

Ne-klauzalni SAT rešavači

Seminarski rad iz kursa Automatsko Rezonovanje

Marko Lazarević

Februar 2026

Sažetak

U ovom radu prikazana je implementacija ne-klauzalnog SAT rešavača koji radi direktno nad formulama u opštem obliku. Implementacija koristi metode istinitosnih tablica i pristup zasnovan na usmerenom acikličnom grafu (DAG). Testiranjem na različitim formulama, uključujući tautologije, kontradikcije i složene iskazne izraze, upoređujemo ispravnost i vreme izvršavanja obe metode, ističući prednosti i ograničenja svake, kao i situacije u kojima DAG pristup pokazuje efikasnost u odnosu na klasične tablice.

Sadržaj

| | | |
|----------|---------------------------------------|-----------|
| 1 | Uvod | 2 |
| 2 | Problem SAT | 2 |
| 2.1 | Definicija | 2 |
| 3 | Metod istinitosnih tablica | 2 |
| 3.1 | Opis metode | 2 |
| 3.2 | Implementacija | 3 |
| 3.2.1 | Formula | 3 |
| 3.2.2 | Algoritam grube sile | 4 |
| 4 | Metod DAG | 5 |
| 4.1 | Opis metode | 5 |
| 4.2 | Implementacija | 6 |
| 4.2.1 | DAGNode i pomoćne strukture | 6 |
| 4.2.2 | Izgradnja DAG-a | 7 |
| 4.2.3 | Algoritam označavanja | 8 |
| 5 | Poređenje rezultata | 10 |
| 6 | Zaključak | 12 |

1 Uvod

Problem zadovoljivosti (SAT) predstavlja osnovni izazov u oblasti računarskih nauka, logike i veštačke inteligencije. Cilj je odrediti da li postoji dodela istinitosnih vrednosti promenljivama koja čini datu iskaznu formulu istinitom. Industrijski standard današnjice predstavljaju SAT rešavači zasnovani na predstavljanju formule u konjunktivnoj normalnoj formi (KNF). Međutim, kroz proces transformacije formule iz opšteg u klauzalni oblik mogu se izgubiti važne informacije o samoj formuli, pa je interpretacija rezultata često teška.

U ovom radu istražujemo dva pristupa rešavanju SAT problema nad formulama u ne-klauzalnoj formi: klasičnu metodu istinitosnih tablica i pristup baziran na usmerenom acikličnom grafu (DAG). Dok metoda istinitosnih tablica pruža jednostavno i intuitivno rešenje, DAG pristup teži efikasnijem rešavanju i iskorišćavanju strukture formule. Implementacijom i testiranjem ispitujemo ispravnost i vreme izvršavanja obe metode na različitim formulama, uključujući tautologije, kontradikcije i složenije iskazne izraze.

Implementacija opisana u daljem tekstu dostupna je na GitHub-u [1].

2 Problem SAT

Problem zadovoljivosti iskaznih formula (SAT) predstavlja jedan od centralnih problema teorijskog računarstva i ima ključnu ulogu u razvoju savremenih metoda automatskog rezonovanja. Iako se u ovom radu polazi od pretpostavke da čitalac poseduje osnovno predznanje o ovom problemu, ovde se daje njegova formalna definicija radi potpunosti izlaganja i preciznog utemeljenja daljih razmatranja.

2.1 Definicija

Problem zadovoljivosti iskaznih formula (*eng. Propositional satisfiability problem (SAT)*) jeste problem odlučivanja da li za datu iskaznu formulu postoji valuacija njenih promenljivih u kojoj se formula evaluira na tačno. Drugim rečima, ispituje se postojanje dodele logičkih vrednosti promenljivima takve da je data iskazna formula zadovoljiva [2].

U okviru ovog rada, SAT problem se posmatra kao polazna formalna osnova za razmatranje struktura i metoda koje se koriste u savremenim SAT rešavačima, pri čemu se akcenat ne stavlja na samu teorijsku složenost problema, već na način njegove reprezentacije i obrade.

3 Metod istinitosnih tablica

Prvi metod koji razmatramo je metod istinitosnih tablica. Iako ovaj metod, zbog svoje neefikasnosti, nema veliki praktični značaj, njegova vrednost se ogleda u jednostavnosti implementacije i lakoći razumevanja. Upravo zbog toga, metod istinitosnih tablica predstavlja dobru referentnu tačku za proveru ispravnosti i validaciju rezultata dobijenih naprednjim metodama, koje će biti predstavljene kasnije u radu.

3.1 Opis metode

Osnovna ideja metode istinitosnih tablica zasniva se na principu grube sile: za datu formulu razmatraju se sve moguće valuacije njenih promenljivih. Ukoliko se pronađe makar

jedna valuacija za koju formula ima vrednost **TRUE**, formula je zadovoljiva. U suprotnom, ako se iscrpe sve moguće valuacije bez pronalaženja zadovoljavajuće, zaključuje se da formula nije zadovoljiva.

3.2 Implementacija

Implementacija metode istinitosnih tablica realizovana je u programskom jeziku C++. Formula je modelovana kao stablo, pri čemu svaki čvor predstavlja ili logičku promenljivu ili logički operator. Ovakav pristup omogućava prirodno rekurzivno izračunavanje vrednosti formule za datu valuaciju, kao i jednostavno proširivanje sistema novim operatorima.

Evaluacija formule za konkretnu valuaciju vrši se pozivom odgovarajuće metode nad korenim čvorom stabla, dok se sam prolazak kroz sve moguće valuacije ostvaruje rekurzivnom enumeracijom dodela logičkih vrednosti promenljivima.

3.2.1 Formula

Formula je implementirana kao apstraktna (virtualna) struktura koja definiše interfejs zajednički svim tipovima formula. Svaki konkretan tip formule predstavlja poseban čvor u stablu izraza i implementira logiku evaluacije u skladu sa semantikom odgovarajućeg logičkog operatora.

Osnovna apstraktna struktura **Formula** sadrži sledeće virtualne metode:

- **print()** – služi za ispis formule u čitljivom obliku,
- **solve(const Valuation&)** – izračunava vrednost formule za datu valuaciju,
- **get_vars()** – vraća skup promenljivih koje se pojavljuju u formuli,
- **signature()** – generiše jedinstveni string-potpis formule, koji se kasnije koristi za identifikaciju strukturno jednakih podformula.

Valuacija promenljivih je predstavljena mapom koja svakom imenu promenljive dodeljuje logičku vrednost:

```
Valuation = map<string, bool>
```

Sve konkretne strukture koje predstavljaju logičke izraze (promenljive i operatori) nasleđuju apstraktну strukturu **Formula**. Na ovaj način svaki čvor stabla predstavlja instancu tipa **Formula**, što omogućava uniforman tretman svih podformula kroz polimorfizam. Implementirane strukture navedene su u nastavku.

Promenljiva. Struktura **Variable** predstavlja logičku promenljivu. Njena evaluacija se svodi na čitanje vrednosti iz date valuacije, dok se skup promenljivih sastoji isključivo od same promenljive.

Negacija. Struktura **Not** predstavlja unarni operator negacije. Sadrži pokazivač na operand i negira njegovu vrednost prilikom evaluacije. Skup promenljivih identičan je skupu promenljivih operanda.

Binarni operatori. Svi binarni logički operatori (`And`, `Or`, `Implies`, `Eq`) nasleđuju zajedničku strukturu `BinaryOp`, koja sadrži pokazivače na levi i desni operand. Ova struktura implementira zajedničku funkcionalnost za prikupljanje skupa promenljivih iz oba podizraza.

Konkretnе semantike operatora definisane su u odgovarajućim izvedenim strukturama:

- `And` – logička konjunkcija,
- `Or` – logička disjunkcija,
- `Implies` – logička implikacija,
- `Eq` – logička ekvivalencija.

Metoda `signature()` kod binarnih operatora dodatno sortira potpise leve i desne podformule kod komutativnih operatora (`And`, `Or`, `Eq`), čime se obezbeđuje jedinstvena reprezentacija strukturno ekvivalentnih formula.

Ovakav dizajn omogućava jasnu i modularnu implementaciju metode istinitosnih tablica, kao i jednostavno proširivanje sistema. Iako sama metoda nije efikasna za veće instance, ona predstavlja pouzdanu osnovu za poređenje sa naprednjijim algoritmima, što je i njen osnovni cilj u ovom radu.

3.2.2 Algoritam grube sile

Algoritam grube sile za rešavanje SAT problema zasniva se na potpunoj enumeraciji svih mogućih valuatora promenljivih koje se pojavljuju u formuli. Za svaku takvu valuatoru računa se vrednost formule, a ukoliko se pronađe makar jedna valuatora za koju formula ima vrednost `TRUE`, zaključuje se da je formula zadovoljiva. U suprotnom, ako nijedna valuatora ne zadovoljava formulu, ona je nezadovoljiva.

Algoritam se prirodno implementira rekurzivno. U svakom koraku bira se jedna nevaluirana promenljiva i dodeljuje joj se najpre vrednost `TRUE`, a zatim `FALSE`. Rekurzija se nastavlja sve dok se ne dodele vrednosti svim promenljivim. U tom slučaju, vrednost formule se izračunava pozivom metode `solve` nad korenim čvorom stabla formule.

U nastavku je prikazana kompletan implementacija algoritma.

```
bool truthtable(Formula*& f, std::set<std::string>& vars, Valuation& v) {
    if (vars.size() == v.size()) {
        return f->solve(v);
    }

    std::string current_var;
    for (const auto& var : vars) {
        if (v.find(var) == v.end()) {
            current_var = var;
            break;
        }
    }

    v[current_var] = true;
    if (truthtable(f, vars, v)) {
        v.erase(current_var);
```

```
    return true;
}

v[current_var] = false;
if (truthtable(f, vars, v)) {
    v.erase(current_var);
    return true;
}

v.erase(current_var);
return false;
}

bool sat_truthtable(Formula*& f) {
    auto vars = f->get_vars();
    Valuation v;
    return truthtable(f, vars, v);
}
```

Implementacija je korektna jer algoritam sistematski razmatra sve moguće valuacije skupa promenljivih formule. Rekurzivno grananje garantuje da nijedna valuacija neće biti preskočena, dok se evaluacija formule vrši isključivo kada su sve promenljive instancirane. Na ovaj način se postiže potpuna pretraga prostora rešenja, čime se obezbeđuje tačnost rezultata.

4 Metod DAG

U klasičnim SAT rešavačima iskazne formule se najčešće prevode u konjunktivnu normalnu formu (KNF), čime se često gubi deo strukturalnih informacija prisutnih u originalnoj formuli. Alternativni pristup, predložen u radu [3], zasniva se na direktnoj obradi formule kroz njenu grafovsku reprezentaciju, čime se omogućava efikasnije ponovno korišćenje zajedničkih potformula i prirodnije rezonovanje nad strukturom izraza.

4.1 Opis metode

Osnovna ideja metode, opisana u [3], zasniva se na posmatranju iskazne formule kao stabla operatora, gde unutrašnji čvorovi predstavljaju logičke operatore, a listovi odgovaraju iskaznim promenljivama. Umesto rada nad stablom, formula se transformiše u usmereni aciklički graf (DAG) u kojem se sve strukturno identične potformulae objedine u jedan čvor. Na taj način se izbegava višestruko evaluiranje istih potformula i dobija se kompaktnija reprezentacija formule, često nazvana i *Boolean circuit* [3].

Transformacija stabla u DAG vrši se odozdo nagore, korišćenjem tehnika poput heširanja kako bi se identifikovale i spojile ekvivalentne potformulae [3]. Svaki čvor u dobijenom DAG-u predstavlja jednu potformula i povezan je sa svojim neposrednim podformulama. Nakon formiranja DAG-a, cilj algoritma je da pronađe dosledno dodeljivanje istinitosnih vrednosti čvorovima tako da čvor koji predstavlja celu formulu bude označen kao tačan, pri čemu dodata mora poštovati semantiku logičkih operatora.

Rešavanje problema sprovodi se pretragom sa povratkom nad vrednostima čvorova u DAG-u, analogno DPLL postupku [3]. U svakom koraku bira se čvor kojem još nije

dodeljena vrednost i pokušava se dodela istinitosne vrednosti, nakon čega se propagiraju logičke posledice te odluke kroz graf, kako ka roditeljskim čvorovima, tako i ka deci. Ukoliko se tokom propagacije otkrije kontradikcija, postupak se vraća unazad i ispituje se alternativna dodela. Ako se pronađe dosledno označavanje svih relevantnih čvorova bez kontradikcija, formula je zadovoljiva, a u suprotnom se zaključuje da rešenje ne postoji.

Ovakav pristup može se posmatrati kao direktna primena pretrage nalik DPLL-u nad strukturu formule, pri čemu dodeljivanje vrednosti potformulama odgovara dodeljivanju vrednosti pomoćnim promenljivama u Cajtinovom kodiranju, dok propagacija u DAG-u odgovara jediničnoj propagaciji u klauzalnom obliku [3]. Prednost ovog pristupa je u tome što omogućava eksploraciju strukturalnih informacija prisutnih u originalnoj formuli, bez eksplisitne transformacije u CNF oblik.

4.2 Implementacija

Implementacija metode zasnovane na DAG reprezentaciji realizovana je u programskom jeziku C++. Fokus implementacije je na eksplisitnoj reprezentaciji potformula kao čvorova usmerenog acikličkog grafa i na algoritmu označavanja koji vrši propagaciju istinitosnih vrednosti kroz graf uz podršku povratnog pretraživanja.

Formula se najpre transformiše u DAG korišćenjem funkcije `build_dag`, pri čemu se identične potformule objedinuju u jedan čvor. Nakon izgradnje DAG-a, nad dobijenom strukturu se primenjuje algoritam označavanja koji pokušava da konzistentno dodeli istinitosne vrednosti čvorovima tako da čvor koji predstavlja celu formulu bude označen kao TRUE. Implementacija ne koristi dodatne optimizacije poput watched čvorova ili učenja klauza, već se oslanja na osnovni skup pravila propagacije.

4.2.1 DAGNode i pomoćne strukture

Centralna struktura implementacije je struktura `DAGNode`, koja predstavlja jedan čvor u DAG-u i odgovara jednoj potformuli ulazne iskazne formule. Svaki čvor sadrži tip operatora, informacije o roditeljskim i potomnim čvorovima, kao i trenutno dodeljenu istinitosnu vrednost. Potpis strukture `DAGNode` dat je u nastavku:

```
struct DAGNode {
    DAGOp op;
    std::string var; // only used if op == VAR
    std::vector<DAGNode*> parents;
    std::vector<DAGNode*> children;
    TruthValue truth_value;
    TruthValueChange last_change;
    DAGNode() : truth_value(TruthValue::UNKNOWN),
                last_change(TruthValue::UNKNOWN, 0, TruthValueChangeReason::TRIGGER)
    {}

    // try to label this node with new_value at the given level,
    // return true if successful, false if it leads to a conflict
    bool label(TruthValue new_value, int level, TruthValueChangeReason reason);
    bool propagate_children(TruthValue new_value, int level);
    bool propagate_parents(TruthValue new_value, int level);
```

```
    void print() const;  
};
```

Tip operatora čvora određen je enumeracijom `DAGOp`, koja obuhvata promenljive, unarne i binarne logičke operatore (`VAR`, `NOT`, `AND`, `OR`, `IMPLIES`, `EQ`). Za čvorove koji predstavljaju promenljive, čuva se i ime promenljive u polju `var`.

Radi podrške povratnom pretraživanju, svakom čvoru se pridružuje struktura nazvana `TruthValueChange`, koja beleži prethodnu vrednost čvora, nivo odlučivanja na kojem je promena nastala i razlog promene. Ovo omogućava vraćanje stanja sistema prilikom backtracking-a.

Polja `parents` i `children` omogućavaju dvosmernu propagaciju istinitosnih vrednosti kroz DAG. Budući da DAG nije stablo, jedan čvor može imati više roditelja, što je ključno za pravilno modelovanje deljenih potformula.

4.2.2 Izgradnja DAG-a

Funkcija `build_dag` odgovorna je za konstrukciju DAG-a iz stabla formule. Tokom izgradnje koristi se mapa potpisa potformula kako bi se izbeglo kreiranje duplikata i obezbedilo da se strukturno identične potformule mapiraju na isti `DAGNode`. Na taj način se direktno ostvaruje kompresija strukture formule. Implementacija funkcije prikazana je u nastavku:

```
DAGNode* build_dag(Formula *f,  
                     std::map<std::string, DAGNode*>& node_map,  
                     DAGNode* parent) {  
  
    auto sig = f->signature();  
    if(node_map.count(sig)){  
        auto node = node_map[sig];  
        if(parent){  
            node->parents.push_back(parent);  
        }  
        return node;  
    }  
  
    DAGNode* node = new DAGNode();  
  
    if(parent){  
        node->parents.push_back(parent);  
    }  
  
    if (auto var = dynamic_cast<Variable*>(f)) {  
        node->op = DAGOp::VAR;  
        node->var = var->name;  
    }  
    else if (auto not_op = dynamic_cast<Not*>(f)) {  
        node->op = DAGOp::NOT;  
        node->children.push_back(  
            build_dag(not_op->operand, node_map, node)
```

```
    );
}

else if (auto and_op = dynamic_cast<And*>(f)) {
    node->op = DAGOp::AND;
    node->children.push_back(build_dag(and_op->left, node_map, node));
    node->children.push_back(build_dag(and_op->right, node_map, node));
}

else if (auto or_op = dynamic_cast<Or*>(f)) {
    node->op = DAGOp::OR;
    node->children.push_back(build_dag(or_op->left, node_map, node));
    node->children.push_back(build_dag(or_op->right, node_map, node));
}

else if (auto impl = dynamic_cast<Implies*>(f)) {
    node->op = DAGOp::IMPLIES;
    node->children.push_back(build_dag(impl->left, node_map, node));
    node->children.push_back(build_dag(impl->right, node_map, node));
}

else if (auto eq = dynamic_cast<Eq*>(f)) {
    node->op = DAGOp::EQ;
    node->children.push_back(build_dag(eq->left, node_map, node));
    node->children.push_back(build_dag(eq->right, node_map, node));
}

node_map[sig] = node;
return node;
}
```

4.2.3 Algoritam označavanja

Algoritam označavanja predstavlja jezgro implementacije i realizuje pretragu sa povratkom nad istinitosnim vrednostima čvorova u DAG-u. Osnovna operacija algoritma je metoda `label`, koja pokušava da dodeli novu istinitosnu vrednost čvoru na datom nivou odlučivanja.

Prilikom pokušaja označavanja, najpre se proverava da li nova vrednost ulazi u konflikt sa postojećom vrednošću čvora. Ukoliko konflikt postoji, označavanje se prekida i signalizira neuspeh. Ako je čvor već označen istom vrednošću, operacija se smatra uspešnom bez daljih akcija.

U slučaju uspešne dodele, pokreće se propagacija posledica nove vrednosti. Propagacija se odvija u dva smera: ka potomnim čvorovima i ka roditeljskim čvorovima, u zavisnosti od tipa operatora. Na primer, ako se čvor tipa AND označi kao TRUE, propagira se TRUE na svu njegovu decu, dok označavanje nekog deteta kao FALSE dovodi do propagacije FALSE ka roditeljskom AND čvoru.

Sve promene koje nastaju tokom propagacije prolaze kroz istu funkciju označavanja, čime se osigurava uniformno detektovanje konflikata. Ukoliko se tokom propagacije naiđe na konflikt, algoritam se vraća na prethodni nivo odlučivanja i pokušava alternativnu dodelu vrednosti. Osnovna implementacija je data u nastavku.

```
bool DAGNode::label(TruthValue new_value,
                     int level,
```

```
    TruthValueChangeReason reason) {

        if(truth_value == TruthValue::TRUE && new_value == TruthValue::FALSE ||
           truth_value == TruthValue::FALSE && new_value == TruthValue::TRUE){
            return false; // conflict
        }

        if(truth_value == new_value){
            return true; // already labeled with the same value
        }

        auto old_value = truth_value;
        auto old_change = last_change;
        last_change = TruthValueChange(truth_value, level, reason);
        truth_value = new_value;

        bool can_propagate_children = this->propagate_children(new_value, level);
        if(!can_propagate_children){
            truth_value = old_value;
            last_change = old_change;
            return false; // conflict during propagation to children
        }
        bool can_propagate_parents = this->propagate_parents(new_value, level);
        if(!can_propagate_parents){
            truth_value = old_value;
            last_change = old_change;
            return false; // conflict during propagation to parents
        }

        return true; // successfully labeled this node
    }
}
```

Pravila propagacije po tipu operatora:

- **NOT:** Ako čvor postane TRUE, propagira se FALSE na dete; ako čvor postane FALSE, propagira se TRUE na dete. Obrnuto, promene deteta propagiraju se ka NOT čvoru.
- **AND:** Ako AND postane TRUE, propagira se TRUE ka svima potomcima. Ako AND postane FALSE i samo jedno dete još nije označeno, propagira se FALSE na to dete. Ako bilo koje dete postane FALSE, propagira se FALSE ka roditelju. Ako su sva deca TRUE, propagira se TRUE ka roditelju.
- **OR:** Ako OR postane FALSE, propagira se FALSE ka svima potomcima. Ako OR postane TRUE i samo jedno dete još nije označeno, propagira se TRUE na to dete. Ako bilo koje dete postane TRUE, propagira se TRUE ka roditelju. Ako su sva deca FALSE, propagira se FALSE ka roditelju.
- **IMPLIES:** Ako IMPLIES postane FALSE, propagira se TRUE ka antecedentu i FALSE ka konsekventu. Promene u antecedentu i konsekventu propagiraju se ka IMPLIES čvoru u skladu sa logikom implikacije.

- **EQUIVALENCE:** Ako EQUIVALENCE postane TRUE, propagira se ista vrednost ka obe strane. Ako postane FALSE, propagira se suprotna vrednost ka obe strane. Promene u čvorovima A ili B propagiraju se nazad ka EQUIVALENCE u skladu sa logikom ekvivalencije.

Ovakav način implementacije omogućava jasno razdvajanje logike označavanja, propagacije i backtracking-a, uz minimalan broj pomoćnih struktura. Iako implementacija ne koristi napredne optimizacije kao što su *watched literals* ili učenje konflikata, ona verno prati osnovnu ideju neklauzalnog DPLL pristupa nad DAG reprezentacijom formule, što je dovoljno za demonstraciju metode.

5 Poređenje rezultata

U ovoj sekciji upoređujemo rezultate izvršavanja algoritma nad DAG reprezentacijom formule sa klasičnom metodom istinitosnih tablica. Testirane su razne logičke formule, uključujući osnovne tautologije, kontradikcije, kao i složenije formule sa više operatora i promenljivih.

Korišćene formule:

1. $A \vee \neg A$
2. $p \wedge \neg p$
3. $(p \vee q) \wedge \neg p \wedge \neg q$
4. $\neg(p \vee \neg p)$
5. $(p \Rightarrow q) \wedge p \wedge \neg q$
6. $(p \Leftrightarrow q) \wedge p \wedge \neg q$
7. $A \Rightarrow B$
8. $(X \wedge Y) \Rightarrow X$
9. $(\neg p \Rightarrow q) \Leftrightarrow (p \vee q)$
10. $(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee C)$
11. $(P \Rightarrow Q) \wedge (Q \Rightarrow R) \wedge \neg R$
12. $(X \Leftrightarrow Y) \wedge (Y \Rightarrow Z) \wedge (\neg Z \vee X)$
13. $(A \vee B \vee C) \wedge (\neg A \vee \neg B \vee C) \wedge (\neg C \vee A)$
14. $((P \wedge Q) \Rightarrow (R \wedge S)) \wedge (\neg R \vee \neg S)$
15. $((A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee C)) \wedge ((D \vee E) \wedge (\neg D \vee F) \wedge (\neg E \vee F))$
16. $((P \Rightarrow Q) \wedge (Q \Rightarrow R) \wedge \neg R) \vee ((S \Rightarrow T) \wedge (T \Rightarrow U) \wedge \neg U)$
17. $((((A \vee B) \wedge (\neg A \vee C)) \wedge ((B \vee D) \wedge (\neg B \vee E))) \Rightarrow (F \wedge G)$

18. $(X \Leftrightarrow Y) \wedge (Y \Rightarrow Z) \wedge (\neg Z \vee X) \wedge (U \Rightarrow V) \wedge (V \Rightarrow W) \wedge \neg W$
19. $((A \vee B \vee C) \wedge (\neg A \vee \neg B \vee C) \wedge (\neg C \vee A) \wedge (D \vee E) \wedge (\neg D \vee F) \wedge (\neg E \vee F)) \vee G$
20. $((A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee C)) \wedge ((D \vee E) \wedge (\neg D \vee F) \wedge (\neg E \vee F)) \wedge ((A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee C))$
21. $((P \Rightarrow Q) \wedge (Q \Rightarrow R) \wedge \neg R) \vee ((S \Rightarrow T) \wedge (T \Rightarrow U) \wedge \neg U) \vee ((P \Rightarrow Q) \wedge (Q \Rightarrow R) \wedge \neg R)$
22. $((A \vee B) \wedge (\neg A \vee C)) \wedge ((B \vee D) \wedge (\neg B \vee E)) \wedge ((A \vee B) \wedge (\neg A \vee C)) \Rightarrow (F \wedge G)$
23. $((X \Leftrightarrow Y) \wedge (Y \Rightarrow Z) \wedge (\neg Z \vee X) \wedge (U \Rightarrow V) \wedge (V \Rightarrow W) \wedge \neg W \wedge (X \Leftrightarrow Y) \wedge (Y \Rightarrow Z))$
24. $((A \vee B \vee C) \wedge (\neg A \vee \neg B \vee C) \wedge (\neg C \vee A) \wedge (D \vee E) \wedge (\neg D \vee F) \wedge (\neg E \vee F)) \vee ((A \vee B \vee C) \wedge (\neg A \vee \neg B \vee C))$
25. $((p \wedge q) \Rightarrow r) \wedge ((p \wedge q) \Rightarrow s) \wedge ((p \wedge q) \Rightarrow t) \wedge (p \wedge q)$
26. $(a \Rightarrow b) \wedge (b \Rightarrow c) \wedge (c \Rightarrow d) \wedge (d \Rightarrow e) \wedge (e \Rightarrow \neg a) \wedge a$
27. $((x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_3))$
28. $(p \vee q) \wedge (p \vee r) \wedge (p \vee s) \wedge (p \vee t) \wedge \neg p$

Uporedni rezultati: Za svih 28 testiranih formula, i DAG implementacija i metod istinitosnih tablica su dali korektne rezultate, što potvrđuje ispravnost implementacije. Vremena izvršavanja pokazuju da u većini slučajeva DAG algoritam ne donosi značajnu vremensku prednost nad jednostavnom tablicom istinitosti zbog male veličine testiranih formula:

- **Istinitosne tablice:** 28 tačnih, 0 netačnih
- **DAG:** 28 tačnih, 0 netačnih
- **DAG brži:** 6 testova
- **DAG sporiji:** 22 testa

Iz ovoga vidimo da je implementacija nad DAG-om ispravna i pouzdana, ali zbog odsustva optimizacija (npr. *watched literals*, učenje konflikata) performanse nisu značajno bolje od klasične brute-force pretrage. Ovo je očekivano za male formule, dok bi prednosti DAG pristupa postale očigledne na većim i složenijim formulama sa više promenljivih.

Napomena o performansama: Primetno je da DAG algoritam u nekim slučajevima brže detektuje konflikte kod UNSAT formula. Razlog je što propagacija istinitosnih vrednosti kroz DAG često dovodi do ranog otkrivanja neusaglašenosti, bez potrebe da se ispitaju sve moguće kombinacije vrednosti promenljivih, dok metod istinitosnih tablica uvek mora da proveri sve redove tablice istinitosti. Kod SAT formula, prednost DAG pristupa nije uvek vidljiva jer propagacija mora da pređe kroz više čvorova pre nego što se potvrdi da je formula zadovoljiva.

6 Zaključak

Na samom kraju možemo zaključiti da se moćni SAT rešavači ne moraju oslanjati na formule u KNF-u, ali se implementacija nad formulama u opštem obliku znatno usložnjava. To je verovatno jedan od razloga zbog kojih su SAT rešavači bazirani na klauzama i dalje industrijski standard već duži niz godina. Ipak, potencijal ne-klauzalnih SAT rešavača se ne sme zanemariti, pogotovo zato što se ne gubi osnovno značenje formule. Rezultati se mogu direktno interpretirati u početni problem koji je opisan iskaznom formulom.

Literatura

- [1] Marko Lazarević. *NonClausalSatSolver: Implementation of a Non-Clausal SAT Solver*. <https://github.com/marko-lazarevic/NonClausalSatSolver>. GitHub repository. 2026.
- [2] Filip Marić. „Formal Verification of a Modern SAT Solver by Shallow Embedding into Isabelle/HOL”. U: (2010.). URL: <https://poincare.matf.bg.ac.rs/~filip/phd/sat-verification-shallow.pdf>.
- [3] Xavier Thiffault, Fahiem Bacchus i Toby Walsh. *Solving Non-Clausal Formulas with DPLL Search*. Tehn. izv. CP-2004-13. École Polytechnique Fédérale de Lausanne (EPFL), 2004. URL: https://lara.epfl.ch/w/_media/projects:thiffaultetalcp2004.pdf.