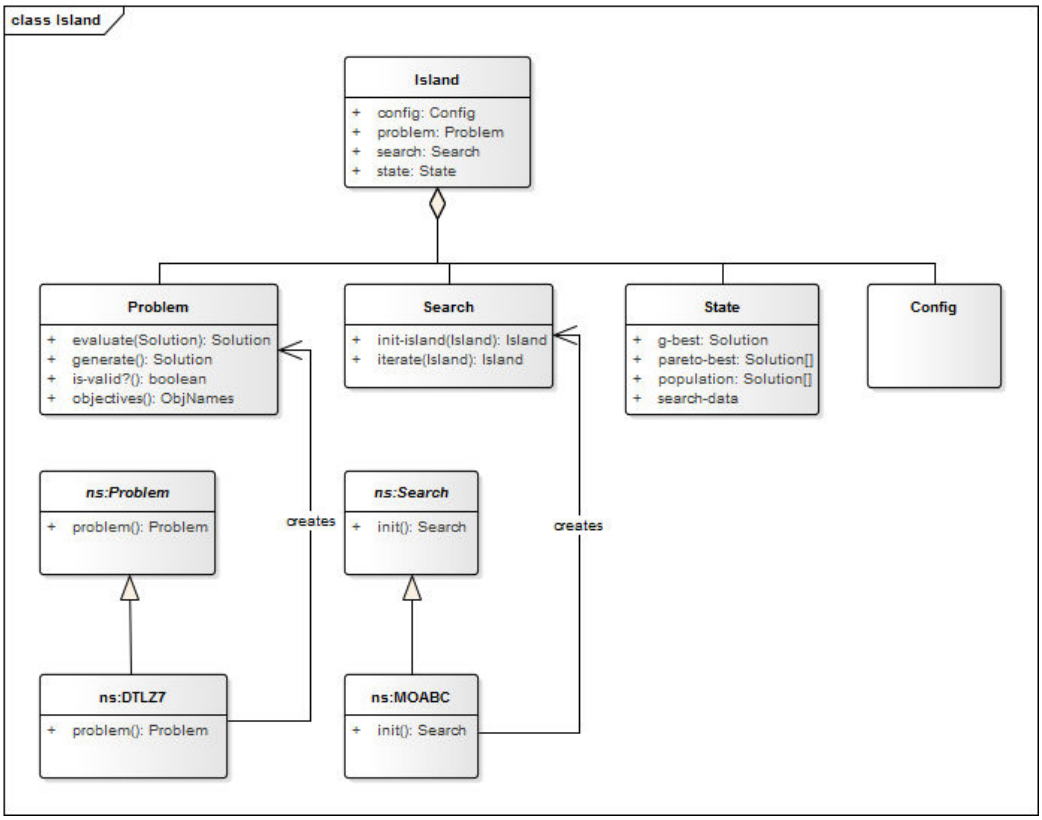
7.2 Ostrov

Ostrov je základná jednotka nášho algoritmu, ktorý bol opísaný v kapitole č. 6. Každý ostrov slúži ako abstrakcia, ktorá vykonáva jeden konkrétny optimalizačný algoritmus nad daným problémom. Aj keď náš algoritmus nie je programovaný v paradigme OOP, použili sme diagramy tried na to, aby sme opísali jednotky programu a ich vzťahy. Diagram, ktorý opisuje ostrov, je možné vidieť na obrázku č. 11.

Každý ostrov je definovaný štyrmi jednotkami:

- *config*
- *problem*
- *search*
- *state*

[12] Vo formáte EDN – http://edn-format.org/

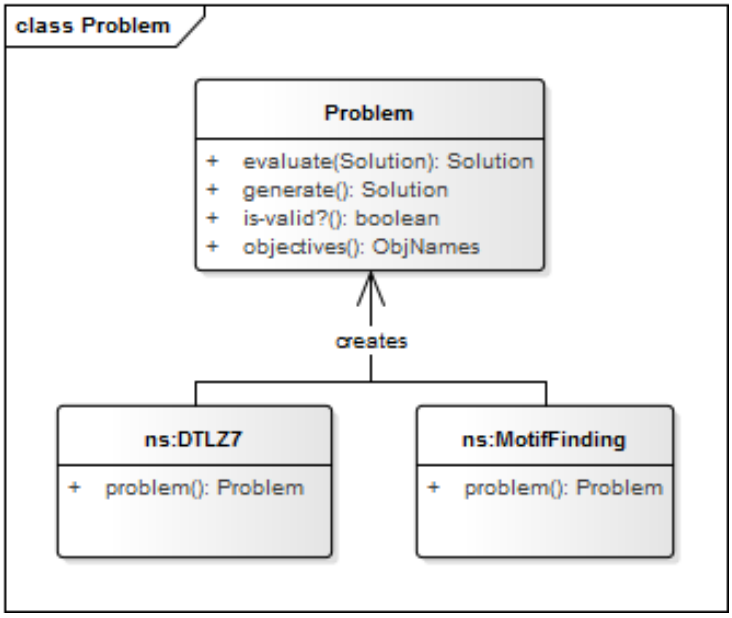


Obr. 11: *Diagram tried opisujúci ostrov.*

7.2.1 Config

Jednou z požiadaviek na ostrov je, aby bežal v samostatnom procese čo možno najoddelenejšie od ostatných procesov. Preto si každý ostrov nesie vlastnú inštanciu konfigurácie, ku ktorej môže pristupovať bez toho, aby musel riešiť problémy spojené so synchronizáciou.

Táto konfigurácia je využívaná ostatnými jednotkami ostrova. Kedže všetky parametre sú oddelené od implementácie problému alebo vyhľadávacieho algoritmu, viaceré ostrovy môžu využívať rovnaký kód no s inými nastaveniami. Takýto návrh zjednodušuje aj testovanie, pretože každý parameter, ktorý ovplyvňuje beh programu, je dostupný na jednom mieste.



Obr. 12: *Diagram tried opisujúci objekt* Problem.

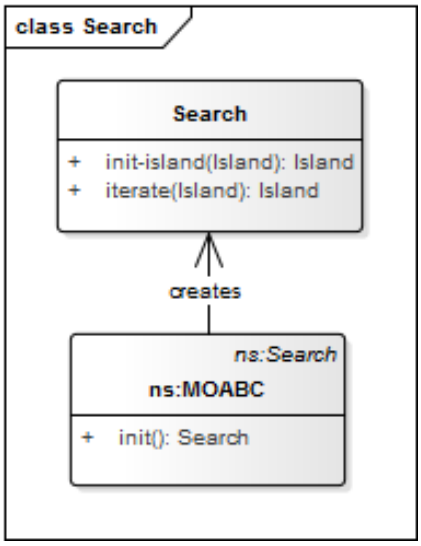
7.2.2 Problem

Tento objekt, znázornený na obrázku č. 12, predstavuje inštanciu problému ako napríklad problém hľadania motívov (kapitola č. 2) alebo DTLZ7, ktorý opíšeme v kapitole č. 7.8.

Pre problém ako taký, resp. pre jeho abstrakciu sme definovali 4 funkcie, ktoré treba implementovať, aby ho vedel ostrov optimalizovať:

- *evaluate* – táto funkcia slúži na ohodnotenie jedinca definovaného v kapitole 7.1. Po vykonaní tejto funkcie musí mať jedinec nastavené výsledky svojho ohodnotenia a pre zjednodušenie aplikácie aj príznak, ktorý popisuje, či je daný jedinec validný.
- *generate* – vytvorí validného a ohodnoteného jedinca na náhodnej pozícii v doménovom priestore.
- *is-valid?* – vráti *true* alebo *false* podľa toho, či je jedinec validný.
- *objectives* – vráti kľúče, na základe ktorých sa dá pristupovať k ohodnoteniu jedinca. Tieto kľúče môžu byť jednoduché čísla, ale aj symboly ako *:similarity* alebo *:length* v závislosti od konkrétneho problému.

7.2.3 Search



Obr. 13: *Diagram tried opisujúci objekt* Search.

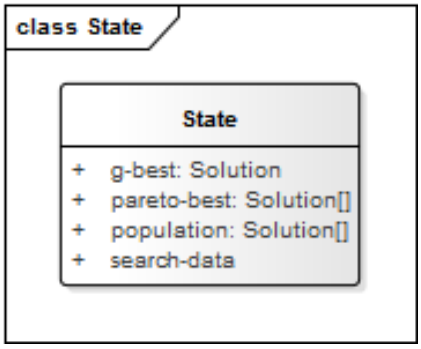
Diagram na obrázku č.13 opisuje objekt *Search*. Ten je inštanciou metaheuristiky, ktorú využíva ostrov na prehľadávanie doménového priestoru. Tento objekt poskytuje dve metódy – *iterate* a *init-island*.

Metóda *iterate* predstavuje jednu iteráciu metaheuristiky. Jej argumentom je celý ostrov, ktorého stav následne mení. Vyhľadávanie MOABC (kapitola č. 7.4) napríklad vykoná všetky fázy včiel.

Metóda *init-island* slúži na inicializáciu ostrova a jeho populácie z pohľadu použitej metaheuristiky. MOABC potrebuje vytvorené paretové rady a dodatočné ohodnotenia opísané v kapitole 7.6.

7.2.4 State

Tento objekt (obrázok č. 14) je nositeľom stavu ostrova resp. vyhľadávania. Obsahuje súčasnú populáciu a množinu jedincov, ktorí tvoria paretový rad. Okrem toho obsahuje dáta špecifické pre vyhľadávací algoritmus a atribút *g-best*. Tento atribút predstavuje jedinca, ku ktorému sú ostatné jedince akokeby ťahané (rovnica č. 12).



Obr. 14: *Diagram tried opisujúci objekt* State.

7.2.5 Vyhľadávací proces ostrova

V algoritme č. 7 opisujeme vyhľadávací proces ostrova. Na riadkoch 1-4 vytvárame ostrov ako kombináciu riešeného problému, vyhľadávacieho algoritmu a konfigurácie. Nasleduje iteratívny proces, kde vyhľadávací algoritmus S vykoná iteráciu metaheuristiky. Následne na riadkoch 8 - 10 sa s určitou pravdepodobnosťou závislou na čase t vykoná funkcia *ponukniJedincov*. Na riadkoch 11 - 14 sa znova s nejakou pravdepodobnosťou vykonajú funkcie *zoberJedincov* a *asimiluj*.

Algorithm 7 Vyhľadávanie ostrova

```
1:  P ← vytvorProblem(:dtlz7)
2:  S ← vytvorSearch(:moabc)
3:  C ← vytvorKonfiguraciu()
4:  I_0 ← vytvorOstrov(C, S, P)
5:  while ukoncit?(I_t, t) = false do
6:      t ← t + 1
7:      I_t ← iteruj(S, I_{t-1})
8:      if p(t) ≥ random(0, 1) then
9:          ponukniJedincov(I_t)
10:     end if
11:     if p_2(t) ≥ random(0, 1) then
12:         M ← zoberJedincov(I_t, t)
13:         I_t ← asimiluj(I_t, M)
14:     end if
15: end while
```

Funkcie *ponukniJedincov*, *zoberJedincov* a *asimiluj* predstavujú mechanizmy

komunikácie medzi ostrovmi, ktoré opíšeme v nasledujúcich kapitolách.

7.3 Migrácie jedincov

Migrácia jedincov je implementovaná pomocou zdieľanej pamäte (kapitola č. 4.2.3). Každý ostrov má k dispozícii dve entity:

- *zdieľaná pamäť* – tá obsahuje štruktúry pre každý ostrov, kde môžu zapisovať svoje riešenia. Tieto štruktúry sú kritické oblasti z pohľadu paralelného programovania. Každý ostrov môže písať iba do svojej štruktúry, no čítať môže zo všetkých. Tieto kritické oblasti sú chránené pomocou *Atom*-ov[13], ktoré zabezpečujú chránený zápis a čítanie. Kedže používame perzistentné dátové štruktúry (kapitola č. 5) nemôže sa stať, že jeden ostrov si zoberie dáta zo zdieľanej pamäte a niekto iný mu ich zmení.

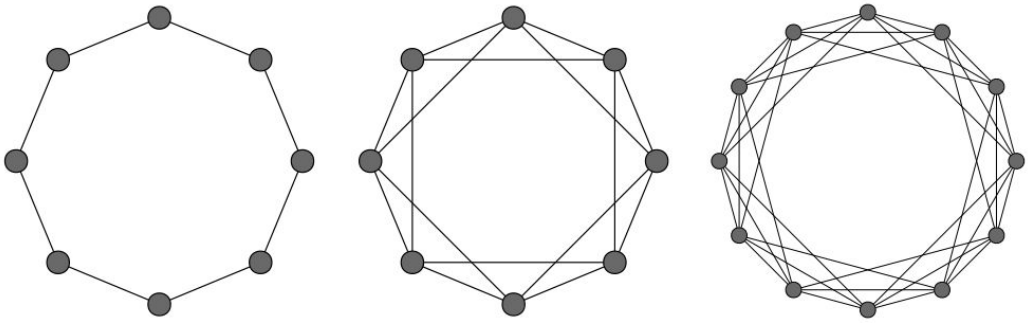
- *susedstvo* – táto štruktúra je špecifická pre každý ostrov tj. každý ma inú inštanciu. Pre ostrov I_k sa skladá z dvojíc (I_i, d_{ki}), kde $I_i \neq I_k$ a d_{ki} je vzdialenosť medzi ostrovmi I_k a I_i. Obsahuje teda informáciu o tom ako sú ostatné ostrovy vzdialené od ostrova I_k.

7.3.1 Susedstvo a topológia

Topológie určujú cesty, ktorými môžu ostrovy komunikovať. Na obrázku č. 15 uvádzame príklad kruhových topológií, kde ostrovy komunikujú iba s ostrovmi, s ktorými majú spoločnú hranu. V implementácií nášho algoritmu nemáme takto explicitne určenú topológiu, resp. každý ostrov je prepojený s každým iným ostrovom a komunikácia závisí od času a vzdialenosti medzi ostrovmi.

Ostrovy sú v našej implementácii rozmiestnené v mriežke pričom v prvom a v poslednom bode tejto mriežky sú umiestnené ostrovy, ktoré optimalizujú všetky ohodnocovacie funkcie. V budúcej práci by bolo vhodné vyskúšať aj iné rozmiestnenie. Z tejto topológie a vzdialeností sa pre každý ostrov vyrátavá jeho susedstvo.

[13] `https://clojure.org/reference/atoms`



Obr. 15: *Topológie (zľava) – Kruh, Kruh+1+2, Kruh+1+2+3[33].*

7.3.2 Funkcie ponúkni jedincov, zober jedincov a asimiluj

Funkcie *ponukniJedincov*, *zoberJedincov* a *asimiluj* zabezpečujú výmenu jedincov medzi ostrovmi a sú volané po každej iterácii metaheuristiky s určitou pravdepodobnosťou.

Funkcia *ponukniJedincov* zapíše prvý paretový rad do zdieľanej pamäte, čím poskytne jedincov ostatným ostrovom na čítanie. Funkcia *zoberJedincov* pracuje so susedstvom ostrova. Najprv na základe času obmedzí množstvo kandidátov, s ktorými môže ostrov komunikovať. Toto množstvo závisí od vzdialenosti ostrovov. Na začiatku vyhľadávania môže ostrov komunikovať iba s blízkymi ostrovmi. Okolie, s ktorým si môže vymieňať riešenia, sa časom zväčšuje. Potom si náhodne vyberie ostrov od ktorého chce čerpať riešenia pomocou rulety. Ostrovy, ktoré sú bližšie, majú väčšiu šancu na výber. Kedže ostrovy poskytujú celý paretový rad, ostrov prijímajúci riešenia sa musí rozhodnúť, ktoré riešenie z paretového radu si zoberie. V našej implementácii je to prvý jedinec v rade. Je možné vyskúšať aj iný výber, napríklad v [27] vyberali jedincov pomocou ohodnotenia *crowding distance*. Funkcia *asimiluj* nastaví prijatého jedinca ako nového *g-best*.

7.4 MOABC

Metaheuristika, ktorú používame v ostrovoch, sa volá Multiobjective Artifical Bee Colony (MOABC) [17]. Jej fungovanie sme už načrtli v kapitole 3.2.2. V algoritme 8 uvádzame pseudokód metódy *iterate*, ktorú poskytuje naša implementácia MOABC ostrovu.

Algorithm 8 MOABC - iterate

```
1:  procedure ITERATE(Island I)
2:      Stav ← stav(I)
3:      P ← populacia(Stav)
4:      W, O, S ← rozdelPopulaciu(P)
5:      W ← fazaPracovniciek(P, W)
6:      O ← fazaPrizeracov(P, W, O)
7:      S ← fazaPrieskumiciek(S)
8:      P_2 ← spoj(W, O, S)
9:      Fronts ← vytvorParetoFronty(P_2)
10:     Fronts ← zoradVRamciFrontov(Fronts)
11:     P_3 ← vytvorNovuPopulaciu(Fronts)
12:     I_2 ← zmenStav(I, Stav, P_3, Fronts)
13:     return I_2
14: end procedure
```

V riadkoch 2 - 4 získame populáciu zo súčastného ostrova a následne ju rozdelíme na 3 skupiny – pracovníčky, prizeračov a prieskumníčky. Rozdelenie prebieha takým spôsobom, že napríklad prvých 50% považujeme za pracovníčky, dalších 40% za prizeračov a zvyšné včely za prieskumníčky.

Po tomto delení nasledujú fázy na riadkoch 5 - 7 tak, ako sú definované v ABC [19]. V pôvodnom algoritme vo fáze prizeračov si včely tejto skupiny náhodne zvolia jednu pracovníčku a jej pozíciu sa snažia zlepšiť. Pravdepodobnosť, že táto včela bude vybraná, závisí od jej ohodnotenia. Kedže pôvodné ABC optimalizovalo iba 1 ohodnocovaciu funkciu, takýto výber sme nemohli presne implementovať. Pravdepodobnosť výberu pracovníčky v našom algoritme môže závisieť buď od pozície v paretovom rade alebo od nejakého iného ohodnotenia v rámci tohto radu. Tomuto problému sa venujeme v nasledujúcich kapitolách.

Po týchto fázach nasledujú kroky spojené s optimalizáciou viacerých ohodnocovacích funkcií. Na riadku 8 vytvoríme novú populáciu spojením vyššie opísaných skupín. Následne na riadku 9 vytvoríme z tejto populácie paretové rady pomocou *non-dominated sort* opísaného v nasledujúcej kapitole. Tento spôsob zoradenia nám vráti množinu radov, kde v prvom rade sú všetky včely lepšie ako v druhom atď. Nehovorí ale nič o tom, v akom vzťahu sú včely v rámci jedného radu. Preto musíme v 10 zoradiť tieto včely iným spôsobom.

Následne na riadkoch 11 a 12 vieme z týchto radov vyskladať novú populáciu a zmeniť stav ostrova.

7.5 Non-dominated sort

V NSGA-II [9] opísali algoritmus, ktorý na zoradenie jedincov používal tzv. *non-dominated sort*. Tento zaraďovací algoritmus slúži na rýchle vytvorenie paretových radov z populácie. Je založený na princípe dominancie opísanej v kapitole č. 3.2.1. Jeho výstupom je množina paretových radov, kde prvý rad obsahuje jedince, ktoré dominujú jedince v ostatných radoch. Nasledujúce rady obsahujú zasa jedince, ktoré dominujú ostatné v radoch za nimi.

7.6 Usporiadanie v rámci paretových radov

Pri opise pseudokódu MOABC v kapitole 7.4 sme narazili na problém náhodného výberu jedincov vo fáze pracovníčiek. Potrebujeme metriku, ktorou vieme ohodnotiť jedinca v rámci paretového frontu. Na základe tejto metriky vieme následne ovplyvniť náhodný výber jedinca. V nasledujúcich kapitolách uvádzame dve také metriky. Obe sú založené na vzdialenosti jedincov v doméne ohodnotení a preferujú jedince, ktoré sú v redších oblastiach. Táto preferencia by mala zlepšiť aproximáciu paretového radu.

7.6.1 Crowding distance

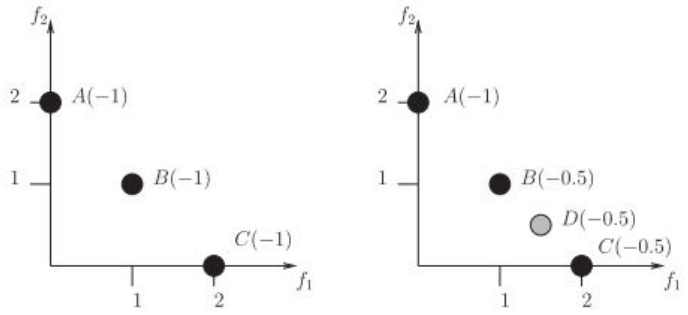
Tento algoritmus bol spolu s *non-dominated sort* definovaný v NSGA-II [9]. Ohodnocuje jedincov spôsobom, pri ktorom sleduje polohu jedinca v porovnaní s jeho susedmi. V rámci jednej dimenzie ohodnocovacieho priestoru vyrátame pre jedného jedinca súčet vzdialeností k dvom najbližším susedom (jednému menšiemu a druhému väčšiemu). Ohodnotenie jedinca je potom súčet týchto vzdialeností cez všetky dimenzie.

7.6.2 Maximin

Maximin ohodnotenie [4] je definované ako:

$$fitness^i = max_{j \neq i}(min_k(f_k^i - f_k^j))$$

kde f_k sú ohodnocovacie funkcie. Na obrázku č. 16 môžeme vidieť ohodnotenie touto funkciou v dvoch prípadoch. Kedže ide o minimalizáciu, môžeme vidieť, že jedince, ktoré sú ďalej od susedov, majú lepšie ohodnotenie.



Obr. 16: *Ohodnotenie maximin[29].*

7.6.3 Usporiadanie a výber jedincov

Po tom, čo prebehnú fázy včiel v MOABC potrebujeme jedincov usporiadať. Prvým krokom je vytvorenie paretových radov pomocou NSGA-II. Nasleduje usporiadanie v rámci týchto radov použitím *crowding distance*. Tieto usporiadané rady sa následne spoja a vytvoria ďalšiu iteráciu populácie.

V pôvodnej verzii algoritmu ABC [19] si vo fáze prizeračov včely vyberajú zdroje potravy, ktoré chcú preskúmať, na základe ich ohodnotenia fitness. Kedže pri probléme hľadania motívov máme viaceré ohodotenia, nemôžeme použiť pôvodný návrh.

V našej implementácii si prizerači vyberajú zdroje potravy pomocou rulety, kde je fitness odvodená od príslušnosti k paretovému radu a zároveň aj od *crowding distance*:

$$fitness(x) = 2 \cdot \frac{MAX - rank(x)}{MAX} + normCD(x) \tag{13}$$

kde $rank(x)$ je číslo paretového radu od 0 (najlepšie) po N (najhoršie) do ktorého patrí jedinec a $MAX = N + 1$. $normCD(x)$ je hodnota *crowding distance* normalizovaná v rámci radu.

7.7 Postprocessing

Vstupom pre náš algoritmus sú DNA sekvencie vo formáte $fasta$ z testovacieho datasetu [35]. Tento dataset vyžaduje, aby výstupom boli reťazce nukleotidov, ktoré predstavujú motív a ich presná poloha. V nasledujúcich kapitolách opíšeme mechanizmy, ktoré okrem toho, že z najlepších jedincov extrahujú reťazce DNA motívov, sú schopné ich aj dodatočne ohodnotiť a prefiltrovať.

7.7.1 Extrahovanie motívu pomocou konsenzu

V kapitole č. 2.1.1 sme opísali proces ohodnocovania jedincov, ktorý použili v MOABC. V tomto procese vystupoval konsenzus motív, čo je akási abstrakcia nad motívmi, na ktoré ukazuje poloha včely. V extrakcii motívu pomocou konsenzu využívame práve tento konsenzus motív. V algoritme 9 opisujeme priebeh takejto extrakcie.

Algorithm 9 Extrakcia motívu pomocou konsenzu

```
1: procedure EXTRAHUJMOTIVKONS(Solution s)
2:     motivy ← vyberMotivyPodlaPolohy(s)
3:     k1 ← konsenzus(motivy)
4:     kandidati ← filtrujPodlaMinPodpory(motivy, k1, 0.5)
5:     k2 ← konsenzus(kandidati)
6:     instanciaMotivu ← vyberNajblizsi(kandidati, k2)
7:     return instanciaMotivu
8: end procedure
```

Riadky 2 - 4 filtrujú reťazce, na ktoré ukazuje jedinec rovnakým spôsobom ako pri jeho vyhodnocovaní. Z kandidátov, ktorých dostaneme na riadku 4, by sa pri ohodnocovaní vytvorila matica PFM. Sú to práve tieto motívy, z ktorých vyberáme riešenie. Vytvoríme si z nich ďalší konsenzus motív a za výsledok považujeme ten motív, ktorý je k nemu najbližšie. Tento prístup extrahuje z každého jedinca iba jedno riešenie problému.

7.7.2 Vedierkové skóre

Vedierkové skóre slúži na ohodnotenie reťazca nukleotidov k, ktorý má predstavovať motív. Pre kandidáta k sa z každej sekvencie vytvoria n-gramy, ktoré majú rovnakú dĺžku. Tieto n-gramy sa ohodnotia podľa toho, akú majú Hammingovú vzdialenosť voči k a zahodia sa tie, ktoré majú vzdialenosť väčšiu ako e[14]. N-gramy, ktoré ostali, sa zatriedia do vedierok B_i. N-gram patrí k vedierku B_i ak je Hammingová vzdialenosť n-gramu od k rovná i. Vedierkové skóre je definované rovnicou 14:

$$vedierkoveSkore = \sum_{i=0}^{|B|-1} \frac{|B_i|}{i+1} \tag{14}$$

kde $|B|$ je počet vedierok a $|B_i|$ je počet n-gramov v i-tom vedierku.

7.7.3 Weed-out

Tento prístup je založený na fungovaní algoritmu Weeder, ktorý sme opísali v kapitole č. 2.2.1. Na rozdiel od prístupu v algoritme 9 *Weed-out* využíva pri extrakcii všetky jedince naraz.

Algorithm 10 Extrakcia motívov pomocou weed-out

```
1:  procedure WEED-OUT(Solution solutions[])
2:      kandidati ← ∅
3:      for each s ∈ solutions do
4:          motivy ← vyberMotivyPodlaPolohy(s)
5:          kons ← konsenzus(motivy)
6:          k ← filtrujPodlaMinPodpory(motivy, kons, 0.5)
7:          for each k' ∈ k do
8:              kandidati ← kandidati ⋃ vedierkoveOhodnotenie(k)
9:          end for
10:     end for
11:     kandidati ← vedierkovyFilter(kandidati)
12:     kandidati ← occFilter(kandidati)
13:     return kandidati
14: end procedure
```

[14] e v našom prípade závisí od dĺžky k.

Na riadku 2 najprv inicializujeme prázdnu množinu kandidátov. Potom pre každé riešenie, podobne ako pri extrakcii pomocou konsenzu, vyberieme z každého jedinca kandidátov. Následne ohodnotíme každého kandidáta vedierkovým ohodnotením. Na riadku 11 si ponecháme iba najlepších 5 jedincov podľa vedierkového skóre. *occFilter* je skóre výskytov tak, ako bolo definované v kapitole č. 2.2.1. V tomto ohodnotení sa použijú n-gramy blízke kandidátovi na vytvorenie matice PFM. Podľa nukleotidov kandidáta a rovnice 8 sa vyráta skóre medzi 0 a 1. Všetci kandidáti, ktorí majú skóre menšie ako 0,9, sú zahodení.

7.7.4 Bucket-out

Bucket-out využíva rovnaký algoritmus ako *Weed-out*, no vynechá filtrovanie pomocou *occFilter*.

7.7.5 Instance-filter

Instance-filter kombinuje vedierkové skóre a extrahovanie motívov konsenzom. Funguje spôsobom, pri ktorom sa vedierkovým skóre ohodnocujú iba také reťazce, ktoré boli pred tým extrahované z jedincov pomocou konsenzu (kapitola 7.7.1). Takto ohodnotené reťazce sa potom zoradia podľa skóre a následne sa zoberie iba n najlepších.

7.7.6 Filtrovanie výsledkov

Náš algoritmus poskytuje 4 módy postprocesingu a filtrácie, medzi ktorými si môžeme vyberať. Všetky tieto módy pracujú s n najlepšími jedincami zo všetkých populácií:

- *:only-pareto-n* – použije extrakciu motívu pomocou konsenzu,
- *:only-weeder* – použije *Weed-out*,
- *:bucket-out* – použije *Bucket-out*,
- *:instace-filter* – použije *Instance-filter*.

7.8 DTLZ 7

Keďže návrh nášho algoritmu umožňuje optimalizovať viacero problémov, implementovali sme za účelmi testovania aj jednoduchší problém ako je hľadanie motívov a to problém DTLZ7 [10]. Ten na rozdiel od MFP (Motif Finding Problem) je matematicky jednoznačne popísaný a je známe jeho úplné riešenie. Vďaka tomu vieme presne určiť, ako vyzerá jeho paretový rad a napríklad aj to, ako ďaleko sú od neho vzdialené nami nájdené riešenia. Pri hľadaní motívov vyhodnocujeme algoritmus na základe prekryvu nami nájdených motívov a motívov zasadených v testovacom datasete. DTLZ7 vieme vyhodnotiť na základe pokrytia paretových radov. Vieme teda presnejšie sledovať správanie algoritmu.

7.8.1 Opis problému

Problém DTLZ7 patrí do skupiny problémov DTLZ, ktorú definoval K. Deb a spol. v [10]. V tejto skupine je definovaných 9 optimalizačných problémov, ktoré testujú rôzne vlastnosti algoritmu. DTLZ7 testuje schopnosť algoritmu udržiavať podpopulácie v rôznych pareto optimálnych regiónoch. Je definovaný nasledujúco:

Minimalizuj:

$$f_1(x_1) = x_1,$$

$$f_2(x_2) = x_2,$$

$$\vdots$$

$$f_{M-1}(x_{M-1}) = x_{M-1},$$

$$f_M(x) = (1 + g(x_M))h(f_1, f_2, ..., f_{M-1}, g),$$

kde

$$g(x_M) = 1 + \frac{9}{|x_M|} \sum_{x_i \in x_m} x_i,$$

$$h(f_1, f_2, ..., f_{M-1}, g) = M - \sum_{i=1}^{M-1} \left[\frac{f_i}{1+g}(1 + \sin(3\pi f_i)) \right],$$

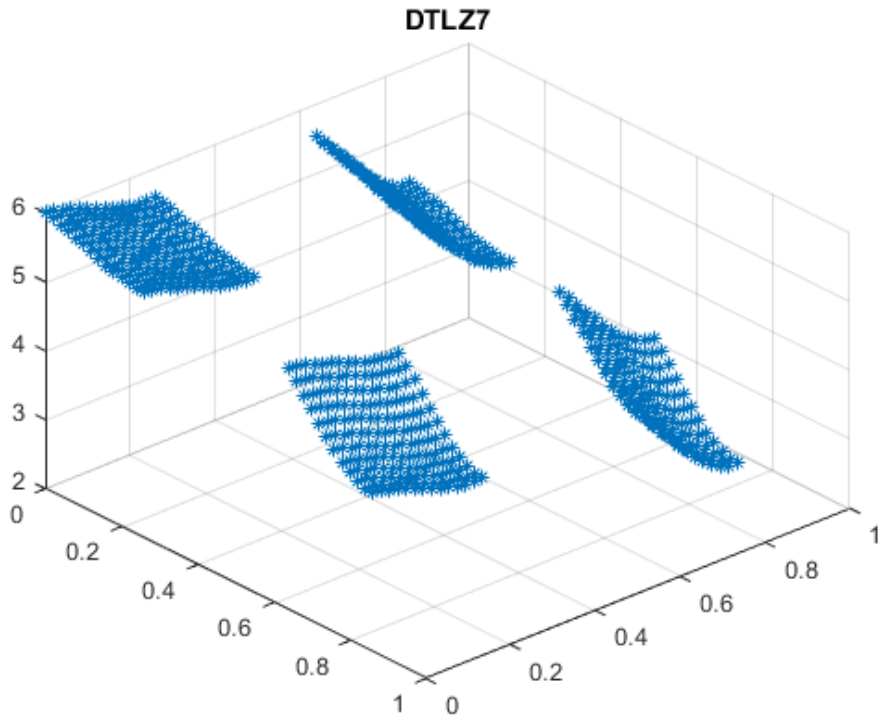
zatiaľ čo platí

$$0 \leq x_i \leq 1, \qquad i = 1, 2, ..., n$$

Funkcie f_i predstavujú ohodnocovacie funkcie, zatiaľ čo vektor x predstavuje pozíciu riešenia v doménovom priestore. n je počet rozmerov tohto priestoru a M je celkový počet ohodnocovacích funkcií. Takáto parametrizácia problému je ďalšou výhodou problémov z triedy DTLZ, kde si môžeme sami určiť počet ohodnocovacích funkcií alebo veľkosť doménového priestoru.

Autor problému ďalej uvádza, že funkcia g potrebuje $k = |x_m|$ rozhodovacích premenných (pozície vo vektore x) pričom celkový počet premenných je $n = M + K + 1$. Autor odporúča $k = 20$. Pri našom testovaní sme používali $M = 3$ ohodnocovacie funkcie, teda počet dimenzií doménového priestoru bol v súlade s odporúčaným k $n = 22$.

Pre tri ohodnocovacie funkcie sme sa rozhodli preto, lebo v knižnici *jMetal*[13] je dostupný pre tento počet pareto optimálny rad. Tento rad pre problém DTLZ7 uvádzame na obrázku č. 17, kde osi predstavujú ohodnotenia jedinca. Výsledky predbežných testov uvedieme v ďalších kapitolách.



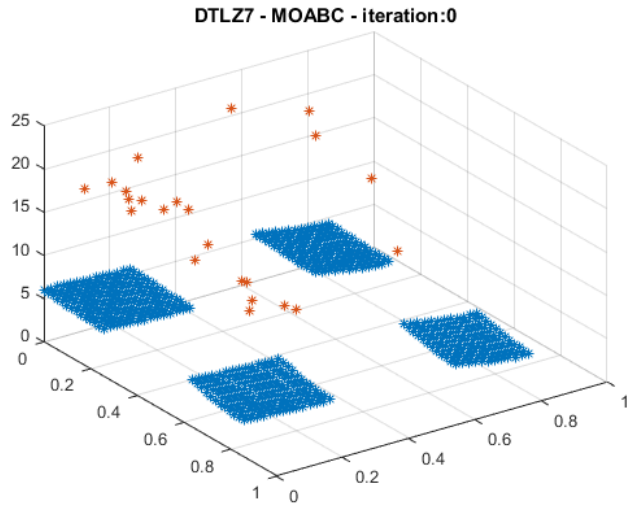
Obr. 17: *Paretový rad pre problém DTLZ7*[15].

7.8.2 Testovanie pomocou DTLZ 7

Pomocou problému DTLZ7 sme sledovali správanie a vlastnosti nášho algoritmu. Vyhodnocovali sme to spôsobom, pri ktorom sledujeme priestor ohodnotení resp. ako sa počas iterácií vyvíja paretový rad. Pri testovaní sme použili MOABC bez *g-best* člena. Počet včiel bol 100, pričom pracovníčiek bolo 50, prizeračov 40 a prieskumníčok 10. Testovali sme schopnosť algoritmu konvergovať k riešeniu, ale aj správanie sa vyhľadávania pri uvoľnení ohodnocovacích funkcií, tak ako je opísané v kapitole č. 6.

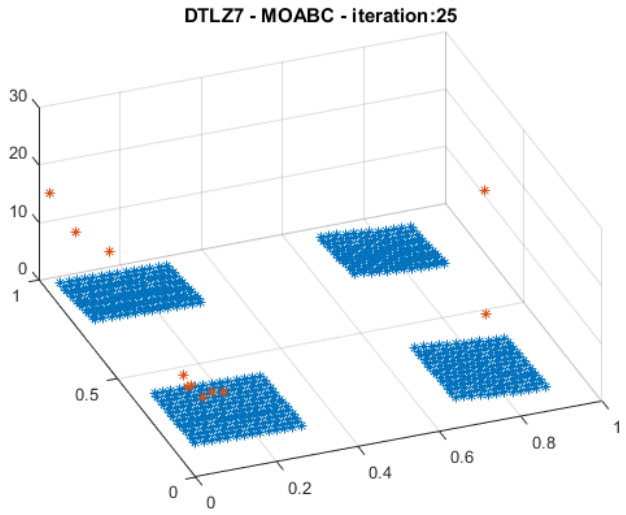
Na nasledujúcich obrázkoch uvádzame správanie sa algoritmu pri vyššie spomenutých parametroch, pričom sme neuvoľnili žiadnu funkciu. Na obrázku č. 18 zobrazujeme graf riešení v priestore ohodnocovacích funkcií definovaných v kapitole č. 7.8.1. Modrou farbou sú znázornené body riešenia daného problému. Červenou farbou sú znázornené body, ktoré tvorili pareto rad nášho riešenia pre danú iteráciu.

Na obrázkoch č. 18, 19, 20 a 21 uvádzame riešenia v rôznych momentoch vyhľadávania.

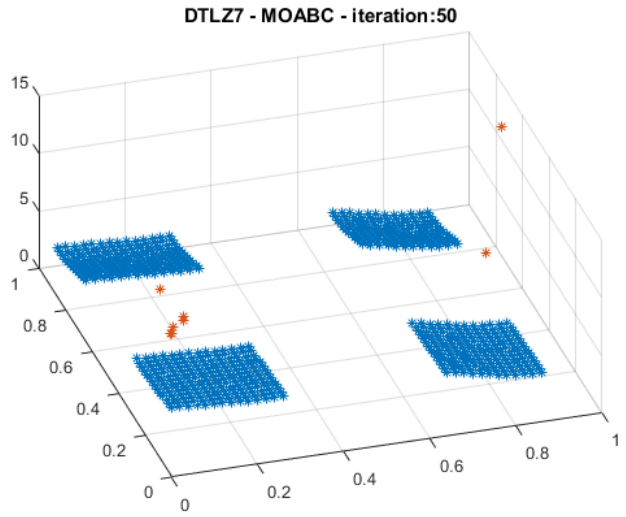


Obr. 18: *MOABC – prvá iterácia pre všetky ohodnocovacie funkcie.*

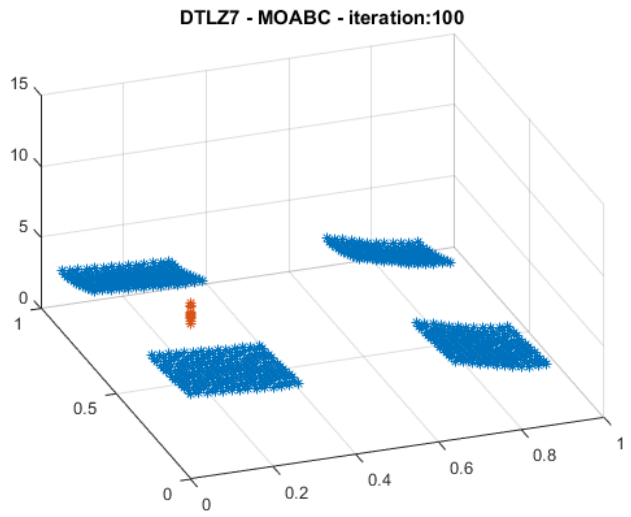
[15] Vygenerované pomocou nástroja MATLAB – https://www.mathworks.com/products/matlab.html



Obr. 19: *MOABC – 25. iterácia pre všetky ohodnocovacie funkcie.*



Obr. 20: *MOABC – 50. iterácia pre všetky ohodnocovacie funkcie.*



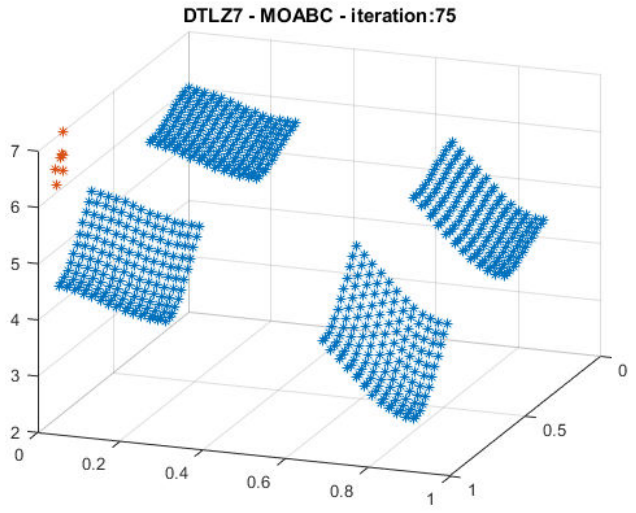
Obr. 21: *MOABC – 100. iterácia pre všetky ohodnocovacie funkcie.*

Z daných výsledkov sme vedeli vyvodiť niekoľko záverov. Naše riešenie iteratívnym spôsobom konvergovalo k lepším riešeniam. DTLZ je problém, kde sa snažíme minimalizovať ohodnocovacie funkcie. Po stej iterácii algoritmu sa všetky najdominantnejšie riešenia nachádzali nad bodom [0,0,0]. To znamená, že dve z troch ohodnocovacích funkcií optimalizovali najlepšie, ako sa dalo.

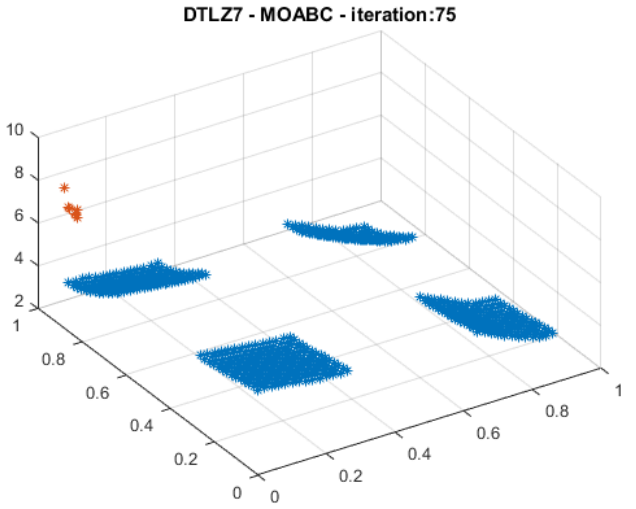
V čom náš algoritmus zlyhával bolo, že sa nedokázal dostať do skutočného optima, ktoré bolo na obrázkoch vidieť modrou farbou. To môže byť spôsobené tým, že sme sa zasekli v lokálnom optime. Skonvergovali sme do bodu [0,0,˜14] pričom skutočné riešenia sa nachádzajú na celom rozsahu prvých dvoch súradníc. Pri pohľade na ohodnocovacie funkcie DTLZ nie je prekvapivé, že sme sa zasekli práve v týchto bodoch. Tieto ohodnocovacie funkcie sa optimalizujú „jednoducho“, kedže ide v podstate iba o jednotlivé hodnoty pozičného vektora v priestore domény ($f_i(x) = x_i$). Takéto ohodnocovacie funkcie sa ale môžu vyskytovať aj v iných problémoch, napríklad pri hľadaní motívov je to ohodnocovacia funkcia, ktorá popisuje dĺžku motívu. „Zložitá“ funkcia je práve tá tretia, ktorú sa nám nepodarilo príliš optimalizovať. Táto funkcia závisí od celej polohy.

Druhou chybou nášho algoritmu bola slabá diverzifikácia riešení, práve na základe týchto testov sme zistili potrebu použitia ohodnotení ako *crowding distance* opísaného v kapitole č. 7.6.1.

Na nasledujúcich obrázkoch uvádzame výsledky algoritmu po 75 iteráciach pri postupnom uvoľňovaní požiadaviek na ohodnocovacie funkcie.



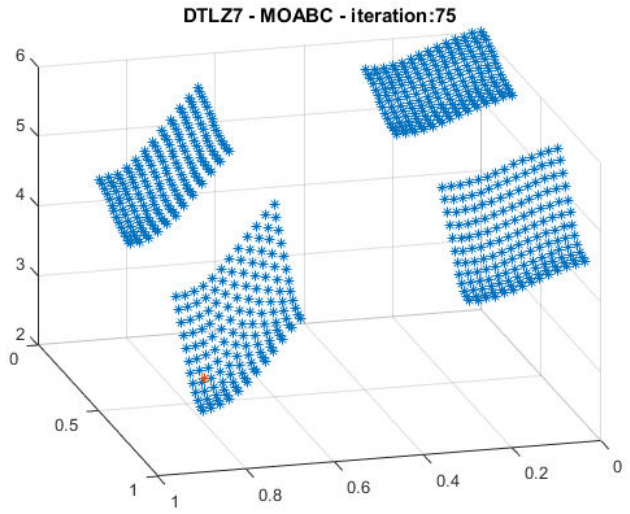
Obr. 22: *MOABC – 75. iterácia pri uvoľnení prvej „jednoduchej“ ohodnocovacej funkcie.*



Obr. 23: *MOABC – 75. iterácia pri uvoľnení druhej „jednoduchej“ ohodnocovacej funkcie.*

Na prvých dvoch grafoch č. 22 a 23 je znázornená 75. iterácia pri uvoľnení jednej z ľahkých ohodnocovacích funkcií. Pri týchto vyhľadávaniach sme optimalizovali teda iba 2 funkcie – jednu „jednoduchú“ a jednu „zložitú“. Ako môžeme vidieť, znovu sme sa zasekli v lokálnom optime, no bolo to oveľa bližšie ku skutočnému riešeniu. Pri „zložitej“ funkcii sme namiesto 14 dosiahli ohodnotenie približne 7. Keď sme interaktívne prezerali tieto grafy, videli sme aj to, že vý-

sledky sa nachádzajú akoby na opačných stenách, teda už iba kombináciou týchto dvoch riešení by sme dosiahli väčšiu diverzifikáciu.



Obr. 24: *MOABC – 75. iterácia pri uvoľnení oboch „ľahkých“ ohodnocovacích funkcií.*

Na poslednom grafe na obrázku č. 24 je vyhľadávanie, pri ktorom sme uvoľnili obe „jednoduché“ funkcie. Z povahy fungovania paretových radov vraciame iba 1 riešenie – to, ktoré dominuje všetky ostatné. Na tomto grafe vidíme, že sme sa dostali takmer úplne k dokonalému riešeniu.

Pomocou problému DTZL7 sme testovali neúplnú implementáciu algoritmu. Mohli sme sledovať správanie algoritmu ešte pred tým, než sme mali implementovaný problém MFP. Zistili sme potrebu použitia *crowding distance*. V budúcej práci by bolo vhodné nad týmto problémom otestovať úplnú implementáciu. Okrem DTLZ7 by sme mohli použiť aj ostatné DTZL verzie a zistiť ako efektívny je náš algoritmus na iných problémoch ako MFP.

8 Testovanie a vyhodnotenie

V tejto kapitole opíšeme testovací dataset a použité metriky na vyhodnocovanie. Potom zhrnieme parametre algoritmu a metodológiu testovania. Nakoniec opíšeme dosiahnuté výsledky a porovnáme ich s inými algoritmami.

8.1 Testovací dataset

Algoritmus sme testovali nad testovacím datasetom od Tompa a spol. [35]. Tento dataset obsahuje sekvencie zo štyroch živočíchov – človek, myš, mucha (*D. melanogaster*) a droždie (*S. cerevisiae*). Využíva databázu TRANSFAC [37], ktorá obsahuje známe motívy spolu s génmi, v ktorých sa vyskytujú.

Keďže tento dataset slúži na testovanie algoritmov, ktoré hľadajú nové motívy, autori museli riešiť dva problémy. Pri použití reálnych sekvencií s už známymi motívmi nikto nepozná úplne riešenie, teda či sa v nich nenachádzajú neobjavené motívy. Pri použití vygenerovaných sekvencií zasa nevieme s istotou povedať, či ich generujeme správnym spôsobom.

Preto sa autori rozhodli pre každého živočícha použiť tri typy vstupov:

- skutočné sekvencie,
- vygenerované sekvencie s vloženými motívmi,
- náhodne zvolené sekvencie s vloženými motívmi.

Každý z týchto typov obsahuje 56 vstupov. Každý vstup môže mať od 1 po 35 sekvencií, ktoré môžu mať dĺžku až 3000 nukleotidov.

K tomuto datasetu je verejne dostupný testovač[16], ktorý vyžaduje aby bol výstupom algoritmu zoznam reťazcov nukleotidov spolu s ich pozíciami v DNA sekvenciách. Taktiež je možné meniť parametre algoritmu v závislostí od vstupu.

[16] http://bio.cs.washington.edu/assessment/index.html

8.2 Použité metriky

Použitý testovač ohodnocoval výsledky viacerými metrikami na úrovni nukleotidov ako aj motívov. Pri nukleotidoch sledoval presné počty pozícií, v ktorých sa nukleotidy nachádzali na pozíciach známych motívov. Na úrovní motívov akceptovali reťazec ak prekrýval známy motív.

Testovač vracal celkové počty nájdených motívov resp. metriky ako skutočné a falošné pozitíva (TP/FP) alebo skutočné a falošné negatíva (TN/FN) na úrovni motívov aj nukleotidov.

Z týchto metrík potom poskytoval:

- *senzitivitu* – $Sn = \frac{TP}{TP+FN}$, známe aj ako *recall*, na úrovni motívov aj nukleotidov,
- *precíznost'* – $PPV = \frac{TP}{TP+FP}$, známe aj ako *positive predictive value (ppv)*, na úrovni motívov aj nukleotidov.

Okrem toho uvádzajú aj:

- *špecifickost'* – $nSp = \frac{nTN}{nTN+nFP}$ na úrovní nukleotidov,
- *koeficient výkonu* – $nPC = \frac{nTP}{nTP+nFN+nFP}$ na úrovní nukleotidov,
- *korelačný koeficient* –

  $$nCC = \frac{nTP \cdot nTN - nFN \cdot nFP}{\sqrt{((nTP + nFN)(nTN + nFP)(nTP + nFP)(nTN + nFN))}}$$

  známy aj ako *matthews correlation coefficient* na úrovni nukleotidov,
- *priemerný výkon motívov* – $sASP = \frac{sSn+sPPV}{2}$ na úrovni motívov.

Z týchto metrík sme pri testovaní sledovali najviac senzitivitu a precíznosť.

8.3 Parametre algoritmu

V našom algoritme sa dá nastavovať viacero parametrov. Rozdelili sme ich preto do dvoch skupín podľa toho, či ich používa vyhľadávací algoritmus (MOABC) alebo či ovplyvňujú optimalizačný problém (MFP). Parametre uvádzame s ich kľúčom, ktorý používame v konfiguráciách a popisom.

Parametre pre MOABC:

- *:coef-C* – koeficient „priťahovania“ (Ψ) v rovnici pohybu včiel (rovnica 12),
- *:population-count* – celkový počet včiel,
- *:size* – vektor pomerov typov včiel. $[0.5\ 0.4\ 0.1]$ by zodpovedalo 50% pracovníčiek, 40% prizeračov a 10% prieskumníčok,
- *:num-of-iterations* – počet iterácii MOABC,
- *:share-prob* – pravdepodobnosť, s akou ostrov zdieľa jedincov,
- *:take-prob* – pravdepodobnosť, s akou ostrov príma jedincov,
- *:pareto-epsilon* – hodnota ϵ v ϵ-dominancii (kapitola 3.2.1),

Parametre pre MFP:

- *:support-coeff* – percentuálny podiel nukleotidov, ktorý musí mať motív spoločný s konsenzom, aby sa s ním rátalo pri podpore motívu počas ohodnocovania včely,
- *:min-length* – minimálna dĺžka motívu,
- *:max-length* – maximálna dĺžka motívu,
- *:min-obj* – minimálny počet ohodnocovacích funkcií, ktoré musí ostrov optimalizovať.

V kapitole č. 6 sme opísali ako každý ostrov ohodnocuje inú podmnožinu ohodnocovacích funkcií. Rozhodli sme sa parametrizovať tieto podmnožiny obmedzením ich minimálnej veľkosti. Ak je *:min-obj* rovný počtu ohodnocovacích funkcií, použijeme fixný počet ostrovov, ktoré optimalizujú všetky ohodnocovacie funkcie[17].

Okrem vyššie uvedených parametrov je potrebné nastaviť aj metódu postprocessingu opísaného v kapitole č. 7.7.6.

8.4 Metodológia testovania

Algoritmus sme museli testovať viackrát z dôvodu nastavovania parametrov. Začali sme so základnými parametrami pre MFP takými aké boli definované v MOABC. K nim sme menili hodnoty vyhľadávacieho algoritmu. Začali sme pri malých populáciách (50 jedincov) a nízkom počte iterácií (20). Potom sme postupne zvyšovali počet iterácií až po 300 a veľkosť populácie až po 150. Najlepšie výsledky sme dosiahli pri nízkych počtoch populácii a aj iterácií. Potom sme menili konfiguráciu MFP pričom sme ju skúšali s rôznymi MOABC konfiguráciami.

Z týchto testov vyplynula aj potreba doplniť postprocessing (kapitola č. 7.7). Preto sme ten testovali už iba nad niekoľkými najlepšími dvojicami konfigurácií MFP a MOABC.

Nakoniec sme menili hodnotu *:pareto-epsilon* pri najlepšej konfigurácii. Dokopy sme vykonali viac ako 70 testov, pričom sme niektoré konfigurácie opakovali, aby sme zistili ako stabilné sú výsledky. Tie sa pre jednu konfiguráciu menili v rozsahu $\pm 1\%$.

Dĺžka testovania závisí od konfigurácie. Pre jeden vstup môže hľadanie motívov trvať od niekoľko sekúnd až po niekoľko minút, ak máme veľké množstvo iterácií a veľké populácie. Vstupov je zhruba 160 a pre všetky trvá test od niekoľko minút až po niekoľko hodín.

[17] Testovali sme s 8 ostrovmi.

8.5 Výsledky testov a porovnanie

V tabuľke č. 2 uvádzame nastavenia algoritmu, pri ktorých sme dosiahli najlepšie výsledky[18]. Pri týchto nastaveniach sme používali postproces *:only-pareto-n*, teda extrakciu motívov pomocou konsenzu.

Tabuľka 2: *Parametre vyhľadávania.*

MOABC		MFP	
parameter	hodnota	parameter	hodnota
:coef-C	0.5	:support-coeff	0.5
:population-count	60	:min-length	15
:size	[0.7 0.25 0.05]	:max-length	64
:num-of-iterations	20	:min-obj	4
:share-prob	0.3		
:take-prob	0.3		
:pareto-epsilon	0.1		

V tabuľke 3 uvádzame nami dosiahnuté výsledky. Úplné výsledky ostatných algoritmov uvádzame v prílohe E.

Tabuľka 3: *Dosiahnuté výsledky.*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	242	15595	1330	86333	20	299	97		0.153	0.015	0.846	0.014	0.0003	0.170	0.062	0.116
Human	1674	65740	13683	779903	102	1353	792		0.109	0.024	0.922	0.020	-	0.114	0.070	0.092
Mouse	1026	25411	3568	117495	70	587	205		0.223	0.038	0.822	0.034	0.020	0.254	0.106	0.180
Yeast	776	24419	2854	146951	59	469	163		0.213	0.030	0.857	0.027	0.028	0.265	0.111	0.188
Total	3718	131165	21435	1130682	251	2708	1257		0.147	0.027	0.896	0.023	0.525	0.166	0.084	0.125

Pri našich testoch sme najviac sledovali senzitivitu a precíznosť na úrovní motívov cez všetky živočíchy. Pracovali sme aj s myšlienkou použitia rôznych nastavení pre rôzne vstupy, ale nepodarilo sa nám nájsť konfigurácie, ktoré by mali stále lepšie výsledky iba pre niektoré druhy.

Pri porovnávaní s inými algoritmami sme sledovali rovnaké metriky. Najlepším algoritmom bol *Weeder*, ktorý dosiahol senzitivitu 16,1% a precíznosť 28,9%. Najhorším bol *QuickScore* s 3,3% resp. 1,9%. My sme dosiahli senzitivitu 16,6% a precíznosť 8,4% k čomu je najbližšie algoritmus *ANN-Spec* s 15,5% a 8,5%.

[18] Výsledky takmer všetkých testov sú dostupné na elektronickom médiu.

Pri porovnávaní výsledkov v rámci jednotlivých živočíchov treba vyzdvihnúť naše výsledky pre sekvencie z DNA muchy, kde sme dosiahli senzitivitu 17% a precíznosť 6,4%. Dosiahli sme najlepší výsledok, pričom *Weeder* mal senzitivitu 2% a precíznosť 3,4% a *ANN-Spec* 2% a 0,9%.

V metrike *nSn*, čo je senzitivita na úrovní motívov, sme dosiahli 14,7%, čo je tiež najlepší výsledok. Treba ale spomenúť, že sme mali pomerne nízku precíznosť, resp. vysoký počet falošných pozitív. Ak sa pozrieme na celkové hodnoty nájdených motívov, skutočných pozitív (TP) sme mali 251 a falošných pozitív (FP) 2708. *Weeder* mal TP rovné 84 a FP 207. *ANN-Spec* mal TP 81 a FP 874.

Na to, aby sme znížili počet falošných pozitív, sme sa ich snažili dodatočne odfiltrovať. Náš najlepší výsledok sme dosiahli, keď sme vracali motívy z 20 najlepších včiel po spojení všetkých populácií. Z postprocesov v kapitole 7.7.6 iba *:instace-filter* dosahoval porovnateľné no trocha horšie výsledky ako bez filtrovania. Ostatné filtre sa nám nepodarilo nastaviť tak, aby nezahadzovali príliš veľa motívov.

Sledovali sme aj vplyv paralelizácie, kde sme obmedzovali komunikáciu medzi ostrovmi. Skúšali sme aj pravdepodobnosti výmeny 0,1 a 0,5. Pri 0,3 sme dosiahli najlepší výsledok.

Pri uvoľňovaní ohodnocovacích funkcií sme dosiahli najlepší výsledok keď 8 ostrovov optimalizovalo všetky ohodnocovacie funkcie. Porovnateľný výsledok sme dosiahli, ak sme uvoľňovali iba jednu funkciu.

Prekvapivé bolo nastavenie *:pareto-epsilon*, ktoré sme opísali v kapitole 3.2.1. Jeho zvýšenie z 0 na 0,1 zlepšilo výsledok.

9 Záver

V tejto práci sme sa venovali problému hľadania motívov. Pozerali sme sa na neho ako na problém optimalizácie a riešili sme ho pomocou metaheuristík. Navrhli asynchrónne paralelné vyhľadávanie kombináciou niekoľkých existujúcich riešení.

Na najnižšej úrovní sme použili sekvenčnú metaheuristiku MOABC [17], ktorú sme rozšírili o entropiu pri vyhodnocovaní jedincov a o použitie *g-best* člena z [40] pri ich pohybe po doménovom priestore. Museli sme riešiť problém náhodného výberu, ktorý sme vyriešili ohodnotením v kapitole 7.6.3.

Na paralelizáciu sme použili model asynchrónnych ostrovov. Ostrovy si jedince nezačleňovali do populácie, ale preberali ich iba ako *g-best* člena. Jednotlivé ostrovy na seba vplývali len nepriamo. Ako metódu komunikácie sme navrhli spôsob, kde to, či si ostrovy vymenia jedincov, nezávisí priamo od topológie, ale skôr od času vyhľadávania a vzdialeností medzi ostrovmi. Na túto komunikáciu sme použili zdieľanú pamäť.

Experimentovali sme s myšlienkou špecializovaných ostrovov, kde každý ostrov ohodnocuje inú podmnožinu ohodnocovacích funkcií, ktorých sme mali štyri. Pri probléme MFP takéto vypúšťanie ohodnocovacích funkcií ale neprinieslo zlepšenie výsledkov.

Priebežné výsledky ukázali na potrebu dodatočného ohodnotenia nájdených motívov s cieľom ich prefiltrovať. Iplementovali sme preto viacero ohodnotení, ktoré sme opísali v kapitole 7.7. Ohodnotenia boli založené na algoritme *Weeder* [31]. Okrem nich sme navrhli a implementovali aj naše vedierkové ohodnotenie.

Výsledky ukázali, že metaheuristické algoritmy dávaju porovnateľné výsledky voči ostatným algoritmom. Problémom stále ostáva vysoký počet falošných pozitív. Výsledkom pomohlo použitie ϵ-dominancie.

Algoritmus sme implementovali takým spôsobom, že je možné modulárne vymieňať problém alebo metaheuristiku ostrova. Preto sme predbežné testy robili aj na probléme DTLZ7 [10].

9.1 Budúca práca

Pri pohľade na našu diplomovú prácu ako na metaheuristiku by sa dalo v budúcnosti vylepšiť viacero vecí. Zmeny je možné urobiť takmer na všetkých úrovniach. Viaceré práce pri paralelnom vyhľadávaní využívali rôzne metaheuristiky, aby sa mohli dopĺňať. My sme implementovali iba MOABC. Preskúmať by sa dal aj vplyv topológie na naše vyhľadávanie. V našej práci sme umiestňovali ostrovy do pevne stanovenej mriežky. Okrem toho by sa dala upraviť aj selekcia jedincov pri výbere tej včely, ktorú chceme akceptovať z poskytovaného paretového radu. Algoritmus by bolo vhodné vyskúšať aj na iných problémoch ako je MFP, napríklad na problémoch z triedy DTLZ7.

Pri pohľade na problém hľadania motívov by bolo vhodné doplniť lepšie filtrovanie výsledkov. Navyše, vo veľa prípadoch (ohodnocovanie, postprocessing) sa využíva na výpočet chybovosti Hammingová vzdialenosť. Kedže DNA sekvencie nepodliehajú iba substitúcii, vyskúšať by sa dali aj iné vzdialenosti, napríklad Levenshteinova.

Literatúra

[1] AL MOUBAYED, N. – PETROVSKI, A. – MCCALL, J. A novel smart multi-objective particle swarm optimisation using decomposition. In *Parallel Problem Solving from Nature, PPSN XI*. Springer, 2010. s. 1–10.

[2] ASHLOCK, D. *Evolutionary computation for modeling and optimization*. Springer Science & Business Media, 2006.

[3] BAILEY, T. L. – ELKAN, C. The value of prior knowledge in discovering motifs with MEME. In *Ismb*, 3, s. 21–29, 1995.

[4] BALLING, R. – WILSON, S. The maxi-min fitness function for multi-objective evolutionary computation: Application to city planning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2001)*, s. 1079–1084. Citeseer, 2001.

[5] BLUM, C. – ROLI, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*. 2003, 35, 3, s. 268–308.

[6] ČERNỲ, V. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*. 1985, 45, 1, s. 41–51.

[7] CRAINIC, H. T. N. Parallel meta-heuristics applications. *Alba, E. (ed.) Parallel Metaheuristics: A New Class of Algorithms*. 2005, s. 447–494.

[8] DAS, M. K. – DAI, H.-K. A survey of DNA motif finding algorithms. *BMC bioinformatics*. 2007, 8, Suppl 7, s. S21.

[9] DEB, K. et al. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*. 2002, 6, 2, s. 182–197.

[10] DEB, K. et al. *Scalable test problems for evolutionary multiobjective optimization*. Springer, 2005.

[11] D'HAESELEER, P. What are DNA sequence motifs? *Nature biotechnology*. 2006, 24, 4, s. 423–425.

[12] DORIGO, M. – MANIEZZO, V. – COLORNI, A. The Ant System: Optimization by a colony of cooperating agents. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS-PART B*. 1996, 26, 1, s. 29–41.

[13] DURILLO, J. J. – NEBRO, A. J. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*. 2011, 42, s. 760–771. ISSN 0965-9978. doi: DOI:10.1016/j.advengsoft.2011.05.014. Dostupné z: <http://www.sciencedirect.com/science/article/pii/S0965997811001219>.

[14] FREDKIN, E. Trie memory. *Communications of the ACM*. 1960, 3, 9, s. 490–499.

[15] GENDREAU, M. – POTVIN, J.-Y. *Handbook of metaheuristics*. 2. Springer, 2010.

[16] GONZÁLEZ-ÁLVAREZ, D. L. et al. Using a Parallel Team of Multiobjective Evolutionary Algorithms to Solve the Motif Discovery Problem. In *DCAI*, s. 569–576. Springer, 2010.

[17] GONZÁLEZ-ÁLVAREZ, D. L. et al. Finding motifs in DNA sequences applying a multiobjective artificial bee colony (MOABC) algorithm. In *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*. Springer, 2011. s. 89–100.

[18] HERTZ, G. Z. – STORMO, G. D. Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. *Bioinformatics*. 1999, 15, 7, s. 563–577.

[19] KARABOGA, D. Artificial bee colony algorithm. *scholarpedia*. 2010, 5, 3, s. 6915.

[20] KARGER, D. 6.854J Advanced Algorithms, Fall 2005. http://ocw.mit.edu/courses/

electrical-engineering-and-computer-science/6-854j-advanced-algorithms-fall-2005, 2005. [Navštívené 6.5.2016].

[21] KAYA, M. MOGAMOD: Multi-objective genetic algorithm for motif discovery. *Expert Systems with Applications*. 2009, 36, 2, s. 1039–1047.

[22] KIRKPATRICK, S. – VECCHI, M. P. – OTHERS. Optimization by simmulated annealing. *science*. 1983, 220, 4598, s. 671–680.

[23] KRISHNANAND, K. N. – GHOSE, D. Glowworm Swarm Optimisation: a New Method for Optimising Multi-Modal Functions. *Int. J. Comput. Intell. Stud.* May 2009, 1, 1, s. 93–119. ISSN 1755-4977. doi: 10.1504/IJCISTUDIES.2009.025340. Dostupné z: <http://dx.doi.org/10.1504/IJCISTUDIES.2009.025340>.

[24] LAUMANNS, M. et al. Combining convergence and diversity in evolutionary multiobjective optimization. *Evolutionary computation*. 2002, 10, 3, s. 263–282.

[25] LAUMANNS, M. et al. Archiving With Guaranteed Convergence And Diversity In Multi-objective Optimization. In *GECCO*, s. 439–447, 2002.

[26] LONES, M. Sean Luke: Essentials of metaheuristics. *Genetic Programming and Evolvable Machines*. 2011, 12, 3, s. 333–334.

[27] MÄRTENS, M. – IZZO, D. The asynchronous island model and NSGA-II: study of a new migration operator and its performance. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, s. 1173–1180. ACM, 2013.

[28] MARTIN, W. – LIENIG, J. – COHOON, J. P. C6. 3 Island (migration) models: evolutionary algorithms based on punctuated equilibria. *B ack et al. BFM97], Seiten C*. 6.

[29] MENCHACA-MENDEZ, A. – COELLO, C. A. C. Selection mechanisms based on the maximin fitness function to solve multi-objective optimization problems. *Information Sciences*. 2016, 332, s. 131–152.

[30] NIELSEN, S. S. et al. Novel efficient asynchronous cooperative co-evolutionary multi-objective algorithms. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, s. 1–7. IEEE, 2012.

[31] PAVESI, G. – MAURI, G. – PESOLE, G. An algorithm for finding signals of unknown length in DNA sequences. *Bioinformatics*. 2001, 17 Suppl 1, s. S207–14. ISSN 1367-4803. doi: 10.1093/bioinformatics/17.suppl˙1.S207. Dostupné z: <http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=11473011>.

[32] PAVESI, G. Competition Results of Weeder. 2004, s. 1–5. Dostupné z: <http://bio.cs.washington.edu/assessment/parameters/Weeder.pdf>.

[33] RUCIŃSKI, M. – IZZO, D. – BISCANI, F. On the impact of the migration topology on the Island Model. *Parallel Computing*. 2010, 36, 10-11, s. 555–571. ISSN 01678191. doi: 10.1016/j.parco.2010.04.002.

[34] SCHNEIDER, T. D. – STEPHENS, R. M. Sequence logos: a new way to display consensus sequences. *Nucleic acids research*. 1990, 18, 20, s. 6097–6100.

[35] TOMPA, M. et al. Assessing computational tools for the discovery of transcription factor binding sites. *Nature biotechnology*. 2005, 23, 1, s. 137–144.

[36] WEINER, P. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, s. 1–11. IEEE, 1973.

[37] WINGENDER, E. et al. TRANSFAC: A Database on Transcription Factors and Their DNA Binding Sites. *Nucleic Acids Research*. January 1996, 24, 1, s. 238–241. ISSN 0305-1048. doi: 10.1093/nar/24.1.238. Dostupné z: <http://dx.doi.org/10.1093/nar/24.1.238>.

[38] XIAO, N. – ARMSTRONG, M. P. A specialized island model and its application in multiobjective optimization. In *Genetic and Evolutionary Computation Conference*, s. 1530–1540. Springer, 2003.

[39] YANG, X.-S. – KARAMANOGLU, M. – HE, X. Multi-objective flower algorithm for optimization. *Procedia Computer Science*. 2013, 18, s. 861–868.

[40] ZHU, G. – KWONG, S. Gbest-guided artificial bee colony algorithm for numerical function optimization. *Applied Mathematics and Computation*. 2010, 217, 7, s. 3166–3173.

A Elektronické médium

Obsah CD priloženého k dokumentu:

/dokument

- Diplomová Práca spolu s anotáciami v slovenskom a anglickom jazyku

/dokument/latex

- súbory projektu v LaTeX

/source

- zdrojové kódy

/source/outputs/results

- výstupy aplikácie, tj. výsledky jednotlivých testov + poznámky

/source/target/

- spustiteľné jar súbory

/source/target/coverage

- pokrytie kódu testami vo formáte HTML

/source/target/doc

- vygenerovaná dokumentácia vo formáte HTML

/test data/

- testovacie vstupy

readme.txt

- popis obsahu CD

B Plán práce

Na tento semester sme mali plán práce v nasledujúcom znení:

V nasledujúcom semestri by sme chceli čím skôr doimplementovať chýbajúcu funkcionalitu. Následne na základe výsledkov testov a ďalšieho výskumu iteratívne upravovať riešenie tak, aby dosiahlo čo najlepšie výsledky.

Tabuľka B.1: *Plán práce*

Týždeň	Plán
1.	Implementácia Crowding-Distance a/alebo Maximin
2. - 4.	Implementácia mechanizmov na komunikáciu medzi ostrovmi
5. - 6.	Implementácia problému hľadania motívov DNA spolu s testovačom
7. -	Testovanie, nastavovanie parametrov a písanie DP

Pri plnení tohto plánu sme nedodržiavali postupnosť implementácie, kedže sme začali problémom hľadania motívov. Následne sme implementovali *crowding distance* a komunikácie. Nakoniec sme implementovali testovač. Z výsledkov priebežných testov vyplynula potreba implementovať postprocesing. Teda ku koncu semestra sme testovali a implementovali rôzne verzie filtrovania výsledkov.

C Používateľská príručka

Diplomová práca bola implementovaná ako konzolová aplikácia. Pre jej spustenie je potrebné mať nainštalovanú Javu vo verzii 8 a prístup k internetu, aby mohla vyhodnotiť výsledky. Aplikácia potrebuje mať parametrami nastavené cesty ku vstupom ako aj konfigurácie, ktoré ma použiť pri vyhľadávaní. Príkaz na spustenie môžeme vidieť v ukážke C.1.

```
java -jar fiit-dp-standalone.jar <parametre>
```

Ukážka C.1: *Zapnutie vyhľadávania.*

Aplikácia pracuje s nasledujúcimi parametrami:

- `-i, --input INPUT-FOLDER` – (vyžadované) cesta k adresáru so vstupnými .fasta súbormi. Môže obsahovať podadresáre.
- `-c, --config CONFIG-FOLDER` – (vyžadované) cesta k adresáru s konfiguráciami vo formáte .edn.
- `-s, --search SEARCH-SYMBOL` – (vyžadované) meno konfigurácie vyhľadávania.
- `-p, --problem PROBLEM-SYMBOL` – (vyžadované) meno konfigurácie problému.
- `-o, --postprocess POSTPROCESS-SYMBOL` – meno postprocesu (kapitola č. 7.7.6).
- `-f` – Filtrovanie výsledkov na základe ohodnotenia.
- `-n, --notes NOTE-STRING` – poznámka k vyhľadávaniu.
- `-h, --help` – vypíše HELP.

Po ukončení vyhľadávania aplikácia v adresári *./results/* vytvorí nový adresár s výsledkami. Tento novovytvorený adresár bude mať meno vo formáte: `-<search>-<problem>-<note>-<time>`

kde:

- ▷ `<search>` je meno konfigurácie vyhľadávania,
- ▷ `<problem>` je meno konfigurácie problému,
- ▷ `<note>` je poznámka, ak bola špecifikovaná parametrom, a
- ▷ `<time>` čas kedy sa ukončilo vyhľadávanie.

Tento adresár bude obsahovať 3 súbory:

- ▷ `config` súbor s informáciami o použitých nastaveniach,
- ▷ `res.html` súbor s výslednými metrikami (kapitola č. 8.2),
- ▷ `<results>.fasta` súbor vo formáte $fasta$ s nájdenými motívmi. Meno tohto súboru je podobné ako meno adresára.

Program počas vyhľadávania informuje o svojom priebehu výpismi do konzoly.

C.1 Konfigurácie

Medzi vstupné parametre aplikácie patria konfiguračné súbory a symboly pre vyhľadávanie a problém. Aplikácia funguje tak, že prečíta všetky konfigurácie z adresára a spojí ich. Preto konfigurácie pre vyhľadávanie musia byť v súbore na ceste `:search :algorithm <meno_konfiguracie>` a konfigurácie problému na ceste `:search :problem <meno_konfiguracie>`. Tieto súbory sú vo formáte edn.

V ukážke č. C.2 uvádzame príklad konfiguračného súboru, ktorý obsahuje konfiguráciu pre vyhľadávanie s menom $moabc$. Väčšina parametrov je opísaná v kapitole č. 8.3. Okrem nich sa v týchto parametroch nachádzajú ale aj kľúče *:ns* a *:search-fn*. Tie slúžia na špecifikovanie balíka s vyhľadávacím algoritmom, resp. na špecifikovanie inicializačnej funkcie.

V ukážke č. C.3 uvádzame príklad konfiguračného súboru, ktorý obsahuje konfiguráciu pre problém s menom mfp. Ten podobne ako konfigurácia vyhľadávania obsahuje kľúče *:ns* a *:problem-fn*.

```
{:search
  {:algorithm
    {:moabc {
        :ns fiit-dp-pmsoa.search.moo.moabc
        :coef-C 1.5
        :size [0.5 0.4 0.1]
        :search-fn search
        :population-count 50
        :num-of-iterations 20
        :share-prob 0.5
        :take-prob 0.5}}}}
```

Ukážka C.2: *Príklad konfigurácie zo súboru moabc.edn.*

```
{:search
  {:problem
    {:mfp {
      :ns fiit-dp-pmsoa.search.problem.mfp
      :support-coeff 0.5 ;; minimal same nucleotides 50%
      :problem-fn problem
      :min-length 6
      :max-length 64
      :min-obj 1}}}}
```

Ukážka C.3: *Príklad konfigurácie zo súboru mfp.edn.*

Tieto konfigurácie sú rozdelené do dvoch súborov iba kvôli prehľadnosti. Súbor môže obsahov ľubovoľné množstvo konfigurácií. Pri spúštaní algoritmu je iba potrebné špecifikovať meno konfigurácie vyhľadávania a problému (v tomto prípade *moabc* a *mfp*).

D Technická dokumentácia

Program bol implementovaný v programovacom jazyku Clojure[19] vo verzii 1.9.0-alpha10, ktorý využíva Java JVM. Na kompilovanie a správu závislostí využívame nástroj Leiningen[20].

D.1 Leiningen

V tejto kapitole uvádzame základne príkazy na prácu s projektom. Všetky treba volať v adresári so zdrojovými súbormi. Pre spustenie aplikácie:

```
lein run -- <args>
```

Ukážka D.1: *Zapnutie vyhľadávania pomocou leiningen.*

Pre nainštalovanie závislostí:

```
lein deps
```

Ukážka D.2: *Inštalovanie závislostí.*

Spustenie testov:

```
lein test
```

Ukážka D.3: *Spustenie testov.*

[19] https://clojure.org/
[20] https://leiningen.org/

Spustenie pokrytia testami[21]:

```
lein cloverage
```

Ukážka D.4: *Pokrytie testami.*

Vygenerovanie dokumentácie[22]:

```
lein codox
```

Ukážka D.5: *Generovanie dokumentácie.*

D.2 Základný prehľad balíkov

V tejto časti uvádzame hlavné balíky spolu s ich popisom. Vygenerovaná dokumentácia k funkciám v balíkoch sa dá nájsť na elektronickom médiu.

Hlavné balíky[23]:

- *cli.main* – funkcionalita spojená s CLI.
- *parallel-search.** – balíky, ktoré sa starajú o zdieľanú pamäť a migrácie. Okrem toho obsahujú funkcie slúžiace na beh vyhľadávania ostrova vo vlákne.
- *postprocess.** – obsahuje funkcionalitu postprocesu opísanú v kapitole č. 7.7.
- *search.island* – obsahuje definíciu štruktúry ostrova.
- *search.solution* – obsahuje definíciu štruktúry riešenia.
- *search.moo.** – balíky metaheuristík. Obsahuje balík *moabc* na vyhľadávanie včelami.
- *search.problem.** – balíky problémov. Obsahuje problémy *dtlz7* a *mfp*.

[21] Treba mať nainštalované rozšírenie *lein-cloverage*
[22] Treba mať nainštalované rozšírenie *lein-codox*
[23] Všetky majú prefix projektu, teda *fiit-dp-pmsoa*.

- *parallel-search* – funkcionalita vytvárania ostrovov a spúšťanie paralelného vyhľadávania a volanie postprocesov.
- *test.test-runner* – spúšťanie vyhľadávaní na základe mien konfigurácii, vyhodnotenie a ukladanie výsledkov.

D.3 Problém

Ostrov sa pozerá na problém ako na štruktúru, ktorá mu poskytuje funkcie a dáta. Takáto štruktúra je znázornená v ukážke D.6.

```
{:is-valid? is-valid-fn
 :evaluate evaluate-fn
 :generate generate-fn
 :objectives objectives}
```

Ukážka D.6: *Štruktúra problému.*

Na vytvorenie nového problému je treba zadefinovať tri funkcie:

- *is-valid-fn* – má za parameter jedinca a vráti $true$ alebo $false$ podľa toho, či je validný,
- *evaluate-fn* – má za parameter jedinca, ktorého ohodnotí,
- *generate-fn* – vytvorí nového, validného jedinca.

Okrem toho obsahuje aj $objectives$, čo sú kľúče, podľa ktorých sa dá pristupovať k ohodnoteniu jedinca. Bližšie sme tieto funkcie opísali v kapitole č. 7.2.2.

V konfigurácii treba potom špecifikovať meno balíka, kde sa nachádza tento problém a meno funkcie, ktorá nám vráti vyššie opísanú štruktúru. Táto funkcia bude zavolaná s konfiguráciou problému ako parametrom a slúži na inicializáciu celého problému.

D.4 Vyhľadávacia metaheuristika

Tak ako pri probléme, aj na vyhľadávaciu metaheuristiku sa ostrov pozerá ako na dátovú štruktúru. V ukážke D.7 uvádzame príklad takejto štruktúry.

```
{:iterate do-iterate
 :generate-population generate-population
 :init init-island}
```

Ukážka D.7: *Štruktúra problému.*

Funkcia *do-iterate* má za parameter ostrov a vykoná jednu iteráciu metaheuristiky. Funkcia *generate-population* má ako parameter objekt problému a počet jedincov. *init-island* berie ako argument ostrov a slúži na jeho inicializáciu z pohľadu metaheuristiky. Bližšie sú tieto funkcie opísané v kapitole 7.2.3.

V konfigurácii treba opäť uviesť meno balíka, kde sa nachádza táto metaheuristika a meno funkcie, ktorá nám vráti vyššie opísanú štruktúru.

Takýto opis štruktúr pre problém a vyhľadávanie by mohol byť v OOP paradigme analogický k rozhraniam alebo abstraktným objektom.

D.5 Spustenie vyhľadávania

Na to aby sme vedel spustiť vyhľadávanie, potrebujeme najprv vyžiadať balíky *fiit-dp-pmsoa.test.test-runner* a *fiit-dp-pmsoa.core*. Potom treba nastaviť príslušné cesty k vstupom, konfiguráciam a výstupom. Následne je potrebné zavolať funkciu *run-test!* z balíka *fiit-dp-pmsoa.test.test-runner*. Táto funkcia má za parametre symboly pre konfigurácie vyhľadávania a problému; potom symbol, ktorý určuje typ postprocesu; boolean, ktorý zodpovedá filtrovaniu na základe ohodnotenia a na koniec číslo behu a voliteľné poznámky. Celý postup je možno vidieť v ukážke D.8.

```
(require
  '[fiit-dp-pmsoa.test.test-runner :as tr]
  '[fiit-dp-pmsoa.core :as core])

;; nastavenie ciest
(alter-var-root #'tr/*seq-path* (constantly "./input/"))
(alter-var-root #'core/*config-path* (constantly "./cfgs/"))
(alter-var-root #'tr/*res-path* (constantly "./results/"))

;; spustenie vyhladavania
;; prva konfiguracia
(tr/run-test! :moabc2-300-50 :mfp2-4obj :instance-out false 1)
;; druha konfiguracia
(tr/run-test! :moabc2-300-50 :mfp3-4obj :instance-out false 1)
```

Ukážka D.8: *Spustenie vyhľadávania v Clojure.*

E Výsledky ostatných algoritmov

V tejto prílohe uvádzame výsledky, ktoré dosiahli ostatné algoritmy nad rovnakým datasetom ako my. Tieto výsledky sú dostupné na stránke datasetu[24].

Tabuľka E.1: *YMF.*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	0	512	671	41817	0	79	51		0.000	0.000	0.988	0.000	-0.014	0.000	0.000	0.000
Human	210	1961	4909	281920	22	252	276		0.041	0.097	0.993	0.030	0.052	0.074	0.080	0.077
Mouse	166	657	1473	53204	20	90	78		0.101	0.202	0.988	0.072	0.125	0.204	0.182	0.193
Yeast	178	362	1056	59404	21	41	54		0.144	0.330	0.994	0.112	0.208	0.280	0.339	0.309
Total	554	3492	8109	436345	63	462	459		0.064	0.137	0.992	0.046	0.082	0.121	0.120	0.120

Tabuľka E.2: *Weeder.*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	8	224	663	42105	1	28	50		0.012	0.034	0.995	0.009	0.011	0.020	0.034	0.027
Human	278	734	4841	283147	32	92	266		0.054	0.275	0.997	0.047	0.115	0.107	0.258	0.183
Mouse	101	475	1538	53386	12	55	86		0.062	0.175	0.991	0.048	0.088	0.122	0.179	0.151
Yeast	361	315	873	59451	39	32	36		0.293	0.534	0.995	0.233	0.386	0.520	0.549	0.535
Total	748	1748	7915	438089	84	207	438		0.086	0.300	0.996	0.072	0.152	0.161	0.289	0.225

Tabuľka E.3: *SeSiMCMC*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	68	1194	603	41135	5	35	46		0.101	0.054	0.972	0.036	0.054	0.098	0.125	0.112
Human	235	8159	4884	275722	20	297	278		0.046	0.028	0.971	0.018	0.013	0.067	0.063	0.065
Mouse	102	2393	1537	51468	10	95	88		0.062	0.041	0.956	0.025	0.015	0.102	0.095	0.099
Yeast	125	2067	1109	57699	7	90	68		0.101	0.057	0.965	0.038	0.050	0.093	0.072	0.083
Total	530	13813	8133	426024	42	517	480		0.061	0.037	0.969	0.024	0.049	0.080	0.075	0.078

Tabuľka E.4: *QuickScore*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	0	660	671	41669	0	115	51		0.000	0.000	0.984	0.000	-0.016	0.000	0.000	0.000
Human	26	2590	5093	281291	0	474	298		0.005	0.010	0.991	0.003	-0.006	0.000	0.000	0.000
Mouse	59	597	1580	53264	8	110	90		0.036	0.090	0.989	0.026	0.039	0.082	0.068	0.075
Yeast	66	1009	1168	58757	9	198	66		0.053	0.061	0.983	0.029	0.039	0.120	0.043	0.082
Total	151	4856	8512	434981	17	897	505		0.017	0.030	0.989	0.011	0.008	0.033	0.019	0.026

[24] http://bio.cs.washington.edu/assessment/assessment_result.html

Tabuľka E.5: *Oligodyad-Analysis*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	0	600	671	41729	0	84	51		0.000	0.000	0.986	0.000	-0.015	0.000	0.000	0.000
Human	190	698	4929	283183	18	102	280		0.037	0.214	0.998	0.033	0.083	0.060	0.150	0.105
Mouse	44	368	1595	53493	6	57	92		0.027	0.107	0.993	0.022	0.039	0.061	0.095	0.078
Yeast	111	225	1123	59541	14	32	61		0.090	0.330	0.996	0.076	0.164	0.187	0.304	0.246
Total	345	1891	8318	437946	38	275	484		0.040	0.154	0.996	0.033	0.069	0.073	0.121	0.097

Tabuľka E.6: *MITRA*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	0	167	671	42162	0	16	51		0.000	0.000	0.996	0.000	-0.008	0.000	0.000	0.000
Human	125	2527	4994	281354	12	244	286		0.024	0.047	0.991	0.016	0.021	0.040	0.047	0.044
Mouse	10	637	1629	53224	2	59	96		0.006	0.015	0.988	0.004	-0.009	0.020	0.033	0.027
Yeast	137	761	1097	59005	12	66	63		0.111	0.153	0.987	0.069	0.115	0.160	0.154	0.157
Total	272	4092	8391	435745	26	385	496		0.031	0.062	0.991	0.021	0.031	0.050	0.063	0.057

Tabuľka E.7: *MEME*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	28	637	643	41692	3	51	48		0.042	0.042	0.985	0.021	0.027	0.059	0.056	0.057
Human	195	3034	4924	280847	18	204	280		0.038	0.060	0.989	0.024	0.034	0.060	0.081	0.071
Mouse	120	777	1519	53084	14	66	84		0.073	0.134	0.986	0.050	0.079	0.143	0.175	0.159
Yeast	238	388	996	59378	23	37	52		0.193	0.380	0.994	0.147	0.260	0.307	0.383	0.345
Total	581	4836	8082	435001	58	358	464		0.067	0.107	0.989	0.043	0.071	0.111	0.139	0.125

Tabuľka E.8: *Improbizer*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	10	558	661	41771	1	43	50		0.015	0.018	0.987	0.008	0.002	0.020	0.023	0.021
Human	213	4261	4906	279620	21	413	277		0.042	0.048	0.985	0.023	0.028	0.070	0.048	0.059
Mouse	177	1275	1462	52586	22	115	76		0.108	0.122	0.976	0.061	0.089	0.224	0.161	0.193
Yeast	194	1848	1040	57918	20	130	55		0.157	0.095	0.969	0.063	0.099	0.267	0.133	0.200
Total	594	7942	8069	431895	64	701	458		0.069	0.070	0.982	0.036	NaN	0.123	0.084	0.103

Tabuľka E.9: *GLAM*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	2	406	669	41923	0	42	51		0.003	0.005	0.990	0.002	-0.008	0.000	0.000	0.000
Human	121	3170	4998	280711	12	188	286		0.024	0.037	0.989	0.015	0.016	0.040	0.060	0.050
Mouse	12	627	1627	53234	1	64	97		0.007	0.019	0.988	0.005	-0.007	0.010	0.015	0.013
Yeast	88	1416	1146	58350	11	179	64		0.071	0.059	0.976	0.033	0.043	0.147	0.058	0.102
Total	223	5619	8440	434218	24	473	498		0.026	0.038	0.987	0.016	NaN	0.046	0.048	0.047

Tabuľka E.10: *Consensus*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	0	329	671	42000	0	30	51		0.000	0.000	0.992	0.000	-0.011	0.000	0.000	0.000
Human	0	0	5119	283881	0	0	298		0.000	NaN	1.000	0.000	NaN	0.000	NaN	NaN
Mouse	80	673	1559	53188	10	72	88		0.049	0.106	0.988	0.035	0.053	0.102	0.122	0.112
Yeast	98	392	1136	59374	11	35	64		0.079	0.200	0.993	0.060	0.115	0.147	0.239	0.193
Total	178	1394	8485	438443	21	137	501		0.021	0.113	0.997	0.018	0.040	0.040	0.133	0.087

Tabuľka E.11: *ANN-Spec*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	17	953	654	41376	1	105	50		0.025	0.018	0.977	0.010	0.002	0.020	0.009	0.015
Human	462	4016	4657	279865	49	449	249		0.090	0.103	0.986	0.051	0.081	0.164	0.098	0.131
Mouse	71	1578	1568	52283	8	180	90		0.043	0.043	0.971	0.022	0.014	0.082	0.043	0.062
Yeast	204	1252	1030	58514	23	140	52		0.165	0.140	0.979	0.082	0.133	0.307	0.141	0.224
Total	754	7799	7909	432038	81	874	441		0.087	0.088	0.982	0.046	NaN	0.155	0.085	0.120

Tabuľka E.12: *AlignACE*

Data set	nTP	nFP	nFN	nTN	sTP	sFP	sFN		nSn	nPPV	nSp	nPC	nCC	sSn	sPPV	sASP
Fly	0	110	671	42219	0	10	51		0.000	0.000	0.997	0.000	-0.006	0.000	0.000	0.000
Human	201	1759	4918	282122	22	156	276		0.039	0.103	0.994	0.029	0.053	0.074	0.124	0.099
Mouse	47	911	1592	52950	3	80	95		0.029	0.049	0.983	0.018	0.015	0.031	0.036	0.033
Yeast	229	1009	1005	58757	21	83	54		0.186	0.185	0.983	0.102	0.168	0.280	0.202	0.241
Total	477	3789	8186	436048	46	329	476		0.055	0.112	0.991	0.038	0.066	0.088	0.123	0.105

F Článok na IIT.SRC 2017

Na nasledujúcich stranách uvádzame článok na IIT.SRC 2017 spolu s prezentovaným posterom.

Motif Finding In DNA Sequences

Róbert CUPRIK*

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
robertcuprik@hotmail.sk

Abstract. Motif finding in DNA sequences is a problem that is currently intensively studied in bioinformatics. The problem is to locate short nucleotide sequences called motifs. These sequences are important as they – among other functions – indicate binding sites used for transcription. We approach this problem as a problem of optimization where the goal is to optimize a set of given objective functions while satisfying constraints. Our approach is based on using metaheuristics. In this paper, we describe a parallel cooperative algorithm that manages cooperation of multiple sequential metaheuristics similar to the island model known in evolutionary computation.

1 Introduction

Motifs are short sequences in DNA also called binding sites [5]. This parts precede other parts of DNA that are used by individual to produce proteins. Therefore knowledge of their position helps scientists to better understand DNA.

Search for these motifs is complicated as the only thing we know is that they tend to repeat between genes. This is further complicated by the fact that motifs suffer from mutations such as nucleotide substitution [2].

This problem can be viewed as a problem of optimisation. For this category of problems metaheuristic algorithms can be used. Metaheuristics especially bio-inspired algorithms are versatile as they can solve various problems from DNA assembly problem in [1] or story tracking in [13] to interplanetary trajectory design in [9].

In this paper we describe one approach to this problem. We combine known metaheuristic MOABC [6] with parallelization using Specialized Island Model [16]. We also experiment with using g-best from [17] as means of individual assimilation. As topology in migration we propose usage of neighbourhoods.

This paper is structured as follows. In section 1.1 we describe Motif Finding Problem (MFP) as a problem of optimization. In section 1.2 we specify solution definition as well as objectives being optimized. In sections 2 and 3 we describe sequential metaheuristic that is used in islands. Sections 4 and 5 are dedicated to Island model and migration. In section 6 and 7 are evaluation and conclusions.

1.1 Motif Finding Problem

Motif finding problem is in [8] defined as multiple sequence local alignment problem, where task is to detect overrepresented motifs in multiple sequences [2]. If objective functions are given, such problem can be solved as a problem of optimization described mathematically [8] for M objectives as :

$$\text{Minimize} f(x) = [f_i(x), i = 1, \ldots, M] \quad (1)$$

satisfying constraints:

$$g_j(x) \leq 0 \qquad k = 1, 2, ...J \quad (2)$$

$$h_i(x) = 0 \qquad j = 1, 2, ...K \quad (3)$$

In MOABC [6] following objectives to optimize were used:

- maximize length of the motif,
- maximize similarity,
- maximize support

subjected to following constraints:

- minimal support,
- minimal complexity,
- motif length in range $[6, 64]$

1.2 Solution evaluation

Solution that is subject to optimization is defined as vector of decision variables where the first variable

* Master study programme in field: Information Systems
Supervisor: Professor Pavol Návrat, Institute of Informatics, Information Systems and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

represents motif length and following variables specify motif position in each input sequence.

Motifs on these positions in each sequence are called candidate motifs. From this motifs consensus motif is computed as a sequence of most dominant nucleotides at each position. Only candidate motifs that have at least 50% of nucleotides same as consensus motif are considered as a part of the final motif.

This is optimized using support objective which is directly relative to the amount of candidate motifs used in final motif. Position Frequency Matrix (PFM) is then computed from considered candidate motifs as a matrix of frequencies of nucleotides (A,C,T,G) at each position. Example of such PFM can be seen in Table 1.

Table 1. Part of PFM for motif YJL056C in ZAP1 gene[1].

A	0.502	0.9243	0.0137	0.0029	...
T	0.0912	0.0485	0.006	0.0129	...
G	0.3905	0.0168	0.0022	0.003	...
C	0.0163	0.0104	0.9781	0.9812	...

This PFM is subsequently used in computation of other objectives. In equation 4 similarity is computed, where $f(b,i)$ is frequency of nucleotide $b \in \{A,T,G,C\}$ on i-th position and l is length of the motif.

$$Similarity = \frac{\sum_{i=1}^{l} max_b f(b,i)}{l} \quad (4)$$

Complexity objective is described in equation 5. n_i is number of nucleotides $i \in \{A,C,G,T\}$ in a final motif sequence.

$$Complexity = \log_N \frac{l!}{\prod(n_i)!} \quad (5)$$

We use entropy (equation 6) as an addition to objective functions used in MOABC. Entropy is similar to similarity, but uses whole distribution of frequencies for given position not only the most dominant nucleotide.

$$Entropy = -\frac{\sum_{i=1}^{l} \sum_b f(b,i) \log_2(f(b,i))}{l} \quad (6)$$

2 Gbest-guided ABC

Artificial Bee Colony (ABC) [7] is bio-inspired algorithm utilizing concepts of Swarm Intelligence (SI) based on foraging behaviour of honey bees. In ABC there are three groups of bees – employed bees, onlooker bees and scouts.

Employed bees locate food source, fly back to hive and then dance to share information about the found food source with other bees. Time spent on dancing reflects the amount of nectar found. In ABC food source represents one solution to domain problem and amount of nectar is corresponding to its fitness. Movement of employed bee is performed using equation 7.

$$v_{ij} = x_{ij} + \phi_{ij}(x_{ij} - x_{kj}) \quad (7)$$

Current position of bee (position of food source) is represented by vector of decision variables. In equation 7 x_i represents position of bee i. x_{ij} is value of j-th decision variable of i-th bee. New position of employed bee v_i is computed using its previous position x_i and random other bee position x_k. ϕ_{ij} is random number in range $[-1,1]$.

In Gbest-guided ABC [17] was this equation extended using global best bee (g-best) which is the best bee that was found in previous iterations. In equation 8 is g-best represented by vector y. Ψ_{ij} is random number between $[0,C]$.

$$v_{ij} = x_{ij} + \phi_{ij}(x_{ij} - x_{kj}) + \Psi_{ij}(y_j - x_{ij}) \quad (8)$$

This extension was proposed to improve exploitation of ABC. We use g-best term as means of influence between different islands and their respective populations.

Onlooker bees watch employed bees dancing and then choose which food source to visit. In ABC onlooker bees use same movement equation as worker bees. Difference is in bee x_i which is chosen from worker bees with probability based on its fitness.

Scount bees search randomly for new solutions and improve exploration.

Each of these groups has its respective phase in ABC – employed bees phase, onlookers phase and scouts phase. In iteration process these phases alternate until ending criterion is met e.g. maximal number of iterations is reached.

3 MOABC

Multi-objective Artificial Bee Colony (MOABC) [6] is multi-objective optimization algorithm based on ABC. As ABC was designed to solve single objective optimization problems, MOABC was proposed to solve multi-objective Motif Finding Problem.

Individuals in multi-objective space are evaluated by more than one objective function. Therefore final solution is not only one individual but multiple

[1] http://yetfasco.ccbr.utoronto.ca/showPFM.php?mot=YJL056C_2097.0

individuals that make up group of pareto optimal solutions called pareto front. Solutions belong to pareto front if they are not dominated by any other solution. Solution x it's said to dominate solution y if for every objective function f_i, $f_i(x)$ is better or equal than $f_i(y)$ and there exists at least one i for which $f_i(x)$ is strictly better than $f_i(y)$.

In MOABC population is represented as a vector of bees where bee position in vector depends on its fitness. Better solutions present at start of the vector are followed by worse ones. Vector is then split into three parts. First N_e positions represent employed bees, second N_o onlooker bees and final N_s scout bees. $N_e + N_o + N_s = N$, where N is number of bees in population.

Simple comparison is not sufficient to sort bees in this vector. MOABC uses non-dominated sort and crowding distance from NSGA-II [3]. Non-dominated sort is based on dominance described above. In this sort multiple ranked fronts are created. First front contains pareto optimal solutions that are not dominated by any other solution. Second front contains solutions that are only dominated by solutions in first front. Third contains solutions that are dominated by solutions from second and so on.

Membership in ranked fronts its not enough to get precise position in population vector. Additional metric to determine differences inside of front is needed. This is where crowding distance is used as it favours solutions that are not present in dense positions of objective space, therefore promoting diversification of solutions. In final position vector bees are sorted in a way where at the beginning are bees that are not dominated and are not present in crowded areas.

In MOABC same bee phases as in ABC are followed with non-dominated sort and crowding distance.

In our implementation we use g-best in movement computation in addition to original MOABC.

4 Asynchronous Island cooperation

Island model [10] is parallel cooperative algorithm that works with sequential Evolutionary Algorithms (EA). Each island has its own population and using EA iteratively searches for better solutions. Islands work in parallel for predefined amount of iterations. When this iterations are finished migration begins.

In migration, individuals (solutions) are exchanged between islands using migration topology. Sequential EA in parallel and subsequent migration make up one iteration of island called epoch. Epochs repeat for predefined amount of times.

There are multiple existing topologies [12] such as chains, rings or hypercubes. We use fully connected topology, but migration is not based on connections between islands but on their distances.

Island only sees other islands in its neighbourhood. For each island I_j its neighbourhood in time t is defined as $N_{jt} = \{I_k, d(I_j, I_k) \leq nd(t) \wedge k \neq j\}$. $d(I_j, I_k)$ is distance between islands I_j and I_k. $nd(t)$ is a function of time that changes neighbourhood size. As search progresses neighbourhood gets bigger. Island chooses n other islands with which it exchanges solutions from its neighbourhood with probability that is based on their respective distances.

In our implementation we use asynchronous variation of island model similar to those used in [9] and [11]. In these models islands do not wait for each other but merely provide their solutions for others to retrieve asynchronously using shared memory.

Moreover, in our implementation assimilation of individual after migration does not consists of injecting it into population. We use it as a g-best individual for a given island.

5 Specialized Island Model

To promote diversification in solutions Specialized Island Model (SIM) [16] was proposed. In this model island optimizes only a subset of objective functions. Each island then progresses to different part of global pareto front. We use this model asynchronously with topology based on neighbourhood.

6 Evaluation

Tompa et al. [14] created dataset designed to test and evaluate motif finding algorithms. As not all real motifs are known, this dataset uses both real and generated data. It consists of three types of input sequences – real sequences from TRANSFAC database [15] with known motifs, Markov chain generated sequences with planted motifs and randomly selected sequences with planted motifs. It contains 56 inputs from each group and sequences are from human, mouse, fly and yeast genome.

There are also known results from various other algorithms ranging from exhaustive search to neural network based ones. In Table 2 we can see results obtained by our algorithm in preliminary testing compared to other results. *MOABC+Islands* is our current algorithm and *MOABC+Entropy* is the best result we achieved in our previous work.

We show true/false positive motifs found (sTP/sFP) and sensitivity (sSn) computed as $nSn = \frac{nTP}{nTP+nFN}$.

Table 2. Results obtained by Tompa et. al. in comparison to our result.

algorithm	**sTP**	**sFP**	**sSn**
MEME	58	358	0.111
GLAM	24	473	0.046
Weeder	**84**	**207**	**0.161**
QuickScore	17	897	0.033
ANN-Spec	81	874	0.155
MOABC+Entropy	52.96	491.13	0.1189
MOABC+Islands	*86*	*1521*	*0.057*

7 Conclusion

In this paper we describe parallel cooperative metaheuristic algorithm that is based on Specialized Island Model (SIM) with neighbourhood. Islands use MOABC as their sequential algorithm used for optimization. Our contribution is in usage of g-best as a method of assimilation as well as neighbourhood for migration. We apply this algorithm to a real life problem of finding DNA motifs.

We performed preliminary testing using benchmark problem DTLZ7 [4]. Results with single island showed the need for diversification using crowding distance or SIM. Results also showed improvements in fitness once some of the objective functions were partially ignored as they would be in SIM.

After implementation of crowding distance we obtained results for Motif Finding Problem shown in Table 2. This result is currently worse than results obtained in our previous work as the amount of false positive motifs is too high. We aim to improve this result by better parametrization – which requires more test runs – and by tweaks to our algorithm such as result filtering to suppress false positives.

Acknowledgement: This work was partially supported by the Scientific Grant Agency of Slovak Republic, grant No. VG 1/0752/14.

References

[1] BouEzzeddine A., Kasala S., Návrat P.: Applying the Firefly Approach to the DNA Fragments Assembly Problem. *Annales Univ. Sci. Budapest., Sect. Comp. (2014)*, 2014, vol. 42, pp. 69–81.

[2] Das, M.K., Dai, H.K.: A survey of DNA motif finding algorithms. *BMC bioinformatics*, 2007, vol. 8, no. Suppl 7, p. S21.

[3] Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*, 2002, vol. 6, no. 2, pp. 182–197.

[4] Deb, K., Thiele, L., Laumanns, M., Zitzler, E.: *Scalable test problems for evolutionary multiobjective optimization*. Springer, 2005.

[5] D'haeseleer, P.: What are DNA sequence motifs? *Nature biotechnology*, 2006, vol. 24, no. 4, pp. 423–425.

[6] González-Álvarez, D.L., Vega-Rodríguez, M.A., Gómez-Pulido, J.A., Sánchez-Pérez, J.M.: Finding motifs in DNA sequences applying a multiobjective artificial bee colony (MOABC) algorithm. In: *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*. Springer, 2011, pp. 89–100.

[7] Karaboga, D.: Artificial bee colony algorithm. *scholarpedia*, 2010, vol. 5, no. 3, p. 6915.

[8] Kaya, M.: MOGAMOD: Multi-objective genetic algorithm for motif discovery. *Expert Systems with Applications*, 2009, vol. 36, no. 2, pp. 1039–1047.

[9] Märtens, M., Izzo, D.: The asynchronous island model and NSGA-II: study of a new migration operator and its performance. In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, ACM, 2013, pp. 1173–1180.

[10] Martin, W., Lienig, J., Cohoon, J.P.: C6. 3 Island (migration) models: evolutionary algorithms based on punctuated equilibria. *B ack et al. BFM97], Seiten C*, vol. 6.

[11] Nielsen, S.S., Dorronsoro, B., Danoy, G., Bouvry, P.: Novel efficient asynchronous cooperative coevolutionary multi-objective algorithms. In: *Evolutionary Computation (CEC), 2012 IEEE Congress on*, IEEE, 2012, pp. 1–7.

[12] Ruciński, M., Izzo, D., Biscani, F.: On the impact of the migration topology on the island model. *Parallel Computing*, 2010, vol. 36, no. 10, pp. 555–571.

[13] Sabo S., Kovarova A., Návrat P.: Multiple developing news stories identified and tracked by social insects and visualized using the new galactic streams and concurrent streams metaphors. *International Journal of Hybrid Intelligent Systems*, 2015, vol. 12, pp. 27–39.

[14] Tompa, M., Li, N., Bailey, T.L., Church, G.M., De Moor, B., Eskin, E., Favorov, A.V., Frith, M.C., Fu, Y., Kent, W.J., et al.: Assessing computational tools for the discovery of transcription factor binding sites. *Nature biotechnology*, 2005, vol. 23, no. 1, pp. 137–144.

[15] Wingender, E., Dietze, P., Karas, H., Knüppel, R.: TRANSFAC: A Database on Transcription Factors and Their DNA Binding Sites. *Nucleic Acids Research*, 1996, vol. 24, no. 1, pp. 238–241.

[16] Xiao, N., Armstrong, M.P.: A specialized island model and its application in multiobjective optimization. In: *Genetic and Evolutionary Computation Conference*, Springer, 2003, pp. 1530–1540.

[17] Zhu, G., Kwong, S.: Gbest-guided artificial bee colony algorithm for numerical function optimization. *Applied Mathematics and Computation*, 2010, vol. 217, no. 7, pp. 3166–3173.

Motif Finding in DNA Sequences

Using Multiobjective Artificial Bee Colony and Asynchronous Island Model

Róbert Cuprik
email: robertcuprik@hotmail.com
supervised by Professor Pavol Návrat

What Is DNA Motif?

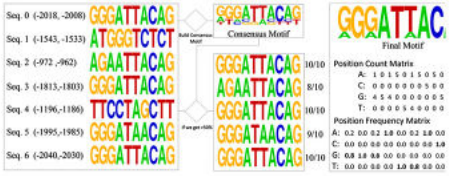
- **short substring pattern**
- **repeatedly present at various positions**
- **subject to mutations such as insertion, deletion or substitution**
- **holds some biologically significant information**

Optimization Problem

Maximize given objectives:
- Length of motif
- Support
- Similarity / Entropy

Satisfy given constraints:
- Motif length 7 - 64
- Minimum Support value
- Minimum motif complexity

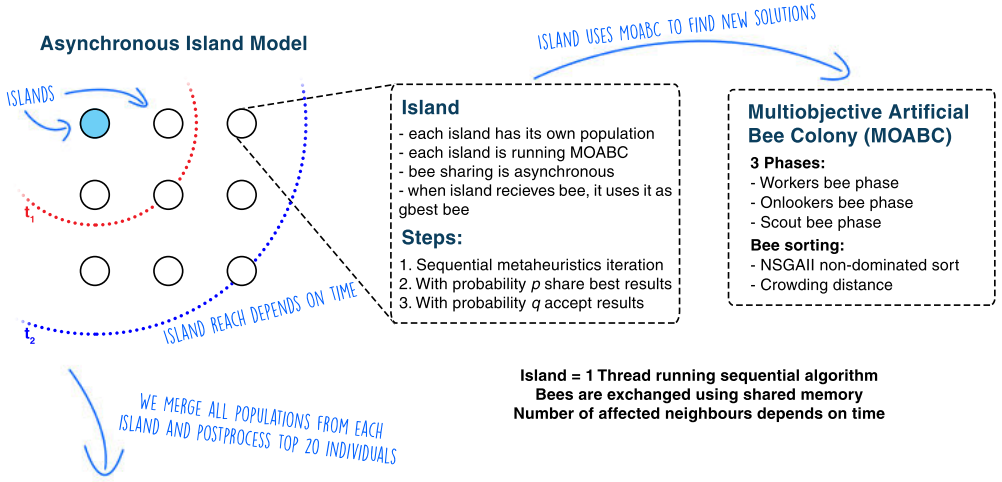
Bee Evaluation



Solution (bee position) contains information about motif position in each input sequence and its length. From these candidate motifs, consensus motif is created using most dominant nucleotide for each position. Only motifs that have at least 50% nucleotides same as consensus are considered. It is these motifs from which position frequency matrix (PFM) is computed.

Using PFM we can compute similarity, entropy and complexity. Support is number of motifs that passed consensus filter. Length is given directly by bee position.

Asynchronous Island Model



Island

- each island has its own population
- each island is running MOABC
- bee sharing is asynchronous
- when island recieves bee, it uses it as gbest bee

Steps:

1. Sequential metaheuristics iteration
2. With probability *p* share best results
3. With probability *q* accept results

Multiobjective Artificial Bee Colony (MOABC)

3 Phases:
- Workers bee phase
- Onlookers bee phase
- Scout bee phase

Bee sorting:
- NSGAII non-dominated sort
- Crowding distance

Island = 1 Thread running sequential algorithm
Bees are exchanged using shared memory
Number of affected neighbours depends on time

Postprocessing

For each solution we consider candidate motif with the smallest hamming distance to consensus motif as a result. Subsequently we score these motifs and output only *n* best of them.

Motif Score

Given motif length we compute n-grams from input sequences. N-grams are scored by hamming distance to motif. Scored n-grams are then filtered by maximum hamming distance that is derived from motif length. N-grams are then split into buckets by their score. Bucket B_j contains n-grams with hamming distance *i*. Motif score is computed as

$$\sum_{i=0}^{|B|-1} \frac{|B_i|}{i+1}$$

Where $|B|$ is number of buckets and $|B_i|$ is number of motifs in given bucket.

Usage of gbest bee

During MOABC we use bee step equation extended by gbest. Each bee is being pulled to gbest bee. Gbest is recieved from neighbouring island during migration.

$$v_{ij} = x_{ij} + \phi_{ij}(x_{ij} - x_{kj}) + \Psi_{ij}(y_j - x_{ij})$$

Results

	Our Solution					Weeder					MEME3				
	TP	FP	FN	Sn	PPV	TP	FP	FN	Sn	PPV	TP	FP	FN	Sn	PPV
Fly	23	256	94	0,196	0,082	1	28	50	0,019	0,03	3	64	48	0,058	0,04
Human	94	1135	800	0,105	0,076	32	92	266	0,107	0,258	19	222	279	0,063	0,07
Mouse	67	482	208	0,243	0,122	12	55	86	0,122	0,179	19	76	79	0,193	0,2
Yeast	40	373	182	0,18	0,096	39	32	36	0,52	0,549	24	55	51	0,32	0,303
Total	224	2246	1284	0,148	0,09	84	207	438	0,16	0,288	65	417	457	0,124	0,124