SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 3304

Optimizacija putanje rezača za postupke automatskog rezanja

Marko Deak

Zagreb, lipanj 2013.

Zahvala

Ovaj rad izrađen je na Zavodu za elektroniku, mikroelektroniku, računalne i inteligentne sustave pod vodstvom mentora prof. dr. sc. Domagoja Jakobovića.

Zahvaljujem mentoru na uloženom vremenu i trudu, te prijedlozima i svoj ostaloj pomoći koju je pružio pri izradi ovog rada.

Također zahvaljujem kolegama koji su sudjelovali u izradi rada vezanog uz automatsko gniježđenje bez kojeg ne bi bilo ovog rada. Abecednim redom to su: Josip Feliks, Mario Kostelac, Edi Smoljan.

SADRŽAJ

1. Uvod

U većini problema koji se danas riješavaju u svijetu računarstva više nije moguće deterministčkim pristupom razviti algoritme koji će koristeći trenutnu tehnologiju do rješenja doći u nekom prihvatljivom vremenu. Budući da se tu najčešće radi o problemima gdje prostor rješenja raste eksponencijalno ovisno o broju ulaznih podataka govorimo o "netraktabilnim" ili tzv. NP-teškim problemima (engl. ***N****on-deterministic* ***P****olynomial-time hard*). Očito je da razvoj tehnologije nije toliko brz da može svladati ovakve probleme, te se tu okrećemo raznim pristupima koji ne traže optimalno već samo "dovoljno dobro" rješenje. Takve algoritme nazivamo heurističkim algoritmima.

U skupinu heurističkih algoritama pretrage spadaju i genetski algoritmi (engl. *genetic algorithms*, GA), ponekad nazivani i metaheuristikama. Oni spadaju u veću skupinu evolucijskih algoritama (engl. *evolutionary algorithms*, EA) koji traže rješenja koristeći razne tehnike inspirirane evolucijom koju nalazimo u prirodi. Zajednička odlika svih genetskih algoritama je da ne definiramo točno postupak rješavanja nekog problema, već ostavimo mogućnost modificiranja različitih parametara rješenja. Tada se nasumično generira određen broj potencijalnih rješenja problema, odnosno koristeći terminologiju genetskih algoritama generira se populacija s određenim brojem jedinki. Populaciju algoritam održava te iterativno provodi proces evolucije, simuirajući prirodnu evoluciju tako da nad jedinkama provodi selekciju, križanje, mutaciju i slično. Nakon svega, najbolja jedinka (rješenje) predstavlja rješenje problema.

U ovom radu demonstrirana je primjena genetskog algoritma na problem optimiranja putanje rezača u postupcima automatskog rezanja. Problem je takav da je unaprijed definiran broj oblika i njihovi položaji u 2D prostoru te je potrebno naći putanju kojom bi se kretao rezač kada bi bilo potrebno iz plohe izrezati sve zadane oblike, tako da se prijeđe što manji put, uz minimalan utrošak energije. Kako je pokazano već u radovima (Castelino, 2002), ovaj problem je NP-težak, te za dovoljno velik broj oblika nije moguće naći rješenje u prihvatljivom vremenu koristeći egzaktne postupke.

U dva poglavlja koja slijede nakon uvoda daje se uvod u područje genetskih algoritama, uz prikaz općenitih elemenata i principa rada genetskog algoritma, te detaljan opis problema automatskog rezanja. U četvrtom poglavlju prikazana je primjena genetskih algoritama na problem nalaženja putanje rezača, uz detaljno objašnjenje korištenih pristupa i dobivenih rezultata, te utjecaja različitih parametara algoritama na konačno rješenje. Na kraju kroz raspravu i zaključak su dodatno pojašnjeni dobiveni rezultati te su dane ideje za daljnji rad na problemu.

2. Genetski algoritmi

Genetski algoritmi su stohastička metoda optimizacije čiji se rad temelji na korištenju tehnika inspiriranih prirodnom evolucijom, kao što su mutacija, križanje, selekcija i slični. Kao što u prirodi evolucija omogućava preživljavanje i reprodukciju samo sposobnim (odnosno dobro prilagođenim) jedinkama, koje zatim reprodukcijom stvaraju još sposobnih jedinki, tako je i cilj genetskog algoritma počevši od populacije nasumično generiranih jedinki (rješenja) simulacijom evolucijskog procesa ostaviti samo ona rješenja koja su najbolja za dani problem te ih evoluirati dalje dobivajući bolja rješenja sve dok je to moguće.

Za razliku od deterministčkih algoritama, početna rješenja koja genetski algoritam daje potpuno su nasumična, te tek nakon provedenog procesa evolucije dolazi se do nekog "dovoljno dobrog" rješenja. Ovo se može činiti kao mana postupka, no dok god se ovakvi algoritmi pravilno koriste za određeni problem, konvergirat će prema dobrom rješenju u određenom broju generacija. Važno je napomenuti da rješenja koja su loša u jednoj generaciji, mogu u idućem koraku evolucije primjenom nekog od operatora genetskog algoritma dati najbolje rješenje. Zahvaljujući tom svojstvu i često relativno velikim populacijama koje se koriste, za razliku od tradicionalnih metoda optimizacije, genetski algoritam rijetko zapinje u nekim lokalnim maksimumima dok god je dobro konfiguriran za dani problem.

2.1. Elementi genetskog algoritma

Iako se mogu upotrebljavati u najrazličitijim situacijama, svi genetski algoritmi imaju određene zajedničke elemente koje je pri rješavanju problema korištenjem genetskih algoritama potrebno definirati. To su (Golub, 2004):

- Prikaz rješenja
- Inicijalizacija i uvjet zaustavljanja evolucijskog procesa

– Funkcija dobrote

– Postupak selekcije

– Genetski operatori

Pri izradi programskog rješenja ovog rada korišten je ECF (engl. ***E**volutionary **C**omputation* **F***ramework*), radni okvir za programske jezike C++ i Javu razvijen na Fakultetu elektrotehnike i računarstva pod vodstvom prof. dr. sc. Domagoja Jakobovića, koji omogućuje lako definiranje i promjenu svih navedenih elemenata genetskog algoritma uz korištenje velikog broja unaprijed definiranih elemenata.

2.1.1. Prikaz rješenja

Svi podaci koji obilježavaju jednu jedinku genetskog algoritma, odnosno rješenje problema, zapisuju se u jedan kromosom. Kromosom općenito definiramo kao bilo kakvu strukturu podataka koja opisuje svojstva jedinke. Primjeri prikaza koji se češće koriste su niz bitova (u ECF-u se ovo zove *Binary* genotip), niz realnih brojeva prikazanih kao brojevi s pomičnom točkom jednostruke preciznosti (ECF: *Floating Point* genotip) i dr.

Važno je da svaki kromosom predstavlja rješenje problema koji genetski algoritam rješava, no veoma je važno dekodiranje prikaza rješenja. Tako se svaki prikaz rješenja može koristiti na gotovo neograničenom broju problema bez promjene same strukture podataka, samo uz promjenu dekodiranja dobivenog kromosoma kako bi se dobilo rješenje pripadajućeg problema.

2.1.2. Inicijalizacija i uvjet zaustavljanja evolucijskog procesa

Kako bi se mogao provesti evolucijski proces nad populacijom rješenja, istu populaciju je potrebno inicijalizirati. Početnu populaciju potencijalnih rješenja predstavlja N jedinki sa strukturom podataka koja zadovoljava definirani prikaz rješenja. N je konstanta algoritma, odnosno kroz evolucijski proces ne mijenja se broj jedinki populacije. Proces inicijalizacije sam po sebi je jednostavan - svaka jedinka generira se nasumično, odnosno njena struktura podataka popunjava se nasumično generiranim vrijednostima. Neki od elemenata populacije također mogu biti rješenja dobivena nekim drugim optimizacijskim postupcima, pri čemu se onda i ta rješenja evoluiraju zajedno sa svim ostalim rješenjima u populaciji.

U jednom trenutku potrebno je zaustaviti evolucijski proces kako bismo uzeli najbolje rješenje. Najčešći uvjet zaustavljanja procesa je dosegnut određen broj iteracija, odnosno u kontekstu genetskog algoritma govorimo o generacijama. Ipak, mogući su razni kriteriji zaustavljanja, te njihovo kombiniranje. Neki od korištenih su dosezanje unaprijed postavljene ciljne vrijednosti funkcije dobrote, broj generacija bez poboljšanja (stagnacija), broj evaluacija funkcije dobrote, itd.

2.1.3. Funkcija dobrote

Mogućnost generiranja funkcije dobrote (engl. *fitness function*) može se smatrati prvim i najvažnijim preduvjetom izrade genetskog algoritma. Funkcija dobrote mora se moći izračunati za svaku pojedinu jedinku, te govori koliko je ta jedinka blizu idealnom rješenju. Vrijednost funkcije dobrote se izračunava za svaku jedinku u svakoj generaciji koristeći isti postupak (dekodiranje prikaza rješenja). Ovisno o implementaciji, veća ili manja vrijednost funkcije dobrote može označavati bolju jedinku, dok god se konvencije drži cijeli algoritam. Postupkom izračuna vrijednosti funkcije dobrote simulira se provjera prilagođenosti određenim uvjetima kakva postoji u prirodi te se omogućava kasniji postupak selekcije.

2.1.4. Postupak selekcije

Jedan od glavnih uvjeta za nalaženje dovoljno dobrog rješenja problema je taj da lošija rješenja nestaju iz sustava, odnosno "izumiru", dok bolja opstaju i iz njih se stvaraju nova rješenja. Kako bi se osiguralo eliminiranje lošijih rješenja u svakoj generaciji vrši se postupak selekcije. Postoje različiti načini provođenja selekcije, a najčešći modeli provedbe su generacijski jednostavni odabir (engl. *roulette wheel*), eliminacijski jednostavni odabir te turnirski odabir.

Kod generacijskog jednostavnog odabira vjerojatnost odabira neke jedinke u novu generaciju proporcionalna je dobroti te jedinke. Selekcija se provodi tako da se skalarne vrijednosti dobrota svih jedinki naslažu na brojevni pravac, nakon čega se slučajnim odabirom odabiru točke na tom pravcu. U novu generaciju ulaze one jedinke u čijem segmentu pravca se nalaze odabrane točke, tako da se na kraju jedinke odabiru svojevrsnim ruletom, od kud i dolazi englesko ime postupka. Eliminacijski jednostavni odabir ima sličan model, s razlikom da umjesto vrijednosti dobrota koristimo vrijednost "nekvalitete" jedinke

Turnirski odabir provodi se tako da se iz populacije stvori određeni broj podgrupa iz kojih se eliminira određen udio najgorih jedinki, te na njihovo mjesto dolaze nove jedinke dobivene genetskim operatorima. Preporučljivo je u svakoj generaciji (iteraciji procesa evolucije) promijeniti način na koji su jedinke grupirane kako bi se spriječilo zapinjanje u lokalnim maksimumima i proširio prostor pretraživanja.

2.1.5. Operatori genetskog algoritma

Operatori genetskog algoritma su križanje i mutacija. Upravo su oni najveća specifičnost genetskih algoritama i ono što njihove mogućnosti odvaja od mogućnosti nekih tradicionalnijih metoda optimizacije. Naime, upravo se primjenom genetskih operatora u populaciju rješenja uvode raznolikosti i noviteti i populaciju. Zahvaljujući tome, često je moguće iz početne skupine relativno loših jedinki (rješenja) dobiti neko "dovoljno dobro" u idućoj generaciji i izbjeći zapinjanje u lokalnim maksimumima vrijednosti funkcije dobrote. Naravno, moguća je i situacija da se baš ona dobra rješenja slučajno izgube, no kod pažljive uporabe operatora ovo se gotovo nikad ne bi smjelo događati u toj mjeri da utječe na rezultate algoritma.

Mutacija

Mutacija je unarni operator korišten za održavanje raznolikosti kroz generacije genetskog algoritma. Analogan je biološkoj mutaciji, te mijenja jednu ili više vrijednosti gena iz kromosoma (prikaza rješenja) iz početnog stanja jedinke. Ovisno o korištenom prikazu rješenja, mutacija se može provoditi na razne načine, no najjednostavniji i najčešće korišteni način je generiranje nasumične vrijednosti koja mijenja jednu od vrijednosti prisutnih u strukturi podataka koja čini kromosom. Vjerojatnost mutacije, kao i tip mutacije koju algoritam koristi, je parametar algoritma koji korisnik podešava pri inicijalizaciji. Vjerojatnost ne bi smjela biti preniska kako bi algoritam mogao izbjeći lokalne ekstreme funkcija. Također, ekstremno niske vjerojatnosti mutacije mogu rezultirati time da jedinke unutar populacije postanu preslične jedna drugoj, tako usporavajući ili čak zaustavljajući evoluciju. S druge strane, previsoka vjerojatnost mutacije pretvara algoritam u jednostavno slijepo pretraživanje.

Križanje

Križanje (engl. *crossover*) binarni je operator koji također omogućava održavanje raznolikosti unutar populacije stvarajući nove jedinke iz već postojećih. Općenito, odabire se par jedinki koje se u ovom kontekstu nazivaju "roditeljima", te se procesom križanja koji također može biti izveden na više različitih načina ovisno o prikazu rješenja stvara nova jedinka koju ovdje se zove "dijete". Važno je uočiti da dok se u procesu selekcije odbacuju jedinke koje nisu dovoljno dobre, procesom križanja dobivaju se nove jedinke kojima se popunjava populaciju sve dok nije iste veličine kao u prijašnjoj generaciji (kao što je rečeno ranije, ***N***, odnosno broj jedinki u populaciji je konstanta algoritma). Osim ovog općenitog postupka, također je moguće novu jedinku, tj. dijete, stvarati iz više od jednog roditelja, ovisno o algoritmu. Ono na što treba pripaziti je da prevelika učestalost rekombinacije gena može uzrokovati preranu konvergenciju algoritma te spriječiti dolazak do nekih boljih rješenja.

2.2. Općeniti pseudokod genetskog algoritma

Algoritam 2.1 Općeniti pseudokod genetskog algoritma

```
brojGeneracija ← 0
brojGeneracija ← generirajPocetnuPopulaciju()
dok uvjetZaustavljanjaNijeIspunjen() radi
    za i = 1 → broj_jedinki() radi
        jedinka.dobrota ← izracunajDobrotu()
    kraj za
    populacija ← provediSelekciju(populacija)
    provediKrizanje(populacija)
    provediMutaciju(populacija)
    brojGeneracija = brojGeneracija + 1
kraj dok
vrati nadiNajboljuJedinku(populacija)
```

3. Automatsko rezanje

Pod pojmom automatskog rezanja u ovom radu se podrazumijevaju dva problema koji se nadovezuju jedan na drugi. Prvi, koji se također sam po sebi naziva automatskim rezanjem (unatoč drugačijem imenu na engleskom jeziku) je automatsko gniježđenje, opisano u idućem odjeljku. Problem koji se nadovezuje na to je nalaženje optimalne putanje rezača pri postupku automatskog rezanja čime se ovaj rad više bavi, te je opisan nešto kasnije.

3.1. Opis problema automatskog gniježđenja

Problem automatskog gniježđenja, nekad nazvan i problem automatskog rezanja/krojenja (engl. *automatic nesting/cutting stock*), proizašao je iz potreba određenih grana industrije kao što su tekstilna, brodogradnja, autoindustrija i sl. Formulacija problema u stvarnom svijetu je jednostavna - na raspolaganju je ograničena količina materijala iz kojeg se mora dobiti (izrezati, ispiliti...) određene oblike. Potrebno je razviti metodu kojom će se naći optimalan raspored rezanja oblika kojim će se minimizirati neiskorišteni materijal. Problem se može formulirati kao

$$W_{total} = A_{total} - \sum_{i=1}^{n} AC_n$$

gdje je A_{total} ukupna površina iskorištenog materijala i jednaka je umnošku iskorištenih jedinki (ploča, traka...) materijala, AC_n označava površinu n-tog oblika koji je izrezan iz danog materijala, dok je W_{total} neiskorišteni materijal (engl. *waste*). Rješavanje problema automatskog gniježđenja se onda svodi na minimiziranje vrijednosti W_{total}. Naravno, iz motivacije za rješavanje problema proizlazi da preklapanje oblika nije dozvoljeno (eventualno dodirivanje bridova) jer bi predstavljalo beskorisno rješenje u stvarnom svijetu.

Problem sam po sebi spada u domenu problema cjelobrojnog linearnog programiranja (engl. *integer linear programming*), te kao većina problema te domene je NP-težak. Varijacije problema mogu se klasificirati na razne načine (Dyckhoff, 1990), no najčešća klasifikacija je ona po dimenzionalnosti problema. Najčešće su problemi iz ove domene jednodimenzionalni (rezanje dugih traka na dijelove zadane duljine) ili dvodimenzionalni (rezanje oblika iz zadanog broja ploha ograničene širine i duljine). Ipak, trodimenzionalne varijante problema postoje, iako se njih više razmatra u domeni problema pakiranja (engl. *packing problem*), koji je poprilično sličan i na 2D razini se može smatrati ekvivalentnim problemom.

3.1.1. Dosadašnji rad na problemu automatskog gniježđenja

Na projektu pod vodstvom prof. dr. sc. Domagoja Jakobovića obrađivao se baš problem automatskog gniježđenja uporabom genetskih algoritama. Problem se razmatrao na svojevrsnoj 1.5D razini, gdje se simulirala situacija sa 2D plohom ograničene širine, te se na nju morao smjestiti proizvoljan skup oblika, u opsegu projekta reduciranih samo na poligone, tako da nakon što se svi smjeste na površinu ukupna dosegnuta visina morala je biti što manja.

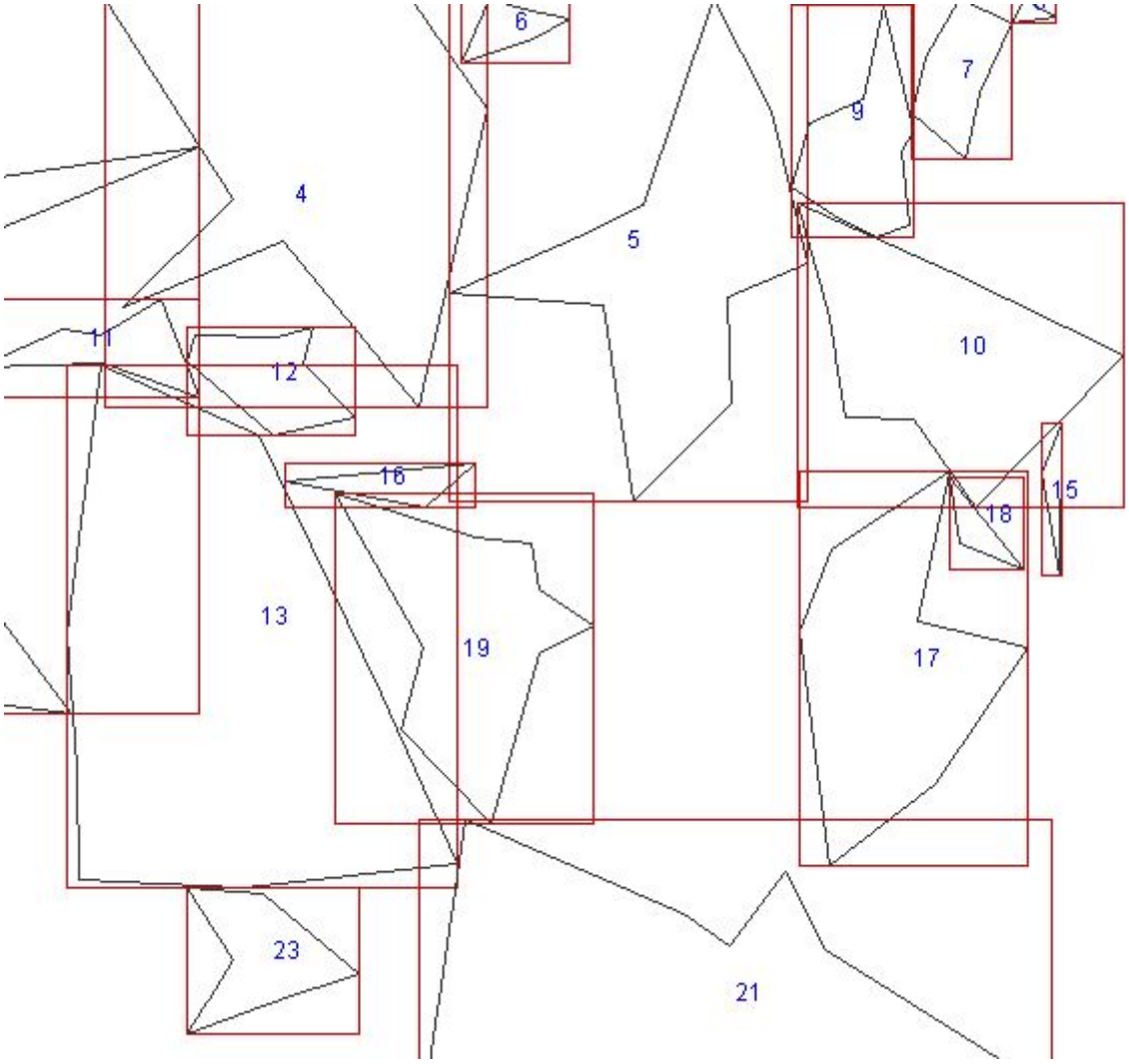
Najjednostavnije, i često ne baš učinkovito rješenje ovakvog problema je primjena *bottom-left* (BL) algoritma koji jednostavno uzima oblike redom kojim dolaze te ih smješta uvijek tako da im je donji lijevi rub smješten u slobodnu koordinatu što bližu ishodištu. Provjera je li neka pozicija slobodna da se poligon smjesti u nju rađena je tako da se jednostavno provjeravalo postoji li preklapanje površina pravokutnika koji okružuju pojedine poligone. Iako algoritam dolazi do rješenja, očito je da ono nije niti blizu zadovoljavajućeg budući da dosta mjesta koje se moglo iskoristiti za smještanje manjih poligona ostaje prazno. Algoritam je zato poboljšan tako da je raspored poligona koje algoritam smješta optimiziran uporabom genetskog algoritma. Genotip koji je korišten je ECF-ov *Permutation* genotip koji u suštini traži onu permutaciju danog niza koja rezultira najboljom vrijednošću funkcije dobrote. Dekodiranje i računanje vrijednosti funkcije dobrote u ovom slučaju su jednostavni jer je svaka permutacija zapravo raspored poligona koji se onda smješta koristeći BL algoritam, te se kao vrijednost funkcije dobrote koristi dosegnuta visina, gdje je manje bolje jer to znači da je iskorištena ukupno manja površina za smještaj poligona.

Nakon primjene genetskog algoritma uočena su znatna poboljšanja u iskorištenju dostupne plohe, no i dalje je postojao problem neiskorištenih površina koji je proizlazio iz prirode BL algoritma, odnosno pozicioniranja korištenjem pravokutnika koji okružuje poligon. Za tu svrhu korišten je algoritam lokalne pretrage Hooke-Jeeves čiji je pseudokod pokazan u nastavku. Postupak se vrši svaki put kada BL algoritam dodaje novi poligon, te se njime poligoni više približavaju međusobno nego u slučaju da se provjerava samo preklapanje pravokutnika, što za rezultat također ima bolje iskorištavanje površine.

Algoritam 3.1 Hooke-Jeeves postupak primjenjen na problem automatskog gniježđenja

$[x, y] \leftarrow [xDLpocetno, yDLpocetno]$ ▷ x i y koordinate donje lijeve točke novog poligona
$D \leftarrow [sirinaNovogPoligona/2, visinaNovogPoligona/2]$ ▷ Pomak po koordinati x i y u svakom koraku
dok |D[0]| > 1 ili |D[1]| > 1 **radi**
    $[x, y] \leftarrow [x - D[0], y]$ ▷ Pokušava se pomak po x koordinati
    ako preklapanje(noviPoligon, stariPoligon) **onda**
        $[x, y] \leftarrow [x + D[0], y]$ ▷ Ako ne valja, vraća se nazad
    kraj ako
    $[x, y] \leftarrow [x, y - D[1]]$ ▷ Pokušava se pomak po y koordinati
    ako preklapanje(noviPoligon, stariPoligon) **onda**
        $[x, y] \leftarrow [x, y + D[1]]$
    kraj ako
    $D \leftarrow [D[0]/2, D[1]/2]$
kraj dok

Sličan algoritam kasnije je primijenjen i za lokalnu pretragu koja se ne tiče koordinata, već rotacije novog poligona. Jedan primjer konačnog rada algoritma, tj rezultata njegovog rada prikazan je na slici 3.1., te je prilično zadovoljavajuć, pogotovo uzevši u obzir da je korišten samo dosta jednostavan genetski algoritam u kombinaciji sa osnovnom lokalnom pretragom.



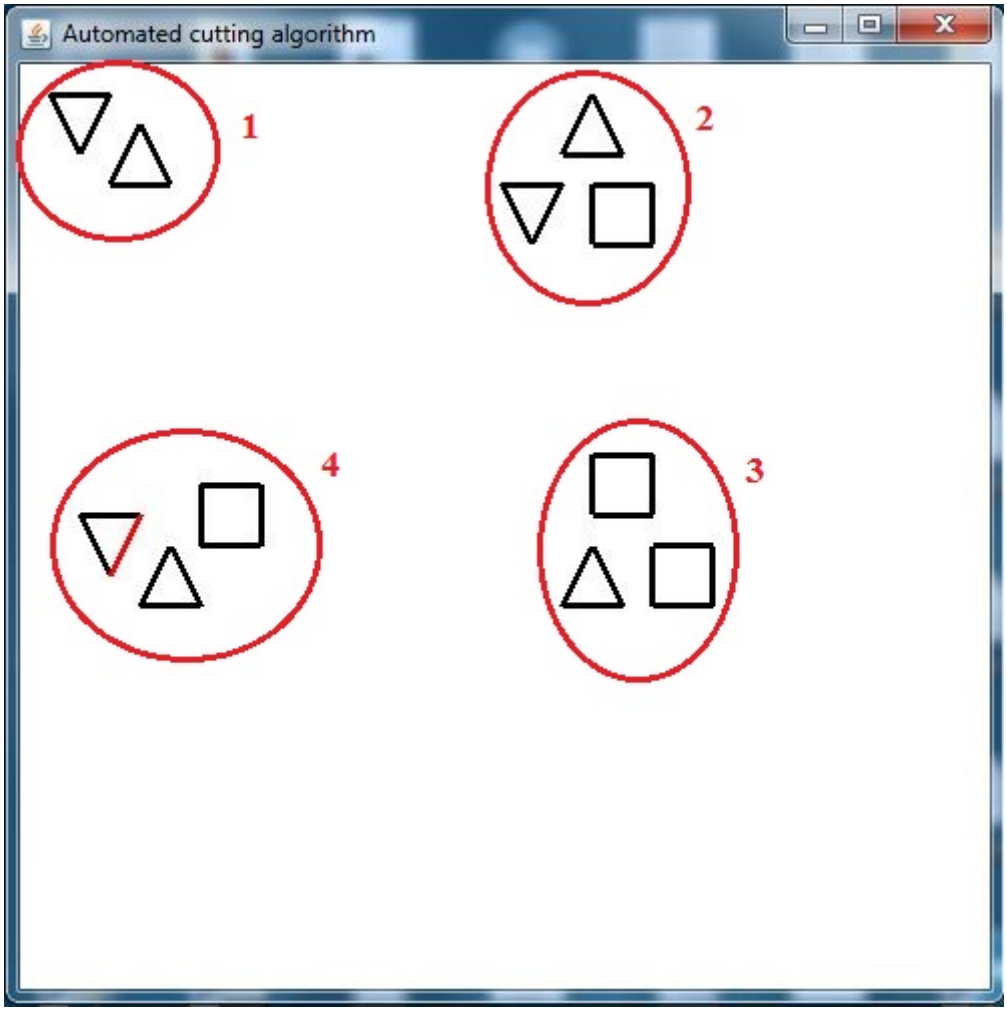
Slika 3.1: Dio plohe iz koje se režu dani poligoni. Sami poligoni označeni su crnom bojom, dok su crveni okviri oko njih najmanji pravokutnici koji ih obuhvaćaju

Iz slike je očito kako zahvaljujući lokalnoj pretrazi poligoni dobro prianjaju jedan uz drugoga te iskorištavaju prostor. Rješenje je dobiveno uporabom genetskih algoritama, a daljnji napredak je moguć korištenjem ostalih metoda optimizacije, no već je i ovo dovoljno dobro za traženje putanje rezača.

3.2. Problem nalaženja putanje rezača

Nakon rješavanja problema gniježđenja logično se postavlja pitanje kojom putanjom bi se trebao kretati rezač kako bi izrezao dane oblike (odnosno u ovom slučaju poligone). Intutivno je jasno da kada je prisutan skup od više poligona, nije svejedno kojim se putem rezač kreće kako bi prošao sve bridove oba poligona. Uzme li se tako

za primjer četiri grupe poligona, unutar grupe međusobno bliskih, ali da su u 2D prostoru grupe raspoređene tako da je put od jedne grupe poligona do druge puno veći nego što je to duljina bilo kojeg brida poligona iz bilo koje grupe očito je da je isplativije prolaziti bridove tako da se prvo prođu bridovi svih poligona u jednoj grupi, tada se prebaci na drugu, te nakon što se nju završi na treću, nego da se jednostavno ide od poligona do poligona, neovisno u kojoj su grupi, prelazeći nepotrebno mnogo puta veće udaljenosti nego u prvom primjeru. Ovakav primjer prikazan je na slici 3.2., gdje će očito kraći put biti onaj koji npr. prvo prolazi kroz sve poligone grupe 1, pa prelazi na one grupa 2, 3 i na kraju 4, nego onaj put koji obradi po jedan poligon iz grupe 1, nakon čega prelazi na jedan poligon grupe 2, itd.



Slika 3.2: Primjer situacije s grupiranim poligonima

Ovaj problem se javlja baš i u stvarnom svijetu - nakon što je dobiven donekle optimalan raspored oblika, te oblike je potrebno izrezati. Problemi iz ove domene najčešće se rješavaju za potrebe neke industrije, a u svakoj industriji se uvijek teži što

učinkovitijoj proizvodnji, gdje se pod učinovitošću podrazumijeva minimizacija otpada (neiskorištenih sirovina, tj. materijala), ali i vremena potrošenog u proizvodnom procesu. Zbog toga se ne može nikako dopustiti da rezač troši na svakoj ploči materijala više vremena nego što je potrebno.

Budući da je u ovom slučaju problem reduciran samo na poligone, može se predstaviti kao svojevrsni problem nalaženja puta kroz graf čiji su vrhovi upravo vrhovi svih poligona, gdje je cilj proći sve zadane bridove grafa. Na mjestima gdje ne postoji povezanost između vrhova (odnosno gdje ne postoji brid poligona), svejedno se može ići između dva vrha, kao što bi rezač mogao proći, ali u tom slučaju se potencijalno stvara potreba dizanja rezača od podloge i premještanja za što se pretpostavlja da također usporava proces i pokušava se izbjeći nepotrebno korištenje tog svojstva.

Ovaj problem optimizacije bi se matematički mogao prikazati kao:

$$min(P_{total}) = min \sum_{i=1}^{k} P_{i,i+1}$$

gdje je P_{total} ukupan prijeđeni put rezača, a $P_{i,i+1}$ put od i-tog vrha poligona do onog idućeg, gdje je k ukupan broj takvih puteva koji ovisi o ukupnom broju bridova poligona. Što manji k će najčešće rezultirati manjim ukupnim putem, a na algoritmu optimizacije je da odredi točan put, te točan k koji proizlazi iz tog puta.

4. Primjena genetskih algoritama na problem nalaženja putanje rezača

Iz opisa problema intuitivno je jasno da je prostor pretraživanja poprilično velik i da se za velik broj poligona rješavanje deterministčkim algoritmima pretrage jednostavno neće završiti u nama prihvatljivom vremenu. Zato u obzir dolazi primjena genetskih algoritama kao dobar način rješavanja problema. Prije primjene genetskih algoritama potrebno je definirati neka zajednička svojstva svih algoritama koji će se primjeniti.

Budući da je cilj algoritma posjetiti vrhove poligona tim redom da na kraju put sadržava sve bridove svih zadanih poligona, očito je da algoritam mora imati na raspolaganju listu vrhova koje će posjećivati, te uz nju također i listu bridova poligona po kojoj će se provjeravati jesu li posjećeni svi bridovi koji su trebali biti posjećeni. Dodatno, kako bi se postigla željena povezanost grafa, mora se i pridružiti svakom vrhu indeks poligona kojem on pripada. Naime, tako se može znati kada je potrebno dizati rezač, što se podrazumijeva kao skuplja opcija od običnog kretanja. Algoritam kao konačno rješenje mora dati listu vrhova onim redom kojim se posjećuju, te na neki način označiti kod kojih pomaka se rezač diže.

Problemu se pristupilo kao traženju najbolje permutacije s ponavljanjem liste indeksa vrhova svi poligona, ali uz nepoznat broj ponavljanja svakog pojedinog indeksa u listi. Kao najveći problem ovakvog pristupa problemu pokazala se baš činjenica da je broj ponavljanja nepoznat, što otežava dekodiranje genotipa i zahtijeva u nekim slučajevima dodatne pretrage.

Za razvoj genetskih algoritama korišten je ECF, odnosno njegova Java inačica, te su isprobani algoritmi sa različitim genotipima i dodatnim poboljšanjima. Korišteni genotipi su standardni u ECF-u, te je promjena parametara algoritma moguća jednostavnom promjenom konfiguracijskih datoteka, bez modifikacije postojećeg koda.

4.1. Funkcija dobrote

Iako su se koristili različiti prikazi rješenja, svaki korišteni algoritam je morao poštivati dano sučelje, odnosno vraćati putanju rezača u istom obliku. Zahvaljujući tome, za svaki algoritam korištena je ista funkcija dobrote, što olakšava međusobnu usporedbu algoritama i također smanjuje potrebu za modifikacijama postojećeg koda u slučaju dodavanja novih algoritama. Pseudokod izračuna funkcije dobrote dan je u algoritmu 4.1. Kao što bi trebalo biti očito iz pseudokoda, manja vrijednost funkcije dobrote je u ovom slučaju bolja (jedinka s manjom vrijednošću funkcije dobrote je bolje rješenje).

Algoritam 4.1 Izračun vrijednosti funkcije dobrote

$duljinaPuta \leftarrow 0$ ▷ Pretpostavka: putanja rezača prikazana je kao povezana lista vrhova poligona

$posjeceniBridovi \leftarrow []$ ▷ Potrebno je voditi i listu posjećenih bridova i kazniti svako rješenje koje ne odreže neke bridove

za svaki vrh vi iz putanje **radi**

ako postojiBrid(vi, vip) **onda** ▷ vip označava idući vrh, provjerava se postoji li brid koji sadrži ta dva vrha

$duljinaPuta \leftarrow duljinaPuta + sqrt((vi.x - vip.x)^2 + (vi.y - vip.y)^2)$ ▷ Ako postoji, samo se dodaje udaljenost između točaka

$posjeceniBridovi \leftarrow brid(vi, vip)$

inače

$duljinaPuta \leftarrow duljinaPuta + cijenaDizanja(vi, vip)$ ▷ Inače se pribraja cijena dizanja rezača, funkcija koja se može modificirati po potrebama korisnika

kraj ako

kraj za

za svaki brid b iz liste bridova **radi**

ako posjeceniBridovi ne sadrzi b **onda**

$duljinaPuta \leftarrow duljinaPuta * 2$

kraj ako

kraj za

vrati duljinaPuta

4.2. Pristup uporabom vektora realnih vrijednosti

Prvi iskušani pristup je uporabom ECF-ovog *FloatingPoint* genotipa. Kromosom ovog genotipa ustvari je vektor realnih brojeva prikazanih u memoriji pomoću broja s pomičnom točkom jednostruke preciznosti. Veličinu vektora, te granice unutar kojih se mora nalaziti svaki realni broj korisnik definira pri inicijalizaciji algoritma.

4.2.1. Prikaz rješenja i dekodiranje kromosoma

Kao što je rečeno, kromosom *FloatingPoint* genotipa je vektor realnih brojeva. U ovom slučaju svi brojevi nalaze se u intervalu $[0, 1]$ kako bi bilo što lakše univerzalno primijeniti razne postupke lokalne pretrage ili slične algoritme neovisno o broju poligona s kojim radimo. Pri dekodiranju tada svaki realni broj iz genotipa se množi sa ukupnim brojem vrhova poligona i zaokružuje na najbliži cijeli broj, te se iz vektora realnih brojeva dobiva vektor cijelih brojeva koji predstavljaju indekse vrhova. Red kojim se vrhovi tada posjećuju je zapravo red kojim su njihovi indeksi zadani u vektoru.

Pri dekodiranju se problematičnom pokazala činjenica da kod velikog broja vrhova zbog nasumičnosti se događala situacija da bi se određeni vrhovi posjećivali puno više puta nego što je to realno potrebno. Zbog toga pri dekodiranju se u slučaju da je vrh posjećen već 2 puta tražilo prvi vrh koji mu je najbliži po indeksu koji nije posjećen 2 puta. Točan pseudokod postupka dekodiranja kromosoma i dobivanja putanje rezača iz njega dan je u algoritmu 4.2.

Algoritam 4.2 Dobivanje putanje rezača iz jedinke prikazane *FloatingPoint* genotipom

$putanja \leftarrow []$ ▷ Putanja je prikazana kao lista vrhova
za $i = 0 \rightarrow velicinaVektora$ **radi**
$vrh \leftarrow vrhovi[zaokruzi(i * brojVrhova)]$
ako brojPosjecivanja(vrh) > 2 **onda**
za $j = 0 \rightarrow velicinaVektora/2$ **radi** ▷ Redom se gledaju najbrzi vrhovi i izlazi se cim se nade neki koji nije posjećen 2 puta
vrh = vrhovi[zaokruzi(i*brojVrhova)+j]
ako brojPosjecivanja(vrh) < 2 **onda** Izadi
kraj ako
vrh = vrhovi[zaokruzi(i*brojVrhova)-j]
ako brojPosjecivanja(vrh) < 2 **onda** Izadi
kraj ako
kraj za

kraj ako
$putanja \leftarrow vrh$
$brojPosjecivanja(vrh) \leftarrow brojPosjecivanja(vrh) + 1$
$posjeceniBridovi \leftarrow brid(vrh, putanja[velicina_putanje - 2])$
ako posjeceniBridovi sadrzi sve bridove **onda** Izadi
kraj ako
kraj za
vrati putanja

Problem koji se postavlja ovdje je da je pretpostavljeno da više od 2 posjeta kroz isti vrh nisu potrebna, dok se u stvarnosti ta granica pomiče ovisno o složenosti problema, odnosno o broju poligona. Isto tako bi se onda morala dinamički određivati svaki put i veličina korištenog vektora ovisno o broju poligona. Naime, očito je da duljina vektora realnih brojeva kojeg se koristi mora biti barem 2 puta veća od ukupnog broja vrhova svih poligona. Prevelika duljina vektor pak može nepotrebno otežati rješavanje i pretvoriti algoritam u slučaju pretragu. U rezultatima je kasnije prikazano kako zbog ovih problema ovaj pristup ipak ne daje prihvatljiva rješenja.

4.2.2. Poboljšanje uporabom Hooke-Jeeves postupka

Kako su svi brojevi unutar vektora normirani na interval $[0, 1]$ veoma je lako Hooke-Jeeves postupkom provesti lokalnu pretragu kako bi se nađena rješenja poboljšala. Pseudokod primjene Hooke-Jeeves postupka u kombinaciji s *FloatingPoint* genotipom za ovaj problem dan je u algoritmu 4.3. Postupak je primjenjivan na 2 načina: prvo je primjenjivan samo na konačnom (najboljem) rješenju koje algoritam daje, a zatim je i primjenjivan na određen udio jedinki (gdje bez obzira koliki je udio bio, uvijek se primjenjivao redom na najboljim jedinkama) u svakoj generaciji. Druga verzija se naravno pokazala vremenski iscrpnijom, no kako na ovoj razini to nije razmatrano kao veliki problem budući da je općenito vrijeme izvođenja algoritma dovoljno kratko. Usporedba oba načina primjene postupka lokalne pretrage dana je u odjeljku gdje je predstavljen utjecaj svih parametara algoritma na rješenje.

Algoritam 4.3 Primjena Hooke-Jeeves postupka za poboljšanje rješenja prikazanog *FloatingPoint* genotipom

$trenutniVektor \leftarrow jedinkaGA$ ▷ Sama predaja jedinke određuje se u algoritmu ovisno o tome želi li se samo najbolju ili više jedinki poboljšavati lokalnom pretragom

$delta \leftarrow 0.5$

dok delta > 0.01 **radi**

za $i = 0 \rightarrow duljinaVektora$ **radi**

$uvecani \leftarrow promijeni(i, delta, trenutniVektor)$ ▷ Funkcija promijeni vraća vektor kojem je element na indeksu danom kao prvi parametar promijenjen za vrijednost danu kao drugi parametar u odnosu na vektor predan kao treći parametar

$umanjeni = promijeni(i, -delta, trenutniVektor)$

ako izracunajFitness(uvecani) < izracunajFitness(trenutniVektor) **onda**

$trenutniVektor \leftarrow uvecani$

kraj ako

ako izracunajFitness(umanjeni) < izracunajFitness(trenutniVektor) **onda**

$trenutniVektor \leftarrow umanjeni$

kraj ako

$delta \leftarrow delta/2$

kraj za

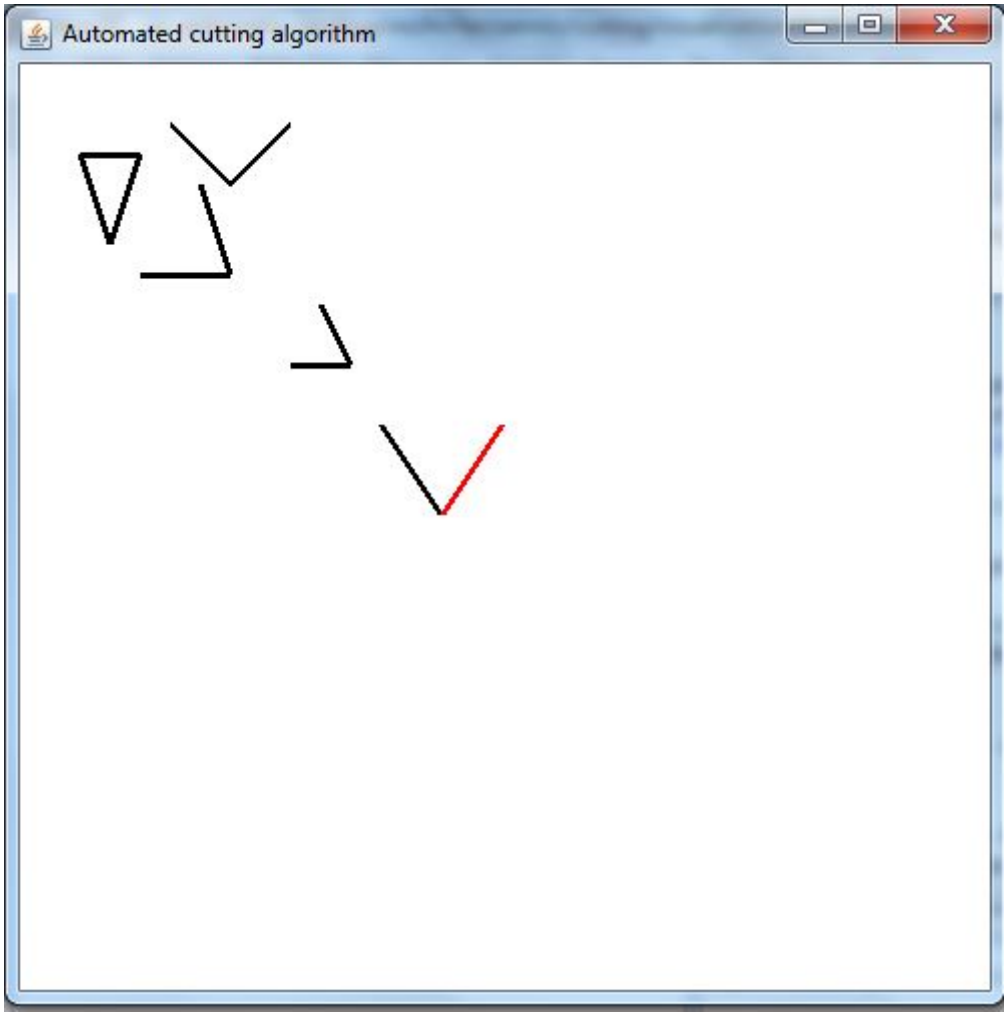
kraj dok

4.2.3. Rezultati primjene algoritma

Nakon implementacije, unatoč dobrom radu na potpuno jednostavnim primjerima sa 2-3 poligona, rezultati primjene algoritma na bilo kakvim složenijim primjerima nisu se pokazali zadovoljavajućima. Naime, putanja nađena očigledno je daleko od optimalne za bilo kakve slučajeve sa više manjih grupacija poligona. Rješenja dobivena ovim algoritmom će jednostavno "skakati" između tih grupacija, pritom očito trošeći previše vremena na te skokove. Primjer pojavljivanja ovog problema prikazan je na slici 4.1., gdje je prikazana vizualizacija procesa rezanja nakon određenog broja koraka. Radi se o 2 odvojene grupacije trokuta i pravokutnika s jednim trokutom između, gdje je onda intuitivno jasno da optimalna putanja mora prvo izrezati bilo koju

od te dvije grupacije, zatim trokut u sredini, te preći na drugu grupaciju. Crveno obojani brid označava brid koji se trenutno reže, te je očito da algoritam ovo ne poštuje, već započinje rezati bridove iz jedne grupacije, zatim skače na drugu potpuno bez opravdanja. Nešto kasnije prikazano je kako drugačiji pristup puno pravilnije rješava ovaj problem.

Ipak, potrebno je napomenuti kako na jednostavnim primjerima se vidi očita "pametna" konvergencija rješenja, gdje budući da je vektor realnih brojeva prevelik za stvaran broj koraka, algoritam konvergira prema rješenju koje više puta u nizu posjeti isti vrh. Time se postiže broj koraka jednak broju vrhova, a zapravo se ne stvara nikakav put, što je zapravo jednako kao da se spomenuti vrh samo jednom pojavio. Iz toga može se zaključiti da bi se pomoću *FloatingPoint* genotipa mogle prikazati permutacije s nepoznatim brojem ponavljanja, no samo dekodiranje genotipa kako bi se dobilo rješenje mora biti puno kompleksnije i obrađivati veći broj različitih slučajeva.



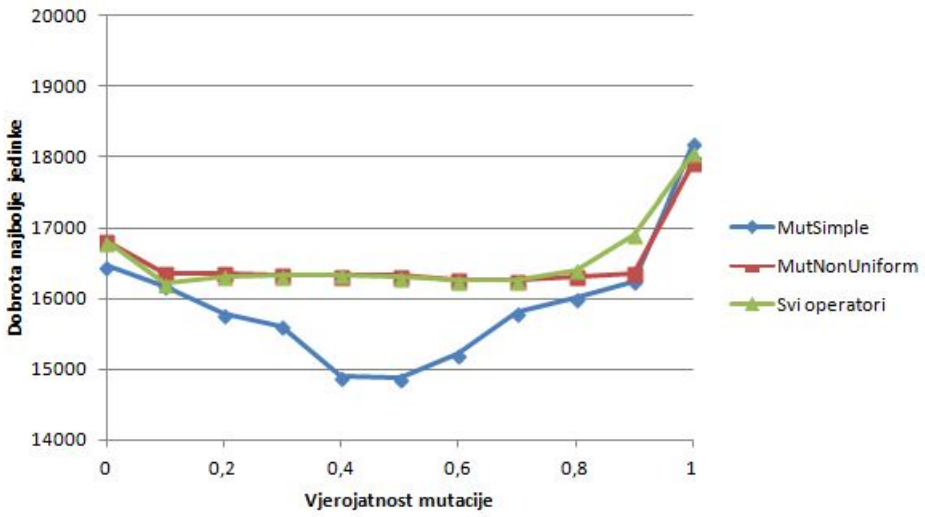
Slika 4.1: Problem sa skakanjem s poligona na poligon pri uporabi *FloatingPoint* genotipa

4.2.4. Utjecaj parametara algoritma na konačno rješenje

Ispitni skup korišten za ispitivanje utjecaja parametara kako ovog, tako i svih kasnije korištenih algoritama, sadrži 20 skupova nasumično generiranih poligona. Svaki skup sadrži nasumičan broj poligona između 10 i 30, od kojih svaki ima od 3 do 6 vrhova. Nad istim skupovima se testira svaki algoritam te se nakon izvođenja nad svih 20 skupova kao rezultat uzima srednja vrijednost funkcije dobrote. Ovaj postupak se ponavlja 50 puta te se na kraju zapisuje srednja vrijednost dobivenog rezultata, te je to vrijednost funkcije dobrote koja se prikazuje na grafovima koji slijede.

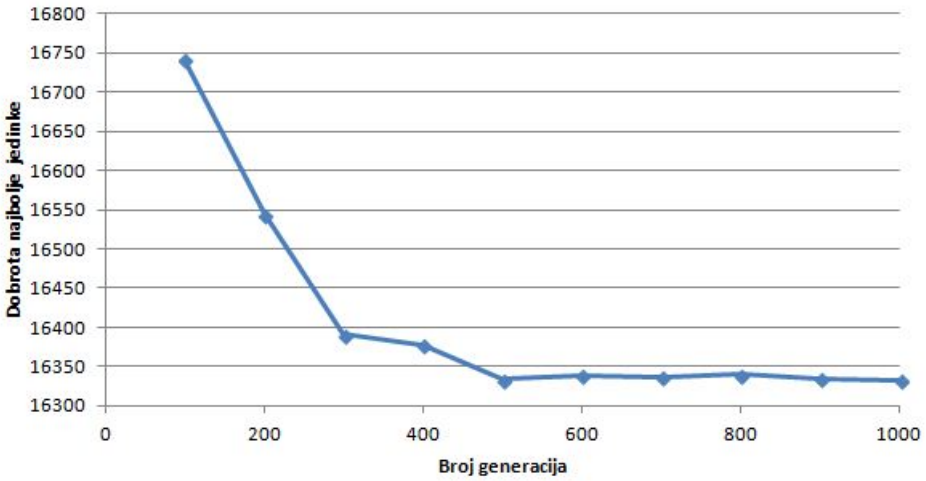
Prvi parametar čiji je utjecaj na konačno rješenje algoritma razmatran je genetski operator mutacije. U sklopu ovoga ispitan je utjecaj vjerojatnosti mutacije kao i uporabe različitih operatora mutacije. Točni rezultati predstavljeni su na slici 4.2. Prvi

operator mutacije korišten je jednostavan operator koji jedan nasumično odabran element iz vektora mijenja potpuno nasumično odabranim brojem iz zadanog intervala i na grafu je označen sa *MutSimple*. Drugi operator, na grafu označen kao *MutNonUniform*, te vidljivo lošiji odabir za ovaj postupak, također mijenja jedan element ali tako da se pri nižim generacijama uniformno pretražuje prostor rješenja, a pri višim generacijama se taj prostor smanjuje na lokalni. Korištenjem oba operatora mutacije zajedno dobivaju se rezultati bliski korištenju *MutNonUniform*, što i nije pretjerano zadovoljavajuće. Očito je da se na rubovima intervala vjerojatnosti dobivaju najgora rješenja, kao što je i očekivano, a iz grafa se pokazuje da je za ovaj algoritam najbolja vjerojatnost mutacije oko 0.4.



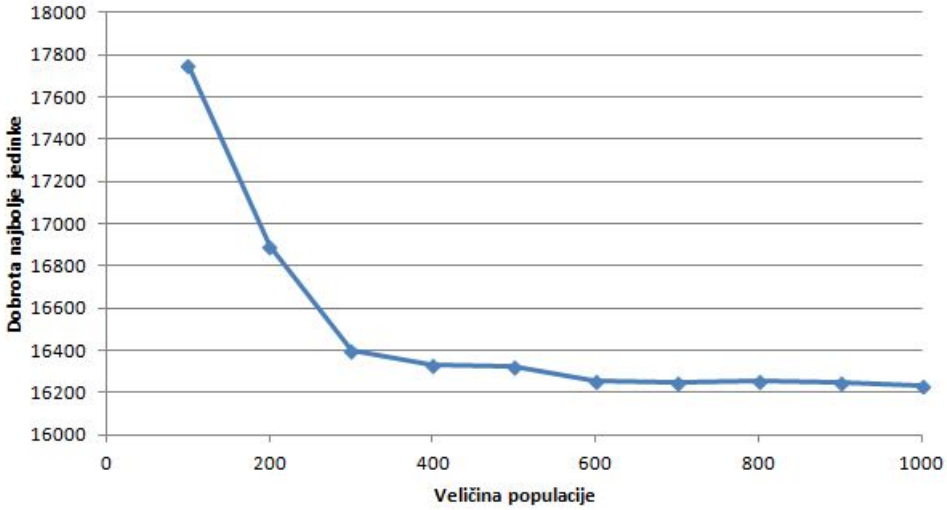
Slika 4.2: Utjecaj vjerojatnosti mutacije i korištenog operatora mutacije na dobrotu najbolje jedinke pri korištenju *FloatingPoint* genotipa

Drugi razmatrani parametar je broj generacija nakon kojih se postupak zaustavlja. Ovisnost dobrote o ovom parametru prikazana je na slici 4.3. i iz nje je očito kako nakon otprilike 300-400 generacija dobrota najbolje jedinke konvergira, te je nepotrebno koristiti više od toga. Korištenje što manjeg broja generacija naravno skraćuje vrijeme izvođenja algoritma, no kada se dobrota najbolje jedinke usporedi sa rezultatima dobivenim drugim algoritmima, ova vrijednost ipak nije nešto čemu bi se trebalo težiti.



Slika 4.3: Utjecaj broja generacija na dobrotu najbolje jedinke pri korištenju*FloatingPoint* genotipa

Razmatran je i utjecaj veličine populacije, odnosno broja jedinki, na konačno rješenje algoritma, te je ovisnost prikazana na slici 4.4. Kao i kod broja generacija, očita je relativno brza konvergencija koja se smatra pozitivnim faktorom za vrijeme izvođenja algoritma. Najboljom kombinacijom pokazuje se korištenje 400 jedinki i 400 generacija. Prije ovoga za sve algoritme korišteno je 400 jedinki i 500 generacija, pa zapravo pretpostavljene vrijednosti nisu bile daleko od onih opravdanih rezultatima.



Slika 4.4: Utjecaj veličine populacije na dobrotu najbolje jedinke pri korištenju *FloatingPoint* genotipa

Na kraju je razmatrano kolika točno poboljšanja unosi u algoritam korištenje Hooke-Jeeves postupka lokalne pretrage. Pri korištenju postupka samo na najboljoj jedinci na danom skupu je zabilježeno poboljšanje koje uglavnom uzima vrijednosti između 5% i 10%. Kada se Hooke-Jeeves postupak provodi na svakoj drugoj jedinci u svakoj generaciji, ukupno poboljšanje doseže vrijednosti od čak 20%, odnosno dobrota najbolje jedinke se sa prosjeka od 14892.7 mijenja na 12201.3 , no vrijeme izvođenja je za red veličine veće, odnosno povećava se sa prosječnog vremena od 52 sekunde na prosječno vrijeme od 3 minute i 12 sekundi, što kod primjera sa većim brojem poligona koje treba izrezati predstavlja priličan problem. Postupak lokalne pretrage je tako u ovom algoritmu možda i najkorisnija stvar od svih koje su istražene, no problemi koje unosi što se tiče vremena izvođenja su prilično teško premostivi. Eventualno se može razmatrati mogućnost primjene Hooke-Jeeves postupka na najbolje rješenje dobiveno nekim drugim algoritmom te onda pretvoreno u *FloatingPoint* genotip, no ovo dodaje nove komplikacije u dekodiranju genotipa.

4.3. Pristup uporabom permutacije vrhova

Kako je rečeno da se u osnovi radi o problemu nalaženja najbolje permutacije s ponavljanjem, isproban je i postupak uporabom ECF-ovog *Permutation* genotipa. U osnovi *Permutation* genotip je vektor cijelih brojeva koji se inicijalizira na željenu veličnu. Brojevi unutar vektora su tada u rasponu $[0, S-1]$ gdje S predstavlja veličinu vektora. Kao što ime govori, radi se o permutaciji brojeva, pa se niti jedan broj ne ponavlja, već je svaki cijeli broj intervala zastupljen jednom, a nasumičan je samo njihov raspored unutar vektora. Važno je uočiti da činjenica da je svaki broj zastupljen jednom otežava primjenu na problem nalaženja putanje rezača budući da iz opisa problema proizlazi da će barem jedan vrh morati biti posjećen dva puta, no ovaj problem riješava se pri dekodiranju genotipa kako je opisano u idućem odjeljku.

4.3.1. Prikaz rješenja i dekodiranje kromosoma

Budući da se kao rješenje traži permutacija s ponavljanjem, a genotip sam po sebi podržava permutaciju bez ponavljanja, pri dekodiranju je potrebno voditi računa o mogućim ponovnim prolascima kroz već posjećene vrhove. Za te svrhe vodi se lista "otvorenih" bridova. Otvorenim bridom ovdje se naziva onaj brid čiji je jedan vrh posjećen, no kroz sam brid se još nije prošlo. Tada u slučaju prijelaza s poligona na poligon, u slučaju da je vrh koji se posjećuje vrh jednog od otvorenih bridova, tada se ponovno

posjećuje drugi vrh brida. Ovakvo dekodiranje zapravo stvara i nepotreban put, no pri složenijim slučajevima sa većim brojem poligona ponovni prelazak jednog brida nije značajan faktor. Točan pseudokod dekodiranja kromosoma dan je algoritmom 4.4.

Algoritam 4.4 Dobivanje putanje rezača iz jedinke prikazane *Permutation* genotipom, permutiranjem vrhova

$putanja \leftarrow []$
$otvoreniBridovi \leftarrow []$
za $i = 0 \rightarrow velicinaVektora$ **radi**
  $putanja \leftarrow vrhovi[i]$
  $n \leftarrow duljina(putanja)$
  za svaki b iz otvoreniBridovi **radi**
    ako b sadrzi vrhovi[i] **onda**
      $drugiVrh \leftarrow nadiDrugiVrh(b, vrhovi[i]$ ▷ Funkcija traži drugi vrh brida
      ako !prije(drugiVrh, vrhovi[i], putanja) **onda** ▷ Funkcija provjerava je li prvi argument funkcije točno jedan indeks ispred argumenta predanog kao drugi argument funkcije u vektoru predanom kao treći argument
        $putanja \leftarrow drugiVrh$
        $putanja \leftarrow vrhovi[i]$
      kraj ako
    otvoreniBridovi.ukloni(b)
    kraj ako
  kraj za

kraj za
vrati putanja

4.3.2. Poboljšanje uporabom lokalne pretrage

Kako je u rješenju s *FloatingPoint* genotipom razmatran algoritam lokalne pretrage, tako je radi usporedbe i u ovom slučaju implementirana određena vrsta pretrage. Kako se radi sa cijelim brojevima iz intervala koji ovisi o ukupnom broju vrhova te razlika između elemenata može biti poprilično velika, postupci poput Hooke-Jeevesovog nisu tako lako primjenjivi. Zato je implementirana lokalna pretraga koja jednostavno pokušava zamijeniti redoslijed kojim se vrhovi svakog pojedinog brida posjećuju. Iako

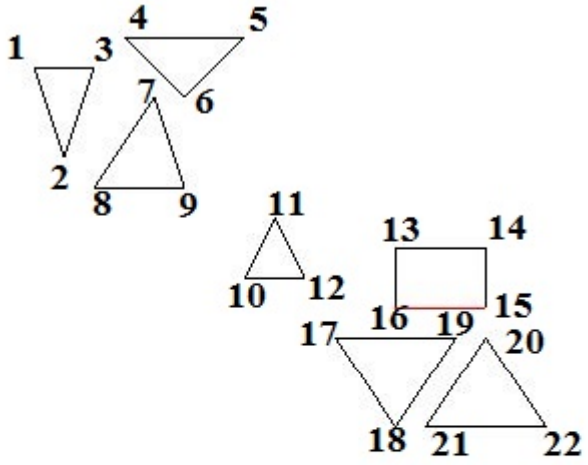
često neće naći znatna poboljšanja, ovaj postupak u situacijama kad se prelazi s pojedinog poligona na drugi može donekle poboljšati putanju. Postupak je prikazan u algoritmu 4.5.

Algoritam 4.5 Lokalna pretraga korištena za poboljšanje rješenja dobivenih uporabom algoritma

$trenutniVektor \leftarrow jedinkaGA$
za $i = 0 \rightarrow duljinaVektora - 2$ **radi**
$novaJedinka \leftarrow zamijeni(trenutniVektor, i, i+2)$ ▷ Funkcija zamijeni vraća novi vektor tako da zamijeni brojeve na indeksima danim kao drugi i treći parametar
ako izracunajFitness(novaJedinka) < izracunajFitness(trenutniVektor) **onda**
$trenutniVektor \leftarrow novaJedinka$
kraj ako
kraj za

4.3.3. Rezultati primjene algoritma

Rezultati primjene algoritma su prilično zadovoljavajući, te pokazuju značajno poboljšanje u odnosu na rezultate dobivene uporabom *FloatingPoint* genotipa. Ovdje više nije prisutan problem preskakanja bridova, kao ni problem skakanja s poligona na poligon. Za praćenje daljnjeg objašnjenja potrebno je promotriti sliku 4.2.



Slika 4.5: Rezultati dobiveni uporabom *Permutation* genotipa pri kraju crtanja. Numeriranje je kasnije dodano radi objašnjenja u radu

Algoritam temeljen na *Permutation* genotipu nalazi sljedeću putanju danu po indeksima vrhova (indeksi su dodani kasnije na sliku samo radi objašnjenja):

$$1 \to 2 \to 3 \to 1 \to 3 \to 4 \to 5 \to 6 \to 4 \to 6 \to 7 \to 8 \to 9 \to 7 \to 9 \to 11 \to 10 \to 12 \to 11 \to 12 \to 17 \to 19 \to 18 \to 17 \to 18 \to 21 \to 22 \to 20 \to 21 \to 20 \to 15 \to 14 \to 13 \to 16 \to 15 \to 16$$

U programu ova putanja je dekodirana iz sljedećeg vektora permutacija:

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 9, 11, 16, 18, 17, 20, 21, 19, 14, 13, 12, 15]$$

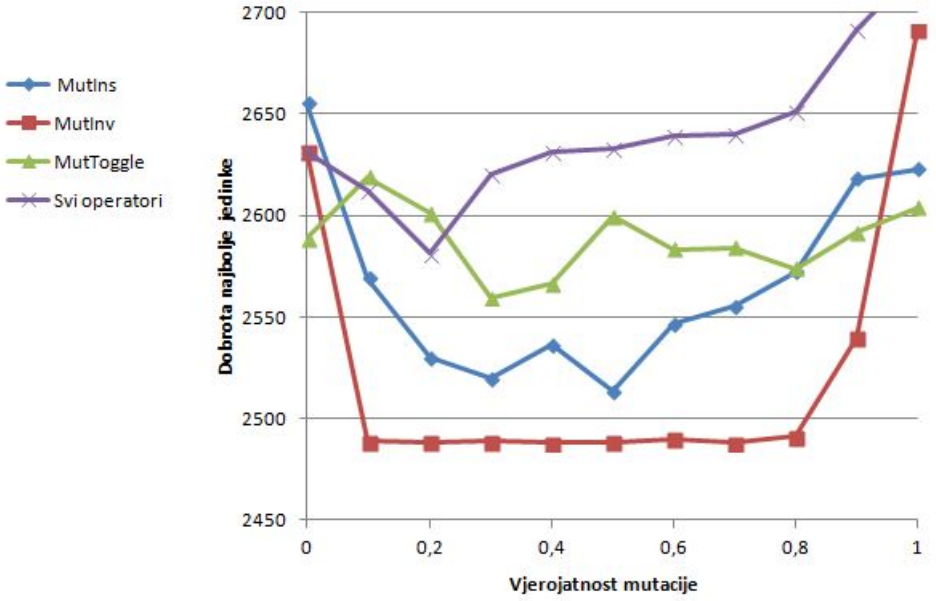
Promotrivši putanju, očito je da nema više preskakanja bridova prisutnih kod uporabe *FloatingPoint* genotipa, te se smisleno rezač prebacuje na jednu grupaciju poli-

gona tek nakon što je završio onu prijašnju kako bi si minimizirao put. Na ostalim primjerima, koji nisu predstavljeni slikama u radu, algoritam pokazuje slično "inteligentno" ponašanje, te se može zaključiti da ovaj algoritam generalno nalazi dovoljno dobra rješenja.

4.3.4. Utjecaj parametara algoritma na konačno rješenje

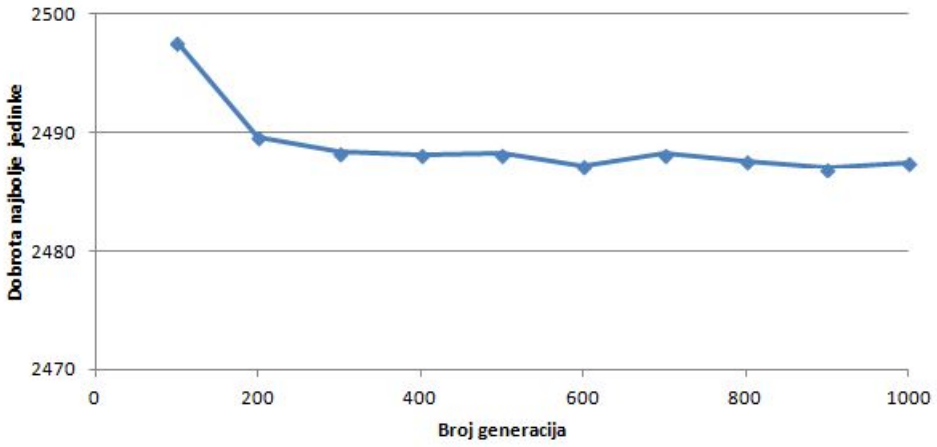
Pri ispitivanju utjecaja parametara algoritma ponovno je korišten isti skup ulaznih podataka kao i za testiranje algoritma koji radi sa *FloatingPoint* genotipom, pa se uz rezultate može i razmotriti prilično velika razlika u vrijednosti funkcije dobrote za najbolju dobivenu jedinku.

Ponovo je prvi parametar čiji se utjecaj na konačno rješenje razmatra operator mutacije. Graf koji pokazuje ovisnost vrijednosti funkcije dobrote za najbolju jedinku zadnje generacije pokazan je u nastavku, a također se uz vjerojatnost mutacije razmatraju i različiti operatori mutacije. Prvi nasumično određuje podniz jedinke čiji se elementi pomiču u desno (krajnji desni element postaje prvi element podniza) i na grafu je označen kao *MutIns*. Drugi nasumično određuje podniz elementa unutar kojeg se mijenjaju mjesta elemenata - prvi sa zadnjim, drugi sa predzadnjim itd. - te je na grafu označen kao *MutInv*. Zadnji je jednostavan operator koji nasumično mijenja 2 elementa permutacijskog niza, te je označen sa *MutToggle*. Ponovno je očita situacija da na rubovima područja, kako se približavamo vrijednostima 0 i 1 vrijednost funkcije dobrote je najlošija, jer ulazimo ili u područje gdje nema mutacije pa se ne unosi dovoljno raznolikosti u populaciju, ili je mutacija prečesta pa se praktički događa nasumično pretraživanje. Ipak, zanimljivo je uočiti kako ovdje puno više utječe pravilan odabir operatora mutacije na konačno rješenje - operator *MutInv* daje uvjerljivo najbolje rezultate, te već pri vjerojatnosti od 0.1 konvergira prema rješenju koje je uvjerljivo najbolje. Uporaba svih operatora mutacije zajedno, što ECF podržava, daje lošije rezultate od bilo kojeg od operatora zasebno, iako pri niskim vjerojatnostima daje nešto bolje rezultate nego *MutToggle*



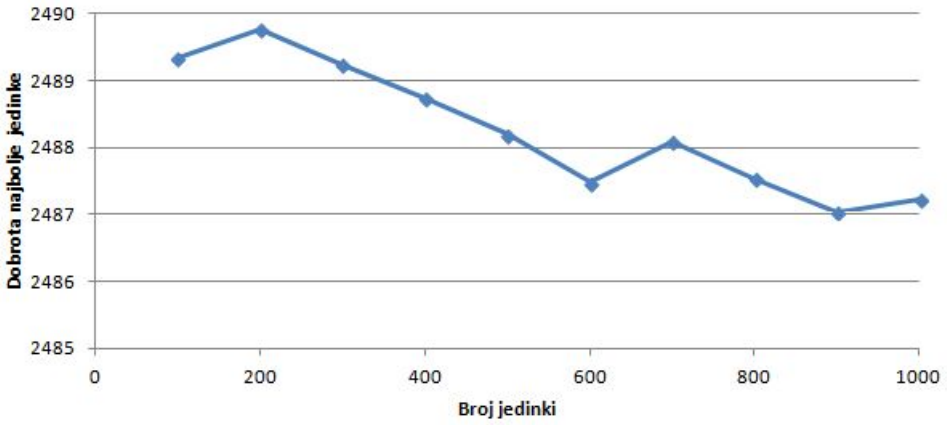
Slika 4.6: Utjecaj vjerojatnosti mutacije i korištenog operatora mutacije na dobrotu najbolje jedinke pri korištenju *Permutation* genotipa

Idući parametar koji se razmatra je broj generacija nakon kojih se algoritam zaustavlja. Na slici 4.6. prikazana je ovisnost konačne funkcije dobrote o korištenom broju generacija, te se može uočiti prilično brza konvergencija. Budući da algoritam daje dosta dobra rješenja, ovdje se tako brza konvergencija može smatrati izuzetno dobrim svojstvom. Naime uzmu li se u obzir slučajevi kad treba raditi s većim brojem poligona te je jedinka algoritma puno veća i memorijski zahtjevnija, veoma je dobro to što nije potrebno provesti evoluciju kroz ogroman broj generacija i tako dodatno usporiti izvođenje algoritma.



Slika 4.7: Utjecaj broja generacija na dobrotu najbolje jedinke pri korištenju *Permutation* genotipa

Također, razmatra se ovisnost dobrote najbolje jedinke o veličini populacije. Iz slike 4.7. ponovo se dolazi do zaključka da algoritam relativno brzo konvergira, te da nije potrebno koristiti ni preveliku populaciju, zbog čega se algoritam dosta brzo izvodi čak i za slučajeve kada je ukupan broj vrhova prilično velik.



Slika 4.8: Utjecaj veličine populacije na dobrotu najbolje jedinke pri korištenju *Permutation* genotipa

Posljednja stvar koju je potrebno razmotriti je utjecaj implementirane lokalne pretrage. Nakon nekoliko testiranja, dolazi se do zaključka da lokalna pretraga primjenjena na najboljem rješenju dobivenim algoritmom daje zanemarivo poboljšanje koje se kreće između 1% i 2% ukupne duljine putanje. Ako se lokalna pretraga primjeni na 50% jedinki u svakoj generaciji, poboljšanje u određenim slučajevima naraste i do 5% ukupne duljine putanje, no s obzirom na to da se vrijeme izvođenja algoritma poveća preko 2 puta, zapravo dobiveno poboljšanje nije nimalo isplativo s obzirom na uložene resurse.

4.4. Pristup uporabom permutacije bridova

Unatoč relativnom uspjehu postignutom uporabom permutacije vrhova, i dalje dano rješenje nije optimalan put zbog spomenutog problema višestrukog "skakanja" na već posjećene vrhove koje se pojavljuje zbog potrebe da se izrežu svi bridovi, odnosno da se postigne permutacije s ponavljanjem. Sagleda li se problem s druge strane, dolazi se do opažanja da bi se ovaj problem mogao izbjeći kada se ne bi permutirali vrhovi poligona, već bridovi. Zato se ponovno koristi ECF-ov *Permutation* genotip, ali uz drukčije dekodiranje rezultata.

4.4.1. Prikaz rješenja i dekodiranje kromosoma

Za razliku od permutacije vrhova, nije potrebno voditi računa o tome je li potrebno vraćati se na prijašnji element pri dekodiranju ili ne, budući da sigurno niti jedan brid nije potrebno ponovno rezati. Ono na što treba paziti je kojim redoslijedom se posjećuju točke brida, budući da se svaki brid može proći u 2 različita smjera. Zato se uvijek kao prva točka koja se posjećuje uzima ona manje udaljena od zadnje posjećene točke, osim u slučaju prvog brida. Točan algoritam dekodiranja dan je u obliku pseudokoda.

Algoritam 4.6 Dobivanje putanje rezača iz jedinke prikazane *Permutation* genotipom, permutiranjem bridova

$putanja \leftarrow []$
$permutacija \leftarrow najboljaJedinka$
$prviBrid \leftarrow bridovi[najboljaJedinka[0]]$
$udaljenostTockeA = nadiUdaljenost(prviBrid.a, bridovi[najboljaJedinka[1]])$
▷ Funkcija nalazi minimalnu udaljenost točke do jednog od vrhova brida
$udaljenostTockeB = nadiUdaljenost(prviBrid.b, bridovi[najboljaJedinka[1]])$
ako udaljenostTockeA < udaljenostTockeB **onda**
    $putanja \leftarrow prviBrid.a$
    $putanja \leftarrow prviBrid.b$
inače
    $putanja \leftarrow prviBrid.b$
    $putanja \leftarrow prviBrid.a$
kraj ako
za $i = 1 \rightarrow brojBridova$ **radi**
    $tockaA \leftarrow bridovi[najboljaJedinka[i]].a$
    $tockaB \leftarrow bridovi[najboljaJedinka[i]].b$
    ako udaljenost(tockaA, putanja[duljina(putanja)-1]) < udaljenost(tockaB, putanja[duljina(putanja)-1]) **onda**
        $putanja \leftarrow tockaA$
        $putanja \leftarrow tockaB$
    inače
        $putanja \leftarrow tockaB$
        $putanja \leftarrow tockaA$
    kraj ako
kraj za
vrati Putanja

4.4.2. Poboljšanje uporabom lokalne pretrage

Budući da je korišteni genotip identičan onome korištenom kod permutacije vrhova, te se algoritmi razlikuju samo u dekodiranju genotipa, za potrebe ispitivanja korišten je isti algoritam lokalne pretrage korišten kod permutacije vrhova i prikazan pseudokodom u odjeljku 4.3.2. Nažalost, kao i kod permutacije vrhova, značajna poboljšanja uporabom dane lokalne pretrage nisu uočena. Koristi li se lokalna pre-

traga na 50% najboljih jedinki u svakoj generaciji, vrijeme izvođenja algoritma raste sa prosjeka od 1.721 sekunde na čak 11.328 sekundi, dok se srednja vrijednost funkcije dobrote najboljeg rješenja poboljšava sa 2373.15 na 2128.3, što se ne može smatrati poboljšanjem koje opravdava toliko dulje vrijeme izvođenja.

4.4.3. Rezultati primjene algoritma

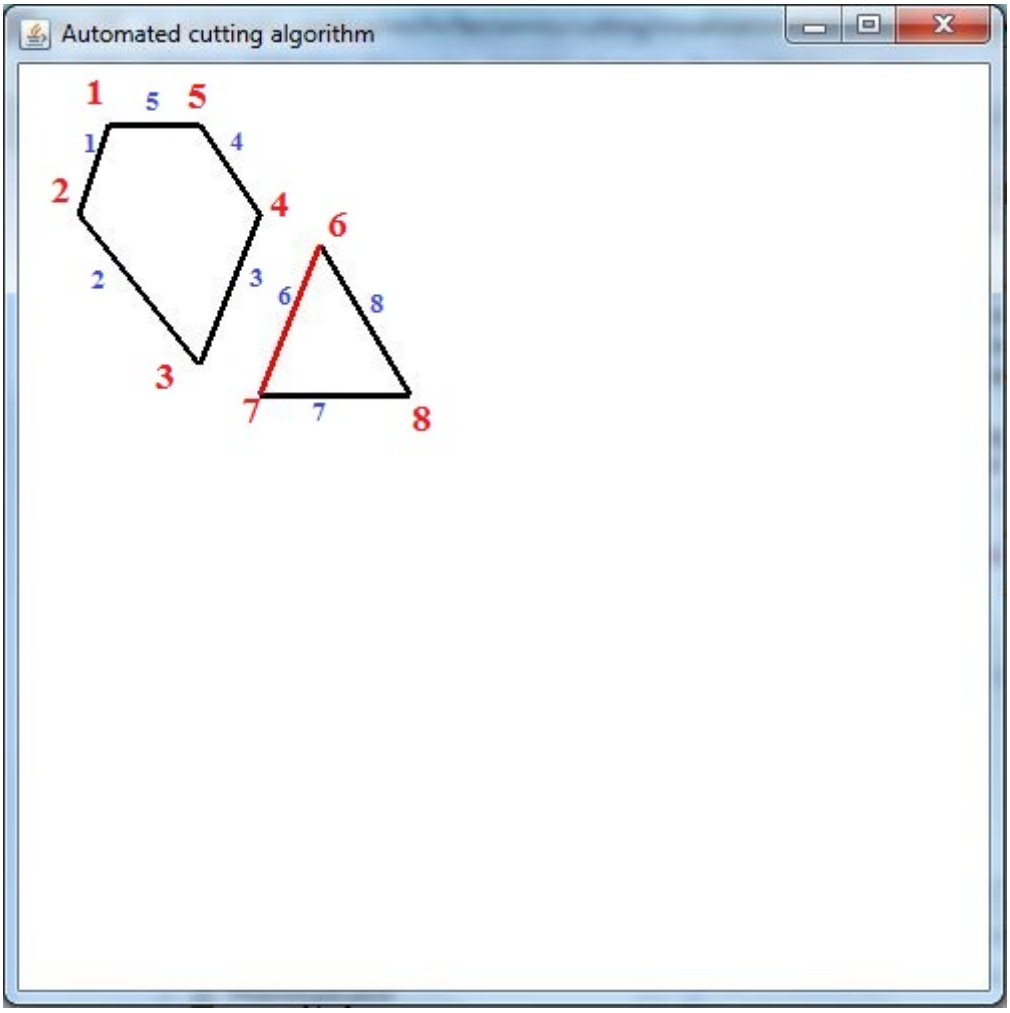
Rezultati dobiveni permutacijom bridova pokazali su se čak i boljima od rezultata dobivenih permutacijom vrhova. Kao prvo, problem bezrazložnog skakanja s jednog poligona na drugi nije prisutan, što je očekivano budući da se radi o jako sličnom pristupu, ovaj put uz garanciju da će se vrhovi jednog brida uvijek u dobivenom rješenju posjećivati jedan nakon drugog. Također je otklonjeno i mogućnost nepotrebnog vraćanja na već posjećene vrhove, jer se umjesto korištenja liste otvorenih bridova kao garancija rezanja koristi činjenica da će svaki brid biti izrezan (budući da se baš bridove permutira).

Zbog svega ovog, algoritam permutacije bridova može se smatrati trenutno najboljim pristupom od svih korištenih u ovom radu, te su rezultati nad posebnim slučajevima koji nisu nasumično generirani već pisani baš za ispitivanje ponašanja algoritma u osnovnim slučajevima i rubnim situacijama prikazani ovdje.

Prvi primjer sa samo 2 poligona prikazan je na slici 4.9., a putanja koja je dobivena radom ovog algoritma glasi (po indeksima vrhova):
$4 \to 5 \to 1 \to 2 \to 3 \to 4 \to 6 \to 8 \to 7 \to 6$

Intuitivnim pristupom i prosječna osoba bi došla do sličnog rješenja, pa iako nije provjereno je li ovo uistinu najkraći put, ako i nije vrlo vjerojatno po duljini se ne razlikuje bitno od optimalnog rješenja. U programu ova putanja je dekodirana iz sljedećeg vektora permutacija bridova. Potrebno je uočiti dvije stvari - indeksi u vektoru ne odgovaraju indeksima vrhova, već indeksima bridova koji se posjećuju, te indeksiranje u programu počinje od 0, za razliku od zapisa koji se ovdje koristi za prikaz.

$$[3, 4, 0, 1, 2, 7, 6, 5]$$



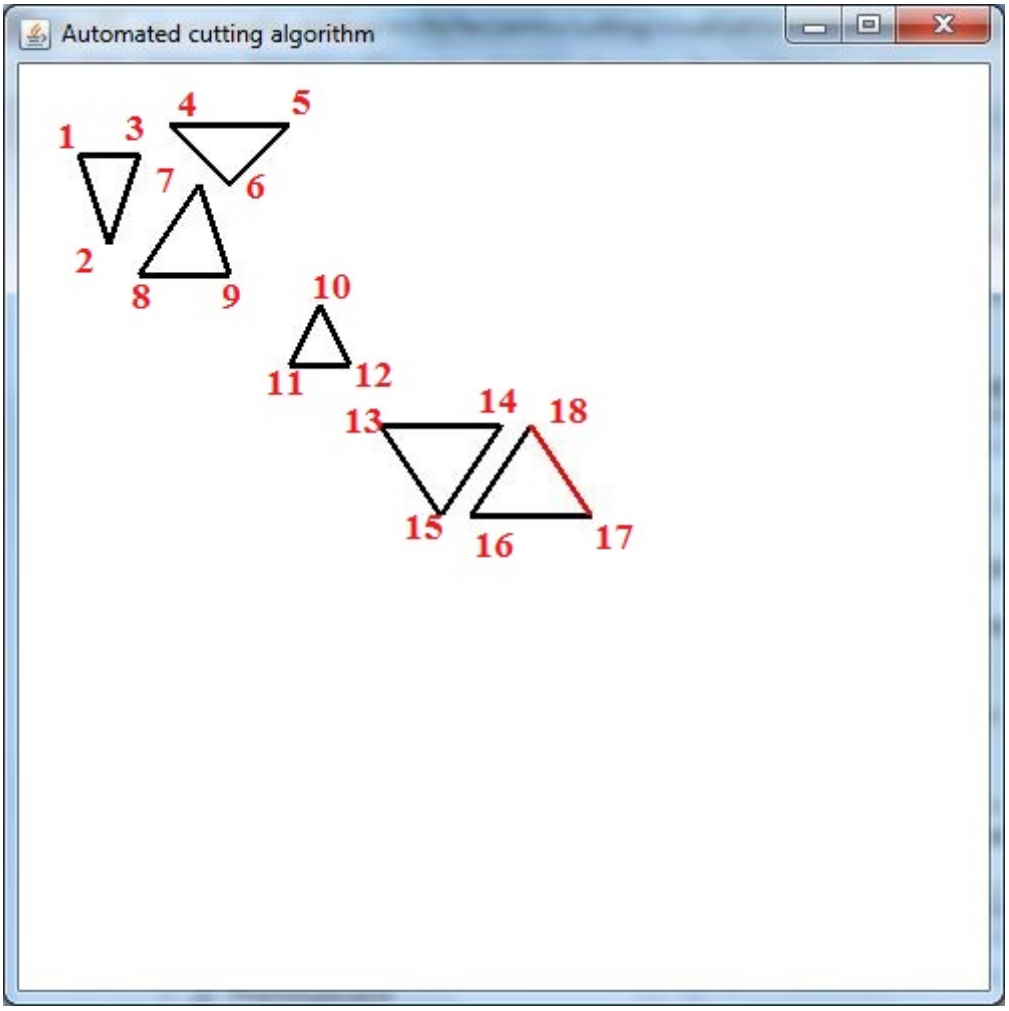
Slika 4.9: Primjer sa samo 2 poligona. Crveno obojanim brojevima označeni su indeksi vrhova, a plavim indeksi bridova

Drugi primjer je onaj koji je pokazan i u rezultatima algoritma s permutacijom vrhova i prikazan na slici 4.10.. Dobivena putanja je:

$3 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 4 \to 9 \to 7 \to 8 \to 9 \to 11 \to 10 \to 12 \to 11 \to 13 \to 15 \to 14 \to 13 \to 18 \to 16 \to 17 \to 18$

Zbog relativno velike zgusnutosti poligona na malom prostoru na slici na slikama 4.10. i 4.11. bridovi nisu indeksirani posebno, no slijedi se konvencija kao na slici 4.9., da je brid 1 između vrhova 1 i 2, brid 2 između vrhova 2 i 3 itd. uz iznimke bridova koji su između npr. vrhova 5 i 1 na slici 4.9., gdje je taj vrh indeksiran kao 5. Bridovi su u programu tada prikazani tako da indeksiranje počinje od 0 umjesto od 1. Dobivena putanja tada je u programu prikazana sljedećim genotipom:

$$[2, 0, 1, 3, 4, 5, 8, 6, 7, 9, 11, 10, 14, 13, 12, 17, 15, 16]$$



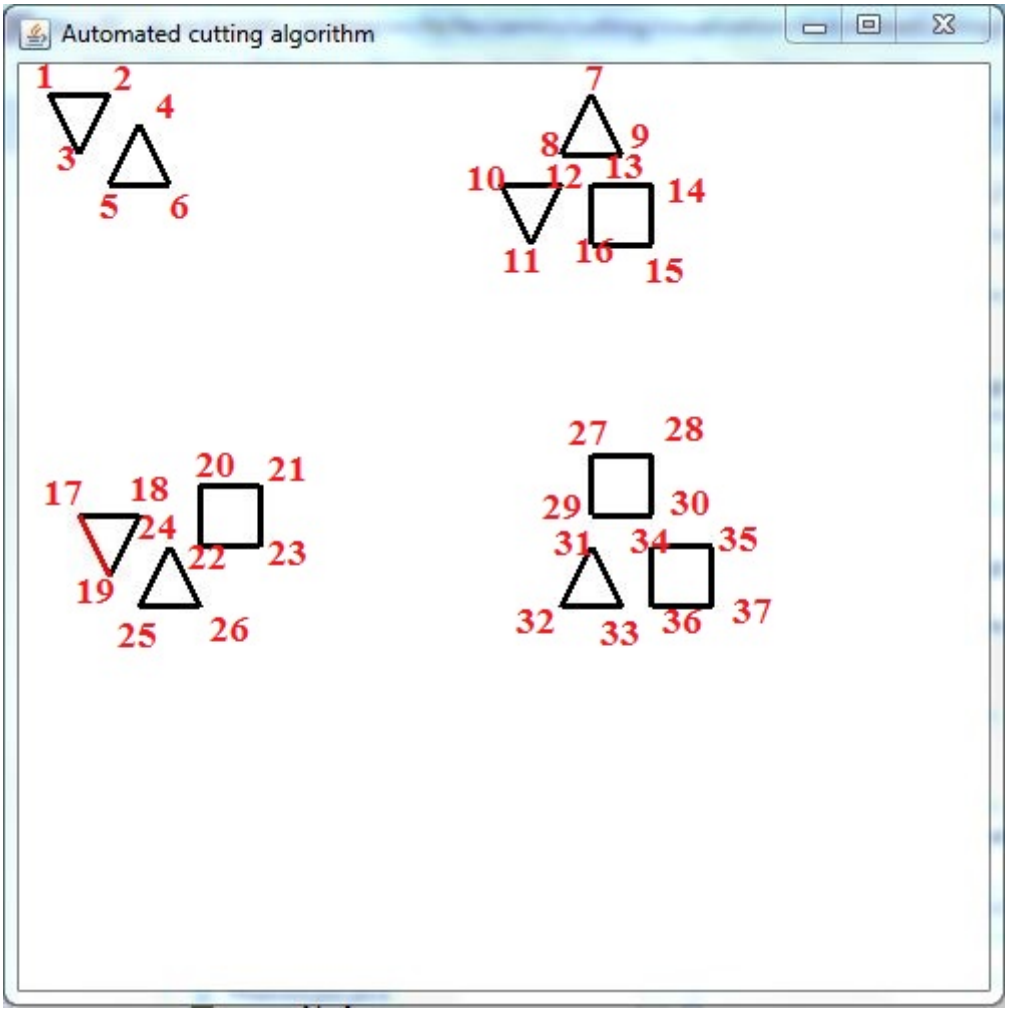
Slika 4.10: Primjer sa poligonima poredanim u dijagonali prikazan i za permutaciju vrhova

Zadnji primjer je onaj prikazan već i kod pojašnjenja problematike grupiranja poligona, sa 4 jasno istaknute grupe poligona. Skup je prikazan na slici 4.11. Putanja koju algoritam nalazi je:

$2 \to 1 \to 3 \to 2 \to 4 \to 5 \to 6 \to 4 \to 10 \to 11 \to 12 \to 10 \to 8 \to 7 \to 9 \to 8 \to 13 \to 14 \to 15 \to 16 \to 13 \to 29 \to 27 \to 28 \to 30 \to 29 \to 34 \to 36 \to 37 \to 35 \to 34 \to 31 \to 33 \to 32 \to 31 \to 23 \to 22 \to 20 \to 21 \to 23 \to 22 \to 24 \to 26 \to 25 \to 24 \to 19 \to 18 \to 17 \to 19$

Držeći se i dalje opisane konvencije za numeriranje bridova uz male promjene uzrokovane pojavljivanjem kvadrata, dana putanja u programu je prikazana kao vektor:

$[0, 2, 1, 3, 4, 5, 9, 10, 11, 6, 8, 7, 12, 13, 14, 15, 29, 26, 34, 28,$
$36, 35, 37, 33, 32, 31, 30, 21, 22, 25, 24, 23, 17, 16, 18]$

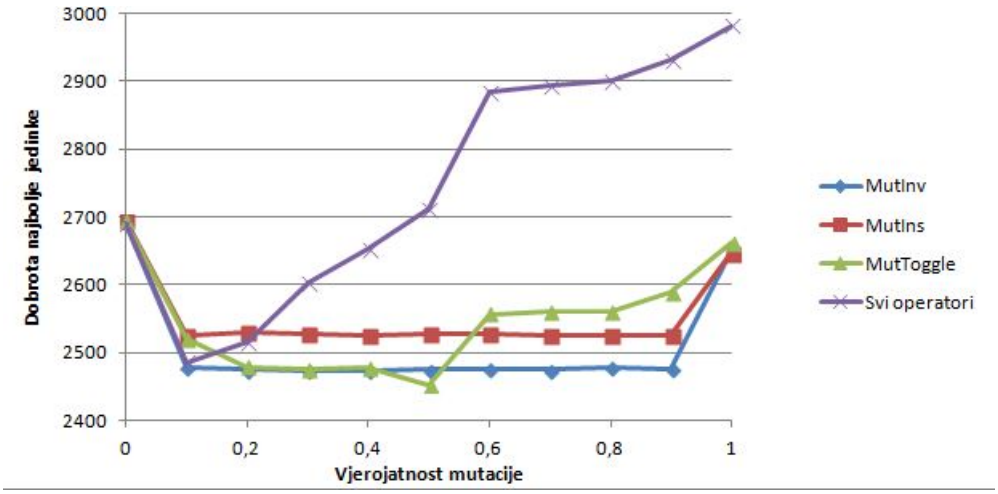


Slika 4.11: Primjer sa četiri odvojene grupe poligona

Također, važno je istaknuti da zbog jednostavnog dekodiranja i samim time evaluacije svakog rješenja unutar populacije, vrijeme izvođenja ovog algoritma je puno kraće od vremena izvođena algoritma koji se zasniva na permutaciji vrhova. Srednje vrijeme izvođena ovog algoritma na ispitnom skupu iznosilo je 1.721 sekundu, dok je permutaciji vrhova bilo potrebno čak 15.298 sekundi na istom skupu. Iz toga se dolazi do zaključka da iako permutacija vrhova potencijalno može dati jednako dobra rješenja kao permutacija bridova uz uporabu deterministčkih algoritama koji bi eliminirali nepotrebna višestruka vraćanja na već posjećene vrhove, vrijeme izvođenja algoritma s permutacijom bridova je toliko kraće da se može smatrati trenutno najboljim rješenjem problema.

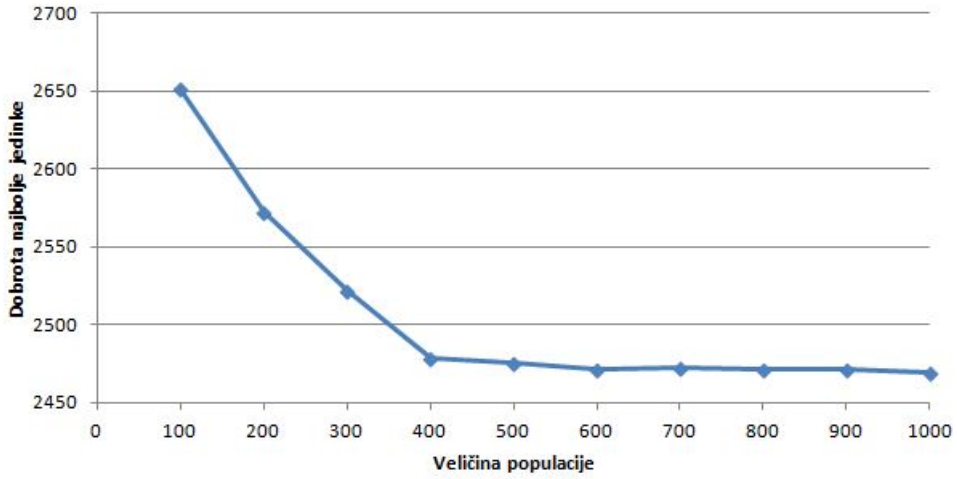
4.4.4. Utjecaj parametara algoritma na konačno rješenje

Ponovno je prvi parametar čiji se utjecaj na konačno rješenje ispituje operator mutacije, te budući da se ponovno radi sa ECF-ovim *Permutation* genotipom, koriste se isti operatori mutacije. Važno je uočiti da se u ovom slučaju *MutIns* pokazao kao najbolji operator za korištenje, za razliku od permutacije vrhova gdje je to bio *MutInv*. Ipak, ovdje razlika između operatora mutacije nije toliko uočljiva kao kod permutacije vrhova, te se svi otprilike jednako dobro ponašaju. Također, ponovno je uporaba svih operatora mutacije zajedno loša ideja, ovdje više nego inače jer dovodi do potpuno nepredvidljivih rezultata, koji su opet bolji od onih dobivenih drugim pristupima.

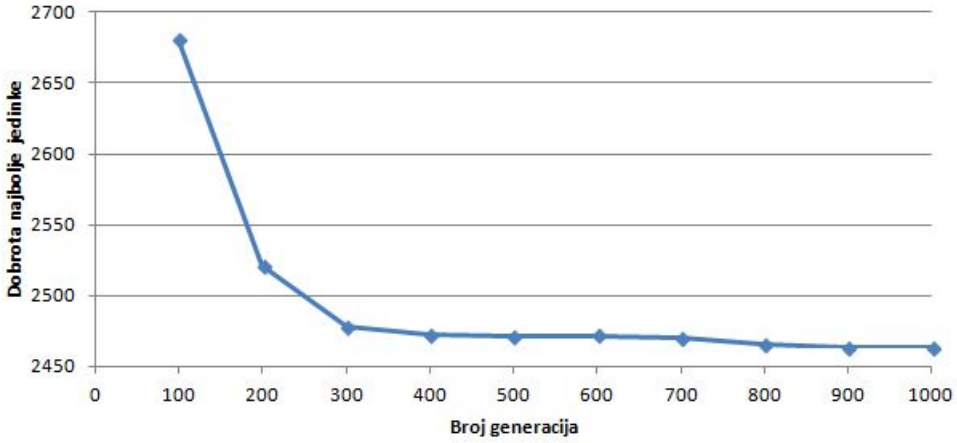


Slika 4.12: Utjecaj operatora mutacije na dobrotu najbolje jedinke pri korištenju *Permutation* genotipa, permutacijom bridova

Parametri koji su također ispitani su broj jedinki populacije te broj generacija nakon kojih se algoritam zaustavlja. Rezultati su ovdje prilično slični onima dobivenim kod ostalih algoritama pa nije potrebna detaljna rasprava. Obje ovisnosti mogu se viditi na sljedećim slikama



Slika 4.13: Utjecaj veličine populacije na dobrotu najbolje jedinke pri korištenju *Permutation* genotipa, permutacijom bridova



Slika 4.14: Utjecaj broja generacija na dobrotu najbolje jedinke pri korištenju *Permutation* genotipa, permutacijom bridova

5. Rasprava

Kako se pokazuje, genetski algoritmi su primjenjivi na ovaj NP-težak problem. Iako rezultati uporabom *FloatingPoint* genotipa nisu previše zadovoljavajući, rezultati dobiveni uporabom *Permutation* genotipa pokazuju dosta potencijala za uporabu uz mala poboljšanja. Također, same mogućnosti daljnjeg rada na ovom problemu i poboljšanja su višestruke, te se načini poboljšanja dobivenih rješenja nameću već u ovoj fazi.

Prvi i najočitiji način je uporaba dodatnih genotipa dostupnih u ECF-u, od kojih se posebno ističe često korišteni Binary. Naravno, potrebno je u tom slučaju smisliti dekodiranje rješenja iz odabranog genotipa, no testiranje novog algoritma samog po sebi zahvaljujući donekle ostvarenoj modularnosti sustava ne bi trebao biti problem. Drugi način bi bio kombiniranje više različitih genotipa, gdje bi se npr. *FloatingPoint* genotip koristio za prvu fazu algoritma, grublju pretragu, a zatim bi se dobivena rješenja dodatno evoluirala uporabom npr. *Permutation* genotipa. Ovakav pristup otvara mjesta i uporabi genetskog programiranja za potrebe finijeg pretraživanja i optimizacije. Zadnji način bi bio uporaba dodatnih lokalnih pretraga i determinističkih metoda za poboljšanje dobivenih rješenja, uz oprez da se vrijeme izvođenja algoritma ipak zadrži u nekim prihvatljivim granicama.

Dodatan problem koji treba istaknuti je da ni u jednom pristupu nije osobita pažnja posvećena točnom procesu dizanja rezača, odnosno rezač se dizao svaki put kad je prelazio na novi poligon, kad je u stvarnosti možda mogao radi brzine bez dizanja proći do željene točke (nema poligona između čije rezanje bi mogao upropastititi). Ovaj problem potrebno je detaljnije razmotriti, gdje je dobra početna točka prikazana u postojećim radovima (Dyckhoff, 1990), no za dani problem optimizacije nije bio toliko ključan da bi se na ovoj razini obrađivao.

6. Zaključak

Za problem nalaženja optimalne putanje rezača za procese automatskog rezanja, kao i za ostale probleme iz klase NP-teških problema, genetski algoritmi pokazuju se kao dobar kompromis između nalaženja optimalnog rješenja te brzog izvođenja algoritma. Za ostvareni genetski algoritam ne treba koristiti ni veliki broj jedinki niti veliki broj generacija (iteracija), zahvaljujući čemu se algoritam izvodi relativno brzo u ovisnosti o broju oblika koje je potrebno izrezati.

Rezultati samog rada su dosta zadovoljavajući, a u raspravi su dani i daljnji prijedlozi za poboljšanje trenutnog rješenja koji se mogu testirati. Ipak, problem predstavlja dekodiranje genotipa, odnosno dobivanje putanje iz korištenog prikaza rješenja. Bilo kakva nepažnja pri dekodiranju može uzrokovati konvergenciju prema neiskoristivim rješenjima, kao što je na neki način prikazano u slučaju algoritma koji koristi *FloatingPoint* genotip.

Nažalost, lokalna pretraga baš na onim prihvatljivim rješenjima ne daje bitan doprinos, dok je kod algoritma koji daje lošija rješenja njen dobar doprinos nedovoljan. Ipak, pokazano je da se lokalna pretraga i slični deterministički postupci mogu kombinirati sa genetskim algoritmima kako bi se poboljšalo dobiveno rješenje, te bi se tim pristupom također trebalo dalje baviti.

LITERATURA

Kenneth Castelino. *Tool-path Optimization for Minimizing Airtime during Machining*. Mechanical Engineering Department, University of California at Berkeley, 2002.

Harald Dyckhoff. A typology of cutting and packaging problems. *European Journal of Operational Research*, 1990.

Marin Golub. *Genetski algoritam - prvi dio*. Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu, 2004.

Domagoj Jakobović. *Genetski algoritmi - predavanje*. Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu, 2001.

Pedro Larranaga. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 1999.

Jean Yves Potvin. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research*, 1996.

Optimizacija putanje rezača za postupke automatskog rezanja

Sažetak

Problem automatskog rezanja može se podijeliti na 2 NP-teška problema. Prvi od njih je automatsko gniježđenje, a drugi nalaženje optimalne putanje rezača. Postupak nalaženja optimalne putanje rezača odgovara generaliziranom problemu trgovačkog putnika, gdje je potrebno naći minimalan put takav da se prođe bridovima svih odvojenih poligona. Kako se NP-teški problemi deterministčkim algoritmima ne mogu riješiti u prihvatljivom vremenu koriste se nedeterminističke metode. Jedna od takvih je genetski algoritam, heuristička metoda pretrage koja ne nalazi optimalno, već samo dovoljno dobro rješenje. Svrha ovog rada je pokazati primjenu više različitih genetskih algoritama na problem nalaženja optimalne putanje rezača za postupke automatskog rezanja te provjeriti ponašanje tih algoritama u ovisnosti o korištenim parametrima. Konkretno su implementirana dva različita postupka koja koriste različite prikaze rješenja, te su testirana sa različitim operatorima mutacije, vjerojatnostima mutacije, brojem generacija te brojem jedinki. Dodatno, u oba algoritma implementirani su i deterministički algoritmi lokalne pretrage za poboljšanje rješenja te je predstavljen njihov utjecaj na konačno rješenje algoritma. Predstavljene su ideje za daljnja poboljšanja i istraživanje.

Ključne riječi: genetski algoritam, automatsko rezanje, traženje putanje, lokalna pretraga.

Optimal cutter path generation for automated cutting procedures

Abstract

Automatic cutting problem can be divided into two NP-hard optimization problems. First of those is automatic nesting (cutting stock), and the other is finding the optimal cutter path. Cutter pathfinding corresponds to the generalized travelling salesman problem, where the minimal path which passes through all the edges of the given polygons must be found. Seeing as NP-hard problems cannot be solved in acceptable time using deterministic algorithms, nondeterministic methods are used. One of those is the genetic algorithm, heuristic search method which doesn't find the optimal solution, but only that which is 'good enough'. The goal of this study is to show the application of several genetic algorithms to the problem of finding the optimal cutter path for automated cutting procedures and to check the behavior of these algorithms as their parameters change. More specifically, two different methods were implemented which use different solution representations and were tested with different mutation operators, mutation probabilities, numbers of generations and population sizes. Additionally, deterministic local search algorithms were implemented in both of those algorithms for improving the solutions and their effect on the final solution given by the algorithm was presented. Ideas for further improvement and experimentation were presented.

Keywords: genetic algorithm, cutting stock, automated nesting, pathfinding, local search.