SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO-MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Tihomir Lolić, Matej Mihelčić

NOVA PARALELNA HEURISTIKA ZA RJEŠAVANJE PROBLEMA ISPUNJIVOSTI LOGIČKE FORMULE

Zagreb, 2011.

Ovaj rad izrađen je na Zavodu za numeričku matematiku i računarstvo Prirodoslovno-matematičkog fakulteta, Matematičkog odsjeka Sveučilišta u Zagrebu pod vodstvom Doc. dr. sc. Goranke Nogo i predan je na natječaj za dodjelu Rektorove nagrade u akademskoj godini 2010./2011.

KRATICE

UW-ob	Obični (osnovni) UnitWalk algoritam
UW-mod	Modificirani UnitWalk algoritam
UW+SGAV	Modificirani UnitWalk algoritam poboljšan genetskim algoritmom
UW+gr	UnitWalk algoritam poboljšan pohlepnim (greedy) algoritmom
TripleU	Paralelni solver koji konkurentno izvodi UW-mod, UW+SGAV i UW+gr algoritam

Sadržaj

Uvod

Problem ispunjivosti logičke formule (SAT) je jedan od najpoznatijih i najistraživanijih matematičkih problema. SAT je snažno povezan s matematičkom logikom, teorijom računarstva, softverskim inženjerstvom i operacijskim istraživanjima. Mnogi problemi iz navedenih polja se reduciraju na SAT; na takav reducirani problem možemo primjenjivati mnoge matematičke tehnike koje nam omogućavaju brže i jednostavnije pronalaženje rješenja problema. Problem SAT spada u klasu svih problema koji su rješivi u polinomnom vremenu na nedeterminističkom Turingovom stroju (NP). Još uvijek nije poznato postoji li algoritam koji bi probleme iz NP rješavao u polinomnom vremenu na determinističkom Turingovom stroju.

Martin Davis i Hilary Putnam su 1960. godine konstruirali eksponencijalni potpuni algoritam koji rješava problem SAT ([9]). Taj algoritam je baza svih današnjih potpunih SAT algoritama.

Stohastički algoritmi za rješavanje problema SAT su prvi puta predstavljeni 1992. godine od strane Gu [5] i Selman et ala [16]. Njihov GSAT algoritam je bio znatno učinkovitiji u pronalaženju rješenja kod raznih formula od tadašnje varijante potpunih DPLL algoritama. Nakon godinu dana nastale su brojne varijante pohlepnih algoritama.

Modificiranjem pohlepnog algoritma je dobiven osnovni WalkSAT algoritam ([17]). WalkSAT se empirijski pokazao puno boljim od svih pohlepnih algoritama. Novelty+ je varijanta WalkSAT algoritma razvijena 1998. godine i za nju je pokazano da može rješavati problem ispunjivosti logičke formule s proizvoljno visokom vjerojatnosti. Poboljšanje Novelty+ algoritma, Adaptive Novelty+, razvijen 2002. godine je još uvijek jedan od najučinkovitijih stohastičkih algoritama za rješavanje problema SAT.

U ovom radu predstavljamo tri nova algoritma bazirana na UnitWalk algoritmu od kojih gradimo paralelni heuristički softverski modul. Koristimo genetski algoritam za poboljšavanje početno generiranih interpretacija kao i za povećavanje raznolikosti takvih interpretacija. Promatrat ćemo poboljšanja u odnosu na osnovni UnitWalk algoritam svakog od ta tri algoritma i usporediti njihove performanse s osnovnim UnitWalk algoritmom.

Dokazat ćemo da je naša implementacija vjerojatnosno aproksimativno kompletna, odnosno da

za svaku ispunjivu formulu pronalazi rješenje u konačno mnogo koraka s vjerojatnošću 1.

Poznato je da ne postoji stohastički algoritam koji bi mogao u polinomnom vremenu riješiti problem ispunjivosti za proizvoljno veliku formulu kao i činjenica da su algoritmi najčešće uspješni na određeno strukturiranim formulama dok su na nekim formulama izrazito loši. Jedan od ciljeva u radu je proučiti učinak paralelizacije, odnosno konkurentnog izvođenja tri navedene modifikacije UnitWalk algoritma na formule koje su osnovnom UnitWalk algoritmu bile teško rješive. Statistički gledano, proučavat ćemo utjecaj paralelizacije na količinu rubnih vrijednosti i na njihovu udaljenost od medijana (srednje vrijednosti) na nekom skupu podataka.

Pokušat ćemo utvrditi koliko teške formule može riješiti naš softverski modul. Vjeruje se da su najteže instance formula za 3-SAT formule koje imaju u prosjeku 4.2 klauzule za svaku varijablu u formuli ([21]). Mi ćemo taj faktor proširivati, odnosno povećavati broj klauzula za velike formule (formule s više od 250 varijabli) dok ne dobijemo formule koje će modul učinkovito riješiti.

U prvom odjeljku rada navodimo osnovne definicije i tvrdnje potrebne za razumijevanje problema koji rješavamo, strukture i tehnike korištene u našim algoritmima za pojednostavljivanje i brže rješavanje problema.

U drugom odjeljku objašnjavamo principe rada korištenih algoritama, navodimo njihova, ograničenja i mane.

U trećem odjeljku detaljno opisujemo UnitWalk algoritam na čijim principima gradimo sva tri nova algoritma.

U četvrtom odjeljku opisujemo strukturu i funkcionalnost našega softverskog modula.

U petom odjeljku analiziramo i uspoređujemo performanse naših algoritama s osnovnim UnitWalk algoritmom.

1 Uvodne definicije i tvrdnje

1.1 Osnovne tvrdnje logike sudova

U ovom odjeljku rada navesti ćemo osnovne tvrdnje logike sudova nužne za razumijevanje problema.

Definicija 1.1. ***Alfabet logike sudova*** *je unija skupova A_1, A_2 i A_3, pri čemu je:*

$$A_1 = \{P_0,\ P_1,\ P_2,\ \dots\}$$

prebrojiv skup čije elemente nazivamo propozicionalne varijable;

$$A_2 = \{\neg\ ,\wedge\ ,\vee\ \rightarrow\ ,\leftarrow\}$$

skup logičkih veznika;

$$A_3 = \{(,)\}$$

skup pomoćnih simbola (zagrade);

Definicija 1.2. ***Atomarna formula*** *je svaka propozicionalna varijabla. Pojam* ***formule*** *definiramo induktivno:*

1. *svaka atomarna formula je formula;*
2. *ako su A i B formule tada su i riječi $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ i $(A \leftrightarrow B)$ također formule;*
3. *riječ alfabeta logike sudova je formula ako je nastala primjenom konačno mnogo koraka 1. i 2.*

Definicija 1.3. *Svako preslikavanje sa skupa propozicionalnih varijabli u skup {0,1}, tj. $I\colon \{P_0, P_1 \dots\} \rightarrow \{0,1\}$, nazivamo* ***totalna interpretacija*** *ili kratko* ***interpretacija****. Ako je preslikavanje definirano na podskupu skupa propozicionalnih varijabli tada kažemo da je to* ***parcijalna interpretacija****. Kažemo da je parcijalna interpretacija I* ***adekvatna*** *za formulu $A(P_1, \dots,\ P_n)$ ako je funkcija I definirana na P_i za sve $i = 1, \dots, n$.*

Definicija 1.4. *Neka je I interpretacija (totalna ili parcijalna). Ako se radi o parcijalnoj interpretaciji I smatramo da je I adekvatna za formule na kojima se definira njena vrijednost. Tada vrijednost interpretacije I na proizvoljnoj formuli definiramo induktivno:*

$$I(\neg A) = 1 \text{ ako i samo ako } I(A) = 0;$$

$$I(A \wedge B) = 1 \text{ ako i samo ako } I(A) = 1 \text{ i } I(B) = 1;$$

$$I(A \vee B) = 1 \text{ ako i samo ako } I(A) = 1 \text{ ili } I(B) = 1;$$

$$I(A \to B) = 1 \text{ ako i samo ako } I(A) = 0 \text{ ili } I(B) = 1;$$

$$I(A \leftrightarrow B) = 1 \text{ ako i samo ako } I(A) = I(B).$$

Ako alfabet sadrži i konstante $\top$ i $\bot$, tada za svaku interpretaciju I definiramo $I(\top) = 1$ i $I(\bot) = 0$.

Definicija 1.5. *Neka je S skup formula, a F neka formula. Kažemo da formula F* ***logički slijedi*** *iz skupa S, u oznaci $S \models F$, ako za svaku interpretaciju I, za koju je $I(S) = 1$, vrijedi $I(F) = 1$. Relaciju $\models$ nazivamo* ***relacija logičke posljedice****. Ako je S jednočlan skup, tj. $S = \{A\}$, tada činjenicu $\{A\} \models B$ zapisujemo i kao $A \Rightarrow B$.*

Definicija 1.6. *Kažemo da su formule A i B* ***logički ekvivalentne*** *i označavamo $A \Leftrightarrow B$ ako za svaku interpretaciju I vrijedi $I(A) = I(B)$.*

Definicija 1.7. *Za formulu F kažemo da je* ***ispunjiva*** *ako postoji interpretacija I takva da vrijedi $I(F) = 1$.*

Za formulu F kažemo da je ***oboriva*** *ako postoji interpretacija I takva da vrijedi $I(F) = 0$.*

Za formulu F kažemo da je ***valjana (tautologija, identički istinita)*** *ako je istinita za svaku interpretaciju.*

Za formulu F kažemo da je ***antitautologija*** *ili* ***identički neistinita*** *ako je neistinita za svaku interpretaciju.*

Definicija 1.8. ***Klauzula*** *je formula oblika $l_1 \vee l_2 \vee \cdots \vee l_k$ gdje je l_j, $1 \leq j \leq k$ literal.*

Definicija 1.9. *Atomarnu formulu i njezinu negaciju nazivamo **literal**. S $|l|$ označavamo propozicionalnu varijablu pridruženu literalu l.*
***Predznak (polarnost)** literala l, $sgn(l)$, je 1 (pozitivna) ako je l varijabla, a 0 (negativna) ako je l negacija varijable. Sa $\bar{l}$ označavamo literal dualan literalu l.*

1.2 Normalne forme

Svi algoritmi koji rješavaju problem ispunjivosti logičkih formula kao ulazni parametar primaju formulu napisanu u konjunktivnoj normalnoj formi. Takav oblik formule omogućuje brojne optimizacije koje ubrzavaju nalaženje rješenja. U nastavku ćemo definirati normalne forme i dati osnovne rezultate koji pokazuju da svaka formula ima logički ekvivalentnu normalnu formu.

Definicija 1.10. *Formulu oblika $A_1 \wedge A_2 \wedge \cdots \wedge A_n$ nazivamo konjunkcija (A_i su proizvoljne formule). Formulu oblika $A_1 \vee A_2 \vee \cdots \vee A_n$ nazivamo disjunkcija. **Elementarna konjunkcija** je konjunkcija literala, a **elementarna disjunkcija** je disjunkcija literala. **Konjunktivna normalna forma** je konjunkcija elementarnih disjunkcija. **Disjunktivna normalna forma** je disjunkcija elementarnih konjunkcija.*

Neka je A neka formula, te B konjunktivna normalna forma i C disjunktivna normalna forma. Kažemo da je B konjunktivna normalna forma za A ako vrijedi $A \Leftrightarrow B$. Kažemo da je C disjunktivna normalna forma za A ako vrijedi $A \Leftrightarrow C$.

Nama je važna činjenica da za svaku formulu logike sudova postoji konjunktivna normalna forma, pa ćemo nju i dokazati ([19]).

Teorem 1.11. *Za svaku formulu logike sudova postoje konjunktivna i disjunktivna normalna forma.*

Dokaz. Dokazat ćemo da za proizvoljnu formulu postoji konjunktivna normalna forma. Dokaz egzistencije disjunktivne normalne forme je sličan. Ako je A valjana formula tada je npr. formula $(P \vee \neg P) \wedge (P \vee \neg P)$ konjunktivna normalna forma za A. Promotrimo sada slučaj kada je formula $A(P_1, \ldots, P_n)$ oboriva. Neka su $I_1, \ldots, I_m$ sve parcijalne interpretacije, čija je domena $\{P_1, \ldots, P_n\}$,

takve da za njih vrijedi $I_1(A) = \cdots = I_m(A) = 0$. Za sve $i \in \{1, \ldots, m\}$ i sve $j \in \{1, \ldots, n\}$ definiramo literale P_{ij} ovako:

$$P_{ij} \equiv \begin{cases} \neg P_j, \text{ ako je } I_i(P_j) = 1; \\ \\ P_j, \text{ ako je } I_i(P_j) = 0. \end{cases} \tag{1}$$

Uočite da vrijedi $I_i(P_{ij}) = 0$. Neka je sada formula B definirana kao

$$(P_{11} \wedge \cdots \wedge P_{1n}) \vee \cdots \vee (P_{m1} \wedge \cdots \wedge P_{mn}),$$

što ćemo kraće zapisati s

$$B \equiv \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n} P_{ij}.$$

Očito je B konjunktivna normalna forma. Preostalo je još dokazati da vrijedi $A \Leftrightarrow B$. Prvo ćemo dokazati da vrijedi $A \Rightarrow B$. Neka je I interpretacija takva da $I(B) = 0$. Pošto je B konjunkcija tada postoji $i \in \{1, \ldots, m\}$ tako da vrijedi $I(P_{i1} \wedge \cdots \wedge P_{in} = 0)$. Tada dalje imamo $I(P_{ij}) = 0$, za sve $j \in \{1, \ldots, n\}$. No, tada je $I/\{P_1, \ldots, P_n\} = I_i$. To znači da je $I(A) = I_i(A) = 0$. Dokažimo još da vrijedi $B \Rightarrow A$. Neka je I interpretacija takva da je $I(A) = 0$. Tada postoji $k \in \{1, \ldots, m\}$ takav da vrijedi $I/\{P_1, \ldots, P_n\} = I_k$. Promotrimo konstrukciju literala P_{ij} za $i = k$. Uočimo da vrijedi $I_k(P_{kj}) = 0$ za sve $j \in \{1, \ldots, n\}$. To posebno znači da vrijedi $I_k(\bigvee_{j=1}^{n} P_{kj}) = 0$ pa je onda i $I_k(\bigwedge_{i=1}^{m} \bigvee_{j=1}^{n} P_{ij}) = 0$ tj. $I(B) = I_k(B) = 0$.

□

Prethodni teorem je jako važan. Garantira nam da možemo svaku formulu pretvoriti u ekvivalentnu konjunktivnu normalnu formu s kojom će naš algoritam raditi. Međutim ostavlja nam preveliki izbor zato što za svaku formulu možemo generirati beskonačno mnogo konjunktivnih i disjunktivnih normalnih formi. Pošto se radi o algoritmima, želimo da nam ulazni podaci (KNF forme) budu donekle jedinstveni.

Definicija 1.12. *Neka je $A(P_1, \ldots, P_n)$ konjunktivna normalna forma. Kažemo da je to* ***savršena konjunktivna normalna forma*** *ako u svakoj njezinoj elementarnoj disjunkciji svaka propozicionalna*

varijabla P_i nastupa točno jednom (s ili bez negacije), te su sve elementarne disjunkcije međusobno logički neekvivalentne.
Neka je $A(P_1, \ldots, P_n)$ disjunktivna normalna forma. Kažemo da je to ***savršena disjunktivna normalna forma*** *ako u svakoj njezinoj elementarnoj konjunkciji svaka propozicionalna varijabla P_i nastupa točno jednom (s ili bez negacije), te su sve elementarne konjunkcije međusobno logički neekvivalentne.*

Prethodna definicija nam daje željeni rezultat ([19]).

Korolar 1.13. *Za svaku oborivu formulu postoji savršena konjunktivna normalna forma. Za svaku ispunjivu formulu postoji savršena disjunktivna normalna forma. Savršene forme su jedinstvene do na permutaciju varijabli u elementarnim disjunkcijama, odnosno konjunkcijama, te do na permutaciju elementarnih konjunkcija, odnosno disjunkcija.*

Dokaz. Pažljivim čitanjem dokaza teorema 1.11 se vidi da je u njemu dana egzistencija savršene konjunktivne normalne forme za oborivu formulu i egzistencija savršene disjunktivne normalne forme za ispunjivu formulu.
Promotrimo sljedeće formule:
$(P \vee \neg P) \wedge (Q \vee \neg Q)$ i $(P \wedge \neg P) \vee (Q \wedge \neg Q)$. Prva formula je tautologija, a druga antitautologija. Za njih ne postoji savršena konjunktivna odnosno savršena disjunktivna normalna forma. □

Navodimo KNF algoritam koji pokazuje kako proizvoljnu formulu svodimo na konjunktivnu normalnu formu ([7]).

Ulaz: Proizvoljna formula F.

1. Sve podformule formule F oblika $F_1 \to F_2$ zamijenimo s $(\neg F_1 \vee F_2)$ i sve podformule oblika $F_1 \leftrightarrow F_2$ s $(\neg F_1 \vee F_2) \wedge (\neg F_2 \vee F_1)$. Postupak završavamo kada u F nema više podformula oblika $A \to B$ i $A \leftrightarrow B$ i prelazimo na drugi korak.

2. Rješavamo se svih dvostrukih negacija i primjenjujemo De Morganove zakone:

$\neg\neg F_1$ zamijenimo s F_1,

$\neg(F_1 \wedge F_2)$ s $(\neg F_1 \vee \neg F_2)$,

$\neg(F_1 \vee F_2)$ s $(\neg F_1 \wedge \neg F_2)$.

Kada u formuli F nema više podformula takvih oblika prelazimo na treći korak.

3. Primjenjujemo svojstvo distributivnosti za $\wedge$ gdje je moguće tako da zamijenimo sve podformule oblika $(F_1 \vee (F_2 \wedge F_3))$ ili $((F_2 \wedge F_3) \vee F_1)$ s $((F_1 \vee F_2) \wedge (F_1 \vee F_3))$.

Izlaz: Logički ekvivalentna formula u konjunktivnoj normalnoj formi.

Primjer 1.14. *Neka je dana formula* $(\neg(F_1 \to F_2) \to \neg(F_3 \vee \neg F_4)) \wedge (\neg(F_5 \wedge F_6) \to F_7)$.
Sada primjenom pravila 1. na podformule $F_1 \to F_2$ *i* $\neg(F_5 \wedge F_6) \to F_7$ *dobivamo formulu* $(\neg(\neg F_1 \vee F_2) \to \neg(F_3 \vee \neg F_4)) \wedge (\neg\neg(F_5 \wedge F_6) \vee F_7)$.
Ponovno primijenimo pravilo 1. na podformulu $\neg(\neg F_1 \vee F_2) \to \neg(F_3 \vee \neg F_4)$ *pa dobivamo* $(\neg\neg(\neg F_1 \vee F_2) \vee \neg(F_3 \vee \neg F_4)) \wedge (\neg\neg(F_5 \wedge F_6) \vee F_7)$.
Prelazimo na 2. pravilo jer više nema podformula oblika $P \to Q$ *niti* $P \leftrightarrow Q$. *Nakon obavljenih zamjena dobivamo formulu* $((\neg F_1 \vee F_2) \vee (\neg F_3 \wedge F_4)) \wedge ((F_5 \wedge F_6) \vee F_7)$. *Preostaje još primijeniti pravilo 3. te dobivamo* $((\neg F_1 \vee F_2) \vee (\neg F_3 \wedge F_4)) \wedge ((F_5 \vee F_7) \wedge (F_6 \vee F_7))$.

Navedeni algoritam je nažalost eksponencijalne složenosti stoga nam nije od praktične koristi. Na sreću postoje i algoritmi polinomne složenosti za pretvaranje proizvoljne formule logike sudova u konjunktivnu normalnu formu ([2], str. 32-33).

U nastavku pretpostavljamo da je zadana formula u savršenoj konjunktivnoj normalnoj formi.

1.3 Problem ispunjivosti logičke formule

Problem ispunjivosti logičke formule (SAT) glasi:

Postoji li za proizvoljnu formulu F u konjunktivnoj normalnoj formi interpretacija $I\colon \{x_1, \ldots, x_n\} \to \{0, 1\}$ takva da vrijedi $I(F) = 1$? $x_1, \ldots, x_n$ su sve propozicionalne varijable koje se javljaju u formuli F.

Postoje razne vrste SAT problema, no mi ćemo se ograničiti na $k - SAT$, odnosno njegovu specifičniju varijantu $3 - SAT$.

Definicija 1.15. *Za formulu F logike sudova kažemo da je k-KNF formula, ako je ona u konjunktivnoj normalnoj formi, sadrži više od jedne klauzule i svaka njezina klauzula sadrži točno k literala.*

Definicija 1.16. *k-SAT*= $\{F|F$ *je ispunjiva k-KNF formula logike sudova.*$\}$.

Problem SAT ima jedno jako važno svojstvo:

Teorem 1.17. *(Cook-Levin)*
SAT *je NP-potpun.*

Ovaj teorem nam govori da ukoliko uspijemo pronaći polinomni algoritam koji rješava problem SAT, tada smo uspjeli riješiti sve probleme iz NP u polinomnom vremenu, pošto se oni mogu polinomno reducirati na SAT. Ovaj teorem nećemo dokazivati, dokaz se može pogledati u [12], str. 171-172.

Još jedan za nas važan rezultat je da je dovoljno pronaći polinomni algoritam za rješavanje $3 - SAT$ problema da bi mogli riješiti sve probleme iz NP u polinomnom vremenu ([18], str. 282-283).

Teorem 1.18. $3 - SAT$ *je NP-potpun.*

2 Heuristički algoritmi

Pošto je gornja ograda trenutno poznatih algoritama za rješavanje problema SAT prevelika za praktične primjene detaljno se istražuju i heuristički algoritmi. Heuristički algoritmi su dosta teški

za teorijsku analizu, no pokazuju dobre rezultate u praksi, kako na slučajno generiranim formulama tako i na strukturiranim instancama koje su dobivene pretvaranjem raznih problema kao što su planiranje ili dizajn mreža u problem ispunjivosti logičke formule. Teoretsko znanje o ovoj skupini algoritama je maleno i većinom se analizom složenosti dobivaju eksponencijalne gornje ograde izvršavanja.

Heuristički algoritmi mogu biti:

1. **Potpuni:** Daju rješenje problema s apsolutnom sigurnošću, ako ne uspiju pronaći rješenje za zadani problem tada ono ne postoji.

2. **Nepotpuni:** Ako nepotpuni algoritam pronađe rješenje ono je sigurno točno, međutim ako naš nepotpuni algoritam ne pronađe rješenje tada ne znamo ništa o rješenju niti o njegovoj egzistenciji.

Primijetimo da nemamo ogradu na vjerojatnost pogreške, stoga ne možemo samo ponavljati algoritam dovoljan broj puta smanjujući vjerojatnost pogreške na neku predefiniranu konstantu.

Nepotpuni heuristički algoritmi se mogu podijeliti na:

1. **Esencijalno nepotpune:** Kod algoritama ove skupine postoje slučajno generirani elementi prostora koji pretražujemo iz kojih algoritam nikako neće poboljšavanjem uspjeti doći do konačnog rješenja. Vjerojatnost da će algoritam uspjeti doći do rješenja iz takvog elementa je strogo manja od 1. Kod esencijalno nepotpunih algoritama se uvode ponovna pokretanja, odnosno **restartovi**. Ako u određenom broju iteracija ne uspijemo doći do rješenja, na slučajan način generiramo novi početni element kojeg pokušamo dalje poboljšati.

2. **Vjerojatnosno aproksimativno potpune:** Algoritmi iz ove skupine će bez obzira na element iz kojeg kreću u pretraživanje prostora rješenja doći sigurno do rješenja, ako takvo postoji, u određenom (konačnom) broju koraka.

Trenutno poznatiji heuristički algoritmi za rješavanje problema SAT: GSAT, GWSAT, HSAT, HWSAT, SDF, IDB, WalkSAT i njegove podvrste: WalkSAT/Tabu, Novelty, R-Novelty, Novelty+,

R-Novelty+. Prisutni su i GA (genetski) algoritmi koji se najčešće kombiniraju s jednim od algoritama lokalnog traženja.
U zadnjih nekoliko godina se radi i na razvijanju algoritama koji bi rješavali problem SAT na kvantnim računalima.

2.1 Genetski algoritmi

Genetski algoritmi su evolucijske meta-heurstike koje se koriste za rješavanje teških problema. Njihova primjena na kompleksne optimizacijske probleme u nekim slučajevima daje izuzetno dobre rezultate. Ideja algoritma je bazirana na oponašanju prirodne evolucije. Algoritam kreće od početne populacije koju čine kandidati za rješenja koji predstavljaju jedinke koje će se poboljšavati te na taj način povećavati kvalitetu populacije. Princip samog algoritma se sastoji od niza reprodukcija između jedinki, mutacije jedinke dobivene reprodukcijom te na kraju odabirom bolje jedinke koja će ući u populaciju. Ovaj proces se ponavlja konačan broj puta i cilj je stvaranje što kvalitetnije jedinke.

Genetski algoritam često konvergira što znači da populacija gubi na svojoj raznovrsnosti pa algoritam gubi svoju efikasnost jer mu se može dogoditi da lako zaglavi te na taj način nikada neće pronaći rješenje. Samu konvergenciju ponekad možemo koristiti kao zaustavni kriterij. Veliki problem predstavlja preuranjena konvergencija koja je svojstvena za klasične genetske algoritme. Ovo svojstvo onemogućava algoritmu pretraživanje većeg dijela domene samog problema.

Pokazuje se da korištenje klasičnog genetskog algoritma za MAX-SAT daje relativno loše performanse ([3]).

Križanje i mutacija se mogu izvesti na različite načine. Opišimo ukratko neke od načina.

1. Operator križanja s jednom točkom prekida: na slučajan način odaberemo točku prekida na kromosomu.[1] Dio kromosoma koji ide od početka do točke prekida se uzima od prvog roditelja, dok se ostatak kopira od drugog roditelja. Promotrimo kako to izgleda na binarnom

[1] Kromosom je kodirani prikaz jedinke koji ovisi o problemu

stringu:

$$\mathbf{1010}|1011 + 1111|\mathbf{1001} = \mathbf{1010}|\mathbf{1001}$$

2. Operator križanja s dvije točke prekida: na slučajan način odaberemo dvije točke prekida na kromosomu te napravimo razmjenu roditeljskih kromosoma između tih točaka i na taj način dobivamo novu jedinku. Na binarnom stringu bi to značilo da dio od početka kromosoma pa do prve točke prekida i dio od druge točke prekida pa do kraja stringa uzimamo od prvog roditelja, dok ono što se nalazi između točaka prekida uzimamo od drugog roditelja.

$$\mathbf{11}|0011|\mathbf{01} + 10|\mathbf{0101}|00 = \mathbf{11}|\mathbf{0101}|\mathbf{01}$$

3. Operator uniformnog križanja: na slučajan način određuje doprinos pojedinog gena u novoj jedinki tako što ga uzima od jednog roditelja. Za binarni string to znači odabir bitova koji se na slučajan način uzimaju od nekog roditelja.

$$111010100|001100111 = \mathbf{111100101}$$

4. Operator aritmetičkog križanja: nova jedinka nastaje primjenom nekog aritmetičkog operatora na genima. Za binarni string to može biti logičko **ILI**.

$$1101010001 + 0101110011 = \mathbf{1101110011}$$

5. Operator križanja s dvije točke prekida i parametrom d(d-fold): na slučajan način odaberemo dvije točke prekida te potom uzimamo d lijevih i d desnih bitova od točke prekida jednog roditelja i to zamijenimo s odgovarajućim bitovima drugog roditelja. Na opisani način dobivamo dvije nove jedinke koje se razlikuju za najviše $4d$ bitova ([4]). Slijedi primjer za $d = 2$.

$$1\mathbf{01}|\mathbf{10}1\mathbf{01}|\mathbf{11}1 + 0\mathbf{11}|\mathbf{10}0\mathbf{11}|\mathbf{10}1 = \{11110111101, 00110001111\}$$

6. Operator mutacije: na binarnom stringu to predstavlja promjenu na slučajan način odabranog bita.

$$101011111 \rightarrow 1010\mathbf{0}1111$$

Reprezentacija jedinki

Jedinka je reprezentirana vektorom X duljine n čiji elementi su nule i jedinice. Svaka komponenta vektora ima vrijednost 0 (Laž) ili 1 (Istina). Kvaliteta jedinke (*fitness*) se mjeri brojem klauzula koje su istinite za danu jedinku. Cilj je maksimizirati broj istinitih klauzula. Jedinka koja ispunjava najviše klauzula odgovara rješenju.

Funkcija raznovrsnosti

Udaljenost jedinke od skupa B se naziva raznovrsnost jedinke. Može se definirati kao minimalna vrijednost Hammingove udaljenosti između jedinke i svih ostalih jedinki iz skupa B. Hammingova udaljenost je broj različitih bitova između dvije jedinke.

Primjer 2.1. *Udaljenost između* $X_1 = 0100$ *i* $X_2 = 0010$ *je jednaka* 2.

Način odabira populacije

Kod klasičnog genetskog algoritma imamo samo na slučajan način generiranje populacije. U ovoj verziji algoritma, osim generiranja, slijedi i odabir određenih jedinki koji se temelji na raznovrsnosti i kvaliteti generiranih jedinki. Jedinke se biraju na sljedeći način:

1. Prvo odabiremo skup od $|B_1|$ najboljih jedinki iz trenutne populacije P. Skup B je na početku jednak skupu B_1 najboljih jedinki.

2. Za svaku jedinku V iz ostatka populacije tj. iz $P - B$ računamo udaljenost do trenutnog skupa B. Minimalna vrijednost Hammingove udaljenosti između jedinke V i jedinki iz B predstavlja raznovrsnost jedinke V.

3. Iz skupa $P - B$ odabiremo $|B_2|$ jedinki, onih koje imaju najveću raznovrsnost.
   Uočimo da jedinka koja treba biti dodana u trenutni skup B je raznovrsna s obzirom na skup jedinki koje pripadaju skupovima B_1 i B_2.

Stohastičko lokalno pretraživanje

Kako bi pretražili najvažniji dio prostora potencijalnih rješenja opisujemo tehniku temeljenu na Walkast ([3]) principu kombiniranu s raznovrsnosti, determinizmu i slučajnom odabiru što je korisno prilikom obilaska potencijalnih rješenja.

Počinjemo s rješenjem V. Provodimo određeni broj koraka koji se sastoje od odabira varijable te promjene njene vrijednosti. U svakom koraku varijabla čija se vrijednost mijenja se odabire na jedan od sljedeća tri načina:

1. Prvi način je da se varijabla *var* čiju vrijednost mijenjamo odabire na slučajan način uz fiksnu vjerojatnost $wp > 0$.

2. Drugi način je da na slučajan način odabiremo klauzulu *cl* iz skupa nezadovoljenih klauzula iz koje na slučajan način odabiremo varijablu *var* s fiksnom vjerojatnošću dp tako da $dp > wp$.

3. Treći način se sastoji od odabira najbolje varijable tj. one varijable čija promjena vrijednosti će davati najbolju interpretaciju.

Sada navodimo pseudokod SLS(stohastical linear search) algoritma.

```
Algoritam SLS.
Ulaz: formula F u KNF, interpretacija V, Max_Flips, dp, wp.
Izlaz: Poboljšana interpretacija V.

for i = 0 to Max_Flips(F)

   Generiraj slučajan broj r između 0 i 1
    if r < wp then
          Promijeni vrijednost na slučajan način
          odabrane varijable iz V
```

```
    else if r < dp then
            Na slučajan način odaberi nezadovoljenu klauzulu cl
               Promijeni vrijednost varijable iz V koja je na
               slučajan način odabrana iz klauzule cl
   else
            Promijeni vrijednost najbolje varijable iz V
           end for
  return interpretacija V
```

Sada navodimo pseudokod SGAV algoritma.

```
Algoritam SGAV.
Ulaz: formula F u KNF.
Izlaz: Najbolja interpretacija I tj. interpretacija koja
       zadovoljava najviše klauzula.

 Na slučajan način generiraj inicijalnu populaciju P
 for i = 1 to MAX_TRIES(F)
             Izaberi listu kandidata od B jedinki iz P
             while nije dostignut maksimalan broj generacija do
               while nije ispitan maksimalan broj kombinacija
                      roditelja do
                 Odaberi dvije jedinke iz B
                 Generiraj slučajan broj r između 0 i 1
                             if r < vjerojatnost križanja then
                   Primjeni operator uniformnog križanja
```

```
                da dobiješ novu jedinku V
                Primjeni operator SLS na jedinku V
                if V poboljšava kvalitetu od B  then
                  Dodaj V u B_1
                  Izbaci iz B najlošiju jedinku
                else if V poboljšava raznovrsnost od B then
                  Dodaj V u B_2
                  Izbaci iz B najmanje raznovrsnu jedinku
                end if
              end if
            end while
          end while
          Na slučajan način generiraj novu populaciju
        end for
return najbolja pronađena jedinka V
```

Prednosti algoritma: Genetski algoritmi su populacijski algoritmi koji omogućavaju rješavanje teških problema.

Mane algoritma: Mogu zaglaviti u lokalnom optimumu.

Primjer 2.2. *Demonstrirat ćemo rad SGAV algoritma na formuli:*
$F = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_2 \vee x_4 \vee x_5) \wedge (x_2 \vee x_4 \vee x_5)$.

Budući da se algoritam odvija u nizu iteracija njegov rad ćemo opisati samo u jednoj iteraciji i to u prvoj kad dolazi do inicijalizacije populacije. Pretpostavimo da je početno $velicinaPopulacije = 6$, $MAX - tries = 2$, $velicinaB1 = 2$, $velicinaB2 = 2$, $r = 0.5$, $maxFlips = 3$, $dp = 0.7$, $wp = 0.4$, $d = 1$. *Varijable MaxGeneration i MaxCombination nam nisu bitne jer opisujemo samo jednu iteraciju algoritma. Zbog lakše vizualizacije, interpretaciju ćemo reprezentirati kao string* 0 *i* 1

takav da bit na poziciji i u stringu predstavlja trenutnu istinitosnu vrijednost varijable x_i, pri čemu indeksiranje kreće od 1.

+ *Neka je* $P = 10100, 11000, 10111, 00011, 10101, 10001$ *slučajno generirana populacija interpretacija.*

  - *Prvo moramo izračunati koje su dvije najbolje interpretacije tj. one koje zadovoljavaju najviše klauzula. Sve interpretacije osim prve zadovoljavaju po tri klauzule dok prva zadovoljava dvije. Stoga odabiremo prve dvije interpretacije koje zadovoljavaju po tri klauzule i na taj način inicijaliziramo skupove B odnosno* B_1 *tj. imamo* $B = B_1 = \{11000, 10111\}$
  - *Sada računamo koje jedinke od ostatka populacije se najviše razlikuju s obzirom na B. Za svaku jedinku iz skupa* $P - B$ *računamo Hammingovu udaljenost od skupa B i dobivamo da jedinka* 10101 *ima Hammingovu udaljenost jedan, dok ostale imaju Hammingovu udaljenost dva. Sada dodajemo prve dvije najraznolikije jedinke u skup B, a to su* 10100 *i* 00011.
  - *Nakon generiranja skupa B može početi križanje. Na slučajan način odabiremo dvije jedinke iz skupa B. Pretpostavimo da smo slučajnim odabirom odabrali upravo jedinke* 11000 *i* 00011 *i da je na slučajan način generiran broj p čija je vrijednost* 0.42 *te će stoga doći do križanja.*
  - *Sada na slučajan način odabiremo točke križanja. Pretpostavimo da su to točke jedan i tri. Primjenom operatora* $Cross\_d$ *dobit ćemo dvije jedinke od kojih biramo bolju. Križanjem dobivamo jedinke* 00000 *i* 10111. *Obje jedinke zadovoljavaju po tri klauzule pa je svejedno koju uzeti, stoga uzmimo npr. prvu tj. jedinku* 00000.
  - *Sada na dobivenu jedinku primijenimo operator SLS. Parametar* $maxFlips$ *nam govori koliko iteracija će biti, konkretno u primjeru je odabrano* 3 *da ilustriramo sve mogućnosti mijenjanja vrijednosti. U prvom prolazu je generiran slučaja broj* $rand = 0.38$. *Kako je* $rand < wp$ *to znači da radimo promjenu vrijednosti varijable koja je odabrana na slučajan način. Neka je napravljena zamjena varijable* x_3 *čime dobivamo*

interpretaciju 00100*. Neka je u drugom prolazu generiran slučajan broj rand* = 0.643*. Tada radimo promjenu vrijednosti na slučajan način odabrane varijable iz na slučajan način odabrane nezadovoljene klauzule. Kako je samo jedna nezadovoljena klauzula za interpretaciju* 00100 *to znači da biramo varijablu iz klauzule* $(x_2 \vee x_4 \vee x_5)$*. Neka je npr. odabrana varijabla* x_2*. Sada je naša interpretacija* 01100 *čime je gotova druga iteracija. U trećoj iteraciji neka je na slučajan način generiran broj rand* = 0.88*. Sada je potrebno odrediti koju varijablu je najbolje promijeniti da dobijemo poboljšanje. Lako se vidi da promjenom vrijednosti od varijabli* x_4 *ili* x_5 *dobivamo maksimalan broj zadovoljenih klauzula. Budući da kod provjere koja je promjena najbolja idemo redom od* x_1 *prema* x_5*, odabiremo promjenu varijable* x_4*. Ovom promjenom SLS završava i njegova povratna vrijednost je interpretacija* 01110*.*

- *Sada slijedi provjera poboljšava li dana interpretacija skup B. Ona ga poboljšava i stoga iz B trebamo izbaciti najlošiju jedinku te u* B_1 *dodati novodobivenu jedinku. Najlošija jedinka je* 10100 *pa nju izbacujemo čime smo došli do kraja iteracije.*

2.2 Pohlepni algoritmi

Hill climbing je matematička tehnika optimizacije koja pripada u tehnike lokalnog traženja. To je iterativni algoritam koji počinje pretragu s proizvoljnim rješenjem problema. Algoritam u svakom koraku pokušava pronaći bolje rješenje inkrementalno mijenjajući jedan element trenutnoga. Ako promjena stvori bolje rješenje, ono se prihvaća kao trenutno. Postupak se ponavlja sve dok algoritam pronalazi bolja rješenja.

Greedy algoritmi za rješavanje problema SAT u oznaci (GSAT) su slučajni hill-climbing algoritmi. GSAT na slučajan način generira interpretaciju te zatim provodi postupak hill-climbinga. Algoritam mijenja vrijednost interpretacije za onu varijablu za koju se dobije najveće povećanje u broju istinitih klauzula u odnosu na prethodni korak. Ukoliko moramo izabrati između dvije jednako dobre promjene, greedy algoritam na slučajan način izabere jednu od njih. Ako niti jedna promjena ne može popraviti rezultat tada se uzima ona promjena koja najmanje smanjuje broj kla-

uzula koje su istinite za trenutnu interpretaciju. GSAT počinje od slučajne pozicije u prostoru svih rješenja i traži globalno rješenje koristeći samo lokalnu informaciju.

Pseudokod osnovnog greedy algoritma za rješavanje problema SAT ([1]):

```
Algoritam GSAT.
Ulaz: formula F u KNF.
Izlaz: Interpretacija I takva da I(F) = 1 ili rješenje nije pronađeno.

 for i = 1 to Max-tries
             T=slučajno generirana interpretacija
              for j = 1 to Max-flips
                    if T(F)=1 return T.
                    else
                            Poss-flips:=skup koji sadrži one
                            varijable na kojima promjena
                            vrijednosti interpretacije donosi
                            najveći rast istinitih klauzula
                            formule F.
                            V= slučajni element iz skupa
                               Poss-flips.
                            T=T s promijenjenom vrijednošću na V.

             end

 end

 return Rješenje nije pronađeno!
```

Prednosti algoritma: Greedy algoritmi su najjednostavniji i vrlo brzi algoritmi lokalnog pretraživanja.

Mane algoritma: Često zaglave u lokalnom optimumu.
Esencijalno nepotpuni algoritmi.

Primjer 2.3. *Demonstrirat ćemo rad greedy algoritma na formuli:*
$F = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_2 \vee x_4 \vee x_5) \wedge (x_2 \vee x_4 \vee x_5)$.

Pretpostavimo da je početno $MAX - tries = 2$, $MAX - flips = 5$. *Zbog lakše vizualizacije, interpretaciju ćemo reprezentirati kao string* 0 *i* 1 *takav da bit na poziciji i u stringu predstavlja trenutnu istinitosnu vrijednost varijable* x_i.

+ *Neka je* $I = 10100$ *slučajno generirana interpretacija.*

  - *Računamo za koje varijable bi promjena istinitosti dovela do najvećeg porasta u broju istinitih klauzula formule, označimo taj broj s* $C(x_i)$. $C(x_1) = 1 - 0 = 1$, $C(x_2) = 2 - 1 = 1$, $C(x_3) = 1 - 1 = 0$, $C(x_4) = 1 - 0 = 1$, $C(x_5) = 1 - 0 = 1$, *stoga je* $Pos - flips = \{x_1, x_2, x_4, x_5\}$.

    Pretpostavimo da smo slučajnim odabirom promijenili vrijednost varijabli x_1, *odnosno* $I = 00100$.
  - *Računamo za koje varijable bi promjena istinitosti dovela do najvećeg porasta u broju istinitih klauzula formule.* $C(x_1) = 0 - 1 = -1$, $C(x_2) = 1 - 1 = 0$, $C(x_3) = 0 - 0 = 0$, $C(x_4) = 1 - 0 = 1$, $C(x_5) = 1 - 0 = 1$, *stoga je* $Pos - flips = \{x_4, x_5\}$.

    Pretpostavimo da smo slučajnim odabirom promijenili vrijednost varijabli x_5, *odnosno* $I = 00101$.

    Pošto vrijedi $I(F) = 1$, *algoritam vraća interpretaciju I kao rješenje.*

2.3 Slučajne šetnje

Riječ je o algoritmima lokalnog pretraživanja. Umjesto da pokušavamo pronaći globalno najbolju varijablu, prvo na slučajan način odabiremo nezadovoljenu klauzulu, a potom varijablu unutar

te klauzule čiju vrijednost mijenjamo. Zbog toga što slučajna šetnja može previdjeti globalno kretanje onda kažemo da ona izvodi *hill-climbing*. Činjenica da je neka klauzula nezadovoljena znači da je vrijednost najmanje jedne varijable mora biti promijenjena da bi dosegnuli globalno rješenje. Označimo s k duljinu najveće klauzule u formuli. Kada je $k = 2$ tada će slučajna šetnja riješiti problem zadovoljivosti formule od n varijabli s visokom vjerojatnošću u vremenu $O(n^2)$ [8]. Za veće k samo najgori slučajevi garantiraju da je slučajna šetnja eksponencijalna. U praksi se varijable čiju vrijednost mijenjamo biraju na slučajan način iz neke nezadovoljene klauzule koju pak izabiremo koristeći neku pohlepnu heuristiku.

Originalna heuristika uvodi pojam *breakcount*, koji predstavlja broj klauzula koje su trenutno zadovoljene, a koje će postati nezadovoljene promjenom vrijednosti varijable. Slično, pojam *makecount* predstavlja broj klauzula koje su trenutno nezadovoljene, a koje će postati zadovoljene promjenom vrijednosti varijable. Heuristika odabira je sljedeća:

1. Ako postoje varijable za koje vrijedi $breakcount = 0$, onda na slučajan način odabiremo jednu od njih.

2. Inače, s nekom fiksnom vjerojatnosti p biramo varijablu s najmanjom vrijednosti *breakcount*, a ako je više takvih onda na slučajan način odabiremo jednu.

3. Inače, biramo varijablu s minimalnom vrijednosti *breakcount*, a ako je više takvih onda na slučajan način odabiremo jednu.

Provjera postoje li varijable za koje vrijedi $breakcount = 0$ je ključ za dobre performanse ove heuristike.

Vrijednosti svih *breakcount* i *makecount* se mogu održavati inkrementalno. Na početku su te vrijednosti inicijalizirane. Kada radimo promjenu vrijednosti varijable x iz istine u laž, tada *breakcount* možemo ažurirati tako da posjetimo sve klauzule koje sadrže pozitivnu varijablu x, te ako klauzula sadrži samo jedan istinit literal l povećamo *breakcount* varijable l za jedan. Potrebno je također posjetiti sve klauzule koje sadrže negativnu varijablu x te ako klauzula sadrži samo još jedan istinit literal l potrebno je umanjiti *breakcount* varijable l za jedan. Sličnu operaciju izvodimo kod promjene vrijednosti neke varijable iz laži u istinu. Odabir vrijednosti za p ovisi o konkretnom

problemu. 3-SAT problemi se obično najbrže rješavaju s parametrom $p = 0.5$. Postoji dokaz da je optimalan p približno jednak μ/σ gdje je μ prosječan broj nezadovoljenih klauzula tijekom izvršavanja algoritma i σ je standardna devijacija od ove veličine ([13]).

Postoji dosta dobra heuristika za slučajnu šetnju koja se zove *Rnovelty*. Ideja je pokušati izbjeći mijenjanje malog skupa varijabli. Prilikom svake promijene varijable zabilježimo vremenski žig promjene. Kada je odabrana nezadovoljena klauzula tada *Rnovelty* radi sljedeće:

1. Sortira varijable u klauzuli po *breakcount* i *makecount*.
2. Ako dvije ili više varijabli daju najbolju vrijednost, odabiremo onu varijablu koja nije iz klauzule koje je nedavno mijenjana što određujemo na temelju vremenskog žiga.
3. Inače promotri prve dvije najbolje varijable. Ako najbolja varijabla nije nedavno mijenjana tada napravi promjenu varijable u klauzuli.
4. Inače ako je razlika između najbolje i druge najbolje veća od jedan odaberi najbolju inače odaberi drugu najbolju varijablu.

Sada navodimo pseudokod WalkSAT algoritma.

```
Algoritam WalkSAT.
Ulaz: formula F, interpretacija I, vjerojatnostFlipa.
Izlaz: Interpretacija I.
 brNezKl:= izbNezKl(F,I)
 while brNezKl>0 do
         nezKl:= odRandNezKl(F,I,brNezKl)
       slučajanBroj:= randomReal(0,1)
       if slučajanBroj< vjerojatnostFlipa then
           slučajanIndeks:= rendomInt(1, duljina(nezKl))
           varijablaZaFlip:= abs(nezKl[slučajanIndeks])
       else
```

```
            varijablaZaFlip:= odaberiNajboljuVarZaFlip(F,I,nezKl)
        end if
        I[varijablaZaFlip]:= 1 - I[varijablaZaFlip]
        brNezKl:= izbrojiNezKl(F,I)
    end while
    return I
```

Pseudokod WalkSAT algoritma je preuzet iz [10].

Prednosti algoritma: Učinkovit i veoma brz algoritam.
Pretražuje nešto veći prostor od pohlepnog algoritma.

Mane algoritma: Može zaglaviti u lokalnom optimumu.
Esencijalno nepotpun algoritam.

Primjer 2.4. *Demonstrirat ćemo rad WalkSAT algoritma na formuli:*
$F = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_2 \vee x_4 \vee x_5) \wedge (x_2 \vee x_4 \vee x_5)$.

Budući da se algoritam odvija u nizu iteracija, bit će opisan rad algoritma samo u dvije iteracije da se vidi rad svih dijelova algoritma. Pretpostavimo da je ulaz $I = 10100$, $vjerojatnostFlipa = 0.4$. *Zbog lakše vizualizacije, interpretaciju ćemo reprezentirati kao string* 0 *i* 1 *takav da bit na poziciji i u stringu predstavlja trenutnu istinitosnu vrijednost varijable* x_i, *pri čemu indeksiranje kreće od* 1.

+ *Neka je* $I = 10100$ *na slučajan način generirana interpretacija.*

  - *Na početku iteracije na slučajan način biramo nezadovoljenu klauzulu, no budući da u ovom slučaju imamo dvije nezadovoljene klauzule, na slučajan način odaberemo* $x_2 \vee x_4 \vee x_5$ *i potom generiramo slučajan broj* $rand = 0.24$.
  - *Kako je* $rand < 0.4$ *na slučajan način biramo varijablu iz klauzule koje smo odabrali na početku iteracije tj. iz* $x_2 \vee x_4 \vee x_5$. *Neka je na slučajan način odabrana varija-*

bla x_2. *Sada mijenjamo njenu vrijednost u* 1 *pa dobivamo* $I = 11100$ *i sad je broj nezadovoljenih klauzule jednak* 1.

- *U drugoj iteraciji na samom početku na slučajan način biramo nezadovoljenu klauzulu, no kako je samo jedna nezadovoljena biramo upravo nju. To je klauzula* $\neg x_2 \vee x_4 \vee x_5$. *Na slučajan način generiramo broj* $rand = 0.67$.
- *Budući da imamo* $rand \geq 0.4$ *biramo najbolju varijablu čiju vrijednost mijenjamo. Najbolji rezultat dobivamo zamjenom varijable* x_4 *ili* x_5. *Pretpostavimo da radimo zamjenu na varijabli* x_4 *tj. njena nova vrijednost je sada* 1 *pa dobivamo interpretaciju* $I = 11110$.
- *Sada računamo broj nezadovoljenih varijabli i dobivamo da je to nula što znači da je algoritam gotov i da je rješenje pronađeno.*

Važno je napomenuti da u trenutku kad smo dobili interpretaciju koja zadovoljava sve klauzule algoritam staje i vraća tu interpretaciju, ali je samo radi opisa kako izgleda jedna iteracija algoritma taj dio provjere zanemaren.

3 UnitWalk algoritam

U ovom odjeljku predstavljamo UnitWalk algoritam koji čini bazu našega SAT softverskog modula (solvera).

UnitWalk algoritam je nepotpuni algoritam za rješavanje problema SAT. UnitWalk algoritam koristi lokalno traženje, kao kod eksperimentalno najboljih algoritama iz skupine WalkSAT algoritama u kombinaciji s eliminacijom jediničnih klauzula (klauzule koja sadrži samo jedan literal).

Heuristika za promjenu vrijednosti generirane interpretacije na nekoj varijabli formule je razvijena prema algoritmu Paturi, Pudlák, Saks i Zane-a ([15]). Jezgru toga algoritma čini funkcija koja radi sljedeće.

Za ulaznu formulu F generiramo interpretaciju na slučajan način i generiramo slučajnu permutaciju varijabli formule F. Uzmimo varijable po redoslijedu određenom permutacijom i za svaku izabranu varijablu, ako se varijabla v nalazi u jediničnoj klauzuli tada namjestimo vrijednost interpretacije I tako da vrijedi $I(v) = 1$. Vrijednost interpretacije je određena jediničnom klauzulom. U suprotnom uzmemo vrijednost slučajno generirane interpretacije. Napravimo odgovarajuću supstituciju u formuli u oznaci $F[v \leftarrow True]$ ili $F[v \leftarrow False]$ tako da iz formule eliminiramo sve klauzule koje sadrže varijablu v i za koje vrijedi $I(v) = 1$, a iz svih klauzula koje sadrže varijablu $\neg v$ eliminiramo tu varijablu iz klauzule.

Ovaj postupak eliminacije varijabli se u logici naziva redukcija formule po literalu.
Definiramo redukciju formule F u odnosu na literal l na sljedeći način:

$$G := \{C \backslash \{\bar{l}\} : C \in F \text{ i } l \notin C\}$$

Primijetimo da literal l određuje dvije redukcije: u odnosu na l i $\bar{l}$. Za zadanu formulu F, te dvije redukcije označavamo s F_l i $F_{\bar{l}}$.

Dokazati ćemo lemu koja nam daje važnu poveznicu između redukcije formule i njezine ispunjivosti ([9]).

Lema 3.1 (Davis-Putnam lema)**.** *Neka je F skup klauzula koje nisu tautologije i neka je l neki literal formule F. Skup klauzula F nije ispunjiv ako i smo ako nisu ispunjivi niti F_l niti $F_{\bar{l}}$.*

Dokaz. Pretpostavimo da je F_l ili $F_{\bar{l}}$ ispunjiva formula. Radi određenosti pretpostavimo da je F_l ispunjiva. Slučaj kada je formula $F_{\bar{l}}$ ispunjiva dokazuje se sasvim analogno. Pošto F ne sadrži klauzule koje su tautologije, F_l ne sadrži l i $\bar{l}$. Pošto je F_l ispunjiva, postoji neka interpretacija v takva da $v(F_l) = 1$. Definiramo interpretaciju ω na sljedeći način:

$$\omega(m) = \begin{cases} v(m), & m \notin \{l, \bar{l}\} \\ 1, & m = l \end{cases}$$

Tada za sve $C \in F$ takve da $l \notin C$, pošto vrijedi $v \models C$, tada i $\omega \models C$. Ako je $l \in C$ tada $\omega \models C$ po konstrukciji.

Obrnuto, pretpostavimo da F_l i $F_{\bar{l}}$ nisu ispunjive, te da postoji interpretacija v takva da $v(F) = 1$. Pretpostavimo da je $v(l) = 1$ i promatrajmo F_l. Ako je $v(l) = 0$ promatramo $F_{\bar{l}}$ i primijenimo analogne argumente na taj skup.
Za $C \in F_l$ moguća su dva slučaja:

1. $C \in F$ i $l, \bar{l} \notin C$. U ovom slučaju redukcija po literalu l nije utjecala na klauzulu C, ona je identična kao i u formuli F. Pošto je po pretpostavci $v(F) = 1$, za svaku klauzulu $C_k \in F$ vrijedi $v(C_k) = 1$, stoga $v \models C$.

2. U drugom slučaju je $C = D \backslash \{\bar{l}\}$ i $\bar{l} \in D$, $D \in F$. Sada imamo $v \models D$ i $v(\bar{l}) = 0$ zato što je $v(l) = 1$, pa mora vrijednost interpretacije v za jedan od preostalih literala u D biti 1. U tom slučaju $v \models C$, pa $v \models F_l$ što je kontradikcija.

□

Iz prethodne leme očito slijedi tvrdnja sljedećeg korolara:

Korolar 3.2. *Neka je F skup klauzula koje nisu tautologije i neka je l neki literal formule F. Formula F je ispunjiva ako i samo ako je barem jedna od formula F_l, $F_{\bar{l}}$ ispunjiva.*

Ovi teoretski rezultati pokazuju da je postupak pojednostavljivanja formule koji koristi procedura korektan i da će on dati ispravno rješenje.

Paturi, Pudlák i Zane ([14]) pokazuju da ta procedura pronalazi interpretaciju za koju je ulazna formula istinita, za 3-KNF formulu s vjerojatnošću najmanje $O(2^{-2n/3})$. Ponavljajući proceduru $O(2^{2n/3})$ puta dobivamo konstantnu vjerojatnost greške. Algoritam se može derandomizirati tako da ga pokrenemo na $2^{2n/3}$ interpretacija i polinomni broj permutacija.

Navedenu proceduru koristimo u UnitWalk algoritmu za poboljšavanje slučajno generirane interpretacije. Ako nam se pri izračunavanju pojave jedinične klauzule odmah ih obrađujemo, ako imamo više od jedne jedinične klauzule tada jediničnu klauzulu koju ćemo obrađivati biramo na

slučajan način. Ako postoje dvije jedinične klauzule suprotnih istinitosnih vrijednosti tada ne mijenjamo interpretaciju jer takva promjena nas ne bi dovela do rješenja, formula bi i dalje bila neistinita za modificiranu interpretaciju. Ako u periodu nismo modificirali interpretaciju niti jednom, izaberemo slučajnu varijablu i promijenimo vrijednost interpretacije na njoj. Promjena vrijednosti interpretacije na varijabli v se definira kao: $I(v) = 1 - I(v)$.

U našem solveru koristimo osnovnu verziju UnitWalk algoritma koju ćemo sada opisati.

3.1 Osnovni UnitWalk algoritam

UnitWalk algoritam su razvili Edward A. Hirsch, *Steklov Institute of Mathematics at St. Petersburg* i Arist Kojevnikov *St. Petersbutg State Univerity, Department of Mathematics and Mechanics, St. Petersburg* ([6]).

Kao tipični algoritam lokalnog pretraživanja algoritam generira na slučajan način početnu interpretaciju i modificira ju korak po korak. Glavna razlika u odnosu na ostale algoritme koji koriste slučajne šetnje je taj što UnitWalk algoritam modificira i ulaznu formulu. Slučajna šetnja je podijeljena na periode. Tijekom jednog perioda se napravi barem jedna modifikacija generirane interpretacije, odnosno flip. Period počinje izborom slučajne permutacije varijabli. Nakon toga algoritam korak po korak modificira kopiju ulazne formule i generiranu interpretaciju. U svakom koraku algoritam supstituira vrijednost varijable u formulu, mijenja formulu G u $G[v \leftarrow t]$ za varijablu v i istinitosnu vrijednost t. Ako postoje jedinične klauzule tada uzimamo v iz jedne od jediničnih klauzula. Ako nema jediničnih klauzula, algoritam supstituira vrijednost varijable koja je sljedeća po generiranoj permutaciji. Ako se varijabla v nalazi u jediničnoj klauzuli i vrijedi $I(v) = 1$ za trenutnu interpretaciju i ne postoji klauzula u kojoj se nalazi $\neg v$, tada mijenjamo vrijednost interpretacije na v, odnosno definiramo $I(v) = 1 - I(v)$. Ako završi period bez mijenjanja vrijednosti generirane interpretacije algoritam bira varijablu na slučajan način i mijenja vrijednost interpretacije na njoj. Nakon završetka perioda algoritam generira novu slučajnu permutaciju varijabli, postavlja trenutnu formulu (u koracima algoritma mijenjamo privremenu kopiju ulazne formule) na

ulaznu i izračunava novi period s trenutno generiranom permutacijom. Broj perioda je ograničen s MAX_PERIODS(F) do $+\infty$ (u ovom slučaju algoritam neće nikada stati ako ulazna formula nije ispunjiva).

Sada navodimo pseudokod osnovnog UnitWalk algoritma.

Algoritam UnitWalk.
Ulaz: formula F u KNF.
Izlaz: Interpretacija I takva da $I(F) = 1$ ili rješenje nije pronađeno.

```
for i = 1 to MAX_TRIES(F)
        A=slučajno generirana interpretacija
          za formulu koja ima n varijabli;
         for j = 1 to MAX_PERIODS(F)
            π := slučajna permutacija od 1... n;
            G:=F;
            f:=0;
           for p = 1 to n do
                   while G sadrži jedinične klauzule radi:
                        -Izaberi jediničnu klauzulu {x_j} ili
                         {¬ x_j} iz G na slučajan način;
                        -Ako je vrijednost interpretacije A
                          na toj klauzuli 0 i G ne sadrži
                          suprotnu jediničnu klauzulu, tada
                          A[j] = 1 − A[j] i f := 1;
                        -G := G[x_j ← A[j]];
                   Ako se varijabla x_π[i] još javlja u
                   formuli G, tada G := G[x_π[i] ← A[π[i]]].
             Ako G ne sadrži niti jednu klauzulu
```

```
        I(G) = 1, return A;
        Ako f = 0, izaberi j na slučajan način
        iz 1... n i napravi zamjenu A[j] = 1 − A[j].
return Rješenje nije pronađeno!
```

U nastavku iskazujemo i dokazujemo teorem koji pokazuje da je osnovni UnitWalk algoritam vjerojatnosno aproksimativno kompletan, ako stavimo MAX_PERIODS(F) na $+\infty$ i MAX_TRIES(F) na 1. Tada za svaku ispunjivu formulu i svaku početno generiranu interpretaciju, UnitWalk pronalazi rješenje problema s vjerojatnošću 1.

Definiramo skraćeni zapis interpretacije: $A = \{a_1, a_2, \ldots a_n\}$ označava da je parcijalna interpretacija A definirana samo na varijablama $a_1, a_2 \ldots a_n$ i da vrijedi $A(a_1) = 1,\ A(a_2) = 1, \ldots, A(a_n) = 1$, gdje je svaki a_i jedna propozicionalna varijabla ili negacija propozicionalne varijable.

Teorem 3.3. *Algoritam UnitWalk je vjerojatnosno aproksimativno kompletan.*

Dokaz. Dokazujemo da se za proizvoljnu interpretaciju A i za svaku interpretaciju S za koju je formula F istinita tijekom jednog perioda ili Hammingova udaljenost između A i S smanjuje s odozdo ograničenom vjerojatnošću ili algoritam vraća interpretaciju za koju je F istinita. Promatrajmo proizvoljnu interpretaciju S za koju je formula F istinita i interpretaciju A koja je na Hammingovoj udaljenosti d od interpretacije S. Konstruiramo permutaciju π takvu da ako je izabrana na početku perioda i početna interpretacija je A, tada će interpretacija dobivena na kraju perioda biti bliže interpretaciji S od interpretacije A (primijetimo da se svaka permutacija izabire s vjerojatnošću $\frac{1}{n!}$). Neka je $\pi[1] = i_1, \ldots, \pi[n-d] = i_{n-d}$, gdje $x_{i_1}, \ldots, x_{i_{n-d}}$ predstavljaju varijable na kojima interpretacije A i S imaju identične vrijednosti. Očito se vrijednosti tih varijabli neće mijenjati kroz period. Vrijednosti interpretacija se na ostalim varijablama razlikuju. Ako promijenimo vrijednost interpretacije A na barem jednoj varijabli na kojoj se razlikuje od interpretacije S kroz period, dokazali smo tvrdnju. Jedino na tim varijablama može biti vrijednost interpretacije A uvjetovana jediničnim klauzulama. Ako nije promijenjena vrijednost interpretacije tijekom perioda niti na jednoj varijabli

i nije pronađena interpretacija za koju je F istinita, UnitWalk izabire varijablu na slučajan način i mijenja vrijednost interpretacije A za tu varijablu. S vjerojatnošću najmanje $\frac{1}{n}$ algoritam izabire varijablu čija vrijednost u A je različita od vrijednosti odgovarajuće varijable u S. Svaki period smanjuje Hammingovu udaljenost između A i S s vjerojatnošću $\frac{1}{n \cdot n!}$. Vjerojatnost da ćemo dobiti interpretaciju za koju je ulazna formula istinita u najviše n koraka je stoga najmanje $p(n) = \frac{1}{(n \cdot n!)^n}$ (bez obzira na početno generiranu interpretaciju). Ukupna vjerojatnost da ćemo dobiti interpretaciju za koju je ulazna formula istinita je najmanje $p(n) + (1 - p(n))p(n) + (1 - p(n))^2 p(n) + \cdots = 1$. □

Svojstvo UnitWalk algoritma da ne mijenja vrijednosti generirane interpretacije na varijablama koje su u skladu s nekom interpretacijom za koju je ulazna formula istinita koristimo kod našega solvera; detalje opisujemo u odjeljku Solver.

Primjer 3.4. *Demonstrirat ćemo rad osnovnog UnitWalk algoritma na formuli:*
$F = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_2 \vee x_4 \vee x_5) \wedge (x_2 \vee x_4 \vee x_5)$.

Pretpostavimo da je početno $MAX\_TRIES = 2$, $MAX\_PERIODS = 5$. *Zbog lakše vizualizacije, interpretaciju ćemo reprezentirati kao string* 0 *i* 1 *takav da bit na poziciji i u stringu predstavlja trenutnu istinitosnu vrijednost varijable* x_i.

+ *Neka je* $I = 00101$ *slučajno generirana interpretacija.*
  - *Neka je* $\pi = (15423)$ *slučajno generirana interpretacija u prvom periodu.*
  - $G := F$.
    * *Vršimo redukciju formule varijablom* x_1 *i dobivamo* $G = (x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_2 \vee x_4 \vee x_5) \wedge (x_2 \vee x_4 \vee x_5)$.
    * *Vršimo redukciju formule varijablom* x_5 *i dobivamo* $G = (x_2 \vee x_3)$.
    * *Varijabla* x_4 *je već eliminirana iz formule pa nastavljamo s postupkom eliminacije.*
    * *Vršimo redukciju formule varijablom* x_2 *i dobivamo* $G = (x_3)$.

** *U ovom trenutku formula G sadrži jednu jediničnu klauzulu* (x_3)*, stoga ulazimo u while petlju algoritma. Pošto vrijedi* $I(x_3) = 1$*, vršimo eliminaciju formule* x_3 *iz formule G i dobivamo* $G = ()$*.*

- *Pošto G ne sadrži niti jednu klauzulu, algoritam vraća rješenje I.*

Demonstrirajmo rad algoritma u slučaju da ulazna formula nije istinita za slučajno generiranu interpretaciju u periodu.

Primjer 3.5. *Demonstrirat ćemo rad osnovnog UnitWalk algoritma na formuli:*
$F = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_2 \vee x_4 \vee x_5) \wedge (x_2 \vee x_4 \vee x_5)$.

Pretpostavimo da je početno $MAX\_TRIES = 2$*,* $MAX\_PERIODS = 5$*. Zbog lakše vizualizacije, interpretaciju ćemo reprezentirati kao string* 0 *i* 1 *takav da bit na poziciji i u stringu predstavlja trenutnu istinitosnu vrijednost varijable* x_i*.*

+ *Neka je* $I = 10101$ *slučajno generirana interpretacija.*

  - *Neka je* $\pi = (15423)$ *slučajno generirana interpretacija u prvom periodu.*
  - $G := F$*.*

    * *Vršimo redukciju formule varijablom* x_1 *i dobivamo* $G = (x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_2 \vee x_4 \vee x_5) \wedge (x_2 \vee x_4 \vee x_5)$*.*
    * *Vršimo redukciju formule varijablom* x_5 *i dobivamo* $G = (x_2 \vee \neg x_3) \wedge (x_2 \vee x_3)$*.*
    * *Varijabla* x_4 *je već eliminirana iz formule pa nastavljamo s postupkom eliminacije.*
    * *Vršimo redukciju formule varijablom* x_2 *i dobivamo* $G = (\neg x_3) \wedge (x_3)$*.*

      ** *U ovom trenutku formula G sadrži dvije jedinične klauzule* (x_3) *i* $(\neg x_3)$*, stoga ulazimo u while petlju algoritma. Na slučajan način biramo jediničnu klauzulu čiju varijablu ćemo eliminirati u sljedećem koraku, pretpostavimo da je ta klauzula* (x_3)*. Pošto postoji negacija varijable* x_3 *u formuli G vršimo eliminaciju varijable iz formule G te dobivamo* $G = \{\emptyset\}$*.*

- *Pošto je* $G = \emptyset$ *i nismo promijenili vrijednost niti jednoj varijabli, izaberemo varijablu na slučajan način i promijenimo joj vrijednost te ulazimo u novi period algoritma. Pretpostavimo da smo promijenili vrijednost varijable* x_4 *odnosno* $I(x_4) = 1$*, stoga je sada* $I = 10111$.
- *Neka je* $\pi = (5213)$ *novo generirana permutacija u drugom periodu.*

* *Vršimo redukciju formule varijablom* x_5 *i dobivamo* $G = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3)$.

* *Vršimo redukciju formule varijablom* x_2 *i dobivamo* $G = (\neg x_1 \vee \neg x_3) \wedge (x_3)$.

** *U ovom trenutku formula G sadrži jediničnu klauzulu* (x_3) *stoga ulazimo u while petlju algoritma. Pošto postoji negacija varijable* x_3 *u formuli G vršimo eliminaciju varijable iz formule G te dobivamo* $G = (\neg x_1)$.

** *Pošto formula G i u ovom koraku sadrži jediničnu klauzulu ostajemo u while petlji i vršimo eliminaciju varijable* x_1*. Pošto se u formuli G ne javlja* x_1 *i* $\neg x_1$ *i* $I(\neg x_1) = 0$*, definiramo* $I(\neg x_1) = 1$*, odnosno* $I(x_1) = 0$ *te vršimo eliminaciju varijable* x_1 *iz formule G. U ovom koraku dobivamo* $G = ()$.

- *Pošto vrijedi* $I(F) = 1$ *vraćamo interpretaciju* $I = 00111$ *kao rješenje.*

3.2 Implementacijski detalji

U ovom pododjeljku ćemo opisati našu implementaciju algoritma, korištene strukture podataka te demonstrirati rad algoritma na jednom jednostavnom primjeru.

Kod naše implementacije osnovnog UnitWalk algoritma implementirali smo strukture:

1. **Citac** koja se sastoji od:
   + imena datoteke iz koje čitamo podatke.
   + matrice dimenzija ($nx3$) u kojoj svaki redak predstavlja klauzulu.
   + varijabli koje nam govore: koliko formula ima varijabli, koliko formula ima klauzula.

+ metode za učitavanje podataka iz datoteke i spremanje u odgovarajuće gore navedene varijable.

2. **Klauzula** koja se sastoji od:

   + varijable koja nam govori veličinu klauzule.
   + varijable koja nam govori redni broj klauzule.
   + metode za postavljanje veličine klauzule.
   + metode koja stavlja trenutnu klauzulu u pretinac hash tablice varijabli koje sadrži.

3. **Formula** koja se sastoji od:

   + polja referenci na stvorene klauzule.
   + hash liste referenci klauzula u kojima se nalaze varijable x_i.
   + liste jediničnih klauzula.
   + varijable koja nam govori koliko imamo klauzula u formuli.
   + varijable koja nam govori koliko ima istinitih klauzula u svakom koraku rada algoritma.
   + metode koja računa redukciju formule.
   + metode koja vrši potpuno kopiranje formule.
   + metode koja vrši kopiranje samo potrebnih dijelova formule.
   + metode koja provjerava postoji li negacija varijable koju trenutno eliminiramo u formuli.

Koristimo klasu **Merssene Twister:**
Klasa koja sadrži implementaciju Merssene Twister pseudo slučajnog generatora koji koristimo za slučajno generiranje interpretacija (gotovi kod generatora je preuzet s Interneta).

Uz navedene strukture implementirali smo i globalne funkcije:

+ funkcija koja generira slučajnu permutaciju varijabli

+ funkcija koja generira slučajnu interpretaciju.

+ glavna funkcija koja implementira pseudokod algoritma.

U nastavku ćemo opisati bitnije metode i izvesti njihove složenosti:

Metoda **PrimjeniOP** iz klase formula je metoda koja računa redukciju formule. Pošto imamo u hash listi sve klauzule u kojima se nalazi varijabla koju reduciramo, umjesto m koraka, gdje je m broj klauzula, u prosjeku napravimo $log_2(m)$ koraka. U svakom od tih koraka prolazimo po svim varijablama klauzule do njezine duljine i radimo odgovarajuće supstitucije. Početno duljina klauzula je tri, no ona se smanjuje nakon svake njene reducirane varijable. Ukoliko je klauzula istinita za trenutno generiranu interpretaciju, njezina veličina se stavlja na 0. Demonstrirat ćemo proces redukcije na primjeru:

Primjer 3.6. *Pretpostavimo da u proizvoljnom koraku k, reduciramo varijablu x_i i da imamo klauzulu $C_k = \{\neg x_i \vee x_j \vee x_k\}$. Klauzula C_k je u memoriji k-ti redak matrice koja reprezentira formulu, označimo ga sa: $|-i|\ |j|\ |k|$, početna veličina klauzule je* 3. *Nakon redukcije klauzule C_k dobivamo ekvivalentnu klauzulu $|k|\ |j|\ |-i|$ veličine* 2 *(proces stvaranja ekvivalentne klauzule je složenosti $O(1)$). U sljedećem koraku ukoliko reduciramo neku varijablu iz klauzule C_k, promatramo samo varijable x_k i x_j, varijabla x_i je eliminirana i ona se ne razmatra do sljedećeg perioda.*

Ovakvim supstitucijama osiguravamo da kada reduciramo klauzulu do veličine 1, znamo točno da nam se preostala varijabla nalazi na prvoj poziciji u ekvivalentnoj klauzuli.

Funkcije koje generiraju slučajne permutacije varijabli **perm** i slučajnu interpretaciju **randInt** su složenosti $O(n)$, gdje je n broj varijabli ulazne formule.

Glavna funkcija koja implementira pseudokod algoritma poziva funkciju PrimjeniOP:
$MAX\_TRIES * MAX\_PERIODS * n$ puta.
Eksperimentalno najbolje rezultate smo dobili za $MAX\_TRIES = n$, $NAX\_PERIODS = 15 \cdot n$, gdje je n broj varijabli ulazne formule.

Algoritam smo implementirali u programskom jeziku C#.

Daljnje izmjene algoritma opisujemo u odjeljku Solver u kojemu ćemo opisati i strukturu, način paralelizacije i rad našega solvera.

4 Paralelna heuristika

U ovom odjeljku opisujemo strukturu i funkcionalnost našega paralelnog heurističkog 3-SAT solvera.

4.1 Struktura

Za potrebe solvera dodatno smo implementirali jedan greedy algoritam i jedan genetski algoritam u kojem koristimo razne tehnike mutacija i kombinacija s ciljem proširenja raspon interpretacija koje UnitWalk algoritam dalje dorađuje do rješenja problema.

Pseudokod varijante greedy algoritma koju koristimo za brzo povećavanje broja istinitih klauzula za neku slučajno generiranu interpretaciju:

```
Algoritam Greedy.
Ulaz: formula F u KNF.
Izlaz: Skup interpretacija I_k poboljšanih u zadanom
broju perioda algoritma.

 for i = 1 to Max-tries
             I=slučajno generirana interpretacija
              for j = 1 to Max-flips
                          p:=slučajan broj iz intervala [0,1]
                          (vjerojatnost izbora koraka).
                    if p < 0.8
```

```
                    if I(F) = 1 stavi I u listu
                            generiranih interpretacija na
                            prvu poziciju i vrati skup
                            generiranih interpretacija.
                    else
                            Izaberi klauzulu iz liste
                            klauzula koje nisu istinite
                            za generiranu interpretaciju i
                            izaberi varijablu za koju se
                            maksimizira broj:
                            broj istinitih − broj neistinitih
                            klauzula nakon izmjene
                            vrijednosti varijabli.
            else
                            Izaberi na slučajan
                            način klauzulu iz
                            skupa neistinitih klauzula
                            za generiranu interpretaciju
                            i promijeni joj vrijednost.
        end

                Stavi poboljšanu interpretaciju u
                skup svih interpretacija.
end

return Skup svih generiranih interpretacija.
```

Za implementaciju pohlepnog algoritma koristimo iste strukture kao i za implementaciju

UnitWalk algoritma. Stoga ćemo opisati samo dodatne funkcije i njihove složenosti.

Implementirali smo:

+ Funkciju koja izbacuje iz skupa klauzula sve klauzule koje su postale istinite promjenom vrijednosti zadane varijable i stavlja u skup neistinitih klauzula sve klauzule koje su postale neistinite nakon te promjene.

+ Funkciju koja ispituje koliko je neistinitih klauzula u svakom pokušaju rada algoritma.

+ Funkciju koja računa koliki je stvarni porast u broju istinitih klauzula nakon promjene vrijednosti varijable.

+ Funkciju koja implementira rad gore opisanog greedy algoritma.

Funkcija **azuriraj** izbacuje iz skupa klauzula sve klauzule koje su postale istinite promjenom vrijednosti zadane varijable i stavlja u skup neistinitih klauzula sve klauzule koje su postale neistinite nakon te promjene. Pošto imamo hash listu svih klauzula u kojima se nalazi pojedina varijabla, u prosjeku ova funkcija napravi $log_2(m)$ koraka, gdje je m broj klauzula ulazne formule, a u svakom koraku ispituje vrijednosti 3 varijable.

Funkcija **brojNezadovoljenihKlauzula1** se poziva jednom i ispituje koliko je klauzula istinito odnosno neistinito za početno generiranu interpretaciju. Funkcija radi točno m koraka gdje je m broj klauzula ulazne formule i u svakom koraku ispituje vrijednost 3 varijable.

Funkcija **procjeni** računa stvarni porast u broju istinitih klauzula nakon promjene vrijednosti varijable. Funkcija radi u prosjeku $log_2(m)$ koraka, gdje je m broj klauzula ulazne formule, a u svakom tom koraku ispituje vrijednost 3 varijable.

Glavna funkcija koja implementira rad Greedy algoritma poziva gornje funkcije $Max-periods$ puta. U našem solveru $Max-periods = n$, gdje je n broj varijabli ulazne formule, a $Max-Tries = 1$.

Algoritam je brzi korak poboljšavanja interpretacije koja se zatim prosljeđuju UnitWalk algoritmu na daljnju obradu. Ako algoritam pronađe rješenje tada odmah vrati generiranu interpretaciju. Pošto smo vidjeli da se Hammingova udaljenost između interpretacije istinite za ulaznu varijablu i interpretacije koju poboljšava UnitWalk algoritam nužno asimptotski smanjuje, ima smisla pretpostaviti da bi algoritam mogao prije pronaći rješenje ako mu kao ulaz damo interpretaciju za koju je veći broj klauzula ulazne formule istinit.

U nastavku navodimo pseudokod UnitWalk+greedy algoritma:

```
Algoritam UnitWalk+Greedy.
Ulaz: formula F u KNF.
Izlaz: Interpretacija I takva da I(F) = 1 ili rješenje nije pronađeno.


for i = 1 to MAX_TRIES(F)
 Za sve parne i radi
                A=slučajno generirana interpretacija
                  za formulu koja ima n varijabli;
 Za sve neparne i radi
                A je jedna poboljšana interpretacija
                Greedy algoritmom.
            for j = 1 to MAX_PERIODS(F)
                π := slučajna permutacija od 1... n;
                G:=F;
                f:=0;
               for p = 1 to n do
                       while G sadrži jedinične klauzule radi:
                            -Izaberi jediničnu klauzulu {x_j} ili
                             {¬ x_j} iz G na slučajan način;
```

```
                        -Ako je vrijednost interpretacije A
                          na toj klauzuli 0 i G ne sadrži
                          suprotnu jediničnu klauzulu, tada
                          A[j] = 1 − A[j] i f := 1;
                        -G := G[x_j ← A[j]];
                    Ako se varijabla x_π[i] još javlja u
                    formuli G, tada G := G[x_π[i] ← A[π[i]]].
                Ako G ne sadrži niti jednu klauzulu
                I(G) = 1, return A;
                Ako f = 0, izaberi j na slučajan način
                iz 1... n i napravi zamjenu A[j] = 1 − A[j].
    return Rješenje nije pronađeno!
```

Pošto je suština osnovnog UnitWalk algoritma ostala netaknuta vrijedi:

Teorem 4.1. *UnitWalk+Greedy algoritam je vjerojatnosno aproksimativno kompletan.*

Dokaz. Dokaz direktno slijedi iz teorema 3.4. □

Sada navodimo pseudokod naše varijante SGAV algoritma. Za razliku od originalne verzije naš algoritam vraća skup interpretacija. Razlika je i u tome što se sad veličina skupa B_2 za jedan u svakoj iteraciji u kojoj mijenjamo taj skup.

```
Algoritam SGAV.
Ulaz: formula F u KNF.
Izlaz: Skup interpretacija I.

 Na slučajan način generiraj inicijalnu populaciju P
```

```
for i = 1 to MAX_TRIES(F)
            Izaberi listu kandidata od B jedinki iz P
            while nije dostignut maksimalan broj generacija do
              while nije ispitan maksimalan broj kombinacija
                    roditelja do
                Odaberi dvije jedinke iz B
                Generiraj slučajan broj r između 0 i 1
                             if r < vjerojatnost križanja then
                  Primjeni operator uniformnog križanja
                  da dobiješ novu jedinku V
                  Primjeni operator SLS na jedinku V
                  if V poboljšava kvalitetu od B  then
                    Dodaj V u B_1
                    Izbaci iz B najlošiju jedinku
                  else if V poboljšava raznovrsnost od B then
                    Izbaci iz B_2 najmanje raznovrsnu jedinku
                    Dodaj V u B_2
                    Napravi d-fold sa jedinkom V i prvom u skupu
                    B_2 te ju dodaj u B_2
                  end if
                end if
              end while
            end while
            Na slučajan način generiraj novu populaciju
          end for
return skup jedinki B_1
```

Kod naše implementacije varijante SGAV algoritma implementirali smo strukture:

1. **Citac** koju smo opisali ranije.
2. **HashTablica** koja se sastoji od:
   + polja listi čije vrijednosti su cjelobrojne.
   + konstruktora koji na temelju pročitane formule popuni polje listi čime se ubrzavaju provjere prilikom promjene istinitosti pojedine varijable.

Uz navedene strukture implementirali smo i globalne funkcije:

+ funkcija koja računa Hammingovu udaljenost.
+ funkcija koja radi d-fold križanje.
+ funkcija koja radi uniformno križanje.
+ funkcija koja vraća broj nezadovoljenih klauzula.
+ funkcija koja vraća listu nezadovoljenih klauzula.
+ funkcija koja računa raznolikost(minimalna udaljenost jedinke od skupa).
+ funkcija koja iz inicijalne populacije P vraća B_1.
+ funkcija koja iz inicijalne populacije P vraća B_2.
+ funkcija SLS koja radi poboljšanja na dobivenoj jedinki.
+ glavna funkcija koja implementira pseudokod algoritma.

U nastavku ćemo opisati bitnije metode i izvesti njihove složenosti:

Funkcija **HammingDistance** računa Hammingovu udaljenost dviju jedinki. Za izračunavanje udaljenosti je potrebno usporediti sve bitove dviju jedinki za što je potreban jedan prolaz po obje jedinke koji traje n koraka, gdje je n broj varijabli ulazne formule, stoga je složenosti $O(n)$.

Funkcija koja generira novu jedinku **Cross_d** je složenosti $O(n)$, gdje je n broj varijabli ulazne formule jer je potrebno napraviti $4 * d$ zamjena varijabli pri čemu najviše može biti n zamjena.

Funkcija koja generira novu jedinku uniformnim križanjem **UCO**(uniform crossover) je složenosti $O(n)$, gdje je n broj varijabli ulazne formule jer je potrebno napraviti jedan prolaz kroz dvije interpretacije koje su duljine n.

Funkcija koja računa raznolikost **raznolikost** je složenosti $O(n)$, gdje je n broj varijabli ulazne formule jer je potrebno napraviti $|B|$ poziva funkcije **HammingDistance** čija je složenost $O(n)$. Budući da veličinu skupa B uzimamo empirijski i onda je fiksirana možemo pretpostaviti da je ograničena konstantom, stoga je ukupna složenost $O(n)$.

Funkcija koja poboljšava jedinku **SLS** je složenosti $O(n * f(n))$, gdje je n broj varijabli ulazne formule, a funkcija f nam govori o tome kako parametar $Maxtries$ ovisi o n.

Funkcije **listaNezadovoljenihKlauzula** i **listaNezadovoljenihKlauzula** su složenosti $O(n)$ varijabli jer je potrebno napraviti $3 * n$ provjera istinitosti varijabli unutar klauzula, gdje je n broj ulazne formule.

Glavna funkcija koja implementira pseudokod algoritma poziva funkcije: **UCO Cross_d**, **raznolikost**, **brojNezadovoljenihKlauzula**, **SLS** u najgorem slučaju
$MAX\_Tries * Max\_Generations * Max\_Combinations$ puta jer se križanja odvijaju s nekom vjerojatnošću koja se empirijski određuje, a ona nije uvijek 1.
Eksperimentalno se pokazuje da je potrebno varirati veličine parametara ovisno o problemu da se dobiju što bolji rezultati, a da vremena ostanu u određenim granicama.

Algoritam smo implementirali u programskom jeziku C#.

U nastavku navodimo pseudokod UnitWalk+SGAV algoritma:

```
Algoritam UnitWalk+SGAV.
Ulaz: formula F u KNF.
```

Izlaz: Interpretacija I takva da $I(F) = 1$ ili rješenje nije pronađeno.

if za broj_varijabli formule F vrijedi $200 \leq n < 500$:
Generiraj 15 interpretacija SGAV algoritmom, tako da inicijalno generiraš populaciju od 40, populaciju od 5 najboljih jedinki i populaciju od 5 najraznovrsnijih jedinki. Dvije jedinke križaj s vjerojatnošću 0.6. Uzmi točno 15 lijevih i desnih bitova od dvije točke prekida jednog roditelja i zamijeni ih jedinkama drugog roditelja.

else if za broj_varijabli formule F vrijedi $n \geq 500$:
Generiraj 50 interpretacija SGAV algoritmom, tako da inicijalno generiraš populaciju od 100, populaciju od 5 najboljih jedinki i populaciju od 5 najraznovrsnijih jedinki. Dvije jedinke križaj s vjerojatnošću 0.6. Napravi $n/100$ iteracija SLS algoritma tako da s vjerojatnošću 0.6 odabereš neistinitu klauzulu iz skupa svih neistinitih klauzula za trenutno generiranu interpretaciju, a s vjerojatnošću 0.3 promijeni vrijednost na slučajan način odabranoj varijabli. Uzmi točno 30 lijevih i desnih bitova od dvije točke prekida jednog roditelja i zamijeni ih jedinkama drugog roditelja.

for $i = 1$ to MAX_TRIES(F)
if Varijabla sadrži više od 175 varijabli:
if paran i takav da je i manji od broja generiranih interpretacija SGAV algoritmom
A=jedna od poboljšanih interpretacija SGAV

```
                algoritmom.
        else
               A=slučajno generirana interpretacija.
else
         A=slučajno generirana interpretacija.
            for j = 1 to MAX_PERIODS(F)
                π := slučajna permutacija od 1... n;
                G:=F;
                f:=0;
               for p = 1 to n do
                       while G sadrži jedinične klauzule radi:
                            -Izaberi jediničnu klauzulu {x_j} ili
                             {¬ x_j} iz G na slučajan način;
                            -Ako je vrijednost interpretacije A
                              na toj klauzuli 0 i G ne sadrži
                              suprotnu jediničnu klauzulu, tada
                              A[j] = 1 − A[j] i f := 1;
                            -G := G[x_j ← A[j]];
                       Ako se varijabla x_π[i] još javlja u
                       formuli G, tada G := G[x_π[i] ← A[π[i]]].
                Ako G ne sadrži niti jednu klauzulu
                I(G) = 1, return A;
                Ako f = 0, izaberi j na slučajan način
                iz 1... n i napravi zamjenu A[j] = 1 − A[j].
return Rješenje nije pronađeno!
```

Treći pomoćni algoritam je modifikacija UnitWalk algoritma koja pri učitavanju podataka iz

datoteke učitava i informaciju o pojavljivanjima propozicionalnih varijabli i njihovih negacija u pojedinim klauzulama te tu informaciju koristi za aproksimativno određivanje varijable čiju bi vrijednost trebalo promijeniti u neistinitoj klauzuli. Pseudokod je identičan osnovnom UnitWalk algoritmu, razlika je u funkciji koja vrši redukciju varijabli iz formule. Dok osnovni UnitWalk algoritam u slučaju kada naiđe na klauzulu koja nije istinita za generiranu interpretaciju kod eliminacije varijable radi promjenu samo na varijabli koju trenutno eliminiramo ako ne postoji negacija te varijable u formuli, mi korištenjem učitane informacije mijenjamo varijablu za koju je $max(br_t - br_f)$ gdje je br_t broj klauzula koje će postati istinite ukoliko promijenimo vrijednost varijable, a br_f broj klauzula koje će možda postati neistinite ukoliko promijenimo vrijednost iste varijable neovisno o postojanju negacije varijable čiju vrijednost mijenjamo u formuli.

4.2 Funkcionalnost

Solver u ovisnosti o broju varijabli ulazne formule određuje koji od tri navedena algoritma se izvršava i broj algoritama koji se izvode paralelno.

$$\begin{cases} n \leq 100, \text{ Osnovni UnitWalk algoritam} \\ 100 < n \leq 200, \text{ UnitWalk+Greedy algoritam} \\ 200 \leq n, \text{ UnitWalk, UnitWalk+SGAV, UnitWalk+Greedy algoritmi} \end{cases}$$

Za formule koje sadrže manje od 200 varijabli paralelizacija se ne isplati. Vrijeme trostrukog učitavanja podataka, pokretanja threadova, provjere i gašenja threadova je veće od vremena potrebnog navedenim algoritmima da izračunaju rješenje sekvencijalno. Za $n \geq 200$ solver konkurentno izvršava UnitWalk+Greedy, UnitWalk+SGAV i UnitWalk algoritam s dodatno učitanim podacima na tri threada. Glavni thread periodički provjerava je li koji od algoritama završio s radom, ako je to slučaj gasi preostale threadove i vraća rješenje. Vrijeme provjere main threada ovisi o broju varijabli formule, što formula ima više varijabli to su rjeđe provjere jer je vjerojatnost da ćemo pronaći rješenje manja. Stoga više procesorskog vremena trošimo za izvođenje algoritama koji računaju rješenje.

Vrijeme čekanja glavnog threada:

$$Time_S = \begin{cases} 10\text{ms}, & 200 \leq n < 225 \\ 30\text{ms}, & 225 \leq n < 240 \\ 40\text{ms}, & 240 \leq n \leq 250 \\ 200\text{ms}, & 250 < n < 500 \\ 2000\text{ms}, & 500 \leq n \end{cases}$$

Pošto sva tri navedena algoritma pri računanju interpretacije mijenjaju i ulaznu formulu, učitamo podatke za svaki algoritam posebno te tako dobivamo potpuno konkurentno izvršavanje bez upotrebe tehnika međusobnog isključivanja.

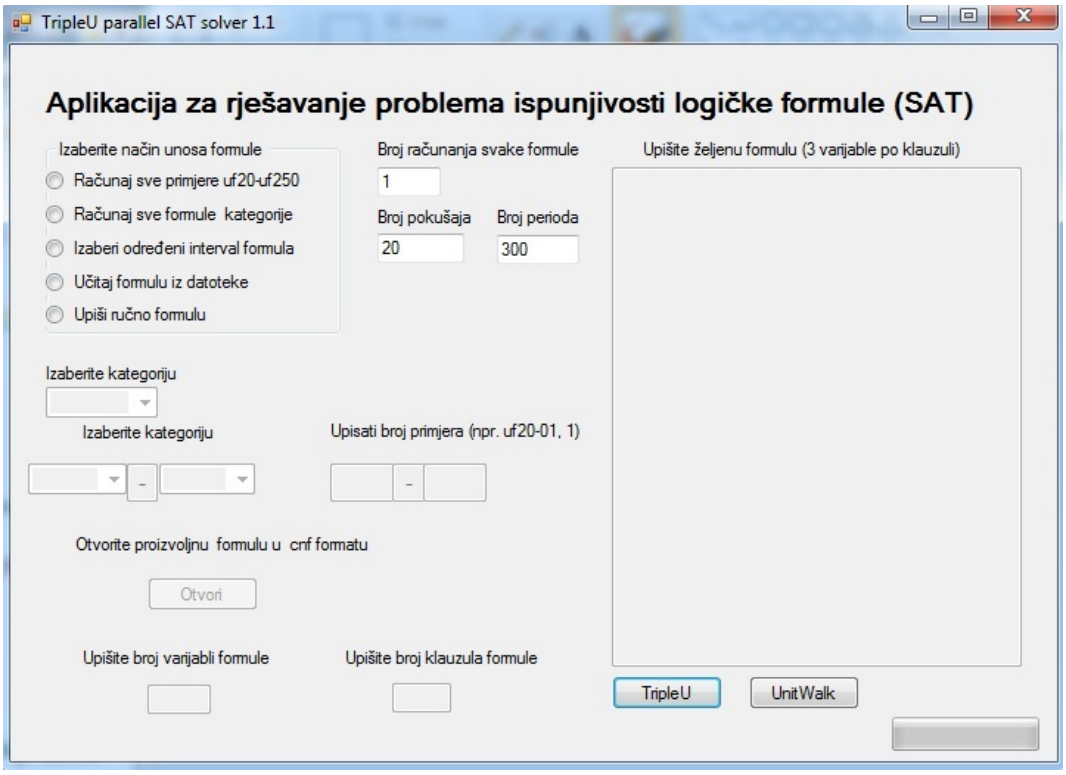
Kao i osnovni UnitWalk algoritam, sve pomoćne algoritme i sam solver smo implementirali u programskom jeziku C#. Napravili smo odgovarajuće korisničko sučelje pomoću kojega se može odrediti način testiranja rada solvera i usporediti njegovo vrijeme izvršavanja s vremenom izvršavanja UnitWalk algoritma. Solver sadrži bazu testnih formula skinutih sa SATLIB stranica ([20]) $uf20 - uf250$, gdje uf označava da su formule generirane na slučajan način iz uniformne distribucije, a broj označava broj varijabli formule. Uz to, postoji mogućnost učitavanja proizvoljne formule u standardnom DIMACS formatu te ručni unos formule u formu. U slučaju ručnog unosa aplikacija će stvoriti datoteku `generirana.cnf` u kojoj će se nalaziti unesena formula u standardnom formatu.

Teorem 4.2. *Solver TripleU je vjerojatnosno aproksimativno kompletan.*

Dokaz. Po teoremu 3.4 znamo da je osnovni UnitWalk algoritam vjerojatnosno aproksimativno kompletan. Pošto se solver za sve formule s $n \leq 100$ varijabli ponaša kao UnitWalk algoritam, tada je on vjerojatnosno aproksimativno kompletan za te ulaze. Za formule s $100 < n \leq 200$ varijabli solver se ponaša kao UnitWalk+Greedy algoritam koji je vjerojatnosno aproksimativno kompletan po teoremu 4.1, stoga je i TripleU vjerojatnosno aproksimativno kompletan za takve formule. Za formule s $200 \leq n$ varijabli solver izvršava konkurentno modificirani UnitWalk algoritam, UnitWalk+SGAV algoritam i UnitWalk+Greedy algoritam. Pretpostavimo sada da smo

definirali $MAX\_TRIES = 1$ i $MAX\_PERIODS = +\infty$. Pošto je UnitWalk+Greedy algoritam vjerojatnosno aproksimativno kompletan on će s vjerojatnošću 1 poboljšati generiranu interpretaciju do interpretacije za koju je ulazna formula istinita. U tom koraku će $UnitWalk + Greedy$ algoritam vratiti rješenje, a glavni thread će zaustaviti izvršavanje UnitWalk i UnitWalk+SGAV algoritma. Od tuda slijedi da će TripleU solver pronaći rješenje za svaku ispunjivu ulaznu formulu s vjerojatnošću 1 bez obzira na njezine dimenzije u konačno mnogo vremena, odnosno TripleU je vjerojatnosno aproksimativno kompletan. □

U nastavku opisujemo grafičko sučelje aplikacije.



Slika 1: Grafičko sučelje TripleU solvera.

Na lijevoj strani grafičkog sučelja se nalaze kontrole za način unosa formule i izbor test primjera.

- Prvi izbor testira rad solvera na svim primjerima $uf20 - uf250$.

- Drugi izbor omogućava testiranje solvera na svim instancama određene kategorije.

- Treći izbor omogućava testiranje solvera na podskupu jedne ili više kategorija formula.

- Četvrti izbor omogućava učitavanje formule u DIMACS formatu iz datoteke.

- Peti izbor omogućava ručno upisivanje formule na kojoj ćemo testirati rad solvera.

Polje **Broj računanja svake formule** definira koliko ćemo puta pokrenuti solver na svakoj zadanoj instanci. Polja **Broj pokušaja** i **Broj perioda** definiraju koliko pokušaja i koliko perioda će napraviti svaki od konkurentno izvođenih algoritama. Parametri se računaju po formuli $MAX\_TRIES = bt * br\_var/20$, $MAX\_PERIODS = bp * br\_var/20$, gdje je bt definirani broj pokušaja, a bp definirani broj perioda od strane korisnika. Izborom odgovarajućeg načina unosa formule se omogućava daljnji izbor, učitavanje ili upisivanje formule na kojoj će se solver testirati.

Solver kao ulaz prima formule u DIMACS formatu:

Primjer 4.3. *Pretpostavimo da želimo učitati formulu* $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$.
U odgovarajuću formu treba upisati:

```
1 2 -3 0
-2 3 4 0
-1 -3 -4 0
```

Na primjeru vidimo da se varijabla prikazuje kao cijeli broj, negirana varijabla se prikazuje kao negativan broj, a varijabla koja se javlja bez negacije kao prirodan broj, 0 *se koristi kao delimiter između klauzula.*

Solver će generirati datoteku `Generirana.cnf` koja izgleda:

```
c This Formular is generated by UnitWalk 2.1
c
```

```
c    horn? no
c    forced? no
c    mixed sat? no
c    clause length = 3
c
p cnf 4 3
1 2 -3 0
-2 3 4 0
-1 -3 -4 0
%
0
```

Kao rješenje solver vraća:

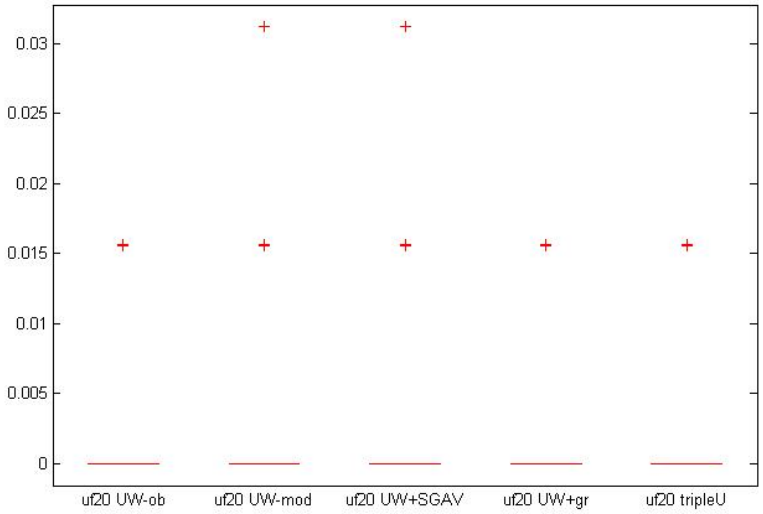
```
22.4.2011. 23:46:59
Test rada TripleU solvera.
Svaki test primjer se izvršava 1 puta.

Ime datoteke:     Vrijeme izvrsavanja:          Uspjeh:
Generirana.cnf        00:00:00                   +
                      :1 0 0 1                    Br. Ist. Kl: 3

22.4.2011. 23:46:59

Ukupno vrijeme:      00:00:00
```

Gumb **TripleU** pokreće TripleU solver, a gumb **UnitWalk** pokreće UnitWalk algoritam na prije određenim test primjerima određeni broj puta.

5 Analiza rezultata testiranja

U ovom odjeljku analiziramo rezultate testiranja našega solvera i uspoređujemo sa svim pomoćno implementiranim varijantama UnitWalk algoritma.

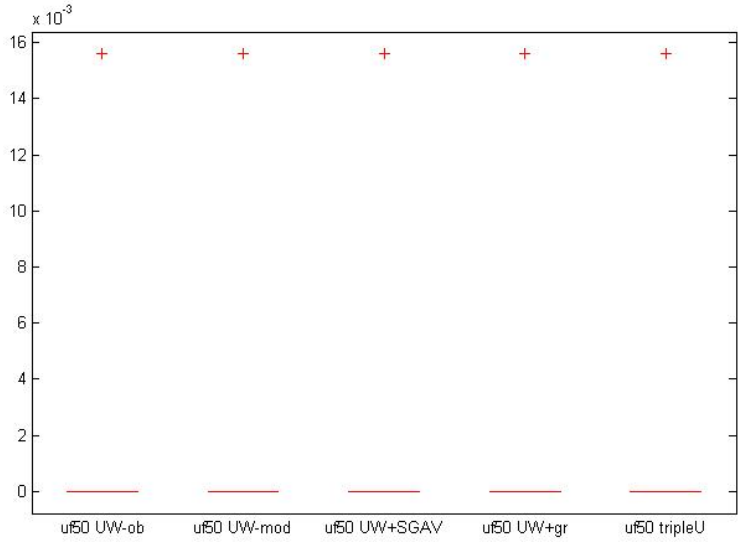
Solver i svi pomoćni algoritmi su testirani na test primjerima $uf20 - uf250$ skinutim s [20]. Prikazat ćemo usporedne boxplotove izvođenja algoritama i solvera redom za svaku kategoriju formula, prikazati rast prosječnog vremena izvođenja u odnosu na povećanje broja varijabli formule, usporediti prosječna vremena izvršavanja solvera s UW-ob, UW-mod, UW+SGAV i UW+gr algoritmom. Sva vremena su u sekundama. Algoritmi i solver su implementirani u programskom jeziku C# i testirani na Intel i3, 2.4Ghz procesoru.



Slika 2: Usporedni boxplot izvođenja algoritama na formulama $uf20$ 1 − 1000 (svaki testni primjer je pokretan 10 puta).

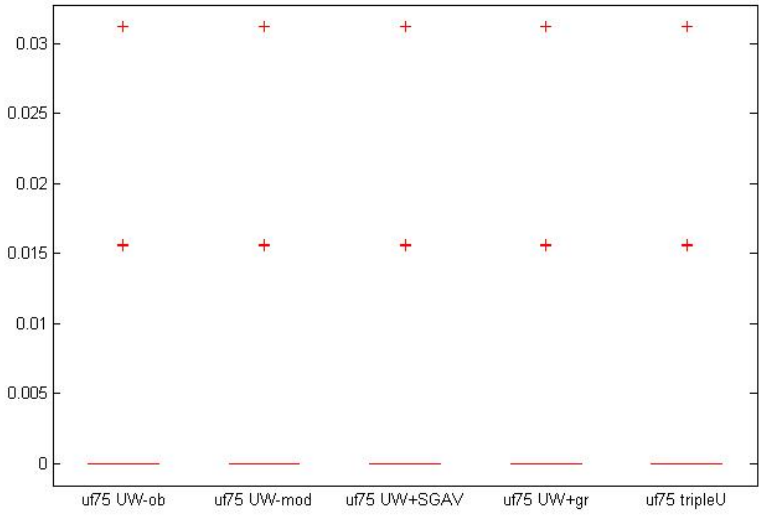
Za $n < 100$ naš solver radi kao osnovni UnitWalk algoritam što je i vidljivo iz test podataka. Skupina formula $uf - 20$ je namijenjena testiranju rada algoritama, stoga na toj klasi svi algoritmi

imaju prosječno vrijeme izvođenja 0 sekundi. Uočavamo da je modifikacija osnovnog UnitWalk algoritma, koja je pokušala poluinformiranim greedy pristupom poboljšati heuristiku za promjenu vrijednosti varijable dovela do pojavljivanja dodatne rubne vrijednosti (”outliera”) u vremenima izvršavanja.



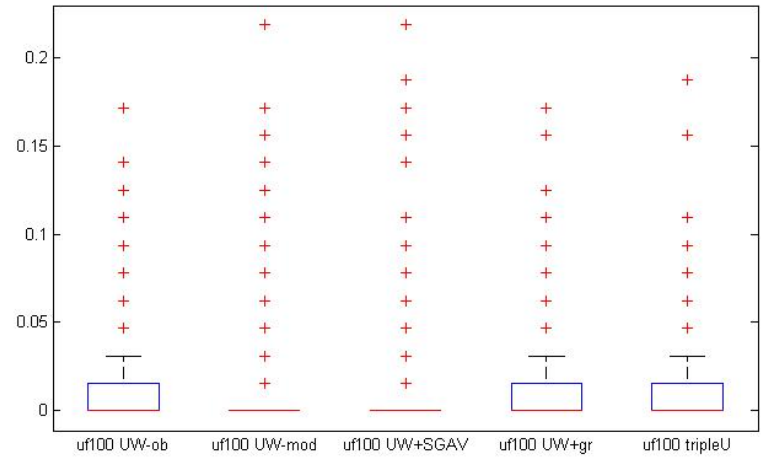
Slika 3: Usporedni boxplot izvođenja algoritama na formulama $uf50\ 1-1000$ (svaki testni primjer je pokretan 10 puta).

Klasa formula $uf50$ sadrži još uvijek malene i osnovnim UnitWalk algoritmom lako rješive formule. Zbog toga još uvijek UnitWalk+SGAV algoritam ne primjenjuje poboljšanja interpretacija evolucijskim tehnikama, a solver izvršava osnovni UnitWalk algoritam.



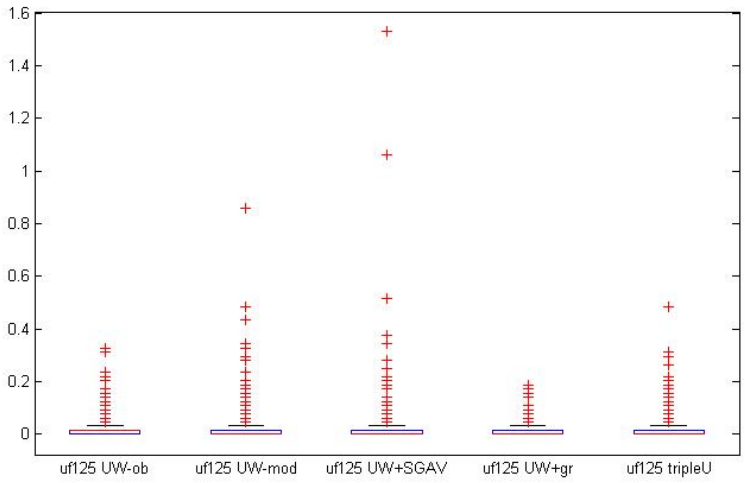
Slika 4: Usporedni boxplot izvođenja algoritama na formulama $uf75$ $1-100$ (svaki testni primjer je pokretan 10 puta).

Kao i za klasu formula $uf50$ i klasa $uf75$ je suviše trivijalna da bi uočili neke bitnije razlike u algoritmima.



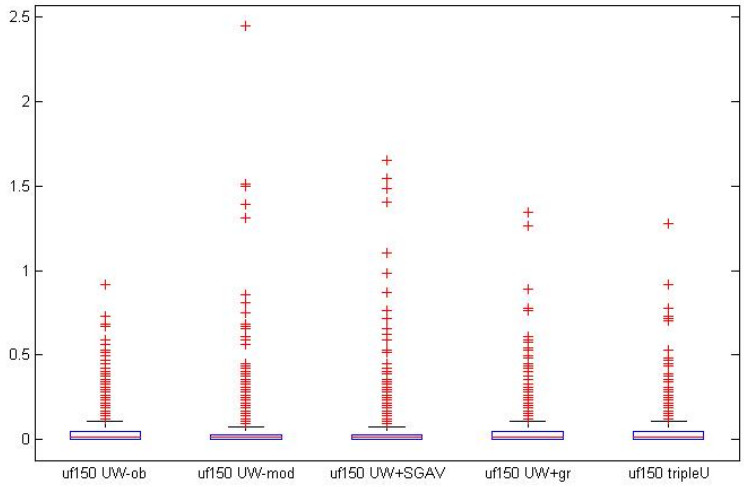
Slika 5: Usporedni boxplot izvođenja algoritama na formulama $uf100$ $1-1000$ (svaki testni primjer je pokretan 10 puta).

Na klasi formula $uf100$ se već počinje uočavati da modificirani UnitWalk algoritam u prosjeku brže rješava problem od osnovnog UnitWalk algoritma. Međutim, ako ulazna formula nije pogodna za rješavanje greedy pristupom, tada algoritam dolazi do rješenja sporije nego osnovni UnitWalk algoritam. Stoga vidimo veći broj "outliera" kod modificiranog UW-mod algoritma. Kod modificiranog UnitWalk algoritma smo zadržali korak u kojemu na slučajan način izaberemo varijablu i promijenimo joj vrijednost ukoliko u trenutnom periodu nismo sa sigurnošću (slučaj kada za varijablu x_i vrijedi $I(x_i) = 0$ i ne postoji negacija varijable x_i u ulaznoj formuli) promijenili vrijednost niti jednoj varijabli; stoga se ne može dogoditi da zaglavimo u lokalnom optimumu.



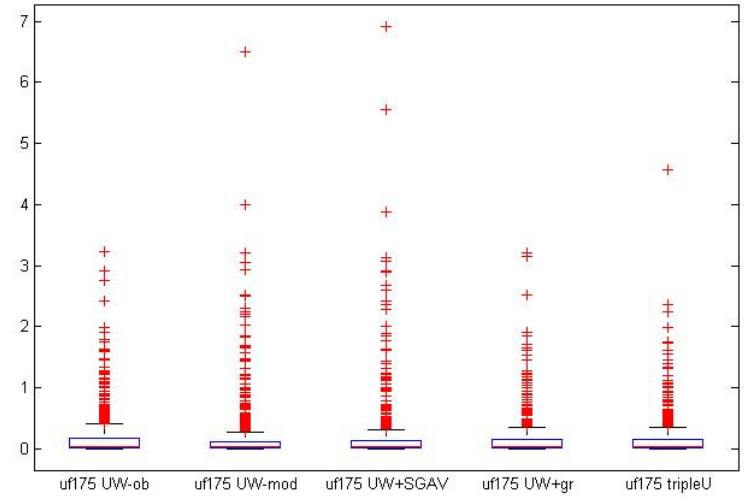
Slika 6: Usporedni boxplot izvođenja algoritama na formulama $uf125\ 1-100$ (svaki testni primjer je pokretan 10 puta).

U ovom trenutku solver radi identično kao UnitWalk algoritam koji greedy pristupom pokušava poboljšati ulaznu interpretaciju u svakom neparnom periodu. Vidimo da je srednja vrijednost izvršavanja sva četiri algoritma osim osnovnog UnitWalk algoritma 0 sekundi.



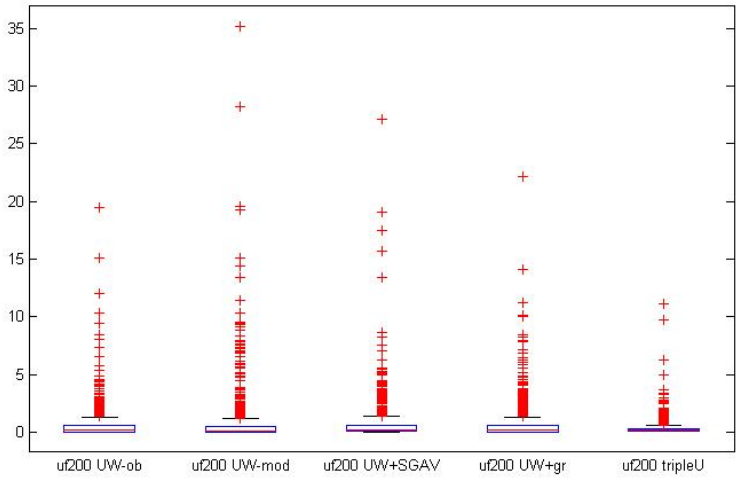
Slika 7: Usporedni boxplot izvođenja algoritama na formulama $uf150\ 1-100$ (svaki testni primjer je pokretan 10 puta).

Uočavamo da su opet UW-mod i UW-SGAV algoritmi nešto bolji u prosjeku, ali imaju povećani broj ”outliera” u odnosu na ostala tri algoritma.



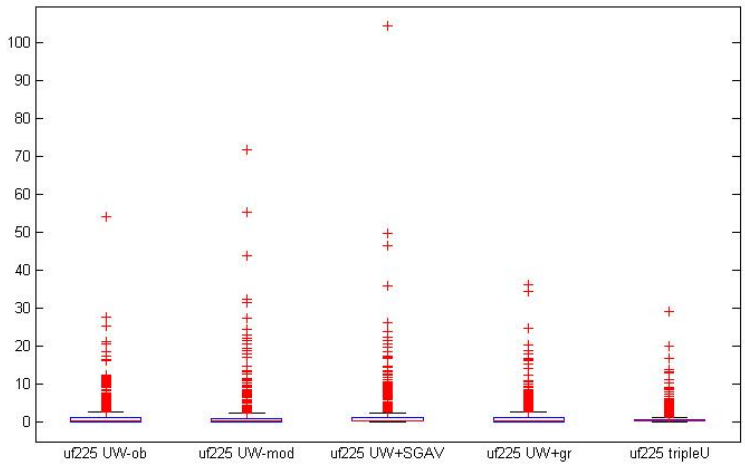
Slika 8: Usporedni boxplot izvođenja algoritama na formulama $uf175\ 1-100$ (svaki testni primjer je pokretan 10 puta).

Prema rezultatima $uf100 - uf175$ uočavamo da bi mogli popraviti prosjek izvršavanja solvera simuliranjem modificiranog UnitWalk algoritma umjesto osnovnog. No, tada bi izgubili svojstvo vjerojatnosno aproksimativne kompletnosti na formulama s manje od 175 varijabli, što ne želimo. Vremena izvršavanja svih algoritama su toliko malena na ulaznim formulama s manje od 175 varijabli da vremenska ušteda na svim testiranim instancama ne bi premašila niti 2 minute.

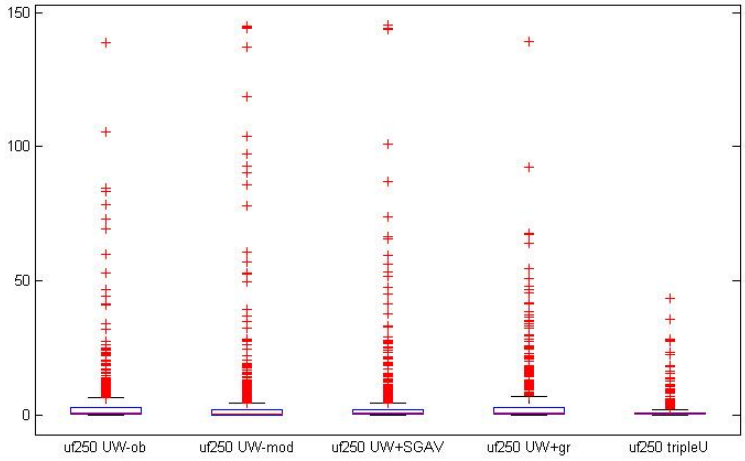


Slika 9: Usporedni boxplot izvođenja algoritama na formulama $uf200\ 1 - 100$ (svaki testni primjer je pokretan 10 puta).

Za formule s više ili jednako 200 varijabli solver konkurentno izvodi UW-mod, UW+SGAV i UW+gr algoritam. Uočavamo dosta veliko smanjenje prosječnog izvršavanja solvera u odnosu na ostale algoritme kao i nestajanje najudaljenijih ”outliera”.

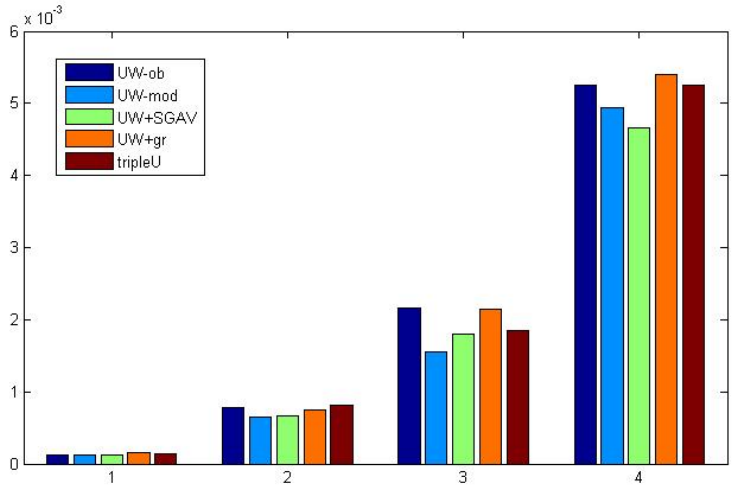


Slika 10: Usporedni boxplot izvođenja algoritama na formulama $uf225\ 1-100$ (svaki testni primjer je pokretan 10 puta).

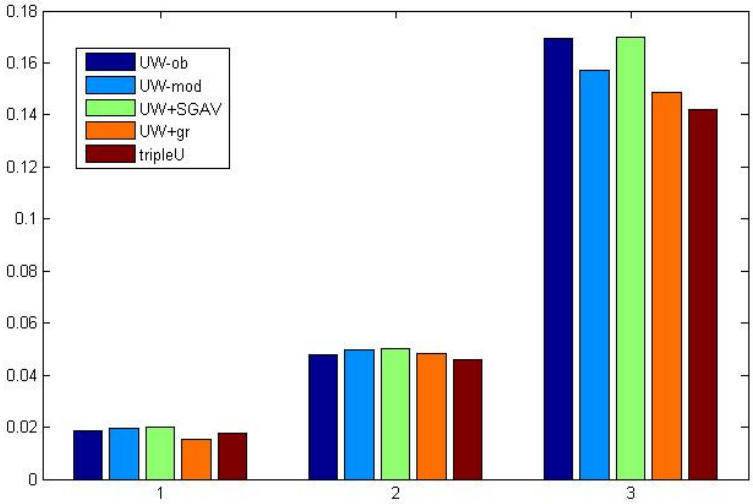


Slika 11: Usporedni boxplot izvođenja algoritama na formulama $uf250\ 1-100$ (svaki testni primjer je pokretan 10 puta).

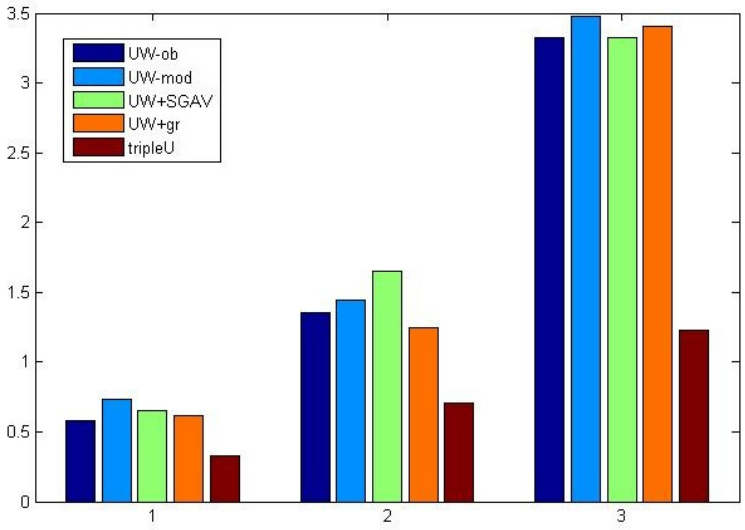
Promotrimo sada prosječna vremena izvršavanja algoritama:



Slika 12: Usporedba prosječnih vremena izvođenja algoritama za instance formula iz klasa $uf20$, $uf50$, $uf75$, $uf100$.



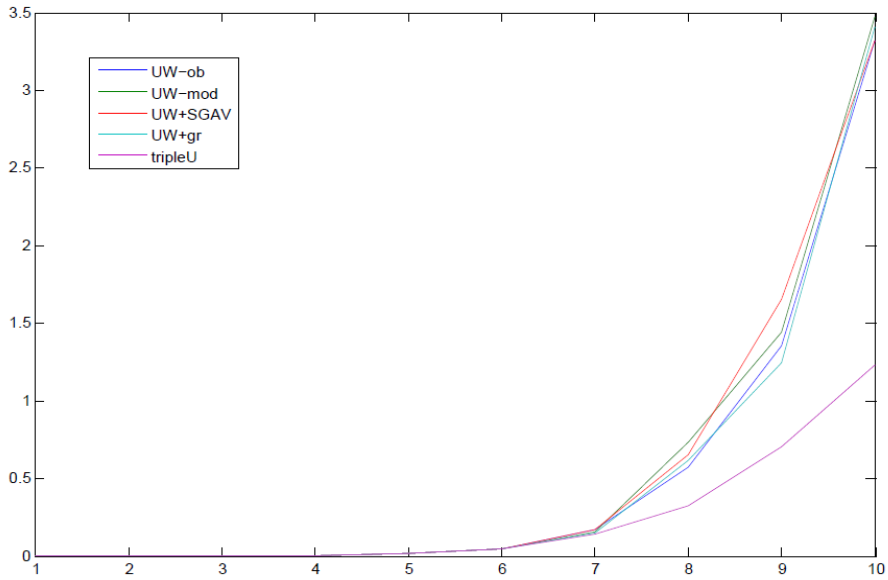
Slika 13: Usporedba prosječnih vremena izvođenja algoritama za instance formula iz klasa $uf125$, $uf150$ $uf175$.



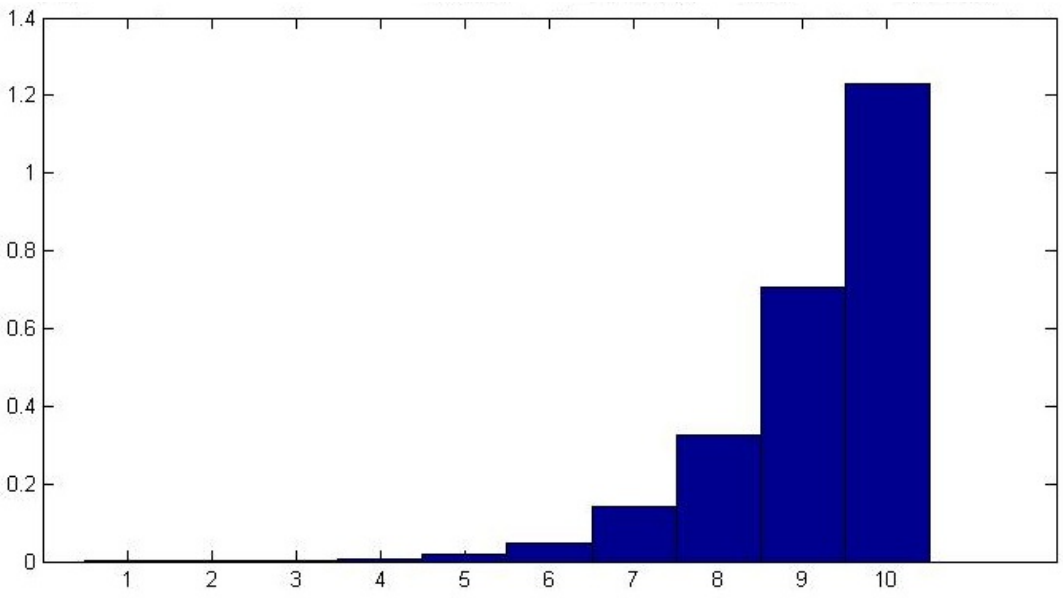
Slika 14: Usporedba prosječnih vremena izvođenja algoritama za instance formula iz klasa $uf200$, $uf225$ $uf250$.

Vidimo da s povećanjem broja varijabli ulazne formule raste razlika između prosječnog vremena izvršavanja TripleU solvera i osnovnog UnitWalk algoritma.

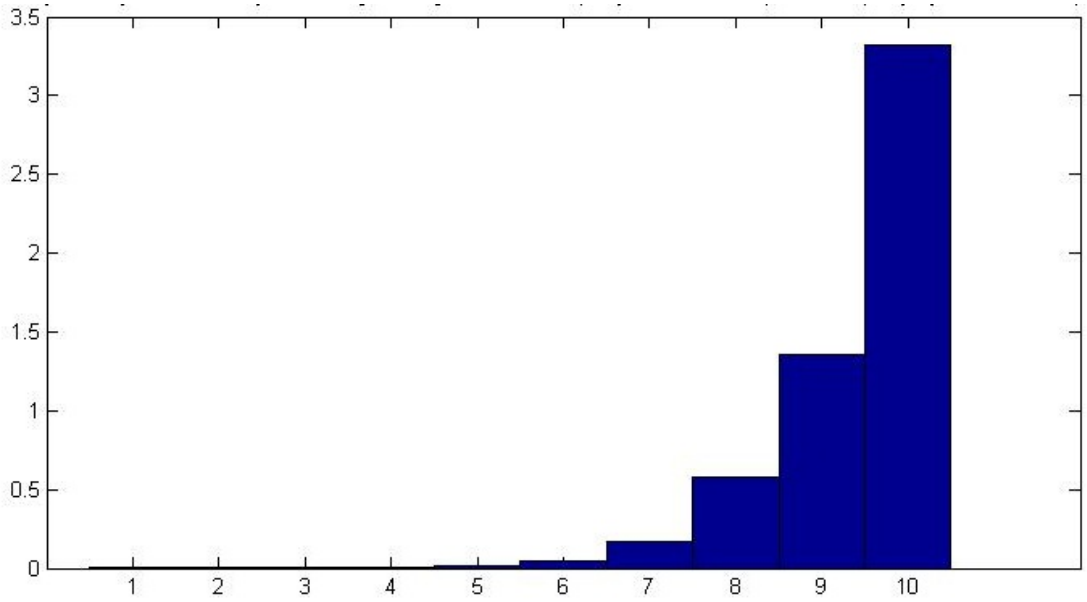
Promotrimo brzinu rasta prosječnih vremena izvođenja algoritama i solvera:



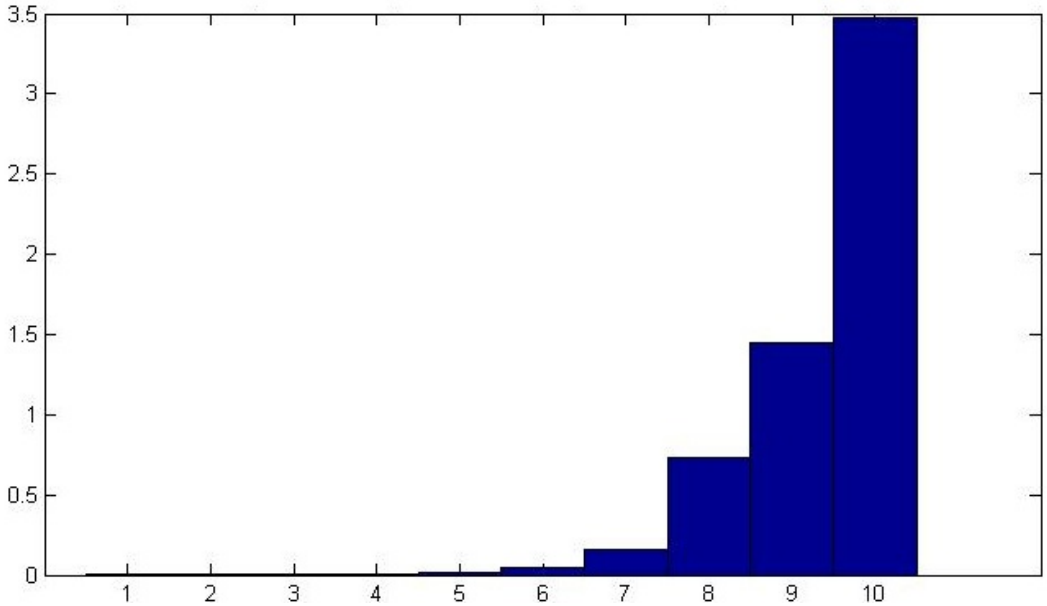
Slika 15: Porast vremena prosječnog izvršavanja algoritama i solvera s povećanjem broja varijabli formule.



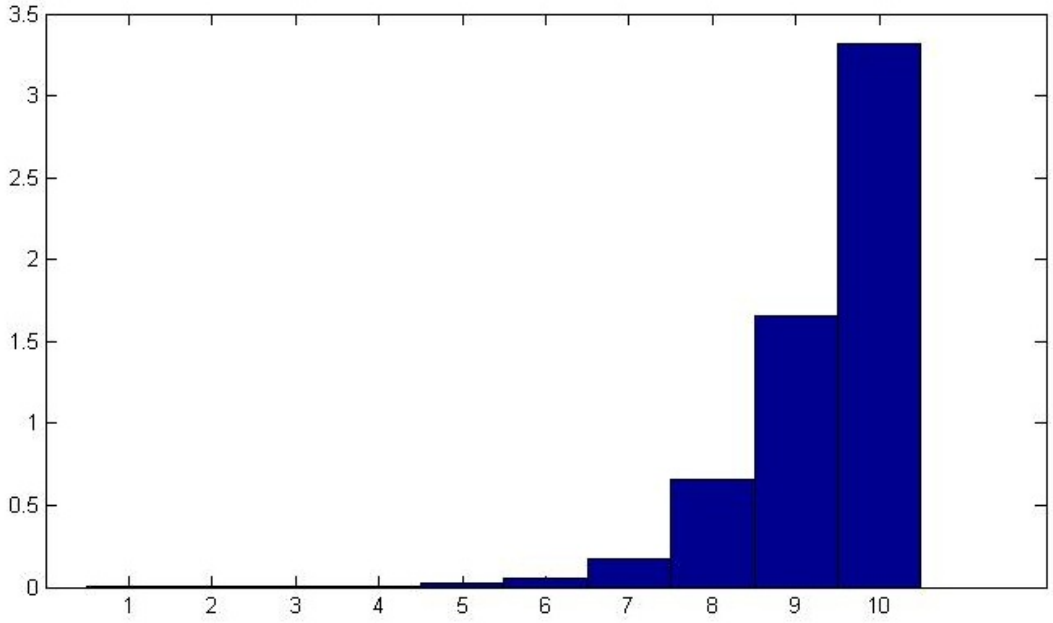
Slika 16: Prosječna vremena izvođenja TripleU solvera na formulama iz klasa $uf20 - uf250$.



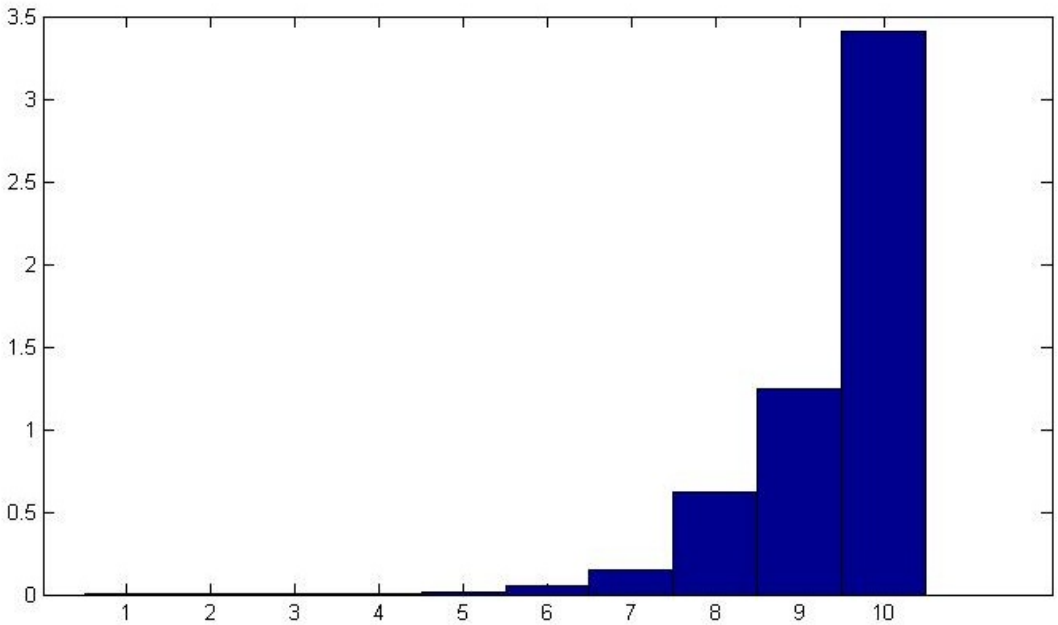
Slika 17: Prosječna vremena izvođenja osnovnog UnitWalk algoritma na formulama iz klasa $uf20 - uf250$.



Slika 18: Prosječna vremena izvođenja UW-mod algoritma na formulama iz klasa $uf20 - uf250$.

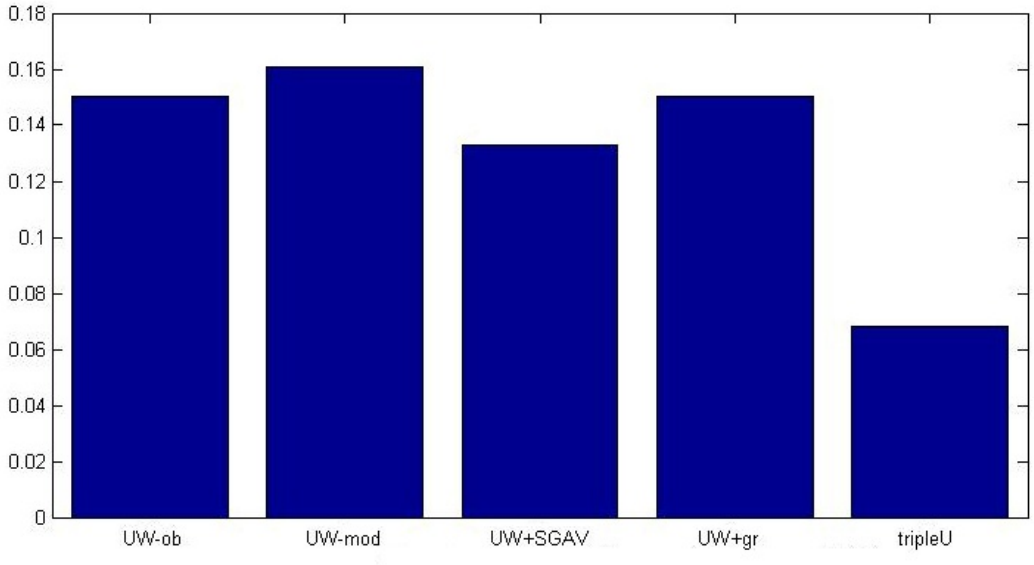


Slika 19: Prosječna vremena izvođenja osnovnog UW+SGAV algoritma na formulama iz klasa $uf20 - uf250$.



Slika 20: Prosječna vremena izvođenja osnovnog UW+gr algoritma na formulama iz klasa $uf20 - uf250$.

Promotrimo sada prosječno vrijeme izvršavanja na svim instancama $uf20 - uf250$.



Slika 21: Usporedba prosječnih vremena izvođenja algoritama na svim formulama iz klasa $uf20 - uf250$.

Iz ovih testiranja podataka možemo zaključiti da UW-mod algoritam, iako ima manja prosječna vremena izvođenja od osnovnog UnitWalk algoritma, zbog povećanog broja ”outliera” ima slabije ukupno prosječno vrijeme izvođenja od osnovnog UnitWalk algoritma. Naš algoritam UnitWalk+SGAV primjenom genetskog algoritma uspijeva UnitWalk algoritmu predati raznolikije interpretacije, koje on kasnije lakše dovodi do rješenja, što se vidi na ukupnom vremenu izvođenja. Također, vidimo znatno poboljšanje u vremenu izvođenja kod TripleU solvera s povećanjem broja varijabli ulazne formule. Sigurni smo da bi se razlika između vremena izvođenja TripleU solvera i ostalih algoritama dodatno povećala na primjerima s 300, 400 i 500 varijabli, kojih na žalost nema na SATLIB stranicama. Poboljšano vrijeme izvođenja je donekle posljedica i činjenice da smo uspjeli eliminirati najudaljenije ”outliere” koji se javljaju u UW-ob, UW-mod, UW+SGAV i UW+gr algoritmima.

Pošto znamo da TripleU solver nije potpuni algoritam, uz vrijeme izvršavanja zanima nas i us-

pjeh pronalaska interpretacija za zadane ispunjive formule. Ukoliko damo solveru na ulaz formulu koja nije ispunjiva, stavimo $MAX\_TRIES = 1$ i $MAX\_PERIODS = +\infty$ algoritam neće stati. Međutim na malim formulama (s < 150 varijabli) možemo s visokom vjerojatnošću zaključiti da formula nije ispunjiva ukoliko TripleU solver ne pronađe rješenje.

Tablica 1: Točnost izvođenja TripleU solvera na formulama iz klasa $uf20 - uf250$ (svaki testni primjer je izvršavan 10 puta).

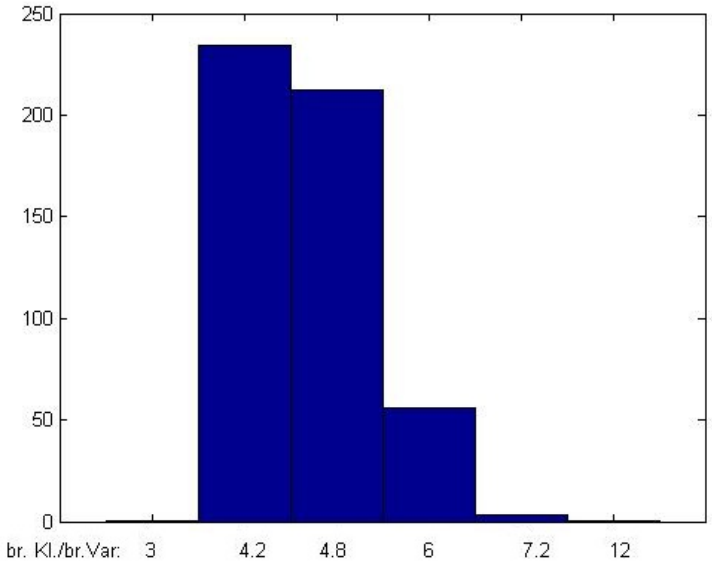
$uf20$	$uf50$	$uf75$	$uf100$	$uf125$	$uf150$	$uf175$	$uf200$	$uf225$	$u250$
100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Od teže rješivih formula, solver je uspio riješiti formulu $f600$ iz klase veoma velikih generiranih formula skinutih s [20].

```
23.3.2011. 1:51:48
Test rada TripleU solvera.
Svaki test primjer se izvršava 1 puta.

Ime datoteke:                              Vrijeme izvrsavanja:  Uspjeh:
C:\Users\matej\Documents\uf600-01.cnf    00:07:00.8263391       +
0 1 0 1 1 0 1 1 1 0 1 1 0 1 0 1 0 1 1 1 0 0 1 0 0 0 0 0 0 1 1 0 0 0
0 1 0 1 1 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 1 0 0 0 1
0 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 1 1 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 1
0 0 1 0 1 1 0 0 0 1 1 1 1 1 1 0 1 0 0 1 0 1 0 0 1 1 0 1 0 1 1 1 1 1
1 1 0 1 0 1 1 1 0 0 1 0 1 1 0 0 0 1 0 0 1 0 0 1 0 1 0 1 1 1 1 1 0 1
0 1 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 0 0 0 0 0 1 0 1 1 0 1 1 1 0 0
0 1 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 0 1 0 1 1 0 1 0 0 0 0
1 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1
0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 1 1 1 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 1
```

```
0 0 1 0 0 0 1 0 0 1 1 0 1 1 1 0 1 0 1 1 1 0 0 0 1 1 1 0 0 0 0 1 1 1
1 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 0 1 1 0 0 0 1 0 1 1 0 0 1 0 1 0 0
0 1 0 1 1 0 0 0 0 1 1 0 0 1 1 1 1 1 0 1 0 0 1 0 1 0 0 1 0 1 1 0 1 0
1 0 1 1 1 0 0 1 1 0 1 0 1 0 0 1 1 1 0 1 0 1 1 0 1 0 1 0 1 1 0 0 1 0
0 1 0 1 1 1 1 1 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 1 0 0 1 0 1 0 0 1 0 0
0 1 0 1 1 1 0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 1 1 0 0 1 0 0 1 1 0 0 0 1
0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 0 1 1 0 0 0 1 0 0 1 0 1 0 0 1 0 1 1 0
1 1 1 1 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0 1
1 0 1 1 1 0 1 0 1 1 0 1 0 1 0 1 1 0 0 1 1 0        Br. Ist. Kl: 2550

23.3.2011. 1:58:49

Ukupno vrijeme:      00:07:00.8419391
Parametri: 20,60
```

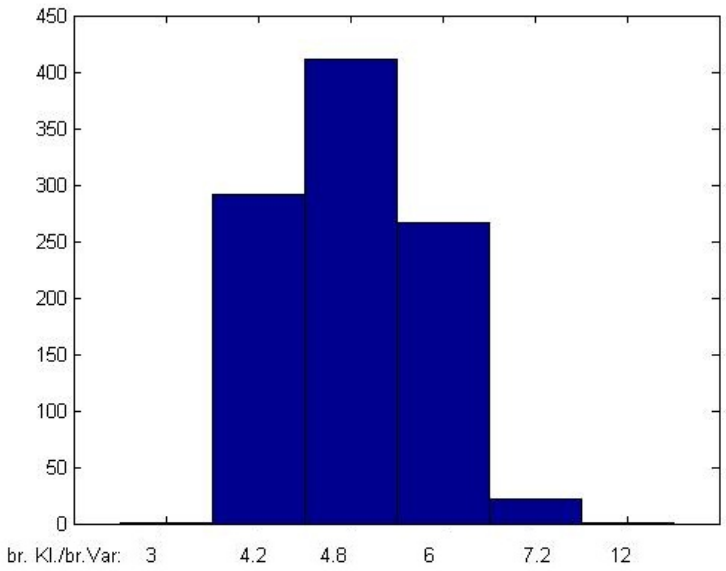
Solver smo testirali i na formulama koje su s visokom vjerojatnošću istinite za samo jednu interpretaciju. Vjeruje se da su formule za 3-SAT teško rješive kada je odnos broja klauzula i broja varijabli blizu 4.2, odnosno imamo u prosjeku oko 4.2 klauzule po varijabli formule ([21]). Za generiranje takvih formula koristimo algoritam `G3.C` ([11]) skinut s [21]. Generirali smo po pet formula s 300, 350, 400 varijabli i pokrenuli svaku 10 puta na TripleU solveru. Generirali smo formule za koeficijente (odnos broja klauzula i broja varijabli) 3, 4.2, 4.8, 6.0, 7.2, 12. Uz to, ispitali smo izvršavanje solvera na formulama s 450, 575, 600 varijabli za koeficijente 3, 12, na formuli s 575 varijabli za koeficijent 4.2 te formulu s 1000 varijabli za koeficijent 12. Sve test primjere smo testirali s parametrima $bt = 20$, $bp = 300$ na Inter Core2 Quad 3.00GHz procesoru.



Slika 22: Usporedni boxplot izvođenja solvera na 5 slučajno generiranih formula s 300 varijabli i različitim omjerima broja klauzula i broja varijabli.

Tablica 2: Točnost izvođenja TripleU solvera na 5 slučajno generiranih formula s 300 varijabli i različitim omjerima broja klauzula i broja varijabli.

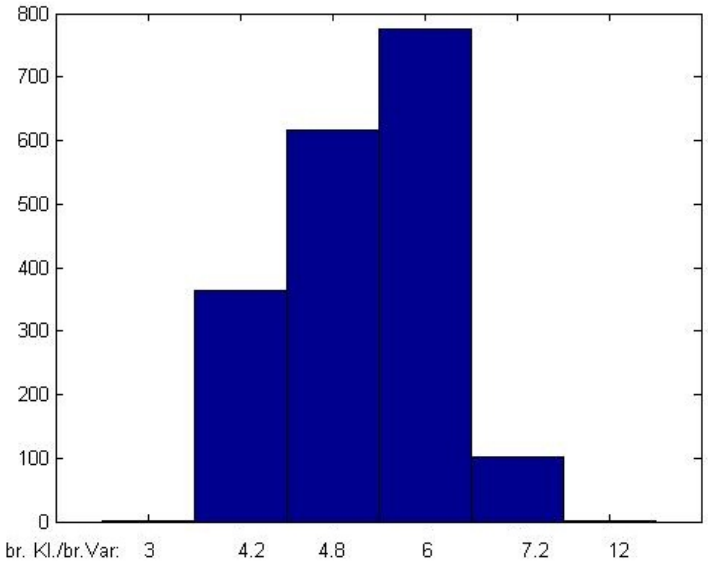
Koeficijent:	3.0	4.2	4.8	6.0	7.2	12.0
Točnost:	100%	8%	48%	98%	100%	100%



Slika 23: Usporedni boxplot izvođenja solvera na 5 slučajno generiranih formula s 350 varijabli i različitim omjerima broja klauzula i broja varijabli.

Tablica 3: Točnost izvođenja TripleU solvera na 5 slučajno generiranih formula s 350 varijabli i različitim omjerima broja klauzula i broja varijabli.

Koeficijent:	3.0	4.2	4.8	6.0	7.2	12.0
Točnost:	100%	10%	0%	78%	100%	100%



Slika 24: Usporedni boxplot izvođenja solvera na 5 slučajno generiranih formula s 400 varijabli i različitim omjerima broja klauzula i broja varijabli.

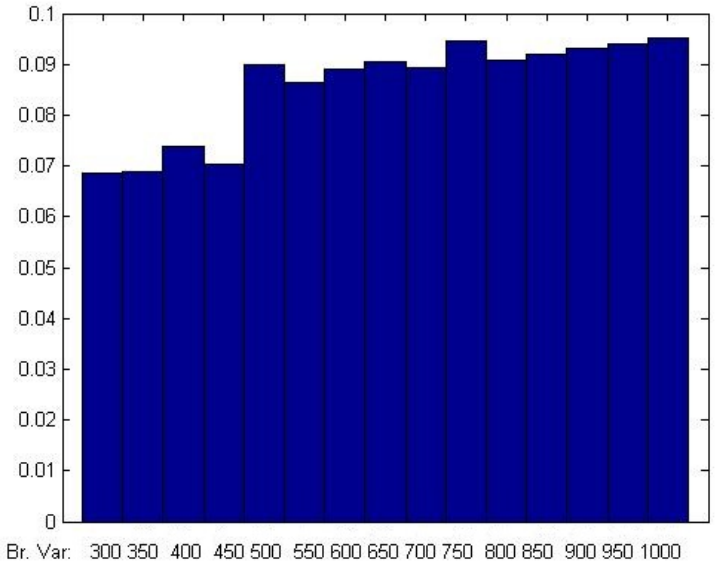
Tablica 4: Točnost izvođenja TripleU solvera na 5 slučajno generiranih formula s 400 varijabli i različitim omjerima broja klauzula i broja varijabli.

Koeficijent:	3.0	4.2	4.8	6.0	7.2	12.0
Točnost:	100%	32%	0%	4%	100%	100%

Iz ovih testiranja možemo uočiti kako interval težine rješavanja problema raste s povećanjem broja varijabli ulazne formule. Uz fiksne parametre *bt* i *bp* dobili smo da je vrijeme izvršavanja za problem od 400 varijabli s koeficijentom 6 najveće za razliku od vremena izvršavanja za formule s 300 i 350 varijabli u kojima je vrijeme izvršavanja najveće za koeficijente 4.2 odnosno 4.8. Eksperimentalni podaci za naš solver se djelomično poklapaju s poznatim eksperimentalnim saznanjima o težini rješivosti formule u odnosu na omjer broja njenih klauzula i broja varijabli. Za koeficijente 3 i 12 uočavamo veliki pad u vremenima izvršavanja neovisno o broju varijabli ulazne formule.

Podaci pokazuju kako je točnost najlošija za koeficijente 4.2 i 4.8 što utječe i na vrijeme izvršavanja jer algoritam izvodi maksimalni broj iteracija koji ovisi o parametrima bt i bp. Također uočavamo da je točnost izvršavanja na formulama s koeficijentima 3, 7.2 i 12 100% što je povezano s vremenima izvršavanja, koja su zbog točnosti bitno manja. Za formule s 400 varijabli uočavamo da je vrijeme izvršavanja na formulama s koeficijentom 6 veće od vremena izvršavanja na formulama s koeficijentom 4.8 što je direktna posljedica pada u točnosti solvera za taj problem.

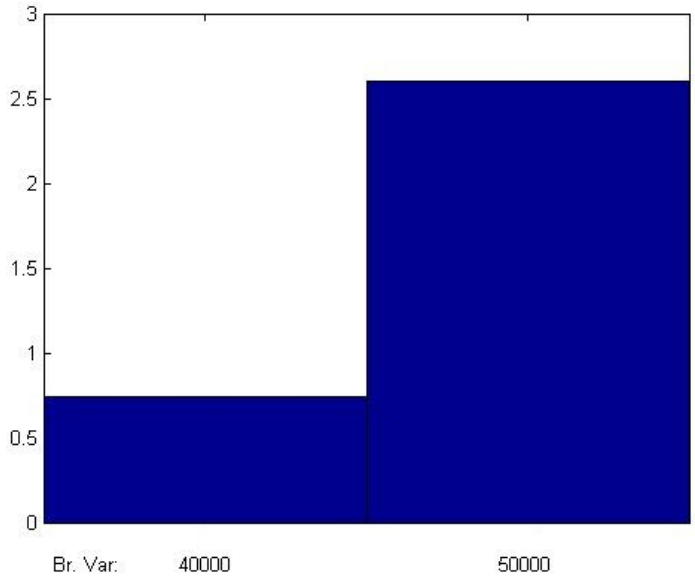
Uočili smo zanimljivo kretanje vremena izvršavanja solvera na formulama s koeficijentima 3 i 12 s različitim brojem varijabli stoga smo odlučili detaljnije testirati izvođenje na takvim formulama. Svaki test primjer je računat 10 puta i izvođen na i3 $2,4$ GHz procesoru.



Slika 25: Usporedni boxplot izvođenja solvera na 5 slučajno generiranih formula s koeficijentom 3 i $300 - 1000$ varijabli.

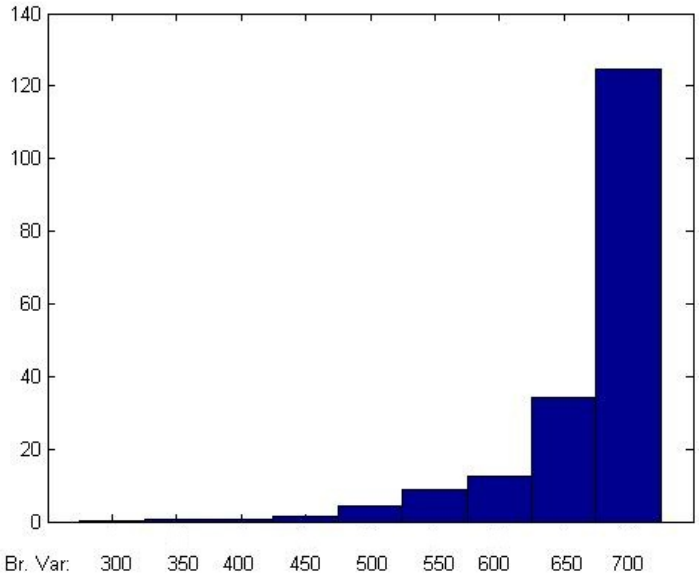
Na slici 25 vidimo blagi rast vremena izvođenja solvera unatoč činjenici što smo testirali njegovo izvršavanje na formulama s velikim rasponom broja varijabli. Razlika u vremenu izvršavanja između formula s 300 varijabli i formula s 1000 varijabli je oko $0.02s$, dok je razlika u prostoru rješenja ogromna 2^{300} i 2^{1000}. Za potrebe izvršavanja na ovako lako rješivim formulama, reducirali

smo vrijeme čekanja glavnog threada u solveru na 20*ms* bez obzira na dimenziju problema. Solver je s 100%-tnom točnošću riješio sve test primjere.



Slika 26: Usporedni boxplot izvođenja solvera na 5 slučajno generiranih formula s koeficijentom 3, 40000 i 50000 varijabli.

Zbog sporog rasta vremena izvršavanja za formule s koeficijentom 3 generirali smo i dvije skupine formula s 40000 i 50000 varijabli. Iz rezultata testiranja možemo zaključiti da su formule s koeficijentom 3 generirane algoritmom `G3.C` jako lako rješive i istinite su za više od jedne interpretacije što našemu solveru omogućava da tako efikasno dođe do rješenja. Točnost izvršavanja solvera na ovim instancama je 100%.



Slika 27: Usporedni boxplot izvođenja solvera na 5 slučajno generiranih formula s koeficijentom 12 i 300 − 700 varijabli.

Sa slike 27. vidimo da solver može riješiti puno veće formule u puno kraćem vremenu s koeficijentom 12 nego s koeficijentima 4.2, 4.8, točnost rješavanja solvera na testiranim formulama s koeficijentom 12 je bila 100%. Iz slike vidimo kako vrijeme izvršavanja ipak puno brže raste za koeficijent 12 nego koeficijent 3, stoga bi već za formule s 1000 varijabli vrijeme izvršavanja prešlo $1min$.

6 Zaključak

U ovom radu dizajnirali smo tri nova heuristička algoritma za rješavanje problema ispunjivosti logičke formule. Danas se intenzivno istražuje problem ispunjivosti logičke formule. Razvijaju se novi potpuni i heuristički algoritmi, pokušava se bolje razumijeti struktura problema i pronaći učinkovitiji algoritam za njegovo rješavanje.

Trenutno su eksperimentalno najbolji algoritmi bazirani na slučajnim šetnjama. Stoga smo i mi kao bazu svojih algoritama koristili slučajne šetnje kombinirane s eliminacijom jediničnih klauzula koje smo dodatno poboljšali upotrebom pohlepnog i genetskog algoritma. Osnovna ideja kombiniranja pohlepnog i genetskog pristupa sa slučajnim šetnjama je ta da genetski i pohlepni pristup generiraju bolje ulazne interpretacije iz kojih ćemo onda slučajnim šetnjama brže doći doći do rješenja.

Da bi poboljšali performanse i točnost rješavanja, napravili smo paralelni softverski modul koji objedinjuje sve prethodno opisane tehnike. Dokazali smo da naš modul ima jedno jako bitno svojstvo, a to je da je vjerojatnosno aproksimativno kompletan. Ukoliko je formula ispunjiva, on može s vjerojatnošću jedan doći do rješenja iz proizvoljno generirane početne interpretacije.

Na osnovi testiranja smo zaključili da naša implementacija pokazuje izvrsne rezultate na testnim primjerima $uf20 - uf250$.

Vjeruje se da su najteže rješive formule za $3-SAT$ one s omjerom broja klauzula i broja varijabli oko 4.2. Proučavanjem odnosa broja klauzula i broja varijabli na slučajno generiranim formulama `G3.C` algoritmom, uočili smo da se povećavanjem broja varijabli formula povećava i interval omjera broja klauzula i broja varijabli na kojem su te formule teško rješive. Izvan toga intervala naš algoritam daje izuzetno dobre rezultate u vremenima izvršavanja i točnosti, a unutar njega je po točnosti najlošiji u okolini 4.2 što potvrđuje navedeno vjerovanje da su upravo u okolini te vrijednosti formule najteže rješive. Primijetili smo da interval na kojima su formule generirane algoritmom `G3.C` teško rješive raste brže na desnom rubu. Na kraju smo proveli testiranja na formulama koje imaju u prosjeku 3 klauzule po varijabli te 40000 i 50000 varijabli. Rezultati pokazuju da je naš softverski modul vrlo učinkovit u rješavanju takvih problema.

Zahvale

Zahvaljujemo se našoj mentorici, Doc. dr. sc. Goranki Nogo što nas je s ovako zanimljivom temom potaknula na samostalna istraživanja o problemu ispunjivosti logičke formule kao i o heurističkim metodama za njegovo rješavanje. Svojim velikim razumijevanjem, stručnim i korisnim savjetima te nesebičnom pomoći je omogućila stvaranje ovoga rada.

Zahvaljujemo se Doc. dr. sc. Mladenu Vukoviću na poticaju za stvaranje ovoga rada i Prof. dr. sc. Robertu Mangeru na stručnoj pomoći.

Literatura

[1] A. Biere, M. Heule, H. van Maaren, T. Walsh, *Handbook of Satisfiability*, IOS Press, 2009.

[2] H. Böning, T. Lettmann, *Propositional Logic: Deduction and Algorithms*, Cambridge University Press, 1999.

[3] D. Boughaci, B. Benhamou, H. Drias, *Scatter Search and Genetic Algorithms for MAX-SAT Problems*, Springer Science, 2008.

[4] G. Folino, C. Pizzuti, G. Spezzano, *Parallel Hybrid Method for SAT That Couples Genetic Algorithms and Local Search*, IEEE Transactions on Evolutionary Computation, vol.5, no. 4, pp. 323-334, August 2001.

[5] J. Gu. *Efficient local search for very large scale satisfiability problems*, SIGART Bulletin, 3(1):8-12, 1992.

[6] E. A. Hirsc, A. Kojevnikov, *UnitWalk: A new SAT solver that uses local search guided by unit clause elimination*
`http://www.cs.ubc.ca/labs/beta/Courses/CPSC532D-03/Resources/HirKoj02b.pdf`

[7] I. Ivanuš, *Rezolucija u logici sudova*, Diplomski rad, PMF-MO, Zagreb, 2008.

[8] H. Kautz, D. McAllester, B. Selman, *WalkSAT in the 2004 SAT Competition*, Toyota Technical Institute at Chicago, 2004.

[9] V. W. Marek, *Introduction to Mathematics of Satistiability*, CRC Press, 2009.

[10] A. McDonald, G. Gordon, *Parallel WalkSAT with Clause Learning*
`http://www.ml.cmu.edu/current_students/DAP_mcdonald.pdf`

[11] M. Motoki, R. Uehara, *Unique solution generation for* 3*-SAT*, SAT2000, 2000.

[12] C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.

[13] C. H. Papadimitriou, K. Steiglitz, *Combinatorial Optimization*, Prentice-Hall, 1982.

[14] R. Paturi, P. Pudlák, F. Zane, *Satisfiability coding lemma*, Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS'97. str. 566-574.

[15] R. Paturi, P. Pudlák, M. E. Saks, F. Zane, *An improved exponential-time algorithm for k-SAT*, Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, FOCS'98. str. 628-637.

[16] B. Selman, H. Levesque, D. G. Mitchell, *A new method for solving hard satisfiability problems*, 10th National Conference on Artificial Intelligence, 440-446, AAAI Press/The MIT Press, 1992.

[17] B. Selman, H. A. Kautz, and B. Cohen, *Noise strategies for improving local search*, 12th National Conference on Artificial Intelligence, 337-343, AAAI Press/The MIT Press, 1994.

[18] M. Sipser, *Introduction to the Theory of Computation*, Thomson Course Technology, 2006.

[19] M. Vuković, *Matematička Logika*, Element, 2009.

[20] *SATLIB benchmark primjeri*
`http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

[21] *SAT Instance Generation page*
`http://www.is.titech.ac.jp/~watanabe/gensat/index.html`

Sažetak

Tihomir Lolić, Matej Mihelčić: **Nova paralelna heuristika za rješavanje problema ispunjivosti logičke formule**

Problem ispunjivosti logičke formule (SAT) je problem u kojemu za zadanu formulu logike sudova ispitujemo postojanje interpretacije takve da je ta formula za nju istinita. SAT je jedan od najpoznatijih i najproučavanijih matematičkih problema.

U ovom radu razvijene su tri nove heuristike za rješavanje problema SAT bazirane na UnitWalk algoritmu: modificirani UnitWalk algoritam, UnitWalk+SGAV algoritam i UnitWalk+Greedy algoritam. Za svaki od navedenih algoritama su izvedene analize složenosti te su detaljno opisani implementacijski detalji.

Nadalje, objašnjen je način rada pripadnog paralelnog softverskog modula, njegova struktura i funkcionalnost te mogući načini testiranja njegovoga rada. Dokazano je da je naša implementacija vjerojatnosno aproksimativno kompletna.

Na kraju je provedena detaljna eksperimentalna evaluacija naših algoritama. Promatran je učinak paralelizacije na prosječnu brzinu pronalaženja rješenja kao i na broj rubnih vrijednosti i njihovu udaljenost od srednje vrijednosti. Analiziran je i učinak primjene genetskog i pohlepnog algoritma kao metoda poboljšanja UnitWalk algoritma.

Eksperimentalni rezultati na klasi formula $uf20 - uf250$ pokazuju da je naša implementacija bitno brža od implementacija originalnih algoritama na kojima je bazirana.

Ključne riječi: SAT, slučajne šetnje, pohlepni algoritmi, eliminacija jediničnih klauzula, paralelno računanje

Summary

Tihomir Lolić, Matej Mihelčić: **A new parallel heuristic for solving the satisfiability problem**

Satisfiability problem (SAT) is a problem of determining the existence of an interpretation for an arbitrary formula of proposition logic such that the input formula is true. SAT is one of the most famous and explored mathematical problems.

In this paper, we design three new heuristics for solving the SAT problem based on the UnitWalk algorithm: the Modified UnitWalk algorithm, the UnitWalk+SGAV algorithm, and the UnitWalk+Greedy algorithm. We analyze complexity of each of these algorithms and explain their implementation specifics in detail.

Further, we explain the structure, design, and functionality of our parallel solver as well as propose several ways of testing it. We prove that our implementation is probabilistically approximately complete.

Finally, we present a detailed experimental evaluation of our algorithms. We observe effects of parallelization to an average execution, determine the number of outliers in data results and their distance from the median of the data set. We analyze effect of genetic and greedy algorithms applied to improve the UnitWalk algorithm.

Experimental results on $uf20 - uf250$ class of formulas show that our solver performs significantly better compared to the original algorithm on which it was based.

Key words: SAT, random walks, greedy algorithms, unit clause elimination, parallel computing