SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1972

Model za optimizaciju razmještaja raspodijeljene aplikacije

Rudolf Lovrenčić

Zagreb, lipanj 2019.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 8. ožujka 2019.

DIPLOMSKI ZADATAK br. 1972

Pristupnik: **Rudolf Lovrenčić (0036484937)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Model za optimizaciju razmještaja raspodijeljene aplikacije**

Opis zadatka:

Proučiti područje razvoja i rada raspodijeljenih aplikacija, te mogućnosti postavljanja aplikacije strukturirane u obliku skupa podatkovnih i programskih elemenata uz korištenje više neovisnih računalnih oblaka. Na osnovi zadane strukture, definirati model ocjene performansi raspodijeljene aplikacije s obzirom na način raspodjele pojedinih podatkovnih i programskih elemenata na skup računalnih oblaka. Predložiti i ostvariti prikladne algoritme za optimizaciju svojstava raspodijeljene aplikacije. Metode optimiranja trebaju obratiti posebnu pažnju na ograničenja raspodjele po pitanju sigurnosnih zahtjeva korisnika aplikacije. Predložiti model uključivanja više korisnički definiranih kriterija u postupak optimizacije. Ispitati učinkovitost ostvarenih postupaka s obzirom na ispitne primjere raspodijeljenih aplikacija. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Zadatak uručen pristupniku: 15. ožujka 2019.
Rok za predaju rada: 28. lipnja 2019.

Mentor:
Prof. dr. sc. Domagoj Jakobović

Djelovođa:
Izv. prof. dr. sc. Tomislav Hrkać

Predsjednik odbora za diplomski rad profila:
Doc. dr. sc. Marko Čupić

Hvala dobrim prijateljima, strpljivoj obitelji i prekrasnoj Dani.

Sadržaj

1. Uvod

Računarstvo u oblaku (engl. *cloud computing*) je popularna paradigma računalne tehnologije kojom se pružaju ekonomske i poslovne prednosti. Računalni resursi za pohranu i obradu podataka nude se kao usluga putem Interneta kako bi se korisnik računarstva u oblaku rasteretio upravljanja i održavanja tih resursa. Dodatna prednost je fleksibilnost – korisnik lako može promijeniti količinu računalnih resursa koju zakupljuje. Može smanjiti količinu resursa kako bi smanjio cijenu usluge ili ju povećati ako mu je to potrebno za postizanje određenog poslovnog cilja.

Usprkos prednostima, velik broj organizacija ne želi koristiti računarstvo u oblaku zbog nedostatka povjerenja u pružatelje usluge. Manjak povjerenja je opravdan. Brojne povrede sigurnosti podataka u proteklih dvadesetak godina uzrokovale su veliku financijsku štetu [1]. Računarstvo u oblaku ima i dodatne sigurnosne rizike koji su posljedica velike fleksibilnosti usluge – kvalitetna i kontinuirana revizija sigurnosti je skupa ili neizvediva.

Istraživanja iz područja računarstva u oblaku i općenito raspodijeljenih aplikacija koncentrirana su na poboljšanje performansi i učinkovitosti [2, 3, 4]. Poboljšanja sigurnosti oblaka ponuđena su pomoću specifičnih tehnologija [5, 6, 7] ili tako da se kritični podaci kriptiraju i pohrane na poslužitelje kojima korisnik vjeruje (engl. *hybrid cloud*). Nedostatak enkripcije je nemogućnost obrade podataka u oblaku i činjenica da korisnik mora brinuti o pohrani ključeva.

Kako korisnik često na raspolaganju ima više pružatelja usluga računarstva u oblaku, u ovom se radu istražuju mogućnosti poboljšanja sigurnosti raspoređivanjem podataka i aplikacije na više pružatelja usluga. Na taj se način nastoji smanjiti sigurnosni rizik jer korisnik više ne treba bezuvjetno vjerovati jednom pružatelju usluga, a istovremeno je očuvana mogućnost obrade podataka u oblaku. Ovakav pristup korištenja računarstva u oblaku prigodno je nazvan *multi-cloud*.

Poglavlje 2 postavlja pretpostavke te daje model aplikacije i podataka korišten za procjenu i optimizaciju sigurnosnog rizika. U poglavlju 3 se opisuje programska implementacija koja koristi model aplikacije i podataka te informacije o raspoloživim pružateljima usluga kako bi se minimizirao rizik korištenja računarstva u oblaku.

2. Model procjene rizika

Sistematična procjena rizika se obavlja na ograničenom modelu. Kvaliteta procjene uvelike ovisi o preciznosti modela. Detaljniji opis aplikacije i podataka rezultira boljom procjenom, no kako se rizik mora moći procijeniti za sve aplikacije, opis se ne smije oslanjati na informacije specifične za pojedinu aplikaciju. Model stoga nastoji što bolje zadovoljiti dva oprečna kriterija:

– **općenitost** (model vrijedi za sve aplikacije)
– **specifičnost** (model omogućava kvalitetu procjenu rizika)

Model se sastoji od tri cjeline. U prvoj se opisuju sigurnosne zavisnosti podataka međusobno i važnost fragmenata podataka zasebno. Druga cjelina opisuje kako aplikacija koristi podatke. Informacije o okruženju dane su u trećem dijelu opisa – u toj cjelini se definiraju raspoloživi pružatelji usluga.

2.1. Opis podataka i sigurnosnih zavisnosti

Model pretpostavlja unaprijed podijeljene fragmente podataka kojima aplikacija pristupa. Fragmenti podataka mogu, primjerice, biti stupci vertikalno fragmentirane tablice (slika 2.1).

IME	PREZIME	ISPLATA
Marko	Horvat	300
Sara	Petek	750
Dino	Milas	600

(a) Tablica prije podjele na fragmente.

F_1		F_2
IME	PREZIME	ISPLATA
Marko	Horvat	300
Sara	Petek	750
Dino	Milas	600

(b) Podaci podijeljeni u dva fragmenta.

Slika 2.1: Vertikalna fragmentacija tablice na fragmente F_1 i F_2.

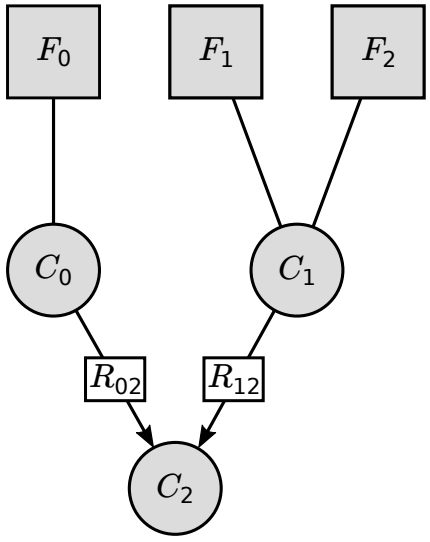
Glavna motivacija za fragmentaciju podataka je činjenica da je zbroj informacija fragmenta manji od informacije svih fragmenata zajedno. Primjerice, informacija o iznosu isplate napadaču puno manje koristi nego informacija o iznosu isplate i osobi kojoj je taj iznos isplaćen. Ako se fragmenti smještaju na istog pružatelja usluge, potrebno je puno povjerenja u tog pružatelja. Ukoliko se fragmenti smještaju na različite pružatelje usluge, korisnik treba imati manju količinu povjerenja u svakog pružatelja. Zbroj tih povjerenja je manji od povjerenja u slučaju bez fragmentacije.

U skupu svih fragmenata $\{F_1, \ldots, F_N\}$ je potrebno pronaći sve kritično zavisne podskupove K_i. Kritično zavisni skup fragmenata $K_i = \{F_1, \ldots, F_n\}$ definira svojstvo da je zbroj informacija fragmenata zasebno manji od informacije cijelog skupa, pa je stoga poželjno da se cijeli skup fragmenata ne nalazi kod istog pružatelja usluge. Drugim riječima, kritično zavisni skup definira sigurnosno ograničenje kojim se izražava želja da se barem jedan fragment tog skupa ne nalazi na mjestu ostalih fragmenata u skupu. Ograničenje $K_1 = \{F_1, F_2\}$ dano uz primjer na slici 2.1 izražava želju da se podaci o isplati ne pohranjuju kod istog pružatelja usluge kao i naziv osobe.

Prepoznavanje svih kritično zavisnih skupova može biti zahtjevno jer je mogući broj podskupova 2^N, a kako proces zahtjeva poznavanje semantike podataka, nemoguće ga je potpuno automatizirati. Olakšavanje pronalaska zavisnosti u velikim bazama podataka moguće je korištenjem entropije [8] i sličnih mjera korelacije podataka. Nedostatak empirijskih mjera je određivanje semantike na temelju postojećih podataka, pa je preduvjet za njihovu primjenu velik skup reprezentativnih podataka.

2.2. Opis aplikacije

Raspodijeljena aplikacija se također raspoređuje po pružateljima usluga prilikom postavljanja u oblak. Svaka aplikacijska komponenta C_i može pristupati podatkovnim fragmentima, primati i slati podatke drugim komponentama. Slika 2.2 prikazuje aplikaciju od tri komponente, tri fragmenta podataka te dva međurezultata koji se razmjenjuju između komponenti.



Slika 2.2: Prikaz tri podatkovna fragmenta, tri aplikacijske komponente, pristup podacima i komunikacija između komponenti. Komponenta C_0 obrađuje samo podatke iz fragmenta F_0 te nakon obrade šalje rezultat R_{02} komponenti C_2. Komponenta C_1 pristupa dvama fragmentima i svoj rezultat također šalje komponenti C_2.

Međurezultati predstavljaju resurse koje je potrebno zaštititi na jednak način kao i fragmente podataka. Primjerice, ako komponenta aplikacije radi jednostavno prebrojavanje zapisa u bazi podataka, to za napadača može biti korisna informacija. Iz tog razloga model omogućava definiranje sigurnosnih zavisnosti između međurezultata i fragmenata podataka.

2.2.1. Doseg resursa

Resursi koje procjena sigurnosnog rizika uzima u obzir su fragmenti podataka i međurezultati koje aplikacijske komponente razmjenjuju. Ponašanje međurezultata je drugačije od fragmenata jer međurezultat nije nužno smješten na jednom mjestu – on je poznat na svojem izvorištu i na svojem odredištu. Doseg resursa definira se kao skup fragmenata i komponenti na kojima je taj resurs poznat.

Iz definicije dosega resursa slijedi da je doseg fragmenta skup koji sadrži taj fragment i sve aplikacijske komponente koje tom fragmentu pristupaju. Doseg međurezultata je skup od dvije komponente: izvorišta i odredišta međurezultata. Za primjer na slici 2.2, doseg fragmenta F_2 je skup $D(F_2) = \{F_2, C_1\}$, a doseg međurezultata R_{02} je $D(R_{02}) = \{C_0, C_2\}$.

Pojam dosega pomaže u provjeri je li neko sigurnosno ograničenje zadovoljeno nekim rasporedom komponenti i fragmenata. Za prethodno napisane dosege $D(F_2)$ i $D(R_{02})$, ograničenje $\{F_2, R_{02}\}$ je zadovoljeno ako se niti jedan element skupa $D(F_2)$ ne nalazi na istom mjestu kao i bilo koji element skupa $D(R_{02})$. Drugim riječima, presjek skupa mjesta na kojima je poznat prvi resurs i skupa mjesta na kojima je poznat drugi resurs mora biti prazan skup. Za ograničenja s više resursa postupak je analogan: odrede se mjesta na kojima je dostupan prvi resurs te se taj skup uzastopno presijeca s mjestima dostupnosti ostalih resursa. Ako je konačni skup prazan, ograničenje je zadovoljeno.

Doseg resursa omogućava i laganu provjeru je li neko ograničenje moguće zadovoljiti u idealnom slučaju[1]. Kako bi ograničenje bilo zadovoljivo, presjek dosega resursa u ograničenju mora biti prazan skup (nužan uvjet zadovoljavanja sigurnosnog ograničenja).

2.3. Opis okruženja

Pružatelj usluga ili poslužitelj nekog pružatelja predstavlja mjesto postavljanja podatkovnih fragmenata i aplikacijskih komponenti. Model je neovisan o definiciji mjesta postavljanja. Popis raspoloživih mjesta je potreban kako bi se odredio rizik potencijalnog rasporeda. Svako raspoloživo mjesto S_i ima određenu razinu povjerenja. Važniji resursi smještaju se na sigurnija mjesta. Jednako tako, ograničenja koja se ne mogu zadovoljiti nastoje se prekršiti na sigurnijim mjestima.

[1] dostupan je proizvoljan broj mjesta za postavljanje fragmenata i komponenti

2.4. Kriteriji procjene rizika

Meta-heuristički optimizacijski algoritmi zahtijevaju mogućnost ocjene dobrote rješenja (engl. *evaluation*). Za proizvoljnu raspodjelu podatkovnih fragmenata i aplikacijskih komponenti po raspoloživim mjestima potrebno je procijeniti sigurnosni rizik.

2.4.1. Prekršena sigurnosna ograničenja

Očiti kriterij procjene rizika kojeg valja uzeti u obzir je broj prekršenih sigurnosnih ograničenja. Ako je neko sigurnosno ograničenje prekršeno, rizik predloženog rasporeda fragmenata i komponenti se uvećava za predodređen iznos.

U obzir se uzima i povjerenje u mjesto na kojem je ograničenje prekršeno. Veća razina povjerenja u mjesto treba rezultirati manjom kaznom. Rizik r kršenja ograničenja može se izračunati koristeći jednadžbu 2.1, gdje p predstavlja kaznu za kršenje sigurnosnog ograničenja, a t povjerenje u mjesto na kojem je ograničenje prekršeno.

$$r = \frac{p}{t} \tag{2.1}$$

Ukoliko je sigurnosno ograničenje prekršeno na više mjesta, u izračunu se koristi mjesto najmanje razine povjerenja. Kazna p može biti definirana za svako ograničenje zasebno ili je moguće koristi jednaku kaznu za sva sigurnosna ograničenja.

2.4.2. Važnost resursa zasebno

Svaki resurs nosi određenu količinu informacije zasebno, bez doticaja s drugim resursima. Ovaj kriterij kažnjava dostupnost važnijih resursa na mjesta s manjim razinama povjerenja. Rizik dostupnosti resursa važnosti v na mjestu razine povjerenja t moguće je odrediti jednadžbom 2.2.

$$r = \frac{v}{t} \tag{2.2}$$

Mjesta na kojima je resurs poznat određuju se korištenjem dosega resursa: za sve elemente dosega resursa odredi se mjesto smještanja. Dobiveni skup određuje mjesta na kojima je resurs dostupan.

2.4.3. Zajednički smještaj podataka i aplikacije

Smještanje podataka i aplikacije na isti poslužitelj nije preporučljivo zbog sigurnosnih rizika. Korištenje ove mjere rizika se preporučuje u slučaju kada mjesta raspoređivanja predstavljaju poslužitelje, a ne, primjerice, pružatelje usluga računarstva u oblaku. Svaki zajednički smještaj fragmenta podataka i aplikacijske komponente kažnjava se fiksnim iznosom.

3. Implementacija modela

Programska implementacija služi kao omotač oko nekoliko biblioteka za optimizaciju. Na je taj način omogućeno unificirano zadavanje modela i parametara algoritama, a velik dio programskog koda može se implementirati na razini omotača te se zatim koristiti u bibliotekama (npr. procjena rizika ponuđenog rasporeda fragmenata i komponenti).

Konfiguracijskim datotekama se zadaje opis podataka i aplikacije, odabire se optimizacijski algoritam i parametri algoritma. Ukoliko je konfiguracija ispravna, pokreće se odabrani algoritam koji nastoji pronaći najbolje rješenje – onaj raspored podatkovnih fragmenata i aplikacijskih komponenti koji po zadanim kriterijima ima najmanji sigurnosni rizik. Moguće je koristiti i višekriterijsku optimizaciju čiji je rezultat skup nedominiranih rješenja – rješenja za koje ne postoji rješenje bolje po svim kriterijima.

Implementacija je prigodno nazvana Lopt (engl. *Layout optimizer*) i pokreće se zadavanjem imena dviju konfiguracijskih datoteka preko naredbenog retka:

```
Lopt <app_config_file.xml> <alg_config_file.xml>
```

Prva datoteka opisuje aplikaciju i podatke, a u drugoj se odabire algoritam i parametri. Poglavlje 3.4 detaljno opisuje sadržaj ulaznih datoteka.

3.1. Tehnologija i biblioteke

Programski kod napisan je u jeziku C++, a konfiguracijske datoteke se očekuju u XML formatu [9]. C++ je popularan izbor za implementaciju optimizacijskih algoritama zbog brzine izvođenja i mogućnosti apstrakcije. XML je izabran zbog standardiziranosti, čitljivosti i mogućnosti dodavanja komentara. Za učitavanje XML datoteka se koristi biblioteka RapidXML.

3.1.1. ECF

Programski okvir ECF (engl. *Evolutionary Computation Framework*) [10] pruža potporu za mnoštvo algoritama iz područja evolucijskog računarstva. Većina algoritama ne ovisi o prikazu rješenja, pa se mogu koristiti za sve vrste optimizacijskih problema.

Okvir se koristi nasljeđivanjem razreda `Genotype` kako bi se definirao prikaz rješenja. Na temelju tog prikaza, potrebno je definirati operatore mutacije, križanja i evaluacije također pomoću nasljeđivanja. U ulaznoj datoteci okvira je potrebno navesti naziv prikaza rješenja (genotipa) koji se koristi. Konfiguracijska datoteka okvira ECF je skrivena od korisnika Lopt programa – ta se datoteka automatski generira na temelju opcija zadanih u datoteci za konfiguraciju algoritma. Na taj je način omogućeno lakše korištenje programa, no manjim izmjenama programskog koda moguće isključiti automatsko generiranje kako bi se omogućila potpuna sloboda konfiguracije programskog okvira.

3.1.2. OpenGA

OpenGA je C++ programska biblioteka za implementaciju genetskih algoritama [11]. Omogućava jednostavno definiranje tipa rješenja i tipa ocjene dobrote rješenja preko argumenta predloška (engl. *template argument*). Funkcije za inicijalizaciju, evaluaciju, mutaciju i križanje dobivaju prototip u skladu s danim tipovima. Od korisnika biblioteke se zahtjeva implementacija tih funkcija i registriranje pomoću pokazivača na funkcije.

```
EA::Genetic<Solution, Risk> algorithm;
void init_genes(Solution& s);
bool eval_solution(const Solution& sol, Risk& risk);
Solution mutate(const Solution& sol);
Solution crossover(const Solution& s1, const Solution& s2);
```

Isječak 3.1: Definicija tipa genetskog algoritma u biblioteci OpenGA.

Biblioteka se isporučuje kao jedna datoteka zaglavlja (engl. *header-only library*) što dodatno pospješuje jednostavnost korištenja. Omogućena je paralelna evaluacija jedinki koristeći standardnu C++ potporu za višedretvenost: `std::thread`. Implementirana je potpora za jednokriterijske i višekriterijske probleme.

Nedostatak biblioteke OpenGA je manjak kontrole nad algoritmom – selekcija je potpuno skrivena od korisnika. Ako težinska selekcija nije dovoljna, korisnik mora mijenjati izvorni kod same biblioteke.

3.1.3. Gnuplot

Za praćenje napretka višekriterijskih algoritama koristi se Gnuplot [12]. Omogućava interaktivnu vizualizaciju podataka i detaljno stiliziranje. Gnuplot se koristi zadavanjem naredbi preko naredbenog retka, pa je u sklopu Lopt programa izrađena apstrakcija koja otvara cjevovod operacijskog sustava prema Gnuplot procesu. Preko cjevovoda se šalju naredbe u tekstualnom formatu. Trenutna implmentacija apstrakcije radi samo na Unix operacijskim sustavima, no prilagodba za ostale je trivijalna.

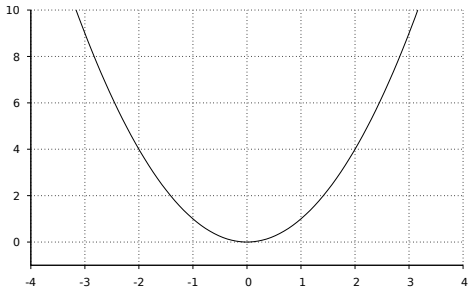
```
class Gnuplot {
private:
   FILE* pipe;
public:
   Gnuplot() { pipe = popen("gnuplot -persist", "w"); }
   ~Gnuplot() { if (pipe) pclose(pipe); }

   Gnuplot(const Gnuplot& other) = delete;
   void operator=(const Gnuplot& other) = delete;

   Gnuplot& operator<<(const std::string& s)
   {
      fputs(s.c_str(), pipe);
      return *this;
   }
   void flush() { fflush(pipe); }
};
```

Isječak 3.2: Apstrakcija cjevovoda prema Gnuplot procesu.

Cjevovod se otvara pomoću naredbe `popen`, te se automatski zatvara po isteku životnog vijeka objekta razreda `Gnuplot`. Uklanjanjem konstruktora za kopiranje i operatora pridruživanja za kopiranje sprječava se otvaranje više cjevovoda iz programa Lopt prema Gnuplot procesu. Naredbe se šalju korištenjem operatora << (isječak 3.3).

```
Gnuplot gnuplot;
gnuplot << "plot x*x";
gnuplot.flush();
```

Isječak 3.3: Upotreba razreda `Gnuplot`.



Slika 3.1: Rezultat izvođenja isječka 3.3.

3.2. Algoritmi

Program Lopt sadrži četiri algoritma. Jednokriterijski genetski algoritam ostvaren je na dva načina: pomoću okvira ECF i koristeći biblioteku OpenGA. Motivacija za dvije implementacije je usporedba performansi algoritama i činjenica da iako OpenGA ima manje mogućnosti, nema zahtjeva za ostalim knjižnicama (ECF zahtjeva Boost [13] i MPI [14] ukoliko je potrebna paralelizacija).

Višekriterijska optimizacija ostvarena je algoritmom NSGA-III u sklopu biblioteke

OpenGA. Danas se taj algoritam smatra vodeći u rješavanju višekriterijskih optimizacijskih problema. Drugi višekriterijski algoritam je PPES (engl. *Predator-Prey Evolution Strategy*). Osnovna inačica algoritma implementirana je bez korištenja biblioteka. Motivacija za ovaj algoritam je jednostavnost implementacije i potpuno drugačiji pristup od algoritma NSGA-III.

3.2.1. Genetski algoritam

Genetski algoritam je stohastička metoda optimizacije osnovana na principu prirodnog evolucijskog procesa. Evolucija se imitira operatorima selekcije, križanja i mutacije. Genetski algoritam je primjeren za probleme koje nije moguće riješiti egzaktnim postupcima, no za razliku od tih postupaka, genetski algoritam ne garantira pronalazak najboljeg mogućeg rješenja.

Evolucijski proces se temelji na generacijama jedinki – svaka jedinka predstavlja moguće rješenje problema. Bolje jedinke imaju veću vjerojatnost preživljavanja i reprodukcije. Na taj način se tijekom puno generacija dobivaju rješenja koja sadrže dobra svojstva predaka. Mutacija u genetskom algoritmu izaziva nasumične promjene rješenja u nadi da se pronađu dobra svojstva koja bi se inače teško (ili nemoguće) postigla isključivo križanjem.

Pseudokod 3.1: Genetski algoritam.

```
generiraj početnu populaciju P(0);   i = 1;
dok nije zadovoljen uvjet zaustavljanja radi
    evaluiraj P(i - 1);
    odaberi P(i) iz P(i - 1);
    križaj P(i);
    mutiraj P(i);
kraj
vrati najbolju jedinku iz populacije;
```

Odabir jedinki za križanje se u OpenGA implementaciji radi proporcionalno dobroti jedinke. Ukoliko je rizik nekog rasporeda podatkovnih fragmenata i aplikacijskih komponenata manji, vjerojatnost odabira tog rješenja je veća. Algoritam je generacijski što znači da odabir zahtjeva dovoljno memorije za pohranu stare i nove populacije.

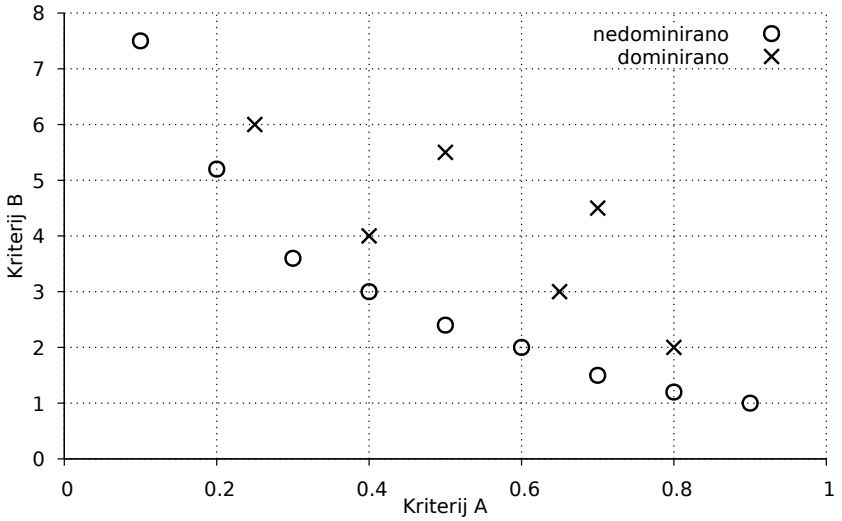
Genetski algoritam implementiran pomoću okvira ECF koristi troturnirsku selekciju u kojoj se nasumično odabiru tri jedinke iz populacije. Križanjem dviju boljih jedinki dobiva se dijete koje zamjenjuje treću odabranu jedinku. Na taj je način zauzeta memorija samo za jednu populaciju (engl. *Steady-State*).

Oba algoritma imaju poželjno svojstvo elitizma – najbolja jedinka iz trenutne populacije nepromijenjena prelazi u sljedeću.

3.2.2. NSGA-III

Algoritam NSGA-III (engl. *Non-dominated Sorting Genetic Algorithm*) je prilagođeni genetski algoritam za rješavanje višekriterijskih optimizacijskih problema [15]. Od običnog genetskog algoritma se razlikuje samo u koraku selekcije. Algoritam je izmjena elitističke verzije originalnog algoritma NSGA [16] – promijenjen je način čuvanja raznolikosti rješenja.

Cilj višekriterijskih algoritama nije odrediti najbolje rješenje jer takvo najčešće ne postoji. Zadatak je pronaći najbolji skup nedominiranih rješenja. Nedominirano rješenje je ono za koje ne postoji drugo rješenje koje je jednako po svim kriterijima, a barem po jednom i bolje. Ako takvo postoji, drugo rješenje dominira nad prvim.



Slika 3.2: Prikaz dominiranih i nedominiranih rješenja za minimizaciju dva kriterija.

Višekriterijska optimizacija je prikladna kada korisnik nije siguran koja je važnost svakog od kriterija. Zadaća algoritma je ponuditi što više nedominiranih rješenja različitih važnosti kriterija. Slika 3.2 prikazuje skup dominiranih i nedominiranih rješenja za problem minimizacije dvije funkcije cilja. Interpretacija nekih rješenja:

$(0.1, 7.5)$ – dobro po prvom kriteriju, loše po drugom.

$(0.4, 4.0)$ – osrednje po oba kriterija.

$(0.9, 1.0)$ – loše po prvom kriteriju, dobro po drugom.

Algoritam u svakoj iteraciji izvršava nedominirano sortiranje kako bi se dobile fronte rješenja. Rješenja $(0.5, 5.5)$ i $(0.7, 4.5)$ na slici 3.2 spadaju u treću frontu, ostala dominirana rješenja u drugu, a u prvoj se fronti nalaze nedominirana rješenja. Složenost ovog postupka je $O(MN^2)$ gdje je M broj kriterija, a N veličina populacije.

Operator selekcije iz trenutne populacije roditelja stvara novu populaciju djece pomoću operatora križanja. Unija tih dviju populacija se nedominirano sortira. Fronte jedinki se redom kopiraju u novu populaciju. Ukoliko cijela fronta ne stane u novu

populaciju, algoritam bira jedinke koje osiguravaju maksimalnu raznolikost kako bi pokrivenost fronte bila najveća. Ovakav postupak selekcije osigurava elitizam.

3.2.3. PPES

Algoritam PPES pristupa višekriterijskom optimizacijskom problemu imitirajući ponašanje grabežljivca i plijena [17]. Lopt program implementira originalnu inačicu algoritma iz 1998. godine, no od tada su predložene izmjene koja donose znatna poboljšanja [18, 19]. Osnovna inačica nema svojstvo elitizma i generalno sporije konvergira od modernih algoritama za višekriterijsku optimizaciju. Jednostavnost algoritma omogućava brzu implementaciju koja služi kao osnova za usporedbu.

Plijen u algoritmu predstavlja moguće rješenje, a grabežljivac jedan od ciljeva optimizacije. Algoritam započinje inicijalizacijom – početna populacija se nasumično raspoređuje svijet koji se obično predstavlja dvodimenzionalnim poljem. Grabežljivci se također nasumično raspoređuju. Veličina svijeta i broj grabežljivaca su hiperparametri algoritma.

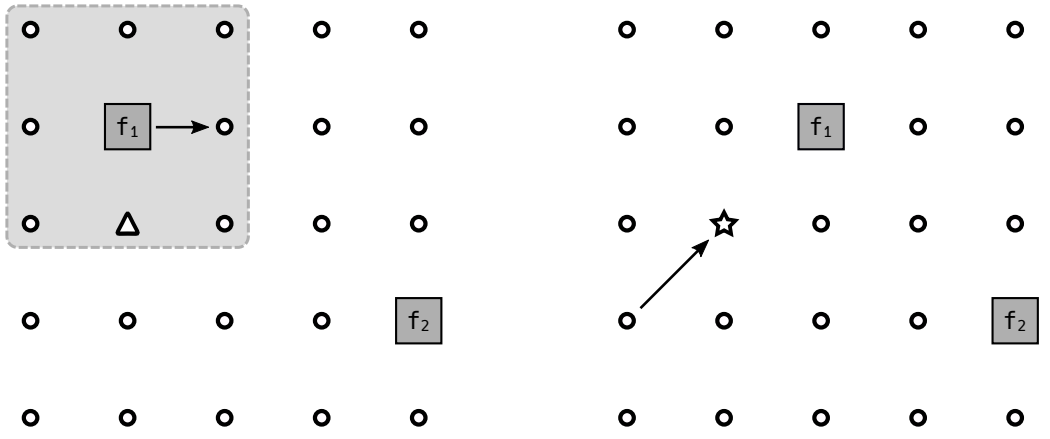
Pseudokod 3.2: Algoritam PPES.

```
inicijaliziraj svijet;
dok nije zadovoljen uvjet zaustavljanja radi
    za svakog grabežljivca radi
        evaluiraj okolne plijnove po svojoj funkciji cilja;
        pojedi najgori plijen;
        odaberi nasumični plijen oko pojedenog;
        mutiraj odabrani plijen;
        muitirani plijen stavi na mjesto pojedenog;
        nasumično pomakni grabežljivca;
    kraj
kraj
vrati nedominirana rješenja;
```

U glavnoj petlji algoritma, svaki grabežljivac evaluira plijen po svojoj funkciji cilja, jede plijen i nasumično se pomiče na jedno od okolnih polja (slika 3.3). Pojedeni plijen se zamjenjuje mutacijom jednog od okolnih plijenova.

Kako bi osigurali da grabežljivci jednako često posjećuju sva polja, svijet se najčešće predstavlja kao dvodimenzionalni torus. Iako u prikazu na slici 3.3 postoje rubovi svijeta, torus omogućava prijelaz s jednog ruba polja na drugi.

Testiranja su pokazala da uvođenje križanja bez elitizma ubrzava konvergenciju, ali iznimno šteti raznolikosti rješenja [18], pa je križanje izostavljeno. Dokazano je da algoritam konvergira koristeći samo operator mutacije [17].



(a) Grabežljivac jede najgori plijen u svojem susjedstvu (oznaka △) i kreće se udesno.

(b) Donji lijevi susjed najgoreg plijena se mutira i zamjenjuje najgori plijen (oznaka ⋆).

Slika 3.3: Rad algoritma PPES. Kvadrat simbolizira grabežljivca, a krug plijen.

3.3. Prikaz rješenja

Performanse optimizacijskih algoritama uvelike ovise o reprezentaciji rješenja. Cilj prikaza je izbjeći što više rješenja koja nisu prihvatljiva[1], a istovremeno omogućiti što jednostavniju i efikasniju implementaciju operatora evaluacije, križanja i mutacije.

```
class Solution {
private:
   std::unordered_map<const Fragment*, const Server*> fm;
   std::unordered_map<const Component*, const Server*> cm;
public:
   template<typename T>
   void set_server(const T& assignable, const Server& server);

   template<typename T>
   const Server& get_server(const T& assignable) const;
};
```

Isječak 3.4: Implementacija rješenja.

Rješenje je mapiranje fragmenata podataka i aplikacijskih komponenti na mjesta postavljanja. Iako se u isječku 3.4 mjesto postavljanja naziva poslužitelj (engl. *Server*), ponovno se napominje da to može predstavljati pružatelja usluga ili ostala mjesta za pohranu podataka i izvršavanje aplikacijskih dijelova.

Korištenje razreda `std::unordered_map` za mapiranje osigurava pretragu, brisanje i dodavanje s konstantnim vremenom u prosjeku. Postavljanje i dohvaćanje

[1] rješenja koja se mogu prikazati, ali ne zadovoljavaju ograničenja problema.

mjesta fragmenta ili komponente omogućeno je preko funkcija `set_server` i `get_server`. Informacija radi li se o fragmentu podataka ili komponenti aplikacije šalje se kao argument predloška.

Objekti na koje pokazuju pokazivači unutar mapiranja stvaraju se prilikom učitavanja konfiguracije programa. Korištenjem pokazivača spriječena je potreba za nepotrebnim kopijama objekata u memoriji.

3.3.1. Evaluacija

Implementirani operatori evaluacije omogućavaju procjenu rizika rješenja po kriterijima definiranim u poglavlju 2.4. Svi evaluatori su izvedeni iz baznog razreda `Evaluator` (isječak 3.5). Od svih operatora evaluacije se očekuje da na temelju opisa aplikacije, okruženja i sigurnosnih zavisnosti mogu procijeniti rizik danog rješenja. Ponašanje se osigurava nadjačavanjem funkcije `evaluate`.

```
class Evaluator {
protected:
   const ApplicationConfiguration& app_config;
public:
   Evaluator(const ApplicationConfiguration& app_config);
   virtual Risk evaluate(const Solution& solution) = 0;
};
```

Isječak 3.5: Bazni razred za sve operatore evaluacije.

Tri navedena načina procjene rizika iz poglavlja 2.4 se direktno preslikavaju u tekst programa. Svaki resurs omogućava izračunavanje svog dosega, pa je evaluacija rješenja dodatno olakšana. Razred `ApplicationConfiguration` sadrži podatke o aplikaciji i okruženju opisane u poglavlju 2.2.

3.3.2. Križanje

Jednostavan operator križanja implementiran je na način da se mjesto postavljana određenog fragmenta u djetetu dobiva iz nasumično odabranog roditelja. Na jednak način se određuje mjesto postavljanja aplikacijske komponente.

```
for (const Fragment& f : app_config.fragments) {
   const Server& s = Random::boolean() ?
         parent1.get_server(f) : parent2.get_server(f);
   child.set_server(f, s);
}
```

Isječak 3.6: Odlučivanje mjesta podatkovnog fragmenta.

Dio funkcije križanja koji određuje mjesto smještanja za svaki podatkovni fragment prikazan je isječkom 3.6. Roditelji i dijete su objekti razreda `Solution`.

3.3.3. Mutacija

Lopt program nudi dva operatora mutacije:

- stavljanje jednog fragmenta ili jedne komponente na nasumično mjesto
- generiranje potpuno novog nasumičnog rješenja

Svi implementirani algoritmi koriste oba operatora – nasumično se izabire jedan operator prije primjene mutacije.

```
void mutate(Solution& s)
{
   const Server& server = Random::server(app_config);
   bool reassign_fragment = Random::boolean();
   if (reassign_fragment) {
      const Fragment& fragment = Random::fragment(app_config);
      s.set_server(fragment, server);
   } else {
      const Component& component = Random::component(app_config);
      s.set_server(component, server);
   }
}
```

Isječak 3.7: Mutacija – stavljanje fragmenta ili komponente na nasumično mjesto.

3.4. Konfiguracija

Lopt program se konfigurira preko dvije konfiguracijske datoteke. Jedna datoteka opisuje fragmentaciju i važnost podataka, pristup podacima iz aplikacije, razmjenu međurezultata između aplikacijskih komponenti, sigurnosne zavisnosti te raspoloživa mjesta postavljanja fragmenata i komponenti. Druga datoteka služi za odabir i konfiguraciju algoritma optimizacije sigurnosnog rizika.

3.4.1. Opis aplikacije

Datoteka za opis aplikacije podijeljena je u četiri sekcije:

- opis resursa za koje se radi procjena rizika
- opis ulaza i izlaza aplikacijskih komponenti
- opis okruženja (raspoloživa mjesta postavljanja fragmenata i komponenti)
- popis sigurnosnih zavisnosti resursa

Resursi se opisuju kao djeca XML čvora `resources`. Za dodavanje fragmenata i međurezultata koriste se čvorovi `fragment` i `result`. Obavezan atribut tih čvorova je `ID` koji mora biti jedinstven niz znakova. Identifikator služi za referenciranje resursa

u nastavku konfiguracije i u rezultatima algoritma. Svaki resurs mora imati i čvor dijete u kojem se navodi važnost resursa kao prirodan broj. Isječak 3.8 prikazuje definiciju dva resursa: fragment važnosti 100 i međurezultat važnosti 50.

Aplikacijske komponente se opisuju na analogan način kao djeca čvora `application`. Svaka komponenta navodi resurse koje koristi pomoću čvora `input` i rezultate koje stvara koristeći čvor `output`. Isječkom 3.9 definiraju se dvije komponente. Prva pristupa fragmentu podataka i šalje svoj rezultat drugoj komponenti.

```
<resources>
   <fragment ID="F0">
      <importance>100</importance>
   </fragment>
   <result ID="R01">
      <importance>50</importance>
   </result>
</resources>
```

Isječak 3.8: Opis resursa.

```
<application>
   <component ID="C0">
      <input>F0</input>
      <output>R01</output>
   </component>
   <component ID="C1">
      <input>R01</input>
   </component>
</application>
```

Isječak 3.9: Opis aplikacije.

Popis raspoloživih mjesta za postavljanje fragmenata i komponenti zadaje se unutar `deployment` čvora. Mjesto se definira `server` čvorom sa identifikatorom. Čvor `trust` određuje povjerenje u mjesto – povjerenje se zadaje kao prirodan broj (isječak 3.10).

Sigurnosna ograničenja navode se kao djeca čvora `security` koristeći čvor `constraint`. Svako ograničenje ima svoj identifikator i skup resursa koji se nalaze u ograničenju. Sigurnosno ograničenje je prekršeno ukoliko su svi resursi u ograničenju dostupni na jednom mjestu. Isječak 3.11 definira sigurnosnu zavisnost između fragmenta i međurezultata definiranih u isječku 3.8. Dodatak A prikazuje cjelokupni opis aplikacije na slici 2.2.

```
<deployment>
   <server ID="S0">
      <trust>10</trust>
   </server>
</deployment>
```

Isječak 3.10: Opis mjesta.

```
<security>
   <constraint ID="0">
      <resource>F0</resource>
      <resource>R01</resource>
   </constraint>
</security>
```

Isječak 3.11: Definicija sigurnosne zavisnosti.

3.4.2. Parametri algoritma

Druga konfiguracijska datoteka koju Lopt program zahtjeva definira koji se algoritam koristi i sve parametre tog algoritma. Svi XML čvorovi u konfiguraciji algoritma su djeca čvora `algorithm`.

Čvor `type` definira algoritam koji se koristi za optimizaciju sigurnosnog rizika. Moguće vrijednosti su: `ECF::SOGA`, `OpenGA::SOGA`, `OpenGA::MOGA` i `PPES::PPES`. Pomoću prve dvije vrijednosti odabire se jednokriterijski genetski algoritam, a druge dvije omogućavaju višekriterijsku optimizaciju. Algoritmi su opisani u poglavlju 3.2.

Evaluatori se zadaju kao djeca čvora `evaluators`. Svaki procjenitelj rizika definiran je svojim imenom (atribut `name`) u čvoru `evaluator`. Implementirani evaluatori sigurnosnog rizika su navedeni u tablici 3.1, a poglavlju 2.4 ih detaljno opisuje.

Kazna za zajednički smještaj podataka i komponenti aplikacije te za kršenje sigurnosnih ograničenja je fiksno postavljena na 100 u tekstu programa. Fiksna kazna ne predstavlja problem jer je korištenjem koeficijenata omogućeno balansiranje između važnosti pojedinih kriterija kod jednokriterijskih algoritama. Višekriterijski algoritmi ne kombiniraju kriterije, pa ova odluka na njih ne utječe.

Naziv	Izračun rizika	Poglavlje
`Constraint`	Prekršena sigurnosna ograničenja.	2.4.1
`Single Resource`	Važnosti resursa zasebno.	2.4.2
`Resident`	Zajednički smještaj fragmenata i komponenti.	2.4.3

Tablica 3.1: Implementirani procjenitelji sigurnosnog rizika.

Jednokriterijski algoritmi

Oba implementirana jednokriterijska genetska algoritma definiraju četiri parametra:

1. `generation_max` – broj generacija
2. `population_size` – veličina populacije
3. `mutation_rate` – vjerojatnost mutacije
4. `log_frequency` – učestalost ispisa stanja algoritma

Prva tri parametra određuju ponašanje algoritma, a `log_frequency` određuje broj generacije između dva uzastopna ispisa stanja algoritma. Jedini uvjet zaustavljanja algoritma je maksimalan broj generacija i on se zadaje `generation_max` XML čvorom. Veličina populacije i vjerojatnost mutacije zadaju se koristeći preostala čvora. Mutacija i broj jedinki u populaciji uvelike utječu na rad algoritma jer se tim parametrima određuje omjer istraživanja novih rješenja i korištenja dobrog genetskog materijala poznatih rješenja.

Ukoliko je navedeno više od jednog evaluatora u konfiguraciji algoritma, svaki od njih mora imati čvor dijete `coefficient` kojim se definira realni broj. Koeficijent množi rizik dobiven iz pripadnog evaluatora kako bi korisnik mogao balansirati između važnosti pojedinih evaluatora i koristiti jednokriterijsku optimizaciju. Za razliku od višekriterijske optimizacije, korisnik prije pokretanja algoritma zadaje važnost svakog od kriterija. Višekriterijska optimizacija rasterećuje korisnika od te odluke na način da korisnik odlučuje o važnosti kriterija tek kada mu se ponude sva nedominirana rješenja.

Isječak 3.12 prikazuje konfiguraciju koja koristi jednokriterijski genetski algoritam iz knjižnice ECF. Korištenjem koeficijenata, zadano je da je kršenje sigurnosnih ograničenja dvostruko važnije od sigurnosti resursa zasebno.

```
<algorithm>
   <type>ECF::SOGA</type>
   <population_size>1000</population_size>
   <generation_max>500</generation_max>
   <log_frequency>50</log_frequency>
   <mutation_rate>0.2</mutation_rate>
   <evaluators>
      <evaluator name="Constraint">
         <coefficient>1.0</coefficient>
      </evaluator>
      <evaluator name="Single Resource">
         <coefficient>0.5</coefficient>
      </evaluator>
   </evaluators>
</algorithm>
```

Isječak 3.12: Konfiguracija jednokriterijskog algoritma.

Višekriterijski algoritmi

Višekriterijski algoritmi se međusobno razlikuju u parametrima. PPES algoritam nema veličinu populacije – veličina svijeta pomoću čvorova `world_width` i `world_height`. Veličina svijeta na jednak način utječe na pretraživanje kao i veličina populacije. Broj grabežljivaca po svakoj funkciji cilja se zadaje koristeći `predators_per_objective` XML ključ. Algoritam NSGA-III ima sve iste parametre kao i jednokriterijski genetski algoritam.

Parametri koji upravljaju praćenjem napretka algoritama PPES i NSGA-III su: `plot_frequency`, `pause_after_plot` i `plot_range`. Prvim se određuje broj generacija između uzastopnih iscrtavanja dobrote populacije po dva kriterija. Ukoliko je zastavica `pause_after_plot` postavljena na `true`, algoritam se zaustavlja prilikom

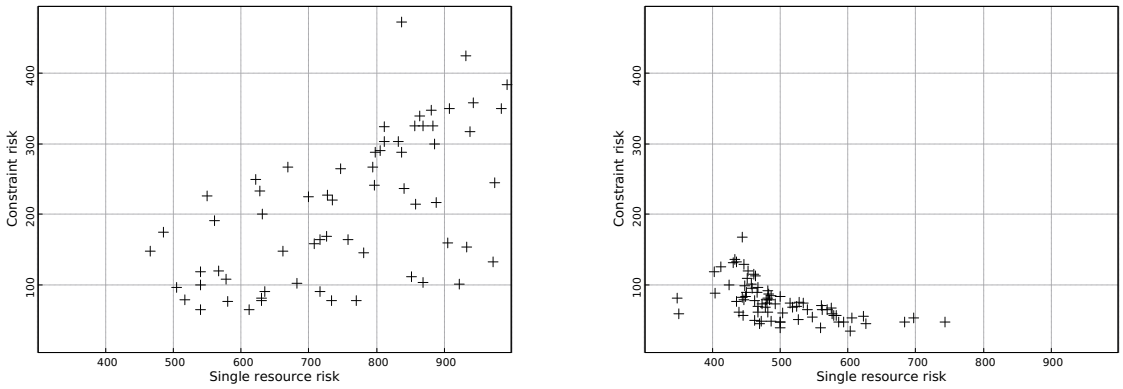
iscrtavanja i čeka dozvolu za nastavak od korisnika. Rad algoritma se korištenjem ove zastavice lakše proučava i demonstrira. Parametar `plot_range` s atributima `lower` i `upper` definira granice iscrtavanja. Parametar je dijete `evaluator` čvora.

Implementirani višekriterijski algoritmi imaju potporu za samo dva evaluatora kako bi iscrtavanje napretka u ravnini bilo moguće. Drugih razloga za ograničenje na dvije funkcije cilja nema, pa je proširenje na više trivijalno ako se onemogući prikaz napretka. U slučaju da je navedeno više evaluatora u konfiguracijskoj datoteci, koriste se samo prva dva uz ispis upozorenja.

```
<algorithm>
   <type>OpenGA::MOGA</type>
   <population_size>75</population_size>
   <generation_max>100</generation_max>
   <log_frequency>10</log_frequency>
   <plot_frequency>1</plot_frequency>
   <pause_after_plot>true</pause_after_plot>
   <mutation_rate>0.2</mutation_rate>
   <evaluators>
      <evaluator name="Single Resource">
         <plot_range lower="300" upper="1000" />
      </evaluator>
      <evaluator name="Constraint">
         <plot_range lower="0" upper="500" />
      </evaluator>
   </evaluators>
</algorithm>
```

Isječak 3.13: Konfiguracija višekriterijskog algoritma NSGA-III.

Konfiguracija algoritma NSGA-III u isječku 3.13 koristi dva evaluatora. Slika 3.4 prikazuje napredak algoritma zadanog gornjim isječkom nakon pet generacija.



(a) Početna generacija. **(b)** Peta generacija.

Slika 3.4: Prikaz rada višekriterijskog algoritma NSGA-III.

4. Rezultati

Raspodijeljene aplikacije nerijetko sadržavaju nekoliko desetaka komponenti i podatkovnih fragmenata [20]. Testiranja su provedena na zamišljenim modelima koji su inspirirani realnim raspodijeljenim aplikacijama.

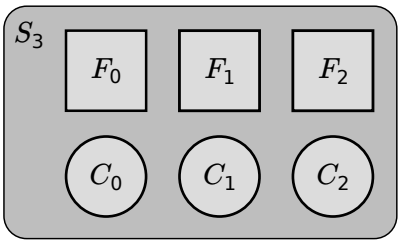
4.1. Jednostavna aplikacija

Aplikacija prikazana slikom 2.2 služi za provjeru ispravnosti rada algoritama i procjene sigurnosnog rizika. Okruženje aplikacije čine četiri mjesta rastućih razina povjerenja (tablica 4.1). Populacija je veličine sto jedinki, a iz dodatka A se vide važnosti resursa.

Jednokriterijski algoritmi pronalaze optimalno rješenje u manje od deset iteracija ako se samo ocjenjuje sigurnost resursa zasebno. Najbolje rješenje ovog slučaja je postavljanje svih fragmenata i komponenti na najsigurnije mjesto (slika 4.1). Korištenje `Single Resource` procjenitelja je uvijek preporučljivo, osobito s manjim koeficijentima. Na taj način algoritam bira rješenje koje koristi najsigurnija mjesta između svih rješenja koja imaju jednaku dobrotu po ostalim procjeniteljima sigurnosnog rizika.

Mjesto	Povjerenje
S_0	1
S_1	2
S_2	3
S_3	4

Tablica 4.1: Raspoloživa mjesta.



Slika 4.1: Smještaj na mjesto S_3.

Dodavanjem `Constraint` evaluatora dvostruko veće važnosti i sigurnosnog ograničenja $\{F_0, R_{12}\}$, optimalno rješenje se mijenja: na mjesto S_2 smještaju se F_0 i C_0. Fragment i komponenta čine doseg resursa F_0, pa je on dostupan samo na mjestu S_2. Najsigurnije mjesto pohranjuje ostale fragmente i komponentu C_1, pa je sigurnosno ograničenje zadovoljeno (slika 4.2).

Odabir koeficijenata evaluatora je hiperparametar koji ima velik utjecaj na dobivena rješenja – parametar oblikuje prostor pretraživanja. Da važnost `Constraint` evaluatora nije povećana, rasporedi na slikama 4.2 i 4.1 imali bi isti rizik, pa bi algoritam u jednakom broju slučajeva vraćao jedno i drugo rješenje. Izračun rizika za ta dva rješenja dan je u nastavku.

Resurs	Doseg	Dostupnost	Rizik
F_0	F_0, C_0	S_3	$\frac{100}{4}$
F_1	F_1, C_1	S_3	$\frac{200}{4}$
F_2	F_2, C_2	S_3	$\frac{300}{4}$
R_{12}	C_1, C_2	S_3	$\frac{40}{4}$
R_{02}	C_0, C_2	S_3	$\frac{50}{4}$

Tablica 4.2: `Single Resource` izračun rizika za raspoređivanje na mjesto S_3.

Tablica 4.2 prikazuje sve resurse aplikacije, njihove dosege, mjesta dostupnosti i rizika resursa zasebno u ponuđenom rješenju. Zbrajanjem navedenih rizika i dodavanjem rizika `Constraint` evaluatora jer se sigurnosno rješenje krši, dobiva se ukupni rizik rješenja:

$$1.0 \cdot \left(\frac{100}{4} + \frac{200}{4} + \frac{300}{4} + \frac{40}{4} + \frac{50}{4} \right) + 1.0 \cdot \frac{100}{4} = 197.5$$

Rješenje koje zadovoljava sigurnosnu zavisnost smješta resurse i na manje sigurno mjesto S_2, pa je rizik resursa zasebno veći (tablica 4.3). Nema kršenja ograničenja, no sigurnosni rizik je jednak:

$$1.0 \cdot \left(\frac{100}{3} + \frac{200}{4} + \frac{300}{4} + \frac{40}{4} + \frac{50}{3} + \frac{50}{4} \right) = 197.5$$

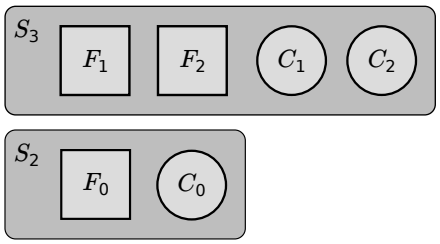
Udvostručivanjem koeficijenta `Constraint` evaluatora, kršenje ograničenja se dvostruko kažnjava:

$$1.0 \cdot \left(\frac{100}{4} + \frac{200}{4} + \frac{300}{4} + \frac{40}{4} + \frac{50}{4} \right) + 2.0 \cdot \frac{100}{4} = 222.5$$

Na ovaj način, algoritam preferira rješenje koje ne krši ograničenje (slika 4.2) jer njegov rizik strogo manji. Sigurnosne zavisnosti resursa su glavna motivacija za fragmentaciju i optimizaciju razmještaja, pa se u stvarnim aplikacijama očekuje da `Constraint` evaluator ima najveći utjecaj.

Resurs	Doseg	Dostupnost	Rizik
F_0	F_0, C_0	S_2	$\frac{100}{3}$
F_1	F_1, C_1	S_3	$\frac{200}{4}$
F_2	F_2, C_2	S_3	$\frac{300}{4}$
R_{12}	C_1, C_2	S_3	$\frac{40}{4}$
R_{02}	C_0, C_2	S_3, S_3	$\frac{50}{3} + \frac{50}{4}$

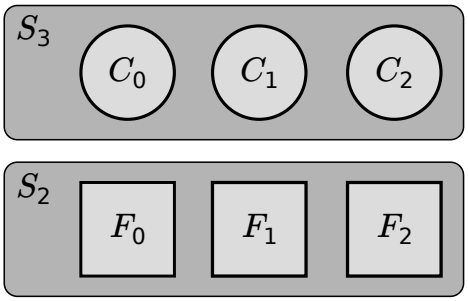
Tablica 4.3: `Single Resource` rizik za sliku 4.2.



Slika 4.2: Rješenje bez kršenja sigurnosne zavisnosti $\{F_0, R_{12}\}$.

Zamjenom `Constraint` procjenitelja `Resident` evaluatorom dovoljne važnosti, optimalno rješenje razdvaja komponente i fragmente. Rješenje koje smješta fragmente

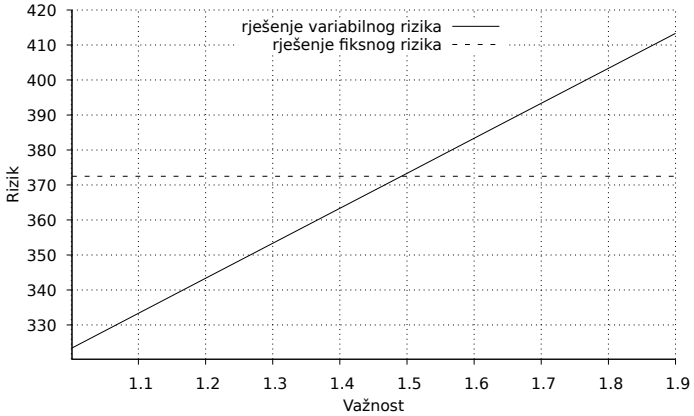
podataka na manje sigurno mjesto ima manji rizik od onog koje smješta fragmente na sigurnije mjesto. Razlog za to su dosezi resursa – su svi fragmenti dostupni na oba poslužitelja u oba rješenja. Smještanjem komponenti na sigurnije mjesto dobiva se sigurniji smještaj međurezultata, pa je takav razmještaj manje rizičan (slika 4.3).



Slika 4.3: Rješenje uz `Resident` evaluator.

Eksperimentalno je utvrđeno da je potrebna važnost za `Resident` procjenitelja približno jednaka 1.5 kako bi se dobilo rješenje na gornjoj slici. Manja važnost daje rješenje u kojoj su fragmenti i komponenta C_1 na najsigurnijem mjestu, a ostale dvije komponente na mjestu S_2. Takvo rješenje prihvaća jednostruku kaznu `Resident` evaluatora kako bi se osiguralo da su fragmenti F_1 i F_2 dostupni samo na najsigurnijem mjestu. Na taj se način sigurnosni rizik minimizira.

Rješenje na slici 4.3 ima fiksnu dobrotu jednaku 372.5, a rizik rješenja koje prihvaća kaznu `Resident` evaluatora ovisi koeficijentu evaluatora. Slika 4.4 prikazuje rizike tih rješenja u ovisnosti o važnosti `Resident` procjenitelja. Za neposredno manju važnosti od 1.5, rizici rješenja su jednaki.

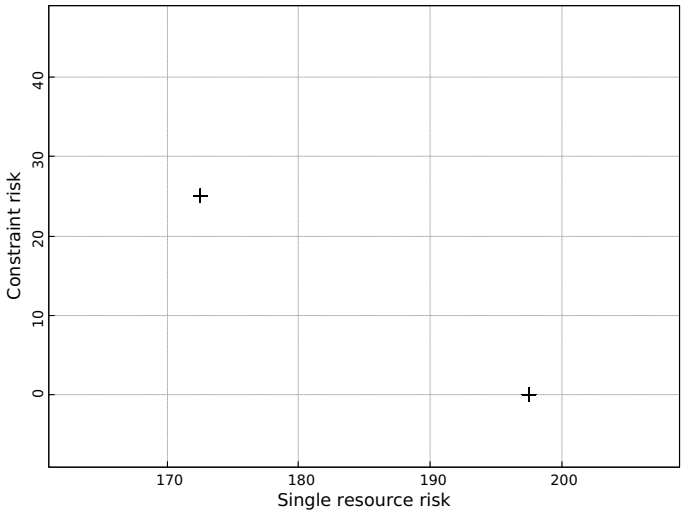


Slika 4.4: Rizici dva rješenja u ovisnosti o važnosti `Resident` evaluatora.

Određivanje koeficijenata kriterija koji rezultiraju željenim prostorom rješenja je glavni nedostatak upotrebe jednokriterijske optimizacije za višekriterijske probleme. Eksperimentalno je moguće odrediti vrijednosti koje daju očekivane odnose nekih rješenja, no za veći prostor pretraživanja i veći broj kriterija, postupak postaje nepraktičan.

Višekriterijska optimizacija ne zahtijeva poznavanje koeficijenata kriterija – važnost se implicitno određuje prilikom odabira jednog od nedominiranih rješenja. Slika 4.5 prikazuje rizike nedominiranih rješenja uz `Single Resource` i `Constraint` evaluatore. Ponuđena su dva rješenja:

1. Smještaj svih komponenti i fragmenata na najsigurnije mjesto (slika 4.1)
   - minimalni rizik `Single Resource` evaluatora: $x = 172.5$
   - kršenje sigurnosnog ograničenja na mjestu S_3: $y = 25.0$
2. Postavljanje fragmenta F_0 i komponente C_0 na mjesto S_2 (slika 4.2)
   - rizik `Single Resource` evaluatora: $x = 197.5$
   - nema kršenja sigurnosnog ograničenja: $y = 0.0$



Slika 4.5: Pareto fronta dobivena algoritmom NSGA-III za jednostavnu aplikaciju.

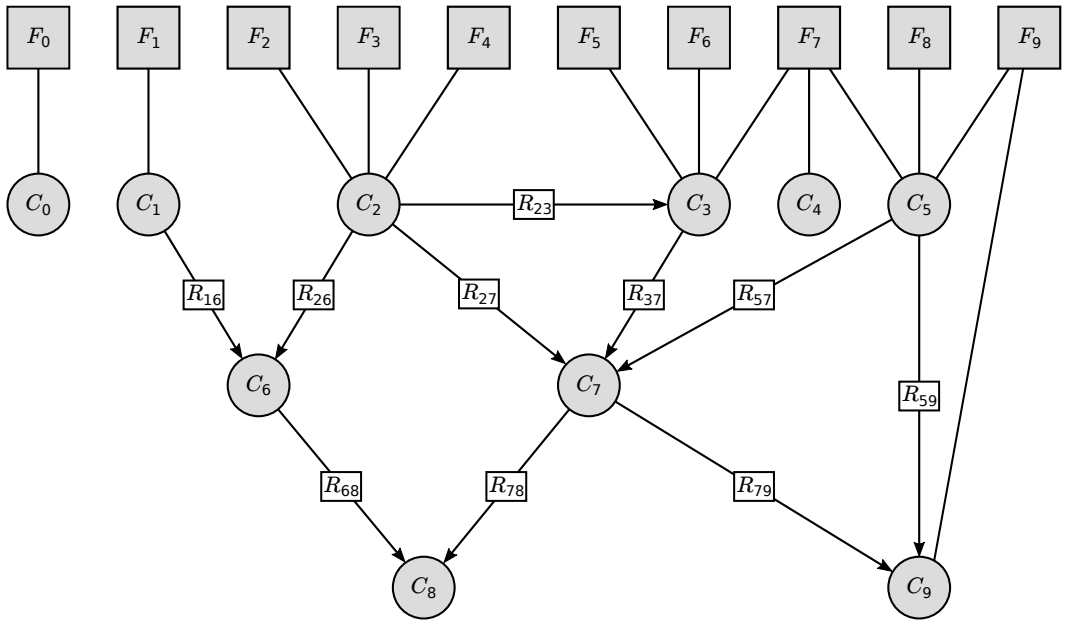
Algoritam PPES vrlo brzo dolazi do iste fronte, no u nerijetko gubi rješenje koje smješta sve fragmente i komponente na mjesto S_3. Kako to rješenje nije najbolje po `Constraint` kriteriju, grabežljivac zadužen taj evaluator ga može pojesti. Drugo nedominirano rješenje se također može izgubiti, ali se to događa rjeđe jer ne postoji puno rješenja koja su bolja po `Single Resource` kriteriju.

Višekriterijska optimizacija rješava problem određivanja važnosti kriterija, pa ju je preporučljivo koristiti. Kako Predator-Prey algoritam ne osigurava prelazak nedominiranih rješenja u sljedeću generaciju, preporučuje se koristiti NSGA-III jer daje konzistentnije rezultate – više pokretanja algoritma daje sličnije fronte.

4.2. Složenija aplikacija

Drugi primjer aplikacije je znatno složeniji. Sastoji se od deset fragmenata podataka, deset aplikacijskih komponenti i deset međurezultata. Slika 4.6 prikazuje navedenu

aplikaciju. Model sadrži komponente koje komuniciraju s nijednom ili više komponenti, komponente koje pristupaju jednom ili više fragmenata i fragmente kojima se pristupa s više komponenti.



Slika 4.6: Aplikacija od deset komponenti i deset fragmenata.

Okruženje aplikacije čini sedam raspoloživih mjesta rastuće sigurnosti (S_0 najmanje sigurno, S_6 najsigurnije). Prostor pretraživanja je veličine 7^{20} jer za svaki fragment i svaku komponentu postoji izbor od sedam mjesta. Definirano je deset sigurnosnih ograničenja: $\{F_0, R_{23}\}$, $\{F_2, F_6\}$, $\{F_3, F_4\}$, $\{F_3, F_6\}$, $\{F_3, R_{79}\}$, $\{F_7, R_{59}\}$, $\{F_9, R_{68}\}$, $\{R_{16}, R_{27}\}$, $\{R_{16}, R_{26}, R_{37}\}$, $\{R_{16}, R_{26}, R_{59}\}$.

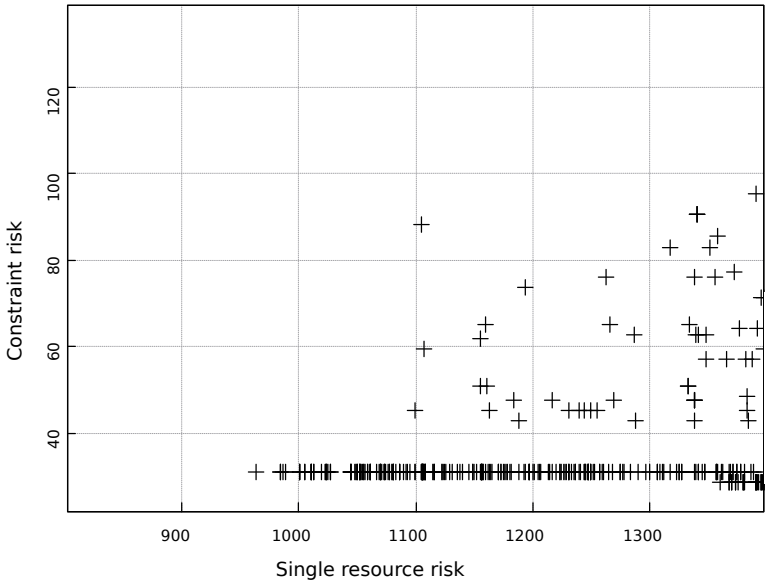
Bez stvarnih podataka, ograničenja ne mogu biti potpuno realna. Navedena su osmišljena da što bolje opišu ograničenja iz stvarnih aplikacija. Primjerice, ograničenje $\{F_7, R_{59}\}$ nemoguće je zadovoljiti jer presjek dosega tih resursa nije prazan skup.

U slučaju korištenja samo `Single Resource` evaluatora, troturnirski genetski algoritam pronalazi nabolje rješenje[1] u stotinjak generacija uz veličinu populacije od tisuću jedinki. Vrijednost 0.2 koristi se kao stopa mutacije.

Jednokriterijski algoritam implementiran korištenjem OpenGA knjižnice daje lošije rezultate uz istu konfiguraciju – samo se u 11% pokretanja algoritma dobiva optimalno rješenje. Povećanjem broja elitnih jedinki na desetinu populacije, optimalno rješenje je ponuđeno u 43% slučajeva. Daljnjim povećanjem broja elitnih jedinki ne postižu se bolje performanse algoritma. Uz veće stope mutacije, algoritam ne pokazuje znatno bolje čuvanje raznolikosti te unutar pedeset generacija dolazi do konvergencije. Za jednokriterijske probleme optimizacije razmještaja aplikacije, preporučuje se troturnirski genetski algoritam knjižnice ECF.

[1] Smještaj svih komponenti i fragmenata na najsigurnije mjesto S_6

Višekriterijska optimizacija je primjerenija za ovaj problem jer je računanje sigurnosnog rizika složeno, pa je određivanje prikladnih vrijednosti koeficijenata kriterija vrlo teško. Veličina problema također oslikava razlike u performansama algoritma NSGA-III i PPES. Predator-Prey višestruko sporije nalazi rješenja na istoj fronti, a glavni nedostatak je mogući gubitak rješenja na fronti što rezultira manjom pokrivenosti fronte.



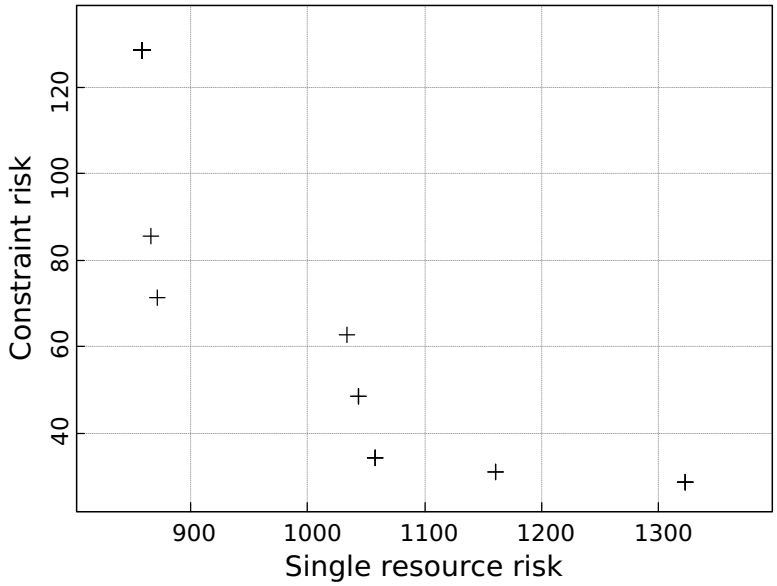
Slika 4.7: Rješenja ponuđena algoritmom PPES.

Slika 4.7 prikazuje populaciju PPES algoritma veličine četiristo jedinki nakon deset tisuća generacija. Problem ponuđenih rješenja je jako mala pokrivenost nedominirane fronte – samo rješenja $(931.5, 30.9)$ i $(1317.8, 28.6)$ čine pareto frontu. Izgled fronte upućuje da postoje rješenja s većim `Constraint` rizikom i manjim `Single Resource` rizikom od ponuđenih.

Velika koncentracija rješenja s `Constraint` rizikom približno jednakim trideset također je problematična, a javlja se zbog velikog broja rješenja koja krše malo ograničenja. Sva ova rješenja preživljavaju prisustvo `Constraint` grabežljivca jer su gotovo optimalna po tom kriteriju.

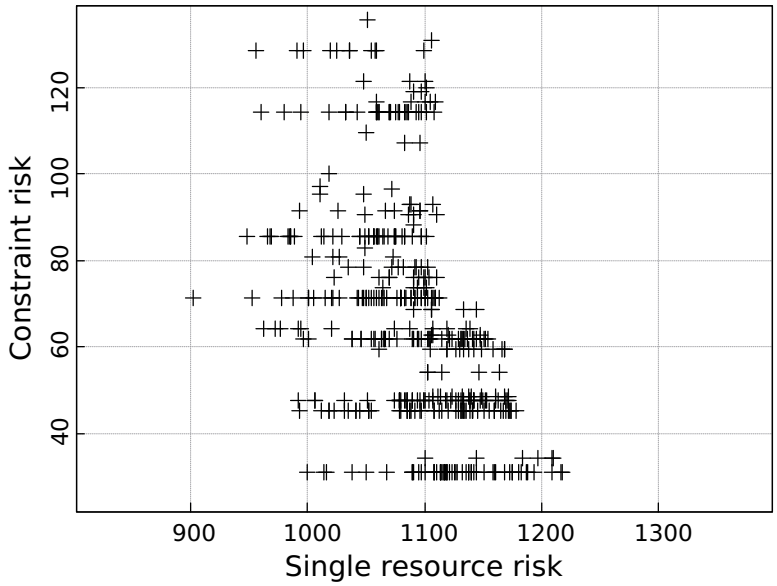
Algoritam NSGA-III očekivano daje bolja rješenja i brže dolazi do konvergencije. Slika 4.8 prikazuje pareto frontu koju nudi algoritam NSGA-III u stotinjak generacija. Fronta je opisana s osam vrlo raznolikih rješenja.

Stotinjak generacija algoritma NSGA-III traje oko četiri puta kraće od deset tisuća generacija PPES algoritma. Algoritam PPES izvrši dva reda veličine više evaluacija funkcija cilja i još uvijek dolazi do znatno lošijih rješenja, pa se preporučuje nedominirajući genetski algoritam za višekriterijski problem minimizacije sigurnosnog rizika.



Slika 4.8: Rješenja ponuđena algoritmom NSGA-III.

Nedominirano sortiranje omogućava veliku prednost algoritmu NSGA-III. U novu generaciju se prenose samo dobre fronte, a loše fronte ispadaju. Posljedica je da se rješenja u prostoru funkcija cilja gibaju prema lijevo i prema dolje, a prostor iza njih ostaje prazan. Slika 4.9 prikazuje dvadesetu generaciju algoritma na problemu optimizacije rizika složenije aplikacije.



Slika 4.9: Rad algoritma NSGA-III.

5. Zaključak

Predloženi model omogućava opis svih vrsta raspodijeljenih baza podataka i aplikacija. Procjena sigurnosnog rizika na temelju modela omogućava pronalazak razmještaja koji minimizira rizik. Procjena može sadržavati nekoliko komponenata koje se linearno kombiniraju kako bi se primijenila jednokriterijska optimizacija. Pristup se pokazao nepraktičan za velike aplikacije jer je vrijednost važnosti svakog kriterija vrlo teško unaprijed odrediti. Višekriterijska optimizacija provedena na modelu veličine realnih aplikacija pokazuje potencijal modela. Algoritam NSGA-III daje bolje rezultate od osnovne verzije algoritma PPES.

Osnovna motivacija za model su sigurnosne zavisnosti podataka – zbroj informacije dijelova podataka je manji od informacije njihove cjeline. Fragmentacijom i optimalnim razmještajem podataka i aplikacije smanjuje se sigurnosni rizik. Ispituje se i rizik svakog resursa zasebno te kažnjavanje zajedničkog smještaja aplikacije i podataka.

Predloženi model moguće je, primjerice, proširiti informacijama o cijeni i kvaliteti mjesta postavljanja podataka i aplikacije, pa bi se osim sigurnosti optimizirali troškovi i performanse aplikacije. Produbljivanje opisa sigurnosti također je moguće.

Literatura

[1] **Databreaches.net**, URL: https://www.databreaches.net.

[2] Anne Koziolek, Heiko Koziolek i Ralf Reussner, **PerOpteryx: Automated Application of Tactics in Multi-objective Software Architecture Optimization**, 2011.

[3] Christoph Rathfelder i Benjamin Klatt, **Palladio Workbench: A Quality-Prediction Tool for Component-Based Architectures**, 2011.

[4] Felix Willnecker i Helmut Krcmar, **Multi-Objective Optimization of Deployment Topologies for Distributed Applications**, 2018.

[5] Florian Kelbert i dr., **SecureCloud: Secure Big Data Processing in Untrusted Clouds**, 2017.

[6] Sergei Arnautov i dr., **SCONE: Secure Linux Containers with Intel SGX**, 2016.

[7] Victor Costan i Srinivas Devadas, **Intel SGX Explained**, 2016.

[8] Tie Hong i dr., **A Novel Vertical Fragmentation Method for Privacy Protection Based on Entropy Minimization in a Relational Database**, 2018.

[9] **Extensible Markup Language (XML)**, URL: https://www.w3.org/TR/xml.

[10] **Evolutionary Computation Framework (ECF)**, URL: http://ecf.zemris.fer.hr.

[11] Arash Mohammadi i dr., **OpenGA, a C++ Genetic Algorithm Library**, 2017.

[12] Thomas Williams i Colin Kelley, **Gnuplot**, URL: http://www.gnuplot.info.

[13] **Boost C++ Libraries**, URL: https://beta.boost.org.

[14] The MPI Forum, **MPI: Message Passing Interface Standard**, 1993.

[15] Kalyanmoy Deb i Himanshu Jain, **An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints**, 2013.

[16] Kalyanmoy Deb i dr., **A fast and elitist multiobjective genetic algorithm: NSGA-II**, 2002.

[17] Marco Laumanns, Günter Rudolph i Hans-Paul Schwefel, **A Spatial Predator-Prey Approach to Multi-objective Optimization: A Preliminary Study.** 1998.

[18] Kalyanmoy Deb i Udaya Bhaskara Rao, **Investigating Predator-Prey Algorithms for Multi-Objective Optimization**, 2002.

[19] Souma Chowdhury, Ramon Moral i George Dulikravich, **Predator-Prey Evolutionary Multi-Objective Optimization Algorithm: Performance and Improvements**, 2009.

[20] Mohammad Hajjat i dr., **Cloudward bound: planning for beneficial migration of enterprise applications to the cloud**, 2010.

Model za optimizaciju razmještaja raspodijeljene aplikacije

Sažetak

Predloženi model opisa raspodijeljenih aplikacija omogućava procjenu sigurnosnog rizika razmještaja komponenti aplikacije i fragmenata podataka. Ponuđena su tri procjenitelja sigurnosti: prekršene sigurnosne zavisnosti, sigurnost resursa zasebno te zajednički smještaj podataka i aplikacije. Pronalazak optimalnog razmještaja ispitan je jednokriterijskim i višekriterijskim optimizacijskim algoritmima. Napravljena je usporedba oba pristupa, dana je preporuka koje algoritme koristiti i opisana su moguća proširenja modela.

Ključne riječi: računarstvo u oblaku, raspodijeljene aplikacije, fragmentacija podataka, genetski algoritam, sigurnost aplikacija, višekriterijska optimizacija

Optimization model for distributed application deployment

Abstract

Proposed model for describing distributed applications allows for risk assessment of a given deployment of data fragments and application components. Three risk estimators have been described: broken security constraints, safety of individual resource and application and data co-residence. Optimal deployment strategy has been searched for using single objective multi objective optimization algorithms. Both approaches have been compared, suggestion has been made on which algorithm to use in which scenario and possible model extensions have been suggested.

Keywords: cloud computing, distributed application, data fragmentation, genetic algorithm, application security, multi objective optimization

Dodatak A
Cjelokupni opis aplikacije

Potpuni opis može biti zahtjevan za velike aplikacije, no korak procjene sigurnosti mjesta i važnosti resursa je daleko složeniji, pa se opis ne smatra kritičnim korakom. Dodatno, moguće je razviti alate koji mogu pomoći prilikom opisivanja. Konfiguracija koja opisuje jednostavnu aplikaciju sa slike 2.2 prikazana je u nastavku.

```
<resources>
   <fragment ID="F0">
      <importance>100</importance>
   </fragment>
   <fragment ID="F1">
      <importance>200</importance>
   </fragment>
   <fragment ID="F2">
      <importance>300</importance>
   </fragment>
   <result ID="R12">
      <importance>40</importance>
   </result>
   <result ID="R02">
      <importance>50</importance>
   </result>
</resources>

<application>
   <component ID="C0">
      <input>F0</input>
      <output>R02</output>
   </component>
   <component ID="C1">
      <input>F1</input>
      <input>F2</input>
      <output>R12</output>
   </component>
   <component ID="C2">
      <input>R12</input>
      <input>R02</input>
   </component>
</application>
```

```
<deployment>
   <server ID="S0">
      <trust>1</trust>
   </server>
   <server ID="S1">
      <trust>2</trust>
   </server>
   <server ID="S2">
      <trust>3</trust>
   </server>
   <server ID="S3">
      <trust>4</trust>
   </server>
</deployment>

<security>
   <constraint ID="0">
      <resource>F0</resource>
      <resource>R12</resource>
   </constraint>
</security>
```